Scripting Guide

IndigoRose
SOFTWARE DESIGN CORP.
www.indigorose.com

Proprietary Notice
The software described in this document is a proprietary product of Indigo Rose Software Design Corporation and is furnished to the user under a license for use as specified in the license agreement.
The software may be used or copied only in accordance with the terms of the agreement.

Information in this document is subject to change without notice and does not represent a commitment on the part of Indigo Rose Software Design Corporation. No part of this document may be reproduced, transmitted, transcribed, stored in any retrieval system, or translated into any language without the express written permission of Indigo Rose Software Design Corporation.

Trademarks
AutoPlay Media Studio and the Indigo Rose logo are trademarks of Indigo Rose Software Design Corporation. All other trademarks and registered trademarks mentioned in this document are the property of their respective owners.

Copyright
Copyright © 2003 Indigo Rose Software Design Corporation.
All Rights Reserved.

LUA is copyright © 2003 Tecgraf, PUC-Rio.

Table of Contents

INTRODUCTION 7

A QUICK EXAMPLE OF SCRIPTING IN AUTOPLAY MEDIA STUDIO 7

IMPORTANT SCRIPTING CONCEPTS 9

SCRIPT IS GLOBAL 9

SCRIPT IS CASE-SENSITIVE 9

COMMENTS 10

DELIMITING STATEMENTS 11

VARIABLES 11

WHAT ARE VARIABLES? 11

VARIABLE SCOPE 12

Local Variables 13

VARIABLE NAMING 14

RESERVED KEYWORDS 15

TYPES AND VALUES 15

Number 16

String 16

Nil 19

Boolean 19

Function 20

Table 20

Variable Assignment 22

EXPRESSIONS AND OPERATORS 22

ARITHMETIC OPERATORS 23

RELATIONAL OPERATORS 23

Logical Operators 24
Concatenation 25
Operator Precedence 25

Control Structures 26
If 26
While 27
Repeat 28
For 29

Tables (Arrays) 30
Creating Tables 31
Accessing Table Elements 31
Numeric Arrays 32
Associative Arrays 32
Using For to Enumerate Tables 34
Copying Tables 35
Table Functions 37

Functions 38
Function Arguments 39
Returning Values 40
Returning Multiple Values 41
Redefining Functions 41
Putting Functions in Tables 42

String Manipulation 42
Concatenating Strings 43
Comparing Strings 43

Counting Characters 44
Finding Strings 45
Replacing Strings 45
Extracting Strings 46
Converting Numeric Strings into Numbers 47

OTHER BUILT-IN FUNCTIONS 49
Script Functions 49
dofile 49
require 49
type 50
Actions 51

DEBUGGING YOUR SCRIPTS 51
Error Handling 52
Syntax Errors 52
Functional Errors 53
Debug Actions 54
Application.GetLastError 54
Debug.ShowWindow 57
Debug.Print 57
Debug.SetTraceMode 58
Debug.GetEventContext 60
Dialog.Message 60

FINAL THOUGHTS 60
Other Resources 60
Help File 60
User's Guide 61

AutoPlay Media Studio Web Site *61*
Indigo Rose Technical Support *61*
The Lua Web Site *61*

Introduction

One of the powerful new features of AutoPlay Media Studio 5.0 is its scripting engine. This document will introduce you to the new scripting environment and language.

AutoPlay scripting is very simple, with only a handful of concepts to learn. Here is what it looks like:

```
a = 5;
if a < 10 then
    Dialog.Message("Guess what?", "a is less than 10");
end
```

(Note: this script is only a demonstration. Don't worry if you don't understand it yet.)

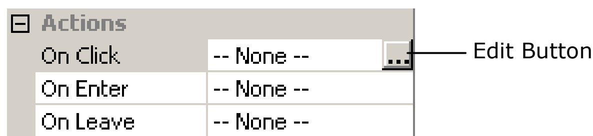
The example above assigns a value to a variable, tests the contents of that variable, and if the value turns out to be less than 10, uses an AutoPlay action called "Dialog.Message" to display a message to the user.

