Myro:Home Helper

NESS / ELK Controller Guide

Version 5.50

ELK automation, saved configuration, solar schedules, supported features, and controller-scoped AI references.

Myro Control, LLC | myrocontrol.com

Contents

Start with your controller

Connect your ELK controller

Configuration and live status

Sunrise and sunset

Supported panel controls

Features unavailable on ELK

Events and shared integrations

Give an AI the ELK reference

References and maintenance

- Editor namespaces

JavaScript API Reference

- MyroAPI - Device Control
- MyroController - Controller & Object Data
- MyroAudio - Audio Zone Control
- MyroSensors - Read Device States

Event-Based Scripts

- Available Event Types

Scheduled Scripts

- ELK solar schedule setup

HTTP and MCP controller operations

- Object numbering and reads
- Typed control
- Security control
- Weather and Sense integration

Start with your controller

This guide describes the NESS / ELK M1 adapter in Myro:Home Helper 5.50 (preview). It describes what Helper exposes, rather than every capability of an ELK panel. Optional lighting, thermostat and audio hardware must be installed and configured before those controls can work.

Open the controller's Scripting window. The **API Reference** button shows feature badges:

Badge	Meaning
HAI / Leviton	Supported by the HAI adapter; a row with only this badge is unavailable on ELK
ELK	Supported by the ELK adapter
ELK: limited	ELK supports a narrower parameter set or different behavior; read the row and this guide
Savant	Savant room/media APIs; unavailable through ELK panel APIs
Control4	Control4 room/activity APIs (`MyroControl4`); unavailable through ELK panel APIs
Helper	A shared Helper service or integration, usable with HAI / Leviton, NESS / ELK, Savant, or Control4 when configured

Badges explain availability; they do not bypass runtime checks or promise that optional hardware exists. Unsupported ELK controls return `false` with a scripting error before sending a command or updating cached values. Unsupported event registrations return `false` and register no handler. Unavailable object fields return `null`.

For Helper APIs, script management, scheduling syntax and integrations, use the shared Helper Scripting Guide. The controller API contracts appear below in this guide. For scripts targeting ELK, copy the scoped AI reference described below rather than asking an assistant to infer compatibility from a HAI example.

For a Savant system on the same Helper, use the Savant Controller Guide. Savant has separate room/service IDs and MyroSavant controls. Cross-controller scripting requires local Send To and Accept From grants; panel methods in this guide do not control Savant rooms. A Control4 destination, reached with `MyroControllers.get(guid).control4`, works the same way under the Control4 Controller Guide; it has rooms and activities, with no panel objects, and these panel methods do not control it.

Connect your ELK controller

1. Click **+** under the controller list and choose **NESS / ELK**. Helper adds the controller and opens its **Connection** tab.
2. Enter a system name, choose the **Panel** (M1 / M1 Gold, EZ8 or EZ24), and enter the interface's host/IP address. The port is 2601 for a secure connection, or 2101 without one.
3. Leave **Secure connection** on and enter the interface username and password. Without it, codes and panel traffic cross your network unencrypted. Turning it off or on switches the port between 2101 and 2601.
4. Check the user code length (it updates from the panel as Helper connects), and optionally enter an **arming code for HomeKit and automations**. HomeKit, scripts and automations can't type a code, so Helper uses this one to arm and disarm. It is stored only in this Mac's Keychain; without it, those requests are refused. Once saved, the field reads "saved; enter to replace". Codes typed in web control or sent through the HTTP API are checked by the panel itself. ELK controllers appear in web control's controller list; sign in with an API key or a Control Login.
5. Select **Connect**. Helper saves the settings and connects. The first time, and if the interface's certificate ever changes, Helper shows the certificate and asks whether to trust it.
6. **Pair your phone (optional).** Open **Connection > Pairing**. Helper makes a pairing code for this controller automatically and shows it as a QR code and in groups of four characters. In Myro:Home Pro, open the ELK controller's settings, choose **Pair with Helper**, and scan the QR code or type the code. The app can then find this Helper on your network and receive the controller's home address. **Copy** puts the code on the clipboard; **Regenerate...** makes a new one, after which phones must pair again. The code identifies the controller to your apps; it is not your panel's login.

Change these settings later on the same Connection tab: select the controller, edit the fields, and they are saved as you leave each one (and again when you select Connect). The address, port, login and code length can be changed only while the controller is disconnected. Removing the controller also removes its interface login and arming code from the Keychain.

Helper reads the panel's user code length and temperature units from the panel as it connects; there is nothing to set for them. Initial setup and **Sync Controller Data / Sync Now** download names/configuration; subsequent app opens and reconnects keep the persisted inventory and refresh status/values. Lights, outputs, tasks, zones, areas, thermostats, temperature sensors, counters and M1XEP audio zones and sources appear in Edit Objects through the existing Object Editor. Outputs appear as Output objects numbered 256 + the ELK output number, and counters as Flag objects numbered 1000 + the counter number.

To expose supported ELK objects to Apple Home and Siri, open **HAP > Edit Objects**, enable HAP on the desired areas/security, zones, lights, outputs, tasks and thermostats, and select the appropriate accessory types. Start the HAP bridge and pair it in Apple Home with the displayed setup code. Restart HAP after changing the exposed objects. Optional equipment and the panel's capabilities determine which objects are available. ELK HomeKit arming/disarming uses the locally saved code described above; it does not add HAI-only locks, UPB links or blinds. See Getting Started for the shared pairing workflow.

ELK support is a preview. The Status tab's HAI-only details, such as numerical battery readings, may stay empty for an ELK panel. These are not available on ELK: locks, UPB links, blinds, user settings, area lights and HAI's "off for a time". Ethernet Quality (`MyroSystem.getEthernetQuality`, `ScriptEvents.onEthernetQuality`, the web control view and the graphs route) is not available for ELK in the preview, because its signals are Omni-Link specific.

Configuration and live status

Helper persists discovered names and configuration in Core Data. Opening the app or reconnecting keeps that inventory and refreshes lightweight status/values. Initial setup, **Sync Now / Sync Controller Data**, and completion of installer programming can perform a full configuration walk. A reconnect is not a reason to download all names again.

ELK uses a persistent M1 ASCII socket. Live panel notifications update known objects; startup/reconnect status requests and periodic reconciliation refresh the cache. This combines socket push with status requests, rather than repeatedly walking names. Notifications depend on the panel/interface configuration and the message family; do not assume every field is pushed.

Before acting on a sensor, check `MyroSensors.isDataFresh(name)` and handle unavailable sentinels: device state `-1`, temperature `-999`, freshness `false`. A saved name is not proof of a current value.

`MyroController.refreshStatus()` requests/coalesces a known-object status refresh without downloading names. It returns a request ID or `null`. Register `ScriptEvents.onStatusRefresh(callback)` first, match `result.requestID`, and check `result.success`. `getRefreshStatus()` exposes the latest result without requesting more traffic. A successful refresh confirms cached observations, not completion of a device command.

Sunrise and sunset

ELK schedules use Helper's local calculation from a location saved for that controller.

