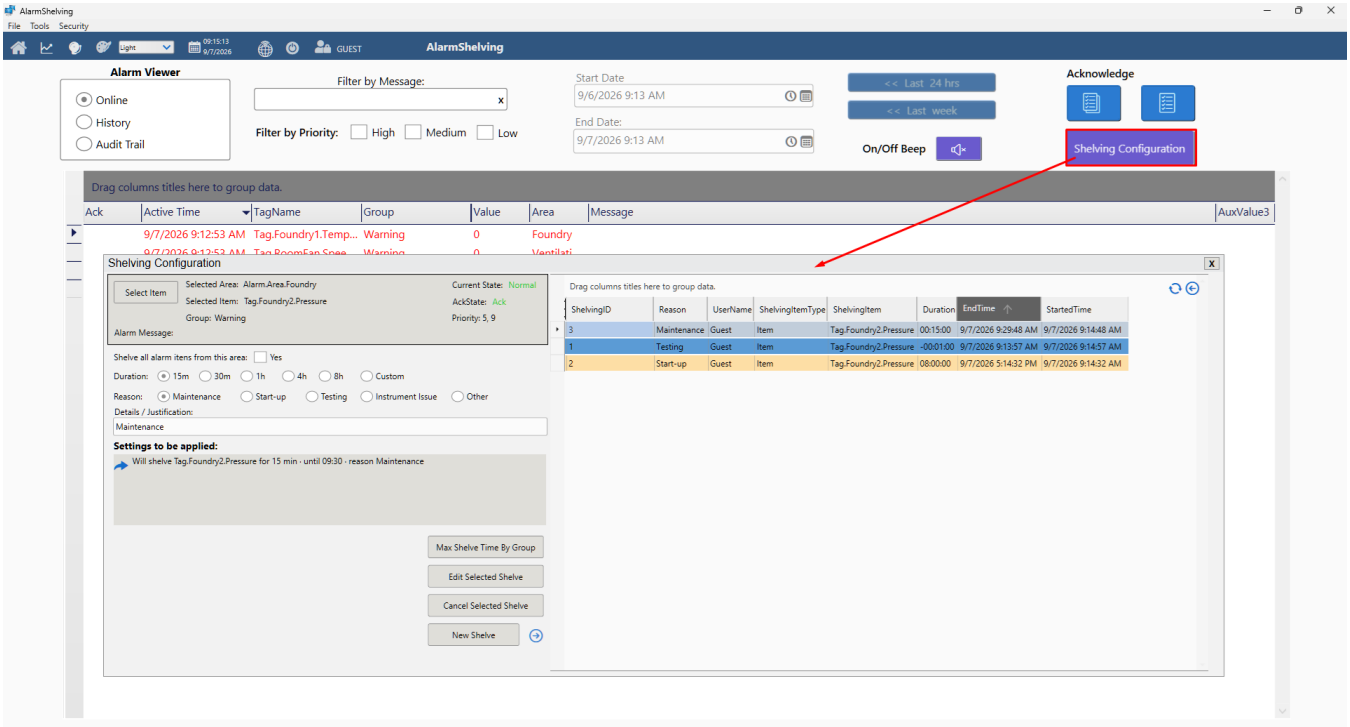
Alarm Shelving Example

Suspend alarms for a fixed period, with automatic return — the ISA-18.2 shelving mechanism.

Download the solution Alarm Shelving Example.dbsln



This example implements *shelving*: the temporary, reversible suspension of an alarm, initiated by an operator, with automatic return when a deadline expires. It exists to fight alarm fatigue during maintenance, commissioning or a known process condition — without permanently disabling monitoring, and without relying on someone remembering to switch it back on.

Scope and standard

The standard that defines the mechanism is **ANSI/ISA-18.2, *Management of Alarm Systems for the Process Industries***, which treats *Shelved* as a state in the alarm lifecycle, distinct from *Suppressed by design* and from *Out of service*. The international equivalent is **IEC 62682**.

The solution implements the mechanism as the standard describes it. That is not the same as certified compliance: the standard also covers rationalization, documentation and alarm system performance metrics, which are plant engineering activities rather than software. What the solution delivers are the technical controls the standard requires for shelving.

