

BuilderRunner2 / Graphic Builder Operator Manual

Visual node-graph editor for control-diagram logic. Screenshots show cdcontrol.json loaded in the light theme.

Ed Chapman 2026



Hey-san,

This is Ingrid. Last night, while I was having fun square dancing with my girlfriends, Astrid and Freja, I thought it might be a nice idea if today I tried to string a few words together explaining the new Graphic Control Builder from Vremsoft LLC. From time to time, the algorithms library will be updated with more and more algorithms; this will be done automatically from your distribution if you have a license. So sit back and enjoy, and if you decide to make some empty candy wrappers while watching this video, that is OK too—please do so! By the way, my favorite empty candy wrappers are from White Chocolate Kisses.

Tack för att du lyssnar

Introduction

The Graphic Builder is a PySide6 visual node-graph editor. Operators place nodes, connect them with wires, and run the resulting diagram as a live STEC Multiverse application. Each node executes a dynamically-loaded Python algorithm on a trigger and propagates signals through the parent/child graph.

This manual covers the V2 prototype interface: the canvas, menus, alias table, declarations, bridges, runtime panels, preferences, and the CD control example shipped as `cdcontrol.json`.

Main Window Overview

The main window is organised around a central canvas with dockable panels on the left, right, and bottom. The title bar shows the loaded project file. The status bar shows the project name, node count, and selection count. Toolbars mirror the File, Edit, Tools, Communication, View, and Help menus.

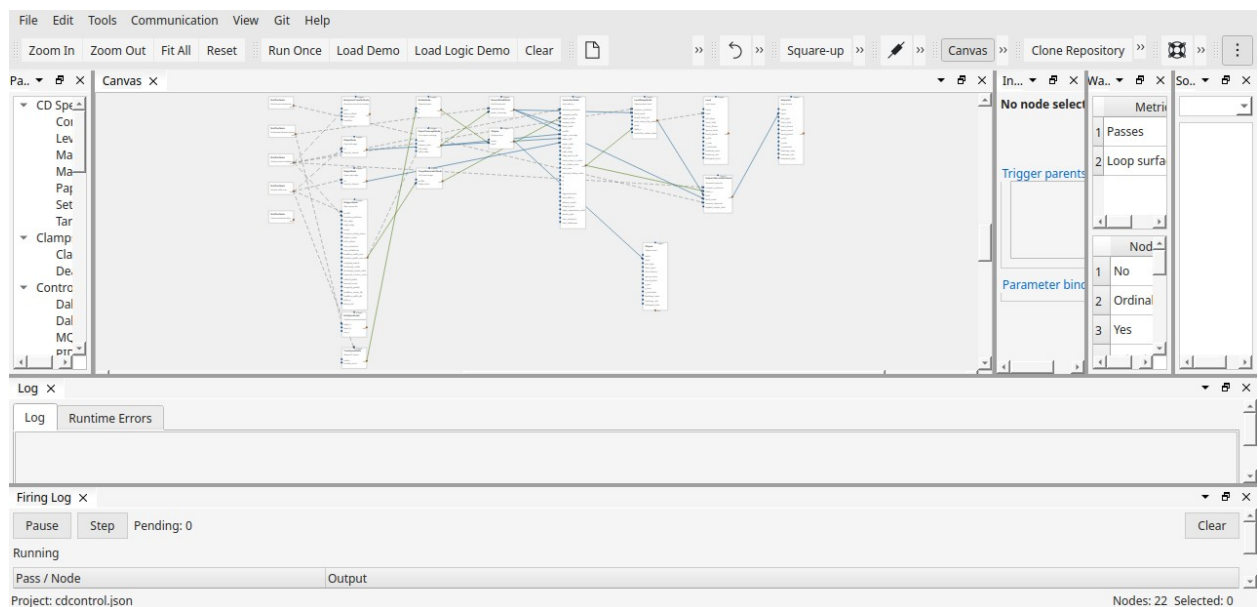


Figure 1: Main window with `cdcontrol.json` loaded.

Default dock panels

- Palette (left) — lists discovered algorithms grouped by category.
- Inspector (right) — edits the selected node's parameter bindings.
- Watch (right) — displays live engine status and resolved alias values.
- Sources (right) — live sources from enabled communication bridges.
- Log / Firing Log (bottom) — application output and stream debugger.

Menu Reference

The following sections summarise each top-level menu. The screenshots were captured by opening the actual menus in the offscreen harness.

File

- New Project — create a project directory and first .json file.
- New File in Project — add another .json to the current project.
- Load Demo / Load Logic Demo — sample drawings for exploration.
- Open / Open Recent / Save / Save As — project file operations.
- Print / Export PDF — render the canvas or a panel to paper or PDF.
- Close — exit the application.

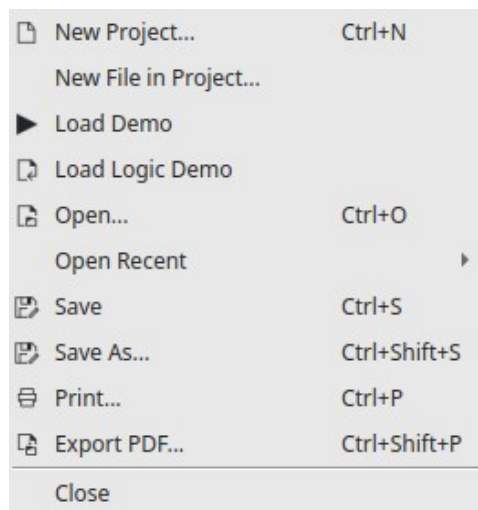


Figure: File menu.

New Project

File → New Project... (Ctrl+N) creates a new project directory and saves the first JSON drawing file inside it. A project is simply a folder that groups related drawings; the first file has the same base name as the folder.

When you choose New Project, the Create Project dialog asks for two values:

- Project name — the name of the new folder and the base name of the first file. Only letters, digits, spaces, underscores, hyphens, and periods are recommended.
- Create in — the parent folder where the project folder will be created. The default comes from the Grade/Clone Folder preference. If that preference is empty, it falls back to the Project Folder; if that is also empty, it uses ~/graph_builder.

After you click OK, the app clears the current canvas, creates <parent>/<name>/<name>.json, saves an empty project, and makes it the active file. Both the Project Folder and the Grade/Clone Folder preferences are updated to the new location so that subsequent Open, Save

As, and Git operations start in the same workspace. The new project is also added to the Open Recent list.

If a file already exists at the target path, a confirmation dialog asks whether to overwrite. If you choose not to overwrite, the dialog closes and the current project is unchanged.

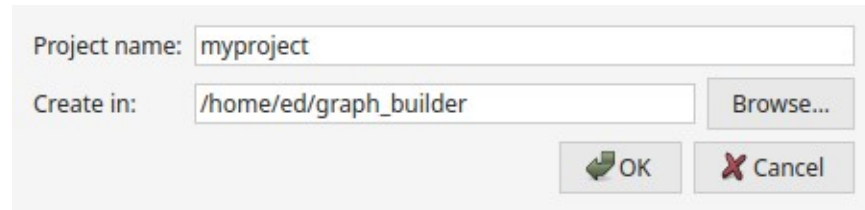


Figure: Create Project dialog with default parent folder and sample name.

New File in Project

File → New File in Project... clears the canvas and prompts for another JSON file in the current project directory. Use this when a control system needs more than one drawing — for example, a `cdcontrol.json` drawing for the live control chain and a separate `cdmapping.json` drawing for mapping diagnostics.

The save dialog starts in the folder of the currently loaded file, or in the Project Folder if no file is loaded. The new file becomes the active project, the Project Folder preference is updated, and the file is added to Open Recent. The current canvas is cleared before saving, so the new file begins empty.

Load Demo

File → Load Demo clears the current canvas and loads a sample distributed-digital-control (DDC) loop. The demo graph is a five-node chain: a `NotifierNode` on the process variable (`water_pv`) triggers a `SubtractNode` that computes `error = pv - setpoint`, followed by a `PIDNode`, a `ClampNode`, and a `ValveOutputNode` that writes the result to `water_valve`.

The alias table is pre-populated with external `stdcomQtPS` topics for `water_pv` and `water_valve`, constants for setpoint (50.0), gain (2.0), and high limit (3.0), and an enable ordinal. The required subscriptions and the enable ordinal are declared automatically. After the graph is built, the engine injects a demo value of 45.0 on `water_pv` so the loop starts with live data.

Load Logic Demo

File → Load Logic Demo clears the canvas and loads a small Boolean example that demonstrates source-bound value inputs. The graph contains a `NotifierNode` clock (`clk`), two `ConstantNodes` (True and False), an `AndNode`, and a `ValveOutputNode`.

One `ConstantNode` is wired to the `AndNode` with a source-only edge (it only supplies a value), while the other is wired with a trigger+source edge (it also wakes the `AndNode`). This makes the distinction between the two edge modes concrete. The alias table declares `stdcomQtPS` topics

for `logic_clock` and `logic_out` plus `True/False` constants, and the clock is injected with 1.0 after the engine starts.

Open

File → Open... (Ctrl+O) opens a saved graph JSON file. The file dialog starts in the Project Folder. After you choose a file, the Project Folder preference is updated to that file's directory, the current canvas is cleared, and the project is loaded through the migration path so that older schema versions are upgraded automatically.

Loading restores nodes, edges, alias table, parameter bindings, trigger parameters, and strict declarations. If the project was saved before multi-bridge support, any external alias rows with an empty Bridge cell are silently assigned to the sole enabled bridge, or you are prompted to pick one when several bridges are enabled. The graph is rebuilt, edge styles are refreshed, the view is fitted to all nodes, the window title is updated, and the file is added to the Open Recent list.

Open Recent

File → Open Recent is a submenu that lists the last 10 projects you worked with, newest first, deduplicated. The list is populated just before the menu opens, so it is always current.

Each entry shows a number and the file name; hovering shows the full path in a tooltip. Choosing an entry loads it exactly like File → Open. If the file has been moved or deleted, it is removed from the list and a message is printed to the Log panel.

Save

File → Save (Ctrl+S) writes the current graph to the loaded project file. If no file is loaded — for example, after File → New Project or Clear — Save behaves the same as Save As and prompts for a path.

The saved data includes the canvas nodes and edges, the alias table, parameter bindings, trigger parameters, and strict declarations. After saving, the green staged-change indicators are cleared, the window title is refreshed, and the file is added to Open Recent.

Save As

File → Save As... (Ctrl+Shift+S) saves the current graph to a new JSON file. The save dialog starts in the Project Folder. If the chosen name does not end in `.json`, the suffix is added automatically.

After saving, the staged-change indicators are cleared, the Project Folder preference is updated to the new file's directory, the new file becomes the active project, the window title is updated, and the file is added to Open Recent.

Print

File → Print... (Ctrl+P) opens the Print dialog. First you choose a target: the entire window, the canvas, or any dock panel (Inspector, Watch, Sources, Log, Firing Log, or Palette). For the canvas you also choose page scaling:

- Fit to one page — scales the whole drawing onto a single sheet. Best for small drawings; large control drawings will have very small port labels.
- Readable (labels ≥ 8 pt, tiled) — splits the drawing across the smallest sheet grid that keeps labels at least 8 pt, then enlarges to fill the grid. This is the default for canvas prints.
- Actual size (100 %), tiled) — prints 1 canvas pixel = 1/96 inch, tiled across sheets.

A Vector print checkbox sends a vector PDF to the printer through CUPS at High quality; when unchecked, the panel is rendered as a bitmap at the printer's native resolution. After you click Print, the system print dialog appears for final printer selection.

Export PDF

File → Export PDF... (Ctrl+Shift+P) opens the Export to PDF dialog. It offers the same target and page-scaling choices as Print, but without the vector-print option (the output is always a PDF file).

After you choose the file name, the selected widget is rendered to PDF. Canvas exports use tiling when the Readable or Actual size scaling modes are selected, and the page is rotated to match the drawing's aspect ratio. This is the same mechanism used to produce the canvas printout shown later in this manual.

Close

File → Close exits the Graphic Builder. Before the window closes, the current project path is saved for session restore, every enabled bridge plugin is disconnected, and any remaining top-level windows such as floating docks, inspectors, or dialogs are closed so the application shuts down cleanly.

Edit

- Undo / Redo / Clear — canvas edit history.
- Search — find nodes by algorithm, display name, or alias (with wildcards) and highlight them in yellow (Ctrl+F).
- Alias Table — cross-reference table for aliases, constants, externals, and node outputs.
- Algorithm Builder — generate a new algorithm skeleton.
- Declarations — strict ordinal/subscription declaration manager.
- Preferences — theme, bridges, shortcuts, and folders.

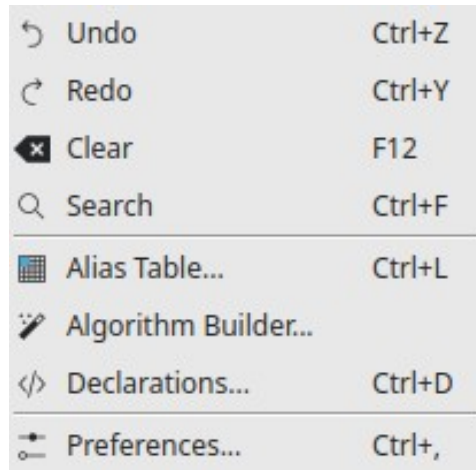


Figure: Edit menu.

Undo / Redo

Edit → Undo (Ctrl+Z) reverses the last canvas edit. Edit → Redo (Ctrl+Y or Ctrl+Shift+Z) reapplies it. The undo stack tracks node add/move/delete, edge add/delete/routing/waypoint changes, port-layout changes, and node display-name/description edits. Loading a project or clearing the canvas resets the stack.

Clear

Edit → Clear (F12) removes every node and edge from the canvas, clears the alias table, declarations, parameter bindings, and runtime state. The current file path is also cleared, so the next Save will behave like Save As. This is the same action as the Clear button on the Run toolbar.

Search

Edit → Search (Ctrl+F) — also at the bottom of the blank-canvas right-click menu — finds nodes on a large drawing and paints their background yellow so they are easy to spot. The popup offers three searches, each listed alphabetically:

- Algorithms — every algorithm used on the canvas, listed once no matter how many instances exist. Picking one highlights all its instances: with two DahlinController nodes and one PIDNode on the drawing, the list shows DahlinController and PIDNode, and choosing DahlinController highlights both.
- User Names — every Descriptor display name placed on the page (F2 on a node). Nodes without a display name are listed under their algorithm name, so unnamed nodes are easy to find for QC.
- Aliases — every alias referenced by a node parameter. Picking one opens a small dialog with the alias in an editable line: press Search (or Enter) to highlight the nodes using it, or edit the line first — "*" matches any text, "?" one character. Choosing Ordinal.AllowCLID and searching highlights exactly its users; backspacing it to Ordinal.* highlights every block using an alias starting with Ordinal.

Two shortcuts skip the popup entirely: right-click a row in the Alias Table → Search highlights every node using exactly that alias, and right-click an algorithm in the Palette → Search highlights every block on the canvas using that algorithm.

The yellow highlight paints the node body only, so the green staged-change outline and the selection border still show. It stays until you click off the node, press Esc, or run any zoom operation (mouse wheel, Zoom In/Out, Fit All, Reset Zoom, Unzoom, or Zoom to Area) — the idea is to highlight, zoom to the instance you want, and carry on editing.

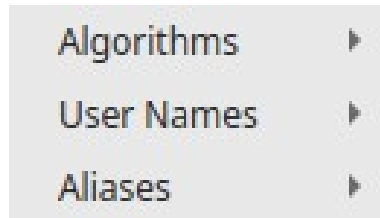


Figure: Search popup with its Algorithms, User Names and Aliases submenus.

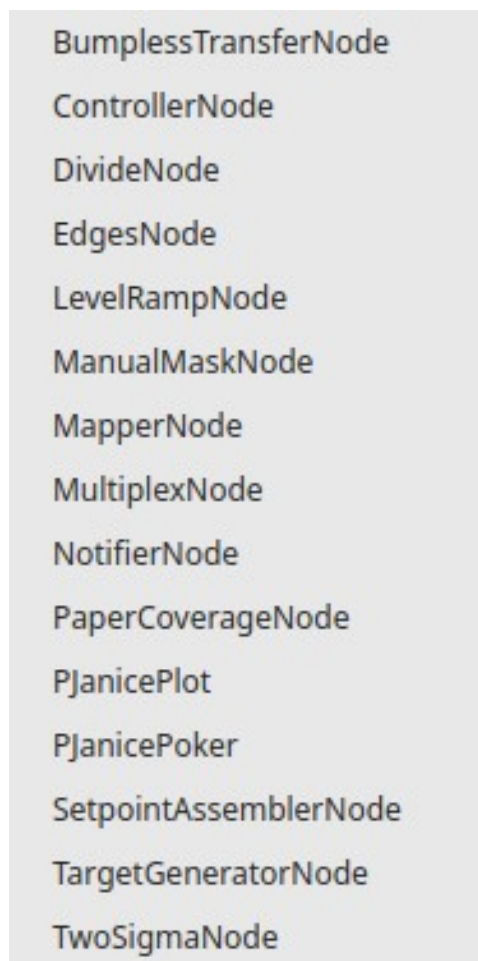


Figure: Algorithms submenu listing the algorithms used on the loaded drawing.

@BumplessTransferNode_da3734_out	dilution_mode
@ControllerNode_82c771_out	estimated_coeffs
@DivideNode_1a488e_out	fixed_mask
@EdgesNode_523690_out	frame_left
@EdgesNode_68b097_out	frame_right
@LevelRampNode_045d18_out	headbox_width_mm
@ManualMaskNode_b0789c_out	learned_count
@MapperNode_82b1e0_out	level
@MultiplexNode_02147f_out	level_ramp_step_pct
@NotifierNode_19ae10_out	mapped_2sig
@NotifierNode_67c339_out	mapped_profile_selected
@NotifierNode_cae1d2_out	mapper_headbox_center
@NotifierNode_ea186a_out	mapper_headbox_width
@NotifierNode_ed4d86_out	mapper_refined_width
@PaperCoverageNode_e7a752_out	mapping_matrix
@PjanicePlot_04c5e7_out	mapping_quality
@PjanicePlot_17ee90_out	No
@PjanicePlot_57ba05_out	num_actuators
@SetpointAssemblerNode_96c2ab_applied	num_databoxes
@TargetGeneratorNode_831ab7_out	Ordinal.AllowCLID
@TwoSigmaNode_eebb7a_out	paper_coverage
actuator_positions	paper_left
actuator_setpoints	paper_right
actuator_target_level	profile
actuator_width_mm	response_centers_norm
always_false	shrinkage_model_valid
always_true	startup_bumpless
applied_delta_u	target_profile
control_aggressiveness	transport_delay_scans
control_anchor_gain	use_simple_zone_control
control_auto_tune	Yes
control_edge_ignore_db	
control_edge_suppression_mult	
control_integral_gain	
control_max_delta_u	
control_Q	
control_R	
control_S	
control_T	
delta_u	
dilution_automanual	
dilution_automanual_request	
dilution_manual_setpoints	

Figure: Aliases submenu listing every alias referenced by the loaded drawing's nodes.

Highlight every node whose parameters use an alias matching the pattern.
 * = any text, ? = one character. Example: Ordinal.* finds every Ordinal-prefixed alias.

Figure: The alias search dialog opened from the Aliases submenu — search as picked, or edit the line into a wildcard pattern first.

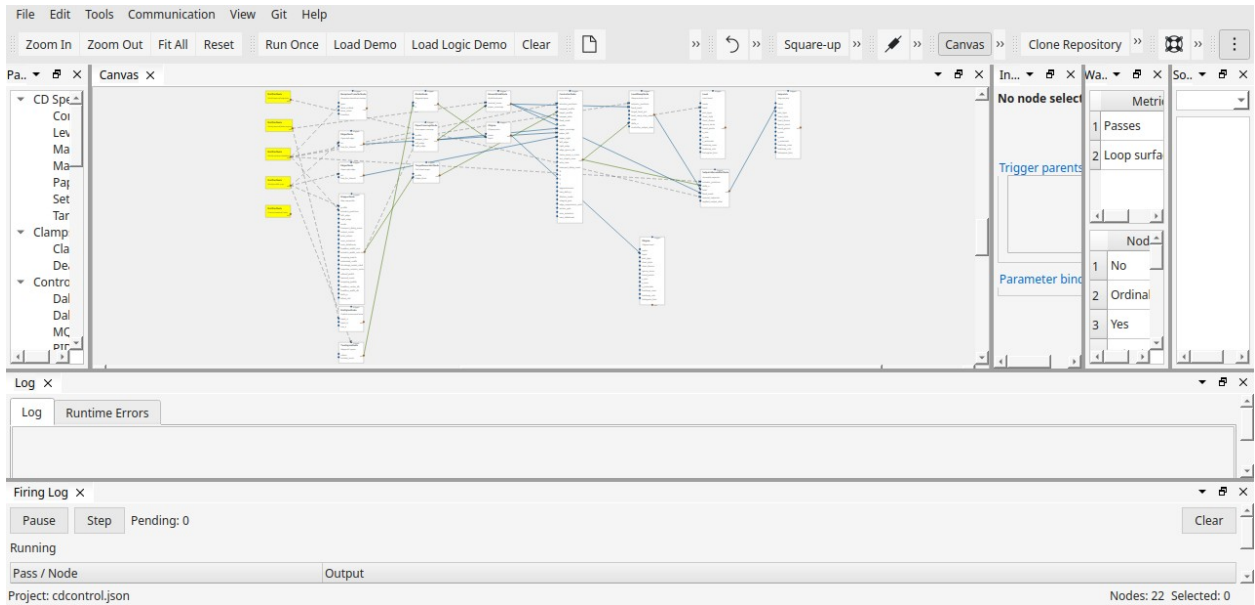


Figure: All NotifierNode instances highlighted in yellow on cdcontrol.json after picking NotifierNode in Search → Algorithms.

