designer_action Reference

10.1.5 additions to `designer_action` and related DesignerMCP tools — new `build` action parameters, compile feedback on write/read responses, and a `context_menu` action for grid right-click verification.

10.1.5 adds build-visibility signals to the DesignerMCP surface. The `build` action gains a `rebuild_all` parameter mirroring the `RuntimeBuildAndPublish` dialog's *Rebuild All* checkbox and a `comment` parameter recorded in build history, `write_objects` responses carry per-row `compile` feedback for the four tables that hold user-authored .NET, `get_objects(detail='full')` surfaces a `lastCompile` field on those same rows, and the solution context carries a compact `build_state` section.

The JSON shape returned by the build surface is defined once on MCP SDK Reference — the "`build` Block" section there is the authoritative shape. This page covers only the DesignerMCP-specific entry points.

designer_action('build', …) — new parameters
write_objects — per-row compile field
get_objects(detail='full') — lastCompile field
SolutionContext / RefreshContext — build_state section
designer_action('context_menu', …) — grid right-click verification
Grid clipboard actions: copy_rows, paste_rows, select_all_rows, commit_edit, cancel_edit
list_wizard_options and pick_wizard_variant — wizard symbols
set_object_value — live Object Model write
Error Codes

---

designer_action('build', …) — new parameters

The `build` action invokes the same pipeline as the Designer "Build and Publish" dialog. In 10.1.5 it accepts a `rebuild_all` parameter mirroring the dialog's *Rebuild All* checkbox, and a `comment` parameter that annotates the build-history entry.

```
designer_action(
  action:        "build",           // the verb (required)
  options:       str  = null,       // the verb's argument; unused by 'build'
  rebuild_all:   bool = false,      // false = incremental, true = full rebuild
  comment:       str  = null,       // free-text note recorded in build history
  solution_id:   str  = null,       // REQUIRED routing key (see below)
  solution_name: str  = null        // fallback address by solution name
) -> { build: <block> }
```

These are the parameters of the `designer_action` tool as a whole; `rebuild_all` and `comment` apply only to the `build` action and are ignored by every other verb.

Parameter	Default	Effect
action	— (required)	The verb to perform — `build`, `navigate`, `start_runtime`, `context_menu`, and so on. Also accepts any value listed in a previous response's `tabActions[]`.
options	`null`	The verb's argument. Shape varies per verb; multi-field arguments pack with `|`. The `build` action takes no argument.
rebuild_all	`false`	When `false`, only objects modified since the last build are recompiled; unchanged objects appear in `summary.skipped`. When `true`, every object in the four affected tables is recompiled regardless of modification state. Mirrors the *Rebuild All* checkbox in the Build and Publish dialog. Each call is authoritative: the value does not inherit the Designer checkbox's state and does not alter it.
comment	`null`	Free-text note captured in the solution's build history (the `SolutionHistory` table) alongside the build entry. Ignored for non-build actions.
solution_id	`null`	Required routing key — the `solution_id` returned by `open_solution` / `create_solution`. It selects which Designer instance receives the call. Omitting it (with no `solution_name`) fails with `MISSING_SOLUTION_ID`.
solution_name	`null`	Fallback address by solution name, for when a `solution_id` has been invalidated by a sibling session. Either identifier satisfies the routing requirement.

There is no `validate_displays` parameter. Display validation renders every XAML display during the build, which takes minutes, so it remains a user-only operation in the Build and Publish dialog and is not exposed on the MCP surface. A `build` issued through `designer_action` always compiles the Scripts tables and `DisplaysList` code-behind on the standard pipeline.

The response is the shared `build` block. See MCP SDK Reference for the full shape.

```
# Fast path during iterative development
designer_action('build', solution_id=SID)
# == designer_action('build', rebuild_all=false, solution_id=SID)

# Pre-deploy check - recompile everything
designer_action('build', rebuild_all=true, solution_id=SID)

# Annotate the build history entry
designer_action('build', rebuild_all=true, comment='pre-release smoke', solution_id=SID)

# SID is the solution_id returned by open_solution / create_solution.
# Where that id has been invalidated, address the session by name instead:
designer_action('build', solution_name='MyPlantSolution')
```

---

write_objects — per-row `compile` field

When a `write_objects` call touches a row in one of the four affected tables (`ScriptsTasks`, `ScriptsClasses`, `ScriptsExpressions`, `DisplaysList`), the response entry for that row carries a `compile` field populated from the save-time incremental compile.

```
{
  "results": [
    {
      "table": "ScriptsTasks",
      "name":  "Line1_Cycle",
      "status": "written",
      "compile": {
        "status": "error",
        "diagnostics": [
          { "line": 12, "msg": "The name 'tagx' does not exist in the current context" }
        ]
      }
    },
    {
      "table": "ScriptsClasses",
      "name":  "TankUtils",
      "status": "written",
      "compile": { "status": "ok", "diagnostics": [] }
    }
  ]
}
```

Writes to rows outside the four affected tables carry no `compile` field — those objects do not participate in the .NET compile pipeline.

