RedEye Advanced Programming Manual

for software version 2.13.0

Contents

Part I: Overview

Introduction

This manual is a companion to the *RedEye User Manual*, specifically focused on the advanced customization options available beginning with the release of RedEye app v2.5.0. Customers looking for basic usage information and general reference on the RedEye application should refer to the *RedEye User Manual*.

Although designed to be intuitive and simple to use, the topics covered in this manual are by nature more complex than other aspects of RedEye configuration and use. The flexibility which some of these features provide is invaluable in creating a fully-integrated and robust system, but that flexibility can also lead to problems if not approached properly. As a result, we recommend that customers first familiarize themselves with basic RedEye operation before approaching the advanced customization topics discussed here. Before jumping into advanced configuration, we recommend creating a backup of your RedEye configuration so that you can “reset” back to a known good state.

Of course if you do find yourself stuck, please make use of our many technical support options, including our technical support line and online customer support forums. As always, we are here to help.

Technical Support Contacts

By phone: 617-299-2000, option 1
By email: support@thinkflood.com
Online: http://thinkflood.com/support/redeye/

RedEye Architecture

From the very beginning we designed RedEye to be more than a simple remote control. Remote controls – even “universal” ones – have come to take on a rather specific meaning. Typically infrared-based, these wand-style controllers come covered in buttons and have a single function: to control specific pieces of equipment. By contrast, in RedEye we were taking something more generic – a smartphone – and repurposing it to function also as a remote control.

This repurposing of the smartphone comes with both advantages and disadvantages. On the negative side, smartphone batteries do not last as long as dedicated remote control batteries, and therefore the smartphone needs to be turned off periodically to conserve power. Also, the multi-purpose nature of the smartphone means that the remote control function exists as only one application among many, with the implication that there is latency in launching the application. There are other disadvantages, too – the lack of dedicated, physical buttons being one that often comes up in discussion.

To make RedEye a useful product, its advantages need to overcome these disadvantages. Some of these advantages are easy to tease out. For example, the touchscreen interface of today’s smartphones provides nearly infinite customization potential, so that was a no-brainer. But some of the possible advantages are not so evident, and can unachievable if not specifically accommodated in the original hardware design.

True Client-Server Design

One of our early realizations was that while using a single smartphone as a remote control introduces several disadvantages, the ability to use many smartphones – or, taken a step further, just about any networked computing device – as control interfaces provides major opportunities. At its essence, RedEye is not about converting a phone into a remote control. Instead, it is about bringing all kinds of previously isolated devices onto the Internet. In other words, RedEye becomes the network interface for TVs, A/V receivers, DVD players, air conditioners, and a host of other products. In theory, these devices are now available wherever you get an Internet connection, and you can control them with whatever networked computer you have handy, whether it be a smartphone, a tablet, or a personal computer.

When we talk of adding network capability to existing devices, there are a couple of options. One option is to provide a simple gateway – a network interface that translates commands sent from networked devices into the protocol (infrared,

RS-232, etc) that the standalone devices understand. The gateway solution is inexpensive to create (the gateway itself does not need to be particularly "smart"), but it presents some significant complexities for the networked devices. Specifically, each networked controller must "know" a lot about the devices it is controlling. This is particularly challenging when there is more than one networked controller, as those controllers must now share this information with one another. In the case of mobile devices (smartphones, tablets) which are sometimes on the network and sometimes powered down or away, sharing this information reliably becomes a practical impossibility.

The second option is to provide a server – a central repository of information that is constantly mediating requests form networked devices and sending out the control signals which the standalone devices require. In this situation networked clients require much less information about the stand-alone devices, so it is easier to add more clients, and to include clients of various different types. In addition, the control server provides a single version of the "state" of the standalone devices, so synchronization with multiple client controllers becomes relatively simple. Finally, the constant network presence of the control server means that it can handle and respond to events that happen even when there is no client controller present.

Because of the distinct advantages of the server approach, we designed RedEye from the ground up as a server — in the parlance of the control industry, RedEye is a "control processor." Thus, the same software that allows you to control RedEye from multiple iOS devices at one time has also become the foundation for things like RS-232 control and our browser-based personal computer interface.

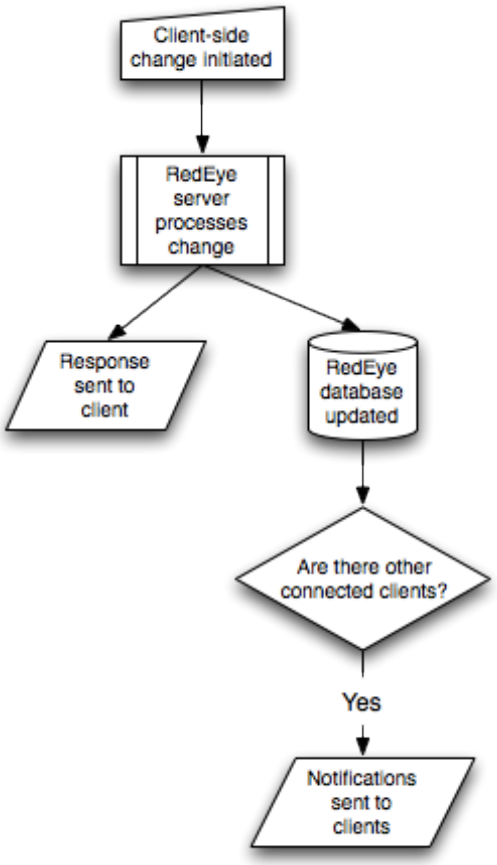
In order to take advantage of the wide array of software already available and to give us a strong foundation on which we could build future functionality, we made RedEye a full Linux server. This choice has some cost implications – running the full version of Linux (as opposed to μLinux or another embedded operating system) brings with it some relatively expensive requirements on the hardware side. On the other hand, the flexibility we gain in return has meant that RedEye can do things that were previously only possible on much more expensive systems and requiring several days of custom programming for each installation.

Relational Database

At the heart of each RedEye server is a full relational database. This database stores information about the RedEye unit itself, as well as the devices it is configured to control, and the activities used to control those devices.

Using a relational database for configuration storage gives us a couple of advantages. First, the database structure helps ensure a logical organization to the data, which makes it easy to access data in new ways and add new functionality without having to reorganize everything when we make relatively minor changes to RedEye software. Second, the database provides us with a transaction mechanism to ensure that we make meaningful changes and leave the database in a known good state if an error occurs in the middle of an update.

This last point about transactions is particularly important to RedEye's client-server design. When one client sends down a request to update RedEye's configuration, we use the transaction capability of RedEye's internal database to track the requested changes. Once those changes are complete, RedEye sends out a message to all connected client controllers notifying them of the change.



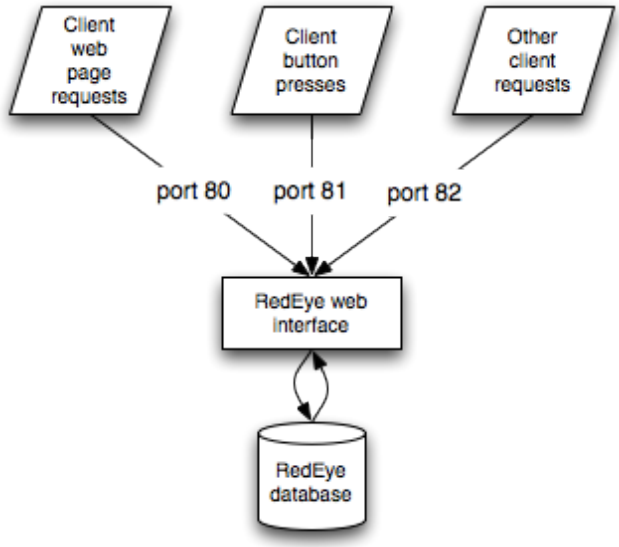
Web Interface

Another important component in the RedEye design is its web interface. Each RedEye unit runs a full web server. This not only enables RedEye's built-in browser application, it also provides a common interface that just about any networked device can access, be it a personal computer, a smartphone, or something else we haven't yet considered.

Client controllers connect to RedEye using HyperText Transfer Protocol (HTTP), and send documents in an eXtensible Markup Language (XML) format.

Although early RedEye prototypes used a regular HTTP server on port 80, in testing we quickly found the need to optimize for performance. As a result, if you

"sniff" RedEye traffic on a network today you will notice that very little information travels over the standard web server on port 80[1]. Instead, you will notice a persistent TCP connection on port 81, and transient HTTP connections on port 82. The port 81 traffic is designed for low-latency functions that need to fire rapidly – think button presses in an activity – and the port 82 traffic is for all other requests. Basically these are streamlined versions of the web server running on port 80, there to reduce overhead and memory usage so that RedEye can respond more quickly and handle more client requests. Thus, if you are configuring RedEye for remote access using port forwarding, you need to make sure you set up forwarding on three ports – 80, 81, and 82 – in order to make sure the application can work fully.



Scripting Engine

Beginning with v2.5.0 of the RedEye application, RedEye includes a scripting engine, which provides access to all of RedEye's basic functionality, plus tools that simplify the process of communicating with external devices which might use more complicated data structures. RedEye's scripting engine essentially allows you to "crack open the hood" to create your own custom commands, launch and shutdown activities, fine tune user interface elements, monitor and respond to information coming in from external devices, and even store and retrieve data using the file system and custom databases.

[1] Technically a web server can run on any port, but most publicly available web servers "listen" on port 80 (those that do not usually employ port 8080).
[2] You should also avoid variable names that begin with an underscore followed by all capital letters, as this is the format for many of Lua's internal variables

Scripting is another one of these basic pillars that makes RedEye so powerful. Without a scripting interface it would be difficult to handle inbound data except in very simple ways. Limited primarily to the one-way sending of commands, RedEye would be no more than a basic remote control. With a proper scripting interface, however, you can customize RedEye to respond to various external events in meaningful ways, such as launching activities, updating client controller interfaces, and broadcasting information to client controllers.

Because this scripting engine provides much of the current customization capability in your RedEye system, most of the remainder of this manual will focus on scripting and related topics.

Part II: Scripting Basics

Introduction to Lua

One of the funny things about software developers is that they tend to have a different programming language for just about everything. Indeed, it almost seems like something developers do for fun – when you have a little free time, why not create a new programming language?

While some languages appear to exist only to satisfy their creators' desire for something that appeals to their own tastes, there are some good reasons for this variety. Perhaps the most compelling reason is that computers themselves come in many different shapes and sizes. Big enterprise servers are optimized for one kind of processing, while small, embedded microcontrollers are optimized for another, and developers have created different languages to make it easier to address these different platforms.

When we set out to add scripting capabilities to RedEye we were not interested in creating our own scripting language, but we did want to choose a language that works well with RedEye's capabilities. The Lua programming language (http://www.lua.org/) fits the bill nicely. In particular, the following traits attracted us to Lua:

1. **Lua is lightweight.** Lua compiles down to only a few hundred kilobytes, which means more of the storage space in RedEye is available for your scripts and your data, and not consumed by the scripting engine itself.
2. **Lua works well with C/C++ code.** Lua is written in C – just like RedEye – and so we were able to compile it directly into the RedEye application itself. This means that when you call system functions from your Lua scripts, often you are getting right down to "bare metal" – taking advantage of the maximum speed and efficiency that compiled code can offer.
3. **Lua is well established.** Lua is already a popular scripting language, which means that it is well tested and reliable. In addition, there are many freely available resources that make it easy to extend Lua for new purposes. We take advantage of several of these extensions in RedEye (more about that later).
4. **Lua is familiar.** Perhaps you have not programmed with Lua before. Even so, Lua's syntax shares much in common with other programming languages. It also uses a minimum of obscure symbols, instead opting for common, easily recognized keywords.

Basic Syntax

With programming languages, nothing explains better than a few examples, and so we provide those here. A quick note before we begin, though: this manual is not intended to stand as a comprehensive reference on programming in Lua, but rather just to provide a quick introduction so that we can proceed with learning how to customize RedEye. For the interested reader (or the more serious programming project) there are other learning resources available. In particular, the book *Programming in Lua* offers a solid treatment of the language and is available not only in English, but also German, Korean, Chinese, and Japanese.

First, Lua is case-sensitive. That is, it differentiates between “For” and “for”. Thus, the following are not the same:

```
print("Hello Lua")
```

versus

```
Print("Hello Lua")
```

In fact, the latter does nothing for you (well, you will get an error), whereas the former writes text to the standard output (usually the console if you are programming interactively). This brings up another point: Lua keywords are all lowercase. Since Lua is case sensitive, this helps because you do not have to use the Shift key as much when typing.

Lua programs execute from top to bottom. Thus if you write something on the first line of a script, it will execute first. If you wish to call a function, however, you must define the function before the line of code on which it executes. Therefore the following will fail:

```
printHello()

function printHello()
    print("Hello Lua")
end
```

whereas the following is perfectly fine:

```
function printHello()
    print("Hello Lua")
end

printHello()
```

Lua doesn't care about line endings (much). Some languages let you continue a line on and on until you type an end of line character (such as a semi-colon). Others require you to type some kind of continuation character (such as an underscore) if you spill over onto a new line. Lua doesn't really care where you break your lines.

Instead, Lua pays attention to "chunks" of code. That is, it is looking for you to use correct syntax. For example, all **function** statements must finish with an **end** statement. Likewise, **if** conditions end with the word **then**, and the code they execute ends with the word **end**. Thus, the following is valid:

```
if today == "Monday" and
   thisMonth == "April" then
    expectRain()
end
```

while the following will fail:

```
if thisMonth == "December"
    expectSnow()
end
```

There are a couple of exceptions here, the biggest being the way Lua handles multi-line string literals. Specifically, if you start a string on one line using double quotation marks (") you must end it on the same line using double quotation marks. Therefore the following is *not* valid:

```
local longString = "This is a very long string of text. It has to
remain on one line if it is in double quotes like this."
```

If you want to create a multi-line string, you should surround it with double square brackets ([[and]]), as follows:

```
local longString = [[This is a very long string of text. It can
spill over to additional lines if you use double square brackets
like this.]]
```

Comments are also sensitive to line breaks. For example, two dash characters (--) denote a comment that runs to the end of the line:

```
-- The following function prints a greeting
function printHello()
    print("Hello Lua")
end
```

If you want to stretch a comment over multiple lines, or if you want to end a comment before the end of a line, you must enclose it in the aforementioned double square brackets, as follows:

```
--[[ The following function prints a greeting
    parameters: none
    return: none
]]
function printHello()
    print("Hello Lua")
end
```

Variables and Scope

Like most other scripting languages, in Lua you do not have to declare a variable before you use it, and the variable will take on a type based on the value it is assigned. For example, the following are all valid statements:

```
projectName = "Dvorak"
projectDays = 10
project2Name = "Brahms"
_projectCost2 = 512.37
project_has_started = true
local projectDescription = "Something musical"
```

As you can see, variable names can have letters, numbers, and underscore characters (although they cannot begin with a digit)[2]. They can use upper- and lower-case letters (remember, Lua is case sensitive), and there is no limitation on length.

Note that the keyword “local” preceding the last variable assignment. When you declare a variable implicitly, you are creating a global variable – one that exists during the life of the Lua environment (and since RedEye’s Lua environment is running from the moment the RedEye box boots until it shuts down, this is a long time). As a result, we generally recommend creating variables using the **local** keyword. This gives them local scope – meaning that outside the function or block of code in which they are declared they are no longer used. This allows Lua to reclaim memory from them, making your programs more efficient. It also avoids sticky situations in which a variable assignment made in one program is accidentally used somewhere else.

[2] You should also avoid variable names that begin with an underscore followed by all capital letters, as this is the format for many of Lua’s internal variables (e.g., _VERSION), which you might accidentally overwrite.

Another point of interest regarding the above variables is that we have three of the six possible Lua types represented. Specifically, we have **strings**, **numbers**, and a **Boolean** (true/false). The other types are **nil** (for undefined variables), **tables**, and **functions**.

The last two types deserve special mention. In Lua, a table is similar to an array – it can hold multiple values. In fact, tables can hold other tables, so you can easily create multi-dimensional arrays. Unlike arrays, however, tables automatically size themselves, so you can easily add and remove member elements. To define a table, we use curly braces. The following creates an empty table:

```
local playlist = {}
```

Once you have defined a table, it is easy to add elements to it. One thing that makes tables really powerful is the ability "index" them using strings, numbers, etc. For example,

```
local songName = "Haitian Fight Song"
playlist[songName] = [[/MusicServer/iTunes/iTunes Music/Charles
Mingus/The Clown/1-01 Haitian Fight Song.m4a]]
```

This flexibility allows you to use Lua tables like arrays (ordered lists of elements) or like dictionaries (indexed using keys). To remove an element from a dictionary, you simply set it to nil:

```
playlist[songName] = nil
```

Functions in Lua are also quite powerful, because you can treat a function just like any other data. This means that you can assign a function to a variable, or store a list of functions in a table. The ultimate implication of this approach is that Lua is capable of object-oriented programming. Although the object-oriented approach to programming is outside the scope of this manual, we will encounter several ways in which Lua's objected-oriented capabilities make scripting much more simple and intuitive than it would be otherwise.

Operators and Keywords

We have already been dancing around some of these key bits of the language, so now it is time for a formal introduction. Lua is a streamlined language and by default has but a few operators and keywords that you need to learn. Most of the other functionality is available through extensions to the language (more on that in a minute).

As you may have picked up from the above code samples, Lua includes a handful of basic mathematical operators. Specifically, we have the following, sorted by their order of operations:

Operator	Function
^	exponentiation
not	negation (logical)
#	length (of a string or table)
-	negation (numeric)
*	multiplication
/	division
%	modulo (remainder)
..	concatenation (of strings)
+	addition
-	subtraction
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equality
~=	inequality
and	logical and
or	logical or
=	assignment

As in most languages, you can use parentheses to group operations together and thus override the default order of operations.

Another point to note is that the logical operators (and, or) are "short-circuiting" – that is, they will stop evaluating as soon as the expression becomes true. This is particularly helpful when the second half of a statement will break if the first half is false, as in the following example:

```
if (playlist[songName] ~= nil) and
   (#playlist[songName] > 0) then
    -- play the song
end
```

In the above code, if the playlist table does not contain an entry for songName, the entire condition is automatically false, and therefore Lua will not attempt to evaluate the length of the (nil) entry in the second half.

As you can tell, these operators are not terribly useful without the ability to include them in loops, conditional statements, and functions.

Conditional Statements

Conditional statements are one of the most common programming idioms, and the syntax is similar across many languages. In Lua, we format a conditional statement as follows:

```
if some_condition then
    -- do something
end
```

In the above, the **if**, **then**, and **end** keywords are required. Lua evaluates statement(s) between the **if** and **then** keywords. If they are true, then Lua proceeds to execute the statements between **then** and **end**; otherwise, Lua skips on to the next line of code after the **end** keyword.

A close cousin to this if … then logic is the if … then … else statement, which allows us to propose an alternate sequence of steps to execute if the condition is false:

```
if some_condition then
    -- do something
else
    -- do something different
end
```

We can extend this further with the if … then … elseif statement:

```
if some_condition then
    -- do something
elseif some_other_condition then
    -- do something different
else
    -- do yet something else
end
```

Conditional Loops

Conditional statements are great, but what if we need to repeat something again and again? Another common idiom is the conditional loop. It looks a lot like a conditional statement, except that it continues to execute while the condition is true.

Lua provides two types of conditional loops: the **while** loop, which evaluates its condition before each execution and loops as long as the condition is true, and the **repeat** loop, which evaluates its condition at the end of each execution and

repeats until the condition becomes true. Which flavor you choose is really a matter of programming style.

```
while some_condition do
    -- do something
end
```

```
repeat
    -- do something
until some_condition
```

Iterative Loops

Perhaps more common than conditional loops are iterative loops, which execute for a certain number of times before exiting. Lua provides two flavors of iterative loop, both using the **for** keyword.

The first iterative loop is the “numeric for” loop – a loop with a defined start and end point, and some fixed increment on each iteration. In Lua, the syntax for this kind of loop is quite concise when compared with other languages:

```
for counter = 1, 11, 2 do
    -- do something
end
```

The above loop statement does three things:

1. It initializes a variable called “counter” to the value 1
2. Executes the code inside the loop, each time adding 2 to the value of counter
3. When counter reaches the value 11, the loop executes one final time and then exits

Let’s take a look at some of the particulars. First, we do not need to include the **local** keyword before declaring “counter” – because the declaration is part of a **for** loop, Lua knows that the scope of counter should be limited only to this function.

Second, the increment value for each loop execution is 2. This number can be positive or negative. It can even have a fractional value (in Lua, all numbers are floating point, so all of the above numbers can be non-integers). In fact, we can even leave out the increment value – Lua will simply assume that we want each loop to increment by +1.

Finally, we could just as easily substitute variables for any of these numbers. For example:

```
for songCounter = 1, playlist.count do
    playSong(playlist[songCounter])
end
```

In the above loop, we use the default increment value of +1 and loop through the playlist until we have played every song. A couple of points to note here:

1. **It is common practice in Lua to begin counting at 1.** Many other programming languages are 0-based. Although you can do whatever you like in your own loops, there are places in the code (string parsing, for example) where you need to know this, so it is helpful to follow the convention everywhere.
2. **You should never modify the value of songCounter within the loop.** Unlike many other programming languages, modifying the counter variable inside a **for** loop can lead to unexpected results.

While the above "numeric for" loops are fine, Lua also provides a much more convenient version of the iterative loop, something called the "generic for" loop – what in some languages has come to be known as a "foreach" loop. "Generic for" loops look like "numeric for" loops, except that you do not have to set up the start, stop, and increment values. The following file system example illustrates:

```
for line in io.lines() do
    -- do something with each line
end
```

What's going on here? It turns out that the file system object (io) provides a special function called an *iterator*. This iterator will keep track of all that information we had to provide in the "numeric for" loop automatically. Even better, it returns an object (in this case, a string corresponding to a single line in the file) that we can use immediately – we do not need to pull the value out of a table or anything. Pretty cool, right? Of course you can write your own iterators, but just being able to use the ones that come with existing Lua objects often is helpful enough.

One thing that can happen with these "generic for" iterators is that you may want to exit the loop early – for example, if you are searching for a particular value it does not make much sense to keep looping through once you have found that value. In this situation we can make use of the **break** keyword as follows:

```
local foundText = false
for line in io.lines() do
    if line == "text we're looking for" then
        foundText = true
        break
    end
end
```

On point of interest here is that, unlike other programming languages, in Lua the **break** keyword must be the last line of code within a chunk. In the above example, it is the last line of code in the "if … then" conditional statement, so that's fine – we can go ahead and execute other code after the end of the **if** statement. The reason for this rule is that **break** immediately halts execution, and therefore any code following it is wasted.

Function Declarations

The last keyword that we need to cover is the all-important **function**. Of course you don't *have* to write functions in your Lua scripts, but if you have a chunk of code that you want to run again and again, it's much easier to write it once in a function declaration and call it over and over rather than copying and pasting it everywhere.

Lua actually provides two methods for declaring a function. The first is the most familiar, and it looks like this:

```
function playSong(filePath, repeat)
    -- do something here
end
```

Simple, no? At the top we let Lua know we are creating a function. Then we give it a name (in this case, "playSong") and some input parameters (in this case there are two: "filePath" and "repeat"). Then we write our code and finish with the **end** keyword.

The alternative method of declaring a function reminds us that in Lua functions are variables, too. Thus, we could write the above function as follows:

```
playSong = function(filePath, repeat)
    -- do something here
end
```

Normally, it does not matter which kind of function declaration you use, but the second version provides some additional options when writing object-oriented Lua scripts.

Functions in Lua are extremely flexible. You can give them as many input parameters as you like, for example. You can also call them with extra parameters (the extra ones are discarded) or missing parameters (which receive the default value of **nil**). In fact, you can even give them a variable number of parameters, which may be familiar from some other languages.

Functions can also return values. What may be unexpected here is that Lua functions can actually return *more than one* value if you want, as well.

How does all this work? The following example illustrates:

```
function add(...)
    local sum = 0

    for index, value in ipairs{...} do
        sum = sum + value
    end

    return sum
end
```

1. We declare our function, giving it a name ("add") and a variable list of parameters (denoted by "…")
2. The variable parameter list is actually a Lua table, so we can iterate through it using Lua's **ipairs** function. This function actually returns two values: the index of a table member, and its value. (We only need the value, so we ignore the index.)
3. We sum the values from the table
4. We use the **return** keyword to send back the results

Just as with the **break** keyword in loops, the **return** keyword must be the last statement of a function; Lua will not execute any code after the return.

If you wanted to modify this function to return multiple values – for example, a count of the parameters passed alongside the sum – you could tweak the return statement as follows:

```
return count, sum
```

In this case, count would be a local variable which we could define as the length (remember the length operator?) of the parameter table:

```
local count = #...
```

Referencing Other Libraries

One of the most significant things you can do in any language is include functions from other code libraries. Lua allows you to include other scripts and libraries through its **require** statement.

When you include a require statement in your script, Lua will load (and run) the library you specify. Although technically you can put a require statement anywhere in your code (remember, Lua runs from top to bottom), standard practice is to place your require statements at the top of your scripts.

There is at least one Lua library that you will want to reference in your RedEye scripts: RedEye's built-in library of system functions. At the top of each of RedEye's script templates, you will find the following line of code:

```
require "systemScript"
```

What do you get for including the systemScript library? In addition to plumbing required to set up port listeners and message queues (we'll dive more into those later), RedEye's system script includes a number of functions that allow you to send commands, launch activities, and so forth.

Strings

The string data type merits particular attention because so much of programming deals with text – formatting, parsing, and presenting. From this perspective, whenever we encounter a new language it is important to understand how strings are stored and manipulated.

Lua strings are immutable – in other words, once you create a string you cannot modify it. If you concatenate one string with another, the result is a new string, not an update to the original.

To concatenate strings in Lua, we use the double dot (..) operator, as follows:

```
local firstName = "John"
local lastName = "Doe"
local fullName = lastName .. ", " .. firstName
```

The last line here takes the last name, adds a comma and a space, and then appends the first name to the end. The result is a new string, which we store in the local variable fullName. Neither lastName nor firstName change in the process.

Simple enough, right? There is one implication of immutable strings that we need to note with caution, and that happens when concatenating large numbers of strings (or even small numbers of large strings). Let's take a closer look at our above example. It turns out that fullName is not the only new string we have created. There is an intermediate result – created and then immediately discarded – that may not be immediately obvious. This intermediate string variable exists because the last line of code contains two string concatenation statements. The first adds the comma and space to the last name, and the second – the one we store in fullName – adds the first name onto that result. Thus, under the covers Lua is actually doing the following:

```
local intermediateResult = lastName .. ", "
local fullName = intermediateResult .. firstName
```

On first blush this may not seem like a big deal, and in this case it really isn't. But now let's imagine that our program contains a loop, and inside this loop we are continually adding a bit more text onto the end of a string variable. Perhaps we are reading the contents of a file into memory, or maybe we are formatting an XML document as part of some communication with another device. We could easily create and destroy hundreds or thousands of these intermediate string objects. Ultimately, all of that memory allocation and string concatenation has a cost, and it can be significant.

The solution is to grab all of those bits of string that we want to concatenate, store them somewhere we can get to them, and then to perform one big concatenation at the end, with no intermediate results. Lua provides a means to do this using a special operator on the table data type – the concat function. Here's how it works:

1. We create a table to store all of the string bits we care about
2. We add our strings to the table in order
3. At the end, we call the table.concat() function on our table, which gives us a single string that we can then use as needed.

In code, here's how this looks:

```
require "table"

local stringTable = {}                          -- Step 1
local tableCounter = 1
local working = true

while working do
    local newString
    -- do some work here
    stringTable[tableCounter] = newString       -- Step 2
    tableCounter = tableCounter + 1
end

local resultString = table.concat(stringTable) -- Step 3
```

Admittedly, this requires a bit of extra effort and it is not necessary in many simple cases, but it is important to note for those times when you are doing a lot of string manipulation.

Another common need is to include special characters within a string – whether it be a newline character, or perhaps a control character that an RS-232 device requires to mark the start or end of a line. Similar to many other programming languages, Lua allows the inclusion of special characters by preceding them with a backslash (\). The following reserved characters come built into Lua:

Lua escape sequence	Result
\a	bell
\b	backspace
\f	form feed
\n	newline
\r	carriage return
\t	horizontal tab
\v	vertical tab
\\	backslash
\"	double quote
\'	single quote

In addition to the above, you can include any ASCII value by prefixing it with the backslash followed by three numbers indicating its decimal value. Thus, the start text (STX) character – ASCII value 2 – can be represented by \002, and the end text (ETX) character – ASCII value 3 – can be represented by \003.

Although Lua's basic string manipulation functions are limited to the concatenation (..) and length (#) operators, the string library includes a number of useful additions. To use the string library, you have to **require** it first:

```
require "string"
```

String Casing

It's pretty common to want to compare a couple of strings together, but sometimes the straight equality (==) comparison isn't sufficient. For example, what if you don't care about the "case" (upper or lower) of the string? Lua is case sensitive, so how do we get around this? The simplest answer is to convert everything to one case and then compare:

```
function compareNames(firstName, secondName)
    local same = false

    if string.lower(firstName) == string.lower(secondName) then
        same = true
    end

    return same
end
```

As you can see here, once we require the string library, there is a new **string** object available to us. This object defines several functions. Here we used **string.lower** to create lowercase versions of the names for comparison, but we could have used **string.upper** just as easily to compare the uppercase versions.

Converting Between ASCII and Byte Values

Particularly when programming RS-232 drivers we may need to convert from ASCII strings to byte values and back. We have already seen that we can use the backslash character to create a string literal for an ASCII value (e.g., \065 = "A"). The string library defines a function that accomplishes the same thing: **string.char**.

