Myro:Home Helper

HAI and Leviton Controller Guide

Version 5.50

HAI connection, panel scripting APIs and events, solar schedules, and HTTP control.

Myro Control, LLC | myrocontrol.com

Contents

About this guide

Connect your HAI or Leviton controller

- Editor namespaces

Moving From HaikuHelper

Optional audio receiver mode (build 3033)

JavaScript API Reference

- MyroAPI - Device Control
- MyroController - Controller & Object Data
- MyroAudio - Audio Zone Control
- MyroSensors - Read Device States
- HaikuHelper Migration Mapping (4.50)

Event-Based Scripts

- Available Event Types
- Additional Omni Events

Scheduled Scripts

- Sunrise and sunset for HAI

Controller examples

- Pattern 1: Temperature Monitoring with AC Control
- Pattern 2: Morning Routine
- Pattern 3: Security Monitoring with Event Handler
- Pattern 4: Away Mode
- Pattern 5: Conditional Device Control with State Tracking
- Pattern 6: Temperature Change Tracking
- Pattern 7: Multi-Condition Logic
- Pattern 8: Flag Counter Usage
- Pattern 9: Wiring Up a Virtual RA2 Keypad
- Pattern 10: Controller Down While the Mac Is Fine

HTTP controller control

- Units and Blinds
- Flags and Outputs
- Native Relative Controls (4.50)
- Buttons, UPB Links, and Messages
- Locks
- Thermostats
- Audio Zones
- User Settings, Area Lights, and Energy Cost

HTTP security control

- Set Security Mode
- Bypass or Restore a Zone

HTTP audio sources and metadata

- List Sources
- Read or Refresh One Source
- Legacy Audio Route

Legacy HAI HTTP routes

- Unit
- Flag
- Output
- Energy Cost
- HAI status example
- HAI object example
- Object Fields

About this guide

Use this guide for HAI and Leviton Omni/Lumina controllers. It contains this adapter's connection steps, controller APIs, panel events, solar source and HTTP command contracts. Optional hardware still needs panel configuration.

Read the shared Helper Scripting Guide for script management, Helper APIs, scheduling syntax and integrations. The HTTP API guide covers authentication and shared endpoints; the MCP guide covers client setup.

The selected controller's AI reference combines the shared scripting guide, this controller guide and its saved device names. Use **Help > HAI / Leviton Controller Guide** to open this document. API Reference badges identify controller and Helper features.

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

Connect your HAI or Leviton controller

Have an Omni-Link II network controller, its host/IP address, port (normally `4369`) and the two-part private key ready. Helper requires macOS 12 or later on a trusted local network.

Helper creates an initial controller entry automatically. Select it in the controller list, then open the **Connection** tab.

1. Enter a recognizable **System Name**.
2. Enter the controller's **Host/IP Address**.
3. Confirm the **Port**. Omni-Link II normally uses `4369`.
4. Enter **Private Key Part A** and **Private Key Part B** exactly as configured for the panel.
5. Select **Connect**.

The **Status** tab should change from disconnected to connected and begin showing the controller model, firmware, date and time, battery level, system features, and trouble indicators.

Run **Sync Controller Data / Sync Now** for initial discovery or after changing panel names/configuration. Helper persists the inventory in Core Data. Opening Helper or reconnecting refreshes lightweight status and values using the saved inventory; it does not require downloading names on every connection.

For shared services and keeping Helper running, continue with Getting Started.

Editor namespaces

Namespaces: `Device.` (everything), `Zone.`, `Unit.`, `Button.`, `Area.`, `Thermostat.`, `Message.`, `UserSetting.`, `Lock.`, `AccessReader.`, `AuxSensor.`, `Flag.`, `Output.`, `AudioZone.`, `AudioSource.`, `UPBLink.`, `Blind.`

Namespaces insert exact saved names as quoted JavaScript strings; they are editor shortcuts, not runtime objects. Optional families depend on the panel's configuration.

Moving From HaikuHelper

You do not need to replace the programming in your controller. Start by connecting Helper to the existing panel and completing a name/property synchronization. Keep an independent backup of your existing panel configuration and Haiku scripts.

HaikuHelper Workflow	Myro:Home Helper
Omni web interface	Browser control at the Helper Mac's port 9393
Temperature/humidity graphs	Logging settings and range-based graphs
Panel event log	Browser Events or the panel-events HTTP/MCP tools
Heating/cooling history	Reported-action intervals with unknown time shown explicitly
JavaScript automation	Scripts window and the current AI scripting reference
Notifications	Helper push registration, HomeKit, or supported integrations

Haiku scripts are not drop-in compatible. Port one automation at a time using Helper's scripting guide, actual controller object names and explicit tests. Start with logging-only handlers, verify events after an idle period and a reconnect, then enable actions. Avoid running equivalent Haiku and Helper automations simultaneously, which can issue duplicate commands.

There is no automatic Haiku graph/database import. Preserve your Haiku archives separately. Helper's original CSVs remain the source of its own history. Helper does not replace PC Access for panel programming or directly import Haiku's custom interfaces, generic TCP/serial scripts, AppleScript, SMTP or camera streams.

Optional audio receiver mode (build 3033)

Choose **File > HAI Audio Receivers (Prototype)...** with a HAI / Leviton controller selected. After connecting and syncing its names, enable the audio zones you want to test. Each enabled zone is a separate receiver accessory with its own pairing QR code and PIN. Select its row, then use **Add Accessory** in Apple Home. Existing HAP audio switches, bridge IDs and pairings remain in place; receiver mode does not change their settings.

The receiver exposes zone power, absolute volume, mute and configured audio-source selection. Test power and input selection in Home, and volume buttons in Control Center's Remote. Absolute volume and mute are available to compatible HomeKit clients; Apple's own controls vary by OS and need validation on your devices. Remote volume buttons step by two percentage points. A requested target is retained for at most two seconds so rapid presses accumulate and intermediate panel feedback does not reset the target. Reads and notifications still show only confirmed HAI values. There are no automatic retries; power-off, source changes and disconnect clear pending intent. The linked speaker follows zone power so Remote audio controls can stay active while the zone is on. Sources use their controller numbers; unavailable source slots stay hidden and cannot be selected.

Enable **Home slider** beside an enabled receiver for a 0–100% light-style volume slider within that accessory. Its on/off unmutes/mutes and never changes zone power. Build 3033 removes the redundant Mute switch added in 3032; use the slider’s on/off or a compatible client’s native speaker mute instead. Volume/unmute controls reject writes while the zone is off. Because the slider is a light service, room/home “all lights” commands affect mute. The receiver already exposes the native speaker service for Remote volume, but that service does not give Apple Home a speaker-style volume slider. Home controls the icons and layout; grouping does not guarantee compact controls. The option is off by default. Updating preserves receiver pairing and all remaining service/characteristic IDs, including the slider; only the redundant switch is removed. Disabling the receiver keeps the slider preference. Allow Home to refresh its accessory database after updating; avoid resetting pairing as a first step.

Mute follows PRO’s system-specific command selection: Russound, NuVo and HAI Hi-Fi use audio key 5, Xantech key 12 and Speaker Craft key 10. Russound, NuVo, Xantech and Speaker Craft infer mute from confirmed zero-volume feedback; HAI Hi-Fi uses its mute flag. Unknown systems keep explicit HAI mute/unmute commands. A toggle is sent only when confirmed state differs from the request. Duplicate requests while waiting send no second toggle; the opposite request is rejected until confirmation or a five-second timeout with fresh feedback. An off zone cannot receive a toggle. There is no automatic retry or reconnect replay.

The selected receiver shows its last command and whether HAI has confirmed it. After five seconds without matching feedback it reports **not confirmed; no retry sent**. This distinguishes missing Remote commands from controller feedback delays.

Values come only from current HAI audio-zone feedback. Connection loss or a status older than two minutes makes the receiver unavailable. A complete status is required for its first baseline; malformed or incomplete updates do not replace the last confirmed values. Reconnect needs a new baseline. Commands are bounded, sent once and never replayed; sending a command does not prove the equipment changed. Power, volume, mute and source changes made through an existing HAP switch, HAI control or app should appear on the receiver after panel feedback. Receiver control should likewise update the existing switch's power state.

Disable a receiver in this window to stop its separate server while keeping its pairing. Re-enabling or renaming it preserves its identity. **Reset This Receiver's Pairing...** resets only that receiver after confirmation; remove its old tile in Home and pair it again. The prototype servers run independently from the existing switch bridge's Start/Stop control. This prototype does not provide AirPlay streaming, track metadata, artwork or remote navigation/playback controls, and is unavailable for ELK and Savant controllers.

JavaScript API Reference

MyroAPI - Device Control

Control lights, switches, dimmers, thermostats, locks, flags, outputs, and other devices.

⚠️ **Important:** All device control methods require an active controller connection. If the controller is not connected, these methods will fail silently (return `false`).

Basic Control

`turnOn(deviceName)`

Turn a device on.

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

Parameters:

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

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

`turnOff(deviceName)`

Turn a device off.

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

Parameters:

- `deviceName` (string): Device name

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

`setLevel(deviceName, level)`

Set dimmer or device level (0-100).

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

Parameters:

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

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

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

These relative controls and the native counter methods below require an updated Helper 4.50 build; earlier builds may not expose them.

Request a relative lighting change using the panel's native command, without reading or overwriting Helper's cached level. `unitName` must identify an Omni **Unit**, by controller or custom name. Flags, outputs, blinds and directly connected Lutron bridge devices are not targets for these methods.

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

Parameters:

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

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

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

`toggle(deviceName)`

Toggle device between on and off.

