

Physics Systems Platform

Script API

Platform Version 4.4.0-alpha6

Robert M. DeLuca

support@physicssystems.com

Last Modified: 27-Jul-2017

NOTE: All examples assume the server is running under a Java 8 or greater runtime environment. Java 7 and below is no longer supported.

TABLE OF CONTENTS

Scripting Overview.....	7
Client Side Scripting.....	7
Server Side Scripting.....	7
Synchronous vs. Asynchronous.....	8
Advanced Example: Running a Command Inside a Thread (Server Side).....	8
Printing Messages To Remote Builder's Console Log.....	9
Example: Printing To Remote Builder's Console Log (Server Side).....	9
Exception Handling.....	10
Server Side Exceptions.....	10
Example: Exception Handling in Server Side Scripts.....	10
Client Side Exceptions.....	10
Example: Checking for Exceptions in Client Side Scripts.....	11
The Exception Object.....	11
getType().....	11
getMessage().....	11
Exception Types.....	12
Access Denied.....	12
Command Failed.....	12
Command Not Supported.....	12
Does Not Exist.....	12
Invalid Command.....	12
Invalid Destination.....	12
Timed Out.....	12
Basic API Commands.....	13
API.getUsername().....	13
API.wolWakeUp(physicalAddress).....	13
Alerts.....	14
How Alerts Reach Users.....	14
Alert Properties.....	15
Testing Alert Notifications.....	15
API.testNotifications().....	15
Temporarily Muting Alert Notifications.....	16
API.muteNotifications(duration).....	16
Generating Alerts.....	16
API.alert(typeName, deviceName, message).....	16
Example: Generating an Alert (Server Side).....	17
Example: Generating an Alert and Handling Errors (Client Side).....	17
The Stack.....	18
STACK.push(value).....	18
STACK.pop().....	18
STACK.peek().....	18

STACK.size().....	18
Example: Using the Stack (Server Side).....	18
Command Responses on the Stack.....	19
Example: Working With Command Responses (Server Side).....	19
Global Values.....	20
API.saveGlobalValue(name, value).....	20
API.loadGlobalValue(name).....	20
API.deleteGlobalValue(name).....	20
Example: Saving and Loading Global Values (Client Side).....	20
Example: Saving and Loading Global Values (Server Side).....	21
GPIO Commands.....	22
GPIO.getPinState(commanderName, pin).....	22
Example: Displaying the State of a GPIO Pin (Client Side).....	22
Example: Displaying the State of a GPIO Pin (Server Side).....	23
Relay Commands.....	24
Relay.turnOn(commanderName, relayNumber).....	24
Relay.turnOff(commanderName, relayNumber).....	24
Relay.toggle(commanderName, relayNumber).....	24
Relay.momentaryToggle(commanderName, relayNumber, duration).....	24
Relay.momentaryOnOff(commanderName, relayNumber, duration).....	24
Relay.momentaryOffOn(commanderName, relayNumber, duration).....	24
Relay.isRelayOn(commanderName, relayNumber).....	24
Timer Commands.....	25
Timer Ownership.....	25
Timer.start(timerName, unitOfTime, duration, macroName).....	25
Timer.cancel(timerName).....	25
Timer.increment(timerName, unitOfTime, amount).....	25
Timer.decrement(timerName, unitOfTime, amount).....	26
Timer.getTimeLeft(timerName, unitOfTime).....	26
Example: Displaying the Date and Time when a Timer Will Run (Client Side).....	26
Macro Commands.....	28
Macro.run(macroName).....	28
Client Side.....	28
Example: Message Passing to and from Macros (Client Side).....	28
Server Side.....	29
Macro.abort().....	29
Macro.restart().....	29
Example: Looping a Macro Three Times.....	29
Macro.skipNextStep().....	29
Component Property Commands.....	30
API.getComponentPropertyValue (componentName ,propertyName ,<responseCallbackFunction> ,<arguments...>).....	30
API.setComponentPropertyValue (componentName ,propertyName ,value	

,<responseCallbackFunction> ,<arguments...>.....	30
Example: Setting a Component Property Value (Client Side).....	30
Component Commands.....	31
API.runComponentCommand(componentName, commandName, arguments.....)	31
Example: Running Component Commands (Server Side).....	31
Example: Running Component Commands (Client Side).....	31
Matrix Commands.....	32
API.matrixGetDescription(matrixName).....	32
API.matrixSetCurrentSourceByName.....	33
(matrixName.....	33
,dstPortName.....	33
,srcPortName).....	33
API.matrixGetCurrentSourceName(matrixName, dstPortName).....	33
Example: Setting the current source port of a destination on the matrix.....	33
Example: Getting the current source port of a destination on the matrix (Server Side).....	33
Example: Getting the current source port of a destination on the matrix (Client Side).....	33
API.matrixGetCurrentDestinationNames.....	34
(matrixName.....	34
,srcPortName.....	34
,<responseCallbackFunction>.....	34
API.matrixConnectToDefaultVLANByName(matrixName, dstPortName).....	34
Internal APIs.....	35
Internal API Basics.....	36
Internal API.....	36
Internal Device.....	36
Internal Device API.....	36
Internal Device API Basics.....	37
getPropertiesJSON().....	37
Example: Using getPropertiesJSON() to do Device Capability Detection.....	38
Example: Using getPropertiesJSON() With Devices that Support Multiple APIs.....	39
Internal Device API: Thermostat.....	40
JSON Properties.....	40
API Commands.....	41
API.Thermostat.isModeAllowed(name, mode).....	41
API.Thermostat.getMode(name).....	41
API.Thermostat.setMode(name, mode).....	41
API.Thermostat.getScale(name).....	41
API.Thermostat.setScale(name, scale).....	41
API.Thermostat.getAmbientTemperature(name, scale).....	41
API.Thermostat.getHumidity(name).....	41
API.Thermostat.getTargetTemperature(name, scale).....	41
API.Thermostat.setTargetTemperature(name, temperature, scale).....	41
API.Thermostat.getTargetTemperatureHigh(name, scale).....	42

API.Thermostat.setTargetTemperatureHigh(name, temperature, scale).....	42
API.Thermostat.getTargetTemperatureLow(name, scale).....	42
API.Thermostat.setTargetTemperatureLow(name, temperature, scale).....	42
API.Thermostat.getEcoTemperatureHigh(name, scale).....	42
API.Thermostat.getEcoTemperatureLow(name, scale).....	42
Internal Device API: FireAlarm.....	43
JSON Properties.....	43
API Commands.....	43
API.FireAlarm.getBatteryState(name).....	43
API.FireAlarm.getAlarmState(name, alarmType).....	43
Internal Device API: Light.....	44
JSON Properties.....	44
API Commands.....	45
API.Light.isOn(name).....	45
API.Light.setOn(name, on).....	45
API.Light.getBrightness(name).....	45
API.Light.setBrightness(name, brightness).....	45
API.Light.getColorRGB(name).....	45
API.Light.setColorRGB(name, red, green, blue).....	45
API.Light.getColorTemperature(name).....	45
API.Light.setColorTemperature(name, temperature).....	45
Server Side Only Commands.....	46
API.runCommanderCommand(commanderName, commandName, arguments...).....	46
API.addDynamicResource(name, resourceBytes, resourceType).....	46
API.getDynamicResourceData(name).....	46
API.getDynamicResourceType(name).....	46
API.removeDynamicResource(name).....	46
API.sendEmail(recipient, subject, message).....	46
Client Side Only Commands.....	47
API.setLabelValue(labelScriptId, value).....	47
API.setLabelImage(labelScriptId, imageId).....	47
API.setLabelImageURL(labelScriptId, imageURL).....	47
API.setLabelVisible(labelScriptId, visible).....	47
API.setButtonVisible(buttonScriptId, visible).....	47
API.setButtonLabel(buttonScriptId, label).....	48
API.setButtonImages(buttonScriptId, upImageId, downImageId).....	48
API.setSliderVisible(sliderScriptId, visible).....	48
API.jumpToPage(pageScriptId).....	48
API.goBack().....	48
API.logout().....	48
Slider commands.....	49
API.Slider.setValue(sliderScriptId, value).....	49
API.Slider.assignProperty(sliderScriptId, apiName, deviceName, propertyName).....	49

Currently Assignable Internal Device API Properties.....	49
Example: using API.Slider.assignProperty().....	49
Dynamic Page Loading.....	50
API.loadPage(pageScriptID, handler).....	50
API.showPage(page, x, y, zIndex).....	50
Example: Loading and Displaying a Page Dynamically (Client Side).....	51
API.makePageBackgroundTransparent(page).....	51
API.hidePage(page).....	51
API.unloadPage(page).....	51

SCRIPTING OVERVIEW

The Physics Systems Platform is tremendously extensible. When Remote Builder's drag-and-drop GUI isn't enough, you also have the ability to create widgets and modules as well as script both the client (user interface) devices and the server. Widget creation is documented in the "Widget Specification". Modules are documented elsewhere. This manual covers the functionality provided by the platform API to client and server side scripts.

