Lua Scripting in Q-Sys

A Guide for Navigating the Wide World of Control Scripting

By Aaron Hood

1st Edition
The Farm AV

Contents

1 Getting Started

Welcome, traveler, to my guide on navigating the world of the Q-Sys control programming platform, powered by Lua. The purpose of this guide is to familiarize you with the concepts and applications of custom solutions via Lua scripting from the ground up. While this document does not have any sections which specifically address some of the fundamentals of coding such as syntax, the structure of this guide should serve as an effective communicator for these concepts, as well as some best-practices for creating a clear, efficient, and effective custom-scripted solution.

Control applications involving simple logical comparisons can be already be done without using any code within the Q-Sys environment by using logic components. While this method of control is viable for solutions of varying complexity, they serve as one of the best examples of how coding can simplify both the look and the flow of control. Using just a few simple lines of code, an adept programmer can take a jungle of various logic components and wires, and turn it into a single sleek component that keeps all of the work hidden under the hood. Additionally, the intuitive nature of coding allows programmers to create solutions whose function and flow can be easily understood, allowing for more efficient troubleshooting when unintended consequences inevitably arise.

While it may seem like a daunting task to learn how to code, it is my intention to show the reader through this guide that everybody has what it takes to get started. Once you have made your first functioning program, you will begin to understand the limitless potential that custom-scripted solutions give you when providing solutions to your customers. By working backwards from the unlimited control provided with scripting solutions, to the limits of control that you will provide the end-user, whether or not a request is possible to accomplish becomes a matter of creativity, and not one of availability.

With that, it's time to begin learning how to create your own custom-scripted solutions! It is assumed that the reader is certified to a minimum of Level One, but it is highly recommended that you are certified to Level Two or above. To get started, open Q-Sys Designer and create a new design. All of the initial lessons can be done using a Control Script component, but as we get into creating full-fledged control solutions, it will be necessary to transition into using the Text Controller component. Once you have the scripting component (Control Script or Text Controller) in your design, you can put the design into Emulation Mode while you work. Whenever you make a change and click "Save Changes", the code will recompile and run, providing you with the ability to work on the code and see its outputs in real-time. Additionally, you can restart the script from the dialogue box in the top-left corner of the scripting console.

1.1 The Print Function

The ability to print to the debug console is arguably the most important tool in any programmer's arsenal. Printing allows you to see important details in your code that you can use to better understand what you're trying to get the program to do. The most ubiquitous first program that every new coder creates is "hello world". As such, it will be the first program we will be creating.

Because Lua is a "flexible" language, there are several different ways that you can print to the debug console. The first is a simple application that prints a single statement:

```
print "Hello, world!"
>>Hello, world!
```

This is the most simple way to print to the debug menu, but it has some shortcomings. The biggest one is that you can only print one statement at a time, which forces you to print every new statement to a new line, which can turn troublesome when you're dealing with a lot of information.

The next method for printing is the preferred method, as it allows you to print multiple statements to a single line, and is more syntactically clear:

```
print("Hello, world!")
>>Hello, world!
```

In the case of multiple statements:

```
print("Hello" , "world!")
>>Hello    world!
```

Because printing to debug converts any values you give it to a string, the later section on strings will amplify the utility of this function by several orders of magnitude.

1.2 Types of Data in Q-Sys

There are many different types of data in the world of computer science. Luckily, for our purposes, there are relatively few that we will have to consider.

1.2.1 Comments and the "nil" Value

Comments are used in programming to provide explanations for what is happening in your code. They come in two flavors: the first is a double-hyphen, which creates a single-line comment. The second is a double-hyphen, double-square bracket, which creates a multi-line comment:

```
-- This is a single-line comment

--[[This is
a multi-line
comment]]
```

The "nil" value is a special value in Lua which is indeterminate. This value will arise in a program when the programmer references something which does not exist.

```
print(NonExistentVariable)
>>nil
```

Nil values are not just indicators of a bug; the ability to determine whether or not something exists is a powerful condition in the creation of logic sequences in a program.

1.2.2 Booleans

Boolean values are considered the most basic types of information in that they are either "true" or "false". Underneath Lua, these values reduce even further to one and zero respectively, but when a boolean-type value is printed to the debug console, it will display as "true" or "false". Additionally, there is an operator which "flips" a boolean value to its opposite value, which comes with some utility that will be seen later on:

```
print(true)
>>true

print(false)
>>false

print(not false)
>>true
```

Boolean values are considered the "bread and butter" of how a program functions at its most basic level. Comparison operators are those which return a boolean value:

```
print(1==1, 1 ~= 1, 1 > 0, 1 >= 0, 1 <= 0, 1 < 0)
>>true    false    true    true    false    false
```

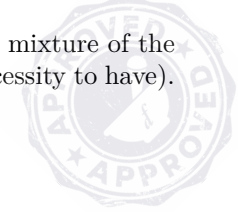
You can additionally use the equivalency operator to see if strings are exact matches of each other:

```
print("cat" == "dog")
>>false
```

1.2.3 Numbers

Numbers come in two distinct flavors in Lua: integers and decimal numbers. The first flavor is a whole number that does not have any trailing decimal points and is printed without a decimal point. The second flavor is a decimal number, which will always have a trailing decimal point when printed, even when it is a whole number.

Lua will always prefer the decimal form of a number when performing calculations that involve a mixture of the two flavors, which is important to consider when an integer-type of number is required (a common necessity to have). The whole gamut of arithmetic operations are available for use between numbers, as is shown below:



```
--Add
print(1 + 1)
>>2

--Subtract
print(1 - 1)
>>0

--Multiply
print(4 * 4)
>>16

--Divide
print(6 / 3)
>>2

--Power
print(2 ^ 2)
>>4

--Modulus
print(4 % 3)
>>1

--Adding a decimal to an integer
print(1 + 1.0)
>>2.0
```

1.2.4 Strings

The final singular type of data you will be dealing with is strings. Strings are arrays of characters that are used for the transmission of information that cannot be easily expressed through the usage of pure numbers or boolean values. Strings can contain readable characters such as numbers and letters, and can also contain non-readable characters, such as hexadecimal codes or formatting characters. Every character has a correlated hexadecimal code that can be viewed by changing the debug output to "Hex", if one is so interested, but this also serves as an additional tool that a programmer can use to view strings that would otherwise be unreadable (as is the case when composing/reviewing hex-commands).

