Display Design Best Practices

Display Design Best Practices

Best practices and known patterns for creating displays, symbols, and dashboards via MCP.

Table Types

Use `DisplaysList` for creating and editing displays. `DisplaysDraw` is the Designer visual editor tab — not a writable table. Similarly, use `DisplaysSymbols` for symbols. Always call `get_table_schema('DisplaysList')` for the display schema.

Display vs Symbol Sizing

Displays use `Width` and `Height` properties directly on the display object. Symbols use a `Size` property with format `"Width,Height"` (e.g., `"400,300"`). Do not mix them — `Size` on a display is ignored, `Width`/`Height` on a symbol are ignored.

Canvas Positioning

Every element on a Canvas display needs explicit position:

- `Left` — horizontal offset from left edge (pixels)
- `Top` — vertical offset from top edge (pixels)

Elements without position stack at (0,0). For a 1920×1080 display, typical layout starts elements at Left=20, Top=20 with spacing.

Dashboard Cell Format

Dashboard displays use a flat `Cells` array. Each cell needs:

- `Column` and `Row` — zero-based grid position
- `ColumnSpan` and `RowSpan` — how many cells to span (default 1)
- `Elements` — array of controls within that cell

```
{
  "Type": "DashboardDisplay",
  "Columns": 3,
  "Rows": 2,
  "Cells": [
    { "Column": 0, "Row": 0, "Elements": [ ... ] },
    { "Column": 1, "Row": 0, "ColumnSpan": 2, "Elements": [ ... ] }
  ]
}
```

Do not create CellGroup, Grid, RowDefinition, or ColumnDefinition objects — those are internal WPF representations. The importer builds them from the flat Cells format.

Controls That Cannot Be Written

Some controls are read-only and will be rejected by `write_objects`. Check the schema — if `writeSupported` is `false`, do not include that control. The import will return an error.

Current read-only controls include: Image (use IndustrialIcon or shapes instead), RadioButton, WPF Control, Blazor Control, ToggleButton, MapsESRI, MapsGMap, MapsOSM, FlowPanel.

When a write is rejected, the error message suggests using `get_table_schema` with the specific control name for alternatives.

IndustrialIcon

Use the `IconName` property with a human-readable name like `"Home"`, `"Valve"`, `"Pump"`, `"Tank"`, `"Motor"`. Do not set FontFamily, FontSize, or Text — those are internal properties managed by the import pipeline.

Call `get_table_schema('IndustrialIcon')` to see the full list of available icon names.

LinkedValue — The Universal Data Binding

`LinkedValue` is the primary data-binding property for almost every control. It accepts tag paths with the `@` prefix: `"@Tag.FolderPath/TagName"`. The `@` prefix is required to mark the value as a tag reference; a bare `"Tag.X"` (without `@`) parses as a literal identifier and fails resolution silently at runtime.

On text-display elements (TextBlock, Label, Button.LabelLink, Button.CaptionLink), `LinkedValue` also supports composite-template form for labeled values: `"Temperature: {@Tag.Reactor/Temp} °C"`. The `@` prefix MUST appear inside the curly braces.

For Button captions, use `LabelLink` (not `Text`). `Text` is design-time fallback only and does NOT interpolate `{...}` or live-bind `@Tag.X` at runtime — it renders both as literal text. The importer auto-promotes a non-empty Text to LinkedValue when LinkedValue is empty, but emit `LinkedValue` directly for clarity.

Exception: BarChart data items use `FieldNameValue` instead of `LinkedValue`.

Four-Family Rule for *Link Properties

Every `StringWithExpressions`-typed property — `LinkedValue`, `ObjectValueLink`, `VisibleLink`, `DisableLink`, `LabelLink`, `CaptionLink`, and every other `*Link` on dynamics — accepts four value families. Pick by intent.