Perhaps more useful is the inverse function, **string.byte**. Let's say that we have some inbound text and we need to determine the starting character, which is a flag from the RS-232 device telling us which function to execute. Our processing function might look something like this:

```
function ProcessInputData(inputData)
    local operation = string.byte(inputData, 1)
    if operation == 1 then
        -- do something
    elseif operation == 2 then
        -- do something else
    else
        -- etc
    end
end
```

In the above code, we receive some data from the serial port through the local function parameter inputData. Then we use string.byte to pull out the numeric ASCII value of the first byte (don't forget – Lua is 1-based). We could have pulled the second byte by using the number 2 here, or we could have pulled the last byte by passing -1 (yes, Lua is smart enough to count backwards from the end of the string if we give it a negative number). We store this value in the local variable "operation," which we can then examine to determine what our next action should be.

Substrings

Another important part of pulling apart ("parsing") strings is being able to extract their individual pieces. Sometimes we know the absolute string length and can easily parse them based on position alone. In these situations the **string.sub** function is perfect.

In this example, let's assume that we are in the same ProcessInputData function we wrote in the last example. Let's say that if the operation character has a value of 1, then that means the RS-232 device has set a value. The first three characters after the function byte might tell us what kind of value was set, and the second two characters tell us the actual value. In this case, we could modify our ProcessInputData function using **string.sub** as follows:

```
function ProcessInputData(inputData)
    local operation = string.byte(inputData, 1)
    if operation == 1 then
        local type = string.sub(inputData, 2, 4)
        local value = string.sub(inputData, 5, 6)
        if type == "PWR" then
            if tonumber(value) == 1 then
                Scripting.SetVariable("Power", "On")
            else
                Scripting.SetVariable("Power", "Off")
            end
        elseif type == "VOL" then
            Scripting.SetVariable("Volume", value)
        else
            -- handle other types here
        end
    elseif operation == 2 then
        -- do something else
    else
        -- etc
    end
end
```

What's going on here? We start just as we did in the earlier example by grabbing the ASCII value of the first byte. Then we read the next three characters (between index 2 and index 4 in the string) into a variable called "type" and the last two characters into a variable called "value"[3]. At this point all we have to do is examine the information in "type" and "value" and decide what to do with them.

In this example, we are storing the power state of the device in a custom variable (called "Power"). If the value is 1, then the power is on; otherwise, it is off. We are also storing the volume level in a variable called "Volume". In both cases we are using RedEye's built-in system function to store these variables.[4]

Pattern Matching

Absolute string positions are fine when strings have fixed lengths, but what can we do with something a little more fluid? Lua's string library includes a number of

[3] Given Lua's ability to search from the end of a string using negative numbers, if we assume that the string is always 6 characters long can you think of an alternative way to extract the same information into "value"?

[4] The benefit of storing these values using RedEye's custom variable facility is that we can then use this information to drive updates to the user interface on client controllers – perhaps updating a volume slider when the volume level changes, or changing a power button from red to green when system power is turned on.

powerful pattern-matching functions. While we will not cover them in depth here, a quick preview is useful.

Here's a real-world example. Some of Panasonic's Viera TVs offer an RS-232 control option. The general protocol is pretty simple: all functions begin with the STX character (ASCII 002) and end with the ETX character (ASCII 003). In addition, query responses arrive in the format of x:y, where x denotes the parameter which has changed, and y is the new value. Now, technically the first part of this response (the "x") is a fixed, three-character string, but the second half is not. Also, the colon (:) in the middle is just too tempting to pass up, because Lua makes it really easy to handle strings in this format.

Ideally what we want to do is grab the first half and put it into a variable, then grab the second half and put it into a second variable. From there we can evaluate the first half to see what changed, and then look at the second half to see how it changed. Lua's **string.match** function makes it possible to do this in essentially one line of code:

```
function ProcessInputData(inputData)
    local trimmedData = string.sub(inputData, 2, -2)  -- Step 1
    local command, parameter =
        string.match(trimmedData, "(.+):(.+)")        -- Step 2
    if command == "QPW" then                           -- Step 3
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Power", "On")
        else
            Scripting.SetVariable("Power", "Off")
        end
    elseif command == "QAV" then
        Scripting.SetVariable("Volume", value)
    elseif command == "QAM" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Mute", "On")
        else
            Scripting.SetVariable("Mute", "Off")
        end
    elseif command == "QMI" then
        Scripting.SetVariable("Input", parameter)
    end
end
```

There it is – the entire inbound processor for Panasonic TVs in about 20 lines of code. Here is what is happening:

1. We start by trimming off the STX and ETX characters surrounding the input string

2. We use Lua's **string.match** function to "capture" two variables – any number of characters to the left of the colon into one variable ("command"), and any number of characters to the right of colon into the other ("parameter"). (This is where the ability for functions to return multiple values really shines.)
3. Then we have a basic set of conditional statements to evaluate the command value. Depending on the command value, we set the appropriate custom variable using information from the parameter value.

Clearly, a lot of the magic here is happening in the second argument to **string.match** – that mess of punctuation which reads "(.+):(.+)". This is a basic pattern-matching string. Because of size concerns, Lua eschews the full regular expression library by which Perl programmers swear. Here are the patterns that Lua recognizes:

Pattern	Meaning
`.`	any character
`%a`	letters
`%b`	"balanced" strings
`%c`	control characters
`%d`	digits
`%l`	lowercase letters
`%p`	punctuation
`%s`	whitespace (including newline, tab, etc)
`%u`	uppercase letters
`%w`	alphanumeric characters (%a or %d)
`%x`	hexadecimal digits
`%z`	null character (\000)
`%`	escape character
`[ and ]`	character sets
`+`	one or more repetitions
`-`	zero or more repetitions – shortest match
`*`	zero or more repetitions – longest match
`?`	optional (zero or one repetitions)
`^`	complement character set or starting match
`$`	ending match

Returning back to our example, we were looking to split the input string into two based on the location of the colon. Therefore we put together a pattern expression that looked for any character (.) in some number of 1 or more (+) followed by a colon, followed by any character (.) in some number of 1 or more (+). By surrounding each of these pieces in parentheses, we tell Lua to capture them inside two return variables.

As with most regular expressions, these are easier to put together when you know what you want than they are to read and understand on paper.

Some of the above patterns meanings are not entirely obvious. Here are a few highlights:

A **balanced** string match (%b) looks for a pair of opening and closing characters surrounding some text. This is particularly useful if you are looking for strings inside of parentheses, braces, etc. For example, the syntax to find text inside of square brackets is "%b[]".

The **escape** character (%) is a special pattern-matching escape character – it does not replace backslash (\), the standard Lua string escape character. Thus if you want a pattern that finds backslashes, you would write "\\". However, since pattern-matching adds some new special characters (e.g., $ and *), you need to escape these with % when you are searching for them. And just as you include a backslash by escaping it with itself (\\), you also include a percent sign by escaping it with itself (%%).

The **character set** delimiters ([and]) allow for the creation of custom ranges of characters. You can include a number of characters directly (as in "[abcd]"), or if your characters have sequential Unicode values, you can include them in a range separated by a dash character (thus "[abcd]" is the same as "[a-d]"). You can also use the **complement** indicator (^) to get all characters *except* those listed (i.e., "[^a-d]" will match on anything except the letters a, b, c, or d).

As we have already mentioned, the plus sign (+) requires at least one repetition. The asterisk (*) and minus sign (-) both accept any number of repetitions, but when you use * you find the longest repetition, and when you use - you find the shortest repetition. Similarly, the question mark (?) finds up to one instance, but will not match if there are more repetitions.

Finally, the carat (^) and dollar sign ($) are useful in that they help specify whether the match occurs at the beginning of a string (in the case of ^) or at the end of a string (in the case of $).[5]

Again, don't be too scared off by the regular expression syntax here – these things are usually easier to use than they are to read, and they are quite

[5] Actually, in our example it turns out that using a ^ at the front of the pattern "(.+):(.+)" is a safer way to do the search. If we were to get back some text from the television which did not contain a colon, this "starting anchor" would prevent Lua from slicing the string into a bunch of small pieces and searching each one again exhaustively until it could determine that no colon existed.

powerful, so using them is worth a couple of minutes of head scratching every so often.

Scripting Basics

That's enough of the language primer for now. What you really want to do is start writing scripts, right?

Two Kinds of Scripts

The first thing to know is that although RedEye has one scripting environment (comprised of the Lua programming language and RedEye's built-in functions as supplied by the systemScript library), there are really two kinds of scripts.

Command Scripts

One type is the **command script**. If you have used RedEye before, commands will be familiar to you. They are the individual signals used to invoke specific functions on each device you are controlling. What you may not know is that each command is stored in a file on the RedEye unit. Therefore commands have two unique features:

1. **Each command is associated to a particular device.** Also, because devices are associated to ports, we can draw a line between a command and a port.
2. **Each command is stored in a file.** Thus, commands are persistent and they have a defined location where we can access them.

Action Scripts

The other kind of script is an **action script**. You may recognize actions from their use in the RedEye app today – buttons have actions, as do activities. Often, actions point right back to commands, and, indeed, they can point to script commands as well as infrared commands. However, actions are by nature more transient then full-blown commands. Specifically:

1. **Not all actions are stored.** Sometimes a client controller will create a special action, execute it, and then throw it away. For example, when you launch an activity, the RedEye app will create an action script that contains instructions to launch the activity. This script is never stored or reused, but rather generated at the point of need.
2. **When they are stored, action scripts are stored as part of something else.** RedEye does store some actions, but always as part of a button, an activity launch sequence or activity shutdown sequence, etc. Actions

scripts do not have their own files and cannot be independently executed as commands can.

3. **Actions can change based on context.** In the same way that RedEye may create an action script on the fly, it can also add additional commands to an action script depending on what is happening in the application at a given time.
4. **Action scripts are not tied directly to devices or ports.** Because of their positioning on things like buttons and activities and their transient nature, there is no clear association between them and individual devices or ports.

Impact on Scripting

The above differences have some significant implications on the way we write scripts. Chief among these is our use of the **SendMessage** function in the systemScript library.

SendMessage is a workhorse of RS-232 and IP communications[6] – it allows us to transmit data to an external device. It takes a single argument – the data (text) to transmit. On its own, this is not enough to send a message properly – we also have to know which port use for transmission (because all external devices are connected to a particular RedEye port). If you are writing a command script, then there is no problem here, because we know the device – and therefore the port – for the command. However, if you are writing an action script, any call to **SendMessage** will fail.

In an action script, the appropriate function to use is **SendMessageToPort**, which explicitly specifies the port to which the message go. In general, however, explicitly specifying the port is undesirable because it ties your scripts to a particular configuration. If you later move your device to another port, your script will break. Therefore when creating scripts that transmit data to external devices, we recommend creating them as commands and using the **SendMessage** function. After all, you can always execute a command as part of a button or activity launch action, so this really is not a significant limitation.

Another consideration when writing scripts is the purpose of the script and likelihood of reuse. Because you can write scripts for buttons and activity launch sequences, you could create a RedEye configuration with minimal commands and devices, and jump right into customizing your activities using scripting. While your activities would be functional, they would also be inflexible and hard to

[6] We use **SendMessage** for IP communication when the port protocol is TCP or UDP. For HTTP ports, we have an analogous function called **SendHttpMessage**, which we will investigate later.

maintain. For example, we will assume that you have created a play button on one activity using an action script. What if you want to invoke the same function on a button in another activity? You could use copy and paste to duplicate your action script, but the best solution is to create a play command script once and then have each button invoke that command.

Generally speaking, our bias should be toward creating command scripts. Commands are designed for reuse, and they are visible and available throughout the application. There is nothing wrong with writing action scripts here and there, but we recommend limiting your use of them to things that are specific to a particular activity. For example, adding a button to your “Watch TV” activity that runs a script to launch the “Watch Blu-Ray” activity is a good, specific use of an action script.

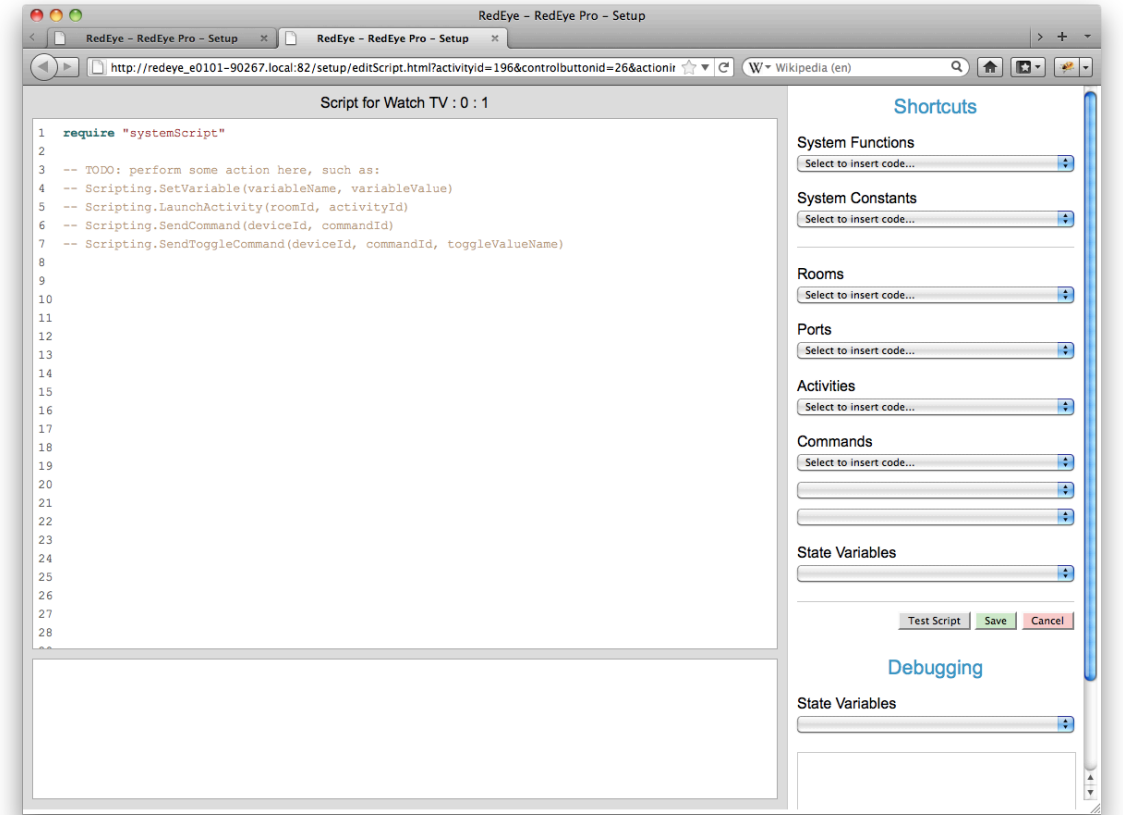
A Simple Launch Activity Script

In fact, the “Watch Blu-Ray” activity button is one of those oft-requested bits of functionality, so let’s use it as our introduction to scripting. First we need to get into our “Watch TV” activity so we can make some changes there. Scripting is a lot easier using RedEye’s built-in browser application than on a mobile device not only because you can use a regular keyboard to type your scripts, but also because a computer’s larger screen affords more space for the tools that make your scripting easier. You can edit scripts from your iPhone if you wish, but we have put a lot more effort into developing a real scripting environment through the browser app, and recommend going there whenever possible.

So, let’s launch RedEye’s browser app, switch over to the setup page, and then select the appropriate tab for our RedEye room. From there we can click on the **Activities** section and edit the button layout for our “Watch TV” activity. At the moment, this is how our activity appears onscreen:

From here let's go ahead and add the button we will use to switch to the "Watch Blu-Ray" activity. After positioning it in the upper-left corner of the layout we are ready to create the script.

First, click on the **Actions/Toggles** section in the menu to the right. Then click on the **Choose Button Action** link. From the **Type** drop-down, switch from **Command** to **Script** and then click the **Edit** button. RedEye opens a new script editing window for you to use:

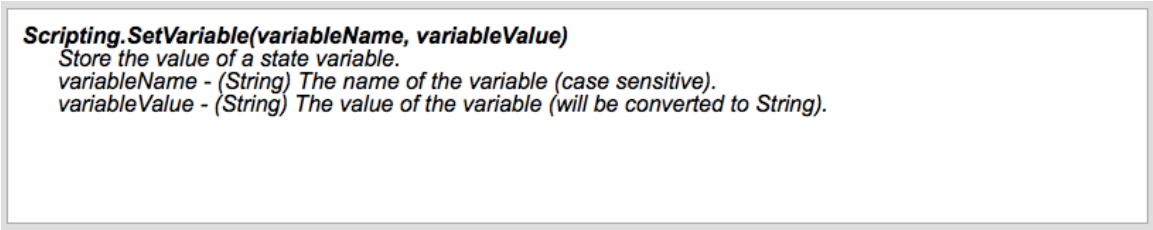


Let's take a moment to look around at the tools available to us. The script editor itself takes up most of the screen – you will notice that it is pre-populated with a template. RedEye provides different templates depending on the type of script you are editing – this is the one we get for button actions.

On the right-hand side of the screen there is a **Shortcuts** pane, which contains several quick links that insert bits of code and other useful values into your scripts when you select them. At the bottom of this pane are three buttons – **Test Script**, **Save**, and **Cancel**. If you click on the **Cancel** button, the script editor throws away any changes you have made and reloads with the last saved version of your script. Clicking **Save** stores your changes (in this case, on the button within the activity layout we are editing). You can click on **Test Script** at any time to test the script in process – there is no need to click **Save** first, nor will running a test overwrite your original script in any way.

Below the **Shortcuts** pane is the **Debugging** pane. One of the tricky things about programming RedEye is that you cannot look into the hardware to inspect what is going on. One useful debugging trick is to store information in state variables. As these are modified during your testing, you can retrieve their current values in the window below. We'll look more into debugging in a minute.

The last area in the editing window is the **Help** pane, which appears just below the editor. When your cursor is on a line of script that contains a system function or constant, this pane will display useful information and sometimes hyperlinks.



Let's go back to the editor for a minute. As you can tell, the editor will highlight text as you type to bring out Lua keywords, comments, literals, and even RedEye system functions. In fact, the editor will even help you indent your code and otherwise keep your scripts looking neat and tidy.

The first line of the template is

```
require "systemScript"
```

If you plan to invoke any of RedEye's system functions, you will need to include this statement. As a general rule, this line appears at the top of every script you write.

Below you will see a few lines of comments. For now let's go ahead and delete them – it is easy enough to add what we want using the **Shortcuts** pane. For this button we are interested in launching an activity, so let's take a look at the **System Functions** drop-down over in **Shortcuts**. Third from the bottom in that menu you will see the function "Scripting.LaunchActivity" – go ahead and select it from the list. In the editor on the left you should see the function inserted in your script (if you don't, make sure you have first clicked in the editor to position the cursor there, and then select from the drop-down menu).

At the bottom of the window in the **Help** pane you should see some information about this function. As the function name implies, it launches an activity. We need to tell it which room and which activity. We can grab those from the **Shortcuts** pane again, but first let's select the text inside the function parentheses. Why? When we select the activity out of the **Shortcuts** pane, the activity information will be pasted into the editor wherever our cursor is. If we first select the "roomId, activityId" text inside the function call, then we won't have to move around parentheses or otherwise reformat the script later.

```
Scripting.LaunchActivity(roomId, activityId)
```

After inserting the activity, there appears to be a lot going on inside those parentheses:

```
Scripting.LaunchActivity(-1--[[RedEye Pro]],
                         196--[[Activity: Watch TV]])
```

In fact, all RedEye needs to run the LaunchActivity function is a room ID (a number), and an activity ID (another number), so the following would work just fine:

```
Scripting.LaunchActivity(-1, 196)
```

The issue is that room IDs and activity IDs don't exactly come to mind easily (indeed, where have you ever seen them in the RedEye app before now?), and so while this form is compact, it is not particularly helpful. Instead, RedEye inserts a short comment after each ID so that you know the name of the room, activity, device, command, etc.

Debugging the Launch Activity Script

At this point you are probably dying to hit that **Test Script** button and see if this stuff really works. Before you do, though, let's add one more line to our script so that we can learn a bit about debugging. After the LaunchActivity line, hit <Enter> and then add the system function SetVariable:

```
Scripting.SetVariable(variableName, variableValue)
```

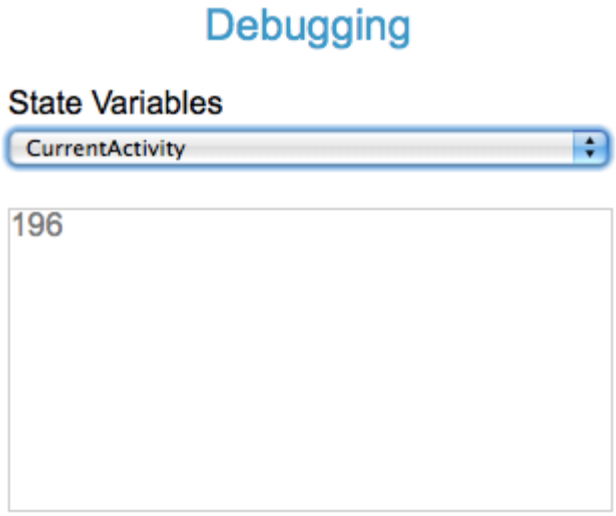
For variable name, insert the name "CurrentActivity" (within quotation marks). Then for the variable value, let's insert another function: GetActivityForRoom. This one takes an argument – room ID – so go ahead and fill that in with the room ID matching the room in which you just launched your activity:

```
Scripting.SetVariable("CurrentActivity",
    Scripting.GetActivityForRoom(-1--[[RedEye Pro]]))
```

By now you are probably getting the hang of the **Shortcuts** pane, but what exactly are we doing with this code? The SetVariable function is going to set a variable (CurrentActivity) to the value returned by GetActivityForRoom. You probably don't have a variable called CurrentActivity, but that's OK – RedEye will create one for you when you set its value for the first time. If everything goes right, then that value will be the activity ID we specified in the LaunchActivity function call on the line before.

OK – now go ahead and click the **Test Script** button.

You may notice that it takes several seconds to run this script. Why? The script execution itself is quite fast, but we have to remember what we are doing here – launching an activity. Depending on the state of your system, this could involve turning on a few pieces of equipment and waiting through delays that last up to a several seconds each before changing inputs, etc. When you are finished however, your “Watch Blu-Ray” activity should be running.



Now let’s take a gander down at the **Debugging** pane. In the **State Variables** drop-down you should have an entry for CurrentActivity, which, when selected, should display the ID for your “Watch Blu-Ray” activity.

Congratulations – you have completed your first script. Don’t forget to click the **Save** button before closing the editing window. You might also want to set the button name (“Watch BD” fits well), and icon (“Text Only” seems appropriate here), and then save your activity layout.

The PortListener Framework

Maybe you have written a couple of scripts at this point, and now you are ready to start controlling serial devices and responding to sensor inputs and that kind of thing. If so, then you'll want to take a minute or two to understand RedEye's PortListener framework.

RedEye allows you to handle inbound data and communication port scripts. By default, RedEye does not create any port scripts – you need to create them when you configure devices on each port. However, once you have created a port script, RedEye will launch that script whenever it boots up, and subsequently whenever you make changes to the port script or port configuration.

Port scripts are a little different from other scripts. They need to do a couple of things:

1. **Port scripts need to be persistent**. Most scripts do a few things and then exit. Port scripts need to hang around and wait for new information to come in from the outside.
2. **Port scripts must meter traffic on the port**. RedEye is a multi-client control system. That means any number of people can be interacting with it at one time. However, devices on the other end need to have one clear picture of what to do. We cannot mix signals, sending part of a command from one place interspersed with parts of other commands from elsewhere. All of those client connections need to funnel down into one, single-file list of marching orders.

RedEye's PortListener framework solves these two problems with a minimum amount of effort required on your part. Where possible we have provided a generic solution that works for all kinds of ports and devices, which allows you to focus on what makes your system unique. As a result, there is a fair amount going on "under the covers" and it is helpful to get at least a basic understanding of this so that you can make things work the way you need.

Behind the Curtain

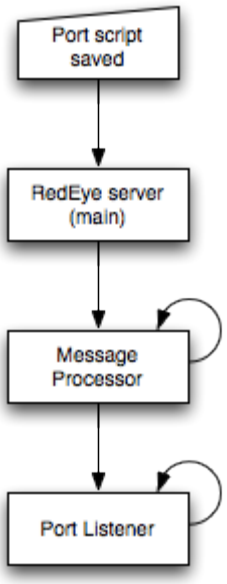
What happens when you create a port script? First RedEye saves the port script in a file. Although you cannot test or execute a port script on its own, these scripts are not transient like action scripts, but instead are stored permanently and independently like command scripts. Next, RedEye updates the port configuration table to point to your new port script. This allows RedEye to kick off

your port script when the server boots. From there it shuts down any currently executing script for the port and fires up a new set of processes using your script.

Although your port script is a single file, your change actually affects three separate processes:

1. **RedEye server.** The RedEye server is the process which handles requests from client controllers and generally governs the mainline RedEye functionality on your hardware. It determines when to start the other port control processes, and keeps track of which processes are active, shutting them down as necessary (on reboot or when a port script changes).
2. **Message processor.** When you save a port script, the RedEye server does not launch that script directly. Instead, it launches an intermediate message processing script, which is the process responsible for making sure that messages sent from various client controllers to the port are handled one at a time, on a first-in-first-out (FIFO) basis.
3. **Port listener.** Once the message processor is ready to go, it launches the port listener script that you have customized. All port listeners derive from a generic port listener which takes care of all the input/output (I/O) operations. All you need to do is provide a function or two to handle the processing of inbound data, and port listener base class[7] takes care of the rest.

[7] If you aren't familiar with object oriented programming (OOP) or the concept of a base class, don't worry – you don't really have to know too much about it to use it. Suffice it to say, Lua allows us to writes scripts which "inherit" functionality from other scripts. Your individual port listener scripts make use of some generic blocks of code that take care of opening connections and listening on ports. When new data comes in, they invoke certain functions that you specify in your script. Then when your functions are called, you can deal with the data and processing you care about without having to worry about the mechanics of managing the low-level data flow.



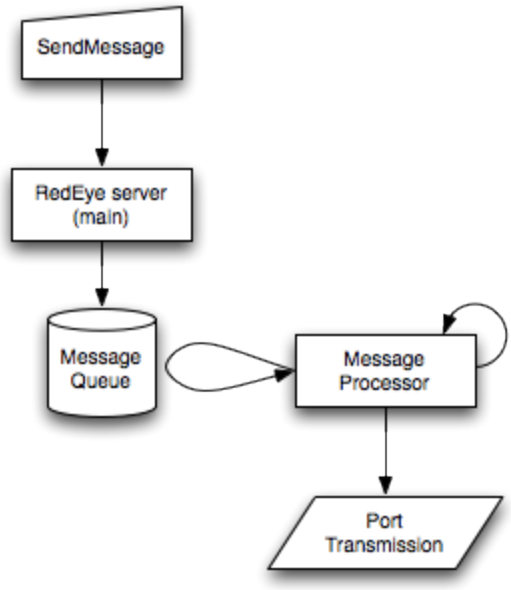
It is important to note that the code which makes both the message processor and port listener persistent is already written for you – all you need to do is handle data as it comes down the pipe. Likewise, when it is time to shut down or restart, the message processor and port listener base class take care of that work, as well. Not only do you write fewer lines of code, but you also avoid having to change your code if the internals of the RedEye server change.

Message Queuing

Now we know how everything starts up, but how do these pieces function minute to minute? There are two flows to consider here, the first being the route of an outbound message being transmitted from the RedEye server to a device you want to control.

In most cases, these messages originate as commands invoked by a client controller (although they could certainly be commands triggered by some inbound data, but we'll consider that in a moment). Whenever you call the **SendMessage** or **SendMessageToPort** functions, the main RedEye server process takes your message and writes it out to a message queue for the destination port. Each port has its own, independent message queue, and each queue works FIFO.

The message processor for the port in question is looking for changes in the message queue, and goes into action when a new message is available. It reads the message from the queue and sends the data out to the port. Then it processes the next message in the queue or waits until there is a new message available to process.



There is an important footnote here: the message processor only starts when you create the port control script. Therefore even if you do not plan to process inbound data on the port, you must create a port listener and save it before you can send any outbound messages. For example, if you add a serial port device and immediately begin writing command scripts, you will find that your commands do not have any effect (i.e., there is no message queue to receive them, or message processor to send them) until you first save the control script.

Inbound Data Processing

The second flow in the port control process is the handling of data coming into each port from external (controlled) devices. This is where your port listener-derived control script comes into play. The port listener base class opens a connection to read data coming into the port. When it receives a chunk of data, it passes this data directly to the function(s) you have defined in your script.

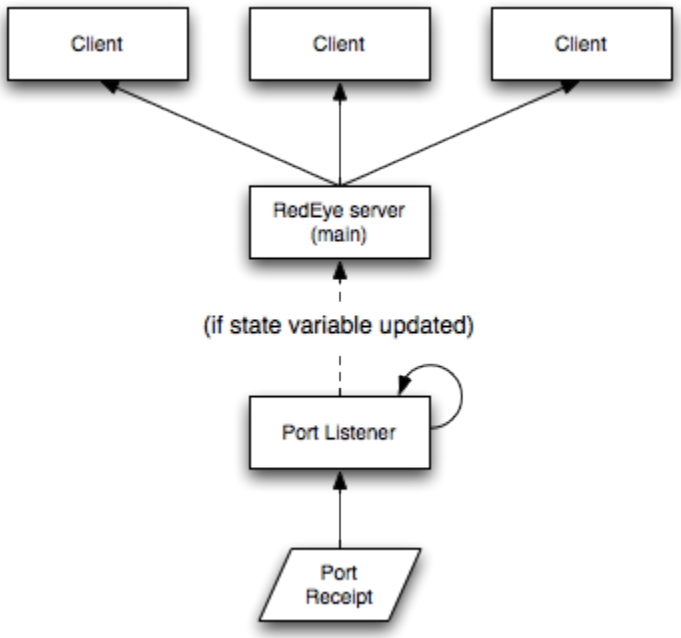
Once you have this data, it is up to you what to do with it – you can store it in a database or the file system, for example. However, one common need is to push notifications out to client controllers. For example, if you are controlling an A/V receiver and someone turns the volume knob on the receiver, you may want to update the clients to let them know that the volume level has changed. RedEye's facility for sending these updates lies in custom state variables.

