Myro:Home Helper

Helper Scripting Guide

Version 5.50

Shared Helper APIs, integrations, script management, scheduling, and controller-scoped AI authoring.

Myro Control, LLC | myrocontrol.com

Contents

Introduction

- Key Features
- ⚠️ Important Disclaimers

Getting Started

- Opening the Scripting Window
- Creating Your First Script
- Example of a shared Helper script

Writing Scripts with AI

- Copying the reference
- Using it
- Direct authoring through MCP
- What is included
- Before you enable an AI-written script
- Keeping it current

Script Management

- Script List
- Script Editor
- Execution Log
- Understanding Device Names

Siri and Shortcuts

- From Shortcuts and Siri into Helper (macOS 13+)
- From Helper into Shortcuts

JavaScript API Reference

- Savant controllers
- Control4 controllers
- MyroLibraries - Shared Local Modules (4.50)
- MyroControllers - Explicit Destinations (4.50)
- MyroSystem - System Functions
- MyroTimeline - Timeline Notes, Moments and Rules
- MyroTime - Date & Time
- MyroNetwork - Network & HTTP Requests
- MyroNotifications - Push Notifications
- MyroVestaboard - Vestaboard Display
- MyroLaMetric - LaMetric Time Display
- MyroLutron - Unified Lutron Control (LEAP + RA2)
- MyroLutronLEAP - Lutron LEAP Control
- MyroMQTT - MQTT Publishing
- MyroSonos - Sonos Announcements
- HTTP API from outside Helper

Event-Based Scripts

- Auto-Registration at App Launch
- Registering Event Handlers
- Available Event Types
- Example: Writing the Timeline from an Event Script
- Clearing Event Handlers
- Event Handler Best Practices

Scheduled Scripts

- Creating a Schedule
- Interval-Based Scheduling
- Cron-Based Scheduling
- Sunrise/Sunset Scheduling
- Schedule Management
- Schedule Behavior
- Schedule Best Practices

Import/Export Scripts

- ⚠️ Security Warning
- Exporting Scripts
- Importing Scripts
- Import/Export Best Practices

Script Templates

- Accessing Templates
- Available Template Categories
- Example Templates

Best Practices

- Script Organization
- Connection Requirements
- Error Handling
- Performance
- Security

Security & Safety

- ⚠️ Important Security Notes
- Script Execution Limits
- Connection Requirements
- Imported Scripts Safety
- Script Validation
- Error Handling Philosophy
- Best Practices for Security

Troubleshooting

- Script Won't Run
- Event Handler Not Firing
- Device Not Found
- Stale Sensor Data
- Notifications Not Sending
- Import/Export Issues
- Common JavaScript Errors
- Getting Help

Appendix: JavaScript Basics

- Variables
- Functions
- Conditionals
- Loops
- Arrays
- String Concatenation
- Timers

Conclusion

- Final Reminders

Introduction

This guide covers the shared Helper scripting system: script management, JavaScript execution, scheduling, Helper APIs and external integrations. Controller commands, panel-event signatures, readings and solar setup are documented separately.

Use the HAI and Leviton Controller Guide, the NESS / ELK Controller Guide, the Savant Controller Guide, or the Control4 Controller Guide for your selected controller. The API Reference window retains feature badges. Copying an AI reference includes the shared guide and only the selected controller family; an unscoped mixed home includes explicitly labeled references for every configured controller family, including Savant and Control4.

Key Features

Helper also includes browser control, bounded history queries and panel-event history. Those are HTTP/MCP surfaces, not new JavaScript globals: do not invent `MyroHistory` or panel-history scripting methods. Use this guide's documented exports for scripts and the HTTP API/MCP guides for history tools. Historical runtime is based on reported action intervals; missing history is unknown, not evidence that equipment was off. HaikuHelper scripts must be ported to these APIs rather than pasted unchanged.

Helper 5.50 includes Siri and Shortcuts on the Mac, alongside Savant discovery, live room feedback, controls, Control4 Link room activities, and cross-controller scripting. `MyroSystem.runShortcut` runs a Mac shortcut and returns its text result to a callback, and `ScriptEvents.onShortcut` publishes a script function as a **Run Script Handler** action in the Shortcuts app after local approval. The Shortcuts app also gets **Set Light / Unit**, **Press Button / Macro** and **Get Object Status**. Incoming actions run in the background, even while the Mac is locked, so an always-on Helper Mac needs no one at it. See **runShortcut** and **ScriptEvents.onShortcut** under MyroSystem.

Helper 5.50 also lets scripts write the Timeline. `MyroTimeline.note` and `MyroTimeline.moment` add a line or a named moment to the Timeline that browser control, Myro:Home Pro and the Apple TV show; `mute`, `rename` and `mode` are rules that quiet a chatty sensor, reword an automatic event or hand the whole Timeline to your script; `events` and `moments` read it back. The Timeline now also records a light the panel changed quietly, learned from a status poll, marked `approximate`. See **MyroTimeline** below and **Writing the Timeline from an Event Script** under Event-Based Scripts.

Helper requires macOS 12 Monterey or later and is built with Xcode 27 for macOS 27. Speech now follows the Mac's System voice (choose a Siri, Premium or Enhanced voice in Spoken Content), "Myro" is pronounced "Mai-row" (MY-row), and every speak function takes an optional per-call voice (see **Voices** under MyroSystem, and **MyroSonos - Sonos Announcements**). `MyroSystem.runUserScript` can also run a user-installed executable after an explicit local opt-in; see its setup and security requirements below.

- **JavaScript-based scripting** - Write scripts using familiar JavaScript syntax (ECMAScript 5)
- **Event-driven automation** - Scripts can automatically run when zones, units, thermostats, or other devices change state
- **Scheduled execution** - Scripts can run on a schedule (interval-based, cron-based, or sunrise/sunset)
- **Import/Export** - Share and backup scripts using JSON format
- **Siri and Shortcuts** - Run Mac shortcuts from scripts, and offer script actions, lights, macros and status to the Shortcuts app and Siri

- **Timeline from scripts** - Add notes and named moments to the Timeline, quiet a chatty sensor, reword an event, or take over the Timeline entirely with `MyroTimeline`
- **Controller control** - HAI / Leviton, NESS / ELK, Savant, and Control4 use the supported actions documented in their controller guides
- **Savant media** - Discover rooms and activities, read live state, and control them with MyroSavant or permitted destination handles
- **Control4 rooms** - List rooms and Watch/Listen activities, read observed state, control them with `MyroControl4` (or `MyroControllers.get(guid).control4`), and react to room changes with `ScriptEvents.onControl4RoomChange`
- **Sensor readings** - Read supported panel values; availability follows your controller guide
- **Push notifications** - Send push notifications to iOS devices (with throttling protection)
- **Network & HTTP requests** - Check connectivity, ping URLs, and make HTTP GET/POST/PUT/DELETE requests to local or remote APIs
- **Lutron LEAP integration** - Direct LEAP protocol connection to Lutron LEAP-compatible processor for real-time bidirectional control
- **Script templates** - Pre-built templates to get started quickly
- **Autocomplete support** - Code editor provides autocomplete for all API methods

⚠️ Important Disclaimers

Before using the scripting system, please read the following:

1. **Scripts run with system-level access** - Scripts can control your home automation devices. Always test scripts before enabling them in production.

2. **Controller connection required** - Most automation scripts require an active connection to your controller. Scripts using `MyroAPI` or `MyroSensors` may fail or use stale data if the controller is not connected.

3. **Imported scripts are disabled by default** - For security, all imported scripts are disabled when imported. Review and validate imported scripts before enabling them.

4. **Scripts are stopped after 30 seconds** - A script or event handler that is still running 30 seconds in (for example a `while (true)` loop, or a loop that waits for a device state to change) is stopped at its next loop iteration or function call, and the log reads `Script "Name" was stopped after 30 seconds`. Use `setTimeout` / `setInterval` or an event handler for anything that repeats or waits. In the rare case a script cannot be stopped — a single native call that blocks — the log reads `Script "Name" is still running after 30 seconds and could not be stopped. Scripts for this controller are blocked. Restart Myro:Home Helper to restore scripting.` and the Scripts window shows a red banner until that script returns or you restart. The app itself and controller processing keep running either way.

5. **No code created at run time** - `eval()`, `new Function()` and `import()` are not available: code created while a script runs cannot be stopped if it loops forever. Validation refuses a script that uses them, with the line number. Names beginning with `__myro` are reserved for the script engine.

6. **Error handling** - Script errors are logged but do not crash the application. Always check execution logs for errors.

7. **No warranty** - Scripts are provided as-is. Always test scripts in a safe environment before deploying to production.

Getting Started

Opening the Scripting Window

1. Launch Myro:Home Helper
2. Select a controller from the main window
3. Click the **Scripts icon** in the toolbar (or use **Window** → **Scripting**)
4. The Scripting window will open, showing scripts for the selected controller

Creating Your First Script

> **Tip:** you never need to type a device name by hand. In the code editor, type `Unit.` (or `Zone.`, `Thermostat.`, `Flag.`, or `Device.` for everything) and pick from your controller's actual devices — the name is inserted correctly quoted. Device names must match the controller exactly, and a wrong one fails silently, so this is the safest way to write them.

1. Click the **"+"** button in the script list
2. Enter a name for your script (e.g., "Turn on lights at sunset")
3. Write your JavaScript code in the editor (autocomplete helps with API methods!)
4. Click **Save**
5. Toggle **Enabled** to activate the script

Example of a shared Helper script

```
MyroSystem.log("Helper script is running");
```

For device control and temperature examples, follow your controller guide and use exact names from its inventory.

Writing Scripts with AI

You do not have to write scripts from scratch. Myro:Home Helper can hand an AI assistant everything it needs to write one for you — the complete scripting API *and* your controller's actual device names.

Copying the reference

In the menu bar, choose **Help ▸ Copy AI Scripting Reference**. The entire reference goes to your clipboard, and the confirmation tells you which controller's device names were included — worth checking if you have more than one controller configured.

Help ▸ Save AI Scripting Reference... writes the same content to a Markdown file instead, which is useful if you want to attach it to a Claude Project, a custom GPT, or any assistant that supports persistent reference files. Attach it once and every future request already has the context.

Using it

Paste it into Claude, ChatGPT, or any AI assistant, then describe what you want in plain language:

> *"Write a script that turns on the porch light when the front gate opens after sunset, and turns it off again 10 minutes later."*

The reference begins with instructions telling the assistant to use only the documented functions and only device names that actually exist on your controller, so what comes back should run as-is. Paste the result into a new script in the Scripts window.

Direct authoring through MCP

With **Myro:Home Helper MCP 0.2.0** and Helper 4.07 or later, an assistant can complete the same workflow without manual copy and paste. It can call:

- `scripting_reference()` for this installed Helper's API and real names
- `list_scripts()` and `get_script()` to inspect current automations
- `list_devices()`, `list_picos()`, and `list_vestaboards()` to resolve targets
- `validate_script()` to parse and lint without executing
- `save_script()` to create or edit a script, disabled by default
- `enable_script()` after you explicitly approve activation
- `reload_scripts()` only when you explicitly request a full handler reload

For example, you can ask:

> *"Create a disabled script that sends a push notification and writes `Lights On` to my Vestaboard when Office Lts turns on. Validate it and show me any warnings before enabling it."*

The assistant should read the live reference and inventories, validate the result, save it disabled, and wait for approval before enabling it. When editing, it should use the immutable `scriptID` returned by `get_script()` so a duplicate name cannot select the wrong automation.

These are external MCP tools, **not JavaScript functions**. Do not place `listScripts()`, `saveScript()`, or similar calls inside script source. A running script cannot rewrite itself through the scripting API.

What is included

- Shared Helper API contracts, integrations and scheduling syntax.
- The selected controller family's explicit guide and its real device names.
- A version stamp matching the installed Helper.

Open the controller's Scripting window before copying. HTTP/MCP reference requests use the resolved controller ID. A scope with only one family includes only that family; an unscoped mixed home includes its HAI / Leviton, NESS / ELK, Savant, and Control4 contracts in labeled sections, as applicable. Cross-controller handles apply the destination's rules; request its reference before using controller features not documented for the current scope.

The API Reference window shows HAI / Leviton, ELK, ELK: limited, Savant, Control4 and Helper badges. Help also offers a separate controller guide for each family.

Before you enable an AI-written script

Treat it exactly like a script you wrote yourself:

1. **Read it.** You are the one responsible for what it does to your home.
2. **Validate it**, then run it once manually with the Run button and watch the Execution Log when a one-time run is safe. Event-only scripts can instead be enabled and tested with their intended event.
3. **Check the device names resolved** — a name that does not exist fails silently, so a script that "does nothing" usually means a name is wrong.
4. **Be careful with security and locks.** Have an assistant explain any script that arms, disarms, or unlocks before you enable it.
5. **Protect remote script management.** It can create and run any local automation, including security-capable scripts. Enable it only on a dedicated API key for a trusted client, and never expose the Helper API, port 9393, or MCP credentials directly to the public Internet.

Keeping it current

The reference is a snapshot. Copy it again after you add or rename devices on your controller, or after updating Myro:Home Helper, so the assistant is not working from a stale list.

Script Management

Script List

The left pane shows all scripts for the current controller:

- **Script name** - Click to select and edit
- **Enabled indicator** - Green checkmark means the script is enabled
- **Last run info** - Shows when script last executed
- **Error indicator** - Warning icon if script has errors
- **Search** - Filter scripts by name
- **Context menu** - Right-click for Duplicate, Export, Delete options

Script Editor

The main editor area includes:

- **Script Name** - Editable text field at the top
- **Enabled Toggle** - Enable/disable the script
- **Validate Button** - Check for JavaScript syntax errors and verify device names exist (shows warnings for missing devices)
- **Run Button** - Execute the script immediately
- **Save Button** - Save changes to the script
- **Code Editor** - Syntax highlighting and autocomplete for all API methods
- **Device name autocomplete** - Type `Unit.`, `Zone.`, `Thermostat.` (or `Device.` for everything) to pick from your controller's real device names; the chosen name is inserted already quoted, so it always matches exactly. See Understanding Device Names.
- **Schedule Section** - Configure scheduled execution (see Scheduled Scripts)

Execution Log

The bottom pane shows this script's output — from a manual Run, and live from its event handlers and timers as they fire, so "did my trigger fire?" is answerable without opening the main Log window. Tick **All scripts** to see every script on the controller together, which is the quickest way to find out which script did something. Clear removes what the pane is showing: this script's lines, or everything when All scripts is on. The copy button puts the log on the clipboard as plain text — with the script's source too, if you choose — ready to paste into Claude or ChatGPT with "what went wrong here?"

The pane shows:

- **Timestamp** - When the log entry was created
- **Log level icon** - Info (blue), Warning (orange), Error (red), Debug (gray)
- **Message** - Log message text
- **Clear button** - Clear the log

Understanding Device Names

Controller name or custom name — both work. If you have given a device a custom name under Edit Objects, scripts can use either that or the controller's original name; API calls, sensor reads, and event handlers all match both. Autocomplete and the AI reference offer the custom name when there is one, since that is the name you actually use. Event handlers still receive the controller's name in their arguments.

What are device names?

Device names are downloaded from the controller during initial setup and Sync Controller Data (Helper's Sync Now), then persisted in Core Data. Controllers reuse those stored names when the app opens or reconnects and refresh statuses and values. Names identify devices, zones, units, and macros in scripts.

⚠️ Critical: Device Name Requirements

1. **Exact match required** - Device names must match exactly as stored in the controller
2. **Case-sensitive** - "Kitchen Light" is different from "kitchen light"
3. **Downloaded from controller** - Sync Controller Data / Sync Now refreshes names; reconnects reuse the Core Data cache
4. **Validation checks device names** - The Validate button checks JavaScript syntax AND verifies device names exist
5. **Validation warnings** - If devices are not found during validation, warnings are shown (validation still passes, but you should verify device names)

⚠️ What happens if device names change?

If you rename a device on the controller:

- Scripts using the old name will **fail to find the device**
- Device control methods will return `false`
- Sensor methods will return `-1` or `-999`
- You must **update your scripts** to use the new device name
- Use Sync Controller Data / Sync Now to refresh the name database

Let the editor type the name for you

Rather than copying names by hand, type a device namespace followed by a dot in the script editor and pick from the list — the editor reads the real names from your controller and inserts the chosen one as a quoted string:

```
Unit.mast          // pick "Master Bed Room"  ->  becomes "Master Bed Room"
MyroAPI.turnOn(Unit.mast    // pick  ->  MyroAPI.turnOn("Master Bed Room")
```

Shared editor namespaces include `Device.`, `Zone.`, `Unit.`, `Button.`, `Area.`, `Thermostat.`, `AuxSensor.`, `Flag.`, `Output.`, `AudioZone.` and `AudioSource.`. Your controller guide lists its available object families and additional namespaces.

Matching is case-insensitive and also matches later words, so `Unit.bed` finds "Master Bed Room". The namespace itself is not JavaScript — it is replaced entirely by the quoted name when you choose one. Names come from the persisted controller database. After renaming devices on the panel, use Sync Controller Data /

Sync Now, then reopen the Scripts window if its list needs refreshing.

Two objects with the same name

Nothing stops a controller from having two objects with the same name. There are two quite different cases.

Different kinds — this is fine. A security zone and a light both called "Front Door" is an ordinary setup. Control methods (`turnOn`, `turnOff`, `setLevel`, `toggle`) always prefer something you can actually control, so they pick the light and never the zone. Nothing to do.

The same kind — this needs fixing. Two *units* both called "Porch Light" is genuinely ambiguous:

- The script resolves to the object with the **lowest number** and controls that one. It is consistent, but it may not be the one you meant.
- The script still reports success — you get no error, just the wrong device.
- Autocomplete lists the name once, because both objects insert the same text.