New programmers and experienced coders alike will find that AutoPlay Media Studio is a powerful, flexible yet simple scripting environment to work in.

A Quick Example of Scripting in AutoPlay Media Studio

Here is a short tutorial showing you how to enter a script into AutoPlay Media Studio and preview the results:

1. Start a new project.
2. Create a new button object.
3. In the Actions category of the properties inspector, click on the On Click event. A small edit button should appear next to the word "-- None --" on the right.

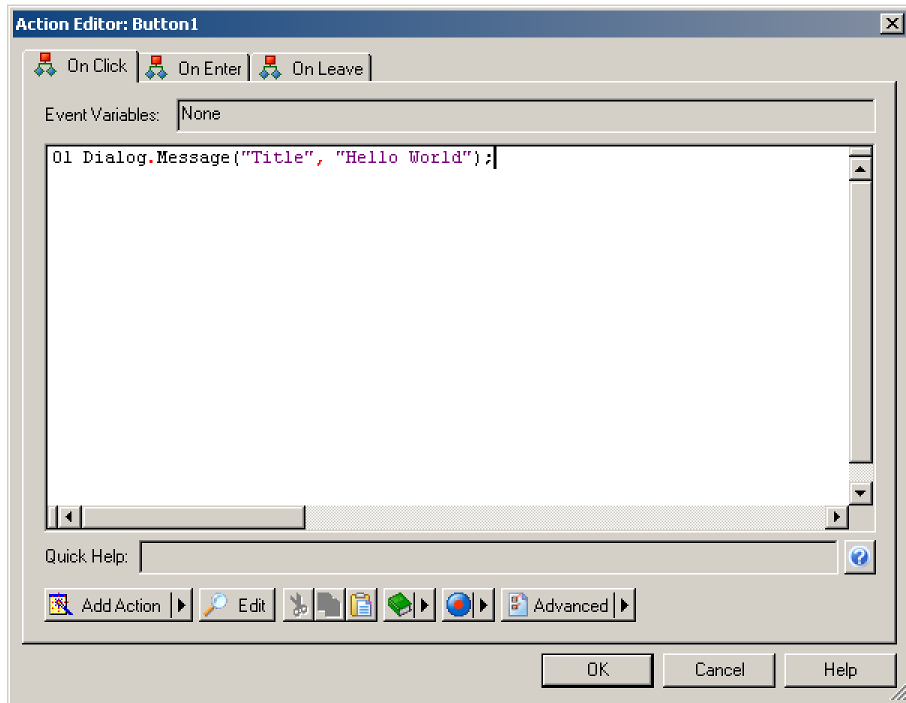


4. Click the edit button to open the action editor. Note that it opens directly to the On Click tab.

5. Type in the following text:

```
Dialog.Message("Title", "Hello World");
```

It should look like this when you're done:



6. Click OK to close the action editor.

7. Choose Project > Preview from the menu.

8. Once the application is running, click on the button that you created.

   This will trigger the button's On Click event, so that the script you entered will be performed.

You should see the following dialog appear:

Congratulations! You have just made your first script. Though this is a simple example, it shows you just how easy it is to make something happen in your AutoPlay application. You can use the above method to try out any script you want in AutoPlay Media Studio.

Important Scripting Concepts

There are a few important things that you should know about the AutoPlay Media Studio scripting language in general before we go on.

Script is Global

The scripting engine is global to the runtime environment. That means that all of your events will "know" about other variables and functions declared elsewhere in the product. For example, if you assign "myvar = 10;" in the project's On Startup event, myvar will still equal 10 when the next event is triggered. There are ways around this global nature (see *Variable Scope* on page 12), but it is generally true of the scripting engine.

Script is Case-Sensitive

The scripting engine is case-sensitive. This means that upper and lower case characters are important for things like keywords, variable names and function names.

For example:

```
ABC = 10;
aBC = 7;
```

In the above script, ABC and aBC refer to two different variables, and can hold

different values. The lowercase “a” in “aBC” makes it completely different from “ABC” as far as AutoPlay is concerned.