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

Parameters:

- `deviceName` (string): Device name

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

Timer-Based Control

`turnOnTimer(deviceName, seconds)`

Turn device on for a specified duration (auto-off after timer expires).

```
MyroAPI.turnOnTimer("Porch Light", 99);  // On for 99 seconds
MyroAPI.turnOnTimer("Fan", 60);           // On for 1 minute
MyroAPI.turnOnTimer("Closet Light", 30);  // On for 30 seconds
```

Parameters:

- `deviceName` (string): Device name
- `seconds` (number): Duration in seconds (1-99, automatically clamped)

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

Use cases:

- Temporary lighting (closets, hallways)
- Timed ventilation fans
- Temporary device activation

`turnOffTimer(deviceName, seconds)`

Turn device off for a specified duration (auto-on after timer expires).

```
MyroAPI.turnOffTimer("Alarm", 30);  // Off for 30 seconds then back on
```

Parameters:

- `deviceName` (string): Device name
- `seconds` (number): Duration in seconds (1-99, automatically clamped)

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

Flag Control

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

`setFlag(flagName)`

Set a flag (turn on).

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

Parameters:

- `flagName` (string): Flag name

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

Use cases:

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

`clearFlag(flagName)`

Clear a flag (turn off).

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

Parameters:

- `flagName` (string): Flag name

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

Messages

`showMessage(messageName, beep)`

Show a controller text message on the consoles.

Parameters:

- `messageName` (string): The message's name
- `beep` (boolean): `true` sounds the console beeper and lights the LED; `false` lights the LED only

Returns: `true` if the message exists and the command was sent

Example:

```
MyroAPI.showMessage("Garage Door Open", true);
```

`clearMessage(messageName)`

Clear a displayed message from the consoles.

Example:

```
MyroAPI.clearMessage("Garage Door Open");
```

`setFlagCounter(flagName, value)`

Set flag counter value (0-255).

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

// Native increment (updated Helper 4.50+); no cached read/modify/write
MyroAPI.incrementFlag("Event Count");
```

Parameters:

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

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

Use cases:

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

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

Request one native counter increment or decrement on the controller. The name must identify a **Flag**, by controller or custom name, not a Unit or output. There is no amount or timer argument: each call requests exactly one step.

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

Unlike `setFlagCounter(name, cachedValue + 1)`, these commands use the panel's current counter, so a stale Helper cache cannot overwrite a newer count. Rapid calls remain separate and ordered. The counter is still the panel's 8-bit value; the panel owns behavior at 0 and 255. The protocol command table does not specify wrap versus saturation, so do not rely on either without testing your panel. Changing a flag can trigger any program configured to respond to it.

Parameters:

- `flagName`: A configured Flag's controller or custom name. Each call requests one native counter step; no amount or timer argument and no cached read/modify/write.
- Flags may trigger controller programs. Behavior at counter endpoints 0/255 is panel-defined; the protocol table does not promise wrapping or saturation.

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

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

Extended Omni Controls

These methods expose the remaining non-security Omni object controls. Every method returns `true` when the object exists and the command was sent, or `false` otherwise.

```
MyroAPI.setBlindLevel("Living Room Shade", 50);       // 0–100
MyroAPI.setOutput("Pool Relay", true);
MyroAPI.setOutputTimer("Pool Relay", true, 30);       // 0–99 seconds
MyroAPI.setUPBLink("Evening Scene", "on");           // on, off, store
MyroAPI.blinkUPB("Hall Light");                      // UPB units: blink 30 s to find it
MyroAPI.stopBlinkUPB("Hall Light");
MyroAPI.setLock("Front Door Lock", true);
MyroAPI.unlockFor("Front Door Lock", 30);             // seconds, up to 18 hours
MyroAPI.setAllLocks(true);                           // every door
MyroAPI.pressPhantomButton(4);                       // RadioRA 2 phantom button 1–255
MyroAPI.setLevitonScene(7, "on");                    // on, off, set

MyroAPI.sayMessage("Welcome Home");
MyroAPI.logMessage("Garage Door Open");
MyroAPI.phoneAndSay("Water Alarm", 1);               // stored phone number 1–8
MyroAPI.setKeypadBeepers(false);
MyroAPI.stopKeypadBeeping();

MyroAPI.setThermostatMode("Main", "auto");           // off, heat, cool, auto, emergency
MyroAPI.setThermostatFan("Main", "auto");            // auto, on
MyroAPI.setThermostatHold("Main", true);
MyroAPI.setHeatSetpoint("Main", 68);
MyroAPI.setCoolSetpoint("Main", 76);
MyroAPI.setHumidifySetpoint("Main", 35);
MyroAPI.setDehumidifySetpoint("Main", 55);

MyroAPI.setUserSetting("Sprinkler Minutes", 20);      // raw value 0–65535
MyroAPI.setAreaLights("House", false);
MyroAPI.setAllLights(false);
```

`setHeatSetpoint` and `setCoolSetpoint` use the controller's configured Fahrenheit or Celsius format. Humidity methods take a percentage. `adjustHeatSetpoint` and `adjustCoolSetpoint` shift a setpoint by a relative amount instead, which lets a script nudge a thermostat without reading its current value first.

`setBlindLevel(blindName, level)`

Move a blind or shade to a position.

Parameters:

- `blindName` (string): Blind name
- `level` (number): 0 (closed) to 100 (open)

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

`setOutput(outputName, on)`

Switch a controller output on or off.

Parameters:

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

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

`setOutputTimer(outputName, on, seconds)`

Switch an output for a limited time, after which the controller reverts it.

Parameters:

- `outputName` (string): Output name
- `on` (boolean): State to hold for the duration
- `seconds` (number): 0-99

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

`setUPBLink(linkName, state)`

Activate, deactivate, or store a UPB Link.

Parameters:

- `linkName` (string): UPB Link name
- `state` (string): `"on"`, `"off"`, or `"store"`

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

`blinkUPB(unitName)`

Blink a UPB device for 30 seconds at a one-second rate, so you can find which switch or module it is. Sends Omni command 26 with P1 = 30 (seconds) and P2 = unit number plus blink rate 2 (1 s) in the top bits. Helper 5.01 and later.

Parameters:

- `unitName` (string): Name of a unit whose unit type is UPB

Returns: `true` if the command was sent, `false` if no unit has that name, the unit is not a UPB unit, or the controller is not connected (the reason is written to the Execution Log)

`stopBlinkUPB(unitName)`

Stop a UPB device blinking before its 30 seconds are up (Omni command 27). Helper 5.01 and later.

Parameters:

- `unitName` (string): Name of a unit whose unit type is UPB

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

`setLock(lockName, locked)`

Lock or unlock an access-control lock.

Parameters:

- `lockName` (string): Lock name
- `locked` (boolean): `true` to lock, `false` to unlock

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

`unlockFor(lockName, seconds)`

Unlock a door for a time; the controller locks it again when the time is up. Sends Omni command 106 with P1 as the Omni duration: 1-99 seconds, then whole minutes up to 99, then whole hours up to 18 (rounded up). Helper 5.01 and later.

Parameters:

- `lockName` (string): Lock name
- `seconds` (number): 1 to 64,800 (18 hours)

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

`setAllLocks(locked)`

Lock or unlock every door at once (Omni command 105 or 106 with object 0). Helper 5.01 and later.

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

`sayMessage(messageName)` / `logMessage(messageName)`

Speak a stored message on the controller's voice output (Omni command 83), or write it to the controller's event log (command 81). Helper 5.01 and later.

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

`phoneAndSay(messageName, phoneNumber)`

Dial a phone number stored in the controller (1-8) and say the message (Omni command 84). Helper 5.01 and later.

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

`setKeypadBeepers(on)` / `stopKeypadBeeping()`

Turn every keypad's beeper on or off (Omni command 102), or silence keypads that are beeping now (command 103). Helper 5.01 and later.

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

`pressPhantomButton(number)`

Press a RadioRA 2 phantom button (Omni command 109). The controller keeps phantom buttons by number only, so scripts use the number from PC Access (1-255). Helper 5.01 and later.

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

`setLevitonScene(number, state)`

Turn a Leviton scene on or off, or set it to the lights' present levels (Omni commands 97, 96 and 98). Scenes are addressed by number (1-255). Helper 5.01 and later.

Parameters:

- `number` (number): Scene number from PC Access
- `state` (string): `"on"`, `"off"`, or `"set"`

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

`setThermostatMode(thermostatName, mode)`

Set a thermostat's system mode.

Parameters:

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

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

`setThermostatFan(thermostatName, mode)`

Set a thermostat's fan mode.

Parameters:

- `thermostatName` (string): Thermostat name
- `mode` (string): `"auto"` or `"on"`

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

`setThermostatHold(thermostatName, hold)`

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

Parameters:

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

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

The panel command only sends off or hold, so a script cannot set a vacation hold; that is set at the thermostat or panel. A thermostat's `holdMode` field (Helper 5.01 and later) reports `"off"`, `"hold"` or `"vacation"`, while `hold` stays 0 or 1.

`setHeatSetpoint(thermostatName, temperature)`

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

Parameters:

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

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

`setCoolSetpoint(thermostatName, temperature)`

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

Parameters:

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

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

`setHumidifySetpoint(thermostatName, percent)`

Set the humidify setpoint.

Parameters:

- `thermostatName` (string): Thermostat name
- `percent` (number): Relative humidity, 0-100

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

`setDehumidifySetpoint(thermostatName, percent)`

Set the dehumidify setpoint.

Parameters:

- `thermostatName` (string): Thermostat name
- `percent` (number): Relative humidity, 0-100

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

`adjustHeatSetpoint(thermostatName, degrees)`

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

Parameters:

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

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

`adjustCoolSetpoint(thermostatName, degrees)`

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

Parameters:

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

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

`showCamera(camera, touchscreen)`

Display a camera on an HAI touchscreen. The controller names neither, so both are numbers.

Parameters:

- `camera` (number): Camera 1-6
- `touchscreen` (number): Touchscreen 1-255

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

`setUserSetting(settingName, value)`

Write a controller user setting by name.

Parameters:

- `settingName` (string): User setting name
- `value` (number): Raw value, 0-65535

Returns: `true` if the setting exists and the value is in range, `false` otherwise

`setAreaLights(areaName, on)`

Turn every light in an area on or off.

Parameters:

- `areaName` (string): Area name
- `on` (boolean): `true` for on, `false` for off

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

`setAllLights(on)`

Turn every light on the controller on or off.

Parameters:

- `on` (boolean): `true` for on, `false` for off

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

---

Security Control

`setSecurityMode(areaName, mode)`

Arm or disarm an area.

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

Parameters:

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

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

`disarm(areaName)`

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

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

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

No panel code is needed, and none should be written into a script. These arm as the controller's own program would, using its Programmed user rather than a person's code; it is the same mechanism the Helper's HomeKit bridge uses when Siri arms the house, and the panel's event log shows the arm as programmed. Every queued request is written to the main log with the name of the script that made it and is labeled as requested, not as a confirmed state change.

Scripts run on your Mac and are written by whoever uses it, so they are treated as local control, in the same way a keypad macro is. Direct HTTP security requests are a separate path and need an API key created with arm/disarm enabled plus a user code that the panel validates. Remote script saving, enabling, disabling, and reloading instead require a key created with script-management permission. That permission includes the ability to run security-capable scripts without a panel code.

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

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

Macro/Button Control

`runMacro(macroName)`

Execute a macro or button defined in your controller.

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

Parameters:

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

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

Use cases:

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

Energy Cost Control

`setEnergyCost(level)`

Set the energy cost level on the controller. Accepts numeric (`0`-`3`) or named values (`Low`, `Medium`, `High`, `Critical`), case-insensitive. Sends the hardware command, updates CoreData, and posts an `energySyncUpdate` notification so `onEnergySync` event handlers fire.