Server side and client side scripts are both written in JavaScript, but their execution environments are very different. They also serve different purposes:

Client Side Scripting

The primary purpose of client side scripting is to enhance the user interface. Client side scripts run directly on the device being used as a remote control (for instance, a tablet). They run within an HTML5 / CSS3 web browser environment and thus have access to the HTML DOM, AJAX and other HTML5 technologies. Some of the uses of client side scripts include:

- Performing animation
- Hiding and showing pieces of the interface
- Responding to gestures
- Updating the interface to display the current state of the system

In general, client side scripting should focus on the user interface and allow macros / server side scripts to handle everything else.

Server Side Scripting

The primary purpose of server side scripts is to decouple the user interface devices from the execution of complex or long running series of commands. Server side scripts run directly on the automation controller (Installation Server or Remote Builder) and thus are not subject to the same network and reliability issues that affect client side scripts.

Server side scripts exist as one or more steps within a macro. They have the ability to change their behavior according to previous steps as well as alter or abort the execution of future steps. They run within a Java virtual machine and have access to most of the Java 8 SE API, in addition to the standard ECMAScript functionality.

As such, they are a double-edged sword of power and danger. Although they are currently jailed so that they cannot access the controller's file system or run forever, a poorly written script still has the potential to render an Installation Server unresponsive for a period of time, possibly requiring a reboot. Complex server side scripts should be vetted with Remote Builder before being uploaded.

Synchronous vs. Asynchronous

In order for the user interface to remain interactive, *API commands called from a client side script run asynchronously*. When you use an API command on the client, it operates in the background while your script immediately continues running.

API commands called from a server side script are synchronous. Each command will block the script until it has completed (and returned a response, if one was expected). If you need asynchronous behavior in server side scripts, that can be achieved using threads (see below example).

Because of the synchronous/asynchronous distinction, API commands that return a response are used differently depending on where they are being called. Client side scripts use response callback functions to return their result whereas server side scripts return their result directly.

The differences are shown in detail by these two examples: [Example: Displaying the State of a GPIO Pin \(Client Side\)](#), [Example: Displaying the State of a GPIO Pin \(Server Side\)](#).

Advanced Example: Running a Command Inside a Thread (Server Side)

This example prints the message "ONE" to the console, (usually) followed by the message "TWO":

```
var thread = new java.lang.Thread(new java.lang.Runnable() {
    run: function() {
        java.lang.Thread.sleep(100);
        print("TWO");
    }
});

thread.start();
print("ONE");
```

Some important notes about threading on server side scripts:

- Threads are not guaranteed to run in any particular order. The above example, for instance, could actually print "TWO" before it prints "ONE"
- Long running threads run the risk of their execution context being re-used before the thread completes. You should not count on the global scope that existed before the thread was executed being the same as when it finally executes.
- Threads bypass the built-in script maximum allowed run time protection. In other words, the server will normally terminate a script that attempts to run for longer than five minutes. This detection does not operate on any script run within a separate thread, so threaded script has the potential to permanently tie up one or more CPUs on the server if they have a bug.

Printing Messages To Remote Builder's Console Log

Server side scripts can print messages to Remote Builder's console log (available in the Window menu) using the **print()** function. These print statements are safe to leave in code uploaded to the Installation Server.

Example: Printing To Remote Builder's Console Log (Server Side)

For example, the following server side script will display "Hello World!" in Remote Builder's console log when its parent macro is run:

```
print("Hello World!");
```

EXCEPTION HANDLING

The API commands throw various exception objects when errors occur. By responding to exceptions, you can create robust installations that handle failures gracefully instead of seemingly doing nothing or bombarding the user with error messages.

Exceptions are utilized differently depending on whether the script is server side or client side.

Server Side Exceptions

You can catch exceptions directly using a try/catch block. Thrown API exceptions are instances of **com.physicssystems.publicapi.InstallationServerException**.

Example: Exception Handling in Server Side Scripts

Here's an example that intentionally attempts to access a commander that doesn't exist:

```
print("Attempting to access an invalid commander");

try {
    GPIO.getPinState("Non Existent Commander",1);
} catch(e) {
    if(e instanceof com.physicssystems.publicapi.InstallationServerException) {
        print("EXCEPTION TYPE:      "+e.getType());
        print("EXCEPTION OCCURRED: "+e.getMessage());
    }
}
```

The output from this would be something similar to:

Attempting to access an invalid commander

EXCEPTION TYPE: Does Not Exist

EXCEPTION OCCURRED: commander Non Existent Commander does not exist

Client Side Exceptions

On the client side, you have to wait for the response callback function to be called in order to get the exception. Every API command can use a response callback function, even one-way commands that don't return a response (the server still returns "OK" for one-way commands). By checking the **isException** field of the response object, which is always the first argument passed to the callback function, you can tell if the response is an exception or not.

You are not required to use response callbacks. If you don't pass a callback function to an API command and an exception occurs, the user will be alerted with a detailed error message.

Example: Checking for Exceptions in Client Side Scripts

This example first inquires the current state of relay 1 on the commander named “Main Commander”. When the response callback (**callbackFunction**) is called, it will display the response. Then, the script attempts to turn on an invalid relay, passing the callback function in order to check for success or failure:

```
function callbackFunction(response) {
  if(response.isException) {
    alert("An error occurred: "+response);
    return;
  }
  alert("Command executed properly. Here's the response: "+response);
}
Relay.isRelayOn("Main Commander",1,callbackFunction);
// Attempt to turn on the 12,345th relay, which obviously doesn't exist
Relay.turnOn("Main Commander",12345,callbackFunction);
```

The Exception Object

On both the client and the server, exception objects provide these methods:

getType()

This returns a terse description of the exception such as “Access Denied” or “Command Failed”.

getMessage()

This returns the actual error message, which is usually a helpful explanation of the problem according to the server.

Exception Types

These are the possible values returned by getType():

Access Denied

The user does not have access to the resource.

Command Failed

The command failed for a reason that could not be classified under one of the other possible exceptions. This is usually thrown when hardware or network errors occur.

Command Not Supported

If this is being thrown then you are attempting to do something that is not supported. This should never occur during normal use.

Does Not Exist

The resource you are attempting to use does not exist. Generally, this would indicate whatever name you provided did not match anything.

Invalid Command

You are trying to execute an invalid command. This should not occur during normal use.

Invalid Destination

The port, pin or relay number you specified is invalid.

Timed Out

The device did not respond to the server in time.

BASIC API COMMANDS

Simple stuff.

API.getUsername()

Returns the username of the user running the current script. Note, macros run by the scheduler are always run by the user "Scheduler." Macros run by triggers are always run by the user "Trigger."

API.wolWakeUp(physicalAddress)

Attempts to wake up the network device with the physical (MAC) address **physicalAddress** according to the Wake-on-LAN (WoL) standard. WoL must be supported and properly configured on the target device in order for this command to work.

For convenience, the physical address may be specified with or without colons, dashes and spaces. The user invoking this command must have the "WoL" API privilege.

WoL "magic packets" are usually not routed to alternate subnets by default, so the server must be on the same subnet as the target device.

ALERTS

The alert system is used to inform users of a significant error or event, for instance, losing connection to a camera or an inability to contact PSNET. Once generated, alerts are logged into the alert history and have the potential to trigger one or more notifications, dependent upon each user's notification settings.

How Alerts Reach Users

Users can become aware of alerts via two methods: accessing the built-in alert history page (available at the top-level URI <http://<SERVERADDRESS>/alerts>) and notifications.

The built-in alerts page requires the user to periodically check it in order to discover new alerts whereas notifications attempt to contact the user in real-time to inform them of an alert.

It is important to note that good practice requires that both methods are utilized. Notifications are not guaranteed to be sent on each alert or reach the user for multiple reasons:

1. Notifications are rate-limited. A (customizable) period of time must elapse between the sending of two notifications to the same user. Rate limiting is done by alert type, by device name AND by each particular method of notification delivery (notification services such as Gmail, SMS and Apple Push Notifications). Rate limiting is done by alert type and device name to prevent hogging of the available notifications (say if a particular camera keeps failing), and rate limiting of each notification delivery method is done to prevent their respective services from banning the notification account.
2. Notifications will fail to send if the server does not have an active connection to the Internet. Notifications DO NOT queue up and wait for delivery to be successful! If a notification fails to send, there is no retry. This is to prevent flooding of notification services that will result in temporary (or permanent) bans.
3. Notifications depend upon the reliability of whatever service they utilize. For instance, if Gmail is down or overload at the time of a notification that uses Gmail, the notification will fail to send. Likewise, Google may enforce a policy change at any time that blocks sending of all emails for a certain (or perhaps indefinite) period of time.