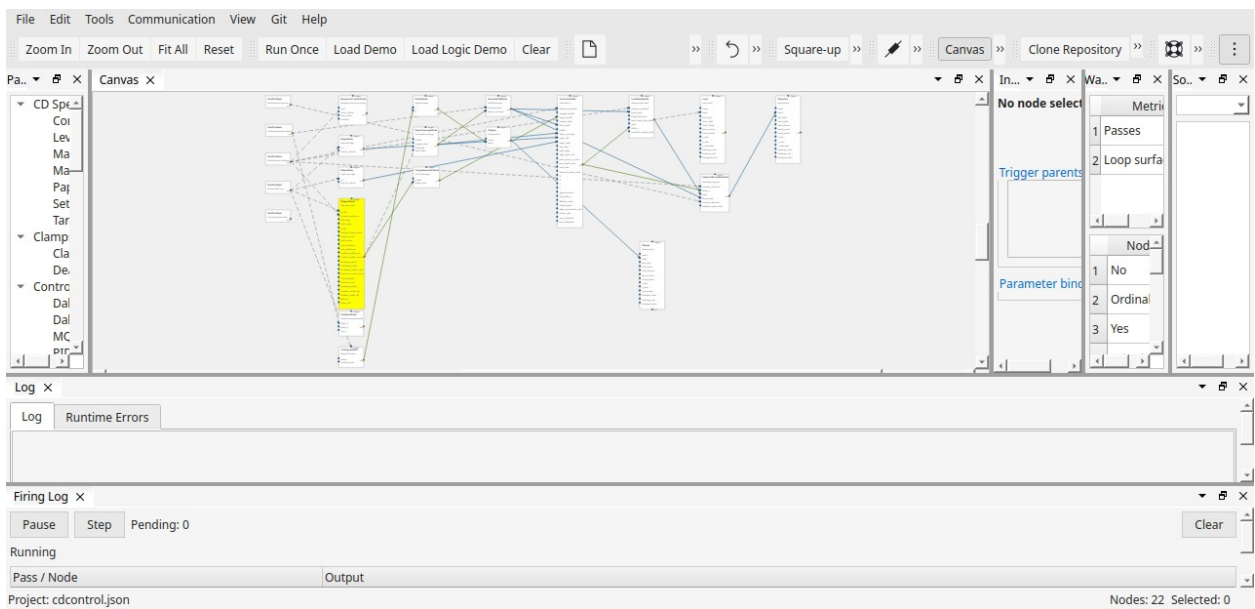


Figure: Alias search for the wildcard Ordinal.* on cdcontrol.json — the MapperNode (which binds Ordinal.AllowCLID) is highlighted.

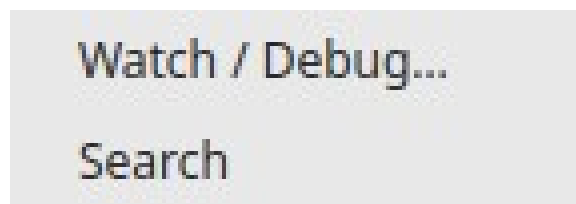


Figure: Alias Table right-click menu — Search sits below Watch / Debug... and jumps straight to the nodes using that alias.



Figure: Palette right-click menu — Search highlights every block on the canvas using that algorithm.

Alias Table

Edit → Alias Table... opens the non-modal alias cross-reference table. Each row maps a symbolic alias to a value source. The columns are:

- Alias — the symbolic name used in parameter bindings (e.g., `water_pv`, `enable`, `@pid_out`).
- Source Type — constant, preload, persistent, ordinal, external, or `node_output`. Each type means:
 - constant — a fixed value saved in the project. Use for geometry, gains, limits, or any value that should not change at runtime. The value lives in the Default column.
 - preload — a tunable value that starts from the saved Default but can be overridden at runtime (for example from a debug panel or a supervisory system). The override lasts until the engine stops.
 - persistent — a value backed by a file on disk. It starts from Default the first time, but every runtime change is saved and reloaded the next time the project starts.
 - ordinal — a global boolean flag used for control flow (e.g., `enable`, `startup_done`). Nodes can read and write ordinals, and a node can be triggered when an ordinal changes.
 - external — a live value from a bridge plugin such as `stdcomQtPS` or `MQTT`. Source Ref is the qualified topic/source name and Bridge selects which plugin owns it.
 - `node_output` — a published proxy for a node's internal output alias (e.g., `@pid_out`). Use this when a computed value must be written to an external Multiverse topic or reused by name elsewhere.
- Source Ref — for external aliases, the bridge source name; for constants, the literal value stored in Default.
- Bridge — the enabled bridge plugin that owns an external alias (e.g., `stdcomQtPS`).
- Default — the fallback value for constants/preloads/persistent aliases, or the resident/default value packet for external aliases.

You can populate an external row by dragging a source from the Sources dock (the plugin display). The dragged text arrives qualified as `bridgename://source`, so the Bridge cell is filled automatically and the Source Type becomes external. If the Alias cell is empty, it is also filled with the source name.

When you click OK in the Alias Table dialog, the app automatically creates subscription declarations for every external alias (direction sub) and for every `node_output` alias that

publishes a Multiverse topic (direction pub). This removes the manual declaration step for topics that the bridge plugin already knows about.

If you need to declare a topic that the plugin does not yet know — for example, a tag that will be owned and published by the Graphic Builder through the stdcomQtPS plugin — you must add that declaration manually in the Declarations dialog. It will not appear in the Sources dock until the plugin discovers it, so the automatic path cannot create it for you.

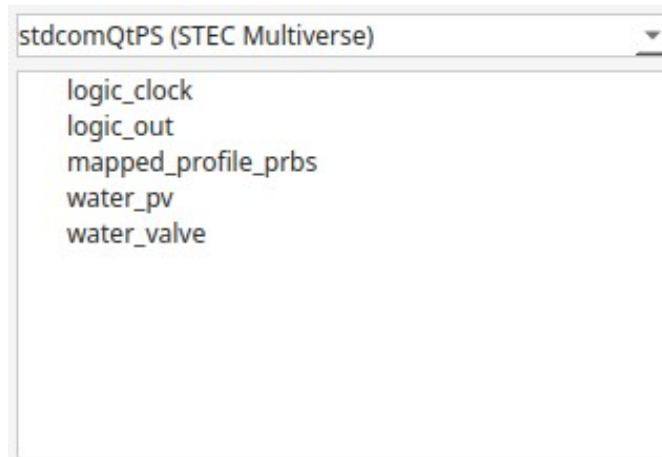


Figure: Sources dock showing live stdcomQtPS topics.

Add RowDelete RowConfigure External Source...

	Alias	Source Type	Source Ref	Bridge	Default
1	No	constant			False
2	Ordinal.AllowC...	persistent			False
3	Yes	constant			True
4	actuator_posit...	external	dilutions_posit...	stdcomQtPS	
5	actuator_setp...	external	dilutions_setp...	stdcomQtPS	
6	actuator_targe...	constant			50.0
7	actuator_widt...	constant			50.0
8	always_false	constant			False
9	always_true	constant			True
10	applied_delta_u	node_output	@SetpointAss...		[]
11	control_Q	preload			1.0
12	control_R	preload			0.0592
13	control_S	preload			0.0592

Node outputs (internal, not editable here):
@BumplessTransferNode_da3734_out
@ControllerNode_82c771_out
@DivideNode_1a488e_out
@EdgesNode_523690_out
@EdgesNode_68b007_out

OKCancel

Figure: Alias Table dialog with external, constant, and node_output rows.

Algorithm Builder

Edit → Algorithm Builder... is a separate scaffold tool that writes a new algorithm template for you. When the dialog opens, fill in the metadata and contract:

- Class name — the Python class name (must be a valid identifier, e.g., MyAverageNode). This becomes the node type shown in the Palette.
- Description — a short human-readable sentence explaining what the node does.
- Group — the Palette category the node appears under (e.g., Math, Controllers, Filter).
- Output alias param — optional contract item name. When set, the engine also publishes the run() result to the alias bound to that parameter, so the node can write to an external topic.

Below the metadata are three contract sections. Each contract item has Name, Type (float/int/bool/string for values), Required (True/False), Default value, and Description:

- Value inputs — data the node reads from upstream wires or aliases.
- Ordinals — boolean flags the node can read or write.
- Subscriptions — external bridge topics the node consumes directly.

Click Preview at any time to see the generated Python skeleton. When you click OK, the tool:

- Saves the file as `graphic_builder/algorithms/<lowercase_class_name>.py`. This is the required location; the Palette only scans modules inside `graphic_builder/algorithms/`.
- Adds an import line to `graphic_builder/algorithms/__init__.py` so the module is loaded on startup.
- Registers the new class immediately, so it appears in the Palette right away without restarting the application.
- Regenerates `algohelp.html` automatically so the new algorithm appears in Help → Algorithm Help and in F1 context help.

The generated file contains a class decorated with `@REGISTRY.decorator()`, a CONTRACT list, and a stub `run()` method. Open the file in an editor, replace the ... in `run()` with your logic, and save. If you later change the contract, the description, or the display group, run Tools → Rebuild Algorithm Help again to refresh the generated help page.

Class name:

Description:

Group:

Output alias param:

Value inputs

values	type	boolean	default value	
values	float	True	Default value	values to average

Ordinals

Preview:

```

"""Compute the average of input values."""

from graphic_builder.model import Context, Node, REGISTRY, ordinal, subscription, value
from typing import Any

@REGISTRY.decorator()
class MyAverageNode(Node):

```

Figure: Algorithm Builder dialog showing metadata, contract inputs, and generated preview.

Declarations

Edit → Declarations... opens the strict declaration manager. Ordinals are declared automatically from alias-table rows whose Source Type is ordinal; subscription declarations are created automatically when the Alias Table dialog is accepted, and can also be imported or edited here.

Each subscription declaration has four fields:

- Name — the fully-qualified topic, usually bridgename://source (for example, stdcomQtPS://water_pv).
- Direction — sub (read from the bridge), pub (write to the bridge), or subpub (both).
- Value type — float, int, bool, or string; used by the engine to coerce wire values.
- Description — optional human-readable note.

Use the Import from Alias Table button to refresh declarations after changing aliases, or Add to enter a topic that does not yet exist in the plugin. The Problems dialog flags any alias that references an undeclared topic.

Declare every external subscription topic used by the graph. Aliases that reference undeclared external topics are rejected in strict mode.

	Name	Direction	Value Type	Description
1	stdcomQtPS://reel_btw	sub	float	
2	stdcomQtPS://dilutions_positions	sub	float	
3	stdcomQtPS://dilution_automanual_request	sub	float	
4	stdcomQtPS://dilution_manual_setpoints	sub	float	
5	stdcomQtPS://dilutions_setpoints	subpub	float	
6	stdcomQtPS://dilution_automanual	subpub	float	
7	stdcomQtPS://mapped2sigma	subpub	float	

Add Edit Remove
Import from Alias Table
Close Apply

Figure: Declarations dialog with subscription declarations.

Preferences

Edit → Preferences... opens the settings dialog. The General tab covers theme, help browser, enabled bridges, enabled utility plugins, default wire routing (Direct or XY), and the Project/Grade/Persistent folders. The Shortcuts tab lets you assign or change key sequences for every registered action. Preferences are saved to the application INI file and persist across sessions. The full Preferences & Shortcuts chapter below expands every setting.

View

- Show/hide each dock: Canvas, Palette, Inspector, Watch, Sources, Log, Firing Log.
- Problems — design-time diagnostics and runtime warnings.
- Save Layout / Reset Layout — dock arrangement persistence.
- Zoom In / Out / Fit All / Reset Zoom — canvas view controls.
- Toolbars submenu — toggle the per-menu toolbars.

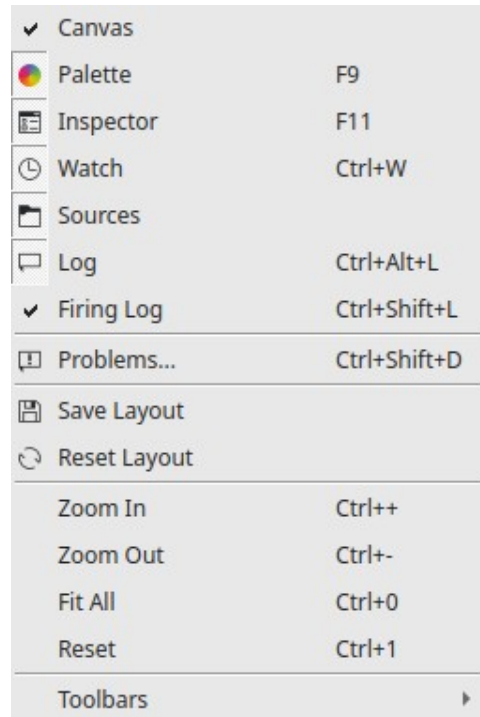


Figure: View menu.

The View menu toggles the dockable panels and canvas view commands. Each panel has its own detailed section later in this manual; the list below shows where to find the full explanation.

- Canvas — see Canvas & Interaction.
- Palette — see Nodes & Algorithms → Palette.
- Inspector — see Canvas & Interaction → The Inspector panel.
- Watch — see Runtime & Debugging → Watch panel.
- Sources — see Communication & Bridges → Source Browser.
- Log — see Runtime & Debugging → Log panel and Runtime Errors.
- Firing Log — see Runtime & Debugging → Firing Log.
- Problems — see Runtime & Debugging → Problems dialog.
- Save Layout / Reset Layout — save or restore the dock layout.
- Zoom In / Zoom Out / Fit All / Reset Zoom — see Canvas & Interaction → Zoom, pan, and fit.
- Toolbars — show or hide the per-menu toolbars.

Tools

- Square-up — auto-arrange nodes and re-route wires to Lucidchart-style standards.
- Rebuild Algorithm Help / Open Algorithm Help — regenerate or view the algorithm reference.
- Utility plugins (when enabled) — Import automap4 Grade, Import / Merge Drawing, etc.

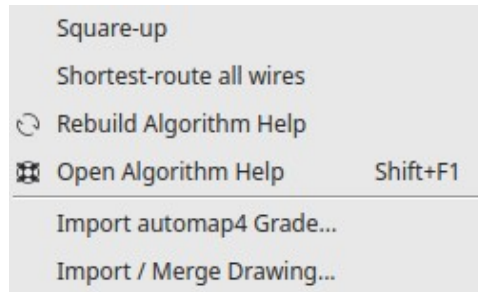


Figure: Tools menu.

Square-up

Tools → Square-up automatically arranges the current drawing and re-routes every wire to the Lucidchart-style standard. It is the fastest way to clean up a messy or hand-placed graph.

Square-up performs four steps:

- Column assignment — nodes are placed in columns by topological depth (the longest path from any source node). Nodes with no incoming edges form the first column; nodes that depend only on the first column form the second, and so on.
- Row stacking — within each column, nodes are sorted by id and stacked using their real height plus a small gap. Tall debug nodes therefore do not overlap the next row, which was a problem with the old fixed-height stride.
- Port layout — the output port moves to the right edge, value inputs move to the left edge, and the trigger port stays on top. This makes left-to-right data flow the default.
- Orthogonal routing with obstacle avoidance — every wire is re-routed as horizontal and vertical segments. Long wires pass through channels above or below intermediate nodes so no line runs behind a node. A lane registry keeps parallel segments offset from each other (± 12 px) so they do not overlap; shared fan-out stubs are exempt so connection points remain visible.

Because square-up changes every node position and every wire, the entire drawing is marked as staged (green outlines and wires) until you save. The example below shows the same cdcontrol.json graph before and after square-up.

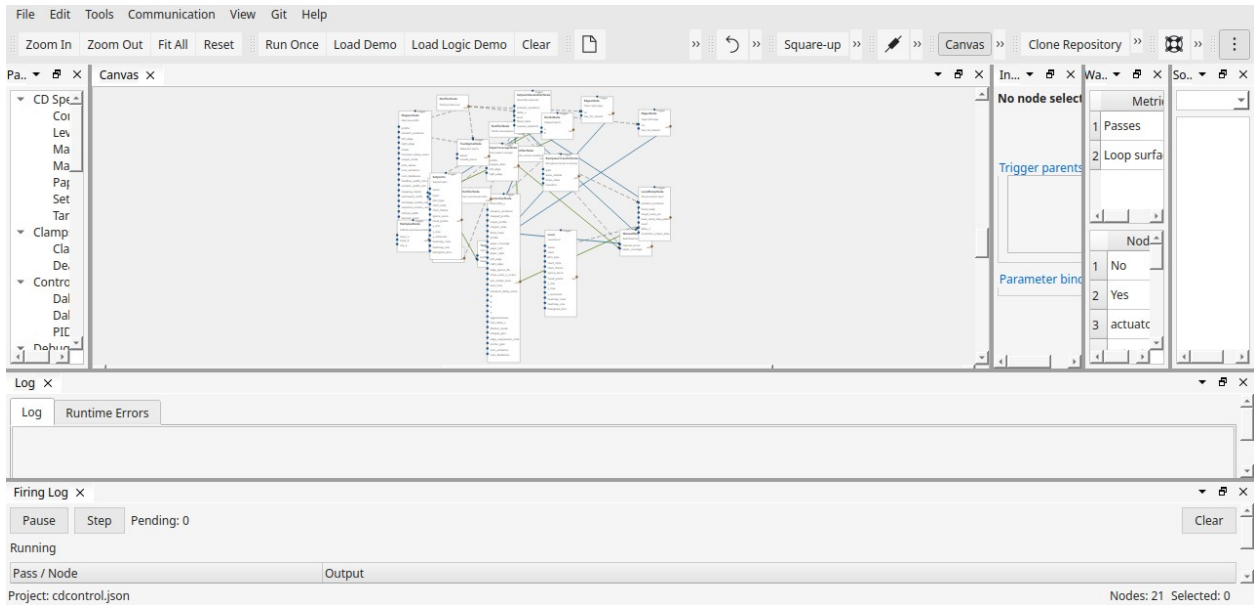


Figure: cdcontrol.json after random manual placement — nodes overlap and wires cross.

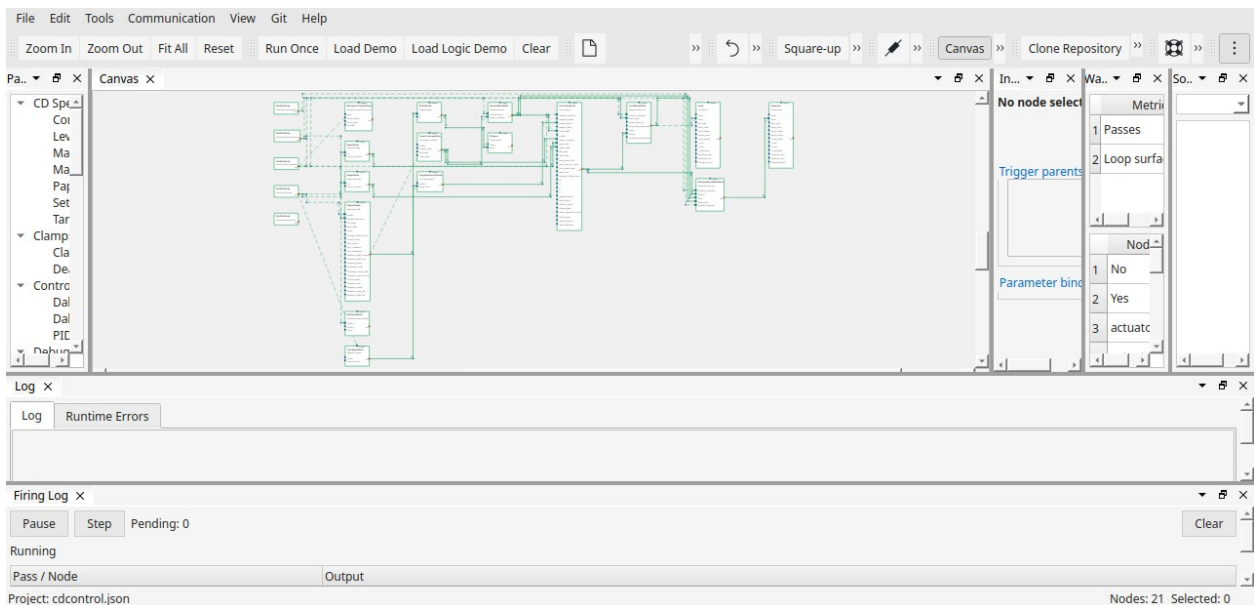


Figure: cdcontrol.json after Tools → Square-up — nodes in columns, wires orthogonal and non-overlapping.

