Skill Discovery Services

Understand the three connection patterns for external data in the UNS: Local Tags + Device Points, Linked Tags (DataLink), and Dynamic Tags (Asset). Learn when to use each, how Discovery Services extend the namespace, and the DataLink shortcut for hybrid workflows.

FrameworX has two independent systems for getting external data into the Unified Namespace: the **Device Module** and **Discovery Services**. They share nothing architecturally. Some protocols (MQTT, OPC UA) are available in both, but the workflow and mental model are completely different. This skill teaches the three connection patterns and when to use each.

When to Use This Skill

Use when the user mentions or needs:

- Connecting to external data sources without creating individual device points
- TagProvider, Tag Discovery, Discovery Services, or dynamic tags
- DataLink property, linked tags, or auto-managed communication
- Asset() syntax for dynamic data binding
- Choosing between Device Module and Discovery Services
- Large-scale IoT/IIoT with dynamic or changing tag structures
- Understanding the difference between local tags, linked tags, and dynamic tags

Do NOT use when:

- The user wants traditional SCADA device communication (Channel Node Points) — that's the Device Module, use protocol-specific skills
- The user only needs simulated data with ValueSimulator
- The user is building displays without external connectivity

Key Concept: Two Independent Systems

```
DEVICE MODULE                          DISCOVERY SERVICES
(Pillar 2: Devices)                    (Technology under Pillar 1: UNS)
---------------------------            ---------------------------------
• Define local tags FIRST              • NO local tags needed
• Create Channel -> Node -> Points       • Give address, system discovers
• Explicit polling, scan rates         • On-demand, auto-managed
• ~100 protocols                       • ~17 protocols (discovery-capable)
• Full control, deterministic          • Dynamic, flexible
• Governance: LOCAL                    • Governance: EXTERNAL
```

Critical rule for AI: Never create UnsTags for data that will be accessed through Discovery Services dynamic tags. The whole point is zero local tags. Only create UnsTags when:

- Using the Device Module (traditional pattern)
- Using DataLink (hybrid pattern — you want local tag governance but auto-managed communication)
- You need alarms or historian on specific dynamic data points

The Three Connection Patterns

These are not stages of development — they are three equally valid architectural patterns chosen based on data characteristics.

Pattern 1: Local Tags + Device Points (Traditional SCADA)

Define tags locally in UNS, then map them to physical device addresses through the Device Module.

When to use:

- Regulatory compliance (FDA, EPA) requiring deterministic behavior
- Safety-critical control loops
- Legacy protocols without discovery (Modbus, BACnet, DNP3, serial)
- Explicit timing requirements (custom scan rates per address)
- Full audit trail requirements

Configuration workflow:

1. Create tags in UNS (UnsTags)
2. Create Device Channel with protocol
3. Create Device Node with station address
4. Create Device Points mapping tags to addresses

5. Configure AccessType for polling behavior

Access syntax: `@Tag.FolderPath/TagName` or `Tag.FolderPath/TagName.Member`

Modules involved: UNS (tags) + Devices (channel, node, points, access types)

Pattern 2: Linked Tags — DataLink Property (Hybrid)

Create local tags in UNS but set the **DataLink** property to a Discovery Services path. The runtime automatically manages communication — no Device Module configuration needed.

When to use:

- You want local tag governance (tag fields, alarms, historian, metadata) BUT
- You don't want to configure Device channels/nodes/points
- Multi-site deployments with standard tag models but changing sources
- Enterprise data integration where tag names must be stable
- Systems with many unused points (communication activates on-demand only)

Configuration workflow:

1. Create tags in UNS (UnsTags)
2. Set the `DataLink` property to the external path
3. A Discovery Services connection for that protocol must exist
4. Runtime auto-manages communication when the tag is accessed

DataLink field example:

```
Tag: Plant1/TankLevel
Type: Float
DataLink: ProductionMQTT.("plant/tank01/level")
```

ProductionMQTT is the Name of the Discovery Services connection. DataLink is written `ProviderName.("path")`, with the provider's own separator inside the quotes (MQTT uses /). `/Provider/path` and `Provider/path` do not link.

Access syntax: `@Tag.Plant1/TankLevel` — same as any local tag. The DataLink binding is transparent.