1. In Helper's main window, select the ELK controller and open **Connection > Location**.
2. Type a city or ZIP code and click **Look Up**. Helper fills in the latitude, longitude and time zone (you can also type them).
3. Click **Save**. Helper persists the location in Core Data for this controller and shows today's sunrise and sunset; the Status tab's Sunrise / Sunset row uses it too.
4. In a script's Schedule settings, choose Sunrise or Sunset, or use a solar token in a cron expression.

The same location can also be entered in a script's Schedule settings (latitude, longitude, IANA time zone, **Save Location**).

There is no JavaScript, HTTP or MCP setter for the solar location. An AI assistant should guide the user to Connection > Location rather than invent a location API.

Examples of schedules:

Schedule	Meaning
`0 @sunrise * * *`	At sunrise each day
`0 @sunset-30 * * 1-5`	Thirty minutes before sunset on weekdays
`0 @sunrise+15 * * *`	Fifteen minutes after sunrise each day

The solar token supplies both hour and minute. Offsets range from -720 to +720 minutes. Helper recalculates daily using the saved time zone and DST; day restrictions use the actual offset run date. Helper must be running and the Mac awake. Wake handling does not replay missed executions.

Read `MyroController.getStatus().solarConfigured`, `solarSource` and `solarTimeZone` to inspect setup. ELK's source is `"calculated"`. Without a valid saved location, or on a polar day with no sunrise/sunset event, the corresponding solar time is `null` and that solar schedule does not run. Do not substitute a guessed time.

Supported panel controls

Feature	ELK behavior
Lights/units	On/off, level, cached relative dim/brighten and toggle on supported configured units
Counters/flags	Set, clear, assign value, increment/decrement; cached values are bounded by Helper's counter representation
Outputs	On/off and timed on; timed off is unavailable
Areas/security	Supported modes and disarm; requires the programmed ELK arming code
Tasks/buttons	`runMacro(name)` invokes a mapped ELK task
Thermostats	Mode, heat/cool setpoints, relative adjustments, hold, fan auto/on
Keypad text	`showMessage(text, beep)` sends literal text for 60 seconds in two 16-character lines; `clearMessage(...)` clears it
Audio	Power, volume, source and mapped transport keys when panel audio hardware is configured

Generic `turnOn`, `turnOff`, `setLevel` and `toggle` address units, counters and outputs. Use typed controls for other objects. Timed on supports lights/outputs, not counters. `turnOffTimer` is unavailable; `setOutputTimer(name, true, seconds)` is supported, but `on=false` is rejected. Timers use the documented 0-99-second range.

Helper maps ELK light numbers directly. Outputs use Helper unit numbers `256 + panel output number`; counters use flag numbers `1000 + panel counter number`. Prefer exact saved device names and the Object Editor over manually deriving numbers.

Audio `sendKey(zone, code)` supports only 1 (power), 2 (source step), 3/4 (volume up/down), 5 (mute toggle), 6 (play), 7 (stop), 8 (pause), 11 (previous) and 12 (next). Mute state is unknown; key 5 toggles it without offering an absolute mute setter. `getSources()` returns cached configured source names with `metadata: null`.

Features unavailable on ELK

These are HAI features without a supported Helper ELK equivalent:

Feature	Result on ELK
HAI energy-cost level	`setEnergyCost` returns false; `energyCost` is null
Stored messages	No stored-message inventory/events; literal keypad text remains supported
Blinds, UPB links, locks and access readers	Controls rejected; objects omitted
User settings, area/all-lights commands, touchscreen cameras	Controls rejected
Thermostat humidity/setpoints, fan cycle	Controls rejected; unsupported readings null
Absolute audio mute and source metadata	Controls/refresh rejected; readings null
Numerical battery reading/percent, freeze, features, modelCode	Fields null; supported panel trouble indicators remain available

Phone-line state is `"ok"` or `"trouble"`, rather than HAI ringing/off-hook states. Thermostat `humidity`, humidity setpoints, `actionMode` and `freezeAlarm` are null. Audio snapshots and audio-zone events have a null mute state.

Read `MyroController.getStatus().backend` (`"elk"`) and `.capabilities` when choosing optional behavior. `solarSchedules` is true, subject to saved-location setup. The following capabilities are false: `energyCost`, `batteryReading`, `freezeStatus`, `upbLinks`, `locks`, `accessControl`, `storedMessages`, `userSettings`, `audioSourceMetadata`, `systemStatusEvents`, `humidity`, `thermostatFanCycle`, `audioMuteState`, `absoluteAudioMute`, `areaLights`, `cameraCommands`.

HAI cost levels and Sense energy readings are different features. Configured Sense readings and `onEnergySync` remain available. Never emulate `setEnergyCost` by creating or substituting an `EnergyCost` counter. A user can explicitly automate their own named counter using the ordinary counter API.

Events and shared integrations

Use zone, unit, flag/counter, output, auxiliary-sensor, area, thermostat and supported audio-zone events for native ELK objects. Connection, Ethernet-quality and status-refresh events remain available. Start an event script with `ScriptEvents.clear()` and use exact saved names or a documented wildcard.

These nine registrations are unavailable and return false without a handler:

- `onBlindChange`, `onUserSettingChange`, `onAudioSourceMetadataChange`
- `onLockChange`, `onAccessControlEvent`, `onMessageChange`
- `onUPBLink`, `onOmniEvent`, `onSystemStatus`

Read panel troubles through `getStatus()` rather than registering the HAI system-status event. Configured weather/Sense events, Sonos, Lutron, MQTT, network requests, notifications, speech and display integrations retain their shared Helper contracts. Their availability depends on integration setup, credentials and permissions.

Give an AI the ELK reference

With the ELK controller's Scripting window open, use **Help > Copy AI Scripting Reference**, **Copy AI Reference (Compact Text)** or **Save AI Scripting Reference**. Helper uses the scoped controller from the scripting window. The copied/saved file is named `MyroHome_Helper_ELK_AI_Scripting_Reference.md` for an ELK-only scope.

The reference composes the shared Helper guide with this guide's explicit ELK scripting section, includes supported controller APIs and shared integration contracts, excludes callable HAI-only APIs and examples, and appends the user's actual cached device names and configured integration names. The compact form retains signatures and key boundaries while dropping most worked examples.

The HTTP `/helper/{controllerID}/scripts/reference` endpoint and MCP scripting-reference tool use the same builder and the resolved controller ID. An unscoped home containing both HAI and ELK receives both independently maintained controller sections with explicit family labels. An unscoped home containing only ELK receives the ELK reference.

Cross-controller permission handles apply the destination controller's backend and grants. For a HAI destination reached from an ELK script, request that destination's reference before using its HAI-only features. A device-name list or permission grant does not make an unsupported feature available on the originating panel.

References and maintenance

- **Helper Scripting Guide**: shared Helper APIs, execution limits, integration events and scheduling syntax.
- **HTTP API Reference / MCP Guide**: external control and script-management interfaces; these are separate from JavaScript globals.
- **API Reference** in the Scripting window: feature badges and quick method descriptions.
- **NESS / ELK Controller Guide** in the Help menu: this controller-specific guide.