```
MyroAPI.setEnergyCost("High");
MyroAPI.setEnergyCost("2");       // Same as "High"
MyroAPI.setEnergyCost("low");     // Case-insensitive
MyroAPI.setEnergyCost("0");       // Same as "Low"
```

Parameters:

- `level` (string): Energy cost level — `"0"`/`"Low"`, `"1"`/`"Medium"`, `"2"`/`"High"`, `"3"`/`"Critical"`

Returns: `true` if successful, `false` if the level string is invalid

Use cases:

- Implement custom peak/off-peak pricing schedules
- Integrate with external energy price APIs
- React to solar production or battery state

> **Push notification note:** Energy cost data (`energyCost`, `energyProj`) is automatically included in every visible push notification's data payload. Any `sendNotification` API call or `MyroNotifications.send()` script call will deliver updated energy cost to iOS devices — no separate silent push is needed.

MyroController - Controller & Object Data

`MyroController` exposes the same decoded controller and object snapshots used by the Helper's HTTP API and MCP server. Results are plain JavaScript objects and arrays, so scripts can inspect complete HAI state without issuing raw Omni-Link requests.

```
var controller = MyroController.getStatus();
MyroSystem.log("Model " + controller.model + ", connected=" + controller.connected);

var thermostats = MyroController.getObjects("Thermostat");
var main = MyroController.getObjectByName("Thermostat", "Main");
var zone1 = MyroController.getObject("Zone", 1);
var everything = MyroController.getObjects("*");
```

- `getStatus()` returns connection, panel model and firmware, troubles, clock/sunrise/sunset, synchronization timestamps, object counts, and Ethernet-quality data.

Controller capabilities

- `getStatus().backend` is `"hai"`. Read `capabilities` for adapter support and verify that optional hardware is configured before using it. Cross-controller handles use the destination controller's contract.
- `solarSchedules` is true; solar times come from the panel, with `solarSource: "controller"`. Schedules need valid downloaded times.
- The HAI adapter exposes `energyCost`, `batteryReading`, `freezeStatus`, `upbLinks`, `locks`, `accessControl`, `storedMessages`, `userSettings`, `audioSourceMetadata`, `systemStatusEvents`, `humidity`, `thermostatFanCycle`, `audioMuteState`, `absoluteAudioMute`, `areaLights` and `cameraCommands`. A capability describes adapter support, not installed hardware.
- HAI energy-cost levels are Low, Medium, High and Critical. Sense energy measurements are a separate Helper integration; neither implies the other is configured.
- Scalar unavailable readings use state `-1`, temperature `-999` and freshness `false`. Do not treat missing data as a valid zero.

Object inventory

- `getObjects(type)` returns all objects matching an Object Editor type. Pass `"*"`, `"all"`, or an empty string for every object.
- `getObject(type, number)` returns one object or `null`.
- `getObjectByName(type, name)` accepts either the controller name or custom display name and returns one object or `null`.

Type names match the Object Editor, including `Zone`, `Unit`, `Button`, `Area`, `Thermostat`, `Message`, `User Setting`, `Access Control`, `Access Control Reader`, `Access Control Lock`, `Auxiliary Sensor`, `Flag`, `Output`, `Audio Zone`, `Audio Source`, `UPB Link`, and `Blind`. The fields returned depend on the object kind; common fields include `type`, `typeCode`, `number`, `name`, `displayName`, `state`, `value`, and `updatedAt`.

Snapshots are read-only. Use `MyroAPI` and `MyroAudio` to make changes.

`refreshStatus()`

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

- Requests standard status for known zones, units/flags/outputs/blinds, areas, thermostats, messages, auxiliary sensors, audio zones, user settings, and access-control objects. It does not download names or scan unused capacity.
- With each access-control lock request, Helper also asks for the reader that shares the lock's number, so last-access fields are available before anyone badges in. Reader replies are not counted toward completion; a panel without readers cannot hold the refresh open. See Access control status.
- Concurrent calls share the in-flight request ID instead of adding panel traffic. Automatic startup/reconnect refreshes use the same tracking.
- Finishing a configuration sync supersedes an older in-flight refresh so newly discovered objects are included. The old listener receives `superseded` with `success: false` and its original request ID; the new refresh has a new ID.
- Register `ScriptEvents.onStatusRefresh` first and match `result.requestID`. Confirm `result.success` before using the refreshed data. A timeout leaves the connection alone.
- Standard replies update the cache without replaying zone/unit changes, alarm actions, or push alerts. A dedicated refresh-result event is delivered instead.
- This confirms cached observations received during the refresh, not an atomic panel snapshot, exact request/reply sequence correlation, or completion of a device action. An intervening live event can legitimately provide a newer value.
- Audio-source metadata and extended-only thermostat humidity are not part of this refresh. Use `MyroAudio.refreshSourceMetadata` for source metadata. Object freshness does not promise every optional field was refreshed.

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

`getRefreshStatus()`

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

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

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

MyroAudio - Audio Zone Control

Control an Omni-managed whole-house audio system — Russound, NuVo, Xantech, Speakercraft, or HAI Hi-Fi. Turn zones on/off, mute, set volume, select sources, run transport (play/pause/stop/next/previous), or send raw audio keys.

⚠️ **Important:** Audio control requires an active controller connection. `zone` and `source` are the 1-n numbers shown in the Object Editor (Audio Zones / Audio Sources). All methods return `true` if the command was sent, `false` otherwise.

Audio systems & transport keys: The transport helpers (`play`, `pause`, `stop`, `next`, `previous`, `volumeUp`, `volumeDown`) use the **Russound** key mapping — the most common Omni audio system. If your zones use a different system (NuVo, Xantech, Speakercraft), use `sendKey(zone, code)` with the code from that system's audio key table instead.

`power(zone, on)`

Turn an audio zone on or off.

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

Parameters:

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

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

`mute(zone, muted)`

Mute or unmute an audio zone.

```
MyroAudio.mute(1, true);   // Mute zone 1
MyroAudio.mute(1, false);  // Unmute zone 1
```

Parameters:

- `zone` (number): Audio zone number
- `muted` (boolean): `true` = mute, `false` = unmute

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

`setVolume(zone, percent)`

Set a zone's volume.

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

Parameters:

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

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

`setSource(zone, source)`

Select the audio source for a zone.

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

Parameters:

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

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

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

Transport controls for a zone (Russound key mapping).

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

Parameters:

- `zone` (number): Audio zone number

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

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

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

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

Parameters:

- `zone` (number): Audio zone number

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

`sendKey(zone, code)`

Send a raw audio key code (1-40) to a zone — for functions not covered by the transport helpers (favorites, numeric keys, tier-2 source selects, channel up/down) or for non-Russound audio systems. Use the code from your system's column in the Omni audio key table.

```
// Russound key codes: 1=Power, 2=Source step, 3=Vol up, 4=Vol down,
//   5=Mute, 6=Play, 7=Stop, 8=Pause, 11=Previous, 12=Next
MyroAudio.sendKey(1, 6);   // Play (Russound)
```

Parameters:

- `zone` (number): Audio zone number

- `code` (number): Audio key code (1-40), per your audio system's key table

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

Audio source metadata

The Helper caches the metadata fields reported by Omni-managed audio sources. Field meanings can vary by attached audio system, so both the raw position and field ID are retained.

```
var sources = MyroAudio.getSources();
var fields = MyroAudio.getSourceMetadata(2);
var requested = MyroAudio.refreshSourceMetadata(2);

fields.forEach(function(field) {
    MyroSystem.log("position=" + field.position +
                   " fieldID=" + field.fieldID +
                   " value=" + field.value);
});
```

- `getSources()` returns each configured source with its name, number, and cached metadata.
- `getSourceMetadata(source)` returns cached fields for one source.
- `refreshSourceMetadata(source)` requests all documented metadata positions from the panel. Responses arrive asynchronously; use `onAudioSourceMetadataChange` to react to them.

Example: "Welcome Home" Music on Disarm

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

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

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

MyroSensors - Read Device States

Read current states and values from devices and sensors.

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

`getDeviceState(deviceName)`

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

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

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

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

Parameters:

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

Returns:

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

`getTemperature(sensorName)`

Get temperature or humidity reading from sensors and thermostats.