Notifications are sent directly from the originating machine that the alert was generated on - they do not go through PSNET first. This is done to reduce the number of potential failure points when sending notifications. For this reason, some caution must be utilized if re-using the same notification service account on multiple servers because each server is not aware of the other's notifications and can not properly enforce an account-wide rate limit.

Alert Properties

Alerts have the following properties:

- **Timestamp** – when the alert was generated. This is handled automatically by the system.
- **Subsystem** – Installation Server has multiple subsystems such as the NVR, Scheduler and PSNET. Alerts indicate which subsystem they originated from. Alerts generated by the API (see below) always have a subsystem of "CUSTOM." This is done so that internally generated alerts by the system are never confused with custom alerts.
- **Type Name** – in addition to the subsystem, alerts are categorized by their type name (e.g., "Camera Down", "I/O Error" or "Connect Failed"). **The subsystem and type name combined indicate the alert type.** This is necessary because some type names may be present in multiple subsystems (such as "I/O Error" – I/O error of what?)
- **Device Name** – some alert types are device specific (e.g., "Camera Down"). When those alerts are generated they will include the name of the device that triggered them.
- **Message** – includes additional details about what triggered the alert. For instance, a "Camera Down" alert might indicate that a read timeout occurred after a period of time. A "Camera Connection Failed" alert might tell the user that the camera couldn't be contacted at its current IP address, or is refusing to operate as expected by the system.

Testing Alert Notifications

The alert notification system can be tested at any time on both client and server side with this command:

```
API.testNotifications()
```

This generates a test alert that will be added to the alert history. All users who are configured to receive alert notifications on the current system will receive a test notification via each enabled delivery method.

NOTE: the alert system is rate limited to prevent notifications for system test alerts from being sent more often than once per 60 seconds. Attempts to initiate another test before this time has elapsed will not trigger notifications. This rate limiting IGNORES each user's individual rate limits for alert type and device. Delivery method rate limits ARE taken into account however.

The user calling this function must have the "Alerts → API" privilege or an **AccessDeniedException** will be thrown.

Temporarily Muting Alert Notifications

Sometimes it is helpful to be able to prevent notifications from being generated by alerts on a live system. To do this you may use the following command both client and server side:

```
API.muteNotifications(duration)
```

This will mute notifications for **duration** seconds. Any alerts generated while the mute is in effect will not trigger notifications. Durations of 0 or less will immediately un-mute notifications.

NOTE: muting will NOT prevent notifications from being generated by system tests i.e. via **API.testNotifications()**.

On success, "OK" is returned server side, and client side the first argument of the handler function will have the value "OK". On failure, a **CommandFailedException** is thrown (server side) or is passed as the first argument of the response handler function (client side).

The user calling this function must have the "Alerts → API" privilege or an **AccessDeniedException** will be thrown.

Generating Alerts

Alerts can be generated both client and server side with the same command:

```
API.alert(typeName, deviceName, message)
```

This generates an alert with type name **typeName**, device name of **deviceName** and with message **message**. If the alert should not specify a device, use **null** for **deviceName**.

On success, "OK" is returned server side, and client side the first argument of the handler function will have the value "OK". On failure, a **CommandFailedException** is thrown (server side) or passed as the first argument to the response handler function (client side). Because of the importance of alerts, failure is not normal on a properly functioning system, and indicates either an inability to contact the server (client side) or that the alert system is so overloaded with alerts it can not queue up any more. This is usually indicative of catastrophic failure.

Success does not indicate a notification was or will be sent!

The user must have the "Alerts → API" privilege or an **AccessDeniedException** will be thrown.

Example: Generating an Alert (Server Side)

The following snippet will generate an alert with no device, and then another one with a device. Depending on the rate limit settings, only one (or neither) will trigger a notification:

```
API.alert("Test Alert", null, "This is the message");
API.alert("Test Alert", "Test Device", "This is the message");
```

Example: Generating an Alert and Handling Errors (Client Side)

The previous server side example will work fine on the client side too, and will suffice for most purposes. If you want to know if the alert was generated successfully on the system, the following snippet will work:

```
API.alert
("Test Alert"
,"Test Device"
,"This is the message"
,function(response,typeName,deviceName,message) {
    if(response.isException) {
        console.log("Failed to generate alert: "+response);
        console.log(" Alert type name: "+typeName);
        console.log(" Alert device name: "+deviceName);
        console.log(" Alert message: "+message);
        return;
    }

    if(response === "OK") {
        console.log("Alert generated successfully");
        return;
    }

    // neither an exception was returned nor "OK", something is wonky!
    console.log("Unexpected response to API.alert(): "+response);
});
```

THE STACK

Server side scripts have access to a stack variable called STACK that can be used to pass values back and forth between steps. The contents of the stack are preserved during the entire execution of a macro (including nested macros).

STACK.push(value)

Pushes the given value onto the top of the stack.

STACK.pop()

Removes the topmost value from the stack and returns it.

STACK.peek()

Returns the value at the top of the stack but doesn't remove it.

STACK.size()

Returns how many elements are on the stack.

Example: Using the Stack (Server Side)

This simple example passes some numbers from one script to the next. The first script pushes the numbers 2 and 50 onto the stack. The second script pops the stack and checks to see if the value is 50 and prints a message indicating success:

Script 1:

```
STACK.push(2);  
STACK.push(50);
```

Script 2:

```
if(STACK.pop() == 50)  
    print("Yes, the number 50 was on the top!");  
else  
    print("No, the number 50 wasn't on the top!");
```

Command Responses on the Stack

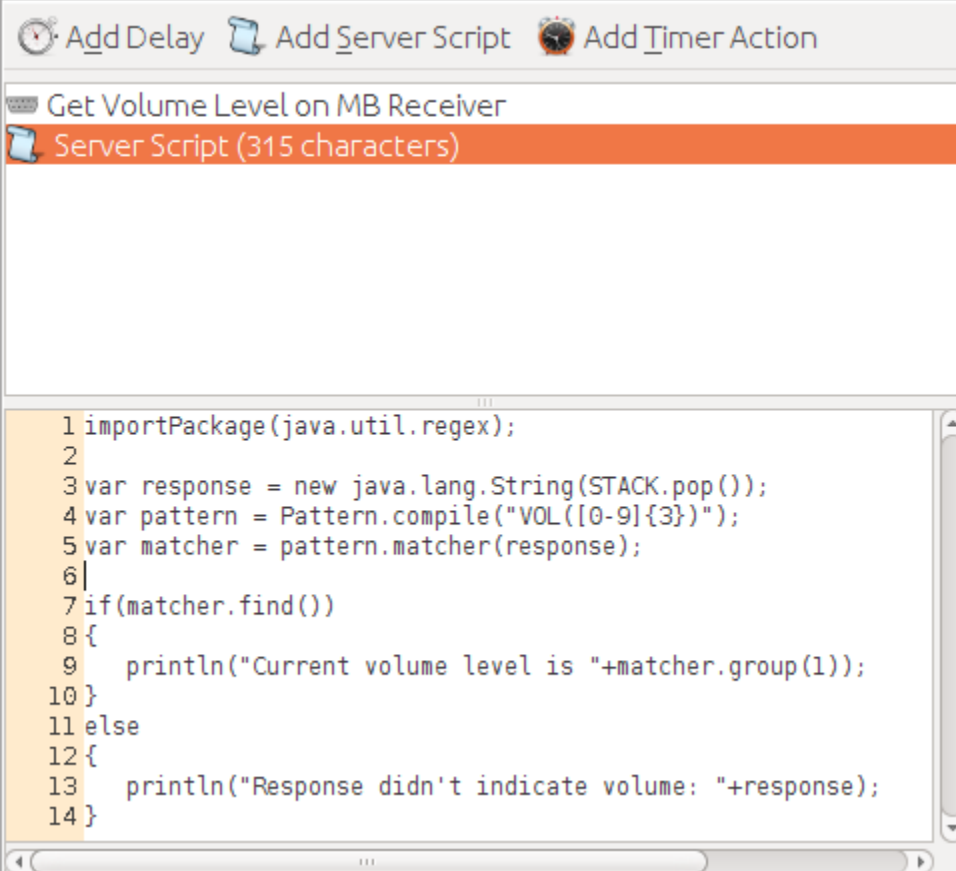
Component command actions that are set to return a response will push the response as an array of bytes onto the stack. This allows you to use the GUI to drag-and-drop component commands into a macro and then use scripts to process their responses. If the response is supposed to be an ASCII string, it can be converted from an array of Java bytes to a Java String like this:

```
var response = new java.lang.String(STACK.pop());
```

Now the variable **response** can be compared to or concatenated with a normal JavaScript string.

