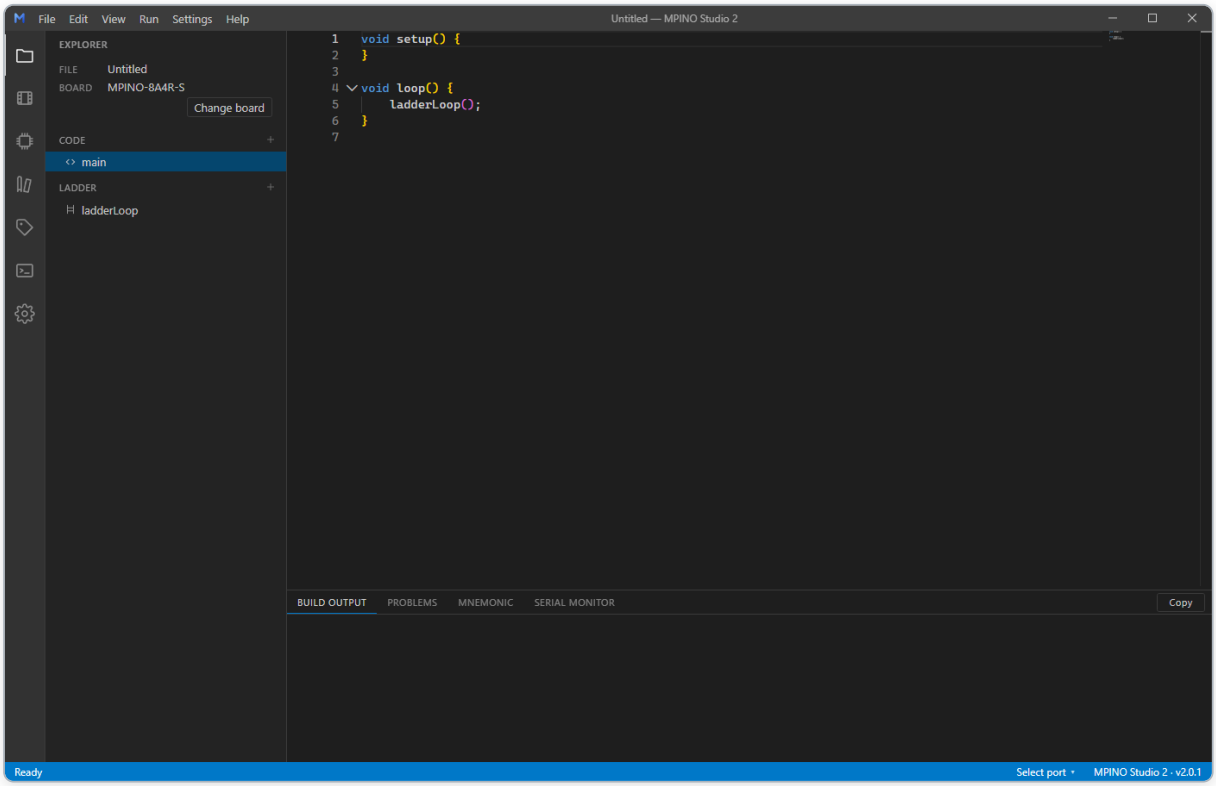
Index

1 Your First Project: Turning an Output On from an Input

1-1 What You Will Build

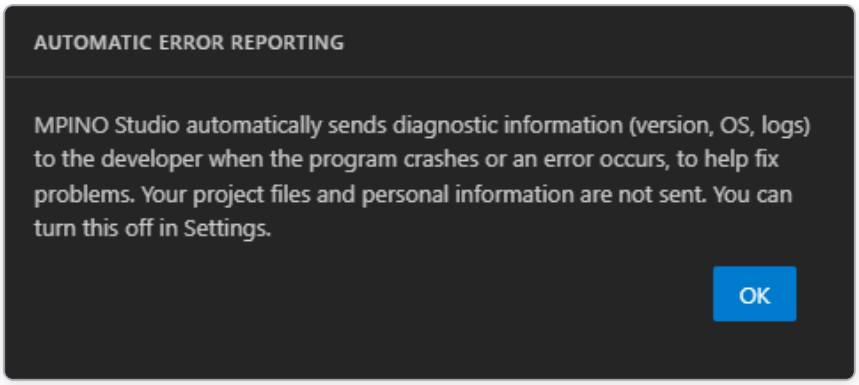
MPINO Studio 2 is a desktop IDE for ILOGICS MPINO industrial PLC boards. You edit Arduino C code and ladder logic side by side on one screen, and when you build, the ladder logic is automatically converted into a C function that runs inside `ladderLoop()`. This chapter follows the whole flow in one pass: from the moment you first launch the app, through creating a new project, drawing the simplest possible circuit in which one input contact turns on one output relay, building it, and uploading it to a real board to confirm that it works.

When you launch the app for the first time, no empty placeholder screen such as a "welcome screen" or a "no project" message appears. Instead, a fully working default (Untitled) project is already open. The board defaults to MPINO-8A4R(T)-S, and the code tab **main** and the empty ladder tab **ladderLoop** are ready, so you can start editing right away.



< Figure 1-1 The default screen on first launch — not an empty screen, but a default project that already works >

The very first time you run MPINO Studio 2, a one-time notice dialog titled "Automatic error reporting" appears together with the screen above. It explains that when the program terminates abnormally or an error occurs, diagnostic information (version, OS, logs) is sent automatically to the developer and used to solve the problem, and it also states that your project files and personal information are not sent. Once you close it with the **OK** button, it does not appear again on later launches. If you do not want automatic sending, you can turn it off later with the **Send error reports automatically**(3-1-1) checkbox on the **Preferences › General** tab.



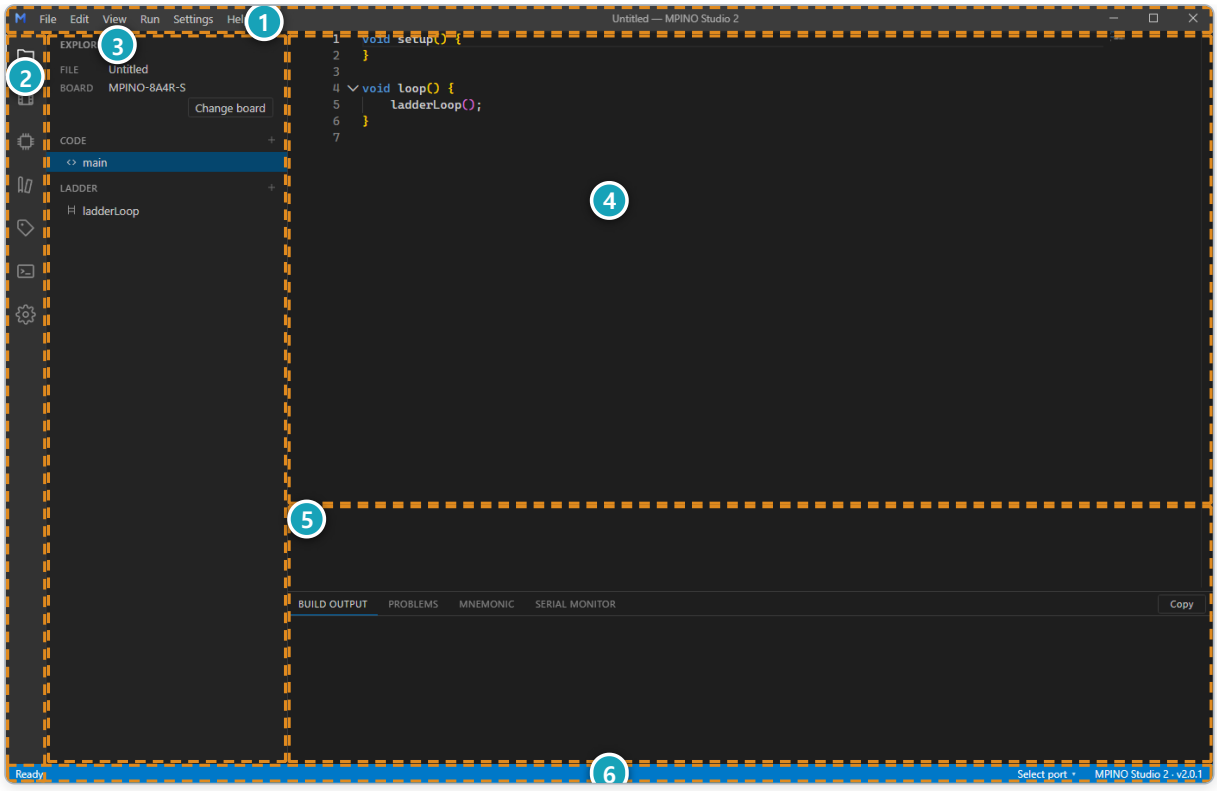
< Figure 1-2 The first-run notice — the Automatic error reporting dialog (it does not appear again once you press OK) >

Every time the app launches, it **checks once, automatically, whether an update is available**. If there is a new version, a "Check for Updates" window appears with the version information and the [Update Now] / [Later] buttons; if you are already on the latest version, or if the server cannot be reached, it passes over silently with no notice at all. To check whenever you like, use **Check for Updates** in the Help menu — checked that way, the result is always shown as a toast (see **Help**(3-6)).

> Project files (`.mp2`) are in a plain text (TOML) format, which makes them well suited to tracking change history with version control tools.

1-2 A Tour of the Screen

Before you get to work in earnest, take a moment to get familiar with the six areas that make up the screen. The screen below is the default layout with **Explorer** selected in the Activity Bar on the left; the name and the main behavior of each area are summarized in the legend below the figure.



< Figure 1-3 The default MPINO Studio 2 layout — menu bar, Activity Bar, Sidebar, Editor, bottom panel, and Status Bar >

① **Menu Bar** — The six top-level menus File, Edit, View, Run, Settings, and Help. Almost every command starts here, from New and Save to Build and Upload

② **Activity Bar** — Seven icons: Explorer, Ladder Settings, Ladder Pin Map Settings, Library, Symbols, Serial Monitor, and Settings. Clicking one switches the sidebar panel or opens a separate modal (Serial Monitor moves to the bottom panel)

③ **Sidebar (Explorer)** — The current project's file and board metadata together with the Code and Ladder tab tree. Click a row and that code or ladder is shown in the editor area

④ **Editor Area** — The space where the code editor and the ladder editor are stacked. Switch between them by clicking the sidebar tab tree (there is no tab strip at the top)

⑤ **Bottom Panel** — Four tabs: Build Output, Problems, Mnemonic, and Serial Monitor. Running a build or an upload switches to the Build Output tab automatically

⑥ **Status Bar** — The Ready text on the left, and the port dropdown and the version display on the right. The only way to select a port for uploading and for serial communication in general

A few of them do not work the way their names suggest. The **Ladder Settings** icon in the Activity Bar is not a way to swap boards; it opens a side panel that shows the current board's pin mapping and memory areas read-only — the panel itself has no button for changing the board. There is, however, a separate way to change only the target board without creating a new project — the **[Change board]** button next to the "Board" row in the Explorer panel, or **Change Board** in the Settings menu, replaces only the target board while leaving your code and ladder as they are (for details, see **The Explorer Panel**(9-2) and **Change Board**(3-5) in the Settings menu). And the icon shaped like a RAM chip is not a memory editor, despite its looks — it opens the **Ladder Pin Map Settings** modal (its tooltip reads "Ladder Pin Map Settings" as well).

The left side of the **Status Bar** at the very bottom of the screen always shows "Ready". This is a static label that does not change even while a build or monitoring is in progress. On the right, the port button for

choosing a serial port (for details on how to use it, see the port dropdown (Figure 1-9) in the Selecting a Port and Uploading section below) and the app version "MPINO Studio 2 · v2.0.4" are shown side by side.

1-3 Reading On-Screen Notifications

The results of many actions — a successful save, an error, a short piece of information — are shown as a notification (a toast) that appears briefly in the lower right of the screen and then disappears.

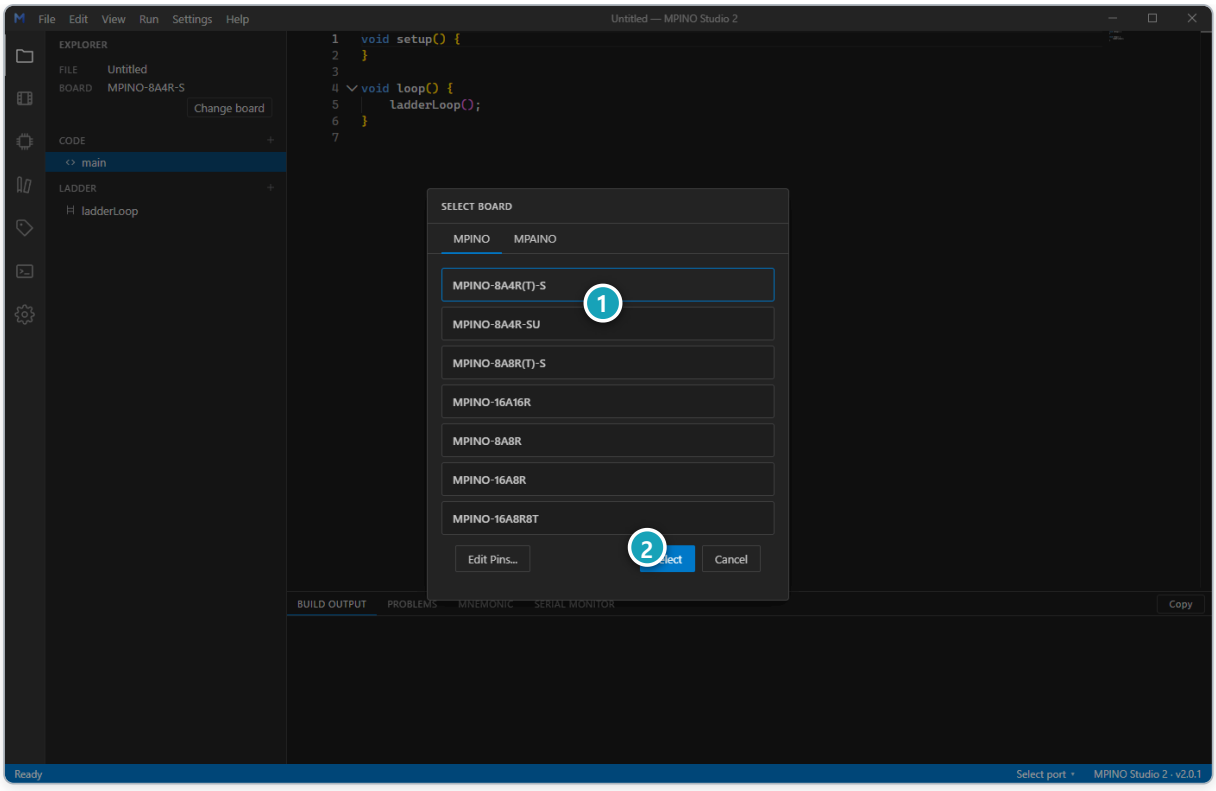
< Figure 1-4 An example of a notification (toast) that appears briefly in the lower right of the screen and then disappears — the green toast shown when the pin map is saved successfully >

A toast disappears on its own after about **4 seconds**. To close one sooner, click the ✕ button on the notification or press **Esc**. There are three kinds — **Error**, **Info**, and **Success** — distinguished by the color of the vertical line on the left (red for error, the accent color for info, green for success). At most **3** are stacked vertically at a time; a new notification is added at the bottom (the side closer to the corner of the screen) and earlier ones are pushed up above it. A toast differs from a modal or a dialog that you have to acknowledge to dismiss, in that it does not cover the screen behind it and does not require any separate action.

1-4 Creating a Project and Selecting a Board

Now let us create a real project. The menu path is **File > New**, and the shortcut is **Ctrl+N**.

1. Click **New** in the **File** menu, or press `Ctrl+N`.
2. If the project you were editing has unsaved changes, a **Discard changes** confirmation dialog appears first. Press **Discard** to continue, or **Cancel** to stop.
3. The **Select Board** modal opens immediately. It is unusual in that it does not ask for a project name first — the name and the save location are decided together later, when you save (Ctrl+S).
4. Check that the **MPINO** tab is the one open of the two tabs — **MPINO** / **MPAINO** — at the top of the modal (you switch tabs by clicking them, or with the ←/→ arrow keys while the modal has focus).
5. Click the **MPINO-8A4R(T)-S** card in the list to select it. This board has digital inputs P0–P7 and digital outputs P32–P35, which is an exact fit for this tutorial's P0 input / P32 output circuit.
6. Confirm by clicking the **Select** button or by double-clicking the card (Enter also confirms; Esc or a click outside cancels).



< Figure 1-5 The Select Board modal — with MPINO-8A4R(T)-S selected >

① **Board card** — Click to select the board you want to use

② **Select button** — Confirms the selection (double-clicking or Enter also works)

When you confirm the selection, nothing is written to disk; a new project with the code tab **main** and the empty ladder tab **ladderLoop** is created in memory only.

The board list is split into the two category tabs **MPINO** and **MPAINO**, and the two together hold fifteen boards. Besides this board, the MPINO tab has MPINO-8A4R-SU, MPINO-8A8R(T)-S, MPINO-8A8R, MPINO-16A16R, MPINO-16A8R, and MPINO-16A8R8T, and the MPAINO tab has eight boards, from MPAINO-8A8R through MPAINO-64A64R. Out in the field you simply pick the one that matches the board in your hand.

Select a board on the MPAINO tab and an **Option modules** section appears below the list as well. This is where you state how many of each of the four module kinds — **Analog input : X**, **Analog output : Y**, **PT100Ω input : F**, and **Pulse output : K** — to attach, with the [−]/[+] buttons. A kind you do not attach reads "none", and one you do reads as the prefix letter plus the count, such as "X1" or "Y2"; all four start at "none". Each kind has an upper limit (X 5, Y 4, F 5, K 2), and once a kind can go no higher its [+] button is disabled and hovering over it shows the reason as a tooltip. There are limits across the kinds as well — "Analog input (X) and PT100Ω input (F) share a budget of 5 modules total." and "Analog output (Y) and pulse output (K) are limited to 3×Y + 6×K ≤ 12 channels." When you attach pulse output (K) together with analog output (Y), or two pulse output (K) modules, the notice "Digital input terminals move from D2–D9 to D22–D29." is shown along with them — in those combinations K takes over the D2–D8 physical pins.

Below the list there is a **Serial RX Buffer** section as well. This is where you choose the receive buffer size — 64, 128, 256, or 512 bytes — for each UART channel the board has. It is collapsed by default and expands when you click its title row. If any channel has been changed from the board default, the changed channels alone are summarized next to the title even while it is collapsed, as in "Serial1 256 bytes"; if every channel is

at its default, no summary is shown. The section appears only on boards that support choosing a buffer, so it is not visible on the MPINO-8A4R(T)-S chosen for this tutorial.

1-4-1 Turning On the Pins You Will Use

There is one step you have to go through before you draw the ladder. **Every pin in a new project starts out as "not used".** Unless you turn on the pins you are going to use first, the build is rejected no matter how correctly you draw the circuit.

> **The default for a pin is "not used".** If you build while the ladder references a pin that is turned off, the build output shows the error "pin P0 (Din_1) is disabled in board definition (usepin = false). Enable it in the board pin editor or use a different pin." and the compile stops. This default is there to stop the firmware from overwriting, with 0 on every scan, output pins the ladder does not even use — which would otherwise hold down outputs you drive directly from your Arduino code.

The pins this tutorial uses are the input **P0** and the output **P32**. Turn them on as follows.

1. Press `Ctrl+Shift+P`, or click **Ladder Pin Map Settings** in the Settings menu. The pin editing table, which lists the board's pins one per row, opens.
2. Find the **P0** row and the **P32** row and tick the checkbox in the **Use** column. If you have many pins to turn on, you can also press **the checkbox in the header** of the **Use** column once to turn them all on, and off, at once (while only some of them are on, the header checkbox is shown in the indeterminate state).
3. Press the **[OK]** button to close the window.
4. Save the project with `Ctrl+S` — the pin settings are stored in the current project (`.mp2`), so you have to save for them to be written to disk.

Turning the pins on has one more side effect. **Only pins that are turned on appear among the autocomplete candidates** when you type an address into a ladder cell later on. If the autocomplete list looks empty in a new project, it means not a single pin has been turned on yet.

> **The same applies when you open a project you made earlier.** If the `.mp2` you saved before does not record any pin as turned on, it opens with every pin "not used". If a project that used to build fine is suddenly blocked by the error above, turn the pins it uses back on in the pin editing table and save.

The columns of the pin editing table and what each of them means are covered in detail in **Pin Editing**(2-4).

1-5 Editing the Ladder: From a P0 Contact to a P32 Coil

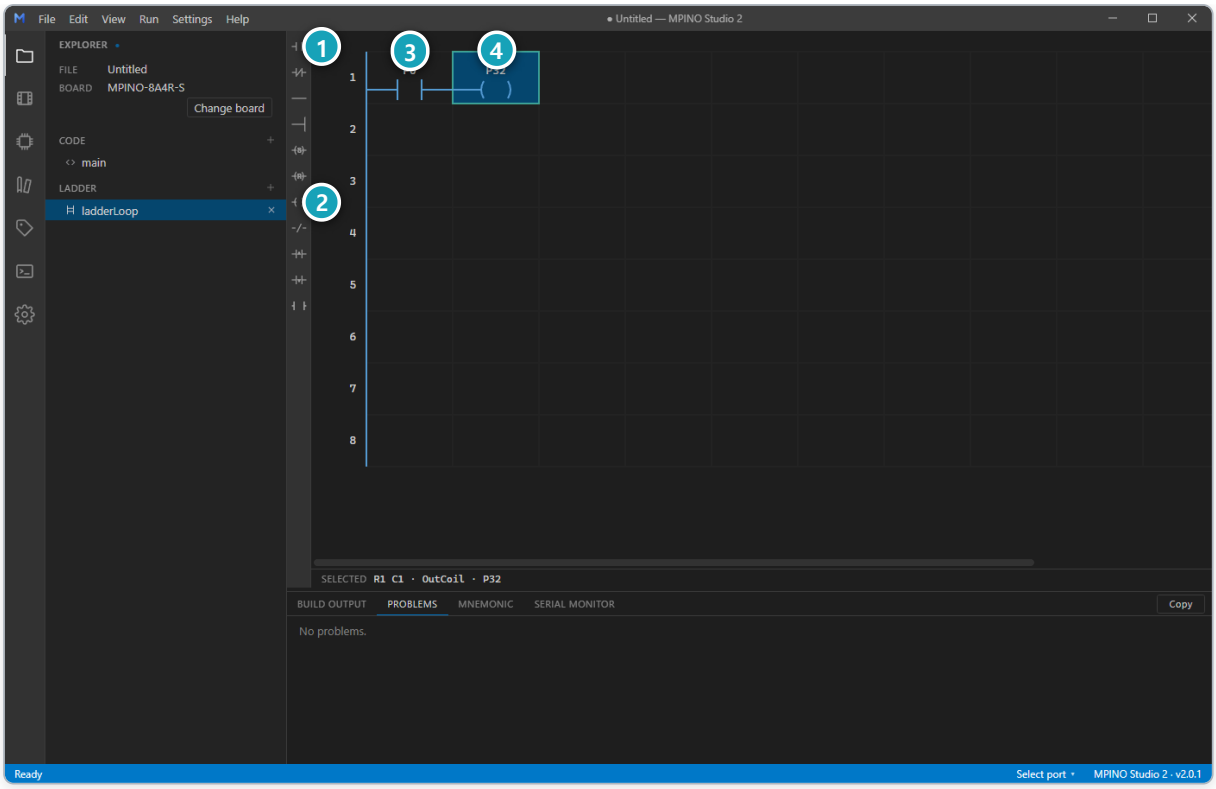
Now for the heart of it. You will draw the minimal circuit "when P0 turns on, P32 turns on" directly on the ladder grid. First, click the **ladderLoop** row under the **Ladder** group in the sidebar tab tree to move to the ladder editor.

The toolbar arranged vertically along the left side of the ladder editor lists eleven tools from top to bottom. The top ten map to the function keys (F2–F12) in order; only Box Function, at the very bottom, uses

Ctrl+Enter. With a single cell selected, clicking one of them or pressing its shortcut applies it to that cell immediately, with no separate "tool mode".

Shortcut	Tool	Symbol	Description
F2	NO Contact (A contact)	`┤ ├`	Normally open contact
F3	NC Contact (B contact)	`┤/├`	Normally closed contact
F5	Line	`──`	Horizontal line that passes the signal through unchanged
F6	Vertical	`|`	Toggles a vertical connecting line on the right of the cell
F7	Set Coil	`-(S)-`	Latches (holds) when the condition turns on
F8	Reset Coil	`-(R)-`	Un-latches when the condition turns on
F9	Output Coil	`-( )-`	Outputs the condition as it is on every scan
F10	Signal Invert	`-/-`	Inverts (`!`) the result of the preceding condition
F11	Positive Edge	`┤↑├`	Turns on for a single scan on the rising edge
F12	Negative Edge	`┤↓├`	Turns on for a single scan on the falling edge
Ctrl+Enter	Box Function	`[TON|T0|100]`	Timers (TON), counters (CTU), and the like — one column per parameter

1. Click the empty cell at row 1, column 1 of the grid to select it.
2. Click the **NO Contact** tool (F2) in the toolbar — the selected cell immediately turns into the NO contact symbol.
3. Click the next cell in the same row 1 (row 1, column 2) to select it.
4. Click the **Output Coil** tool (F9) in the toolbar — this cell turns into the output coil symbol.
5. Double-click the contact cell you just placed (row 1, column 1). The **Edit Contact** modal opens.
6. In the input field (placeholder text: "Address (P0), symbol, or comparison (D0 > 100)"), type "P0" and click **OK**.
7. Double-click the coil cell (row 1, column 2). A coil is also a cell that references a pin, so the same modal as for a contact opens — the modal title still reads "Edit Contact", but it really is the screen for editing the coil's value.
8. Type "P32" in the input field and click **OK**.



< Figure 1-6 The ladder toolbar and the finished circuit — a contact (P0) and an output coil (P32) placed in row 1 >

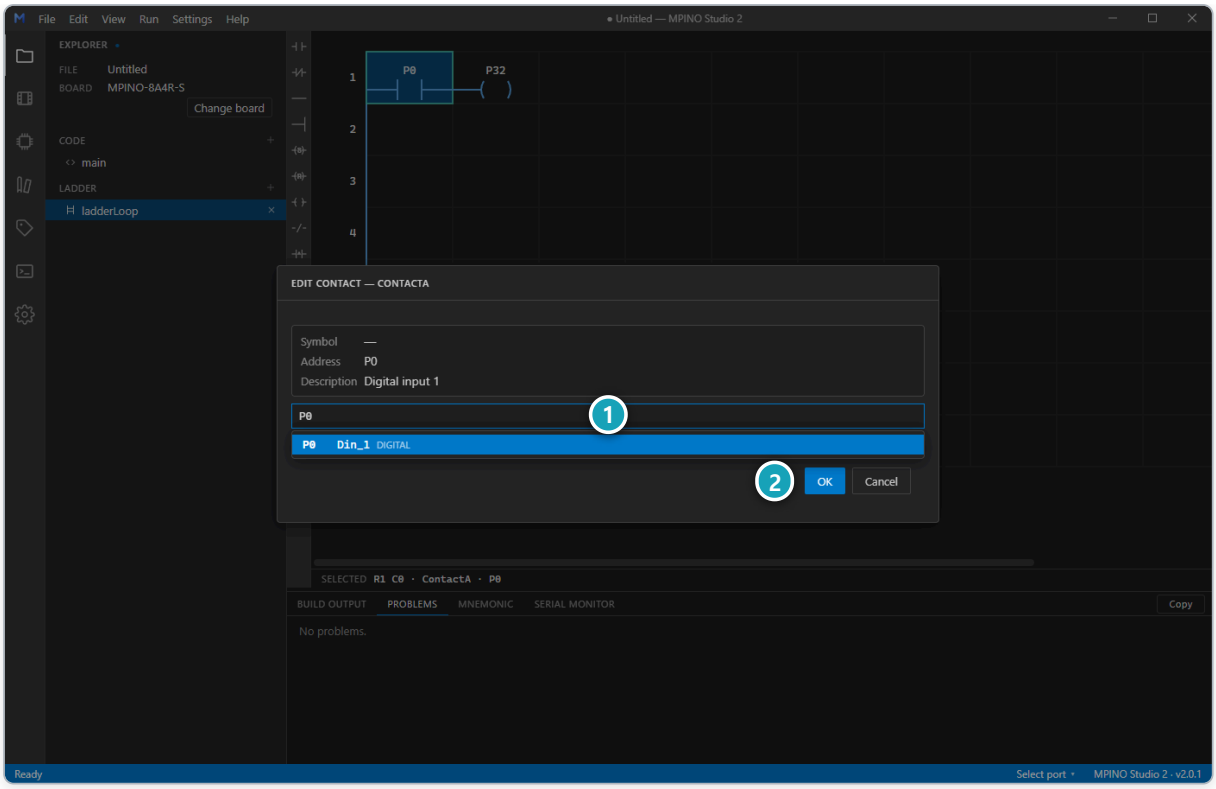
① **NO Contact tool** — Shortcut F2

② **Output Coil tool** — Shortcut F9

③ **P0 contact cell** — The place that tests the circuit's input condition

④ **P32 output coil cell** — The place that sends the result of the condition out to the relay output

The cell edit modal behaves the same whether you open it with the Enter key or by double-clicking. The modal title shows "Edit Contact" followed by the cell type (the internal type name, for example ContactA). Type a new address or symbol in the input field and press the **OK** button, and the value is committed and the modal closes. An address that has not been registered yet is still saved, and it resolves automatically once the symbol or the pin definition is filled in.



< Figure 1-7 The Edit Contact modal — entering P0 in the address field >

① **Address field** — The field where you enter what the cell will reference. It accepts all three formats

② **OK button** — Commits the entered value to the cell

This completes a circuit in row 1 that runs from `ContactA(P0)` to `OutCoil(P32)`. It is the simplest form of ladder example: "if P0 is ON, P32 is ON". The remaining tools, such as F6 (Vertical) and Box Function, become necessary in more complex circuits that branch several conditions or use timers and counters, so this minimal example does not cover them.

1-6 How a Ladder Becomes Arduino C Code

Let us take a moment to see how the circuit you just drew actually runs on the board. Through a separate conversion engine called `mpinoc`, MPINO Studio 2 breaks the ladder grid down into rungs (horizontal lines), turns them into expression trees, and generates code from them as a pair of functions inside a header file named `ladder_generated.h`. Because the ladder tab is named "ladderLoop", the functions generated here are `ladderLoopSetup()` and `ladderLoop()`.

The transpiler does not rewrite the `.ino` code you edit on your behalf. Open the **main** tab of the default project and you will find that a minimal sketch that calls `ladderLoop()` inside `loop()`, as shown below, is already prepared, so the very first build works without any extra setup (the auto-generated header `#include` lines at the top of the file are omitted here for convenience).

```
void setup() {
}

void loop() {
```

```
    ladderLoop();
}
```

Note that `setup()` is empty here. You do not write the ladder's initialization function `ladderLoopSetup()` yourself. Instead, at build time the transpiler puts it into the `setup()` of the sketch being compiled automatically (this injection happens only in the build output; the original code you see on screen and the `.mp2` file are left untouched). The `ladderLoop()` call inside `loop()`, by contrast, is not injected automatically — its call order is the user's intent, so you are responsible for it yourself.

> There is no need to write `ladderLoopSetup()` inside `setup()` a second time by hand. The transpiler does not filter out duplicates, so if you write it yourself the initialization function is called twice.

The overall flow runs as follows.

```
.mp2 project
  → mpinoc transpile
      → main.ino
      + ladder_generated.h
      + mpino_runtime.h/.cpp
      + board_pins.h
  → arduino-cli compile → .hex
  → arduino-cli upload → runs on the board
```

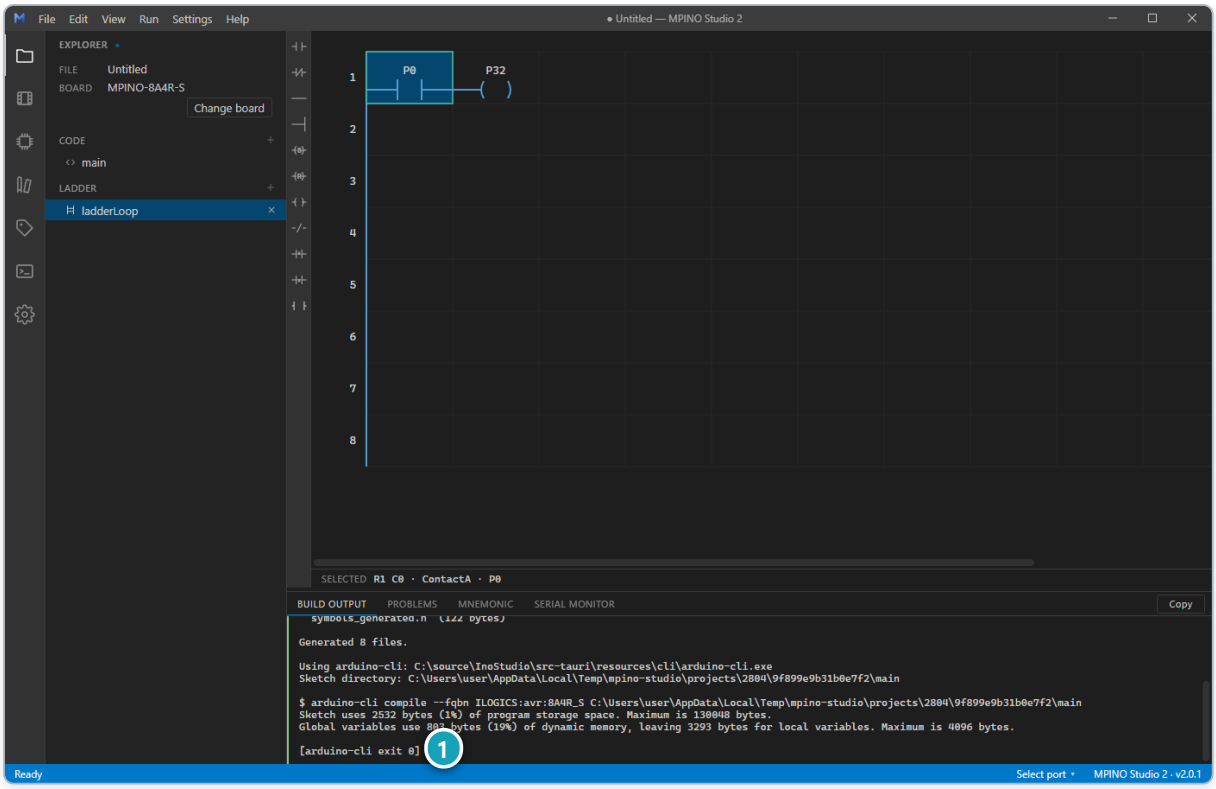
`mpinoc` was made a standalone module and CLI, separate from the GUI, in order to avoid the trap of the original IDE, which locked the transpiler inside the GUI and so made automation and CI verification impossible without it. You really can build straight from the command line without launching the app, in the form `mpinoc compile project.mp2 -o build/`.

1-7 Building

Once the circuit is drawn, try compiling it. Use the **Run > Build** menu item or the shortcut **Ctrl+R**.

> Ctrl+R is the same combination as the "Refresh" shortcut in a web browser, but inside MPINO Studio 2 this combination is intercepted and used only for **Build** (compile). There is no need to worry about the screen being refreshed.

1. Click **Build** in the **Run** menu, or press `Ctrl+R`.
2. The bottom panel switches to the **Build Output** tab automatically and shows the "Building..." log.
3. The process of arduino-cli actually compiling the transpiled sketch is printed as it happens.
4. If the last line reads `[arduino-cli exit 0]`, the build succeeded.



< Figure 1-8 The Build Output panel — a successful compile finishes with exit 0 >

① **Build succeeded (exit 0)**

The build panel shows the progress state as one of four values: Idle, Building, Success, and Failed. On failure a one-line summary error message is shown below the log as well, so you can see where it got stuck immediately, without scanning the whole log. Here the app distinguishes "the board core is not installed" from "the FQBN or the menu settings in the board definition are wrong" — for the latter it points out which item to look at, together with the sentence "This is not a hardware package installation problem."

> **The first build does not need an internet connection.** arduino-cli and the ILOGICS board core and toolchain used to compile are all shipped inside the installer, so you can build right after installing, even on a field PC with no internet.

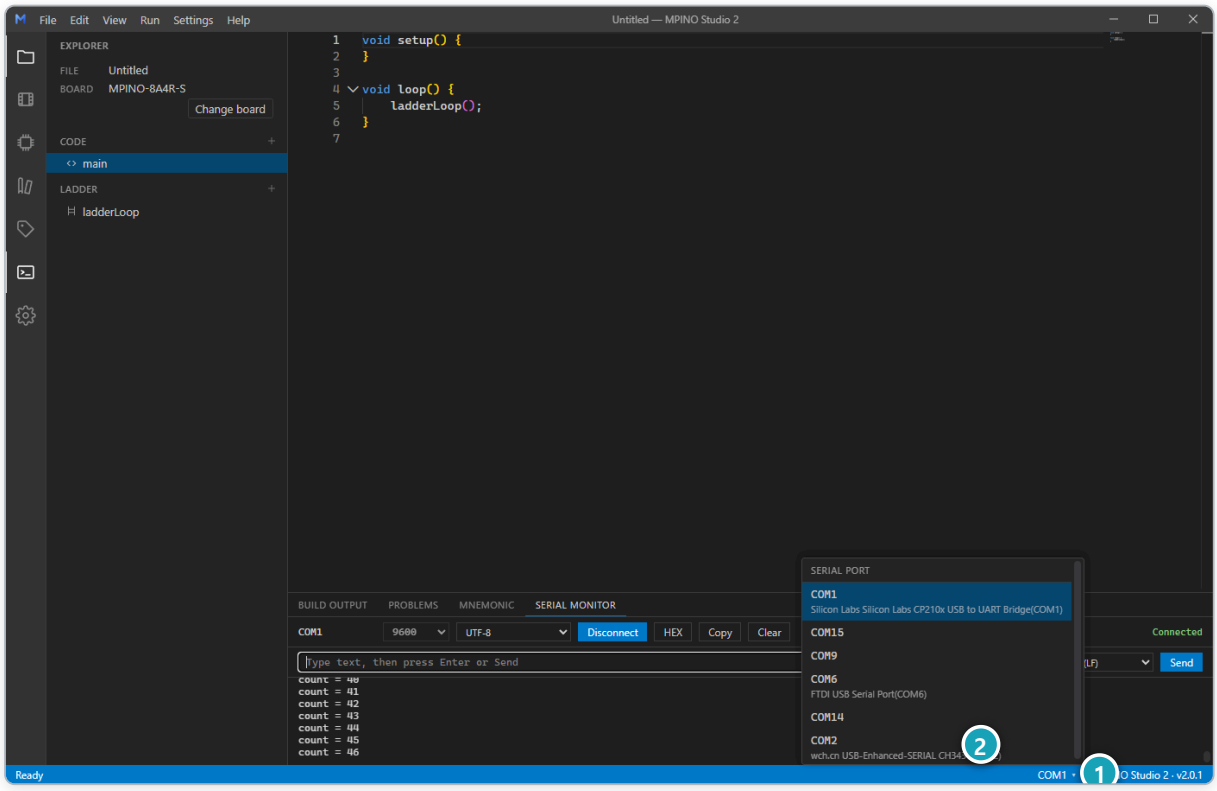
1-8 Selecting a Port and Uploading

With the build finished, it is time to put the program on a real board. You have to follow the order: choose the port first, then run the upload.

