UNS Tags Reference

Define and manage tags in your UNS.

> ⚠ **Two distinct "reference" concepts in FrameworX — do not confuse**
>
> The word "reference" covers two different mechanisms in FrameworX. Both appear on this page, so disambiguating up front:
>
> **(a) The `Reference` tag type** — covered in the **Reference Type** section below. A runtime tag whose `.Link` property points to another tag of the Reference's target type, used for runtime pointer / alias behavior (one display driving many equipment instances). See the Tag Reference Open Display Example.
>
> **(b) A `Reference`-type UDT member** — covered on the sibling UNS UserTypes Reference page. A member declared with `Type=Reference` whose per-instance value points to another tag's `/Attr` envelope (for example, `Mixer_M101.feedsInto = Reactor_R101/Attr`). This is the FX representation of an OWL `owl:ObjectProperty` — an ontology relation between named individuals — and round-trips through the RDF/OWL importer and exporter. See Industrial Ontology Integration How-to.

UNS Tags (Reference) represent real-time variables and their associated historical records, providing a structured way to model process information by linking to physical devices, databases, or calculated values.

Tags in the Unified Namespace provide:

- Real-time data representation
- Historical record association
- Device communication mapping
- Hierarchical organization
- Type-safe data handling
- Array and template support

Tags form the foundation of the solution's data model, connecting field devices to visualization, logic, and storage.

Tag Types
Configuration Properties
Ontology metadata columns (10.1.5+)
AI / MCP page action (10.1.5+)
Creating Tags
Tag Parameters, Type Specific Settings
Per-instance enumeration override (10.1.5+)
Domain Configuration
Visibility Levels
Retentive Options
Format Strings
Array Configuration
DataTable Tags
Runtime Attributes
Quality on a comms or sync drop
Additional Configuration
Best Practices Checklist
Troubleshooting

Tag Types

Built-in Types

Type	.NET Type	Description	Range
Digital	Int32	Binary value	0 or 1
Integer	Int32	Whole numbers	±2,147,483,647
Long	Int64	Extended integers	$\pm 9.2 \times 10^{18}$
Double	Double	Floating point	$\pm 1.79 \times 10^{308}$
Decimal	Decimal	High precision	28-29 significant digits
Text	String	Unicode text	2GB max
Json	String	JSON structured data	Built-in parsing
DateTime	DateTimeOffset	Date and time	Year 0001-9999
TimeSpan	TimeSpan	Duration	Days to milliseconds
Guid	Guid	Unique identifier	128-bit
DataTable	DataTable	In-memory table	Structured data
Image	Byte[]	Binary content	Any size
Timer	Int32	Time patterns	Multiple modes

Digital Tag Values and Events

From FrameworX 10.1.6, a Digital tag raises an event on a Device read or a normal write only when its value (0 or 1) or its quality changes, like an Integer tag. Its timestamp updates only on such a change. Forced writes still raise an event every time.

A Digital tag stores 1 or 0, never the number as written: for example, 5 is stored as 1 and -3 as 0. From FrameworX 10.1.6, this also applies when the value comes from a DataTable copied with `TK.CopyDataTableToTag`, from a UserType member named Json filling a Digital member, from a locked tag with a LockValue other than 0 or 1, from the retentive store, or from a StartValue applied again by `TK.ClearTag`. In 10.1.5, the tag keeps the number as written in these cases.

> **Changed in FrameworX 10.1.6**
>
> In 10.1.5, a Digital tag fed a value other than 0 or 1, for example by a Device point reading a whole register, refreshes its timestamp on every read, and the server sends it to the other modules as a forced write. A Device channel writing the tag then queues one write per read. From 10.1.6, such a tag raises an event only when its 0/1 value or quality changes. Review scripts and other logic triggered by these tags. To keep the full register value, map the point to an Integer tag.