Rebuild Algorithm Help

Tools → Rebuild Algorithm Help regenerates graphic_builder/algorithmhelp.html from the metadata of every algorithm registered in the Palette. Run this after adding a new algorithm, renaming one, or changing its contract (inputs, outputs, parameters) so that F1 context help and Help → Algorithm Help stay current.

Open Algorithm Help

Tools → Open Algorithm Help opens the generated algohelp.html in the help viewer or external browser (depending on the Use system browser for help preference). This page lists every registered algorithm, its description, and its full contract table. It is the fastest way to browse the complete algorithm reference.

For help on one specific algorithm, click a node on the canvas to select it and press F1. The help viewer jumps directly to that algorithm's section in algohelp.html. If the node is not selected, hover over it and press F1; the help system resolves the node under the cursor and opens the same algorithm-specific anchor. You do not need to open the full Algorithm Help index first.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	actuator_positions	float	yes		Actuator positions (NumberOfZones elements)
value	mapped_profile	float	yes		Mapped profile (NumberOfZones elements)
value	target_profile	float	yes		Target profile (NumberOfZones elements)
value	mapper_alias	string	yes		Output alias of the MapperNode that owns the mapping matrix
value	fixed_mask	float	no	[]	Fixed/manual/off-sheet mask (NumberOfZones elements)
value	profile	float	no	[]	Raw databox profile (0 = no paper). Gives the paper mask for the gain matrix and the paper extent for the error taper
value	paper_coverage	float	no	[]	Per-zone paper coverage 0..1 (PaperCoverageNode); zones with 0 get error weight 0 and measurement = target
value	paper_left	float	no		First non-zero databox (EdgesNode low); anchors the 5-box error taper. None = from profile
value	paper_right	float	no		Last non-zero databox (EdgesNode high); anchors the 5-box error taper. None = from profile
value	left_edge	float	no	0.0	Mapper ruler left end in databoxes (edge-free frame: 0)
value	right_edge	float	no		Mapper ruler right end in databoxes (edge-free frame: NumberOfDataboxes-1)
value	edges_inner_th	float	no	25.0	Edges band in databoxes over which the move cost ramps up to edge suppression multi

Figure: Algorithm Help viewer opened for a single node (ControllerNode) via F1.

Utility plugins

The bottom of the Tools menu contains any utility plugins enabled in Preferences. Unlike bridge plugins, utilities run on demand and do not participate in live data transport.

Import automap4 Grade

Tools → Import automap4 Grade... reads an automap4 grade .npz file (the saved result of S-curve discovery, PRBS, and tuning) and imports its machine/tuning data into the alias table as

alias rows. After you choose the file, a preview dialog shows every row that will be imported with five columns: Import (checked by default), Alias, Type, Value, and Status (New or Replace).

The grade keys are mapped to alias source types as follows:

- Constant aliases — fixed machine geometry that does not change for a given machine: mapper_headbox_center, mapper_headbox_width, headbox_width_mm, actuator_width_mm, and paper_length_m.
- Preload aliases — tunable control and mapping scalars that you may still adjust at runtime: control_Q, control_R, control_S, control_T, control_aggressiveness, control_max_delta_u, control_max_adj_diff, control_integral_gain, control_integral_decay, control_integral_max, control_edge_suppression_mult, control_anchor_gain, control_estimated_gain, control_estimated_response_width, control_estimated_tau, control_estimated_tc, control_horizon, mapping_progress_threshold, and estimated_coeffs (the S-curve cubic). Also from the grade's preferences snapshot: control_edge_ignore_db, actuator_target_level, level_ramp_step_pct, use_simple_zone_control, dilution_mode, and refine_from_control.
- Persistent alias — the mapping_matrix is imported as a persistent alias so it is file-backed and survives application restart.

You can change the Type drop-down for any row before importing, for example to make a normally fixed geometry value adjustable (preload) or to make a tunable value fixed (constant). Click OK to upsert the rows into the alias table. Existing aliases are replaced; node_output aliases with the same name are skipped so the running graph is not corrupted. Re-running the import on the same grade replaces the values again, which is useful after re-tuning in automap4.

Import	Alias	Type	Value	Status
☒	actuator_target_level	preload	50.0	New
☒	actuator_width_mm	constant	50.0	New
☒	control_Q	preload	1.0	New
☒	control_R	preload	0.0592	New
☒	control_S	preload	0.0592	New
☒	control_T	preload	0.0592	New
☒	control_aggressiveness	preload	0.7	New
☒	control_anchor_gain	preload	0.0	New
☒	control_edge_ignore_db	preload	25	New
☒	control_edge_suppression_mult	preload	3.0	New
☒	control_estimated_gain	preload	11.178200760041864	New
☒	control_estimated_response_w...	preload	4.45945690199613	New
☒	control_estimated_tau	preload	2.0	New

Existing aliases are replaced (a node_output alias with the same name is skipped). New aliases are added to the table.

OK Cancel

Figure: Import automap4 Grade dialog showing constant and preload alias rows from a grade.

Import / Merge Drawing

Tools → Import / Merge Drawing... copies one or more BuilderRunner2 project JSON files into the currently open drawing. The current drawing is the base; each selected file is merged in order. This is a one-time snapshot, not a live link — changes to the merged result do not affect the original files, and fixes to an original do not flow back.

The merge performs four steps:

- Node ids — ids that already exist in the base are prefixed with the source file stem (for example, cdcontrol_MapperNode). Unique ids are kept as-is so cross-references stay readable.
- Declarations — ordinal and subscription declarations are unioned by qualified name. Later sources override earlier ones on conflict.
- Alias table — external, subscription, and node_output aliases are shared Multiverse topics; on conflict the later source wins. Internal alias conflicts are resolved in one of two modes: override (seed mode, later source replaces the base) or module mode (the later alias is renamed with the source prefix and every reference inside that source is rewritten).
- Geometry — node positions and edge routing from each source are appended so the merged drawing can be laid out and squared-up as a whole.

A second tab in the same dialog exports an automap4 grade .npz as a minimal mapping-seed drawing: it creates aliases for the grade's tuning and mapping constants but no nodes, so the seed can be merged into a generic control chain.

Merge example

The sequence below uses the test drawings drawingtest1.json and drawingtest2.json. Both drawings have the same internal node ids (notifier, mean/divide, out_publisher) and the same internal constant alias name (scale), but different external topics (topic1 vs topic2) and different output aliases (out1_alias vs out2_alias).

Step 1 — start with File → New Project or File → Clear so the canvas is empty.

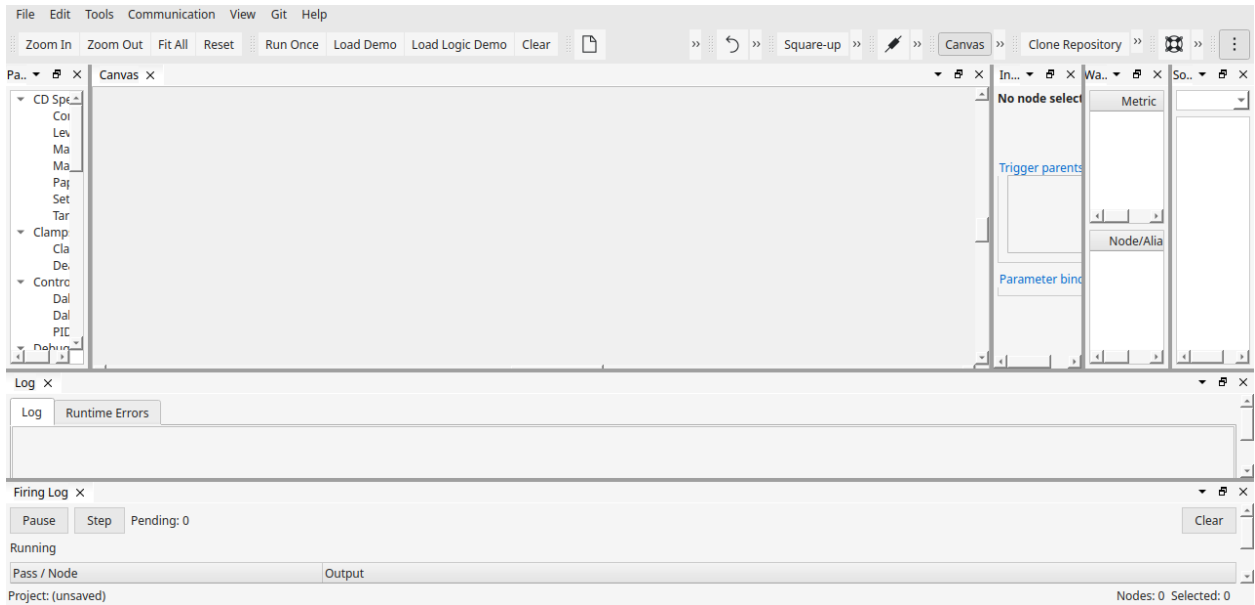


Figure: Empty canvas before any merge.

Step 2 — choose Tools → Import / Merge Drawing..., select drawingtest1.json, and import it in seed/override mode. The three nodes from drawingtest1 appear: notifier → MeanNonZeroNode → out_publisher. The alias table now contains topic1, scale=10, and out1_alias.

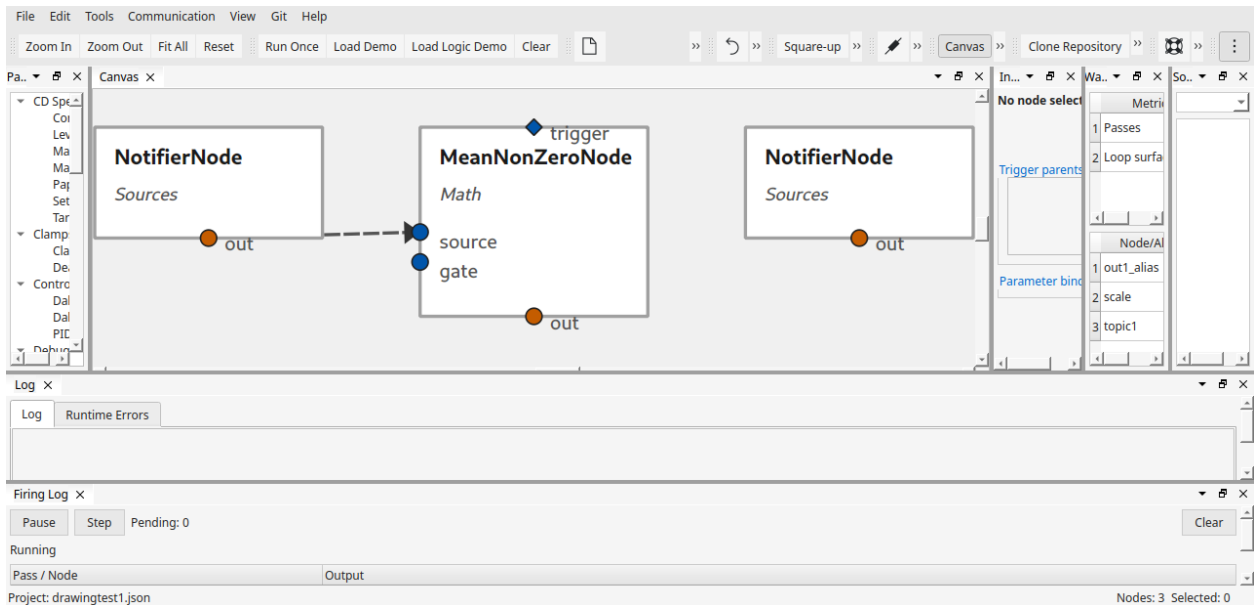


Figure: Canvas after merging drawingtest1.json.

Step 3 — run Import / Merge Drawing... again and add drawingtest2.json. The merge prefixes the colliding node ids from drawingtest2 with drawingtest2_ (so notifier becomes drawingtest2_notifier, and so on). The external topics topic1 and topic2 coexist, as do the published outputs out1_alias and out2_alias. Because both drawings define an internal constant

named scale, the later source wins in override/seed mode and scale becomes 100; in module mode it would be renamed drawingtest2_scale. The result has six nodes and both data chains on the same canvas.

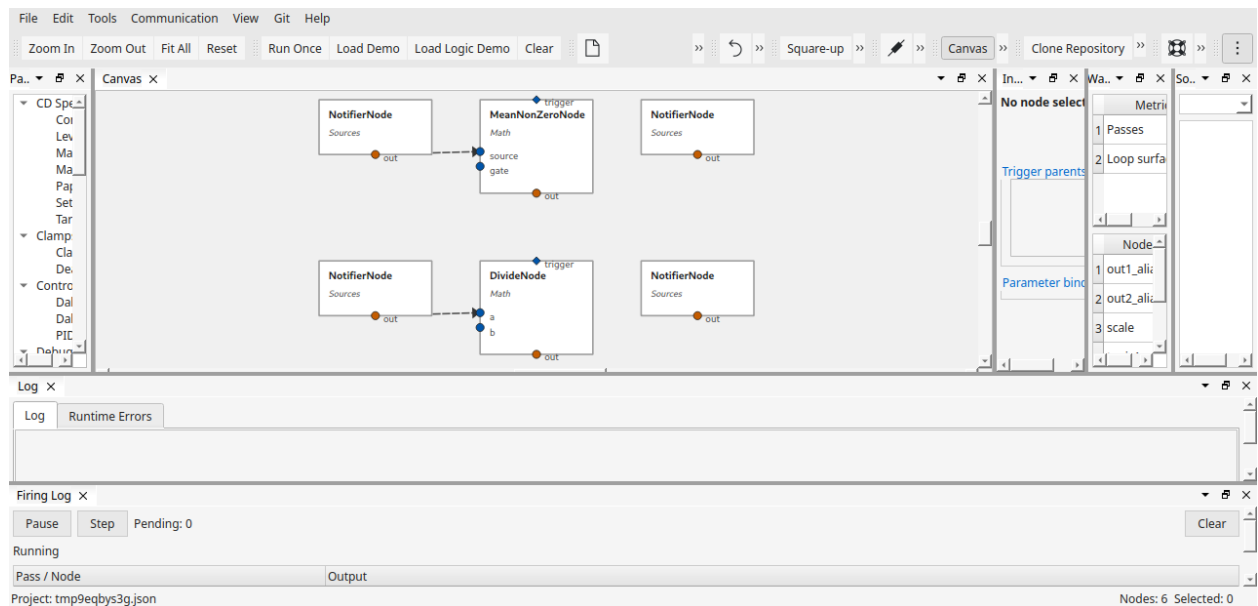


Figure: Canvas after merging drawingtest2.json on top of drawingtest1.json.

Communication

- Bridges — enable, configure, and connect communication bridge plugins.
- Connect / Disconnect — all enabled bridges at once.



Figure: Communication menu.

The Communication menu contains Bridges..., Connect, and Disconnect. See the Communication & Bridges chapter for the full explanation of the Bridges manager, the Source Browser, how bridge plugins work, and how to build a new bridge plugin.

Git

- Clone / Open Repository — get an existing grade project under version control.
- Commit / Pull / Push / Fetch — basic distributed version-control workflow.

- New Branch / Tags / Stashes / Merge / Rebase / Branches — branch and history management.
- Show Log / Show Diff / Status — inspect repository state.

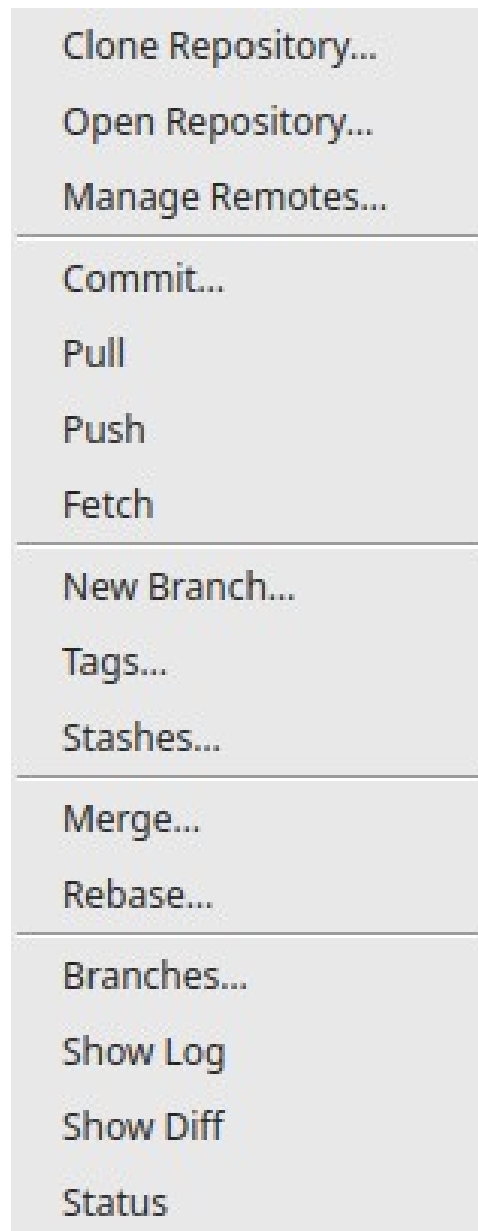


Figure: Git menu.

Git menu

The Git menu is a top-level plugin menu for version-controlling grade projects. It shells out to the system git command, so the usual Git installation on your machine is used. All operations target the directory of the currently loaded project JSON.

Basic workflow items:

- Clone Repository... — clone a remote grade repository into the Grade/Clone Folder.
- Open Repository... — open an existing local Git repository in the Graphic Builder.
- Manage Remotes... — add, remove, or inspect remote URLs.
- Commit... — stage modified project files and write a commit message.
- Pull — fetch and merge changes from the current branch's remote.
- Push — send local commits to the current branch's remote.
- Fetch — download remote refs without merging.

Branch and history items:

- New Branch... — create a branch, optionally starting from another branch or commit.
- Branches... — list, switch, and delete branches.
- Tags... — create and delete annotated or lightweight tags.
- Stashes... — save and restore unfinished changes.
- Merge... / Rebase... — integrate another branch; the dialog shows the exact git command and warns that conflicts must be resolved outside the Graphic Builder.
- Show Log — view recent commits.
- Show Diff — view uncommitted changes.
- Status — quick summary of modified/staged/untracked files.

Git is a powerful distributed version-control system. The menu covers the most common operations, but it is not a complete Git client. For advanced tasks — interactive rebase, conflict resolution, reflog, submodules, hooks, etc. — use the command line or a dedicated Git tool. If you are new to Git, start with the official documentation:

- Git official book and reference: <https://git-scm.com/doc>
- Pro Git book (free): <https://git-scm.com/book>
- GitHub Git cheat sheet: <https://education.github.com/git-cheat-sheet-education.pdf>

The menu plugin uses your existing SSH keys and git configuration, so authentication works the same way as running git from a terminal.

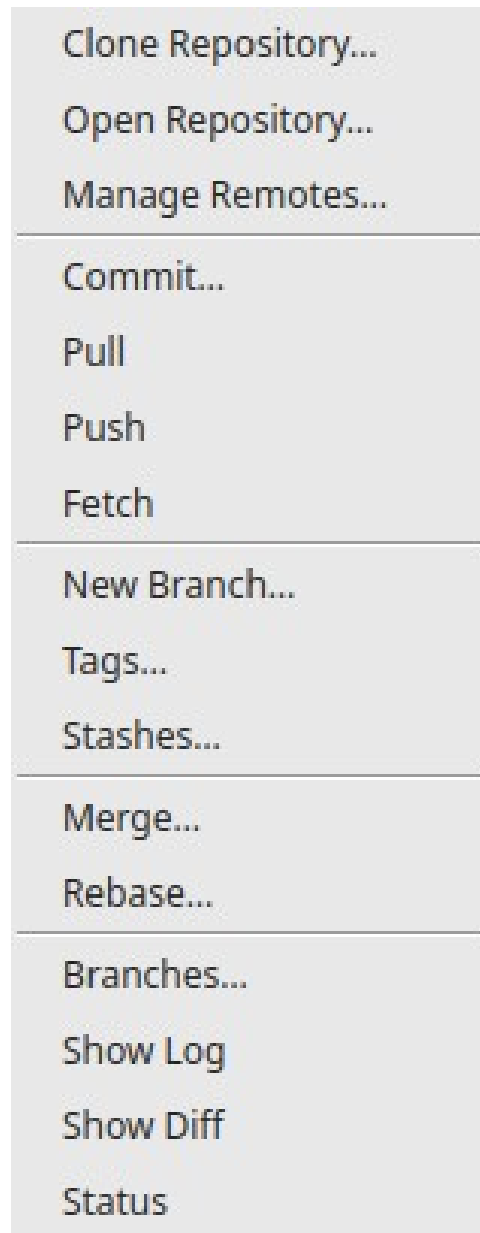


Figure: Git top-level menu.