> Upload (Ctrl+U) stops immediately with a notice if no port is selected. Always select a port in the status bar first.

1. Click the port button on the right of the status bar (it reads "Select port" while no port is chosen, and shows the port name itself once one is).
2. The port list refreshes automatically the moment the dropdown opens (there is no separate refresh button).

3. Click the port you want to use (COM3, for example) in the list to select it.
4. Click **Upload** in the **Run** menu, or press `Ctrl+U`.
5. In the **Build Output** tab of the bottom panel, the upload log follows below the build log, preceded by a `--- Upload ---` divider.



< Figure 1-9 The status bar port dropdown — the only way to select a serial port >

① **Status bar port button** — The button that opens the port dropdown. The button label itself also shows which port is currently selected

② **Port list item** — The port you pick here is used both for uploading and for connecting the Serial Monitor

The status bar dropdown is the only way to select a port. While an upload is in progress, a **Cancel upload** button appears on the right of the panel, and if you request a cancel it changes briefly to "Cancelling...".

When you start an upload, if something else is also using the port ladder monitoring is set to use, the upload does not begin straight away: a confirmation window carrying the port name — such as **"Serial1 already in use"** — appears first. Its body takes one of three forms, depending on what it collides with.

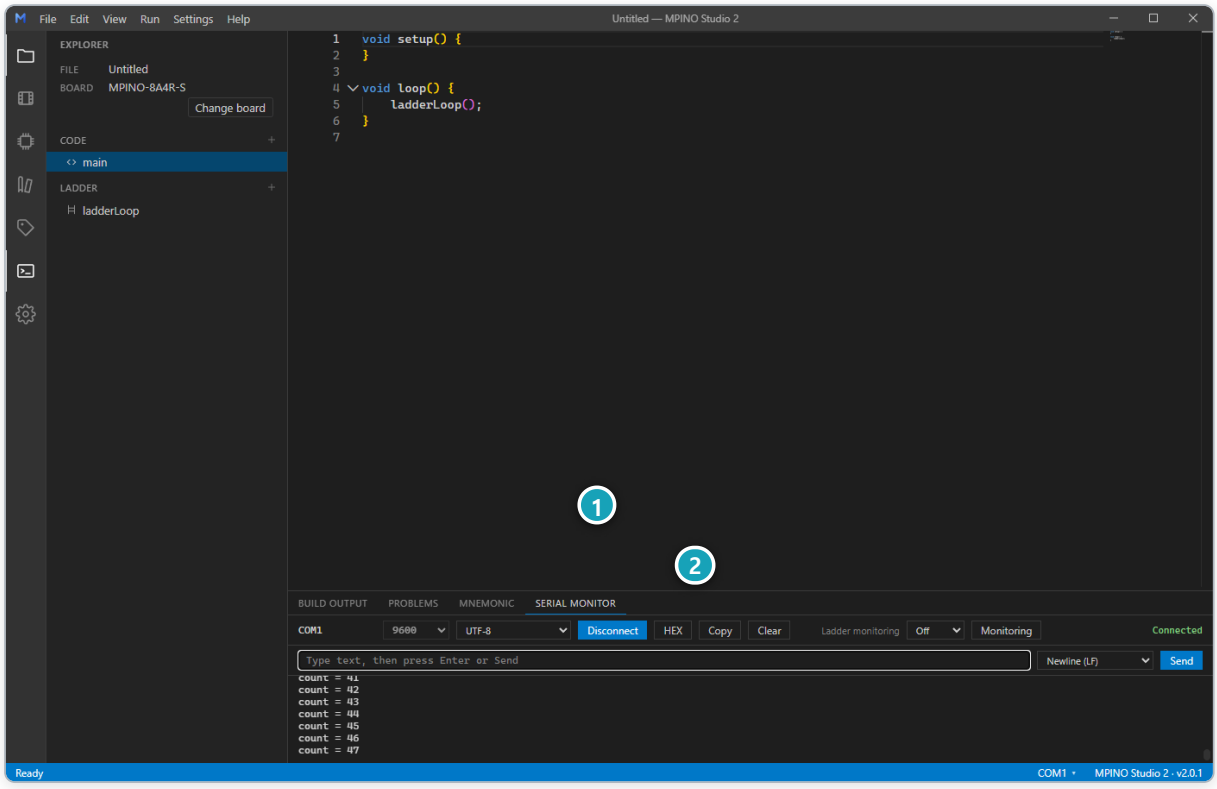
What it collides with	What it tells you
A channel in the communication settings (Run › Communication)	Set that channel's protocol to **None**, or choose a different monitoring port
A communication init call in your Arduino code	Pass a different port to that call in your code (it shows you the function name as it is)
Your Arduino code using that port directly	Use a different port in your code, or change the monitoring port

In all three cases this is **a warning, not a block**. Press [Upload] and it goes ahead as it is; press [Cancel] and the upload does not run.

1-9 Verifying Operation: Serial Monitor and Live Monitoring

Finally, you check how the board actually behaves in two ways. One is the **Serial Monitor**, which shows the raw strings going back and forth over the serial port; the other is live monitoring, which shows the conducting state of the ladder circuit directly on screen. The two are different features, so they can be turned on and off independently.

1. Click the **Serial Monitor** icon in the Activity Bar — it goes straight to the **Serial Monitor** tab of the bottom panel, not to the sidebar.
2. Check the baud rate (115200 by default, chosen from 16 values in total) and the encoding (5 options including UTF-8) in the header.
3. Click the **Connect** button. Once connected, the button label changes to "Disconnect" and is shown in the accent color.
4. Received strings pile up in the log area as they are. You can change the display format with the ASCII/HEX button.



< Figure 1-10 The Serial Monitor — checking raw strings along with the baud rate and encoding after connecting >

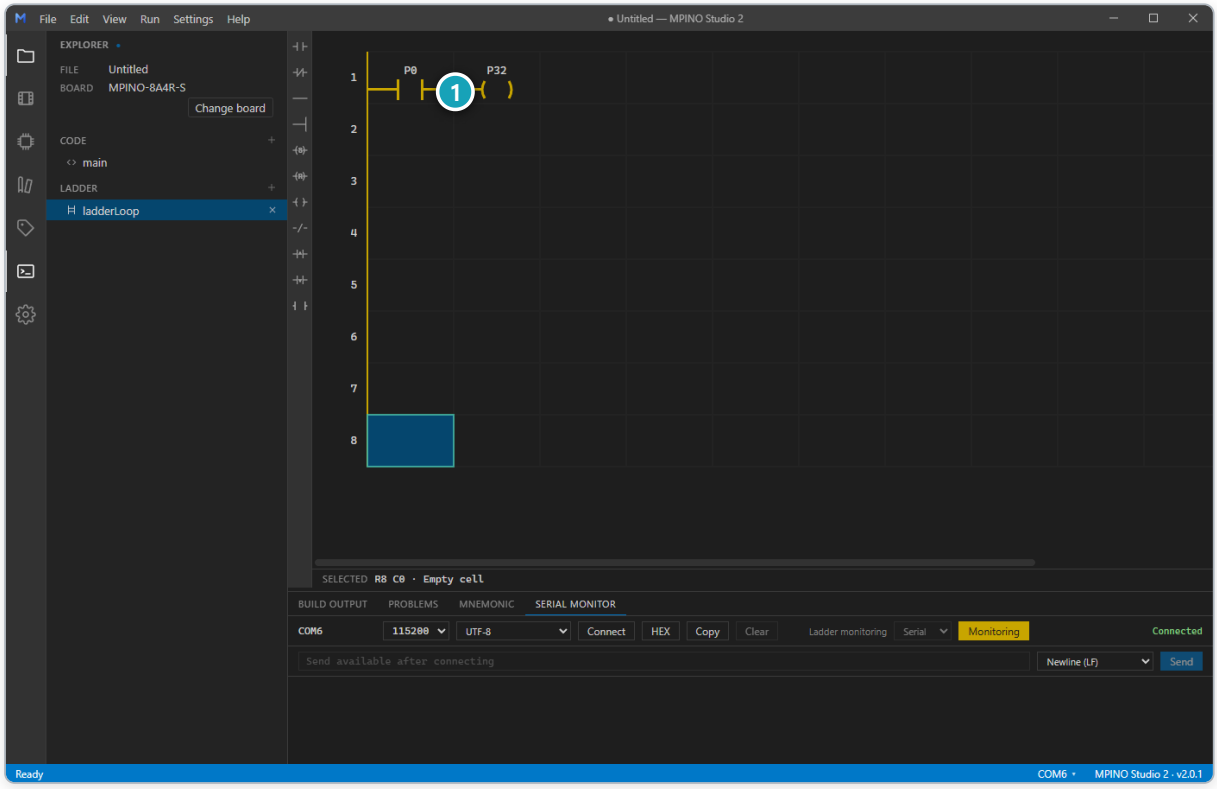
① **Connect button** — A toggle that turns the serial connection on and off

② **Monitoring button** — A toggle that turns the live ladder power-flow display on and off. It works independently of the Connect button beside it

> The **Monitoring** button is a separate switch that turns on live ladder monitoring (the power-flow display). The first time you press it a notice modal appears, and its exact wording is: "Monitoring requires a serial port. Set it in Preferences → Ladder Settings → Monitoring Port." The place it refers to is not the **Ladder Settings** side panel in the Activity Bar that you saw in "A Tour of the Screen" (1-2), which is the read-only view of the pin map and memory areas, but the **Ladder Settings** tab inside the **Settings** (Preferences) modal — in English the two carry exactly the same name even though they are completely different screens, so take care not to confuse them. That tab offers the radio options **Off / Serial / Serial1 / Serial2**, but it keeps only **the UARTs the current board actually has** — the MPINO-8A4R(T)-S used in this tutorial has just two UARTs, so only "Off / Serial / Serial1" are shown. Once you pick one (Off means it is turned off), the firmware actually transmits monitoring data from the next build/upload onward.

You can make the same choice from the Serial Monitor header as well. The **Monitoring channel** dropdown in the **Ladder monitoring** group on the right of the header carries the same value as the radio buttons in Preferences, so it makes no difference which of the two you change (hover over the dropdown and the description "The channel the firmware sends monitoring data over. Upload after changing to apply it to the board." appears).

Once the monitoring port is set, upload the firmware that contains your circuit and then press the **Monitoring** button again. `Ctrl+M`, or **Monitoring ON** in the ladder grid context menu, turns on the same behavior. Contacts and coils that are conducting change to the accent color and their lines thicken so that they stand out — but for this highlight to actually appear, a person has to apply a physical signal to the board's P0 input.



< Figure 1-11 Live monitoring — when a signal arrives at P0, the P0 contact and the P32 coil are highlighted in the power-flow color >

① **Power-flow highlight** — Contacts and coils that are conducting are shown in the accent color with thickened lines (it appears only once an actual signal is applied to the P0 input)

To turn monitoring off, press the **Escape** key while the ladder grid has focus and it stops immediately.

1-10 Shortcuts Used in This Tutorial

Shortcut	Command
Ctrl+N	New
Ctrl+Shift+P	Ladder Pin Map Settings (turning pins on)
Ctrl+S	Save
Ctrl+R	Build
Ctrl+U	Upload
Ctrl+M	Toggle Monitoring
F2	Place NO Contact
F9	Place Output Coil

That is the first MPINO Studio 2 project, from start to finish. You created a new project, chose a board, drew a minimal circuit with one contact and one coil, built it and put it on a real board, and confirmed with your own eyes that it works. Later chapters cover how to combine several contacts, box functions such as timers and counters, and the settings for memory areas and communication protocols.

2 Essential Reading

2-1 Data Memory

Memory is the storage space that ladder circuits and Arduino C code read values from and write values to. MPINO Studio 2 divides that storage space into six areas with different characters, and addresses are written as a letter identifying the area followed by a decimal index (`P0`, `M10`, `D0`, and so on).

Area	Type	Size	Example addresses	Use
P	Port (bit)	1 bit	P0, P32	Digital input/output
M	Bit	1 bit	M0, M10	Internal bit relay (auxiliary contact)
D	Word	16 bits	D0, D10	General-purpose data (integers)
C	Counter	16 bits	C0, C1	For counter box functions (also usable as an ordinary word)
T	Timer	16 bits	T0, T1	For timer box functions (also usable as an ordinary word)
R	Float	32 bits	R0, R1	Floating point (IEEE754 single precision)

P is the digital input/output area connected to the board's physical pins. **M** is an internal bit relay used only inside the circuit, unrelated to the board pins; you use it to hold a contact value for a moment or to link several rungs together. **D** is a general-purpose word area that holds integer values. **C** and **T** are reserved for counter and timer box functions respectively, but the remaining addresses that you do not use in box functions can also be used as ordinary words, just like D. **R** is a 32-bit float area (IEEE754 single precision) that holds values with a decimal point.

> **Whether a P address is an input or an output is not fixed — the pin direction (IN/OUT) of the board decides it.** On the MPINO-8A4R(T)-S used in Chapter 1, for example, P0–P7 are inputs and P32–P35 are outputs, but that is only how that board's definition (`boards/*.toml`) laid them out; it is not a rule the app enforces, such as "P0–P31 is always an input". Choose a different board and the very same P0 may be an output rather than an input.

> **R (float) carries floating-point error.** Even if you store 7.1, for example, the actual value may be off by a tiny amount, such as 7.10001, so allow for that margin of error in conditional expressions that have to compare R values exactly.

2-1-1 Value Ranges

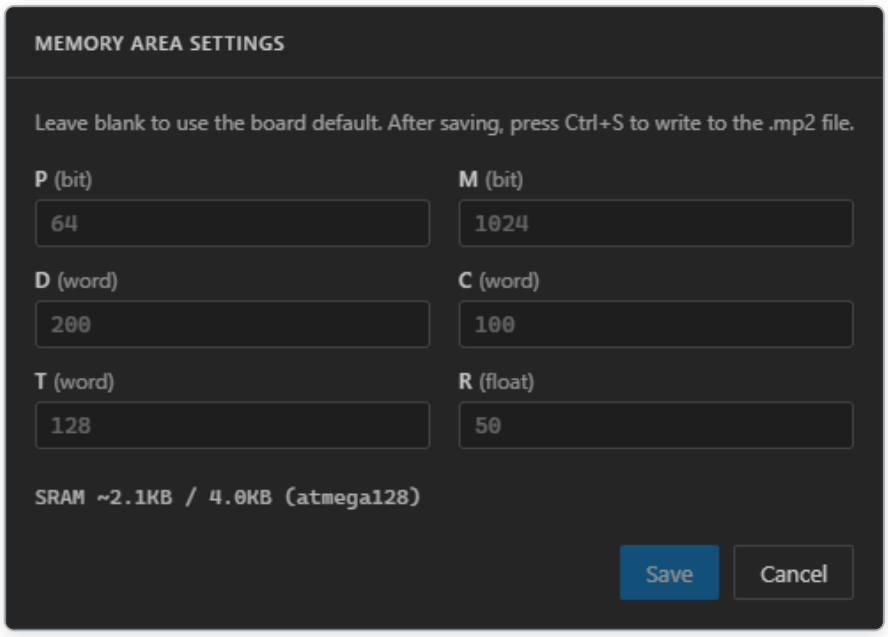
Besides the native width of each area, memory is also classified into the following four units according to the range of values it can hold.

Unit	Value range	Composition
BIT	0 or 1	The smallest unit of memory
BYTE	−128 to +127	8 bits

Unit	Value range	Composition
WORD	−32,768 to +32,767	2 bytes
DWORD	−2,147,483,648 to +2,147,483,647	2 words

2-1-2 Area Sizes and Memory Settings

The number of addresses in the six areas (how many P, how many M, and so on) has a different default on each board. On the MPINO-8A4R(T)-S, for example, the defaults are P 64, M 1024, D 200, C 100, T 128, and R 50. To adjust those defaults per project, open the **Memory Area Settings** modal (**Ctrl+Shift+M**). Leave any of the six fields blank and the board default applies as it is; fill in a value and it overrides the default. The bottom of the modal shows the total SRAM usage for the current values in real time, and **if the total exceeds the chip's SRAM capacity it is highlighted in red and you cannot save in that state.** Press Save and the values take effect immediately; after that, **Ctrl+S** writes them to the project (`.mp2`).



< Figure 2-1 The Memory Area Settings modal — adjust the number of addresses in the six areas and check SRAM usage at the bottom >

2-1-3 Prefixes and Bit Access

Besides the default (native) unit, you can attach a prefix to read the same address at a different width, or attach `.n` to designate one specific bit.

Access	Example notation	Areas allowed
Default (native)	P0, M0, D0, C0, T0, R0	All
B — byte view	BP0, BM1	P/M
W — word view	WP0, WM1	P/M
D — double-word view	DM0, DD2, DC31	P/M/D/C/T
.n — bit view	D0.0, D0.15, WM1.3	P/M width views, D/C

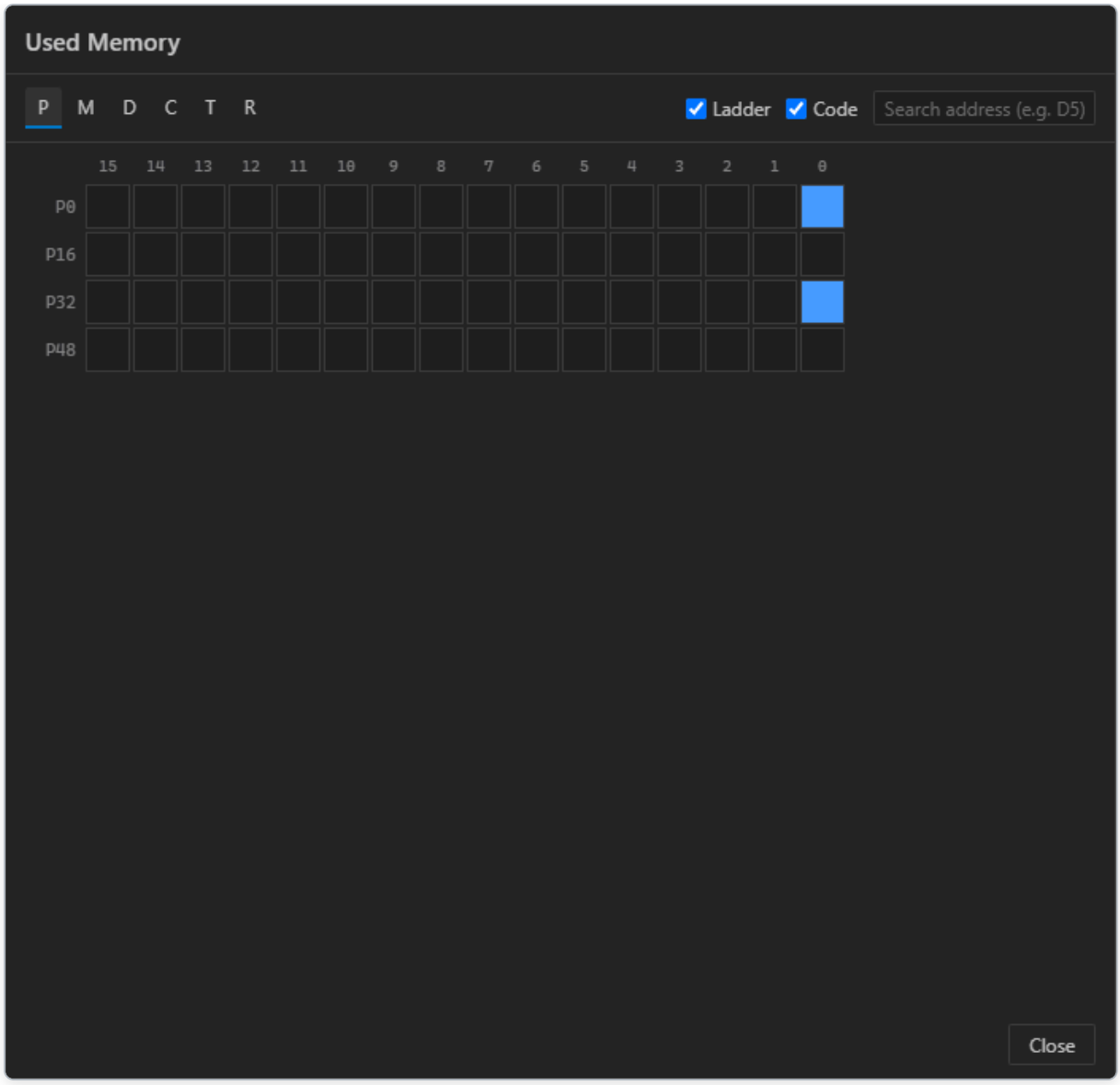
> **The byte (B) and word (W) views can only be used on the P and M areas.** If you attach a B or W prefix to the D, C, or T areas — as in `BD0` (a notation for viewing the D area as bytes) from the examples in the original manual — it cannot be used in this app (the only prefix you can attach in D, C, and T is the double-word `D` prefix). R (float) supports neither width prefixes nor `.n` bit access.

2-1-4 Checking Used Memory

To see which addresses the current project actually uses, open the **Used Memory** modal (**Ctrl+Shift+L**). The P/M/D/C/T/R tabs at the top let you switch between areas, and cells that are in use are marked with a color on the grid. Click a cell that is in use and a list of the ladder cells and code lines that reference that address appears; the Go to button beside each entry takes you straight to that location.

The modal also supports the following three things.

- Address search box: type an address into the field in the upper right of the modal (placeholder "Search address (e.g. D5)") and press **Enter**, and the view switches to the area tab that address belongs to and the square in question is highlighted. It does not filter the list; it navigates to the address and highlights it, so the other squares in the grid remain visible. Case does not matter, the format must be an area letter (P/M/D/C/T/R) followed by a number, and if the format does not match, nothing happens at all. Clear the search text and the highlight is released.
- Ladder/Code checkboxes: the **Ladder** and **Code** checkboxes above the grid let you show or hide cells in use by source (whether they are used in the ladder circuit or in the C code). Both are on by default.
- Tab and scroll position are preserved: the selected tab and the grid scroll position stay as they were even if you close the modal and open it again (they are reset if you quit the app and launch it again, though).



< Figure 2-2 The Used Memory modal — check the addresses actually in use in the project on a per-area grid >

2-2 Special Memory

Unlike ordinary memory, where a person writes the values in, a special contact is a special bit whose ON/OFF state the app updates automatically as time passes. You write a notation beginning with `@` instead of an address, and you can place it **only in a contact (input) cell or an edge cell** — you **cannot place it in an output coil cell**. A special contact is, after all, an input-side expression that builds a condition, not an output-side expression that emits a value.

Only the following two kinds of special contact are supported.

Notation	Meaning	Examples
`@<ms>`	one-shot — ON for only the single scan that crosses the specified millisecond (ms) boundary	`@100` (once every 100ms), `@1000` (once every second)
`@F<ms>`	cyclic toggle — repeats ON for the given ms, then OFF for the same ms	`@F1000` (blinks 1 second ON / 1 second OFF)

You write the millisecond number immediately after `@`, as in `@100`. Even if the scan cycle is slower than the specified millisecond boundary and so skips the exact instant, the one-shot (`@<ms>`) detects on the next scan that the boundary was crossed and fires exactly once, so no pulse is missed.

Example 1 — counting values with a one-shot

Connect an `@100` contact to the `INC D0` box function (or `D0 = D0 + 1`) and the condition turns on for exactly one scan every 100ms, so D0 increases by 1.

Example 2 — blinking

Connect an `@F1000` contact to output coil P32 and P32 repeats turning on for one second and off for one second.

< Figure 2-3 A special contact example — a circuit that blinks an output coil at one-second intervals with an @F1000 contact >

> **The notation differs from the old MP STUDIO.** This app writes it without parentheses, as in `@100` and `@F100`, and the unit is ms (not the old `@(n)` parenthesized notation, 10ms units, and limit of eight kinds).

A condition that is "always ON" or "always OFF" is written as a **constant contact**. Enter `@ON` or `@OFF` (or `TRUE` or `FALSE`) in an NO or NC contact cell and that contact conducts permanently, or blocks permanently, respectively. A rung that runs straight to a coil with no input contact also executes on every scan and so means the same thing, but a constant contact makes the intent — "this place is deliberately held on" — visible in the circuit. For how to use them in detail, and for their restrictions, see **Constant Contacts (@ON / @OFF)**(5-2-3).

2-3 Keyboard Shortcuts

This is the complete list of MPINO Studio 2 shortcuts, organized by function.

File

Shortcut	Command
Ctrl+O	Open Project
Ctrl+S	Save
Ctrl+Shift+S	Save As
Ctrl+N	New project (select board)
Ctrl+P	Print (code if a code tab is active, the ladder diagram if a ladder tab is)
Ctrl+Shift+M	Memory Area Settings

Shortcut	Command
Ctrl+Shift+D	Ladder Design Settings
Ctrl+Shift+P	Ladder Pin Map Settings
Ctrl+Shift+L	Used Memory (memory map)
Ctrl+Shift+E	EEPROM Retain Settings

View

Shortcut	Command
Ctrl+B	Toggle Sidebar
Ctrl+J	Toggle the bottom panel
Ctrl+M	Toggle live monitoring

Build and Upload

Shortcut	Command
Ctrl+R	Build (compile)
Ctrl+U	Upload

Ladder Navigation and Selection

Shortcut	Command
↑ ↓ ← →	Move the cell selection
Shift+ ↑ ↓←→	Extend the range (rectangular) selection
Delete	Clear the selected cell
Ctrl+ ↑ / Ctrl+ ↓	Move the active sidebar item up/down

Ladder Tools

Shortcut	Tool
F2	NO Contact
F3	NC Contact
F5	Horizontal Line
F6	Vertical Line
F7	Output Coil SET
F8	Output Coil RESET
F9	Output Coil

Shortcut	Tool
F10	Signal Invert
F11	Positive Edge (rising)
F12	Negative Edge (falling)
Ctrl+Enter	Box Function (TON, CTU, etc.)

Ladder Clipboard and Editing

Shortcut	Command
Ctrl+C / Ctrl+X / Ctrl+V	Copy / Cut / Paste cells
Ctrl+Z	Undo Ladder
Insert	Insert Cell
Ctrl+Delete	Delete Cell (shifts the rest left)
Alt+Insert / Alt+Delete	Insert Row / Delete Row
Ctrl+F	Find / Replace in Ladder
Ctrl+Shift+W	Force write to memory

> **Some shortcuts behave differently depending on where you are.** The arrow keys (↑ ↓ ←→) move the selected cell on the ladder editing screen, and move the text cursor in the code editor. The ladder tools (F2–F12), Shift+arrow keys, and Ctrl+C/X/V/Z/F, meanwhile, do what the tables above describe **only while the ladder grid has focus**; while the code editor has focus they act as ordinary text-editing shortcuts. In addition, **Enter or a double-click** opens the edit window for the selected ladder cell (a cell operation, not a global shortcut), and **Esc** (while the ladder grid has focus) turns live monitoring off.

2-4 Pin Editing

The phrase "pin editing" refers to two different screens in this app. One is **Board Pin Map Editing**, the screen where you change the board hardware's pin definitions themselves. The other is **Ladder Cell Editing**, the screen where you enter the value (an address, a symbol, and so on) that an individual ladder cell will reference. The names are similar, but what they change is completely different, so this manual keeps them apart.

2-4-1 Board Pin Map Editing

Press **Ctrl+Shift+P** (Ladder Pin Map Settings), or press the **[Edit Pins...]** button in the Select Board window, to open it. You edit each pin defined on the board in a table. The table's columns are as follows.

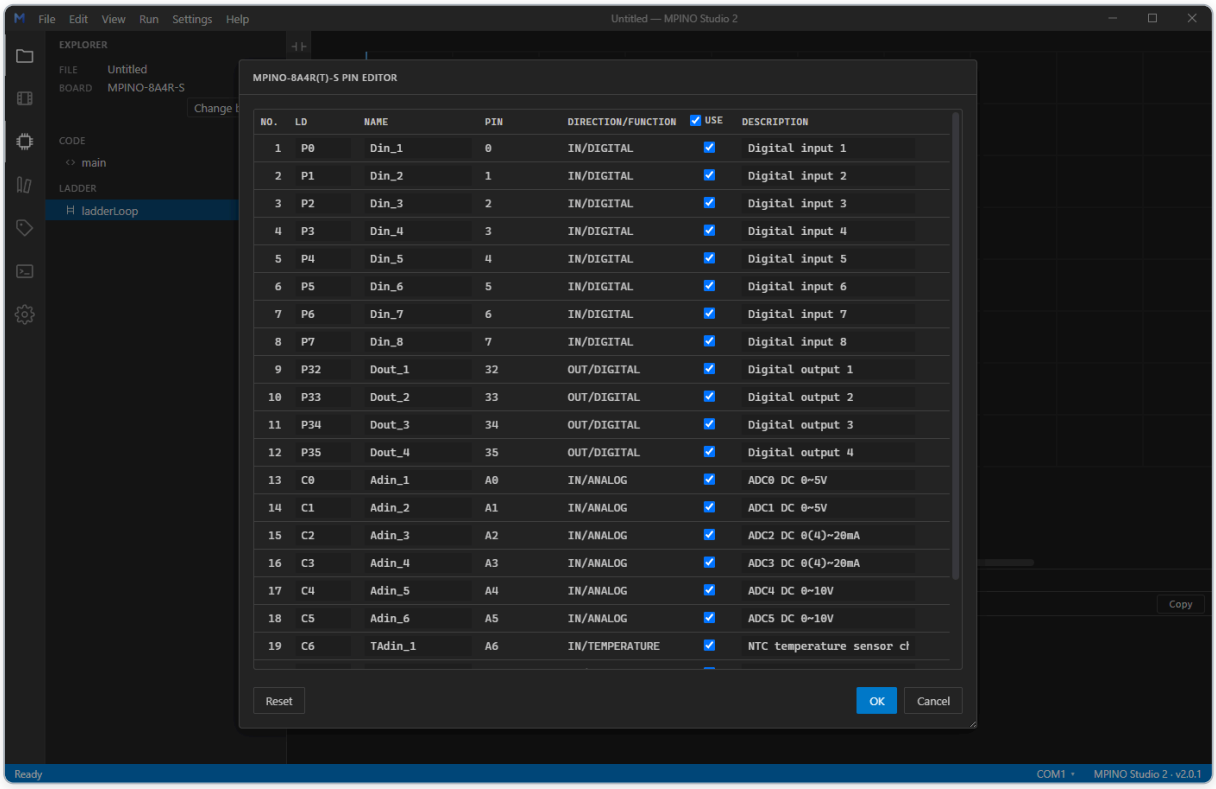
Column	Meaning	Editing
No.	The row number in the table	Read-only
LD	This pin's LD address (P0, for example). The area and number can be edited freely	Editable

Column	Meaning	Editing
Name	The pin's display name	Editable
Pin	The actual Arduino pin number (an integer, or an alias such as `A0`)	Read-only
Direction/Function	IN/OUT (direction), DIGITAL/ANALOG/TEMPERATURE/PULSE (function)	Read-only
Use	Whether to use this pin in the ladder (a checkbox)	Editable
Description	Free text	Editable

Because this app is dedicated to ILOGICS industrial boards, Pin and Direction/Function are values fixed in the hardware and are therefore shown **read-only**, and there is no way to add a new row or delete an existing one either — you can edit only four fields: LD, Name, Use, and Description.

The **Use** column has a checkbox in its header as well. Press that checkbox to turn every pin in the table on, or off, at once; while only some of them are on, it is shown in the indeterminate state.

> **The default for a pin is "not used".** A newly created project has every pin turned off, and a project you made earlier that does not record any pin as turned on likewise opens with all of them off. If the ladder references a pin that is turned off, the build stops with the error "pin P0 (Din_1) is disabled in board definition (usepin = false). Enable it in the board pin editor or use a different pin." And since **only pins that are turned on appear among the autocomplete candidates** in the ladder cell edit window, if no candidate at all comes up, turn the pins you intend to use on in this table first.



< Figure 2-4 Board pin map editing — Pin and Direction/Function are read-only; only LD, Name, Use, and Description can be edited >

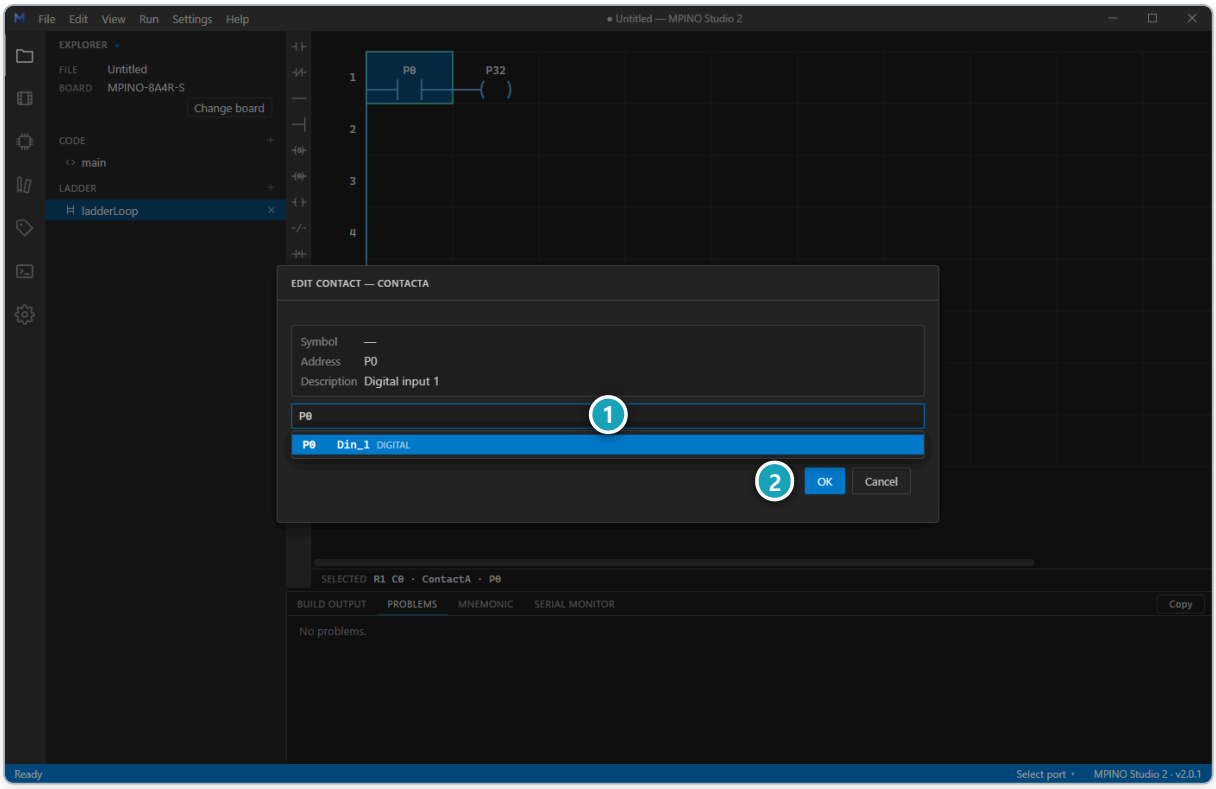
Below the table there are three buttons: [Reset], [OK], and [Cancel]. [Reset] returns the pins to the factory default configuration (it reverts only the editable values — LD, Name, Use, Description, and so on; and since pins themselves cannot be deleted in the first place, there is nothing to bring back even if you think you deleted one earlier), [OK] actually applies the changed values, and [Cancel] closes the window without any change. [Reset] asks once more before it reverts anything: "Reverts the pin mapping to the factory defaults. … Continue?"

> **This editing is stored in the current project (`.mp2`), not in the board definition file.** Press [OK] and the changes are reflected in the project that is open; the moment they are written to disk is when you save with **Ctrl+S**. The board definition file (`boards/<board>.toml`) is a read-only set of factory defaults and does not change, so **other projects are not affected at all.** To use the same pin settings in another project, open that project and set it up the same way.

2-4-2 Ladder Cell Editing

It opens when you **double-click** a contact, edge, or coil cell in the ladder grid, or select one and press **Enter**. In this window you enter the **address, symbol, or special contact value** that this one cell will reference (`P0` , a symbol name, or `@F1000` , for example). As you type, matching pins, symbols, and timer/counter done bits appear as autocomplete candidates, and you can pick one from the list to fill the field immediately. Only pins you have turned on with **Use** in Board Pin Map Editing appear among the pin candidates.

The board definition is left untouched; **only that cell's value** changes. An address or symbol that has not been registered yet is still saved, and it resolves (connects) automatically once that pin or symbol is defined.



< Figure 2-5 Ladder cell editing — enter the address or symbol the cell will reference (the same window as the Edit Contact modal seen in Chapter 1) >

① **Address field** — The field where you enter what the cell will reference. It accepts all three formats

② **OK button** — Commits the entered value to the cell

3 Menus and Toolbar

The main menu of MPINO Studio 2 consists of the six menus in the menu bar at the top (File, Edit, View, Run, Settings, and Help) and the items inside each of them. This chapter explains what each menu does and how to use it.

3-1 File

The File menu consists of seven items for creating, opening, and saving a project, for printing code and ladders, and for reaching the app's preferences or quitting the program.

Item	Shortcut	Description
New	Ctrl+N	Creates a new project after you select a board
Open	Ctrl+O	Loads an existing `.mp2` project file
Save	Ctrl+S	Saves the current project
Save As	Ctrl+Shift+S	Saves after you specify the location and file name again
Print	Ctrl+P	Prints the contents of the active tab (code or ladder)
Preferences	—	Opens the Preferences modal on the "General" tab (see the Preferences section below for details)
Quit	—	Closes the program window and exits the app

New (Ctrl+N) first brings up a dialog asking whether to discard any unsaved changes in the project you were editing, and then the board selection window opens. For the whole procedure, see **Creating a Project and Selecting a Board**(1-4).