Timer Types

Mode	Behavior	Use Case
SquareWave	Toggle 0/1 at interval	Blinking, heartbeat
Pulse	Momentary 010	Triggers, events
DelayOff	Hold value then reset	TOF timer
Comparer	Daily schedule	Time-based control

Reference Type

Dynamic tag addressing:

```
// Set reference target
@Tag.Reference1.Link = @Tag.TankFarm/Tank1/Level.GetName();
```

```
// Use reference
double level = @Tag.Reference1.Value;
```

Dynamic Target Type

Each Reference has a target type, set in the **Parameters** column or in **ReferenceType** on the Tag Properties dialog. A Reference with the target type **Dynamic** accepts a tag of any type as its target. From FrameworX 10.1.6a, this includes a tag of a UserType (UDT).

Address a Dynamic reference by its name alone, for example `@Tag.Reference1`. Member access is not supported: FrameworX rejects an expression such as `@Tag.Reference1.Value` or `@Tag.Reference1.Speed`. The `Link` property is the exception. To read the members of a UDT tag through a reference, set the target type to the UserType of the tag.

The result of reading or writing a Dynamic reference depends on its target:

- **A tag of a basic type, or an element of a tag array:** the reference reads and writes the value of the target tag.
- **A UDT tag:** the reference has no value, the same as an empty reference. A display binding on the reference behaves as on an empty reference. `TK.GetObjectValue` on the reference returns no value, and `TK.SetObjectValue` on it changes nothing in the UDT tag. A script statement such as `@Tag.Reference1 = 5;` raises a run-time error, as on an empty reference.

The same rules apply to each element of an array of Dynamic references. To find which tag a Dynamic reference points to, read its `Link` property, for example `@Tag.Reference1.Link`.

Configuration Properties

Property	Description	Required

Name	Unique tag identifier	Yes
Type	Data type selection	Yes
Path	Asset tree location	No
Array	Array size (0 to N)	No
StartValue	Initial value at startup	No
Parameters	Type specific settings	No
Min/Max	Value limits	No
ScaleMin /Max	Engineering scale	No
Retentive	Value persistence	No
Domain	Server/Client scope	No
Visibility	External access level	No
Format	Display formatting	No
Units	Engineering units	No
Enumeration	Number to String (show result in Tag1.ValueAsString). Instance-level — each tag, or each member of a UserType instance, can resolve a different enumeration set (see **Per-instance enumeration override** below).	No
DefaultSymbol	Symbol used when dropping/pasting tags on a display	No
Labels	Semicolon-delimited alternative names. Imports from `skos:altLabel`, searchable via `search_uns` and `runtime_search_uns`. **Runtime-mutable and retentive** — scripts, bindings, and (planned) operator UI can edit at runtime; writes persist across restarts. Unbounded text (10.1.5+).	No
SourceIri	Full source IRI for the individual. External-system join key. Populated by the ontology importer, editable at design time only. **Immutable at runtime** — the IRI is the tag's identity; if a rename is needed, re-import with `owl:sameAs`. Unbounded (10.1.5+).	No
Attributes	JSON column for ontology annotations not covered by other columns (`dcterms:*`, alt-language labels keyed `rdfs:label@pt`, `skos:note`, `owl:versionInfo`, custom predicates). **Runtime-mutable and retentive** — edit via `@Tag.X.SetAttribute(...)` or the (planned) operator UI (10.1.5+).	No
Description	Documentation (1024 char)	No

Ontology metadata columns (10.1.5+)

Tags in 10.1.5 carry three ontology columns alongside their real-time data: `Labels`, `SourceIri`, and `Attributes`. These columns populate from RDF/OWL individuals imported by the ontology importer and round-trip back through **Export UNS RDF / JSON**.

- `Labels` — semicolon-delimited alternative names. Imports from `skos:altLabel` and `skos:hiddenLabel`. Searchable via `search_uns` (Designer) and `runtime_search_uns` (runtime) with one canonical scoring algorithm.
- `SourceIri` — full source IRI for the individual. External-system join key. Preserved on re-export, so round-trip is lossless.
- `Attributes` — JSON column for ontology annotations not covered by other columns (`dcterms:*`, alt-language labels keyed `rdfs:label@pt`, `skos:note`, `owl:versionInfo`, custom predicates).