Key behavior:

- Communication activates only when the tag is accessed (on-demand)
- If nobody reads the tag, no polling occurs (resource efficient)
- All UNS features work: alarms, historian, min/max, retentive, engineering units
- The DataLink path must match a path reachable through an existing Discovery Services connection

Modules involved: UNS only (tags with DataLink). No Device Module tables.

Pattern 3: Dynamic Tags — Asset() Syntax (Zero Configuration)

Direct binding to external paths without creating any local tags. Discovery Services dynamically discovers and extends the UNS namespace. Tags are created in the real-time database (RTDB) on-demand when accessed.

When to use:

- Data structure is dynamic or unknown upfront — the external system defines what exists
- Monitoring/visualization applications (not tight SCADA control)
- IoT systems with thousands of changing endpoints
- Diagnostics, temporary connections, fleet/rental equipment
- Exploring new data sources before committing to a tag model
- Systems where the external system governs the namespace (not your solution)

Configuration workflow:

1. Create a Discovery Services connection (UnsTagProviders table) with protocol and address
2. Create an Asset Tree folder linked to the connection (a UnsAssetTree row with TagProviderLink), inside a folder of the tree: a link cannot sit at the root. Asset() paths run from the root of the Asset Tree through this folder
3. Access data using Asset() syntax — tags appear dynamically

Access syntax:

Asset() is the form for displays and expressions. Plant is a folder of the Asset Tree, and the linked folder is a folder under it that is linked to the TagProvider's root or to one of its branches. The linked folder sits inside a folder of the Asset Tree, not at its root. After the linked folder, write the rest of the path with the TagProvider's own separator (MQTT uses /, OPC UA uses \).

```
// In displays and expressions:
Asset("/Plant/<linked folder>/<path on the TagProvider>")

// In display element bindings (LinkedValue field), a path built from the selected tree node:
Asset(Client.Context.AssetPath + "/Temperature")
```

C# and VB.NET code reads the same path with `TK.Data.AssetValue`. In C#, double each backslash in the string; in VB.NET, write each backslash once.

```
object value = TK.Data.AssetValue("/Plant/<linked folder>/<path on the TagProvider>");
```

Key behavior:

- No local tags exist — data is accessed directly through the Discovery Services namespace
- The Asset() function takes a string that can be an expression (enabling dynamic UIs)
- Discovery Services auto-discovers the external tree structure at runtime
- Tags in the RTDB are created on-demand and cleaned up when no longer accessed
- Alarms and historian are NOT available on dynamic tags (no local tag to attach them to)

Modules involved: UNS (Discovery Services connection only). No tags, no devices.

Pattern Selection Guide

If your data...	Use	Why
Requires deterministic polling with explicit scan rates	Pattern 1 (Local Tags + Devices)	Only way to control timing
Uses legacy/proprietary protocols without discovery	Pattern 1	Only option for non-discovery protocols
Needs local governance + alarms + historian, but flexible sourcing	Pattern 2 (DataLink)	Best of both worlds
Has stable structure but multiple sites or changing sources	Pattern 2	Template once, link many
Is temporary or diagnostic	Pattern 3 (Dynamic/Asset)	No configuration overhead
Has unknown or changing structure	Pattern 3	Dynamic discovery at runtime
Has thousands of tags, mostly for monitoring	Pattern 3	Zero engineering per tag
Needs alarms on some points from a dynamic source	Mixed: Pattern 3 for browsing + Pattern 2 for alarm points	Use DataLink for the critical subset

Most production systems use multiple patterns simultaneously. A typical split:

- ~30% Local Tags (safety, control loops, legacy PLCs)
- ~60% Linked Tags / DataLink (enterprise metrics, standard equipment)
- ~10% Dynamic Tags / Asset() (IoT fleet, diagnostics, dynamic sources)

Discovery Services Connections (UnsTagProviders Table)

Supported Protocols for Tag Discovery

Only protocols with built-in discovery capability can be used. The following connectors ship today (call `list_protocols` for the live list):