Myro:Home Helper warns you when this happens (and only then — the harmless zone-and-unit case stays quiet). The Execution Log and the main Log window will show:

```
Device name "Porch Light" matches 2 objects of the same kind on this controller
— using #14 (also matched #22). Rename one on the controller (appending its
number works well) so scripts can tell them apart.
```

The fix is to make the names unique on the controller. Appending the object number is the easiest convention — rename them to "Porch Light 14" and "Porch Light 22". Then use Sync Controller Data / Sync Now so Helper picks up the new names, and update any scripts that used the old ambiguous name.

Best Practices:

1. **Test scripts after creating** - Run scripts manually to verify all device names are correct
2. **Use exact names** - Copy device names from the controller interface to avoid typos
3. **Check return values** - Device methods return `false` if the device is not found
4. **Update after renaming** - If you rename devices on the controller, update all scripts that reference them
5. **Monitor execution logs** - Check logs for device not found errors

Example:

```
// Check if device was found
const success = MyroAPI.turnOn("Kitchen Light");
if (!success) {
    MyroSystem.log("ERROR: Device 'Kitchen Light' not found - check name or reconnect to
        controller");
}
```

Siri and Shortcuts

Helper works with the Mac Shortcuts app in both directions. Everything runs in the background, even while the Mac is locked, so an always-on Helper Mac can serve shortcuts and automations with no one at it.

From Shortcuts and Siri into Helper (macOS 13+)

In the Shortcuts app, search for **Myro:Home Helper**. Six actions are available:

Action	What it does
Set Light / Unit	Turns a light or unit on or off, or sets a level
Press Button / Macro	Runs a controller button (macro)
Get Object Status	Reads a zone, unit, thermostat or sensor, flag or output; offline or stale readings are an error
Arm Security	Arms an area in Day (Stay), Night, Away, Vacation, Day Instant or Night Delayed. No code is needed
Disarm Security	Disarms an area with your security code
Run Script Handler	Runs a function your script published with `ScriptEvents.onShortcut` and returns its text

1. To publish a script action, call `ScriptEvents.onShortcut("Name", function (input) { … })`, or start from **Templates > Utilities > Shortcut Action**.
2. Enable and save the script, then turn on **Allow this script's handlers in Siri & Shortcuts** in the script editor. Editing the code turns approval off until you allow it again.
3. Build a shortcut with the action, and name it to run it with Siri ("Hey Siri, Evening routine"). The first time a shortcut uses Helper, macOS asks you to allow it.

Arming and disarming

- **Arm Security** needs no code. Helper arms as the panel's Programmed user, the same way HomeKit and scripts do; on NESS / ELK panels it uses the arming code saved in the Mac's Keychain. European (EN 50131) HAI controllers refuse to arm without a code, so fill in the action's optional **Code** for those.
- **Disarm Security** needs your security code. The controller checks it, exactly as for the HTTP API and web control, and repeated wrong codes lock out further tries for a while. For an unattended disarm, such as a morning automation, save the code in the shortcut. Anyone who can open that shortcut can read the code, and Shortcuts syncs shortcuts through iCloud to your other devices (only a Mac running Helper can carry out the action). Choose **Ask Each Time** instead if you want someone to say or type the code.
- Every request appears in Helper's log as a Security entry from Mac Shortcuts, never with the code.
- A successful result means the controller accepted the request. Read the area's status if you need to confirm it.

An approved script handler or a macro can also perform security actions, so approve only scripts you trust. Shortcuts on an iPhone can't use the Mac's actions; use the Myro:Home apps or Helper's HTTP API.

From Helper into Shortcuts

`MyroSystem.runShortcut(name, input, callback)` runs any Mac shortcut and hands its text result to your callback. Start from **Templates > Utilities > Run a Shortcut**. The first time, macOS asks to let Helper control **Shortcuts Events** (**System Settings > Privacy & Security > Automation**). One shortcut runs at a time; a second call while one is running returns `null`.

Full details: runShortcut and ScriptEvents.onShortcut below.

JavaScript API Reference

Savant controllers

Add **Savant** from Helper's Add Controller menu. In **Connection**, use **Discover** or enter a host address and choose **Verify Host**. Confirm the host/home identity and certificate, then select a **Savant Local User** and enter that user's local password. This is not the Mac mini administrator login; the cloud password may differ. No cloud token or agent ID is required. **Remember Local User in Helper Keychain** saves an accepted login and also works when checked while already connected. Unchecking removes the saved login without disconnecting. Enable automatic connection separately if desired. See the Savant Controller Guide for setup and troubleshooting. Helper discovers the current WebSocket port rather than relying on a fixed port. Removing the controller removes its Helper credentials and inventory cache.

The **Rooms** tab loads room and AV source inventory and shows reported activity, volume, mute, and timestamps. Unknown values remain unknown. Disconnection marks feedback stale; reconnecting subscribes again and establishes a new baseline. Commands are sent once and are never replayed after reconnecting. Panel security and object editing are unavailable for a Savant controller. Its **HAP** tab exposes only selected Savant activities as Apple Home switches. An optional **Custom Name**, matching HAI/ELK terminology, sets a short label for Apple Home and Siri without changing activity routing or pairing identity. ON starts an activity, OFF targets only that activity, and live Savant feedback controls the reported state. Unknown or disconnected state reports **No Response**. Shared scripts and configured integrations such as Vestaboard remain available. PRO can connect directly to Savant with Helper optional; adding Savant here does not merge PRO rooms.

This is a preview of Savant's app protocol, not a vendor-supported API contract. Local User authentication needs no cloud token or agent identity. Compatibility depends on the installed host software and equipment. Inventory comes from the host's local HTTP configuration service without account credentials; account authentication and control use the pinned encrypted WebSocket.

MyroSavant

`MyroSavant` controls the script's own Savant controller. For another controller on this Helper, use `MyroControllers.get(guid).savant`; the origin's **Send To** and destination's **Accept From** grants are checked on every call. The generic `MyroAPI`, `MyroAudio`, and panel sensor APIs do not control Savant rooms.

Method	Behavior
`rooms()`	Cached rooms as `{id, name}`.
`services(room)`	AV sources with IDs, labels, variants, `commandsAvailable` and optional `commands` / `commandsUpdatedAt`.
`status(room)`	Observed state, or null for an unavailable room.
`start(room, service)`	Start the selected AV service, allowing Savant to route the activity.
`off(room)`	Stop all AV sources confirmed active in fresh feedback; a confirmed off room sends nothing.

Method	Behavior
`volume(room, level)`	Request an integer volume from 0 through 100.
`mute(room, muted)`	Request an absolute Boolean mute state.
`play(room)`, `pause(room)`, `stop(room)`, `next(room)`, `previous(room)`	Send transport requests when exactly one source is confirmed active and advertises the command. Multiple active sources fail.
`transport(room, service, action)`	Target one source with `play`, `pause`, `stop`, `next`, `previous`, or `off`. OFF leaves other sources alone.

Room/source arguments accept discovered IDs or an unambiguous name (case insensitive). Known older service UUIDs recover only through a unique saved route in the same trusted Savant system and room; unknown or ambiguous IDs fail. Duplicate names fail; use IDs. A command's `true` return means validated and queued; check the script log for sent/failure receipts and use observed feedback to determine the result. A sent request or protocol acknowledgement does not prove a physical TV turned on. Volume/mute writes and transport requests still require hardware validation on each installation.

`status(room)` includes `roomID`, `roomName`, `fresh`, `propertyUpdatedAt`, and `inventory` refresh status. Source commands require successful discovery; missing metadata is unknown and blocks commands until a refresh succeeds. The native Rooms pane shows service type/variant labels and enables Play, Pause, Stop, Next and Previous only for supported commands on the selected active source. Inventory refreshes on reconnect, manually, and every five minutes while connected. When reported, it also includes `activeService`, `activeServices`, `activeAudioService`, `activeVideoService`, `volume`, `muted`, `isActive`, `hasAudio`, `hasVideo`, and `updatedAt` (ISO date). Unknown optional fields are omitted rather than fabricated as false/zero. Activity values may be full Savant service strings; they are not necessarily the source's display name. Inspect each property's timestamp when freshness matters.

```
// Discover actual IDs/names before enabling a command script.
MyroSystem.log(JSON.stringify(MyroSavant.rooms()));
MyroSystem.log(JSON.stringify(MyroSavant.services("Theater")));

// Replace these with discovered IDs or unambiguous names.
// MyroSavant.start("Theater", "Apple TV");
```

ScriptEvents.onSavantRoomChange

`ScriptEvents.onSavantRoomChange(room, handler)` watches a room ID, unambiguous name, or `"*"` on the script's own Savant controller. The callback receives `(roomName, state)`. Numeric/string values are normalized before comparison. Initial snapshots and reconnect baselines do not trigger change actions. Callbacks run on the script queue and are removed when the script is stopped, disabled, or cleared. This method returns false for a non-Savant controller or an unresolved room. Cross-controller command handles do not subscribe to destination events.

```
ScriptEvents.onSavantRoomChange("Theater", function (roomName, state) {
    MyroSystem.log(roomName + ": " + JSON.stringify(state));
    // Call configured Helper integrations here, e.g. MyroVestaboard.
});
```

`MyroController.getStatus()` identifies `backend: "savant"` and reports native connection/inventory capabilities. `refreshStatus()` queues an inventory refresh and returns a queue marker; it does not represent a correlated refresh acknowledgement. `getRefreshStatus()` reports connection/status, room count, and the latest state update. Use `MyroSavant` for room state and controls.

Control4 controllers

Add **Control4 Link** from Helper's Add Controller menu and import the installer's connection file under **Connection > Import Trusted Connection...**. A Control4 controller has rooms and Watch/Listen activities, not panel objects: use `MyroControl4` (documented in the Control4 Controller Guide) for `rooms()`, `activities(room)`, `status(room)`, `start`, `off`, volume, mute and transport control, and `ScriptEvents.onControl4RoomChange(roomOrStar, handler)` for observed room changes. From another controller use `MyroControllers.get(guid).control4` with the same Send To and Accept From grants. `MyroController.getStatus()` identifies `backend: "control4"`. `refreshStatus()` asks the connector for its room inventory again and returns a `control4-inventory-...` queue marker (null before a connection file is imported). It is not a correlated acknowledgement: nothing is sent while Helper is not signed in to the connector, or within a second of the previous refresh. `getRefreshStatus()` reports `backend`, connection `state`, `statusMessage`, `roomCount`, `inventory` and `lastUpdated`. On a Control4 controller `getObjects` returns an empty list and `getObject` / `getObjectByName` return null, and panel `ScriptEvents` handlers such as `onZoneChange` and `onUnitChange` return false and register nothing. The generic `MyroAPI`, `MyroAudio` and panel sensor APIs do not control Control4 rooms.

MyroLibraries - Shared Local Modules (4.50)

`MyroLibraries.require(path)`

- Loads a UTF-8 JavaScript file from the Helper's own Documents `Scripts/Libraries` folder. Open it using **Scripts > Cross-Controller Access** (network icon) > **Open Libraries Folder**. Create/edit library files locally; there is no remote library-management API.
- Paths are relative, case-sensitive filesystem paths. The `.js` suffix is optional. Use letters, digits, `_`, `-`, `.`, and `/`; no empty components, `..`, absolute paths, backslashes, URLs, or symlinks (including folder/root symlinks).
- A module uses `exports.name = value` or `module.exports = value`. Its local `require("./child")` resolves beside that module; `require("folder/child")` resolves from Libraries. No Node built-ins, package resolution, ES imports, `eval`, or Function constructors.
- Exports and module-local variables are cached separately for each script/run. Another script gets its own instance. Manual rerun, code reload, disable/delete, or restart discards the old run's cache. Editing a library does not hot-reload existing exports: explicitly rerun the consuming scripts. Script import/export does not package library files.
- The engine still shares its JavaScript global scope between scripts. A global `var lighting` or controller handle can be overwritten/deleted by another script's execution. Capture helpers/handles in an IIFE-local

variable for retained callbacks, as below; per-run module caching does not isolate your script's global variables. Avoid repeated top-level `let` / `const` declarations on rerun.

- Libraries use the same syntax validation, reserved-name checks, dynamic-code lockdown, and loop/function instrumentation as script bodies. Initialization shares the caller's 30-second budget; later exported-function calls use the calling event/timer/native-callback budget. Loading never resets a deadline. Invalid/uninstrumentable source is not executed.
- Limits: 128 KiB per file, 1 MiB total loaded source per script/run, 64 modules including failed initializations, 16 nested loads. Cycles throw with the module chain; a failed initialization is not retried within that run. These are trusted shared code files, not an independent security sandbox or a place for secrets.

Returns: `module.exports` (object, function, or other JavaScript value). Throws on invalid path/source, load failure, cycle, expired require closure, or quota. Validation of a script body does not execute or preflight its filesystem dependencies; modules are validated when loaded.

Method Signature and Minimal Example:

```
// Scripts/Libraries/lighting.js
exports.offIfFresh = function(name) {
    return MyroSensors.isDataFresh(name) && MyroAPI.turnOff(name);
};
// Consuming script (not part of lighting.js)
(function() {
    var lighting = MyroLibraries.require("lighting");
    ScriptEvents.onUnitChange("Hall", function(name, state) {
        if (state > 0) lighting.offIfFresh("Porch");
    });
})();
```

MyroControllers - Explicit Destinations (4.50)

`MyroControllers.list()`

Returns: Array of `{guid, name, backend, connected}` (`backend` is `"hai"`, `"elk"`, `"savant"` or `"control4"`) for the originating controller and bilaterally permitted configured destinations. It does not expose credentials. There is no implicit first/default destination.

`MyroControllers.get(guid)`

Returns: A handle `{guid, controller, api, sensors, audio, savant, control4}`, or `null` when the GUID is invalid, unknown, not permitted, or the 64-distinct-handle engine limit is reached. Supply a full controller GUID, never a display name.

- Native permission setup: in the origin controller's Scripts window, open **Cross-Controller Access** and enable **Send To** for the destination. In the destination's Scripts window, enable **Accept From** for the origin. Both grants default off; each enable requires local confirmation. The sheet shows effective incoming/outgoing access. Either side can revoke immediately for future operations.
- A grant trusts **all scripts on the origin**, including scripts edited through its privileged script-management API key. It permits the destination's existing scripting controls, including arm/disarm. Ordinary

status/control keys gain no script or library management. Do not grant this trust to an origin whose scripts or script-management keys are not trusted for the destination.

- `handle.controller` has the same methods as `MyroController`; `handle.api` as `MyroAPI`; `handle.sensors` as `MyroSensors`; `handle.audio` as `MyroAudio`; `handle.savant` as `MyroSavant`; `handle.control4` as `MyroControl4`. Existing globals and ScriptEvents remain bound to the original controller. No cross-controller event subscription is introduced.
- Destination permission is rechecked on every operation, even after extracting `var api = handle.api`. Managers/connections are resolved by the destination GUID on each call, not from the selected UI controller or a saved connection. Bridges remain retained by the originating engine. Handles confer no independent authority outside an active script run.
- Commands require the destination connection. A true result means the existing bridge accepted/sent the request, not physical completion. Destination sensor reads require confirmation during its current uninterrupted connection: `getDeviceState` returns `-1`, `getTemperature` returns `-999`, and `isDeviceOn` returns `false` when unavailable/stale. Check `isDataFresh` before automation decisions. Cross-controller `toggle` also refuses stale state.
- Object inventory remains readable while offline and is cached data, not a live read. Each object includes `fresh` and, when available, `updatedAt`; `fresh` requires a sync timestamp at or after the destination connection began. Refresh requests and audio metadata retain their existing completion/cache semantics. Snapshot nulls/false values also indicate revoked permission.

Method Signature and Minimal Example:

```
(function() {
    var destinations = MyroControllers.list();
    var remote = MyroControllers.get("11111111-2222-3333-4444-555555555555");
    if (remote && remote.sensors.isDataFresh("Porch")) {
        remote.api.setLevel("Porch", 25);
    }
})();
```

- Never automatically retry an uncertain relative command. Read reported state and confirm the outcome before choosing another action.

MyroSystem - System Functions

System-level functions for logging, alerts, sounds, variable storage, and trusted user-installed programs.

`runShortcut(name, input, callback)`

Run a shortcut on this Mac from any Helper script or event handler. No wrapper installation is needed. The Mac and Helper must be running. Start with **Templates > Utilities > Run a Shortcut**, edit the light and shortcut names, then enable and save. macOS asks permission for Helper to control **Shortcuts Events** the first time. You can change it in **System Settings > Privacy & Security > Automation**. Actions inside the shortcut may have their own privacy prompts or require interaction.

```
var id = MyroSystem.runShortcut("Evening routine", "Kitchen", function (result) {
    if (result.success) {
        // result.output is the shortcut's text result.
        MyroSystem.log("Evening routine finished");
    } else {
        MyroSystem.log("Shortcut: " + result.state);
    }
});
if (!id) {
    MyroSystem.log("Shortcut not accepted; check the log before trying again");
}
```

- `name` is the exact Mac shortcut name, at most 1 KiB. `input` is text (at most 16 KiB) or `null` for empty input. Neither may contain NUL. `callback` must be a function. The call returns immediately with a request ID, or `null` when it cannot be admitted; rejected calls do not invoke the callback.
- The callback receives one result: `{id, kind, state, success, output, outputTruncated, completionUnknown, errorCode?, error?}`. State is `finished`, `failed`, `cancelled` or `timedOut`. Text output is limited to 32 KiB; oversized output is discarded and reported as failure. Use text as the shortcut's final output. Lists, files and other non-text results are not accepted. Missing output is an empty string.
- One outgoing shortcut runs at a time across Helper, with no waiting queue. This slot is separate from user-installed executables. A shortcut that asks for input can wait for the user. Native execution waits up to 30 seconds, with a 60-second overall deadline including launch. Timeout means completion is unknown. Resolve any pending shortcut, then restart Helper before another outgoing call. Helper never retries automatically.
- Disabling, deleting or rerunning the script discards its old callback. It does not terminate a shortcut already sent. Input/output and target error text are not automatically logged; take care before logging results yourself. The existing per-run callback limits also apply (8 per script run, 128 per engine).