Example: Working With Command Responses (Server Side)

This example prints the current volume level of a Pioneer receiver. It uses Java pattern matching to look for and retrieve it from the response to the previous command ("?V"). The receiver returns the volume level as "VOLXXX" (without the quotes), where XXX is a number. For example, "VOL074".



```
1 importPackage(java.util.regex);
2
3 var response = new java.lang.String(STACK.pop());
4 var pattern = Pattern.compile("VOL([0-9]{3})");
5 var matcher = pattern.matcher(response);
6
7 if(matcher.find())
8 {
9     println("Current volume level is "+matcher.group(1));
10 }
11 else
12 {
13     println("Response didn't indicate volume: "+response);
14 }
```

GLOBAL VALUES

The script API provides a way to permanently store small, globally accessible values on the server. These values will persist through reboots of the installation server and are not removed when an installation XML is uploaded. They are not stored inside the installation XML, so any values saved in your development environment (i.e., when hosted from Remote Builder), will not carry to the installation server from an upload.

The API is capable of properly saving and loading booleans, numbers and strings. Any other types (such as a Date) must be converted to one of the three basic types in order to be stored properly. String values larger than 4096 characters will be truncated when saved.

Global values are not unique per user. Any user who has access to global values can overwrite, load and delete global values set by any other user. Global values are available in platform versions 1.1 and above. The following API commands are available on both the client and server:

API.saveGlobalValue(name, value)

Saves the given **value** under the name **name**, where **name** is a string and **value** is either a *boolean*, a *number* or a *string*. Throws a **CommandFailedException** if an error occurred while saving the value. On the client-side, the response value returned to the handler will be "OK" on success.

API.loadGlobalValue(name)

Retrieves the given global value named **name**. Returns **null** if no value is saved under that name.

API.deleteGlobalValue(name)

Deletes the global value named **name**. Afterwards, any calls to **loadGlobalValue()** with that name will return null until a new value is saved. It is your responsibility to delete global values you no longer need.

Example: Saving and Loading Global Values (Client Side)

The following client-side script implements two functions. **saveDate()** saves the current date and time into the global value "date", **loadDate()** displays the saved date:

```
function saveDate() {
    var date = new Date();

    API.saveGlobalValue
        ("date"
        ,date.getTime() // Convert the date into milliseconds
        ,function(response,name) {
            if(response == "OK") {
```

```
        alert("Saved date is: "+date);
        loadDate();
    }
});
}

function loadDate() {
    API.loadGlobalValue
    ("date"
    ,function(response,name) {
        var date = new Date();
        date.setTime(response);
        alert("Loaded date is: "+date);
    });
}
```

Example: Saving and Loading Global Values (Server Side)

Here is the server-side version of the previous script:

```
function saveDate() {
    var date = new Date();
    API.saveGlobalValue("date",date.getTime());
    print("Saved date is: "+date);
}

function loadDate() {
    var date = new Date();
    date.setTime(API.loadGlobalValue("date"));
    print("Loaded date is: "+date);
}
```

GPIO COMMANDS

These commands are available in both client and server side scripts.

GPIO.getState(**commanderName**, **pin**)

Reads and returns the current state of the given GPIO pin. This value will be 1, 0 or -1. -1 indicates the current state could not be determined.

Note that the way this value should be interpreted depends on what type of device the commander is. For instance, on Global Caché units, 1 could mean either the IR pin is being held high OR is disconnected, and 0 indicates the IR pin is definitely being held low.

This command returns a response, and thus is used differently on the client side vs. server side.

Example: Displaying the State of a GPIO Pin (Client Side)

The following client script will pop up a dialog box informing you what the current state of pin #2 is on commander "Main":

```
// This function will be called when the pin state is returned by
// the server. Note that the arguments passed to the original
// GPIO.getState() command are also passed to the callback function.
function pinStateCallback(response, commanderName, pin) {
  // Check if the response is an exception object
  if(response.isException) {
    alert("An error occurred: "+response);
    return;
  }

  switch(response)
  {
    case 1:
      alert("Pin #"+pin+" is high");
      break;
    case 0:
      alert("Pin #"+pin+" is low");
      break;
    default:
      alert("The state of pin #"+pin+" could not be determined");
      break;
  }
}

// Do the API command. Wrapped in a try/catch block so that script
// execution is not disabled if an error occurs.
try {
  GPIO.getState("Main (GC-100-12)", 5, pinStateCallback);
}
```

```
} catch(e) {  
    alert("Failed to connect to server: "+e);  
}
```

Example: Displaying the State of a GPIO Pin (Server Side)

The following example is similar to the client side one, but it prints the current pin state to Remote Builder's console log:

```
try {  
    switch(GPIO.getPinState("Main (GC-100-12)",5))  
    {  
        case 1:  
            print("Pin #5 is high");  
            break;  
        case 0:  
            print("Pin #5 is low");  
            break;  
        default:  
            print("The state of pin #5 could not be determined");  
            break;  
    }  
} catch(e) {  
    print("An error occurred: "+e);  
}
```

RELAY COMMANDS

All relay commands operate on a specific commander and relay. The commander is specified by name as a quoted string, case insensitive and whitespace is trimmed. The relay is specified by its number, with 1 being the first relay of the commander.

These commands are available in both client and server side scripts.

Relay.turnOn(commanderName, relayNumber)

Activates the relay (turns it ON).

For example, to activate relay #4 on commander "Master Bedroom":

```
Relay.turnOn("Master Bedroom", 4);
```

Relay.turnOff(commanderName, relayNumber)

Deactivates the relay (turns it OFF).

Relay.toggle(commanderName, relayNumber)

Toggles the state of the relay. If the relay was ON before, it will be OFF after this command, and vice versa.

Relay.momentaryToggle(commanderName, relayNumber, duration)

Toggles the relay, waits the specified duration and then toggles the relay back to its original state. At the conclusion of this command, the relay will be in the same state that it was before the command.

The duration is specified in milliseconds. Durations greater than 30000 (30 seconds) are not permitted.

Relay.momentaryOnOff(commanderName, relayNumber, duration)

Turns the relay ON, waits the specified duration and then turns the relay OFF. Duration is in milliseconds and may not be greater than 30 seconds.

Relay.momentaryOffOn(commanderName, relayNumber, duration)

Turns the relay OFF, waits the specified duration and then turns the relay ON. Duration is in milliseconds and may not be greater than 30 seconds.

Relay.isRelayOn(commanderName, relayNumber)

Returns a boolean true if the relay is definitely ON, false otherwise.

TIMER COMMANDS

Timers allow a macro to be run once at a later time. All timer commands refer to timers by an exact name of your choosing. When specifying a timer name in script, it must be a quoted string. Timer names are case insensitive and leading/trailing whitespace is trimmed.

Timer commands that indicate a duration or amount of time require the unit of time to be specified. Accepted values are `Timer.MILLISECONDS`, `Timer.SECONDS`, `Timer.MINUTES`, `Timer.HOURS` and `Timer.DAYS`.

These commands are available in both client and server side scripts.

Timer Ownership

For security purposes, timers belong to the user who started them. For instance, if user "Bob" starts a timer named "Set Alarm", user "Alice" cannot see or change Bob's timer. In fact, Alice is free to start her own timer named "Set Alarm", which will run independently of Bob's timer.

Timers started inside macros belong to the user who ran the macro. If the macro was run by the scheduler, no user can access the timer (but other macros run by the scheduler can).

Timer.start(timerName, unitOfTime, duration, macroName)

Schedules a timer to run after the given duration has elapsed. Durations must be whole numbers (no decimals). If a timer with the same name was already scheduled (and hasn't run yet), it will be replaced.

For example, to schedule a timer named "Sleep" that will run the macro named "System Off" in 30 seconds:

```
Timer.start("Sleep", Timer.SECONDS, 30, "System Off");
```

Timer.cancel(timerName)

Cancels the timer.

Timer.increment(timerName, unitOfTime, amount)

Adds the given amount of time to the timer. The new amount of time remaining will be returned.

The amount of time must be a whole number, but it may be negative (indicating time should be removed instead of added).

A timer that is decremented so far that its remaining time becomes negative will execute immediately.

Timer.decrement(timerName, unitOfTime, amount)

Removes the given amount of time from the timer. The new amount of time remaining will be returned.

A timer that is decremented to run in the past will execute immediately.

Timer.getTimeLeft(timerName, unitOfTime)

Returns the amount of time remaining before the given timer will execute its macro. If no timer with the given name is scheduled to run (or has already run), -1 will be returned.

Note that partial durations are not returned. For example, if there are 45 seconds left on a timer and you request the time left in MINUTES, 0 will be returned, not 1 or 0.75.