Mutability & retention

Tag ontology columns follow a **class-vs-instance** split shared with UserType and member-def columns (see UNS UserTypes Reference):

Layer	SourceIri	Labels	Attributes
UserType	Design-time · immutable identity	Design-time	Design-time
Member-def	Design-time	Not modeled	Design-time
Tag (this page)	Design-time · immutable identity	Runtime-editable · retentive	Runtime-editable · retentive

Tag `Labels` and `Attributes` are the only ontology columns an operator, script, or UI can mutate at runtime. Writes land in the retentive database and survive restarts. Use `promote_retentive_attributes` before export to roll runtime overlays back into the cold-start data so the exporter produces a deterministic RDF/OWL file. Bindings can *read* all three columns but writes go through the property setter (script) or the planned operator UI — never through binding-write directly. `SourceIri` is the tag's ontology identity and never changes at runtime: if a rename is needed, re-import with `owl:sameAs`.

See Industrial Ontology Integration How-to for the full model.

AI / MCP page action (10.1.5+)

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

- `designer_action("export_graph", "rj")` — mirrors the **Export UNS** toolbar dropdown. Accepts four W3C RDF formats — `format=rj` (RDF/JSON, default; the format the ontology importer round-trips and the most compatible with graph stores like GraphDB / Stardog / Fuseki), `format=jsonld` (JSON-LD), `format=turtle` (alias `ttl`), and `format=ntriples` (alias `nt`). All four are native emitters in 10.1.5. The exported file lands in the solution's Exchange folder.

The action is listed in the `tabActions` array returned by `get_state()` only when the Tags tab is active. See Export your UNS to RDF/OWL /GraphDB for the broader export reference.

Creating Tags

Method 1: Asset Tree

1. Navigate to **Unified Namespace Asset Tree**
2. Right-click folder or click **New Tag** icon
3. Enter tag name and properties

Method 2: Tags Grid

1. Go to **Unified Namespace Tags**
2. Options:
    - Click **New Item** for dialog
    - Type directly in Name column
    - Paste from external source

Method 3: Import

- Copy from another solution
- Import from CSV files
- Use Solution Import Tools
- EngWrapper API

Tag Parameters, Type Specific Settings

Deadband

Limits value updates:

```
Absolute: ±5 units from last value
Percentage: 10% change required
```

EnumerationSet

Map values to text:

```
0  "Off"
1  "On"
2  "Auto"
```

Per-instance enumeration override (10.1.5+)

The **Enumeration** assigned to a tag is an **instance-level** property — it lives on the tag, not on the tag's type. A single UserType (UDT) can therefore be reused across many equipment instances and screens while each instance, or each individual member of an instance, resolves a *different* enumeration set. Setting the override on one instance never changes the type default or any sibling instance.

Plain tag

Set the **Enumeration** directly on the tag: show the Enumeration column in the Tags grid (right-click the column headers) and pick the set, or open the **Tag Properties** dialog and set it there. At runtime `@Tag.MyTag.ValueAsString` resolves text from that tag's own enumeration.

UserType (UDT) member

When a tag's type is a UserType and one of its members uses an enumeration, each instance can override that member's enumeration independently:

1. Navigate to **Unified Namespace Tags**.
2. Select the tag whose **Type** is the UserType.
3. In the **Tag Properties** panel at the bottom, use **Edit Properties of member** to select the member.
4. Set the **Enumeration** for that member.

The member's enumeration is an instance-level override and is set through the Tag Properties dialog — like `StartValue`, `Min`, and `Max` on a member, it is not editable in the Designer DataGrid and does not propagate to other instances (see UNS UserTypes Reference). At runtime `@Tag.Pump1.State.ValueAsString` reflects the override chosen for that instance.

> Scripts can read the `Enumeration` property of a tag but cannot assign it. Set the override in the Tag Properties dialog, as described above. See UNS Enumerations Reference Dynamic Enumeration Selection.