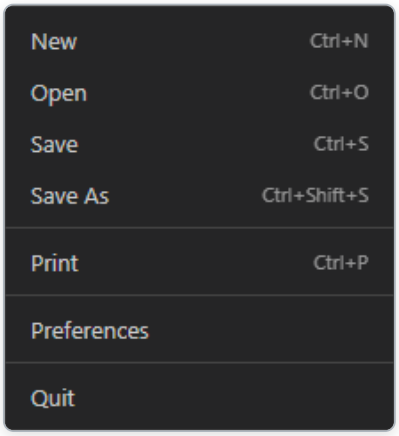
Open (Ctrl+O) also shows the same confirmation dialog first if there are unsaved changes, exactly as New does. The OS file open dialog then opens, and you select a project file from a list filtered to the `.mp2` extension. You can also skip the menu and **double-click a `.mp2` file in the file explorer** — the `.mp2` extension is associated when the app is installed, so double-clicking one launches the app with that project already open.

Save (Ctrl+S) overwrites the path directly if the project already has a path it was saved to once. For a project that has never been saved yet — right after New, for example — it moves on to the "Save As" procedure automatically.

Save As (Ctrl+Shift+S) always asks again for the save location and file name every time you run it. The newly decided file name is then **synchronized automatically with the name of the main code tab** — the main tab name must always match the project file name (if another code tab already has the same name, the save is rejected).

Print (Ctrl+P) decides what to print based on **the type of the currently active tab**, not on what you see on screen. If a code tab is active, all of the project's code (.ino) tabs are collected in tab order starting from main and printed with each tab starting on a new page (if there is only one code tab, only that one is printed). If a

ladder tab is active, all of the project's ladder circuits are printed as static diagrams (SVG). If there is no code or ladder to print (a project with no ladders at all, for example), nothing happens.



< Figure 3-1 The File menu — the dropdown showing all seven items from New to Quit >

> **Quit closes the window immediately without asking whether to save.** If you have unsaved changes, save first with Ctrl+S (or Ctrl+Shift+S) before quitting.

3-1-1 Preferences

You open Preferences with the **Preferences** item (no shortcut) in the File menu, and opening it this way always lands on the **General** tab. You can also open it with the ⚙ icon in the Activity Bar (the Activity Bar in detail is outside the scope of this chapter). The top of the modal has four tabs (**General** / **Shortcuts** / **Ladder Settings** / **Ladder Design**), and you switch between them by clicking a tab or, while the modal has focus, with the ←/→ arrow keys.

General

Item	Description
User Library Folder	The location used by library install, import, and open-folder actions. The current path is shown in code formatting
Platform Folder	The ILOGICS board core and toolchain folder. Build and autocomplete reference this folder
Language	Three radio options: 한국어 / English / 日本語
Error reporting	A checkbox for whether reports are sent automatically (on by default)

For **User Library Folder** you designate a new folder with **[Browse]**. Once you pick one in the OS folder selection dialog, the path is re-pointed there, and **existing library files are not moved** (the same way changing the sketchbook location works in the Arduino IDE — it is a re-pointing, not a move). **[Restore Default]** returns the folder to its default location, and this button is disabled if it is already at the default. When either action succeeds, a library refresh runs automatically once (re-syncing the vendor channel, re-querying the installed list, and restarting the autocomplete engine), and the following notice appears.

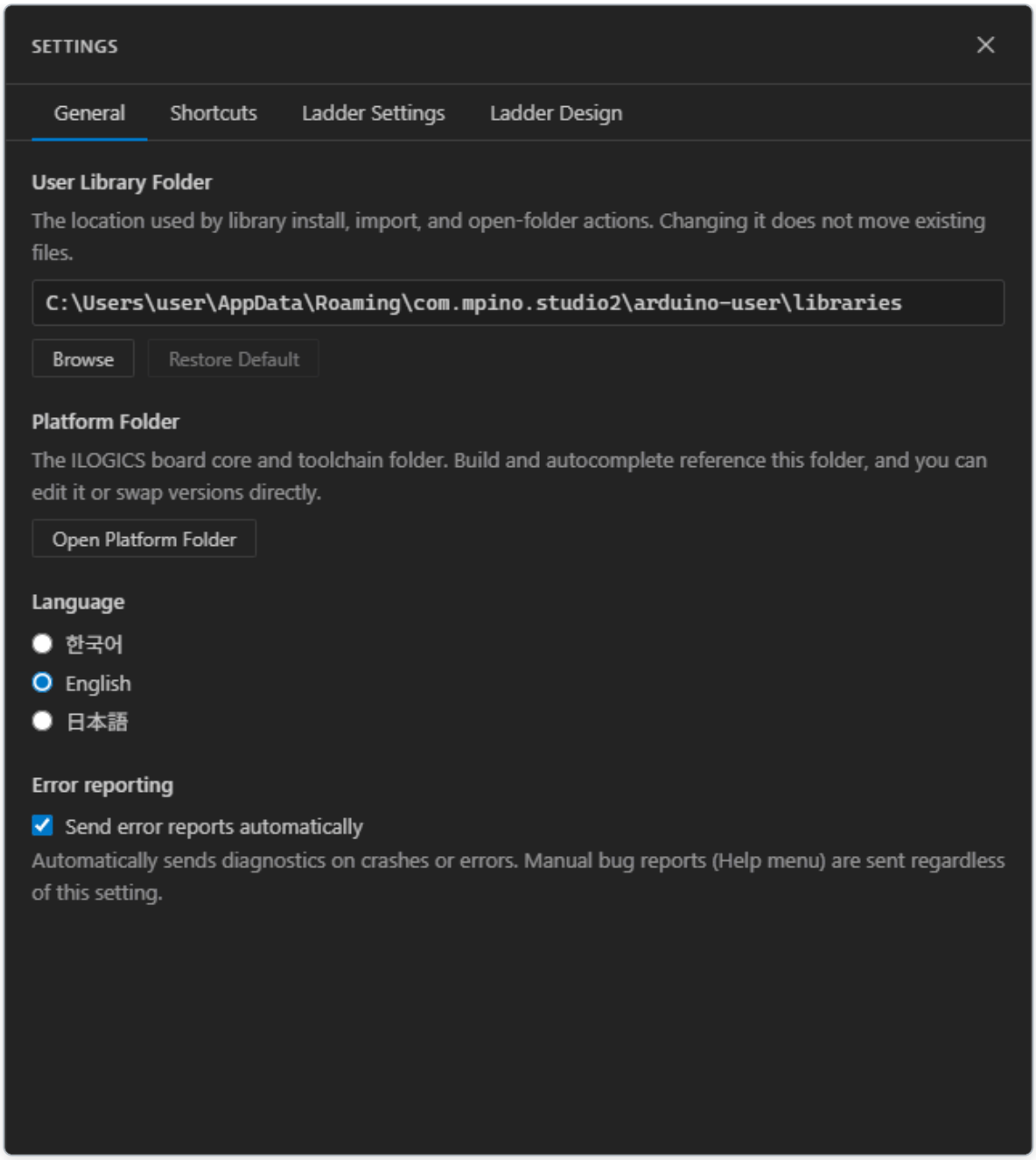
> **"Refreshing libraries — updating autocomplete (warm ~3s, cold up to ~1 min)"** — right after a refresh, autocomplete may be slow or look empty for a moment. This is normal behavior, and it recovers

> if you wait a little.

For **Platform Folder**, pressing **[Open Platform Folder]** opens it directly in the OS file explorer. Because this is the folder that build and autocomplete reference, you can edit its contents or swap versions directly in the explorer if you need to.

Language switches the whole app UI to that language the moment you select a radio button (there is no separate save button). The language it starts in on the very first launch is decided by **the edition you installed** — the installer comes in three kinds, Korean, English, and Japanese, and since there is only one app, it first comes up in the language of the edition you installed. Once you have chosen a language here, that choice takes priority, so it is kept even if you later update with a different edition.

The **Send error reports automatically** checkbox is on by default, and while it is on, diagnostic information is sent automatically when the app terminates abnormally or an error occurs. Report a Bug in the Help menu is always sent regardless of this setting.



< Figure 3-2 The Preferences modal — the General tab: User Library Folder, Platform Folder, Language, and Send error reports automatically >

Shortcuts

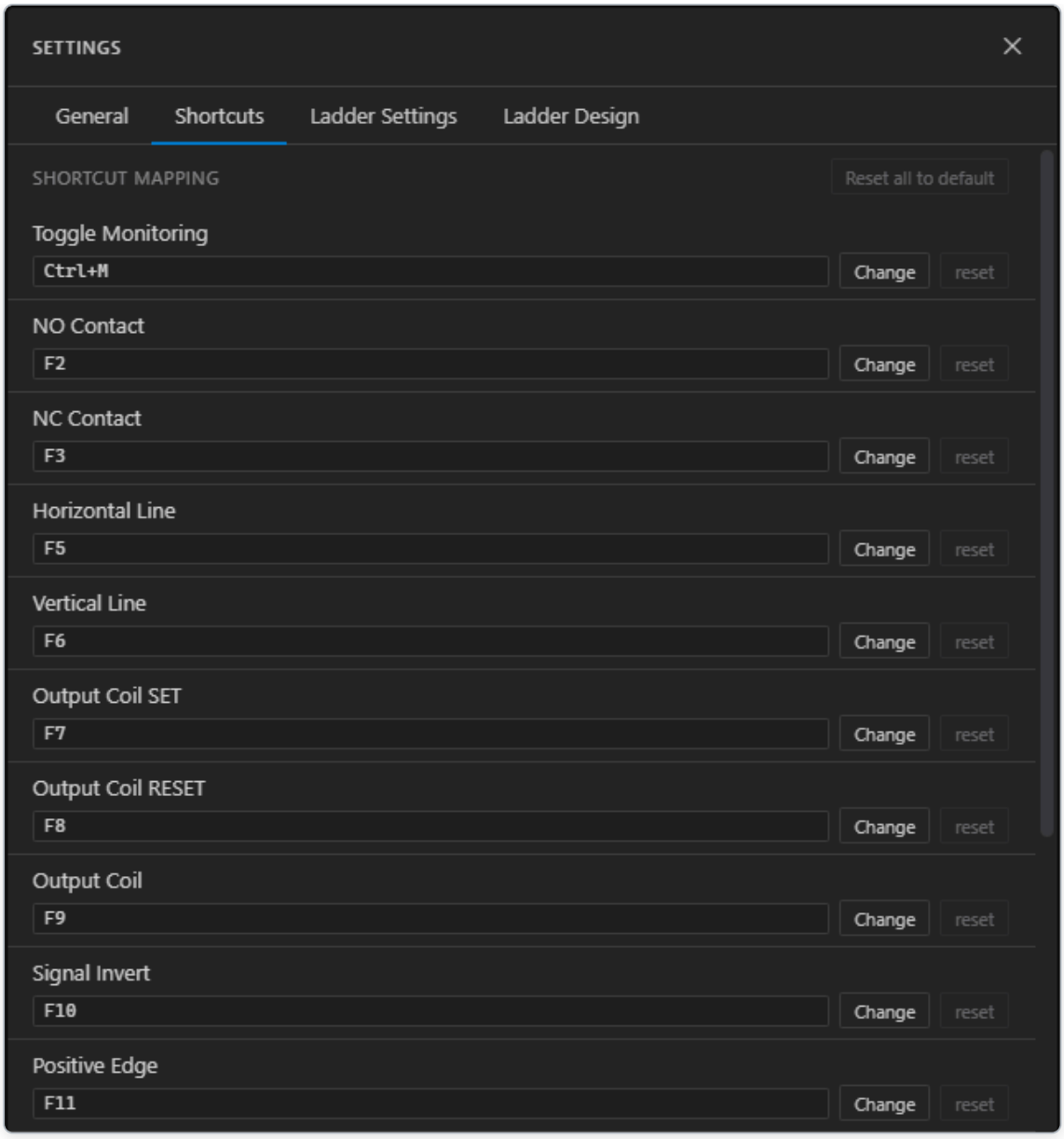
Only 16 shortcuts can be redefined on this tab — the ladder tools, cell/row editing, and the monitoring toggle (for the full list of every other shortcut, see **Keyboard Shortcuts**(2-3)).

Command	Default shortcut
Toggle Monitoring	Ctrl+M
NO Contact	F2
NC Contact	F3
Horizontal Line	F5
Vertical Line	F6

Command	Default shortcut
Output Coil SET	F7
Output Coil RESET	F8
Output Coil	F9
Signal Invert	F10
Positive Edge	F11
Negative Edge	F12
Box Function	Ctrl+Enter
Insert Cell	Insert
Delete Cell	Ctrl+Delete
Insert Row	Alt+Insert
Delete Row	Alt+Delete

Every row has a **[Change]** and a **[reset]** button, and the header above the table has a **[Reset all to default]** button that reverts everything. Pressing **[Change]** brings up the "Change Shortcut" dialog, and, as its prompt says ("Press the new shortcut. Enter to confirm, Esc to cancel."), pressing the key (combination) you want shows that combination on the spot. **Enter** confirms and **Esc** cancels, and pressing another key before you confirm overwrites the combination you just captured with the new one (for recovering from a mistype). If the combination you captured is already used by another command, a warning that includes the name of the existing command appears — such as "This key is already assigned to 'Save' — applying it will still conflict" — but it is still applied if you press **[Confirm]**. **[reset]** reverts only that one row to its default (disabled if it has never been redefined), and **[Reset all to default]** is disabled when not a single item has been redefined. Everything on this tab takes effect **immediately**, and there is no separate save button.

> **Conflict checking is done over the whole range of shortcuts, not just the 16 in this table.** If you change NO Contact to Ctrl+S, for example, it conflicts with the Save command, which is not in the table. When there is a conflict, a "Conflicting shortcuts" warning list appears at the top of the tab, the one registered first fires with priority, and the rest do nothing when pressed. Even so, saving (applying) itself is not blocked — sometimes you deliberately want to disable a particular function.



< Figure 3-3 The Preferences modal — the Shortcuts tab: the 16 redefinable commands with [Change]/[reset]/[Reset all to default] >

Ladder Settings

At the very top of the tab is the notice "How symbols (address aliases) are shown in ladder cells — a global user setting (this PC). Separate from the per-project ladder design." Of the items on this tab, only **Monitoring port** is stored per project (`.mp2`); all the rest are **global user settings** for this PC.

Item	Stored in	Description
Ladder symbol display mode	Global	Two ways of showing the address and symbol in a ladder cell
Use Ladder	Global	When unchecked, builds from the Arduino code alone
Duplicate coil check	Global	A build error when two or more output coils use the same memory

Item	Stored in	Description
Auto input box on contact creation	Global	Opens the input box automatically when you use a contact creation tool
Monitoring port	**Per project** (`.mp2`)	The serial port that ladder monitoring data is sent over

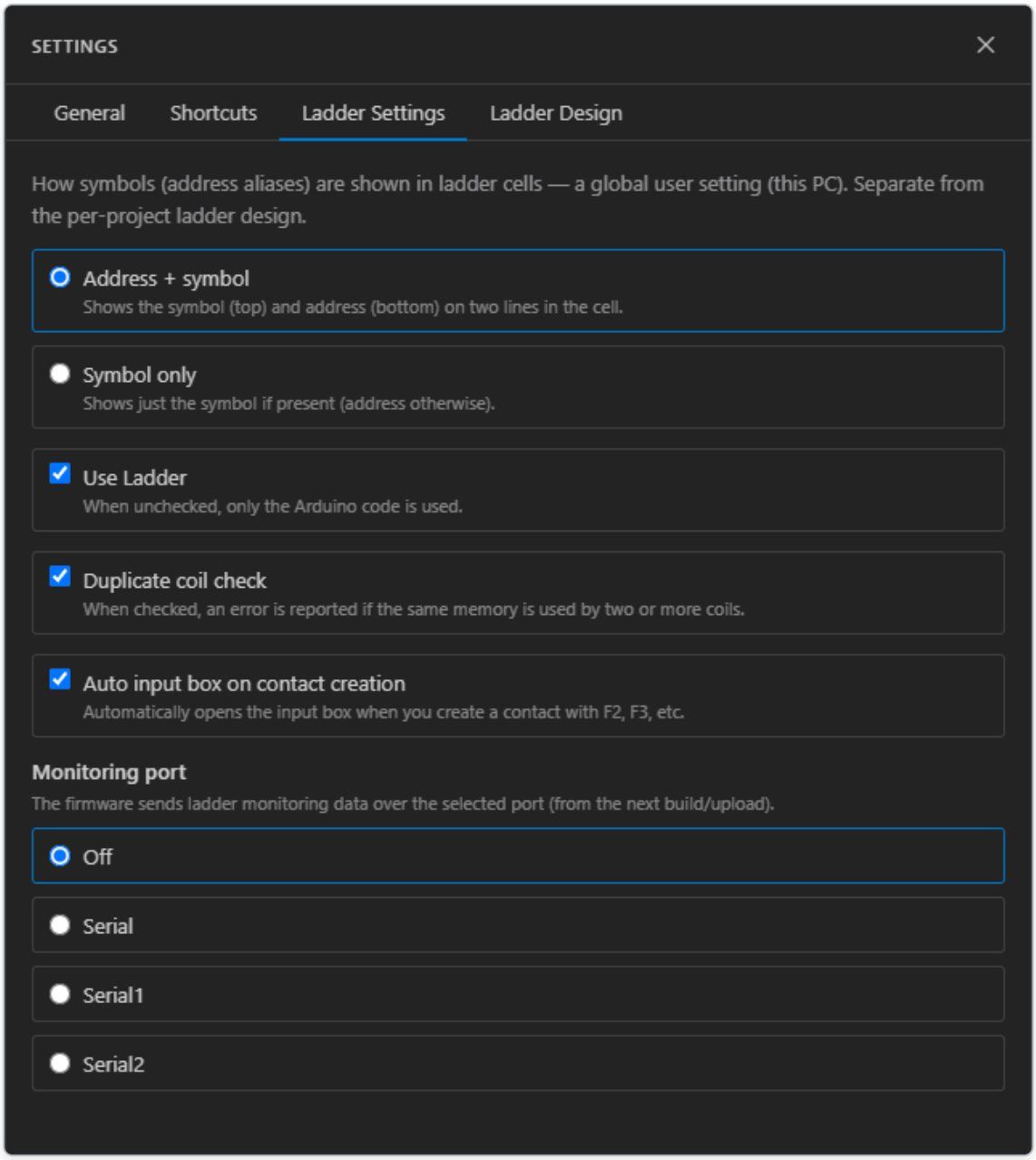
The Ladder symbol display mode has two radio options. **Address + symbol** shows the symbol (top) and the address (bottom) together on two lines in the cell, and **Symbol only** shows just one line with the symbol if there is one (the address otherwise). It is a display method used only on this PC, separate from the "Ladder Design" tab (cell colors, sizes, and so on) covered next.

Unchecking the **Use Ladder** checkbox excludes the ladder-generated code at build time and builds from pure Arduino code alone (the ladder tabs themselves remain and can still be edited).

With **Duplicate coil check** checked, the build is blocked with an error when the same output address is used as an output coil in two or more circuits.

With **Auto input box on contact creation** checked, the cell edit input box opens automatically every time you use a contact creation tool such as F2 or F3.

Monitoring port is chosen with radio buttons. The full set of options is **Off** / **Serial** / **Serial1** / **Serial2**, but only **the UARTs the current board actually has** are left on screen (on the MPINO-8A4R(T)-S, for example, which has just two UARTs, only "Off / Serial / Serial1" are shown). This is so that nothing can be chosen here and then blocked at build time; if you took a project saved with a board that has more UARTs and changed it to one that has fewer, that value stays visible in the list, and you simply correct it to another port. Ladder monitoring data is sent over the port you chose from the next build/upload onward — choosing the port doubles as choosing whether monitoring is used at all (Off means it is turned off). If your Arduino code already uses that serial port directly (for example, choosing **Serial** in code that uses `Serial.begin(...)`), a "Serial already in use" warning appears ("Your Arduino code already uses Serial. Using it for both may cause ladder monitoring to malfunction."), but the selection is applied as it is once you acknowledge it.



< Figure 3-4 The Preferences modal — the Ladder Settings tab: the symbol display mode, Use Ladder, Duplicate coil check, Auto input box on contact creation, and Monitoring port >

Ladder Design

The top of the tab shows the notice "Applies only to the current theme (Dark). Leave blank to use the default." (this app currently offers only a dark theme). Unlike the other three tabs, changing a value here is not reflected right away — you have to press **[Save]** for it to apply — and if you switch to another tab or try to close the modal without saving, a dialog appears asking whether to discard the changes.

There are six colors, of which only four actually apply; the remaining two are marked "(planned)" on screen (you can enter them and the values are saved, but they are not applied anywhere yet).

Color	Status
Cell color	Applied
Selected cell color	Applied

Color	Status
Line color	Applied
Text color	Applied
Symbol color	(planned)
Monitoring color	(planned)

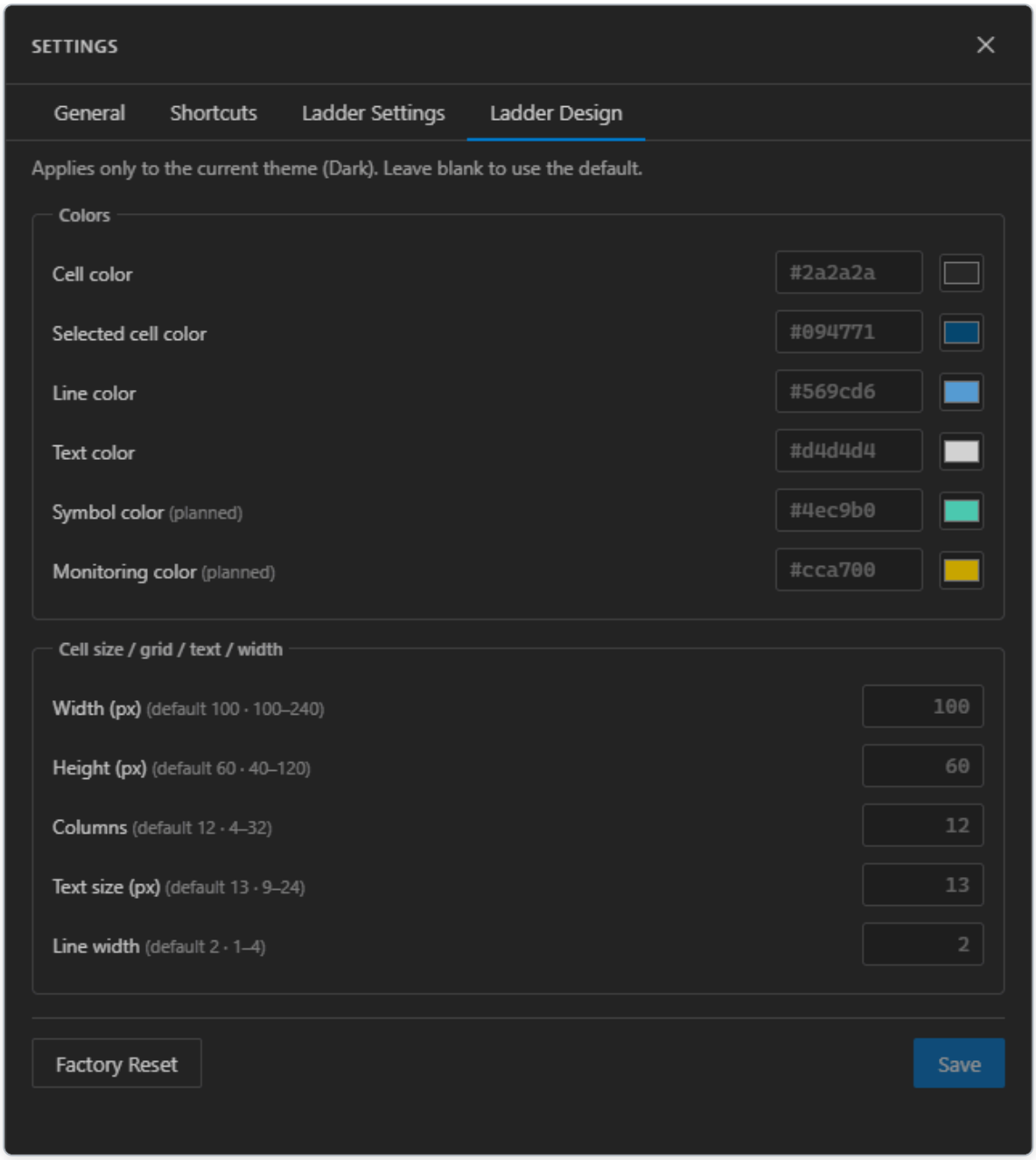
Each color has a hex text input in `#RRGGBB` form and a color picker (swatch) side by side, and the two stay in sync with each other. Leave one blank and the default is used; enter a value that is not a 6-digit `#RRGGBB` value, and a warning appears and **[Save]** is disabled.

There are five numeric items, and the range guide on each input also shows the actual default.

Item	Default	Range
Width (px)	100	100–240
Height (px)	60	40–120
Columns	12	4–32
Text size (px)	13	9–24
Line width	2	1–4 (in steps of 0.5)

Leave one blank and the default is used; enter a value outside the range, or one that is not an integer (not a multiple of 0.5 for line width), and a warning appears and **[Save]** is disabled.

[Factory Reset] at the bottom returns every color and numeric input to blank (the screen reflects it the moment you press it, but you have to press **[Save]** for it to actually apply). Pressing **[Save]** records the changes and closes the Preferences modal.



< Figure 3-5 The Preferences modal — the Ladder Design tab: six colors (four applied, two planned), five numeric items, Factory Reset, and Save >

3-2 Edit

The Edit menu consists of five items — Undo, Cut, Copy, Paste, and Delete — and every one of them is a command that edits a ladder circuit.

Item	Shortcut	Description
Undo	Ctrl+Z	Reverts the most recent ladder edit by one step
Cut	Ctrl+X	Moves the selected ladder cells to the clipboard and clears the originals
Copy	Ctrl+C	Copies the selected ladder cells to the clipboard
Paste	Ctrl+V	Pastes the cells on the clipboard at the current selection

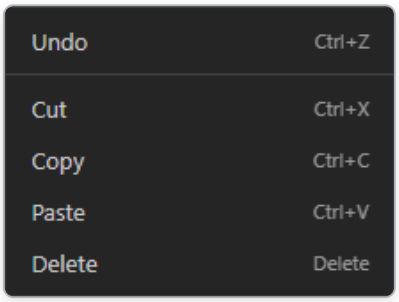
Item	Shortcut	Description
Delete	Delete	Clears the selected ladder cells

Undo (Ctrl+Z) cancels the single most recent undoable ladder edit, such as applying a cell tool, cutting, pasting, or deleting. If there is no edit to undo, nothing happens, and redo is not supported.

Cut (Ctrl+X) puts the selected cells on the clipboard and clears their place. **Copy (Ctrl+C)** leaves the originals as they are and only puts them on the clipboard. **Paste (Ctrl+V)** pastes the cells held on the clipboard using the currently selected cell as the top-left anchor (or from the first column of the first row of the grid if no cell is selected), and does nothing if the clipboard is empty.

Copied and cut cells are also placed on the system clipboard. That means you can keep two copies of MPINO Studio 2 open and paste a circuit copied in one window straight into the ladder of the other.

Delete (Delete) clears the selected cells. When a single cell is selected, a cell that also has a vertical connecting line (F6) loses only the connecting line on the first delete, and you have to delete once more to clear the whole cell. When you have selected a range of several cells, one delete erases the connecting lines and the contents together.



< Figure 3-6 The Edit menu — the dropdown showing all five items from Undo to Delete >

Edit menu items behave differently depending on where you run them.

- **Cut, Copy, Paste, and Delete work only when a ladder tab is active.** Clicking any of these four while a code tab is active does nothing at all.
- **Undo always works regardless of the type of the active tab** — even if you click it while you are looking at a code tab, the most recent ladder edit, if there is one, is undone (the target is always the ladder only, and the text content of a code tab is not subject to Undo).
- Pressing Ctrl+Z, Ctrl+X, Ctrl+C, or Ctrl+V directly inside the code editor runs Monaco's ordinary text undo, cut, copy, and paste, independently of these menu commands.

For the other editing shortcuts, such as insert/delete cell and add/delete row, see **Keyboard Shortcuts**(2-3).

3-3 View

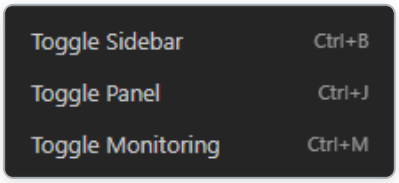
The View menu consists of three items — Toggle Sidebar, Toggle Panel, and Toggle Monitoring — and each item opens and closes one area of the screen.

Item	Shortcut	Description
Toggle Sidebar	Ctrl+B	Opens and closes the sidebar on the left
Toggle Panel	Ctrl+J	Opens and closes the bottom panel below the editor
Toggle Monitoring	Ctrl+M	Turns live ladder monitoring communication on and off

Toggle Sidebar (Ctrl+B) opens and closes the sidebar on the left of the screen. The sidebar shows the panel selected in the Activity Bar, and which panel was selected is kept even after you close the sidebar and open it again (the Activity Bar in detail is outside the scope of this chapter).

Toggle Panel (Ctrl+J) opens and closes the bottom panel below the editor (the Build Output, Problems, Mnemonic, and Serial Monitor tabs). The panel height and the tab you last selected are kept even after you close it and open it again.

Toggle Monitoring (Ctrl+M) turns live ladder monitoring communication on and off. If no monitoring port is designated in Preferences, a notice window telling you to set the port appears instead of the communication turning on. How the monitoring screen is laid out and how the data is displayed are outside the scope of this chapter.



< Figure 3-7 The View menu — the dropdown showing all three items from Toggle Sidebar to Toggle Monitoring >

3-4 Run

The Run menu consists of five items — build, upload, and cancel upload, plus the ones that open the Communication and Memory Write List windows.

Item	Shortcut	Description
Build	Ctrl+R	Transpiles and compiles the current project to verify that there are no errors
Upload	Ctrl+U	Builds the current project, then flashes it to the board over the selected port
Cancel Upload	—	Aborts an upload in progress
Communication	—	Opens the communication channel (protocol) settings window
Memory Write List	—	Opens the Memory Write List window

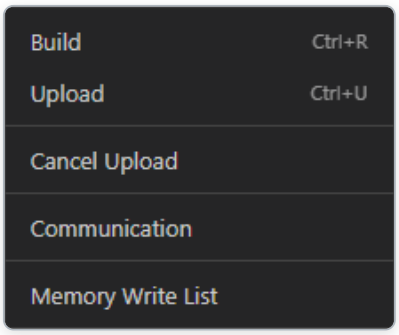
Build (Ctrl+R) transpiles and compiles the current project to verify that there are no syntax or build errors (nothing at all is written to the board). The result is shown in the Build Output tab of the bottom panel as a state (Idle / Building / Success / Failed) and a log.

Upload (Ctrl+U), like Build, transpiles and compiles the current project, and then goes straight on to flash it to the board over the selected port, all in one run (there is no need to build separately first). If no port is

selected, an error notice telling you to select a port first appears (select it by clicking the ▾ on the right of the status bar) and the upload does not run. Upload progress is shown in the bottom panel in real time.

If something else is also using the port ladder monitoring is set to use, a confirmation window carrying the port name — such as **"Serial1 already in use"** — appears before the upload starts. The body and the suggested fix differ depending on whether the other party is a channel in the communication settings, a communication init call in your Arduino code, or your code using that port directly. It is a warning rather than a block, so pressing [Upload] goes ahead as it is, and pressing [Cancel] means the upload does not run. For the three cases in detail, see **Selecting a Port and Uploading**(1-8).

Cancel Upload only means anything while an upload is actually in progress (clicking it in any other state does nothing at all), and it aborts the upload in progress the moment you click it. A cancelled upload is shown not as a failure but as a separate "Cancelled" state.



< Figure 3-8 The Run menu — the dropdown showing all five items from Build to Memory Write List >

> **Both build and upload run on the current contents of the editor, regardless of whether you have saved.** Unsaved changes are reflected as they are, and no confirmation dialog demanding that you save appears.

3-4-1 Communication

Clicking **Communication** (no shortcut) brings up the "Communication Settings" window. Every time the window opens it reloads the communication channel information of the current board; while it is loading it shows "Loading channel information...", and if it fails, "Failed to load board information." If the board you have selected has no communication channels, the only thing shown is the notice "This board has no communication channels."

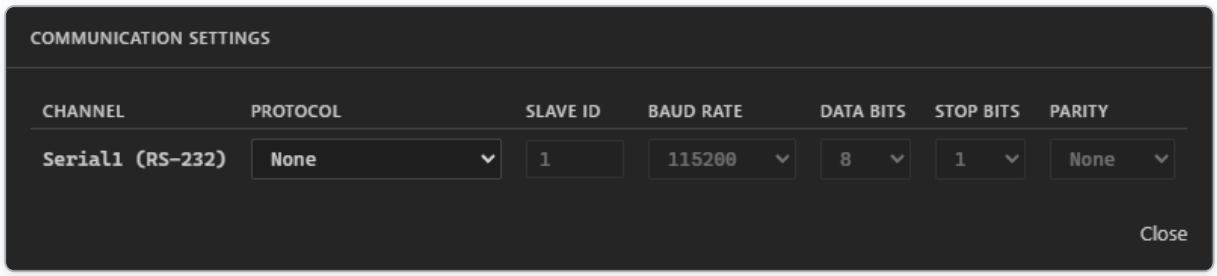
A board that does have communication channels gets a table with one row per channel, and each row consists of the following seven columns.

Column	Description
Channel	The channel name (differs per board, for example Serial1 (RS-232))
Protocol	None / Modbus RTU Slave / LS Electric Cnet Slave / Modbus RTU Master (Coming soon) / Modbus TCP/IP (Coming soon)
Slave ID	1–247
Baud Rate	Chosen from 300–500000 bps (115200 by default)

Column	Description
Data Bits	5/6/7/8 (8 by default)
Stop Bits	1/2 (1 by default)
Parity	None/Even/Odd (None by default)

Leaving **Protocol** as "None" means that channel is not used, and the other five columns are shown disabled. Choosing **Modbus RTU Slave** or **LS Electric Cnet Slave** enables the other five columns. **Modbus RTU Master** and **Modbus TCP/IP** are marked "(Coming soon)" and cannot be selected yet. Switching between the two working protocols (Modbus RTU Slave ↔ LS Electric Cnet Slave) keeps the five values you have already entered, while switching afresh from None (or from one of the coming-soon options) to a working protocol resets them to the defaults of Slave ID 1, baud rate 115200, data bits 8, stop bits 1, and parity None.

Changes are reflected in the project immediately (there is no separate save button) and take effect right from the next build or upload — you do not have to save the project file separately. Close the window with the **Close** button.



< Figure 3-9 The Communication Settings window — Protocol, Slave ID, Baud Rate, Data Bits, Stop Bits, and Parity for each channel >

3-4-2 Memory Write List

Clicking **Memory Write List** (no shortcut) brings up a window for writing values to the M/D/C/T/R memory addresses of the running board and watching their current values in real time. It is a feature separate from the individual write-value popup you open by right-clicking a ladder cell (anything to do with the ladder editing screen is outside the scope of this chapter). Unlike other settings modals it is a **modeless** window that does not cover the background, so you can keep editing the ladder while it is open, and you can drag the header to move it. Close it with the ✕ in the header or with Esc.

To write a value, monitoring communication has to be turned on first with the **Monitoring** button in the header (if no monitoring port is set, pressing the button only brings up a window pointing you to the settings — the same behavior as **View › Toggle Monitoring**). While monitoring is off, every write-related button in the window is disabled.

Enter a value in the input row at the top of the window (address + value to write) and press **Write** or hit Enter, and the value is written to the board on the spot; on success that address is added to the list below as a new row (if the address is already in the list, only that row's value is updated). If the write fails, the list is unchanged and an error toast appears.

There are four buttons on the toolbar above the list.

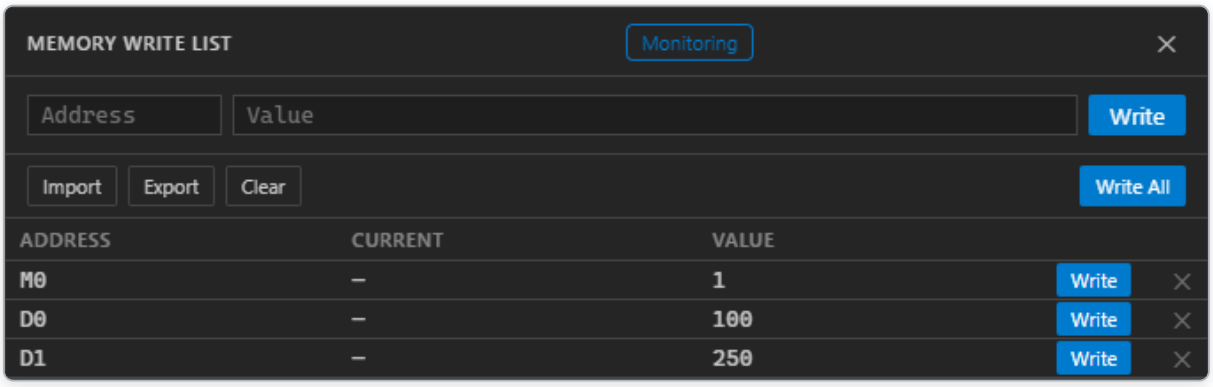
Button	Action
Import	Loads a list from a file and replaces the current list wholesale (the existing list is kept if you cancel the selection)
Export	Saves the current list to a file
Clear	Deletes the whole list (it goes through a confirmation dialog, and is disabled when the list is empty)
Write All	Writes every row in the list to the board in order

Write All is disabled if monitoring is off, if an operation is in progress, or if even one row in the list has a syntax or range error. When it runs, rows with no value to write are skipped and the rest are written one after another, but if even one row fails to write along the way, it stops immediately at that point and shows the error (the rows after it are not attempted).

The list table is shown in exactly the order you registered the rows (no sorting), and after the three columns Address, Current, and Value comes a column of write and delete buttons that has no column heading.

- Clicking an **Address** or **Value** cell turns it into an input right there on the spot (Enter commits, Esc cancels, and clicking elsewhere also commits). If you try to change an address to something that is not `M` / `D` / `C` / `T` / `R` followed by digits, or to an address already in the list, the change is rejected and the reason is shown below the table.
- **Current** shows the live value received through monitoring, and when there is no value it shows "—".
- An address whose format is correct but which falls outside the memory range of the current board is marked as an error on that row, and a notice telling you the range appears below the table as well. In this case the edit itself is not rejected, but the **Write** button on that row is disabled.
- The **Write** button on each row writes just that one row to the board. The ✕ button on the same row (tooltip: Delete) writes nothing at all to the board and only removes that row from the list.

