Built-in Namespaces Reference

Use built-in namespaces to access runtime objects across all modules.

Reference **Code** Namespaces | Concept | Reference

Overview

Everything you create in a FrameworX solution — tags, alarm groups, device channels, displays — becomes a live .NET object at runtime. These objects are organized into **namespaces** that mirror the Designer module structure, and every object exposes properties and methods you can use in displays, scripts, and bindings without any programming setup.

How It Works
The Namespaces
Dot Notation
Config vs. Runtime: Two Views of the Same Object
Server vs. Client Domain
Using Namespaces in Code
Discovering Properties

How It Works

When you create an Alarm Group called **ProductionAlerts** in Designer, the platform automatically makes it available at runtime as:

```
@Alarm.Group.ProductionAlerts.TotalCount
```

No API calls, no configuration — the object exists because you created it. This applies to everything: tags, device nodes, historian tables, dataset connections, scripts, displays, and security users.

Because these are .NET objects, you also get the full .NET class library for free. A DateTime tag exposes `.Value.DayOfYear`, `.Value.AddHours(2)`, and every other `System.DateTime` method — not because FrameworX implemented them, but because your tag value *is* a .NET DateTime.

The Namespaces

Every FrameworX solution has the same namespace roots: `Tag` (the UNS) plus the built-in namespaces. Each contains the runtime objects created by its module's configuration. The table lists the module and system namespaces. `Layout`, `Enumeration`, and `TagProvider` are also built-in namespace roots.

Namespace	Module	What It Contains
Tag	Unified Namespace	All tags organized by asset path. Each tag exposes Value, Quality, Timestamp, Min, Max, Units, alarm conditions, and .NET type members.
Alarm	Alarms	Groups, Items, and Areas with counts, acknowledgment, and status. `Alarm.TotalCount`, `Alarm.Group.Warning.TotalCount`.
Device	Devices	Channels and Nodes with connection status, error counts, and communication diagnostics.
Historian	Historian	Tables, Tags, and storage locations with pending event counts and database status.
Dataset	Datasets	Database connections, queries, tables, and files with execution status.
Script	Scripts	Tasks, Classes, and Expressions with execution state and control properties.
Report	Reports	Forms and WebData objects for report generation.
Display	Displays	Display pages with navigation properties (MainPage, SelectPage, LogOn).
Security	Security	Users, Policies, and Permissions with authentication methods and role management.
Server	System	Server computer info: IP, hostname, DateTime, SystemMonitor (CPU, memory, disk), redundancy, and retentive storage.
Client	System	Per-client properties: current user, locale, session, UI state, trend charts. Values are local to each connected client.
Info	System	Solution metadata: version, license, module status, execution profiles, and diagnostics.

Dot Notation

All runtime access uses dot notation, drilling from namespace category instance property:

```
Alarm.Group.Warning.Sound           -- sound file for Warning group
Device.Node.PLC1.Status             -- connection status of PLC1
Historian.Table.ProcessData.PendingEvents  -- buffered records waiting to write
Server.DateTimeInfo.Now             -- server current time
Client.CurrentUser.UserName         -- logged-in user on this client
Tag.Plant1/Line2/Motor1.Speed.Value  -- tag value (slash for folders, dot for members)
```

Tags use **slash** (`/`) for asset path folders and **dot** (`.`) for member access. All other namespaces use dot notation throughout.

Config vs. Runtime: Two Views of the Same Object

Every object has two representations:

Config properties are the fields you set in Designer — the alarm threshold, the Modbus address, the display layout. These are what `get_objects` reads and `write_objects` modifies.

Runtime properties are the live state — the current alarm count, whether the device is connected, the tag's current value. These are what `browse_object_model` discovers and what displays and scripts bind to.

They are two views of the same object. Creating an alarm item in AlarmsItems (config) automatically creates `Alarm.Item.MyAlarm` (runtime) with properties like `.IsActive`, `.AckStatus`, `.TagName`, and `.Limit`.

Need	Tool	Example
Read or write configuration fields	`get_objects` / `write_objects`	Alarm thresholds, tag data types, device addresses
Discover runtime properties and methods	`browse_object_model`	`Alarm.Group.Warning` shows TotalCount, AckAll, Sound, etc.
See where an object is referenced	`designer_action ('find_object')`	Cross-references across displays, scripts, alarms

Server vs. Client Domain

Some namespaces have a **domain** — they exist on the server, the client, or both.

Server domain (`@Server`, and all module namespaces like `@Alarm`, `@Device`, `@Historian`): Values are shared across all connected computers. `Alarm.TotalCount` shows the same number everywhere.

Client domain (`@Client`): Values are local to each connected display client. `Client.CurrentUser.UserName` is different on each machine. Client properties are only available in displays and client-side scripts.

When running on a single computer, both domains exist — the server process handles data acquisition, alarms, and historian, while the client process runs displays.

Using Namespaces in Code

In **display bindings and expressions**, use the path directly:

```
Tag.Plant1/Line1/Motor1.Speed
Alarm.Group.Critical.TotalCount
Server.DateTimeInfo.Now
```

In **Script Tasks and CodeBehind**, prefix with `@` to distinguish runtime objects from local .NET variables:

```
@Tag.Plant1/Line1/Motor1.Speed.Value = 100;
int activeAlarms = @Alarm.TotalCount;
string serverIP = @Server.ComputerIP;
```

The `@` prefix is not needed in grid fields, dialog inputs, or display bindings — only in C# script code where it prevents name collisions with local variables.

Discovering Properties

You don't need to memorize property names. Three discovery mechanisms are available:

IntelliSense in Designer auto-completes as you type. Start typing `@Alarm.Group.` and it shows all groups; select one and it shows all properties of that group.

browse_object_model (MCP tool) provides the same discovery for AI sessions. Call `browse_object_model('Alarm')` to see child categories, then `browse_object_model('Alarm.Group.Warning')` to see all properties and methods of that group instance.

In this section...