Family	Form	Example
Numeric literal	Bare digits	`"1"`, `"100"`, `"3.14"`, `"-1"`
Boolean literal	Bare keyword	`"true"`, `"false"`
Tag / expression reference	`@` prefix	`"@Tag.Setpoint"`, `"@Tag.Mode == 1"`
String literal	Escape-wrapped quotes	`"\"Dark\""` — writes the literal four-character string `Dark`

A bare `"Dark"` (no `@`, no quote-wrap) parses as a tag/object reference, fails resolution, and the action silently no-ops with no error logged. The escape-wrap is JSON-source — the on-disk in-memory string is `"Dark"` (six characters including the quotes); the parser sees the inner quotes and recognizes a string literal.

Composite template form (on `LinkedValue`, `LabelLink`, `CaptionLink` for text-display elements only): `"Label: {@Tag.X} unit"` — literal text outside braces; `@Tag.X` inside braces with the `@` prefix required.

Dynamics

Dynamics add runtime behavior to any element. Each dynamic is a separate JSON object within the element's `Dynamics`array.

```
{
  "Type": "Ellipse",
  "Width": 40,
  "Height": 40,
  "Dynamics": [
    {
      "Type": "FillColorDynamic",
      "LinkedValue": "@Tag.Process/Status",
      "ChangeColorItems": [
        { "Type": "ChangeColorItem", "ChangeLimit": 0, "LimitColor": "Gray" },
        { "Type": "ChangeColorItem", "ChangeLimit": 1, "LimitColor": "Green" }
      ]
    }
  ]
}
```

Key rules:

- Always use `list_dynamics` for the specific dynamic type to see its exact property names
- `ColorDynamic` is abstract — use `FillColorDynamic`, `LineColorDynamic`, or `TextColorDynamic`
- `ShineDynamic` controls glow/shine effects — it does not have a `LinkedValue` property
- `ActionDynamic` properties use internal names: `ObjectLink`, `ObjectValueLink`, `ActionType`

Collections (Pens, Columns, Data Items)

Controls with child collections (TrendChart pens, DataGrid columns, PieChart items) use wrapper arrays. The wrapper needs a `Type` that names the collection class:

```
{
  "Type": "TrendChart",
  "Pens": {
    "Type": "TrendPenList",
    "Items": [
      { "PenName": "Temperature", "TagName": "@Tag.Process/Temp" }
    ]
  }
}
```

Collection types: `TrendPenList`, `PieChartDataList`, `GridColumnList`, `ColumnDataList`, `TimelineColumnList`.

Do not set `Id` on collection items — it is auto-managed by the engine.

Themes

To apply a theme color, use the `"theme:ColorName"` prefix on any color property:

```
{ "Fill": "theme:Accent1", "Foreground": "theme:Text" }
```

To clear a theme (revert to explicit color), set the color to any explicit value like `"#FF0000"` — the engine removes the theme binding automatically.

Shapes

All shapes require `StrokeThickness` (default is usually 1 but should be explicit). Common shapes: Ellipse, Rectangle, Line, Polyline, Path, Border.

Use `Fill` for interior color, `Stroke` for border/outline color.

Groups and Containers

- `Group` — groups child elements that move together. Children go in `ChildElements` array.
- `Border` — a container with optional border and background. Single child in `Child` property.
- `ChildDisplay` — embeds another display. Use `DisplayName` property to reference the target display.

Layouts and Navigation

The MainPage display (ID=0 in DisplaysList) is the default display every solution starts with. Configure it as the main process overview. The Startup layout (ID=0 in DisplaysLayouts) controls which displays appear on startup.

Use `get_table_schema('DisplaysLayouts')` for layout structure. Layouts define regions; displays fill those regions.

Verification After Creating

After calling `write_objects`, the success response is your primary confirmation. Do not enter a screenshot verification loop. At most, take one screenshot to confirm visual rendering, then deliver your report and let the user review.

If something looks wrong in a screenshot, mention it in the report — do not silently retry or investigate further.