Domain Configuration

Domain	Scope	Use Case
Server	Global across all clients	Device data, shared values
Client	Local to each session	User settings, local states

Visibility Levels

Level	External Access	Use Case
Private	None	Internal only
Protected	Read-only	Monitor only
Public	Read/Write	Full access

Affects: OPC UA Server, MQTT Broker, TcpDataAccess

Retentive Options

Setting	Saves	Use Case
None	Nothing	Temporary values
ValueOnly	Value only	Operator setpoints
Properties	All including value	Complete state
PropertiesOnly	All except value	Configuration

Storage: Dataset.DB.Retentive database

Format Strings

Numeric Formats

```
N0      123         (no decimals)
N2      123.45      (2 decimals)
C       $123.00     (currency)
P       12.34%      (percentage)
X       7B          (hexadecimal)
E       1.23E+02    (scientific)
```

DateTime Formats

```
d      3/15/2024  (short date)
T      14:30:00   (long time)
yyyy-MM-dd HH:mm:ss  2024-03-15 14:30:00
```

Array Configuration

Array size N creates elements [0] to [N]:

```
Array = 5 creates:
Tag[0], Tag[1], Tag[2], Tag[3], Tag[4], Tag[5]
(6 elements total)
```

DataTable Tags

Using Queries

```
@Tag.DataTableTag = @Dataset.Query.MyQuery.SelectCommand();
```

Manual Population

```
@Tag.DataTableExample.StartBlockSet();
@Tag.DataTableExample[1].Column1 = value1;
@Tag.DataTableExample[2].Column2 = value2;
@Tag.DataTableExample.CommitBlockSet();
```

Runtime Attributes

Core Properties

Property	Type	Description
Value	Tag Type	Current value
Quality	Integer	OPC quality (0/64/192)
Timestamp	DateTimeOffset	Last change time

Quality Codes

- 0: Bad quality
- 64: Uncertain quality
- 192: Good quality

Quality on a comms or sync drop

When the connection that feeds a tag is lost — a device communication failure (read retries exhausted, node or channel failure) or a broken redundancy / remote-server sync link — the runtime does **not** leave the affected tags frozen at their last-good value pretending to still be live. Instead it **forces `Quality` to Bad (0)** and raises the change as a normal *Changed* event, so every consumer — displays, scripts, alarms, the Historian, OPC UA / MQTT publishers — sees the staleness on the next update rather than reading a stale value as if it were current.

Lifecycle

Event	Value	Quality	Raised as
Connection drops (read retries exhausted, or node / channel / sync failure)	Last-good value is retained (the runtime does not blank it)	**Bad (0)**	*Changed* event

Reconnect attempt also fails	Unchanged	Stays **Bad (0)**	—
Connection recovers (next successful read)	Fresh value from the field	**Good (192)** resumes	*Changed* event

The value being kept (rather than blanked) means a display still shows the last reading, but it is now flagged Bad — the quality, not the value, is what tells you it is no longer live.

> **Design consequence — react to .Quality, not just .Value**
>
> Because a dropped source holds its last `.Value` while only `.Quality` flips to Bad, control, interlock, and permissive logic must test `.Quality` — not the value alone. A rung that reads only `@Tag.X.Value` will keep acting on a stale number after the source is gone; gating it on `@Tag.X.Quality == 192` (Good) makes the logic fail safe when the data goes stale. The same applies to alarm conditions and any calculation that aggregates field data.

Additional Configuration

The TagNames are frequently reference in these other configuration Tables:

DevicesPoints

- **Node**: Communication node
- **Address**: PLC/device address
- **AccessType**: Read/Write/ReadWrite

AlarmsItems

- **Condition**: Trigger logic
- **Groups**: Alarm categorization
- **Limits**: Threshold values

HistorianTags

- **Table**: Storage location
- **Deadband**: Recording threshold
- **DeadbandType**: Absolute/Percentage

Best Practices Checklist

- **Use meaningful names** - Follow naming conventions
- **Set appropriate types** - Match data characteristics
- **Configure retentive** - Preserve critical values
- **Apply deadbands** - Reduce unnecessary updates
- **Document tags** - Use Description field
- **Organize in assets** - Logical hierarchy
- **Set security** - Control access levels

Troubleshooting

Tag not updating:

- Check device communication
- Verify Quality = 192
- Review deadband settings
- Confirm not disabled

Wrong value format:

- Check Format string
- Verify Units configuration
- Review scale settings

Array issues:

- Confirm index in range
- Check array size setting
- Verify element access

Retentive not working:

- Check database connection
- Verify retentive setting
- Review database permissions

In this section...