`ScriptEvents.onShortcut(name, handler)`

Make a script action available to **Siri and Shortcuts on this Mac** (macOS 13+). Registration itself works on macOS 12, but incoming App Intents require 13+. The registered function is called with text input; its result becomes the action's output. Calling it does not rerun the script body or reset existing timers.

```
ScriptEvents.onShortcut("Kitchen temperature", function (input) {
    if (!MyroSensors.isDataFresh("Kitchen")) {
        return "Kitchen temperature is unavailable";
    }
    return MyroSensors.getTemperature("Kitchen");
});
```

1. Enable and save the script. In the script editor, turn on **Allow this script's handlers in Siri & Shortcuts**. Approval is local to this Mac and is not included in exported scripts. Changing the script's code requires approval again. Names can change without invalidating the approval.

2. In Mac Shortcuts, add **Myro:Home Helper > Run Script Handler**, choose the handler and optionally provide text. Name the shortcut to invoke it with Siri. The built-in **Run a script in Myro:Home Helper** phrase can also prompt for a handler; Siri's phrasing and app-name recognition are controlled by macOS.
3. Handlers run in the background, even while the Mac is locked. Approved handlers have the script's full authority, including code-free arm/disarm commands and any trusted libraries or external integrations it calls, for anything that can run a shortcut on this Mac. Approve only scripts you trust. Scripts remain responsible for checking fresh state.

`name` is a nonempty name of at most 128 UTF-8 bytes, without NUL. `handler` must be a function. Returns `true` when registered, otherwise `false`. A repeated name within the same script replaces that function. Identical names in different scripts/controllers remain separate choices. Up to 32 handlers per script and 512 across Helper can be registered. The list uses stable controller/script/name identities; renaming a handler creates a new choice for saved shortcuts.

The handler must return text, a finite number, a boolean or nothing (`null` and `undefined` return empty text), up to 32 KiB. Objects and Promises are errors: the action does not wait for timers or asynchronous callbacks. Input is limited to 16 KiB. One incoming handler request runs at a time, without event throttling or a waiting queue. The usual 30-second script guard applies, with a 35-second request deadline. Cancellation/deadlines prevent queued work from starting; work already started may still finish and keeps the slot until it exits. Exceptions are reported without copying their text into the intent or log.

`ScriptEvents.clear()`, disabling/deleting a script, reloading its code or revoking local approval makes its handlers unavailable. Pending callers receive an error. Normal script reload re-registers handlers and preserves approval only when the saved code is unchanged. The app caches handler choices for cold launch and revalidates the enabled script, source approval and live registration before execution. Input/output is not automatically logged.

Mac Shortcuts also includes **Set Light / Unit**, **Press Button / Macro** and **Get Object Status**. All run in the background, even while the Mac is locked. Light/button results mean the controller request was accepted, not physical confirmation. Object status supports zones, units, thermostats/sensors, flags and outputs, and refuses offline or stale readings. Choose objects by name, controller and number; saved actions retain their identity when display names change. Hidden objects are excluded. **Arm Security** (no code) and **Disarm Security** (code checked by the controller) are described under **Siri and Shortcuts**; locally approved handlers and macros may also perform security actions.

Incoming intents run on the Mac. An iPhone shortcut needs Helper's authenticated HTTP API (for example over LAN or Tailscale) rather than the Mac's App Intent.

`runUserScript(filename, args, callback)`

- Runs a user-installed executable asynchronously through macOS `NSUserUnixTask`. This is useful for a Python integration such as TinyTuya without an HTTP listener. It does not install Python, packages, or scripts for you.
- **Local setup:** In **Scripting**, click the terminal icon (**User-Installed Scripts**), review the warning, and enable access for this controller. Use **Open Application Scripts Folder** and place your trusted executable there yourself. This is not the JavaScript Libraries folder. Helper cannot write the executable for you. For Python, use an executable script with a shebang pointing to your Python virtual environment's absolute

interpreter path, with all required packages installed there. A wrapper may invoke an existing Python program elsewhere. Do not rely on your Terminal's PATH or activated environment. Make the installed file executable, for example `chmod +x` followed by its quoted full path.

- **Security:** Disabled by default. User scripts execute outside Helper's sandbox with the Mac user's permissions, subject to macOS privacy controls. The grant trusts every automation script on this controller, including code saved or enabled through a privileged API/MCP script-management credential. Helper restricts which installed executable it launches, not what that executable can do. Review executable code and argument handling; never install a general-purpose command evaluator. MCP cannot install executables or enable this grant. Never expose Helper or its credentials directly to the Internet. Do not pass secrets as command-line arguments; keep device credentials in a protected local store that the executable reads itself.

Parameters and result:

- `filename`: a plain filename, up to 128 UTF-8 bytes, using ASCII letters, digits, hyphens, underscores and dots; it cannot begin with a dot. Subdirectories, absolute paths and symbolic links are rejected. The file must be a regular executable in Application Scripts.
- `args`: an array of at most 32 strings, each at most 8 KiB and collectively at most 16 KiB of UTF-8, without NUL characters. Helper passes arguments literally, with no shell expansion. Pass JSON with `JSON.stringify(...)` as one argument when useful.
- `callback`: a required function. Returns a request ID when accepted, or `null` on rejection (access disabled, another task running, invalid arguments, or callback capacity exhausted). A rejected request logs a reason and does not call the callback. Acceptance does not prove the file can launch or that a device was controlled.
- The callback receives one terminal result: `{id, kind, state, success, stdout, stderr, outputTruncated, error?}`. `state` is `finished`, `failed`, `timedOut`, or `accessRevoked`. `success` is true only for a normal successful process exit. No exit-code number is exposed; the script must exit unsuccessfully when its operation fails. A successful exit is not independent confirmation of physical device state.
- **Limits and lifecycle:** One external process at a time across all controllers, with no waiting queue. The callback timeout is 60 seconds. A timeout or permission revocation does **not** terminate the process; new executions remain blocked until macOS reports that it exited. Do not automatically retry an uncertain physical command. A stuck process must be resolved outside Helper, for example in Activity Monitor. Restarting Helper does not guarantee termination of that process. Executables should perform bounded work and exit, not launch detached background services.
- Helper continuously drains output, retaining at most 32 KiB each of stdout and stderr; `outputTruncated` indicates discarded output. Output is decoded as UTF-8. Arguments and captured output are not automatically logged; take care before logging them yourself. Disabling, deleting or rerunning the JavaScript automation discards its old callback, but does not kill the external process. Revoking the controller's local grant blocks future calls and reports `accessRevoked` to an active callback.
- **Operational boundary:** This is a trusted-program launcher, not a process supervisor or security sandbox. The timeout bounds how long Helper waits to report an outcome, not the executable's CPU, memory, disk, network use, or child processes. A timed-out run receives no second callback on eventual exit. Give

device/network operations their own finite timeouts, shorter than 60 seconds, and keep external scripts short-lived. Do not parse truncated stdout as complete JSON. Invalid UTF-8 is replaced when converting output to text.

First-run setup check (no devices controlled)

- A ready-to-use example is included in **Scripting > Templates > Utilities > User Script: Setup Check**. Choose **Create Script**; this example starts disabled. It includes the executable's two-line contents and installation instructions in comments. Review the source and use **Run** once after setup. Do not enable or schedule the probe. Other templates retain their existing defaults.

1. Open **Scripting > User-Installed Scripts > Open Application Scripts Folder**. In that folder, create a plain-text file named `helper-probe.sh` containing the shell example below. It needs Unix line endings and the shebang on its first line.
2. In Terminal, type `chmod +x` and drag that file from Finder onto the command line, then press Return. Review the file before allowing execution. The Application Scripts folder is shared by controllers on this Helper installation; the permission is per controller.
3. Locally enable external scripts for the intended controller. Create a new JavaScript script with **Script Enabled** unchecked, paste the JavaScript setup check below, and click **Run** once. Do not enable or schedule the probe.
4. Expect `finished`, `success=true`, and the supplied text in stdout. This verifies installation and the callback only; it does not validate a Python environment or a device integration.

```
#!/bin/sh
printf '%s\n' "$1"
```

```
var probeID = MyroSystem.runUserScript("helper-probe.sh", ["Helper setup OK"],
    function (result) {
        MyroSystem.log("Probe: " + result.state + "; success=" + result.success);
        // Safe here: the probe prints only the supplied test phrase.
        if (result.success) MyroSystem.log(result.stdout);
        else MyroSystem.log(result.error || "Probe failed");
    });
if (!probeID) MyroSystem.log("Probe rejected; check the script log");
```

Python and device integrations

- Use a dedicated virtual environment with the integration's dependencies installed. If the interpreter path contains spaces, use a reviewed shell wrapper that calls its quoted absolute path, followed by the quoted absolute Python script path and `"$@"`. Do not use `eval`, `sh -c`, or concatenate incoming arguments into command text. Helper passes arguments literally, but an unsafe wrapper can undo that protection.
- Test the Python program manually using the same interpreter, arguments, and Mac account. Use absolute paths for credentials and other files; do not depend on the working directory or interactive shell configuration. A TinyTuya program must validate allowed device names and commands, read its local key from protected storage, set finite network timeouts, and return a nonzero exit on failure.
- Start with a disabled automation, verify one device action and the actual resulting state, then attach the call to the intended event. Filter repeated status reports so one switch action does not launch repeated

programs. Saving an enabled automation or reloading scripts executes its top-level body again. Never put an unintended device action at top level.

Complete example: Lutron to TinyTuya

- **Included files:** Open **Scripting > User-Installed Scripts > Open Examples Folder** for `README.md`, `tuya-light.py`, `tuya-light.sh` and `tuya-devices.example.json`. The README contains complete installation commands and a deliberate manual bulb test. Copy the files to the locations it specifies; never edit the app bundle. No background Python listener is needed. The Mac and Helper still need to be running.
- **Included automation:** Create **Templates > Utilities > Lutron to TinyTuya: Color LED**. This template starts disabled and contains the complete JavaScript event handler. Change `lutronName` to a light name unique across all connected Lutron bridges, and set the bulb alias, color and brightness. Test the Python program first, grant external-script access locally, then enable and save the automation.
- **Trigger behavior:** Works with Caseta/LEAP and RA2 level events. The first valid report establishes a baseline without controlling the bulb. Observe OFF then ON to set the bulb's color and brightness. Repeated ON and dimming reports are ignored. Scenes and app commands count too; this is not a physical-button event. OFF leaves the bulb unchanged. Rerunning resets the baseline. Busy or failed actions are not queued or automatically retried.
- **Credentials:** The Python program reads device IDs, reserved LAN IPs, protocol versions and local keys from `~/Library/Application Support/MyroHomeHelper/UserScripts/tuya-devices.json`, an owner-only (`chmod 600`) local file. Use a private (`chmod 700`) parent folder. Keys never belong in the Helper script, command arguments, AI chat or shared logs. This is protected plaintext, not Keychain storage. The script accepts only configured aliases and a limited color/brightness request.
- **Compatibility:** The example pins TinyTuya 1.20.0 and targets standard Type A/B Wi-Fi RGB bulbs with protocol 3.1, 3.3, 3.4 or 3.5. It is not a universal Tuya, Zigbee or gateway adapter. Named colors are red, green, blue, yellow, cyan, magenta and RGB white; brightness is an integer 1-100. RGB white does not select the bulb's white-temperature channel.
- **Confirmation and limits:** One absolute color assignment is followed by a fresh device-status check. Failure or missing confirmation exits nonzero; it does not prove the bulb stayed unchanged. TinyTuya can internally retry transport operations or fall back to individual data-point writes with the same intended setting. The example sets finite socket timeouts and a 20-second Python process alarm, but this is not general process supervision. Helper's own timeout does not kill the program. Tests cover real TinyTuya encoding with fake transport; verify compatibility and actual light output on your hardware.

The template passes this non-secret request to the installed launcher:

```
// The included template calls this only after an observed OFF-to-ON event.
MyroSystem.runUserScript("tuya-light.sh", [
    JSON.stringify({device: "Office LED", color: "blue", brightness: 60})
], function (result) {
    // The full template validates result.success and the JSON confirmation.
    // It never dumps stdout/stderr or retries a failed action automatically.
    MyroSystem.log("LED task: " + result.state);
});
```

See the bundled README and TinyTuya project for setup and device-compatibility details. After setup, normal bulb commands stay on the LAN; installing Python dependencies and obtaining the device's local key are separate steps.

User-script troubleshooting

Result or symptom	What to check
`null` with no callback	Read the script log: check the local grant, a running task, filename/argument limits, and the required function callback. Disabled access can show a local alert offering **Open User-Installed Scripts** or **Not Now**, once per controller per app session when no other dialog is active. Opening settings does not grant access or retry the command. There is no pending queue.
`failed` before output	Use the folder opened by Helper, a regular non-symlink file, executable permission, a valid shebang and an installed absolute interpreter. macOS privacy restrictions still apply; do not disable system security to make a script work.
`failed` with stderr	Inspect stderr privately for program, dependency, credential, or device errors. Do not paste secrets into AI chat or logs.
`timedOut`, subsequent calls rejected	The program may still be running and may already have acted. Inspect its process and the physical device state before any further action. Resolve the stuck process locally. Restarting Helper is not a reliable process cleanup method.
`accessRevoked`	Access was turned off locally. New calls are blocked; an already launched program is not terminated.
No callback after disabling or editing the automation	Old callbacks are intentionally discarded on disable, delete or rerun. The external program can continue until it exits.
`finished` but the device did not change	Success means the program exited successfully, not independent device confirmation. Make the program check device responses and fail explicitly when appropriate.

`log(message)`

Write to execution log and main Log window.

```
MyroSystem.log("Script started");
MyroSystem.log("Temperature: " + temp + "°F");
MyroSystem.log("Device state: " + state);
```

Parameters:

- `message` (string): Message to log

Use cases:

- Debug script execution
- Track automation decisions

- Record important events

`speak(text, options)`

- Queues Mac speech and returns `true` only when admitted. This is not a start/completion acknowledgement. Uses the same bounded, script-owned queue as `speakQueued`; rerun/disable/delete requests cancellation.
- `options` is optional: `{voice: "Ava"}` (or just the voice as a string) speaks this one call in a particular voice. See **Voices** below. Omit it to use the Mac's System voice. Helper speaks with the Mac's System voice (System Settings ▸ Accessibility ▸ Spoken Content ▸ System voice); choose a Siri, Premium or Enhanced voice there for a more natural voice.

`speakQueued(text, callback, options)`

Returns: Request ID string when admitted, otherwise `null`. Callback may be omitted or null. `options` is the same optional `{voice}` object as `speak()`; pass `null` for the callback to give options without one. Helper speaks with the Mac's System voice (System Settings ▸ Accessibility ▸ Spoken Content ▸ System voice); choose a Siri, Premium or Enhanced voice there for a more natural voice. An invalid callback, empty/NUL-containing text, text exceeding 4096 UTF-8 bytes, full queue, or unavailable/recovering native speech rejects admission without lifecycle callbacks.

Callback Function Signature:

```
function(event) { /* event = {id, state, error?} */ }
```

- Outcomes: `queued` means admitted; `started` comes from the native synthesizer delegate; `finished` only from its native finish delegate; `cancelled` means removed before playback or native cancellation acknowledged; `failed` includes queue expiry, native failure, and recovery timeout. Every admitted request has at most one terminal outcome; callbacks stop when the owning run is torn down.
- Delivery is asynchronous on the originating engine's serialized JavaScript queue with the caller's script identity and a guarded code budget. Callback functions remain rooted in a JS global until terminal delivery or teardown. They can register events/timers for their own script.
- Limits: 32 admitted requests across script engines, 8 per script run; at most 8 undelivered requests/outcomes per run and 128 per engine. No estimated-duration completion timer. Active requests have a 120-second recovery deadline, queued requests expire after 240 seconds, and cancellation waits at most 5 seconds for acknowledgement before failing.
- A timeout reports unknown completion and requests a native stop. The queue then allows 5 seconds to recover; waiting requests fail if native acknowledgement is still missing. New admission is refused during recovery. A late native terminal event restores availability without generating a second completion. No replacement synthesizers or uncertain overlapping playback are started.
- Mac speech is independent of existing HTTP speech, sounds, and Sonos queues. This API cannot cancel those services or another script's speech. Native completion is not proof that a speaker/amplifier was audible.

`cancelSpeech(id)`

Returns: `true` if this run owned a queued/active request and cancellation was requested; otherwise `false`. A finish racing cancellation remains `finished`. Never cancels another caller's request.

`cancelAllSpeech()`

Returns: Number of this script run's queued/active requests for which cancellation was requested. Rerun, disable, delete, and engine teardown perform the same ownership-scoped cancellation automatically and discard old callback roots; no expired callback is invoked in the next run.

Mac Speech Example

Use text-to-speech to speak a message. Helper speaks with the Mac's System voice (System Settings ▸ Accessibility ▸ Spoken Content ▸ System voice); choose a Siri, Premium or Enhanced voice there for a more natural voice.

```
MyroSystem.speak("Welcome home");
MyroSystem.speak("Temperature is " + temp + " degrees");
MyroSystem.speak("Front door is open", {voice: "Ava"});
```

Parameters:

- `text` (string): Text to speak
- `options` (object, optional): `{voice: "identifier or name"}`

Returns: `true` when admitted to the speech queue, `false` when rejected. This does not acknowledge playback.