There are several characters (escape sequence characters) that are useful for printing sets of data to the debug console in a more readable form. These characters rely on an "escape", which in Lua is the backslash. Note that some of these escape sequences will seem to perform the same functions as others; as time progresses, their different characteristics will become more apparent, and their uses more specific:

Escape Sequence	Use
\a	Bell
\b	Backspace
\f	Formfeed
\n	New Line
\r	Carriage Return
\t	Tab
\v	Vertical Tab
\\	Backslash
\"	Double Quote
\'	Single Quote
\[	Left Square Bracket
\]	Right Square Bracket

1.3 Tables And Arrays

Tables and arrays are tools that provide a programmer with the ability to organize their data in such a way that they can access it more intuitively, and save themselves an appreciable amount of time in the process of creating a custom-scripted solution. Tables and arrays in Lua must be initialized before they can be accessed.

1.3.1 Arrays

In Lua, the main difference between a table and an array is how the data is organized. In the case of an array, the table's values are organized by integer-type indices, starting at the value 1, which can be initialized and accessed like so:

```
testArray = {"cat", 34}
print(testArray[1], testArray[2])
>>cat    34
```

The advantage of using an array over a table is that the integer-value indices allow for the use of several useful functions. Some of the most common use cases are:

```
--Table concatenation
testArray = {"cat", 34}
print( table.concat(testArray, " ") )
>>cat 34

--Array entry insertion (note that a third argument will
--specify where to insert the entry)
table.insert(testArray, "tree")
print( table.concat(testArray, " ") )
>>cat 34 tree

--Array entry removal (this function also returns the removed entry,
--allowing you to use it after removal)
table.remove(testArray, 2)
print( table.concat(testArray, " ") )
>>cat tree

--Size of array
print( #testArray )
>>2
```

1.3.2 Tables

A table can be created in a similar fashion as the array, with the main difference being that instead of using integers as indices in the table, you create a system of linked "keys" and "values" that are used to access and change the various elements of the table. There are a few different ways that you can edit these tables:

```
-- The simplest method
-- (keys cannot have spaces using this method)
testTable = {cat = "meow", dog = "woof"}
print(testTable.cat, testTable.dog)
>>meow    woof

-- The simple "generative" method
-- (keys cannot have spaces using this method)
testTable = {}
testTable.cat = "meow"
testTable.dog = "woof"
print(testTable.cat, testTable.dog)
>>meow    woof
```

```
-- The "preferred" generative method
-- (keys may have spaces using this method)
testTable = {}
testTable["cat"] = "meow"
testTable["dog"] = "woof"
testTable["blue jay"] = "squawk"
print(testTable["cat"], testTable.dog, testTable["blue jay"])
>>meow    woof    squawk
```

You can construct these tables in a variety of ways, and the method that you use to do so will depend on the specific needs and function of the table. Below is what is considered to be the "best" way to populate a table with predetermined values:

```
testTable = {
  ["cat"] = "meow",
  ["dog"] = "woof",
  ["speckled hen"] = "cluck"
}
```

This method of construction allows the programmer to easily read the entries in the table line-by-line to confirm, without any ambiguity, that they are indeed the intended key-value pairs to be used by the program.

1.3.3 Multidimensional Tables and Arrays

Further convoluting the topic of tables and arrays is the concept of multidimensionality. Not only can you create simple lists using tables and arrays, you can create lists of lists, lists of lists of lists, etc through the use of multidimensional tables and arrays. These allow you to be able to create multiple sets of data that all live under one roof:

```
testTable = {
  ["fruits"] = {
    "apple",
    "orange",
    "banana"
  },
  ["vegetables"] = {
    "carrot",
    "lettuce",
    "fennel"
  }
}
print(testTable.fruits[1], testTable["vegetables"][3])
>>apple    fennel
```

Tables and arrays are some of the most flexible entities within Lua, and their utility is often overlooked when first starting out with custom scripting. When combined with the techniques that follow, however, they can quickly become one of the most powerful tools in your kit.

1.4 Logic

At some point in nearly every custom-scripted solution, your program will have to make some decisions about how to proceed. Every decision that your program will make will be based on the evaluation of some boolean argument. Logic is often considered to be organized into "trees", where each decision is called a "branch". To create a new logic decision, you use "if this then do stuff end" statements:

```
cat = "meow"
dog = "woof"
  if cat == dog then
  print("The dog is a cat")
  else
  print("The dog is not a cat")
  end
>>The dog is not a cat
```

You can also create several branches using "elseif":

```
cat = "meow"
dog = "woof"
dave = "meow"
  if cat == dog then
  print("the cat is a dog")
  elseif dave == cat then
  print("Dave is pretending to be a cat")
  end
>>Dave is pretending to be a cat
```

Logic statements can be imagined as forks in the trail of where the program will go next. When one of the conditions of a logic branch is met, the program will proceed performing the task which ensues, ignoring all of the other logic choices which come after it, even if they are met as well. Because of this, it is important to consider every imaginable case when creating a logic branch to avoid "exclusion" errors such as the one below:

```
num = 1
  if num > 0 then
  print("the number is positive")
  elseif num == 1 then
  print("the number is unity")
  else
  print("the number is neither positive, nor unity")
  end
>>the number is positive
```

Because we put the more general case ahead of the more specific case, the program took the first acceptable path available. Thus, we have effectively "excluded" the other cases from ever being a possible choice. When creating a logic tree, it is advisable to always start with the most specific cases as the first branches. The final "else" statement is usually reserved for either the most general case, or as a catch-all for logic cases which were unforseen at the onset of programming.

1.5 Looping

In order to perform a large amount of processes in a small amount of code, it is extremely common to employ looping. Loops are when a program performs a specific "block" of code several times over until the end-condition of the loop is met, at which point the loop terminates, and the program moves on. To start a loop, you use a "for" statement, followed by something called an "iterating argument", followed by "do", and finalized with "end":

```
num = 0
  for i = 1, 5 do
  num = num + i
  end
print(num)
>>15
```

Loops are especially useful for accessing elements of an array or table:

```
testArray = {"this", "that", "something else"}
  for i = 1, #testArray do
  print(testArray[i])
  end
>>this
>>that
>>something else

testTable = {
  ["january"] = 1,
  ["february"] = 2,
  ["march"] = 3
  }
  for key, value in pairs(testTable) do
  print(key, value)
  end
>>january1
>>march  3
>>february    2
```

Note that in the second case, the "pairs" function does not usually follow the order in which the elements are originally initialized. While small, this factor is an important one in deciding whether a table or an array is the right choice for the application at hand.

1.6 String Manipulation

While seemingly simple in form, the potential functions that strings can perform in a program are numerous. To account for the massive utility that strings provide a programmer, Lua has been equipped with a wealth of string functions that allow you to manipulate them with near-infinite variety. While the list of string functions is long and arduous, it is important to understand them, as strings are one of the fundamental forms of information that is used in many custom-scripted solutions.