For developers, update the shared guide for Helper API changes and this guide for ELK controller contracts. Feature badges and explicit AI composition share `ScriptReferenceProfile`; runtime gates live in the bridges and `ELKScriptPolicy`. Run `Tools/run-script-reference-profile-regressions.cjs` and the normal scripting/documentation checks when changing either contract. Regenerate this PDF with `Tools/build-helper-elk-scripting-guide.sh` or all bundled guides with `Tools/build-helper-docs.sh`.

Editor namespaces

Use `Device.`, `Zone.`, `Unit.`, `Button.` (tasks), `Area.`, `Thermostat.`, `AuxSensor.`, `Flag.` (counters), `Output.`, `AudioZone.` and `AudioSource.` for configured objects. Namespaces insert saved names as quoted strings; they are not runtime objects. Stored messages, blinds, user settings, locks/readers and UPB links are unavailable object families.

JavaScript API Reference

MyroAPI - Device Control

- Control ELK lights/units, counters, outputs, areas, thermostats and tasks through the documented bridges. An active connection is required. Generic on/off/level/toggle only address units, counters and outputs; use typed methods for other objects.

Basic Control

`turnOn(deviceName)`

Turn a device on.

```
MyroAPI.turnOn("Kitchen Light");
MyroAPI.turnOn("AC Unit");
MyroAPI.turnOn("Porch Light");
```

Parameters:

- `deviceName` (string): Device name (case-sensitive, must match controller exactly)

Returns: `true` if successful, `false` if device not found

`turnOff(deviceName)`

Turn a device off.

```
MyroAPI.turnOff("Kitchen Light");
MyroAPI.turnOff("AC Unit");
```

Parameters:

- `deviceName` (string): Device name

Returns: `true` if successful, `false` if device not found

`setLevel(deviceName, level)`

Set dimmer or device level (0-100).

```
MyroAPI.setLevel("Bedroom Dimmer", 50);  // 50% brightness
MyroAPI.setLevel("Bedroom Dimmer", 0);   // Off
MyroAPI.setLevel("Bedroom Dimmer", 100); // Full brightness
```

Parameters:

- `deviceName` (string): Device name
- `level` (number): 0-100 (automatically clamped to valid range)

Returns: `true` if successful, `false` if device not found

`dim(unitName, steps)` / `brighten(unitName, steps)` (4.50)

- These relative helpers require a configured ELK light. The current adapter calculates a new absolute light level from its cached reading, changes it by 10 percentage points per step and clamps it to 0-100. This is a cached read/modify/write, not an atomic native panel-relative command. A stale reading or concurrent panel change can affect the result.

```
MyroAPI.dim("Bedroom Dimmer", 1);
MyroAPI.brighten("Bedroom Dimmer", 2);
```

Parameters:

- `unitName`: A configured ELK light's controller or custom name; not a Flag, output, blind or directly connected Lutron bridge device.
- `steps`: Required whole number from 1 to 9 steps of 10 percentage points each. Fractional/out-of-range values fail without sending. No timer argument.
- Support and resulting level depend on the panel and lighting technology. Use `setLevel` for an absolute target.

Returns: `true` means submitted to the connected controller's command queue, not a panel ACK or observed final brightness. `false` means invalid input, no matching Unit, disconnection or denied cross-controller access.

- Observe later unit status for the reported result. Never automatically retry an uncertain relative command: the adjustment could apply twice.

`toggle(deviceName)`

Toggle device between on and off.

```
MyroAPI.toggle("Porch Light");  // On → Off or Off → On
MyroAPI.toggle("Hallway Light");
```

Parameters:

- `deviceName` (string): Device name

Returns: `true` if successful, `false` if device not found

Timer-Based Control

`turnOnTimer(deviceName, seconds)`

- deviceName: exact light/unit or output name. seconds: integer 0-99. ELK supports timed on for lights/outputs; timed counters and timed off are rejected. **Returns:** Bool: sent or rejected, not action completion.

Flag Control

Flags are system variables that can be on/off or have counter values (0-255). Use flags for state tracking, counters, and conditional logic.

`setFlag(flagName)`

Set a flag (turn on).

```
MyroAPI.setFlag("System Armed");
MyroAPI.setFlag("Guest Mode");
MyroAPI.setFlag("Vacation Mode");
```

Parameters:

- `flagName` (string): Flag name

Returns: `true` if successful, `false` if flag not found

Use cases:

- System state tracking (armed/disarmed, home/away)
- Mode flags (guest mode, vacation mode, party mode)
- Conditional automation triggers

`clearFlag(flagName)`

Clear a flag (turn off).

```
MyroAPI.clearFlag("System Armed");
MyroAPI.clearFlag("Guest Mode");
```

Parameters:

- `flagName` (string): Flag name

Returns: `true` if successful, `false` if flag not found

Messages

`showMessage(messageName, beep)`

- ELK interprets messageName as literal keypad text, not a stored-message name or number. The text is shown on keypads for 60 seconds, in two 16-character lines (32 characters total). beep requests the keypad beep. **Returns:** Bool: sent or rejected.

`clearMessage(messageName)`

- ELK clears keypad text. messageName is retained for API signature compatibility, not stored-message lookup. **Returns:** Bool: sent or rejected.

`setFlagCounter(flagName, value)`

Set flag counter value (0-255).

```
MyroAPI.setFlagCounter("Visit Counter", 0);    // Reset counter
MyroAPI.setFlagCounter("Visit Counter", 42);   // Set to 42
MyroAPI.setFlagCounter("Event Count", 255);    // Max value

// ELK relative helper: derives an absolute write from the cached counter
MyroAPI.incrementFlag("Event Count");
```

Parameters:

- `flagName` (string): Flag name
- `value` (number): Counter value (0-255, automatically clamped)

Returns: `true` if successful, `false` if flag not found

Use cases:

- Event counters (door opened count, motion detections)
- Threshold tracking (run action after N events)
- Persistent state storage

`incrementFlag(flagName)` / `decrementFlag(flagName)` (4.50)

Request a one-step change for a configured native ELK counter mapped as a Flag. There is no amount or timer argument.

```
MyroAPI.incrementFlag("Visit Counter");
MyroAPI.decrementFlag("Visit Counter");
```

- The current ELK adapter calculates the new value from its cached native counter reading and sends an absolute set. This is a cached read/modify/write, not an atomic panel increment. It defaults a missing

cached value to zero, then clamps the result to 0-65535. Helper's ordinary Flag reads and explicit assignments remain limited to 0-255.
- A stale reading or concurrent panel change can overwrite a newer count; repeated calls before status arrives do not promise consecutive increments. Do not use these helpers when atomic counter behavior is required.

Parameters:

- `flagName`: A configured ELK counter's controller or custom name. No amount or timer argument.
- Confirm fresh reported data first, and observe subsequent reports. Panel programs can respond to the write.

Returns: `true` means submitted to the connected controller's command queue, not confirmation of a new count. `false` means no matching Flag, disconnection or denied cross-controller access.