Help

- Graphic Builder Help — this help document.
- Algorithm Help — generated reference for every registered algorithm.
- How to Build a Bridge — plugin developer guide.
- Rebuild Algorithm Help — refresh algohelp.html.

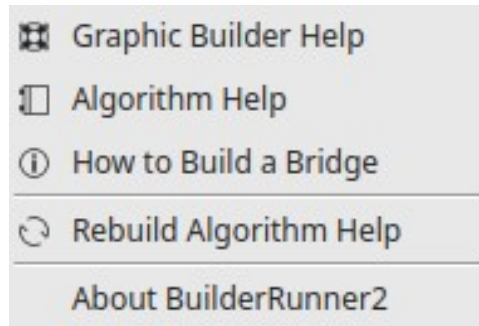


Figure: Help menu.

Run toolbar

- Run Once (F5) — execute the current graph one time.
- Load Demo — load a sample DDC control loop.
- Load Logic Demo — load a sample AndNode with source-bound inputs.
- Clear — remove all nodes and edges from the canvas.

Canvas & Interaction

The canvas is the node-graph workspace. Nodes are boxes, wires are arrows, and the view can be panned and zoomed without changing logical positions.

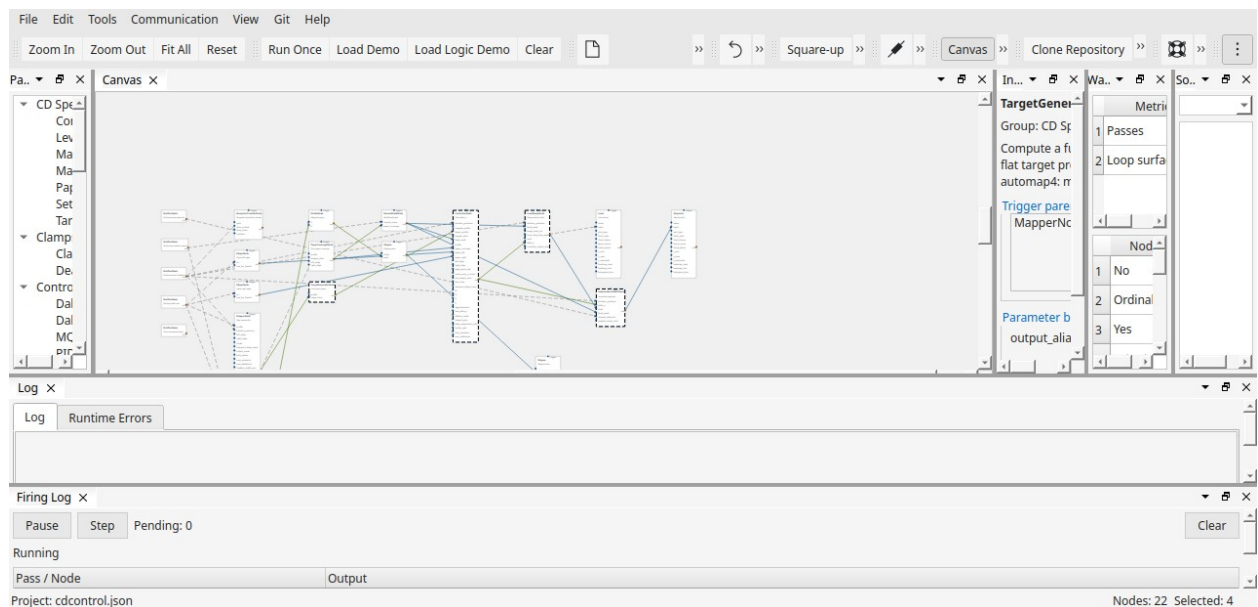


Figure 2: Canvas with several nodes selected.

Zoom, pan, and fit

- Magnifies the canvas view so nodes and wires appear larger. Logical node positions do not change; only the view transform is affected.

- Shrinks the canvas view so more of the graph is visible at once. Logical node positions do not change.
- Adjusts the canvas view so that all existing nodes and edges fit inside the current window.
- Restores the canvas view to its default 1:1 scale.
- Move the canvas without changing the zoom level:
 - Middle mouse drag — pan anywhere on the canvas.

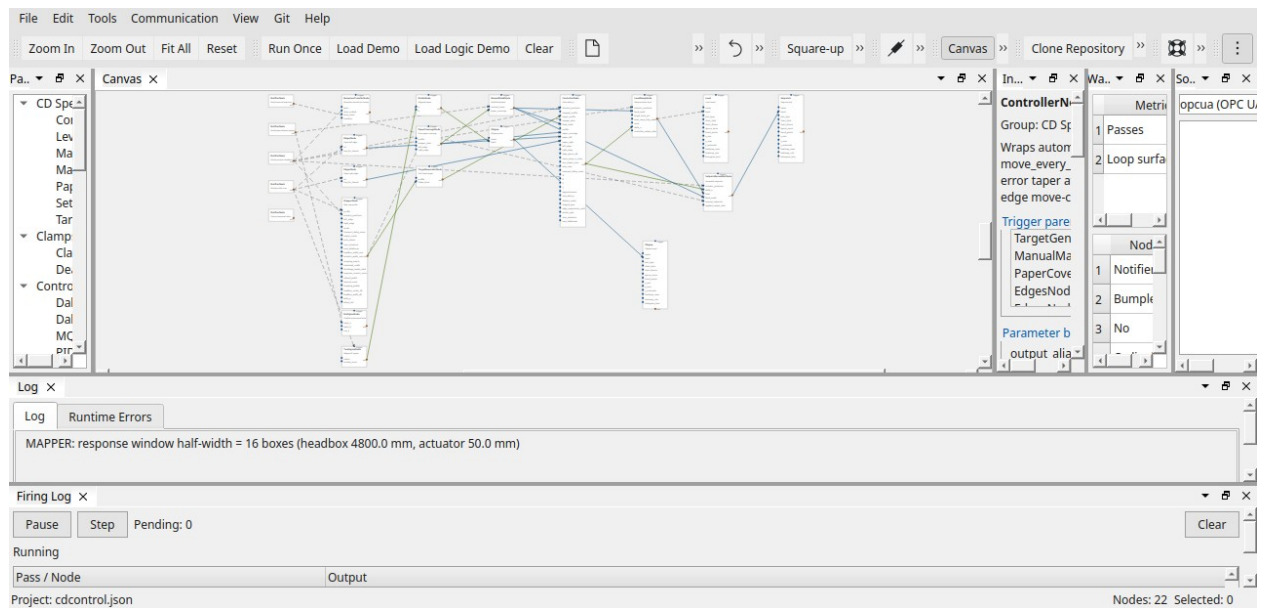


Figure 3: Canvas after Fit All.

Selecting, placing, and wiring nodes

- A visual block that represents one algorithm. Every node is a single concrete subclass of the abstract Node class, with a description, group, toolbar image, and a declared contract of inputs/outputs.
- Right-click a node to open its context menu. Choose Delete to remove the node and its connected edges.
- A node outline drawn in green means the node is staged — added or changed since the last save (like files staged in git but not yet committed). The same applies to green wires. File → Save or Save As... commits the drawing and returns everything to its normal colors.
- Lists all discovered algorithms grouped by their declared category. Click an algorithm to stage it: a translucent ghost of the node follows the mouse, and the node is created exactly where you click on the canvas. Right-click or press Esc to cancel a staged placement.
- Small circles on the top (inputs) and bottom (output) of a node. Drag from an output port to an input port to connect two nodes. Ports are typed according to the node's contract.
- A line connecting an output port to an input port. Edges represent data flow from child to parent in the upward-running block string.
- Every node that produces a value has an output port on its bottom edge. By default the output is wired internally as @<nodeid>_out and downstream wires use that internal name.

Blocks, ports, and trigger semantics

A block (node) is one algorithm instance. It has a trigger port, one or more value input ports, and one output port. The trigger port is the diamond at the top; the value inputs are the small circles on the left (or on another side if you changed the port layout); the output is the circle at the bottom.

There are three ways a block receives data and execution:

- Source only — a value input is bound to an alias (or to a parent block's output alias), but the block does NOT run when that value changes. The wire is solid and usually drawn in a light blue color. Use this when the block is triggered by something else but still needs the current value as a parameter.
- Source + trigger — a value input is bound to an alias (or to a parent block's output alias) AND the block runs whenever that value changes. The wire is solid and usually drawn in a light green color. This is the default when you drop a wire on a value input.
- Trigger only — a wire lands on the top trigger port and only wakes the block; no value is transferred. The wire is dashed. Use this to chain execution from a notifier or from another block whose output you do not need as a value.

Hold Shift while dropping a wire on a value input to force source-only mode. Otherwise a small popup appears with two buttons: Bind & trigger (default) or Bind only. You can also right-click an existing wire and choose Switch to source only / Switch to trigger + source.

Common wiring examples

The examples below show how the same logical idea can be wired in different ways.

- Src input from alias: in the Inspector, type the alias name directly into a value parameter. The block reads the alias value when it runs. Check the Src Also box if you want the alias itself to wake the block when it changes.
- Src input from parent: drag a wire from a parent block's output port to a child value input. The child reads the parent's output alias. This is the same as typing the alias, but the canvas makes the dependency visible.
- Src input + trigger from parent: the default drop on a value input. The child runs whenever the parent runs and receives the parent's output value. Most chains look like this.
- Trigger from parent: drag a wire from a parent output to the child's top trigger diamond. The child runs when the parent runs but reads its parameter values from the alias table, not from the parent's output.
- Special trigger input — NotifierNode: a NotifierNode has a subscription parameter bound to an external alias. When the alias updates, the NotifierNode fires and passes the value through its output. Wire that output (source + trigger) to the first block of your chain, or wire the trigger port only to wake a block without binding the value.

The Inspector panel

Click any block to open it in the Inspector dock. The Inspector shows the algorithm name, group, description, and all parameter bindings.

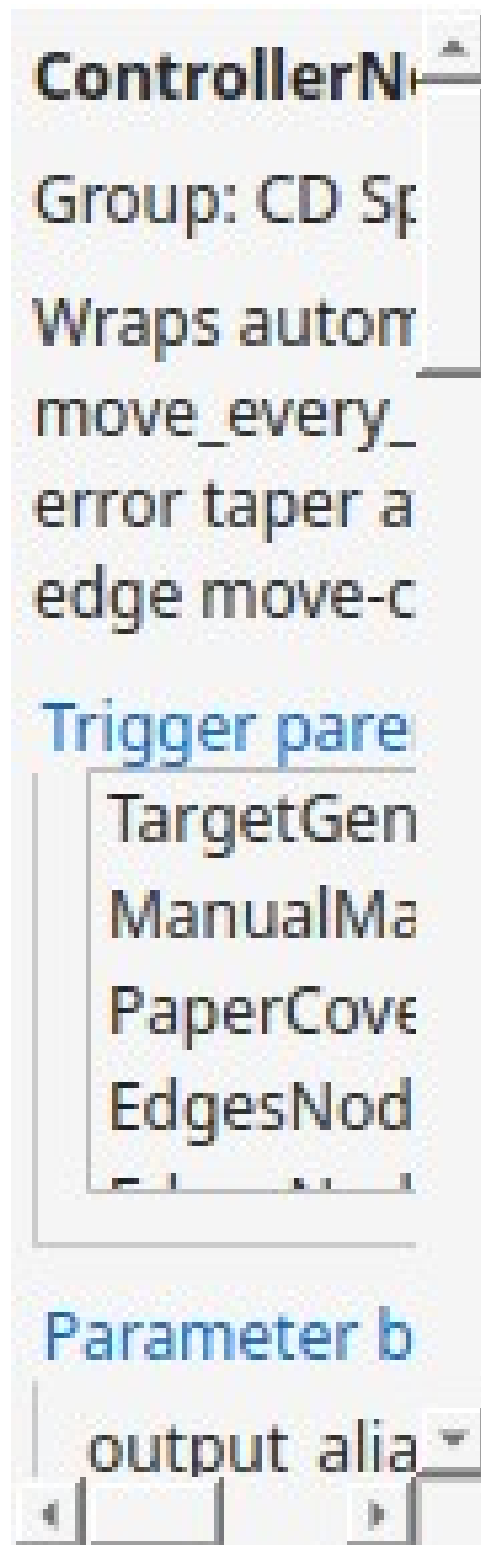


Figure: Inspector panel with the ControllerNode selected.

Each value parameter has a text box. You can:

- Type an alias name directly. Internal aliases like '@node_id_out' are accepted. String parameters are shown unquoted but saved as proper Python string literals automatically.
- Click the '...' button to open the alias picker and choose from the current alias table.
- Drag and drop an alias name from the Sources dock or the Alias Table dialog into the text box.
- Leave the field empty to use the algorithm's default value, if one exists.

For each value input there is a Src Also checkbox. Checked means the bound source wakes the block when it changes; unchecked means the source only supplies a value and the block must be triggered elsewhere. The output alias has an Internal alias checkbox that publishes to '@node_id_out' instead of a custom alias.

The Trigger parents box at the top of the Inspector lists every upstream block whose wire lands on a trigger port or on a Src-Also value input. If a non-source block has no trigger parents, it will not run.

Changes in the Inspector are applied immediately (the block turns green to show it is staged). Use Edit → Undo if you bind the wrong alias or check the wrong Src Also box.

Snapshots of live values

While commissioning, you often want to capture a live value and use it as a starting point. The PJanicePoker debug probe has a Snapshot input button in its companion widget: click it to copy the current input into the Next output editor. You can then edit the value and apply it as a one-shot injection. This is useful for replaying a corner case without retyping a long vector.

For any block, the Watch panel shows the latest output, and the Firing Log records the resolved inputs and outputs of every pass. Both are effectively live snapshots of the running graph.

Square-up and staged indicators

- Re-lays-out the whole drawing to Lucidchart-style standards: nodes are arranged left-to-right in columns by signal flow (sources on the left), each column stacked using the real node heights so nothing overlaps, with inputs on the left side and the output on the right side of every node. Every wire is then re-routed as an XY (orthogonal) elbow line that leaves its output connection point and enters its input connection point at a right angle, routes around any intermediate nodes (including trigger wires into top ports), and is offset into its own lane so no two parallel segments sit on top of each other. Wires that share one connection point fan out at staggered right-angle corners so the connection point stays visible.
- Green node outlines and wires indicate staged changes since the last save. File → Save or Save As commits the drawing and clears the highlights.

Nodes & Algorithms

A node is one concrete algorithm with a declared contract of value inputs, ordinals, subscriptions, and outputs. The Palette groups algorithms by category. See [Canvas & Interaction](#) for how to wire nodes and edit their bindings in the Inspector; see [Available Algorithms](#) and the [Algorithm Reference](#) appendix for the full catalog.

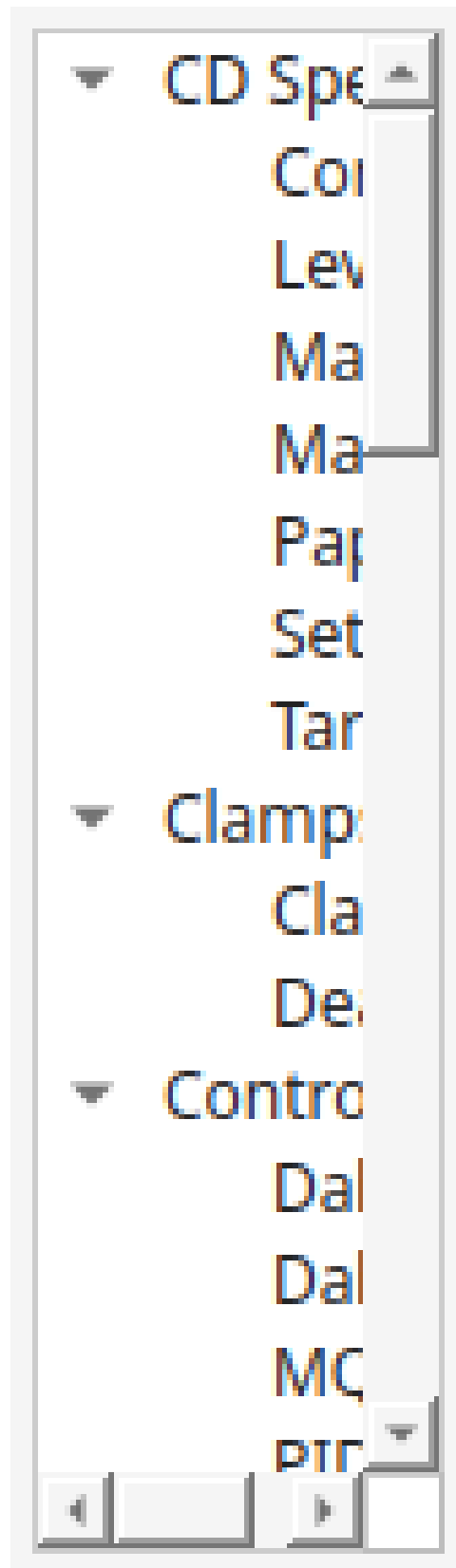


Figure: Palette dock showing algorithm groups.

Palette

Lists all discovered algorithms grouped by their declared category. Click an algorithm to stage it: a translucent ghost of the node follows the mouse, and the node is created exactly where you click on the canvas. Right-click or press Esc to cancel a staged placement.

Published output aliases

- Every node that produces a value has an output port on its bottom edge. By default the output is wired internally as @<nodeid>_out and downstream wires use that internal name.
- Right-click the output port to:
 - Publish as alias... — create a human-readable alias (e.g. my_pid) in the alias table. Existing and new downstream wires immediately switch to the published name.

Algorithm help from the canvas

Every node type has a generated help page in algohelp.html. To view it, select the node and press F1, or hover over the node and press F1. The in-application help viewer (or external browser, depending on Preferences) jumps directly to the algorithm's section, showing its description and the full contract table of value inputs, ordinals, subscriptions, and outputs. To browse every algorithm at once, use Tools → Open Algorithm Help instead.

Aliases & Declarations

The alias table decouples the graph logic from physical data sources. Each row maps a symbolic alias to a constant, ordinal, external bridge source, persistent value, or published node output.

Add Row
Delete Row
Configure External Source...

	Alias	Source Type	Source Ref	Bridge	Default
1	No	constant			False
2	Ordinal.AllowC...	persistent			False
3	Yes	constant			True
4	actuator_posit...	external	dilutions_posit...	stdcomQtPS	
5	actuator_setp...	external	dilutions_setp...	stdcomQtPS	
6	actuator_targe...	constant			50.0
7	actuator_widt...	constant			50.0
8	always_false	constant			False
9	always_true	constant			True
10	applied_delta_u	node_output	@SetpointAss...		[]
11	control_Q	preload			1.0
12	control_R	preload			0.0592
13	control_S	preload			0.0592

Node outputs (internal, not editable here):

@BumplessTransferNode_da3734_out
@ControllerNode_82c771_out
@DivideNode_1a488e_out
@EdgesNode_523690_out
@EdgesNode_68b007_out

OK
Cancel

Figure 6: Alias Table dialog.

- Opens the alias cross-reference table editor. Each row maps a symbolic alias used by the graph's algorithms to a concrete source: a constant value, an ordinal name, an external bridge source, a file-backed persistent value, or a published node output.
- Alias — the symbolic name referenced by the algorithm.
- Source Type — constant, preload, persistent, ordinal, external (bridge subscription), or node_output (proxy for a node output).
- Source Ref — the real topic name, ordinal name, or internal node-output alias such as @NodeID_out.

Declarations

The Declarations dialog manages strict ordinal and subscription declarations. The engine enforces these in strict mode and the Problems dialog reports any design-time gaps.

Declare every external subscription topic used by the graph. Aliases that reference undeclared external topics are rejected in strict mode.

	Name	Direction	Value Type	Description
1	stdcomQtPS://reel_btw	sub	float	
2	stdcomQtPS://dilutions_positions	sub	float	
3	stdcomQtPS://dilution_automannual_request	sub	float	
4	stdcomQtPS://dilution_manual_setpoints	sub	float	
5	stdcomQtPS://dilutions_setpoints	subpub	float	
6	stdcomQtPS://dilution_automannual	subpub	float	
7	stdcomQtPS://mapped2sigma	subpub	float	

Add Edit Remove

Import from Alias Table

Close
Apply

Figure 7: Declarations dialog.