1.6.1 Concatenation

This function is the equivalent of taking a piece of glue, applying it to the back of one string, attaching that string to another string, and returning the glued-together result. Using two periods as the "concatenation operator", this is the simplest and most straightforward string manipulation that one can perform:

```
first = "this"
second = "that"
print(first..second)
>>thisthat
```

1.6.2 Numerical Codes

Earlier on when strings were first introduced, a brief mention was made to this topic that every character in a string can be represented by a unique numerical code that corresponds to the character's size in bytes. There are two functions in the Lua library that allow the programmer to go both directions (they are "inverse" with respect to each other) in this process, from character to byte-size, and from byte-size to character:

```
-- getting the size of a character in bytes
print(string.byte("t") )
>>116

-- getting a character from its byte-size
print(string.char(116) )
>>t
```

These functions are particularly useful in the calculation of checksums, which involve performing a combination of arithmetic and logic operations on the numerical codes of certain characters contained within a given string.

1.6.3 String Length

This function returns the length of the string i.e. how many characters are contained within:

```
str = "this"
print(string.len(str) )
>>4
```

1.6.4 Case Changers

There are two functions that you can perform on a string to return its complementary upper-case, or lower-case form:

```
--put string into upper-case
str = "this"
print(string.upper(str) )
>>THIS

--put string into lower-case
str = "THIS"
print(string.lower(str) )
>>this
```

1.6.5 Parsing

There will be many occasions in which the opportunity to parse a string will present itself as a straightforward solution to an otherwise complex problem. There are several ways in which a string can be parsed, with their main differences being what is returned. Sometimes, simply searching for the existence of a given character within the string is sufficient for the task at hand, but other times, more intricate methods of string analysis are needed to get the job done. Most parsing is done using the identification and exploitation of patterns. The following sections cover the concepts and utilities that are involved in parsing data from a string.

1.6.6 Patterns

The ability to identify patterns within a string is arguably one of the most powerful facets of string parsing. The amount of patterns available to identify is truly staggering once the potential of the available parsing functions is realized. In order to best understand how to maximize the parsing potential of our programs, it is a good idea to review the litany of "character classes" that define the various different characters found in strings:

Pattern Character	Use
.	All Characters
%a	All Letters
%c	All Control Characters
%d	All Digits
%l	All Lowercase Letters
%p	All Punctuation Characters
%s	All Space Characters
%u	All Uppercase Letters
%w	All Alphanumeric Characters
%x	All Hexadecimal Digits
%z	Zero
%Y	Where Y is any non-alphanumeric character; '%' is the escape character
[set]	All characters in 'set'; can be specified for a range using the hyphen
[^set]	All characters NOT in 'set'

It must be noted that these patterns only apply to a single character in a string. However, there are ways to further increase the intricacy of a pattern by specifying sequences of characters that fit a pattern:

Sequence Character	Use
+	One or more repetitions (Longest possible sequence)
*	Zero or more repetitions (Longest possible sequence)
-	Zero or more repetitions (Shortest possible sequence)
?	Zero or one occurence in class
%n	Substring equal to n'th captured string
%bxy	String that starts with 'x' and ends with 'y'

The reason for bringing up these lists of patterns is that they are what makes string manipulation so powerful. Using these patterns, it is possible to do most anything you wish with a string using the functions we are going to go over next.

1.6.7 string.format

One convenient way to insert non-table variables into a string is to use the string.format function. The examples below will explain how to insert each data type into a string using string.format:

```
local FunString = "The string is: \"%s\". The float is: %f. The integer is: %d."
print(string.format(FunString, "Hello!", 2.71828, 21) )
>>The string is: "Hello!". The float is: 2.718280. The integer is: 21.
```

1.6.8 string.find

This function searches a string for the specified pattern and returns the indices in the string where the pattern starts and ends. If it doesn't find any occurrences, it returns 'nil':

```
str = "this is a string"
print(string.find(str, "%a+"))
>>1    4

str = "this is a string"
print(string.find(str, "not in str"))
>>nil
```

Note how in the above code, the function finds the start and end of the first word only. This is because a 'space' character is not alphanumeric. If you wish to start your search further in the program, a third argument can be used to bypass the default value of 1:

```
str = "this is a string"
print(string.find(str, "%a+", 5))
>>6    7
```

1.6.9 string.sub

A great complement to the "find" function is the "sub" function. Using this, you can take a pair of scissors, and "cut" a string at a specified index (starting at 1), forming a substring with those indices as the initial and final characters, respectively. If you would like to count backwards from the end of the string, you can use negative indices to indicate how far back you want to go from the final character (i = -1) of the string:

```
str = "this is a string"
print(string.sub(str, 1, -8))
>>this is a
```

1.6.10 string.match

This function searches a string for a specified pattern and returns the substring that it finds (if there are none, it returns 'nil'). Much like string.find, there is a third argument to begin the search at a specific point:

```
str = "this is a string"
print(string.match(str, "%a+"))
>>this

str = "this is a string"
print(string.match(str, "%a+", 5))
>>is

str = "this is a string"
print(string.match(str, "%a+ %a+ %a+ %a+"))
>>this is a string
```

1.6.11 string.gmatch

While similar in looks to the previous function, this function lives in loops. It will iterate through every match of the string, performing the task which you desire to each match:

```
str = "this is a string"
  for s in string.gmatch(str, "%a+") do
  print(s)
  end
>>this
>>is
>>a
>>string
```

1.6.12 tonumber

You can use this function to convert a number in string-form, to a number in number-form in order to use it as such:

```
str = "12"
num = tonumber(str)
print(num, num + 1)
>>12   13
```

There are more functions that you can use to manipulate strings further, as can be seen in the Lua reference, but these are the ones which are most commonly used in the world of Q-Sys Designer.