If the list is empty, the only thing shown is the notice "No memory registered — add an address (M/D/C/T/R) in the input row above."



< Figure 3-10 The Memory Write List — the input row at the top, the Import/Export/Clear/Write All toolbar, and the table for watching the current value of each address >

3-5 Settings

The Settings menu consists of six items. Five of them are entry points that take you straight to another screen — the essential reading in Chapter 2, the Preferences section earlier in this chapter, or, as with **Change Board**

below, somewhere else — and only EEPROM Retain is introduced for the first time in this section.

Item	Shortcut	Description
Memory Area Settings	Ctrl+Shift+M	Opens the modal for adjusting the per-project memory area sizes
Ladder Design Settings	Ctrl+Shift+D	Opens the modal for adjusting the design, such as ladder cell colors and sizes
Ladder Pin Map Settings	Ctrl+Shift+P	Opens the modal for editing the pin definitions of the current board
Change Board	—	Replaces only the target board of the current project with another board, without opening a new one
Used Memory	Ctrl+Shift+L	Opens the modal for checking which addresses the project actually uses
EEPROM Retain	Ctrl+Shift+E	Opens the modal for setting which memory areas are retained after a power loss and how they are saved

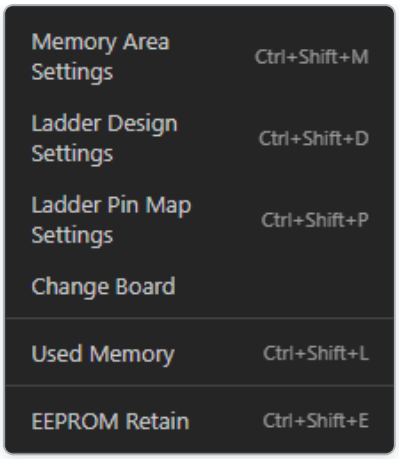
Memory Area Settings (Ctrl+Shift+M) opens a modal for adjusting the counts of the six memory areas per project, beyond the board defaults. For details, see "Area Sizes and Memory Settings" in **Data Memory**(2-1).

Ladder Design Settings (Ctrl+Shift+D) opens the Preferences modal directly on the "Ladder Design" tab — the same screen covered in **Preferences**(3-1-1) earlier in this chapter.

Ladder Pin Map Settings (Ctrl+Shift+P) opens a modal for editing, in a table, the pin definitions of the board the current project uses. For details, see "Board Pin Map Editing" in **Pin Editing**(2-4).

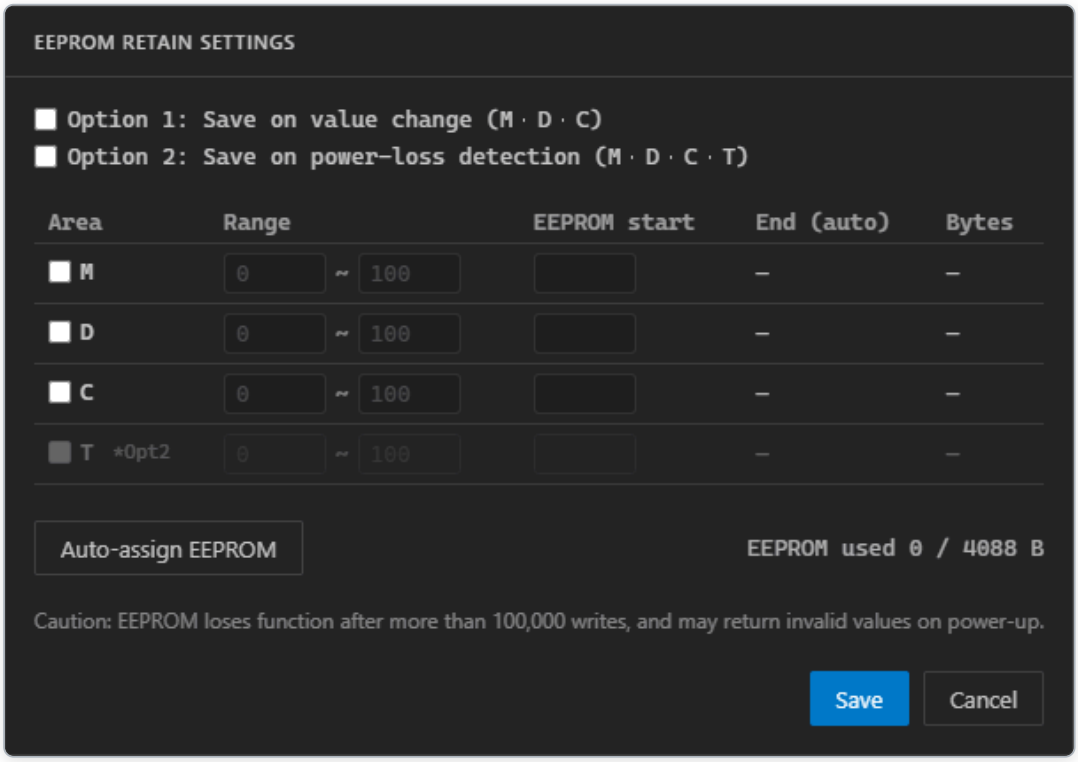
Change Board (no shortcut) opens the board selection modal that changes only the board the current project targets to another board, without creating a new project. It opens with the currently selected board preselected as the initial value, and if you pick another board and confirm, only the target board information is replaced while the code and the ladder are left as they are (the board does not change if you cancel). This modal is the same screen as the one that opens from New, so the two category tabs **MPINO** / **MPAINO**, the **Option modules** counts for MPAINO boards, and the **Serial RX Buffer** section are all available here as well — for details, see **Creating a Project and Selecting a Board**(1-4). The same modal can also be opened with the **[Change board]** button next to the "Board" row in the Explorer panel — for details, see **The Explorer Panel**(9-2).

Used Memory (Ctrl+Shift+L) opens a modal for checking, on a per-area grid, the addresses the project actually uses. For details, see "Checking Used Memory" in **Data Memory**(2-1).



< Figure 3-11 The Settings menu — the dropdown showing all six items from Memory Area Settings to EEPROM Retain >

EEPROM Retain (Ctrl+Shift+E) opens the "EEPROM Retain Settings" window, where you designate the M, D, C, and T memory areas to retain even when the power goes off, and when they are saved. The save timing is decided with two independent checkboxes, Option 1 (save every time a value changes, M·D·C only) and Option 2 (save on power-loss detection, all of M·D·C·T), and you can turn both on at once. To use Option 2 you have to designate a power detect pin (if the detect time (ms) is left blank it is treated as 0). The EEPROM addresses you designate for each area can be laid out without overlap in one go with **[Auto-assign EEPROM]**, and saving is blocked if the total usage exceeds 4088 bytes (8 of the 4096 total bytes are reserved for internal use) or if they overlap each other.



< Figure 3-12 EEPROM Retain Settings — Option 1 and Option 2, the range and EEPROM address for each area, auto-assign, and usage >

EEPROM loses its function after more than 100,000 writes, and invalid values may be read on power-up. Option 1 (save on value change) in particular saves every time a value changes, so if you

include frequently changing addresses in a retained area, the write count piles up quickly.

3-6 Help

The Help menu consists of four items — Report a Bug, Request an Improvement, Check for Updates, and About — and none of them has a shortcut.

Item	Shortcut	Description
Report a Bug	—	Describes a problem and reports it
Request an Improvement	—	Requests a feature or improvement you want
Check for Updates	—	Checks the server for a new version
About	—	Shows the app name and version as a toast

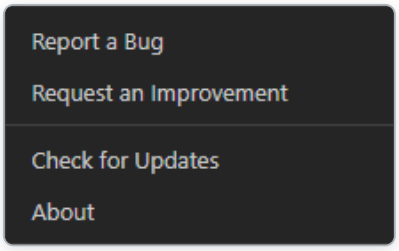
Clicking **Report a Bug** brings up the "Report a Bug" window, where you describe the problem and send it with **[Send]**. Basic diagnostics (version, OS, logs) are always included, and the current project (`.mp2`), a screenshot of the screen, and a contact are optional attachments. The basic diagnostics checkbox is shown **greyed out and locked so that it cannot be turned off, always in the on state**, and only the other three items (project, screenshot, and contact) can be turned on and off.

Clicking **Request an Improvement** brings up the "Send an Improvement Request" window, where you enter the feature or improvement you want and send it with **[Send]**, optionally attaching a screenshot of the screen and a contact. Unlike the Report a Bug window, the improvement request window **has no diagnostics item** — only a screenshot of the screen and a contact are optional attachments.

Clicking **Check for Updates** sends a check request to the update server right away. If you are already on the latest version, a "You are using the latest version." info toast appears; if the server connection fails, an "Unable to connect to the update server." error toast appears; and if there is a new version, an update window appears with the version information and the **[Update Now]** / **[Later]** buttons.

Quite apart from this menu item, the app **checks for an update once automatically every time it launches**. The automatic check raises the update window only when there is a new version; if you are already on the latest version, or if the server cannot be reached, it passes over silently with no notice at all — the side that always shows a toast is checking from this menu yourself.

Clicking **About** brings up a "MPINO Studio 2 · v2.0.4" info toast.



< Figure 3-13 The Help menu — the dropdown showing all four items from Report a Bug to About >

4 Ladder Grid Editing

There are two broad ways to actually edit the ladder grid. One is the context menu you open by right-clicking a cell, an entry point that gathers commands such as turning monitoring on, switching the number format, and writing a value to memory together with the cell and row editing commands. The other is the operations that change the arrangement of the grid's squares itself, such as inserting and deleting cells and adding and deleting rows. This chapter explains these two axes in turn, but first it covers the **cursor movement and selection** that every editing command rests on in common. Reading it before you place **Contacts** (Chapter 5) or **Box Functions** (Chapter 6) lets you learn the basic operations for handling cells in advance.

4-1 The Grid Status Bar

The status bar just below the ladder grid canvas shows the coordinates and the type of the cell you have selected right now, in real time. It is separate from the Status Bar at the very bottom of the screen used for selecting a port; this one appears only inside the ladder editor. Before you run a context menu command or a shortcut, you can always check on this line which cell is selected.

The coordinates are shown in the form `R{row} C{col}`, where row numbers count from 1 and column numbers from 0. Select the very first cell at the top left of the grid and the status bar shows `R1 C0` — note that, unlike the Chapter 1 tutorial, which calls the same spot "row 1, column 1" (with both the row and the column counted from 1), the column number in the status bar starts at 0.

If the selected cell has an element in it, the cell type is appended after the coordinates. The type is shown exactly as the name the app uses internally: contacts appear as `ContactA` (A contact) and `ContactB` (B contact), coils as `OutCoil`, `SetCoil`, and `ResetCoil`, and box functions as `Func`. Beyond those, edge-detection contacts (`RisingEdge` and `FallingEdge`), Signal Invert (`Reverse`), and the horizontal connecting line (`LineH`) are each shown under their own name as well. The vertical connecting line you place with F6 (Vertical) is not a separate cell type but a property attached to the existing cell (`link_to_below`), so applying F6 leaves the type shown in the status bar as that cell's original type (`None` for an empty cell, `ContactA` for a contact, and so on) — a type called `LineV` never appears. A cell that has a memory address or a function code entered in it gets that code appended once more after the type, so it looks like `R1 C0 · ContactA · P0`.

Select an empty cell — that is, one where no cell exists in the array at those coordinates at all — and "Empty cell" is appended in place of the type, giving `R{row} C{col} · Empty cell`. When the cell does remain in the array but no tool has been applied to it, however, as with a cell you cleared with Delete, it is shown with the internal type name as it is: `R{row} C{col} · None` — both cases look exactly the same on the grid, as an empty square, but this is how the status bar tells them apart. When no cell is selected at all, the operation hint "Click a cell or move with arrow keys, Enter to edit, F-keys for tools, Delete to clear." is shown instead of a type.

4-2 Cursor Movement and Selection

Almost every command in the ladder grid works on "the cell the cursor is on right now". That makes how you move the cursor, and how far you pick out at once, the starting point of all editing. While the grid has focus,

the arrow keys (↑, ↓, ←, →) move the cursor one square at a time, **Enter** opens the edit window for that cell, the **F-keys** apply a tool, and **Delete** clears the cell — the same operations as the hint the status bar shows when nothing is selected.

4-2-1 The Cursor on a Box Function

A box function (Chapter 6) takes up **several squares** on screen. Below the command strip one argument takes one square each, and the body widens over the empty squares on its right or over the horizontal lines on its left. In the ladder data, though, a box function is still **a single cell**. So whichever of the squares the box occupies you pick, the F-keys, Delete, Enter (edit), the right-click menu, and the toolbar tools all act on the **box as a whole**. It never happens that only a horizontal line hidden under the body is erased on its own.

The ← and → arrow keys are the exception: **inside a box they move one argument square at a time**.

- Come into the box from outside with → and the cursor lands on the **first argument** square; come in with ← and it lands on the **last argument** square (it carries on in the direction you were pressing).
- Press ← or → inside the box and the argument square moves one at a time; press it **once more at the last argument** and only then does the cursor leave for the next square outside the box.
- ↑ and ↓ move one square up or down while keeping the **column the cursor stands in on screen**. If the square they arrive at is covered by another box, the cursor goes into the argument square that the same column of that box points at.
- At the edge of the grid the cursor stays where it is, and the argument square cursor is kept as well.

The argument square the cursor is on is marked with a **border and a faint fill** in the same color as the cell cursor (the shading of a write target square has the fill only, so the two can be told apart). Press Enter or double-click in that state and the edit window opens with **only that argument's text selected**, rather than the whole box, so you can rewrite a single argument straight away.

> **Some boxes do not get an argument square cursor.** A **free-form** (6-9) box such as `D0 = D0 + 1` is not divided into squares but drawn as a single line of its original text, and a box that has been **compressed** into fewer columns than it has arguments, because there was not enough room to widen the body, does not line its columns up with its arguments one to one. In these two cases ← and → **skip over the box as a single square** as they used to, and opening it with Enter selects the whole line.

4-2-2 Range Selection

Picking several cells at once is called a **range selection**. There are four ways to make one.

Way	Operation	Result
Shift+arrow keys	Pick one cell, then hold Shift and press an arrow key	A rectangle widens one square at a time from the first cell as its anchor (only while the ladder grid has focus)
Shift+click	Pick one cell, then hold Shift and click another cell	The rectangle that has the anchor and the clicked cell as its two ends
Ctrl+click	Hold Ctrl and click a cell	Puts just that one cell into the selection, or takes it out (a toggle) — for gathering up cells that lie apart
Drag	Hold the mouse button down on a cell and drag	A rectangular outline follows while you drag, and the cells inside it are selected

The unit of a range selection is the **cell**, not the argument square of a box function. So Shift+arrow keys skip over a box as a single square, and when the rectangle catches a box, the **box function cell and the covered horizontal lines on its left go into the selection as one lump**. They come out as one lump when you take them out with Ctrl+click as well — this is to stop a horizontal line hidden under the body from being left behind in the selection and quietly breaking the power path.

Once you have picked a range, these are the commands you can use.

- **Delete** — clears every selected cell at once (a single Ctrl+Z brings it all back).
- **Ctrl+C** / **Ctrl+X** / **Ctrl+V** — see **Clipboard and Undo** below.
- **The F-key and Ctrl+Enter tools** — these are not applied to several cells at once. The selection is first narrowed to **the single cell the cursor was last on**, and the tool is applied to that cell only.

4-3 The Ladder Cell Context Menu

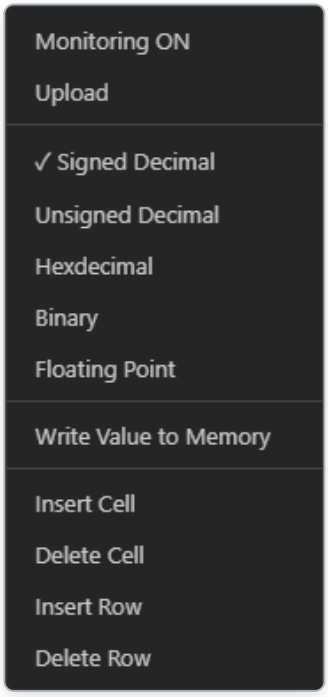
Right-click a ladder cell and that cell is selected first (the selection marker moves to it), and then the context menu opens. Every insert, delete, and number-format command in the menu works on this newly selected cell, so which cell you right-clicked decides the result of the command that follows.

The menu always shows the same 12 items regardless of the cell type — contact, coil, box function, empty cell, and so on. In the actual menu, separators divide them into four groups — ① monitoring and upload, ② the five number formats, ③ write value to memory, and ④ the four grid editing commands — and the table below lists them in that same group order. No item is ever added or removed depending on the cell type; the only parts that change dynamically are the ON/OFF label of item 1 and the position of the radio check among items 3–7.

Item	Description
Monitoring ON / Monitoring OFF	Turns live ladder monitoring communication on and off. The label changes according to the current state (when it is off, "Monitoring ON"; when it is on, "Monitoring OFF") — for details, see **Getting Started with Ladder Monitoring**(8-5)
Upload	Builds the current project, then flashes it to the board over the selected port — for details, see **Run**(3-4)
Signed Decimal	Shows monitoring values as signed decimal (the default)
Unsigned Decimal	Shows monitoring values as unsigned decimal
Hexdecimal	Shows monitoring values in hexadecimal (this is the app's actual spelling, not a typo)
Binary	Shows monitoring values in binary
Floating Point	Shows monitoring values as floating point
Write Value to Memory	Opens a popup for writing (forcing) a value directly to the memory address of the selected cell. It overwrites the value at that address on the running board with the value you enter, right away — for details, see **Write Value to Memory** below
Insert Cell	Inserts one cell into the grid — for details, see **Inserting and Deleting Cells** below
Delete Cell	Deletes one cell from the grid — for details, see **Inserting and Deleting Cells** below

Item	Description
Insert Row	Adds one row to the grid — for details, see **Adding and Deleting Rows** below
Delete Row	Deletes one row from the grid — for details, see **Adding and Deleting Rows** below

The number formats in items 3–7 are radio items, and a check mark appears on the format currently selected. For details such as how each format actually represents a value and how the time-unit display is combined with it, see **Number Formats and Special Displays**(8-7).



< Figure 4-1 The ladder cell context menu — the same 12 items always appear >

4-4 Write Value to Memory

Selecting "Write Value to Memory" in the context menu (or pressing the shortcut) opens an on-the-spot force write popup. It forces the value at the memory address of the cell the cursor (selection) is on, overwriting it on the running board with the value you enter, right away.

How to open it — right-click a ladder cell and select "Write Value to Memory", or press the shortcut **Ctrl+Shift+W** (the shortcut works only while the ladder grid has focus). Both routes give the same result, and the popup always opens for the cell the cursor (selection) is on.

The address field at the top of the popup (placeholder `"M0"`) is filled in automatically with a writable address. Only the M/D/C/T/R areas can be written (P contacts are excluded), and the address is extracted automatically from the contents of the selected cell — when there are several candidate addresses, as with a box function, you can pick one from a dropdown; when there is a single address, as with a contact or a coil, that address is filled in; and for an empty cell the field is left blank for you to type into yourself. Enter an invalid address and the notice "... is not writable (M/D/C/T/R only)" appears inline.

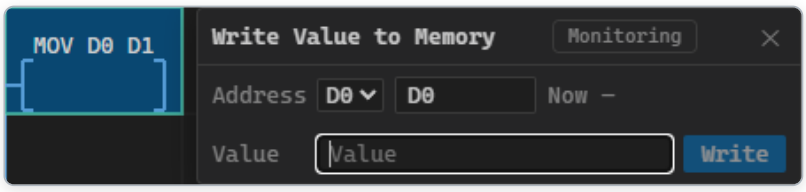
Enter a value in the value field (placeholder `"Value"`) and press the **Write** button or hit Enter, and the value is written to the board on the spot. The format of the value is interpreted on the board side according to the data type of the area. Writing is enabled only while monitoring communication is on; if it is off, you can turn it on with the **Monitoring** button in the popup header (if there is no monitoring port, a window pointing you to the settings appears). On a successful write the popup stays open and only the value field is cleared, so you can force the same address or another address one after another, and the current value keeps being shown in the popup in real time.

How to close it — close the popup with **Esc**, by clicking outside it, or with the ✕ button in the header. Unlike other settings windows it is a modeless window that does not cover the background, so you can see other parts of the ladder while it is open, and you can drag the header to move it.

It is easy to confuse this with the similarly named **Memory Write List**(3-4-2) in the Run menu, but the two differ as follows.

Item	Write Value to Memory (on-the-spot popup)	Memory Write List (registration window)
How to open it	Right-click a ladder cell, or Ctrl+Shift+W	"Memory Write List" in the Run menu
Character	A one-shot popup attached to the cursor cell	A watch-list window that does not cover the background
Address	Extracted automatically from the selected cell (you can edit it directly)	Registered by you as rows
Target areas	M/D/C/T/R	M/D/C/T/R
Extra features	None (one at a time)	Import, Export, Clear, and Write All

Both write values straight to the running board (monitoring has to be on), and both use the same address syntax (M/D/C/T/R). For the list window, see **Memory Write List**(3-4-2).



< Figure 4-2 The Write Value to Memory popup — the on-the-spot force write window with the selected cell's address filled in automatically >

4-5 Inserting and Deleting Cells

You adjust the arrangement of the grid's squares by inserting or deleting a single cell. Both operations work on the cell the cursor is on (or the cell newly selected by a right-click), and the result is the same whether you run them from the shortcut or from the context menu.

Insert Cell — the **Insert** shortcut, or "Insert Cell" in the context menu

It pushes everything from the square the cursor is on one square to the right. It does not push the whole row unconditionally, though: in each row it pushes only as far as the nearest empty spot (an empty square or an empty horizontal line) and absorbs that spot. If there is a coil at the end of the row, the coil is not pushed

and keeps its place, and for a vertical connecting line between a contact and a coil the squares above and below move together so that the alignment is not broken. That is because an empty spot to absorb the shift is caught before the coil — and **a box function likewise** is not pushed but keeps its place. Run Insert Cell in the last column of the grid and there is no place left to push into, so the grid does not change and the notice toast below appears.

> If you run it in the last column of the grid, or if even one of the affected rows has no empty spot on the right to absorb the shift (that is, it is filled solid all the way to the right end), the grid as a whole does not change and the notice toast "Cannot shift further — the rightmost column is full." appears. It never happens that some rows shift while the rest stay as they are.

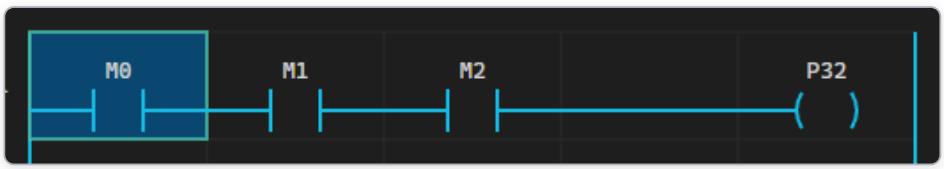
< Figure 4-3 Before Insert Cell — the cell the cursor is on >

< Figure 4-4 After Insert Cell — the cells to the right have been pushed as far as could be absorbed >

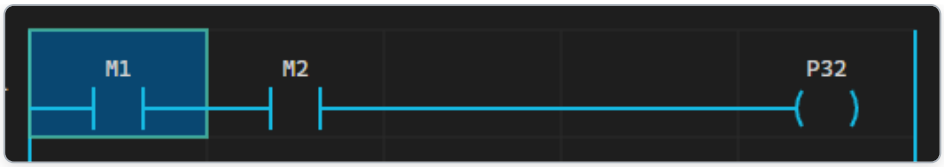
Delete Cell — the **Ctrl+Delete** shortcut, or "Delete Cell" in the context menu

It erases the cell the cursor is on and pulls the cells to its right one square to the left to fill the gap. As with Insert Cell, a coil at the end of the row is not pulled and stays in place, and the gap that opens up in front of the coil is joined with a horizontal line. Put the cursor on the coil's own square and run Delete Cell, and the coil is not deleted — nothing happens at all. What counts as the endpoint (sink) of a row here is four things — the output coil, the SET coil, the RESET coil, **and the box function** — so a box function at the end of a row is protected in just the same way.

> **A vertical connecting line with a contact at both of its ends is not pulled in.** When the top and bottom ends of a parallel branch tied together by a two-row vertical connecting line are **both contacts** and you run Delete Cell on one of them, instead of pulling in the cells on the right that contact alone is turned into a **horizontal line** where it stands — the vertical connecting line is left as it is, so the parallel branch is not broken. If one of the end points is an empty square or a horizontal line, it is not taken for a meaningful parallel branch and is pulled in as usual.



< Figure 4-5 Before Delete Cell — the cell the cursor is on >



< Figure 4-6 After Delete Cell — the cells to the right have been pulled to the left >

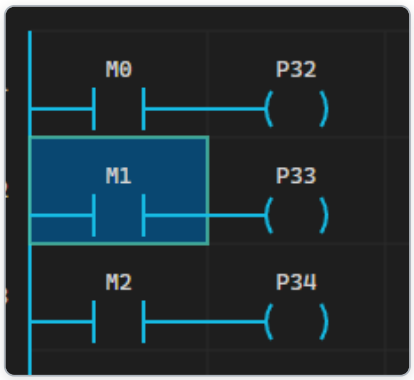
For both Insert Cell and Delete Cell, if you made a mistake you can undo it with **Ctrl+Z** — for details, see **Clipboard and Undo** below.

4-6 Adding and Deleting Rows

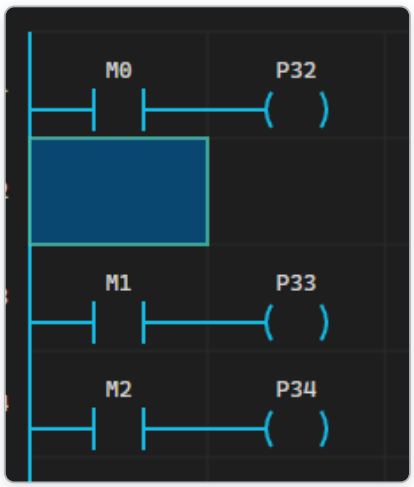
You put an empty row into the grid, or erase an existing row whole, a row at a time.

Insert Row — the **Alt+Insert** shortcut, or "Insert Row" in the context menu

It inserts one empty row above the row the cursor is on. A vertical connecting line that ran through that position is not broken but is automatically extended through the new empty row.



< Figure 4-7 Before Insert Row — the row the cursor is on >

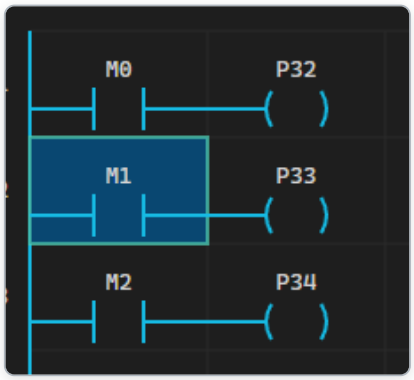


< Figure 4-8 After Insert Row — an empty row has been inserted above the cursor row >

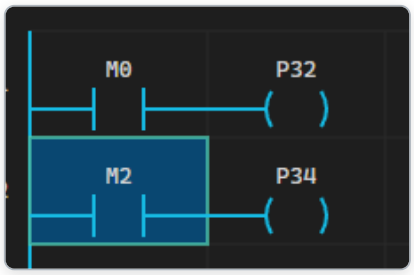
Delete Row — the **Alt+Delete** shortcut, or "Delete Row" in the context menu

It erases the row the cursor is on whole and pulls the rows below it up by one. A vertical connecting line that ran through the deleted row is automatically reconnected.

> Running Delete Row when only one row is left empties the ladder. If you did it by mistake, undo it with Ctrl+Z.



< Figure 4-9 Before Delete Row — the row the cursor is on >



< Figure 4-10 After Delete Row — the rows below have been pulled up >

For both Insert Row and Delete Row, if you made a mistake you can undo it with **Ctrl+Z** — for details, see **Clipboard and Undo** below.

4-7 Clipboard and Undo

Copying, Cutting, and Pasting Cells — the **Ctrl+C** / **Ctrl+X** / **Ctrl+V** shortcuts

These copy, cut, and paste the selected cell (or the cells of a **range selection**(4-2-2)). These shortcuts do what is described above only while the ladder grid has focus; while the code editor has focus they act as ordinary text-editing shortcuts.

Copied and cut cells are also placed on the system clipboard, so you can paste them not only into another ladder tab of the same project but also **between two windows of MPINO Studio 2**. If something was copied in another window, that is what gets pasted first.

Copying and Pasting Only a Box Function Argument Value

Select a single box function cell, put the cursor on an **argument square that has a value in it**, and press Ctrl+C, and **that argument's text** is taken along with the cell. The shortcuts are still the single pair Ctrl+C / Ctrl+V, and **the place you paste into** decides which of the two goes in. The argument text is kept **inside this window only**, though, so when you paste something copied in another window it is always the whole box that goes in.

- If the place you paste into is **the argument square of another box function** → only the value in that square changes. The cell type and the remaining arguments stay as they are, and the argument square

cursor stays where it is, so you can paste one after another.
- If the place you paste into is **an empty square or an ordinary cell** → the whole box function cell is pasted, as before.

The only boxes that take a value-only paste are **fixed-form commands** such as `MOV D0 D1` and **function call** boxes such as `setMotor(100)`. A box whose argument squares cannot be told apart, such as an assignment expression (`D0 = D0 + 1`) or a label, is not a target, and in that case it falls through, with no toast, to a cell paste that puts in the whole box.

Ordinary text copied **from outside the app**, in Notepad or a code editor, can be pasted into an argument square as well. In that case only the **first line that has content in it** is used, with the whitespace at either end stripped off — if there is more than one line with content in it, a notice that some lines were dropped appears with it (it does not appear in the common case where only a single trailing newline came along).

> **Paste a box function cell and the square you pasted into always becomes the first argument position.** So that the result is the same whether you paste into an empty square or into the middle of a stretch joined up by horizontal lines, the app either spreads the box body out to the right from the cursor square (when the right is empty) or shifts the position so that the cursor square becomes the first argument.

A paste is never left quietly in a place that does not line up. A notice toast appears in the cases below.

Situation	Notice
The box's text and its argument breakdown do not line up, so the text range of a square cannot be found	"This function box does not expose its argument positions, so a value cannot be pasted into it. Press Enter to open the box and edit it directly."
Only the first of several lines was put in (the paste succeeded)	"The clipboard held several lines, so only the first line was pasted

Situation	Notice
	into the argument."
There is not enough room for the box body (nothing is pasted)	"There is not enough room for the function box — it needs N columns starting at the cursor. Paste it further left or on an empty row." (N = the number of columns needed)

Undo — the **Ctrl+Z** shortcut

It undoes every operation that changes the grid — inserting and deleting cells, adding and deleting rows, pasting, and so on (a no-op path where nothing changed at all, such as an Insert Cell with the right end full, has nothing to undo in the first place). If you make a mistake while editing, do not hesitate to press Ctrl+Z.

These shortcuts are also summarized in the Chapter 2 shortcut tables (Ladder Clipboard and Editing) — for details, see **Keyboard Shortcuts**(2-3).

5 Contacts

5-1 Contact Types

There are ten cell tools in the contact, line, and coil families that you can place on the ladder grid. They are arranged vertically in the toolbar on the left in F-key order, and the "Name" column in the table below gives the name the app uses for each tool.

Symbol	Name	Description	Shortcut
⊣ ⊢	NO Contact	A contact that is normally (with the condition bit OFF) open and does not conduct, and that closes and conducts once the condition bit turns ON	F2
⊣/⊢	NC Contact	A contact that is normally (with the condition bit OFF) closed and conducts, and that opens and cuts the power flow once the condition bit turns ON	F3
──	Line	Wiring that joins one cell to the next horizontally	F5
|	Vertical	Not a cell type but a toggle that turns the up–down connecting line on the right of the selected cell on and off. Contacts and coils already placed there are kept as they are, and it cannot be attached to a box function cell	F6
−(S)−	Set Coil	Turns the output bit on the moment the condition turns ON, and keeps the value as it is afterwards even if the condition turns OFF (latch)	F7
−(R)−	Reset Coil	Turns the output bit off the moment the condition turns ON, and keeps the value as it is afterwards even if the condition turns OFF (latch)	F8
−( )−	Output Coil	Reflects the condition state in the output bit as it is on every scan (condition ON = output ON, condition OFF = output OFF)	F9
−/−	Signal Invert	Inverts (NOT) the condition accumulated up to that point in the rung and passes it on to the coil	F10
⊣↑⊢	Positive Edge	A contact that conducts for only the single scan in which the condition bit changes from OFF to ON	F11
⊣↓⊢	Negative Edge	A contact that conducts for only the single scan in which the condition bit changes from ON to OFF	F12

There are three ways to apply a tool to a cell.

- Select the cell and then press the shortcut (function key) from the table above.
- Select the cell and then click the button for the tool you want in the toolbar on the left.
- Drag and drop a toolbar button onto the cell you want (it is applied even without a separate selection).

For every tool other than the three coils and the box function, the three ways give the same result. For **the three coils and the box function**, though, where the tool lands differs.

> **Place one of the three coils (F7, F8, F9) or a box function (Ctrl+Enter) with a shortcut and it goes not into the cursor square but into the rightmost square of that row.** A coil is the endpoint (sink) of a row, so the app uses only the **row** the cursor is on, moves the column to the far right, and then fills the

> empty squares in front of it with straight lines (horizontal wire) to join up the power path (it stops when it meets a vertical connecting line coming down from the row above — power already comes into that point). Contacts and addresses already placed are kept as they are. The selection then moves to that square and **the edit window opens automatically.**
>
> **Click** a toolbar button, or **drag and drop** it onto a cell, and this move does not happen — the tool lands in exactly the cell you selected (or dropped it on), and the edit window does not open automatically either. Use the toolbar when you want to put a coil in a particular square yourself.

For the full list of shortcuts, including how to redefine them, see **Keyboard Shortcuts**(2-3); for how to enter an address or a symbol after placing a cell, see "Ladder Cell Editing" in **Pin Editing**(2-4).

The same toolbar also has the Box Function (Ctrl+Enter) tool. Unlike contacts and coils, it is for placing function blocks such as TON and CTU, and a separate chapter covers it in detail.

< Figure 5-1 The ladder toolbar — the ten contact, line, coil, and edge-detection tools laid out in F-key order, with the Box Function button at the very bottom >

5-2 NO Contacts and NC Contacts

NO (Normally Open, A contact) and NC (Normally Closed, B contact) are the two most basic kinds of contact, and their state when the condition bit is off (normally) is the exact opposite of each other.

5-2-1 NO Contact (A Contact)

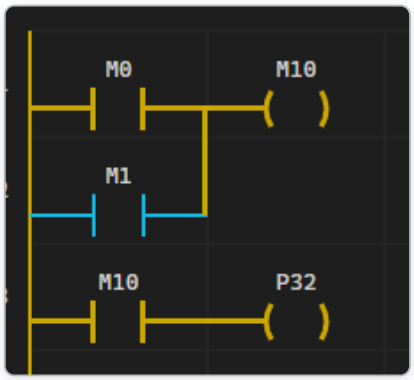
An NO contact is normally (when the condition bit is OFF) open and does not conduct, and it closes and conducts once the condition bit turns ON. You place it with F2 or the **NO Contact** button in the toolbar.

The simplest form is `M0 NO → P32 coil`. If M0 is OFF, P32 is OFF as well; if M0 is ON, P32 is ON as well. The physical input addresses P0–P15 behave in the same way.

< Figure 5-2 A basic NO contact example — M0 OFF, P32 OFF (no power flow) >

< Figure 5-3 A basic NO contact example — M0 ON, P32 ON (power flow highlighted) >

Place several contacts in parallel and they behave as an OR. Put `M0 NO` and `M1 NO` in parallel and take the result with an `M10 coil` (`M0 ‖ M1 → M10 coil`), and M10 turns ON even if only one of the two is ON. When a coil turns on, the contacts that reference the same address (auxiliary contacts) close along with it, so if you follow it with `M10 NO → P32 coil` on the next rung, P32 turns on as well while M10 is ON.



< Figure 5-4 A parallel OR example with NO contacts — if M0 or M1 is ON, P32 turns on by way of M10 >

5-2-2 NC Contact (B Contact)

An NC contact is normally (when the condition bit is OFF) closed and conducts, and it opens and cuts the power flow once the condition bit turns ON — it behaves the exact opposite way to an NO contact. You place it with F3 or the **NC Contact** button in the toolbar.

Place it as `M0 NC → P32 coil` and P32 is ON (conducting) while M0 is OFF, and turns to OFF once M0 turns ON. The physical input addresses P0–P15 behave in the same way.

< Figure 5-5 An NC contact example — M0 OFF, P32 ON (conducting) >

< Figure 5-6 An NC contact example — M0 ON, P32 OFF (power flow cut) >

On the monitoring screen, the contacts and coils that are conducting are shown in the accent color (how the monitoring screen itself is laid out is outside the scope of this chapter).

5-2-3 Constant Contacts (@ON / @OFF)

In the address position of an NO or NC contact you can put a constant that is **always true** or **always false**, instead of an actual memory address. You use it at the head of a rung you want to run on every scan with no condition, or to cut one branch of a circuit temporarily without touching it.

Entry	Meaning
`@ON` or `TRUE`	Always true — a permanently conducting source
`@OFF` or `FALSE`	Always false — a permanently cut source

`@ON` and `TRUE` mean exactly the same thing, and so do `@OFF` and `FALSE`. **Case is not distinguished**, so `@on`, `true`, and `False` are recognized just as they are, and the cell keeps the text you typed unchanged.

Put one in an NC contact and it is **inverted**, as the nature of an NC contact requires. The four combinations therefore give the following results.

Arrangement	Result
NO contact + `@ON`	Permanently conducting
NO contact + `@OFF`	Permanently cut
NC contact + `@ON`	Permanently cut
NC contact + `@OFF`	Permanently conducting

During monitoring too, a constant contact does not fetch a value from the board but is always shown in the same state, as in the table above.

> **Constant contacts (`@ON`, `TRUE`, `@OFF`, `FALSE`) can be used only in NO and NC contacts.** Put a constant in the input position of a rising or falling edge contact (5-5) or in the target address position of a coil and it is rejected at build time. A constant whose value never changes has no edge to detect, and a coil position calls for an address to **write** a value into.

> The similarly named **pulse special contacts** `@<ms>` and `@F<ms>` (**Special Memory**(2-2)) are a separate feature. `@100` is a one-shot pulse that turns ON for a single scan every 100 milliseconds, `@F100` is a pulse that repeats ON and OFF at 100-millisecond intervals, and `@ON` is a constant whose value does not change.