- Observe `ScriptEvents.onFlagChange` for reported values; do not assume your cached count plus one or automatically retry an uncertain command.
- Also available on a permitted `MyroControllers.get(guid).api` handle, with destination access checked on every call.

Extended Controls

- Every supported control returns true when the command was sent, or false when unavailable. A sent command is not a hardware acknowledgement.

`setOutput(outputName, on)`

Switch a controller output on or off.

Parameters:

- `outputName` (string): Output name
- `on` (boolean): `true` to energize, `false` to release

Returns: `true` if the object exists and the command was sent, `false` otherwise

`setOutputTimer(outputName, on, seconds)`

- outputName: exact output name. on: true only for ELK; false is rejected. seconds: integer 0-99. **Returns:** Bool: sent or rejected, not action completion.

`setThermostatMode(thermostatName, mode)`

Set a thermostat's system mode.

Parameters:

- `thermostatName` (string): Thermostat name
- `mode` (string): `"off"`, `"heat"`, `"cool"`, `"auto"`, or `"emergency"`

Returns: `true` if the object exists and the command was sent, `false` otherwise

`setThermostatFan(thermostatName, mode)`

- thermostatName: exact thermostat name. mode: "auto" or "on". ELK rejects "cycle". **Returns:** Bool: sent or rejected.

`setThermostatHold(thermostatName, hold)`

Hold the current setpoints against the thermostat's schedule, or release the hold.

Parameters:

- `thermostatName` (string): Thermostat name
- `hold` (boolean): `true` to hold, `false` to release

Returns: `true` if the object exists and the command was sent, `false` otherwise

`setHeatSetpoint(thermostatName, temperature)`

Set the heating setpoint, in the controller's configured Fahrenheit or Celsius.

Parameters:

- `thermostatName` (string): Thermostat name
- `temperature` (number): -1 to 122 F, or -18 to 50 C; values outside the controller's range are rejected

Returns: `true` if the object exists and the command was sent, `false` otherwise

`setCoolSetpoint(thermostatName, temperature)`

Set the cooling setpoint, in the controller's configured Fahrenheit or Celsius.

Parameters:

- `thermostatName` (string): Thermostat name
- `temperature` (number): -1 to 122 F, or -18 to 50 C; values outside the controller's range are rejected

Returns: `true` if the object exists and the command was sent, `false` otherwise

`adjustHeatSetpoint(thermostatName, degrees)`

Move the heat setpoint by a relative amount, without reading it first.

Parameters:

- `thermostatName` (string): Thermostat name
- `degrees` (number): -50 to 50, in the controller's configured temperature format

Returns: `true` if the object exists and the command was sent, `false` otherwise

`adjustCoolSetpoint(thermostatName, degrees)`

Move the cool setpoint by a relative amount, without reading it first.

Parameters:

- `thermostatName` (string): Thermostat name
- `degrees` (number): -50 to 50, in the controller's configured temperature format

Returns: `true` if the object exists and the command was sent, `false` otherwise

Security Control

`setSecurityMode(areaName, mode)`

Arm or disarm an area.

```
MyroAPI.setSecurityMode("Main", "away");
MyroAPI.setSecurityMode("Main", "night");
MyroAPI.setSecurityMode("Main", "off");
```

Parameters:

- `areaName` (string): Area name, exactly as the controller spells it
- `mode` (string): One of `"off"` (or `"disarm"`), `"day"`, `"night"`, `"away"`, `"vacation"`, `"day_instant"`, `"night_delayed"`

Returns: `true` if the area exists and the command was queued for the controller, `false` otherwise. This does not confirm the area's resulting state; observe a later area-status event when confirmation matters.

`disarm(areaName)`

Disarm an area. Shorthand for `setSecurityMode(areaName, "off")`.

```
MyroAPI.disarm("Main");
```

Returns: `true` if the area exists and the command was queued for the controller, `false` otherwise. This does not confirm the area's resulting state.

- **A saved ELK arming code is required.** Configure it locally in Helper's ELK Connection settings; Helper stores it in this Mac's Keychain. Scripts and HomeKit use that code. Without it, arming/disarming is refused. Keep the code out of script source. Submission does not confirm a changed area state; observe the reported status.
- Direct HTTP security is separate and requires an API key with arm/disarm enabled plus a panel-validated code. Remote script management requires its own highly privileged API key. That permission allows scripts to use the locally configured ELK arming code; it does not supply a code or bypass a missing one.

> **Think about what re-runs.** An event handler that disarms will disarm every time its event fires. Guard it with a flag or a time check, and prefer arming over disarming when a script is deciding on its own.

> **Keep it local.** Never expose the Helper HTTP service, port 9393, or an MCP/API credential directly to the public Internet. A script-management key is equivalent to remote code-execution authority over the devices the Helper can control.

Macro/Button Control

`runMacro(macroName)`

Execute a macro or button defined in your controller.

```
MyroAPI.runMacro("Good Night");
MyroAPI.runMacro("Coffee Maker");
MyroAPI.runMacro("Morning Routine");
```

Parameters:

- `macroName` (string): Macro or button name

Returns: `true` if successful, `false` if macro not found

Use cases:

- Execute pre-programmed controller macros
- Trigger button actions
- Complex multi-device sequences

MyroController - Controller & Object Data

- MyroController.getStatus() returns the normalized ELK snapshot including backend, capabilities, connection, troubles, solar times and freshness information.
- getObjects(type), getObject(type, number), getObjectByName(type, name) return cached objects, or null when a requested object is unavailable. Use refreshStatus() and onStatusRefresh to confirm a refresh.

Controller capabilities

- getStatus().backend is "elk". Read capabilities before using optional features. Cross-controller handles apply the destination backend's rules.
- solarSchedules is true; solarConfigured, solarSource ("calculated") and solarTimeZone describe the saved location. Solar times are null without setup or when no event occurs today.
- energyCost, batteryReading, freezeStatus, upbLinks, locks, accessControl, storedMessages, userSettings, audioSourceMetadata, systemStatusEvents, humidity, thermostatFanCycle, audioMuteState, absoluteAudioMute, areaLights and cameraCommands are false.
- energyCost, battery.reading/percent, freezeStatus, features, modelCode, thermostat humidity/humidity setpoints/actionMode/freezeAlarm, audio mute and source metadata are null. Phone status is "ok" or "trouble".

- Scalar unavailable values: device state -1, temperature -999, freshness false. Test these before using a value.
- Sense energy and weather integrations remain available when configured; HAI energy-cost level is a separate unsupported feature. Never substitute an EnergyCost counter.
- Unsupported controls return false before transport/cache updates/events. Unsupported registrations return false and register no handler.

Object inventory

- getObjects(type) returns objects of that type; "*", "all" or an empty string requests every supported object. getObject(type, number) and getObjectByName(type, name) return one object or null.
- ELK inventory includes configured Zone, Unit, Button, Area, Thermostat, Auxiliary Sensor, Flag, Output, Audio Zone and Audio Source objects. Unsupported HAI families are omitted.
- Common fields: type, typeCode, number, name, displayName, state, value, updatedAt. Snapshots are read-only; they do not confirm a device action.