Internally, RedEye uses state variables to keep track of key information about your RedEye system. For example, we store the ID of the current activity for a given room in a state variable, and we keep track of the toggle state of toggling infrared commands in state variables. When these variables change, client

controllers need to know, and so RedEye automatically sends a notification to them.

RedEye also allows you to create your own, custom state variables. These are handled in the same way as system state variables – that is, they are stored in the same place and the notification facility is the same. The difference is that you have complete control over them – unlike system state variables, you can add, delete, or change them at will.

With RedEye's state variable notifications going out automatically, if you want to update client controllers and their user interfaces, all you need to do is update the value of your state variable(s). If necessary, you can even update more than one state variable at a time. RedEye wraps your inbound data processing inside a transaction, so all changes you make within an update cycle are batched together and then broadcasted all at once. This reduces the amount of network traffic and also frees you from having to worry that clients might receive one bit of information about an update but be missing another critical piece for some period of time.



Command Scripts

All the above may sound great, but by now you probably want to get down to actually controlling a device, and that means writing some command scripts. For now the important part to remember is: if you don't edit and save a port script, there will be no message processor running on that port, and therefore any commands that you attempt to send will have not effect. As a result, your first step should be to open and save the default port script that RedEye provides.

Using the SendMessage Function

Most commands consist of a single call to the **SendMessage** function. As covered earlier, we can avoid using **SendMessageToPort** because RedEye always knows the port when we are executing a command:

```
Scripting.SendMessage("tray:open")
```

Sometimes you need to do a little pre- or post-processing around the message you send out. For example, a imagine an RS-232 Blu-Ray player that defines three commands for controlling the disc tray:

1. tray:open opens the tray
2. tray:close closes the tray
3. tray:status queries the open/closed status of the tray

This is very specific, which is exactly what we want for an RS-232 protocol. However, what if we want a single "Eject" command? The thing about the familiar Eject function is that if the tray is open, Eject closes it, and vice versa. Therefore our command script is a bit more involved (but not much). The trick here is knowing whether the tray is open or closed. For this we need to tweak the port listener script, too. Let's start there.

Let's assume that the disc player responds to the tray:status query with tray:0 if the tray is closed, and tray:1 if it is open. In the initialization section of our port listener, we would fire off a query to find out the current status of the tray when RedEye starts up:

```
Scripting.SendMessageToPort("tray:status", portId)
```

Then in our **ProcessInputData** function we store the response to this query in a state variable[8]:

```
if command = "tray" then
    if tonumber(parameter) == 1 then
        Scripting.SetVariable("TrayStatus", "Open")
    else
        Scripting.SetVariable("TrayStatus", "Closed")
    end
end
```

This should take care of keeping track of the tray state. Now we can write our Eject command:

```
require "systemScript"

local trayState = Scripting.GetVariable("TrayStatus")
if trayState == "Open" then
    Scripting.SendMessage("tray:close")
else
    Scripting.SendMessage("tray:open")
end
```

Presumably when you send one of the “tray:” commands the disc player will respond with an update on the tray status, but if it doesn’t, we have a couple of options.

1. We could add a post-processing step to the command. Again, we have a couple of choices here. We could set the state variable based on whether we have just opened or closed the tray, or we could send the “tray:status” query again.
2. We could add a pre-processing step to query the tray status. This is the safest choice, but we’ll have to give the disc player a chance to respond, which introduces a bit of delay. Here’s how that script would work:

[8] If you don’t see a need to notify clients of this change – and in the case of tray status, we may not – then you could store the value somewhere else. Since we have not yet reviewed other storage options yet, we’ll stick with a state variable for now.

```
require "systemScript"

Scripting.SendMessage("tray:status")
Scripting.Wait(200) -- give the disc player a chance to respond

local trayState = Scripting.GetVariable("Tray")
if trayState == "Open" then
    Scripting.SendMessage("tray:close")
else
    Scripting.SendMessage("tray:open")
end
```

Obviously the ideal situation is one in which the disc player keeps us updated when the tray opens and closes, but not all RS-232 devices have ideal protocols.

IP Command Scripts

One of the most exciting recent developments in the RedEye line is the addition of control via Internet Protocol (IP). IP control extends bi-directional communication to our entry-level RedEye product, so it is accessible to more people. Another great thing is that IP control allows for control of an increasingly wide array of devices without any physical port limitations — because RedEye is already on your network, anything else on the network is fair game.

There are actually a few different flavors of IP control. Today RedEye's PortListener framework includes built-in support for the three most common choices: User Datagram Protocol (UDP), Transmission Control Protocol (TCP), and Hypertext Transfer Protocol (HTTP).[9] To keep things simple, the PortListener framework for UDP and TCP control is identical to that for RS-232 control – we use the same **SendMessage** (and **SendMessageToPort**) functions to transmit commands over the network pipe, and we implement the same port listener control function to handle the responses we receive. One benefit in reusing these functions is that we can easily take a driver written for RS-232 and repurpose it for TCP or UDP. In fact, many RS-232 controllable devices that provide an IP control mechanism use the identical command set over TCP that they do over RS-232. That means less code to write, and more control options for you.

[9] Our implementation of IP control is 100% Lua scripting, so you are not limited to these three choices alone. By using the LuaSocket library (see http://w3.impa.br/~diego/software/luasocket/home.html), you can use almost any network protocol conceivable, including SMTP (for sending email), FTP (for sending files), RTSP (for streaming media), etc. RedEye may not be suitable for all types of network operations (for example, we do not recommend storing lots of large media files on your RedEye unit), but what you do with the product is ultimately up to you.

Because TCP and UDP follow the same pattern as RS-232, here we will focus on HTTP, which adds a number of new options and therefore has its own set of functions and behaviors.

Sending HTTP Requests

Under the covers, HTTP uses TCP to move data over the network.[10] Thus, you could simply use a TCP connection and deal with the HTTP messaging information yourself. But our goal is to minimize the amount of data you have to manipulate (and therefore the amount of code that you have to write).

While TCP and UDP are open data pipes that allow information to be sent and received as needed, HTTP is structured into request and response pairs. That is, HTTP clients (which in the case of IP control is your RedEye unit) send requests to HTTP servers (the devices you want to control), in reply to which the HTTP servers then send responses. In terms of RedEye's PortListener framework, we put HTTP requests in our command scripts, and handle HTTP responses in our PortListener-derived control scripts. Here we focus on HTTP requests – the command scripts.

An HTTP request message typically consists of four parts:

1. **The HTTP request type.** This identifies what the message is doing – requesting information from a server, or making modifications to information stored on a server, for example. Retrieving information — known as a "GET" request — is by far the most common type, and this is normally what happens when you type a URL into a browser address bar. Another common type of request is "POST," which sends data to the server for storage or other processing. Most of your HTTP messages will use either GET or POST, depending on what you are trying to do (retrieve information, or modify data?) and the design of the API for the device you are controlling.
2. **The request path.** This is the "location" on the server for the resource that you want. For example, the homepage for most websites is /index.html. If there is a "product" section on a website, then the main page within that section could be located a "/product/index.html" or "/product.html". Exactly how the website is organized – and even whether the path represents an

[10] In the Open Systems Interconnection model (see http://en.wikipedia.org/wiki/OSI_model), TCP is classified as a "transport layer" protocol (layer 4), while HTTP is an "application layer" protocol (layer 7). Basically, HTTP is a set of message formatting rules, while the actual communication takes place over a TCP pipe.

actual file location or just some logical organization – depends on how the server has been programmed.[11]

3. **Header fields.** HTTP headers are key-value pairs that identify particular information about the message. For example, they can specify how long the message is (the "content-length" header), and the browser/platform being used for communication (the "user-agent" header). Although you may want to insert custom values into your HTTP message headers, in practice most of these should be transparent to your application, and so RedEye will attempt to handle them automatically for you. For example, when sending POST data, the best practice is to include a "content-length" header that specifies the size of the data you are sending. Since RedEye can calculate this value easily enough based upon the message you provide, we automatically add this header and fill in the value for you.
4. **Message body.** Depending on the HTTP request type, an HTTP message body is optional. In the case of GET requests, the request path is all you need to specify the resource you want. In the case of a POST request, however, often you need to provide data to be stored, and so typically a POST request contains a message body.

In order to accommodate these new message parts, we have added two new scripting functions: **SendHttpMessage**, and **SendHttpMessageToPort**. These methods are analogous to the **SendMessage** and **SendMessageToPort** functions that work with RS-232, TCP, and UDP ports. Specifically, you should use **SendHttpMessage** inside your command scripts because RedEye can infer the target port automatically and therefore your device can be moved to other ports freely. If you must send an HTTP request message outside of a command (e.g., within an activity or macro action) then you will have to use **SendMessageToPort** and call out the port number explicitly.

SendHttpMessage takes a four arguments corresponding to the four HTTP request parts, plus two "bonus" arguments which tell RedEye how to handle the delivery of your request:

path identifies the resource path on the HTTP server.

method is the HTTP request method – usually GET or POST. The default value is GET.

[11] In addition to paths that look like file system locations, many web servers allow the use of a path extension called a "query string" which provides a list of parameters about which information to access. These query strings begin with the question mark character (?). For example, if the path to the products section is "/products/", then the full request path for product ID 7 might look like this: "/products/?id=7".

postData is the data you want to send as part of an HTTP POST request.

httpHeaders is a Lua table containing the HTTP headers. The keys to the table are the header keys, and the table values are the corresponding header values.

proxyUrl is the URL of an optional proxy server through which to route the HTTP request. Proxy servers are typically used for security purposes, and should be uncommon in home control applications.

timeout is the amount of time (in seconds) that RedEye should allow the request to linger before cancelling. Normally the timeout value for a control message should be short (the default is 5 seconds), but you can cause RedEye to wait indefinitely by passing the value -1.[12]

Because six arguments is a lot of things to juggle, we have made them all optional — you have to supply some information, but whenever possible, RedEye will handle what you do not provide with default values. Which information should you provide? The answer depends on what you are trying to do. For an HTTP GET request, typically the only thing you need to supply is a path (since the **method** argument itself defaults to GET). For a simple POST request, you may not need to specify a path, but may need to provide **postData**. In most cases, you should be able to avoid specifying **httpHeaders**, **proxyUrl**, and **timeout** altogether.

Here are a few examples:

1. Send HTTP GET request to “/query/apps” to determine which applications are installed on the device.

```
Scripting.SendHttpMessage("query/apps")
```

2. Send HTTP POST request to “keypress/Play” with no post data.

```
Scripting.SendHttpMessage("keypress/Play", "POST")
```

3. Send HTTP POST request to the default path with post data and HTTP headers.

[12] If you choose to wait indefinitely for a response, please be aware that all subsequent messages sent to this port will queue until a response comes back. A safer choice would be a relatively long timeout (e.g., 30 or 60 seconds).

```
local xmlData = [[
    <request>
        <!-- some request data here -->
    </request>
]]

local headers = {
    ["Accept-Charset"] = "utf-8",
    ["Connection"] = "close"
}

Scripting.SendHttpMessage(nil, "POST", xmlData, headers)
```

4. Send HTTP GET request to “/query/apps” with a timeout of 10 seconds.

```
Scripting.SendHttpMessage("query/apps", nil, nil, nil, nil, 10)
```

Customizing PortListener Scripts

Now that you can issue commands by sending messages, you may be wondering how difficult it is to process responses. The answer depends greatly on how complicated the protocol is for the device you are controlling, but generally speaking you can have a basic port listener up and running in just a few minutes.

Although there is a single PortListener base class which works for all types of ports (sensor, serial, IP, etc), in practice you will use port listener a little differently in each case. Sensor port listeners are the simplest example, so let's start there.

A Sensor PortListener

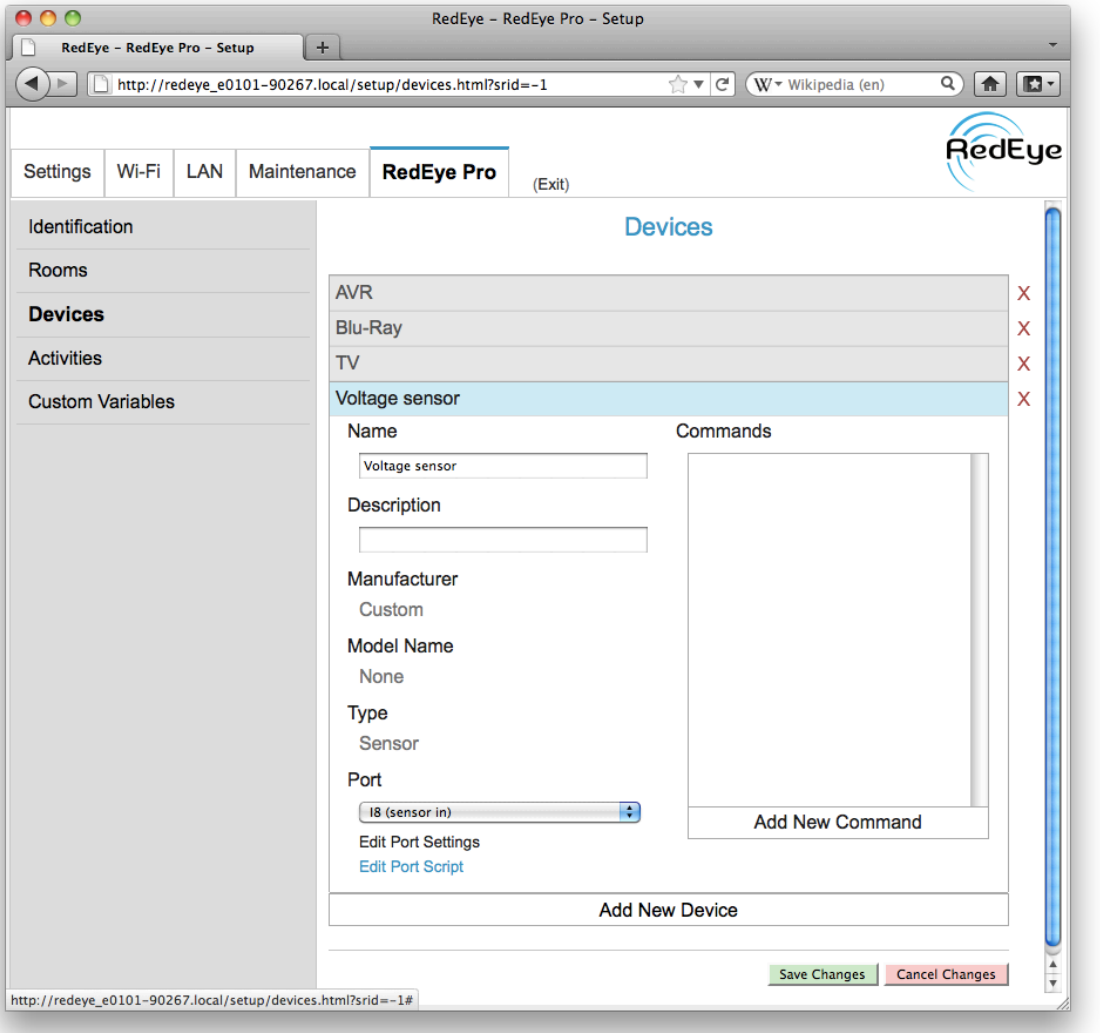
One of the "problems" that people cite for touchscreen remote controls is a lack of "hard" buttons – physical buttons on the controller that you can operate without having to look down or find a place on the screen.[13] With two-way communication through a PortListener script, you can break free from pure touchscreen control and use many other devices – light switches, strike plates, infrared sensors – to invoke commands.

For purposes of illustration, let us assume that we have a voltage sensor wired to one of RedEye Pro's contact closure sensor ports and monitoring whether the TV in our setup is on or off. With this voltage sensor in place, it would be fantastic if we could detect when someone hits the power button on the TV and respond by launching the "Watch TV" activity when the TV is turned on, or shutting down all activities in the room when the TV is turned off. Thankfully, we can wire this up with just a couple of lines of code in a PortListener script.

Let's assume that we have already plugged our voltage sensor into port I8, created a voltage sensor device on our RedEye Pro, and configured the port for

[13] Having to look down at the screen to invoke a command is indeed an issue for touchscreen controllers, but hard buttons are not the only possible solution. One feature which RedEye has offered from the beginning is the ability to use multi-touch or motion gestures as button shortcuts. Both of these options provide the ability to "press" a button without having to look down at the screen. Also, when using RedEye's browser application with a PC, you can assign shortcuts to just about any key on the keyboard, giving you many more hard button options than most dedicated remote controls.

sensor input.[14] RedEye's browser app is our script-editing vehicle of choice, so let's fire up a browser, flip over to the setup page for our Pro, and click on the Devices section. Here we can select the voltage sensor device:



When you click on the **Edit port script** button, RedEye opens a new script editing window and presents you with a template for a sensor port listener. Here we pause to note that there is no port control script running on this port yet – RedEye has only served up a template for us, and the script will not "go live" until we click the **Save** button.

[14] When you change the mode of RedEye Pro's dual-purpose ports to sensor input, RedEye Pro puts 5V on the line to power the sensor. It is important that you do not have an infrared emitter cable plugged into the port at this time, because leaving a constant voltage on the emitter could burn out the LED inside.

At this point we could save the template and we would have a fully operational sensor port listener on I8 – it just wouldn't do anything other than listen. We'll provide functionality in a minute, but first let's take a look at what the template gives us.

At the top of the script you will notice that we have a new **require** statement in additional to the usual systemScript:

```
require "systemScript"
require "portListener"
```

Remember that RedEye provides you with a PortListener base class which gives you the basic functionality required by all port listeners. You could ignore this base class and write your own, completely custom port listener, but why go to the trouble? Instead, let's understand how the base PortListener works and what you have to do to use it.

Right after the **require** statements, there are a couple of functions: SensorOpened() and SensorClosed(). This is the place where we will put in our custom code in a minute. For the moment we'll move on to the latter part of the script to see what is happening there.

The next line is perhaps the least intuitive part of the whole script, but it is absolutely vital to the proper function of our port listener:

```
local portId = arg[1]
```

It's clear enough that we are creating a local variable to store the port ID for our particular port, but what is "arg[1]" and where does it come from? From the bracket notation, we can assume that arg is a Lua table. We can also assume that it must be a global variable, otherwise how else would we be able to access it here?

Remember our earlier discussion about the launch process for message processors and port listeners? When the RedEye server application launches the message processor, it uses a library called Lua Task. In the task creation process, we can specify launch arguments to be passed to the spawned task, and these are available through the global variable "arg." When RedEye server launches the message processor, it populates the arg table with two bits of information, the first being the port ID, and the second being the path to our port listener script. Then when the message processor launches our port listener script, it populates the arg table with a single piece of information: the port ID. Thus in our script we have access to the port ID through the global variable arg.

It turns out that the port ID and our two custom functions are all that we need to make everything work. Thus, in the next line we create an instance of PortListener that we will use:

```
local listener = PortListener:new()
```

Then we tell it which functions to call when the sensors open and close. (This is where it convenient to be able to treat functions like data.)

```
listener:SetSensorOpenedCallback(SensorOpened)
listener:SetSensorClosedCallback(SensorClosed)
```

As you can see, we could rename our SensorOpened() and SensorClosed() functions whatever we want – we just have to make sure we pass the right name into SetSensorOpenedCallback() and SetSensorClosedCallback().

Now we can set the port ID:

```
listener:SetPortId(portId)
```

Everything is set up and ready to go. All we have to do is turn it on, which we do by telling the PortListener to listen:

```
listener:Listen()
```

So, that's it – we have everything here to make our sensor operational. But we still haven't put in any of the custom code to respond to the TV turning on and off. Let's go back up to the SensorOpened() and SensorClosed() functions at the top of the script. In the future we can ignore everything else, because this is where the real action happens.

Let's start with SensorOpened(), which is the function that the listener calls when the TV is powered down. If the TV is off, then we want to shutdown any running activity in the room. We can do this with a single call the the LaunchActivity system function:

```
Scripting.LaunchActivity(623--[[Media Room]], -1)
```

The first argument in LaunchActivity is the room ID, which in this case is 623 for the Media room. The second argument is the activity ID. Since we want to shut everything down, we pass -1 here. This single line of code is all that we need to have everything shutdown automatically when the TV turns off.

SensorClosed() is the function that the listener calls when someone turns on the TV. Of course it's possible that the TV could be turning on because we launched

an activity from within the RedEye app (e.g,. “Watch DVR” or “Watch Movie”). We don’t want to immediately switch to a different activity, so we’ll put a check in here to make sure that there was no activity running first:

```
if (Scripting.GetActivityForRoom(623--[[Media Room]]) == -1) then
    Scripting.LaunchActivity(623--[[Media Room]],
                             624--[[Watch TV]])
end
```

That’s all there is to it. With the addition of a voltage sensor and a couple of lines of code, RedEye Pro is now much more than a smartphone remote system – it is now smartly integrated with the hard power button on the front of the TV. You can imagine how this type of customization could extend to things like light switches and dimmer controls if you were to wire in a lighting system.

A Serial PortListener

Sensors are pretty easy to integrate – they are either on or off – but what about RS-232 devices? The PortListener base class for serial devices is the same, with just a minor initialization difference. What matters here is the code that we write to handle the inbound data.

Back in our introduction to the Lua programming language, we used an RS-232-controlled Panasonic TV to demonstrate some of Lua’s string manipulation functions. Let’s go back to that example now and look at the rest of the port listener code.

As was the case when we added the voltage sensor and edited its port script, RedEye gives us a port listener template. This one looks very similar to the last, with a couple of key differences. At the top of the page we have the same **require** statements, because we are drawing on the same scripting libraries:

```
require "systemScript"
require "portListener"
```

After that we have a place where we can enter our custom code. Rather than a pair of functions for SensorOpened() and SensorClosed(), however, we have a single function called ProcessInputData(). Again, we’ll pass on this for a minute and come back.

Next we have the now-familiar portId assignment and instantiation of the PortListener instance:

```
local portId = arg[1]
local listener = PortListener:new()
```

Then, rather than setting the sensor callback functions, we set the single data received callback function:

```
listener:SetBytesAvailableCallback(ProcessInputData)
```

And, once again we set the port ID and tell the listener to start waiting for input data:

```
listener:SetPortId(portId)
listener:Listen()
```

Now all we have to do is implement the ProcessInputData() method. We figured that one out before, but here it is again for reference. The inputData parameter we receive in the function call is a string, so most of this is just string parsing. And to notify clients of the changes, we also include some calls to SetVariable().

```
function ProcessInputData(inputData)
    local trimmedData = string.sub(inputData, 2, -2)
    local command, parameter =
        string.match(trimmedData, "(.+):(.+)")
    if command == "QPW" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Power", "On")
        else
            Scripting.SetVariable("Power", "Off")
        end
    elseif command == "QAV" then
        Scripting.SetVariable("Volume", value)
    elseif command == "QAM" then
        if tonumber(parameter) == 1 then
            Scripting.SetVariable("Mute", "On")
        else
            Scripting.SetVariable("Mute", "Off")
        end
    elseif command == "QMI" then
        Scripting.SetVariable("Input", parameter)
    end
end
```

An IP PortListener

When we examined the creation of IP command scripts, we mentioned that for TCP and UDP ports the functions we use are the same as for RS-232 ports. The same holds true for our PortListener-derived port scripts – when writing for TCP

and UDP ports, you use the same **ProcessInputData** function to handle information coming from the devices you control.

As with command scripts, things change a bit for HTTP ports. In particular, HTTP responses contain a number of different components, which we break out for easier processing.

Receiving HTTP Responses

HTTP response messages are similar to HTTP requests, and consist of the following three parts:

1. **Status line.** The first line of an HTTP response tells us the status of the request. The status line consists of a numeric code, followed by a short message. Common status lines include “200 OK” and “404 Page Not Found”. This information is useful in debugging and in confirming the successful processing of a request.
2. **Header fields.** Although formatted in the same way as HTTP request headers, there is a different set of response header key-value pairs. Normally you should not need to worry about these.
3. **Message body.** Whereas HTTP GET request message bodies are often blank, the message body is where you receive most of the interesting information from an HTTP response (e.g., the HTML to render a web page).

As with all of our other port scripts, we start off by requiring the systemScript and portListener classes:

```
require "systemScript"
require "portListener"
```

Next we have a function where we can enter our custom code. This time the function is called ProcessHttpResponse(), but before we dive in there, let’s take a look at the remainder of the boilerplate code.

Here is the portId assignment and instantiation of the PortListener instance:

```
local portId = arg[1]
local listener = PortListener:new()
```

After that, we call a special method on our port listener which is specific to HTTP ports:

```
listener:SetHttpResponseCallback(ProcessHttpResponse)
```

Finally, we set the port ID and tell the listener to start waiting for input data:

```
listener:SetPortId(portId)
listener:Listen()
```

With all the plumbing out of the way, let's go back to the ProcessHttpResponse() function. The first thing to notice is that we receive three pieces of data, corresponding to each of the three HTTP response parts:

statusCode is the numeric code from the HTTP response status line (e.g., 200, 404). This can be useful when debugging to determine why sent commands are not working properly.

responseHeaders is a Lua table containing the HTTP headers from the response. Keys in the table are HTTP response header keys, and the table values are the corresponding HTTP response header values.

responseBody is a string containing the response message body. This part of the response is the one we will use most often within our port script.

Most HTTP servers respond with some HTML or XML message which we have to parse in our script to find the information we want. The following example takes an XML response message and transforms it into an HTML document that is stored in a state variable for display on RedEye clients through an HTML control. In case of error, it dumps the **statusCode** and **responseHeaders** into a state variable for debugging.

```
function ProcessHttpResponse(statusCode,
                             responseHeaders,
                             responseBody)
    if statusCode == 200 then
        if #responseBody > 6 and
            string.sub(responseBody, 1, 6) == "<apps>" then
            local channelXml = xml.eval(responseBody)
            local htmlTable = { "<div style=\"width:100%;
height:100%; text-align:center; overflow: auto;\">" }

            for index = 1, #channelXml do
                local channel = channelXml[index]
                local channelId = channel.id
                local channelName = channel:value()
                htmlTable[index + 1] =
    "<a href=\"redeye#" .. tostring(channelId) .. "\">" ..
    tostring(channelName) .. "</a><br /><br />"
            end

            htmlTable[#channelXml + 2] =
    "<a href=\"redeye#-1\">(refresh channel list)</a></div>"

            Scripting.SetVariable("roku-channelHtml",
                                  table.concat(htmlTable))
            Scripting.SetVariable("roku-selectedChannelId", "")
        end
    else
        Scripting.SetVariable("roku-error",
                              tostring(statusCode) .. ": " ..
                              table.concat(responseHeaders))
    end
end
```

Handling Message Fragments

As control protocols have become more sophisticated, the amount of information transmitted within the protocols has grown. For example, because infrared is a relatively slow control medium, infrared signals are short and contain only a little data. With IP control – which is both bi-directional and also significantly faster than RS-232 or infrared – sometimes request and response messages can be several thousand characters long.

One issue with increasing length is that individual request and response messages may need to be broken into small chunks – a process called fragmentation. RedEye and the underlying IP software stack hide most of the fragmentation issues from your code, but sometimes when dealing with TCP or UDP connections you may receive response fragments from devices you are controlling.

To simplify the management of these message fragments, RedEye's scripting library contains a new class called **StringAggregator**. StringAggregator accumulates message fragments until it reaches a predetermined message termination string. Once it hits this message termination string, it combines all of the message fragments together and returns the complete message to you for processing.

The following example illustrates the most common usage of StringAggregator. Here we have the ProcessInputData() function for a TCP port. If fragmentation were not an issue, we would write the function as follows:

```
function ProcessInputData(inputData)
    local id, sequence, message =
        string.match(inputData, "(.-)/(.-)/(.+)/.*")

    if (id and sequence and message) then
        -- process the message here
    end
end
```

If upon closer inspection of the protocol documentation or through testing we discover that we are receiving message fragments (or even combinations of multiple messages) in our ProcessInputData method, then we add a couple of lines to our script.

First, we need to require the StringAggregator library:

```
require "stringAggregator"
```

Next, we create a new StringAggregator instance:

```
local inputAggregator = StringAggregator:new()
```

Optionally, we also tell this StringAggregator about our message termination string. The default message termination string is "\r\n" (which is a common IP message termination string), but this could be just about anything. In this case, let's assume it is "</message>":

```
inputAggregator:SetLineTerminationString("</message>")
```

Now inside our ProcessInputData method we pass all our data through the StringAggregator instance, and it returns complete messages to us. Then we can loop through the complete messages we receive and process them as we had originally planned:

```
function ProcessInputData(inputData)
    local dataStrings = inputAggregator:ProcessData(inputData)

    for key, value in ipairs(dataStrings) do
        local id, sequence, message =
            string.match(value, "(.-)/(.-)/(.+)/.*")

        if (id and sequence and message) then
            -- process the message here
        end
    end
end
```

Other Scripting Tools

In addition to the built-in system functions, RedEye provides a number of other scripting tools that you can use. Some of these tools could merit an entire manual unto themselves, and most are only really necessary when you are doing some major programming, so we will leave the details for you to explore. Still, a quick introduction to what is available is in order.