Example: Displaying the Date and Time when a Timer Will Run (Client Side)

The following client script will pop up a dialog box telling you approximately what time and date the timer "Sleep" will run:

```
function timerTimeLeftCallback(response,timerName,unitOfTime) {
  if(response.isException) {
    alert("An error occurred: "+response);
    return;
  }

  if(response == -1) {
    alert("Timer '"+timerName+"' is not set");
    return;
  }

  switch(unitOfTime) {
    case Timer.MILLISECONDS:
      break;
    case Timer.DAYS:
      response *= 24;
    case Timer.HOURS:
      response *= 60;
    case Timer.MINUTES:
      response *= 60;
    case Timer.SECONDS:
      response *= 1000;
      break;
  }

  var now    = new Date();
  var when  = new Date(now.getTime() + response);

  alert
    ( "Timer '"+timerName+"' will run on "
```

```
+ when.toLocaleDateString()
+ " at "
+ when.toLocaleTimeString() );
}

Timer.getTimeLeft("Sleep",Timer.SECONDS,timerTimeLeftCallback);
```

MACRO COMMANDS

Macro script commands function differently depending on whether they are called from server side or client side scripts.

Macro.run(*macroName*)

Runs the specified macro immediately. Macro names are quoted strings, case insensitive and whitespace is trimmed. The behavior of this command depends on whether it is called from a client or server side script:

Client Side

When called from a client script, this command will return immediately and the specified macro will execute independently. You can pre-populate the macro's stack by passing parameters to `run()`, but if you wish to do this, you must provide a value for the callback function (you can use **null** if you don't want to use a callback). The parameters you provide are pushed onto the stack in the order that you provide them.

When the macro has finished running, the top element of the stack is returned to the response callback function. If the stack is empty when the macro finishes, the message "OK" is returned instead.

Example: Message Passing to and from Macros (Client Side)

First, create a macro named "Multiply" that has this server side script:

```
var a = parseInt(STACK.pop());
var b = parseInt(STACK.pop());
STACK.push(a*b);
```

Now on the client side, this script will display an alert with the result of multiplying 35 times 27:

```
Macro.run
("Multiply"
, function(response) {
    alert("Answer is: "+response);
}
, 35, 27);
```

Server Side

If called from a server side script, `run()` will not return until the specified macro has finished. The sub macro has access to the same stack as the calling macro.

For example, to run the macro named "Turn Sprinklers On" from a server side script:

```
Macro.run("Turn Sprinklers On");
```

Macro.abort()

Terminates the current macro and attempts to terminate the currently running script. If the current macro is a sub macro (i.e., was called from another macro), only the sub macro will terminate.

This command is server side only. If not called within a try/catch block, it will terminate the currently running script (which is usually what you want).

Macro.restart()

Restarts the current macro from the beginning. If the current macro is a sub macro, only the sub macro will restart.

This command is server side only. If not called within a try/catch block, it will terminate the currently running script.

Example: Looping a Macro Three Times

The following server side script, if placed at the end of a macro, will cause the macro to repeat itself three times:

```
var count = 2;
if(STACK.size() != 0)
{
    count = STACK.pop();
    if(--count <= 0)
        Macro.abort();
}
STACK.push(count);
Macro.restart();
```

Macro.skipNextStep()

Skips the next step of the current macro. If there are no more steps left, the macro will end.

This command is server side only. If not called within a try/catch block, it will terminate the currently running script.

COMPONENT PROPERTY COMMANDS

Component properties reduce tedium and repetition by wrapping ordinary device commands with code to gather state, parse it and convert it to something human-readable. They also perform the reverse to facilitate setting state.

```
API.getComponentPropertyValue  
  (componentName  
  ,propertyName  
  ,<responseCallbackFunction>  
  ,<arguments...>)
```

Returns the current value of the property **propertyName**, of component **componentName**. If the underlying get command has arguments, they may be passed as the last arguments to `getComponentPropertyValue()`. **responseCallbackFunction** is for client side scripts only. Server side scripts return the value directly.

```
API.setComponentPropertyValue  
  (componentName  
  ,propertyName  
  ,value  
  ,<responseCallbackFunction>  
  ,<arguments...>)
```

Sets a component property to **value**. **responseCallbackFunction** is for client side scripts only.

Note that **value** must be in human-readable form. For enum properties, this means **value** must be a quoted string that EXACTLY matches one of the available labels of the property. For number properties, **value** must be a number within the valid range.

Example: Setting a Component Property Value (Client Side)

This example sets the Z-Wave thermostat (node id: 5) mode to "Cool" and pops up an alert with the server's response (which should just be "OK"):

```
API.setComponentPropertyValue  
  ("MB Z-Wave Controller"  
  ,"Thermostat Mode"  
  ,"Cool"  
  ,function(response) {  
    alert("Server responded: "+response);  
  }  
  ,5);
```

COMPONENT COMMANDS

These commands are available in both client and server side scripts.

API.runComponentCommand(componentName, commandName, arguments...)

Runs the command named **commandName** belonging to the component named **componentName**.

If the command uses a command generator, any arguments after **commandName** will be passed to the command's generator function (if it has one). Otherwise, any additional arguments will be ignored. Currently, up to five arguments are supported.

The return value of **runComponentCommand()** is an array of Java bytes. If the command is set to return a response, the bytes will be the exact device response (HTTP commands will only return the response body). Otherwise, the return value will contain the ASCII characters "OK".

Example: Running Component Commands (Server Side)

This example will instruct the component named "Receiver" to run the command named "Set Volume", passing the argument 50:

```
API.runComponentCommand("Receiver", "Set Volume", 50);
```

This example will retrieve the current status of video input 1 on the component "Kubincam" and print it to the console log:

```
var response = API.runComponentCommand
  ("Kubincam","Get Input Status",1);
print("Video input is :"+response);
```

Example: Running Component Commands (Client Side)

This example will instruct the component named "Doug's Showroom Russound" to execute the command "MM Select Menu Item" with parameters 4, 1 and 1. After the command has been executed and a response returned, the function `updateNavigation()` will be called with the response:

```
API.runComponentCommand
  ("Doug's Showroom Russound"
  ,"MM Select Menu Item"
  ,updateNavigation
  ,4,1,1);
```

MATRIX COMMANDS

The matrix commands are available both server and client side.

API.matrixGetDescription(matrixName)

Retrieves a description of the given matrix, specified by **matrixName**.

Server side, the return value will be a JSON string that you must parse with `JSON.parse()` to turn into an object. On the client side, the response will already be parsed for you before it is passed to your handler function. The response object will follow this template:

```
{
  "name": "Matrix Name",
  "id": 1,
  "srcPorts": [
    {
      "name": "Source Port 1 Name",
      "port": 12,
      "id": 1
    },
    {
      "name": "Source Port 2 Name",
      "port": 13,
      "id": 2
    }
  ],
  "dstPorts": [
    {
      "name": "Destination Port 1 Name",
      "port": 14,
      "id": 3
    },
    {
      "name": "Destination Port 2 Name",
      "port": 15,
      "id": 4
    }
  ]
}
```

The order that the ports appear in the object will match the order they appear in Remote Builder.

API.matrixSetCurrentSourceByName

```
(matrixName  
,dstPortName  
,srcPortName)
```

Tells the matrix named **matrixName** to connect the destination port named **dstPortName** to the source port named **srcPortName**.

API.matrixGetCurrentSourceName(matrixName, dstPortName)

Gets the name of the source port that the destination port named **dstPortName** is currently connected to, on the matrix named **matrixName**. If the source port could not be determined, the value "UNKNOWN" will be returned instead. This will normally only happen if the destination port is not connected to any source, or the matrix configuration is out of sync with the installation.

On server-side scripts, the return value will be the source port name. On client-side scripts, the source port name will be passed as the first argument to the handler function.

Example: Setting the current source port of a destination on the matrix

The following will connect the source port named "TiVo" to the destination port named "Master TV" on the matrix "Main":

```
API.matrixSetCurrentSourceByName("Main", "Master TV", "TiVo");
```

Example: Getting the current source port of a destination on the matrix (Server Side)

The following will display the current source of the "Master TV" destination port on the matrix "Main":

```
var port = API.matrixGetCurrentSourceName("Main", "Master TV");  
print("The current port is: "+port);
```

Example: Getting the current source port of a destination on the matrix (Client Side)

The following will print the current source of the port named "Master TV" on the matrix "Main" to the browser console log:

```
API.matrixGetCurrentSourceName  
("Main"  
, "Master TV")
```

```
,function(response,matrix,destination) {
  console.log
    ("Port "+destination
    +" on matrix "+matrix
    +" is currently connected to source port "+response);
});
```

API.matrixGetCurrentDestinationNames

```
(matrixName
,srcPortName
,<responseCallbackFunction>)
```

Returns an array of the names (as strings) of the destination ports on matrix **matrixName** that are currently connected to the source port named **srcPortName**. The array will be empty (i.e., its **length** property will === 0) if there are no destination ports currently connected to the given source port.

As always, the response is directly returned on server side scripts, and passed as the first argument to the **responseCallbackFunction** on client side scripts.