From FrameworX 10.1.6, the compile of a `DisplaysList` row covers the display's tag and object-model bindings and expressions, not only its code-behind, so the write reply names an unknown tag or a broken binding. Read `status` and `diagnostics[]` together: `diagnostics[]` carries warnings on an `ok` row, and `ok` means no errors were recorded, not proof the compile ran. AI Console runs no compiler on write.

The row-level `compile` reports only what the incremental pipeline saw for that single object. Cross-object breakage — for example, an edit to a `ScriptsClasses` row that breaks a `ScriptsTasks` row referencing it — is surfaced by a subsequent `designer_action('build')` call, not by the write response alone.

---

get_objects(detail='full') — `lastCompile` field

When `get_objects` is called with `detail='full'` against one of the four affected tables, each returned row carries a `lastCompile` field holding the most recent compile result for that object.

```
{
  "table": "ScriptsTasks",
  "name":  "Line1_Cycle",
  "code":  "...",
  "lastCompile": {
    "status": "ok",
    "at":     "2026-04-21T16:02:11Z",
    "diagnostics": []
  }
}
```

Field	Meaning
`lastCompile.status`	`ok` or `error`, matching the shared `build` block convention.
`lastCompile.at`	ISO-8601 timestamp of the compile that produced this result.
`lastCompile.diagnostics[]`	Same shape as elsewhere — `line` and `msg` per entry, empty when status is `ok`.

Calls with `detail='summary'` or the default detail level do not include `lastCompile` — it is only emitted on the full-detail path to keep the summary response compact.

SolutionContext / RefreshContext — `build_state` section

The context packages returned to the agent on connect and on refresh now include a compact `build_state` section. This gives the agent a one-shot view of compile health without needing to call `designer_action('build')` at session start.

```
{
  "build_state": {
    "built":     42,
    "failed":    1,
    "skipped":   0,
    "timestamp": "2026-04-21T16:02:11Z"
  }
}
```

The fields match the `summary` portion of the shared `build` block. Per-object diagnostics are not included here — callers that need them issue `designer_action('build')` or `get_objects(detail='full')` on the affected table.

---

designer_action('context_menu', …) — grid right-click verification

The `context_menu` action drives a row's right-click context menu through the same UI surface a Designer operator would use. It serves automated verification workflows that previously needed a person for any menu-driven step (Tag editor rows, Security Secrets, Devices / Datasets grids, and every other table whose row carries actions accessible only via right-click).

Two modes, discriminated by the presence of a `|` separator in the options string:

```
# Inspect: list the items the context menu would show for this row.
designer_action('context_menu', '<TableType>.<RowName>', solution_id=SID)
# Invoke: fire the named menu item on this row.
designer_action('context_menu', '<TableType>.<RowName>|<MenuAction>', solution_id=SID)
```

Inspect form

Returns the snapshot of the menu items after the page's `ContextMenu.Opened` gates fire against the selected row. Items reflect the post-gate state, so visibility and enablement match what a user would see on right-click.

```
{
  "mode": "inspect",
  "table": "SecuritySecrets",
  "row":   "AdminPass",
  "items": [
    { "header": "Copy SecretValue to clipboard",
      "tag":    "TDEV1412_CopySecretValue",
      "isEnabled": true,
      "isVisible": true }
  ]
}
```

Invoke form

Matches `MenuAction` against **`MenuItem.Tag`** first (case-sensitive — Tags are programmer-set stable identifiers), then against **Header text** (case-insensitive). Fires `RaiseEvent(MenuItem.ClickEvent)` on the matched item; the response carries any error the Click handler raised.

```
{
  "mode": "invoke",
  "table": "SecuritySecrets",
  "row":   "AdminPass",
  "match": { "matchedBy": "tag", "tag": "TDEV1412_CopySecretValue" },
  "status": "ok"
}
```

Prefer Tag matching for stable AI scripts — Header text often carries localization or accelerator markers that drift release to release. Tags are author-set in code-behind and stay stable across cosmetic UI changes.