1.6.13 Another Flavor of Syntax

If you would like to use any of the string manipulation functions on a string-type variable, there is a neater way to call these functions, such that you only need the supplementary arguments to the manipulating functions, with the string-argument being implicitly defined:

```
str = "this is a string"
print(str:match("%a+") )
>>this
```

1.7 Functions

We have already seen several functions throughout this guide, but these have all been functions which are already included in the Lua library. In this section, we will be focused on creating functions of our own. There are two ways to declare a function in Lua:

```
function hithere()
print("hi there")
end
hithere()
>>hi there

hithere = function()
print("hi there")
end
hithere()
>>hi there
```

Many times, we will need to put arguments into our function. This is a very straightforward process in most cases:

```
function add(arg1, arg2)
print(arg1 + arg2)
end
add(1, 4)
>>5
```

It is important to note that you cannot call a function "above" where the function is declared, much like calling a variable "above" where the variable is declared:

```
add(1, 4)
function add(arg1, arg2)
print(arg1 + arg2)
end
>>attempt to call a nil value (global 'add')
```

When creating functions, oftentimes you will want to have the function return something. To do this, you use "return". Sometimes you want to return nothing as a way to stop the function from continuing further, and other times, you may want to return something meaningful that the function has computed. The return argument must come at the end of any logic, loops, or other operations ("chunks") as well.

```
function add(arg1, arg2)
return arg1 + arg2
end
print(add(1, 4) )
>>5
```

1.7.1 os.date and os.time

One important pair of functions that deserve mentioning are os.date and os.time. Using these two functions, one can access past, present, and future dates and times. In §22.1 of the Lua Reference Manual, you will find an excellent synopsis of the use of these tools. For quick reference, the tables of keys for os.time and the table of tags for os.date are listed below:

os.time

Key	Values
year	Full year
month	01 - 12
day	01 - 31
hour	00 - 23
min	00 - 59
sec	00 - 59
isdst	Boolean: true if on daylight savings time

os.date

Tag	Return
%a	Abbreviated Weekday Name (e.g. Mon)
%A	Full Weedkay Name
%b	Abbreviated Month Name (e.g. Jan)
%B	Full Month Name
%c	Date and Time (e.g. 03/15/2021 17:43:08)
%d	Day of the Month
%H	Hour in 24-Hour Format
%I	Hour in 12-Hour Format
%M	Minute
%m	Month
%p	Either "am" or "pm"
%S	Second
%w	Weekday (0-6 = Sunday - Saturday)
%x	Date (e.g. 08/15/2021)
%X	Time (e.g. 01:31:55)
%Y	Full Year
%y	Two-Digit Year
%%	The Character %

2 Coding for Q-Sys

Now that the basics of coding in Lua have been covered, it's time to start putting our new skills to use. This next section will cover some of the most commonly-used extensions that Q-Sys has integrated into their Lua coding environment. While it is by no means comprehensive or exhaustive, it should serve to prepare up-and-coming programmers for the fundamentals of Q-Sys extensions in Lua such that they will be able to learn the other extensions pain-free.

From hereon out, all of the concepts that were discussed above are going to be used heavily, and I will be operating under the assumption that you have a salient understanding of them, such that none of the code you come across further down will be something that you have not seen. If you come across a concept or a piece of code that you cannot follow, revisit its related section for any clarification, and if that doesn't help, there are much more detailed guides readily available for review on the internet.

2.1 Controls IO

Accessing controls is easily the most common application of Lua scripting in Q-Sys. Whether it's reading inputs or setting outputs, nearly every program that you make will have some sort of controls attached to it. Controls for scripting components i.e. the Control Script or Text Controller, exist in a "Controls" table. To make the process of accessing these controls simpler, it is often a good idea to assign your controls to variables:

```
--Control Script
led = Controls.Inputs[1]
status = Controls.Outputs[1]

--Text Controller with custom names
led = Controls.LED
status = Controls["Status Output"]
```

The story does not end there however; we may have assigned a variable to a control, but we have not yet accessed the value of the control. It is smart to think of a control as a table of functions which return various properties of the control (something which becomes useful later on down the line). When we call a function that is attached to the control, the function then returns the value associated with the control. As such, it's important not to assign variables with the value of a control with the belief that the variable will change its value when the control changes value. The value of a control should only be accessed when and where it is needed, otherwise you may run into problems where you are using control values that are outdated (unless that is your intention). The next portion of this section is dedicated entirely to accessing control values of the various types that Q-Sys makes available to us, as well as the types of user-interface objects that these controls are usually attached to.

2.1.1 EventHandler

The event handler of a control calls a function whenever the control's value is changed. To assign a new even-handling function to a control, you assign it like any other function:

```
Controls.Inputs[1].EventHandler = function()
  print("My value was changed!")
end
```

2.1.2 Boolean

Booleans are controls that return a boolean-type value when called. They are accessed like so:

```
Controls.Inputs[1].EventHandler = function()
  print(Controls.Inputs[1].Boolean)
end
>>false
```

Because they lend themselves so well to logic trees, it is worth mentioning at this point that within your "if" statements, you do not necessarily need to include an equivalency operator comparing the boolean control to "true" or "false", because "if" only looks for a boolean type as the condition:

```
--Checking for "true" statement
Controls.Inputs[1].EventHandler = function()
  if Controls.Inputs[1].Boolean then
  print("the button is on")
  end
end

--Checking for "false" statement
Controls.Inputs[1].EventHandler = function()
  if not Controls.Inputs[1].Boolean then
  print("the button is off")
  end
end
```

One final note: because buttons which are not triggers have individual "on" and "off" states, it is important to include a logic tree in every event handler you create for them. Otherwise, the button does the same event handling function doubly.

2.1.3 Value

The "Value" of a control is one which can be an integer, a decimal number, an audio level, a frequency, or a time. For the next example, a text controller was created, and a fader with the custom name of "fader" was added to the control list. By analogy, we can extend what was discussed with booleans to values:

```
--Print values from a fader to the debug console
Controls.fader.EventHandler = function()
print(Controls.fader.Value)
end
```

2.1.4 String

Similarly, you can access the string of a control in the same way, using a text controller and a custom-named string control:

```
--Print values from a string to the debug console
Controls.str_input.EventHandler = function()
print(Controls.str_input.String)
end
```

It is important to mention that when a string-type control is changed to the identical string as was in there before, the event handler will not be called.

2.1.5 Status

This type of control is very similar to a string-type control, but it is actually a combination of an integer-type, and a string-type control. Each status type has a correlated integer code that can be used to change its overall status. The string value can also be used to adjust the status; if you place the status you would like displayed at the beginning of the string i.e. "OK Connected", the status will appear as "OK" and the message following will be "-Connected". Either method works perfectly well, but in some situations, you may prefer one method over the other.

Below is a table containing the statuses and their correlated integer codes. Note that going any higher than what is on the table results in a "Fault" code.