The same principle applies to function names as well. For example:

```
Dialog.Message("Hi", "Hello World");
```

...refers to a built-in AutoPlay function. However,

```
DIALOG.Message("Hi", "Hello World");
```

...will not be recognized as the built-in function, because DIALOG and Dialog are seen as two completely different names.

Note: It’s entirely possible to have two functions with the same spelling but different capitalization—for example, GreetUser and gREeTUSeR would be seen as two totally different functions. Although it’s definitely possible for such functions to coexist, it’s generally better to give functions completely different names to avoid any confusion.

Comments

You can insert non-executable comments into your scripts to explain and document your code. In a script, any text after two dashes (--) on a line will be ignored. For example:

```
-- Assign 10 to variable abc
abc = 10;
```

...or:

```
abc = 10; -- Assign 10 to abc
```

Both of the above examples do the exact same thing—the comments do not affect the script in any way.

You can also create multi-line comments by using --[[and]]-- on either side of the comment:

```
--[[ This is
a multi-line
comment ]]--
a = 10;
```

You should use comments to explain your scripts as much as possible in order to make them easier to understand by yourself and others.

Delimiting Statements

Each unique statement can either be on its own line and/or separated by a semi-colon (;). For example, all of the following scripts are valid:

Script 1:

```
a = 10
MyVar = a
```

Script 2:

```
a = 10; MyVar = a;
```

Script 3:

```
a = 10;
MyVar = a;
```

However, we recommend that you end all statements with a semi-colon (as in scripts 2 and 3 above).

Variables

What are Variables?

Variables are very important to scripting in AutoPlay. Variables are simply "nicknames" or "placeholders" for values that might need to be modified or re-used in the future. For example, the following script assigns the value 10 to a variable called "amount."

```
amount = 10;
```

Note: We say that values are "assigned to" or "stored in" variables. If you picture a variable as a container that can hold a value, assigning a value to a variable is like "placing" that value into a container. You can change this value at any time by

assigning a different value to the variable; the new value simply replaces the old one. This ability to hold changeable information is what makes variables so useful.

Here are a couple of examples demonstrating how you can operate on the “amount” variable:

```
amount = 10;
amount = amount + 20;
Dialog.Message("Value", amount);
```

This stores 10 in the variable named amount, then adds 20 to that value, and then finally makes a message box appear with the current value (which is now the number 30) in it.

You can also assign one variable to another:

```
a = 10;
b = a;
Dialog.Message("Value", b);
```

This will make a message box appear with the number 10 in it. The line “b = a;” assigns the value of “a” (which is 10) to “b.”

Variable Scope

As mentioned earlier in this document, all variables in AutoPlay Media Studio are *global* by default. This just means that they exist project-wide, and hold their values from one script to the next. In other words, if a value is assigned to a variable in one script, the variable will still hold that value when the next script is executed.

For example, if you enter the script:

```
foo = 10;
```

...into the current page’s On Open event, and then enter:

```
Dialog.Message("The value is:", foo);
```

...into a button object’s On Click event, the second script will use the value that was assigned to “foo” in the first script. As a result, when the button object is clicked, a message box will appear with the number 10 in it.

Note that the order of execution is important...in order for one script to be able to use the value that was assigned to the variable in another script, that other script has to be executed first. In the above example, the page's On Open event is triggered *before* the button's On Click event, so the value 10 is already assigned to foo when the On Click event's script is executed.

Local Variables

The global nature of the scripting engine means that a variable will retain its value throughout your entire project. You can, however, make variables that are non-global, by using the special keyword "local." Putting the word "local" in front of a variable assignment creates a variable that is local to the current script or function.

For example, let's say you have the following three scripts in the same project:

Script 1:

```
-- assign 10 to x
x = 10;
```

Script 2:

```
local x = 500;
Dialog.Message("Local value of x is:", x);
x = 250; -- this changes the local x, not the global one
Dialog.Message("Local value of x is:", x);
```

Script 3:

```
-- display the global value of x
Dialog.Message("Global value of x is:", x);
```

Let's assume these three scripts are performed one after the other. The first script gives x the value 10. Since all variables are global by default, x will have this value inside all other scripts, too. The second script makes a *local* assignment to x, giving it the value of 500—but only inside that script. If anything else inside that script wants to access the value of x, it will see the local value instead of the global one. It's like the "x" variable has been temporarily replaced by another variable that looks just like it, but has a different value.