Communication & Bridges

Graphic Builder communicates with the outside world through bridge plugins. Each enabled bridge contributes sources under its own bridgenamespace://source namespace. Multiple bridges can run side by side.



Figure: Sources dock (empty when no bridge is connected).

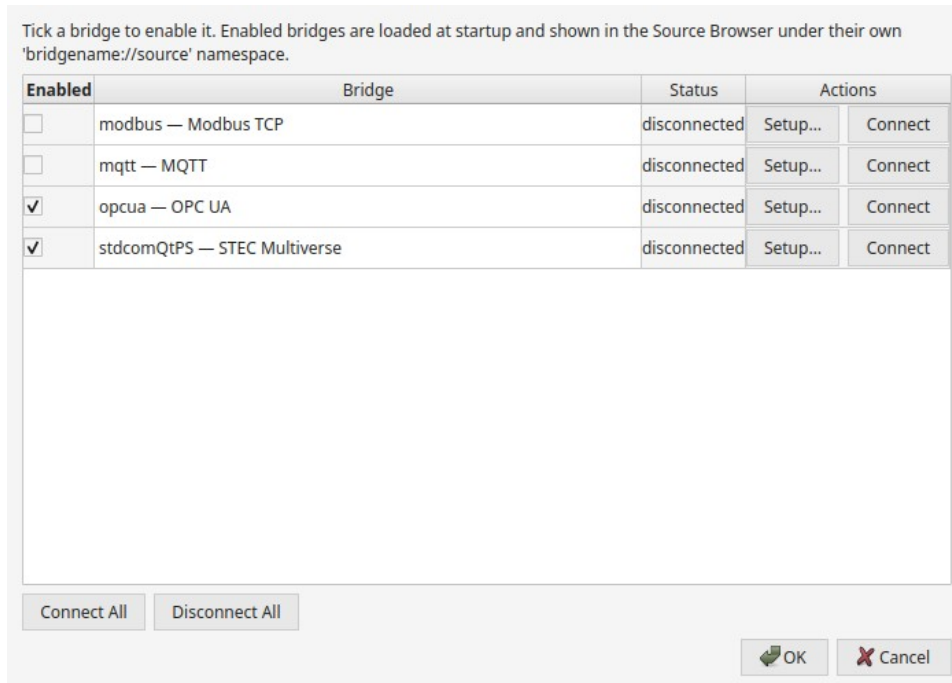


Figure: Bridges manager dialog.

Bridges manager

Communication → Bridges... opens the Bridges manager. The dialog lists every installed bridge plugin (for example, stdcomQtPS, mqtt, opcua, modbus). Each row has an Enabled checkbox, the bridge name, a live status label (connected or disconnected), a Setup... button, and a Connect/Disconnect button.

Tick the Enabled checkbox for every bridge you want to use. The enabled list is saved to the application INI file and is persistent across restarts: the next time you start the Graphic Builder, all enabled bridges are loaded and automatically connected. The live Connect/Disconnect buttons control only the current session and are not saved.

Click Setup... to open the plugin-specific configuration dialog (host, port, credentials, discovery topic, etc.). The dialog returns True only if you changed something; the Graphic Builder then calls connect() or disconnect() as needed. Multiple bridges can be enabled at the same time; each contributes its sources to the Source Browser under its own bridgename://source namespace.

Connect / Disconnect

Communication → Connect connects every enabled bridge to its transport. For STEC Multiverse this opens the TCP connection to the Multiverse server and starts discovering available subscription names; for MQTT it connects to the configured broker and begins listening on the discovery wildcard; for OPC UA and Modbus it opens their respective transports.

Communication → Disconnect closes every bridge transport and clears the Source Browser. This is useful when you want to stop live updates or before changing network settings. Connect and Disconnect affect only the current session; the Enabled list decides what reconnects on the next startup.

Source Browser

View → Sources toggles the Source Browser dock. It is empty when no bridge is connected. As soon as a bridge connects and reports its available sources, those sources appear in the tree under the selected bridge's namespace. Use the pulldown at the top to pick which bridge's sources are displayed. You can drag or copy a source name into an alias-table cell; the name is automatically qualified as `bridgename://source` so the alias table knows which transport owns it.

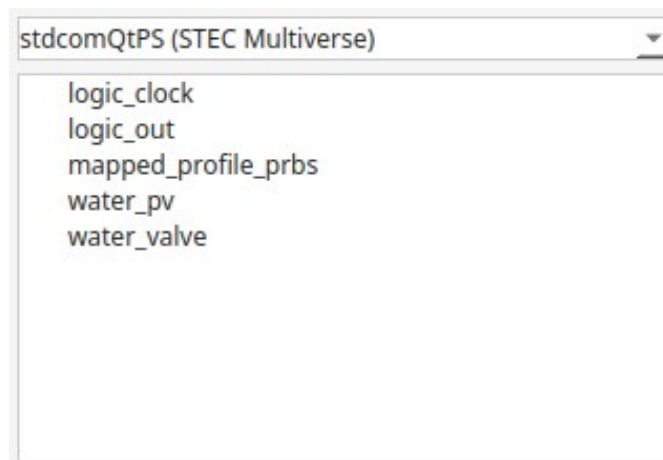


Figure: Source Browser showing live stdcomQtPS topics after the bridge connected.

How bridge plugins work

A bridge plugin is a Python class that inherits from `BridgePlugin` in `graphic_builder/bridge.py`. It is the only part of the Graphic Builder that knows how to talk to a particular transport; the rest of the application and the runtime engine see only the common interface. A finished control drawing must be able to run headlessly, so the interface is deterministic and contains no GUI-specific behaviour except the optional `setup()` and `configure_subscription()` dialogs.

The plugin must implement:

- `name()` — human-readable plugin name shown in menus.
- `parameters()` — list of `BridgeParameter` descriptors exposed in Preferences → Enabled Bridges (host, port, discovery topic, etc.).
- `setup(parent)` — show a modal plugin-specific configuration dialog; return `True` only if settings changed.
- `connect()` / `disconnect()` / `is_connected()` — open/close the transport and report state.
- `read(source)` — return the latest value list for a source, or `[]`.
- `write(source, value)` — publish a value list to a source.

- `_do_subscribe(source, config) / _do_unsubscribe(source)` — create and tear down real wire subscriptions (the base class tracks subscribers by callback identity, so you normally do not override `subscribe()/unsubscribe()`).
- `load_settings(settings) / save_settings(settings)` — persist parameter values in a unique `SETTINGS_GROUP`.
- Signals: `sig_sources_changed`, `sig_data_changed`, `sig_connected`, `sig_disconnected`, `sig_error`.

The plugin owns the source list. Use the inherited helpers `_set_sources()`, `_add_sources()`, and `_remove_sources()` to update it, and emit `sig_sources_changed` whenever the list changes so the Source Browser refreshes. Sources may be discovered incrementally at any time, not only at connect.

Building a new bridge plugin

There is no separate code-generation tool for bridge plugins. To add a new transport, create one Python module under `graphic_builder/plugins/` that subclasses `BridgePlugin`, implements every method, and registers itself in `graphic_builder/plugins/__init__.py` with `BRIDGE_REGISTRY.register(key, PluginClass)`. After registration, the plugin appears in Communication → Bridges and in Preferences → Enabled Bridges.

Rules a bridge plugin must follow:

- Implement every abstract method; raising `NotImplementedError` is not enough.
- Keep GUI code inside `setup()` and `configure_subscription()` only; the rest of the class must run headlessly.
- `setup()` must be modal, must not connect or disconnect itself, and must return `True` only when settings actually changed.
- `connect()` must call `self._set_connected(True)` after the transport is open; `disconnect()` must clear sources with `self._set_sources([])` and call `self._set_connected(False)`.
- `read()` must always return a list (empty if no data); `write()` must accept a list value.
- Coerce wire strings to native Python values (`bool/int/float`) before emitting `sig_data_changed`.
- On reconnect, refresh the source list and re-apply stored subscription configs; the base class keeps `self._subscriptions` and `self._subscription_configs` alive across disconnect/connect cycles.
- Use a unique `SETTINGS_GROUP` so the plugin's Preferences values do not collide with other bridges.

The canonical reference implementation is `graphic_builder/plugins/stdcom_qt.py` (STEC Multiverse). A compact second example is `graphic_builder/plugins/mqtt_bridge.py`. Open Help → How to Build a Bridge for the full developer guide, method table, minimal skeleton, registration example, and review checklist.

Runtime & Debugging

When the engine runs, the Watch panel shows live values, the Log panel captures printed output and runtime errors, the Problems dialog flags design-time issues, and the Firing Log records every pass for stream-level debugging.

Watch panel

View → Watch toggles the Watch dock. When the engine is running, it displays live values for aliases and node outputs. Each row shows the alias name, its current value (truncated to fit), and the source type. Double-click a row to open an Alias Inspector where you can set or inject a value; right-click a row for Watch / Debug options such as logging on change.

Log panel and Runtime Errors

The Log dock has two tabs. The Log tab is a plain-text view of every `print()`, status, alarm, and bridge message produced while the application runs. It is useful for tracing engine startup, subscription events, and diagnostics output.

The Runtime Errors tab is a table with Time, Node, and Error columns. Whenever a node raises an exception during a pass, the engine logs the node id, a short summary, and the full traceback here. Double-click any row and the canvas jumps to the failing block: the node is selected, centered in view, and outlined so you can open its Inspector or algorithm help (F1) immediately. Hover over a row to see the complete traceback as a tooltip. Click Clear to remove all runtime error rows; the tab label also shows the current error count.

Tip: keep the Log dock visible while commissioning a new drawing. A red Runtime Errors badge is usually the fastest way to find a mismatched alias, a dimension mismatch, or a missing subscription.

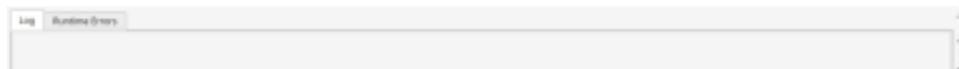
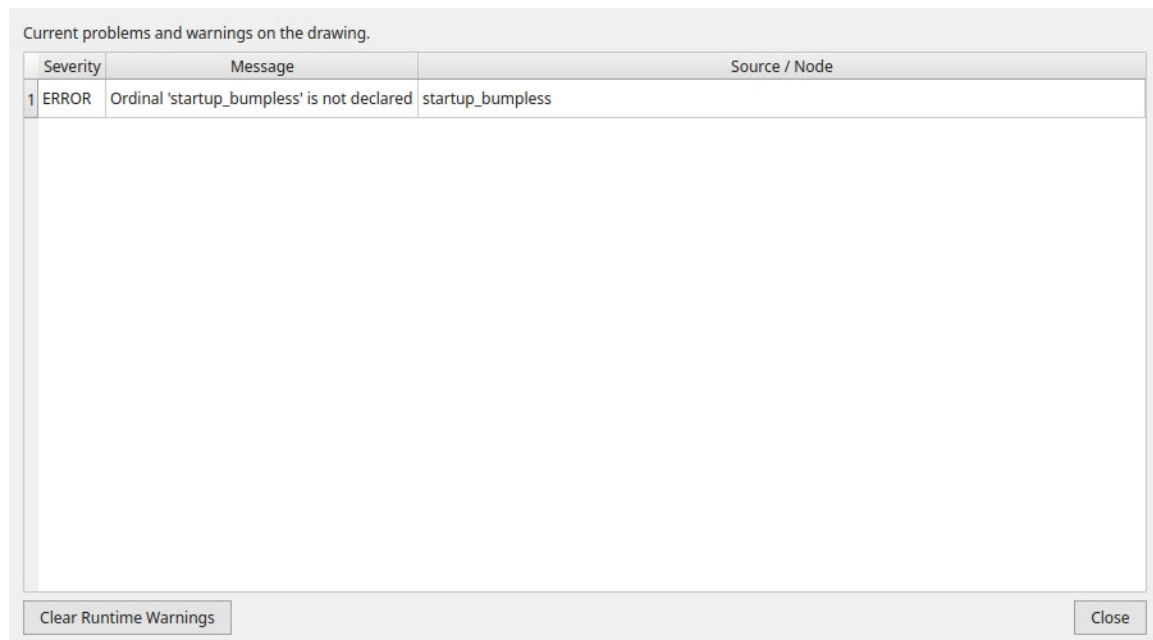


Figure: Log dock with Normal and Runtime Errors tabs.

Problems dialog

Open the drawing problems and warnings dialog. This dialog lists design-time problems (for example a Figure 8 feedback loop or a missing alias) and runtime warnings reported by the engine. Double-click a row to jump to the related node on the canvas. Runtime warnings can be cleared with the Clear Runtime Warnings button; design-time problems disappear automatically when the drawing is corrected.



Severity	Message	Source / Node
1 ERROR	Ordinal 'startup_bumpless' is not declared	startup_bumpless

Figure: Problems dialog.

Firing Log

View → Firing Log toggles the stream-debugger dock. When the engine runs, it records one row per pass with child rows for each fired node, its resolved inputs, and its output. Use Pause, Step, and Resume to control execution, and double-click a node row to open its inspector. This is the most detailed view of exactly what the engine did and in what order.

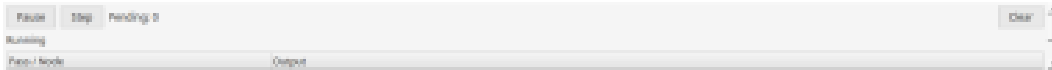


Figure: Firing Log dock with Pause/Step/Resume controls.

Preferences & Shortcuts

Edit → Preferences... (Ctrl+,) opens the settings dialog. Changes apply when you click OK. Preferences are saved to the application INI file and reloaded on the next startup. The dialog has two tabs: General and Shortcuts.

General tab

The General tab contains application-wide settings that affect appearance, available transports, default canvas behavior, and default folders.

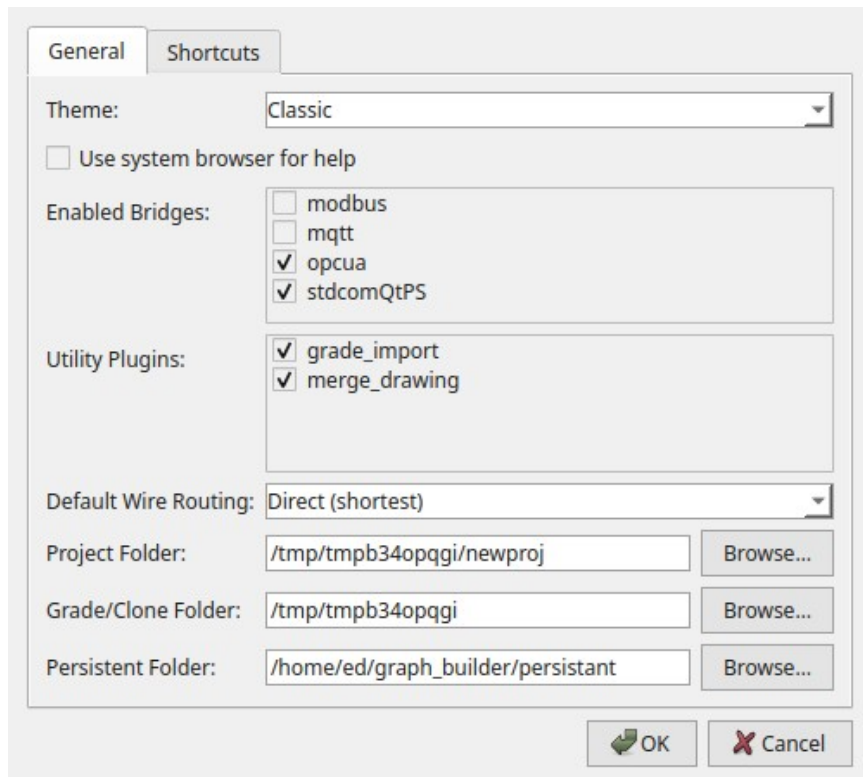


Figure 14: Preferences — General tab.

Theme

Choose the application color theme. Available themes are Dark, Light, Classic, and Solar. The theme is applied immediately when you click OK and affects the window chrome, canvas, nodes, docks, and dialogs.

Use system browser for help

When this checkbox is checked, pressing F1 or opening help from a menu opens the help file in the default external browser. When unchecked, help is shown in an in-application QTextBrowser viewer so no external browser is required.

Enabled Bridges

This checklist shows every installed bridge plugin (for example, stdcomQtPS, mqtt, opcua, modbus). Tick the transports you want to use. Enabled bridges are loaded at startup, connected, and shown in the Source Browser under their own bridgename://source namespace. Multiple transports can run side by side; each alias-table row then chooses which bridge to use.

Utility Plugins

This checklist shows every registered utility plugin (for example, Import automap4 Grade, Import / Merge Drawing). Ticked utilities appear as actions at the bottom of the Tools menu. They run on demand, not at runtime, and stay listed until unticked. On the first run with no saved list, all utilities are enabled.

Default Wire Routing

Sets the default style for new connections. Direct (shortest) draws a straight line between ports. XY (orthogonal) draws horizontal and vertical segments. You can change the style of individual wires from their right-click menu regardless of this default.

Project Folder

Folder where project JSON files are opened and saved. File → Open and File → Save As start here. Choosing a file in a different folder updates this preference automatically so the app follows your workflow. Type a path or click Browse to select one.

Grade/Clone Folder

Default folder for git clone/open of grade repositories and for File → New Project. If this field is empty, it defaults to the Project Folder; if the Project Folder is also empty, it falls back to ~/graph_builder. This lets grade/Git operations start where you already work.

Persistent Folder

Folder where persistent alias values are stored. Each persistent alias gets its own file here, with a backup copy for power-fail safety. Choose a location that is writable and backed up if the persisted values are important.

Shortcuts tab

The Shortcuts tab lists every registered action and its current key sequence. Click a shortcut cell and press the desired key combination to change it. The table has two columns: Action and Shortcut.

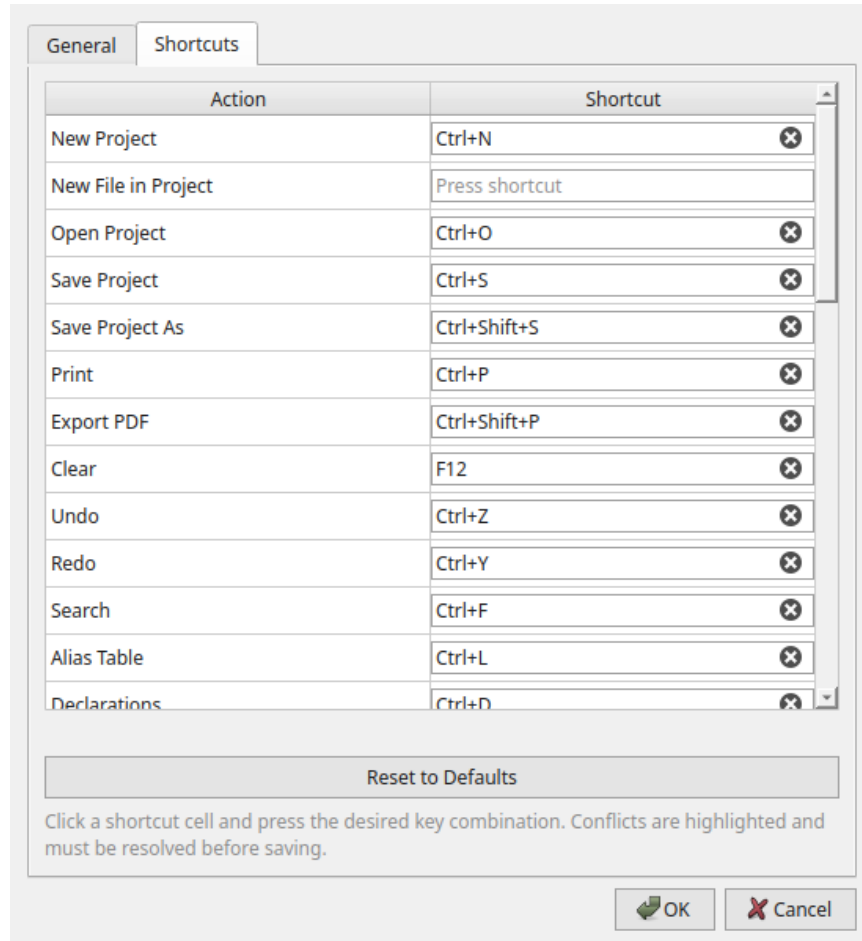


Figure 15: Preferences — Shortcuts tab.

To change a shortcut, click the Shortcut cell for the action and press the new key combination. To clear a shortcut, press Escape while the cell is capturing. If two actions are assigned the same sequence, a red warning appears below the table listing the conflicting actions. You must resolve the conflict before the OK button is enabled.

Click Reset to Defaults to restore every shortcut to its factory value. The default shortcuts are defined in the action registry and include common items such as New Project (Ctrl+N), Open (Ctrl+O), Save (Ctrl+S), Save As (Ctrl+Shift+S), Print (Ctrl+P), Export PDF (Ctrl+Shift+P), Undo (Ctrl+Z), Redo (Ctrl+Y), Clear (F12), Preferences (Ctrl+.), and Run Once (F5).