Number	Status
0	OK
1	Compromised
2	Fault
3	Not Present
4	Missing
5	Initializing

Below is an example of providing a status of "Missing" with a message to the status control named "stat":

```
Controls.stat.Value = 3
Controls.stat.String = "I can't Find it!"
```

2.1.6 Values

Another type of information that you may come across is that which is transmitted in the form of an array. This is uncommon, but is worth mentioning in case you need to create a custom-scripted solution which references an RTA output pin or something of the sort:

```
--Print the 5th band from an RTA
print(Controls.RTA.Values[5])
```

2.1.7 Color

This is a property that is available for every control in Designer. There are several colors available to use through their literal string-name, but others can be created from an RGB string, where the values (ranging from 0-255) are encoded in hexadecimal format. I often find myself draggint the button into the schematic page and manually changing the color of the button in the Properties window to the one that I want to use, then coping the color string and pasting it into my code.

```
--Using literal color names
Controls.Button.Color = "blue"

--Using hex color names
Controls.Button.Color = "#0000FF"
```

2.1.8 Control Count

When using a Text Controller, there is the option to increase the count of controls of a given label. When you have more than one control of a given label, their correlated control functions are placed in an array:

```
--Print the 3rd string from a multi-count control:
print(Controls.text_inputs[3].String)
```

2.2 Timers

Timers are incredibly useful tools for sequencing events temporally. If you want to have a touch panel go to a "dim" state after a certain period of time, or would like to have a text-field be erased after several seconds, then timers are going to be the tool for the job.

2.2.1 The Timer Entity

To use a timer in its fullest capacity, you will have to initialize the entity itself. Once this is done, you now have a timer that behaves much like any other control within Q-Sys in that it has an associated event handler that you can program to do whatever it is that you wish. The following example contains the full syntax of the timer entity; note that the argument passed to "Timer:Start(x)" is the time-step of the entity in seconds:

```
count = 0
counter = Timer.New()
counter.EventHandler = function()
  if count < 3 then
  count = count + 1
  print(count)
  else
  counter:Stop()
  count = 0
  print("Finished Counting.")
  end
end
counter:Start(1)
>>Starting Script
(1 second later)
>>1
(1 second later)
>>2
(1 second later)
>>3
(1 second later)
>>Finished Counting.
```

2.2.2 Timer.CallAfter

Sometimes, you only need a timer to perform a simple task once after a certain period of time. In this case, it is often simpler to employ the use of the "Timer.CallAfter" function:

```
Timer.CallAfter(function()
print("This was delayed by 1.5 seconds")
end, 1.5)
>>Starting Script
(1.5 seconds later)
>>This was delayed by 1.5 seconds
```

2.2.3 Temporal Sequences

If you need to have a series of events happen in a time-ordered fashion, you can use either the timer entity, or Timer.CallAfter, depending on which one makes more sense to you. If you are more comfortable with logic trees, then the entity is a great choice. If you are more comfortable with nested functions, then Timer.CallAfter is the choice for you. Below are examples of both methods:

```
--The Timer Entity:
seq = 0
s_timer = Timer.New()

s_timer.EventHandler = function()
  if seq == 0 then
  first_func()
  elseif seq == 1 then
  second_func()
  elseif seq == 2 then
  third_func()
  else
  s_timer:Stop()
  seq = 0
  return
  end
```

```
seq = seq + 1
end

s_timer:Start(1)

--Nested Timer.CallAfter functions:

Timer.CallAfter(function()
first_func()
  Timer.CallAfter(function()
  second_func()
    Timer.CallAfter(function()
    third_func()
    end, 1)
  end, 1)
end, 1)
```

There are benefits and drawbacks to both methods of sequential timing; the more that you use timers, the more you will understand which method best-suits both your programming style, and the application you're programming.

2.3 Components

When you're creating a solution that involves various controls from a single component, it can start to get a little confusing to deal with the many wires or control tags that can be involved. This can be compounded by the fact that many designs also implement extensive use of control tags and wires to route audio signal, and to another person looking at your design, it can quickly become a daunting task to keep track of all of the tags and wires in a design. The best way to maintain forward-facing simplicity is to reference component controls directly in your script. For the entirety of this section, we will be referencing an 8 input by 8 output Matrix Mixer component with the Label Controls property enabled.

2.3.1 Initializing a New Component

In order to access the controls of a component in your design, you must give the component a unique name. Once this is done, you can initialize the component within the script by referencing that custom name:

```
mixer = Component.New("Near-End Mixer")
```

2.3.2 Viewing Accessible Controls

The way that component controls are labeled for access is not as simple as is the case with multi-count controls, where they are arranged in arrays. This is mainly because arrays and multidimensional arrays are not easily converted to a human-readable format. To view the controls available for access by your script, click "Tools", then click "View Component Control Info". At this point, I like to move the Control Info window into one of the side-banks to be able to keep it out of the way. Finally, click on the component that you would like to view the controls of. The list that is returned in the case of our mixer can seem quite daunting at first glance, but take some time to notice the patterns between all of the controls. These patterns were created with the intention of providing programmers with the capability to more easily create new control tables within their own script.

2.3.3 Accessing Controls

Mercifully, accessing the controls of a custom-named component is identical to how it is done internally, with the biggest difference being that you will always be referencing a key from the table of controls:

```
mixer = Component.New("Near-End Mixer")
print(mixer["input.1.gain"].Value )
>>0.0
```

2.3.4 Organizing Controls

As is probably apparent, accessing controls in this fashion can become quite tedious. Luckily, with our knowledge of looping and string manipulation, we can construct control tables that allow us to reference these controls in a more streamlined fashion (HINT: copy+paste is your best companion in this quest):

```
mixerComp = Component.New("Near-End Mixer")
mixer = {
  inputs = {},
  outputs = {}
}
  for i = 1, 8 do
  mixer.inputs[i] = {
    gain   = mixerComp["input." .. i .. ".gain"],
    invert = mixerComp["input." .. i .. ".invert"],
    label  = mixerComp["input." .. i .. ".label"],
    mute   = mixerComp["input." .. i .. ".mute"],
    solo   = mixerComp["input." .. i .. ".solo"],
    trim   = mixerComp["input." .. i .. ".trim"]
  }
  mixer.outputs[i] = {
    gain    = mixerComp["output." .. i .. ".gain"],
    invert  = mixerComp["output." .. i .. ".invert"],
    label   = mixerComp["output." .. i .. ".label"],
    mute    = mixerComp["output." .. i .. ".mute"],
    prepost = mixerComp["output." .. i .. ".pre.post"]
  }
  end
```

Now, we can access the controls that we want in a way that is much more intuitive to the style of programming that Lua lends itself to. Instead of having to write out those horrendous keys every time you wish to reference an input or output control, you can now reference your control table. For instance, if you wanted to change the status of the mute button of any input to "true" every time that its fader goes below -40dB, you can do it like this:

```
for i = 1, #mixer.inputs do
  mixer.inputs[i].gain.EventHandler = function()
    if mixer.inputs[i].gain.Value < -40 then
    mixer.inputs[i].mute.Boolean = true
    end
  end
end
```

Hopefully the massive utility of multidimensional tables and arrays is now apparent, as you can create control structures that involve various components and internal controls, all under a common roof that is easier to reference than the controls themselves. Further, it allows you to easily iterate through arrays in order to make bulk adjustments.