Voices

Helper speaks with the Mac's System voice (System Settings ▸ Accessibility ▸ Spoken Content ▸ System voice); choose a Siri, Premium or Enhanced voice there for a more natural voice. Everything Helper says uses it — `MyroSystem.speak`, `speakQueued`, the HTTP `/helper/speak` route, MCP's `speak` tool and Sonos announcements — and a change there takes effect on the next announcement, no relaunch needed. Download more voices with System Voice ▸ Manage Voices. A Siri voice works here because Helper speaks through the Mac's own speech engine whenever no voice is named; a voice named for one call (below) is spoken by AVSpeechSynthesizer, which cannot use Siri voices. With a Siri System voice, Helper keeps the voice loaded in the background. macOS can take up to a minute to load a Siri voice for an app, so speech right after launch waits for it (up to 60 s, logged as `Speech: waiting for the System voice to load…`) and is then spoken in Siri; the Log window says when the System voice is ready. If the System voice cannot speak, Helper falls back to the best installed voice for the Mac's language: Premium, then Enhanced. Helper names the System voice in the Log window at launch (`Speech: using System voice (Siri, com.apple.siri.natural.Nora)`) and again whenever it changes.

Helper says Myro as "Mai-row" (MY-row).

A script can pick a voice for one call by identifier (`"com.apple.voice.premium.en-US.Ava"`) or by name (`"Ava"`). When several voices share a name, the one for the Mac's language and the best quality wins. A voice that is not installed on this Mac is not an error: the call speaks in the System voice and the Log window

says so once. When macOS replaces a voice with a larger download of the same voice (a Compact voice becoming Enhanced, say), an identifier naming the old size still finds it. If a voice produces no audio at all, Helper speaks the words with the Mac's speech synthesizer instead of staying silent, and logs that too.

`voices()`

Returns: An object describing the voices on this Mac:

```
{
    language: "en-US",                 // the Mac's speech language
    selected: "automatic",             // Helper follows the System voice
    automatic: {identifier, name, language, quality, category, group},   // the fallback voice if
        the System voice cannot speak (null: none)
    current:   {identifier, name, language, quality, category, group},   // same as automatic unless
        a voice is named
    voices: [ {identifier, name, language, quality, category, group}, ... ]  // every installed
        voice, in menu order
}
```

`quality` is `"siri"`, `"premium"`, `"enhanced"` or `"standard"`, as macOS reports it. `category` is the same, except that novelty voices are `"novelty"` (their `quality` stays `"standard"`). `group` is `"main"` for the Mac's language (novelty excluded) and `"more"` for everything else. `voices` is ordered: `main` voices best first, then each other language, then the Mac's language's novelty voices (Bubbles, Zarvox and the other sound-effect voices, and the character voices Eddy, Flo, Grandma, Grandpa, Reed, Rocko, Sandy and Shelley).

```
var premium = MyroSystem.voices().voices.filter(function (v) {
    return v.category === "premium" && v.language.indexOf("en") === 0;
});
if (premium.length) MyroSystem.speak("Using " + premium[0].name, {voice: premium[0].identifier});
```

`beep()`

Play system beep sound.

```
MyroSystem.beep();  // Simple alert beep
```

Returns: `true`

`alert(message)`

Display a macOS alert dialog.

```
MyroSystem.alert("Temperature too high!");
MyroSystem.alert("System armed");
MyroSystem.alert("Zone 1 has been tripped!");
```

Parameters:

- `message` (string): Alert message

Returns: `true`

Note: Alerts are modal and will block script execution until dismissed.

`playSound(filename)`

Play a system sound file.

```
MyroSystem.playSound("Glass.aiff");
MyroSystem.playSound("Hero.aiff");
MyroSystem.playSound("Ping.aiff");
```

Parameters:

- `filename` (string): Name of sound file (e.g., "Glass.aiff")

Supported locations:

- `/System/Library/Sounds/`
- `~/Library/Sounds/`
- App bundle resources

Returns: `true` if sound found and playing, `false` otherwise

`stopSound(filename)`

Stop a sound started by this script on this controller.

```
MyroSystem.stopSound("Glass.aiff");
```

Parameters:

- `filename` (string): Name of sound file

Returns: `true` if an active sound was stopped, `false` if none was playing for this script or stopping failed.

- Sounds are isolated by script/controller. Stopping `Glass.aiff` does not stop another script's copy, Mac speech, Sonos, or another application's audio.

- Replaying the same filename from the same script replaces its previous playback. Helper retains at most 32 active script sounds; additional playback fails until a slot is free.

`setVar(key, value)`

Store a legacy in-memory variable in the controller's scripting engine.

```
MyroSystem.setVar("lastTemp", 72);
MyroSystem.setVar("userName", "John");
MyroSystem.setVar("isArmed", true);
```

Parameters:

- `key` (string): Variable name
- `value` (string, number, or boolean): Value to store

Returns: `true` if successful, `false` if invalid type

Note: These variables are not durable and can be shared by scripts in the same engine. Use `setPersistentVar` for script-local state that must survive a restart. Panel flags remain an alternative when the panel itself needs the value.

`setPersistentVar(key, value)`

Parameters:

- `key` (string): 1-128 UTF-8 bytes; not a file path.
- `value`: A JSON-serializable value: string, number, boolean, null, array, or object. Standard `JSON.stringify` conversion rules apply: functions/prototypes are not preserved, dates become strings, and non-finite numbers become null. Cyclic values and top-level undefined are rejected.

Returns: `true` after an atomic disk write; `false` on invalid input, quota, or file errors. Errors are logged without logging the saved value.

- Storage is scoped to the controller GUID and stable script ID. Renaming/re-running/re-enabling the same script retains its values; another script or controller has a separate namespace.
- Limit: 64 KiB serialized JSON and 256 keys per script; 4 MiB of JSON per controller. Use this for small state, not logs, histories, media, or frequent sensor samples.
- Files are stored under Helper's Application Support `MyroHomeHelper/ScriptState` directory. Writes are atomic; unreadable/corrupt existing files are preserved and reported, not silently replaced.
- This is local application data, not encrypted secret storage or an isolation boundary against a privileged script. Do not store alarm codes, API keys, or passwords here.
- Values are not included in script export or MCP script source. Copying a script to a new ID starts a new namespace. Deleting a script leaves its stored file for recovery; remove unwanted keys before deletion or remove its file while Helper is stopped.

```
var count = MyroSystem.getPersistentVar("count");
if (typeof count !== "number") count = 0;
if (!MyroSystem.setPersistentVar("count", count + 1)) {
    MyroSystem.log("Counter was not saved");
}
```

`getPersistentVar(key)`

Parameters:

- `key` (string): The key saved by this script/controller.

Returns: The saved JSON value, or `undefined` if absent or a read fails (errors are logged). A stored `null` remains null. Mutating a returned object does not persist changes; call `setPersistentVar` again.

`removePersistentVar(key)`

Parameters:

- `key` (string): The saved key to remove from this script/controller only.

Returns: `true` after removal, including an already absent key; `false` for invalid input or a file error. JSON null is a value, not a deletion.

`getVar(key)`

Retrieve a stored variable.

```
const lastTemp = MyroSystem.getVar("lastTemp");
const userName = MyroSystem.getVar("userName");
const isArmed = MyroSystem.getVar("isArmed");

if (lastTemp === null) {
    MyroSystem.log("Variable not found");
}
```

Parameters:

- `key` (string): Variable name

Returns: Stored value, or `null` if not found

`isControllerConnected()`

Whether the Helper's controller session to this controller is up right now.

```
if (!MyroSystem.isControllerConnected()) {
    MyroSystem.log("Controller offline");
}
```

Returns: `true` if connected, `false` otherwise. On a Savant or Control4 controller this is Helper's native connection to that system.

Note: This is the controller link. `MyroNetwork.isConnected()` is about this Mac's own network. Read together they say where a problem is: controller down while the Mac is fine points at the controller, its cable or its port; both down points at the network.

`getEthernetQuality()`

The Helper's Ethernet Quality facts for this controller, the same ones the Ethernet Quality graph and CSV are built from.

```
var q = MyroSystem.getEthernetQuality();
MyroSystem.log("Connected: " + q.connected
    + ", CRC errors since launch: " + q.crcErrors);
```

Returns: `null` on NESS / ELK, Savant and Control4 controllers (these signals are Omni-Link specific); otherwise an object with these properties:

- `connected` (boolean): The controller session is up
- `secondsSinceLastMessage` (number): Seconds since the controller last sent anything, or `-1` if nothing has been received since launch
- `crcErrors`, `linkDrops`, `halfOpen`, `connectionFailures`, `ackTimeouts` (number): Counts since the Helper launched. An ACK timeout records a missed command confirmation; protocol details are in the controller guide. It is a missed confirmation, not proof that the command failed, the controller disconnected, or the panel's Ethernet hardware is defective.
- `macNetworkKnown` (boolean): The OS has reported this Mac's network state at least once
- `macNetworkUp` (boolean): This Mac has a working network path
- `physicalInterfaces` (number): Live Wi-Fi plus wired Ethernet interfaces on this Mac. `2` means both are up, which causes periodic drops when they share a subnet
- `othersConfigured` (boolean): At least one Lutron bridge is set up, so there is something to compare the controller against
- `othersHealthy` (boolean): At least one of those bridges is connected

Note: These are facts, not a verdict. Counts are cumulative since launch, so store a value and compare against it to detect new errors. A single ACK timeout is normally treated as a transient observation. The Ethernet Quality graph uses its longer CSV history to label recurrence as Low, Medium, or High concern; that history-based assessment is intentionally not synthesized inside a script snapshot. Compare the controller connection with `MyroNetwork.isConnected()` before deciding whether to alert.

MyroTimeline - Timeline Notes, Moments and Rules

Write to and read from the controller's Timeline: the **Timeline** page in browser control, the Timeline in Myro:Home Pro and the Apple TV's Today row. `MyroTimeline` exists on every controller, because HAI / Leviton, NESS / ELK, Savant and Control4 all have a Timeline. Writes are handed to Helper's Timeline queue and return at once; they never wait on the main thread. Reads take the history's own lock.

A script note never notifies anyone, never takes a picture and never sends a command. It is a line in the Timeline and nothing more.

Object ids are the favorites ids used across Helper: `"Unit:12"`, `"Zone:7"`, `"Thermostat:1"`, `"Area:1"`, `"Output:3"`, and `"Activity:<room>:<id>"` for a Savant or Control4 activity. Where a call takes an object, an object's name is accepted as well and is resolved to its id when it matches exactly one object.

Event kinds: `lights`, `door`, `window`, `motion`, `security`, `climate`, `lock`, `audio`, `media`, `scene`, `power`, `system`, `doorbell`, plus `note` (the default for `note()`) and `moment` (written by `moment()`).

Limits: a title is at most 80 characters and `detail` at most 200. Each script may write 120 notes and moments per hour; over that, calls return `null` and a warning goes to the script log. Helper keeps 1,000 script events in the 3-day store and drops the oldest script events first, separately from the 6,000-event cap on the panel's own events.

`MyroTimeline.note(title, options)`

Write an event into the 3-day history with `source: "script"`.

```
var id = MyroTimeline.note("Pool pump started", {
    kind: "power",
    object: "Pool Pump",
    room: "Backyard",
    importance: 0,
    detail: "Started by the pool schedule script"
});
```

Parameters:

- `title` (string): What happened, at most 80 characters
- `options` (object, optional):
  - `kind` (string): One of the event kinds above; default `"note"`
  - `object` (string): A favorites id such as `"Unit:12"`, `"Zone:7"` or `"Activity:<room>:<id>"`, or an object's name (resolved when it matches one object)
  - `room` (string): A room name
  - `at` (string or `Date`): When it happened, as ISO 8601 or a `Date`; default now. It must be within the last 72 hours and not in the future
  - `importance` (number): `0` routine, `1` notable, `2` headline; default `1`
  - `detail` (string): Up to 200 characters shown with the event

Returns: The event id (string), or `null` when the note was refused: a missing or too long title, a bad `kind`, `at` or `importance`, an `object` that matches no object or more than one, or the hourly limit. A refusal also writes a `MyroSystem.log` warning.

Note: `GET /helper/web/history` lists `note` events only when `note` is named in its `kinds`, as it does for the other non-legacy kinds; the Timeline, `events()` and `moments()` always include them.

`MyroTimeline.moment(title, options)`

Write a *named moment anchor*: an event of kind `"moment"` with importance 2 that the Timeline turns into a moment with exactly this title. The title is never rewritten and never merged into another moment's name. By default the moment reaches 10 minutes either side of its anchor and gathers the events in that window.

```
var movie = MyroTimeline.moment("Movie night", {
    symbol: "film",
    tone: "info",
    objects: ["Unit:12", "Unit:14"]
});
```

Parameters:

- `title` (string): The moment's title, at most 80 characters
- `options` (object, optional):
  - `symbol` (string): An SF Symbol name, at most 40 characters; default `"sparkles"`
  - `tone` (string): `"info"`, `"good"`, `"warn"` or `"alert"`; default `"info"`
  - `objects` (array of strings): Object ids the moment covers
  - `start`, `end` (string or `Date`): ISO 8601 dates; default now and open. Both must be within the last 72 hours and not in the future
  - `detail` (string): Up to 200 characters

Returns: The event id (string), or `null` when refused. Keep the id to close the moment with `end(id)`.

`MyroTimeline.end(id)`

Close an open moment or span written by `moment()` or `note()`: its `end` becomes now.

```
MyroTimeline.end(movie);
```

Parameters:

- `id` (string): The id returned by `moment()` or `note()`

Returns: `true` when the event was found and closed, `false` otherwise (an unknown id, an event that is already closed, or one this script did not write).

`MyroTimeline.mute(selector)`

A rule: the automatic Timeline rules make no event for whatever the selector names. Lanes are unaffected, so a muted light still shows in the Lights lane and the playhead still rewinds it; only the event line disappears.

```
MyroTimeline.mute("Hallway Motion");   // one chatty zone, by name
MyroTimeline.mute("Unit:12");          // one object, by id
MyroTimeline.mute("Thermostat");       // every object of a type
MyroTimeline.mute("motion");           // every event of a kind
```

Parameters:

- `selector` (string): An object id (`"Unit:12"`), an object name (`"Kitchen Lts"`), an object type (`"Unit"` , `"Zone"` , `"Thermostat"` , `"Area"` , `"Output"` , `"Lock"` , `"AudioZone"` , `"Activity"` ...), or an event kind (`"lights"` , `"motion"` , `"door"` , `"climate"` , `"audio"` ...)

Returns: `true`

Note: Doorbell rings and alarms are never muted, whatever the selector says. See **Who owns a rule** below.

`MyroTimeline.unmute(selector)`

Remove a mute this script set earlier. It takes the same selector forms as `mute` and cannot lift a mute another script set.

```
MyroTimeline.unmute("Hallway Motion");
```

Returns: `true`

`MyroTimeline.rename(selector, template)`

A rule: automatic events for the selector use `template` as their title. `{name}` is the object's name, `{state}` the word the rule would have used (on, off, opened, closed, armed Away, unlocked ...) and `{room}` the room.

```
MyroTimeline.rename("Unit:12", "{name} lamp {state}");
MyroTimeline.rename("Zone:7", "Front door {state}");
```

Parameters:

- `selector` (string): The same forms as `mute`
- `template` (string): The new title, with `{name}` , `{state}` and `{room}` substituted; at most 80 characters

Returns: `true`

`MyroTimeline.mode(mode)`

Choose how much of the Timeline the automatic rules write for this controller.

```
MyroTimeline.mode("manual");   // from now on this script writes the Timeline
```

Parameters:

- `mode` (string): `"auto"` (the default): automatic events plus your notes and moments. `"manual"`: the automatic rules make **no** events for this controller; the lanes, offline gaps, arming spans and UniFi Protect detections still come from the ledger and the cameras, and only `note()` and `moment()` write events

Returns: The mode now in force (`"auto"` or `"manual"`) after every script's rules are combined, so a script asking for `"auto"` while another holds `"manual"` is told `"manual"`.

Note: Doorbell rings and alarms are the exception: neither `mode` nor `mute` ever silences them.

`MyroTimeline.events(options)`

Read events from the 3-day history, newest first.

```
var lights = MyroTimeline.events({hours: 2, kind: "lights", limit: 50});
lights.forEach(function (e) {
    MyroSystem.log((e.approximate ? "~" : "") + e.time + " " + e.title);
});
```

Parameters:

- `options` (object, optional):
  - `hours` (number): 1–72; default 8
  - `kind` (string or array): One kind, or an array of kinds
  - `object` (string): An object id
  - `limit` (number): At most 500; default 100

Returns: An array of `{id, time, kind, title, object, room, importance, source, approximate, detail, end}`. `time` and `end` are ISO 8601 strings (`end` is absent while a span is open). `source` is `"script"` for `MyroTimeline`, `"api"` for the HTTP route, `"hai"`, `"elk"`, `"savant"` or `"control4"` for a live controller change, `"panelLog"` for the panel's event log, `"helper"` for Helper's own notes and `"protect"` for a UniFi Protect camera. `approximate` is `true` for a light the panel changed quietly and Helper learned from a status poll rather than an announcement; the browser shows these with a "~" before the time.

`MyroTimeline.moments(hours)`

The moments of the window, as the Timeline page builds them, without pictures.

```
MyroTimeline.moments(8).forEach(function (m) {
    MyroSystem.log(m.start + " " + m.title + " (" + m.eventIDs.length + " events)");
});
```

Parameters:

- `hours` (number, optional): 1–24; default 8

Returns: An array of `{id, title, kind, start, end, tone, symbol, importance, objects, eventIDs}`, oldest first, where `eventIDs` are the ids of the events inside the moment (readable with `events()`).