`refreshStatus()`

Returns: A request-ID string when a lightweight status refresh is accepted, or `null` when offline/unavailable or within the 30-second cooldown. This is not a completion or command acknowledgement.

- Requests standard status for known ELK zones, units/counters/outputs, areas, thermostats, auxiliary sensors and audio zones. It does not download names or scan unused capacity.
- Concurrent calls share the in-flight request ID instead of adding panel traffic. Automatic startup/reconnect refreshes use the same tracking.
- Finishing a configuration sync supersedes an older in-flight refresh so newly discovered objects are included. The old listener receives `superseded` with `success: false` and its original request ID; the new refresh has a new ID.
- Register `ScriptEvents.onStatusRefresh` first and match `result.requestID`. Confirm `result.success` before using the refreshed data. A timeout leaves the connection alone.
- Standard replies update the cache without replaying zone/unit changes, alarm actions, or push alerts. A dedicated refresh-result event is delivered instead.
- This confirms cached observations received during the refresh, not an atomic panel snapshot, exact request/reply sequence correlation, or completion of a device action. An intervening live event can legitimately provide a newer value.
- Audio-source metadata and extended-only thermostat humidity are not part of this refresh. ELK source metadata is unavailable. Object freshness does not promise every optional field was refreshed.

```
var refreshID = null;
ScriptEvents.onStatusRefresh(function(result) {
    if (result.requestID !== refreshID) return;
    if (!result.success) {
        MyroSystem.log("Refresh " + result.state + "; missing=" + result.missingCount);
        return;
    }
    var garage = MyroController.getObjectByName("Zone", "Garage 1");
    MyroSystem.log("Confirmed garage snapshot: " + JSON.stringify(garage));
});
refreshID = MyroController.refreshStatus();
if (!refreshID) MyroSystem.log("Refresh not accepted; check connection/cooldown");
```

`getRefreshStatus()`

Returns: An object describing the latest refresh; safe to read without requesting more panel traffic.

Field	Meaning
`requestID`	Refresh ID, or null before a refresh
`state`	`idle`, `queued`, `refreshing`, `complete`, `empty`, `timedOut`, `failed`, `superseded`, `disconnected`, or `unavailable`
`success`	True only for `complete`; disconnect invalidates a previous successful result
`expectedCount`, `confirmedCount`	Unique standard-status objects expected and stored
`missingCount`, `missing`	Total missing objects and up to 50 diagnostic `statusType:number` identifiers; auxiliary sensor type is Helper's `110` alias
`startedAt`, `finishedAt`	ISO-8601 timestamps, or null

- Check `getRefreshStatus()` when installing a listener after startup: terminal events are not replayed to new listeners. `unavailable` may contain only `state` and `success`.
- The wait is bounded to 120 seconds plus five seconds per planned request, capped at ten minutes. A timeout can contain useful partial results, but `success` remains false. An empty inventory is `empty`, not a successful refresh.

MyroAudio - Audio Zone Control

- ELK audio uses the adapter's mapped Russound keys and requires configured panel audio hardware. Zone/source parameters are numeric. Power, volume, source and documented transport controls remain available. Mute state and source metadata are null.

`power(zone, on)`

Turn an audio zone on or off.

```
MyroAudio.power(1, true);   // Zone 1 on
MyroAudio.power(1, false);  // Zone 1 off
```

Parameters:

- `zone` (number): Audio zone number (1-n; `0` = all zones)
- `on` (boolean): `true` = on, `false` = off

Returns: `true` if the command was sent, `false` otherwise

`setVolume(zone, percent)`

Set a zone's volume.

```
MyroAudio.setVolume(1, 30);  // 30%
MyroAudio.setVolume(1, 0);   // Silent (zone still on)
```

Parameters:

- `zone` (number): Audio zone number
- `percent` (number): Volume 0-100 (automatically clamped)

Returns: `true` if sent, `false` otherwise

`setSource(zone, source)`

Select the audio source for a zone.

```
MyroAudio.setSource(1, 2);  // Zone 1 -> source 2
```

Parameters:

- `zone` (number): Audio zone number
- `source` (number): Source number (1-n, as shown in the Object Editor)

Returns: `true` if sent, `false` otherwise

Transport — `play(zone)`, `pause(zone)`, `stop(zone)`, `next(zone)`, `previous(zone)`

Transport controls for a zone (Russound key mapping).

```
MyroAudio.play(1);
MyroAudio.pause(1);
MyroAudio.stop(1);
MyroAudio.next(1);
MyroAudio.previous(1);
```

Parameters:

- `zone` (number): Audio zone number

Returns: `true` if sent, `false` otherwise

`volumeUp(zone)` / `volumeDown(zone)`

Step a zone's volume up or down by one increment.

```
MyroAudio.volumeUp(1);
MyroAudio.volumeDown(1);
```

Parameters:

- `zone` (number): Audio zone number

Returns: `true` if sent, `false` otherwise

`sendKey(zone, code)`

- zone: audio zone number. code: only 1 (power), 2 (source step), 3/4 (volume up/down), 5 (mute toggle), 6 (play), 7 (stop), 8 (pause), 11 (previous), 12 (next).
- Key 5 toggles mute; there is no readable mute state or absolute mute setter. Other keys return false. **Returns:** Bool: sent or rejected.

Audio source metadata

`getSources()`

- Returns configured cached sources as an array of {number, name, displayName, metadata}; metadata is null for ELK. There is no source-metadata retrieval or refresh on ELK.

Example: "Welcome Home" Music on Disarm

```
// Play streaming music in the Kitchen when the alarm is disarmed.
// (Run on demand, on a schedule, or from a security/flag event handler.)
var KITCHEN = 1;     // audio zone number
var STREAMING = 2;   // audio source number

MyroAudio.power(KITCHEN, true);
MyroAudio.setSource(KITCHEN, STREAMING);
MyroAudio.setVolume(KITCHEN, 30);
MyroAudio.play(KITCHEN);
MyroSystem.log("Welcome-home music started in the Kitchen");
```

> **Tip:** Pair with `ScriptEvents.onAudioZoneChange(...)` (see Audio Zone Changes) to react to changes a Russound keypad or app makes and keep other systems in sync.

MyroSensors - Read Device States

Read current states and values from devices and sensors.

⚠️ **Important:** Sensor reads return the Helper's latest cached value. Check the controller connection when a current reading matters. Use `isDataFresh` only when the automation specifically requires proof that this object was confirmed during the current connection.