2.4 User Interface

In the pursuit of an optimal user experience (UX), it can be to your benefit to familiarize yourself with the various functions that you can call that can help you maximize the efficiency and ease-of-use of your interfaces.

2.4.1 UCI Functions

These are functions that affect a UCI such that the changes can be viewed from directly within Q-Sys Designer.

Uci.SetLayerVisibility

This function makes a layer of a UCI visible, or invisible, depending on the boolean argument passed to it. The function takes in a total of four arguments: First, the name of the UCI, followed by the name of the page within the UCI that this layer lives in, then the name of the layer itself. Next, the boolean value of whether or not the layer is visible, and finally, the transition type. There are six transitions you can choose from: "none", "fade", "left", "right", "bottom", and "top". Using the code is as simple as this:

```
vis_button = Controls.MakeVisible
vis_button.EventHandler = function()
  Uci.SetLayerVisibility(
  "User UCI",
  "Audio Page",
  "Fader Layer",
  vis_button.Boolean,
  "none"
  )
end
```

Uci.SetSharedLayerVisibility

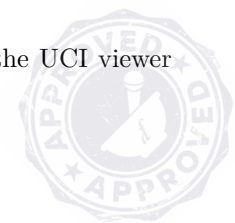
Sometimes, you would like to change the visibility of a shared layer such that many pages of a UCI are affected simultaneously. The arguments to this function are identical to the ones above, except the page-name is not required (as shared layers transcend pages). This function can be useful for UCI's that will be used for several different applications:

```
mode_button = Controls.MakeVisible
mode_button.EventHandler = function()
Uci.SetLayerVisibility(
  "User UCI",
  "Concert Layer",
  mode_button.Boolean,
  "none"
)
Uci.SetLayerVisibility(
  "User UCI",
  "Tradeshow Layer",
  not mode_button.Boolean,
  "none"
)
end
```

Note how the second call of the function used "not" to invert the value of the button. In effect, we have created a button which makes the UCI alternate between the two layers.

2.4.2 Touch Panel And UCI Viewer Functions

The functions in this section are only able to manifest themselves either inside of a UCI, or in the UCI viewer program.



Uci.ShowDialog

If you want to give users access to more control, but don't want to overpopulate a page with buttons, this function allows you to do so by interrupting the UX with a pop-up dialog box populated with buttons that the user can press to perform the desired control. A good way to set up this control would be to create the handling function to be called inside of the dialog first:

```
function switchInput(choiceInt)
print("Input "..choiceInt.." was picked")
send_input_cmd(choiceInt)
end

function ShowInputDialog()
  Uci.ShowDialog(
    "User UCI",
    {
      Title = "Input Selection",
      Message = "Select your desired input"
      Buttons = {
        "HDMI 1",
        "HDMI 2",
        "DisplayPort"
      },
      Handler = SwitchInput
    }
  )
end

Controls.InputSelect.EventHandler = ShowInputDialog
```

Uci.SetScreen

This function is helpful for placing screens in a "standard" screen state at the beginning or end of the day, whether that's fully on, off, or just dim. This can be done using the os.date and os.time functions (covered in "Scheduling Controls"), but for simplicity's sake, this example will only cover the syntax of the function:

```
Uci.SetScreen("User Touch Panel", "Dim")
```

Uci.SetUCI

This function is useful if and only if a touch screen has the **Dynamic UCI Assignment** property enabled. Much like the function above, this is useful for returning a touch panel to its "standard" UCI:

```
Uci.SetUCI("User Touch Panel", "User Login UCI")
```

Uci.SetPage

If you are only using a single UCI in your touch panel or UCI viewer, then this function will give you the ability to change which page the UCI that you're using is on:

```
Uci.SetPage("User Touch Panel", "Home Page")
```

Uci.LogOff

If you've configured your touch panel or UCI Viewer to require a PIN to be able to use from the Core Manager (and this is the only way that this function can be used), then you'll probably want an automated way to log the device out from being able to access the controls. This is the function that will provide you with that capability:

```
Uci.LogOff("User Touch Panel")
```

2.5 Serial Ports

Serial ports are one of the two primary media through which controls are transferred between a 3rd party device and a Q-Sys device such as a Core. Whenever a serial connection is made, **it is important to check the manufacturer's serial control Application Programming Interface (API) manual to see what baud rate, data bits, and parity the device uses.** If they are different from what is considered to be the industry standard (9800 baud, 8 data bits, and no parity), then you must specify some extra optional arguments beyond the baud rate when connecting to a serial port. The QSC help file has a great description of the methods and events available to programmers to use with their serially controlled devices.

2.6 TcpSocket and TcpSocketServer

TCP Sockets are the second way to control a 3rd party device using the Q-Sys platform. These are my preferred type of connection, as they allow for the connection of multiple devices in the most straightforward manner.

The TCP Socket Server is very similar to the TCP Socket, with its purpose being the receipt of inbound control messages from other devices on a certain allowed port that Q-Sys is "listening" on. Once again, the QSC help file contains all of the needed details about these objects, and I will refer the reader to that section for more information.

2.7 JSON

This library is useful whenever you need to control a device that uses a REST API, or some other JSON-based control method. The ability to encode Lua tables into JSON and the other way around allows for the composition and interpretation of JSON strings that would otherwise require a great deal of string manipulation. To initialize JSON, just require the extension within your script:

```
json = require("json")
```

Then, encoding a table of any sort into JSON is as easy as:

```
luaTable = { array = {"a", "b", "c"}, table = {a = 1, b = 2, c = 3} }
jsonTable = json.encode(luaTable)
print(jsonTable)
>>{"table":{"c":3,"b":2,"a":1},"array":["a","b","c"]}
```

2.8 HttpClient

The HTTP client is a great way to control a device or use services that employ the use of web hooks. When you are "sending" a command to an HTTP server, you will be employing the "Upload" function, using either the POST or PUT method, depending on what their API specifies. When you are trying to gather data or telemetry through web hooks, you will want to use the GET method, which is automatically specified by the "Download" function.

For basic or digest-authenticated sites, there are fields in which you can include a username and password, but considering that this is very sensitive information, it is a very good idea to place the scripting component into a container that you lock with a PIN stored on a secure database that only trusted individuals know, because it's more than likely stored in the program in plain-text.