When you click OK, the shortcut map is saved, applied to the current window's actions, and written to the INI file so the new bindings are available on the next startup.

Available Algorithms

The palette groups every installed algorithm by category. This chapter gives a short operator-oriented explanation of what each block is for and how it is normally wired. For the full contract — every input, output, ordinal, subscription, default value, and type — see the Algorithm Reference appendix, or press F1 on a node in the palette or on the canvas.

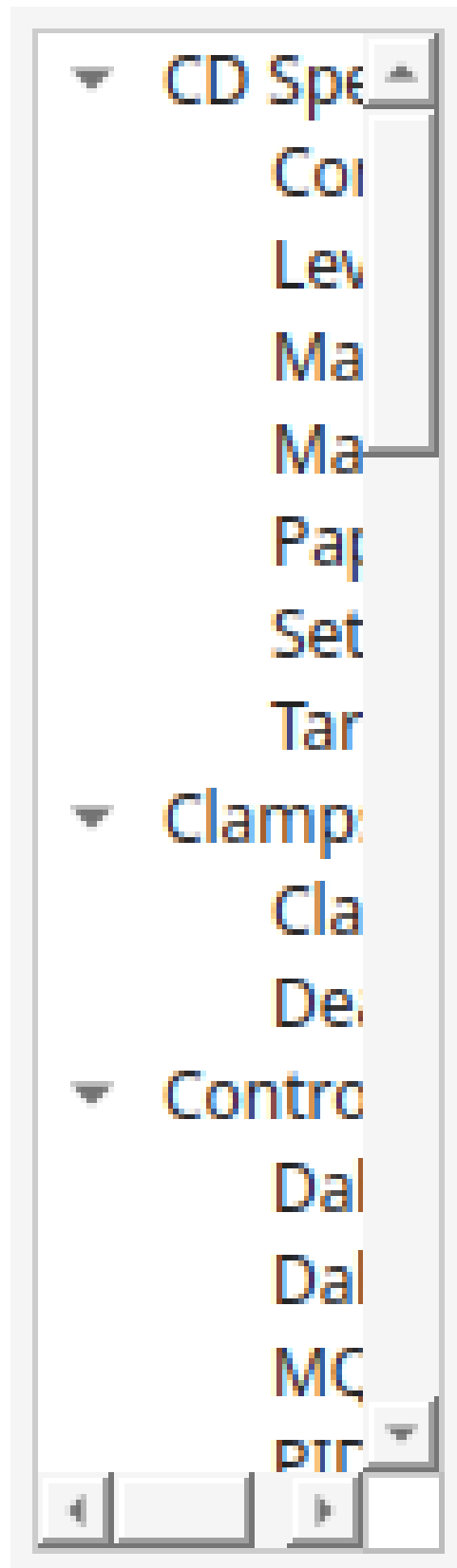


Figure: Palette dock showing algorithm groups.

CD Specific

ControllerNode

Wraps automap_control CDControllerMPC the way automap4 runs it: one move every move_every_n_scans scans (returns [] on held scans so the setpoint assembler holds), a 5-databox error taper at the paper extent, zones over air carry no error, paper-aware simple-zone gain matrix, edge move-cost weights from the S-curve. Outputs raw delta_u.

Typical use: Place this after the MapperNode and TargetGeneratorNode in a CD control chain. Wire actuator_positions from the process, mapped_profile and target_profile from the mapper/target nodes, and mapper_alias to the MapperNode's output alias. The node outputs raw delta_u, which is normally fed into a LevelRampNode or SetpointAssemblerNode. Use fixed_mask, profile, and paper_coverage to pin manual or off-sheet zones.

LevelRampNode

Hold the counted-zone (non-fixed) actuator mean and ramp it toward `target\_level\_pct` by at most `level\_ramp\_step\_pct` per move. Re-seeds from the current counted mean on first use, whenever the counted set (fixed_mask) changes, or when `reset` is true — automap4's bumpless actuator level.

Typical use: Insert between the ControllerNode and the SetpointAssemblerNode. It holds the counted-zone actuator mean, walks it toward the Preferences target level by at most the configured ramp step per move, and ensures CD moves remain zero-mean over the active zones. Wire delta_u from the controller and actuator_positions from the process.

ManualMaskNode

Build the controller fixed mask. A zone is fixed (True) if it is manual/auto-hold in `manual\_zones` OR off-sheet in `paper\_coverage`. If `paper\_coverage` is empty, only `manual\_zones` is used.

Typical use: Use this to build the fixed mask sent to the ControllerNode. Combine auto-hold ordinals (from speed or quality gates), manual-zone bits, and paper coverage so the controller leaves the right zones untouched.

MapperNode

Wraps automap_control OnlineAdaptiveMapper. Each scan feeds the raw profile and actuator positions into the mapper history and optionally runs closed-loop identification (scan/control/prbs). Outputs the footprint-weighted (.prbs) or simple-zone (.simple) mapped profile.

Typical use: This is normally the first node in a CD chain after the profile NotifierNode. Feed it the raw profile, actuator positions, machine speed, and optional grade seed aliases. It publishes the mapped profile and learns the S-curve/mapping matrix online. Other nodes read its output alias.

PaperCoverageNode

Fraction of paper (0..1) under each actuator's simple-zone window, from the S-curve owned by the MapperNode referenced by `mapper_alias`. 0 = zone over air (must be pinned). All ones if the mapper has no valid S-curve.

Typical use: Place this after the MapperNode and wire `mapper_alias` to the mapper's output. It computes per-actuator paper coverage 0..1 from the S-curve footprint. Feed the result into the ControllerNode as `paper_coverage`.

SetpointAssemblerNode

Add `delta_u` to `actuator_positions`, shift the counted-zone mean to `level`, clamp to 0..100, and preserve fixed/manual zones. When `delta_u` is empty (controller throttled by the transport delay) the last published setpoints are held, exactly like `automap4` holds `_last_setpoints` between moves; before the first move it holds the current feedback (bumpless).

Typical use: Use this as the final node before the actuator setpoint output. It adds `delta_u` to the current positions, applies the level shift, clamps to 0..100 %, and publishes the new actuator setpoints. Wire `delta_u`, `actuator_positions`, `level`, and `fixed_mask`.

TargetGeneratorNode

Compute a full-sheet target scalar as the mean of all finite non-zero values of `profile`, and emit a flat target profile filled with that scalar in the shape of `shape_from` (or of `profile` if not given). `automap4`: mean of the non-zero raw databoxes, broadcast to zone shape.

Typical use: Place immediately before the ControllerNode. It computes a full-sheet target profile (usually a flat shape derived from the mean of non-zero raw databoxes) and publishes it as `target_profile`. Use `shape_from` to choose the source profile.

Clamps and Limits

ClampNode

Clamp a value between +/- a limit. Supports scalar and matrix broadcasting: a single limit applies to all elements; a shorter limit array clamps the prefix and leaves the rest unclamped.

Typical use: Use anywhere a signal must stay inside physical limits. Common examples: clamping control moves before they reach an actuator, or bounding a setpoint between a high and low constant.

DeadbandNode

Pass the input through unchanged unless it is inside the deadband around zero. Supports scalar and matrix broadcasting: a single width applies to all elements; a shorter width affects the prefix and leaves the rest of the input unchanged.

Typical use: Place this after a noisy measurement or before a valve output to suppress tiny changes that would otherwise cause dithering. Values inside the deadband become zero.

Controllers

DahlinController

Dead-time compensating controller (Dahlin) for a first-order-plus-dead-time process. Tuned by the desired closed-loop time constant τ_c . Supports scalar and matrix broadcasting; every zone runs an independent controller.

Typical use: Use for first-order-plus-dead-time processes where you know the process gain, time constant, and dead time. It provides a dead-time-compensating controller in a single block.

DahlinControllerHistory

Dahlin dead-time compensating controller implemented with explicit per-zone history arrays (past errors and past outputs). Numerically identical to the standard Dahlin controller; the history arrays make the dead-time delay line explicit for debugging.

Typical use: Same control law as DahlinController, but keeps explicit per-zone history arrays. Use when you want to inspect or reset the history outside the node.

MQController

MQ controller: one-step model-predictive move removing a fraction aggressiveness of the error each run, $du = Q \cdot K \cdot \alpha \cdot e / (Q \cdot K^2 + R)$, rate-limited by $\max_delta_u$ and integrated into the absolute output u . A plain PID/Dahlin-slot controller for generic loops; the CD profile version is ControllerNode. Supports scalar and matrix broadcasting; every zone runs an independent controller.

Typical use: A generic single-/multi-loop version of the MQ (one-step MPC) controller used in the CD drawings, as a plain drop-in for a PID slot: error in, absolute control signal out, no mapping matrix and no transport-delay throttle. Tune the process gain K and aggressiveness (α); raise R or set $\max_delta_u$ to calm a noisy loop. Use a negative K for a reverse-acting loop. For the full CD profile chain use ControllerNode instead.

PIDNode

Proportional controller with enable gate. Supports scalar and matrix broadcasting: a single enable or gain applies to all elements; a shorter enable/gain array affects the prefix and leaves the rest enabled.

Typical use: A simple proportional controller with an enable gate. Wire the error to input, a boolean enable to gate, and feed the output to a clamp or an actuator. Useful for fast inner loops or secondary control.

Debugger

PJanicePlot

Design-time plot probe. Passes input through unchanged while a companion widget displays it as profile, trend, heatmap, histogram, or FFT. Remove or leave in live mode for headless production runs.

Typical use: A design-time plot probe. Drop it anywhere you want to visualize a 1-row matrix; double-click the node to open the companion widget. Choose profile, trend, heatmap, histogram, or FFT. Set ignore_zeros, chart theme, and Y-axis limits in the Inspector. The node passes its input through unchanged, so it can stay in the chain for headless runs if desired.

PJanicePoker

Design-time value intercept. Passes input through by default; the companion widget can hold or one-shot inject a custom value. Remove or leave in pass-through mode for headless production runs.

Typical use: A design-time value intercept. Place it inline between two nodes, double-click to open the widget, and switch between Pass through, Hold, or One shot. Use it to freeze a signal during commissioning or to inject a test value. Switch back to Pass through before leaving the drawing in production, or remove the node.

Filter

ExponentialFilterNode

Single-pole exponential moving-average filter. $\alpha = 1.0$ passes the input through unchanged; α close to 0.0 heavily smooths the signal. The first real input seeds the output. None or empty inputs hold the last output.

Typical use: Use on a noisy measurement. $\alpha = 1.0$ passes the signal unchanged; smaller α applies more smoothing. The filter seeds on the first real value and holds its last output when the input is None.

MovingAverageFilterNode

Rolling moving-average filter. $\text{window} = 1$ passes the input through; larger windows average the last N real inputs. The first real input seeds the output. None or empty inputs hold the last output.

Typical use: Use to smooth a signal with a rolling window mean. Larger windows give more smoothing but add more lag. $\text{window} = 1$ passes the input through unchanged.

SumNode

Sum multiple inputs element-wise. Inputs are read from a list of alias names. The node outputs None until every input has a real value, so missing externally-owned signals do not inject zeros into the sum.

Typical use: Use to add several 1-row signals element-wise. Provide the input aliases as a JSON list in the inputs parameter. All inputs must have the same length or broadcast according to the matrix rules.

Iterators

EdgesNode

Find the lowest or highest non-zero index in a 1-D profile, or a list of such indices for a 2-D array.

Typical use: Feed it a raw profile and select low or high edge mode. It returns the index of the first or last non-zero databox. Wire the result into the ControllerNode as paper_left/paper_right, or into the MapperNode for framing.

ReverseNode

Reverse a list, or each inner list in a list-of-lists. When reverse is false the input passes through unchanged.

Typical use: Use to reverse the order of a list or each row of a list-of-lists. Handy when scanner direction flips the profile order relative to the actuator array.

Logic

AndNode

Logical AND of two boolean inputs. Supports scalar and matrix broadcasting: a single True leaves the other input unchanged; a shorter second input leaves the rest of the first input unchanged.

Typical use: Wire two boolean conditions (for example, speed_ok AND quality_ok). The output is true only when both inputs are true. Supports scalar and matrix broadcasting.

BumplessTransferNode

Copy one or more source aliases into target aliases when the gate is true, then write a done ordinal. Used for bumpless transfer at startup: snapshot actuator_positions into setpoint aliases so control starts from the current machine state. Check 'Src Also' on the gate input so the node runs when the startup ordinal changes.

Typical use: Use once at startup. Trigger it from a NotifierNode on the live source you want to copy. When gate is true, it copies each source_alias into the matching target_alias, writes the done_ordinal, and never transfers again. This avoids a step change when enabling control.

CounterNode

Count the number of input aliases that are currently true / non-zero. Receives a trigger alias and reads all connected aliases from the alias table. Useful for testing multi-notifier parent chains.

Typical use: Feed it a list of boolean alias names. It counts how many are currently true/non-zero. Useful for alarm summaries or for deciding how many gates are open.

GreaterThanNode

Return true if $a > b$. Supports scalar and matrix broadcasting: a single b compares against all elements; a shorter b compares against the prefix and leaves the rest of the result False.

Typical use: Compare $a > b$. Use for threshold alarms, high-limit checks, or enabling a controller only when a value is above a setpoint.

LessThanNode

Return true if $a < b$. Supports scalar and matrix broadcasting: a single b compares against all elements; a shorter b compares against the prefix and leaves the rest of the result False.

Typical use: Compare $a < b$. Use for low-limit checks or for enabling a controller only when a value is below a setpoint.

MultiplexNode

Select between two inputs. When `use_a` is true the node publishes `input_a`; otherwise it publishes `input_b`. Useful for switching between `mapped_profile_simple` and `mapped_profile_prbs` in the CD control chain.

Typical use: Use as a selector between two inputs. Common use case: switch between `mapped_profile_simple` and `mapped_profile_prbs` by wiring `use_a` to an ordinal or constant.

NotNode

Logical NOT of a boolean input. Supports scalar and matrix (1-row array).

Typical use: Invert a boolean signal. Use to turn an alarm condition into an enable signal, or vice versa.

OrNode

Logical OR of two boolean inputs. Supports scalar and matrix broadcasting: a single False leaves the other input unchanged; a shorter second input leaves the rest of the result unchanged.

Typical use: Wire two boolean conditions. The output is true if either input is true. Use for combining permissives or alarms.

SetOrdinalNode

Write a boolean value to the ordinal named by this node's output alias. Downstream nodes read the flag through an alias of source type 'ordinal' with the same name.

Typical use: Write a boolean input to the ordinal named by `output_alias`. Use it to latch state such as `machine_speed_good` or `control_enabled` based on logic upstream.

Math

DeltaNode

Subtract a reference from every element. The reference can be supplied as ``standard`` or, if empty, computed as the mean of the input. With `include_zeros` off, zeros are ignored when computing the mean and are left as zero in the output.

Typical use: Subtract a reference from every element. Use to compute deviation from a baseline profile, or subtract a constant offset. The reference can be supplied as `standard` or as a direct reference alias.

DivideNode

Divide two numeric inputs (a / b). Supports scalar and matrix broadcasting: a single divisor divides all elements; a shorter divisor affects the prefix and leaves the rest of the first factor unchanged. Division by zero returns 0.0.

Typical use: Compute a / b . Use to scale a signal (for example, divide a 2-sigma value by 2 to get sigma).

InvertNode

Invert the sign of every element: positive becomes negative and vice versa.

Typical use: Negate every element. Use to flip the sign of an error or a control move.

MeanNonZeroNode

Mean of the non-zero elements of a profile. While the gate input is false the node holds and republishes its last value instead of recomputing.

Typical use: Compute the mean of the non-zero elements. Use for BW mean at the reel or any average that must ignore off-sheet zeros. Gate it with a boolean to hold the last value.

MultiplyNode

Multiply two numeric inputs. Supports scalar and matrix broadcasting: a single factor multiplies all elements; a shorter second factor affects the prefix and leaves the rest of the first factor unchanged.

Typical use: Compute $a * b$. Use to apply a gain, convert units, or scale a profile by a tuning factor.

SubtractNode

Compute the difference between two inputs. Supports scalar and matrix broadcasting: a shorter setpoint is applied to the prefix; remaining elements keep the measured value.

Typical use: Compute $a - b$. The most common use is $\text{error} = \text{setpoint} - \text{process_value}$ for a controller.

Sinks

ValveOutputNode

Publish a value to a subscription alias (the final output of a chain). Accepts a scalar or a matrix (1-row array).

Typical use: Use as the final actuator output. It publishes a value to an external alias (typically the actuator setpoint Multiverse topic).

WriteNode

Publish a value to an external alias (Multiverse topic). Use this as the final output of a chain. Accepts a scalar or a matrix (1-row array).

Typical use: Generic output node. Use to publish any computed value to an external alias, not just valve positions.

Sources

ConstantNode

Output a constant value. Accepts a scalar or a matrix (1-row array).

Typical use: Use for fixed values: setpoints, gains, limits, number of zones, or any literal. The value can be a scalar or a 1-row matrix.

NotifierNode

Subscribe to an alias and trigger downstream nodes when it changes. Passes through the latest value of the monitored alias so downstream nodes can read it from the output alias or internal node output.

Typical use: This is the entry point of most chains. Wire its subscription parameter to an external alias. Whenever the alias updates, the node fires and passes the value through to its children.

Statistics

LinearRegressionNode

Fit a straight line $y = \text{slope} * x + \text{intercept}$ to two equal-length sequences. Publishes [slope, intercept] on output_alias and, if set, on slope_output_alias and intercept_output_alias. Returns None if there are fewer than min_points real values or if x/y lengths differ.

Typical use: Feed two equal-length sequences x and y. It fits $y = \text{slope} * x + \text{intercept}$ and can publish the two coefficients to separate output aliases. Set min_points to require a minimum sample size.

TwoSigmaNode

Compute twice the standard deviation (2-sigma) of the input values. Returns a single scalar.

Typical use: Compute twice the standard deviation of the input values. Use as a simple variability metric for a profile; set `include_zeros` to `False` to ignore off-sheet zeros.

Timer

TimerAlarmNode

Fire once per day at each configured wall-clock time. Outputs the time as an HHMM integer (e.g. 800 for 08:00).

Typical use: Schedule daily wall-clock alarms. Set times to a comma-separated list such as 0800,1200,1600 or 08:00,12:00,16:00. The node fires at each configured time and outputs the HHMM integer of the alarm that just fired (e.g. 800 for 08:00). Wire its output to downstream batch or logic nodes.

TimerIntervalNode

Fire repeatedly at a fixed interval in milliseconds and output the interval value. Minimum interval is 100 ms.

Typical use: Fire repeatedly at a fixed interval in milliseconds. Set `interval_ms` to the desired period; the minimum allowed value is 100 ms. Each tick wakes downstream nodes and outputs the interval value (e.g. 1000).

Debug probes in depth

The Debugger group contains two design-time probes: `PJanicePoker` and `PJanicePlot`. Both pass their input through unchanged, so they never alter the computed result; they only give you a window into the signal while you are building or commissioning the drawing.

PJanicePoker

`PJanicePoker` is an inline value intercept. Drop it between two nodes (for example, between a filter and a controller), then double-click the node to open its companion widget. The widget shows the node's display name, the current input value, and three mode buttons:

- **Pass through** — the input is copied to the output unchanged. Use this when you are finished debugging.
- **Hold** — the last output value is frozen. Use this to keep a downstream node awake with a steady value while you inspect upstream logic.
- **One shot** — the next computed value is replaced by whatever you type in the Next output field, then the node returns to Pass through. Use this to inject a single test value.

The active mode button is highlighted in green. Changes take effect immediately. The node title on the canvas comes from the name parameter, so you can label it 'Hold PV' or 'Inject bump' for

clarity. Right-click the node and choose Hide to close the widget without deleting the node. Before you put a drawing into unattended production, switch every PJanicePoker to Pass through or delete the probe entirely.

PJanicePlot

PJanicePlot is an inline plot probe. Drop it on any wire carrying a scalar or 1-row matrix, double-click to open the widget, and choose the plot type from the Inspector or the widget toolbar:

- Profile — line or bar chart of the current 1-row matrix (for example, a CD profile).
- Trend — time-series of scalar values; trend_points controls how many historical points are kept.
- Heatmap — 2-D view of recent profile history; heatmap_rows and heatmap_cols configure the grid (0 = auto).
- Histogram — distribution of values; histogram_bins sets the bin count.
- FFT — frequency spectrum of the current 1-row matrix.

Common Inspector parameters:

- plot_type — profile / trend / heatmap / histogram / fft.
- chart_style — line or bar.
- chart_theme — dark or light background.
- ignore_zeros — exclude zeros from means, histograms, autoscale, and draw gaps in line plots.
- trend_points — history length for trend plots.
- y_min / y_max — fixed Y-axis limits (ignored when y_autoscale is true).
- y_autoscale — let the widget pick the Y range automatically.