File System Access

Lua's Input/Output (I/O) library is available for your use, and RedEye gives you access to a sizeable chunk of the file system, as well as some device ports. Lua.org provides an online tutorial on how the built-in I/O library works:

http://lua-users.org/wiki/IoLibraryTutorial

As far as access goes, all of your Lua scripts are available in RedEye's /devicedata partition under the /lua subfolder. When you access the I/O library the /devicedata/lua folder becomes your root folder. This means two things:

1. You are limited to this folder and its subdirectories.
2. If you want to access "/devicedata/lua/tmp," the correct path is "/tmp".

Proper Use of the Flash File System

Although you can write freely to the file system, it is important to understand a bit about writing to flash memory and the long-term implications of frequent writes. RedEye stores its programs and data on a NAND flash chip rather than use a mechanical disk. NAND flash has several advantages – it is fast, it does not fail mechanically, it is compact and consumes little power, etc.

NAND flash also has some tradeoffs. For example, it has a relatively high percentage of "bad blocks" – unusable chunks of memory. A brand new chip may have up to 1-2% of memory in an unusable state, and these bad blacks grow over time. In addition, each time a block of memory is overwritten, there is a very slight chance that it will become unusable.

RedEye uses a file system designed to work with the deficiencies of NAND flash,[15] so you don't have to worry about your data going away. However, there is

[15] Specifically, we use the Journaling Flash File System, v2 (JFFS2). If you are curious, you can learn more about JFFS2 here: http://sourceware.org/jffs2/

one unavoidable implication of using NAND flash, which is that over time the amount of available storage space slowly decreases as more and more blocks become unusable.

The rate at which storage space shrinks is directly related to the frequency of database writes. Within the RedEye server application we take care to avoid writing to the flash chip too often, and as a general practice, you should too.

All this is not to say that you should be afraid to store information in flash. It is there to use, and we can write millions of times without a noticeable impact on storage. Still, we should be careful that we do not write to flash more often than necessary.

One option that we use often in the RedEye server application which may work for some of your scripts is to store data in the temporary file system. When you write out to the /tmp directory, you are not actually writing to the flash memory. Instead, everything stored in /tmp is kept in memory. You can access it just as with any other file, but when RedEye reboots the information there is cleared out. That is usually fine for things such as the state of RS-232 devices, because under normal circumstances you need to retrieve new state information after boot anyway.

One example of RedEye's use of the /tmp file system is the message queuing system we covered in our discussion of the PortListener framework. RedEye creates its message queues as files in the /tmp/queues directory. When the system reboots, these are automatically cleared, but that is acceptable because we would need to clear out these queues on reboot anyway. Also, since we write to these queues frequently it is particularly important that we save the flash memory from this constant activity.

Accessing Devices Directly

Although RedEye provides a convenient and managed way for you to access serial ports using the message queuing system, your ability to access the file system using Lua's I/O library means that you can directly read from or write to I/O devices in your own scripts. Should you find a need to do so, RedEye currently gives you access to the following devices:

1. **/dev/ttyS1**. This is the S1 serial port.
2. **/dev/ttyS2**. This is the S2 serial port.
3. **/dev/null**. The "null" device provides a black hole where anything you write is consumed immediately. May be useful for testing in some situations.

4. **/dev/zero**. The “zero” device always returns zeros when read from, which can be useful if you need to zero out some memory or fill a stream with empty data for some period of time.

Multithreading

Although we have not discussed it directly, in out treatment of the PortListener framework we mentioned that RedEye uses Lua’s Task library to create messages processors and port listeners. We referred to these as processes, but technically these POSIX threads and not full system processes. Whatever the case, at times you may find it convenient to be able to spin off a separate thread to do some work while your main script thread executes. RedEye gives you this option through the Lua Task library:

http://luatask.luaforge.net/

One point to note with the Lua Task library is that we have made a slight modification to guarantee delivery of messages between tasks. As a result, when you call the task.create() function, you will need to pass in an extra argument to specify the name of the task. RedEye uses this name to create the message queue for your task, and therefore the name must be unique. As long as you avoid names that are purely numeric (e.g., “12”, “57”) you should not stomp on any internal RedEye task queues. Here is an example of task creation with a reasonably unique name:

```
local portId = arg[1]

local channelQueryScript = [[=
    require "systemScript"
    local portId = arg[1]
    Scripting.Wait(5000)
    Scripting.SendHttpMessageToPort(portId, "query/apps")
]]

local taskName = "port" .. portId .. "-channelQueryTask"

local taskId = task.create(taskName,
                           channelQueryScript,
                           { portId })
```

XML Processing

The use of XML in data formatting is becoming increasingly more common as Internet technologies begin to replace older, proprietary formats. To this end, RedEye includes the Lua XML library:

http://viremo.eludi.net/LuaXML/

JSON Processing

If you receive data in JavaScript Object Notation (JSON) format or need to send data to another device using JSON, we have added a basic utility to convert easily between Lua tables and JSON strings. Documentation is available from the library homepage:

http://json.luaforge.net/

You can even use the JSON RPC format outlined here (although doing so would circumvent RedEye's normal IP port handling process).

SQLite Databases

File system storage is fine, but when you need something with a little more structure, SQLite is a great option. RedEye actually uses a SQLite database for its own storage, and with the LuaSQLite3 library you can do so with your own data, as well. You can read more about SQLite here:

http://www.sqlite.org/

And you can read more about LuaSQLite3 here:

http://lua.sqlite.org/

Part III: Advanced Interface Customization

Image Controls

RedEye's scripting engine enables more sophisticated user interface elements, of which custom image controls are the first and most common. The unifying theme of these controls is that they are tied to state variables, which allows you to update them at runtime as the state of your system changes. State variables are *the* mechanism for broadcasting changes to all connected RedEye client controllers, and so we make extensive use of them here.

Conceptually, image controls are simple: they display graphics that you specify by providing a path to the graphics file in the form of a URL. Beyond basic display, however, we allow you to assign an action to your images, which means that you can use them as custom controls, mimicking the functionality of buttons or other interface elements. With some scripting tricks, image controls provide a whole range of interesting possibilities.

Displaying Images

The most basic kind of image control is one that displays a static image. Because the image does not change, you really do not have to deal with scripting, and configuration is straightforward. A common example of a display-only image is a custom background.

Image size

When choosing the images you will display, there are a couple of considerations. The first is proper sizing. With button controls, RedEye gives you a set of pre-defined sizes and you do not have to worry about how everything fits together other than placement on the page. When working with custom images, you have complete control over both position and size.

RedEye activity layouts are 320 pixels wide. The actual display width differs depending on the resolution and size of screen on the controller, but everything scales off this default resolution of 320 pixels. We recommend you create your images at twice the activity layout resolution so that they scale well on other devices. For example, iPad activities display 40% larger than on the iPhone.

Screen height is a bit trickier than width, because the RedEye application scales activity layouts based on width, but different devices provide displays of different heights. On an iPhone you get 368 pixels in height with both toolbars visible. If you hide the bottom toolbar this space increases 416 pixels. If you hide both toolbars, you can use 460 pixels. On an iPad in landscape orientation you have

500 pixels of vertical height, and in portrait orientation you have 683 pixels (remember: these are not iPad screen pixels, but rather RedEye activity layout pixels — you need to multiply by 1.4 to get to actual iPad pixels). As a result, if you want to create a background image that fills up the entire screen on an iPad running in portrait mode and also provides a 2X oversampling rate for high resolution displays, your image should measure 1000 pixels wide by 1,366 pixels high.

Creating images with resolution far beyond what is necessary is not helpful, either – generally speaking, the larger the image, the larger the file size. With larger files load times are slower and we use up more memory on the clients.

Proper Scaling

When you configure an image control, you have complete freedom to set its dimensions with your activity layout. Normally you would want to retain the same height-to-width ratio as the original image. When you do, RedEye will scale your image appropriately depending on the space available for displaying activity layouts on each client controller (iPhone, iPad, PC, etc).

What happens if you enter dimensions with a different height-to-width ratio? The RedEye app will do its best to render your image attractively, but the result may not always be what you want. The best option is to plan ahead and create your images with the final height-to-width ratio in place. The following images illustrate how RedEye handles scaling. The power button is the image being scaled, and the gray area represents the rectangle defined by the width and height dimensions we give the image.

Original Image
100 pixels wide x 100 pixels tall
Suitable for an activity button 50 pixels wide x 50 pixels tall

Oversize Image
200 pixels wide x 200 pixels tall
Image appears pixilated

(*Note*: this is how the image looks on higher resolution controllers when the original image is not at least twice the size specified in the activity layout)

Wide Image
150 pixels wide x 100 pixels tall
Image is displayed with proper proportions, but does not fill all horizontal space

Tall Image
100 pixels wide x 150 pixels tall
Image is displayed with proper proportions, but does not fill all vertical space

Minimum Button Size

When designing your layouts, please keep in mind that it can be difficult to tap on controls that are too small. As a result, we recommend that your image controls be at least 50 pixels square. To reiterate the above point about scaling, the images you use should then be at least 100 pixels on a side to accommodate tablets and other high-resolution devices.

Image hosting

The second consideration when using custom images is hosting. RedEye hardware is not designed to store large graphics files, so you need to host them on an external server. Because of security limitations in most browsers, you need to use a web (HTTP) server.

When configuring your hosting server, you should not require client authentication. The physical location of the server does not matter – it can be on your local area network, or on a publicly-accessible website – but the RedEye app needs to be able to access files without a password.

File formats

Generally speaking, if your browser can display an image file, the RedEye app can, too. GIF, JPEG, and PNG files are the most common, and there are a number of drawing and illustration programs you can use to generate these. We recommend using the PNG file format where possible:

1. PNG uses lossless compression, so your images will look as sharp and clear as the original illustrations.

2. You can use real alpha-blending (variable levels of transparency) for non-rectangular shapes or to add depth to your layouts.

File caching

Because images are stored out across the network somewhere and may take up to a few seconds to load, the RedEye app will cache recently used images in order to speed up the display of your layouts. As a result, if your image files change, you must also change their corresponding URLs. For example, if you have a background image called “background.png”, RedEye will load this file over the network the first time it is used, and then keep a local copy for future reference. If you make a design change to the image and upload it to the hosting server without changing the URL, RedEye will continue to load local copy of the file, and you will not see your changes.

Please note that changing the URL is not the same as changing the file name. The best way around this caching issue is to make use of a “query string” at the end of your URL. Query strings are those bits of text in a URL following the question mark (?), usually used to pass data to a web server. In this case we do not need to pass any data to the web server; instead, we are taking advantage of the fact that the web server will still serve up our image even though we tack on some arbitrary data to the URL in the form of a query string. Thus, the following query strings all load the same image file:

http://thinkflood.com/media/RedEyeBackground.png

http://thinkflood.com/media/RedEyeBackground.png?1

http://thinkflood.com/media/RedEyeBackground.png?version=1

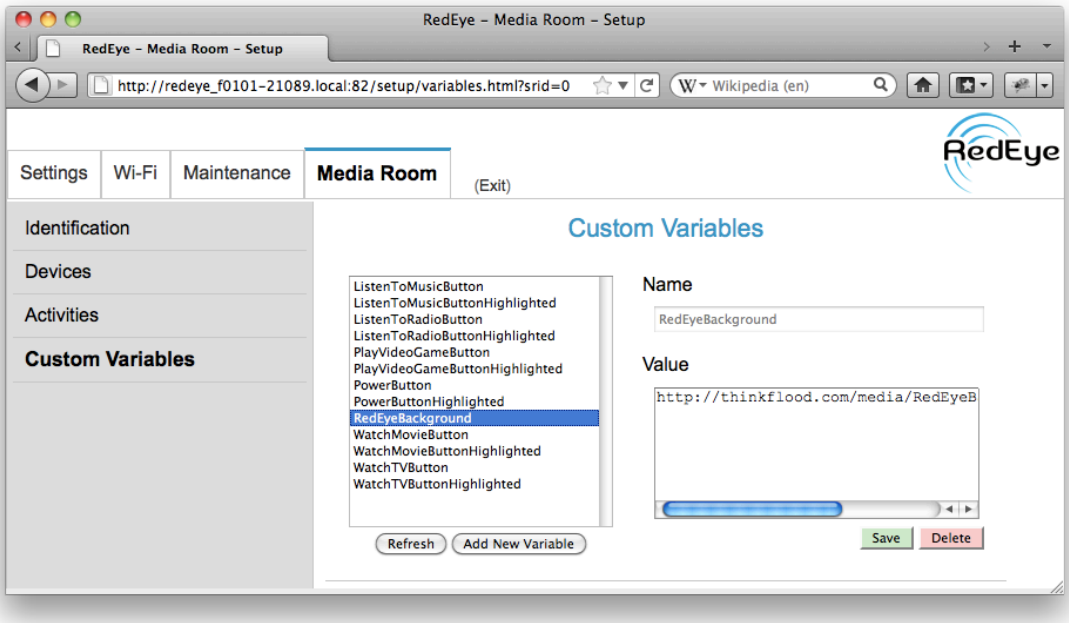
Changing the query string in this way is enough to tell the RedEye app that it should reload the image from the web server rather than from its local cache.

Adding a Background Image

To illustrate how the whole process works, we will add a background image to an activity layout. Following our rules above for image size, we are starting with a background image that is 640 pixels wide by 1,370 pixels high. Our file uses the PNG format and we have put it on a publicly-accessible web server:

http://thinkflood.com/media/RedEyeBackground.png

The next step is to create a custom variable containing this URL. We can do this from an iOS client, but the browser application is much more convenient.



To create a variable using the browser application:

1. Open the browser setup page
2. Click on the tab for your RedEye
3. Select the “Custom Variables” section
4. Click the “Add New Variable” button
5. Type a name (e.g. “RedEyeBackground”) into the “Name” field
6. Paste the URL into the “Value” field
7. Click the “Save” button

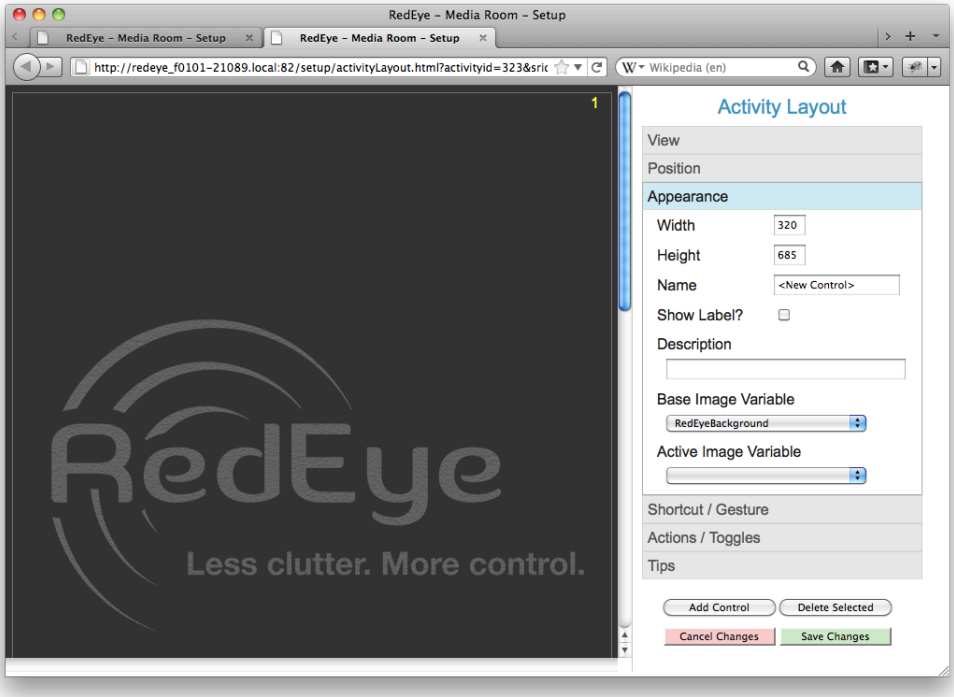
If you are adding more images, this would be a convenient time to create custom variables for them.

Next we will navigate over to the “Activities” section. Choose an activity from your list to which you would like to add the custom background and click on the activity layout button.

To add your image:

1. Click on the “Add Control” button
2. Choose the “Image” control type and click “OK”
3. Select the “Appearance” section of the control details sidebar
4. Enter 320 for the width and 685 for the height. (At this point you should see a large picture of a semi-transparent icon showing a piece of paper with a photo and a paperclip – this is RedEye’s default image, which will display whenever your image custom variable does not point to a valid image URL.)

5. Select our previously-defined custom variable (“RedEyeBackground”) from the “Base Image Variable” drop down list. (After a brief second the image should load.)



6. Click outside the image to deselect it – the image will move to the background behind the other elements of your layout.

Adding a Custom Button

Background images are great, but images can do a lot more than add some aesthetic sizzle. One common use for images is to create custom buttons. With proper use of transparency your buttons can take on just about any shape, and with a few scripting tricks, they can do just about anything, too.

We'll continue to customize the activity from the last example. For purposes of illustration, let's assume that RedEye is controlling a lighting system in the room. Scene 2 for the lighting system is full brightness, and Scene 3 is dimmed for watching television and movies. We want to create a custom button which does the following:

1. Allows us to switch between Scene 2 (lights on) and Scene 3 (lights dimmed)
2. Displays the "state" of the lights – green when they are on, and red when they are dim.

The first step is to add a couple of custom variables:

1. PowerButton. This is the base variable for our image. Its initial value will be http://thinkflood.com/media/PowerButtonRed.png, which is a red button image (lights dimmed).
2. PowerButtonHighlighted. This is the active variable for our image. The URL is http://thinkflood.com/media/PowerButtonGold.png, which is a gold button image.

You might have been expecting a green button image for PowerButtonHighlighted. It's important to draw a distinction here between the highlight (active) image and the other image state. The active image is the one we display when the image is being tapped – it provides a bit of animation, but once our finger is raised, the image goes back to the URL identified by the PowerButton variable. In order to display a green image, we'll have to update the value of PowerButton itself – we'll get to that in a moment.

Once we have our variables in place, we can open up the activity layout for editing. The first order of business is to add a new image control, just as we did with the background image. This time around, we'll use the following values:

Position
Top: 11 pixels
Left: 251 pixels
Z-Index: 2

Appearance
Width: 50 pixels
Height: 50 pixels
Name: Lights
Show Label: Yes
Base Image Variable: PowerButton
Active Image Variable: PowerButtonHighlighted

Z-Index

Right away you probably noticed a couple of new fields that we are using. First, let's take a look at the "Z-Index." Z-Index is a computer graphics term which identifies the "depth" of an image. If you think of the screen as a grid system with X (horizontal) and Y (vertical) coordinates, then you can imagine adding a third, Z-axis which extends from the screen towards the viewer. Z-Index defines an image's position on the Z-axis. Lower values are closer to the screen; higher values are closer to the viewer.

The default Z-Index is 1, and that is fine for our background image – we want it to be in the back, after all. On the other hand, we want to ensure that our new power button remains on top of the background image, so we need to give it a higher Z-Index value. You can pick any number up to 1000, but 2 works fine.

It is important to note here that Z-Index only applies to image controls. Other types of controls (buttons, for example) are not useful when stacked on top of each other, and therefore do not need a Z-Index. As a result, image controls always display *below* other images. In other words, all non-image controls have an effective Z-Index of 1001.

Image Label

Just as with button controls, images can display a text label if you wish. When you choose the “Show Label” option, the image name appears just below the image, and the image’s boundaries expand to include the label. The Height value which you specify applies only to the image itself, however. In this example, the power button is a perfect square, and we have given it dimensions of 50 pixels on each size. Once you select the “Show Label” option, the image boundary rectangle automatically expands to include the label.

Active Image Variable

All images require a Base Image Variable, which identifies the path to the file displayed most of the time. Optionally, your images can also specify an Active Image Variable, which identifies the image displayed when the image is tapped. Even if you do not specify an Active Image Variable, you can still assign an action to your image and have it function as a button; the only thing you lose is the animation which indicates that the image is being tapped. Although this extra bit of animation may seem like frivolous “eye candy,” providing some visual feedback is quite helpful when controlling the system, particularly on touchscreen devices.

There are many different ways to create the illusion of pliability in a custom button, including shading and positional translation. Alternate coloring is one of the easiest effects to manage, as the base and active images can maintain the same dimensions. In our illustration, we are using a gold coloring, which works equally well against both the red and green versions of our button. As a result, we do not have to change the active image when we change the base image (although you could certainly do so if you wanted to).

Image Action

At this point we have an image control which changes color when clicked, but otherwise does not do anything interesting. To give the image some real functionality, we have to assign an action to it.

RedEye buttons provide several different action options, including macro and toggle buttons, both of which allow for more than one action on a single button. With the advent of RedEye scripting, it is possible to achieve macro or toggle functionality through a script, so these additional types are extraneous and therefore not available for image controls. That said, there are still two action types: normal, and repeating. Repeating actions are particularly useful for things like volume or channel controls that “ramp” when you press and hold them. You can create a repeating image control in the same way as a non-repeating (normal) image control – all you need to do is choose “Repeating” from the dropdown menu.

Actions themselves contain either a single command or a script. Choosing an image action is the same as choosing a button action. You can also delete an existing action to revert the image back to a display-only state.

The control we are creating is effectively a toggle button, so we need to create a script action. The following script should do the trick:

```
require "systemScript"

local sceneName = Scripting.GetVariable("SceneName")
if sceneName == "Scene 3" then
    Scripting.SendCommand(453--[[Device: Lights]],
                          435--[[Command: Scene 2]])
    Scripting.SetVariable("SceneName", "Scene 2")
    Scripting.SetVariable("PowerButton",
        "http://thinkflood.com/media/PowerButtonGreen.png")
else
    Scripting.SendCommand(453--[[Device: Lights]],
                          437--[[Command: Scene 3]])
    Scripting.SetVariable("SceneName", "Scene 3")
    Scripting.SetVariable("PowerButton",
        "http://thinkflood.com/media/PowerButtonRed.png")
end
```

Let’s take a look at what is going on here. First, we have a state variable which tells us what the current lighting scene is – this is critical if we are going to toggle between scenes. In this case, if the lighting is on Scene 3 (lights dimmed), then we switch to Scene 2 (lights at full brightness); otherwise, we switch to Scene 3. Notice that after we switch scenes we update the SceneName variable.

Once we have the conditional logic for the toggle in place, swapping the red image out for the green one is pretty easy. Since images are tied to variables, all we have to do is update the PowerButton variable with a new URL. Here is how the two different toggle states look on an iPhone client:

Creating a Launchpad Activity

So far we have made some minor visual improvements to activity layouts using images, but the real power in these controls is that, together with some scripting customization, you can create entire new interfaces to your RedEye system. One example of this is a launchpad – basically a home screen for activities within a RedEye room.

The concept behind the launchpad is pretty simple: it provides a number of buttons to launch activities or perform common functions (such as change lighting scenes). The launchpad should be your primary window into the room's activities; that is to say, when you open the RedEye app and select a room, either the currently running activity is visible there, or you see the launchpad.

The key to creating a launchpad on RedEye is recognizing that the launchpad itself is an activity. It has buttons which launch other activities (easily accomplished through the **Scripting.LaunchActivity** function), and when other activities shut down, they launch the launchpad.

In this example, we'll create a launchpad for a room that has five activities:

1. Watch TV
2. Watch Movie
3. Listen to Music

4. Listen to Radio
5. Play Game

Our launchpad will use a custom background, so we'll need an image (and custom variable) for that. We'll also need the base and active variable images for each of the activity buttons – ten in total. Here is a list of the variables and URLs:

Variable Name	URL
RedEyeBackground	http://thinkflood.com/media/RedEyeBackground.png
WatchTV	http://thinkflood.com/media/Television.png
WatchTVHighlighted	http://thinkflood.com/media/TelevisionHighlighted.png
WatchMovie	http://thinkflood.com/media/FilmReel.png
WatchMovieHighlighted	http://thinkflood.com/media/FilmReelHighlighted.png
ListenToMusic	http://thinkflood.com/media/Headphones.png
ListenToMusicHighlighted	http://thinkflood.com/media/HeadphonesHighlighted.png
ListenToRadio	http://thinkflood.com/media/Radio.png
ListenToRadioHighlighted	http://thinkflood.com/media/RadioHighlighted.png
PlayGame	http://thinkflood.com/media/GameController.png
PlayGameHighlighted	http://thinkflood.com/media/GameControllerHighlighted.png

Next, we need the launchpad activity itself. By creating an activity of type "Other" we get a blank layout to start with. Our launchpad will not include any launch or shutdown actions – we are using it simply to display some buttons. The logic for turning on and off our other activities should be contained within their launch and shutdown action sequences, and RedEye will know what to do when we switch from the launchpad to another activity and back, just as it knows how to switch between other activities.

Once we have our activity, we can add our image controls. We will need six of them, as follows:

Background Image
Top: 0
Left: 0
Z-Index: 1
Width: 320
Height: 685
Name: (blank)
Show Label: No
Base Image Variable: RedEyeBackground
Active Image Variable: None
Action: None

Watch TV Button
Top: 10

Left: 40
Z-Index: 2
Width: 100
Height: 100
Name: Watch TV
Show Label: Yes
Base Image Variable: WatchTV
Active Image Variable: WatchTVHighlighted
Action: Script (more on this in a moment)

Watch Movie Button
Top: 10
Left: 180
Z-Index: 2
Width: 100
Height: 100
Name: Watch Movie
Show Label: Yes
Base Image Variable: WatchMovie
Active Image Variable: WatchMovieHighlighted
Action: Script

Listen to Music Button
Top: 125
Left: 40
Z-Index: 2
Width: 100
Height: 100
Name: Listen to Music
Show Label: Yes
Base Image Variable: ListenToMusic
Active Image Variable: ListenToMusicHighlighted
Action: Script

Listen to Radio Button
Top: 125
Left: 180
Z-Index: 2
Width: 100
Height: 100
Name: Listen to Radio
Show Label: Yes
Base Image Variable: ListenToRadio
Active Image Variable: ListenToRadioHighlighted

Action: Script

Play Game Button
Top: 240
Left: 110
Z-Index: 2
Width: 100
Height: 100
Name: Play Game
Show Label: Yes
Base Image Variable: PlayGame
Active Image Variable: PlayGameHighlighted
Action: Script

The scripts for these buttons are all more or less the same, and they are quite easy to create – one line only, configurable completely using the drop-down menus in the script editor. Here is our example for the Watch TV Button:

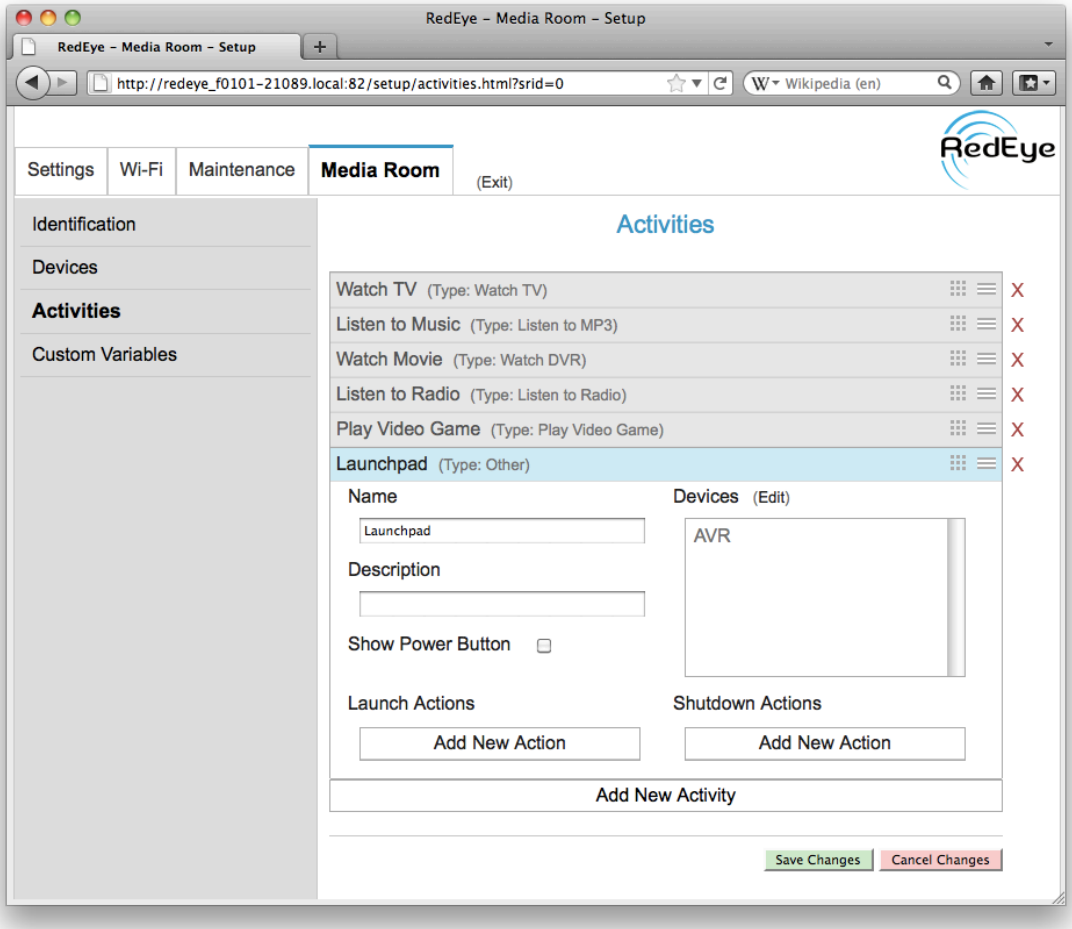
```
require "systemScript"

Scripting.LaunchActivity(0--[[Room: Media Room]],
                         323--[[Activity: Watch TV]])
```

(If you are trying this at home, please note that your script will be slightly different depending on the ID for your room and Watch TV activity.)