Who owns a rule

`mute`, `rename` and `mode` belong to the script that set them. They apply from the call until that script is reloaded, disabled or deleted, when they are dropped. Scripts run again at launch, so a script that wants a rule to be permanent sets it at the top, where it is set again on every run. Two scripts may set rules; the stricter wins: `manual` beats `auto`, and anything muted by any script stays muted until that script lifts it or goes away. The rules are consulted by the automatic Timeline rules on Helper's Timeline queue through a lock-guarded table, never by calling back into JavaScript, so a slow or stopped script can never hold the Timeline up. There is no switch for any of this in Helper's windows: to stop a rule, disable or delete the script.

MyroTime - Date & Time

Get current date and time information.

`getCurrentTime()`

Get current time as a string.

```
const time = MyroTime.getCurrentTime();
// Returns: "2:30 PM"

MyroSystem.log("Current time: " + time);
```

Returns: String (formatted time)

`getCurrentDate()`

Get current date as a string.

```
const date = MyroTime.getCurrentDate();
// Returns: "1/7/26"

MyroSystem.log("Current date: " + date);
```

Returns: String (formatted date)

`isWeekday()`

Check if today is a weekday (Monday-Friday).

```
if (MyroTime.isWeekday()) {
    MyroSystem.log("It's a weekday");
    MyroAPI.runMacro("Morning Routine");
} else {
    MyroSystem.log("It's the weekend");
    MyroAPI.runMacro("Weekend Mode");
}
```

Returns: `true` if Monday-Friday, `false` if Saturday-Sunday

`getHour()`

Get current hour (0-23).

```
const hour = MyroTime.getHour();
// Returns: 14 (2:00 PM)

if (hour >= 7 && hour < 9 && MyroTime.isWeekday()) {
    MyroAPI.runMacro("Morning Routine");
} else if (hour >= 22 || hour < 6) {
    MyroSystem.log("Night time - quiet mode");
}
```

Returns: Number (0-23)

MyroNetwork - Network & HTTP Requests

Check network connectivity and make HTTP requests to local or remote APIs.

`isConnected()`

Check if network is connected.

```
if (MyroNetwork.isConnected()) {
    MyroSystem.log("Network is available");
} else {
    MyroSystem.log("WARNING: Network is offline");
    MyroNotifications.send("Network connection lost!");
}
```

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

`ping(host, callback)`

Ping a network host (basic connectivity check).

```
MyroNetwork.ping("https://www.google.com", function(success) {
    if (success) {
        MyroSystem.log("Google is reachable");
    } else {
        MyroSystem.log("Cannot reach Google");
        MyroNotifications.send("Internet connectivity issue detected");
    }
});

// Check local device
MyroNetwork.ping("192.168.1.1", function(success) {
    if (!success) {
        MyroSystem.log("Router is unreachable");
    }
});
```

Parameters:

- `host` (string): Hostname or IP address
- `callback` (function): Callback receives `(success)` boolean

`get(url, headers, callback)`

Perform an HTTP GET request.

```
// Simple GET request
MyroNetwork.get("http://192.168.1.50/api/status", null, function(status, body, error) {
    if (error) {
        MyroSystem.log("Request failed: " + error);
        return;
    }
    MyroSystem.log("Response: " + status);

    // Parse JSON response
    var data = JSON.parse(body);
    MyroSystem.log("Device count: " + data.length);
});

// GET with custom headers
var headers = { "Authorization": "Bearer YOUR_TOKEN" };
MyroNetwork.get("http://192.168.1.50/api/devices", headers, function(status, body, error) {
    if (error) {
        MyroSystem.log("Request failed: " + error);
        return;
    }
    MyroSystem.log("Authenticated response: " + status);
});
```

Parameters:

- `url` (string): The URL to request
- `headers` (object|null): Optional object with header key/value pairs, or `null` for no custom headers
- `callback` (function): Callback receives `(statusCode, responseBody, error)` — error is `null` on success

---

`post(url, body, contentType, headers, callback)`

Perform an HTTP POST request.

```
// POST JSON data to a local API
var payload = JSON.stringify({ action: "on", level: 100 });
MyroNetwork.post("http://192.168.1.50/api/command", payload, "application/json", null,
    function(status, body, error) {
        if (error) {
            MyroSystem.log("POST failed: " + error);
            return;
        }
        MyroSystem.log("POST response: " + status + " " + body);
    }
);

// POST form data
MyroNetwork.post("http://192.168.1.50/api/action", "key=value&key2=value2",
    "application/x-www-form-urlencoded", null, function(status, body, error) {
        MyroSystem.log("Form POST: " + status);
    }
);
```

Parameters:

- `url` (string): The URL to request
- `body` (string): Request body as a string (use `JSON.stringify()` for JSON payloads)
- `contentType` (string): Content-Type header (e.g., `"application/json"`)
- `headers` (object|null): Optional additional headers, or `null`
- `callback` (function): Callback receives `(statusCode, responseBody, error)`

`put(url, body, contentType, headers, callback)`

Perform an HTTP PUT request. Same signature as `post()`.

```
var update = JSON.stringify({ brightness: 75 });
MyroNetwork.put("http://192.168.1.50/api/device/1", update, "application/json", null,
    function(status, body, error) {
        if (error) {
            MyroSystem.log("PUT failed: " + error);
            return;
        }
        MyroSystem.log("Updated: " + status);
    }
);
```

Parameters: Same as `post()`

`httpDelete(url, headers, callback)`

Perform an HTTP DELETE request.

```
MyroNetwork.httpDelete("http://192.168.1.50/api/schedule/5", null, function(status, body, error) {
    if (error) {
        MyroSystem.log("DELETE failed: " + error);
        return;
    }
    MyroSystem.log("Deleted: " + status);
});
```

Parameters:

- `url` (string): The URL to request
- `headers` (object|null): Optional headers, or `null`
- `callback` (function): Callback receives `(statusCode, responseBody, error)`

`request(method, url, body, contentType, headers, callback)`

Perform a generic HTTP request with any method (GET, POST, PUT, DELETE, PATCH, etc.).

```
// PATCH request
var patch = JSON.stringify({ name: "Updated Name" });
MyroNetwork.request("PATCH", "http://192.168.1.50/api/device/1", patch, "application/json", null,
    function(status, body, error) {
        MyroSystem.log("PATCH response: " + status);
    }
);
```

Parameters:

- `method` (string): HTTP method (e.g., `"GET"`, `"POST"`, `"PATCH"`)
- `url` (string): The URL to request
- `body` (string): Request body (empty string `""` for no body)
- `contentType` (string): Content-Type header (empty string `""` for none)
- `headers` (object|null): Optional additional headers, or `null`
- `callback` (function): Callback receives `(statusCode, responseBody, error)`

HTTP Request Notes

- **Timeout:** Requests timeout after 15 seconds
- **Async:** All HTTP methods are asynchronous — use the callback to process the response

- **Local network:** Works with local IP addresses (e.g., `http://192.168.1.x/...`) for controlling devices like Lutron LEAP-compatible processor, Home Assistant, Hubitat, etc.
- **JSON:** Use `JSON.stringify()` to create request bodies and `JSON.parse()` to read JSON responses
- **Error handling:** When an error occurs, `statusCode` is `0`, `responseBody` is empty, and `error` contains the error message. On success, `error` is `null`.

Example: React to Controller Changes and Call an API

```
// When controller flag changes, send command to external system
ScriptEvents.onUnitChange("Porch Light", function(name, isOn, level) {
    var payload = JSON.stringify({ device: name, state: isOn ? "on" : "off", level: level });
    MyroNetwork.post("http://192.168.1.50/api/sync", payload, "application/json", null,
        function(status, body, error) {
            if (error) {
                MyroSystem.log("API sync failed: " + error);
            } else {
                MyroSystem.log("Synced " + name + " state to external API");
            }
        }
    );
});
```

MyroNotifications - Push Notifications

Send push notifications to registered iOS and Android devices.

> **Setup required:** In Helper, select the controller, enable the appropriate app under **Notifications > App**, and display the QR code under **Notifications > Configuration**. On each iOS or Android device, open the controller's settings in the Myro:Home app, scroll to **Push Notifications**, tap the scanner icon, and scan Helper's QR code. Repeat for every device that should receive alerts, then use **Send Test Push Notification** in Helper to confirm delivery.

⚠️ **Rate Limits:**

- **Per-script:** 30 seconds between notifications from same script
- **System-wide:** 10 notifications per minute (OneSignal limit)

`send(message)`

Send notification with default sound.

```
MyroNotifications.send("Front door opened");
MyroNotifications.send("Temperature: " + temp + "°F");
MyroNotifications.send("System armed");
```

Parameters:

- `message` (string): Notification message

Returns: `true` if sent, `false` if throttled or error

Note: If notification is throttled, a warning message is logged showing time remaining until next notification can be sent.

`sendWithSound(message, sound)`

Send notification with custom sound.

```
MyroNotifications.sendWithSound("Alarm triggered!", "Trouble.caf");
MyroNotifications.sendWithSound("Welcome home", "Info.caf");
MyroNotifications.sendWithSound("Front door opened", "Perimeter.caf");
```

Parameters:

- `message` (string): Notification message
- `sound` (string): Sound file name

Available sounds:

- `Info.caf` (default)
- `Trouble.caf` (alert)
- `Ping.caf` (notification)
- `Perimeter.caf` (security)
- `Message.caf` (message)
- `Arm.caf` (system armed)
- `Disarm.caf` (system disarmed)

Returns: `true` if sent, `false` if throttled or error

MyroVestaboard - Vestaboard Display

Send messages to Vestaboard mechanical flip displays. Boards must be configured in the Add-ons > Vestaboard tab.

`sendMessage(boardID, line1, line2, line3, line4, line5, line6, persist)`

Send a message to a Vestaboard by its numeric ID. Each line is centered automatically. Extra lines beyond the board's row count are ignored (Flagship = 6 rows of 22 characters, Note = 3 rows of 15 characters).

```
// Send a temporary message (restores previous content after 30 seconds)
MyroVestaboard.sendMessage(1, "HELLO", "FROM", "MYRO:HOME", "", "", "", false);

// Send a persistent message (stays on the board permanently)
MyroVestaboard.sendMessage(1, "WELCOME", "HOME", "", "", "", "", true);
```

Parameters:

- `boardID` (integer): Numeric board ID (shown in Add-ons > Vestaboard table)
- `line1` through `line6` (string): Text for each line
- `persist` (boolean): `false` = temporary message (restores previous after 30 seconds), `true` = message stays permanently

Returns: `true` if message was sent, `false` if board not found

Note: If the board is mechanically busy drawing a previous message, the new message will be dropped.

`sendMessage` places lines from the top of the board. For a short message that should sit in the middle, or text you do not want to break into rows yourself, use `sendText`.

`sendText(boardID, text, persist)`

Fit and center text for the board automatically. Words wrap to the board's width, `\n` forces a new line, and the block is centered vertically as well as horizontally, so the same call looks right on a Flagship (6 rows of 22) and a Note (3 rows of 15) without choosing rows yourself. Text beyond the board's rows is dropped and a warning is logged.

```
// One line, centred on the board whatever its size
MyroVestaboard.sendText(1, "Welcome home", false);

// Wraps by words; \n forces a break
MyroVestaboard.sendText(1, "Garage door has been open\nfor 20 minutes", false);
```

Parameters:

- `boardID` (integer): Numeric board ID
- `text` (string): The message; case does not matter, the board is uppercase
- `persist` (boolean): `false` = temporary (restores the previous message after 30 seconds), `true` = stays

Returns: `true` if sent, `false` if the board was not found

`sendCharacters(boardID, characters, persist)`

Send an exact grid of Vestaboard character codes, one array per row, for colour tiles and art. Short rows and missing rows are padded with blanks; extra cells and rows are ignored. Any code outside 0-71 rejects the whole message.

```
// A red bar across the top row of a Note, then a blank row, then a green bar
var red = [], green = [];
for (var i = 0; i < 15; i++) { red.push(63); green.push(66); }
MyroVestaboard.sendCharacters(2, [red, [], green], true);
```

Character codes: 0 blank, 1-26 A-Z, 27-35 digits 1-9, 36 digit 0, 37-60 punctuation, 62 degree sign, 63 red, 64 orange, 65 yellow, 66 green, 67 blue, 68 violet, 69 white, 70 black.

Parameters:

- `boardID` (integer): Numeric board ID
- `characters` (array): Array of rows, each an array of whole numbers 0-71
- `persist` (boolean): As for `sendMessage`

Returns: `true` if sent, `false` if the board was not found or the grid was malformed

---

`getBoards()`

List all configured Vestaboard displays.

```
var boards = MyroVestaboard.getBoards();
for (var i = 0; i < boards.length; i++) {
    MyroSystem.log("Board #" + boards[i].id + " - " + boards[i].name + " (" + boards[i].type + ")");
}
```

Returns: Array of objects, each containing:

- `id` (integer): Board ID
- `name` (string): Board name
- `ipAddress` (string): IP address or hostname
- `type` (string): "Flagship" or "Note"
- `enabled` (boolean): Whether the board is enabled

MyroLaMetric - LaMetric Time Display

Send notifications to LaMetric Time smart clocks and LED displays. Devices must be configured in the Add-ons > LaMetric tab.

`sendNotification(deviceID, text, icon, sound, priority, cycles)`

Send a notification to a LaMetric Time device by its numeric ID. Notifications are displayed instantly and auto-dismiss automatically.

```
// Send a basic notification
MyroLaMetric.sendNotification(1, "Hello from Myro:Home", "i73576", "", "critical", 1);

// Send with a sound
MyroLaMetric.sendNotification(1, "Alert!", "i73576", "notification", "critical", 2);

// Send info priority (may be queued depending on device mode)
MyroLaMetric.sendNotification(1, "Temperature: 72F", "i73576", "", "info", 1);
```

Parameters:

- `deviceID` (integer): Numeric device ID (shown in Add-ons > LaMetric table)
- `text` (string): Notification text to display
- `icon` (string): LaMetric icon ID (e.g., `"i73576"` for Myro:Home logo). Use `""` for no icon
- `sound` (string): Sound ID from notifications category (e.g., `"notification"`, `"notification2"`). Use `""` for silent
- `priority` (string): `"info"` (queued), `"warning"` (higher priority), or `"critical"` (interrupts immediately)
- `cycles` (integer): Number of times to display the notification

Returns: `true` if notification was sent, `false` if device not found

Note: LaMetric devices may reject notifications with `info` or `warning` priority depending on the device's current mode. Use `critical` priority to ensure the notification is always displayed. Errors from the device are reported in the script execution log.

`getDevices()`

List all configured LaMetric Time devices.

```
var devices = MyroLaMetric.getDevices();
for (var i = 0; i < devices.length; i++) {
    MyroSystem.log("Device #" + devices[i].id + " - " + devices[i].name);
}
```

Returns: Array of objects, each containing:

- `id` (integer): Device ID
- `name` (string): Device name
- `ipAddress` (string): IP address or hostname
- `enabled` (boolean): Whether the device is enabled

MyroLutron - Unified Lutron Control (LEAP + RA2)

Control Lutron devices across ANY supported Lutron system with one API. Routes automatically between LEAP (modern systems) and RA2 Telnet (older systems) based on the bridge ID.

Supported Systems:

- **LEAP:** Caseta Smart Hub, Caseta Smart Bridge Pro, RA2 Select, RadioRA 3, HomeWorks QSX
- **RA2 Telnet:** RadioRA 2 Main Repeater, HomeWorks QS Processor

Device Reference:

- For LEAP bridges, pass the LEAP href as a string (e.g., `"/zone/1"`)
- For RA2 bridges, pass the integration ID as a number or string (e.g., `5` or `"5"`)

The same script can control devices on any Lutron system — just use the correct bridge ID and the matching device reference format for that bridge's protocol.

`MyroLutron.getBridges()`

List all configured Lutron bridges across both protocols.

```
var bridges = MyroLutron.getBridges();
for (var i = 0; i < bridges.length; i++) {
    MyroSystem.log("Bridge #" + bridges[i].id + " - " + bridges[i].name +
        " (" + bridges[i].protocol + ") connected: " + bridges[i].connected);
}
```

Returns: Array of objects, each containing:

- `id` (integer): Bridge ID
- `name` (string): Bridge name
- `ipAddress` (string): Bridge IP address
- `connected` (boolean): Connection state
- `protocol` (string): `"LEAP"` or `"RA2"`

`MyroLutron.getDevices(bridgeID)`

List discovered devices on a bridge. The returned objects vary slightly by protocol.

```
var devices = MyroLutron.getDevices(1);
for (var i = 0; i < devices.length; i++) {
    if (devices[i].protocol === "LEAP") {
        MyroSystem.log(devices[i].href + " - " + devices[i].name);
    } else {
        MyroSystem.log(devices[i].integrationID + " - " + devices[i].name + " (area: " +
        devices[i].area + ")");
    }
}
```

`MyroLutron.setLevel(bridgeID, deviceRef, level)`

Set a dimmer/output level (0-100) on any bridge.

```
// LEAP bridge
MyroLutron.setLevel(1, "/zone/3", 75);

// RA2 bridge (integration ID as number)
MyroLutron.setLevel(2, 5, 75);
```

`MyroLutron.turnOn(bridgeID, deviceRef)` / **`turnOff(bridgeID, deviceRef)`**

Convenience methods — shortcut for `setLevel(..., 100)` / `setLevel(..., 0)`.

`MyroLutron.getLevel(bridgeID, deviceRef)`

Read the last-known level. Returns -1 if the device isn't known.

`MyroLutron.activateScene(bridgeID, sceneRef)` ***(LEAP only)***

Activate a LEAP scene. Not supported on RA2 — use `tapButton` on the scene keypad instead.

`MyroLutron.tapButton(bridgeID, integrationID, component)` ***(RA2 only)***

Simulate a Pico/keypad button tap. Component numbers come from the integration report.

