UNS UserTypes Reference (10.1.5 draft)

New for 10.1.5 (draft preview). This is a preview of how the parent page will look in FrameworX 10.1.5 (planned April 2026). The parent page is the current 10.1.4 version. Content under review.

Create reusable equipment templates (UserTypes).

Reference Modules UNS UI Asset | Tags | UserTypes | Enumerations | Services | Monitor

---

UNS UserTypes (Reference), also referred as UDTs, or DataTemplates, extend the platform's predefined data types by defining custom types with properties tailored to business needs, enabling modeling of solution-specific data structures like equipment status and asset attributes. A UserType provide:

- Custom data type definitions
- Reusable tag structures
- Hierarchical data modeling
- Consistent asset representation
- Property inheritance
- Template-based tag creation

UserTypes allow creating complex data structures that standard types cannot represent, such as pumps, motors, tanks, or production lines.

Ontology Metadata (10.1.5+)

Every UserType carries ontology metadata alongside the familiar configuration properties. Import populates these columns from RDF/OWL source files, export serializes them back to triples, and semantic search ranks matches across them. See Industrial Ontology Integration How-to for the full model.

Column	Where	Storage	UI	Round-trip
DisplayText	UserType row	Text, 255 chars	UserType properties grid	`rdfs:label` (primary) on the class IRI
Labels	UserType row	Text, unbounded, semicolon-delimited	UserType properties grid, label-chips	`skos:altLabel` per entry
SourceIri	UserType row	Text, unbounded	UserType properties grid	Subject IRI for class triples
BaseUserType	UserType row	Name reference to another UnsUserTypes row (resolved Name ID at write time). Enables inheritance. Locked cells use `LockedState=BaseType` semantics.	UserType properties grid (dropdown)	`rdfs:subClassOf <parent IRI>`
Attributes	UserType row	JSON column, unbounded. **Design-time only** at UserType level (class-level TBox annotations).	UserType properties grid, expandable JSON editor	Individual triples per `prefix:localName` (language-tagged for `@lang`-suffixed keys)

Member-def ontology columns (10.1.5+)

The UDT member grid gained two ontology columns in 10.1.5:

Column	Where	Storage	UI	Round-trip
SourceIri	Member-def row	Text, unbounded	UDT member grid column	Predicate IRI for the property declaration
Attributes	Member-def row	JSON column, unbounded. Design-time only.	UDT member grid (Attributes column)	Predicate-level annotations on the property declaration

Why no Labels column on member-def: property-level synonyms (`skos:altLabel` on data properties) are rare in real industrial ontologies (IOF, ISA-88, ISA-95) — altLabels cluster at the class level. `DisplayText` already covers the primary human-readable member name. Member-def Labels was considered and dropped in the 2026-04-22 settlement.

Mutability & retention

The ontology columns follow a **class vs. instance** split that matches W3C + industrial-ontology conventions: **IRIs are identity (immutable everywhere); Labels are `skos:altLabel` synonyms (edited on the class at design time, on tag instances at runtime); Attributes is an extensibility bag (design-time at class + member-def levels, runtime-mutable only at tag level)**.

Layer	SourceIri	Labels	Attributes
UserType (this page)	Design-time · immutable identity	Design-time	Design-time
Member-def (this page)	Design-time	Not modeled	Design-time
Tag instance (see UNS Tags Reference (10.1.5 draft))	Design-time · immutable	Runtime-editable · retentive	Runtime-editable · retentive

Bindings can *read* all three columns on every layer; *writes* go through script (`@Tag.X.SetAttribute(...)`), the Designer editors, or a future tag-level operator UI — never through bindings directly. For RDF/OWL export, per-tag runtime mutations on `Labels` / `Attributes` must be snapshotted back to the cold-start data with `promote_retentive_attributes` before running the exporter.

AI / MCP page action (10.1.5+)

When Designer is navigated to the UserTypes page, the following action is exposed via the `designer_action` MCP tool:

- `designer_action("relationship_graph")`. Mirrors the **Open Visual Graph** toolbar button. Generates an interactive HTML view (pan, zoom, filter via cytoscape.js) and opens it in the default browser. A Mermaid Markdown side-car lands next to the HTML in the Exchange folder under `Visualizations/` for diff and source-control use.

The action is listed in the `tabActions` array returned by `get_state()` only when the UserTypes tab is active. See Generate a visual report of your UNS for the broader visual-report reference.