The functions take in a table as their input, and when they receive a response, their corresponding event handler is a function of several arguments pertaining to the response they get from the server:

```
--HTTP UPLOAD EXAMPLE:
URL = https://www.thispersondoesnotexist.com/

function handleResponse(tbl, code, data, err)
  print("Data Received")
  print("\tResponse Code: ".. code)
  print("\tErrors: ".. err or "None")
  print("\tData: "..data)
end

HttpClient.Upload {
  Url = URL,
  Method = "POST",
  User = "timmyjimmy66",
  Password = "supersecretpassword",
  Data = "Hi Timmy, hope you had a great vacation! -Jimmy",
  EventHandler = handleResponse
}
--The following response is purely hypothetical:
>>Data Received
>>Response Code: 200
>>Errors: None
>>Data: This is a response confirming that your message to
  Timmy O'Jimmy's personal HTTP-based messaging server has been received.
  To see if Timmy has responded to your message, check back periodically
  using the GET method. Have a good day!

--HTTP DOWNLOAD EXAMPLE
URL = https://www.thispersondoesnotexist.com/

function handleDownload(tbl, code, data, err, headers)
  print("Download Complete")
  print("Response Code: ".. code)
  print("Errors: ".. err or "None")
  print("Headers:")
    for headerName, headerContent in pairs(headers) do
    print("\t"..headerName.." = "..headerContent)
    end
  print("Data: "..data)
end

HttpClient.Download {
  Url = URL,
  Headers = {["Content-Type"] = "text/html"},
  Timeout = 30,
  EventHandler = handleDownload
}
--The following response is purely hypothetical:
>>Download Complete
>>Response Code: 200
>>Errors: None
>>Headers:
>>  Content-Type = text/html
>>Data: Hi Mr. O'Timmy, hope your mother is doing well.
  My vacation was great; the family really enjoyed the
  beach, and the kids brought home plenty of silly magnets
  to put on the fridge. - Timmy
```

3 Building a Text Controller

Building a text controller is one of the most common applications of Lua programming in Q-Sys. This process is completely open to the creativity of the programmer, and more complex applications start to become as much a work of art as a pragmatic solution. Whether your solution is simple or complex, the following guidelines will be helpful in ensuring that you minimize the amount of time spent programming, while maximizing the results of your program.

3.1 The Plan of Attack

When developing a new program, it is often tempting to begin the process by adding the required controls and typing code. Sometimes, this quick and dirty tactic is sufficient to perform the task at hand. As a young, budding programmer, I also followed this route as it allowed me to complete many projects quickly. However, as the projects became more complex, and the stakes got higher, I started running into more problems with my programs. Because I didn't plan out the development process, if anything went awry, not only did I have to fight with issues with the program, I also had to overcome obstacles of my own design. After a while of this self-destructive methodology, I realized that by adding intention to the program, I was able to not only minimize unforseen bugs, but also clear them up with ease. Because of this, I am confident in my assertion that even smaller programs deserve the same type of careful consideration as the more complex ones, if only to hone the skill of planning out the development process. Before a single keystroke of Lua code is written, it's important to ask yourself some questions:

1: What is the desired functionality of my program?

Make sure that you have a complete and detailed list of the desired functions that the program is going to be responsible for. Use this as a "checklist" to make sure that there isn't a single parameter or requirement that is overlooked; adding functionality to a completed program is much harder than including it in the first place, often requiring workarounds or fundamental changes to the primary structure of the code.

2: Does this program require any additional components?

Once you have the list of functions, it's important to consider all physical devices and associated processing components that will be involved in the program. By finding out which devices are involved, you can begin gathering information on how to control these devices, oftentimes vis-a-vis a 3rd-party control API. By considering both the existing components in the design, and other components that will be needed to make the program work, you can make sure that you have every piece of the machine that you need to make the program do what you want.

3: Are there any additional features I can add that enhance functionality?

While extra features come with extra time and an increased risk of bugs, being able to provide a program that exceeds expectations and delivers functionality that improves the user-experience will make users and clients much happier with their choice to employ your skills. Going that "extra mile" often doesn't significantly increase your workload, but it does give you the satisfaction of a job well-done, and instills a sense of fulfillment that you provided somebody with an amazing program. Discretion is important with this one; will this extra feature be a "wow" experience for the end-user, or just another button on a screen of buttons? Make sure that it is the former, as the latter tends to frustrate simplicity-oriented users.

4: What are the biggest development challenges?

By breaking down the desired functionality into its base-components, you can prepare yourself for what it is going to take to make the program work. Identifying each different aspect of every function will allow you to make more salient decisions about the structure of your code, and also figure out any interdependencies that need to be adressed. Understanding which problems are going to be the hardest to solve will allow you to focus your early efforts on overcoming them in a "vacuum" without the overhead of code you spent hours writing that you now have to work around. For complex programs that have many hurdles to overcome, a good idea would be to rank parts of the program by difficulty of implementation. Then, work your way from hardest to easiest, so long as it makes sense to do so.

5: Do I have any existing programs that I can use as a starting point?

Many times, programs have analogous (or even identical) functions to ones that you have created in the past. Why go through the effort of reinventing the wheel when you can just copy some code from another program, paste it into the new one, and fit it within its new context? While a great deal of time can be saved by stealing from yourself and others this way, it must be done with caution; sometimes the new context has some aspect that renders the old code completely obsolete, and spending time trying to adapt it to its new environment will outweigh the cost of simply developing it from scratch.

6: Should this program be static or dynamic in structure?

The relevancy of this question is highly dependent on the type of program that is being made. Put more simply, should you make the program able to have the number of certain controls changed? If you're developing a third-party control application, then it comes down to what kind of hardware you're going to be controlling. If you're making a program for hardware that has different models with different numbers of the same types of controls, such as a video switcher, then it would be beneficial to allow your program to be "scaled" to control all of those models. If you're making a program that has universal applicability to many different scenarios, then scalability will also be an integral part of what makes it as universal as possible. The inclusion of scalability in a program is usually a trivial application of looping, but care must be taken with more complex programs; scaling is something that should be considered at the onset to minimize the risk of missing scaling functionality in specific aspects of your program.

3.2 Control Structures

While verbose structures like these may seem excessive for simpler programs, it's important to apply them often in order to maintain a consistent programming standard. A good way to keep your structure intuitive and organized is by storing related things in a multidimensional table. The following example will go through setting up eight buttons that turn on a correlated LED when on, as well as placing some text in a correlated text field. When that button is turned back off, the LED turns off, and new text is created, then deleted after a 2-second delay. We are able to do all of this from inside of one multidimensional table.

```
--Button/LED/Text Example:
cTable = {
  Button = {},
  Led = {},
  Text = {}
}

for i = 1, #Controls.Button do
  cTable.Button[i] = Controls.Button[i]
  cTable.Led[i] = Controls.LED[i]
  cTable.Text[i] = {
      ctl = Controls.Text[i],
      clear = function()
        Timer.CallAfter(function()
        cTable.Text[i].ctl.String = ""
        end, 2)
      end
    }

  cTable.Button[i].EventHandler = function()
  cTable.Led[i].Boolean = cTable.Button[i].Boolean
    if cTable.Button[i].Boolean then
    cTable.Text[i].ctl.String = "ON"
    else
    cTable.Text[i].ctl.String = "POWERING OFF"
    cTable.Text[i].clear()
    end
  end
end
```

4 Advanced Concepts

Now that we've covered some of the most important aspects of Q-Sys programming, it's time to take a deep dive into some examples and applications that not only showcase important concepts, but also provide a good deal of utility in the real world.