- `tapButton` sends the press and release directly to the RA2 repeater. See the controller guide for panel-specific bridge examples.

```
// Tap the "On" button on a Pico remote (integration ID 6, component 2)
MyroLutron.tapButton(2, 6, 2);
```

`MyroLutron.findBridge(name)`

Look up a bridge ID by name. Returns -1 if not found.

`MyroLutron.findDevice(bridgeID, name)`

Look up a device reference by name. Returns the href (LEAP) or integration ID (RA2), or null.

Unified Event Handlers

Method Signature:

```
ScriptEvents.onLutronDeviceChange(deviceName, callback)     // LEAP + RA2
ScriptEvents.onLutronButtonPress(deviceName, callback)      // LEAP + RA2
ScriptEvents.onLutronLEAPDeviceChange(deviceName, callback) // LEAP only
ScriptEvents.onLutronLEAPButtonPress(deviceName, callback)  // LEAP only
ScriptEvents.onRA2DeviceChange(deviceName, callback)        // RA2 only
ScriptEvents.onRA2ButtonPress(deviceName, callback)         // RA2 only
```

Handlers registered with `onLutronDeviceChange` and `onLutronButtonPress` fire for events from BOTH LEAP and RA2 bridges, so one handler can react to changes anywhere in your Lutron setup. Pass `"*"` as the device name to match every device or remote.

Action casing. LEAP bridges report `"Press"` and `"Release"`; RA2 bridges report `"press"` and `"release"`. In a unified handler compare with `action.toLowerCase()` so it works on both.

Pico remotes with no assigned load. A Pico enrolled in the Caseta hub triggers scripts even when it does not control any Lutron light — add and name it in the Lutron app and leave it unassigned. A Pico paired *directly* to a dimmer and never enrolled in the hub is invisible over LEAP and cannot trigger anything.

```
ScriptEvents.onLutronDeviceChange("Kitchen", function(deviceName, level, deviceRef) {
    MyroSystem.log(deviceName + " -> " + level + "% (ref: " + deviceRef + ")");
});

ScriptEvents.onLutronButtonPress("Living Room Pico", function(deviceName, button, action, deviceRef)
        {
    // LEAP reports "Press"/"Release"; RA2 reports "press"/"release". Compare
    // case-insensitively so the same handler works on either bridge.
    if (action.toLowerCase() === "press" && button === 2) {
        MyroAPI.activateScene("Evening");
    }
});
```

RA2-only handlers are also available: `onRA2DeviceChange(deviceName, handler)` and `onRA2ButtonPress(deviceName, handler)`.

MyroLutronLEAP - Lutron LEAP Control

Control Lutron devices via the local LEAP protocol. Requires a paired Lutron LEAP-compatible processor configured in the Add-ons tab.

> **Tip:** For new scripts, prefer the unified `MyroLutron` API above — same calls work for both LEAP and RA2 bridges.

Supported Processors:

- Caseta Smart Hub (L-BDG2-WH)
- Caseta Smart Bridge Pro (L-BDGPRO2-WH)
- RA2 Select Main Repeater (RR-SEL-REP2-BL)
- RadioRA 3 All-in-One Processor (RR-PROC3)
- HomeWorks QSX Processor

Note: The standard (non-Smart Hub) Caseta bridge does not support the local LEAP protocol.

`getBridges()`

List all configured Lutron bridges.

```
var bridges = MyroLutronLEAP.getBridges();
for (var i = 0; i < bridges.length; i++) {
    MyroSystem.log("Bridge #" + bridges[i].id + " - " + bridges[i].name +
        " (" + bridges[i].ipAddress + ") connected: " + bridges[i].connected);
}
```

Returns: Array of objects, each containing:

- `id` (integer): Bridge ID
- `name` (string): Bridge name
- `ipAddress` (string): Bridge IP address
- `connected` (boolean): Whether the bridge is currently connected
- `paired` (boolean): Whether the bridge has been paired

`getDevices(bridgeID)`

List all discovered Lutron devices on a bridge.

```
var devices = MyroLutronLEAP.getDevices(1);
for (var i = 0; i < devices.length; i++) {
    MyroSystem.log(devices[i].href + " - " + devices[i].name +
        " (" + devices[i].type + ") level: " + devices[i].level + "%");
}
```

Parameters:

- `bridgeID` (integer): The bridge ID

Returns: Array of objects, each containing:

- `href` (string): LEAP device href (e.g., `/zone/1`) — use this for control commands
- `name` (string): Device name (e.g., "Kitchen")
- `type` (string): Device type (`zone`, `area`, `scene`)
- `area` (string): Associated area href
- `level` (integer): Current level (0-100)

`setLevel(bridgeID, deviceHref, level)`

Set a Lutron device to a specific level.

```
// Set kitchen dimmer to 75%
MyroLutronLEAP.setLevel(1, "/zone/3", 75);

// Turn on (100%)
MyroLutronLEAP.setLevel(1, "/zone/3", 100);

// Turn off (0%)
MyroLutronLEAP.setLevel(1, "/zone/3", 0);
```

Parameters:

- `bridgeID` (integer): The bridge ID
- `deviceHref` (string): The LEAP href of the device (from `getDevices()`)
- `level` (integer): Target level, 0-100

Returns: `true` if the command was sent, `false` if the bridge is not connected

---

`turnOn(bridgeID, deviceHref)` / `turnOff(bridgeID, deviceHref)`

Convenience methods to turn a device fully on (100) or off (0).

```
MyroLutronLEAP.turnOn(1, "/zone/3");   // Sets level to 100
MyroLutronLEAP.turnOff(1, "/zone/3");  // Sets level to 0
```

Parameters:

- `bridgeID` (integer): The bridge ID
- `deviceHref` (string): The LEAP href of the device

Returns: `true` if sent, `false` if not connected

`getLevel(bridgeID, deviceHref)`

Get the current cached level of a Lutron device.

```
var level = MyroLutronLEAP.getLevel(1, "/zone/3");
MyroSystem.log("Kitchen is at " + level + "%");
```

Parameters:

- `bridgeID` (integer): The bridge ID
- `deviceHref` (string): The LEAP href of the device

Returns: Integer level (0-100), or `-1` if device not found

`activateScene(bridgeID, sceneHref)`

Activate a Lutron scene (virtual button).

```
MyroLutronLEAP.activateScene(1, "/virtualbutton/1");
```

Parameters:

- `bridgeID` (integer): The bridge ID
- `sceneHref` (string): The LEAP href of the scene

Returns: `true` if sent, `false` if not connected

Lutron LEAP Event Handlers

React to real-time Lutron device changes. These fire immediately when someone uses a wall dimmer, Pico remote, or the Lutron app.

`ScriptEvents.onLutronLEAPDeviceChange(deviceName, handler)`

Register a handler for Lutron device level changes.

```
// React when a Lutron device changes
ScriptEvents.onLutronLEAPDeviceChange("Kitchen", function(deviceName, level, href) {
    MyroSystem.log("Lutron: " + deviceName + " changed to " + level + "%");

    // Sync back to controller
    MyroAPI.setLevel("Kitchen Light", level);
});
```

Handler parameters:

- `deviceName` (string): Name of the Lutron device
- `level` (integer): New level (0-100)
- `href` (string): LEAP href of the device

`ScriptEvents.onLutronLEAPButtonPress(deviceName, handler)`

Register a handler for Lutron button/Pico remote presses.

Register with the name shown in the Lutron app, or with the device's LEAP href as earlier versions required — both match. Pass `"*"` to match every remote, which is the practical way to discover what a button actually reports before writing the real handler.

```
ScriptEvents.onLutronLEAPButtonPress("Living Room Pico", function(deviceHref, buttonNumber, action,
        deviceName) {
    MyroSystem.log(deviceName + " button " + buttonNumber + " " + action);
});

// Learn what every remote reports
ScriptEvents.onLutronLEAPButtonPress("*", function(href, buttonNumber, action, deviceName) {
    MyroSystem.log(deviceName + " button " + buttonNumber + " " + action);
});
```

> **New in 3.1:** a fourth handler argument, `deviceName`, and registration by name or `"*"`. The first three arguments are unchanged, and registering with an href still works, so existing scripts keep running as they are.

Handler parameters:

- `deviceHref` (string): LEAP href of the button device
- `buttonNumber` (integer): Button number pressed
- `action` (string): Event type (e.g., "Press", "Release")
- `deviceName` (string): Name of the remote as shown in the Lutron app (added in 3.1)

Example: Bidirectional Controller-Lutron Bridge

```
// Bridge an controller unit to a Lutron dimmer with anti-feedback protection
var bridgeID = 1;
var lutronHref = "/zone/3";
var omniUnit = "Kitchen Light";
var lastCommandTime = 0;

// controller -> Lutron
ScriptEvents.onUnitChange(omniUnit, function(name, isOn, level) {
    lastCommandTime = Date.now();
    MyroLutronLEAP.setLevel(bridgeID, lutronHref, level);
    MyroSystem.log("controller -> Lutron: " + name + " level " + level + "%");
});

// Lutron -> controller (with 2-second echo suppression)
ScriptEvents.onLutronLEAPDeviceChange("Kitchen", function(deviceName, level, href) {
    if (Date.now() - lastCommandTime < 2000) return; // skip echo
    MyroAPI.setLevel(omniUnit, level);
    MyroSystem.log("Lutron -> controller: " + deviceName + " level " + level + "%");
});
```

MyroMQTT - MQTT Publishing

Publish messages to your MQTT broker(s) and check connection state. Brokers are configured under **Add-ons ▸ MQTT** — most installations have exactly one (Mosquitto, Home Assistant's broker add-on, EMQX, etc.). Broker passwords are stored in the macOS Keychain. To *receive* MQTT messages, use `ScriptEvents.onMQTTMessage` (see Event-Based Scripts).

The Helper also publishes its own availability to a retained topic — `online` while connected, and the broker publishes `offline` automatically if the Helper dies unexpectedly (MQTT Last Will). This works directly as a Home Assistant `availability_topic`. The exact topic is `myrohome/<Client ID>/status`, and it is written to the Log window each time the broker connects so you can copy it.

Each broker gets a **Client ID** generated when you add it, unique to that Mac (for example `helper-a1b2c3-1`). It does two jobs:

- **It identifies the connection.** MQTT requires a broker to disconnect any existing session when a new client connects with the same ID, so running Helper on two Macs against one broker needs two different Client IDs — otherwise they repeatedly evict each other and neither stays connected. Helper detects this and logs it explicitly.
- **It names the availability topic.** Because that topic is retained and last-writer-wins, a shared topic would let one Helper quitting mark another still-running Helper as offline.

The generated defaults already differ per Mac, so this works out of the box. Edit the Client ID if you want a friendlier availability topic — setting it to `helper-studio` gives you `myrohome/helper-studio/status`.

`publish(topic, payload)`

Publish a message to the default broker (the first connected one) with QoS 0, not retained.

Parameters:

- `topic` (string): Topic to publish to (no wildcards)
- `payload` (string): Message body. To send JSON, pass `JSON.stringify(obj)`.

Returns: `true` if handed to a connected broker, `false` if no broker is connected

Example:

```
MyroMQTT.publish("myrohome/alarm", "armed");
MyroMQTT.publish("myrohome/temps/living", JSON.stringify({ temp: 72.5, unit: "F" }));
```

`publishEx(topic, payload, qos, retain, brokerID)`

Publish with full control.

Parameters:

- `topic` (string): Topic to publish to
- `payload` (string): Message body
- `qos` (integer): 0 (at most once) or 1 (at least once)

- `retain` (boolean): `true` stores the message as the topic's retained value — new subscribers immediately receive it
- `brokerID` (integer): Target broker ID from Add-ons ▸ MQTT; `0` = default broker

Returns: `true` if handed to a connected broker, `false` otherwise

Example:

```
// Retained state topic Home Assistant can bind a sensor to
MyroMQTT.publishEx("myrohome/zones/front-door", "closed", 1, true, 0);
```

`isConnected()`

Returns: `true` when at least one configured broker is connected.

`getBrokers()`

Returns: Array of broker objects: `{brokerID, name, host, port, connected, state, lastError}`.

Example:

```
var brokers = MyroMQTT.getBrokers();
for (var i = 0; i < brokers.length; i++) {
    MyroSystem.log(brokers[i].name + ": " + brokers[i].state);
}
```

MyroSonos - Sonos Announcements

Speak text through your Sonos speakers, or play a sound file on them. Speakers are configured under **Add-ons ▸ Sonos**. Click **Discover** to search the network and tick the rooms you want; you can also add one by hand if you know its address. Either way the whole household layout is read from whatever you add, so a script can address any room by name without you listing them all.

Announcements **duck** rather than interrupt. The Helper uses the speaker's local audio-clip endpoint, which lowers whatever is playing, speaks over it, and brings the music back on its own — nothing is stopped and no queue position is lost. That endpoint requires an S2 speaker. On S1 systems and older players the Helper falls back to briefly taking the speaker over and restoring what was playing afterwards. Either way the announcement is heard; only the smoothness differs, and the Test button under Add-ons ▸ Sonos tells you which path a given speaker gets. Speech is rendered as 44.1 kHz 16-bit stereo audio, which is the uncompressed format Sonos documents support for, so older players accept it too.

Grouped rooms. Sonos plays a group through a single speaker, its *coordinator*; addressing a member of a group directly is silent on the UPnP fallback path. Use `speakToGroup()` when rooms may be grouped and you want everyone in the group to hear it.

Speech is cached. Every fixed phrase written in a script is rendered to audio the moment the script is saved, so the first announcement plays immediately instead of pausing to synthesise. Editing or deleting a script releases phrases it no longer says. Text a script builds at run time (`"Garage open for " + minutes + "`

`minutes"`) is cached by content and trimmed automatically, so nothing accumulates unattended. Changing the Mac's System voice re-renders the fixed phrases in the new voice.

Voice. Helper speaks with the Mac's System voice (System Settings ▸ Accessibility ▸ Spoken Content ▸ System voice); choose a Siri, Premium or Enhanced voice there for a more natural voice. (See **Voices** under MyroSystem.) Every speak function also takes an optional last argument, `voice` — an identifier or name from `MyroSystem.voices()` — for one announcement in a different voice. Existing calls without it are unchanged. A phrase spoken with an explicit voice is rendered the first time it is used and cached from then on.

`speak(text, room, volume, voice)`

Announce spoken text.

Parameters:

- `text` (string): What to say
- `room` (string): Room name. Pass `""` to announce on every configured speaker.
- `volume` (integer, optional): 0-100. Omit to use the speaker's **Announce Volume** from Add-ons ▸ Sonos, or — if that is blank — whatever the speaker is currently set to. Setting an Announce Volume matters: a speaker left turned down for background music delivers an alert nobody hears. A value outside 0-100 is ignored and logged.
- `voice` (string, optional): Voice identifier or name for this announcement. Omit to use the Mac's System voice. Pass `undefined` for `volume` to give a voice without a volume.

Returns: `true` once the announcement is queued. Announcing happens in the background so a handler is never blocked while speech renders and plays, which means the return value cannot report an unknown room — that is written to the Log window instead. Use `getRooms()` if a script needs to check a name first.

Example:

```
MyroSonos.speak("Front gate opened", "Kitchen");
MyroSonos.speak("Water leak detected in the basement", "", 70);   // everywhere, loud
```

`speakToGroup(text, room, volume, voice)`

Announce to the whole group that `room` belongs to, addressed to the group's coordinator. If the room is not grouped, this is just that room.

Parameters: same as `speak()`, except `room` is required.

Returns: `true` once the announcement is queued (see `speak()` above).

Example:

```
// Heard in every room currently grouped with the Kitchen
MyroSonos.speakToGroup("Dinner is ready", "Kitchen");
```

`speakEx(text, room, volume, chime, leadInSeconds, voice)`

Speak with an explicit paging lead-in, ignoring whatever the speaker is set to.

A Sonos wired into a whole-house or paging amplifier cannot simply start talking: the amplifier switches on when it detects audio, and the first syllable is lost while the relay engages. Turning on **Chime** for that speaker under Add-ons ▸ Sonos plays a short two-tone page followed by a pause before the words, which covers the normal case. Use `speakEx()` when one script needs to differ — an urgent alert that should chime everywhere, or an overnight reminder that should not.

The pause is silent. An amplifier that switches on when it detects audio holds for tens of seconds to minutes after the signal stops, so the chime is enough to bring it up and it is still up when the words begin — there is no need to fill the gap with anything.

Parameters:

- `text` (string): What to say
- `room` (string): Room name, or `""` for every configured speaker
- `volume` (integer, optional): 0-100
- `chime` (boolean **or** string): `true` plays the built-in two-tone page. Pass a **file name** instead — `"page-tone.wav"` — to use one of your own sounds as the preamble, which matters when a house already has a page tone its occupants recognise. The file is searched the same way `playSound()` searches, and falls back to the built-in tone (with a note in the Log window) if it is not found. `false` plays no chime.
- `leadInSeconds` (number): Seconds of lead-in before the words, 0-10. `2` suits most paging amplifiers.
- `voice` (string, optional): Voice identifier or name for this announcement, as for `speak()`.

Returns: `true` once the announcement is queued (see `speak()` above).

Example:

```
// Chime everywhere, whatever each speaker is normally set to
MyroSonos.speakEx("Fire alarm activated", "", 90, true, 2);

// No chime and no pause, even on the speaker that usually pages
MyroSonos.speakEx("Laundry is done", "Kitchen", 40, false, 0);

// The building's own page tone instead of the built-in one
MyroSonos.speakEx("Fire alarm activated", "", 90, "evacuation-tone.wav", 2);

// A different voice for one announcement
MyroSonos.speakEx("Turning off bathroom lights", "Paging", 80, true, 2, "Ava");
```

The same `chime` argument is accepted by `speakToGroupEx()` and `playSoundEx()`.

`speakToGroupEx(text, room, volume, chime, leadInSeconds, voice)`

`speakToGroup()` with an explicit paging lead-in, for the same reasons as `speakEx()`.