Creating UserTypes

1. Navigate to **Unified Namespace UserTypes**
2. Click **New** button
3. In dialog:
    - **Name**: Template identifier
    - **Description**: Documentation
4. Click **OK**
5. Add members in grid:
    - Type in first row
    - Press Enter to add
    - Configure properties per member

UserType Properties

Property	Description	Required
Name	Member name within template	Yes
Type	Data type (built-in or nested template)	Yes
Array	Array size for member	No
Parameters	Type-specific settings	No
Min/Max	Value limits	No
ScaleMin /Max	Engineering scale	No
Units	Engineering units	No
Format	Display formatting	No
Enumeration	Value mappings	No
StartValue	Initial value	No
Retentive	Value persistence	No
Domain	Server/Client scope	No
Visibility	External access	No
DisplayText	UI display text. Imports from `skos:prefLabel` or primary `rdfs:label` (10.1.5+).	No

Labels	Alternative names, semicolon-delimited. UserType class level only (not present on member-def). Imports from `skos:altLabel`. Searchable by `search_uns`. Design-time only — see "Mutability & retention" above (10.1.5+).	No
SourceIri	Full source IRI for round-trip with external ontologies. Populated by the ontology importer, user-editable at design time. Immutable at runtime (identity — see "Mutability & retention" above) (10.1.5+).	No
BaseUserType	Parent UserType for single-parent inheritance. Maps to `rdfs:subClassOf` on export (10.1.5+).	No
Attributes	JSON column for ontology annotations not covered by other columns (`dcterms:*`, alt-language labels, custom predicates). **Design-time only** at UserType and member-def level — edit in Designer and re-publish. For runtime-mutable per-tag annotations see the UNS Tags Reference (10.1.5 draft) (10.1.5+).	No
RelativeAddress	Device addressing	No
Description	Member documentation	No

Using UserTypes

Creating Tags from Templates

1. Go to **Unified Namespace Tags**
2. Click **New Item**
3. In **Type** dropdown, select template
4. Tag inherits all template members

Template Instance Example

```
Template: Motor
Members:
  - Running (Digital)
  - Speed (Double)
  - Temperature (Double)
  - AlarmStatus (Integer)

Tag: Motor1 (Type: Motor)
Access:
  @Tag.Motor1.Running
  @Tag.Motor1.Speed
  @Tag.Motor1.Temperature
  @Tag.Motor1.AlarmStatus
```

Nested Templates

Templates can contain other templates:

```
Template: PumpStation
Members:
  - Pump1 (Type: Motor)
  - Pump2 (Type: Motor)
  - FlowRate (Double)
  - Pressure (Double)

Access:
  @Tag.Station1.Pump1.Running
  @Tag.Station1.Pump2.Speed
```

Security Settings

Property	Controls	Applies To
EditSecurity	Modify permission	Design-time

ReadSecurity	Read access	Runtime
WriteSecurity	Write access	Runtime

Member Configuration

Supported Base Types

- Digital
- Integer
- Long
- Double
- Decimal
- Text
- DateTime
- Custom Templates

Array Members

```
Template: TankFarm
Members:
  - Tanks (Type: Tank, Array: 10)

Access:
  @Tag.TankFarm1.Tanks[0].Level
  @Tag.TankFarm1.Tanks[1].Temperature
```

Relative Addressing

For device communication:

```
Template: AnalogInput
Members:
  - Value (RelativeAddress: ".PV")
  - Status (RelativeAddress: ".ST")

Device mapping automatically appends to base address
```

Important Notes

When setting properties for array elements or template members (StartValue, Min, Max), these values will not display in the Designer DataGrid. Access them through Tag Properties dialog or at runtime.

Property changes to one instance do not propagate to others. Each template instance operates independently.

Common Use Cases

Motor Template

```
Members:
  - Running: Digital
  - Speed: Double (0-1800 RPM)
  - Current: Double (0-100 Amps)
  - Temperature: Double (0-200 C)
  - HoursRun: Long
  - LastMaintenance: DateTime
```

Tank Template

```
Members:
  - Level: Double (0-100 %)
  - Temperature: Double (-50-150 C)
  - Pressure: Double (0-10 Bar)
  - HighAlarm: Digital
  - LowAlarm: Digital
  - Product: Text
```

Production Line

```
Members:
  - Station1: Motor
  - Station2: Motor
  - ConveyorSpeed: Double
  - ProductCount: Integer
  - BatchID: Text
  - QualityScore: Double
```

Best Practices Checklist

- **Plan template hierarchy** - Design before implementation
- **Use meaningful names** - Clear member identification
- **Document templates** - Describe purpose and usage
- **Standard units** - Consistent engineering units
- **Reuse templates** - Avoid duplication
- **Version control** - Track template changes
- **Test thoroughly** - Verify all members

Troubleshooting

Template not appearing:

- Save after creation
- Check for naming conflicts
- Verify no circular references
- Review template hierarchy

Members not accessible:

- Confirm tag uses template type
- Check member names
- Verify array indices
- Review security settings

Values not retained:

- Check retentive settings
- Verify database connection
- Review member configuration
- Test with simple template