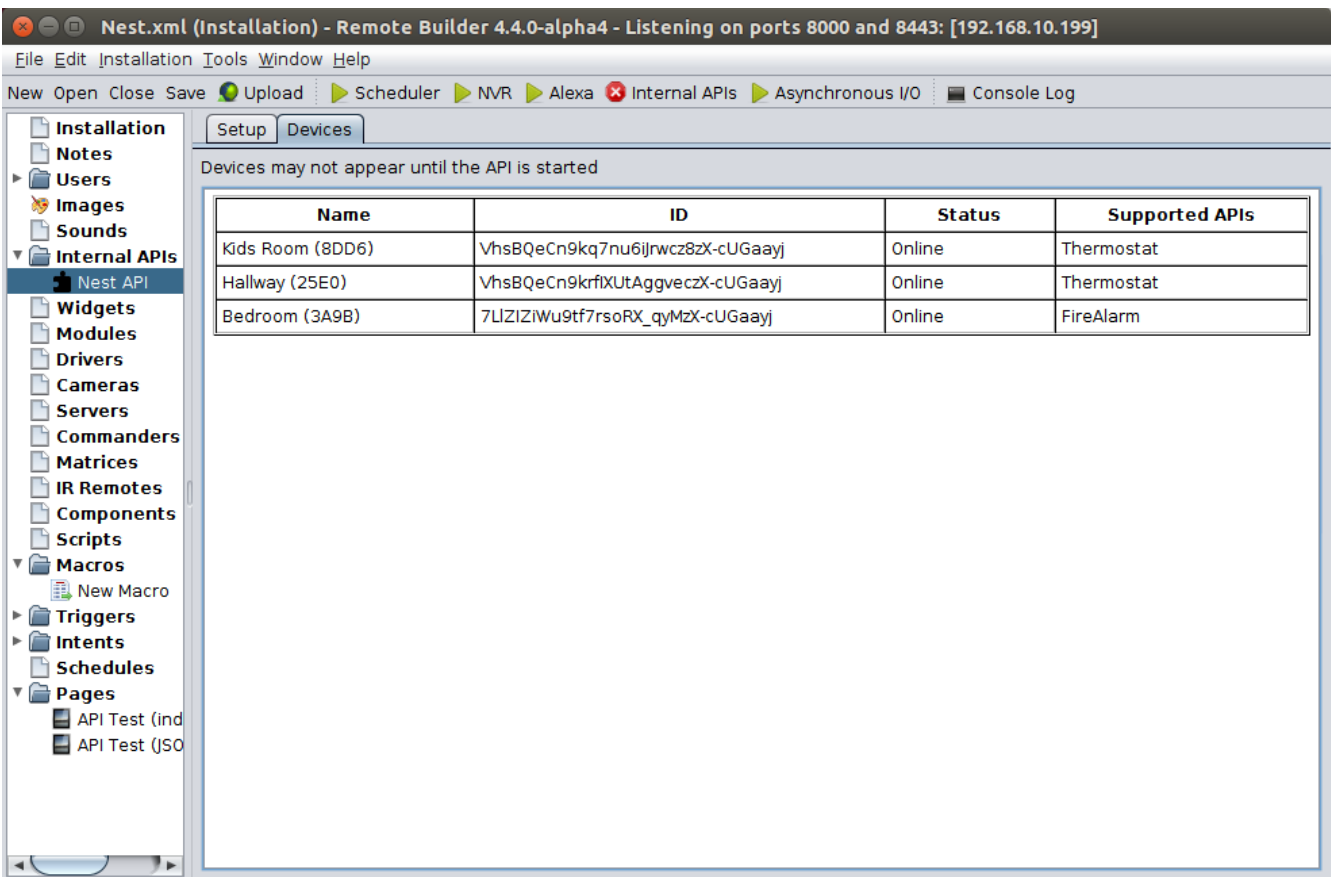
API.matrixConnectToDefaultVLANByName(matrixName, dstPortName)

This special function takes the given destination port off of the matrix **matrixName** and places it back onto the default VLAN for the matrix switch. Doing so allows the port to communicate normally with all non-matrix ports on the switch. This communication will be immediately broken when the destination port is set to a matrix source port again.

INTERNAL APIS

Internal APIs bridge the gap between the proprietary APIs/protocols of certain devices and a standard set of API commands. They arose out of the need to control devices that require authorization and/or need device state to be cached for performance reasons or due to service limitations.

Internal APIs are built-in to the system and may be activated in Remote Builder by choosing "Add Internal API..." from the "Installation" menu while an installation is open. After following the API specific setup procedures, the API exposes access to any devices it finds through device-type-specific APIs documented in this manual. The set of devices and the APIs they support appear in the "Devices" tab when the Internal API is selected:



Support for new Internal APIs will be added to the system based on demand. Contact your dealer to request an Internal API be created for a particular type of device.

The Internal API system was added to platform version 4.4.

Internal API Basics

First, some terminology:

Internal API

An **Internal API** provides access to all devices that use a particular, proprietary protocol or API. For example, the "Nest API" is an Internal API that provides control (as of 4.4) of all Nest thermostat and smoke+CO alarm devices attached to a single Nest account. Internal APIs are somewhat analogous to Drivers. The biggest differences are:

- 1) Internal APIs are built-in and always available to use within Remote Builder. Whereas Drivers must be created from scratch and/or loaded from a repository of installation XMLs.
- 2) The set of commands exposed by Internal APIs is entirely pre-determined by the types of devices they provide access to. In other words, a thermostat is always controlled the same way when an Internal API is available for it, regardless of brand or model. Drivers, on the other hand, are not required to follow any particular convention.

Internal Device

An **Internal Device** is the name of a device discovered (or manually setup) by an Internal API that can be controlled using Physics Systems API commands. For example, the previous screenshot depicts an installation where the internal Nest API has discovered and made available three internal devices – the "Kid's Room" thermostat, the "Hallway" thermostat and the "Bedroom" smoke+CO alarm.

Internal Devices are somewhat analogous to Components. The biggest differences are:

- 1) When possible, they are automatically discovered and made available for control, not manually created
- 2) Instead of using Component API commands to control Internal Devices, you must use the appropriate Internal Device API for that type of device.

Internal Device API

An **Internal Device API** is the API used to control a particular type of Internal Device. For example, the Internal Device API used to control ALL thermostats is called "Thermostat." As new Internal APIs are added to provide support for other brands of thermostats, they will always be controllable with the same "Thermostat" API.

Future versions of the platform are planned to allow voice intents to automatically control all Internal Devices of the system by device name.

Internal Device API Basics

All Internal Devices have at least two intrinsic properties: a name and a device ID. The name is determined by discovery. It is generally the same name that is assigned using the device's proprietary software or app. ***It is the responsibility of the installer to ensure that the name of each device is unique within the set of all devices of a particular type.*** For example, there should never be two thermostats in the same system named "Bob." However, there may be a thermostat named "Bob" and a light named "Bob."

To allow for ambiguous situations, or the case where the end user frequently changes the names of devices, you may also control devices by their ID. The IDs of each device are guaranteed to persist between reboots and reloadings of the same XML. In any Internal Device API command that follows, wherever you are required to specify the name of the device in the command, you may instead use the device's ID.

Names are case insensitive, device IDs are case sensitive.

getPropertiesJSON()

ALL Internal Device APIs support at least one command in both server and client side script: **getPropertiesJSON()**. This command retrieves ALL of the current properties of the given device and returns a simple JSON object representation of those properties and their current values. The properties that will be present are specified in the respective documentation for that API.

If a particular property is not supported by the underlying hardware, it will not be present in the response. Thus it will match with the JavaScript keyword **undefined**. If the current value of a property is unknown, but the property is otherwise supported, the property will be present but its value will be **null**.

This can be used to discover the capabilities of the device!

NOTE: in server side script, the response from `getPropertiesJSON()` must be parsed into an object via `JSON.parse()` before using it. In client side script, this step is done for you automatically.

`getPropertiesJSON()` is the recommend way to retrieve more than one value pertaining to a device at a time. Getting property values individually executes separate commands (and network connections), whereas `getPropertiesJSON()` combines all properties into a single command and network call.

Example: Using `getPropertiesJSON()` to do Device Capability Detection

In this example, we have a light named "Bob" that is dimmable and able to change its color, but does not support getting or setting a color temperature.

Thus, its properties would include "on", "brightness" and "color", but not "colorTemperature."