> **Click handlers must not call `ShowDialog` inline.** The verb invokes the Click handler synchronously on the WPF Dispatcher. A handler that opens a modal dialog inline deadlocks the MCP pipe thread. Day-1 wired items (e.g., `SecuritySecrets.CopySecretValue`) only mutate clipboard / object state — safe. If a future Click handler needs to surface a modal, refactor it to `BeginInvoke` the dialog so the menu invocation completes first.

Page coverage (v1)

v1 wires only pages that expose the canonical `GridControlObj` element name — every configuration grid (Tags, Devices, Alarms, Datasets, Security, Reports, etc.). Display editors and the project tree return `PAGE_NOT_SUPPORTED`; future releases extend the verb's surface area.

Error Codes

Code	When
`MISSING_PARAMETER`	The options string is missing.
`INVALID_PARAMETER`	Options string does not match `<TableType>.<RowName>[|<MenuAction>]`.
`ROW_NOT_FOUND`	The named row does not exist in the table on the active solution.
`NO_DISPATCHER`	WPF dispatcher unavailable (Designer not interactive).
`NO_ACTIVE_PAGE`	No page is currently active in Designer for that table.
`PAGE_NOT_SUPPORTED`	Page does not expose the canonical `GridControlObj` grid (Display editors, project tree, etc.).
`NO_DATA_GRID`	The page exposes `GridControlObj` but it does not host a data grid.
`NO_CONTEXT_MENU`	The grid does not have a context menu attached.
`MENU_ITEM_NOT_FOUND`	Invoke form: no menu item matched `MenuAction` by Tag or Header.
`MENU_ITEM_GATED`	Invoke form: matched item is disabled or hidden under the active permissions. Response includes `isEnabled` / `isVisible` for diagnostics.

CLICK_HANDLER_EXCEPTION	The Click handler raised an exception. The original message is surfaced in the response.
EXCEPTION	Any other unexpected failure.

Grid clipboard actions: copy_rows, paste_rows, select_all_rows, commit_edit, cancel_edit

From FrameworX 10.1.6, designer_action drives the clipboard of the active configuration grid the way a person does. The copied rows travel inside the MCP response. The Windows clipboard is not involved. Cut is not offered.

Action	Options	Effect
select_all_rows	(none)	Selects every row of the active grid. Returns itemCount and the selectedCount read back from the grid. Returns INLINE_EDITOR_OPEN while an inline editor is open.
copy_rows	(none)	Copies the selected rows and returns the payload in the response, in the grid's own copy format. Password and secret columns are emptied and named in credentialColumnsRedacted. On the Identity Providers page the AuthOptions column is emptied and named as well. escapedFormat=true marks a page using the escaped row format. Returns COPY_NOT_ALLOWED when nothing is selected. On DisplaysSymbols and DisplaysImages, use display_clipboard_export.
paste_rows	Payload text, or {"payload":"...","insert":bool,"escaped":bool}	Feeds the payload to the grid's own paste. With insert=false (default) it overwrites the selected rows in order and adds extra lines as new rows. With insert=true or no selection, every line becomes a new row. For a payload from copy_rows, set escaped to its escapedFormat value. A payload with a backslash and no escaped value returns ESCAPED_FORMAT_UNSPECIFIED. The paste never writes password or secret columns, nor AuthOptions on the Identity Providers page. An overwritten row keeps its stored value, and credentialColumnsIgnored names the skipped columns. The response reports overwrittenRows (id, keyBefore, keyAfter), createdRowKeys, overwrittenCount and createdCount. Returns PASTE_NOT_ALLOWED when the page disallows paste or is read-only for the signed-in user. Confirm the values with get_objects.
commit_edit / cancel_edit	(none)	Commits or discards the open inline cell editor, for example the one invoke_command('Rename') opens. Returns NO_INLINE_EDITOR when none is open. commit_edit reports rowState and persisted. cancel_edit works in read-only mode.

get_state(target='designer') reports the grid state in two fields: gridSelection (count and rowKeys of the selected rows) and inlineEditor (present only while an inline editor waits for input). An unrecognized action returns the list of valid actions.

---

list_wizard_options and pick_wizard_variant — wizard symbols

These two paired tools drive the five **wizard symbols** — BLOWER, MOTOR, PUMP, TANK, VALVE. Each is a parameterized graphical symbol whose appearance is assembled from named segments (appearance, orientation, feet, output, and others depending on the symbol). They have nothing to do with solution templates: list_wizard_options reports the legal option values for one symbol type, and pick_wizard_variant resolves a set of chosen options to a concrete variant and applies it to the open solution.