ISA-18.2 requirement for shelving	Where it is implemented
Fixed duration, with automatic return to normal	Rule 4 — expiry by `EndTime`, evaluated every cycle
Maximum duration limit, definable per alarm class	Rule 3 — `MaxShelveTime` per group
Alarms that must never be shelved	Rule 2 — `BlockShelve` category
Reason for the suspension recorded	`Reason` field, required on screen
Identity of who shelved the alarm	`UserName` field, taken from the logged-in user
List of currently shelved alarms, visible to the operator	Grid on the Shelving screen, over the `ShelvingInfo` table
Restriction on who may shelve	FrameworX Security, by permission on the buttons
Manual reversal before the deadline	Rule 5 — cancellation

> **Plant engineering decision.** Which alarms carry the `BlockShelve` category, what `MaxShelveTime` each group gets, and who is allowed to shelve are rationalization decisions — they do not ship configured with the solution. Without them, the mechanism works but does not meet the intent of the standard.

Architecture

The logic lives on the server; the screen only records intent. That separation is what makes the solution behave the same on the Windows client and in the browser, and it is the reason `IsShelved` is never computed in display code-behind.

Object	Type	Role
`ShelvingControl`	Script Class, Server domain	All decision logic: who is shelved, who is protected, what the group cap is
`InitShelving`	Script Task, tag trigger	One-time setup: creates the table, loads alarms and groups
`AlarmShelving`	Script Task, 5 s	Main cycle: reads the database and applies `IsShelved` to each alarm item
`SwitchProcess`	Script Task, 1 s	Persistence on redundant server switchover
`Shelving`	Display, popup	Main operator screen
`SelectItems`	Display, popup	Tag or area selection
`MaxShelveTimeByGroup`	Display, popup	Per-group cap configuration
`ShelvingInfo`	Dataset Table, SQLite	Persisted record of every shelving operation

The point of application

Everything converges on a single write. The runtime exposes each alarm item as an object, and that is where the suspension is recorded:

ShelvingControl.ApplyShelvingAsync

```
foreach (KeyValuePair<int, int> kv in finalStateByItemId)
{
    int itemId = kv.Key;
    int value = kv.Value;

    TK.SetObjectValue("@Alarm.Item.ID" + itemId + ".IsShelved", value);
}
```

This is the only place in the solution that writes `IsShelved`.

Data model

ShelvingInfo — the persisted record

A SQLite table, one row per shelving operation. Cancellation does not delete it: the row stays with `Duration = "-1"`, preserving the audit trail.

Column	Contents
`ShelvingID`	Row key
`ShelvingItem`	Name of the shelved tag or area
`ShelvingItemType`	`"Item"` or `"Area"`
`StartedTime`	Creation timestamp, server clock
`EndTime`	Timestamp of the automatic return
`Duration`	Duration as a `TimeSpan`; `"-1"` marks it cancelled
`Reason`	Operator justification
`UserName`	User logged in at the time of the operation

ShelvingUI and ShelvingData — the tags

`ShelvingUI` carries the operator's intent while the screen is being used; `ShelvingData` mirrors the database columns for the write. The separation keeps a field still being edited from contaminating the record.

AlarmsItems — the configuration

The screen reads the alarm configuration once, on open, and works against that in-memory copy. That is what lets the preview answer instantly, with no server round trip per keystroke:

Shelving display, DisplayOpening

```
alarmsItems = await TK.ProjectDB.GetDataTableAsync("AlarmsItems");
doNotShelveItems = await @Script.Class.ShelvingControl.GetDoNotShelveItems(alarmsItems);
```

The nine rules

These define the behavior. Each comes with the code that implements it.

Rule 1 — Shelving is per alarm item, never per tag

A tag can own several alarm items — one per condition (Hi, Lo, HiHi). The `IsShelved` attribute belongs to the *item*; the tag has no such attribute. Shelving a tag means shelving the set of its items, and each item is decided individually.

The practical consequence: a temperature tag can have its *Hi* alarm shelved and its *Lo* alarm active at the same time — and that is correct behavior, not a defect.

```
foreach (int itemId in itemIds)
{
    if (protectedIds.Contains(itemId))
    {
        finalStateByItemId[itemId] = 0;
        continue;
    }

    finalStateByItemId[itemId] = suspendValue;
}
```

Rule 2 — "Do Not Shelve" protection via the BlockShelve category

An alarm item in the `BlockShelve` category is never shelved, under any circumstance — not by direct selection, and not by inheritance from an area-wide suspension. The check runs every cycle, on the server, and overrides any record in the database.

```
private const string BlockShelveCategory = "BlockShelve";

private bool IsDoNotShelve(DataRow row)
{
    string names = SafeString(row, "CategoryNames");

    if (string.IsNullOrWhiteSpace(names))
        return false;

    foreach (string name in names.Split(';'))
    {
        if (string.Equals(name.Trim(), BlockShelveCategory, StringComparison.OrdinalIgnoreCase))
            return true;
    }

    return false;
}
```