The following server side script checks if the light supports the brightness property (e.g. is dimmable). It also verifies that the light does not support changing its color temperature:

```
var properties = JSON.parse(API.Light.getPropertiesJSON("Bob"));

if(properties.brightness !== undefined) {
  print("Bob the light supports the brightness property!");

  if(properties.brightness !== null) {
    print
      ("Bob the light's brightness is currently "+properties.brightness);
  }
}

if(properties.colorTemperature !== undefined) {
  // because we said that "Bob" does not support color temperature
  print("THIS SHOULD NEVER PRINT!");
}
```

Here is the client-side script version of the previous. Note again that we do NOT need to parse the response first with client-side scripts:

```
API.Light.getPropertiesJSON("Bob",function(properties) {
  if(properties.brightness !== undefined) {
    console.log("Bob the light supports the brightness property!");

    if(properties.brightness !== null) {
      console.log
        ("Bob the light's brightness is currently "+properties.brightness);
    }
  }

  if(properties.colorTemperature !== undefined) {
    // because we said that "Bob" does not support color temperature
    console.log("THIS SHOULD NEVER PRINT!");
  }
});
```

Example: Using `getPropertiesJSON()` With Devices that Support Multiple APIs

If a device ever supports more than one type of API, the properties returned by `getPropertiesJSON()` will only apply to the particular Internal Device API you use. For example, if you had a hypothetical device named "Bob" that had the features of a thermostat AND a fire alarm, to retrieve the thermostat properties you would use:

```
API.Thermostat.getPropertiesJSON("Bob");
```

To retrieve the fire alarm properties, you would use:

```
API.FireAlarm.getPropertiesJSON("Bob");
```

INTERNAL DEVICE API: THERMOSTAT

This API is available for Internal Devices that support the "Thermostat" API.

JSON Properties

The following properties are returned by `getPropertiesJSON()`:

Property	Description	Possible Values / Notes
mode	The current HVAC mode	"Heat" – the thermostat is running in heat-only mode. "Cool" – the thermostat is running in cool-only mode. "Both" – the thermostat will apply heat and cooling as needed to keep the temperature within a specific range. "Eco" – the thermostat is in eco or energy saver mode. "Off" – the thermostat is off.
scale	The temperature scale in use by the thermostat	"F" – for Fahrenheit "C" – for Celsius
ambientTemperature	The current temperature measured by the thermostat	Value will be in the thermostat's temperature scale specified by the scale property.
humidity	The current humidity measured by the thermostat.	A percent (between 0-100).
targetTemperature	The current target temperature of the thermostat.	Only applicable if the thermostat is in "Heat" or "Cool" mode.
targetTemperatureHigh	The highest allowed temperature before cooling will be started.	Only applies when thermostat is in "Both" mode. Not supported by all hardware.
targetTemperatureLow	The lowest allowed temperature before heating will be started.	Only applies when thermostat is in "Both" mode. Not supported by all hardware.
ecoTemperatureHigh	The highest temperature the thermostat will allow when in energy saving mode.	Only applies when thermostat is in "Eco" mode. Not supported by

		all hardware.
ecoTemperatureLow	The lowest temperature the thermostat will allow when in energy saving mode.	Only applies when thermostat is in "Eco" mode. Not supported by all hardware.

API Commands

API.Thermostat.isModeAllowed(**name**, **mode**)

Returns **true** if the specified **mode** is currently supported by the thermostat with name or ID **name**. Valid mode values are specified in the properties table above.

API.Thermostat.getMode(**name**)

Returns the current mode of the thermostat with name or ID **name**.

API.Thermostat.setMode(**name**, **mode**)

Sets the current **mode** of the thermostat with name or ID **name**. Mode is nmay be "He

API.Thermostat.getScale(**name**)

Gets the temperature scale currently in use by the thermostat with name or ID **name**.

API.Thermostat.setScale(**name**, **scale**)

Sets the temperature **scale** used by the thermostat with name or ID **name**. Not supported by all thermostats. Value scale values are specified in the properties table above.

API.Thermostat.getAmbientTemperature(**name**, **scale**)

Returns the current temperature measured by the thermostat with name or ID **name**, in units of the given temperature **scale**.

API.Thermostat.getHumidity(**name**)

Returns the current humidity measured by the thermostat with name or ID **name**. In percent, between 0 and 100. Not supported by all thermostats.

API.Thermostat.getTargetTemperature(**name**, **scale**)

Returns the current target temperature of the thermostat with name or ID **name**, in units of the given temperature **scale**.

API.Thermostat.setTargetTemperature(**name**, **temperature**, **scale**)

Sets the current target **temperature** of the thermostat with name or ID **name**. Temperature is expected to be in units specified by **scale**.

API.Thermostat.getTargetTemperatureHigh(name, scale)

Gets the current maximum temperature of the thermostat with name or ID **name**. Temperature will be returned in units of the given **scale**. Not all thermostats support this feature.

API.Thermostat.setTargetTemperatureHigh(name, temperature, scale)

Sets the current maximum **temperature** of the thermostat with name or ID **name**. Temperature is expected to be in units specified by **scale**. Not all thermostats support this feature, and high/low temperatures may not apply depending on the current thermostat mode.

API.Thermostat.getTargetTemperatureLow(name, scale)

Gets the current minimum temperature of the thermostat with name or ID **name**. Temperature will be returned in units of the given **scale**. Not all thermostats support this feature.

API.Thermostat.setTargetTemperatureLow(name, temperature, scale)

Sets the current minimum **temperature** of the thermostat with name or ID **name**. Temperature is expected to be in units specified by **scale**. Not all thermostats support this feature, and high/low temperatures may not apply depending on the current thermostat mode.

API.Thermostat.getEcoTemperatureHigh(name, scale)

Gets the current maximum eco mode temperature of the thermostat with name or ID **name**. Temperature will be returned in units of the given **scale**. Not all thermostats support this feature.

API.Thermostat.getEcoTemperatureLow(name, scale)

Gets the current minimum eco mode temperature of the thermostat with name or ID **name**. Temperature will be returned in units of the given **scale**. Not all thermostats support this feature.

INTERNAL DEVICE API: FIREALARM

This API is available for Internal Devices that support the "FireAlarm" API.

JSON Properties

The following properties are returned by `getPropertiesJSON()`:

Property	Description	Possible Values / Notes
batteryState	The current state of the battery	"OK" – the battery is fine. "Replace" – the battery needs to be replaced.
alarmStateSmoke	The current alarm state of the smoke detector.	"OK" – no smoke detected. "Warning" – some smoke detected. "Emergency" – smoke detected, GTFO
alarmStateCO	The current alarm state of the carbon monoxide (CO) detector.	"OK" – no CO detected. "Warning" – some CO detected. "Emergency" – CO detected, GTFO

API Commands

API.FireAlarm.getBatteryState(name)

Returns the current state of the battery for the fire alarm with name or ID **name**.

API.FireAlarm.getAlarmState(name, alarmType)

Returns the current alarm state of the fire alarm with name or ID **name**. The type of alarm is specified by **alarmType**. Current supported values are "Smoke" for smoke alarms and "CO" for carbon monoxide alarms. Some fire alarms support only one type.

INTERNAL DEVICE API: LIGHT

This API is available for Internal Devices that support the "Light" API.

JSON Properties

The following properties are returned by `getPropertiesJSON()`:

Property	Description	Possible Values / Notes
on	The powered on state of the light	true or false
brightness	The current dimmer level / brightness of the light.	A floating point value between 0.0 and 100.0, inclusive. 0.0 indicates the light is powered off. Not all lights support dimming.
color	The current color of the light.	An RGB value, with each color component ranging from 0.0 to 1.0, inclusive. The color property has three child properties: red , green and blue . Not all lights support changing color.
colorTemperature	The current color temperature of the light.	A color temperature, in mireds. This unit is simply 1,000,000 divided by the color temperature in Kelvin. For example, a color temperature of 6500K is about 154 mired. Not all lights support changing the color temperature.

API Commands

API.Light.isOn(name)

Returns whether the light with name or ID **name** is on.

API.Light.setOn(name, on)

Turns the light with name or ID **name** on or off, depending on the value of **on**.

API.Light.getBrightness(name)

Returns the current brightness of the light with name or ID **name**.

API.Light.setBrightness(name, brightness)

Sets the current **brightness** of the light with name or ID **name**. Setting the brightness to 0 turns the light off, setting it to any other value with automatically turn it on.

API.Light.getColorRGB(name)

Returns the current color of the light with name or ID **name**. The color is returned as a simple RGB object with the properties **red**, **green** and **blue**.

For example, the following server-side snippet displays the currently color of the light named "Bob":

```
var color = API.Light.getColorRGB("Bob");  
print("Red: "+color.red+", Green: "+color.green+", Blue: "+color.blue);
```

API.Light.setColorRGB(name, red, green, blue)

Sets the **red**, **green** and **blue** color values of the light with name or ID **name**. Remember, color values range between 0.0 and 1.0, inclusive.

API.Light.getColorTemperature(name)

Retrieves the current color temperature of the light with name or ID **name**.

API.Light.setColorTemperature(name, temperature)

Sets the current color **temperature** of the light with name or ID **name**. Remember, color temperature is specified in units of mireds. To convert from Kelvin to mireds and vice versa, divide 1,000,000 by the value.