`getDeviceState(deviceName)`

Get device state (0 = off, 1+ = on/level, -1 = not found).

```
const state = MyroSensors.getDeviceState("Kitchen Light");
// Returns: 0 (off), 1 (on), 2-99 (dimmed), -1 (not found)

if (state === 1) {
    MyroSystem.log("Kitchen light is on");
} else if (state > 1) {
    MyroSystem.log("Kitchen light is dimmed to " + state + "%");
}

// Check flag state
const isArmed = MyroSensors.getDeviceState("System Armed");
if (isArmed === 1) {
    MyroSystem.log("System is armed");
}
```

Parameters:

- `deviceName` (string): Device, flag, or sensor name

Returns:

- Number representing device state
- `-1` if device not found
- `0` = off/clear
- `1` = on/set
- `2-99` = dimmed percentage
- `0-255` = flag counter value

`getTemperature(sensorName)`

- sensorName: exact temperature sensor or thermostat name. Returns cached temperature in the controller's configured Fahrenheit/Celsius units, or -999 when unavailable. Check isDataFresh before acting when a current reading matters.
- ELK thermostat humidity is unavailable; do not treat a missing humidity value as zero or invent a humidity sensor. **Returns:** Number (temperature), -999 when unavailable.

`isDeviceOn(deviceName)`

Check if device is on (convenience method).

```
if (MyroSensors.isDeviceOn("Kitchen Light")) {
    MyroSystem.log("Kitchen light is currently on");
    MyroAPI.turnOff("Kitchen Light");
}

// Check before turning on
if (!MyroSensors.isDeviceOn("AC Unit") && temp > 76) {
    MyroAPI.turnOn("AC Unit");
}
```

Parameters:

- `deviceName` (string): Device name

Returns: `true` if device is on (state > 0), `false` otherwise

`isDataFresh(deviceName)`

Each connection queues a lightweight snapshot of known ELK zones, units/counters/outputs, areas, thermostats, auxiliary sensors and audio zones after configuration is ready. The asynchronous refresh updates the shared cache without replaying device-change handlers or alarm notifications. A script running at startup must wait for confirmation. ELK thermostat humidity is unavailable; object freshness does not promise every optional field is available or has been refreshed.

Check whether MHH confirmed the object's value during the current controller connection. A value remains fresh while that connection stays active, even if it has not changed recently. After a disconnect or restart, the result remains `false` until MHH receives a new status for that object.

```
if (MyroSensors.isDataFresh("Outdoor Temp")) {
    const temp = MyroSensors.getTemperature("Outdoor Temp");
    MyroSystem.log("Current outdoor temp: " + temp + "°");
} else {
    MyroSystem.log("Warning: Outdoor temp sensor data is stale");
}

// Use this guard only when this decision requires a value confirmed
// during the current controller connection.
if (MyroSensors.isDataFresh("Main Thermostat")) {
    const temp = MyroSensors.getTemperature("Main Thermostat");
    if (temp > 76) {
        MyroAPI.turnOn("AC Unit");
    }
} else {
    MyroSystem.log("Skipping automation — temperature data is stale");
}
```

Parameters:

- `deviceName` (string): Device or sensor name

Returns: `true` if the object was confirmed during the current live controller connection; `false` after startup or reconnect until a new status is received, while disconnected, or if the device is not found.

Note: This is a connection-session confirmation flag, not a time-to-live or "recently changed" test. It is optional for ordinary state checks, including zones. Do not add it automatically to every script.

Event-Based Scripts

Available Event Types

Zone Status Changes

Monitor security zones for state changes.

Method Signature:

```
ScriptEvents.onZoneChange(zoneName, callback)
```

Parameters:

- `zoneName` (string): Zone name (case-sensitive, must match controller exactly), or `"*"` for every zone
- `callback` (function): Handler function to execute when zone status changes

Callback Function Signature:

```
function(zoneName, status, details)
```

Callback Parameters:

- `zoneName` (string): Zone name that changed
- `status` (string): One-word summary of the zone, in priority order:
  - `"unknown"` - No confirmed status yet, or the received byte contains an undefined condition/alarm value. Do not treat this as an alarm or a secure zone.
  - `"tripped"` - Zone has tripped (latched alarm)
  - `"trouble"` - Zone has a trouble condition
  - `"bypassed"` - Zone is bypassed, by the user or by the system
  - `"notReady"` - Zone is open / not ready
  - `"ready"` - Zone is secure
- `details` (object): The decoded status byte, for when the summary is not enough:
  - `raw` (number): The Helper-normalized zone status value
  - `condition` (string): `"secure"`, `"notReady"` or `"trouble"`
  - `latched` (string): `"secure"`, `"tripped"` or `"reset"` (was tripped, since reset)
  - `arming` (string): `"disarmed"`, `"armed"`, `"bypassedByUser"` or `"bypassedBySystem"`
  - `bypassed` (boolean): Shorthand for either bypassed value
  - `armed` (boolean): Shorthand for `arming === "armed"`
  - `troubleUnacknowledged` (boolean): A trouble occurred and has not yet been acknowledged; `condition` says whether trouble is still present now
- Detecting a restore: there is no separate restore event. A restore arrives as a zone change whose `details.bypassed` is `false` after it was `true`, so keep the previous value per zone and compare

Helper stores the decoded panel zone status consistently for properties and live status. At launch, cached zone states are marked unknown until confirmed by a properties response, live event, or the zone-status snapshot requested on connection. This also refreshes quiet zones such as fire and water sensors. Snapshot reads update stored state without firing zone-change script handlers or push alerts. While status is unknown, detail fields are `"unknown"` and boolean convenience fields are false; those false values do not confirm that a zone is secure, disarmed, or restored.

A restore has no event of its own. It arrives as a zone change whose `details.bypassed` has gone from `true` to `false`, so keep the previous value per zone and compare, as in the second example below.

> **Changed in Helper 4.06:** `status` used to report `"bypassed"` for any zone in an armed area, restored or not, and `"notReady"` was never reported while armed. If a script relied on that, read `details.arming` instead.

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
// Clear any existing handlers first
ScriptEvents.clear();

// Register handler for front door zone
ScriptEvents.onZoneChange("Front Door", function(zoneName, status) {
    MyroSystem.log(zoneName + " status changed to: " + status);

    if (status === "tripped") {
        // Zone opened/tripped
        MyroAPI.turnOn("Entry Lights");
        MyroAPI.turnOnTimer("Entry Lights", 99); // Auto-off after 99 seconds
        MyroNotifications.sendWithSound("Front door opened", "Perimeter.caf");
        MyroSystem.playSound("Glass.aiff");
    } else if (status === "ready") {
        // Zone closed/ready
        MyroSystem.log("Front door closed");
    } else if (status === "trouble") {
        // Zone has trouble
        MyroNotifications.send("WARNING: Front door zone has trouble!");
    }
});
```

Example: announce bypass and restore on Sonos

```
ScriptEvents.clear();

// Previous bypass state per zone. Starts empty, so the first event
// for a zone is judged against "not bypassed".
var wasBypassed = {};

ScriptEvents.onZoneChange("*", function(zoneName, status, details) {
    var before = wasBypassed[zoneName] === true;
    var now = details.bypassed;
    wasBypassed[zoneName] = now;

    if (now && !before) {
        MyroSonos.speak(zoneName + " has been bypassed", "Kitchen");
    } else if (!now && before) {
        MyroSonos.speak(zoneName + " has been restored", "Kitchen");
    }
});
```

Unit/Device Status Changes

Monitor lights, switches, dimmers, and other units for state changes.

Method Signature:

```
ScriptEvents.onUnitChange(unitName, callback)
```

Parameters:

- `unitName` (string): Unit/device name (case-sensitive, must match controller exactly)
- `callback` (function): Handler function to execute when unit status changes

Callback Function Signature:

```
function(unitName, isOn, level)
```

Callback Parameters:

- `unitName` (string): Unit name that changed
- `isOn` (boolean): Whether the unit is on (`true`) or off (`false`)
- `level` (number): Dimmer level (0-100) - for non-dimmers, this is 0 when off, 100 when on

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
// Monitor garage door opener
ScriptEvents.onUnitChange("Garage Door", function(unitName, isOn, level) {
    if (isOn) {
        MyroSystem.log("Garage door opened");
        MyroNotifications.send("Garage door is open");
        MyroAPI.turnOn("Garage Lights");
    } else {
        MyroSystem.log("Garage door closed");
        MyroAPI.turnOff("Garage Lights");
    }
});

// Monitor dimmer level changes
ScriptEvents.onUnitChange("Living Room Dimmer", function(unitName, isOn, level) {
    MyroSystem.log(unitName + " is " + (isOn ? "on" : "off") + " at " + level + "%");

    if (isOn && level < 20) {
        MyroSystem.log("Warning: Dimmer is very low");
    }
});
```

Flags, Outputs and Auxiliary Sensors

- ScriptEvents.onFlagChange(name, callback): callback(flagName, value, isSet).
- ScriptEvents.onOutputChange(name, callback): callback(outputName, isOn).
- ScriptEvents.onAuxSensorChange(name, callback): callback(sensorName, value, status).
- Pass an exact name or "*" for all objects of the kind. ELK has no blind or user-setting events.

Audio Zone Changes

Monitor audio zones for power, volume, mute, or source changes.

Method Signature:

```
ScriptEvents.onAudioZoneChange(zoneName, callback)
```

Parameters:

- `zoneName` (string): Audio zone name (case-sensitive, must match controller exactly)
- `callback` (function): Handler function to execute when audio zone changes

Callback Function Signature:

```
function(zoneName, power, volume, mute, source)
```

Callback Parameters:

- `zoneName` (string): Audio zone name that changed
- `power` (boolean): Power state (`true` = on, `false` = off)
- `volume` (number): Volume level (0-100)
- `mute` (null): The panel does not report mute state; do not infer it from a toggle command.
- `source` (number): Source number currently selected

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
// Monitor living room audio zone
ScriptEvents.onAudioZoneChange("Living Room Audio", function(zoneName, power, volume, mute, source)
        {
    MyroSystem.log("Audio Zone: " + zoneName);
    MyroSystem.log("Power: " + (power ? "On" : "Off"));
    MyroSystem.log("Volume: " + volume + "%");
    MyroSystem.log("Mute state is unavailable on this controller");
    MyroSystem.log("Source: " + source);

    if (power && volume > 70) {
        MyroNotifications.send("Audio volume is high in " + zoneName);
    }

    // Power and volume do not confirm playback or mute state.
});
```

Thermostat Changes

Monitor thermostats for temperature and mode changes.

Method Signature:

```
ScriptEvents.onThermostatChange(thermostatName, callback)
```

Parameters:

- `thermostatName` (string): Thermostat name (case-sensitive, must match controller exactly)
- `callback` (function): Handler function to execute when thermostat changes

Callback Function Signature:

```
function(thermostatName, temperature, mode)
```

Callback Parameters:

- `thermostatName` (string): Thermostat name that changed
- `temperature` (number): Current temperature (in °F or °C based on system setting)

- `mode` (string or null): System mode - one of:
  - `"heat"` - Heating mode
  - `"cool"` - Cooling mode
  - `"auto"` - Auto mode
  - `"off"` - Off mode
  - `null` - Mode not available

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
// Monitor main thermostat
ScriptEvents.onThermostatChange("Main", function(thermostatName, temperature, mode) {
    MyroSystem.log("Thermostat: " + thermostatName);
    MyroSystem.log("Temperature: " + temperature + "°");
    MyroSystem.log("Mode: " + (mode || "unknown"));

    if (temperature > 76) {
        MyroNotifications.send("Temperature high: " + temperature + "°F");

        // Only turn on AC if not already on
        if (!MyroSensors.isDeviceOn("AC Unit")) {
            MyroAPI.turnOn("AC Unit");
            MyroSystem.log("AC turned on due to high temperature");
        }
    } else if (temperature < 68) {
        MyroNotifications.send("Temperature low: " + temperature + "°F");
    }

    // Log mode changes
    if (mode) {
        MyroSystem.log("Thermostat mode: " + mode);
    }
});
```

Area Status Changes

Monitor security areas for mode changes (armed/disarmed states).

Method Signature:

```
ScriptEvents.onAreaChange(areaName, callback)
```

Parameters:

- `areaName` (string): Area name (case-sensitive, must match controller exactly)

- `callback` (function): Handler function to execute when area status changes

Callback Function Signature:

```
function(areaName, status)
```

Callback Parameters:

- `areaName` (string): Area name that changed
- `status` (number): Current area status/mode:
  - `0` - Disarmed/Off
  - `1` - Day mode
  - `2` - Night mode
  - `3` - Away mode
  - `4` - Vacation mode
  - `5` - Instant mode
  - Other values may be used by your controller

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
// Monitor house security area
ScriptEvents.onAreaChange("House", function(areaName, status) {
    MyroSystem.log("Area " + areaName + " status: " + status);

    if (status === 3) {  // Armed away
        MyroNotifications.sendWithSound("System armed away", "Arm.caf");
        MyroAPI.runMacro("Away Mode");
        MyroAPI.setFlag("System Armed");
    } else if (status === 0) {  // Disarmed
        MyroNotifications.sendWithSound("System disarmed", "Disarm.caf");
        MyroAPI.clearFlag("System Armed");
    } else if (status === 2) {  // Night mode
        MyroSystem.log("System armed in night mode");
        MyroAPI.setLevel("Bedroom Lights", 20); // Dim lights
    } else if (status === 1) {  // Day mode
        MyroSystem.log("System armed in day mode");
    }
});
```

Connection Status Changes

Monitor controller connection status.

Method Signature:

```
ScriptEvents.onConnectionChange(callback)
```

Parameters:

- `callback` (function): Handler function to execute when connection status changes

Callback Function Signature:

```
function(isConnected)
```

Callback Parameters:

- `isConnected` (boolean): Connection state (`true` = connected, `false` = disconnected)

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
// Monitor controller connection
ScriptEvents.onConnectionChange(function(isConnected) {
    if (isConnected) {
        MyroSystem.log("Connected to controller");
        MyroNotifications.send("Controller connected");

        // Refresh sensor data when reconnected
        MyroSystem.log("Refreshing sensor data...");
    } else {
        MyroSystem.log("Disconnected from controller");
        MyroNotifications.send("WARNING: Controller disconnected!");

        // Disable automations that require connection
        MyroSystem.log("Automations requiring connection are disabled");
    }
});
```

Ethernet Quality

Not available for ELK in the preview. On an ELK controller this registers no handler and logs that it is not available for NESS / ELK controllers.