```
const temp = MyroSensors.getTemperature("Main Thermostat");
// Returns: 68 (in Fahrenheit or Celsius based on system setting)

const humidity = MyroSensors.getTemperature("Wine Cellar Hum");
// Returns: 48 (humidity percentage)

if (temp > 76) {
    MyroAPI.turnOn("AC Unit");
    MyroNotifications.send("Temperature high: " + temp + "°F");
}
```

Parameters:

- `sensorName` (string): Temperature sensor, thermostat, or humidity sensor name

Returns:

- Number (temperature in °F/°C or humidity percentage)
- `-999` if sensor not found

Note: Temperature format (Fahrenheit or Celsius) matches your controller's system setting. Humidity sensors return percentage (0-100).

`isDeviceOn(deviceName)`

Check if device is on (convenience method).

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

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

Parameters:

- `deviceName` (string): Device name

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

`isDataFresh(deviceName)`

Starting with Helper 4.09, each connection queues a lightweight status snapshot of known zones, units (including flags/outputs), areas, thermostats, messages, auxiliary sensors, audio zones, user settings, and access-control objects after configuration is ready, including the last-access status of each lock's reader. This refresh updates the shared cache without firing script events or sending alarm notifications. It is asynchronous: a script that runs immediately at startup must not assume the snapshot has completed. Standard thermostat snapshots confirm temperature, setpoints, mode, fan, hold, and fault flags; extended-only fields such as thermostat humidity still depend on extended reports. A freshness result is object-level confirmation, not proof that every optional field has been refreshed. Check whether MHH confirmed the object's value during the current controller connection. A value remains fresh while that connection stays active, even if it has not changed recently. After a disconnect or restart, the result remains `false` until MHH receives a new status for that object.

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

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

Parameters:

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

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

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

HaikuHelper Migration Mapping (4.50)

HaikuHelper pattern	Myro:Home Helper translation	Important difference
`include(...)` and shared globals	`MyroLibraries.require(path)`, `exports`, `module.exports`	Local relative `.js` modules; exports are isolated per script/run, not global includes

HaikuHelper pattern	Myro:Home Helper translation	Important difference
`helper.controllers` and controller lookup	`MyroControllers.list()` and `MyroControllers.get(guid)`	Explicit GUID plus local origin Send To and destination Accept From opt-in
Object methods on a selected controller	`handle.api`, `handle.controller`, `handle.sensors`, `handle.audio`	Use the existing Myro method signatures; stale/connection checks use the destination
Speech start/stop callbacks	`MyroSystem.speakQueued(text, callback, options)`	Inspect `event.state`; `finished` means native completion, `cancelled` and `failed` are distinct
Clear/stop speech queue	`cancelSpeech(id)` / `cancelAllSpeech()`	Only the caller's script run; no control of other scripts, HTTP speech, or Sonos
`localStorage`	`MyroSystem.getPersistentVar` / `setPersistentVar` / `removePersistentVar`	Controller/script-scoped JSON; existing Haiku state is not imported
Per-controller event handlers/timers	`ScriptEvents.on...`, `setTimeout`, `setInterval`	Register in the originating script; rerun/disable/delete cleans its registrations and timers
Native counter increment/decrement	`MyroAPI.incrementFlag(name)` / `decrementFlag(name)` (4.50)	One panel step per call; no cached read/modify/write or automatic retry
Relative unit dim/brighten	`MyroAPI.dim(name, steps)` / `brighten(name, steps)` (4.50)	1-9 native steps, not percentage points; Unit objects only

- This is an API migration, not source compatibility. Replace Haiku callbacks and object names explicitly. The two relative-command families above are added in the updated 4.50 build; blink, HLC presets, additional console commands, generic TCP/serial, SMTP, AppleScript and speech recognition are not added.
- For amplifier/page orchestration, treat `queued` as admission only. Use native `started`/terminal outcomes and retain a separate hardware fail-safe; script teardown deliberately suppresses callbacks from its expired run, and a failed timeout does not prove that audio stopped.

Method Signature and Minimal Example:

```
(function() {
    var pageID = MyroSystem.speakQueued("Front door is open", function(event) {
        if (event.state === "started") MyroSystem.log("Native speech started");
        if (event.state === "finished" || event.state === "cancelled" || event.state === "failed") {
            MyroSystem.log("Speech ended: " + event.state);
        }
    });
    // MyroSystem.cancelSpeech(pageID); // Requests cancellation; observe its outcome.
})();
```

Event-Based Scripts

Available Event Types

Zone Status Changes

Monitor security zones for state changes.

Method Signature:

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

Parameters:

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

Callback Function Signature:

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

Callback Parameters:

- `zoneName` (string): Zone name that changed
- `status` (string): One-word summary of the zone, in priority order:
  - `"unknown"` - No confirmed status yet, or the received byte contains an undefined condition/alarm value. Do not treat this as an alarm or a secure zone.
  - `"tripped"` - Zone has tripped (latched alarm)
  - `"trouble"` - Zone has a trouble condition
  - `"tamper"` - Zone reports tamper (current condition 3); treat it like trouble
  - `"bypassed"` - Zone is bypassed, by the user or by the system
  - `"notReady"` - Zone is open / not ready
  - `"ready"` - Zone is secure
- `details` (object): The decoded status byte, for when the summary is not enough:
  - `raw` (number): The Omni-Link status byte as received
  - `condition` (string): `"secure"`, `"notReady"`, `"trouble"` or `"tamper"`
  - `latched` (string): `"secure"`, `"tripped"` or `"reset"` (was tripped, since reset)
  - `arming` (string): `"disarmed"`, `"armed"`, `"bypassedByUser"` or `"bypassedBySystem"`
  - `bypassed` (boolean): Shorthand for either bypassed value
  - `armed` (boolean): Shorthand for `arming === "armed"`
  - `troubleUnacknowledged` (boolean): A trouble occurred and has not yet been acknowledged; `condition` says whether trouble is still present now

- Detecting a restore: there is no separate restore event. A restore arrives as a zone change whose `details.bypassed` is `false` after it was `true`, so keep the previous value per zone and compare

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

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

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

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

Example:

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

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

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

Example: announce bypass and restore on Sonos

```
ScriptEvents.clear();

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

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

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

Unit/Device Status Changes

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

Method Signature:

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

Parameters:

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

Callback Function Signature:

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

Callback Parameters:

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

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

Example:

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

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

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

Flags, Outputs, Blinds, Sensors, and User Settings

These typed handlers expose extended Omni object-status packets. Pass an exact name or `"*"` for every object of that kind.

```
ScriptEvents.onFlagChange("*", function(name, value, isSet) {
    MyroSystem.log("Flag " + name + " = " + value);
});

ScriptEvents.onOutputChange("*", function(name, isOn) {
    MyroSystem.log("Output " + name + " on=" + isOn);
});

ScriptEvents.onBlindChange("*", function(name, isOpen, level) {
    MyroSystem.log("Blind " + name + " level=" + level);
});

ScriptEvents.onAuxSensorChange("*", function(name, value, status) {
    MyroSystem.log("Sensor " + name + " value=" + value + " status=" + status);
});

ScriptEvents.onUserSettingChange("*", function(name, value, settingType) {
    MyroSystem.log("Setting " + name + " = " + value);
});
```

- `onFlagChange(name, callback)` calls `callback(flagName, value, isSet)`.
- `onOutputChange(name, callback)` calls `callback(outputName, isOn)`.
- `onBlindChange(name, callback)` calls `callback(blindName, isOpen, level)`.
- `onAuxSensorChange(name, callback)` calls `callback(sensorName, value, status)`.
- `onUserSettingChange(name, callback)` calls `callback(settingName, value, settingType)`.

UPB Link Activations

React to UPB Links firing on the Omni controller. UPB Links are programmed in the Omni configuration and represent logical actions (a single link can affect multiple units, scenes, etc.). The Omni reports link activations as part of its "Other Event" notification stream — separate from individual unit changes.

Method Signature:

```
ScriptEvents.onUPBLink(linkNumber, callback)
```

Parameters:

- `linkNumber` (integer): Specific UPB Link number to listen for (e.g., 11). Pass `0` to fire on **any** link activation.
- `callback(linkNumber, command)` (function): Called when the link fires.
  - `linkNumber` (integer): The link number that fired
  - `command` (string): One of `"on"`, `"off"`, `"store"`, `"fade stop"`, or `"unknown"`

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

Examples:

```
// React to a specific link (e.g. Master Bedroom 5min Fade UP — Link 9)
ScriptEvents.onUPBLink(9, function(linkNumber, command) {
    MyroSystem.log("UPB Link " + linkNumber + " " + command);
    if (command === "on") {
        MyroLutron.turnOn(1, "/zone/2");  // bridge to Caseta
    }
});

// Catch every UPB Link for logging or general routing
ScriptEvents.onUPBLink(0, function(linkNumber, command) {
    MyroSystem.log("Link " + linkNumber + " fired: " + command);
});

// Use links as scene triggers — multiple actions per link
ScriptEvents.onUPBLink(99, function(linkNumber, command) {  // Alarm arming inside lights
    if (command === "on") {
        MyroAPI.turnOff("Living Room Lamp");
        MyroAPI.turnOff("Kitchen Lights");
        MyroAPI.turnOn("Hallway Light");
    }
});
```

Notes:

- UPB Links are NOT the same as individual unit changes. A single link can drive many units; reacting to the link is often cleaner than reacting to each unit.
- Link names from your Omni programming are not included in the event payload (only the link number). Maintain a mapping in your script if you want readable code:

  ```
  var LINK_NAMES = { 9: "MBR Fade Up", 11: "LR U44", 99: "Alarm Arming" };
  ```

- Some UPB devices send dim ramps as a sequence of `on` / `fade stop` commands. The handler will fire once per command.

Message Events

Method Signature:

```
ScriptEvents.onMessageChange(messageName, callback)
```

Fires when a controller text message is shown on the consoles, acknowledged, or cleared. Pass `"*"` to hear about every message.