SERVER SIDE ONLY COMMANDS

API.runCommanderCommand(commandName, commandName, arguments...)

This command functions similar to **runComponentCommand()**, except it runs commander commands.

API.addDynamicResource(name, resourceBytes, resourceType)

Makes the given resource available at `/dynamic/name`. **resourceBytes** must be a Java (not javascript) byte array (i.e., a `byte[]`). **resourceType** should specify the MIME type of the data (i.e., "image/jpeg").

Currently, dynamic resources may consume up to 16MB. If an attempt is made to add another resource after 16MB has been used, the least recently used resources will be purged, one at a time, until enough space is available for the new resource. This space limitation may become configurable in the future.

API.getDynamicResourceData(name)

Returns the data (**resourceBytes**) that was previously stored as a dynamic resource by the given **name**. The return value will be a Java `byte[]`. Returns **null** if no resource with that name is available.

API.getDynamicResourceType(name)

Returns the **resourceType** value of the dynamic resource with the given **name**. Returns **null** if no resource with that name is available.

API.removeDynamicResource(name)

Removes the dynamic resource with the given **name** from the pool.

API.sendEmail(recipient, subject, message)

Sends an email **message** to the given **recipient**, with the given **subject**. This functionality will only operate on servers that have an active PSNET account.

CLIENT SIDE ONLY COMMANDS

These commands are only available in client side scripts.

API.setLabelValue(labelScriptId, value)

Changes the currently displayed value of the label with script ID **labelScriptID** to the given string. **value** may be HTML. The script ID is set in the properties window for the label (available by double-clicking the label in Remote Builder's page editor).

For example, to change the text of the label with id "status" to "OFF":

```
API.setLabelValue("status", "OFF");
```

API.setLabelImage(labelScriptId, imageId)

Changes the current image of the label with script ID **labelScriptID** to the image with image ID **imageID**. Image IDs are shown in the main image list (shown by clicking the "Images" tree item in Remote Builder).

For example, to change the image of the label with id "status" to image ID #45:

```
API.setLabelImage("status", 45);
```

API.setLabelImageURL(labelScriptId, imageURL)

Changes the current image of the label with script ID **labelScriptID** to the image at the given URL.

For example, to change the image of the label with id "picture" to our logo:

```
API.setLabelImageURL  
("picture"  
 , "http://physicssystems.com/images/logo164x75.png");
```

API.setLabelVisible(labelScriptId, visible)

Shows or hides the specified label. **visible** must either be **true** or **false**.

API.setButtonVisible(buttonScriptId, visible)

Shows or hides the specified button. **visible** must either be **true** or **false**.

The following example makes the button with script id "power" visible:

```
API.setButtonVisible("power", true);
```

API.setButtonLabel(buttonScriptId, label)

Sets the button label (text) of the button with script id **buttonScriptId**.

API.setButtonImages(buttonScriptId, upImageId, downImageId)

Sets the up and down images for the button with script id **buttonScriptId**. The IDs for the up and down image can be determined in Remote Builder in the master list of images.

API.setSliderVisible(sliderScriptId, visible)

Shows or hides the specified slider. **visible** must either be **true** or **false**.

API.jumpToPage(pageScriptId)

Immediately jumps to the page with the given script id.

API.goBack()

Goes back to the previous page in the page history. Note, this is the page history maintained by the server, not the browser's page history. Currently, the history length is set to 10 pages.

API.logout()

Immediately logs out the current user. NOTE: logging out does not cause the current page to reload. This can cause confusing behavior if the current page uses functionality that requires the user to be logged in. To avoid confusing the user you should load another page after logging out. For example, to logout the current user and then send them to the home page:

```
API.logout();  
window.location = '/';
```


Slider commands

The following commands operate on sliders on the client side.

API.Slider.setValue(*sliderScriptId*, *value*)

Sets the current **value** for the slider with script ID **sliderScriptId**. The value will be automatically truncated to the appropriate precision, if necessary. For example, if the "Step" of the slider is 0.5, and **value** is 0.51483, it will be truncated to 0.5.

API.Slider.assignProperty(*sliderScriptId*, *apiName*, *deviceName*, *propertyName*)

This extremely powerful function assigns an Internal Device property directly to the slider with script ID **sliderScriptId**. The current value will be automatically retrieved. Sliding the thumb will cause the current value to be updated.

apiName specifies which Internal Device API should be used to access the device with the given name or ID of **deviceName**. Neither value is case sensitive e.g. "Light" and "light" are both acceptable for referring to the "Light" Internal Device API. **propertyName** specifies which property to bind the slider to.

Sliders that have been assigned properties will honor the values of the "Overrides" set in Remote Builder.

Currently Assignable Internal Device API Properties

These are the properties that are currently assignable to sliders. More properties will be added in the future:

Internal Device API	Assignable Properties
Light	brightness

Example: using API.Slider.assignProperty()

Assuming there is a light in the system named "Bob", the following snippet will assign the slider with script ID of "sliderBob" to the brightness:

```
API.Slider.assignProperty("sliderBob", "light", "Bob", "brightness");
```

DYNAMIC PAGE LOADING

One of the new features added to the 2.0 platform is the ability to load pages within other pages and display them dynamically. This enables you to create even more powerful interfaces and drastically reduce the amount of time needed to develop multi-room systems.

For example, if multiple rooms in an installation share the same source equipment, you only need to create one page with controls for each source and display them as needed on the per-room pages.

The basic steps for using dynamic pages are:

1. Create a page to be loaded dynamically in Remote Builder and assign it a script ID. We recommend using the prefix “p_” in the ID. For example, p_controls or p_airConditioning.
2. Create a handler function for **loadPage()** that will be called when the page is loaded and ready to be shown. This handler function should store a reference to the loaded page and display it. See the [example](#) further down in this section.
3. Call **loadPage()** using the previously set page script ID and handler function
4. If the page should no longer be displayed but should be quickly accessible, use **hidePage()** to hide it.

API.loadPage(**pageScriptID**, **handler**)

Loads the page with script ID **pageScriptID** for later inline display on the current page. When the page has been loaded it will be passed as the first argument to **handler**. Pages are initially invisible after being loaded and their Enter actions will not be run until **showPage()** is called.

API.showPage(**page**, **x**, **y**, **zIndex**)

After a page has been loaded via **loadPage()**, it can be displayed using **showPage()**. The argument **page** must be the argument previously passed to **handler** by **loadPage()**. The **x** and **y** arguments are optional and indicate the absolute x and y position in the client space to display the page in. The **zIndex** is also an optional argument (but can not be specified unless x and y are also specified). It specifies what z-index the page should be displayed at, with higher z-indexes appearing on top of lower numbered z-indexes. It is only necessary to specify the x and y coordinates and z-index once after loading a page. These attributes will be retained after **hidePage()** is called (but will be lost if **unloadPage()** is used to unload the page).

If a loaded page has Enter actions, they will be run the very first time **showPage()** is called after the page has been loaded. Subsequent calls to **showPage()** for that page will not cause the Enter actions to be-run unless the page is unloaded first via **unloadPage()** and then re-loaded.

Example: Loading and Displaying a Page Dynamically (Client Side)

The following example loads the page with script ID “p_controls” and displays it 100 pixels from the right and 120 pixels from the top of the screen, at z-index 5:

```
var g_controlsPage = null; // This global variable will store the loaded page

function controlsLoaded(controlsPage) {
    // Store the returned page in a global variable so we
    // can use it later to hide the page:
    g_controlsPage = controlsPage;
    // Show the page at 100 pixels from the right, 120 from the top, z-index 5:
    API.showPage(g_controlsPage,100,120,5);
}

// Load the page and use controlsLoaded() for the response handler function:
API.loadPage("p_controls",controlsLoaded);
```

API.makePageBackgroundTransparent (page)

Sometimes it is not desirable to display the background of a dynamically loaded page. For instance, you may design a dynamically loaded page to just be a set of controls that you want to blend into the background of the main page. Use **makePageBackgroundTransparent()** to hide the background of a page previously loaded via **loadPage()**.

API.hidePage (page)

Hides a dynamically loaded page previously shown via **showPage()**. Does NOT run the Leave actions of the page. A page hidden with **hidePage()** can be shown again by calling **showPage()**.

API.unloadPage (page)

This method performs three actions:

1. Hides the given **page**, if it is currently visible
2. Runs the Leave actions of the given page, if any
3. Completely removes the given page from the HTML DOM. All scripts that were part of the loaded page will be removed and no longer accessible by the main page, as well as any buttons, sliders, labels, widgets, etc.,

NOTE: it is not ordinarily necessary to unload a page after you are done using it unless you are loading dozens of large, complicated pages dynamically and you are experiencing browser instability. This method is provided primarily to enable you to run the Leave actions of a dynamically loaded page.