Connector	Type	Notes
AB Rockwell ControlLogix	PLC	Discovers controller tags
Beckhoff TwinCAT	PLC	Discovers TwinCAT symbols
Canary Historian	Historian	Also usable as Storage Location
Codesys PLC Handler	PLC	Discovers Codesys variables
DataHub Communication Service	Application	Connects to Cogent DataHub
FlowTimebaseDB Database	Historian	Time-series database

GE Proficy Historian	Historian	Also usable as Storage Location
InfluxDB Database	Historian	Also usable as Storage Location
LineMonitor	Application	Production monitoring
MQTT Client	Messaging	Standard MQTT broker
MQTT Sparkplug B Collector	Messaging	SparkplugB-specific metadata
OPC UA Client	Industrial	Standard OPC UA discovery
AVEVA PI System	Historian	Also usable as Storage Location
Server to Server	Application	Connect to another FrameworX solution
SQL Database	Database	Generic SQL data source
SQL Database Normalized	Database	Normalized schema variant

Important: The Device Module has ~100 protocols. Discovery Services has ~17. They are independent lists. Do not confuse them.

Historian Connectors — Dual Role

Some connectors (Canary, InfluxDB, GE Proficy, AVEVA PI System) serve two purposes:

1. **Tag Discovery** — browse and read historical data, TrendCharts auto-query through Asset() syntax
2. **Storage Location** — configured in Historian Storage Locations as an archive target

On a historian connector, TrendCharts using Asset() syntax query the historian for time-range data. The connection row has no IsHistorian column; the protocol driver declares the capability.

Creating a Discovery Services Connection via MCP

```
get_table_schema('UnsTagProviders')
list_protocols('<protocol_name>')    // Get PrimaryStation format
```

Write example:

```
{
  "table_type": "UnsTagProviders",
  "data": [{
    "Name": "ProductionMQTT",
    "Protocol": "MQTT",
    "PrimaryStation": "localhost;1883;FrameworX-001;;;None;;AtLeastOnce;10",
    "Separators": "BranchSeparator=/;AttributeSeparator=/;",
    "Description": "Production floor MQTT broker"
  }]
}
```

Key fields:

- **Name** — identifies the connection. A linked folder of the Asset Tree points into it with `TagProviderLink` set to `Name.("path")`, and a tag's `DataLink` uses the same form. Asset() paths run from the root of the Asset Tree through the linked folder (`Asset("/Plant/MQTTData/...")`), never from this name.
- **Protocol** — must be one of the ~17 discovery-capable protocols
- **PrimaryStation** — protocol-specific connection string. Always call `list_protocols()` first to get the format.
- **Separators** — maps external hierarchy delimiters to UNS path delimiters. For MQTT: `BranchSeparator=/;AttributeSeparator=/;`

AssetTree Integration

The AssetTree makes Discovery Services data navigable in the UI. A **linked folder** connects an AssetTree node to a Discovery Services connection.

Creating Linked Folders via MCP

```
get_table_schema('UnsAssetTree')
```

```
{
  "table_type": "UnsAssetTree",
  "data": [
    { "Name": "Plant" },
    {
      "Name": "MQTTData",
      "Parent": "Plant",
      "TagProviderLink": "ProductionMQTT.(\"plant\")",
      "Description": "MQTT production data linked to the Asset Tree"
    }
  ]
}
```

The parent folder comes first: a link cannot sit at the root of the Asset Tree, and the import refuses a link row with no Parent. There is no InitialBranch column. `TagProviderLink` is `ProviderName.("path")`: the connection's Name and the branch the folder starts at, with the provider's own separator inside the quotes. `ProductionMQTT.("")` links the whole namespace of the connection. With the link above, `Asset("/Plant/MQTTData/tank01/level")` reads the topic `plant/tank01/level`.

From this folder down, the tree is dynamic — it auto-expands based on what the Discovery Services connection discovers at runtime.