Callback parameters:

- `messageName` (string): The message's name

- `state` (string): `"off"`, `"displayed"`, `"notAcknowledged"`, or `"displayedNotAcknowledged"`
- `isDisplayed` (boolean): Currently shown on the consoles
- `isUnacknowledged` (boolean): Shown and not yet acknowledged

Example:

```
// Announce a message the controller puts up, and clear it once someone acknowledges it
ScriptEvents.onMessageChange("Garage Door Open", function(name, state, displayed, unacked) {
    if (state === "displayedNotAcknowledged") {
        MyroSonos.speak("The garage door is open", "Kitchen");
    } else if (state === "off") {
        MyroSystem.log(name + " cleared");
    }
});

// Every message, for finding out what the controller sends
ScriptEvents.onMessageChange("*", function(name, state) {
    MyroSystem.log("Message " + name + ": " + state);
});
```

Audio Zone Changes

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

Method Signature:

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

Parameters:

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

Callback Function Signature:

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

Callback Parameters:

- `zoneName` (string): Audio zone name that changed
- `power` (boolean): Power state (`true` = on, `false` = off)
- `volume` (number): Volume level (0-100)
- `source` (number): Source number currently selected
- `mute` (boolean): Reported mute state; true for muted and false for unmuted.

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

Example:

```
// Monitor living room audio zone
ScriptEvents.onAudioZoneChange("Living Room Audio", function(zoneName, power, volume, mute, source)
        {
    MyroSystem.log("Audio Zone: " + zoneName);
    MyroSystem.log("Power: " + (power ? "On" : "Off"));
    MyroSystem.log("Volume: " + volume + "%");
    MyroSystem.log("Mute: " + (mute ? "Yes" : "No"));
    MyroSystem.log("Source: " + source);

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

    if (power && !mute && volume > 0) {
        MyroSystem.log("Audio is playing in " + zoneName);
    }
});
```

Audio Source Metadata Changes

`onAudioSourceMetadataChange(sourceName, callback)` fires for each metadata field returned by the panel. Pass an exact audio source name or `"*"` for all sources.

```
ScriptEvents.onAudioSourceMetadataChange("*", function(sourceName, sourceNumber,
                                                       position, fieldID, value,
                                                       sequenceNumber) {
    MyroSystem.log(sourceName + ": " + value +
                   " (position " + position + ", field " + fieldID + ")");
});

MyroAudio.refreshSourceMetadata(2);
```

The callback values are the packet just received and are authoritative. `sequenceNumber` changes when the attached audio system reports a new metadata set.

Lock Status Changes

Monitor door locks for lock/unlock state changes.

Method Signature:

```
ScriptEvents.onLockChange(lockName, callback)
```

Parameters:

- `lockName` (string): Lock name (case-sensitive, must match controller exactly)

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

Callback Function Signature:

```
function(lockName, isLocked)
```

Callback Parameters:

- `lockName` (string): Lock name that changed
- `isLocked` (boolean): Lock state (`true` = locked, `false` = unlocked)

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

Example:

```
// Monitor front door lock
ScriptEvents.onLockChange("Front Door Lock", function(lockName, isLocked) {
    if (isLocked) {
        MyroSystem.log(lockName + " is now locked");
        MyroNotifications.send("Front door locked");
        MyroAPI.setFlag("Front Door Locked");
    } else {
        MyroSystem.log(lockName + " is now unlocked");
        MyroNotifications.sendWithSound("Front door unlocked!", "Perimeter.caf");
        MyroAPI.clearFlag("Front Door Locked");
    }
});

// Monitor multiple locks
const locks = ["Front Door Lock", "Back Door Lock", "Garage Lock"];
locks.forEach(function(lockName) {
    ScriptEvents.onLockChange(lockName, function(name, isLocked) {
        MyroSystem.log(name + " " + (isLocked ? "locked" : "unlocked"));
    });
});
```

Access-Control Reader Events

`onAccessControlEvent(readerName, callback)` reports credential activity separately from a lock's physical state. Pass an exact reader name or `"*"` for every reader.

Method Signature:

```
ScriptEvents.onAccessControlEvent(readerName, callback)
```

Callback Function Signature:

```
function(readerName, granted, userNumber)
```

```
ScriptEvents.onAccessControlEvent("*", function(readerName, granted, userNumber) {
    MyroSystem.log(readerName + ": access " +
                   (granted ? "granted" : "denied") +
                   " for user " + userNumber);
});
```

The callback is `function(readerName, granted, userNumber)`. `userNumber` is the Omni user number reported by the reader; no user code is exposed. Panel-defined numbers are 251 Duress, 252 Keyswitch, 253 Quick Arm, 254 PC Access and 255 Programming; 0 means no user.

The event fires when the panel pushes a reader status, which it does on every badge or keypad code at that reader. A status refresh also reads each reader's last access to fill the object fields, but it updates the cache without firing this event. To read the last result at any time, use `MyroController.getObject("Access Control Reader", n)` or the lock's `lastAccess...` fields; see Access control status.

Thermostat Changes

Monitor thermostats for temperature and mode changes.

Method Signature:

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

Parameters:

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

Callback Function Signature:

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

Callback Parameters:

- `thermostatName` (string): Thermostat name that changed
- `temperature` (number): Current temperature (in °F or °C based on system setting)
- `mode` (string or null): System mode - one of:
  - `"heat"` - Heating mode
  - `"cool"` - Cooling mode
  - `"auto"` - Auto mode
  - `"off"` - Off mode
  - `null` - Mode not available

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

Example:

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

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

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

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

Area Status Changes

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

Method Signature:

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

Parameters:

- `areaName` (string): Area name (case-sensitive, must match controller exactly)
- `callback` (function): Handler function to execute when area status changes

Callback Function Signature:

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

Callback Parameters:

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

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

Example:

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

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

Connection Status Changes

Monitor controller connection status.

Method Signature:

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

Parameters:

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

Callback Function Signature:

```
function(isConnected)
```

Callback Parameters:

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

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

Example:

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

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

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

Ethernet Quality

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

Method Signature:

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

Parameters:

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

Callback Function Signature:

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

Callback Parameters:

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

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

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

Example:

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

System Status Updates

Monitor system status updates (various system events).

Method Signature:

```
ScriptEvents.onSystemStatus(statusType, callback)
```

Parameters:

- `statusType` (string): Type of system status to monitor (e.g., "battery", "trouble", etc.)
- `callback` (function): Handler function to execute when system status changes

Callback Function Signature:

```
function(statusType, value)
```

Callback Parameters:

- `statusType` (string): Type of status update
- `value` (string or number): Status value

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

Example:

```
// Monitor system trouble status
ScriptEvents.onSystemStatus("trouble", function(statusType, value) {
    MyroSystem.log("System trouble: " + statusType + " = " + value);
    MyroNotifications.send("System trouble detected: " + value);
});

// Monitor battery status
ScriptEvents.onSystemStatus("battery", function(statusType, value) {
    MyroSystem.log("Battery status: " + value);
    if (value < 20) {
        MyroNotifications.send("WARNING: Low battery detected!");
    }
});
```

Additional Omni Events

`onOmniEvent(eventType, callback)`

Method Signature:

```
ScriptEvents.onOmniEvent(eventType, callback)
```

Parameters:

- `eventType` (string): One of the types below, or `"*"` for all listed types. Unknown types are rejected.
- `callback` (function): Called with one event object, in the order events arrived in the panel's batch. Events without a controller identity are dropped, never broadcast to other controllers.

Callback Parameters:

- `event.eventType`: The normalized type below.
- `event.controllerGUID`: The reporting controller.
- `event.statusType`: The underlying CommKit event name. Other fields are the panel's decoded payload, not inferred changes.

Type	Payload Fields
`switchPress`	`unitNumber`, `state` (0 off, 1 on, 2-11 switch/scene 1-10), `stateString`
`button`	`buttonNumber` (Omni macro button activated)
`proLink`	`messageNumber` (configured Pro-Link message event, not raw serial bytes)
`centralite`	`messageNumber` (decoded CentraLite switch number)

Type	Payload Fields
`x10`	`houseCode`, `unitNumber`, `allUnitsFlag`, `state`, `stateString`
`compose`	`houseCode`, `unitNumber`, `state` (0 off, 1 on, 2-13 scenes A-L), `stateString`
`camera`	`action` (camera 1-6, 0 unknown), `actionString`
`upbLink`	`linkNumber`, `commandString` and the existing decoded link fields

Returns: `true` when registered; `false` on invalid type or registration failure.

- A reported UPB/ALC/RadioRA/StarLite switch press is distinct from a UPB link and from a unit-state change. This API is not a UPB bus sniffer: only events reported by the panel are available.
- `onUPBLink(number, callback)` is unchanged, including its `(linkNumber, command)` arguments. Registering both it and the generic `upbLink` listener intentionally delivers the same link to both listeners.
- These events are not throttled like diagnostic system events. Avoid scripts that call the same macro unconditionally from its own `button` event.

```
ScriptEvents.onOmniEvent("switchPress", function(event) {
    MyroSystem.log("Switch unit=" + event.unitNumber + " action=" + event.stateString);
});
ScriptEvents.onOmniEvent("button", function(event) {
    MyroSystem.log("Omni button activated: " + event.buttonNumber);
});
```

`onStatusRefresh(callback)`

Method Signature:

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

Parameters:

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

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

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

Scheduled Scripts

Sunrise and sunset for HAI

Helper runs these schedules locally using sunrise/sunset values downloaded from the controller. Verify that the panel location/time configuration and downloaded solar times are correct. Helper must remain running and the Mac awake; controller actions require a connection.

Choose **Sunrise/Sunset** in Schedule settings, select the event and an offset from -720 to +720 minutes, then enable the schedule. Cron uses the same `@sunrise` and `@sunset` tokens described in the shared scheduling guide. Negative offsets run before the event; positive offsets run afterward.