Disabling the Default Power Button

The above takes care of the overall look and functionality of our launchpad, but our launchpad isn't really a launchpad if we have to launch it first. What we want to do here is make sure that once the launchpad is fired up for the first time, there is no way to kill it. In order to make that happen, we need to disable RedEye's default power off button. Doing so is quite simple – just open the activity settings for our launchpad and turn of the **Show Power Button** switch.



After saving changes, your launchpad should look like this:

At this point we have our launchpad activity, but it is only half functional – we can use it to start other activities, but when those activities shut down, they do not return to the launchpad. The solution to this problem is to create a custom power off button which launches the launchpad.

We will need to add this custom power button to each of the activities. Also, we will want to turn off RedEye's default power button for each activity to ensure that only our custom power button (the one that returns to the launchpad) is available.

Because we can use the power button images we created for the earlier custom button example, and the script to launch the launchpad activity is similar to the ones we use to launch the other activities, we leave the creation of these power buttons to the reader.

Slider Controls

Slider controls are an attractive alternative to simple buttons for a couple of reasons:

1. **Sliders provide direct access to specific equipment settings.** For example, consider a basic volume slider for an AVR in comparison to a pair of volume up/down buttons. With the slider, you can adjust directly to the 25% volume level by dragging to a position one quarter of the distance between the left and right edges of the slider. With volume up/down buttons, you can only increment or decrement relative to the current setting.
2. **Sliders display the current state of the equipment.** In the volume slider example, a quick glance tells us that we are at 25% volume, and therefore that we can as much as quadruple the volume level if we wish.

It is important to note that the first point above signals a limitation of slider controls: it is not practical to construct a slider without the ability to directly set the volume level on your AVR. In other words, if your AVR provides only volume up and volume down commands, then you should stick with the two-button approach.

Value Range

By definition, sliders control your equipment across a range of values. As such, properly configuring a slider control first requires setting the slider range. All ranges have a minimum value, a maximum value, and an increment.

Minimum Value

The minimum value of your slider control should be the lowest value that your equipment allows (alternatively, it could be the lowest setting that you wish to

accommodate). This minimum value can be negative; for example, if you are creating a volume slider for an AVR and the volume is measured in decibels, the lowest setting might be something in the range of -80 or -90 dB. The minimum value can also be a fractional number. For example, if you have a light dimmer that ranges between 0% and 100% but you want to put a floor at 10%, you can set the minimum value to be 0.1.

Maximum Value

Similarly, the maximum value of your slider control is the highest value that your equipment allows (or the highest setting that you wish to accommodate). Again, the maximum value can be negative or fractional; the only restriction here is that your maximum value should be greater than your minimum value (i.e., your slider cannot range from -1 to -2, but instead should range from -2 to -1).

Increment

The slider increment is the smallest possible step along the way between your minimum and maximum values. For example, if you have a volume slider that ranges between -80 dB and +15 dB in steps of 1 dB, then the minimum value is -80, the maximum value is 15, and the increment is 1. Or, if you have a dimmer that ranges between 0% and 100% in 1% increments, the minimum value is 0, the maximum value is 1, and the increment is 0.01. RedEye does not accommodate a “sliding scale” but rather assumes that the increment is constant across the range of your slider control.[16]

Use of State Variables

We rely upon a state variable to store the current value of each slider control. State variables are a logical choice here because RedEye automatically synchronizes state variables across client controllers. As a result, when you change the state variable on one client, you see the value updated on another; in other words, when the state variable behind a slider changes in one place, you see the slider update everywhere else within the system.

Slider controls are particularly effective when used in a system that receives feedback from devices. For example, if you have a climate control system integrated via RS-232, when there is a change to the set temperature outside the RedEye system you can see that change reflected in your temperature slider as

[16] If you are controlling a device that needs to vary the increment, you can still do so. In this case you should set the increment to be the smallest possible step, and then you will need to round to acceptable increment values as necessary as the value of the slider changes.

long as you process the temperature change into the same state variable you use for your slider control.

It is important to note that simply setting a state variable is not enough to control your equipment; you must also do something with the value that the slider control sets.

Whenever you drag the handle on a slider control, RedEye will do two things:

1. Update the slider's current value state variable.
2. Invoke the command or script you specify for the slider action.

To function appropriately, the command or script you use should use the current value state variable. For example, the following script reads the value of a volume state variable and then sends an RS-232 message to set the volume level to the value read from the state variable:

```
require "systemScript"

local volumeLevel = Scripting.GetVariable("nad-t765-volume")
Scripting.SendMessageToPort("\rMain.Volume=" .. volumeLevel
                            .. "\r",
                            13--[[Port: S2]])
```

You can encapsulate this volume level message in a command, or you can write the script directly into the slider action itself. One point to consider is that if you create a standalone command (e.g., "Set Volume"), that command will always be visible to users of your system, but will have no effect when executed alone (since the command itself does not change the volume variable). As a result, it is somewhat cleaner to embed the script directly in your slider control.

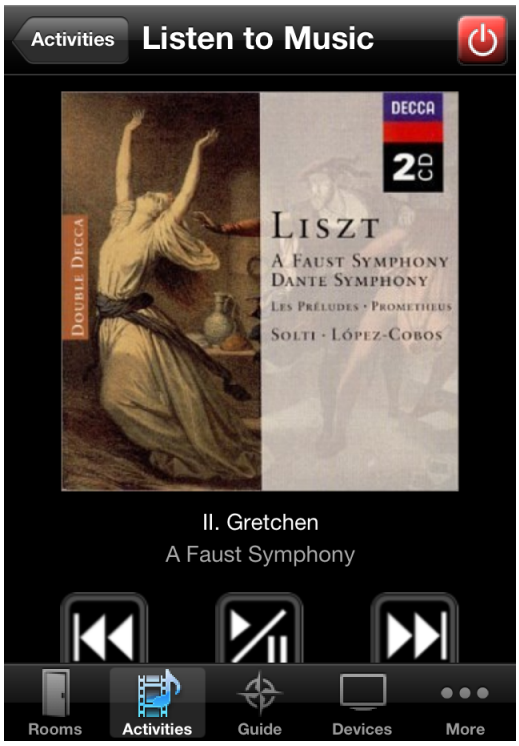
Label Controls

Most layout controls provide the option to show a text label displaying the name of the control. However, there may be times when you want to display some text not directly associated with another control, or when you want to present dynamic text. Label controls provide this functionality. You can even use label controls as hyperlink-style buttons.

State Variables Once Again

If you have been reading straight through, you will have noticed that the overwhelming theme for activity controls is our use of state variables. Label controls are probably the simplest example of this pattern.

RedEye synchronizes configuration between the control processor and connected clients automatically, and state variables are part of that synchronization. As a result, a state variable change from any source – be it a client controller or some direct input from a controlled device – automatically propagates across the network.



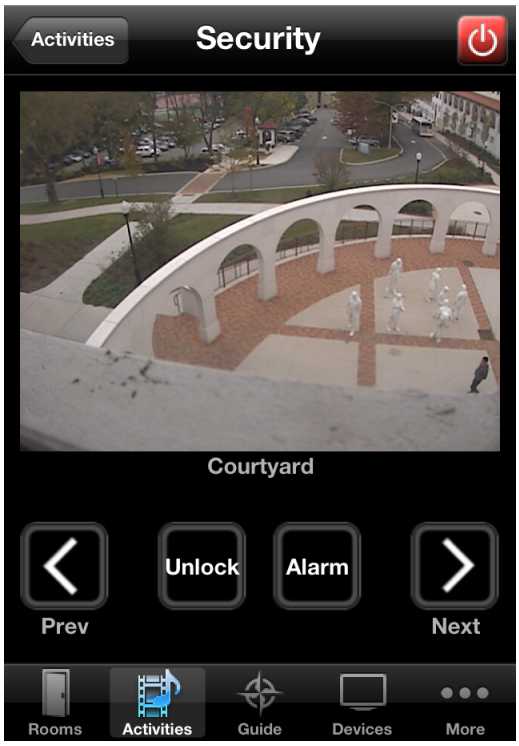
For example, if you are pulling updated track information from a media player, you can store the new updates in state variables and know that the new information is available to all client controllers. By tying these state variables to label controls, we can dynamically update the display text as tracks change. When combined with other dynamically updating content – such as image controls – you can create activity layouts that are both useful and visually striking.

Labels as Buttons

Beyond their underlying state variables, you can set various text properties, such as the font size and text color. One of the more interesting options is the ability to assign a command or script action to the label, turning it into a clickable button. In this way labels are similar to image controls – they can be for display only, or you

can assign an action to make them function as custom buttons. In addition, you can mark them as repeating buttons, and you can also choose their highlight color when tapped.

Camera Controls



RedEye's camera control is unusual among the control types in that it does not actually "control" anything — that is, camera controls are for display only. Even so, it can be useful to have visibility to Internet Protocol (IP) camera feeds when you are using other controls, particularly when it comes to security systems and other monitoring applications.

For the moment RedEye supports only Motion-JPEG video feeds, but as these are common among IP camera devices it is not a major limitation. Motion-JPEG is a constantly updating sequence of JPEG images. The use of JPEG allows for in-frame compression, but each frame is transmitted independently, so there is no opportunity for the higher compression ratios achieved through inter-frame compression as used by newer protocols such as H.264, and therefore Motion-JPEG can be something of a bandwidth hog. On the other hand, Motion-JPEG is also an adaptive frame-rate protocol (i.e., when the system is under load it can simply send frames less frequently), so if you can tolerate slower frame rates, you can include more simultaneous streams. We have tested RedEye activities with up to 8 simultaneous streams running at 320x240 resolution with a 30% compression ratio and found performance to be adequate (single digit frames per second). Real-world performance depends on a number of factors, including the processing capabilities of the IP cameras, client hardware, the network, and the resolution and compression ratio of the source feeds.

Camera Feed URL

We pull the camera feed URL from — you guessed it — a state variable. The URL itself usually ends with .cgi, .mjpg, or .html, and you can stream from both HTTP and HTTPS sources (more about security in a minute). As with other controls, the benefit of using a state variable is that you can update the URL at runtime.

Why update the camera URL at runtime? If you have several cameras that you want to monitor you could drop several camera controls onto your activity screen and have them all loading simultaneously, but doing so chews up network bandwidth and can make your controller a bit sluggish. By using a state variable, you can limit yourself to one or two camera controls, and the create buttons that update the state variable, thereby switching camera views within the application.

Updating the camera URL with a button is simple enough — add a button to your layout, give it a script action, and inside the script call the Scripting.SetVariable() method to change the value. You can have a button for each available camera, or you can create a pair of next camera/previous camera buttons. The following script demonstrates how a “next camera” button can work.

```
require "systemScript"

local cameraId = tonumber(Scripting.GetVariable("cameraId"))
cameraId = cameraId + 1
if cameraId > 5 then
    cameraId = 1
end
Scripting.SetVariable("cameraId", cameraId)

if cameraId == 1 then
    Scripting.SetVariable("camera",
                          "http://192.168.1.180/mjpg/video.mjpg")
elseif cameraId == 2 then
    Scripting.SetVariable("camera",
                          "http://192.168.1.181/mjpg/video.mjpg")
elseif cameraId == 3 then
    Scripting.SetVariable("camera",
                          "http://192.168.1.182/mjpg/video.mjpg")
elseif cameraId == 4 then
    Scripting.SetVariable("camera",
                          "http://192.168.1.183/mjpg/video.mjpg")
elseif cameraId == 5 then
    Scripting.SetVariable("camera",
                          "http://192.168.1.184/mjpg/video.mjpg")
end
```

(Please make sure to create a “cameraId” variable and give it a numeric value before running the above script.)

To create a similar script for the “previous camera” button, you only need to change lines 4-7 to the following:

```
cameraId = cameraId - 1
if cameraId < 1 then
    cameraId = 5
end
```

Network Security

If your IP cameras are available outside of a secure local-area network, you may want to protect their video feeds using a password, encryption, or both. RedEye can connect to these secured feeds, but you have a couple of options regarding how to do so.

Encrypted Feeds

In practice connecting to an encrypted feed is as easy as changing the URL from HTTP to HTTPS (provided that your camera is configured for HTTPS access). In order to serve up an HTTPS stream, your camera must have a security certificate installed. In most commercial settings these security certificates are signed using the "public key infrastructure" — a system by which well-known entities (such as GeoTrust, Thawte, or VeriSign) vouch for the certificates of other companies. There is a cost to this verification, and for purposes of validating a security camera it may not be worth the trouble.

The alternative to such a publicly-signed certificate is a "self-signed" certificate — usually created by the camera itself. RedEye can connect to cameras using self-signed certificates, but in order to do so you need to select the "Allow Self-Signed" option in the camera control settings.

Password-Protected Feeds

Encrypting your video feeds prevents someone else from snooping on your connection, but it does not prevent them from making their own connection to the camera. If you want to prevent unauthorized access, you will need to password protect your camera. RedEye can access camera feeds that use basic HTTP authentication, and you have a couple of options for doing so.

Storing credentials in your RedEye configuration

The first option is the least cumbersome, but also the least secure: you can enter your username and password directly into the camera settings when you create your layout. RedEye will store this information in your configuration, and provide it to the camera automatically each time you connect. The downside is that because your information is stored in the configuration, it is available to technically savvy people who have access to your network. Even so, if your

network is secure and you trust everyone around you, then this may be a viable option.

Entering credentials at runtime

The second option is to leave blank the username and password fields in your camera control settings. If your camera requires login credentials, the RedEye app will prompt you for the username and password at runtime when you attempt to connect to the camera. You can save yourself some typing by opting for a hybrid approach — if you enter your username in the camera control settings but leave the password blank, you need only enter the password at runtime.

Sending credentials over unencrypted connections

While the decision to use authentication or not is independent of whether you encrypt your feed using HTTPS, if you do choose to password-protect an unencrypted feed you should be aware that your username and password will be transmitted over the network in clear text every time your connect to your camera. If it is your intention to do so, please ensure that you select the “Allow Clear Text” option in camera control setup — otherwise RedEye will not attempt to send the password over an unencrypted (HTTP) connection.

HTML Controls

HTML controls provide the ultimate flexibility, enabling the display of styled text, graphics, tables, and even external web content in your layouts. This flexibility comes with a price – configuring HTML controls is more complex and rendering across all client control platforms can be trickier – so if another control can deliver what you need (e.g., a custom image, a dynamic text label, etc), it may be a better choice.

Such is not to say that you should shy away from using HTML controls. They are quite powerful, and doing basic things like displaying columnar data (imagine a playlist that shows track number, title, and duration) or styled, multi-line text is not difficult at all. As with all tools, the trick is knowing when and how to use them appropriately.

What They Are, and What They Are Not

Generally speaking, there are few limitations as to what you can do with HTML controls in RedEye. You can use whatever HTML tags you please, and conform to just about any version of the HTML standard that you choose. That said, your success in creating a consistent presentation across Android, iOS, and PC platforms running various web browsers is largely dependent on your implementation choices. Others have written countless volumes on cross-platform browser compatibility, so this is not a subject that we will dwell on here. Suffice it to say that adhering to widely accepted standards and testing are the two best ways to ensure that you get the intended results.

It is also up to you to determine the particular requirements of your system. If you never use RedEye outside of the iOS platform, then you need not concern yourself with Android or PC compatibility; if you do not use Internet Explorer, then there is no need to test using that particular browser. When you do anticipate running on different platforms, we strongly encourage you to test on all of those platforms, because strange quirks in HTML rendering are a fact of life.

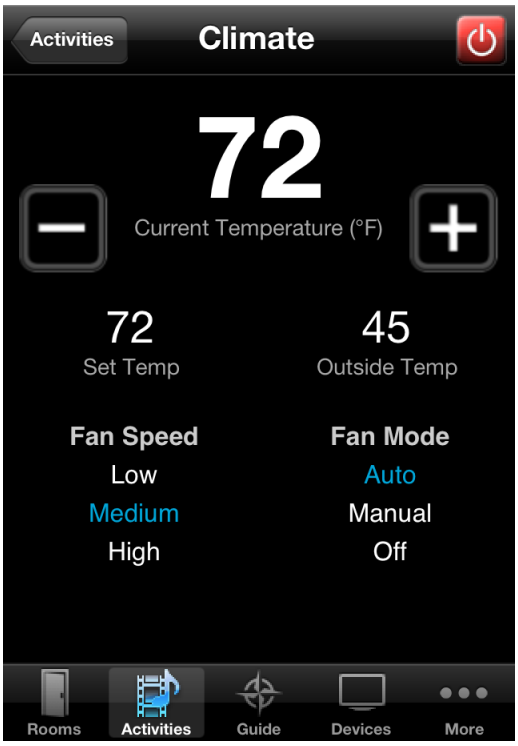
RedEye's HTML controls will render any valid HTML fragments you send them. In other words, you should *not* attempt to use full HTML documents (with <html>, <head>, <meta>, <link>, or <body> tags) — RedEye cannot render these because the browser application itself is already an HTML document.

Under the covers, RedEye takes your HTML fragment and embeds it inside a tag with styling applied so that your HTML content blends well with overall RedEye app styling.

Applying Custom Styles to HTML Content

Because RedEye cannot present full HTML documents, you cannot use stylesheet links in your HTML content. However, you are free to use in-line style tags. Style tags open up a range of possibilities for displaying custom fonts and sizes, as well as positioning elements within the control. On the other hand, the greatest variability among different browsers exists in how they apply styling to HTML, so you should be careful to test thoroughly whenever you employ in-line style tags. For example, Android uses a different set of fonts than do PCs and iOS devices, so you may find that there are differences in font spacing and positioning when using Android.

Let's say that you want to create an HTML control for adjusting the fan settings of a climate control system. There are two settings groups: fan speed, and fan mode. Within each group, there are a handful of settings. You would like to display all of the available settings, but highlight the current settings so that they are readily visible. The following HTML table is not particularly fancy, but it works just fine:



```
<table>
    <tr>
        <th style="border-style: none;">Fan Speed</th>
        <th style="border-style: none;">Fan Mode</th>
    </tr>
    <tr>
        <td style="border-style: none; color: #ffffff;">Low</td>
        <td style="border-style: none; color: #00a6d9;">Auto</td>
    </tr>
    <tr>
        <td style="border-style: none;
                   color: #00a6d9;">Medium</td>
        <td style="border-style: none;
                   color: #ffffff;">Manual</td>
    </tr>
    <tr>
        <td style="border-style: none; color: #ffffff;">High</td>
        <td style="border-style: none; color: #ffffff;">Off</td>
    </tr>
</table>
```

In the above HTML, we have done a couple of things. First, we are highlighting the Medium fan speed and Auto fan mode settings using a cyan color. Second, we are overriding a couple of RedEye's default table styles by setting the header and detail borders to none, and the detail text color to 100% white.

Adding Functionality to Hyperlinks

It is fine to display information in an HTML control, but more useful to be able to actually click on hyperlinks and have some response in your system. Anchor tags ("<a>", also known as hyperlinks) are the means for making things happen on web pages, and we have retained this capability within RedEye HTML controls.

When you populate your controls with HTML content, RedEye scans the anchor tags for links that begin with the string "redeye#". When you click on one of these hyperlinks, it takes the text following the "redeye#" prefix and stores that value in a state variable of your choosing. Then it executes a command or script that you specify. Within your command or script, you can inspect the value of the state variable and determine how to proceed.

Building on the previous example, we can add hyperlinks to the table rows. Note how we avoid putting hyperlinks on the currently selected values (they are already active, so there is no need to make them clickable) and how the color styling moves from the table detail to the anchor.

```
<table>
    <tr>
        <th style="border-style: none;">Fan Speed</th>
        <th style="border-style: none;">Fan Mode</th>
    </tr>
    <tr>
        <td style="border-style: none;">
            <a href="redeye#fanspeed-low"
               style="color: #ffffff;">Low</a>
        </td>
        <td style="border-style: none; color: #00a6d9;">Auto</td>
    </tr>
    <tr>
        <td style="border-style: none;
                   color: #00a6d9;">Medium</td>
        <td style="border-style: none;">
            <a href="redeye#fanmode-manual"
               style="color: #ffffff;">Manual</a>
        </td>
    </tr>
    <tr>
        <td style="border-style: none;">
            <a href="redeye#fanspeed-high"
               style="color: #ffffff;">High</a>
        </td>
        <td style="border-style: none;">
            <a href="redeye#fanmode-off"
               style="color: #ffffff;">Off</a>
        </td>
    </tr>
</table>
```

In this example you would want to change the HTML content whenever a link is tapped. Ideally, the climate system provides some feedback so that when the fan mode or speed changes, you can modify the HTML content variable accordingly. If cannot get any input data from the fan system, you can still update the HTML content variable within the same command or script that process the hyperlink invocation; the only downside here is that if someone changes a setting outside the RedEye system your HTML display may become out of synch with the climate control system.

Using Hyperlinks to Launch Other Apps

Although as a universal remote control RedEye provides a convenient place to control all of your equipment, sometimes you may need to dive deeper into another system. For example, perhaps you want to edit the playlist in a media streaming system that you are controlling via RedEye. Although it may be technically possible to create an interface for editing the playlist within the

RedEye app, the media streaming system's dedicated app is likely to do this more efficiently (and certainly with less configuration work on your part). In such cases it makes sense to "integrate" with other systems by launching their applications.

To this end, when the "href" element in your anchor tag begins with the prefix "launch#", the RedEye app will attempt to launch the app you specify in the URL immediately following. For example, on iOS Apple's Remote app for controlling AppleTV and iTunes is registered to launch when you enter the URL "remote://". In your HTML control, the following link will launch the app:

```
<a href="launch#remote://">Remote app</a>
```

Discovering App URL Schemes

These special app URLs only work if the target app has defined a URL scheme.[17] There is no easy way to know whether an app has defined a URL scheme or what it might be – publishing this information is up to the app developer. You may be able to find this information online. The Dutch company Magnatron hosts a self-registration repository for app URL schemes called "handleOpenURL":

http://handleopenurl.com/

Even so, you may not find a listing for an app that does have a URL scheme. Apple's Remote app is one such example which is defined but not listed. Sometimes the easiest way to know whether you can launch an app by URL is to guess at a few likely URL candidates. In iOS, you can enter your URLs in the Safari browser's address field, and if the scheme is valid, it will launch the app when you hit the page load button. This approach does not work in Android, however, so you will need to populate your HTML control and try that way. We have had reasonable success in discovering URL schemes (including the Remote app) simply by entering the app name (no spaces) followed by colon-slash-slash (://).

Launching Apps without a Scheme (Android only)

Although iOS limits launching to those applications that have explicitly defined a URL scheme, on Android you can launch any application, provided you know the application's unique identifier. App identifiers come in the "reverse domain-name"

[17] We have defined a URL scheme for the RedEye app. If you want to launch RedEye from another application, please use "redeye://" (without the quotation marks).

format, e.g., com.thinkflood.redeye.[18] Once again, however, it is not readily apparent what the reverse domain-name is for a given application. Thankfully, if the app in question is available through Android Market, you can find the identifier easily enough.

For example, the controller app for Sonos multi-room music systems does not define a URL scheme. On iOS, that means RedEye cannot launch the app. However, the app is available through Android Market. A quick search reveals the following URL:

https://market.android.com/details?id=com.sonos.acr

Looking at the tail of the URL, we see a query string with the parameter "id=com.sonos.acr" — that is the app identifier we need.[19]

Now we have the application identifier, but in order to launch the app we need to format our URL a little differently. We still include the "launch#" value in our "href" tag, but in this case we add a new attribute to the link. The "package-name" element identifies that we are launching an Android application using its package name (application identifier). In this example,

```
<a href="launch#" package-name="com.sonos.acr">Open Sonos app</a>
```

Launching Apps to a Specific Page

Sometimes when you launch an app you want to go directly to a particular page so that you can perform a particular operation. Your ability to do this is largely dependent on the app developer's use of the URL scheme. For example, an

[18] Reverse domain names are a convenient way to ensure that application names are globally unique without having to set up a special system to manage the uniqueness. Basically, we end up relying upon the domain naming system to ensure uniqueness (e.g., ThinkFlood has registered for thinkflood.com), and then the individual company to manage uniqueness across all of the apps they develop.

[19] If you actually perform a search on market.android.com for the Sonos app, you may arrive at a page with a URL that looks something more like this:

https://market.android.com/details?id=com.sonos.acr&feature=search_result -?t=W251bGwsMSwyLDEsImNvbS5zb25vcy5hY3IiXQ..

In this case, you want only the text between "id=" and the next ampersand (&) character. The remainder of the URL is specific to your search and is not part of the application identifier.

early version of Kaleidescape's iPad application defined a URL scheme that would you to launch the app to any of the following pages: System, Movies, Music, Remote. Thus, if you are in a "Watch Movie" activity on your RedEye system you might include a link to the Kaleidescape app's Movies page, whereas if you are in a "Listen to Music" activity, a link to the Music page would be more appropriate. For example, a link to the Movies page would look like this:

```
<a href="launch#kscape://?view=Movies">Go to Kaleidescape app</a>
```

Full documentation for the Kaleidescape iPad app's URL scheme is available online at:

http://www.kaleidescape.com/files/documentation/Integrating-with-the-Kaleidescape-App-for-iPad.pdf[20]

If you are using Android, the target application does not need to define a special URL scheme to open a particular page, though you will need to know the internal name of the page (called a "class") in the app. In this case you will need to include both the "package-name" attribute and a new "class-name" attribute.[21] Again, this requires some information from the app developer which may not be readily available. As an example, the following URL opens the home screen within the RedEye app:

```
<a href="launch#"
   package-name="com.thinkflood.redeye"
   class-name="com.thinkflood.redeye.activity_classes.DashboardAc
tivity">Launch RedEye</a>
```

Launching System Applications

In addition to launching apps from third-party developers, there are a handful of system applications that you can launch on both iOS and Android. For example, you can create HTML links that place phone calls, format email messages, send SMS text messages, open Google Maps, or play YouTube videos. The URL

[20] If you plan on integrating with the Kaleidescape iPad app, this document is definitely worth reviewing. For example, Kaleidescape provides access to an app icon that you can include in your HTML control. You can even have their app display a button that will return you to the RedEye app (using our redeye:// URL scheme).

[21] Whenever you include the "package-name" attribute, the RedEye Android app will ignore whatever follows the "launch#" value in the "href" attribute. Thus, you can use a URL scheme for iOS in the "href" attribute and something completely different on Android using the "package-name" attribute. This can be useful if you want to launch a different app on Android than on iOS.

format differs somewhat for each of these options, but they all start with the "launch#" prefix in your anchor tag's "href" element. You can read on for more details, or refer to Apple's online documentation here:

http://developer.apple.com/library/ios/#featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html#//apple_ref/doc/uid/TP40007899

Placing a Phone Call

For example, if you are installing a RedEye system for a client and want the client to be able to call you easily if they require technical support, you can include a phone link within your HTML control. The following link will call ThinkFlood's tech support line:

```
<a href="launch#tel:1-617-299-2000">Call Technical Support</a>
```

Formatting an Email Message

Similarly, you could format a message to send with a technical support question:

Email Technical Support

If you like, you can even include other portions of the email using standard mailto link formatting rules:[22]

```
<a href="launch#mailto:support@thinkflood.com&subject=Help%20with
%20RedEye">Email Technical Support</a>
```

Sending an SMS Text Message

Continuing in this vein, you could also send a text message:

```
<a href="launch#sms:1-617-299-2000">Text Support</a>
```

Opening Google Maps

As long as you prefix the URL with "launch#", most Google Maps URLs will open the Google Maps app on your phone (this has the additional benefit of working on the browser app[23]). For example:

---

[22] See http://www.ietf.org/rfc/rfc2368.txt

[23] Some of the other link formats, such as "sms:" and "tel:" will not work, although depending on your web browser, you may be able to configure them to launch a softphone or other application on your PC. At the very least they will not cause

```
<a href="launch#http://maps.google.com/maps?q=255+bear+hill+road+
waltham+ma">Map of ThinkFlood Office</a>
```

There are a number of different options using Google Maps URLs (including driving directions), most of which are documented here:

http://mapki.com/wiki/Google_Map_Parameters

Playing YouTube Videos

There may be situation in which you want to play a video from YouTube. As with Google Maps, all you need to do is include the YouTube URL following the "launch#" prefix:

```
<a href="launch#http://www.youtube.com/watch?v=qEF1hkGAMM4">Watch
&quot;Getting Started with RedEye&quot; video</a>
```

Widgets: Embedding External Content

RedEye's HTML controls make it possible to embed external web content into your activity layouts, in essence creating web widgets that can feed dynamic content into your activity layouts. For example, you can embed weather forecasts or sports scores.

The mechanism for bringing in outside content is the <iframe> HTML tag. Just like n image tag, iframes use the src element to indicate the source URL for their content. Usage is quite simple:

```
<iframe src="http://thinkflood.com/"
        width="100%"
        height="100%" />
```

While using an iframe is easy, making the content fit on a mobile screen is more challenging. You may find that using the mobile version of a website works better (many are formatted to fit RedEye's default 320 pixel wide layout), but sometimes getting everything exactly right requires putting up your own website specially formatted for RedEye layouts and pulling information from there. Website production and hosting are outside the scope of this manual.

Another consideration in terms of making external content work onscreen is that RedEye's smartphone and tablet applications do not allow the web content to

---

problems in the browser app – they will simply have no effect when you click them.

scroll. The reason is that the activity layout itself scrolls, and it is difficult using a finger on a small mobile device screen to scroll certain parts of a page but not others.