5-3 Coil Output

A coil is the endpoint (sink) placed at the far right of a rung, and it writes the result of the condition accumulated up to that point in the rung into an output bit. Three kinds of coil, which differ in how they write, are provided (Output Coil / SET Coil / RESET Coil), and the last section covers the use of an input pin as a target, which applies to all three coils in common.

5-3-1 Output Coil

An output coil turns the output bit ON when the condition is ON and back OFF when it is OFF — it reflects the condition state in the output bit as it is on every scan. It differs from the SET and RESET coils below in that the value turns off along with the condition. You place it with F9 or the **Output Coil** button in the toolbar.

The example circuit is the same `M0 NO → P32 coil` used in NO Contact (5-2-1). P32 is ON only while M0 is ON, and it returns to OFF immediately once M0 goes back to OFF.

< Figure 5-7 An output coil example — M0 OFF, P32 OFF >

< Figure 5-8 An output coil example — M0 ON, P32 ON >

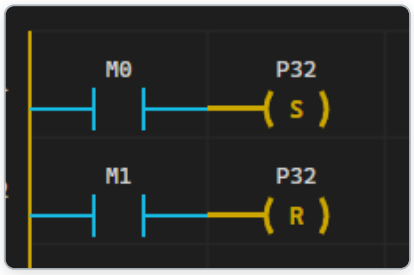
5-3-2 SET Coil

A SET coil writes ON into the output bit the moment the condition turns ON, and it keeps the value as it is even after the condition returns to OFF (latch). You place it with F7 or the **Set Coil** button in the toolbar.

> **Even when the condition of a SET coil returns to OFF, the output bit does not turn off.** To turn the value off you need a separate RESET coil that targets the same address (see the RESET Coil section below).

Place it as `M0 NO → P32 SET` and P32 is written ON the moment M0 turns ON. Even after M0 returns to OFF, P32 keeps holding ON.

< Figure 5-9 A SET coil example — M0 ON, P32 written ON >

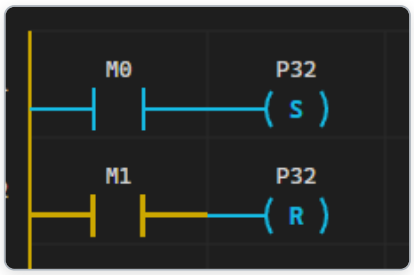


< Figure 5-10 A SET coil example — P32 keeps holding ON even after M0 returns to OFF >

5-3-3 RESET Coil

A RESET coil writes OFF into the output bit the moment the condition turns ON (releasing it), and it keeps the value as it is even after the condition returns to OFF (latch) — the same shape as the SET coil, only in the opposite direction. You place it with F8 or the **Reset Coil** button in the toolbar.

SET and RESET are used together as the standard pair that turns the same address on and off. Lay the two rungs `M0 NO → P32 SET` and `M1 NO → P32 RESET` one after the other and you get a circuit in which M0 turns P32 on and M1 turns it off.



< Figure 5-11 A SET/RESET pair example — P32 switches to OFF the moment M1 turns ON >

P32 in this example is a transistor output. The relay output addresses of a board (P40–P47 on the MPINO-16A8R8T, for example) are handled by a coil in exactly the same way — the board merely decides which physical signal it goes out on, and the behavior of the coil itself (writing ON/OFF) has nothing to do with the type of output.

If you designate a timer (T) or counter (C) address as the target of a RESET coil, it does not act as an ordinary bit release but takes on the special meaning of resetting the accumulated value and the done bit together. The chapter that covers timers and counters explains this in detail.

5-3-4 Using an Input Pin as a Coil Target

For the target address of a coil you can use not only an output pin but also an **input pin** (P0–P15 and the like). PLCs have a customary technique of overwriting a digital input in software, and that is why this is allowed — the representative uses are **contact debouncing** and **deliberately putting a delay** on an input whose signal fluctuates (hunts), such as a water level sensor. All three, the output coil, the SET coil, and the RESET coil, can designate an input pin as their target.

An input pin coil behaves as **a coil that writes to memory only**. A value cannot physically be written to an input pin, so the app produces no hardware output instruction and overwrites only that pin's **memory image**. The contacts of the rungs that follow see this overwritten value.

> **A value written by an input pin coil is held only within that scan.** At the start of every scan the input synchronization reads the physical pin values again and fills the memory image with them, so a value the coil wrote on an earlier scan is overwritten with the physical value at that point. To keep the value, you have to build the circuit so that the coil writes it again on every scan.

For example, drive a timer (TON) from the NO contact of the input `P0` and take its done bit back with a `P0` coil, and the rungs that follow see P0 as ON only while P0 keeps coming in ON for a certain length of time — a chattering signal with a short spike in it is filtered out. To do the same thing with a purely internal bit, use an M address rather than a P address as the coil target.

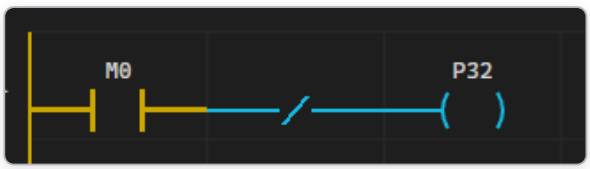
5-4 Signal Invert

Signal Invert (NOT) is not a separate contact but a cell that inverts the rung condition accumulated up to that position and passes it on to the next square. Unlike other contacts, you do not enter an address in the cell. You place it with F10 or the **Signal Invert** button in the toolbar.

> **You can place only one Signal Invert in a single rung (branch).** Place two or more and a compile error occurs at build time.

The simplest form is `M0 NO → Signal Invert → P32 coil`. If M0 is OFF, the inversion makes the condition true and P32 turns ON; if M0 is ON, the inversion makes the condition false and P32 turns OFF — it behaves the exact opposite way to an NO contact (5-2-1).

< Figure 5-12 A basic Signal Invert example — M0 OFF, P32 ON through the inversion >

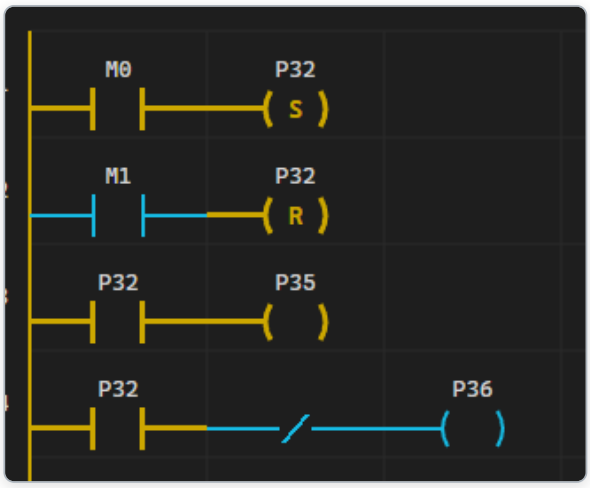


< Figure 5-13 A basic Signal Invert example — M0 ON, P32 OFF through the inversion >

In a lamp application, you split a single state bit with Signal Invert to turn a run lamp and a stop lamp on and off at the same time.

- `M0 NO → P32 SET` — the start signal turns P32 (the motor running state) ON
- `M1 NO → P32 RESET` — the stop signal turns P32 OFF
- A single `P32 NO` contact branches into two coils — the branch that continues straight through is the `P35 coil` (the run lamp), and the branch that goes through Signal Invert is the `P36 coil` (the stop lamp)

If P32 is ON, the run lamp (P35) lights and the stop lamp (P36) goes out. If P32 is OFF, the opposite happens and only the stop lamp lights.



< Figure 5-14 A Signal Invert application — with P32 ON, the run lamp (P35) is lit and the stop lamp (P36) is out >

5-5 Rising and Falling Edge Contacts

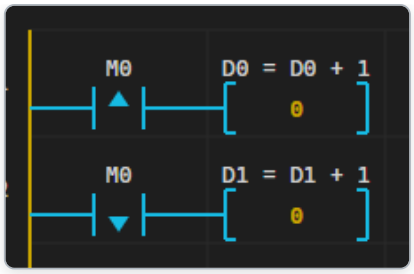
An ordinary contact on the ladder grid keeps responding for as long as the condition is on, but the rising edge contact and the falling edge contact are two kinds of contact that conduct for a single scan only at the moment the condition changes. You use them for logic that you want to run exactly once, such as a count-up or a trigger.

5-5-1 Rising Edge Contact (Positive Edge)

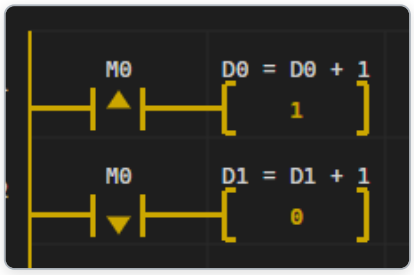
A rising edge contact conducts for only the single scan in which the condition bit changes from OFF to ON. You place it with F11 or the **Positive Edge** button in the toolbar.

It uses one bit internally to remember what state the condition was in on the previous scan. **This bit is internal memory that the compiler manages automatically.** You do not have to designate a separate address for storing the state or give it any thought, and it does not consume user memory such as P or M either — all you enter in the cell is the input address that becomes the condition. Even if you use the same address in several rising or falling edge cells, the state is managed separately for each cell, so they do not affect one another.

The difference from an ordinary contact is clear when you compare examples. Placed without an edge, as `M0 NO → box function D0 = D0 + 1`, D0 keeps increasing on every scan while M0 is ON. Placed as `M0 rising edge contact → box function D0 = D0 + 1`, on the other hand, D0 increases exactly once at the moment M0 turns ON. Turn M0 off and then on again and it increases once more at that point.



< Figure 5-15 A rising edge contact example — M0 OFF, D0 holds its value (no power flow) >



< Figure 5-16 A rising edge contact example — immediately after M0 turns ON, the contact conducts for that single scan and D0 increases by 1 >

You can use the following in the input position of a rising or falling edge contact.

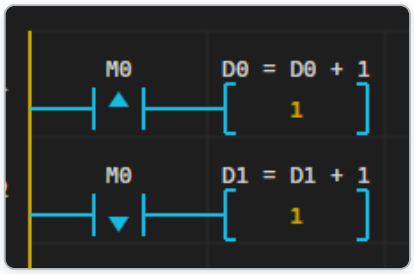
- A bit reference in P, M, D, or C memory (`P0` , `M10` , and so on)
- The done bit of a timer or a counter (`T0` or `C0` , for example)
- The special contacts `@<ms>` and `@F<ms>` (see **Special Memory**(2-2))
- A bit reference in word.bit form (`D0.5` and the like — see Bit Contacts in Word Memory (5-8))

A word value that is not a bit, such as D0, cannot be used as an input, and neither can a constant contact (`@ON` , `TRUE` , `@OFF` , `FALSE` — 5-2-3), which has no edge to detect.

5-5-2 Falling Edge Contact (Negative Edge)

A falling edge contact conducts for only the single scan in which the condition bit changes from ON to OFF. You place it with F12 or the **Negative Edge** button in the toolbar. The way it manages its state and what you can use as an input are the same as for the rising edge contact (5-5-1).

Place it as `M0 falling edge contact → box function D1 = D1 + 1` and D1 increases exactly once at the moment M0 changes from ON to OFF.



< Figure 5-17 A falling edge contact example — the moment M0 changes from ON to OFF, the contact conducts for that single scan and D1 increases by 1 >

5-6 Comparison Contacts

A comparison contact is not a separate cell type; you make one by putting a comparison expression, instead of an address, into the address field of an NO or NC contact. The contact conducts while that expression is true. You enter it in the same cell edit modal field as for any other contact — see "Ladder Cell Editing" in **Pin Editing**(2-4).

The following six comparison operators are supported.

Notation	Meaning
`>`	Conducts if the left side is greater than the right side
`<`	Conducts if the left side is less than the right side
`>=`	Conducts if the left side is greater than or equal to the right side
`<=`	Conducts if the left side is less than or equal to the right side
`==`	Conducts if the left side and the right side are equal
`!=`	Conducts if the left side and the right side are different

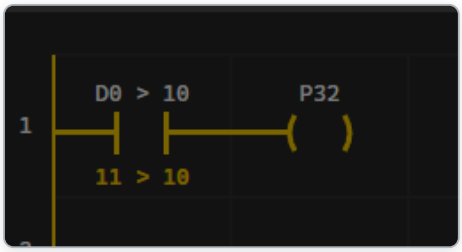
For example, `D0 > D1` conducts when D0 is greater than D1.

> **The `=>`, `=<`, `=`, and `<>` notations of the old MP STUDIO are not supported.** You must use the canonical notations in the table above (`>=`, `<=`, `==`, `!=`).

A comparison expression can be used only in NO and NC contacts (it cannot be used in edge contacts or coils). Also, only one comparison expression goes into a single square — you cannot join several comparison expressions with `&&` or `||`, and to use several conditions together you express them by placing contacts in series (AND) or in parallel (OR). Put a comparison expression in an NC contact and the behavior is inverted, so it conducts while the expression is false.

On the left and right sides of the expression you can put not only memory addresses (word values such as D, C, and T, and float values such as R) and numeric literals but also symbol names as they are, so you can write a registered name directly in the expression instead of an address, as in `Temp > 100`. For how to register and manage symbols (address aliases), see **Symbols**(9-5).

Place `D0 > 10 (NO comparison contact) → P32 coil` as an example and P32 turns on when D0 exceeds 10. During monitoring, the current values of both sides are shown together underneath the comparison expression (`6 > 10`, for example), and the row height of that cell grows automatically by as much as the added values require.



< Figure 5-18 A comparison contact monitoring example — with D0=11 the expression becomes true, so the contact conducts and shows the value 11 > 10, and P32 is ON >

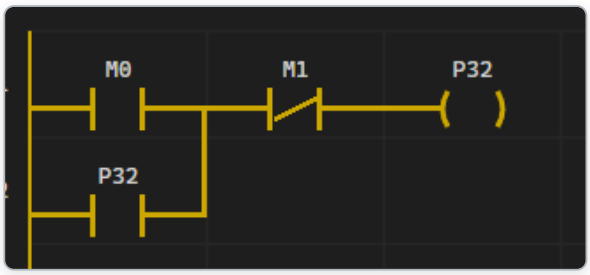
5-7 Self-Holding Circuits

When the start signal (M0) turns ON, the output (P32 — a motor, for example) turns ON, and that output's own auxiliary contact (P32 NO) closes along with it, so the circuit stays on even if you turn M0 back OFF. When the stop signal (M1, an NC contact) turns ON, the circuit is broken and P32 returns to OFF. A circuit that holds its own ON state like this, by means of the output's auxiliary contact, is called a self-holding circuit.

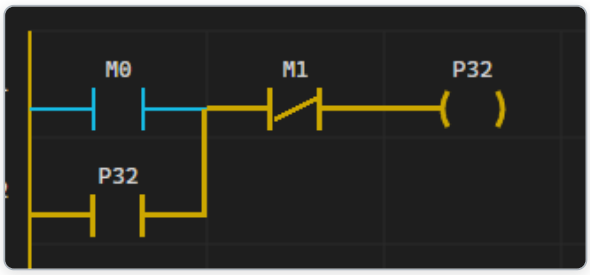
The circuit is `[M0 NO ‖ P32 NO] → M1 NC → P32 coil` — you place the start signal M0 and the auxiliary contact of P32 in parallel, follow them in series with the stop signal M1 (NC), and take the end with a P32 coil. The physical input addresses P0–P15 behave in the same way.

The sequence of operation is as follows.

1. With no signal at all, P32 is OFF.
2. When M0 turns ON (M1 is not pressed, so its NC contact is closed), P32 turns ON and, at the same time, the auxiliary contact of P32 closes as well.
3. Even if you return M0 to OFF, the P32 auxiliary contact that has just closed holds the parallel path in its place, so P32 keeps holding ON.
4. When M1 turns ON, its NC contact opens, the circuit is broken, and P32 returns to OFF while its auxiliary contact opens along with it.



< Figure 5-19 A self-holding circuit example — the moment M0 turns ON, P32 turns ON and its auxiliary contact conducts as well >



< Figure 5-20 A self-holding circuit example — P32 keeps holding ON through the auxiliary contact path even after M0 returns to OFF >

The pair made of a SET coil and a RESET coil (5-3-2, 5-3-3) achieves the same purpose of holding the value even after the condition disappears. A self-holding circuit is the classic way of producing the same effect with nothing but an arrangement of NO and NC contacts, without SET and RESET.

5-8 Bit Contacts in Word Memory

Attach `.n` (a bit number) after an address in a word area and you can use just one bit inside that word as a contact or a coil (`D0.5`, for example, is bit 5 of D0). The areas in which the whole address notation (the

native, byte (B), word (W), and double-word (D) views plus `.n` bit access) can be used are covered in "Prefixes and Bit Access" in **Data Memory**(2-1) — here we cover the correspondences and the restrictions that apply when you actually use that notation in a contact or coil position.

When you reference the P and M areas in 16-bit units through the word (W) view, (word index × 16) + bit number is the physical bit it actually points at. The byte (B) view is calculated the same way with ×8, and the double-word (D) view with ×32. The representative correspondences are as follows.

Notation	Corresponds to	Calculation
`WP0.1`	P1	word index 0 × 16 + bit 1
`WM1.15`	M31	word index 1 × 16 + bit 15
`D10.0` – `D10.15`	bits 0–15 of D10	the native unit is already a word, so you designate the bit directly with no view prefix

The range in which a width view can be used differs from area to area — P and M support all three of the byte (B), word (W), and double-word (D) views, D, C, and T support only the double-word (D) view, and R has no width views at all. The bit range depends on the width: B is 0–7, W is 0–15, and D is 0–31.

However, the only things you can actually use in a contact or coil position with `.n` attached are P and M (by way of a width view) and D and C (the native word or the double-word (D) view). A T (timer) address can be written through the double-word (D) view, but it cannot be used as a contact with `.n` attached.

> **Only bit-value references can be used in a contact or coil position.** A raw word value with no `.n`, such as `D0` or `WM0`, cannot be used as a contact or a coil (it is rejected at build time). To compare a word value as it is, use a comparison expression (5-6) rather than a contact.

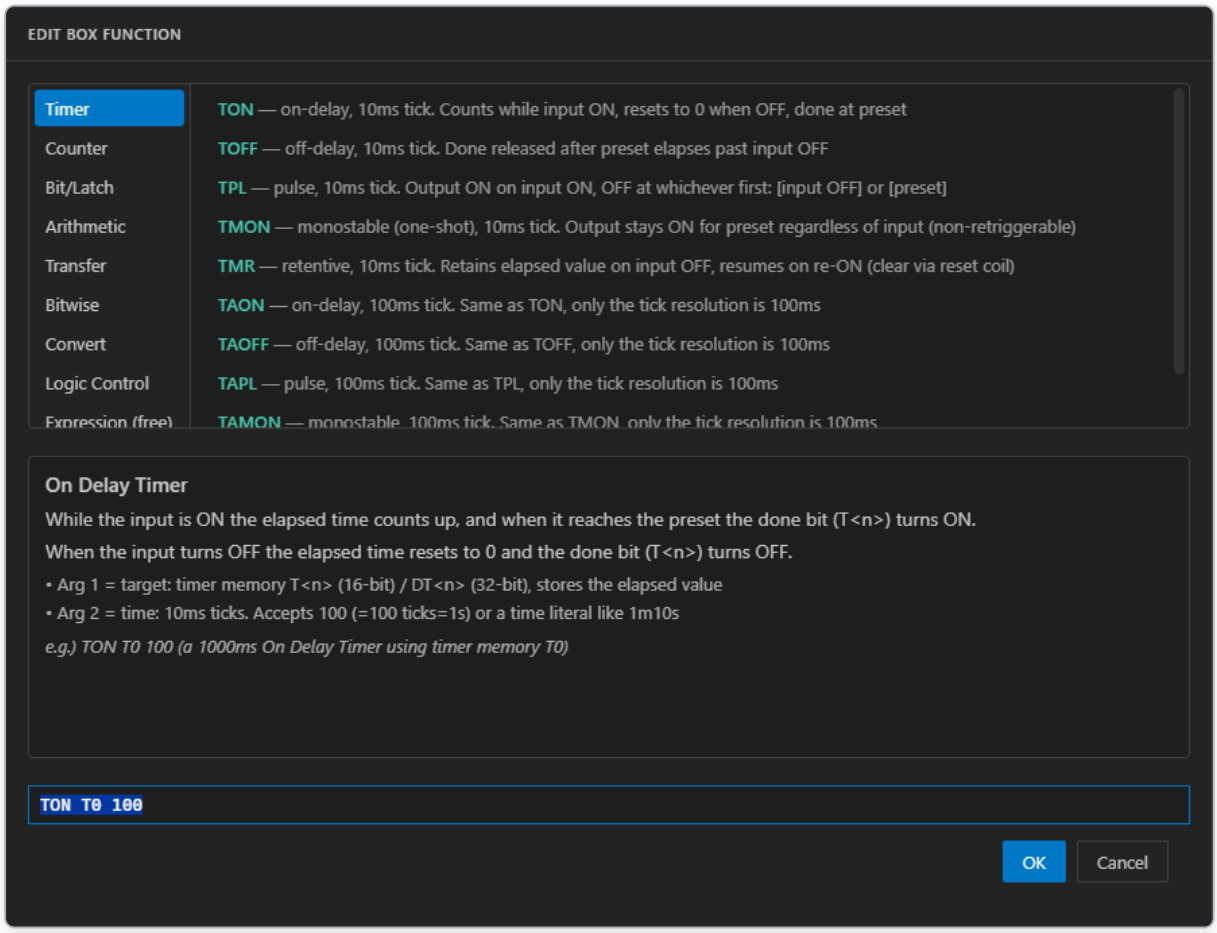
For example, place the two rungs `D0.0 NO → P32 coil` and `D0.1 NO → P33 coil` and put 1 into D0: because 1 has only bit 0 set in binary, only P32, which looks at D0.0, turns on, and P33, which looks at D0.1, stays off — an example that shows that a word value itself and the individual bits inside it are different things.

< Figure 5-21 A word bit contact example — with D0=1, only the D0.0 contact (rung 1) conducts, while the D0.1 contact (rung 2) does not >

6 Box Functions

A box function is a cell you place on the ladder grid just like a contact or a coil, but its position is the same as a coil's — the far right of the rung (the endpoint). If the condition carried up to that rung is true, it executes the instruction written in the cell. It works on the same principle as an output coil, which writes the result of a condition into a single bit as it is; the only difference is that it "executes an instruction" when the condition is true. Everything from behavior that builds up state across several scans, such as timers and counters, through one-shot calculations such as basic arithmetic and bitwise operations, to calls to arbitrary Arduino functions, is handled at this position.

You place a box function cell in an empty cell with **Ctrl+Enter** (or with the **Box Function** button in the toolbar on the left). Double-click the cell you placed, or select it and press **Enter**, and the **Edit Box Function** window opens. At the top of the window is an instruction palette grouped by category: click the instruction you want and a template is filled into the input box (for example `TON T0 100`), and the description panel below it shows the definition, arguments, and example of the instruction you selected. Typing code directly into the input box without going through the palette saves it in exactly the same way.



< Figure 6-1 The Edit Box Function window — the instruction palette grouped by category and the description panel for the selected instruction >

Box functions fall into two branches: the predefined instructions the app offers through the palette, and the free-form ones you write directly into the input box. The predefined instructions are grouped into eight categories — Timer, Counter, Arithmetic, Transfer, Bitwise, Convert, Bit/Latch, and Logic Control — and this

chapter describes them category by category, starting at 6-1. The free-form kind covers assignment expressions such as `D0 = D0 + 1` and calls to arbitrary functions such as `setMotor(100)`, and 6-9 covers it separately.

If you place no contact at all to the left of a box function cell, the condition is treated as always true. How contacts and coils themselves behave is covered in **Contact Types** (Chapter 5), and the dedicated special memory C (counters) and T (timers) that keep appearing in this chapter was already covered in **Data Memory**(2-1).

What a box function looks like

A box function cell is drawn as a wide table. At the top is a **band** carrying the instruction name, and below it the cell is divided into **one slot per parameter** — for `ADD D0 D1 D2` the band reads `ADD` and the three slots below it read `D0`, `D1`, and `D2`. The number of slots is fixed for each instruction, so if you write fewer arguments than that, the empty positions remain as dotted slots and you can see exactly how many are missing.

- **The slot that is written to is shaded.** It marks the slot the result is stored in — `D2` in `ADD D0 D1 D2`, `D1` in `MOV D0 D1` — which helps most in SCALE (6-3-3), where the destination is not the last token. JMP·JMPN·MCS·MCSCLR (6-8), which write to nothing, have no shading.
- The box **takes over the neighboring cells in the same row** for as many slots as it has. It uses the empty cells to its right first, and if those are not enough it covers the horizontal connecting line running to its left. A covered cell loses its outline so that the box reads as one piece, and the box never crosses a vertical connecting line that stands in the way.
- **Each slot shows its value and its symbol separately.** If a symbol (9-5) is registered for that address, the symbol name is shown as the main text, and while ladder monitoring (8-5) is running each slot also carries the current value at that position. A timer target slot is converted into a time rather than shown as a tick count, and if you write a T or C address in the up and down slots of CTUD (6-2-3), the slot shows the done bit as a contact would, not the word value.
- In an argument position you can write a **registered symbol name** instead of an address — write `MOV Temp D0`, for example; see **Symbols**(9-5) for registering and managing symbols.

A free-form box function (6-9) is not divided into slots — the band reads `code`, and the expression you wrote appears below it as a single line, as it is.

Editing one argument slot at a time

You edit a box function one slot at a time. Double-click a slot, or put the cursor on it and press **Enter**, and the edit window opens with **only the text of that slot selected**, so typing straight away does not wipe out the whole line. Inside the box, `←`/`→` move from one argument slot to the next, and the slot the cursor is on gets an outline (the shading of the slot that is written to is a fill, so you can still tell the two apart on the same slot). Press once more at the last argument and the cursor leaves the box. The instruction-name band is not a cursor target, so to change the instruction itself, change it in the edit window.

Copy and paste work slot by slot too. The keys are the usual **Ctrl+C/Ctrl+V**, and what decides the outcome is **where you paste** — paste onto an argument slot of another box function and **only that slot's value** changes; paste onto an empty slot or an ordinary cell and the whole box goes in. When a whole box is pasted, it is placed so that **the position you pasted at is always the first argument**, and if there is not enough room or a vertical connecting line is in the way, it is not pasted and the app tells you why.

On a box that has no slot cursor, `←`/`→` skip over the box as a whole — that means a free-form box function (6-9), which is not divided into slots, and a box drawn compressed because it is narrower than its slot count. The unit of a range selection and of a Ctrl toggle is still the cell, not the argument, and when a box is caught in one it is handled as a single piece, out to the horizontal line it covers on its left. For moving the cell cursor and making a range in the first place, see "Cursor Movement and Selection" in **Ladder Grid Editing** (Chapter 4).

> **In a branching rung that feeds two or more box functions from one condition, only eight predefined instructions can be used.** In a circuit where one contact splits with a vertical connecting line and feeds several box functions at once, the only predefined instructions the compiler allows are the eight **ADD·SUB·MUL·DIV·MOD·INC·DEC·MOV**. The assignment expressions and function calls of the free-form kind (6-9) can always be used in a branch as well. The other 36 — timers (6-1), counters (6-2), SCALE·FLOOR·SQRT (6-3), the block transfers GMOV·FMOV (6-4-2), bitwise (6-5), convert (6-6), SET/RST (6-7), and logic control (6-8) — are **rejected by the compiler** in that position. They are instructions that accumulate state or change the execution flow, and how they should mesh with a parallel branch has not been settled yet. If you want to use several such instructions, draw the condition contact separately on each rung and keep one box function per rung.

6-1 Timers

There are ten timer box functions in total: five behaviors (TON·TOFF·TPL·TMON·TMR) multiplied by two tick resolutions — the unprefixed form (`TON`, for example) is 10ms per count, and the version prefixed with `TA` (`TAON`, for example) is 100ms per count. All five behaviors share the same syntax.

```
<function> <target> <preset>
```

- **Target**: the timer memory address that stores the elapsed value. Use `T<n>` (16-bit, one elapsed value) or `DT<n>` (32-bit, writing the two words `T<n>` and `T<n+1>` together).
- **Preset**: the number of ticks needed to complete. Write an integer such as `100` and it waits for that many ticks (10ms without a prefix, 100ms with the `TA` prefix). You can also write it as a time unit such as `1m10s`, `500ms`, or `2s` (h/m/s/ms can be combined), which is converted internally into a tick count when you save. The preset has an **upper limit set by the width of the target**, and compilation is rejected if the converted tick count exceeds that width — `T<n>` (16-bit) holds at most **65535 ticks**, so the unprefixed `TON T0 1h` (360000 ticks) is an error, while the same one hour as `TAON T0 1h` (36000 ticks, at 100ms per tick) passes. If you need a longer time, use the 32-bit target `DT<n>` (up to 4294967295 ticks).

The same name means different things depending on where it sits. `T0` written in the box function position (the target) points at the elapsed value, but **placing that same `T0` in a contact position points not at the elapsed value but at the done bit**. When the elapsed value reaches the preset, the done bit turns ON, and you can read that state with a `T0` contact and pass it on to the next coil.

If you designate a timer target (`T<n>`/`DT<n>`) as the destination of a **RESET Coil**(5-3-3), it acts with the special meaning of returning both the elapsed value and the done bit to 0, rather than simply clearing a bit. TON·TOFF·TPL·TMON reset naturally from the input state alone, but TMR, whose behavior is retentive, keeps its value as it is even when the input turns off, so a RESET coil is needed to reset it partway through (see the TMR section below).

None of the ten timers **can be used in a branching rung** — place a timer where one contact splits with a vertical connecting line and feeds several box functions, and compilation is rejected (see the warning in the introduction to this chapter). Draw the condition contact separately on each rung and keep one timer per rung.

6-1-1 TON — On Delay Timer

Definition

TON (On Delay Timer) is a timer that accumulates elapsed time while the input is ON and turns the done bit ON once that value reaches the preset. When the input returns to OFF, the elapsed value and the done bit are reset to 0 immediately. It is the most basic delay behavior, in which "completion" requires the input to be held on for the preset time.

Usage

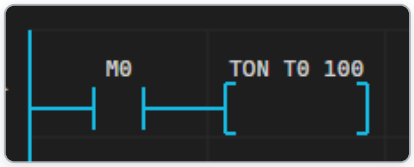
```
TON <target> <preset>
```

`TON T0 100` uses T0 as the elapsed-value target and turns the done bit on after 100 ticks (= 100 × 10ms = 1 second). If you need ticks in units of 100ms, use `TAON`, which adds the `TA` prefix.

Mnemonic	Tick	Meaning
`TON`	10ms per count	The unprefixed version. `TON T0 100` = 100 × 10ms = 1 second
`TAON`	100ms per count	The `TA`-prefixed version. Behaves the same as TON; `TAON T0 30` = 30 × 100ms = 3 seconds

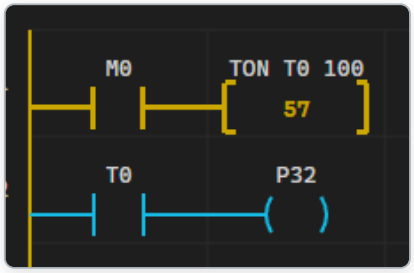
Example

Place it as `M0 NO contact → TON T0 100` and, while M0 is ON, elapsed time builds up, and once 1 second has passed the done bit T0 turns ON.

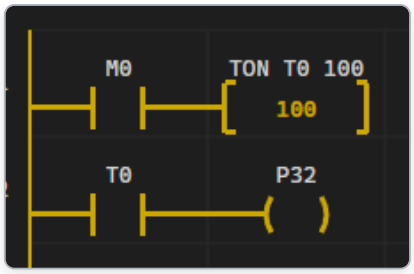


< Figure 6-2 A TON example circuit — driving TON T0 100 from an M0 NO contact >

To take the done bit on another rung, wire it as `T0 NO contact → P32 coil`, and P32 turns on together with T0 when T0 conducts as the done bit.



< Figure 6-3 TON monitoring — the elapsed value increasing while M0 stays ON >



< Figure 6-4 TON monitoring — the elapsed value has reached the preset (100), so the done bit T0 conducts as ON >

Notes

> **A preset written in time syntax has to be a multiple of that tick.** For example, with no prefix (a 10ms tick), writing a time that is not a multiple of 10, such as `TON T0 15ms`, becomes a Syntax error at compile time. Use an integer tick count (`100`) or a time that is a proper multiple (`150ms` , `1s`).

- If the target index exceeds the number of timer memory entries configured for the project, compilation is refused. To adjust the count, see "Area Sizes and Memory Settings" in **Data Memory**(2-1).
- If the input returns to OFF even once, the elapsed value is reset to 0. If you want to keep counting on without a break in between, use TMR (the retentive timer, 6-1-5).

6-1-2 TOFF — Off Delay Timer

Definition

TOFF (Off Delay Timer) is a timer that turns the done bit ON immediately, with no delay, while the input is ON, and that returns the done bit to OFF only once the elapsed time counted from the moment the input changed to OFF reaches the preset. It is called an "off delay" in the sense that turning on is immediate and only turning off is delayed.

Usage

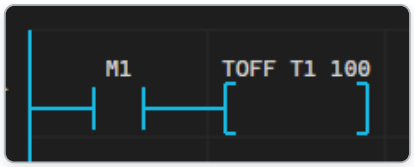
```
TOFF <target> <preset>
```

With `TOFF T1 100`, the done bit turns off only after 100 ticks (= 1 second) have passed since the input changed to OFF. For ticks in units of 100ms, use `TAOFF`.

Mnemonic	Tick	Meaning
`TOFF`	10ms per count	The unprefixed version. `TOFF T1 100` = the done bit turns OFF 1 second after the change to OFF
`TAOFF`	100ms per count	The `TA`-prefixed version. Behaves the same as TOFF; `TAOFF T1 100` = OFF 10 seconds after the change to OFF

Example

Place it as `M1 NO contact → TOFF T1 100` and, while M1 is ON, the done bit T1 holds ON immediately. From the moment M1 changes to OFF the elapsed time starts to build up, and the done bit is released to OFF only once 1 second has passed.



< Figure 6-5 A TOFF example circuit — driving TOFF T1 100 from an M1 NO contact >

Notes

- If the input repeats ON and OFF quickly, the elapsed time builds up again from the beginning each time it changes to OFF — the done bit can keep holding ON and never turn off.
- The rules for the target index range and the preset time syntax are the same as for TON (6-1-1).

6-1-3 TPL — Pulse Timer

Definition

TPL (Pulse Timer) is a timer that turns the done bit on at the instant the input changes from OFF to ON — the rising edge — and starts counting the elapsed time. After that, the done bit turns off automatically at whichever comes first, **the input returning to OFF** or **the elapsed value reaching the preset** — even if the input is held for a long time, the pulse length never exceeds the preset.

Usage

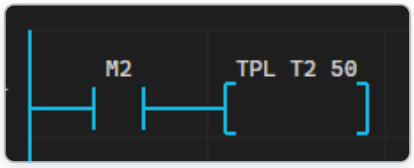
```
TPL <target> <preset>
```

`TPL T2 50` holds the done bit ON for at most 50 ticks (= 0.5 seconds) from the instant of the input rising edge. For ticks in units of 100ms, use `TAPL`.

Mnemonic	Tick	Meaning
`TPL`	10ms per count	The unprefixed version. `TPL T2 50` = a pulse of at most 0.5 seconds from the rising edge
`TAPL`	100ms per count	The `TA`-prefixed version. Behaves the same as TPL; `TAPL T2 50` = a pulse of at most 5 seconds

Example

Place it as `M2 NO contact → TPL T2 50` and, at the moment M2 turns ON (the rising edge), the done bit T2 turns on immediately and a 0.5-second count starts. If M2 returns to OFF first in the meantime, or the 0.5 seconds fills up — whichever comes first — the done bit turns off automatically.



< Figure 6-6 A TPL example circuit — driving TPL T2 50 from an M2 NO contact >

Notes

- Even if the input keeps holding ON, the pulse lasts only for the preset time and then turns off automatically — it does not behave like an ordinary contact that "keeps conducting while it is ON".
- The rules for the target index range and the preset time syntax are the same as for TON (6-1-1).

6-1-4 TMON — Monostable / One-shot Timer

Definition

TMON (Monostable / One-shot Timer) is a timer that turns the done bit on with a single input rising edge and then holds the done bit ON, **regardless of the input state**, until the preset time fills up. While the preset is running, an input that changes from OFF to ON again (a retrigger) is ignored, and the count that started first carries on as it is.

Usage

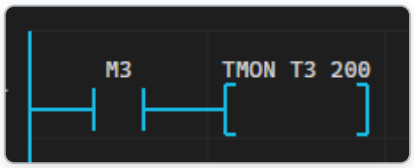
```
TMON <target> <preset>
```

`TMON T3 200` holds the done bit ON for 200 ticks (= 2 seconds) from the instant of the input rising edge. For ticks in units of 100ms, use `TAMON`.

Mnemonic	Tick	Meaning
`TMON`	10ms per count	The unprefixed version. `TMON T3 200` = held ON for 2 seconds from the rising edge
`TAMON`	100ms per count	The `TA`-prefixed version. Behaves the same as TMON; `TAMON T3 200` = held ON for 20 seconds

Example

Place it as `M3 NO contact → TMON T3 200` and, at the moment M3 turns ON, the done bit T3 turns on, and after that it stays on until the 2 seconds have filled up, regardless of the state of M3.



< Figure 6-7 A TMON example circuit — driving TMON T3 200 from an M3 NO contact >

Notes

> **It is easy to confuse this with TPL.** TPL turns off immediately when the input returns to OFF, even before the preset, whereas TMON, once it has turned on, turns off only when the preset ends, regardless of the input state.

- A new rising edge that arrives while the done bit is holding ON does not retrigger it and is ignored (non-retriggerable).
- The rules for the target index range and the preset time syntax are the same as for TON (6-1-1).

6-1-5 TMR — Retentive Timer

Definition

TMR (Retentive Timer) is a timer that builds up elapsed time only while the input is ON and that stops, leaving that value as it is, when the input turns OFF. Unlike TON, the elapsed value is not reset when the input turns off, and when the input turns ON again it carries on accumulating from the value where it stopped — it is for adding up all the time spent ON across several occasions until the preset is reached.

Usage

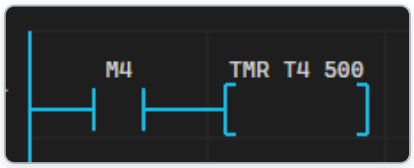
```
TMR <target> <preset>
```

With `TMR T4 500`, the done bit turns on once the total time the input has been ON adds up to 500 ticks (= 5 seconds). For ticks in units of 100ms, use `TAMR`.