When the user clicks a node in an AssetsTree control:

- `Client.Context.AssetPath` updates to the selected path
- `Client.Context.AssetName` updates to the selected node's full name; `Client.Context.AssetNodeName` to its leaf name
- All display elements bound to these properties react automatically

Asset() Syntax Reference

Basic Usage

Asset() is the form for displays and expressions. In C# or VB.NET code, read a value with `TK.Data.AssetValue`. `TK.Data.AssetDouble` and `TK.Data.AssetString` return a typed value.

```
// Read a value
double level = TK.Data.AssetDouble("/Plant/MQTTData/tank01/level");

// Dynamic path from context (in display CodeBehind)
double val = TK.ToDouble(TK.Data.AssetValue(@Client.Context.AssetPath + "/Temperature"));
```

Data Type Handling

`TK.Data.AssetValue` returns the value as an object. Convert it to the type the code needs:

```
int intValue    = TK.ToInt(TK.Data.AssetValue("/Plant/MQTTData/tank01/level"));
double dblValue = TK.ToDouble(TK.Data.AssetValue("/Plant/MQTTData/tank01/level"));
string strValue = TK.ToString(TK.Data.AssetValue("/Plant/MQTTData/tank01/level"));
bool boolValue  = TK.ToBool(TK.Data.AssetValue("/Plant/MQTTData/tank01/level"));
```

Asset() vs Tag Syntax

Syntax	When to use	Example
`@Tag.Folder/Name`	Local tags, Linked Tags (DataLink)	`@Tag.Plant1/TankLevel`
`Asset("path")`	Dynamic tags from Discovery Services	`Asset("/Plant/MQTTData/tank01/level")`
`Asset(expression)`	Dynamic UIs driven by user selection	`Asset(Client.Context.AssetPath + "/Temp")`

Note: `Asset()` also works with local tags. It's sometimes used even with local tags because the string argument can be an expression, enabling dynamic display patterns.

Common Pitfalls

Mistake	Why it happens	How to avoid
Creating UnsTags for dynamic Discovery Services data	Habit from Device Module workflow	Dynamic tags don't need local tags — that's the whole point. Only create UnsTags if you need alarms/historian or want DataLink.
Confusing Device Module protocols with Discovery Services protocols	Both can use MQTT/OPC UA	Device Module has ~100 protocols. Discovery Services has ~17. Check `list_protocols()` for Discovery Services availability.
Setting alarms on dynamic tags	Dynamic tags have no local tag to attach alarms to	Create a local tag with DataLink to the specific path, then configure alarms on that tag.
Using DataLink without a Discovery Services connection	DataLink needs the underlying TagProvider technology	A UnsTagProviders entry for the matching protocol must exist. DataLink binds to paths in that connection.
Thinking Discovery Services replaces Devices	They are independent systems	Both can coexist in the same solution. Use Devices for control, Discovery Services for monitoring/IoT.
Wrong PrimaryStation format	Each protocol has its own format	Always call `list_protocols('<protocol>')` before writing UnsTagProviders.

Mixed-Mode Example

A typical production solution combining all three patterns:

```
UNS Tags:
  Safety/EStop          -> Device Module (Modbus, deterministic polling)
  Safety/FireAlarm      -> Device Module (Modbus)
  Production/LineSpeed  -> DataLink: ProductionMQTT.("plant/line1/speed")  (local tag + auto-comm)
  Production/OEE        -> DataLink: ProductionMQTT.("plant/line1/oee")
  Quality/BatchResult   -> DataLink: MESServer.("mes\quality\result")  (OPC UA separates with \)

Discovery Services Connections:
  MQTT (ProductionMQTT) -> mqtt.factory.local:1883
  OPC  (MESServer)      -> opc.tcp://mes:4840

AssetTree:
  Plant/
    Safety/         -> manual tags, Device Module
    Production/     -> linked to ProductionMQTT via AssetTree folder
    Enterprise/     -> linked to MESServer, browsed via Asset() syntax
```

In this setup, safety tags have full deterministic control, production tags have local governance with auto-managed communication, and enterprise data flows dynamically with zero tag engineering.

See also

- Skill MQTT TagProvider. MQTT-specific Discovery Services implementation (broker connection, SparkplugB, built-in broker/simulator, UI construction).
- Skill MQTT Integration. Router for choosing between TagProvider and Device Module on MQTT brokers.
- TagProvider Services. Concept overview.
- UNS Tag Discovery Reference. Three connection patterns explained.
- UNS TagProvider Services Reference. All properties and settings.
- TagProvider Services Technology. Architecture deep dive.
- Unified Namespace. UNS foundation concepts.