Fires on the hourly Ethernet Quality baseline and on every controller-link incident, whether or not Ethernet Quality logging to CSV is enabled for the controller.

Method Signature:

```
ScriptEvents.onEthernetQuality(callback)
```

Parameters:

- `callback` (function): Handler function to execute for each sample

Callback Function Signature:

```
function(event, quality)
```

Callback Parameters:

- `event` (string): `"baseline"` (hourly), or the incident: `"crcError"`, `"unexpectedDisconnect"`, `"halfOpen"`, `"connectionFailure"`, `"ackTimeout"`
- `quality` (object): The same object `MyroSystem.getEthernetQuality()` returns, plus `event` and `count` (how many of that incident this sample represents; `0` for a baseline)

For `"ackTimeout"`, `count` can be greater than one when several commands lose their confirmations during the same incident. Treat that as one episode rather than several independent Ethernet failures. Do not notify on an isolated ACK timeout alone; corroborate it with a disconnected controller, repeated incidents, or another controller-link signal.

Returns: `true` if handler registered successfully, `false` otherwise

Example:

```
ScriptEvents.onEthernetQuality(function(event, q) {
    if (event === "baseline") { return; }
    MyroMQTT.publish("myrohome/controller/link", JSON.stringify(q));
    if (!q.connected && q.macNetworkUp) {
        MyroNotifications.send("Controller link " + event
            + " while this Mac is online");
    }
});
```

Status refresh completion

`onStatusRefresh(callback)`

Method Signature:

```
ScriptEvents.onStatusRefresh(callback)
```

Parameters:

- `callback` (function): Called once when an accepted refresh completes, times out, fails, finds no objects, is superseded by a configuration refresh, or is interrupted by disconnect. It receives the `getRefreshStatus()` result object plus `controllerGUID`.

Returns: `true` when registered, `false` on registration failure.

- Register before `MyroController.refreshStatus()` and match the returned ID against `result.requestID`. Check `result.success`; partial data is not success.
- The listener also sees automatic startup/reconnect refresh results. A new listener does not replay earlier results; read `getRefreshStatus()` for those.
- This event reports cache readiness, not an alarm, a physical device transition, or acknowledgement of an action. The usual zone/unit handlers are not fired by snapshot reads.

Scheduled Scripts

ELK solar schedule setup

- Choose Sunrise/Sunset or a solar cron token, then enter latitude, longitude and an IANA time zone and click **Save Location**. The location persists in Core Data and applies to this controller's scripts.
- No JavaScript, HTTP or MCP setter exists for this location. Without a valid location, solar timers wait. Do not invent coordinates or a location API.
- Offsets range from -720 to +720 minutes. Day/month/weekday restrictions use the actual run date after the offset in the saved time zone.
- Helper recalculates daily, handles DST, location changes, clock changes and wake, and searches for the next real solar event when today's event does not occur. It must remain running and the Mac awake.

HTTP and MCP controller operations

Use the shared HTTP API guide for authentication, service addresses, read endpoints and script management. The MCP guide covers client installation and local permission setup. Read the resolved controller's inventory and live scripting reference before selecting targets or generating scripts. Tool availability also depends on the installed MCP adapter version.

Object numbering and reads

- Zones, areas, thermostats, tasks/buttons, lights and temperature sensors retain their ELK numbers.
- Outputs use `256 + panel output number`; counters are Flag objects numbered `1000 + panel counter number`. Counter reads above 255 are capped by Helper's representation.
- `GET /helper/{controllerID}/status` returns the normalized panel snapshot. Unavailable fields stay null; do not infer a battery reading, humidity or energy-cost level.
- `GET /helper/{controllerID}/objects` and `/object/{type}/{number}` return the configured supported inventory. Audio-source reads expose configured cached names with null metadata.
- Panel events come from the panel event log; HAI stored-message and access-control inventories are unavailable.

Typed control

`POST /helper/{controllerID}/control` uses the following native mappings. Use the Helper object number from the inventory. Commands depend on configured panel hardware and an active connection. Submission/acceptance is not confirmation of a physical result; never retry an uncertain relative command automatically.

Action	ELK parameters and behavior
`set_unit`	`number`, `value` 0-100; optional `timer` 0-99 seconds for supported timed on
`set_output`	Helper output `number`, boolean `on`; optional timed on `timer` 0-99; timed off rejected
`set_flag`	Helper Flag `number`, `value` 0-255; writes native counter, without a timer
`increment_flag`, `decrement_flag`	Helper Flag `number`; cached relative counter write, no timer/value/amount
`dim_unit`, `brighten_unit`	Configured Unit `number`, integer `steps` 1-9; cached relative write of 10 percentage points per step
`run_button`	Task `number`; invokes the native ELK task
`set_thermostat`	`number`, `operation` mode/heat_setpoint/cool_setpoint/fan/hold, `value`; fan auto/on only
`set_audio_zone`	Numeric `zone`, `operation` power/volume/source/key/transport, `value`; no absolute mute

Action	ELK parameters and behavior
`message`	`operation` show/clear; show uses literal `text`, not a stored-message number

Relative counter/light helpers use cached readings as described in the scripting reference above; they are not atomic native panel commands. Relative thermostat setpoints also derive an absolute target from the latest cached thermostat reading. Confirm fresh data and do not automatically retry an uncertain result.

The mapped audio keys are 1, 2, 3, 4, 5, 6, 7, 8, 11 and 12. Key 5 toggles mute, whose state is unknown. Unsupported audio keys, humidity targets, blinds, locks, UPB links, user settings, area lights and `set_energy_cost` are rejected. Legacy energy-cost requests are rejected too; no EnergyCost counter is substituted.

Keypad text accepts up to 32 characters as two 16-character lines. `presentation: "beep_led"` requests a beep; other presentations are silent. Optional `area` is 1-8 (otherwise every area), and `seconds` defaults to 60. Clear requires no message number.

```
{"action":"message","operation":"show","text":"Garage door
        open","presentation":"beep_led","seconds":120}
```

```
{"action":"set_flag","number":1001,"value":10}
```

Security control

`POST /helper/{controllerID}/security` requires an API key with arm/disarm enabled. The panel validates four- or six-digit codes according to its configured code length. Arming/disarming requires the request's panel code. Zone bypass/restore without a supplied code requires the arming code saved locally in Helper's ELK settings; without one it is refused. Do not assume a Programmed-user fallback.

Scripts and HomeKit use the arming code saved in this Mac's Keychain. MCP direct security uses the client-side stored code and explicit local permissions described in the shared MCP guide. Keep codes out of scripts, source control and chat.

Weather and Sense integration

Configured native counters named RainAdvisory, MonthKWH, TodayKWH, ProjectedKWH and AvgDailyKWH support the existing Helper integrations. RainAdvisory remains forecast rain in whole millimeters capped at 255. Failed weather requests retain the last value and successful-sync timestamp; a valid dry forecast writes zero. Sense measurements are distinct from unsupported HAI energy-cost levels.