Mnemonic	Tick	Meaning
`TMR`	10ms per count	The unprefixed version. `TMR T4 500` = complete at 5 seconds of accumulated ON time
`TAMR`	100ms per count	The `TA`-prefixed version. Behaves the same as TMR; `TAMR T4 500` = complete at 50 seconds of accumulated ON time

Example

Place it as `M4 NO contact → TMR T4 500` and elapsed time builds up only while M4 is ON, and stops when it turns OFF. When M4 turns ON again it carries on counting from the value where it stopped, and once all the ON time adds up to 5 seconds the done bit T4 turns on.



< Figure 6-8 A TMR example circuit — driving TMR T4 500 from an M4 NO contact >

Notes

> **With TMR, the elapsed value is not reset by the input alone.** To return the accumulated value to 0 partway through, you have to place a separate **RESET Coil**(5-3-3) that targets T4 — place it as `M5 NO contact → T4 RESET` and, at the moment M5 turns ON, the elapsed value and the done bit of T4 both become 0.

- The rules for the target index range and the preset time syntax are the same as for TON (6-1-1).

6-2 Counters

There are three counter box functions in total (CTU·CTD·CTUD). Unlike the timers there is no tick-prefixed version (TA), and they count the number of rising edges of the input rather than elapsed time. CTU and CTD

share the same syntax (`<function> <target> <preset>`), but CTUD uses a four-argument syntax that takes the up and down signals separately (`CTUD <target> <up bit> <down bit> <preset>`).

- **Target**: the counter memory address that stores the count value. Use `C<n>` (16-bit, one count value) or `DC<n>` (32-bit, writing the two words `C<n>` and `C<n+1>` together).
- **Preset**: the number of counts needed to complete. Unlike the timers, time syntax (`1s`, `500ms`, and so on) is not supported; write only an integer such as `5`.

As with the timers, the same name means different things depending on where it sits. `C8` written in the box function position (the target) points at the count value, but placing that same `C8` in a contact position points not at the count value but at the done bit. When the count value reaches the preset, the done bit turns ON, and you can read that state with a `C8` contact and pass it on to the next coil.

If you designate a counter target (`C<n>` / `DC<n>`) as the destination of a RESET coil (5-3-3), it acts with the special meaning of returning both the count value and the done bit, rather than simply clearing a bit. The value it returns to differs by counter type — CTU·CTUD return to 0, but CTD returns to the configured preset value (a down counter has to start from the preset so that it can count again; see the CTD section below).

Like the timers, the three counters **cannot be used in a branching rung** either — place one where a single contact feeds several box functions at once and compilation is rejected (see the warning in the introduction to this chapter).

6-2-1 CTU — Count Up Counter

Definition

CTU (Count Up Counter) is a counter that increases the count value by 1 every time a rising edge — a change of the input from OFF to ON — occurs. When the count value reaches the preset the done bit turns ON, and even if further input rising edges arrive after that, the done bit stays on while only the count value keeps increasing. Once the count value reaches its upper limit (65535 for 16-bit, 4294967295 for 32-bit) it increases no further and is held as it is (saturate).

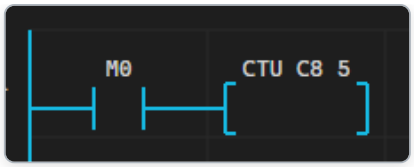
Usage

```
CTU <target> <preset>
```

With `CTU C8 5`, the done bit turns on once the input has produced five rising edges. The CTU helper has rising-edge detection built in, so even if you leave an ordinary NO contact as the input, it is counted only once each time the contact changes from OFF to ON (you do not need to place a separate rising edge contact in front of it).

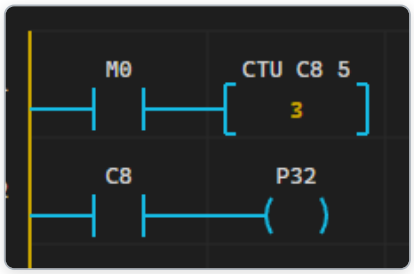
Example

Place it as `M0 NO contact → CTU C8 5` and the count value increases by 1 every time M0 changes from OFF to ON (a rising edge), and at the fifth rising edge the done bit C8 turns ON. Because rising-edge detection is inside the helper, you use an ordinary NO contact as the input as it is, without placing a separate rising edge contact in front of it.

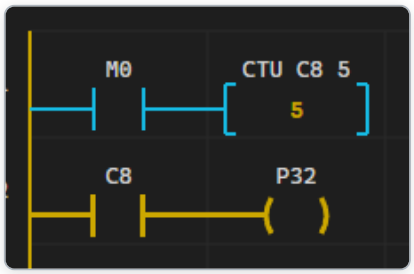


< Figure 6-9 A CTU example circuit — driving CTU C8 5 from an M0 NO contact >

To take the done bit on another rung, wire it as `C8 NO contact → P32 coil`, and P32 turns on together when the count value reaches the preset and C8 conducts as the done bit.



< Figure 6-10 CTU monitoring — the count value increasing each time an M0 rising edge repeats >



< Figure 6-11 CTU monitoring — the count value has reached the preset (5), so the done bit C8 conducts as ON >

Notes

> The preset can only be an integer. Unlike a timer preset, time syntax such as `1s` or `500ms` is not supported, and using such a notation becomes a Syntax error at compile time.

- If the target index exceeds the number of counter memory entries configured for the project, compilation is refused. To adjust the count, see "Area Sizes and Memory Settings" in **Data Memory**(2-1).
- If input rising edges keep arriving after the done bit has turned on, the count value keeps increasing up to its upper limit. To return the count value and the done bit, use a RESET coil (see the introduction above).

6-2-2 CTD — Count Down Counter

Definition

CTD (Count Down Counter) is a counter whose count value is **filled with the preset automatically when it first runs**, and which decreases the count value by 1 every time an input rising edge occurs. When the count value reaches 0 the done bit turns ON, and it decreases no further and is held at 0 (saturate, no negatives). To return it to the preset again during operation, reset that counter with a RESET coil (a reset also fills it with the preset rather than 0). It is a counter that counts in the opposite direction to CTU.

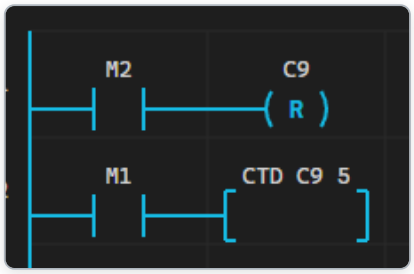
Usage

```
CTD <target> <preset>
```

With `CTD C9 5`, the count value is filled with 5 on the first run, and once five input rising edges have occurred and the count value reaches 0, the done bit turns on.

Example

Place `M1 NO contact → CTD C9 5` and the count value starts at the preset 5 on the first run and decreases by 1 every time M1 produces a rising edge. At the fifth edge the count value becomes 0 and the done bit C9 turns on. As with CTU, CTD has rising-edge detection built into the helper, so you use an NO contact as the input as it is. Add `M2 NO contact → C9 RESET` as in the figure and you can return the count value to the preset at any time during operation (the first-run preset is automatic, so you do not have to place a reset).



< Figure 6-12 A CTD example circuit — driving CTD C9 5 from an M1 NO contact (the preset 5 is loaded automatically on the first run), with an M2 RESET reloading the preset >

Notes

> Because **the count value of a CTD is filled with the preset automatically on the first run (at boot)**, counting down starts from the preset even if you do not add a separate reset before using it. You use a reset (a RESET coil) when you want to return the count value to the preset again during operation.

- The preset can only be an integer, and the rule for the target index range is the same as for CTU (6-2-1).
- **If you have set the C area to be retentive, the value at boot is the saved value, not the preset.** The boot sequence loads the CTD automatic preset first and restores the retentive values from EEPROM afterwards, so the retentive restore, which runs later, overwrites the automatic preset — that is, the count value from just before the power was lost carries on as it is, which is exactly what turning retention on is for. If you want it to start from the preset on every boot, take that counter out of the retentive set, or drive a RESET coil (5-3-3) from a signal that turns on only once right after boot.

6-2-3 CTUD — Count Up/Down Counter

Definition

CTUD (Count Up/Down Counter) is a counter that takes an up signal and a down signal as separate bit contacts, increasing the count value by 1 every time the up signal produces a rising edge and decreasing it by 1 every time the down signal produces a rising edge. Unlike CTU and CTD, the contact you place on the left of the rung is not a count input but acts only as an enable (a gate), and while that contact is off the count value does not move even if the up and down signals change. When the count value reaches the preset (in the up direction) the done bit turns on, and it saturates at the upper and lower limits respectively (see the introduction above).

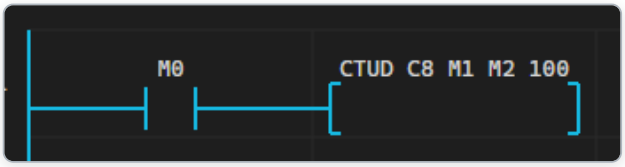
Usage

```
CTUD <target> <up bit> <down bit> <preset>
```

Unlike CTU·CTD, it has four arguments. Place it as `CTUD C8 M1 M2 100` and the contact on the left of the rung is the enable, M1 is the up signal, M2 is the down signal, and 100 is the preset. In the up and down positions you can use the same inputs as an ordinary contact as they are (bit addresses in P/M/D/C and so on, including the timer and counter done bits `T0` and `C0`).

Example

Place it as `M0 NO contact → CTUD C8 M1 M2 100` and, only while M0 is ON, the count value increases by 1 on a rising edge of M1 and decreases by 1 on a rising edge of M2. Note that the M0 contact on the left of the rung is not a count input but acts as the enable (the gate).



< Figure 6-13 A CTUD example circuit — driving CTUD C8 M1 M2 100 from an M0 enable contact (up=M1, down=M2) >

Notes

> If up and down produce rising edges at the same time in the same scan, both are ignored and the count value stays as it is (the same behavior as the IEC 61131-3 standard counter).

- Even if the up and down signals change while the enable contact is off, those changes are recorded as already reflected — when the enable turns back on, the changes that happened while it was off are not counted again.
- The preset can only be an integer (the same as CTU·CTD; none of the timers' time syntax).
- The rule for the target index range is the same as for CTU (6-2-1).

6-3 Arithmetic

There are ten arithmetic box functions in total: five basic arithmetic operations (ADD·SUB·MUL·DIV·MOD), two increment and decrement operations (INC·DEC), scale conversion (SCALE), and two floating-point operations (FLOOR·SQRT). They are all instructions that store a calculated result in a destination address you specify; place a contact to the left of the cell and it calculates only when that condition is true, and place no contact and it calculates on every scan as it is, just like a coil.

The word memory used in these calculations — **D, C, and T — is signed 16-bit**, so its range is **-32768 to 32767** (2-1-1). A calculation that runs past that range has the overflowing digits cut off and comes back with the sign flipped, so when you handle large numbers, use the 32-bit double-word views (`DD0` , `DC0` , `DT0`) or the floating-point area R, or split the calculation into intermediate steps that stay inside the range.

The five basic arithmetic operations (ADD·SUB·MUL·DIV·MOD) and the two increment and decrement operations (INC·DEC) reduce to pure assignments, which makes them seven of the eight instructions that **can also be used in a branching rung** (the eighth is MOV, the transfer instruction, 6-4-1). SCALE (6-3-3) and FLOOR·SQRT (6-3-4), by contrast, are rejected by the compiler in a branching rung even though they sit in the same category (see the warning in the introduction to this chapter).

6-3-1 Basic Arithmetic (ADD·SUB·MUL·DIV·MOD)

Definition

ADD·SUB·MUL·DIV·MOD are five box functions that store the result of a basic arithmetic operation on two values (addition, subtraction, multiplication, division, remainder) in an address you specify. All five instructions have exactly the same number and positions of arguments and differ only in the operator.

Mnemonic	Operator	Meaning
`ADD`	`+`	Addition
`SUB`	`-`	Subtraction
`MUL`	`*`	Multiplication
`DIV`	`/`	Division
`MOD`	`%`	Remainder (modulo)

Usage

```
<function> <S1> <S2> <D>
```

The arguments are always four tokens, and the destination (D) is the last token. `ADD D0 D1 D2` is converted into `D2 = D0 + D1`, and you can also write the destination back to itself, as in `MUL D0 2 D0`. In the S1 and S2 positions you can mix word addresses (`D0`, `C10`, and so on) with integer literals, and the destination D has to be a word address that will store the result.

Example

Place it as `M0 NO contact → ADD D0 D1 D2` and, on every scan in which M0 is ON, the value of D0 plus D1 is stored in D2.

< Figure 6-14 A basic arithmetic ADD example circuit — the ADD D0 D1 D2 box function placed after an M0 NO contact >

For SUB·MUL·DIV·MOD you only change the instruction written in the cell; the wiring and the argument positions (S1 S2 D) are exactly the same. Place `SUB D0 D1 D2` in the same spot and D0−D1 is stored in D2; place `MUL D0 D1 D2` and D0×D1 is stored.

< Figure 6-15 ADD monitoring — the result of D0+D1 being reflected in D2 while M0 is ON >

Notes

> **If the second value (S2) in DIV·MOD is 0, it is not caught at the compile stage.** Check the ranges of the values yourself when you design the ladder so that a division by zero cannot arise at run time.

- The destination (D) is always the last token. SCALE (6-3-3), described later, puts the destination in a different position, so do not confuse the two.
- Because the D area is a 16-bit integer (2-1), DIV is an integer division that discards the fractional part — in `DIV D0 D1 D2`, if D0=7 and D1=2, D2 becomes 3 (not 3.5).

6-3-2 Increment and Decrement (INC·DEC)

Definition

INC·DEC are unary box functions that increase or decrease the value of a single destination by 1. INC works as `D = D + 1` and DEC as `D = D - 1`.

Usage

```
INC <D>
DEC <D>
```

There is only one argument, the destination. `INC D0` increases D0 by 1 on every scan, and `DEC D5` decreases D5 by 1 on every scan.

Example

Place it as `M0 rising edge contact → INC D0` and D0 increases by 1 only on the rising edge where M0 changes from OFF to ON. Because the INC box function is placed after a rising edge contact (5-5-1), D0 increases once at the moment the contact turns on, rather than the whole time the contact is on.

< Figure 6-16 An increment INC example circuit — the INC D0 box function placed after an M0 rising edge contact >

Notes

> **INC·DEC themselves have no edge-detection function.** Place `INC D0` on its own with no contact and D0 keeps increasing on every scan (hundreds to thousands of times per second), so in practice the value shoots up in an instant. The counter CTU (6-2-1) has rising-edge detection built into its helper function,

> so you can connect an ordinary contact to it as it is, but INC·DEC have no such built-in detection, so to make the value "increase only once each time the condition turns on" you must place a rising edge contact (5-5-1) in front of them. A free-form assignment expression (`D0 = D0 + 1`, the example in Rising Edge Contact (5-5-1)) is used together with a rising edge contact for the same reason.

- If the destination index exceeds the number of memory entries configured for the project, compilation is refused.

6-3-3 SCALE — Scale (Linear Interpolation)

Definition

SCALE is a box function that linearly converts (scales) an input value from one range to another. You use it, for example, to change an analog input value into a percentage value in the range 0–100. It is converted into a direct call to the scale function bundled with the ILOGICS core, and the app chooses automatically between the integer variant `Iscale` and the floating-point variant `Iscalef`.

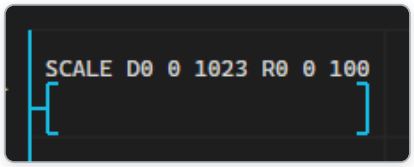
Usage

```
SCALE <IN> <IN_MIN> <IN_MAX> <OUT> <OUT_MIN> <OUT_MAX>
```

There are always six arguments (seven tokens): it treats `IN` as a value in the range `IN_MIN`–`IN_MAX`, converts it proportionally into the range `OUT_MIN`–`OUT_MAX`, and stores the result in `OUT`.

Example

Place it as `SCALE D0 0 1023 R0 0 100` and the value of D0 (assumed to be in the range 0–1023), converted into the range 0–100, is stored in R0. Because no contact is placed in front of it, the conversion happens on every scan as it is. The input range `0–1023` used here is an example based on the analog inputs built into the board — a channel of the MPAINO analog input module (X) has a full scale of **32767**, which is a different range. Check what range the channel you are reading actually produces, and write `IN_MIN` and `IN_MAX` to match it.



< Figure 6-17 A SCALE example circuit — the SCALE D0 0 1023 R0 0 100 box function placed with no contact >

Notes

> **The destination (OUT) is not the last token but the fifth token.** In Basic Arithmetic (6-3-1) earlier and MOV (6-4-1) later, as in the other box functions in this chapter, the destination is always the last token, but SCALE alone is the exception: `OUT` comes in the middle (after `IN_MIN` and `IN_MAX`, before `OUT_MIN` and `OUT_MAX`). If you push the arguments straight through in the usual order, the result is stored at the wrong address, so check the order.

> **All six arguments together decide whether the calculation is done in integers or in floating point.** If **any one** of the six arguments (`IN` , `IN_MIN` , `IN_MAX` , `OUT` , `OUT_MIN` , `OUT_MAX`) is real, the calculation uses the floating-point variant `Iscalef` ; if **all** of them are integers, it uses the integer variant `Iscale` . "Real" here means an address in the R area (or a symbol that points at the R area) and a literal written with a decimal point or an exponent, such as `2.5` or `1e3` ; an integer literal such as `100` does not count as real. In other words, `SCALE D0 0 1023 D1 0 100` is an integer calculation from beginning to end and produces no decimals, and if you want to keep the decimals, either put the destination in R (`SCALE D0 0 1023 R0 0 100`) or write the range arguments as reals, such as `0.0` and `100.0` .

- The result **never leaves the output range (it is clamped)**. Even if `IN` is a value outside `IN_MIN` – `IN_MAX` , the output for the nearest boundary is stored, so a sensor value that overruns the range you expected does not fly outside `OUT_MIN` – `OUT_MAX` .
- If `IN_MIN` and `IN_MAX` are the same, it returns `OUT_MIN` **as it is**, to avoid dividing by zero.
- The ranges **can also be written the other way around.** If `OUT_MIN` is greater than `OUT_MAX` (`SCALE D0 0 1023 R0 100 0` , for example), the conversion becomes inverse — the larger the input, the smaller the output — and writing the input range reversed, `IN_MIN` > `IN_MAX` , is handled the same way.
- If the destination is in the D (integer) area, the decimals are **cut off at the moment the value is stored**, even when the calculation was done in floating point. If you need the decimals, put the destination in the R area (float, 2-1).

6-3-4 Floating-Point Operations (FLOOR·SQRT)

Definition

FLOOR·SQRT are unary box functions that apply a mathematical function to a floating-point value and store the result. FLOOR is a floor operation that discards the fractional part, and SQRT is a square root. Both call `floor()` and `sqrt()` from the C standard library (`math.h`) directly.

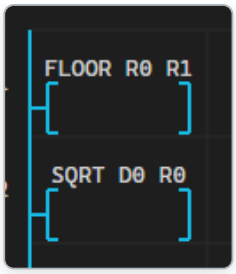
Usage

```
FLOOR <S> <D>
SQRT <S> <D>
```

`FLOOR R0 R1` stores the floor of R0 in R1, and `SQRT D0 R0` stores the square root of D0 in R0.

Example

Place `FLOOR R0 R1` with no contact and the floor of R0 is stored in R1 on every scan; place `SQRT D0 R0` and the square root of D0 is stored in R0. As with the basic arithmetic operations, the destination is the last token, and if you need to, you can place a contact in front so that the calculation happens only under a particular condition.



< Figure 6-18 A floating-point operations example circuit — the FLOOR R0 R1 and SQRT D0 R0 box functions placed with no contact >

Notes

- Because the result is a floating-point value, use the R (float, 2-1) area as the destination. Use the D (integer) area as the destination and the fractional part is truncated.
- Feed a negative number into SQRT and it follows the behavior of `sqrt()` in `math.h` as it is (an undefined result) — filter the input upstream so that it cannot become negative.

6-4 Transfer

Transfer box functions split into two branches: MOV, which moves a value as it is, and GMOV/FMOV (block transfer), which move or fill several words at once. In both branches, if there is a contact to the left of the cell the move happens only while that condition is true; with no contact, the move happens on every scan.

The two branches are treated differently in a branching rung — MOV reduces to a pure assignment, so it is one of the eight instructions that can also be used in a branching rung, whereas the block transfers GMOV and FMOV expand into loops and are **rejected by the compiler** in a branching rung (see the warning in the introduction to this chapter).

6-4-1 MOV — Move

Definition

MOV is a box function that moves one word value to another address as it is. It behaves as `D = S`.

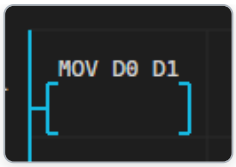
Usage

```
MOV <S> <D>
```

`MOV D0 D1` copies the value of D0 into D1. The source (S) position accepts not only an address but also an integer literal (`MOV 100 D5` → `D5 = 100`), so it is also used to load a constant into word memory.

Example

If you place `MOV D0 D1` alone with no contact, the value of D0 is copied to D1 on every scan.



< Figure 6-19 A MOV example circuit — a MOV D0 D1 box function placed with no contact >

Notes

- The destination (D) is the last token (the same as basic arithmetic (6-3-1); only SCALE (6-3-3) is an exception).
- If the source is a literal, the same value keeps being overwritten on every scan — if you want to initialize only once, consider gating it with a contact (a signal that turns on for a single scan right after boot, for example).

6-4-2 Block Transfer (GMOV·FMOV)

Definition

GMOV and FMOV are box functions that move several words at once. GMOV (Group Move) copies N words in order from the source to the destination as they are, and FMOV (Fill Move) fills all N destination words with the same value from a single source value.

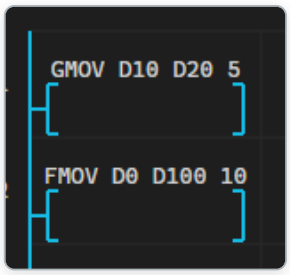
Usage

```
GMOV <S> <D> <N>
FMOV <S> <D> <N>
```

The argument order is source → destination → count. `N` (the count) accepts either an integer literal or a word address.

Example

`GMOV D10 D20 5` copies the five words D10–D14 to D20–D24 as they are. `FMOV D0 D100 10` fills all ten words D100–D109 with the single value of D0. Both instructions run on every scan when they are placed with no contact, and the argument order is source → destination → count.



< Figure 6-20 A block transfer example circuit — GMOV D10 D20 5 and FMOV D0 D100 10 box functions placed with no contact >

Notes

> **Only D/C/T word addresses are allowed for the source and the destination.** Bit areas such as P/M, the R (float) area, and `.bit` or byte/double-word views cannot be used as the source or the destination of a block transfer — use such an address and compilation is rejected. **The same goes for the source of FMOV, so a constant literal cannot be used there** — write `FMOV 0 D100 10`, with the value to fill written straight in as a number, and it is rejected; load the constant into a single word first with `MOV 0 D0` and then spread that word with `FMOV D0 D100 10`, in two steps.

- **If you write the count (N) as an integer literal, the end address of the block is checked too.** For `GMOV D10 D20 5` it checks that the source `D10`–`D14` and the destination `D20`–`D24` are all inside the configured memory count (2-1), and if even one of them runs past it, compilation is rejected. You do not have to work it out for yourself.
- **If you write the count (N) as an address**, however (`GMOV D10 D20 D7`), the value is decided at run time and the end address cannot be known in advance, so only the start address is checked and the end address is not. In that case, check for yourself that the largest value N can take, plus the start address, does not exceed the memory count.

6-5 Bitwise

Bitwise box functions come to eight in total: four logic operations (WAND·WOR·WXOR·WNOT), two shifts (SHL·SHR), and two rotates (ROL·ROR). Like arithmetic (6-3), bitwise logic and shift turn the instruction written in the cell into an assignment expression of the form `destination = expression` and reuse the existing, proven expression conversion path; rotate, however, has no circular shift operator in C, so the app handles it directly with a dedicated inline formula. In all three branches, placing a contact to the left of the cell wraps the code in `if (condition) { ... }`; with no contact, the code runs unconditionally on every scan, just as a coil does.

The eight bitwise instructions do reduce to assignment expressions, but they are not on the list of eight instructions that can be used in a branching rung, so **compilation is rejected in a position where one contact feeds several box functions** (see the warning in the introduction to this chapter). In such a position you can write the same calculation as a free-form (6-9) assignment expression (`D2 = D0 & D1`).

6-5-1 Bitwise Logic (WAND·WOR·WXOR·WNOT)

Definition

WAND, WOR, and WXOR are box functions that store, at a specified address, the result of a bitwise logic operation (AND, OR, XOR) on two values. WNOT is a unary box function that stores the bitwise inversion (NOT) of a single value. Their names resemble the AND/OR combination of contacts, but they are a different concept, so the `W` prefix — meaning a word-level operation — distinguishes them.

Mnemonic	Operator	Arguments	Meaning
`WAND`	`&`	binary	Bitwise AND
`WOR`	`|`	binary	Bitwise OR
`WXOR`	`^`	binary	Bitwise XOR
`WNOT`	`~`	unary	Bitwise inversion (NOT)

Usage

```
<function> <S1> <S2> <D>
```

WAND, WOR, and WXOR use the same argument positions as basic arithmetic (6-3-1) — the destination (D) is always the last token. WNOT takes only one source, so its form is different.

```
WNOT <S> <D>
```

`WAND D0 D1 D2` converts to `D2 = D0 & D1`, and `WNOT D0 D1` converts to `D1 = ~D0`.

Example

If you place `M0 NO contact → WAND D0 D1 D2`, the bitwise AND of D0 and D1 is stored in D2 on every scan in which M0 is ON.

< Figure 6-21 A bitwise logic WAND example circuit — a WAND D0 D1 D2 box function placed after an M0 NO contact >

For WOR and WXOR you only change the instruction written in the cell; the wiring and the argument positions (S1 S2 D) are exactly the same — place `WOR D0 D1 D2` in the same position and the bitwise OR of D0 and D1 is stored in D2, and place `WXOR D0 D1 D2` and the bitwise XOR is stored there. WNOT takes only one source, so if you place it as `WNOT D0 D1`, the bitwise inversion of D0 is stored in D1.

Notes

> **WNOT and INV (6-6-1), covered later, behave and produce results exactly the same.** Both store the bitwise inversion (NOT) of the source value in the destination, and only the palette category (Bitwise/Convert) differs, so it does not matter which one you choose.

- The destination (D) is always the last token for WAND, WOR, and WXOR; for WNOT it is the second (and last) token, right after the source (S).

6-5-2 Shift (SHL·SHR)

Definition

SHL and SHR are box functions that store the result of shifting a value left or right by a specified number of bits. SHL fills the lower bits with 0 as far as it pushes to the left, and SHR fills the upper bits with 0 as far as it pushes to the right.

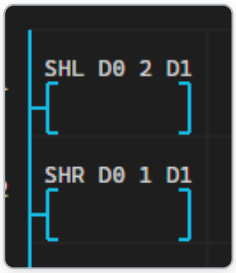
Usage

```
SHL <S> <n> <D>
SHR <S> <n> <D>
```

The arguments are in the order source (S) → shift bit count (n) → destination (D). `n` accepts either an integer literal or a word address.

Example

If you place `SHL D0 2 D1` and `SHR D0 1 D1` on two rungs with no contact, then on every scan SHL stores the value of D0 shifted 2 bits to the left and SHR stores the value of D0 shifted 1 bit to the right, each into D1.



< Figure 6-22 A shift example circuit — SHL D0 2 D1 and SHR D0 1 D1 box functions placed with no contact >

Notes

SHR is a logical shift (zero-fill) only when the source is a word in the D area. D, C, and T are all signed 16-bit values (`int16_t`), so shifting one right as it is would be an arithmetic shift, which spreads the sign bit. The app casts the source to an unsigned value, filling the empty upper positions with 0, **only when the source is a word address in the D area** (or a symbol that points at the D area). So `SHR D0 1 D1` is a logical shift, but `SHR C0 1 D1` **and** `SHR T0 1 D1` **are arithmetic shifts**, and the upper positions are filled with 1 when the source is negative. To shift a C or T word logically, apply the cast yourself in a free-form box function (6-9), as in `D1 = UW(C0) >> 1` .

SHL fills the empty lower positions with 0 whatever area the source is in, so this distinction does not apply to it. If the source is a double-word view (`DD0`) or a `.bit` reference (`D0.3`), the value itself cannot be negative, so the upper positions are filled with 0 even with SHR.

- If the shift bit count is 16 or more, the whole word width (16 bits) is pushed out and the value can become 0 — if you need a 16-bit word rotation, use the rotate instructions (ROL·ROR) in the next section.

6-5-3 Rotate (ROL·ROR)

Definition

ROL and ROR are box functions that store the result of rotating a value left or right within a 16-bit word. Unlike a shift, the bits that are pushed out are not lost but re-enter from the opposite end — for ROL the upper bits pushed out to the left re-enter as the lowest bit, and for ROR the lower bits pushed out to the right re-enter as the highest bit.

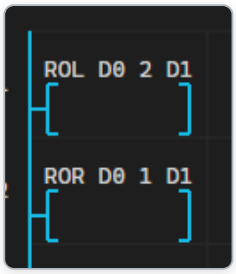
Usage

```
ROL <S> <n> <D>
ROR <S> <n> <D>
```

The argument order is the same as for SHL and SHR (source → rotate bit count → destination). C has no circular shift operator, so instead of converting the instruction into an assignment expression the way shift does, the app generates a dedicated inline masking formula directly.

Example

If you place `ROL D0 2 D1` and `ROR D0 1 D1` on two rungs with no contact, then on every scan ROL stores the value of D0 rotated 2 bits to the left and ROR stores the value of D0 rotated 1 bit to the right, each into D1. Unlike a shift, the bits that are pushed out are not lost but re-enter from the opposite end.



< Figure 6-23 A rotate example circuit — ROL D0 2 D1 and ROR D0 1 D1 box functions placed with no contact >

Notes

> **A rotate always circulates in units of a 16-bit word.** The rotate bit count (n) is masked internally with `n & 15` and is therefore fixed to the range 0–15 (even if you enter 16 or more, only the remainder applies, which blocks undefined shift behavior) — `ROL D0 16 D1` and `ROL D0 0 D1` produce the same result (no rotation).

- The destination (D) is always the last token (the same as for shift and basic arithmetic).

6-6 Convert

Convert box functions come to four in total: two inversions (INV·NEG) and two BCD conversions (H2D·D2H). INV converts to exactly the same assignment expression as WNOT in Bitwise (6-5-1), and NEG converts to a unary-minus assignment expression. Like rotate (6-5-3), H2D and D2H have no matching operator in C, so they are handled with a dedicated inline formula. All four instructions are wrapped in a condition when there is a contact to the left of the cell, and run unconditionally on every scan when there is not.

The four convert instructions are **rejected by the compiler** in a branching rung as well (see the warning in the introduction to this chapter). In a position where one contact feeds several box functions, use a free-form (6-9) assignment expression instead (`D1 = ~D0`, `D1 = -D0`).

6-6-1 Invert and Negate (INV·NEG)

Definition

INV is a unary box function that stores the result of inverting a value bit by bit (NOT), and NEG is a unary box function that stores the two's complement (sign inversion) of a value.

Mnemonic	Expression	Meaning
`INV`	`~S`	Bitwise inversion (NOT)
`NEG`	`-S`	Two's complement (sign inversion)

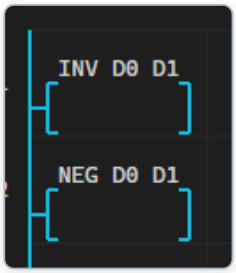
Usage

```
INV <S> <D>
NEG <S> <D>
```

There are only two arguments: source (S) → destination (D). `INV D0 D1` converts to `D1 = ~D0`, and `NEG D0 D1` converts to `D1 = -D0`.

Example

If you place `INV D0 D1` and `NEG D0 D1` on two rungs with no contact, then on every scan INV stores the bitwise inversion of D0 and NEG stores the sign inversion (two's complement) of D0, each into D1.



< Figure 6-24 An invert example circuit — INV D0 D1 and NEG D0 D1 box functions placed with no contact >

Notes

> **INV behaves and produces results exactly the same as WNOT in Bitwise (6-5-1).** Both store the bitwise inversion (NOT) of the source value in the destination, and only the palette category (Bitwise/Convert) differs, so it does not matter which instruction you choose.

- The destination (D) is always the last token.

6-6-2 BCD Conversion (H2D·D2H)

Definition

H2D and D2H are box functions that convert back and forth between a hexadecimal representation (hex) and BCD (Binary-Coded Decimal, a representation that holds one decimal digit in each 4-bit nibble). H2D is a BCD decode (`0x1234` → 1234, for example), and D2H is the reverse, a BCD encode (1234 → `0x1234`).

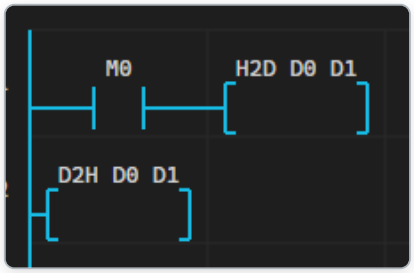
Usage

```
H2D <S> <D>
D2H <S> <D>
```

There are only two arguments: source (S) → destination (D). The D area is 16 bits (four nibbles), so it handles up to four BCD digits at a time.

Example

If you place `M0 NO contact → H2D D0 D1`, the result of weighting each nibble of D0 by its decimal place value and summing them is stored in D1 on every scan in which M0 is ON. If you place `D2H D0 D1` with no contact, the conversion runs in the opposite direction (decimal → BCD).



< Figure 6-25 A BCD conversion example circuit — H2D D0 D1 after an M0 NO contact, and D2H D0 D1 placed with no contact >

Notes

> **A value outside the four-digit BCD range is not converted as intended.** H2D weights and sums each nibble (0–F) of the source directly as a decimal place, so if a nibble value exceeds 9 (the `A` of `0x1A00`, for example) the value is pure hexadecimal rather than BCD and the result goes wrong. If the source of D2H falls outside the range 0–9999, the carry is truncated by the `% 10` operation — for both instructions, check in advance that the input value is valid BCD (H2D) or within 0–9999 (D2H).

- The destination (D) is always the last token.

6-7 Bit/Latch

Bit/latch box functions are the two instructions SET and RST, and all they do is turn a single destination bit on or off, which makes them the simplest box functions of all. Rather than computing or converting a value the way the other categories do, they behave exactly the same as the SET coil and the RESET coil (5-3-2, 5-3-3) in the contact types chapter. This category is therefore closer to a cell coil than any other box function — in effect, these instructions are "a coil placed in the box function position."

They behave like coils, but **they are treated differently in a branching rung.** In a circuit where one contact splits with a vertical connecting line and feeds several outputs, the SET coil and the RESET coil can be used as they are, but the SET and RST box functions are **rejected by the compiler** (see the warning in the introduction to this chapter). If you need a latch in a branch, use the cell coils (5-3-2, 5-3-3) instead of the box functions.

6-7-1 SET·RST

Definition

SET and RST are box functions that write a single destination bit as ON and as OFF respectively. SET writes the destination as ON at the moment the condition turns ON, and holds that value even after the condition returns to OFF (latch). RST writes the destination as OFF (releases it) at the moment the condition turns ON, and likewise holds the value even after the condition turns off. Their behavior is exactly the same as that of the SET coil and the RESET coil (5-3-2, 5-3-3) in the contact types chapter — the SET/RST box functions amount to moving the SET coil and RESET coil cells into the box function position, so there is no difference in how the destination is turned on, turned off, and held (latched).

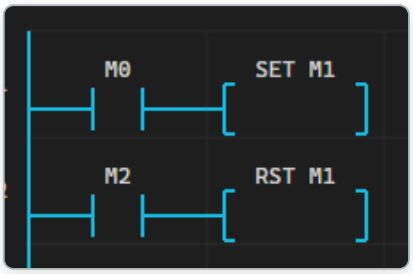
Usage

```
SET <target>
RST <target>
```

There is only one argument, the destination — there is no preset, so the syntax differs from the three- or four-token syntax of timers and counters. As with a coil, you write a bit address such as P or M in the destination position.

Example

If you place the two rungs `M0 NO contact → SET M1` and `M2 NO contact → RST M1`, M1 is written ON at the moment M0 turns ON and stays on even after M0 returns to OFF, until M1 is released to OFF at the moment M2 turns ON.



< Figure 6-26 A SET·RST example circuit — SET M1 after an M0 NO contact, and RST M1 after an M2 NO contact >

This behavior is exactly the same as that of the SET coil and RESET coil cells seen in the contact types chapter (5-3-2/5-3-3); only the destination address differs. This example targets the internal relay M1, so it writes only an internal bit, but if you target a registered output pin (P32, for example), the actual physical pin output is updated as well, just as in the SET coil example (5-3-2, Figure 5-9) — that is only a difference between a pin and an internal relay as the destination, not a difference in the behavior of SET/RST itself.

Notes

> **Even when the condition returns to OFF, SET does not turn the destination off.** To turn the value off you must place a separate RST that targets the same address — exactly the same caution as for the SET coil (5-3-2).

- If you specify a timer (T) or counter (C) address as the destination of RST, it acts with the special meaning of resetting the elapsed value (or the count value) together with the done bit, rather than releasing an

ordinary bit — the same special meaning of the RESET coil covered under timers (6-1) and counters (6-2) (RST goes through the same emit path as a coil, so it applies identically).
- SET and RST take only one argument, so if you append a preset the way you do for timers and counters (`SET M1 5`, for example), the argument count does not match and compilation is rejected.

6-8 Logic Control

Logic control box functions come to four in total: the conditional jumps JMP and JMPN, and the master controls MCS and MCSCLR. Whereas every instruction so far updates a single value inside its own rung and finishes there, this category is the only group of box functions that changes the execution flow of the whole ladder — which rungs to skip in this scan, which section to gate as a whole.

Because they change the execution flow, they **cannot be used in a branching rung** — place JMP, JMPN, MCS, or MCSCLR where one contact feeds several box functions and compilation is rejected (see the warning in the introduction to this chapter). Let these four instructions take a whole rung to themselves.

6-8-1 Conditional Jump (JMP·JMPN)

Definition

JMP and JMPN are box functions that jump to a specified label position depending on the condition to the left of the rung. JMP jumps when the condition is ON, and JMPN inverts the condition and jumps when the condition is OFF. When a jump occurs, the rungs between the jump rung and the label rung are skipped as a whole in this scan and are not executed — the values that the coils and box functions of those rungs were updating are not updated and simply keep the value of the previous scan (hold).