`playSoundEx(filename, room, volume, chime, leadInSeconds)`

`playSound()` with an explicit paging lead-in.

Sound files get a lead-in too — through a paging amplifier the first moment of an alarm tone is lost exactly as the first syllable of a sentence would be. `playSound()` already applies the speaker's Chime and Lead-in settings; this overrides them for one call.

`playSound(filename, room, volume)`

Play an audio file instead of speech — a chime, an alarm tone, a recorded message.

Three places are searched, in order: your own **Sounds folder** (Add-ons ▸ Sonos ▸ **Sounds Folder** opens it — inside the sandbox container it is not somewhere you would find by hand), then `~/Library/Sounds`, then the macOS system sounds. So a file you drop in shadows a system sound of the same name, and the built-in macOS sounds work with no setup at all.

Sonos plays MP3, M4A, WAV, AIFF, FLAC, and OGG. Anything else is refused with a message rather than handed to the speaker to ignore.

Parameters:

- `filename` (string): File name including its extension
- `room` (string): Room name, or `""` for every configured speaker
- `volume` (integer, optional): 0-100

Returns: `true` if the file was found and the sound was queued, `false` if it was not found or is not a format Sonos plays. A miss lists what *is* available in the Log window.

Example:

```
MyroSonos.playSound("doorbell.mp3", "Front Porch", 80);
MyroSonos.playSound("Submarine.aiff", "Kitchen");     // a macOS system sound
```

`getSounds()`

Returns: Array of `{name, location, bytes}` for every playable sound file found, so a script can check rather than guess.

Example:

```
var sounds = MyroSonos.getSounds();
for (var i = 0; i < sounds.length; i++) {
    MyroSystem.log(sounds[i].name + "  (" + sounds[i].location + ")");
}
```

`getRooms()`

Returns: Array of room-name strings discovered on the household — including rooms you never added by hand.

Example:

```
var rooms = MyroSonos.getRooms();
MyroSystem.log("Sonos rooms: " + rooms.join(", "));
```

`getGroups()`

Returns: Array of `{coordinator, rooms}` objects describing how rooms are grouped right now.

Example:

```
var groups = MyroSonos.getGroups();
for (var i = 0; i < groups.length; i++) {
    MyroSystem.log(groups[i].rooms.join(" + ") + " -> " + groups[i].coordinator);
}
```

`getSpeakers()`

Returns: Array of the speakers configured under Add-ons ▸ Sonos: `{speakerID, name, ipAddress, enabled}`.

`refresh()`

Re-read the household topology and return the room names. Grouping changes whenever someone uses the Sonos app, so call this first if your script's behaviour depends on the current grouping.

Returns: Array of room-name strings

Example:

```
// Announce to whichever group the Kitchen is in *right now*
MyroSonos.refresh();
MyroSonos.speakToGroup("Laundry is done", "Kitchen");
```

Sonos + MQTT

Because announcements are just a function call, anything that can trigger a script can speak. Pairing them with MQTT turns any sensor on your broker into a spoken alert without writing an integration:

```
ScriptEvents.clear();

ScriptEvents.onMQTTMessage("zigbee2mqtt/+/water_leak", function(topic, payload, json) {
    if (json && json.water_leak) {
        var sensor = topic.split("/")[1];
        MyroSonos.speak("Water leak detected at " + sensor, "", 80);
        MyroNotifications.send("Water leak: " + sensor);
    }
});

ScriptEvents.onMQTTMessage("frigate/events", function(topic, payload, json) {
    if (json && json.type === "new" && json.after.label === "person") {
        MyroSonos.speakToGroup("Someone is at the " + json.after.camera, "Kitchen");
    }
});
```

HTTP API from outside Helper

JavaScript APIs run inside Helper. External programs use authenticated HTTP routes described in the HTTP API guide. Read your controller guide for panel action parameters. MCP tools and HTTP route names are not JavaScript globals.

Event-Based Scripts

Event-based scripts automatically run when specific events occur in your home automation system. They run in the background and persist until disabled, deleted, or the app closes.

Auto-Registration at App Launch

When Myro:Home Helper starts, all enabled scripts with event handlers are automatically registered. This means:

- Event handlers persist across app restarts
- Scripts with event handlers are ready immediately after launch
- No need to manually run scripts to register handlers

Registering Event Handlers

Start an event script with `ScriptEvents.clear()` before registering callbacks. Keep callbacks short and use timers for delayed work. Enabled event scripts register automatically at Helper launch. Controller-event signatures appear in your controller guide; shared integration events are documented below.

```
ScriptEvents.clear();
ScriptEvents.onWeatherSync(function() {
    MyroSystem.log("Weather forecast updated");
});
```

Available Event Types

- Every handler that takes an object name also accepts `"*"` to match **every** object of that kind. The callback still receives the specific name, so one handler can `switch` on it instead of registering a handler per object. Pass the exact name when you want only one.

Weather Sync Updates

- RainAdvisory sums five-day forecast rain in millimeters, rounds to whole millimeters and caps the panel value at 255. A valid dry forecast writes zero. Failed or incomplete responses preserve the previous value and successful-sync timestamp and do not emit this event. Automatic refresh targets an hour after the last success and retries failures with backoff. A successful sync means receipt of a valid forecast and dispatch of its counter command, not hardware acknowledgement. Monitor weather data synchronization events.

Method Signature:

```
ScriptEvents.onWeatherSync(callback)
```

Parameters:

- `callback` (function): Handler function to execute when weather data is synced

Callback Function Signature:

```
function()
```

Callback Parameters: None

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

Example:

```
// Monitor weather sync updates
ScriptEvents.onWeatherSync(function() {
    MyroSystem.log("Weather data updated");

    // Weather data is now fresh - you can read it using MyroSensors
    // Note: Weather data is stored in the controller, not directly accessible via API
    // You would need to check controller-specific weather sensors if available
});
```

Energy Sync Updates

Monitor energy data synchronization events.

Method Signature:

```
ScriptEvents.onEnergySync(callback)
```

Parameters:

- `callback` (function): Handler function to execute when energy data is synced

Callback Function Signature:

```
function()
```

Callback Parameters: None

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

Example:

```
// Monitor energy sync updates
ScriptEvents.onEnergySync(function() {
    MyroSystem.log("Energy data updated");

    // Energy data is now fresh - you can read it using MyroSensors
    // Note: Energy data is stored in the controller, not directly accessible via API
    // You would need to check controller-specific energy sensors if available
});
```

Lutron LEAP Device Changes

Monitor Lutron device level changes in real-time (wall dimmers, Pico remotes, Lutron app).

Method Signature:

```
ScriptEvents.onLutronLEAPDeviceChange(deviceName, callback)
```

Parameters:

- `deviceName` (string): Lutron device name (as discovered by the bridge)
- `callback` (function): Handler function

Callback Function Signature:

```
function(deviceName, level, href)
```

Callback Parameters:

- `deviceName` (string): Lutron device name
- `level` (integer): New level (0-100)
- `href` (string): LEAP href of the device

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

Example:

```
// Sync Lutron kitchen dimmer back to controller
ScriptEvents.onLutronLEAPDeviceChange("Kitchen", function(deviceName, level, href) {
    MyroSystem.log(deviceName + " changed to " + level + "%");
    MyroAPI.setLevel("Kitchen Light", level);
});
```

Lutron LEAP Button Presses

Monitor Lutron Pico remote button presses.

Method Signature:

```
ScriptEvents.onLutronLEAPButtonPress(deviceName, callback)
```

Parameters:

- `deviceName` (string): Pico remote or button device name
- `callback` (function): Handler function

Callback Function Signature:

```
function(deviceHref, buttonNumber, action)
```

Callback Parameters:

- `deviceHref` (string): LEAP href of the button device
- `buttonNumber` (integer): Button number pressed
- `action` (string): Event type (e.g., "Press", "Release")

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

Example:

```
ScriptEvents.onLutronLEAPButtonPress("Living Room Pico", function(name, btn, action, href) {
    if (action === "Press" && btn === 2) {
        MyroSystem.log("Pico center button pressed");
        MyroAPI.runMacro("Movie Mode");
    }
});
```

MQTT Messages

Trigger scripts from MQTT messages published to your broker (configured under Add-ons ▸ MQTT). This connects Myro:Home to Zigbee2MQTT, Frigate, Home Assistant, Shelly devices, and anything else speaking MQTT.

Method Signature:

```
ScriptEvents.onMQTTMessage(topicFilter, callback)
```

Parameters:

- `topicFilter` (string): Topic filter to subscribe to. Supports MQTT wildcards: `+` matches exactly one topic level, `#` matches all remaining levels (must be last). Examples: `"zigbee2mqtt/front_door/contact"`, `"zigbee2mqtt/+/contact"`, `"frigate/#"`.
- `callback` (function): Handler function

Callback Function Signature:

```
function(topic, payload, json, retained, brokerID)
```

Callback Parameters:

- `topic` (string): The actual message topic (not the filter)
- `payload` (string): Raw message payload as UTF-8 text
- `json` (object or null): The payload pre-parsed as JSON when it is valid JSON, otherwise `null` — no `JSON.parse` boilerplate needed
- `retained` (boolean): `true` when this is a broker-retained message (delivered on subscribe, not a live event)
- `brokerID` (integer): ID of the broker the message arrived from

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

The subscription is registered with the broker automatically and restored on every reconnect — the script only declares its filter once. Messages are never throttled.

Example:

```
// Zigbee door sensor -> controller entry lights
ScriptEvents.onMQTTMessage("zigbee2mqtt/front_door", function(topic, payload, json, retained) {
    if (retained) return; // ignore the startup snapshot
    if (json && json.contact === false) {   // false = opened
        MyroAPI.turnOn("Entry Lights");
        MyroNotifications.send("Front door opened");
    }
});

// Frigate person detection -> announce
ScriptEvents.onMQTTMessage("frigate/events", function(topic, payload, json) {
    if (json && json.after && json.after.label === "person") {
        MyroSystem.speak("Person detected at " + json.after.camera);
    }
});
```

Example: Writing the Timeline from an Event Script

An event script is the natural place to add a moment: it sees the activity start and stop, and `MyroTimeline` turns that into a named moment with exactly the title you give it. Rules live with the script, so they go at the top where they are set again every time the script runs at launch.

This Control4 example opens a *Movie night* moment when the theater's Watch source comes on and closes it when the room is off, and it quiets a motion zone whose on-and-off trips were filling the Timeline (the Doors & motion lane still shows them). On a Savant controller use `ScriptEvents.onSavantRoomChange` and

`state.activeVideoService` instead; on a HAI / Leviton or NESS / ELK panel, an `onUnitChange` handler for the projector's unit does the same job.

```
ScriptEvents.clear();

// Rules belong to this script and return with it at launch.
MyroTimeline.mute("Hallway Motion");                  // a chatty motion zone, by name
MyroTimeline.rename("Unit:12", "{name} lamp {state}"); // "Reading lamp on" instead of "Reading on"

var movieNight = null;

ScriptEvents.onControl4RoomChange("Theater", function (roomName, state) {
    if (!state || !state.fresh) { return; }             // never act on stale feedback
    var watching = state.power === true && state.watchSourceID && state.watchSourceID !== "0";

    if (watching && !movieNight) {
        movieNight = MyroTimeline.moment("Movie night", {
            symbol: "film",
            tone: "info",
            detail: roomName + " is watching"
        });
        if (!movieNight) { MyroSystem.log("Movie night moment refused; see the warning above"); }
    } else if (!watching && movieNight) {
        MyroTimeline.end(movieNight);                    // closes the moment at now
        movieNight = null;
    }
});
```

A note is the lighter form for something that is over as soon as it happens, such as a pump that started or a door code that was used; pass `kind` so it is coloured like the lane it belongs to, and `object` and `room` so the Timeline knows which device and room it belongs to.

Clearing Event Handlers

To remove all event handlers for a script:

```
ScriptEvents.clear();
```

Best practice: Call `ScriptEvents.clear()` at the start of scripts with event handlers to avoid duplicate registrations.

Event Handler Best Practices

1. **Keep handlers simple** - Event handlers should execute quickly
2. **Use throttling** - The system automatically throttles some events to prevent excessive execution
3. **Error handling** - Wrap handler code in try-catch if needed

4. **One handler per event** - Each script can register one handler per event type
5. **Clear before registering** - Call `ScriptEvents.clear()` to remove old handlers

Scheduled Scripts

Scheduled scripts run automatically at specified times or intervals. You can schedule scripts to run:

- **At regular intervals** (e.g., every 60 seconds, every 5 minutes)
- **Using cron expressions** (e.g., daily at 9 AM, every Monday at midnight)
- **At sunrise or sunset** (e.g., daily at sunrise, 30 minutes before sunset)

Creating a Schedule

1. Select a script in the script list
2. Scroll down to the **Schedule** section (below the script code editor)
3. Click **"Create Schedule"**
4. Configure your schedule:
    - **Schedule Enabled** - Toggle to enable/disable the schedule
    - **Schedule Type** - Choose between "Interval", "Cron Expression", or "Sunrise/Sunset"

Interval-Based Scheduling

Interval-based schedules run a script at regular time intervals.

To create an interval schedule:

1. Select **"Interval"** as the schedule type
2. Enter the interval in **seconds** (e.g., `60` for every minute, `3600` for every hour)
3. Toggle **"Schedule Enabled"** to activate
4. The schedule is saved automatically

Example intervals:

- `60` - Every minute
- `300` - Every 5 minutes
- `3600` - Every hour
- `86400` - Every day (24 hours)

Example use cases:

- Check sensor values every 5 minutes
- Send periodic status updates
- Monitor network connectivity every minute

Cron-Based Scheduling

Cron-based schedules use standard cron expressions for precise scheduling.

Cron Expression Format:

```
minute hour day month weekday
```

Fields:

- **minute** (0-59) - Minute of the hour
- **hour** (0-23) - Hour of the day (24-hour format)
- **day** (1-31) - Day of the month
- **month** (1-12) - Month of the year
- **weekday** (0-6) - Day of the week (0=Sunday, 1=Monday, ..., 6=Saturday)

Special Characters:

- `*` - Matches any value (e.g., `*` in hour field means "every hour")
- `*/n` - Step value (e.g., `*/15` in minute field means "every 15 minutes")
- `n-m` - Range (e.g., `9-17` in hour field means "9 AM to 5 PM")
- `n,m,p` - List (e.g., `1,3,5` in weekday field means "Monday, Wednesday, Friday")

Common Cron Examples:

Expression	Description
`0 9 * * *`	Daily at 9:00 AM
`0 0 * * *`	Daily at midnight
`*/15 * * * *`	Every 15 minutes
`0 */2 * * *`	Every 2 hours
`0 9 * * 1`	Every Monday at 9:00 AM
`0 0 1 * *`	First day of every month at midnight
`0 9-17 * * 1-5`	Every hour from 9 AM to 5 PM, Monday through Friday
`0 0,12 * * *`	Twice daily at midnight and noon

Sunrise/Sunset Tokens in Cron:

You can use special tokens in cron expressions to schedule based on sunrise/sunset times:

- `@sunrise` - Replace with sunrise time
- `@sunset` - Replace with sunset time
- `@sunrise-30` - 30 minutes before sunrise
- `@sunset+60` - 60 minutes after sunset

Cron Token Examples:

- `0 @sunrise * * *` - Daily at sunrise
- `0 @sunset * * *` - Daily at sunset
- `0 @sunrise-15 * * *` - 15 minutes before sunrise
- `0 @sunset+60 * * *` - 1 hour after sunset

- `0 @sunrise-30 * * 1-5` - 30 minutes before sunrise, weekdays only

Note: Put solar tokens in the hour field. A token supplies both the hour and minute; use `0` in the minute field. Offsets are minutes, from –720 to +720. See your controller guide for solar time-zone and calendar behavior.

Learning Cron Format:

For more information about cron expressions and to test your cron expressions, see these resources:

- Crontab.guru - Interactive cron expression editor and validator
- Cron Expression Generator - Generate cron expressions with a visual interface
- Wikipedia: Cron - Comprehensive documentation on cron syntax

Note: Our cron parser uses the standard 5-field format (minute hour day month weekday). Some online tools may show 6 fields (including seconds) - ignore the seconds field when using those tools.

Sunrise/Sunset Scheduling

Helper runs solar schedules locally. Set up the solar source using your controller guide before enabling one. The controller guides describe where times come from and what happens when they are unavailable. Helper must remain running and the Mac awake.

Choose Sunrise/Sunset, select the event, set an offset from -720 to +720 minutes and enable the schedule. Cron solar tokens use the same offset range. Negative offsets run before the event; positive offsets run afterward.

Schedule Management

Viewing Schedule Status:

- The Schedule section shows the current schedule configuration
- **Next run time** is displayed when a schedule is enabled
- The next run time updates in real-time as it approaches

Editing a Schedule:

- Change the schedule type, interval, or cron expression
- Toggle "Schedule Enabled" to temporarily disable without deleting
- Changes are saved automatically

Deleting a Schedule:

- Click the **"Delete"** button in the Schedule section
- Confirm deletion in the dialog
- The script will no longer run on schedule (but can still be run manually)

Schedule Behavior

Script Requirements:

- Scripts must be **enabled** for schedules to run

- If a script is disabled, its schedule is automatically paused
- Re-enabling a script will resume its schedule

Connection Requirements:

- Scripts using `MyroAPI` or `MyroSensors` require controller connection
- If the controller is not connected, scheduled scripts will still run but may log warnings
- Use `MyroSensors.isDataFresh()` only when a decision requires a value confirmed during the current connection

Execution Logging:

- All scheduled script executions are logged to the main Log window
- Log entries show script name and execution status (success/error)
- Execution logs in the Scripting window show detailed execution information

Multiple Schedules:

- Each script can have **one schedule** (either interval, cron, or sunrise/sunset)
- Scripts can have both a schedule **and** event handlers
- Event handlers and schedules work independently

Schedule Best Practices

1. **Test before scheduling** - Run scripts manually before enabling schedules
2. **Use appropriate intervals** - Don't schedule scripts too frequently (minimum 1 second)
3. **Consider connection status** - Schedule scripts that require connection during times when the controller is typically connected
4. **Monitor execution logs** - Check logs regularly to ensure scheduled scripts are running correctly
5. **Use cron for precise timing** - Use cron expressions for time-based schedules (daily, weekly, etc.)
6. **Use intervals for frequent checks** - Use interval-based scheduling for monitoring tasks

Import/Export Scripts

The scripting system supports importing and exporting scripts in JSON format. This allows you to:

- **Backup your scripts** - Export scripts for safekeeping
- **Share scripts** - Export and share scripts with other users
- **Migrate scripts** - Move scripts between controllers or installations
- **Version control** - Track script changes by exporting versions

⚠️ Security Warning

Always review imported scripts before enabling them. Imported scripts are disabled by default for your safety. Only enable scripts from trusted sources after reviewing their code.