(This reminds me of those caper movies, where the bank robbers put a picture in front of the security cameras so the guards won't see that the vault is being emptied. Only

in this case, it's like the bank robbers create a whole new working vault, just like the original, and then dismantle it when they leave.)

When told to display the contents of x, the first Dialog.Message action inside script #2 will display 500, since that is the local value of x when the action is performed. The next line assigns 250 to the local value of x—note that once you make a local variable, it completely replaces the global variable for the rest of the script.

Finally, the third script displays the global value of x, which is still 10.

Variable Naming

Variable names can be made up of any combination of letters, digits and underscores as long as they do not begin with a number and do not conflict with reserved keywords.

Examples of **valid** variables names:

```
a
strName
_My_Variable
data1
data_1_23
index
bReset
nCount
```

Examples of **invalid** variable names:

```
1
1data
%MyValue%
$strData
for
local
_FirstName+LastName_
User Name
```

Reserved Keywords

The following words are reserved and cannot be used for variable or function names:

and	break	do	else	elseif
end	false	for	function	if
in	local	nil	not	or
repeat	return	table	then	true
until	while			

Types and Values

AutoPlay's scripting language is dynamically typed. There are no type definitions—instead, each value carries its own type.

What this means is that you don't have to declare a variable to be of a certain type before using it. For example, in C++, if you want to use a number, you have to first declare the variable's type and then assign a value to it:

```
int j;
j = 10;
```

The above C++ example declares j as an integer, and then assigns 10 to it.

As we have seen, in AutoPlay you can just assign a value to a variable without declaring its type. Variables don't really have types; instead, it's the values inside them that are considered to be one type or another. For example:

```
j = 10;
```

...this automatically creates the variable named "j" and assigns the value 10 to it. Although this value has a type (it's a *number*), the variable itself is still typeless. This means that you can turn around and assign a different type of value to j, like so:

```
j = "Hello";
```

This replaces the number 10 that is stored in j with the string "Hello." The fact that a string is a different type of value doesn't matter; the variable j doesn't care what kind of value it holds, it just stores whatever you put in it.

There are six basic data types in AutoPlay: number, string, nil, boolean, function, and table. The sections below will explain each data type in more detail.

Number

A number is exactly that: a numeric value. The number type represents real numbers—specifically, double-precision floating-point values. There is no distinction between integers and floating-point numbers (also known as “fractions”)...all of them are just “numbers.” Here are some examples of valid numbers:

4 4. .4 0.4 4.57e-3 0.3e12

String

A string is simply a sequence of characters. For example, “Joe2” is a string of four characters, starting with a capital “J” and ending with the number “2.” Strings can vary widely in length; a string can contain a single letter, or a single word, or the contents of an entire book.

Strings may contain spaces and even more exotic characters, such as carriage returns and line feeds. In fact, strings may contain any combination of valid 8-bit ASCII characters, including null characters (“\0”). AutoPlay automatically manages string memory, so you never have to worry about allocating or de-allocating memory for strings.

Strings can be used quite intuitively and naturally. They should be delimited by matching single quotes or double quotes. Here are some examples that use strings:

```
Name = "Joe Blow";
Dialog.Message("Title", "Hello, how are you?");
LastName = 'Blow';
```

Normally double quotes are used for strings, but single quotes can be useful if you have a string that contains double quotes. Whichever type of quotes you use, you can include the other kind inside the string without escaping it. For example:

```
doubles = "How's that again?";
singles = 'She said "Talk to the hand," and I was all like "Dude!"';
```

If we used double quotes for the second line, it would look like this:

```
escaped = "She said \"Talk to the hand,\" and I was all like \"Dude!\"";
```

Normally, the scripting engine sees double quotes as marking the beginning or end of a string. In order to include double quotes inside a double-quoted string, you need to *escape* them with backslashes. This tells the scripting engine that you want to include an actual quote character *in* the string.