Controller examples

Pattern 1: Temperature Monitoring with AC Control

```
// Monitor temperature and control AC automatically
const temp = MyroSensors.getTemperature("Main Thermostat");

// Check data freshness first
if (!MyroSensors.isDataFresh("Main Thermostat")) {
    MyroSystem.log("WARNING: Temperature data is stale, skipping automation");
} else {
    MyroSystem.log("Current temperature: " + temp + "°F");

    if (temp > 76) {
        // Only turn on if not already on
        if (!MyroSensors.isDeviceOn("AC Unit")) {
            MyroAPI.turnOn("AC Unit");
            MyroNotifications.send("Temperature high (" + temp + "°F), AC turned on");
            MyroSystem.log("AC turned on due to high temperature");
        }
    } else if (temp < 68) {
        // Turn off if currently on
        if (MyroSensors.isDeviceOn("AC Unit")) {
            MyroAPI.turnOff("AC Unit");
            MyroSystem.log("Temperature comfortable (" + temp + "°F), AC turned off");
        }
    }
}
```

Schedule: Interval, 300 seconds (every 5 minutes)

Pattern 2: Morning Routine

```
// Weekday morning automation
const hour = MyroTime.getHour();

if (hour === 7 && MyroTime.isWeekday()) {
    // Gradual light wake-up
    MyroAPI.setLevel("Bedroom Lights", 30);

    // Start coffee maker
    MyroAPI.runMacro("Coffee Maker");

    // Adjust thermostat
    const temp = MyroSensors.getTemperature("Main Thermostat");
    if (temp < 68) {
        MyroAPI.turnOn("Heater");
    }

    // Speak greeting
    MyroSystem.speak("Good morning! It's " + MyroTime.getCurrentTime());

    // Send notification
    MyroNotifications.send("Morning routine started");

    MyroSystem.log("Morning routine executed");
}
```

Schedule: Cron expression `0 7 * * 1-5` (7 AM weekdays)

Pattern 3: Security Monitoring with Event Handler

```
// Clear old handlers first to avoid duplicates
ScriptEvents.clear();

// Monitor front door zone
ScriptEvents.onZoneChange("Front Door", function(zoneName, status) {
    if (status === "tripped") {  // Zone opened/tripped
        // Turn on entry lights
        MyroAPI.turnOn("Entry Lights");

        // Auto-off after 99 seconds
        MyroAPI.turnOnTimer("Entry Lights", 99);

        // Send notification with security sound
        MyroNotifications.sendWithSound("Front door opened", "Perimeter.caf");

        // Play alert sound
        MyroSystem.playSound("Glass.aiff");

        // Log with timestamp
        MyroSystem.log("Front door opened at " + MyroTime.getCurrentTime());

        // Check if armed
        const isArmed = MyroSensors.getDeviceState("System Armed");
        if (isArmed === 1) {
            MyroNotifications.sendWithSound("WARNING: Front door opened while armed!",
        "Trouble.caf");
        }
    } else if (status === "ready") {  // Zone closed/ready
        MyroSystem.log("Front door closed");
    } else if (status === "trouble") {  // Zone has trouble
        MyroNotifications.send("WARNING: Front door zone has trouble!");
    }
});
```

Note: Event handlers persist until script is disabled or deleted. Always call `ScriptEvents.clear()` at the start of scripts with event handlers to avoid duplicate registrations.

Pattern 4: Away Mode

```
// Turn off all devices when leaving home
const devices = [
    "Living Room Lights",
    "Kitchen Lights",
    "Bedroom Lights",
    "TV",
    "Bedroom Fan",
    "AC Unit"
];

let successCount = 0;
let failCount = 0;

devices.forEach(function(device) {
    if (MyroAPI.turnOff(device)) {
        successCount++;
        MyroSystem.log("✓ Turned off: " + device);
    } else {
        failCount++;
        MyroSystem.log("× Failed to turn off: " + device + " (device not found)");
    }
});

// Set away mode flag
MyroAPI.setFlag("Away Mode");

// Send summary notification
let summary = "Away mode: " + successCount + " devices turned off";
if (failCount > 0) {
    summary += " (" + failCount + " failed)";
}
MyroNotifications.send(summary);

// Confirmation beep
MyroSystem.beep();

MyroSystem.log("Away mode completed");
```

Pattern 5: Conditional Device Control with State Tracking

```
// Only run AC if someone is home
const isAway = MyroSensors.getDeviceState("Away Mode");
const temp = MyroSensors.getTemperature("Main Thermostat");

if (isAway === 1) {
    // Away mode active - turn off AC to save energy
    if (MyroSensors.isDeviceOn("AC Unit")) {
        MyroAPI.turnOff("AC Unit");
        MyroSystem.log("AC turned off (away mode active)");
    }
} else {
    // Someone is home - normal automation
    if (temp > 76 && !MyroSensors.isDeviceOn("AC Unit")) {
        MyroAPI.turnOn("AC Unit");
        MyroSystem.log("AC turned on (home, temp: " + temp + "°F)");
    } else if (temp < 68 && MyroSensors.isDeviceOn("AC Unit")) {
        MyroAPI.turnOff("AC Unit");
        MyroSystem.log("AC turned off (home, temp: " + temp + "°F)");
    }
}
```

Pattern 6: Temperature Change Tracking

```
// Track temperature changes and alert on rapid increases
const currentTemp = MyroSensors.getTemperature("Main Thermostat");
const lastTemp = MyroSystem.getVar("lastTemp");

if (lastTemp === null) {
    // First run - initialize
    MyroSystem.setVar("lastTemp", currentTemp);
    MyroSystem.log("Temperature tracking initialized: " + currentTemp + "°F");
} else {
    const tempChange = currentTemp - lastTemp;

    if (tempChange > 2) {
        // Temperature rising quickly
        MyroSystem.log("Temperature rising quickly: " + lastTemp + "°F → " + currentTemp + "°F");
        MyroNotifications.send("Temperature spike detected: " + currentTemp + "°F (+" + tempChange +
        "°)");
    } else if (tempChange < -2) {
        // Temperature dropping quickly
        MyroSystem.log("Temperature dropping quickly: " + lastTemp + "°F → " + currentTemp + "°F");
    }

    // Update stored temperature
    MyroSystem.setVar("lastTemp", currentTemp);
}
```

Schedule: Interval, 300 seconds (every 5 minutes)

Pattern 7: Multi-Condition Logic

```
// Complex automation with multiple conditions
const temp = MyroSensors.getTemperature("Main Thermostat");
const hour = MyroTime.getHour();
const isWeekday = MyroTime.isWeekday();
const isAway = MyroSensors.getDeviceState("Away Mode") === 1;

if (!isAway && isWeekday && hour >= 7 && hour < 9) {
    // Weekday morning, someone home
    MyroAPI.setLevel("Bedroom Lights", 50);
    MyroAPI.runMacro("Coffee Maker");

    if (temp < 68) {
        MyroAPI.turnOn("Heater");
        MyroSystem.log("Morning routine: heater turned on (temp: " + temp + "°F)");
    }

    MyroSystem.speak("Good morning! It's " + MyroTime.getCurrentTime());

} else if (!isAway && !isWeekday && hour >= 9) {
    // Weekend morning, someone home
    MyroAPI.setLevel("Bedroom Lights", 30);  // Softer lighting
    MyroSystem.log("Weekend morning routine (relaxed mode)");

} else if (isAway) {
    // Away mode - energy saving
    if (MyroSensors.isDeviceOn("Heater")) {
        MyroAPI.turnOff("Heater");
        MyroSystem.log("Away mode: heater turned off");
    }
}
```

Pattern 8: Flag Counter Usage

```
// Count observed openings with a native panel increment (updated Helper 4.50+).
(function() {
    var previousCondition = null;
    ScriptEvents.clear();
    ScriptEvents.onZoneChange("Front Door", function(name, status, details) {
        if (!details) return;
        var opened = details.condition === "notReady" &&
                     previousCondition !== "notReady";
        previousCondition = details.condition;
        if (opened && !MyroAPI.incrementFlag("Door Open Counter")) {
            MyroSystem.log("Door counter increment was not submitted");
        }
    });

    ScriptEvents.onFlagChange("Door Open Counter", function(name, value) {
        MyroSystem.log("Panel reports door counter: " + value);
    });
})();
```

This counts openings observed while the script is running, not a durable audit trail. The counter is 8-bit, and events during disconnection can be missed. Use `setFlagCounter(name, 0)` when an intentional reset is appropriate; do not derive an increment from a value captured when the script started.

Pattern 9: Wiring Up a Virtual RA2 Keypad

- **HAI controllers do not support RA2 keypads.** Naming a keypad in PC Access gives you a *virtual RA2 control*: a unit that appears in Myro:Home but does nothing on the RA2 side, because the Omni sends an output command and a keypad is not an output. HAI's own Snap-Link behaves the same way.
- That virtual control is still useful, because the Omni reports its state. The Helper can watch it and call the RA2 bridge directly, so **On, Off and each scene value can be wired to whatever you want** -- including a real keypad button press via `tapButton`.
- `tapButton` reaches the keypad over the RA2 bridge (`#DEVICE,<id>,<component>,3` then `,4`, straight to the repeater), which is why it works where the Omni unit cannot.

```
// Turn a dead RA2 keypad unit into a working keypad button.
var OMNI_UNIT = "Kitchen Keypad";  // the keypad unit, as on the Omni
var BRIDGE_ID = 1;                 // MyroLutron.getBridges() to list
var KEYPAD_ID = 7;                 // the keypad's Integration ID
var BUTTON    = 1;                 // component number (see the report)

ScriptEvents.clear();
ScriptEvents.onUnitChange(OMNI_UNIT, function(name, isOn, level) {
    if (!isOn) return;             // ignore the turnOff below

    MyroLutron.tapButton(BRIDGE_ID, KEYPAD_ID, BUTTON);
    MyroSystem.log("RA2 keypad: " + name + " -> button " + BUTTON);

    // Momentary: leaving it off keeps it out of the "On" filter.
    MyroAPI.turnOff(OMNI_UNIT);
});
```

One unit covers the whole keypad: the value it carries picks the button, so setting the virtual control to 3 presses button 3.

```
var MAX_BUTTON = 6;

ScriptEvents.onUnitChange(OMNI_UNIT, function(name, isOn, level) {
    if (!isOn) return;
    var button = (level > 0 && level <= MAX_BUTTON) ? level : 1;
    MyroLutron.tapButton(BRIDGE_ID, KEYPAD_ID, button);
    MyroAPI.turnOff(OMNI_UNIT);
});
```