When considering the size and scrolling limitations, you should consider navigation within the iframe. As external content will often contain hyperlinks, it is possible to navigate to a different page within the HTML control. This may be what you want to do, but you should be aware of the implications and make sure to test what happens when you go from page to page.

Sizing Everything to Fit

One of RedEye's most important features is its support for a wide array of client controllers and operating systems, and a critical part of this support is being able to handle different screen sizes and resolutions. For most control types, we can simply scale the control based on the available display resolution, but HTML controls have many more moving parts and do not always scale as nicely. The following practices will help ensure that your HTML content scales as well as possible:

1. **Use relative sizes wherever possible.** Rather than specifying absolute pixel dimensions or font sizes, try to use percentages. The enclosing tag into which RedEye places your content is sized using the absolute pixel resolution of the current display, so width and height percentages work fine for images as well as tables and other controls.
2. **Provide source images that have at least 2X the resolution required for a display 320px wide.** This is in keeping with our recommendations for standalone image controls.
3. **Test various layout sizes to see how things scale.** The browser application is a particular convenient vehicle, as you can resize the layout on the fly by dragging the boundary between the navigation pane and the layout panel. When testing different sizes, look for content clipping – you may need to adjust the HTML control container size slightly in some cases.

Creating HTML Mockups on a PC

When it comes to creating and testing your HTML content, you may find it easier to start by making an HTML document on a PC, tweaking until you get the look you want, and then moving your tested HTML over to a RedEye state variable later. To that end, we have created an HTML document that includes RedEye's basic control styling and dimensions. You can download the test document from our support website:

http://thinkflood.com/products/redeye/html-control-test-page.zip

Text Field Controls

As the electronics in our homes become more sophisticated, advanced functionality such as the ability to search for specific media becomes more commonplace. It is with these more advanced devices that the traditional infrared control approach (button pushing) starts to feel strained and inconvenient. After all, onscreen keyboards are painful to use, and remotes that integrate full-blown keyboards seem clunky. At the same time, we have become accustomed to using the virtual or miniature keyboards on our mobile devices for tapping out short messages, and so it seems natural to turn there for text entry.

RedEye's new text field control makes this possible, at least as far as device and activity layout design are concerned. Whether remote text entry is actually an option depends on the control interface of your equipment. Even basic infrared could allow for keyboard entry if the device defines a separate command signal for each infrared character. Normally, however, the ability to send a chunk of text down to your device requires a more robust interface such as RS-232 or IP.

Text Variable

On a conceptual level, RedEye text field controls are fairly basic. You enter some text onscreen. RedEye stores the text you enter in a state variable, and then it executes a command or script to process the contents of the state variable.

This flow makes a tradeoff in order to keep things simple and maintain a high level of reliability and performance. For example, the RedEye application does not send down each character as you type, because doing so would require an open and active connection to the device you are controlling and therefore become a bottleneck. Instead, once you finish typing in a text field (either by hitting the Enter/Done key on your keyboard or by tapping on another control in the layout), the RedEye client sends down the entire block of text that you have entered for processing. In this sense text fields function in a similar way to slider controls – they do not update dynamically, but instead wait for you to finish

making changes on the client before processing the end result on the RedEye server.

Text Field Action

The most interesting part of the text field is the action that processes the Text Value variable. This action is usually a script which grabs the state variable value and then sends that text over to the controlled device. Whether you send the text one character at a time or in a chunk depends on how the controlled device receives textual data. The following example is a command script that demonstrates how to send text character by character:

```
require "systemScript"

local inputText = Scripting.GetVariable("roku-textInput")

-- Unfortunately, there is no "clear" command in the Roku API;
-- otherwise, we would use it here
if #inputText > 0 then
    for position = 1, #inputText do
        local character = string.sub(inputText,
                                     position,
                                     position)
        local escapedCharacter =
            Scripting.EncodeForUrl(character)
        Scripting.SendHttpMessage("keypress/Lit_" ..
                                  escapedCharacter,
                                  "POST")
        -- Give Roku time to catch up
        Scripting.Wait(500)
    end
end

Scripting.SetVariable("roku-textInput", "")
```

The first step in this process is to grab the value of the state variable we are using. Then we check to make sure it has a length greater than zero before processing it character by character starting from the first to the last. We URL-encode each character (a requirement since the characters will be part of a URL path), and then we send an HTTP request message containing the URL-encoded character. Some testing here reveals that we can send characters too fast for the receiving device to process, so we add a bit of delay before sending the next one. When we are finished, we wipe out the value of the text variable to make sure we do not accidentally send it twice somehow.

Editing Layout XML

In version 2.13.0, we introduced the ability to export the layout for any device or activity as an XML document. This XML can be re-imported onto a different device or activity, effectively copying the layout of the original.

We chose to use XML as the intermediate format for exporting layouts so that you can customize them before importing.[24] In fact, you do not even have to start with an existing layout – you could make your own XML from scratch. Also, when it comes to importing layouts, there is no difference between a device layout and an activity layout – you can import a device layout onto an activity and vice versa.

Please note that everything in the layout XML is case sensitive – if you change "id" to "ID", the XML will no longer be valid.

Changes to Internal Layout XML

Internally, we use XML documents to move information between client controllers and the RedEye server. Exported layout XML is similar to the layout XML we use internally, but there are a few key differences that we have introduced in order to make layouts portable across different RedEye configurations.

Types Rather Than IDs

In an internal layout XML document, we reference devices and commands by their IDs, because IDs are fast and unambiguous. IDs are also specific to a particular RedEye configuration. Device ID 10 on one RedEye might be a Sony TV, while on another RedEye it might be a Yamaha AVR. In order to make the layout portable, we have to resort to device types and command types.

When you import a layout into another configuration, RedEye does its best to match the device and command types to specific device and commands IDs within the target configuration. There are some limitations:

1. **"Other" types.** When a device or command has a type of "other", we do not have enough knowledge about what the original function was, and

[24] XML is a "human-readable" format, which means you can open it with any text editor, such as Notepad (Windows) or TextEdit (Mac). You do not need to purchase any special software, although if you are doing a lot of XML editing, sometimes it is nice to use a special purpose XML editing program to ensure that everything is formatted properly. If you are new to XML, the W3Schools website has a good tutorial: http://www.w3schools.com/xml/default.asp.

therefore we cannot map these effectively. As a result, any control action which references an “other” device or “other” command will not be transferred as part of the import process. We do import the control itself, so you do not lose placement or appearance characteristics of the layout – in effect, all you need to do is select the control and reassign the action.

2. **Multiple devices or commands of the same type.** If your target activity contains two television devices, RedEye will not be able to distinguish between them when importing and will always pick the first one that it finds. Likewise, if you have multiple commands of the same type on a device, RedEye will always pick the first command of that type when importing. As with “other” types, you may need to edit some of the control actions after importing.
3. **We do not modify scripts.** The import/export process does not dive into the text of any embedded scripts in your layouts, so these will continue to reference IDs from the original configuration. You can modify the scripts manually within the XML prior to importing, or you can import the layout and then edit the scripts using RedEye’s script editing window – either approach works fine. This is yet another reason why it is better to put scripts into commands rather than layouts when possible.

Custom Variables

The use of “state” or “custom” variables is an important part of layout customization, and we have tried to make sure they are easy to manage through the export/import process.

When you export a layout, the XML contains both the variable name and its value. When you import it, RedEye will inspect the target configuration to see if a variable of the same name (case-sensitive) is already present. If that variable exists, then RedEye will use the existing variable. If the variable does not exist, RedEye will create it and assign it the value contained in the exported XML. When using an existing variable, RedEye does not update or overwrite the current value.

System Data Reference Page

When it comes to modifying layout XML, it is helpful to have a reference document that lists RedEye constant values such as device types, command types, icon types, and so forth. All of these are available from the **System Data** page on your RedEye unit. To access the System Data page, open the browser application to the Setup pages, click on the Maintenance tab, and the click on the System Data link. Alternatively, you can use the following URL:

http://[RedEye IP Address]:82/setup/data.html

Layout XML Format

The following XML document contains the basic structure used in each layout XML. A full layout would have too much detail to show here, so some sections are commented out to show the basic structure. Subsequent sections of this document will provide the finer-grained detail.

The XML is structured as a dictionary of key-value pairs, within which many of the values are themselves other dictionaries of key-value pairs. At the root level are the global layout properties. Among these are dictionaries containing buttons, images, sliders, labels, web controls, cameras, and text fields. There are also dictionaries for shortcut gestures and shortcut keys.

```
<dict>
    <key>showPowerButton</key>
    <string>yes</string>
    <key>buttons</key>
        <!-- XML for button controls -->
    <key>images</key>
        <!-- XML for image controls -->
    <key>labels</key>
        <!-- XML for label controls -->
    <key>sliders</key>
        <!-- XML for slider controls -->
    <key>webs</key>
        <!-- XML for web (HTML) controls -->
    <key>cameras</key>
        <!-- XML for camera controls -->
    <key>texts</key>
        <!-- XML for text field controls -->
    <key>shortcuts</key>
        <!-- XML pairing shortcut gestures with controls -->
    <key>keyboardShortcuts</key>
        <!-- XML pairing keyboard shortcuts with controls -->
</dict>
```

A full layout XML document is available online here:

http://thinkflood.com/products/redeye/sample-layout.xml

Show Power Button

One of the most basic layout properties is the ability to show or hide the power button. When this property is set to “yes”, the activity will display a default power button in the upper right corner of the screen next to the activity name. When this

property is set to “no”, the power button is not displayed (for example, when the activity provides a custom power button, as when using a launchpad activity).

This property applies only to activity layouts. For device layouts, we display a power toggle button only if the device contains a power toggle command.

```
<key>showPowerButton</key>
<string>yes</string>
```

Button Controls

All button controls in a layout appear within the “buttons” dictionary. For historical reasons, this dictionary has an extra layer of depth when compared with other control dictionaries – within the “buttons” dictionary there is another “buttons” dictionary.

```
<key>buttons</key>
<dict>
    <key>buttons</key>
    <dict>
        <key>0</key>
            <!-- XML for button control with ID = 0 -->
        <key>6</key>
            <!-- XML for button control with ID = 6 -->
    </dict>
    <key>nextId</key>
    <integer>37</integer>
</dict>
```

In the early days of RedEye, layouts had only button controls, and each button was a property of the layout itself. Then we added a property called **nextId** which specified the ID of the next button to be created. Keeping **nextId** separate from the buttons required pushing the **buttons** dictionary one level down, off the main layout.

Later, when we made the switch to having other controls, we wanted to keep each control type separate. We also wanted to leave the existing **buttons** dictionary alone for backwards compatibility, so we wrapped the whole thing in another dictionary layer.

While this is admittedly confusing, there are only two things that you need to do:

1. Follow the above structure.
2. Make sure that the **nextId** key has a value that is 1 greater than the highest ID of any control within the layout.

The second point about the **nextId** property is particularly important and may be non-obvious. RedEye uses this **nextId** property to assign IDs to new controls (of any type – even if you don't have buttons on the layout), and those IDs need to be unique within the layout. For example, if you export a layout with controls 0 through 9, then **nextId** will be 10. If you then add a control to the XML but forget to increment **nextId** to 11, the layout will import properly. However, when you next go to add a control, you will have trouble because the layout will then have two controls with ID 11 (exactly what happens – whether one control is overwritten with another or whether the entire layout becomes invalid – can vary depending on which controls you add, but in any case, the result is not good).

Each individual button has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>buttons</key>
<dict>
    <key>buttons</key>
    <dict>
        <key>6</key> <!-- this must match the button ID -->
        <dict>
            <key>id</key>
            <integer>6</integer> <!-- this must match the key -->
            <!-- more button properties -->
        </dict>
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name. If a "text only" icon (icon type 99999) is assigned to the button, then the name is displayed inside the button. When using any other icon, you can choose to display the name just below the button (see **showLabel**).
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **icon.** The icon which the button will display. Icon type 99999 is a "text only" icon which displays the name of the button. For a current list of icon types, please reference the System Data page in the browser application.

7. **showLabel.** Indicates whether the name of the control will be displayed in a text label just underneath the control. Valid values are “yes” and “no”. If the **icon** property is set to “text only” (icon type 99999), then this setting has no effect.
8. **buttonSize.** The size of the button. The three valid values are 1 (small), 2 (large), and 3 (jumbo).
9. **repeatInterval.** Indicates the frequency with which a control’s action is repeated, in milliseconds. For a repeating control, this must be something greater than 0 – usually 50 milliseconds – and there should be only one action in the control’s action list. If the button is of any other type (normal, toggle, macro), this value must be 0.
10. **timeoutInterval.** Indicates how long a control’s action will be repeated with no communication from a client controller before the repeat will be cancelled (to prevent uncontrolled repetitions in the case of a network failure or other exception). For a repeating control, this must be something greater than 0 – usually 2500 milliseconds. For a button of any other type (normal, toggle, macro) this value must be 0.
11. **isToggle.** Indicates whether the button toggles through a defined list of actions when pressed. Valid values are “yes” and “no”. If “yes”, then the **toggleValues** node must be populated.
12. **nextAction.** For buttons where **isToggle** is set to “yes”. This is referenced by RedEye at runtime, so it should be set to 0 when importing.

```
<dict>
    <key>id</key>
    <integer>6</integer>
    <key>xPosition</key>
    <real>63</real>
    <key>yPosition</key>
    <real>26</real>
    <key>name</key>
    <string>Normal</string>
    <key>description</key>
    <string>This is a sample button control</string>
    <key>icon</key>
    <integer>99999</integer>
    <key>showLabel</key>
    <string>no</string>
    <key>buttonSize</key>
    <integer>2</integer>
    <key>repeatInterval</key>
    <integer>0</integer>
    <key>timeoutInterval</key>
    <integer>0</integer>
    <key>isToggle</key>
    <string>no</string>
    <key>nextAction</key>
    <integer>0</integer>
    <key>toggleValues</key>
        <!-- XML for button toggle values -->
    <key>actions</key>
        <!-- XML for button actions -->
</dict>
```

Toggle Values

Buttons are unique among controls in that they can toggle between different states with each subsequent tap[25]. In the **toggleValues** property of the button, you specify how the button should appear when it is in each state.

For example, consider a power on/off toggle button. When powering off the device, the button should display a “power off” icon; when powering on the device, the button should display a “power on” icon. The **toggleValues** property contains an array of button states that provide the name, description, and icon which the button will use as it cycles through. The order of this array must match the order of the actions on the button – i.e., the first toggle value maps to the first action, the second toggle value to the second action, and so forth. It follows that

[25] You can achieve something similar with image and HTML controls using scripting, but buttons provide this capability natively.

the number of toggle values and the number of actions must match exactly, or you may have runtime errors when you use the button.

The following is a **toggleValues** array for the power toggle button as described above.

```
<array>
    <dict> <!-- first toggle value -->
        <key>name</key>
        <string>Power On</string>
        <key>description</key>
        <string></string>
        <key>icon</key>
        <integer>1</integer>
    </dict>
    <dict> <!-- second toggle value -->
        <key>name</key>
        <string>Power Off</string>
        <key>description</key>
        <string></string>
        <key>icon</key>
        <integer>2</integer>
    </dict>
</array>
```

Button Actions

Actions are what make the button actually do something. All controls (with the exception of cameras) have an **actions** property, but buttons have more options than most.

No actions. Actions always appear inside an array, even if there are fewer than 2 actions. If the button has no actions, then the actions array is empty. You will see this if the button originally had no action, or the actions used device or command type “other” in which case RedEye will not be able to map the action to an appropriate device or command in the target system.

```
<key>actions</key>
<array />
```

Command actions. Within the **actions** array lie individual actions. Each action is a dictionary, which has three properties:

```
<dict>
    <key>deviceType</key>
    <integer>0</integer>
    <key>commandType</key>
    <integer>29</integer>
    <key>toggleValue</key>
    <string></string>
</dict>
```

1. **deviceType.** This property indicates the type of the device on which to perform the action. Please reference the System Data page in the browser application for a full list of available device types.
2. **commandType.** This property indicates the type of command to use. Please reference the System Data page in the browser application for a full list of available command types.
3. **toggleValue.** This property applies only when the command referenced toggles. For example, command type 0 is Power On/Off (toggle). The two toggle values for this command are usually “On” and “Off”. Note that toggle values are identified by their name, not by an ID. Toggle value names are case sensitive, and may vary from configuration to configuration, so you may need to edit these before importing. Remember also that RedEye “intelligently” handles toggle values – if you specify that a button should execute a power toggle command to the “On” value, RedEye will not send the command if it believes the device is already in the “On” state.

When an action has both a **deviceType** and a **commandType**, we call it a “command action” – that is, it is an action which executes a predefined command on a specific device.

Delay actions. Not all actions invoke commands. Particularly when it comes to transmitting infrared command signals, it is important to leave a bit of space between each command so that the receiving equipment can distinguish one action from another. In addition, sometimes devices require time to respond to one command before processing the next. The classic example is powering a device on – many device require a warm-up period of several seconds before they will accept other commands. RedEye’s delay actions provide a means to handle both situations.

A delay action has no device – the RedEye is simply idling for a period of time. As a result, delay actions have a **deviceId** property which is always -1.

Delays are defined by a length of time. RedEye delays are expressed in milliseconds – thousandths of a second - and can be up to several seconds in

length (in the RedEye client applications, we limit them to 20 seconds per delay). Within a delay action, the **commandId** property specifies the length of the delay (in milliseconds).

We do not use the **toggleValue** property for delay actions.

The following is an example of a 5 second delay action:

```
<dict>
    <key>deviceId</key>
    <integer>-1</integer> <!-- indicates this is a delay action -
->
    <key>commandId</key>
    <integer>5000</integer> <!-- delay length: 500 ms = 5 s -->
    <key>toggleValue</key>
    <string></string>
</dict>
```

Script actions. Actions can also contain Lua scripts. Scripts allow for much more sophisticated behaviors to be embedded inside a control.

In the case of a script action, both the **deviceId** and **commandId** properties are set to -1. Then the script itself resides in the **toggleValue** property, as shown here:

```
<dict>
    <key>deviceId</key>
    <integer>-1</integer> <!-- could be a delay action -->
    <key>commandId</key>
    <integer>-1</integer> <!-- no, it's a script -->
    <key>toggleValue</key>
    <string>
require "systemScript"
-- This is a sample script.

Scripting.SendCommand(93, 24)
Scripting.SendToggleCommand(93, 40, "On")
    </string>
</dict>
```

There are a couple of important things when editing layouts that have script actions:

1. **Line breaks matter.** In theory, Lua does not care about line breaks or whitespace inside a script. In practice, this is almost true. The exception is when dealing with a single line comment. Consider the script from the example above:

```
require "systemScript"
-- This is a sample script.

Scripting.SendCommand(93, 24)
Scripting.SendToggleCommand(93, 40, "On")
```

The second line is a comment (beginning with two dash characters). The comment runs to the end of the line. If we were to remove the line breaks, we would have the following:

```
require "systemScript" -- This is a sample script. Scripting.Send
Command(93, 24) Scripting.SendToggleCommand(93, 40, "On")
```

The problem is that everything after those double dashes is now commented out – in this case, the result being a script that does nothing.

There are two ways around this problem. The first is to make sure your chosen XML editor preserves line breaks. Although this sounds easy, the official XML specification says that line breaks and whitespace do not matter in XML, so some editors will actually remove them. The other alternative is to not use the single line comment in your Lua scripts. Instead, you can use the block comment, which encloses your comment inside double square brackets as follows:

```
require "systemScript"
--[[ This is a sample script. ]]

Scripting.SendCommand(93, 24)
Scripting.SendToggleCommand(93, 40, "On")
```

Now if we force everything to one line, the script will still work as expected:

```
require "systemScript" --[[ This is a sample script. ]] Scripting
.SendCommand(93, 24) Scripting.SendToggleCommand(93, 40, "On")
```

2. **Scripts use IDs.** While the layout export process converts most actions to use device and command types in lieu of device and command IDs, it does not make changes to your scripts. Because IDs will vary from configuration to configuration, a script that executes a particular command on one RedEye is unlikely to execute the correct command on another RedEye. As a result, you will need to edit scripts after importing for things to work properly. This is one reason why it is better to keep as much scripting inside commands as possible (command scripts are much more portable than layout scripts).

Normal or repeating button. “Normal”-type buttons have a single action, as do repeating buttons. Repeating buttons function exactly as normal buttons when tapped and released, but when pressed and held will execute the action repeatedly until released. Whether the button repeats or not is determined by the **repeatInterval** and **timeoutInterval** properties (if these are 0, then the button does not repeat). Otherwise, they appear exactly the same as normal buttons.

Here is a complete **actions** array for a normal button:

```
<array>
    <dict>
        <key>deviceType</key>
        <integer>0</integer>
        <key>commandType</key>
        <integer>29</integer>
        <key>toggleValue</key>
        <string></string>
    </dict>
</array>
```

Toggle button. There are toggle commands, and there are toggle buttons – the two are not the same. You can create a non-toggle button that uses a toggle command in its action array (e.g., a “Power On” button that uses the power toggle command with the toggle value of “On”). You can also have a toggle button that uses only discrete (i.e., non-toggle) commands. A classic toggle button example is the Play/Pause button – tap once to start playback, tap a second time to pause. Thus, the primary purpose of using a toggle button in a layout is to save space – they allow for having more than one function on screen, but only take up a single button’s worth of real estate.

If a button’s **isToggle** property is set to “yes” then it is a toggle button and usually has two or more actions. Each time you tap the button, it will cycle through those actions in order: tap once to play, tap again to pause, tap a third time to start over with play again.

The **actions** array will have an action for each available state of the toggle button. The button must also have a corresponding number of items in the **toggleValues** array. In our earlier example for the **toggleValues** array we saw the toggle value XML for a Power On/Off toggle button. Here are the corresponding actions for that button:

```
<array>
    <dict>
        <key>deviceType</key>
        <integer>0</integer>
        <key>commandType</key>
        <integer>1</integer>
        <key>toggleValue</key>
        <string></string>
    </dict>
    <dict>
        <key>deviceType</key>
        <integer>0</integer>
        <key>commandType</key>
        <integer>2</integer>
        <key>toggleValue</key>
        <string></string>
    </dict>
</array>
```

Macro button. A macro button is a button that performs multiple actions in a predetermined sequence. The only thing distinguishing a macro button from a normal button is that the macro button's **actions** array contains more than one action.

The format of each action in the array is the same as what we have seen for other **actions** arrays, but macro buttons always have delay action between their command or script actions. The following macro therefore contains 3 command actions and 2 delay actions:

```
<array>
    <dict>
        <key>deviceType</key>
        <integer>0</integer> <!-- Television -->
        <key>commandType</key>
        <integer>51</integer> <!-- Up -->
        <key>toggleValue</key>
        <string></string>
    </dict>
    <dict>
        <key>deviceId</key> <!-- 200 ms delay -->
        <integer>-1</integer>
        <key>commandId</key>
        <integer>200</integer>
        <key>toggleValue</key>
        <string></string>
    </dict>
    <dict>
        <key>deviceType</key>
        <integer>0</integer> <!-- Television -->
        <key>commandType</key>
        <integer>49</integer> <!-- Left -->
        <key>toggleValue</key>
        <string></string>
    </dict>
    <dict>
        <key>deviceId</key> <!-- 1 s delay -->
        <integer>-1</integer>
        <key>commandId</key>
        <integer>1000</integer>
        <key>toggleValue</key>
        <string></string>
    </dict>
    <dict>
        <key>deviceType</key>
        <integer>0</integer> <!-- Television -->
        <key>commandType</key>
        <integer>10</integer> <!-- Select -->
        <key>toggleValue</key>
        <string></string>
    </dict>
</array>
```

Image Controls

All image controls in a layout appear within the “images” dictionary.

```
<key>images</key>
<dict>
    <key>31</key>
        <!-- XML for image control with ID 31 -->
</dict>
```

Each individual image has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>images</key>
<dict>
    <key>31</key> <!-- this must match the image ID -->
    <dict>
        <key>id</key>
        <integer>31</integer> <!-- this must match the key -->
        <!-- more image properties -->
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name. You can choose to display the name just below the control (see **showLabel**).
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **showLabel.** Indicates whether the name of the control will be displayed in a text label just underneath the control. Valid values are “yes” and “no”.
7. **width.** The width (in pixels) of the control.
8. **height.** The height (in pixels) of the control.
9. **baseVariableName.** The name of the variable that contains the URL for the control’s main image. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.
10. **baseVariableValue.** If the import process creates a variable for **baseVariableName**, it assigns this value.
11. **activeVariableName.** The name of the variable that contains the URL for the control’s highlighted (active) image. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.

12. **activeVariableValue.** If the import process creates a variable for **activeVariableName**, it assigns this value.
13. **zIndex.** Specifies a “depth” level for this image. When layering image controls on top of one another, controls with a higher **zIndex** will display in front.
14. **repeatInterval.** Indicates the frequency with which a control’s action is repeated, in milliseconds. For a repeating control, this must be something greater than 0 – usually 50 milliseconds – and there should be only one action in the control’s action list. If the button is of any other type (normal, toggle, macro), this value must be 0.
15. **timeoutInterval.** Indicates how long a control’s action will be repeated with no communication from a client controller before the repeat will be cancelled (to prevent uncontrolled repetitions in the case of a network failure or other exception). For a repeating control, this must be something greater than 0 – usually 2500 milliseconds. For a button of any other type (normal, toggle, macro) this value must be 0.

```
<dict>
    <key>id</key>
    <integer>31</integer>
    <key>xPosition</key>
    <real>41</real>
    <key>yPosition</key>
    <real>214</real>
    <key>name</key>
    <string>SampleImage</string>
    <key>description</key>
    <string>A sample image control</string>
    <key>showLabel</key>
    <string>no</string>
    <key>width</key>
    <real>100</real>
    <key>height</key>
    <real>100</real>
    <key>baseVariableName</key>
    <string>SampleImageBaseVariable</string>
    <key>baseVariableValue</key>
    <string>http://thinkflood.com/media/Television.png</string>
    <key>activeVariableName</key>
    <string>SampleImageActiveVariable</string>
    <key>activeVariableValue</key>
    <string>http://thinkflood.com/media/TelevisionHighlighted.png
</string>
    <key>zIndex</key>
    <integer>1</integer>
    <key>repeatInterval</key>
    <integer>0</integer>
    <key>timeoutInterval</key>
    <integer>0</integer>
    <key>isBackground</key>
    <string>no</string>
    <key>actions</key>
        <!-- XML for image action -->
</dict>
```

Image Action

The **actions** property of an image control follows the same formatting rules as for a button, with one exception: images never contain more than a single action. The actions array can be empty if the image is for display only, or it can have a single action if the image functions as a custom button (or repeating custom button). The action can reference a command, or contain a script. But there is no concept of a toggle or macro image control, and so the array will never have more than one action.

Label Controls

All label controls in a layout appear within the “labels” dictionary.

```
<key>labels</key>
<dict>
    <key>33</key>
        <!-- XML for label control with ID 33 -->
</dict>
```

Each individual label has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>labels</key>
<dict>
    <key>33</key> <!-- this must match the label ID -->
    <dict>
        <key>id</key>
        <integer>33</integer> <!-- this must match the key -->
        <!-- more label properties -->
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name.
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **width.** The width (in pixels) of the control.
7. **fontSize.** The font size (in pixels) of the label.
8. **justification.** The text justification to use when presenting the label. The three valid values are 0 (left), 1 (center), and 2 (right).
9. **valueVariableName.** The name of the variable that contains the text for the label. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.
10. **valueVariableValue.** If the import process creates a variable for **valueVariableName**, it assigns this value.
11. **color.** The color to apply to the text. Specified as a hexadecimal, RGB triplet (e.g., FFFFFF = white, FF0000 = red, 00FF00 = green, 0000FF = blue).

12. **highlightColor.** The color to apply to the text when the label is tapped. Formatting is the same as for the **color** property.
13. **bold.** Indicates whether the text should use bold-style font. Valid values are “yes” and “no”.
14. **repeatInterval.** Indicates the frequency with which a control’s action is repeated, in milliseconds. For a repeating control, this must be something greater than 0 – usually 50 milliseconds – and there should be only one action in the control’s action list. If the button is of any other type (normal, toggle, macro), this value must be 0.
15. **timeoutInterval.** Indicates how long a control’s action will be repeated with no communication from a client controller before the repeat will be cancelled (to prevent uncontrolled repetitions in the case of a network failure or other exception). For a repeating control, this must be something greater than 0 – usually 2500 milliseconds. For a button of any other type (normal, toggle, macro) this value must be 0.

```
<dict>
    <key>id</key>
    <integer>33</integer>
    <key>xPosition</key>
    <real>157</real>
    <key>yPosition</key>
    <real>245</real>
    <key>name</key>
    <string>SampleLabel</string>
    <key>description</key>
    <string>A sample label control</string>
    <key>width</key>
    <real>100</real>
    <key>fontSize</key>
    <real>14</real>
    <key>justification</key>
    <integer>1</integer>
    <key>valueVariableName</key>
    <string>SampleLabelTextVariable</string>
    <key>valueVariableValue</key>
    <string>This is text displayed in the label control.</string>
    <key>color</key>
    <string>ffffff</string>
    <key>highlightColor</key>
    <string>00a5d9</string>
    <key>bold</key>
    <string>no</string>
    <key>repeatInterval</key>
    <integer>0</integer>
    <key>timeoutInterval</key>
    <integer>0</integer>
    <key>actions</key>
        <!-- XML for label action -->
</dict>
```

Label Action

The **actions** property of a label control follows the same formatting rules as for a button, with one exception: labels never contain more than a single action. The actions array can be empty if the label is for display only, or it can have a single action if the label functions as a hyperlink-style button (or repeating custom button). The action can reference a command, or contain a script. But there is no concept of a toggle or macro label control, and so the array will never have more than one action.