Rule 3 — Duration cap per alarm group

Each group may carry a `MaxShelveTime` in minutes. Zero means no restriction. When a selection spans several groups — the area case — the **lowest** cap among them applies: the most severe restriction wins.

```
int maxTime;
// 0 means no restriction configured for this group — skip it
if (int.TryParse(row["MaxShelveTime"]?.ToString(), out maxTime) && maxTime > 0)
{
    if (maxTime < minMaxTime)
    {
        minMaxTime       = maxTime;
        restrictingGroup = g;
    }
}
```

The cap is used at two moments, for different purposes: `TryGetCap` only *reports*, feeding the preview; `ApplyCapIfNeeded` *enforces*, at the moment of the write. The operator sees the trim before deciding, so the write itself has nothing to ask.

```
private void ApplyCapIfNeeded()
{
    int cap; string group;
    int offTime = @Tag.Shelving/ShelvingUI.OffTime;

    if (TryGetCap(out cap, out group) && offTime > cap)
    {
        @Tag.Shelving/ShelvingUI.OffTime = cap;
        @Tag.Shelving/ShelvingUI.SelectedTime = cap;
    }
}
```

The configured value is also pushed to the group's runtime object, which is what constrains the alarm module itself:

```
string objPath = "@Alarm.Group." + groupName + ".ShelveMaxDuration";
TK.SetObjectValue(objPath, maxShelveTime);
```

Rule 4 — Automatic return on expiry

There is no "undo at the deadline". There is a full re-evaluation every cycle: records whose `EndTime` has passed are simply ignored, and the item falls back to normal. If the server was down at the moment of expiry, the alarm returns on its own as soon as the cycle runs again.

```
string endStr = SafeString(row, "EndTime");
DateTime? end = TryParseDateTime(endStr);
if (!end.HasValue || now >= end.Value)
    continue;
```

Every cycle starts from `IsShelved = 0` for all items. A suspension exists only while a valid record sustains it.

Rule 5 — Cancellation is a sentinel, not a deletion

Cancelling writes `Duration = "-1"` on the existing row. The record stays in the database with its original user, reason and timestamps — the audit trail is not lost.

```
private static bool IsShelvingCancelled(ShelvingRecord rec)
{
    string durationStr = (rec.Duration ?? "").Trim();
    return string.Equals(durationStr, "-1", StringComparison.OrdinalIgnoreCase);
}
```

Rule 6 — Per tag, the newest record wins

The same tag can appear in several active records — for instance an individual suspension plus one inherited from an area-wide suspension. The most recent `StartedTime` wins; ties are broken by the higher `ShelvingID`.

```
if (rec.StartedTime > current.StartedTime ||
    (rec.StartedTime == current.StartedTime && rec.ShelvingId > current.ShelvingId))
{
    newestShelvingPerTag[tagName] = rec;
}
```

Rule 7 — Areas are expanded on the server

An `"Area"` record is not applied to the area: it is expanded into that area's tag list at cycle time. Alarms created after the suspension fall into scope automatically, because the expansion is recomputed on every pass.

```
if (shelvingType.Equals("Area", StringComparison.OrdinalIgnoreCase))
{
    List<string> tagsInArea;
    if (!areaToTags.TryGetValue(shelvingItem, out tagsInArea) || tagsInArea.Count == 0)
        continue;

    tagsToApply = tagsInArea;
}
else
{
    tagsToApply = new string[] { shelvingItem };
}
```

Rule 8 — Preview before the click, confirmation after

The screen does not ask "are you sure?". It shows, before the click and while the operator can still change their mind, exactly what is about to happen: how many alarms out of how many, the return time, the group cap if one applies, and the list of protected items that will stay active.

```
string scope = isArea
    ? eligible + " of " + total + " alarms in " + area
    : (blocked > 0 ? eligible + " of " + total + " alarms on " + item : item);

string mainLine = "Will shelve " + scope + " for " + effective + " min"
                + capNote + " · until " + until.ToString("HH:mm") + reasonNote;
```

After the write, a status band confirms the result and clears itself after five seconds. The confirmation costs no extra click — one "OK" per operation is exactly the confirmation fatigue the screen is designed to avoid. The two bands never speak at once: on a successful write the preview is silenced, because it describes an intent that has already happened.

Rule 9 — Suspensions survive a server switchover

In a redundant architecture, the in-memory table on the active server is the good copy. When a switchover occurs, the `Control.Switch` flag goes to 1 and `SwitchProcess` writes that copy over the SQLite file, so the newly active server takes over with the correct state.