If a physical press re-triggers the script. On some setups the Omni marks the keypad unit On when someone presses the keypad by hand, which re-enters the handler and presses the button a second time. Record real presses and ignore unit changes that follow them:

```
var lastPhysicalPress = 0;

ScriptEvents.onRA2ButtonPress("*", function(name, btn, action, ref) {
    if (ref === KEYPAD_ID) lastPhysicalPress = Date.now();
});

ScriptEvents.onUnitChange(OMNI_UNIT, function(name, isOn, level) {
    if (!isOn) return;
    if (Date.now() - lastPhysicalPress < 1500) return;  // real press
    MyroLutron.tapButton(BRIDGE_ID, KEYPAD_ID, BUTTON);
    MyroAPI.turnOff(OMNI_UNIT);
});
```

Pattern 10: Controller Down While the Mac Is Fine

The controller's Ethernet port can fail while everything else on the network is healthy. This tells the two apart and acts on it: it publishes the link state over MQTT for a dashboard, and pages you only when the evidence points at the controller rather than the network.

```
ScriptEvents.clear();

var lastCRC = MyroSystem.getEthernetQuality().crcErrors;

function report(reason) {
    var q = MyroSystem.getEthernetQuality();
    // "The Mac is fine": it has a network path, and if any Lutron
    // bridge is configured, at least one of them is connected.
    var macFine = q.macNetworkUp
        && (!q.othersConfigured || q.othersHealthy);

    MyroMQTT.publish("myrohome/controller/link", JSON.stringify({
        reason: reason,
        connected: q.connected,
        macNetworkUp: q.macNetworkUp,
        othersHealthy: q.othersHealthy,
        crcErrors: q.crcErrors,
        secondsSinceLastMessage: q.secondsSinceLastMessage
    }));

    if (!q.connected && macFine) {
        MyroNotifications.send("Controller unreachable but this Mac "
            + "is fine. Check the controller, its cable and its port.");
    } else if (!q.connected) {
        MyroSystem.log("Controller unreachable; network looks down too.");
    }

    if (q.crcErrors > lastCRC) {
        MyroSystem.log((q.crcErrors - lastCRC) + " new CRC errors.");
        lastCRC = q.crcErrors;
    }
}

ScriptEvents.onConnectionChange(function(isConnected) {
    // Give a reconnect 30 seconds to land before judging, so a
    // blip that heals itself does not page anyone.
    setTimeout(function() {
        report(isConnected ? "reconnected" : "disconnected");
    }, 30000);
});

ScriptEvents.onEthernetQuality(function(event, q) {
    if (event !== "baseline") { report(event); }
});
```

Why the wait: the Helper reconnects on its own after most drops. Reporting immediately turns every blip into an alert. Waiting lets the reconnect land and reports only what is still wrong.

HTTP controller control

`POST /helper/{controllerID}/control`

This is the preferred write endpoint. Each action is range checked and mapped to a documented Omni command. Arbitrary command bytes are not accepted.

Successful response:

```
{"success":true,"action":"set_unit","accepted":true}
```

Failure to receive the matching controller ACK returns `503` with `success:false` and `accepted:false`.

Units and Blinds

```
{"action":"set_unit","number":35,"value":75,"timer":0}
```

```
{"action":"set_blind","number":4,"value":50,"timer":0}
```

`value` is 0 through 100. Optional `timer` is 0 through 99 seconds.

Flags and Outputs

```
{"action":"set_flag","number":2,"value":10,"timer":0}
```

Flag `value` is 0 through 255. Optional `timer` is 0 through 99.

```
{"action":"set_output","number":3,"on":true,"timer":30}
```

Native Relative Controls (4.50)

Requires an updated Helper 4.50 build with these actions; earlier builds may reject them as unsupported. Do not substitute a cached-value write or retry.

Use the same authenticated `POST /helper/{controllerID}/control` endpoint:

```
{"action":"increment_flag","number":42}
```

```
{"action":"decrement_flag","number":42}
```

```
{"action":"dim_unit","number":35,"steps":1}
```

```
{"action":"brighten_unit","number":35,"steps":2}
```

`number` must be a JSON integer from 1 to 65535 identifying a configured Flag or Unit of the corresponding type on this controller. Query `/objects` first; blinds, outputs and flags are not dim/brighten targets. `steps` is required for dim/brighten and must be a JSON integer from 1 to 9. Booleans, strings, fractions, missing values, unconfigured targets and unsupported fields return `400` without issuing a command. These actions do not accept `timer`, `value` or `amount`; a counter command always requests one step.

The panel applies the change to its current value. Helper neither reads a cached level/count to calculate a replacement nor predicts the result in its cache. Repeated requests remain separate and ordered. Native lighting steps are not percentage points; their effect and support depend on the lighting technology and panel. Counter behavior at 0/255 is panel-defined, not specified by the protocol command table. Flags may trigger controller programs.

`accepted:true` means the panel acknowledged the command, not that the requested physical change or a particular counter value has been observed. An ACK timeout is uncertain: the command might have applied. Do not retry automatically. Read reported state before deciding what to do next. Offline, NAK and ACK-timeout responses use the existing `503` failure contract. Scripts expose the same commands as `incrementFlag`, `decrementFlag`, `dim` and `brighten`, but their Boolean return reports queue submission rather than waiting for an ACK.

Buttons, UPB Links, and Messages

```
{"action":"run_button","number":12}
```

```
{"action":"set_upb_link","number":8,"state":"on"}
```

UPB Link states are `on`, `off`, or `store`.

Helper 5.01 and later can blink a UPB device to find it, for 30 seconds at a one-second rate, or stop it early. Only units whose unit type is UPB are accepted (numbers 1-511); other units answer `400`.

```
{"action":"blink_upb","number":12}
```

```
{"action":"blink_upb","number":12,"operation":"stop"}
```

`operation` is `start` (the default) or `stop`.

```
{"action":"message","operation":"show","number":4,"presentation":"beep_led"}
```

```
{"action":"message","operation":"clear","number":4}
```

Presentation is `beep_led`, `led`, or `silent`.

Helper 5.01 and later can also say a message, log it, or phone a stored number (1-8) and say it:

```
{"action":"message","operation":"say","number":4}
```

```
{"action":"message","operation":"phone","number":4,"phone":1}
```

`operation` `log` writes the message to the controller's event log.

Keypad beepers (all keypads): `operation` is `on`, `off` or `stop` (silence them now).

```
{"action":"keypad_beepers","operation":"off"}
```

RadioRA 2 phantom buttons and Leviton scenes are addressed by their PC Access numbers (1-255). Leviton scene states are `on`, `off` or `set`.

```
{"action":"radiora2_phantom","number":4}
```

```
{"action":"leviton_scene","number":7,"state":"on"}
```

Locks

```
{"action":"set_lock","number":1,"locked":true}
```

`locked: true` sends the Omni lock command and `false` the unlock command. `number` 0 locks or unlocks every door. When unlocking, `seconds` (1 to 64800) unlocks for that long, then the controller locks the door again (Helper 5.01 and later):

```
{"action":"set_lock","number":1,"locked":false,"seconds":30}
```

Confirm the result from the lock's `locked` field. The reader fields on the same object describe credential activity, not whether the door is locked.

Thermostats

```
{"action":"set_thermostat","number":2,"operation":"mode","value":"auto"}
```

Supported operations:

Operation	Value
`heat_setpoint`	Numeric temperature

Operation	Value
`cool_setpoint`	Numeric temperature
`humidify_setpoint`	Numeric percentage
`dehumidify_setpoint`	Numeric percentage
`mode`	`off`, `heat`, `cool`, `auto`, `emergency`, or 0 through 4
`fan`	`auto`, `on`, 0, or 1
`hold`	Boolean (off or hold; vacation hold cannot be set)

Setpoint example:

```
{"action":"set_thermostat","number":2,"operation":"cool_setpoint","value":76}
```

Audio Zones

```
{"action":"set_audio_zone","zone":1,"operation":"volume","value":40}
```

Operations are `power` (boolean), `mute` (boolean), `volume` (0-100), `source` (1-255), `key` (1-40), and `transport`. Russound transport values are `play`, `pause`, `stop`, `next`, `previous`, `volup`, and `voldown`.

User Settings, Area Lights, and Energy Cost

```
{"action":"set_user_setting","number":4,"value":20}
```

```
{"action":"set_area_lights","area":1,"on":false}
```

Area `0` addresses all areas.

```
{"action":"set_energy_cost","level":"high"}
```

Energy levels are `low`, `medium`, `high`, `critical`, or 0 through 3.

HTTP security control

`POST /helper/{controllerID}/security`

Security requests require **an API key created with arm/disarm enabled**. Arming and disarming also require a panel user code in the JSON body: four digits. The Helper validates the code directly with the panel, refuses the duress user, and sends only the resulting Omni user number with the command.

Zone bypass and restore accept a code but do not require one. With a code, the command runs as that user and the panel's event log names them. Without one, it runs as the panel's Programmed user, the same identity the Helper uses when HomeKit or a script arms the house, so a script or an assistant can bypass a zone without a panel code being stored in it. Either way the code is checked against the zone's own area. Security operations are serialized because validation replies do not carry a client request ID.

Security control is off by default. A key's capability is fixed when the key is generated in the Helper's API tab, and cannot be changed afterwards: to move a controller from standard to security control, generate a new key. A key without the capability receives `403` and the attempt is logged.

The arm/disarm capability protects this direct, code-validated route. It does not classify JavaScript source. Remote script changes use the separate script-management capability because arbitrary JavaScript cannot be secured with a method-name denylist. A script-management key can create and run security-capable scripts and must therefore be treated as highly privileged. Scripts written in the Helper's own editor are local control by a person at the Mac.

Five rejected codes lock the route for fifteen minutes and return `429` with a `retryAfter` value in seconds. A rejected code counts toward that limit; a validation timeout or a disconnected controller does not. Every attempt, accepted or refused, is written to the Helper log with the address it came from.

Do not store a panel code in source control. Ask for it at runtime and send it only over a trusted local network.

Set Security Mode

```
{
  "action": "set_mode",
  "area": 1,
  "mode": "away",
  "code": "1234"
}
```

Modes are `off`, `disarm`, `day`, `night`, `away`, `vacation`, `day_instant`, `night_delayed`, or 0 through 6.