Slider Controls

All slider controls in a layout appear within the “sliders” dictionary.

```
<key>sliders</key>
<dict>
    <key>34</key>
        <!-- XML for slider control with ID 34 -->
</dict>
```

Each individual slider has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>sliders</key>
<dict>
    <key>34</key> <!-- this must match the slider ID -->
    <dict>
        <key>id</key>
        <integer>34</integer> <!-- this must match the key -->
        <!-- more slider properties -->
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name. You can choose to display the name just below the control (see **showLabel**).
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **showLabel.** Indicates whether the name of the control will be displayed in a text label just underneath the control. Valid values are “yes” and “no”.
7. **currentValueVariableName.** The name of the variable that contains the current value of the slider. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.
8. **currentValueVariableValue.** If the import process creates a variable for **currentValueVariableName**, it assigns this value.
9. **increment.** The smallest possible step allowed when adjusting the slider value.
10. **maximumValue.** The slider’s highest possible value.
11. **minimumValue.** The slider’s lowest possible value.
12. **sliderType.** Reserved for future use. Should be 0.
13. **showValue.** Indicates whether the slider should display the current value in a text label. Valid values are “yes” and “no”.

```
<dict>
    <key>id</key>
    <integer>34</integer>
    <key>xPosition</key>
    <real>0</real>
    <key>yPosition</key>
    <real>313</real>
    <key>name</key>
    <string>SampleSlider</string>
    <key>description</key>
    <string>A sample slider control</string>
    <key>showLabel</key>
    <string>no</string>
    <key>currentValueVariableName</key>
    <string>SampleSliderValueVariable</string>
    <key>currentValueVariableValue</key>
    <string>21</string>
    <key>increment</key>
    <real>1</real>
    <key>maximumValue</key>
    <real>100</real>
    <key>minimumValue</key>
    <real>0</real>
    <key>sliderType</key>
    <integer>0</integer>
    <key>showValue</key>
    <string>yes</string>
    <key>actions</key>
        <!-- XML for slider action -->
</dict>
```

Slider Action

The **actions** property of a slider control follows the same formatting rules as for a button, with one exception: sliders never contain more than a single action. The action can reference a command, or contain a script. But there is no concept of a toggle or macro slider control, and so the array will never have more than one action.

HTML Controls

All HTML controls in a layout appear within the “webs” dictionary.

```
<key>webs</key>
<dict>
    <key>28</key>
        <!-- XML for HTML control with ID 28 -->
</dict>
```

Each individual HTML control has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>sliders</key>
<dict>
    <key>28</key> <!-- this must match the HTML ID -->
    <dict>
        <key>id</key>
        <integer>28</integer> <!-- this must match the key -->
        <!-- more HTML properties -->
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name. You can choose to display the name just below the control (see **showLabel**).
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **showLabel.** Indicates whether the name of the control will be displayed in a text label just underneath the control. Valid values are “yes” and “no”.
7. **width.** The width (in pixels) of the control.
8. **height.** The height (in pixels) of the control.
9. **selectedValueVariableName.** The name of the variable that contains the selected value when an HTML link is tapped. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.
10. **selectedValueVariableValue.** If the import process creates a variable for **selectedValueVariableName**, it assigns this value.
11. **contentVariableName.** The name of the variable that contains the HTML to be displayed. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.

12. **contentVariableValue.** If the import process creates a variable for **contentVariableName**, it assigns this value.[26]

```
<dict>
    <key>id</key>
    <integer>28</integer>
    <key>xPosition</key>
    <real>0</real>
    <key>yPosition</key>
    <real>367</real>
    <key>name</key>
    <string>0</string>
    <key>description</key>
    <string></string>
    <key>showLabel</key>
    <string>no</string>
    <key>width</key>
    <real>320</real>
    <key>height</key>
    <real>83</real>
    <key>selectedValueVariableName</key>
    <string>SampleHTMLValueVariable</string>
    <key>selectedValueVariableValue</key>
    <string></string>
    <key>contentVariableName</key>
    <string>SampleHTMLContentVariable</string>
    <key>contentVariableValue</key>
    <string>&lt;p&gt;The content displayed in the HTML control.&l
t;/p&gt;</string>
    <key>actions</key>
        <!-- XML for the HTML action -->
</dict>
```

HTML Action

The **actions** property of an HTML control follows the same formatting rules as for a button, with one exception: HTML controls never contain more than a single action. The action can reference a command, or contain a script. But there is no concept of a toggle or macro HTML control, and so the array will never have more than one action.

[26] As with all values inserted into the XML document, this must be XML-escaped. While many other values (names, etc) may not have any characters that need escaping, HTML virtually guarantees the need to escape. There are a number of free online utilities that can do this, including http://www.freeformatter.com/xml-escape.html.

Camera Controls

All camera controls in a layout appear within the “cameras” dictionary.

```
<key>cameras</key>
<dict>
    <key>35</key>
        <!-- XML for camera control with ID 35 -->
</dict>
```

Each individual HTML has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>cameras</key>
<dict>
    <key>35</key> <!-- this must match the camera ID -->
    <dict>
        <key>id</key>
        <integer>35</integer> <!-- this must match the key -->
        <!-- more camera properties -->
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name. You can choose to display the name just below the control (see **showLabel**).
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **showLabel.** Indicates whether the name of the control will be displayed in a text label just underneath the control. Valid values are “yes” and “no”.
7. **width.** The width (in pixels) of the control.
8. **height.** The height (in pixels) of the control.
9. **username.** The username for accessing password-protected camera feeds.
10. **password.** The password for accessing password-protected camera feeds.
11. **allowClearTextPassword.** Indicates whether the password may be sent in “clear text” (i.e., when using an unencrypted (HTTP) network connection). Valid values are “yes” and “no”.

12. **allowSelfSignedCertificates.** Indicates whether when using an encrypted (HTTPS) connection the client should skip validation of the feed's certificate authority. Valid values are "yes" and "no".
13. **urlVariableName.** The name of the variable that contains the URL for the camera feed. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.
14. **urlVariableValue.** If the import process creates a variable for **urlVariableName**, it assigns this value.

```
<dict>
    <key>id</key>
    <integer>35</integer>
    <key>xPosition</key>
    <real>77</real>
    <key>yPosition</key>
    <real>499</real>
    <key>name</key>
    <string>SampleCamera</string>
    <key>description</key>
    <string>A sample camera control</string>
    <key>showLabel</key>
    <string>no</string>
    <key>width</key>
    <real>160</real>
    <key>height</key>
    <real>120</real>
    <key>username</key>
    <string>Username</string>
    <key>password</key>
    <string>Password</string>
    <key>allowClearTextPasswords</key>
    <string>yes</string>
    <key>allowSelfSignedCertificates</key>
    <string>no</string>
    <key>urlVariableName</key>
    <string>SampleCameraURLVariable</string>
    <key>urlVariableValue</key>
    <string>http://127.0.0.1/MyCameraFeed</string>
</dict>
```

Text Controls

All text controls in a layout appear within the "texts" dictionary.

```
<key>texts</key>
<dict>
    <key>36</key>
        <!-- XML for text control with ID 36 -->
</dict>
```

Each individual text field has the following properties:

1. **id**. This is the unique ID of the control. IDs must be unique across the entire layout. The id element within the button XML must also equal the value for the corresponding key node in the parent dictionary. For example:

```
<key>texts</key>
<dict>
    <key>36</key> <!-- this must match the text field ID -->
    <dict>
        <key>id</key>
        <integer>36</integer> <!-- this must match the key -->
        <!-- more text field properties -->
    </dict>
</dict>
```

2. **xPosition.** The X (horizontal) coordinate of the top-left corner of the control. All RedEye layouts are based on a screen that is 320 pixels wide, and then scaled to match the actual display area.
3. **yPosition.** The Y (vertical) coordinate of the top-left corner of the control.
4. **name.** The control name.
5. **description.** The control description. Often used with accessibility (e.g., to inform someone with visual impairment what the control does).
6. **placeholder.** Text to display inside the text field when the field is blank.
7. **width.** The width (in pixels) of the control.
8. **fontSize.** The font size (in pixels) of the text field.
9. **color.** The color to apply to the text. Specified as a hexadecimal, RGB triplet (e.g., FFFFFF = white, FF0000 = red, 00FF00 = green, 0000FF = blue).
10. **backgroundColor.** The color to apply to the region behind the text. Formatting is the same as for the **color** property.
11. **showBorder.** Indicates whether a thin border line should be drawn around the text field. Valid values are “yes” and “no”.
12. **valueVariableName.** The name of the variable that contains the value of the text field. If the target RedEye does not contain a variable with this name (case sensitive), the import process will create one.
13. **valueVariableValue.** If the import process creates a variable for **valueVariableName**, it assigns this value.

```
<dict>
    <key>id</key>
    <integer>36</integer>
    <key>xPosition</key>
    <real>82</real>
    <key>yPosition</key>
    <real>645</real>
    <key>name</key>
    <string>SampleTextField</string>
    <key>description</key>
    <string>A sample text field control</string>
    <key>width</key>
    <real>150</real>
    <key>fontSize</key>
    <real>14</real>
    <key>valueVariableName</key>
    <string>SampleTextFieldTextVariable</string>
    <key>valueVariableValue</key>
    <string></string>
    <key>color</key>
    <string>000000</string>
    <key>backgroundColor</key>
    <string>ffffff</string>
    <key>placeholder</key>
    <string>Placeholder Text</string>
    <key>showBorder</key>
    <string>yes</string>
    <key>actions</key>
        <!-- XML for text action -->
</dict>
```

Text Action

The **actions** property of a text control follows the same formatting rules as for a button, with one exception: text controls never contain more than a single action. The action can reference a command, or contain a script. But there is no concept of a toggle or macro text control, and so the array will never have more than one action.

Gesture Shortcuts

Motion and multi-touch gesture shortcuts appear in a dictionary node under the “shortcuts” key:

```
<key>shortcuts</key>
<dict>
    <!-- gesture shortcuts assigned here -->
</dict>
```

Within this dictionary, each key is a gesture shortcut ID. (The full list of gesture shortcuts is available through the System Data page of the browser application.) The value for each key is a number which indicates the ID of the control to which the shortcut is assigned. For example, if the “channel up” button in a layout has control ID 47, and we want to assign the “two-finger swipe right” (type 2) gesture to that button, then we would include the following key-value pair in the **shortcuts** dictionary:

```
<key>2</key> <!-- shortcut ID 2 = two-finger swipe right -->
<integer>47</integer> <!-- "channel up" button ID -->
```

Multiple Shortcuts on a Single Control

Most controls have a single function – for example, you tap a button, and it executes. Sliders are an exception to this rule – you can nudge them up, or you can nudge them down. As a result, we need to be able to assign a shortcut to each function of the slider.

In order to accommodate multiple functions on a control, we change the value type in the dictionary from “integer” to “real” and then add a decimal point after the control ID. The fractional value indicates the function on the control to which the shortcut is associated. Going back to sliders, “.1” means “nudge down” and “.2” means “nudge up”.

In the following example, the slider’s “nudge down” function is associated with the “flick left” motion gesture, while the “nudge down” function is associated with the “flick right” motion gesture.

```
<key>22</key> <!-- flick left -->
<real>34.2</real> <!-- control ID = 34, function = down -->
<key>21</key> <!-- flick right -->
<real>34.1</real> <!-- control ID = 34, function = up -->
```

Keyboard Shortcuts

Keyboard shortcuts appear in a dictionary node under the “keyboardShortcuts” key:

```
<key>keyboardShortcuts</key>
<dict>
    <!-- keyboard shortcuts assigned here -->
</dict>
```

Keyboard shortcut assignments work in the same way as gesture shortcut assignments – each key is a shortcut ID (taken from the keyboard shortcut enumeration, which is available on the System Data page of the browser application). Each value is a number indicating the control ID – and in the case of controls (such as sliders) that have multiple functions, a decimal point followed by the function ID.

```
<key>39</key> <!-- right arrow key -->
<real>34.2</real> <!-- control ID = 34, function = up -->
<key>37</key> <!-- left arrow key -->
<real>34.1</real> <!-- control ID = 34, function = down -->
<key>76</key> <!-- letter "O" key -->
<integer>5</integer> <!-- control ID = 5 -->
```

Part IV: External Application Programming Interface (API)

Integrating with Other Systems

At its heart, RedEye is an integration platform. As a control system, we usually think of this integration with RedEye at the center, mediating among different devices and protocols. But if we zoom out a bit there is also integration between client controllers and RedEye (provided by RedEye's iOS, Android, and browser applications), and there can also be integration with other control systems and client interfaces.

When designing an Application Programming Interface (API), we had to understand the purpose of integrating with other systems. Ultimately we decided that our RedEye API would facilitate control from other sources – the proliferation of platforms through which a RedEye system could be operated – but not the configuration of the RedEye unit itself.

Much of this decision was practical. Specifically,

1. Configuration is complex.
2. Configuration has its own set of rules.
3. We have to restrict access to certain data to comply with licensing agreements.

As a result, at least for the time being RedEye's API gives you the freedom to control your RedEye system from just about any other device or platform, but you will still need to configured networking, add devices, create activities, and adjust layouts using our software.

To allow for external control, RedEye needs only provide a few basic functions:

1. Sending commands.
2. Launching activities.
3. Updating state variables.

While it may be obvious why you would want to send commands and launch activities from outside the RedEye platform, the way we use state variables has made it important to be able to make changes to these values, as well. Because state variable updates are broadcast to RedEye client applications, they are the primary means for providing a dynamic user interfaces. In fact, when you send toggle commands or launch activities, the RedEye system updates internal state variables in order to communicate these changes to client applications. Thus, it is natural to change your custom state variables directly.

In addition to these functions, the ability to query the RedEye system to discover dynamic information about the configuration is also important. Although our API does not allow external systems to update the configuration, you should not have to modify those systems whenever the RedEye configuration changes. RedEye provides information on the following data entities:

1. RedEye hardware
2. Rooms
3. Devices
4. Commands
5. Activities
6. Variables

If you are integrating with RedEye for the first time you can grab a comprehensive listing of all the configured data entities of a given type (e.g., all devices or all variables). Then while your system is running you can ask for more fine-grained information, such as the details on a particular room, without having to retrieve the whole list.

Getting Some Good ReST

Having scoped out our API, the next question we had to answer was what technology to use. Here we had two overarching goals: first, to make the API as widely accessible as possible, and second, to make it intuitive and simple to access.

In terms of broad access, we knew that we wanted to use web technology – specifically Hypertext Transfer Protocol (HTTP). It is hard to argue against HTTP today, as there are so many platforms that handle it, and there are so many tools that make development quite simple. Indeed, all you is a basic text editor and a free web browser, and you can start building web interfaces that use HTTP.

In terms of making things intuitive and simple, we wanted an interface that has a limited number of easy-to-remember commands. It should be logically organized, human readable, and easy to code.

With these thoughts in mind, RedEye provides an API that uses the principles of Representational State Transfer – ReST. Specifically,

1. Our data hierarchy is clearly visible in the path for each request
2. The internal RedEye implementation is separate from our external API (which makes the API more stable to internal changes)
3. The RedEye unit does not store or depend on external client state
4. We make use of standard HTTP request/response architecture

We did make one compromise in our API which is not strictly “ReSTful” in that we rely solely on HTTP GET messages. GET is the most common kind of HTTP request, easily implemented in an HTML document using anchor tags (`<a href="somepage">link</a>`). Normally, ReSTful designs use GET for queries, but prefer another HTTP request method (POST, UPDATE, etc) when the result is some action on the server. The primary concern is that web crawlers and other “dumb” programs might accidentally invoke functions on the server if GET requests allow them to do so.

While this is a valid concern for websites, RedEye servers typically reside on private networks and are not fodder for web crawlers. The practical simplicity and broader support for HTTP GET requests was more important to us than making allowance for web crawlers. As a result, you can test any RedEye API call simply by pasting a URL into the address bar of a standard web browser.

We do make some distinction between data retrieval requests and those that have some associated action (i.e., sending a command, launching an activity, or updating a variable). In our API, actionable requests always require at least one parameter passed using the query string, whereas simple data retrieval requests ignore the query string. For example:

/redeye/rooms/-1/devices/192/commands/217

will retrieve information about command 217 on device 192, whereas

/redeye/rooms/-1/devices/192/commands/send?commandId=217

actually executes that command.

Accessing the API Server

You may already know that RedEye runs HTTP servers on ports 80 and 82 (the latter being a lightweight server optimized for speed and memory performance). RedEye's API server is yet another HTTP server, this one running on port 8080.[27]

When you send an HTTP (GET) request to the API server, you should always receive one of two HTTP status codes in response: either 200 (OK), or 400 (Bad Request). In the case of a Bad Request, the server returns a document containing the invalid request path.

For all successful requests, the server will return valid HTTP headers. For actionable requests (e.g., sending a command, launching an activity), the request body will be blank. For data retrieval requests, the body will contain a document with the data requested.

The API server will respond to data retrieval requests in either XML or JSON format. The default format is XML. If your request contains the HTTP "Accept" header with a value that contains the string "application: json", then the server will return a JSON document.

[27] Coincidentally, we wrote the API server using only our Lua scripting engine, which says something about the amount of flexibility and power you have available with that tool.

Data Retrieval

Whether you are designing your external RedEye interface or checking for changes before invoking a function, it is useful to be able to “look inside” your RedEye to know how things are organized. In particular, you will need to know the integer IDs for the rooms, devices, commands, and activities you wish to access. (Although internally RedEye associates your variables with IDs, you access them through their unique variable names.)

RedEye Server

The root of the API server is not the path “/” but rather, “/redeye/”. When you query this path, the result is some basic information about the RedEye hardware unit itself – serial number, MAC address, hardware type, etc. For example, the request

http://192.168.1.107:8080/redeye/

might return

```
<redeye serialNumber="A0000-00000"
        wifiMacAddress="00:23:87:00:00:00"
        lanMacAddress="00:23:87:00:00:01"
        hardwareType="RedEye Pro"
        description=""
        name="RedEye Pro" />
```

Please note that while there is no document header, all RedEye responses are encoded using UTF-8, and fully support international characters.

Also of importance: the RedEye API server is case-sensitive, so you must honor the casing documented here. In other words, the following queries are *not* the same as the one above and will result in “400 Bad Request” responses:

http://192.168.1.107:8080/RedEye/
http://192.168.1.107:8080/REDEYE/

As a rule, API requests should use lower-case. The exception is when you pass a variable name, variable value, or toggle value name, in which case you need to maintain the same capitalization that you used when creating the values originally.

State Variables

RedEye state variables are global to each RedEye unit, even if it is a RedEye Pro with multiple rooms configured. As a result, in the data hierarchy they reside just below the root server level:

http://192.168.1.107:8080/redeye/variables/

```
<variables>
    <variable variableName="nad-t765-power"
              variableValue="Off" />
    <variable variableName="nad-t765-tunerBand"
              variableValue="FM" />
    <variable variableName="nad-t765-volume"
              variableValue="-20" />
</variables>
```

You can query a single variable by appending the variable name to the path:

http://192.168.1.107:8080/redeye/variables/nad-t765-tunerBand/

```
<variable variableName="nad-t765-tunerBand"
          variableValue="FM" />
```

Notice that we have maintained the capital “B” in “tunerBand” in order to retrieve the variable correctly.

If your variable name contains foreign characters, you must URL-encode the variable name in order to send it as part of the path. For example, if I have a variable named 變量名, then I would use the following path to look up its value:

http://192.168.1.107:8080/redeye/variables/%E8%AE%8A%E9%87%8F%E5%90%8D/[28]

It is perfectly fine to URL-encode ASCII text, so when in doubt, the safest bet is to use URL-encoding. (Do not, however, URL-encode your text more than once, as the resulting decoded string will be the earlier URL-encoded value, and therefore will not match the variable name you want.)

[28] Most web development platforms – including JavaScript, PHP, Python, and so forth – include basic URL-encoding functions. In addition, there are a number of online URL-encoding utilities, such as this one: http://meyerweb.com/eric/tools/dencoder/.

Rooms

All RedEye units support the concept of rooms, though only RedEye Pro allows for more than one room on the same server. With the exception of variables and the server itself, all RedEye data belongs to a hierarchy chain that begins with a room at the top. For gen1 and gen2 RedEye hardware, there is only one room, and it has the ID 0. For all RedEye Pro hardware, there is a root room at ID -1 which contains all devices. Of course you do not have to memorize all this information, you can simply ask your RedEye unit and it will tell you:

http://192.168.1.107:8080/redeye/rooms/

```
<rooms>
    <room name="RedEye Pro"
          currentActivityId="-1"
          roomId="-1"
          description="" />
    <room name="Family Room"
          currentActivityId="-1"
          roomId="1075"
          description="" />
</rooms>
```

or

http://192.168.1.108:8080/redeye/rooms/

```
<rooms>
    <room name="Family Room"
          currentActivityId="-1"
          roomId="0"
          description="" />
</rooms>
```

You can ask for details on a particular room by appending its ID to the path:

http://192.168.1.108:8080/redeye/rooms/0/

```
<room name="Family Room"
      currentActivityId="-1"
      roomId="0"
      description="" />
```

Notice that one property of the room is the “currentActivityId”. This property tells you which activity is running at the moment (-1 means that there is no activity currently running).

Activities

You can retrieve the list of activities for a particular room by appending the path “activities” after the ID of the room:

http://192.168.1.107:8080/redeye/rooms/1075/activities/

```
<activities>
    <activity name="Launchpad"
              activityId="406"
              description=""
              activityType="99999" />
    <activity name="Listen to Music"
              activityId="324"
              description=""
              activityType="8" />
    <activity name="Watch Movie"
              activityId="326"
              description=""
              activityType="1" />
    <activity name="Watch TV"
              activityId="323"
              description=""
              activityType="0" />
</activities>
```

As with rooms, you can check the current details of a particular activity by appending the activity ID to the request path:

http://192.168.1.107:8080/redeye/rooms/1075/activities/326/

```
<activity name="Watch Movie"
          activityId="326"
          description=""
          activityType="1" />
```

Devices

Just as with activities, you can query the devices in a given room by appending the path “activities” after the ID of the room:

http://192.168.1.107:8080/redeye/rooms/1075/devices/

```
<devices>
    <device manufacturerName="Yamaha"
            description=""
            portType="infrared"
            deviceType="6"
            modelName=""
            displayName="AVR"
            deviceId="217" />
    <device manufacturerName="Roku"
            description=""
            portType="http"
            deviceType="13"
            modelName=""
            displayName="Roku"
            deviceId="528" />
    <device manufacturerName="Samsung"
            description=""
            portType="infrared"
            deviceType="0"
            modelName=""
            displayName="TV"
            deviceId="138" />
</devices>
```

Again, you can retrieve the details of a specific device by appending the device ID to the end of the path:

http://192.168.1.107:8080/redeye/rooms/1075/devices/138/

```
<device manufacturerName="Samsung"
        description=""
        portType="infrared"
        deviceType="0"
        modelName=""
        displayName="TV"
        deviceId="138" />
```

Commands

You can retrieve the list of commands for a particular device by appending the string “commands” after the device ID in the request path:

http://192.168.1.107:8080/redeye/rooms/1075/devices/138/commands/

```
<commands>
    <command commandId="135"
             description=""
             toggles="no"
             commandType="26"
             name="+100" />
    <command commandId="48"
             description=""
             currentToggleValueName="On"
             toggles="yes"
             commandType="0" name="Power">
        <toggleValues>
            <toggleValue name="On" />
            <toggleValue name="Off" />
        </toggleValues>
    </command>
    <command commandId="136"
             description=""
             toggles="no"
             commandType="30"
             name="Volume Up" />
</commands>
```

Notice how toggle command nodes contain additional attributes (including "currentToggleValueName") as well as a child node that contains all of the toggle values, sorted in order of execution.

If you need to check the current status of a command (for example, because you want to know the current toggle value), you can append the command ID to the end of the path:

http://192.168.1.107:8080/redeye/rooms/1075/devices/138/commands/48/

```
<command commandId="48"
         description=""
         currentToggleValueName="Off"
         toggles="yes"
         commandType="0"
         name="Power">
    <toggleValues>
        <toggleValue name="On" />
        <toggleValue name="Off" />
    </toggleValues>
</command>
```

Invoking Functions

Data retrieval is nice, but it does not produce any results. Once you know the IDs and values of the data you want, however, you can invoke RedEye functions in a similar manner.

Sending Commands

Function invocation URLs are similar to data retrieval URLs, except that they contain a verb followed by a query string that contains the command ID. So, while

http://192.168.1.107:8080/redeye/rooms/1075/devices/138/commands/135/

might retrieve this:

```
<command commandId="135"
         description=""
         toggles="no"
         commandType="26"
         name="+100" />
```

By contrast, the following request

http://192.168.1.107:8080/redeye/rooms/1075/devices/138/commands/send?commandId=135

returns a blank document but actually transmits the command. Here, notice that the "I" in "commandId" is capitalized – requests are case-sensitive.

What happens when you want to send a toggle command? Toggle commands can be tricky to manage, and you want to make sure that RedEye remains in sync. The RedEye API provides a variant on the basic send command function that allows you to send a toggle command and update RedEye's internal state tracking appropriately. Here is the syntax:

http://192.168.1.107:8080/redeye/rooms/1075/devices/138/commands/sendToggle?toggleValueName=On&commandId=48

The differences are:

1. Rather than "send" we use "sendToggle".

2. In addition to the “commandId” parameter in the query string, we add a “toggleValueName” parameter (again, case-sensitive) to specify the toggle value we want RedEye to achieve.
3. RedEye calculates the number of times it needs to execute the command to achieve the desired toggle value, plays the command that number of times, and then stores the new toggle state.

A few notes:

- Just as with RedEye’s scripting function (SendToggleCommand), if the toggle value does not exist, or if the current toggle value is the one requested, then nothing happens.
- The entire request path, including the query string (and including the toggle value name itself) is case-sensitive.
- If you want to play a toggle command without updating state (as with the “Adjust” function in RedEye client apps), then you can invoke the normal “send” function on a toggle command. In this case the command will be executed a single time for each request, and the toggle value state will not update.
- If your toggle value name contains non-ASCII characters, you must URL-encode it.

Launching Activities

As with commands, the syntax for launching an activity is not too far from the syntax for retrieving it. So,

http://192.168.1.107:8080/redeye/rooms/1075/activities/406/

yields this

```
<activity name="Launchpad"
          activityId="406"
          description=""
          activityType="99999" />
```

And this:

http://192.168.1.107:8080/redeye/rooms/1075/activities/launch?activityId=406

returns a blank document, but launches the activity. Again, please note the capital “I” in “activityId”.

Whereas most RedEye API calls should return quickly, launching an activity may take several seconds. Exactly how long depends on how many launch and shutdown actions you have and what the delays are between them. In other

words, you only receive a response from the API server when your RedEye has completely finished sending whatever command or launching whatever activity you specify.

As with the Scripting.LaunchActivity function, if you want to shut down the currently running activity, you can pass -1 for the activity ID, like this:

http://192.168.1.107:8080/redeye/rooms/1075/activities/launch?activityId=-1

Updating State Variables

Finally, you can also change the value of state variables. In this case you need to pass two query string arguments – one for the variable name, and another for the new value:

http://192.168.1.107:8080/redeye/variables/set?variableName=MyVariable&variableValue=Test

As with other requests, everything here is case-sensitive. If you are using non-ASCII characters for either the variable name or variable value, then you will need to URL-encode them. Finally, you *can* create a new variable using this request by simply setting its value (just as you would with the Scripting.SetVariable function call). You cannot, however, delete a variable using the API.

As with the rest of the RedEye system, whenever you set a state variable, the update is passed to all connected client controllers.

Part V: Scripting Function Reference

Scripting Class

RedEye's Scripting class provides hooks into the RedEye server application running on your RedEye hardware so that you can do things like send commands and launch activities. It also provides a handful of convenience methods for common data manipulation.

Scripting.BinaryToHex

Description
Converts a binary string to a human-readable hexadecimal format.

Parameters
binaryString
String. The binary string to convert.

Comments
Historically, RS-232 and TCP were designed to be used with teletype terminals and therefore were fundamentally ASCII protocols. Nevertheless, some hardware manufacturers insist on sending binary data in their messages. This method makes it possible to convert a binary string to an ASCII string of human-readable hexadecimal characters.

This function is the inverse of **HexToBinary**.

Examples

```
local hexadecimalString = Scripting.BinaryToHex(binaryString)
```

Scripting.CloseRelay

Description
Closes a contact relay.

Parameters
relayNumber
Integer. The relay port number. Valid values are 1-4. For example, to retrieve the value for relay port R1, this value should be 1.

Comments
Relay port numbers are 1-based (i.e., there is no R0 port).

Examples

```
Scripting.CloseRelay(3)
```

```
local relayNumber =
    tonumber(Scripting.GetVariable("CurrentRelayNumber"))
Scripting.CloseRelay(relayNumber)
```

Scripting.DecodeFromHtml

Description

Decodes a block of text that has been encoded for use inside an HTML or XML document.

Parameters

text

String. The text to be decoded.

Comments

This function decodes text which has been encoded for inserting into an HTML or XML document. It also decodes international characters which have been hexadecimal encoded with their Unicode values.

This function is the inverse of **EncodeForUrl**.

Examples

```
local text = Scripting.DecodeFrom(encodedText)
```

Scripting.DecodeFromUrl

Description

Decodes a block of text that has been URL-encoded.

Parameters

text

String. The text to be decoded.

Comments

Decodes text in accordance with IETF RFC 3986 (see http://tools.ietf.org/html/rfc3986). This function is the inverse of **EncodeForUrl**.