Flow

1. Call list_wizard_options with the symbol type. It returns segments[] (each with name, label, and legal options[]), constraints[] (cross-segment guards), defaultVariant, and labelContract. This is pure data — no Designer session is required.
2. Call pick_wizard_variant with the same symbol type and a JSON option map. The platform applies the cross-segment constraints, resolves the canonical variant name, verifies it exists in the library, and writes the symbol into DisplaysSymbols.
3. Omitting target_symbol_name creates a new symbol instance, auto-named PUMP1, PUMP2, and so on. Supplying it replaces the variant of that existing symbol in place.

```
# 1. Discover the segment/option taxonomy for one symbol type
list_wizard_options(wizard_type='PUMP')
# -> { segments: [ { name: "appearance", label: "Appearance",
#                    options: [...] }, ... ],
#      constraints: [...], defaultVariant: "...", labelContract: {...} }

# 2. Create a new PUMP symbol with the chosen options
pick_wizard_variant(
  wizard_type='PUMP',
  options='{"appearance":"Detailed","orientation":"Right","feet":"None","output":"Top"}',
  solution_id=SID)
# -> a new symbol (PUMP1, PUMP2, ...) in DisplaysSymbols

# 3. Or replace the variant of a symbol that already exists
pick_wizard_variant(
  wizard_type='PUMP',
  options='{"appearance":"Simple","orientation":"Left"}',
  target_symbol_name='PUMP1',
  solution_id=SID)
```

Option values accept either the human label (`Detailed`) or the canonical token (`DET`), and any segment left out of the map falls back to that wizard's default. The `options` argument is a JSON *object* passed as a string; any other JSON shape is rejected with `INVALID_OPTIONS_SHAPE`, and an unrecognized symbol type with `INVALID_WIZARD_TYPE` plus the list of valid ones.

`pick_wizard_variant` writes to the solution — it is not read-only. The symbol lands in `DisplaysSymbols`; placing it on a Display is a separate `DisplaysList` write. `list_wizard_options` writes nothing and can be called safely at any time. Note that `pick_wizard_variant` is DesignerMCP-only in this release; the ConsoleMCP surface returns `DESIGNER_MCP_ONLY`.

---

set_object_value — live Object Model write

`set_object_value` writes a single row in the live Object Model — a tag value, an attribute, or any other runtime-writable object exposed under the Object Model roots. It is distinct from `write_objects`: `write_objects` targets the solution's **configuration tables** (Tags, Devices, ScriptsTasks, etc., as authored in Designer), while `set_object_value` targets **live runtime state** on the connected solution.

Tool	Targets	Persistence
`write_objects`	Configuration tables (the solution graph)	Persists to the `.dbsln`; survives runtime restart.
`set_object_value`	Live Object Model rows in the running runtime	Affects in-memory runtime state; persistence depends on the target row's retention semantics (retentive vs. volatile).

Use `set_object_value` when an AI workflow needs to drive a control loop, exercise a tag-bound dynamic, or seed runtime state for a smoke test against a running solution. Use `write_objects` when the change should land in the saved solution and survive a restart.

```
# Drive a live tag value in the running runtime.
# The path is 'path', and the value is always passed as a string -
# the kernel coerces it to the target object's actual type.
set_object_value(path="Tag.Plant1/Line1/Setpoint", value="72.5", solution_id=SID)

# Booleans use the string forms 'true' / 'false'
set_object_value(path="Tag.Pump1.Running", value="true", solution_id=SID)
```

The AI Runtime connector (RuntimeMCP) has no `set_object_value` tool and no general write to live values. From FrameworX 10.1.6, it offers `runtime_set_test_value`, a Development-only tool for automated tests in disposable test solutions. See Runtime MCP Reference. On the DesignerMCP surface, `set_object_value` requires a runtime to be bound to the active Designer session; calls against a solution that has no running runtime return an error.

Error Codes

Code	When	Recovery
`BUILD_IN_PROGRESS`	Another `build` call is already running for the active solution.	Wait for the in-flight call to complete. The second call does not queue. Poll `build_status` rather than re-issuing `build`.
`MISSING_SOLUTION_ID`	Neither `solution_id` nor `solution_name` was supplied.	Pass the `solution_id` returned by `open_solution` / `create_solution`.
`UNKNOWN_SOLUTION_SESSION`	The identifier supplied does not resolve to a registered Designer session — the Designer was closed, or the id is stale.	Call `open_solution` to establish a session, or retry addressing it with `solution_name`.
`MISSING_PARAMETER`	No `action` was supplied.	Response includes a `validActions` list.
`INVALID_PARAMETER`	A parameter carries a value the verb cannot accept — for example `rebuild_all` is not a boolean.	Response includes an `examples` array with the correct call form.

In this section...