Exporting Scripts

To export scripts:

1. Click the **Export** button (📤) in the script list toolbar
2. Choose **"Export Selected"** to export the currently selected script, or **"Export All"** to export all scripts
3. Choose a location to save the JSON file
4. The export includes:
    - Script code and metadata (name, category, tags, notes)
    - Schedule configuration (if present)
    - Script ID (for tracking)

Export Format:

Scripts are exported as JSON with the following structure:

```
{
  "version": "1.0",
  "scripts": [
    {
      "scriptID": "unique-id",
      "name": "Script Name",
      "code": "// Script code here",
      "category": "automation",
      "tags": "tag1, tag2",
      "notes": "Script description",
      "createdDate": 1234567890,
      "exportDate": 1234567890,
      "schedule": {
        "enabled": true,
        "scheduleType": "interval",
        "interval": 300,
        "cronExpression": null
      }
    }
  ]
}
```

Importing Scripts

To import scripts:

1. Click the **Import** button (📩) in the script list toolbar
2. Select a JSON file containing exported scripts
3. Review the import results in the alert dialog
4. **Important**: Imported scripts are **disabled by default** - you must manually enable them after review

Import Behavior:

- **Single script files** - Imports one script
- **Multiple script files** - Imports all scripts in the file
- **Duplicate handling** - If a script with the same ID already exists, import will fail unless you enable "Replace Existing"
- **Validation** - All imported scripts are validated before import. Invalid scripts are rejected with an error message
- **Schedule import** - Schedules are imported along with scripts if present in the export

Import Errors:

If import fails, you'll see an error message explaining the issue:

- **"Script name is required"** - The exported file is missing required fields
- **"Script code is required"** - The exported file doesn't contain script code
- **"Script validation failed"** - The script code has syntax errors

- **"Script with ID 'xxx' already exists"** - A script with this ID already exists (enable "Replace Existing" to overwrite)
- **"Unsupported export version"** - The export file is from a newer version

Import/Export Best Practices

1. **Regular backups** - Export your scripts regularly as a backup
2. **Review before import** - Always review imported script code before enabling
3. **Test imported scripts** - Test imported scripts in a safe environment first
4. **Version control** - Export scripts before making major changes
5. **Share safely** - Only share scripts with trusted users
6. **Validate exports** - Verify exported files can be imported before sharing

Script Templates

Script templates provide pre-built script structures to help you get started quickly.

Accessing Templates

1. Click **"Browse Templates"** in the script list
2. Select a category (Automation, Monitoring, Notifications, etc.)
3. Choose a template
4. Fill in the parameters
5. Click **"Create Script"**

Available Template Categories

- **Automation** - Device control and scheduling
- **Monitoring** - Sensor monitoring and alerts
- **Notifications** - Push notification scripts
- **Security** - Zone and lock monitoring
- **Climate** - Thermostat control
- **Utilities** - Network checks and system utilities

Example Templates

- **"Zone Alert"** - Monitor a zone and send notification when tripped
- **"Temperature Monitor"** - Check temperature and automatically control AC unit based on thresholds
- **"Morning Routine"** - Weekday morning automation with lights, macros, and speech
- **"Away Mode"** - Turn off multiple devices and send confirmation notification
- Controller-specific templates follow the selected controller's capabilities; see its guide before using a panel feature.
- **"Internet Connectivity Check"** - Ping a URL and send notification if connection fails
- **"Zone Status Change Handler"** - Comprehensive event-based zone monitoring
- **"Thermostat Control"** - Temperature-based thermostat control with automatic adjustment
- **"Device Group Control"** - Control multiple devices together with a single action
- **"Unit Status Speaker"** - Get a unit status and speak the name and status aloud
- **"Vestaboard Message"** - Send a message to a Vestaboard by board ID with optional persistence
- **"LaMetric Notification"** - Send a notification to a LaMetric Time device with icon, sound, and priority

Best Practices

Script Organization

1. **Use descriptive names** - Name scripts clearly (e.g., "Turn on lights when zone tripped")
2. **One purpose per script** - Keep scripts focused on a single task
3. **Comment your code** - Add comments explaining complex logic
4. **Test before enabling** - Use the Validate and Run buttons before enabling event-based or scheduled scripts
5. **Document schedules** - Add comments explaining why a script is scheduled and when it runs

Connection Requirements

1. **Check connection before automation** - Scripts using `MyroAPI` or `MyroSensors` require controller connection
2. **Require confirmed data when appropriate** - Use `MyroSensors.isDataFresh()` only when a decision must use a value confirmed during the current connection
3. **Handle connection failures gracefully** - Check return values from `MyroAPI` methods
4. **Monitor warnings** - Pay attention to connection warnings in the execution log

Error Handling

```
// Always check return values
const success = MyroAPI.turnOn("Light");
if (!success) {
    MyroSystem.log("ERROR: Failed to turn on light - device not found");
}

// Use try-catch for complex operations
try {
    const temp = MyroSensors.getTemperature("Sensor");
    if (temp === -999) {
        throw new Error("Sensor not found");
    }
    // Process temperature...
} catch (error) {
    MyroSystem.log("Error: " + error);
}
```

Performance

1. **Never write an unbounded synchronous loop** - `while (true)`, `for (;;)`, or a loop that waits for a device state change is stopped after 30 seconds and, until then, blocks every other script for that controller. Use `setTimeout` / `setInterval` or an event handler instead

2. **Minimize API calls** - Cache values when possible
3. **Use event handlers wisely** - Don't register handlers for events that fire too frequently
4. **Return quickly** - Scripts and handlers are cut off at 30 seconds; keep synchronous work short and hand long waits to timers or event handlers

Security

1. **Validate inputs** - Check device names exist before using them
2. **Handle errors gracefully** - Don't let script errors crash the system
3. **Test thoroughly** - Test scripts in a safe environment before deploying
4. **Review imported scripts** - Always review code from imported scripts before enabling
5. **Avoid infinite loops** - Scripts with infinite loops are stopped after 30 seconds; until then they block other scripts for that controller
6. **Monitor script performance** - Large or complex scripts may impact system performance
7. **Keep remote management private** - Never expose port 9393, API keys, or MCP credentials directly to the public Internet; a key with script-management permission can run security-capable scripts

Security & Safety

⚠️ Important Security Notes

Scripts have access to your home automation system. Follow these security guidelines:

1. **Review all scripts before enabling** - Especially imported scripts or scripts from untrusted sources
2. **Test in safe environment** - Test scripts that control critical systems (locks, alarms) in a safe environment first
3. **Monitor script execution** - Check execution logs regularly for errors or unexpected behavior
4. **Disable unused scripts** - Disable scripts you're not actively using
5. **Backup before changes** - Export scripts before making major modifications

Script Execution Limits

The system enforces the following limits to prevent issues:

- **Execution cutoff**: a script or event handler still running after 30 seconds is stopped at its next loop iteration or function call and logged as an error
- **Event throttling**: 5 seconds minimum between event handler executions per script
- **System-wide event limit**: 10 event handler executions per minute across all scripts
- **Notification throttling**: Per-script: 30 seconds between notifications from same script

If a script exceeds these limits:

- Throttled events and notifications are dropped, and warnings are logged
- A stopped script's error names the script; fix the loop and run it again
- If a script cannot be stopped (a single native call that blocks), the app and controller processing continue, but other scripts and event handlers for that controller wait until it returns or Myro:Home Helper is restarted; the Scripts window shows a banner meanwhile
- The system does not crash

Connection Requirements

⚠️ **Warning: Stale Data Risk**

Scripts using `MyroSensors` may read **stale data** if the controller is not connected. This can cause:

- Incorrect automation decisions
- False alarms or missed alerts
- Incorrect device state readings

Always check connection status:

- Scripts using `MyroAPI` or `MyroSensors` require controller connection
- The system will log warnings if scripts run without connection

- `MyroSensors.isDataFresh()` is optional and means the object was confirmed during the current connection; it is not a general age test

Imported Scripts Safety

⚠️ **Security Warning for Imported Scripts:**

1. **Imported scripts are disabled by default** - This is intentional for your safety
2. **Review code before enabling** - Always read through imported script code
3. **Validate before use** - Use the Validate button to check for syntax errors and device names
4. **Test before production** - Run imported scripts manually before enabling schedules or event handlers
5. **Trust the source** - Only import scripts from trusted sources

What to look for in imported scripts:

- Suspicious code patterns (infinite loops, excessive API calls)
- Unfamiliar device names or operations
- Missing or unclear comments
- Unexpected notification or logging behavior

Script Validation

The system automatically checks scripts for:

- **Syntax errors** - JavaScript syntax validation
- **Device names** - Verifies device names exist in controller (shows warnings)
- **Security patterns** - Potential infinite loops, excessive recursion
- **Performance issues** - Very large scripts or deep nesting

Validation warnings don't block execution - They're logged for your review. You decide whether to proceed.

Error Handling Philosophy

The scripting system follows a "log and continue" approach:

- ✅ Errors are logged with full context
- ✅ Script execution continues (doesn't crash the app)
- ✅ Error history is maintained for debugging
- ✅ Warnings are shown but don't block execution

This means:

- Script errors won't crash Myro:Home Helper
- You can review errors in the execution log
- The system remains stable even with problematic scripts

Best Practices for Security

1. **Start simple** - Begin with simple scripts and build complexity gradually

2. **Test incrementally** - Test each part of a script before adding more
3. **Use validation** - Always validate scripts before enabling
4. **Monitor logs** - Regularly check execution logs for warnings or errors
5. **Backup regularly** - Export scripts as backups before major changes
6. **Document changes** - Add comments explaining why scripts do what they do
7. **Review periodically** - Periodically review enabled scripts to ensure they're still needed

Troubleshooting

Script Won't Run

1. **Check if script is enabled** - Toggle the Enabled switch
2. **Validate the script** - Click Validate to check for syntax errors and device names
3. **Check execution log** - Look for error messages in the log pane
4. **Verify device names** - Make sure device names match exactly (case-sensitive)
5. **Check controller connection** - If script uses `MyroAPI` or `MyroSensors`, ensure controller is connected

Event Handler Not Firing

1. **Check script is enabled** - Event handlers only work for enabled scripts
2. **Verify event registration** - Make sure `ScriptEvents.onXXX()` is called
3. **Check device/zone names** - Names must match exactly as configured (case-sensitive)
4. **Review throttling** - Some events are throttled to prevent excessive execution
5. **Clear old handlers** - Call `ScriptEvents.clear()` at start of script

Device Not Found

Symptoms: Script returns `false` from device control methods, or sensor methods return `-1` or `-999`

Causes:

1. **Device name doesn't match** - Device names must match exactly as stored in the controller (case-sensitive)
2. **Device was renamed** - If you renamed a device on the controller, scripts using the old name will fail
3. **Device doesn't exist** - The device may not exist in the controller's name database
4. **Wrong controller** - The device may exist in a different controller

Solutions:

1. **Use Validate button** - Checks device names and shows warnings for missing devices
2. **Verify device name** - Check the exact name in the controller or Myro:Home Helper interface
3. **Check case sensitivity** - "Kitchen Light" is different from "kitchen light"
4. **Update scripts after renaming** - If you renamed devices on the controller, update all scripts that reference them
5. **Test device existence** - Run the script manually and check return values:

```
const success = MyroAPI.turnOn("Device Name");
if (!success) {
    MyroSystem.log("ERROR: Device 'Device Name' not found!");
}
```

6. **Check execution logs** - Look for device not found errors in the execution log

7. **Re-download name data** - Use Sync Controller Data / Sync Now to refresh device names

Stale Sensor Data

Symptoms: Sensor readings don't update or are incorrect

Solutions:

- Check controller connection status
- Use `MyroSensors.isDataFresh()` to verify data was confirmed during the current connection:

```
if (!MyroSensors.isDataFresh("Main Thermostat")) {
    MyroSystem.log("WARNING: Temperature data is stale, skipping automation");
}
```

- Reconnect to controller to refresh sensor data
- Check execution log for connection warnings

Notifications Not Sending

Symptoms: `MyroNotifications.send()` returns `false` or no notification received

Causes:

1. **Throttling** - Script-level: 30 seconds between notifications from same script
2. **System limit** - System-wide: 10 notifications per minute
3. **Network issue** - No internet connectivity
4. **Push service unavailable** - OneSignal service issues

Solutions:

1. Check throttling messages in execution log
2. Wait 30 seconds between notification calls from same script
3. Verify network connectivity with `MyroNetwork.isConnected()`
4. Check if notification was sent with return value:

```
if (!MyroNotifications.send("Test")) {
    MyroSystem.log("Notification failed or throttled");
}
```

Import/Export Issues

Import Failed:

- Verify JSON file format is correct
- Check that required fields (name, code) are present
- Validate script code in the export file
- Check for duplicate script IDs (enable "Replace Existing" if needed)

- Review error message for specific issue

Export Not Working:

- Ensure at least one script is selected (for "Export Selected")
- Check file permissions for the save location
- Verify scripts are not corrupted (try validating them first)

Imported Script Not Appearing:

- Imported scripts are disabled by default - check the script list
- Refresh the script list if needed
- Check import result message for errors

Common JavaScript Errors

"ReferenceError: Can't find variable: xxx"

- Variable not defined - check variable names and spelling
- Make sure to declare variables with `var`, `const`, or `let`

"TypeError: undefined is not a function"

- Check API method names and spelling
- Verify you're calling methods correctly (e.g., `MyroAPI.turnOn()` not `MyroAPI.turnOn`)

"SyntaxError"

- Check for missing semicolons, brackets, or quotes
- Use the Validate button to find syntax errors
- Review JavaScript basics in the Appendix

Getting Help

1. **Check this guide** - Review the relevant section
2. **Review execution logs** - Error messages often provide clues
3. **Test with simple scripts** - Start with basic examples and build up
4. **Validate before running** - Always validate scripts before enabling
5. **Use autocomplete** - Type `MyroAPI.` to see available methods

Appendix: JavaScript Basics

Variables

```
var name = "Kitchen Light";
var level = 50;
var isOn = true;

// Modern JavaScript (ECMAScript 5+)
const MAX_TEMP = 80;  // Constant (cannot be reassigned)
let currentTemp = 72;  // Block-scoped variable
```

Functions

```
function turnOnLight() {
    MyroAPI.turnOn("Kitchen Light");
}

turnOnLight();

// Function with parameters
function setLightLevel(lightName, level) {
    return MyroAPI.setLevel(lightName, level);
}

setLightLevel("Bedroom Light", 50);
```

Conditionals

```
if (MyroSensors.isDeviceOn("Light")) {
    MyroAPI.turnOff("Light");
} else {
    MyroAPI.turnOn("Light");
}

// Multiple conditions
const temp = MyroSensors.getTemperature("Main");
if (temp > 76) {
    MyroSystem.log("Hot");
} else if (temp < 68) {
    MyroSystem.log("Cold");
} else {
    MyroSystem.log("Comfortable");
}
```

Loops

```
// For loop
for (var i = 0; i < 5; i++) {
    MyroSystem.log("Count: " + i);
}

// forEach (arrays)
const devices = ["Light 1", "Light 2", "Light 3"];
devices.forEach(function(device) {
    MyroAPI.turnOff(device);
});
```

Arrays

```
const devices = ["Light 1", "Light 2", "Light 3"];
MyroSystem.log("First device: " + devices[0]);
MyroSystem.log("Total devices: " + devices.length);

devices.push("Light 4");  // Add to end
```

String Concatenation

```
const temp = 72;
const message = "Temperature is " + temp + "°F";
MyroSystem.log(message);
```

Timers

```
// Execute after 5 seconds
setTimeout(function() {
    MyroAPI.turnOff("Light");
}, 5000);

// Execute every 10 seconds
const interval = setInterval(function() {
    MyroSystem.log("Still running...");
}, 10000);

// Stop interval after 1 minute
setTimeout(function() {
    clearInterval(interval);
}, 60000);
```

Conclusion

The Myro:Home Helper Scripting System provides powerful automation capabilities for your home automation controller. Start with simple scripts and gradually build more complex automations as you become familiar with the API.

Final Reminders

⚠️ **Always test scripts before enabling them in production**

⚠️ **Review imported scripts before enabling**

⚠️ **Monitor execution logs for errors and warnings**

⚠️ **Backup your scripts regularly using export**

⚠️ **Verify controller connection for automation scripts**

⚠️ **Use `MyroSensors.isDataFresh()` only when current-connection confirmation is required**

For additional help, refer to the execution logs and validate your scripts before enabling them. Use the autocomplete feature in the code editor to explore available API methods.

Happy Scripting! 🏠✨

Version 5.50 documentation: shared Helper behavior is maintained here; panel commands, events and setup are maintained in the separate controller guides. Copy the selected controller's live AI reference after updating Helper or its inventory.