4.1 Checksum Algorithms & Command Composition

In most cases, building 3rd-party control applications with small amounts of controls is a trivial process. However, as the control count grows, so too does the necessity for your program to compose these commands "on the fly". If an API uses hexadecimal commands, this need is compounded; more often than not, these API's require the character-length of the command at the beginning, and a checksum at the end. Further confusing the matter, manufacturers do not follow any standard checksum algorithm, so it's up to you, the programmer, to make sure that you fully understand what it is that the device expects to receive. Checksum algorithms are undoubtedly the hardest part of any comprehensive 3rd party application, taking the most time to troubleshoot and ensure proper functionality. But creating your own checksum algorithms can save an immense amount of time and programmatic complexity, as well as provide your program with the utmost flexibility to add more commands and modify existing ones. The following example will go through a simple checksum algorithm, as well as composing a command to be sent to a 3rd-party device:

```
--I like to make a global variable for the EOC
eoc = "\r"

--Checksum Algorithm
function checksum(cmd)
local sum = 0
  for c in cmd:gmatch("%x") do
    sum = sum + string.byte(c)
  end
return sum
end

--Command Composition
function compose(cmd)
  local cLength = cmd:len()
  local cSum = checksum(cmd)
  return cLength .. cmd .. cSum .. eoc
end
```

4.2 A Comprehensive Look at 3rd-Party Feedback

Part of building a comprehensive 3rd-party application is integrating feedback from the device. Luckily, the engineers at most manufacturers follow an intuitive pseudo-standard for providing this feedback in a way that is analogous to the commands that their device receives. By building our 3rd-party control plugins in a fashion that is conducive to this type of operation, we can simplify the structure of our code significantly. On the following page is an incomplete example of structuring commands, controls, and handlers in such a way.

By handling the incoming command vis-a-vis referencing functions stored within the table, we were able to simplify the data pipeline. Further, this method of handling incoming data allows for a more compartmentalized code-structure, which can make troubleshooting and debugging a simpler task[1].

Taking some time to refine a complete generalized 3rd-party control structure that makes sense to you will leave you with a good template to start with when beginning a new 3rd-party control program.

[1]This example can be expanded upon by attaching event handlers and outgoing command functions to the control table.

```
--Control Table Structure Example
cTable = {
  pwr = {
    ctl = Controls.Power,
    Type = "Boolean",
    cmd = "\x01",
    par = {
      off = "\x00",
      on = "\x01"
    },
    handle = function(par)
      if par == "\x00" then
      cTable.ctl[Type] = true
      else
      cTable.ctl[Type] = false
      end
    end
  }

  vol = {
    ctl = Controls.Volume,
    Type = "Value",
    cmd = "\x02",
    par = vol.ctl[Type],
    handle = function(par)
    vol.ctl[Type] = string.byte(par)
    end
  }
}

address = Controls["IP Address"]
port = Controls.Port
status = Controls.Status

socket = TcpSocket.New()

socket.Data = function(socket)
local data = socket:Read(socket.BufferLength)
local inCmd = data:sub(2,2) -- Assuming the command-name is the second character
local inPar = data:sub(3,3) -- Assuming the parameter is the third character
  for k, tbl in pairs(cTable) do
    if inCmd == tbl.cmd then
    tbl.handle(inPar)
    return
    end
  end
print("Warning: "..data.." is not in the command table (cTable)")
end

function connect()
  socket:Connect(address.String, port.Value)
end

address.EventHandler = connect
port.EventHandler = connect

connect()
```

4.3 Confirm Button: A Function of Variable Arguments

This is an example of a function that can take in a variable number of arguments that is useful when you have controls that make changes that are big enough to warrant asking the user if they would really like to enact them. Note that the button's event handler must be cleared when the user changes their mind about performing the action.

```
confirm = function(...)
Controls.Confirm.Color = "yellow"
local arg = {...}
local func = table.remove(arg , 1)
  for i  = 1, #arg do
    if arg[i] == nil then
      table.remove(arg, i)
    end
  end
  Controls.Confirm.EventHandler = function()
    Controls.Confirm.Color = "white"
    func(table.unpack(arg))
    Controls.Confirm.EventHandler = nil
  end
end

cancel = function()
Controls.Confirm.Color = "white"
Controls.Confirm.EventHandler = nil
end

--Example of Application
function otherprint(string1, string2, string3)
  print(string1..string2..string3)
end

confirm(otherprint, "hi ", "there, ", "friend!")
```

5 About The Farm AV and FarmAssist

The Farm's mission is to promote and support the brands we represent. In the dynamic marketplace of the West Coast, this mission requires an ever expanding collection of abilities. The Farm provides these services from the beginning to the end of the audiovisual project lifecycle. From design and standards-setting, all the way through deployment and commissioning. From marketing to training and ultimately the sale, The Farm serves our partners and their customers with focus and dedication through every step of the process.

We built FarmAssist as a support organization for the West Coast commercial AV channel. We are a team of highly technical design, programming and commissioning professionals working with our AV partners to ensure their deployments function as promised. Whether a client is overwhelmed with projects, timelines or complexities, our promise is to be your outsourced right hand.

At FarmAssist, we specialize in work built around our line card platforms, but we also understand that there are multiple components in each system to be automated, programmed, and operational within a technology ecosystem. For that reason, our team has a diverse education in many industry platforms.

A Message from the Author

Thank you for reading my guide. It was created with the intention of giving people the knowledge that they need to create programs with the same amount of intention and functionality as the ones we provide. By giving users this guide, it is our hope that we can show them just how intuitive and powerful the language is at accomplishing what you need.

If there are any issues with the text, errors with the examples, or sections that you would like to see added to later editions, please leave a note at our FarmAssist help desk. Navigate to `https://www.thefarmav.com/farmassist/` and click on "Take me to the FarmAssist Help Desk!" Then, at the bottom of the page, click "Submit a ticket" to create a support ticket. In the subject, make sure to include "Lua Guide" so that I can easily find it and get working on the change. If that seems like too much of a hassle, you can feel free to e-mail me directly at `aaron@thefarmav.com` with "Lua Guide" in the subject field.

- *Aaron Hood*