The backslash and quote (\") is known as an *escape sequence*. An escape sequence is a special sequence of characters that gets converted or "translated" into something else by the script engine. Escape sequences allow you to include things that can't be typed directly into a string.

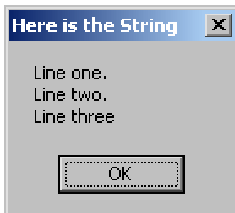
The escape sequences that you can use include:

\a - bell
\b - backspace
\f - form feed
\n - newline
\r - carriage return
\t - horizontal tab
\v - vertical tab
\\ - backslash
\" - quotation mark
\' - apostrophe
\[- left square bracket
\] - right square bracket

So, for example, if you want to represent three lines of text in a single string, you would use the following:

```
Lines = "Line one.\nLine two.\nLine three";
Dialog.Message("Here is the String", Lines);
```

This assigns a string to a variable named Lines, and uses the newline escape sequence to start a new line after each sentence. The Dialog.Message function displays the contents of the Lines variable in a message box, like this:



Another common example is when you want to represent a path to a file such as C:\My Folder\My Data.txt. You just need to remember to escape the backslashes:

```
MyPath = "C:\\My Folder\\My Data.txt";
```

Each double-backslash represents a single backslash when used inside a string.

If you know your ASCII table, you can use a backslash character followed by a number with up to three digits to represent any character by its ASCII value. For example, the ASCII value for a newline character is 10, so the following two lines do the exact same thing:

```
Lines = "Line one.\nLine two.\nLine three";
Lines = "Line one.\10Line two.\10Line three";
```

However, you will not need to use this format very often, if ever.

You can also define strings on multiple lines by using double square brackets ([[and]]). A string between double square brackets does not need any escape characters. The double square brackets lets you type special characters like backslashes, quotes and newlines right into the string. For example:

```
Lines = [[Line one.
Line two.
Line three.]];
```

is equivalent to:

```
Lines = "Line one.\nLine two.\nLine three";
```

This can be useful if you have preformatted text that you want to use as a string, and you don't want to have to convert all of the special characters into escape sequences.

The last important thing to know about strings is that the script engine provides automatic conversion between numbers and strings at run time. Whenever a numeric operation is applied to a string, the engine tries to convert the string to a number for the operation. Of course, this will only be successful if the string contains something that can be interpreted as a number.

For example, the following lines are both valid:

```
a = "10" + 1; -- Result is 11
b = "33" * 2; -- Result is 66
```

However, the following lines would not give you the same conversion result:

```
a = "10+1"; -- Result is the string "10+1"
b = "hello" + 1; -- ERROR, can't convert "hello" to a number
```

For more information on working with strings, see page 42.

Nil

Nil is a special value type. It basically represents the absence of any other kind of value.

You can assign nil to a variable, just like any other value. Note that this isn't the same as assigning the letters "nil" to a variable, as in a string. Like other keywords, nil must be left unquoted in order to be recognized. It should also be entered in all lowercase letters.

Nil will always evaluate to false when used in a condition:

```
a = nil;
if a then
    -- Any lines in here
    -- will not be executed
end
```

It can also be used to "delete" a variable:

```
y = "Joe Blow";
y = nil;
```

In the example above, "y" will no longer contain a value after the second line.

Boolean

Boolean variable types can have one of two values: true, or false. They can be used in conditions and to perform Boolean logic operations. For example:

```
boolybooly = true;
if boolybooly then
     -- Any script in here will be executed
end
```

This sets a variable named boolybooly to true, and then uses it in an if statement. Similarly:

```
a = true;
b = false;
if (a and b) then
     -- Any script here will not be executed because
     -- true and false is false.
end
```

This time, the if statement needs both “a” and “b” to be true in order for the lines inside it to be executed. In this case, that won’t happen because “b” has been set to false.

Function

The script engine allows you to define your own functions (or “sub-routines”), which are essentially small pieces of script that can be executed on demand. Each function has a name which is used to identify the function. You can actually use that function name as a special kind of value, in order to store a “reference” to that function in a variable, or to pass it to another function. This kind of reference is of the *function* type.

For more information on functions, see page 38.