A label is a dedicated cell that marks the arrival point of a jump. Place a single Func cell with no contact to its left and write only the jump target name (`SKIP`, for example) in the code field, and that rung is recognized as a label — but only if at least one JMP or JMPN that refers to the same name exists within that ladder; a bare identifier cell with no reference is simply ignored.

Usage

```
JMP <label>
JMPN <label>
```

A label name is an identifier that starts with a letter or an underscore and is followed only by alphanumeric characters and underscores (`^[A-Za-z_][A-Za-z0-9_]*$`) — it is used as a C label as it is, so the rules are the same as for C identifiers.

Example

If you place `P0 NO contact → JMP SKIP`, then the rung `ADD D0 1 D0` (the target to skip), and the label `SKIP` on the last rung, the ADD rung is skipped in scans in which P0 is ON, so D0 is not updated and keeps its previous value.

< Figure 6-27 A conditional jump example circuit — JMP SKIP after a P0 NO contact, the ADD D0 1 D0 to be skipped, and the destination label SKIP >

JMPN inverts the condition, so it jumps to SKIP in scans in which P0 is OFF instead.

Notes

> **If the target label of a jump is not defined, or if the same label name is defined more than once, compilation is rejected.** The app does not guess at the jump target and move on; it allows only an unambiguous label pair.

- Combinations in which a jump or a label crosses the boundary of a master control section (MCS–MCSCLR, 6-8-2) have not been verified yet, so compilation is rejected — place JMP/JMPN and MCS sections so that they do not overlap.

6-8-2 Master Control (MCS·MCSCLR)

Definition

MCS and MCSCLR are a pair of box functions that gate every coil and box function in the section between them with a single condition. MCS marks the start of the section and MCSCLR marks its end, and if the condition of the rung on which MCS is placed is OFF, every coil inside the section is forced OFF regardless of its own condition. You can overlap several MCS instructions (nesting) to gate a narrower section again, and this nesting priority is managed with a level number (0–15).

Usage

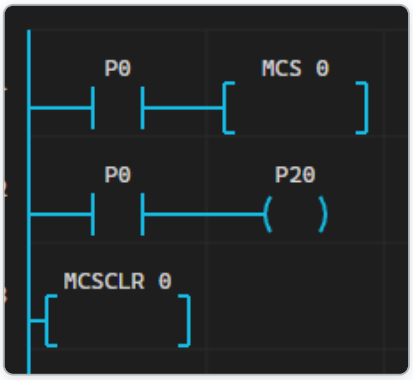
```
MCS <level>
MCSCLR <level>
```

The level is an integer in the range 0–15. A section opened with `MCS <k>` is closed by an `MCSCLR` of level k or below (the same number or a lower one) — `MCSCLR <n>` closes at once every open section of level n or above (including sections nested further inside). When you nest, the level of the inner MCS must be larger than that of the outer one — reopen an MCS with a level equal to or lower than an already open section and compilation is rejected.

Example

If you place `P0 NO contact → MCS 0`, then the rung `P0 NO contact → P20 coil`, and `MCSCLR 0` on the last rung, the P20 coil inside the section works according to its own condition only while P0 is ON, and P20 is

forced OFF when P0 is OFF.



< Figure 6-28 A master control example circuit — MCS 0 after a P0 NO contact, the P20 coil being gated, and MCSCLR 0 closing the section >

When you nest MCS instructions, the inner section combines its own condition again on top of a state in which the outer gate is already applied — the coils of the inner section do not have to account for the outer condition separately, and when the outer section closes, the inner one closes with it.

Notes

> **The levels of MCS and MCSCLR must be paired.** If you leave an MCSCLR without a matching MCS, or overlap the level numbers out of order, compilation is rejected. Keep a nesting structure that closes in the reverse of the opening order (innermost first).

- If the level falls outside the range 0–15 (negative, or 16 or more), compilation is rejected.
- If a timer or a counter is inside an MCS section, then while the section is closed (gate OFF) it is treated exactly as if the input of that rung were off — the result follows each type's usual "input OFF" behavior. Types such as TON·TOFF·TPL·TMON, which reset or change state naturally on input OFF alone, do the same when the gate is closed. A retentive type such as TMR, on the other hand, preserves the elapsed value on input OFF (see **TMR — Retentive Timer**(6-1-5)), so even when the gate closes the elapsed value is not reset and merely stops where it is. Counters (CTU·CTD·CTUD) are also treated as input OFF, so no new rising edge is detected and counting stops, while the count value itself is not reset.

6-9 Free-Form Box Functions

Every instruction covered so far has been a predefined instruction that you choose from the palette. A free-form box function bypasses the palette and writes an expression close to C directly into the input field of the edit window, and it splits broadly into two branches. A free-form box is not divided into slots — the band carries `code`, and the expression you wrote appears below it as a single line, as it is.

The free-form kind is also the only one that **can be used in a branching rung with no restriction at all**. Where most of the predefined instructions are rejected in a circuit in which one contact splits with a vertical connecting line and feeds several box functions (see the warning in the introduction to this chapter), rewriting the same calculation as an assignment expression passes as it is.

An assignment expression takes the form `destination = expression`; the simplest form is `D0 = D0 + 1`, already used in the rising edge contact (5-5-1) example of the contact types chapter. On the left side (the

destination) you can write not only a word address (`D0`, `R0`, and so on) but also a single bit inside a word (`D0.5`) or a width-view bit of the P or M area (see bit contacts in word memory (5-8)), and on the right side you can write an expression that combines arithmetic, comparison, and so on however you like. Mixing the casts and the arithmetic covered below, as in `R0 = REAL(D0) / 10.0`, is also a form of assignment expression. You can write a registered symbol name (9-5) instead of an address.

> **When you write a width-view bit of the P or M area (`WP0.3`, for example) on the left side, turn that pin on first.** If you do not tick **Use** for that pin in "Board Pin Map Editing" under **Pin Editing**(2-4), a bit written through a width view is **silently treated as 0**, with neither an error nor a warning. Write the same pin as a single address such as `P0` and compilation catches it as a "not in use" error, but a width view does not go through that check, so the circuit quietly fails to work. Before writing a width-view bit, **always check that the pin is turned on**.

< Figure 6-29 A free-form assignment expression example circuit — a D0 = D0 + 1 assignment expression placed after an M0 rising edge contact >

An arbitrary function call is a form in which you write only a function call, with no assignment (`setMotor(100)`, `doStuff()`). You use it when you do not need to store a return value and need only the side effect (driving a motor, updating a state, and so on). You can of course also store a return value by writing a function call on the right side of an assignment expression, as in `D0 = readSensor()`. It makes no difference whether the function you call is an Arduino built-in function or a user function defined in another .ino tab of the same project.

< Figure 6-30 A free-form function call example circuit — a setMotor(100) call placed after an M0 NO contact >

> **The app does not check whether a function name actually exists or whether the argument count matches.** That validation is delegated as it is to the arduino-cli compile step — use a function that does not exist and the app generates the code as written, and the compiler raises an error when you build.

Lastly, there are five cast-like functions that change the width and the sign of a value on the spot. All of them are unary, taking a single argument, and cannot be used as the left side (lvalue) of an assignment.

Function	Matching C cast	Meaning
`REAL`	`(float)`	Integer → float
`DW`	`(int32_t)`	Signed 32-bit integer
`WD`	`(int16_t)`	Signed 16-bit integer

Function	Matching C cast	Meaning
`UD`	`(uint32_t)`	Unsigned 32-bit integer
`UW`	`(uint16_t)`	Unsigned 16-bit integer

For example, `R0 = REAL(D0) / 10.0` casts D0 (an integer) to a float and then divides it by 10.0, storing in R0 a result that preserves the fractional part. Casts can also be nested, as in `REAL(WD(D0))`, so you can correct the sign or the width first and then convert to a float.

> **A "bare expression" that is neither an assignment nor a function call is rejected.** If you write a calculation such as `D0 + 1` into a cell without using the result anywhere, it is regarded as code with no effect and becomes an error at the compile step. Once you have calculated a value you must either assign it to a destination (`D1 = D0 + 1`) or write it as a function call that has a side effect.

- Cast names are recognized only in uppercase — write one in lowercase (`real(D0)`) and it is treated not as a cast but as an ordinary function call of that name, which becomes an error at the build step if no such function actually exists.
- If you place an expression that updates itself with no rising edge contact (`D0 = D0 + 1`), it keeps running on every scan while the condition is on — to increment it only once you must put a rising edge contact (5-5-1) in front, just as with increment and decrement (6-3-2) (for the same reason as in the rising edge contact (5-5-1) example above).

7 Code Editing and Autocomplete

MPINO Studio 2 lets you edit Arduino C code directly alongside the ladder logic. This chapter covers the basics of the code editor, working with code tabs (`.ino`), autocomplete and Go to Definition (F12), the find and replace that changes depending on what you are editing, and how to read the bottom panel. One small example — writing a `setMotor()` helper function in a code tab, checking it with autocomplete and F12, and then calling it from the ladder box function `setMotor(100)` — runs on through several sections and shows how code and ladder fit together.

7-1 The Code Editor

The code editor in MPINO Studio 2 is a Monaco-based C/C++ editor. When you create a new project, a single code tab named `main` opens by default, and the name of this tab always changes together with the project file name. The font is fixed-width Cascadia Code at 14 px, with line numbers on the left and a minimap on the right. Code folding arrows are always visible in the gutter next to the line numbers, so you can collapse and expand functions or blocks. The Tab key inserts four spaces, and the line the cursor sits on is highlighted faintly.

How it connects to the ladder

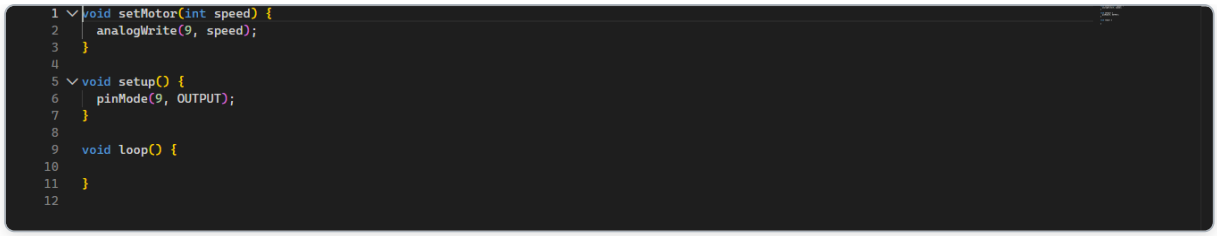
In a project that uses a ladder, automatically generated `#include` lines (`mpino_runtime.h`, `ladder_generated.h`, and `ino_shared.h`, plus `symbols_generated.h` when it is needed) go in at the top of the code. The editor hides this block and numbers the lines from 1 again — it is only a display convenience; the lines really are in the file, and they are reflected unchanged when you save and build. A project that does not use a ladder (a pure Arduino project) has no such block in the first place, so there is nothing to hide.

A `setup()` and `loop()` skeleton is generated along with it, and the `ladderLoop()` call inside `loop()` is the entry point that runs the ladder logic on every scan. If you delete this call, the ladder no longer runs — the transpiler does not rewrite code that you wrote, so keeping this call is your responsibility (for how a ladder becomes code, see **How a Ladder Becomes Arduino C Code**(1-6)). The menu that puts the `ladderLoop()` call in and takes it out is not on the code tab but on the ladder tab, and **Working with Code Tabs**(7-2) covers it.

The context menu

Right-click in the code editor and **Upload** is at the top of the context menu, so you can run build and upload in one go straight away. For the detailed procedure of building and uploading itself, see **Selecting a Port and Uploading**(1-8) and **Run**(3-4).

> Code folding, the minimap, and hiding the generated `#include` lines are all display conveniences only, and they do not affect what you save or build. The hidden `#include` lines really are in the `.ino` file.



< Figure 7-1 The code editor — writing the setMotor function, with line numbers and folding arrows on the left and the minimap on the right >

7-2 Working with Code Tabs

The Code group

The `.ino` code files of a project are listed as tabs in the **Code** group of the **Explorer** in the sidebar on the left. Ladder circuits are gathered below that, in a separate **Ladder** group. Click a row to switch to that tab, add a new tab with the **+** button on the right of the group header, and hover over a row to reveal the ✕ button on the right that deletes that tab.

> Code and ladder tabs are managed in the **Explorer** tree in the sidebar on the left, not above the editor area. The editor itself has no separate tab bar — adding, switching, renaming, deleting, reordering, and the context menu all happen in the Explorer tree described above, and this applies in the same way to both code and ladder.

Adding, renaming, and deleting

Action	How
Add	The **+** button on the group header, or right-click the row › **Add**. Enter a name and a new tab is created and activated right away
Rename	Double-click the row, select the row and press F2 or Enter, or right-click the row › **Rename**
Delete	The ✕ button that appears when you hover over a row, select the row and press Delete, or right-click the row › **Delete**. All three go through a confirmation dialog before deleting

Reordering

To change the tab order within the same group, drag and drop a row to move it, or select a row and move it up and down one step at a time with Ctrl+ ↑ / Ctrl+ ↓. Both methods work.

The main tab is locked

`main`, the first tab in the code group, is specially locked and cannot be renamed, deleted, or moved. That is because the name of the `main` tab always changes automatically together with the project file name (save under a different name and it follows the file name). So right-click the `main` row and the menu has only **Add**.

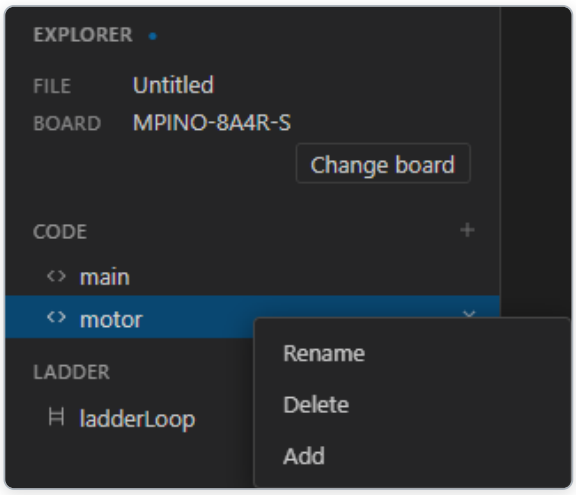
Adding and removing loop()

The menu that puts the `ladderLoop()` call inside `loop()` into the code and takes it out is in the context menu of a row in the **Ladder** group — **Add loop()** and **Remove loop()**. Rows in the code group do not have this menu (this follows on from **The Code Editor**(7-1), where keeping the `ladderLoop()` call is your

responsibility). Rename a ladder tab and the name of the call inside `loop()` is updated automatically along with it.

Several .ino files are one sketch

Even with several code tabs, they are merged into a single sketch when you build. A function defined in one tab can be called from another code tab, and from a ladder box function as well (for how to call a code function from a ladder, see **Free-Form Box Functions**(6-9); for how several tabs are merged and built, see **Building**(1-7)). A new `.ino` tab that you add besides `main` is pre-filled with a single `#include "ino_shared.h"` line at the top — the line that ties the tabs together so they share each other's functions.



< Figure 7-2 The Code group in the Explorer — main and the motor tab that was added, with the context menu of a code row >

7-3 Autocomplete

How it works

A completion list comes up automatically while you type code. When the list is closed, you can open it yourself with Ctrl+Space. Pick a candidate with the up and down arrow keys and press Enter or Tab, and the selected candidate is inserted.

Where the candidates come from

The completion list mixes candidates from four sources.

Source	Example
Words you have just used in the document	The name of a function or variable you defined a moment ago
Arduino built-in functions and constants	`pinMode`, `digitalWrite`, `analogWrite`, `Serial.*`, `HIGH`, `LOW`, and so on
Local functions in the open `.ino` tabs	Functions you defined yourself in a code tab
Symbols of recognized libraries	Functions and constants provided by user libraries and the board core

The one-line description on a candidate

Candidates that are Arduino built-in functions and constants (the second source) come with a **one-line description** beside them — pick `analogWrite`, for example, and "Write a PWM output (0–255)." is shown, `pinMode` carries "Configure pin mode as INPUT/OUTPUT/INPUT_PULLUP.", and `INPUT_PULLUP` carries "Input mode with internal pull-up." These descriptions follow the display language of the app, so change the language and the descriptions change with it (for switching the language, see **Preferences**(3-1-1)). Local functions you defined yourself and library symbols show signature information such as a parameter list instead of a one-line description.

Delay before a library is recognized

Right after you add a new library or refresh, it can take a moment before that library's symbols appear in the completion list (that is the time it takes to prepare code analysis). Type again a little later and it is reflected. For choosing the library folder and refreshing, see **Preferences**(3-1-1).

The running example

Let us write one function in a code tab, as below.

```
void setMotor(int speed) {
  analogWrite(9, speed);
}
```

While you write these lines, `analogWrite` is completed as an Arduino built-in function (the second source). And once you have defined `setMotor` this way, typing only as far as `setM` somewhere else afterwards brings the `setMotor` you just defined up in the completion list as a local function (the third source).

< Figure 7-3 Autocomplete — typing setM shows the setMotor function you have just defined as a candidate >

7-4 Go to Definition

Viewing a definition with F12

Put the cursor on a function or variable name in the code editor and press F12, and you jump straight to where its definition is. It works the same way for a function you defined yourself, such as the `setMotor` above, and for an Arduino built-in function such as `analogWrite`.

Three ways to open a definition

The mouse reaches the same place as F12.

Action	Result
F12	Jumps to the definition of the name the cursor is on
Ctrl+click	Jumps to the definition of the name you clicked. The result is the same as F12

Action	Result
Hold Ctrl and hover the pointer over a name	Does not jump; it shows the definition right there in a **preview popup**. Release Ctrl or move the pointer away and it disappears

While you hold Ctrl down the name is underlined, so you can see with your eyes which name is the jump target at that moment.

Three kinds of jump

The way it jumps splits three ways depending on where the definition is.

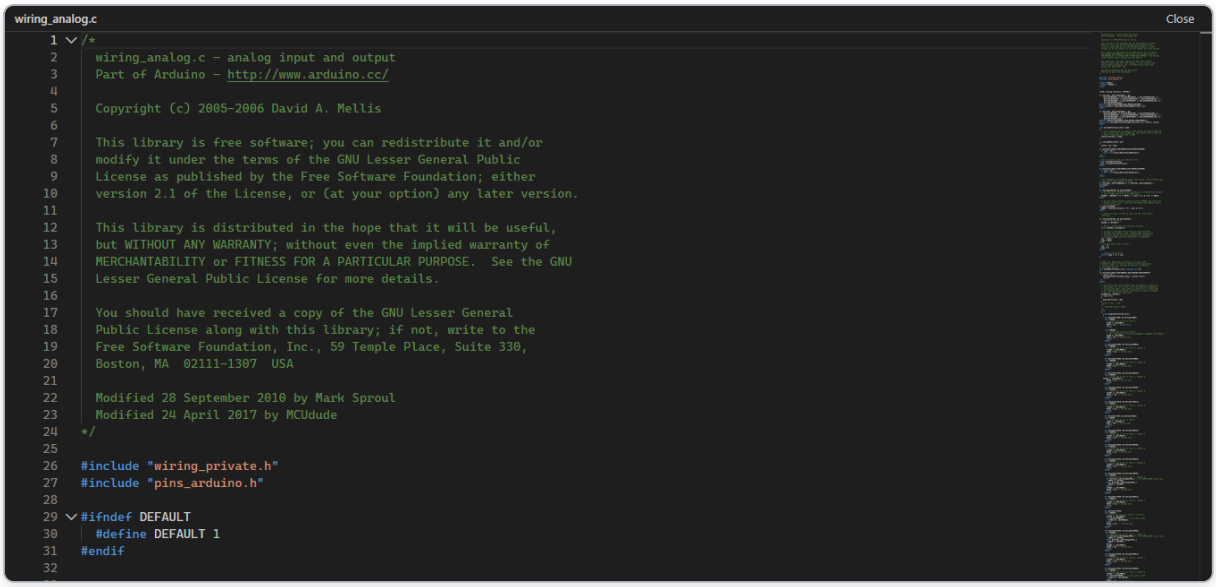
Where the definition is	How it jumps
In the same `.ino` tab	The cursor moves to that line
In another `.ino` tab	It switches to that tab and moves to that line
A built-in or library definition	A read-only viewer opens and shows the original. You close it with Esc or the close button, and you cannot change the contents in this viewer

> Besides Go to Definition (F12), the editor's four commands Go to Declaration, Go to Type Definition, Go to Implementation, and Find All References do not work yet. This is not a bug but something turned off on purpose — the code analysis server can become unstable if it gets back a definition location outside the range of the code you are currently editing, and these four commands alone are disabled to avoid that instability. Go to Definition (F12) is free of this problem and works normally.

> What F12 does changes depending on where the cursor is. With the focus in the code editor, F12 is the Go to Definition just described; with the focus on the ladder grid, F12 is the tool that places a falling edge contact (see the table in **Keyboard Shortcuts**(2-3)). The two functions do not conflict with each other; they are chosen to match whatever you are editing at that moment.

The running example

Put the cursor on `analogWrite` in the code you wrote earlier and press F12, and the Arduino built-in definition opens in a read-only viewer. This viewer is for checking the original, so you cannot change the contents, and you leave it with Esc or the close button.



< Figure 7-4 Go to Definition — press F12 on analogWrite and the built-in definition (the Arduino core source) opens in a read-only viewer >

7-5 Find and Replace

Ctrl+F changes with what you are editing

The find and replace you open with Ctrl+F brings up a different widget depending on whether what you are editing right now is code or a ladder. Press it in a code tab and the standard widget that finds code text opens; press it in a ladder tab and the ladder-specific widget that finds circuits opens (for the find shortcut itself, see **Keyboard Shortcuts**(2-3)).

Code tabs — finding text

In a code tab the familiar standard find widget comes up. You enter the text to find and the text to replace it with, you can turn on the match case, whole word, and regular expression options, and you move between matches with the previous and next arrows.

< Figure 7-5 Find and replace in a code tab — the standard widget for text >

Ladder tabs — finding in the circuit

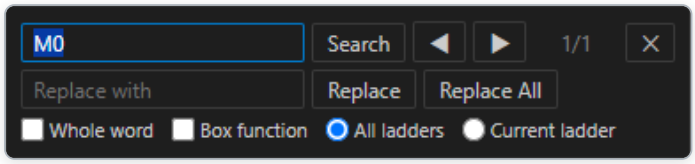
In a ladder tab the ladder-specific widget comes up. It searches contact addresses or the code inside box functions; you run the search with the **[Search]** button and move between matching cells with the previous (◀) and next (▶) arrows. The widget shows the current position together with the total number of matches, and **Replace** and **Replace All** let you replace what you found. The options are as follows.

Option	Description
Whole word	Finds only when the search term matches one complete word
Box function	Searches the code inside box functions. Turn this option on and it becomes find-only, so **Replace** and **Whole word** are disabled along with it

Option	Description
All ladders / **Current ladder**	Choose whether to widen the search scope to every ladder or to limit it to the one ladder you are looking at

The content of the cell the cursor was on when you open the widget is pre-filled in the search box. Unlike find in a code tab, however, **the search does not run yet while you are typing** — you have to press **Enter** or the **[Search]** button, and only then is the circuit swept and the match count filled in. Press Enter again without changing the search term and you move to the next match; **Shift+Enter** goes back to the previous one (the same behavior as the ▶ and ◀ arrows). When nothing matches at all, **No results** is shown where the count would be.

Change an option (whole word, box function, or the search scope) or edit the ladder, and the results are recalculated automatically without your having to press [Search] again. **Esc** closes the widget.



< Figure 7-6 The find and replace widget in a ladder tab — the whole word, box function, and scope options >

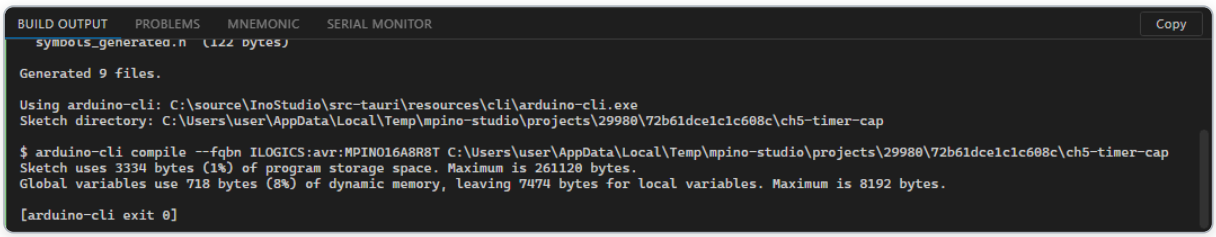
7-6 Reading the Bottom Panel

Opening and closing the panel

You open and close the bottom panel below the work area with Ctrl+J. The panel has four tabs — **Build Output**, **Problems**, **Mnemonic**, and **Serial Monitor** — and this chapter looks at the first three in detail.

Build Output

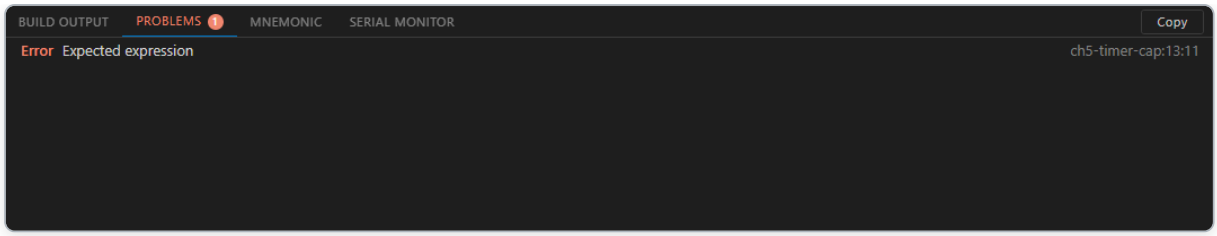
Logs flow through the **Build Output** tab while compiling and uploading are in progress. Whether it succeeded or failed, and where it stopped if it failed, appear in this log. How you run a build or an upload is covered in **Building**(1-7) and **Run**(3-4), so here you only need to know that this is the tab where you read the resulting log.



< Figure 7-7 The Build Output tab — a successful compile log >

Problems

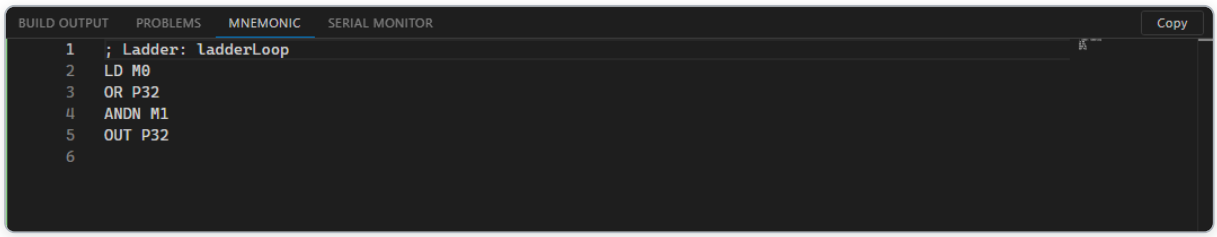
Diagnostics such as errors and warnings are gathered as a list in the **Problems** tab. When there are diagnostics, a count badge appears next to the tab name, with errors in red and warnings in yellow. Click an item in the list and you jump straight to the location that diagnostic points at — to that line of the code tab for a code diagnostic, and to that cell of the ladder for a ladder diagnostic.



< Figure 7-8 The Problems tab — the diagnostics list and the count badge on the tab >

Mnemonic

The **Mnemonic** tab shows the IL (intermediate language) the ladder was converted into, read-only. It is a helper window for checking what order of instructions the ladder circuit actually turns into, and it helps you understand how a ladder becomes Arduino code (see **How a Ladder Becomes Arduino C Code**(1-6)). The IL is filled in only when the build succeeds — when you have not built yet, or the previous build failed, a message is shown instead, and at the moment of failure any IL that was left from before is cleared as well.



< Figure 7-9 The Mnemonic tab — the IL the ladder was converted into, shown read-only >

Serial Monitor

The fourth tab, **Serial Monitor**, is the window for reading the values the board sends. How to connect to a board and check the received values is covered in **Verifying Operation: Serial Monitor and Live Monitoring**(1-9) in the first-run tutorial. More detailed usage, such as sending text or hex values to the board, follows in later chapters.

8 Serial Monitor and Ladder Monitoring

MPINO Studio talks to the board in two ways. The **Serial Monitor** reads and sends the text that your Arduino `Serial` code exchanges, and ladder monitoring overlays the internal state of the ladder program running on the board — the ON/OFF of contacts, memory values, and the power flow through the circuit — on the ladder screen in real time. This chapter covers connecting, receiving, and sending in the Serial Monitor and then the instrumentation, reading the screen, and display formats of ladder monitoring in turn, and it closes with the point that, because the two features share the same serial port between them, only one of them can be on at a time.

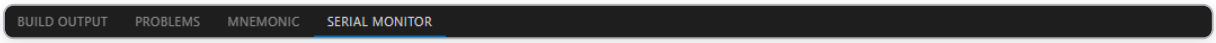
8-1 Serial Monitor Overview

What the Serial Monitor is

The Serial Monitor is the window that exchanges text with the PC over the `Serial` (USART) port of the Arduino board. It reads the values the board sends with `Serial.print` (receiving) and sends the text you type on the keyboard to the board (sending). You use it for firmware debugging and for sending simple commands.

Where the panel is

The Serial Monitor is the rightmost of the four tabs in the bottom panel of the screen — **Build Output / Problems / Mnemonic / Serial Monitor**. For how to open and close the bottom panel and for the Build Output, Mnemonic, and Problems tabs in detail, see **Reading the Bottom Panel**(7-6). Clicking the serial icon in the Activity Bar on the left also opens this tab.



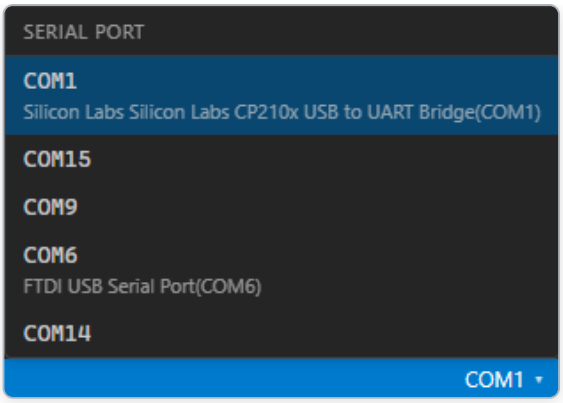
< Figure 8-1 The Serial Monitor tab in the bottom panel — alongside Build Output, Problems, and Mnemonic >

> **If a build or an upload starts, the screen switches to the Build Output tab automatically even while you are looking at the Serial Monitor.** Once the upload has finished, just click the Serial Monitor tab again.

8-2 Connecting

Choosing the port in the status bar

You choose the COM port to connect to from the port dropdown (▾) on the right of the status bar at the bottom of the screen — there is no port selection feature inside the monitor panel. The list refreshes automatically every time you open the dropdown, and each port is shown together with its name and its USB description. If there are no ports, "No ports available" appears. This port selection shares the same value as the one used for uploading — for details, see **Selecting a Port and Uploading**(1-8).



< Figure 8-2 The port dropdown on the right of the status bar — this is where you choose the COM port to connect to >

baud (communication speed)

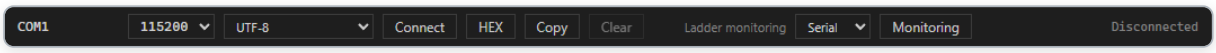
You choose the baud rate from the baud selector in the monitor header (115200 by default, chosen from 16 values). The baud rate **must be exactly the same as the `Serial.begin()` speed in the firmware** — if it is not, the characters simply look garbled, with no warning.

Connecting and disconnecting

Press the connect button in the header and it connects. The button label changes with the state: **Connect** → **Disconnect** (while connected) → **Abort** (while a reconnect is being attempted). The state display distinguishes four values — **Disconnected** / **Connected** / **Reconnecting** / **Error**.

One combination looks contradictory but is correct: when **ladder monitoring** owns the port, the state reads **Connected** while the button still reads **Connect**. The button reflects only the connection *you* opened from this panel, and ladder monitoring did not open that one, so there is nothing here for you to disconnect. Turn monitoring off to release the port (see **Ladder Live Monitoring** later in this chapter). If you press the connect button without having chosen a port, the notice "Select a port first (click the ▾ on the right of the status bar). The monitor cannot open without a port." appears.

The controls in the header sit, from the left, in the order **port display → baud → encoding → the connect button → the HEX/ASCII toggle → Copy → Clear**. To their right, set apart by a gap, stands a group labeled **Ladder monitoring** (**the channel selector + the [Monitoring] button**), and at the far right is the connection **state**. Only that group is set apart because everything before it belongs to the Serial Monitor itself, while the two behind it belong to ladder monitoring (see **Getting Started with Ladder Monitoring**(8-5)).



< Figure 8-3 The Serial Monitor header — from the left, port, baud, encoding, the connect button, the HEX toggle, Copy, and Clear, then after a gap the Ladder monitoring group (the channel selector + the Monitoring button), with the state at the far right >

> Changing the baud rate while connected does not take effect right away — you have to disconnect once and connect again for the new speed to apply. If you unplug the board and plug it back in, the monitor tries to reconnect automatically (up to five times), and on success the line "-- Reconnected automatically --" is printed.

> **For about 0.6 seconds right after you connect, the screen stays empty and then everything that arrived in the meantime appears at once.** The program that opened the port watches the bytes coming in for the first 0.5 seconds to tell whether they are ladder monitoring data or just text. Once it decides they are not monitoring data, it **releases what it held back, in the original order**, onto the screen. Not a single character is dropped, so even the first line of a sketch that prints once in `setup()` and then goes quiet is shown as it is. The brief pause is normal.

8-3 Receiving Data

The receive display

The text the board sends with `Serial.print` / `Serial.println` piles up in the body of the monitor in real time, and every time a new line arrives the display **scrolls to the bottom automatically**. Upload the counter sketch that runs through this chapter (the full code is covered in Sending Data) and you can watch `count = 1`, `count = 2` ... flow past at one-second intervals.

< Figure 8-4 Receiving data — the count values sent by the counter sketch flow past at one-second intervals (state: Connected) >

ASCII / HEX view

You change the display method with the view toggle button in the header. The button carries not the mode you are looking at now but the mode it will switch to — while you are looking at ASCII it reads "HEX", and while you are looking at HEX it reads "ASCII". The HEX view shows the received bytes in two uppercase digits, separated by a space for every byte, as in `AB CD 01 0A`. Changing the mode empties the screen and prints a `-- HEX mode --` or `-- ASCII mode --` line (the previous log disappears).

< Figure 8-5 The HEX view — the same received data shown byte by byte (the -- HEX mode -- line) >

Receive encoding

You choose the character encoding that received text is decoded with from the encoding selector in the header. Five encodings are supported — **UTF-8 / EUC-KR (Korean) / Shift-JIS (Japanese) / Latin-1 (ISO-8859-1) / Windows-1252** — and the default is UTF-8. If Korean text sent by EUC-KR firmware looks garbled, change the encoding to EUC-KR. Changing the encoding empties the screen and prints a `-- Encoding: … --` line (in the HEX view, however, only the setting changes, with no change on screen).

> The encoding choice applies **only to how received data is decoded** (it has no meaning in the HEX view). Sending is always UTF-8 — the send encoding is covered in **Sending Data**(8-4).

Scrolling back over what went past

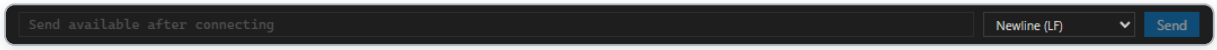
If you want to read something further up while values are streaming past, just **scroll the body up** — the pin to the bottom is released at that moment, so the display is no longer dragged down as new lines keep arriving. Scroll all the way back down and the pin comes back to life and automatic scrolling carries on. Pressing **Clear** or switching between the ASCII and HEX views also pins the display back to the bottom.

> There is no timestamp or pause feature. Use **Clear** to empty the screen and the copy button to copy the contents. Once the screen goes past about 200,000 characters, it is erased automatically from the front.

8-4 Sending Data

Entering and sending

Type text into the input box at the bottom of the monitor panel and send it with the **Enter** key or the **Send** button. While you are not connected the input box is disabled and shows "Send available after connecting". The text you send stays on screen as it is with a `>` prefix (an echo), mixed in chronological order with what comes in from the board.



< Figure 8-6 The state of the input box — disabled while not connected (Send available after connecting), ready for input once connected >

Line endings

The selector next to the input box chooses the line break to append when you send — you pick from **None (no LE) / Newline (LF) / Carriage Return (CR) / Both NL & CR (CRLF)**, and the default is **Newline (LF)**. If your firmware handles line-based input with `Serial.read()`, this setting has to match the line break the firmware expects.

Sending in HEX

In the HEX view you type hexadecimal directly into the input box — separated by spaces as in `AB CD 01`, or run together as in `ABCD01` — and the raw bytes go out as they are. If the format is wrong (a non-hexadecimal character is mixed in, or the number of digits is odd), the notice "Send failed: invalid hex input (e.g. 'AB CD 01' or 'ABCD01')" appears.

> **What you send is always encoded as UTF-8** (regardless of the receive encoding setting). So **to send Korean text** to firmware that understands only EUC-KR, you have to enter those bytes directly in the HEX view. Also, **there is no input history (↑ / ↓)** for calling back what you have sent.

The running example (the counter sketch)

Write the sketch below in a code tab and build and upload it — for how to write code, see **The Code Editor**(7-1). Connect the Serial Monitor at baud 9600 and a `count` value comes in every second; send `R` from the input box and `count` returns to 0 along with `-- reset --`.

```
unsigned long lastTick = 0;
int count = 0;
```

```
void setup() {
  Serial.begin(9600);
  Serial.println("Counter ready. Send R to reset.");
}

void loop() {
  if (millis() - lastTick >= 1000) {   // send the counter once a second
    lastTick = millis();
    count++;
    Serial.print("count = ");
    Serial.println(count);
  }
  if (Serial.available() > 0) {        // receive user input
    char c = Serial.read();
    if (c == 'R' || c == 'r') {
      count = 0;
      Serial.println("-- reset --");
    }
  }
}
```

< Figure 8-7 Sending data — send R and the echo (> R) and the board's reply (-- reset --) appear mixed in with what is received >

This example works **only with ladder monitoring turned off**. Turn ladder monitoring on and the program takes the serial port over, pushing your own `Serial` code aside — the reason in detail is covered in **Serial Monitor vs. Ladder Monitoring**(8-8).