Use PJanicePlot to sanity-check a mapped profile, watch a target converge, or spot noise and drift while tuning a chain. Like PJanicePoker, it can be left in pass-through mode for headless operation, but it is usually removed once commissioning is complete.

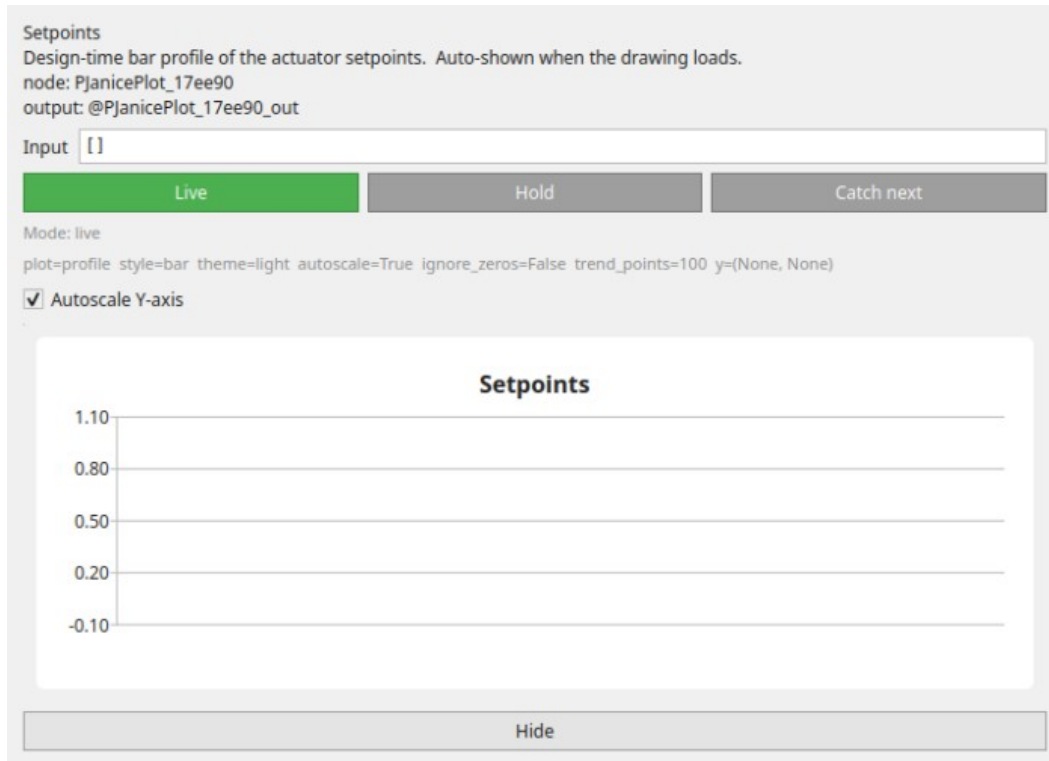


Figure: PJanicePlot widget showing a profile.

Editing block labels and operator notes (F2)

Every node has an internal id, an algorithm type, a short display name, and a free-form operator description. The display name appears on the canvas below the algorithm name; the description is stored with the project and is useful when you (or another operator) modify the drawing later.

To edit these fields, select one node and press F2 (or choose Edit → Edit Node Description). The dialog shows the read-only Node ID and Algorithm, plus two editable fields:

- Display name — a short label such as 'Profile notifier', 'CD controller', or 'Setpoint asm'.
- Description — operator notes explaining what the block does, why it is here, and how it is wired. For example: 'Computes the full-sheet target as the mean of non-zero raw databoxes. Triggered by mapped_profile_prbs.'

Good labels and descriptions make large drawings much easier to maintain. They are saved with the project and are preserved when you merge or import drawings. The node is marked staged (green outline) after you click OK until the project is saved.

CD Control Example (cdcontrol.json)

cdcontrol.json is a complete cross-direction (CD) supervisory control drawing that mirrors automap4's live control chain. It loads a verified grade seed and runs one control move every two scans.

The chain from left to right is:

- NotifierNode on profile scan — triggers each pass from the raw profile.
- MapperNode — seeds the grade S-curve and learns the actuator-to-profile mapping.
- EdgesNode (left/right) — finds the paper extent from the raw profile.
- PaperCoverageNode — per-zone paper coverage from the mapper's S-curve.
- ManualMaskNode — builds the fixed/manual/off-sheet mask.
- MultiplexNode — selects between mapped_profile_simple and mapped_profile_prbs.
- TargetGeneratorNode — flat full-sheet target from the mean of non-zero raw databoxes.
- LevelRampNode — holds and ramps the counted-zone actuator level.
- ControllerNode — solves raw delta_u with the MPC controller.
- SetpointAssemblerNode — adds delta_u, applies the level, clamps, preserves fixed zones, and publishes the actually-applied move (applied_delta_u) for closed-loop identification.
- NotifierNode on automanual state — publishes the assembled setpoints back to Multiverse.
- BumplessTransferNode — snapshots actuator_positions into setpoint aliases at startup.
- TwoSigmaNode / DivideNode / PJanicePoker / PJanicePlot — debug/monitoring probes (including a 2-sigma trend tapped after the poker).

The alias table carries the grade tuning constants (control_Q/R/S/T, aggressiveness, edge terms, actuator target level, ramp step, etc.), the Ordinal.AllowCLID gate, and external stdcomQtPS topics for the machine interface.

Closed-loop identification (CLID)

With control running, the MapperNode can refine the actuator-to-profile mapping from the machine's own response to control moves — the same feature as automap4's "Refine mapping from control moves". Each applied move is paired with the profile it was computed from; once the transport delay has passed, the response is compared and the mapping rows learn at a very conservative rate. The mapper pairs the move the machine actually applied (published by the SetpointAssemblerNode as applied_delta_u — after the level shift and clamp), so a level ramp in flight reads as a plant-wide move and is never misattributed to one zone.

Refinement is guarded so noisy or ambiguous scans teach it nothing:

- Attribution breaker — if more than 15% of zones move at once (a broad correction or a level-ramp step), the pair is skipped: the response cannot be attributed to individual actuators.
- Same-phase pairing — a pair older than transport_delay + 2 scans is discarded, so a stale or gapped scan never pairs across a disturbance.

- Trim event — if the paper extent changed by more than 4 databoxes between the pair's scans, the delta is paper appearing, not a response: skipped.
- Coherence and energy gates — the measured response must be one-signed and strong enough inside the zone's S-curve window.

CLID is gated by the persistent alias `Ordinal.AllowCLID` (default `False` — off until enabled). To switch it on: Edit → Alias Table..., right-click the `Ordinal.AllowCLID` row → Watch / Debug..., type `True` and press Set. The value is saved to the persistent store, so it stays on across restarts until set back to `False`. (Convention: an `Ordinal.` prefix marks an operator flag that should stay `True/False`; the alias itself may be preload or persistent.)

Watch the Log panel: "CLOSED-LOOP ID: Refined N zones from control move" means a localized move (for example a manual-zone release) taught the mapping; "skipped — N/96 zones moving simultaneously" during broad corrections is the breaker working as designed. The wander estimate line is report-only — in this edge-free drawing it never shifts the S-curve frame. If you ever want the seed mapping back, reopen the project: the mapper reseeds from the persistent `mapping_matrix`.

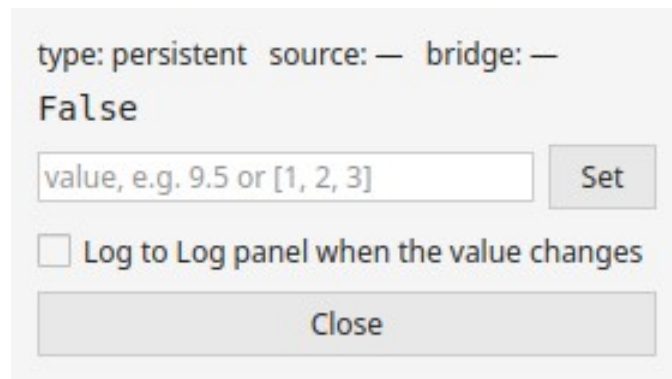


Figure: Enabling CLID — the alias inspector on `Ordinal.AllowCLID` (type `True`, Set).

Canvas Printout

The following multi-page PDF was generated directly from the Graphic Builder canvas with `cdcontrol.json` loaded. It uses the "Readable (labels ≥ 8 pt, tiled)" scaling mode, so the drawing is split across A4 landscape pages and port labels remain legible. This is the same output you get from File → Export PDF → Canvas.

PDF file: `BuilderRunner2/cdcontrol_printout.pdf` (12 pages, A4 landscape)

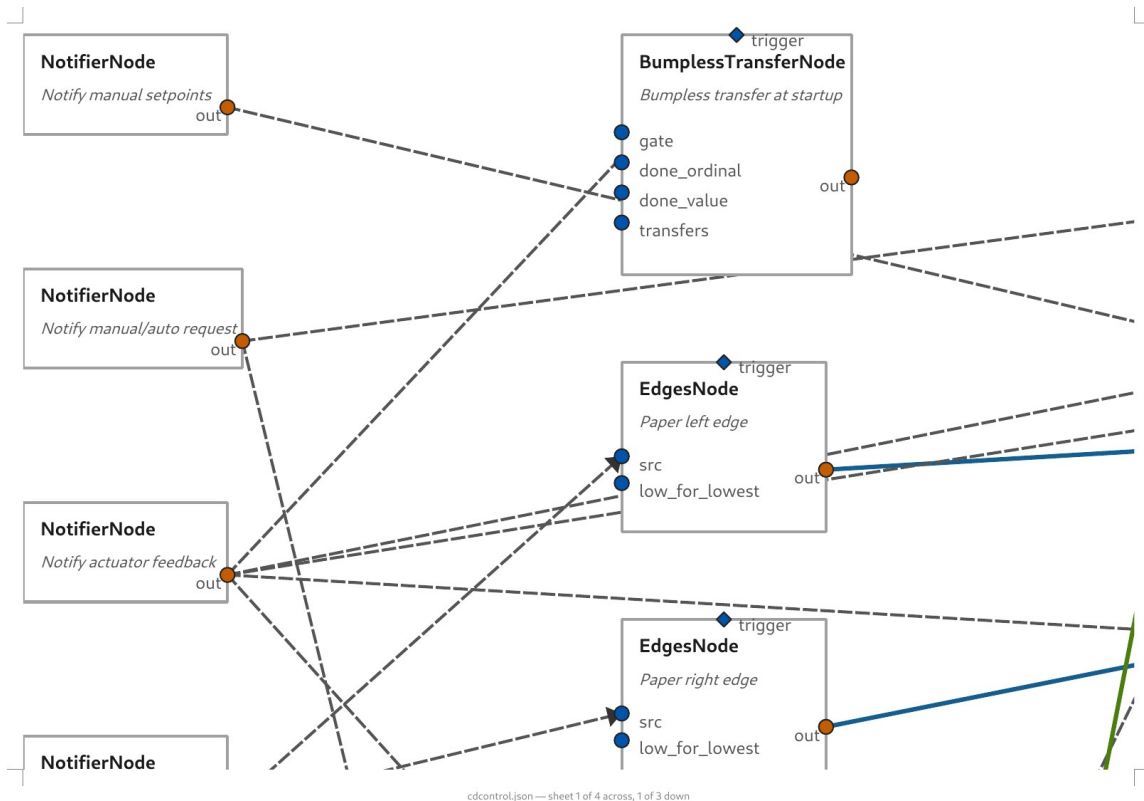


Figure: First page of the `cdcontrol.json` canvas printout.

Running Headlessly

Every BuilderRunner2 project JSON can be loaded and executed without opening the Graphic Builder window. The same runtime classes the GUI uses — `load_project` / `logic_from_dict`, `Engine`, `Graph`, and `BridgeHub` — are available from Python.

Why run headlessly?

- Deploy a verified control drawing as a long-running service on a plant computer.
- Run closed-loop tests, regression checks, or CI without launching Qt widgets.
- Drive the engine from an external scheduler, simulator, or test harness.

Key classes

`graphic_builder.serialization.logic_from_dict()` parses a migrated project JSON into a `TemplateArtifact` containing nodes, edges, alias table, and declarations.

`graphic_builder.runtime.GraphNode` / `GraphEdge` / `Graph` assemble the engine graph from the parsed artifact.

builderv2.bridge.BridgeHub + InMemoryBridge (or RuntimeBridgeAdapter around a bridge plugin) form the transport layer that feeds data into the engine and receives published outputs.

builderv2.engine.Engine is the headless runtime. Call start() after all subscriptions are wired, then publish incoming topics through the bridge.

Example: run_json_headless.py

A complete example ships as BuilderRunner2/run_json_headless.py. It loads the project file given on the command line, builds the engine on an InMemoryBridge, seeds the manual/auto state, and runs 40 scans of the UTP2 simulator against cdcontrol.json.

Command:

```
cd /home/ed/automap
QT_QPA_PLATFORM=offscreen .venv/bin/python BuilderRunner2/run_json_headless.py
/home/ed/graph_builder/cdcontrols/cdcontrol.json
```

The engine prints scan number, current 2σ , and counted-zone actuator mean, exactly as the closed-loop smoke test does, but without any Graphic Builder window.

Wiring a real bridge

For production, replace InMemoryBridge with a RuntimeBridgeAdapter around the desired BridgePlugin instance (stdcomQtPS, mqtt, opcua, modbus). The plugin is configured in the same way as in the GUI — host, port, discovery topic, etc. — then added to the BridgeHub under the name the alias table expects (usually "stdcomQtPS").

Because stdcomQtPS uses Qt networking, even a headless service still needs a QApplication event loop, but it can run with no visible windows (QT_QPA_PLATFORM=offscreen or an Xvfb framebuffer). Bridge plugins that are pure sockets may run with no Qt at all if instantiated directly.

Exporting a canvas printout from the command line

BuilderRunner2/export_cdcontrol_pdf.py loads cdcontrol.json offscreen and writes a tiled PDF to BuilderRunner2/cdcontrol_printout.pdf. You can adapt the same pattern for any project JSON.

Algorithm Reference

The following appendix is generated from algohelp.html. Each algorithm's description is followed by its contract inputs: kind, name, type, whether it is required, default value, and description.

CD Specific

ControllerNode

Wraps automap_control CDControllerMPC the way automap4 runs it: one move every move_every_n_scans scans (returns [] on held scans so the setpoint assembler holds), a 5-databox error taper at the paper extent, zones over air carry no error, paper-aware simple-zone gain matrix, edge move-cost weights from the S-curve. Outputs raw delta_u.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	actuator_positions	float	yes		Actuator positions (NumberOfZones elements)
value	mapped_profile	float	yes		Mapped profile (NumberOfZones elements)
value	target_profile	float	yes		Target profile (NumberOfZones elements)
value	mapper_alias	string	yes		Output alias of the MapperNode that owns the mapping matrix
value	fixed_mask	float	no	[]	Fixed/manual/off-sheet mask (NumberOfZones elements)
value	profile	float	no	[]	Raw databox profile (0 = no paper). Gives the paper mask for the gain matrix and

					the paper extent for the error taper
value	paper_coverage	float	no	[]	Per-zone paper coverage 0..1 (PaperCoverageNode); zones with 0 get error weight 0 and measurement = target
value	paper_left	float	no		First non-zero databox (EdgesNode low); anchors the 5-box error taper. None = from profile
value	paper_right	float	no		Last non-zero databox (EdgesNode high); anchors the 5-box error taper. None = from profile
value	left_edge	float	no	0.0	Mapper ruler left end in databoxes (edge-free frame: 0)
value	right_edge	float	no		Mapper ruler right end in databoxes (edge-free frame: NumberOfDataboxes-1)
value	edge_ignore_db	float	no	25.0	Edge band in databoxes over which the move

					cost ramps up to edge_suppression_mult (automap4 Edge ignore, default 25)
value	move_every_n_scans	int	no	0	Transport-delay throttle: solve every N scans, return [] (hold) otherwise. 0 = transport_delay_scans
value	use_simple_zone	bool	no	True	Use simple-zone gain matrix
value	auto_tune	bool	no	False	Derive R/S/T/max_delta_u from the mapping
value	transport_delay_scans	int	no	2	Transport delay in scans
value	Q	float	no	1.0	Error-cost weight
value	R	float	no		Move-cost weight (None = keep controller value)
value	S	float	no		Move-bending weight (None = keep controller value)
value	T	float	no		Actuator-profile curvature weight (None = keep controller value)

value	aggressiveness	float	no	0.7	Alpha: fraction of error removed per move
value	max_delta_u	float	no		Per-move rate limit (None = keep controller value)
value	dilution_mode	bool	no	True	Dilution actuators (no adjacent-difference bending limit)
value	integral_gain	float	no	0.0	Integral gain
value	edge_suppression_mult	float	no	3.0	Edge move-cost multiplier
value	anchor_gain	float	no	0.0	Anchor gain
value	num_actuators	int	no	96	Number of actuator zones (NumberOfZones); inferred from inputs if 0
value	num_databoxes	int	no	500	Number of databoxes (NumberOfDataboxes); used for controller sizing

LevelRampNode

Hold the counted-zone (non-fixed) actuator mean and ramp it toward `target\_level\_pct` by at most `level\_ramp\_step\_pct` per move. Re-seeds from the current counted mean on first use, whenever the counted set (fixed_mask) changes, or when `reset` is true — automap4's bumpless actuator level.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish

					this node's output to
value	actuator_positions	float	yes		Current actuator position array (NumberOfZones elements)
value	fixed_mask	float	no	[]	Per-zone bool mask (NumberOfZones elements): True = fixed/manual /off-sheet zone
value	target_level_pct	float	no	50.0	Target counted-zone mean level (percent)
value	level_ramp_step_pct	float	no	3.0	Maximum level change per move (percent)
value	reset	bool	no	False	True = re-seed level from current counted-zone mean
value	delta_u	float	no		Optional controller move; when wired and empty [] (throttled scan) the level holds instead of stepping
value	controller_output_alias	string	no	delta_u	Output alias of the ControllerNode whose anchor must

					follow the walking level
--	--	--	--	--	--------------------------

ManualMaskNode

Build the controller fixed mask. A zone is fixed (True) if it is manual/auto-hold in `manual\_zones` OR off-sheet in `paper\_coverage`. If `paper\_coverage` is empty, only `manual\_zones` is used.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	manual_zones	float	yes		Per-zone mask (NumberOfZones elements): non-zero = manual / auto-hold zone
value	paper_coverage	float	no	[]	Optional per-zone mask (NumberOfZones elements): non-zero = on-sheet zone

MapperNode

Wraps automap_control OnlineAdaptiveMapper. Each scan feeds the raw profile and actuator positions into the mapper history and optionally runs closed-loop identification (scan/control/prbs). Outputs the footprint-weighted (.prbs) or simple-zone (.simple) mapped profile.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	profile	float	yes		Raw measurement profile

					(NumberOfDataboxes elements)
value	actuator_positions	float	yes		Actuator position array (NumberOfZones elements)
value	left_edge	float	no	0.0	Left sheet edge in databox units
value	right_edge	float	no		Right sheet edge in databox units (default: len(profile)-1)
value	mode	string	no	none	Learning mode: none (history only — what automap4 does during control), scan (idle learner), prbs (identification). 'control' is the retired legacy learner and is not used by automap4's live path.
value	transport_delay_scans	int	no	2	Transport delay in scans
value	output_mode	string	no	prbs	Output projection: prbs (footprint) or simple (zone average)
value	prbs_values	float	no	[]	PRBS excitation vector

					(NumberOfZones elements), used when mode=prbs
value	num_actuators	int	no	96	Number of actuator zones (NumberOfZones); inferred from actuator_positions if 0
value	num_databoxes	int	no	500	Number of databoxes (NumberOfDataboxes); inferred from profile if 0
value	headbox_width_mm	float	no	4800.0	Headbox width in millimetres
value	actuator_width_mm	float	no	50.0	Actuator zone width in millimetres
value	mapping_matrix	float	no	[]	Optional initial mapping matrix (N_act x N_box) from an imported grade
value	estimated_coeffs	float	no	[]	Grade S-curve cubic (4 numbers, estimated_coeffs); drives zone centres, coverage and gain matrix
value	shrinkage_model_valid	bool	no	False	Grade shrinkage_model_valid flag; True enables the S-curve

					above (False = hard-coded fallback curve)
value	response_centers_norm	float	no	[]	Grade per-zone learned response centres in databoxes (NumberOfZones elements)
value	refined_width	float	no	0.0	Grade mapper_refined_width (response sigma in databoxes); sets the gain scale, 0 = from geometry
value	learned_count	float	no	[]	Grade learned_count per zone (NumberOfZones elements)
value	mapping_quality	float	no	[]	Grade mapping_quality per zone (NumberOfZones elements)
value	headbox_center_db	float	no		Grade mapper_headbox_center (databoxes); None = geometry default
value	headbox_width_db	float	no		Grade mapper_headbox_width (databoxes); None = geometry

					default
value	delta_u	float	no	[]	Controller move vector (source-only wire from the controller output). Read one pass late; paired with the previous scan's profile for CLID.
value	allow_clid	bool	no	False	Closed-loop identification gate (bind Ordinal.Allow CLID). False = no refinement from control moves (automap4 'Refine mapping from control moves').