Table

Tables are a very powerful way to store lists of indexed values under one name. Tables are actually associative arrays—that is, they are arrays which can be indexed not only with numbers, but with any kind of value (including strings).

Here are a few quick examples (we cover tables in more detail on page 30):

Example 1:

```
guys = {"Adam", "Brett", "Darryl"};
Dialog.Message("Second Name in the List", guys[2]);
```

This will display a message box with the word “Brett” in it.

Example 2:

```
t = {};
t.FirstName = "Michael";
t.LastName = "Jackson";
t.Occupation = "Singer";
Dialog.Message(t.FirstName, t.Occupation);
```

This will display the following message box:

You can assign tables to other variables as well. For example:

```
table_one = {};
table_one.FirstName = "Michael";
table_one.LastName = "Jackson";
table_one.Occupation = "Singer";
table_two = table_one;
occupation = table_two.Occupation;
Dialog.Message(b.FirstName, occupation);
```

Tables can be indexed using array notation (my_table[1]), or by dot notation if not indexed by numbers (my_table.LastName).

Note that when you assign one table to another, as in the following line:

```
table_two = table_one;
```

...this doesn't actually copy table_two into table_one. Instead, table_two and table_one both refer to the *same* table.

This is because the name of a table actually refers to an address in memory where the data within the table is stored. So when you assign the contents of the variable table_one to the variable table_two, you're copying the *address*, and not the actual data. You're essentially making the two variables "point" to the same table of data.

In order to copy the contents of a table, you need to create a new table and then copy all of the data over one item at a time.

For more information on copying tables, see page 35.

Variable Assignment

Variables can have new values assigned to them by using the assignment operator (=). This includes copying the value of one variable into another. For example:

```
a = 10;
b = "I am happy";
c = b;
```

It is interesting to note that the script engine supports multiple assignment:

```
a, b = 1, 2;
```

After the script above, the variable “a” contains the number 1 and the variable “b” contains the number 2.

Tables and functions are a bit of a special case: when you use the assignment operator on a table or function, you create an alias that points to the same table or function as the variable being “copied.” Programmers call this copying *by reference* as opposed to copying *by value*.

Expressions and Operators

An expression is anything that evaluates to a value. This can include a single value such as “6” or a compound value built with operators such as “1 + 3”. You can use parentheses to “group” expressions and control the order in which they are evaluated. For example, the following lines will all evaluate to the same value:

```
a = 10;
a = (5 * 1) * 2;
a = 100 / 10;
a = 100 / (2 * 5);
```

Arithmetic Operators

Arithmetic operators are used to perform mathematical operations on numbers. The following mathematical operators are supported:

Operator	Meaning
+	(addition)
-	(subtraction)
*	(multiplication)
/	(division)
unary -	(negation)

Here are some examples:

```
a = 5 + 2;
b = a * 100;
twentythreepercent = 23 / 100;
neg = -29;
pos = -neg;
```

Relational Operators

Relational operators allow you to compare how one value relates to another. The following relational operators are supported:

Operator	Meaning
>	(greater-than)
<	(less-than)
<=	(less-than or equal to)
>=	(greater than or equal to)
~=	(not equal to)
==	(equal)

All of the relational operators can be applied to any two numbers or any two strings. All other values can only use the == operator to see if they are equal.

Relational operators return Boolean values (true or false). For example:

```
10 > 20; -- resolves to false

a = 10;
a > 300; -- false
```

```
(3 * 200) > 500; -- true
"Brett" ~= "Lorne" -- true
```

One important point to mention is that the == and ~= operators test for *complete equality*, which means that any string comparisons done with those operators are case sensitive. For example:

```
"Jojoba" == "Jojoba"; -- true
"Wildcat" == "wildcat"; -- false
"I like it a lot" == "I like it a LOT"; -- false

"happy" ~= "HaPPy"; -- true
```

Logical Operators

Logical operators are used to perform Boolean operations on Boolean values. The following logical operators are supported:

and	(only true if both values are true)
or	(true if either value is true)
not	(returns the opposite of the value)

For example:

```
a = true;
b = false;
c = a and b; -- false
d = a and nil; -- false
e = not b; -- true
```

Note that only nil and false are considered to be false, and all other values are true.