While that write is in progress, the main cycle stands down — the flag acts as a semaphore between the two tasks, preventing one from reading the database in the middle of the other's rewrite.

Execution cycle

Three server tasks, with different cadences and non-overlapping responsibilities.

Task	Cadence	What it does
`InitShelving`	Once	Fires when the `ShelvingData` tag loads. Creates the table if absent, reads the alarm configuration, builds the group list with its caps and the selectable item grid, and performs the first read of the database.
`AlarmShelving`	Every 5 s	The main cycle. Reads the records, recomputes `IsShelved` for every alarm item from scratch, and applies it. If the `Control.Switch` semaphore is raised, it returns without doing anything.
`SwitchProcess`	Every 1 s	Watches the semaphore. When raised, it recreates the table if needed and replaces the entire SQLite contents with the in-memory copy, recording the status code returned.

Maximum latency between an operator action and its effect on the alarm is therefore five seconds. This is deliberate: the cost of re-evaluating everything each cycle buys the property that system state is always a function of the database alone, never of an event history that might have been lost.

Operator workflow

Shelve an alarm

1. Open the **Shelving** screen from the alarm page.
2. Pick the target in **Select Item** — a specific tag or an entire area. Items protected by `BlockShelve` do not appear in the list.
3. Set the duration: one of the presets (15 min, 30 min, 1 h, 4 h, 8 h) or **Custom**.
4. Choose the reason and add the justification.
5. Check the **Settings to be applied** band, which already shows how many alarms will be shelved, until what time, and which will stay active.
6. Confirm. The status band acknowledges the write and disappears after five seconds.

Cancel before the deadline

Select the row in the grid and use **Cancel Selected Shelve**. On the next cycle the alarm returns to normal, and the record stays in history marked as cancelled.

Configure the per-group cap

The **Max Shelve Time By Group** screen lists the groups with their caps in minutes. The saved value then limits every new suspension in that group, and is pushed to the corresponding runtime object.

WPF and HTML5

The solution runs on both clients. That imposes an architectural rule which runs through the whole code-behind and explains several decisions that would otherwise look indirect.

> A **synchronous** call to a Server-domain object, made from a Display, **does not work on the HTML5 client**: it is refused and returns the default value, without throwing. The browser console records `InvokeSync is not supported here`.

Two consequences in the code. First: every call to a Server-domain Script Class uses `await`.

```
doNotShelveItems = await @Script.Class.ShelvingControl.GetDoNotShelveItems(alarmsItems);
```

Second: what the screen needs to know about alarms does not come from a runtime API called directly from the Display. `@Alarm.GetItemList` has no asynchronous variant, so the read was moved inside the Script Class — where the same call is local and legitimate — and the Display consumes the result with `await`.

The general principle, applicable to any Display on this platform: **configuration comes from a table loaded once on open; live state comes from a Server-domain Script Class called with `await`**. Configuration answered from memory keeps the screen instant; live state answered by the server keeps the screen correct.

Tag reference

Tag	Purpose
`ShelvingUI.SelectedItem`	Tag chosen by the operator
`ShelvingUI.SelectedArea`	Area chosen by the operator
`ShelvingUI.AllItensFromArea`	1 = area mode, 0 = item mode
`ShelvingUI.OffTime`	Effective duration in minutes — this is the authoritative one
`ShelvingUI.SelectedTime`	Preset chosen; `-2` indicates Custom
`ShelvingUI.Reason`	Justification entered
`ShelvingUI.AlarmState` / `.AckState`	Live state of the target, aggregated as "any item"
`ShelvingUI.Checks.PreviewText` / `.PreviewKind`	Preview band; Kind 0 neutral, 1 valid, 2 with a caveat
`ShelvingUI.Checks.ExclusionsText`	List of alarms that will stay active
`ShelvingUI.Checks.StatusText` / `.StatusKind`	Result band; Kind 0 hidden, 1 success, 2 failure
`ShelvingData.*`	Mirror of the database columns, for the write
`ShelvingData.Control.ShelvingTable`	In-memory copy of the records
`ShelvingData.Control.AlarmGroups`	Groups with their `MaxShelveTime`
`ShelvingData.Control.Switch`	Server switchover semaphore

`@Alarm.Item.ID<n>.IsShelved`	Shelved state of each alarm item, in the runtime
`@Alarm.Group.<name>.ShelveMaxDuration`	Group duration cap, in the runtime

In this section...