Devices Points Reference

Define and manage device points.

Devices Points (Reference) represent specific data elements within field devices, mapping PLC registers, I/O values, and variables to platform tags for monitoring and control.

Device Points provide:

- Register-to-tag mapping
- Data type conversion
- Read/write access control
- Scaling and modifiers
- Dynamic addressing
- Import capabilities

Each point links a specific device address to a tag, enabling data exchange between field devices and the platform.

Creating Device Points
Point Properties
Address Configuration
Data Types
Modifiers
Scaling Configuration
Access Types
Dynamic Address Example
Importing Points
Performance Optimization
Best Practices Checklist
Troubleshooting
Diagnostics

Creating Device Points

1. Navigate to **Unified Namespace Tags**
2. Copy tags to be mapped
3. Go to **Devices Points**
4. Paste copied tags
5. Configure point properties

Point Properties

Property	Description	Required
TagName	Associated platform tag	Yes
Node	Parent device node	Yes
Address	Device register/location	Yes
DataType	Data format Native, Int32, Float	No
Modifiers	Byte order adjustments SwapBytes, SwapWords	No
AccessType	Read/Write permissions Read, Write, ReadWrite or other DevicesAccessTypes	Yes
Scaling	Value transformation Div:10, Add:32	No

Address Configuration

Static Addressing

Protocol-specific formats:

- **Modbus**: 40001, 30001, 10001
- **ControlLogix**: Controller.Tag[0]
- **Siemens**: DB100.DBW10
- **OPC**: ns=2;s=Channel.Device.Tag

Dynamic Addressing

Change addresses at runtime:

csharp

```csharp
// Retrieve point configuration
DataTable dt = TK.ProjectDB.GetDataTable("DevicesPoints",
    "TagName='Tag.Temperature'");

// Update address
DataRow row = dt.Rows[0];
row["Address"] = "Group01/" + @Tag.Gateway + "/Device01";
row.AcceptChanges();

// Apply changes
string error;
TK.ProjectDB.UpdateDataRows("DevicesPoints", row, out error);
TK.ProjectDB.ApplyProjectChanges();
```

Data Types

Native Type

Default - protocol handles conversion automatically

Override Types

Type	Description	Use Case
Boolean	Single bit	Digital I/O
Int16	16-bit signed	Analog values
Int32	32-bit signed	Counters
Float	32-bit floating	Measurements
Double	64-bit floating	Precision values
String	Text data	Messages

The **DataType** column can only be modified after the initial row validation. Since most protocols do not allow changing this value, it is disabled by default and only becomes editable after validation.

This issue does not occur when copying and pasting data from Excel. However, when editing manually, it is necessary to first add the point. Then, after the row has been inserted, you can use another custom field.

Modifiers

Byte Order Adjustments

Modifier	Description	When to Use
SwapBytes	Reverse byte order	Big/Little endian mismatch
SwapWords	Reverse word order	32-bit value issues
SwapDWords	Reverse double word	64-bit value issues
BitIndex	Extract specific bit	Packed boolean values

Scaling Configuration

Read Operations

```
Tag Value = (Device Value / Div) + Add
```

Write Operations

```
Device Value = (Tag Value - Add) * Div
```

Examples

Temperature scaling:

- Device: 0-1000 (tenths of degree)
- Div: 10, Add: 0
- Result: 0-100.0 degrees

Offset correction:

- Device: 0-100
- Div: 1, Add: -50
- Result: -50 to 50

Access Types

Configure read/write behavior:

- **ReadWrite** - Full bidirectional
- **ReadOnly** - Monitor only
- **WriteOnly** - Control only
- **OnDemand** - Manual trigger

See Devices AccessTypes for detailed configuration

Dynamic Address Example

MQTT Topic Routing

csharp

```csharp
public void UpdateMQTTAddress(string tagName, string gateway)
{
    string error;

    // Get point configuration
    DataTable dt = TK.ProjectDB.GetDataTable("DevicesPoints",
        $"TagName='{tagName}'");

    if (dt.Rows.Count > 0)
    {
        // Update address with gateway
        DataRow row = dt.Rows[0];
        row["Address"] = $"/topic/{gateway}/data";
        row.AcceptChanges();

        // Apply to project
        TK.ProjectDB.UpdateDataRows("DevicesPoints",
            new DataRow[] { row }, out error);

        if (string.IsNullOrEmpty(error))
        {
            TK.ProjectDB.ApplyProjectChanges();
            @Info.Trace($"Address updated: {row["Address"]}");
        }
        else
        {
            @Info.Trace($"Error: {error}");
        }
    }
}
```

Importing Points

Import Methods

- PLC program files (L5K, TIA)
- OPC browsing

- CSV/Excel files
- Online discovery

See Importing PLC Addresses for procedures

Performance Optimization

Efficient Configuration

- Group consecutive registers
- Use block reads when possible
- Minimize individual points
- Optimize polling rates

Address Optimization

```
Good: 40001-40010 (block read)
Poor: 40001, 40003, 40005 (individual reads)
```

Best Practices Checklist

- **Group Related Points** - Organize by function
- **Use Native Types** - Let protocol handle conversion
- **Document Addresses** - Include descriptions
- **Test Scaling** - Verify calculations
- **Plan Access Types** - Minimize unnecessary writes
- **Validate Imports** - Check addressing after import
- **Monitor Point Count** - Stay within license limits

Troubleshooting

Point not updating:

- Verify node connection
- Check address format
- Review access type
- Test with OPC client

Wrong values:

- Check data type match
- Verify byte order
- Review scaling settings
- Test without modifiers

Write failures:

- Confirm write access
- Check PLC program
- Verify address writable
- Review error logs

Import issues:

- Validate source file
- Check address format
- Review tag mapping
- Test with sample

Diagnostics

Monitor point communication:

csharp

```
// Get point status
bool quality = @Tag.Temperature.Quality == 192;

// Check last update
DateTime timestamp = @Tag.Temperature.TimeStamp;

// Monitor errors
string lastError = @Device.Point["Temperature"].LastError;

// Communication statistics
int readCount = @Device.Point["Temperature"].ReadCount;
int writeCount = @Device.Point["Temperature"].WriteCount;
```

In this section...