PaperCoverageNode

Fraction of paper (0..1) under each actuator's simple-zone window, from the S-curve owned by the MapperNode referenced by mapper_alias. 0 = zone over air (must be pinned). All ones if the mapper has no valid S-curve.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	profile	float	yes		Raw measurement profile (NumberOfDataBoxes elements); 0 = no paper

value	mapper_alias	string	yes		Output alias of the MapperNode that owns the S-curve (quoted string literal)
value	left_edge	float	no	0.0	Mapper ruler left end in databoxes (edge-free frame: 0)
value	right_edge	float	no		Mapper ruler right end in databoxes (edge-free frame: len(profile)-1)

SetpointAssemblerNode

Add delta_u to actuator_positions, shift the counted-zone mean to `level`, clamp to 0..100, and preserve fixed/manual zones. When delta_u is empty (controller throttled by the transport delay) the last published setpoints are held, exactly like automap4 holds _last_setpoints between moves; before the first move it holds the current feedback (bumpless).

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	actuator_positions	float	yes		Current actuator position array (NumberOfZones elements)
value	delta_u	float	yes		Control move from the controller (NumberOfZones elements); empty [] =

					throttled scan, hold last setpoints
value	level	float	yes		Target counted-zone mean level (percent)
value	fixed_mask	float	no	[]	Per-zone bool mask (NumberOfZones elements): True = fixed/manual /off-sheet zone
value	manual_setpoints	float	no	[]	Per-zone manual setpoint override (NumberOfZones elements); used only where fixed_mask is True
value	applied_output_alias	string	no		Optional second output: the move actually applied per zone (post level-shift and clamp, 0 on fixed zones) for CLID pairing

TargetGeneratorNode

Compute a full-sheet target scalar as the mean of all finite non-zero values of `profile`, and emit a flat target profile filled with that scalar in the shape of `shape\_from` (or of `profile` if not given). automap4: mean of the non-zero raw databoxes, broadcast to zone shape.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	profile	float	yes		Profile the target mean is taken from (automap4: the RAW databox profile, non-zero boxes only)
value	shape_from	float	no	[]	Optional profile whose shape the output takes (e.g. the mapped zone profile); empty = same shape as profile

Clamps and Limits

ClampNode

Clamp a value between +/- a limit. Supports scalar and matrix broadcasting: a single limit applies to all elements; a shorter limit array clamps the prefix and leaves the rest unclamped.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	x	float	yes		Value to clamp (scalar or matrix)
value	hiLimit	float	no	3.0	Symmetrical limit (scalar or matrix)

DeadbandNode

Pass the input through unchanged unless it is inside the deadband around zero. Supports scalar and matrix broadcasting: a single width applies to all elements; a shorter width affects the prefix and leaves the rest of the input unchanged.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	x	float	yes		Input value (scalar or matrix)
value	width	float	no	1.0	Deadband half-width (scalar or matrix)

Controllers

DahlinController

Dead-time compensating controller (Dahlin) for a first-order-plus-dead-time process. Tuned by the desired closed-loop time constant τ_c . Supports scalar and matrix broadcasting; every zone runs an independent controller.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	error	float	yes		Control error $e = \text{setpoint} - \text{measured}$ (scalar or matrix)
value	K	float	no	1.0	Process steady-state gain (scalar or matrix)
value	tau_p	float	no	1.0	Process time constant in seconds (scalar or matrix)
value	T	float	no	1.0	Sample

					interval in seconds (scalar or matrix)
value	delay	float	no	0.0	Process dead time in seconds (scalar or matrix)
value	tau_c	float	no	1.0	Desired closed-loop time constant in seconds (scalar or matrix)
value	enable	bool	no	True	Run when true (scalar or matrix); disabled zones output 0.0
value	reset	bool	no	False	Clear this node's history when any element is true

DahlinControllerHistory

Dahlin dead-time compensating controller implemented with explicit per-zone history arrays (past errors and past outputs). Numerically identical to the standard Dahlin controller; the history arrays make the dead-time delay line explicit for debugging.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	error	float	yes		Control error $e = \text{setpoint} - \text{measured}$ (scalar or matrix)
value	K	float	no	1.0	Process steady-state

					gain (scalar or matrix)
value	tau_p	float	no	1.0	Process time constant in seconds (scalar or matrix)
value	T	float	no	1.0	Sample interval in seconds (scalar or matrix)
value	delay	float	no	0.0	Process dead time in seconds (scalar or matrix)
value	tau_c	float	no	1.0	Desired closed-loop time constant in seconds (scalar or matrix)
value	enable	bool	no	True	Run when true (scalar or matrix); disabled zones output 0.0
value	reset	bool	no	False	Clear this node's history when any element is true

MQController

MQ controller: one-step model-predictive move removing a fraction aggressiveness of the error each run, $du = Q \cdot K \cdot \alpha \cdot e / (Q \cdot K^2 + R)$, rate-limited by max_delta_u and integrated into the absolute output u. A plain PID/Dahlin-slot controller for generic loops; the CD profile version is ControllerNode. Supports scalar and matrix broadcasting; every zone runs an independent controller.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias

					to publish this node's output to
value	error	float	yes		Control error $e = \text{setpoint} - \text{measured}$ (scalar or matrix)
value	K	float	no	1.0	Process gain: a move du shifts the measurement by $K * du$; negative = reverse acting (scalar or matrix)
value	aggressiveness	float	no	0.7	Alpha: fraction of the error removed per move; lower with dead time (scalar or matrix)
value	Q	float	no	1.0	Error-cost weight (only the ratio R / Q matters) (scalar or matrix)
value	R	float	no	0.0	Move-cost weight; damps noisy loops, 0 = pure alpha / K form (scalar or matrix)
value	max_delta_u	float	no	0.0	Per-move rate limit applied after the solve; 0 = no limit (scalar or matrix)
value	u_start	float	no	0.0	Output at

					startup and after reset (scalar or matrix)
value	u_min	float	no	0.0	Output lower clamp (scalar or matrix)
value	u_max	float	no	100.0	Output upper clamp (scalar or matrix)
value	enable	bool	no	True	Run when true (scalar or matrix); disabled zones output 0.0 and hold their state
value	reset	bool	no	False	Return this node's output to u_start when any element is true

PIDNode

Proportional controller with enable gate. Supports scalar and matrix broadcasting: a single enable or gain applies to all elements; a shorter enable/gain array affects the prefix and leaves the rest enabled.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	error	float	yes		Control error (scalar or matrix)
value	enable	bool	no	True	Run when true (scalar or matrix)
value	gain	float	no	2.0	Proportional gain (scalar or matrix)

Debugger

PJanicePlot

Design-time plot probe. Passes input through unchanged while a companion widget displays it as profile, trend, heatmap, histogram, or FFT. Remove or leave in live mode for headless production runs.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	name	string	no		Display name shown on the node and widget
value	input	float	no		Value to pass through and plot (scalar or 1-row matrix)
value	plot_type	string	no	profile	profile / trend / heatmap / histogram / fft
value	chart_style	string	no	line	line or bar
value	chart_theme	string	no	light	dark or light chart background
value	ignore_zeros	bool	no	True	Exclude zeros from means, histograms, autoscale, and draw gaps
value	trend_points	int	no	100	Number of points to keep for trend plots
value	y_min	float	no		Fixed Y-axis lower limit (None = auto)
value	y_max	float	no		Fixed Y-axis upper limit

					(None = auto)
value	y_autoscale	bool	no	True	Autoscale the Y-axis (ignores y_min/y_max when true)
value	heatmap_rows	int	no	0	Heatmap row count (0 = auto)
value	heatmap_cols	int	no	0	Heatmap column count (0 = auto)
value	histogram_bins	int	no	20	Number of bins for histogram plots

PJanicePoker

Design-time value intercept. Passes input through by default; the companion widget can hold or one-shot inject a custom value. Remove or leave in pass-through mode for headless production runs.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	name	string	no		Display name shown on the node and widget
value	input	float	no		Value to pass through (scalar or 1-row matrix); leave unconnected to use as a source

Filter

ExponentialFilterNode

Single-pole exponential moving-average filter. $\alpha = 1.0$ passes the input through unchanged; α close to 0.0 heavily smooths the signal. The first real input seeds the output. None or empty inputs hold the last output.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	input	float	yes		Signal to filter (scalar or matrix)
value	alpha	float	yes	0.2	Smoothing factor ($1.0 =$ no filter, $0.01 =$ very heavy)
value	reset	bool	no	False	Clear filter history when true

MovingAverageFilterNode

Rolling moving-average filter. $\text{window} = 1$ passes the input through; larger windows average the last N real inputs. The first real input seeds the output. None or empty inputs hold the last output.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	input	float	yes		Signal to average (scalar or matrix)
value	window	float	yes	5.0	Number of samples in the rolling window ($1 =$ no filter)
value	reset	bool	no	False	Clear averaging history when

					true
--	--	--	--	--	------

SumNode

Sum multiple inputs element-wise. Inputs are read from a list of alias names. The node outputs None until every input has a real value, so missing externally-owned signals do not inject zeros into the sum.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	inputs	string	yes	[]	JSON list of alias names to sum (use the table editor)

Iterators

EdgesNode

Find the lowest or highest non-zero index in a 1-D profile, or a list of such indices for a 2-D array.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	src	float	yes		Input profile (1-D list) or list of profiles (2-D list-of-lists)
value	low_for_low est	bool	no	True	True = lowest non-zero index, False = highest non-zero index

ReverseNode

Reverse a list, or each inner list in a list-of-lists. When reverse is false the input passes through unchanged.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	inp	float	yes		Input list or list-of-lists to reverse
value	reverse	bool	no	True	True = reverse, False = pass through

Logic

AndNode

Logical AND of two boolean inputs. Supports scalar and matrix broadcasting: a single True leaves the other input unchanged; a shorter second input leaves the rest of the first input unchanged.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	bool	yes		First input (scalar or matrix)
value	b	bool	yes		Second input (scalar or matrix)

BumplessTransferNode

Copy one or more source aliases into target aliases when the gate is true, then write a done ordinal. Used for bumpless transfer at startup: snapshot actuator_positions into setpoint aliases so control starts from the current machine state. Check 'Src Also' on the gate input so the node runs when the startup ordinal changes.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish the transfer count to
value	gate	bool	yes	False	Enable the

					transfer (bind to a startup ordinal and check Src Also)
value	done_ordinal	string	no		Optional ordinal name to write when the transfer completes
value	done_value	bool	no	False	Value to write to done_ordinal
value	transfers	string	yes	[]	JSON list of [source_alias, target_alias] pairs (use the table editor)

CounterNode

Count the number of input aliases that are currently true / non-zero. Receives a trigger alias and reads all connected aliases from the alias table. Useful for testing multi-notifier parent chains.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	inputs	float	no	[]	Ignored: real inputs come from input_aliases

GreaterThanNode

Return true if $a > b$. Supports scalar and matrix broadcasting: a single b compares against all elements; a shorter b compares against the prefix and leaves the rest of the result False.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	float	yes		First value

					(scalar or matrix)
value	b	float	yes		Second value (scalar or matrix)

LessThanNode

Return true if $a < b$. Supports scalar and matrix broadcasting: a single b compares against all elements; a shorter b compares against the prefix and leaves the rest of the result False.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	float	yes		First value (scalar or matrix)
value	b	float	yes		Second value (scalar or matrix)

MultiplexNode

Select between two inputs. When `use_a` is true the node publishes `input_a`; otherwise it publishes `input_b`. Useful for switching between `mapped_profile_simple` and `mapped_profile_prbs` in the CD control chain.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	input_a	float	yes		First input (selected when <code>use_a</code> is true)
value	input_b	float	yes		Second input (selected when <code>use_a</code> is false)
value	use_a	bool	no	True	True = select <code>input_a</code> , False = select <code>input_b</code>

NotNode

Logical NOT of a boolean input. Supports scalar and matrix (1-row array).

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	bool	yes		Input (scalar or matrix)

OrNode

Logical OR of two boolean inputs. Supports scalar and matrix broadcasting: a single False leaves the other input unchanged; a shorter second input leaves the rest of the result unchanged.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	bool	yes		First input (scalar or matrix)
value	b	bool	yes		Second input (scalar or matrix)

SetOrdinalNode

Write a boolean value to the ordinal named by this node's output alias. Downstream nodes read the flag through an alias of source type 'ordinal' with the same name.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Ordinal name to write (e.g. machine_speed_good)
value	x	bool	yes		Boolean value to latch into the ordinal

Math

DeltaNode

Subtract a reference from every element. The reference can be supplied as `standard` or, if empty, computed as the mean of the input. With include_zeros off, zeros are ignored when computing the mean and are left as zero in the output.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	input	float	yes		Input list or list-of-lists
value	standard	float	no		Reference value/list to subtract; empty = use input mean
value	include_zeros	bool	no	True	True = include zeros in mean and delta; False = ignore zeros

DivideNode

Divide two numeric inputs (a / b). Supports scalar and matrix broadcasting: a single divisor divides all elements; a shorter divisor affects the prefix and leaves the rest of the first factor unchanged. Division by zero returns 0.0.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	float	yes		Dividend (scalar or matrix)
value	b	float	yes		Divisor (scalar or matrix)

InvertNode

Invert the sign of every element: positive becomes negative and vice versa.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	input	float	yes		Scalar, list, or list-of-lists to invert

MeanNonZeroNode

Mean of the non-zero elements of a profile. While the gate input is false the node holds and republishes its last value instead of recomputing.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	source	float	yes		Profile to average; zero elements are excluded
value	gate	bool	no	True	Recompute only when true; otherwise hold the last value

MultiplyNode

Multiply two numeric inputs. Supports scalar and matrix broadcasting: a single factor multiplies all elements; a shorter second factor affects the prefix and leaves the rest of the first factor unchanged.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	a	float	yes		First factor (scalar or matrix)
value	b	float	yes		Second factor (scalar

					or matrix)
--	--	--	--	--	------------

SubtractNode

Compute the difference between two inputs. Supports scalar and matrix broadcasting: a shorter setpoint is applied to the prefix; remaining elements keep the measured value.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	pv	float	yes		Measured value (scalar or matrix)
value	setpoint	float	yes		Target value (scalar or matrix)

Sinks

ValveOutputNode

Publish a value to a subscription alias (the final output of a chain). Accepts a scalar or a matrix (1-row array).

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	value	float	yes		Value to publish (scalar or matrix)
subscription	valve	float	yes		Output subscription alias

WriteNode

Publish a value to an external alias (Multiverse topic). Use this as the final output of a chain. Accepts a scalar or a matrix (1-row array).

Kind	Name	Type	Required	Default	Description
value	value	float	yes		Value to

					publish (scalar or matrix)
subscription	target	float	yes		External alias / topic to write to

Sources

ConstantNode

Output a constant value. Accepts a scalar or a matrix (1-row array).

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	value	float	no	0.0	Constant value (scalar or matrix)

NotifierNode

Subscribe to an alias and trigger downstream nodes when it changes. Passes through the latest value of the monitored alias so downstream nodes can read it from the output alias or internal node output.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
subscription	source	float	yes		Alias to monitor

Statistics

LinearRegressionNode

Fit a straight line $y = \text{slope} * x + \text{intercept}$ to two equal-length sequences. Publishes [slope, intercept] on output_alias and, if set, on slope_output_alias and intercept_output_alias. Returns None if there are fewer than min_points real values or if x/y lengths differ.

Kind	Name	Type	Required	Default	Description
------	------	------	----------	---------	-------------

value	output_alias	string	no		Unique alias to publish [slope, intercept] to
value	x	float	yes		X values (list)
value	y	float	yes		Y values (list, same length as x)
value	min_points	float	no	2.0	Minimum number of points required
value	slope_output_alias	string	no		Optional alias to publish the slope to
value	intercept_output_alias	string	no		Optional alias to publish the intercept to

TwoSigmaNode

Compute twice the standard deviation (2-sigma) of the input values. Returns a single scalar.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	values	float	yes		Input values (1-row matrix)
value	include_zeros	bool	no	True	If false, zero values are excluded from the calculation

Timer

TimerAlarmNode

Fire once per day at each configured wall-clock time. Outputs the time as an HHMM integer (e.g. 800 for 08:00).

Kind	Name	Type	Required	Default	Description
------	------	------	----------	---------	-------------

value	output_alias	string	no		Unique alias to publish this node's output to
value	times	string	yes		Comma-separated alarm times, e.g. 0800,1200,1600 or 08:00,12:00,16:00

TimerIntervalNode

Fire repeatedly at a fixed interval in milliseconds and output the interval value. Minimum interval is 100 ms.

Kind	Name	Type	Required	Default	Description
value	output_alias	string	no		Unique alias to publish this node's output to
value	interval_ms	int	yes		Interval in milliseconds between fires (minimum 100 ms)

Index

The following index lists the key concepts, panels, menus, and algorithms in this manual. For a generated Word index, right-click the field below and choose Update Field.

Right-click and choose Update Field to build the index.

A

- Alias — see Aliases & Declarations; also used in parameter bindings and the Alias Table.
- Alias Table — Edit → Alias Table; non-modal cross-reference table for constants, ordinals, external sources, and node outputs.
- Algorithm — a Python class implementing one node type; see Nodes & Algorithms and Available Algorithms.
- Algorithm Builder — Edit → Algorithm Builder; scaffold tool for new node algorithms.
- Algorithm Reference — appendix with full contract tables for every algorithm.

B

- Block — see Canvas & Interaction → Blocks, ports, and trigger semantics.
- Bridge — see Communication & Bridges.
- Bridge plugin — see Communication & Bridges → How bridge plugins work and Building a new bridge plugin.
- Bridges manager — Communication → Bridges.

C

- Canvas — the main node-graph workspace; see Canvas & Interaction.
- CD control — see CD Control Example (cdcontrol.json).
- Connection modes — source only, source + trigger, trigger only; see Canvas & Interaction → Blocks, ports, and trigger semantics.
- Constant — alias source type for fixed literal values.

D

- Declaration — strict ordinal/subscription declaration; see Aliases & Declarations → Declarations.
- Display name — editable node label; press F2 or Edit → Edit Node Description.

E

- External alias — alias bound to a bridge source, qualified as bridgename://source.

F

- F1 — context-sensitive help; opens algorithm help from a node or menu help from a widget.
- F2 — edit the selected node's display name and description.

- Firing Log — stream debugger; see Runtime & Debugging → Firing Log.

G

- Git — version-control menu plugin; see Menu Reference → Git.

I

- Index — this section.
- Inspector — see Canvas & Interaction → The Inspector panel.
- Internal alias — auto-generated output alias in the form @node_id_out.

L

- Log panel — see Runtime & Debugging → Log panel and Runtime Errors.

M

- Menu Reference — chapter listing every menu and action.
- MQController — generic single-loop MQ (one-step MPC) controller; see Available Algorithms → Controllers.

N

- Node — see Nodes & Algorithms and Canvas & Interaction.
- NotifierNode — source node that triggers a chain when an external alias updates.

O

- Ordinal — boolean flag alias used for gating and latching state.
- Output alias — published node output; see Nodes & Algorithms → Published output aliases.

P

- Palette — see Nodes & Algorithms → Palette.
- PJanicePoker — debug probe; see Available Algorithms → Debug probes in depth.
- PJanicePlot — debug plot probe; see Available Algorithms → Debug probes in depth.
- Preferences — Edit → Preferences; see Preferences & Shortcuts.
- Problems dialog — see Runtime & Debugging → Problems dialog.

R

- Runtime Errors — Log dock tab; see Runtime & Debugging → Log panel and Runtime Errors.

S

- Search — Edit → Search (Ctrl+F), blank-canvas right-click → Search, Alias Table right-click → Search, or Palette right-click → Search; highlight nodes by algorithm, display name, or alias pattern.
- Source Browser — see Communication & Bridges → Source Browser.
- Source-only connection — value binding that does not wake the downstream node.

- Square-up — Tools → Square-up; automatic layout and wire routing.
- Src Also — Inspector checkbox that makes a bound source also trigger the node.
- Staged changes — green outlines on nodes/wires changed since the last save.

T

- Table of Contents — at the front of the manual; right-click and update in Word.
- Trigger — wakes a node; see Canvas & Interaction → Blocks, ports, and trigger semantics.
- Trigger port — top diamond port on every non-source node.

V

- Value input port — left-side circle that binds a value to a parameter.
- View menu — see Menu Reference → View.

W

- Watch panel — see Runtime & Debugging → Watch panel.
- Wire routing — direct (shortest) or XY (orthogonal); see Canvas & Interaction.