For example:

```
iaminvisible = nil;
if iaminvisible then
     -- any lines in here won't happen
     -- because iaminvisible is considered false
     Dialog.Message("You can't see me!", "I am invisible!!!!");
end

if "Brett" then
     -- any lines in here WILL happen, because only nil and false
     -- are considered false...anything else, including strings,
```

```
    -- is considered true
    Dialog.Message("What about strings?", "Strings are true.");
end
```

Concatenation

In AutoPlay scripting, the concatenation operator is two periods (..). It is used to combine two or more strings together. You don't have to put spaces before and after the periods, but you can if you want to.

For example:

```
name = "Joe".." Blow"; -- assigns "Joe Blow" to name
b = name .. " is number " .. 1; -- assigns "Joe Blow is number 1" to b
```

Operator Precedence

Operators are said to have *precedence*, which is a way of describing the rules that determine which operations in a series of expressions get performed first. A simple example would be the expression 1 + 2 * 3. The multiply (*) operator has higher precedence than the add (+) operator, so this expression is equivalent to 1 + (2 * 3). In other words, the expression 2 * 3 is performed first, and then 1 + 6 is performed, resulting in the final value 7.

You can override the natural order of precedence by using parentheses. For instance, the expression (1 + 2) * 3 resolves to 9. The parentheses make the whole sub-expression "1 + 2" the left value of the multiply (*) operator. Essentially, the sub-expression 1 + 2 is evaluated first, and the result is then used in the expression 3 * 3.

Operator precedence follows the following order, from lowest to highest priority:

and	or				
<	>	<=	>=	~=	==
..					
+	-				
*	/				
not	- (unary)				
^					

Operators are also said to have *associativity*, which is a way of describing which expressions are performed first when the operators have equal precedence. In the

script engine, all binary operators are left associative, which means that whenever two operators have the same precedence, the operation on the left is performed first. The exception is the exponentiation operator (^), which is right-associative.

When in doubt, you can always use explicit parentheses to control precedence. For example:

```
a + 1 < b/2 + 1
```

...is the same as:

```
(a + 1) < ((b/2) + 1)
```

...and you can use parentheses to change the order of the calculations, too:

```
a + 1 < b/(2 + 1)
```

In this last example, instead of 1 being added to half of b, b is divided by 3.

Control Structures

The scripting engine supports the following control structures: if, while, repeat and for.

If

An if statement evaluates its condition and then only executes the "then" part if the condition is true. An if statement is terminated by the "end" keyword. The basic syntax is:

if *condition* then
    do something here
end

For example:

```
x = 50;
if x > 10 then
    Dialog.Message("result", "x is greater than 10");
end
```

```
y = 3;
if ((35 * y) < 100) then
    Dialog.Message("", "y times 35 is less than 100");
end
```

In the above script, only the first dialog message would be shown, because the second if condition isn't true...35 times 3 is 105, and 105 is not less than 100.

You can also use else and elseif to add more “branches” to the if statement:

```
x = 5;
if x > 10 then
    Dialog.Message("", "x is greater than 10");
else
    Dialog.Message("", "x is less than or equal to 10");
end
```

In the preceding example, the second dialog message would be shown, because 5 is not greater than 10.

```
x = 5;
if x == 10 then
    Dialog.Message("", "x is exactly 10");
elseif x == 11 then
    Dialog.Message("", "x is exactly 11");
elseif x == 12 then
    Dialog.Message("", "x is exactly 12");
else
    Dialog.Message("", "x is not 10, 11 or 12");
end
```

In that example, the last dialog message would be shown, because x is not equal to 10, or 11, or 12.

While

The while statement is used to execute the same "block" of script over and over until a condition is met. Like if statements, while statements are terminated with the “end” keyword. The basic syntax is:

> while *condition* do
>   *do something here*
> end

The condition must be true in order for the actions inside the while statement (the "do something here" part above) to be performed. The while statement will continue to loop as long as this condition is true. Here's how it works:

If the condition is true, all of the actions between the "while" and the corresponding "end" will be performed. When the "end" is reached, the condition will be reevaluated, and if it's still true, the actions between the "while" and the "end" will be performed again. The actions will continue to loop like this until the condition evaluates to false.