Examples

```
local queryValue = Scripting.DecodeFromUrl(encodedQueryValue)
```

Scripting.EncodeForHtml

Description

Encodes a block of text for use inside an HTML or XML document.

Parameters

text

String. The text to be encoded.

Comments

RedEye uses UTF-8 character encoding natively, so special handling for international characters is not necessary. However, there are a handful of characters reserved for use in HTML and XML tags. This function encodes those characters so that you can safely incorporate them into your HTML or XML strings.

This function is the inverse of **DecodeFromHtml**.

Examples

```
local encodedText = Scripting.EncodeForHtml(text)
```

Scripting.EncodeForUrl

Description

Encodes a block of text for use in a URL (query string) or as HTTP POST data.

Parameters

text

String. The text to be encoded.

Comments

Encodes text in accordance with IETF RFC 3986 (see http://tools.ietf.org/html/rfc3986). Although this standard is relatively new and some HTTP servers will accept data that is not URL-encoded, it is always safest to pass your query string values through this method before including in an HTTP request path. Please note that you should *not* pass the entire query string through this method, as that would encode the ampersand (&) and equal sign (=) characters that should be present.

This function is the inverse of **DecodeFromUrl**.

Examples

```
local encodedQueryValue = Scripting.EncodeForUrl(queryValue)
```

Scripting.GetActivityForRoom

Description

Retrieves the current activity ID for the specified room.

Parameters

roomId

Integer. The ID of the room to which the activity belongs. For “whole-house” activities on RedEye Pro, the ID is -1.

Comment

Returns -1 if there is no activity running in the room.

On RedEye Pro you can create rooms and add activities to them. You can also add activities to the RedEye Pro itself. The latter case is useful for creating “whole house” or “party mode” activities (or if you are using RedEye Pro for single zone control). The “root” RedEye Pro room ID is -1.

Examples

```
local currentActivityId = Scripting.GetActivityForRoom(42)
```

```
local currentActivityId =
    Scripting.GetActivityForRoom(42--[[Family Room]])
```

Scripting.GetRelayValue

Description

Retrieves the current value of a contact relay. 0 is open, 1 is closed.

Parameters

relayNumber

Integer. The relay port number. Valid values are 1-4. For example, to retrieve the value for relay port R1, this value should be 1.

Comments

Relay port numbers are 1-based (i.e., there is no R0 port).

Examples

```
local r1Value = Scripting.GetRelayValue(1)
```

```
local relayNumber =
    tonumber(Scripting.GetVariable("CurrentRelayNumber"))
local relayValue = Scripting.GetRelayValue(relayNumber)
```

```
local relay2IsClosed = (Scripting.GetRelayValue(2) == 1)
```

Scripting.GetSensorValue

Description

Retrieves the current value of a contact sensor. 0 is open, 1 is closed.

Parameters

sensorNumber

Integer. The sensor port number. Valid values are 1-8. For example, to retrieve the value for relay port I1, this value should be 1.

Comments

Sensor port numbers are 1-based (i.e., there is no I0 port).

This function is only valid on ports that have been configured to sensor input mode. Calling this function on a port that is set for infrared output will not return an error, but the result is undefined.

Examples

```
local i7Value = Scripting.GetSensorValue(7)
```

```
local sensorNumber =
    tonumber(Scripting.GetVariable("CurrentSensorNumber"))
local sensorValue = Scripting.GetSensorValue(sensorNumber)
```

```
local sensor4IsOpen = (Scripting.GetSensorValue(4) == 0)
```

Scripting.GetToggleStateForCommand

Description

Retrieves the current toggle state for the command specified.

Parameters

commandId

Integer. The ID of the toggle command in question.

Comments

If the command is not a toggle command, or the toggle state has not yet been stored (because the command has never been executed), returns the empty (zero-length) string.

Examples

```
local toggleState = Scripting.GetToggleStateForCommand(551)
```

```
local toggleState =
    Scripting.GetToggleStateForCommand(551--[[Command: Down]])
```

Scripting.GetVariable

Description

Retrieves the value of a state variable.

Parameters

variableName

String. The name of the variable. Variable names are case sensitive.

Comments

Return values are always of type “string,” so you will need to cast to a number (using Lua’s tonumber() function) if you need a numeric value. If the variable does not exist, returns the empty (zero-length) string.

Examples

```
local trackName = Scripting.GetVariable("TrackName")
if #trackName > 0 then
    -- do something with trackName
end
```

```
local playCount = Scripting.GetVariable("PlayCount")
if tonumber(playCount) > 5 then
    -- do something
end
```

Scripting.HexToBinary

Description

Converts a string of hexadecimal characters to its binary string equivalent.

Parameters

hexString

String. The hexadecimal string to convert.

Comments

Historically, RS-232 and TCP were designed to be used with teletype terminals and therefore were fundamentally ASCII protocols. Nevertheless, some hardware manufacturers insist on sending binary data in their messages. One option for handling generating these binary strings is to first calculate the decimal value of each hexadecimal pair and then using Lua’s string escape sequence (*ddd*,

where *ddd* is the decimal value) to build the string. This method simplifies the process so you can enter hexadecimal values directly based on documentation. Please note that we are still dealing with a string of hexadecimal characters – you must enclose your hexadecimal values in quotation marks (") or Lua will assume you are passing a variable name.

Valid hexadecimal strings will always have an even number of characters. If your hexString parameter contains an odd number of characters, this function will return nil, followed by an error message. Likewise, hexadecimal strings should be composed only of the characters 0-9 and A-F (casing does not matter). If your string contains other characters, this function will return nil followed by an error message.

This function is the inverse of **BinaryToHex**.

Examples

```
local binaryString = Scripting.HexToBinary("FFFFFF")
```

```
local hexString = "A1B2C3D4E5F6e7d8c9ba"
local binaryString = Scripting.HexToBinary(hexString)
```

Scripting.LaunchActivity

Description

Launches an activity within a specified room.

Parameters

roomId

Integer. The ID of the room to which the activity belongs. For “whole-house” activities on RedEye Pro, the ID is -1.

activityId

Integer. The ID of the activity to launch. Send -1 to shutdown the currently running activity.

Comments

When launching an activity, RedEye coalesces the shutdown actions of the current activity with the launch actions of the requested activity. For example, if the current activity would power down the television, but the new activity would power it back on, RedEye will skip the two actions and their associated inter-action delays. After coalescing, RedEye executes the remaining shutdown actions delays in order, followed by the remaining launch actions and delays. Once all actions are complete, the current activity ID is updated to reflect the new activity.

If the activity ID requested is the same as the currently running activity ID, this function does nothing. If you wish to shut down the current activity, the you should pass -1 for the activity ID.

On RedEye Pro you can create rooms and add activities to them. You can also add activities to the RedEye Pro itself. The latter case is useful for creating "whole house" or "party mode" activities (or if you are using RedEye Pro for single zone control). The "root" RedEye Pro room ID is -1.

Examples

```
Scripting.LaunchActivity(42, 357)
```

```
Scripting.LaunchActivity(42--[[Family Room]], 357--[[Watch TV]])
```

```
Scripting.LaunchActivity(42, -1)
```

```
Scripting.LaunchActivity(-1--[[RedEye Pro]],
                         907--[[Listen to Radio]])
```

```
Scripting.LaunchActivity(-1, -1)
```

Scripting.OpenRelay

Description

Opens a contact relay.

Parameters

relayNumber

Integer. The relay port number. Valid values are 1-4. For example, to retrieve the value for relay port R1, this value should be 1.

Comments

Relay port numbers are 1-based (i.e., there is no R0 port).

Examples

```
Scripting.OpenRelay(4)
```

```
local relayNumber =
    tonumber(Scripting.GetVariable("CurrentRelayNumber"))
Scripting.OpenRelay(relayNumber)
```

Scripting.SendCommand

Description

Executes a command.

Parameters

deviceId

Integer. The ID of the device to which the command belongs.

commandId

Integer. The ID of the command to execute.

Comments

This function executes any kind of command – infrared or scripted.

In the case of infrared toggle commands, if you execute this method, the command plays once and RedEye's stored toggle state does not change. In other words, this is equivalent to the "adjust" method in the RedEye client interface. To execute an infrared toggle command with a state adjustment, you should call the related function, **SendToggleCommand**.

Example

```
Scripting.SendCommand(547, 551)
```

```
Scripting.SendCommand(547--[[Device: Apple TV]],
                      551--[[Command: Down]])
```

```
local selectedDeviceId =
    tonumber(Scripting.GetVariable("SelectedDevice"))
local selectedCommandId =
    tonumber(Scripting.GetVariable("SelectedCommand"))
Scripting.SendCommand(selectedDeviceId, selectedCommandId)
```

Scripting.SendHttpMessage

Description

Sends an HTTP request message on the current port.

Parameters

path

String, optional. The relative path (including query string, if needed) to append to the port base URL. Must be URL-encoded.

method

String, optional. The HTTP request method, usually GET or POST. Defaults to GET.

postData

String, optional. When the request method is POST, this text is uploaded to the HTTP server. As with the path, you must properly encode this data (usually XML- or HTML-encoded).

httpHeaders

Lua table, optional. Any request headers that should accompany your request. Table keys map to header names, while table values map to header values. RedEye will provide default headers for common headers such as “User-Agent” and “Content-Length”.

proxyUrl

String, optional. The URL for your HTTP proxy server. This is typically a security feature and uncommon for home control applications.

timeout

Integer, optional. The maximum amount of time that RedEye will wait for a response from the server, in seconds. Send -1 to wait indefinitely (not recommended, as your request will block others). Defaults to 5 seconds.

Comments

All function parameters are optional, but you must provide something. What you specify depends on what kind of HTTP request you are sending (GET versus POST) and whether you need something beyond the default headers and timeout values.

As with **SendMessage**, this function is useful in that it does not specify a particular port ID, which means that your script is generic and can be copied and reused in a number of places. However, it is only valid within a command script, where the device (and therefore the port association) can be inferred. If you need to send a message from another script, use **SendHttpMessageToPort** instead.

Examples

```
Scripting.SendHttpMessage("currentTrackName")
```

```
Scripting.SendHttpMessage("/keypress/Left", "POST")
```

```
local escapedCharacter = Scripting.EncodeForUrl(character)
Scripting.SendHttpMessage("/keypress/Lit",
                          "POST",
                          escapedCharacter)
```

```
local headers = {
    ["Accept"] = "text/plain",
    ["Accept-Charset"] = "utf-8",
    ["Authorization"] =
        "Basic\r\nQWxhZGRpbjpvcGVuIHNlc2FtZQ==\r\n"
}

Scripting.SendHttpMessage("setup/admin/", nil, nil, headers)
```

```
Scripting.SendHttpMessage("currentAlbum",
                          nil,
                          nil,
                          nil,
                          "http://proxy.somedomain.com:8080/")
```

```
Scripting.SendMessage("query/playlist", nil, nil, nil, nil, 10)
```

Scripting.SendHttpMessageToPort

Description

Sends an HTTP request message to the specified port.

Parameters

portId

Integer. The ID of the port to which the message should be sent.

path

String, optional. The relative path (including query string, if needed) to append to the port base URL. Must be URL-encoded.

method

String, optional. The HTTP request method, usually GET or POST. Defaults to GET.

postData

String, optional. When the request method is POST, this text is uploaded to the HTTP server. As with the path, you must properly encode this data (usually XML- or HTML-encoded).

httpHeaders

Lua table, optional. Any request headers that should accompany your request. Table keys map to header names, while table values map to header values. RedEye will provide default headers for common headers such as “User-Agent” and “Content-Length”.

proxyUrl

String, optional. The URL for your HTTP proxy server. This is typically a security feature and uncommon for home control applications.

timeout

Integer, optional. The maximum amount of time that RedEye will wait for a response from the server, in seconds. Send -1 to wait indefinitely (not recommended, as your request will block others). Defaults to 5 seconds.

Comments

Other than portId, all function parameters are optional, but you must provide something. What you specify depends on what kind of HTTP request you are sending (GET versus POST) and whether you need something beyond the default headers and timeout values.

As with **SendMessageToPort**, this function is a companion to **SendHttpMessage** and necessary if you are sending a message from within a non-command script (i.e., port script, action script). However, because it explicitly identifies the port ID, your script is not reusable. One exception is within port scripts where you can use a previously-defined port ID variable and avoid hardcoding a port ID in the function call.

Examples

```
Scripting.SendHttpMessageToPort(915, "system/status/")
```

```
Scripting.SendHttpMessageToPort(915, "system/sendKey(5)", "POST")
```

```
local request = [[
    <request>
        <!-- some request data here -->
    </request>
]]

Scripting.SendHttpMessageToPort(915,
                                "/execute/",
                                "POST",
                                request)
```

```
local headers = {
    ["Accept-Language"] = "en-US"
}

Scripting.SendHttpMessageToPort(915,
                                "/system/status/",
                                nil,
                                nil,
                                headers)
```

```
local encodedRequest = Scripting.EncodeForUrl(request)
local headers = {}
headers["Content-Type"] = "application/x-www-form-urlencoded"
local proxyUrl = "http://myproxy.mydomain.net/"

Scripting.SendHttpMessageToPort("/execute/",
                                "POST",
                                encodedRequest,
                                headers,
                                proxyUrl)
```

```
Scripting.SendMessage("query/nextTrack", nil, nil, nil, nil, 2)
```

Scripting.SendMessage

Description

Sends a message on the current port.

Parameters

message

String. The message to send.

Comments

Used for communication over RS-232, TCP, and UDP ports.

This function is useful in that it does not specify a particular port ID, which means that your script is generic and can be copied and reused in a number of places. However, it is only valid within a command script, where the device (and therefore the port association) can be inferred. If you need to send a message from another script, use **SendMessageToPort** instead.

Examples

```
Scripting.SendMessage("Main.Power=On")
```

```
local requestedTrackId =
    Scripting.GetVariable("RequestedTrackId")

Scripting.SendMessage("Zone1.SetTrackId=" ..
                      requestedTrackId .. "\r\n")
```

Scripting.SendMessageToPort

Description

Sends a message to the specified port.

Parameters

message

String. The message to send.

portId

Integer. The ID of the port to which the message should be sent.

Comments

Used for communication over RS-232, TCP, and UDP ports.

As a companion to **SendMessage**, this function is necessary if you are sending a message from within a non-command script (i.e., port script, action script); however, because it explicitly identifies the port ID, your script is not reusable.

One exception is within port scripts where you can use a previously-defined port ID variable and avoid hardcoding a port ID in the function call.

Examples

```
Scripting.SendMessageToPort("01/1/ALPHABETIZE_COVER_ART:\r\n",
                            898)
```

```
local queryText = Scripting.GetVariable("AlbumQueryText")
local escapedQuery = Scripting.EncodeForUrl(queryText)

Scripting.SendMessageToPort("search?query=" .. escapedQuery, 898)
```

Scripting.SendToggleCommand

Description

Executes a command until a specified toggle state is achieved.

Parameters

deviceId

Integer. The ID of the device to which the command belongs.

commandId

Integer. The ID of the command to execute.

toggleValueName

String. The name of the target toggle state. Toggle names are case sensitive.

Comments

This function executes an infrared toggle command until a particular toggle state is achieved. If you invoke this function on an ID that does not correspond to an infrared to command, nothing will happen.

Internally, this function looks up the current toggle state of the command in RedEye's database and calculates the number of command executions required to arrive at the new toggle state specified in the toggleValueName parameter you pass. Once execution is complete, the new toggle state is stored in RedEye's internal database. If toggleValueName does not match a valid toggle state for the command, or if toggleValueName matches the current toggle state, nothing will happen.

If you want to execute a toggle command without changing the toggle state stored in your RedEye, you should call the related function, **SendCommand**. **SendCommand** will execute a toggle command a single time without updating the database, which is the equivalent of the RedEye client "adjust" function.

Example

```
Scripting.SendToggleCommand(842, 844, "On")
```

```
Scripting.SendToggleCommand(842--[[Device: TV]],
                            844--[[Command: Power]],
                            "On")
```

```
local selectedDeviceId =
    tonumber(Scripting.GetVariable("SelectedDevice"))
local selectedCommandId =
    tonumber(Scripting.GetVariable("SelectedCommand"))
local selectedToggleValue =
    Scripting.GetVariable("SelectedToggleValue")
Scripting.SendToggleCommand(selectedDeviceId,
                            selectedCommandId,
                            selectedToggleValue)
```

Scripting.SetVariable

Description

Stores the value of a state variable.

Parameters

variableName

String. The name of the variable. Variable names are case-sensitive.

variableValue

Any type. The value of the variable. Converted to a string using Lua's tostring() function.

Comments

Overwrites the value of the existing state variable with the same name. If no state variable exists, creates a new one.

All state variables are stored as strings. If you need to store the contents of a table or other complex data type, you should convert to XML or some other serialized form first.

Examples

```
Scripting.SetVariable("CurrentTemperature", "71°F")
```

```
Scripting.SetVariable("CurrentTrackName", "")
```

```
Scripting.SetVariable("PlayCount", 5)
```

```
local serializedPlaylist = table.concat(playlist, "\n")
Scripting.SetVariable("Playlist", serializedPlaylist)
```

Scripting.Wait

Description

Halts script execution for a period of time.

Parameters

milliseconds

Integer. The number of milliseconds to wait.

Comments

This function blocks the current thread and yields execution to other threads in the application.

Examples

```
Scripting.SendMessage("first message")
Scripting.Wait(50)
Scripting.SendMessage("second message")
```

```
local status = Scripting.GetVariable("traystatus")
local waitTime = 0

while status == "closed" do
    if waitTime > 2500 then
        break
    end

    Scripting.SendMessage("tray:open")
    Scripting.Wait(500)
    waitTime = waitTime + 500
end
```

Scripting.WakeOnLan

Description

Sends a Wake-on-LAN “magic packet” to the device with the MAC address specified.

Parameters

macAddress

String. The MAC address of the device to wake up.

broadcastAddress

String, optional. The broadcast address of your network. Defaults to 255.255.255.255 (the local network).

Comments

This function formats the required “magic packet” and transmits it via UDP three times to port 7 and three times to port 9 on the broadcastAddress specified, which is usually more than adequate to wake a device that has WOL enabled. Normally the goal is to wake a device on the local network, so in practice you can omit the broadcastAddress from your function call.

Examples

```
Scripting.WakeOnLan("00:23:87:00:00:0A")
```

```
Scripting.WakeOnLan("00238700000a")
```

```
Scripting.WakeOnLan("00238700000a", "192.168.1.255")
```

PortListener Class

PortListener encapsulates functionality for receiving communication from controlled devices over RedEye ports. Because of its tight integration with RedEye's internal MessageProcessor class (which handles outbound communication on ports), it is the main utility for writing custom port scripts. PortListener provides hooks for communication using

- Sensors
- RS-232
- TCP
- UDP
- HTTP

Moreover, PortListener is structured in an extensible, object-oriented way so that you can easily add functionality as needed.

PortListener:new

Description

Creates a new instance of the PortListener class.

Parameters

metatable

Lua table, optional. If you are subclassing PortListener, you can pass a metatable for PortListener to use for internal storage.

Comments

Before you can use PortListener, you must first create an "instance" of it. Instances are distinct copies that have their own storage. Thus, multiple instances can be running simultaneously without interfering with each other (as long as they are each accessing different ports).

Although you can subclass PortListener, you can access all major functionality simply by creating an instance and making function calls on that instance.

Examples

```
local listener = PortListener:new()
```

```
local baseTable = {}
-- add some values to baseTable here
local listener = PortListener:new(baseTable)
```

[instance]:Listen

Description

Starts monitoring a port for activity and invokes callback methods as necessary.

Parameters

(none)

Comments

Prior to calling the **Listen** function on a PortListener, you must first identify the port (by calling **SetPortId**) and set your callback function(s) as appropriate for your port type. If you fail to set the port ID, then the Listen function will not do anything and your script will exit. If you fail to set the necessary callback function(s), then the PortListener will not be able to do anything when the status of the port changes.

Calling the **Listen** function blocks – that is, no code following the function call will execute. The PortListener remains in a constant listening state until the RedEye application shuts it down. The RedEye application will shut down your PortListener instance in any of the following cases:

1. The system is about to reboot
2. You save a new copy of your port script
3. You save a new port configuration
4. You move or delete the device associated with the port

If the device you are controlling is unavailable for a period of time, the PortListener will attempt to reconnect. If reconnecting has failed, sending an outbound command will trigger another reconnection attempt.

Examples

```
local function ProcessInputData(data)
    -- your custom code here
end

local portId = arg[1]
local listener = PortListener:new()
listener:SetPortId(portId)
listener:SetBytesAvailableCallback(ProcessInputData)
listener:Listen()
```

[instance]:SetBytesAvailableCallback

Description

Assigns the function used to respond when data has been received from an RS-232, TCP, or UDP port.

Parameters

method

Lua function. The function you have written to handle the data received.

Comments

The method you specify here should expect a single string argument when called by PortListener. This string contains the data received over the port.

The method you pass to PortListener through this function will only be invoked if you have also set your PortListener instance to monitor an RS-232, TCP, or UDP port. See **SetPortId** for more information on setting the port assignment.

Examples

```
local function ProcessInputData(data)
    -- your custom code here
end

local listener = PortListener:new()
listener:SetBytesAvailableCallback(ProcessInputData)
```

```
MySerialListener = {}

MySerialListener.ReceiveData(data) = function ()
    -- custom code here
end

local listener = PortListener:new()
listener:SetBytesAvailableCallback(MySerialListener.ReceiveData)
```

[instance]:SetHttpResponseCallback

Description

Assigns the function used to respond when a response has been received from an HTTP port.

Parameters

method

Lua function. The function you have written to handle the response.

Comments

The method you specify here should expect three arguments from PortListener:

statusCode

Integer. The HTTP status code from the response (e.g., 200, 301, 404)

responseHeaders

Lua table. The HTTP response headers. Header names are stored as table keys, and table values correspond to header values.

responseBody

String. The HTTP response message body, typically an HTML or XML document.

The method you pass to PortListener through this function will only be invoked if you have also set your PortListener instance to monitor an HTTP port. See **SetPortId** for more information on setting the port assignment.

Examples

```
local function ProcessHttpResponse(statusCode, headers, message)
    -- your custom code here
end

local listener = PortListener:new()
listener:SetHttpResponseCallback(ProcessHttpResponse)
```

```
MyHttpListener = {}

MyHttpListener.HandleResponse = function (code, headers, message)
    -- custom code here
end

local listener = PortListener:new()
listener:SetHttpResponseCallback(MyHttpListener.HandleResponse)
```

[instance]:SetPortId

Description

Sets the port ID monitored by the listener instance.

Parameters

portId

Integer. The ID of the port that you want to monitor.

Comments

PortListener uses the port ID to look up the characteristics of the port – its type and configuration. Thus, assigning the correct port ID is a critical step in making sure the proper port is monitored and your callback functions are invoked properly. If you do not set the port ID, then the PortListener will not connect to any port, and any call to the **Listen** method will have no effect.

The port ID is RedEye's internal ID, not the port name or number. For example, on a RedEye Pro, port I1 has the port ID 0, and port S1 has the port ID 12. (IP ports IDs are arbitrarily assigned therefore cannot be predetermined.) Normally you do not hardcode an ID, but instead use the port ID passed by RedEye when it launches your port script. If you must use a hardcoded port ID, then you should

be aware that your port script will not work if you move your device to another port, and could conflict with other port scripts.

Examples

```
local portId = arg[1]
local listener = PortListener:new()
listener:SetPortId(portId)
```

[instance]:SetSensorClosedCallback

Description

Assigns the function used to respond when a sensor transitions to a closed state.

Parameters

method

Lua function. The function you have written to handle the sensor closed transition.

Comments

The method you specify here should not expect to receive any arguments when called by PortListener.

The method you pass to PortListener through this function will only be invoked if you have also set your PortListener instance to monitor a sensor port. See **SetPortId** for more information on setting the port assignment.

Examples

```
local function OnSensorClosed()
    -- your custom code here
end

local listener = PortListener:new()
listener:SetSensorClosedCallback(OnSensorClosed)
```

```
MySensorListener = {}

MySensorListener.SensorClosed = function ()
    -- custom code here
end

local listener = PortListener:new()
listener:SetSensorClosedCallback(MySensorListener.SensorClosed)
```

[instance]:SetSensorOpenedCallback

Description

Assigns the function used to respond when a sensor transitions to an open state.

Parameters

method

Lua function. The function you have written to handle the sensor opened transition.

Comments

The method you specify here should not expect to receive any arguments when called by PortListener.

The method you pass to PortListener through this function will only be invoked if you have also set your PortListener instance to monitor a sensor port. See **SetPortId** for more information on setting the port assignment.

Examples

```
local function OnSensorOpened()
    -- your custom code here
end

local listener = PortListener:new()
listener:SetSensorOpenedCallback(OnSensorOpened)
```

```
MySensorListener = {}

MySensorListener.SensorOpened = function ()
    -- custom code here
end

local listener = PortListener:new()
listener:SetSensorOpenedCallback(MySensorListener.SensorOpened)
```

StringAggregator Class

StringAggregator is a utility for managing message fragments. It is particularly useful when processing data received from a port which may have been broken into inconvenient chunks, whether those chunks are incomplete messages or combinations of messages.

You configure a StringAggregator instance by telling it what the message boundary is (a string value). Once configured, you pass it data as it arrives, and StringAggregator returns to you the fully assembled messages.

StringAggregator:new

Description

Creates a new instance of the StringAggregator class.

Parameters

metatable

Lua table, optional. If you are subclassing StringAggregator, you can pass a metatable for StringAggregator to use for internal storage.

Comments

Before you can use StringAggregator, you must first create an “instance” of it. Instances are distinct copies that have their own storage. Thus, multiple instances can be running simultaneously without interfering with each other. This can be useful if you need to break down messages into sub-messages.

Although you can subclass StringAggregator, you can access all major functionality simply by creating an instance and making function calls on that instance.

Examples

```
local inputAggregator = StringAggregator:new()
```

```
local baseTable = {}
-- add some values to baseTable here
local aggregator = StringAggregator:new(baseTable)
```

[instance]:ProcessData

Description

Evaluates data, combining and splitting to create complete messages.

Parameters

dataString

String. The next chunk of data to process.

Comments

This method takes in data, looks for message boundaries, and combines or splits the data to create complete messages. It returns a Lua table containing the complete messages. StringAggregator will hold onto incomplete message parts across calls to this method, but there will never be more than one incomplete message in process at a time.

Although this method will always return a table, you should not assume that the table will always have values (if StringAggregator has only received message fragments, it will return an empty table). Likewise, you should be prepared to receive a table that contains more than one value (as the data could contain more than one message boundary). As a result, the safest way to process the return table is to loop through its members and process one at a time.

Examples

```
local messages = aggregator:ProcessData(receivedData)
for key, value in ipairs(messages) do
    -- process each value entry here
end
```

[instance]:SetLineTerminationString

Description

Changes the message boundary for the StringAggregator instance.

Parameters

terminationString

String. The new message boundary.

Comments

By default, a new StringAggregator instance will break strings at every occurrence of the character combination [Carriage Return][Line Break] (“\r\n”), as this is the most common message termination marker for both RS-232 and IP communications. However, some devices may use other markers, whether it be an XML end document tag, a binary string sequence, or something else. You can invoke this method to set the boundary string as needed.

StringAggregator consumes termination strings as part of its processing. As a result, if you need to preserve the message boundary, you will need to concatenate it to the end of each result which StringAggregator returns to you.

Changing the termination string after your StringAggregator instance has started accumulating message fragments will not break the instance, but it can lead to unexpected return values. The important point to remember is that StringAggregator evaluates and stores message fragments as they are received, so anything processed before your call to this method will be handled using the old termination string, and anything processed after will use the new one.

Examples

```
local aggregator = StringAggregator:new()
aggregator:SetLineTerminationString("</message>")
```

```
local aggregator = StringAggregator:new()
local lineEnding = "\004" -- ASCII 4 (^D)
aggregator:SetLineTerminationString(lineEnding)
```

System Constants

Technically, Lua does not have a facility for storing constant values. To help you avoid having to look up values in reference tables, following tables contain functions which return values for each “constant” of interest.

ActivityTypes

The **ActivityTypes** table contains a function for each available activity type. You can invoke these functions to compare their return types to values in your scripts.

Comments

Remember to include the parentheses in your code; otherwise you will receive a pointer to the function rather than the return value from calling the function.

Examples

```
if myActivity.type == ActivityTypes.WatchTV() then
    -- do something
end
```

DeviceTypes

The **DeviceTypes** table contains a function for each available device type. You can invoke these functions to compare their return types to values in your scripts.

Comments

Remember to include the parentheses in your code; otherwise you will receive a pointer to the function rather than the return value from calling the function.

Examples

```
if myDevice.type == DeviceTypes.Television() then
    -- do something
end
```

CommandTypes

The **CommandTypes** table contains a function for each available command type. You can invoke these functions to compare their return types to values in your scripts.

Comments

Remember to include the parentheses in your code; otherwise you will receive a pointer to the function rather than the return value from calling the function.

Examples

```
if myCommand.type == CommandTypes.PowerOn() then
    -- do something
end
```

PortTypes

The **PortModes** table contains a function for each available port type. You can invoke these functions to compare their return types to values in your scripts.

Comments

Remember to include the parentheses in your code; otherwise you will receive a pointer to the function rather than the return value from calling the function.

Examples

```
if myPort.type == PortTypes.Ip() then
    -- do something
end
```

PortModes

The **PortModes** table contains a function for each available port mode. You can invoke these functions to compare their return types to values in your scripts.

Comments

Remember to include the parentheses in your code; otherwise you will receive a pointer to the function rather than the return value from calling the function.

Examples

```
if myPort.type == PortTypes.Audio() and
   myPort.mode == PortMode.Input() then
    -- do something
end
```