8-5 Getting Started with Ladder Monitoring

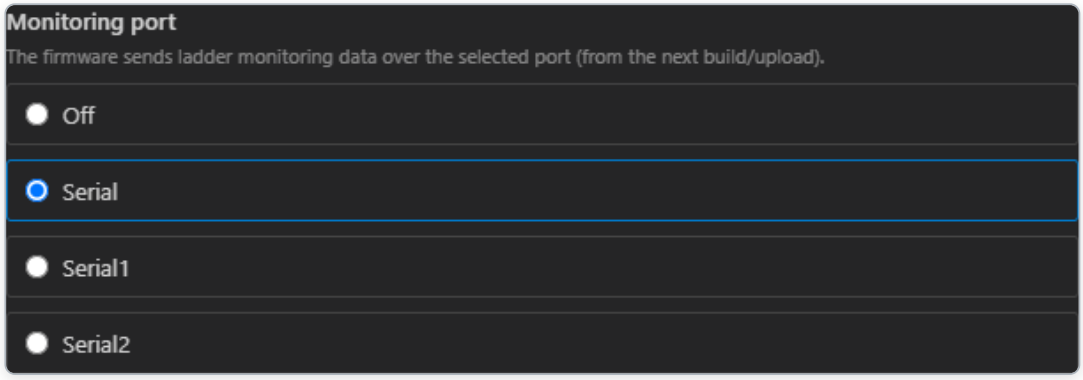
What ladder monitoring is

Ladder monitoring is the feature that overlays the internal state of the ladder program running on the board — the ON/OFF of contacts, memory values, and the power flow through the circuit — on the ladder screen in real time. Unlike the Serial Monitor, which exchanges text, ladder monitoring looks directly into the internal logic state of the PLC.

Three steps of preparation

To see ladder monitoring you have to go through three steps in order. ① Designate the communication port in **Preferences → Ladder Settings → Monitoring port** (for a description of every item there, see **Preferences**(3-1-1)) — this choice plants the code that sends monitoring data out into the firmware

(instrumentation). ② With that in place, **build and upload again** — the hint on the settings screen tells you this order as it is: "The firmware sends ladder monitoring data over the selected port (from the next build/upload)." ③ Turn the communication on with the **Monitoring** button (or Ctrl+M, or "Monitoring ON" in the ladder screen context menu) — there are other places to turn it on as well, and they are gathered under **Five ways to turn it on** below.



< Figure 8-8 Preferences → Ladder Settings → Monitoring port — the firmware is instrumented only once you designate it here >

Two places to choose the port

The port in ① can be chosen **straight from the Serial Monitor header** as well as in Preferences. It is the selector on the left of the **Ladder monitoring** group on the right of the header, and hovering over it brings up the description "The channel the firmware sends monitoring data over. Upload after changing to apply it to the board." The two places point at **exactly the same value**, so changing it in one is reflected in the other immediately, and the confirmations that follow are the same too. Think of the header as a shortcut that saves you opening Preferences every time.

Entry point	Where
The Preferences radio buttons	**Preferences → Ladder Settings → Monitoring port**
The selector in the Serial Monitor header	On the left of the **Ladder monitoring** group on the right of the header

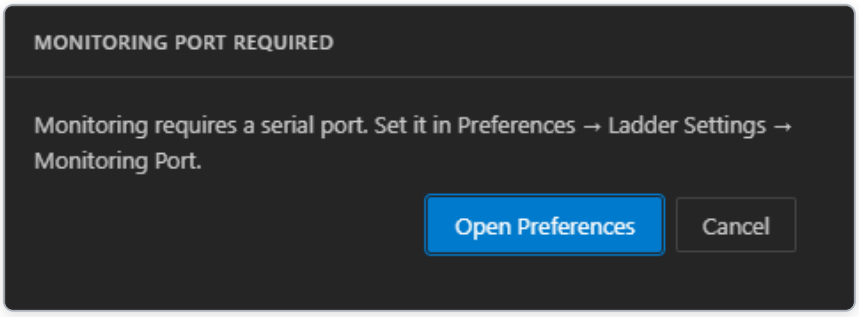
The values you can choose are **Off / Serial / Serial1 / Serial2**, and what is actually listed is **only the communication channels the current board has** — on a board with just two UARTs, for example, Serial2 does not appear in the list. Choose **Off** and the instrumentation itself is left out, so ladder monitoring cannot be used.

The selector in the header is **locked while monitoring communication is on and while an upload is in progress.** Turn the communication off, or wait for the upload to finish, and you can choose again.

> **A confirmation dialog can come up when you change the port — press [Cancel] and it is not changed.**
>
> - If your Arduino code already uses that port (your code has `Serial1.begin()` in it, say, and you chose Serial1 as the monitoring port), the **"Serial1 already in use"** dialog comes up and tells you "Your Arduino code already uses Serial1. Using it for both may cause ladder monitoring to malfunction." Press **[OK]** and the port changes; press **[Cancel]** and both the port and the display on screen stay exactly as they were.

- If you try to turn monitoring on after changing the port but **without having uploaded**, the **"Monitoring Channel Changed"** dialog comes up and asks "The monitoring channel change takes effect only after you upload the program to the board. Upload now?" Choose **[Upload]** and it uploads right away, and monitoring is turned on after it on success. Choose **[Cancel]** and neither the upload nor the monitoring happens.

The "Monitoring" button turns on communication only. What actually makes the firmware send monitoring data out is the port setting in ① and the re-upload in ②. If you try to turn the Monitoring button (or Ctrl+M, or the context menu) on without having set a port, the **"Monitoring Port Required"** notice appears and tells you how to set it: "Monitoring requires a serial port. Set it in Preferences → Ladder Settings → Monitoring Port." A project you saved a while ago may have no instrumentation code in it, so you have to set the port and then build and upload again for monitoring data to come out.



< Figure 8-9 The notice that appears when you turn monitoring on without having set a port >

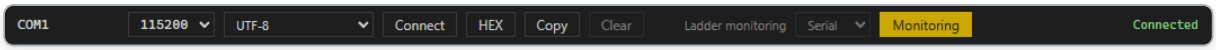
Five ways to turn it on

Once you have finished setting the port and re-uploading, it makes no difference which of the following places you turn the communication on from. All five press **one and the same switch**, so turn it on in one place and the display in the other four changes to the on state along with it.

Where	What you do
The Serial Monitor header	The **[Monitoring]** button on the right of the **Ladder monitoring** group
The ladder screen context menu	The first item — **Monitoring ON** while it is off, **Monitoring OFF** while it is on
The shortcut	**Ctrl+M**
The View menu	**Toggle Monitoring** (see **View**(3-3))
The memory write windows	The **[Monitoring]** button in the header of the **Memory Write List**(3-4-2) window and of the **Write Value to Memory**(4-4) popup

When the button turns on it is highlighted in yellow, and hovering over it shows in a tooltip whether the communication is on or off right now.

The reason a [Monitoring] button sits in the memory write windows as well is that monitoring communication has to be on if you want to write a value to memory and **check the result right away in the current value column**. You can turn it on there without closing the window.

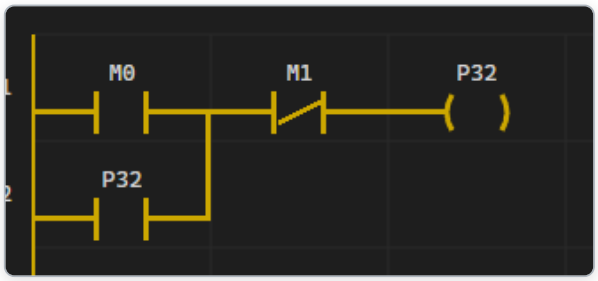


< Figure 8-10 The Monitoring button in its on state (highlighted in yellow) >

8-6 Reading Power Flow and Live Values

Power-flow highlighting

Turn monitoring on and the contacts, coils, and wiring that power flows through are shown **in yellow with thickened lines**, and the power rail on the left is activated along with them. An NC contact (B contact) conducts while its bit is OFF — for how it behaves in detail, see **NC Contact (B Contact)**(5-2-2).

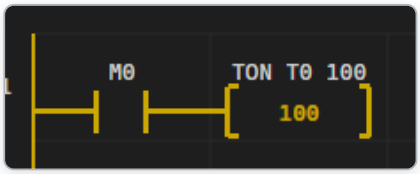


< Figure 8-11 Power-flow highlighting — the contacts and wiring that power flows through are shown in yellow with thickened lines >

Live values per cell

Where a live value is shown and what form it takes differ with the kind of cell.

- Bit contacts (P/M): they show ON/OFF with the power-flow color alone, with no value text.
- Box functions (Func): each column that holds an address shows its current value on the line below it (for example `ADD D0 D1 D2` → `5`, `3`, `8`). Columns with an expression or a constant show no value.
- Word value cells (D/R and the current value of a counter or a timer): shown as a number underneath the cell.
- An empty cell with nothing but an address in it (watch): a bit address is shown as ON/OFF, and a numeric address shows its value.



< Figure 8-12 A box function live value — each column that holds an address shows its current value >

< Figure 8-13 A word value cell — the current value in memory is shown as a number underneath the cell >

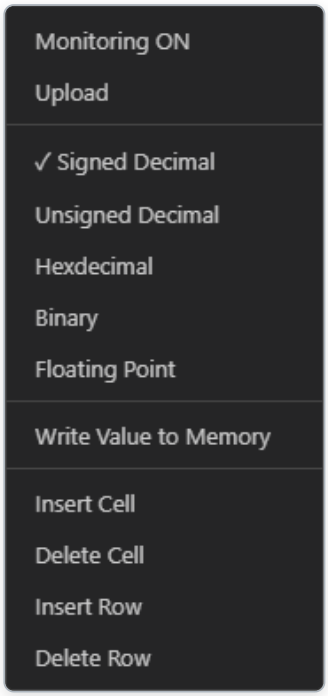
> Place a counter or a timer in a contact position (a done contact) and it is shown only as whether it is done (the power flow), not as a number. A board that was saved and uploaded a while ago does not send this done signal, so a done contact can be shown differently from its actual state — build and upload again after setting monitoring up and it becomes accurate.

> If the board stops or the communication drops, the last values received and the power-flow color stay as they are — only the displays that change with time stop, so that nothing looks falsely live.

8-7 Number Formats and Special Displays

The five number formats

You choose the base you want to see live numeric values in from the ladder cell context menu. The menu lists five formats — **Signed Decimal / Unsigned Decimal / Hexdecimal / Binary / Floating Point** (`Hexdecimal` is the app's actual spelling, not a typo) — and a check mark (✓) appears on the item currently selected. This setting is not applied per cell but is **a global setting that applies to every project in common**, and the default is Signed Decimal.



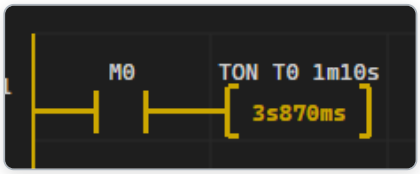
< Figure 8-14 The five number formats — you choose the base from the ladder cell context menu >

Timer time units

Write the preset of a timer box function in the time syntax (`TON T0 1m10s`, for example) and the current value during monitoring is shown in time units as well — only as many units as are needed are joined together, starting from the largest, so it appears as `1m10s`, `43s210ms`, or `500ms`. Write the preset as a plain number (ticks) and the current value is shown as a number as it is.

This time-unit display applies only while the number format is the default, Signed Decimal. Change the number format to another base and you see the number expressed in that base instead of time units, and a

counter does not use the time-unit display in the first place.

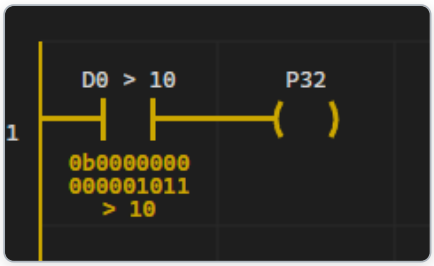


< Figure 8-15 Timer time units — with a time-syntax preset, the current value is shown as 1m10s as well >

Showing comparison contact values

A contact with a comparison expression in it, such as `D0 > D1` (for details, see **Comparison Contacts**(5-6)), shows during monitoring the form `5 > 3`, with both sides replaced by their actual values, underneath the contact as well. While that value is shown the height of that cell grows for a while, and it returns to its original size once you turn monitoring off.

> Only while the number format is Binary is the value split into the left side and the operator plus the right side and shown across several lines. The other formats (Signed Decimal, Unsigned Decimal, Hexdecimal, and Floating Point) are shown on one line. The cell grows in both cases, but the Binary side, which splits across several lines, grows more.



< Figure 8-16 Comparison contact values — during monitoring the actual values are shown underneath the contact, and in the Binary format they split across several lines so the cell grows larger >

8-8 Serial Monitor vs. Ladder Monitoring

One port, two uses

The Serial Monitor's **Connect** and ladder **Monitoring** share the same serial port between them. That is why **they cannot be on at the same time** — turn one on and the other turns off automatically. Turn ladder monitoring on and that session takes the port over at 115200, and the serial send input box is disabled (because only ladder state data should flow over the monitoring communication).

When to use which

What you want to do	Feature to use
Viewing the debug output of Arduino code (`Serial.print`) or sending text commands	Serial Monitor (8-1–8-4)
Viewing the contact, memory, and power-flow state of a ladder program in real time	Ladder monitoring (8-5–8-7)

Relation to the running example

Code in which you use `Serial` yourself, like the counter sketch in 8-4, can be observed in the Serial Monitor **only with ladder monitoring turned off**. That is because turning ladder monitoring on has the program take the serial port over for instrumentation.

> **The Serial Monitor = the window that communicates with your Arduino code in text**, and **ladder monitoring = the screen that looks into the internal state of the PLC**. The two share the same port between them, so only one of them can be on at a time.

9 Activity Bar and Side Panels

Along the left edge of MPINO Studio there is a vertical column of icons called the **Activity Bar**. Press one of these icons and information such as project files, board details, and libraries opens in a side panel on the left of the screen, while information that needs a window of its own — symbols, for instance — opens as a modal, so that you can browse and manage it. This chapter explains the seven icons in the Activity Bar and what each of the screens they open is for.

9-1 Activity Bar Overview

What the Activity Bar Is

It is the column of icons laid out vertically at the far left of the screen. Press an icon and the screen that icon is responsible for opens — some icons unfold a side panel on the left, some raise a separate modal (popup) window, and some move you to the bottom panel below the editor. The Activity Bar itself is always shown on screen, and there is no feature for hiding it.

What the Seven Icons Open

Icon	What it opens	In this manual
Explorer	Side panel	**The Explorer Panel**(9-2) below
Ladder Settings	Side panel	**The Ladder Settings Panel**(9-3) below
Ladder Pin Map Settings	Modal	"Board Pin Map Editing" in **Pin Editing**(2-4)
Library	Side panel	**The Library Panel**(9-4) below
Symbols	Modal (modeless)	**Symbols**(9-5) below
Serial Monitor	Bottom panel	**Serial Monitor Overview**(8-1), **Getting Started with Ladder Monitoring**(8-5)
Settings	Modal	**Preferences**(3-1-1)

The icon labels, in order, are "Explorer" / "Ladder Settings" / "Ladder Pin Map Settings" / "Library" / "Symbols" / "Serial Monitor" / "Settings". Of these, only the three icons Explorer, Ladder Settings, and Library open a side panel; the remaining four move you to a different screen, such as a modal or the bottom panel. The icon that corresponds to the screen currently open has its background highlighted, so you can tell at a glance which screen is open.

< Figure 9-1 The Activity Bar on the left of the screen — each icon opens a side panel, a modal, or the bottom panel >

Opening and Closing the Sidebar

You can open and close the side panel with `Ctrl+B` (the same action as "Toggle Sidebar" in the View menu — see **View**(3-3)). Press one of the three icons that open a side panel (Explorer, Ladder Settings, and Library) and it behaves as follows.

- If the sidebar was closed → that icon's panel opens.
- If the sidebar was open with a different panel selected → it switches to that icon's panel and the sidebar stays open.

- If the sidebar was open and that panel was already selected → the sidebar closes (it is treated as pressing the same icon one more time).

> The width of the side panel is fixed at 280px, and you cannot widen or narrow it by dragging its edge. The bottom panel below the editor (Build Output, Serial Monitor, and so on) can have its height adjusted by dragging its edge, but that is a separate feature with nothing to do with adjusting the width of the side panel. Click the edge of the bottom panel, or move focus to it with the Tab key, and you can adjust the height from the keyboard as well — the **↑ / ↓** keys raise or lower the height by 8px, the **PageUp/PageDown** keys by 32px, the **Home** key jumps straight to the maximum height, and the **End** key to the minimum height. You can use mouse dragging and keyboard operation together.

9-2 The Explorer Panel

The Project Nameplate

At the top of the Explorer panel, information about the project currently open is shown on two rows. The **File** row is the file name of the current project `.mp2`, and the **Board** row is the name of the board the project targets. You can check right here which file is the current build input and which board it is aimed at. If there are unsaved changes, a dot (●) appears at the right of the panel header to let you know.

To the right of the **Board** row is the **[Change board]** button. It is the entry point for changing only the board the current project targets, without creating a new project; press it and the **Select Board** modal opens with the current board already selected (the modal itself is the same screen you saw in **Creating a Project and Selecting a Board**(1-4)). Pick another board and confirm, and only the target board information is replaced while the code and the ladder are left as they are; cancel and the board does not change. You can open the same feature from **Change Board**(3-5) in the Settings menu.

In a project with option modules attached the nameplate has three rows. An **Option modules** row is added below the **Board** row, summarizing which modules the current project uses and how many of each, in the form `X2 · Y1 · K1`. The letter is the kind of module (X = analog input, Y = analog output, F = PT100Ω input, K = pulse output) and the number after it is the count, and **a kind whose count is 0 is not listed at all.** On a board where all four kinds are 0, or which does not support option modules, this row does not appear at all, so you see only the two rows described above. The place where you set the module counts is the board selection modal, covered in **Creating a Project and Selecting a Board**(1-4).

The Code and Ladder Tree

Below the nameplate comes a tree divided into two groups — the **Code** group, which collects the `.ino` files, and the **Ladder** group, which collects the ladder circuits. The editor area has no separate tab bar, so this tree serves as the tabs — click an item in the tree and that file opens in the editor area, and if it is already open the editor switches to it.

Detailed operations such as adding a file or a ladder, renaming (F2), deleting, and reordering by dragging are covered in **Working with Code Tabs**(7-2) — the Code and Ladder groups in the Explorer panel are exactly the tree that section describes, so they are not repeated here.

> Code files and ladder circuits are all held in one and the same `.mp2` project, so the Explorer panel is the

place where you can scan everything the project holds for editing from a single screen.

< Figure 9-2 The Explorer panel — the Code and Ladder file tree with the project and board names (for the operations in detail, see Chapter 7) >

9-3 The Ladder Settings Panel

What You See

Press the "Ladder Settings" icon in the Activity Bar and information about the board the current project uses appears in the side panel. The board name is shown on a single row at the very top of the panel, and below it come two sections, each of which you can collapse or expand by clicking its section heading. This screen is read-only throughout — the panel itself has no button for changing the board, and it is there for checking information about the board the current project uses. To change the board, use the **[Change board]** button in **The Explorer Panel**(9-2) or **Change Board**(3-5) in the Settings menu.

Pin mapping

This is a table that shows the pin numbers used in the ladder (LD) side by side with the actual Arduino pin numbers. Every pin the board defines is listed in the two columns **LD** and **Arduino**, and pins **in the "Unused" state are shown dimmed** (hover over such a row and the "Unused" tooltip comes up).

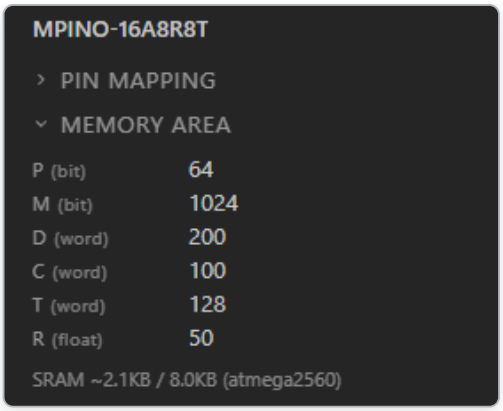
> **In a project you have just created, this table is shown dimmed from top to bottom.** The "Use" setting of a pin defaults to **cleared**, so no pin has been turned on yet. Nothing is wrong with the display; the moment you turn a pin you are going to use on, that row becomes crisp. For how to turn one on, see "Board Pin Map Editing" in **Pin Editing**(2-4) — you tick the **Use** column in the pin map editor that the **Ladder Pin Map Settings** icon in the Activity Bar opens.

Memory area

The counts for the six areas P/M/D/C/T/R are listed together with their respective units (bit/word/float), and below the list the total SRAM usage summed over the six areas is shown together with the chip name. Depending on the board, the board's total SRAM capacity may be shown next to it as well.

> This panel only shows information about the current board. To edit the pin assignments you have to use the pin map editor that **Ladder Pin Map Settings** (the RAM-chip icon) in the Activity Bar opens — for details, see "Board Pin Map Editing" in **Pin Editing**(2-4). To change the counts of the memory areas, use **Memory Area Settings**(3-5) in the Settings menu (see **Data Memory**(2-1)).

> This **Ladder Settings** side panel and the **Ladder Settings** tab in the Preferences window (one of the four tabs in **Preferences**(3-1-1) — it decides the ladder symbol display mode, whether the ladder is used, and so on) carry exactly the same name in English yet are different screens. This panel shows the current board's pin and memory information read-only, whereas the **Ladder Settings** tab in **Preferences** holds the options that decide how the ladder is displayed and handled.



< Figure 9-3 The Ladder Settings side panel — memory areas and the SRAM total (read-only). Each section can be collapsed and expanded by clicking its heading >

9-4 The Library Panel

What the Library Panel Is

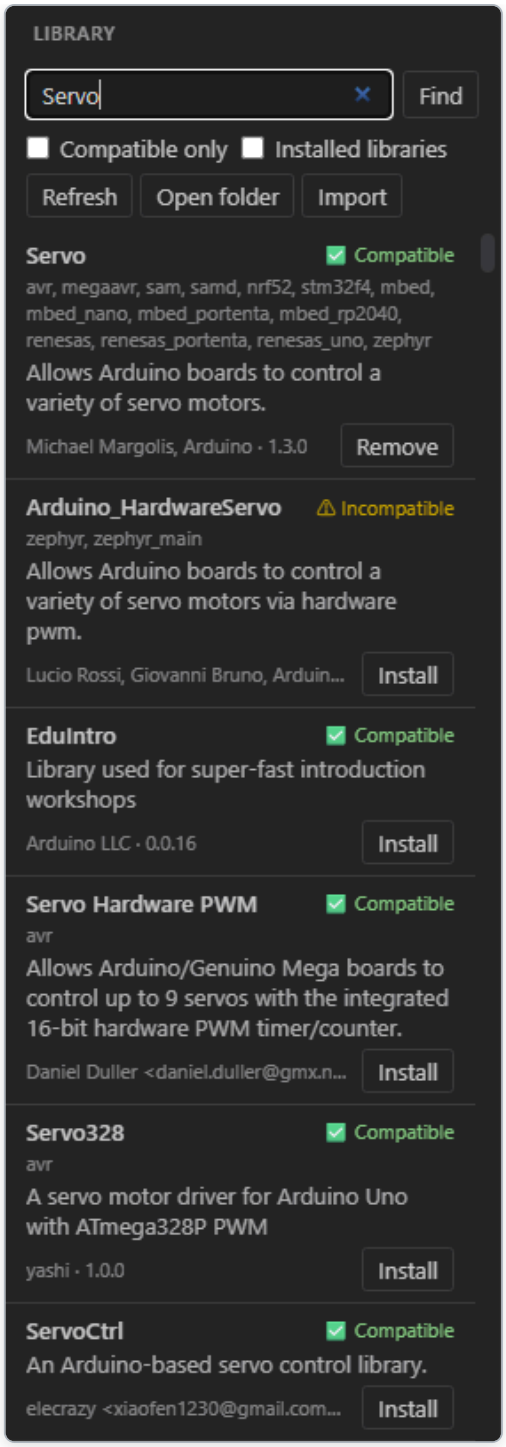
This is the side panel that opens when you press the "Library" icon in the Activity Bar. It is the screen for searching, installing, importing, and managing Arduino libraries, and it wraps the library features of arduino-cli as they are. The functions and constants of a library you install here appear in the code editor's autocomplete list — for details, see **Autocomplete**(7-3).

Search and Install

Type a name into the search box at the top of the panel (placeholder "Search libraries...") and the search runs automatically a short while after you stop typing. The **[Find]** button next to the search box downloads the online library list (the registry) itself afresh and refreshes it.

Each item in the search results shows the library name and a compatibility badge ("✅ Compatible" or "⚠ Incompatible"), the supported architectures (nothing is shown when every architecture is supported; they are listed only when the library is restricted to particular architectures), a one-line description, and the author and version, and on the right there is an **[Install]** or a **[Remove]** button depending on whether it is installed. The two checkboxes below the search box, **Compatible only** and **Installed libraries**, let you filter the list (check **Installed libraries** and the screen switches from the search results to the list of libraries currently installed — see "Managing and Deploying Installed Libraries" below).

> The compatibility badge is only a hint for reference. Many libraries are registered as supporting every board without specifying a particular architecture, so the badge does not guarantee actual operation 100%.



< Figure 9-4 Library search — names, compatibility badges, and an Install or Remove button depending on install state, with the filter checkboxes >

Managing and Deploying Installed Libraries

Turn on the **Installed libraries** checkbox and every library currently installed on this PC is listed regardless of the search term (this covers the ones installed from the registry, the ones added with "Import" below, and the ones dropped straight into the library folder and picked up with **[Refresh]**). Each item shows the name and the version; press **[View]** and the folder the library is installed in opens, and press **[Remove]** and it is deleted.

Every item has a **Deploy** checkbox. Turn this checkbox on and that library is bundled into the project file when you save the `.mp2` project — open that `.mp2` on another PC and there is no need to install the library

separately. Libraries the app provides by default (copies obtained through the vendor channel), however, have no Deploy checkbox at all. Installing the program already brings the same copy with it, so there is no need to carry a duplicate inside the project.

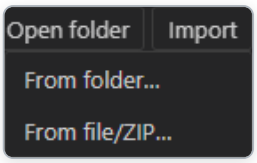
< Figure 9-5 Installed libraries — View and Remove, and the Deploy checkbox that bundles a library into the .mp2 >

Import (Offline Libraries)

A library that is not in the registry can be added by hand with the **[Import]** button. Press the button and two menu items appear, "From folder..." and "From file/ZIP...".

- "From folder..." takes a whole library folder that you pick.
- "From file/ZIP..." lets you pick a ZIP file or individual source files. Pick a ZIP and it is installed right away, but pick individual source files and a window asking for the library name appears (the name cannot be left empty, and characters such as `\ / : * ? " < > |` cannot be used). Pick ZIPs and individual source files mixed together in one go and the selection is rejected with the notice "ZIP and source files cannot be selected together — import them separately.", so you have to import the two separately.

If you dragged files straight into the library folder (without going through Import), you apply that change with the **[Refresh]** button — applying it can take about 3 seconds at the shortest and up to about a minute at the longest, and during that time autocomplete may be briefly slow or look empty. If there are files sitting there without the structure of a library folder, the panel shows the notice "N unrecognized file(s)", and in that case you have to install them properly with **[Import]** → "From file/ZIP..." for them to be reflected in the build and in autocomplete. To look straight at the folder the libraries actually sit in from the file explorer, press **[Open folder]**.



< Figure 9-6 Import — adding an offline library from a folder or from a file/ZIP >

> **User Library Folder** on the "General" tab of **Preferences**(3-1-1) is the setting that specifies the location — which folder libraries are read from and written to — while this Library panel is the screen where you actually search, install, and manage the libraries in that folder. If you want to change the folder location, work in Preferences; if you want to handle the libraries themselves, work in this panel.

9-5 Symbols

Symbols are a feature for attaching an alias that is easy for a person to read to a memory address such as `D0` or `M10` (for example, `D0` → `Temp`). Once you have attached an alias, you can use that name instead of the address in ladder contacts, box functions, and comparisons, and in C code, which makes the circuit considerably easier to read.

> The "symbol" spoken of here is an alias for an address. It is a different concept from the "library symbols" that appear in **Autocomplete**(7-3) (the functions and constants provided by libraries and the board core), so do not confuse the two.

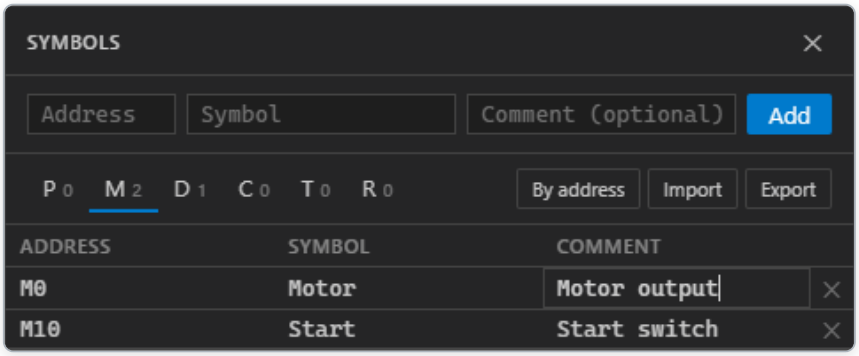
9-5-1 Opening the Symbols Window

It opens when you press the "Symbols" icon (shaped like a tag) in the Activity Bar. Unlike the other Activity Bar modals, this window is modeless — you can go on operating the ladder editor with the window up, and you can grab the header and drag it to wherever you want on the screen. You close it by pressing the same icon again or by pressing Esc. Note, though, that an Esc pressed while input focus is somewhere else, as during ladder cell editing, is used first to cancel that input and does not close the Symbols window. There is no dedicated shortcut or menu item for opening the Symbols window — the Activity Bar icon is the only way in.

9-5-2 Window Layout

There are six tabs, P/M/D/C/T/R, one per memory area, and each tab divides the symbols by the first letter of the address (the `D` tab, for example, lists only addresses that start with `D`). Each tab also shows the number of symbols registered in that area. The table below the tabs is made up of three columns, **Address** / **Symbol** / **Comment**.

The **[By address]** / **[By symbol]** button above the table shows the criterion the list is currently sorted by, and pressing it switches to the opposite criterion. This sorting changes only the order you see on screen; it affects neither the order the symbols are saved in the file nor the data itself.



< Figure 9-7 The Symbols window — the P/M/D/C/T/R tabs and the Address, Symbol, and Comment columns (modeless, and movable) >

9-5-3 Adding, Editing, and Deleting

Fill in the input row at the very top (placeholders, in order, "Address" / "Symbol" / "Comment (optional)") and press the **[Add]** button or hit Enter, and a new symbol is registered. On a successful registration the screen switches automatically to the tab the address belongs to, and the input row is cleared while focus is kept, so that you can carry straight on entering the next item.

A row that is already registered can be corrected on the spot by clicking the cell you want (inline editing). Hit Enter or click outside the cell (focus leaves it) and the value is saved; press Esc and the edit is cancelled and the original value comes back. The × button at the end of a row deletes it on the spot with no confirmation window.

> Adding, editing, and deleting symbols (including the ladder cells that are rewritten along with an address change) are not covered by the ladder editor's undo (Ctrl+Z). To reverse them you have to correct the values by hand again.

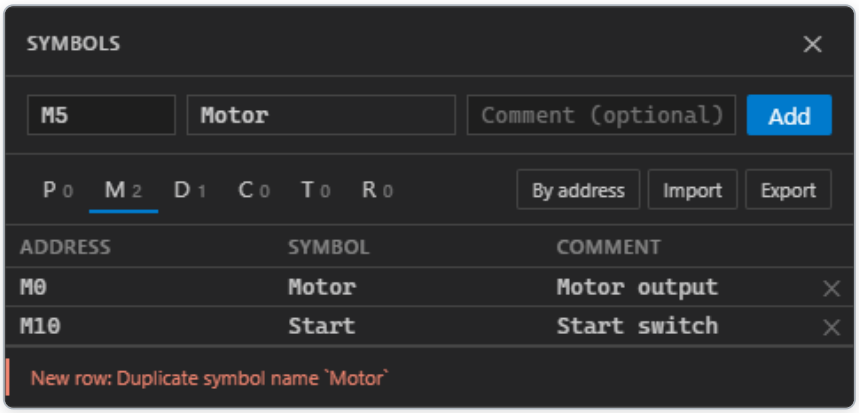
9-5-4 Naming Rules

Both the address and the name have syntax rules, and if you break them the row itself is not registered or (if the row is already registered) is highlighted in red, with the reason shown below the table.

- **Address** — it has to be one area letter (P/M/D/C/T/R) followed by digits (like `P0` or `D10`). Example: `Invalid symbol address syntax 'D0X'`.
- **Name** — it must not be empty, its first character has to be a letter (Korean included) or `_`, and it must not have the same form as an address or start with `@`. Example: `Symbol name is empty`.

- **Neither addresses nor names may be duplicated.** Duplicate names are judged without regard to case (`Motor` and `motor` are treated as the same name). Examples: `Duplicate symbol address `D0``, `Duplicate symbol name `Motor`` (the duplicate address and name messages really do wrap the value in backticks on screen).

A violation coming from the top input row is prefixed with `New row:` in the error list, so the same duplicate-name message reads `New row: Duplicate symbol name `Motor`` there. Violations found on an already-registered row are prefixed with that row's number instead.



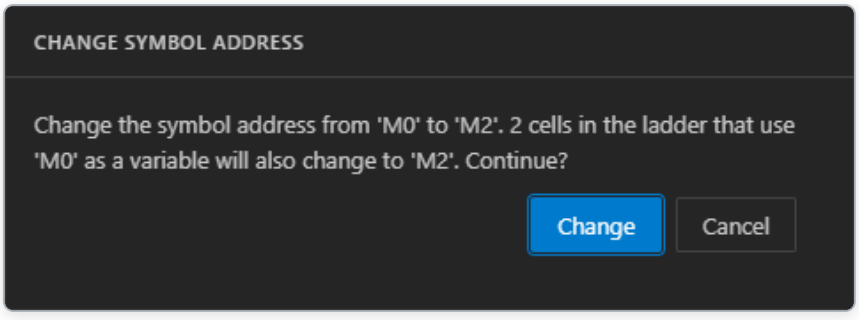
< Figure 9-8 A naming rule violation — an entry that breaks a rule is not registered, and the reason is shown below the table >

9-5-5 Changing an Address or a Name

If you change an address, the value in every ladder cell that used that address changes to the new address along with it — it is a change that is hard to reverse, so a confirmation window like this appears before it happens: "Change the symbol address from '...' to '...'. N cells in the ladder that use '...' as a variable will also change to '...'. Continue?" It is only applied once you press **[Change]**, and if you cancel, both the symbol and the cells stay as they are (there is no partial application). If not a single cell is affected, the change happens right away with no confirmation window.

If you change only the name, the cells that pointed at that symbol follow the new name automatically — no confirmation window appears.

> A ladder cell always stores the actual address, and the symbol is no more than an alias for convenience when displaying and entering that address. That is why changing only the name has no effect at all on how the circuit behaves.



< Figure 9-9 The address change confirmation — it tells you how many ladder cells will change along with it >

9-5-6 Import and Export

The **[Import]** / **[Export]** buttons let you exchange the symbol list with a file. Two formats are supported, Excel (.xlsx) and CSV (.csv), and the column order in the CSV is address / symbol / comment.

> Importing replaces the current list wholesale with the new list (it does not merge it with the existing list). And if even one line in the file breaks a naming rule, the whole import is cancelled and the existing list is kept as it is — there is never a partial application.

9-5-7 Using Symbols in C Code

When you build, a macro of the form `#define <name> <area>[<number>]` is generated automatically for each registered symbol (if you gave `M0` the name `Motor`, for example, `#define Motor M[0]`), so that you can read and write that memory by that name from C code just as it is.

Symbols that fall under any of the following, however, are excluded from this macro — the header file keeps a "(제외)" ("excluded") comment that gives the reason.

- The name does not conform to C identifier syntax (a non-ASCII script such as Korean, for example)
- The name is a C/C++ reserved word
- The name collides with a memory area name (P/M/D/C/T/R) or with a name in other generated code

In other words, a symbol named in Korean (or in any other non-ASCII script) can still be used in the ladder circuit and in comparisons as it is, but it does not appear in the mapping that lets you reach it by name from C code — to handle that memory from C code, you write the address directly.

> Even if you enter an address or a name that has not been registered yet into a ladder cell, the value is stored as it is, and once you later register a symbol with that address or name it is linked up automatically. The behavior in detail is the same deferred resolution described in **Editing the Ladder: From a P0 Contact to a P32 Coil**(1-5) and **Ladder Cell Editing**(2-4-2).

9-6 Other Activity Bar Items

The four icons covered above from 9-2 through 9-5 — Explorer, Ladder Settings, Library, and Symbols — open a side panel or a modeless window. The remaining three icons — Ladder Pin Map Settings, Serial Monitor, and Settings — open a modal or the bottom panel rather than a side panel, and since they are already covered in detail in other chapters, this section only briefly says what each of them opens and where you can find the full explanation.

Ladder Pin Map Settings

Press the icon shaped like a RAM chip (or the shortcut `Ctrl+Shift+P`) and a modal for editing the pin assignments of the board the current project uses opens. Unlike **The Ladder Settings Panel**(9-3) seen above, which only shows the pin mapping read-only, this icon leads to an editing screen where you can actually correct those assignments. This screen is also where you turn the **Use** setting of each pin on and off.

In a project with option modules attached, the pins the modules created (the MPAINO-derived pins) are edited in the same table as the pins of the board itself. So the screen the **[Edit Pins...]** button in the board selection modal opens and the screen this icon opens always show the same rows for the same project.

Whichever route you take, the result of the editing is **saved in the project you have open (`.mp2`)**, and you have to save with `Ctrl+S` to leave it on disk — the board definition file itself is not changed, so other projects are unaffected.

For details, see "Board Pin Map Editing" in **Pin Editing**(2-4).

Serial Monitor

This icon moves you not to a side panel but to the Serial Monitor tab of the bottom panel at the bottom of the screen. If the bottom panel was closed it opens with the Serial Monitor tab selected; if a different tab was open it switches to the Serial Monitor tab; and if the Serial Monitor tab was already selected the bottom panel closes — the same three-way behavior described in "Opening and Closing the Sidebar" above. For details, see **Serial Monitor Overview**(8-1) and **Getting Started with Ladder Monitoring**(8-5).

Settings

Press the gear icon and the Preferences modal opens. It is a screen that comes up as a separate window, with nothing to do with the side panel. For details, see **Preferences**(3-1-1).