Bypass or Restore a Zone

```
{
  "action": "set_zone_bypass",
  "area": 1,
  "zone": 10,
  "bypassed": true,
  "code": "1234"
}
```

```
{
  "action": "restore_all_zones",
  "area": 1,
  "code": "1234"
}
```

Without a code, as the Programmed user:

```
{
  "action": "set_zone_bypass",
  "zone": 10,
  "bypassed": false
}
```

The panel will not restore a bypassed zone that is not ready or in trouble, whatever code is sent — the zone has to be closed or cleared first. When a restore is refused for that reason, the reply adds a `reason` and a plain-language `hint`; `error` is unchanged:

```
{
  "success": false,
  "action": "set_zone_bypass",
  "accepted": false,
  "error": "NAK from controller",
  "reason": "zone_not_ready",
  "hint": "Sump Pump is not ready, so the panel will not restore it. Close or clear the zone, then
       restore it."
}
```

Bypassing is not affected: the panel accepts a bypass of a zone that is not ready.

HTTP audio sources and metadata

List Sources

`GET /helper/{controllerID}/audio/sources`

Returns configured source names plus cached metadata:

```
{
  "success": true,
  "count": 2,
  "sources": [
    {"number":1,"name":"Streamer"},
    {"number":2,"name":"Tuner"}
  ],
  "metadata": []
}
```

Read or Refresh One Source

`GET /helper/{controllerID}/audio/source/{number}/metadata`

Add `?refresh=true` to request all documented metadata positions from the controller before returning the current cache. The panel response is asynchronous; `refreshRequested:true` means the request was sent, not that all new fields are already present in this response.

```
{
  "success": true,
  "source": 1,
  "refreshRequested": true,
  "count": 2,
  "metadata": [
    {"sourceNumber":1,"position":1,"fieldID":1,"value":"Artist"},
    {"sourceNumber":1,"position":2,"fieldID":2,"value":"Track"}
  ]
}
```

Field meanings vary by attached audio system. Use `position` and `fieldID` rather than assuming every system reports the same labels.

Legacy Audio Route

`GET /helper/{controllerID}/audio/{zone}/{cmd}/{value}`

Commands are `power`, `mute`, `volume`, `source`, `key`, and `transport`. New integrations should use the typed control endpoint.

Legacy HAI HTTP routes

These GET routes remain for existing network resources and simple integrations. New code should prefer `POST /control`.

Unit

`GET /helper/{controllerID}/unit/{number}/{value}`

`value` is 0 for off or 1 through 100 for on/dim level.

Flag

`GET /helper/{controllerID}/flag/{number}/{value}`

`value` is 0 through 255.

Output

`GET /helper/{controllerID}/output/{number}/{value}`

`value` is 0 or 1.

Energy Cost

`GET /helper/{controllerID}/controller/setEnergyCost/{level}`

HAI status example

`GET /helper/{controllerID}/status`

The response is a coherent snapshot copied from the Helper's controller store plus live connection-quality state.

```
{
  "success": true,
  "controller": {
    "controllerID": "953d8997-ab41-4aec-9f58-78235e36d46f",
    "name": "Grouse",
    "address": "192.168.1.10",
    "port": 4369,
    "protocol": "OmniLink II - TCP/IP",
    "connected": true,
    "status": "Connected",
    "model": "OmniPro II",
    "modelCode": 16,
    "firmware": "4.0B",
    "features": "Russound",
    "hasAudio": true,
    "temperatureFormat": "F",
    "dateTime": "Sep 3, 2026 10:12 AM",
    "sunrise": "6:30 AM",
    "sunset": "7:34 PM",
    "battery": {"status": 1, "reading": 229, "percent": 100},
    "acPowerOK": true,
    "phoneLine": {"code": 3, "status": "ok"},
    "troubles": [],
    "objectCounts": {"Zone": 48, "Unit": 64, "Thermostat": 2},
    "ethernetQuality": {
      "connected": true,
      "secondsSinceLastMessage": 12,
      "crcErrors": 0,
      "linkDrops": 0,
      "halfOpen": 0,
      "connectionFailures": 0,
      "ackTimeouts": 2,
      "macNetworkKnown": true,
      "macNetworkUp": true,
      "physicalInterfaces": 1,
      "othersConfigured": true,
      "othersHealthy": true
    }
  }
}
```


Additional status fields include `digitalCommunicatorStatus`, `fuseStatus`, `freezeStatus`, `energyCost`, and `sync`. The `sync` object contains ISO-8601 timestamps for `systemStatus`, `troubles`, `names`, `objectProperties`, `extendedSetup`, and `events`. A missing timestamp is JSON `null`.

Ethernet ACK timeout meaning

`ackTimeouts` is cumulative since the Helper launched. An ACK timeout means a command that expected an Omni ACK or NAK did not receive its matching reply within five seconds. It is a missed confirmation, not proof that the command failed, the controller disconnected, or the panel Ethernet hardware is defective.

Use the graph endpoint's historical `assessment` when you need a recurrence level. Do not diagnose a panel from one timeout or from a lifetime total.

HAI object example

`GET /helper/{controllerID}/objects`

Optional query: `?type=Zone`. Type matching is case-insensitive. Omit it to return all objects.


```
{
  "success": true,
  "count": 1,
  "objects": [
    {
      "number": 10,
      "type": "Zone",
      "state": 16,
      "value": 137,
      "status": "ready",
      "details": {
        "raw": 16,
        "condition": "secure",
        "latched": "secure",
        "arming": "armed",
        "armed": true,
        "bypassed": false,
        "troubleUnacknowledged": false
      },
      "loop": 137,
      "options": 0,
      "customName": "Front Door",
      "updatedAt": "2026-09-03T17:12:02Z"
    }
  ]
}
```


Supported type names include `Zone`, `Unit`, `Button`, `Area`, `Thermostat`, `Message`, `User Setting`, `Access Control`, `Access Control Reader`, `Access Control Lock`, `Auxiliary Sensor`, `Flag`, `Output`, `Audio Zone`, `Audio Source`, `UPB Link`, and `Blind`.

Object Fields

Every object can include `number`, `type`, `state`, `value`, `area`, `customName`, `kind`, `hidden`, `room`, and `updatedAt`. Type-specific decoded fields are:

Type	Decoded fields
Zone	`status`, `details`, `loop`, `options`
Unit / Blind	`on`, `level`
Flag	`on`, `counter`
Thermostat	`temperature`, `heatSetpoint`, `coolSetpoint`, `humidity`, `mode`, `fanMode`, `hold`, `holdMode`, `actionMode`, `communicationFailure`, `freezeAlarm`, `humidifySetpoint`, `dehumidifySetpoint`
Auxiliary Sensor	`reading`, `highSetpoint`, `lowSetpoint`
Audio Zone	`power`, `volume`, `source`, `mute`
Area	`mode`, `modeName`, `alarm`, `alarmName`, `entryTimer`, `exitTimer`
Message	`active`
User Setting	`settingValue`
Access Control / Lock	`locked`, `lastAccessGranted`, `lastAccessUser`, `lastAccessAt`
Access Control Reader	`granted`, `lastUser`, `lastAccessAt`

The raw `state` and `value` remain available even when a decoded field is present.

Thermostat hold

`hold` is 0 or 1, as before. `holdMode` (Helper 5.01 and later) keeps the panel's distinction: `"off"`, `"hold"` or `"vacation"` (the panel's raw hold value 2). Vacation hold is set at the thermostat or panel; `setThermostatHold` and the `hold` control operation only send off or hold. Until the first thermostat status arrives after Helper starts, `holdMode` is derived from `hold` and cannot yet say `"vacation"`.

Access control status

An access-control reader and the lock it releases share one object number. The lock reports whether it is locked. The reader reports whether the panel granted or denied the last access and which user number it was; the panel pushes this on every badge or keypad code at that reader. Helper also requests reader status together with lock status whenever it refreshes statuses at startup, on reconnect or through `refreshStatus()`, so the fields are present from the start instead of only after the first access. Reader replies do not count toward refresh completion and do not fire `onAccessControlEvent`.

Field	Object	Meaning
`locked`	Access Control / Lock	`true` when the lock reports locked

Field	Object	Meaning
`lastAccessGranted`	Access Control / Lock	`true` granted, `false` denied; present once the reader has reported
`lastAccessUser`	Access Control / Lock	Omni user number of the last access; present once the reader has reported
`granted`	Access Control Reader	Same as `lastAccessGranted`
`lastUser`	Access Control Reader	Same as `lastAccessUser`
`lastAccessAt`	Both	ISO 8601 time Helper received the reader status

User numbers are the panel's code numbers, not codes. 0 means no user; 251 is Duress, 252 Keyswitch, 253 Quick Arm, 254 PC Access and 255 Programming. Helper never exposes a user code.

The panel does not timestamp reader activity. `lastAccessAt` is when Helper last received the reader's status, so treat it as "last reported": a status refresh can report an access that happened earlier. It is absent until Helper has received a status from that reader since it launched.

Zone status

A zone's `state` is the panel's Omni-Link II zone status byte. Bits 0-1 are the current condition (secure, not ready, trouble), bits 2-3 the latched alarm (secure, tripped, reset but previously tripped), bits 4-5 the arming status (disarmed, armed, bypassed by user, bypassed by system), and bit 6 an unacknowledged trouble.

`status` is a one-word summary, in priority order: `tripped`, `trouble` or `tamper`, `bypassed`, `notReady`, `ready`. An armed zone that is secure is still `ready`; `details.armed` says it is armed. `details` decodes every field: `condition` (`secure`, `notReady`, `trouble`, `tamper`), `latched` (`secure`, `tripped`, `reset`), `arming` (`disarmed`, `armed`, `bypassedByUser`, `bypassedBySystem`), and the booleans `armed`, `bypassed` and `troubleUnacknowledged`. Read `details.bypassed` rather than `status` to tell whether a zone is bypassed: a bypassed zone that is in trouble or tripped reports that in `status`.

`status` is `unknown` only when the byte is not a valid zone status. A `state` of `-1` means the Helper has not yet heard this zone's status from the panel since it launched.