For example:

```
a = 1;
while a < 10 do
    a = a + 1;
end
```

In the preceding example, the "a = a + 1;" line would be performed 9 times.

You can break out of a while loop at any time using the "break" keyword. For example:

```
count = 1;
while count < 100 do
    count = count + 1;
    if count == 50 then
        break;
    end
end
```

Although the while statement is willing to count from 1 to 99, the if statement would cause this loop to terminate as soon as count reached 50.

Repeat

The repeat statement is similar to the while statement, except that the condition is checked at the *end* of the structure instead of at the beginning. The basic syntax is:

> repeat
>   *do something here*
> until *condition*

For example:

```
i = 1;
repeat
    i = i + 1;
until i > 10
```

This is similar to one of the while loops above, but this time, the loop is performed 10 times. The “i = i + 1;” part gets executed before the condition determines that a is now larger than 10.

You can break out of a repeat loop at any time using the “break” keyword. For example:

```
count = 1;
repeat
    count = count + 1;
    if count == 50 then
        break;
    end
until count > 100
```

Once again, this would exit from the loop as soon as count was equal to 50.

For

The for statement is used to repeat a block of script a specific number of times. The basic syntax is:

> for *variable* = *start*,*end*,*step* do
>   *do something here*
> end

The *variable* can be named anything you want. It is used to “count” the number of trips through the for loop. It begins at the *start* value you specify, and then changes by the amount in *step* after each trip through the loop. In other words, the *step* gets added to the value in the *variable* after the lines between the for and end are performed. If the result is smaller than or equal to the *end* value, the loop continues from the beginning.

For example:

```
-- This loop counts from 1 to 10:
for x = 1, 10 do
     Dialog.Message("Number", x);
end
```

The above example displays 10 dialog messages in a row, counting from 1 to 10. Note that the step is optional—if you don't provide a value for the step, it defaults to 1.

Here's an example that uses a step of "-1" to make the for loop count backwards:

```
-- This loop counts from 10 down to 1:
for x = 10, 1, -1 do
     Dialog.Message("Number", x);
end
```

That example would display 10 dialog messages in a row, counting back from 10 and going all the way down to 1.

You can break out of a for loop at any time using the "break" keyword. For example:

```
for i = 1, 100
    if count == 50 then
        break;
    end
end
```

Once again, this would exit from the loop as soon as count was equal to 50.

There is also a variation on the for loop that operates on tables. For more information on that, see *Using For to Enumerate Tables* on page 34.

Tables (Arrays)

Tables are very useful. They can be used to store any type of value, including functions or even other tables.

Creating Tables

There are generally two ways to create a table from scratch. The first way uses curly braces to specify a list of values:

```
my_table = {"apple","orange","peach"};

associative_table = {fruit="apple", vegetable="carrot"}
```

The second way is to create a blank table and then add the values one at a time:

```
my_table = {};
my_table[1] = "apple";
my_table[2] = "orange";
my_table[3] = "peach";

associative_table = {};
associative_table.fruit = "apple";
associative_table.vegetable = "carrot";
```

Accessing Table Elements

Each “record” of information stored in a table is known as an *element*. Each element consists of a key, which serves as the index into the table, and a value that is associated with that key.

There are generally two ways to access an element: you can use array notation, or dot notation. Array notation is typically used with numeric arrays, which are simply tables where all of the keys are numbers. Dot notation is typically used with associative arrays, which are tables where the keys are strings.

Here is an example of array notation:

```
t = { "one", "two", "three"};
Dialog.Message("Element one contains:", t[1]);
```

Here is an example of dot notation:

```
t = { first="one", second="two", third="three"};
Dialog.Message("Element 'first' contains:", t.first);
```

Numeric Arrays

One of the most common uses of tables is as arrays. An array is a collection of values that are indexed by numeric keys. In the scripting engine, numeric arrays are one-based. That is, they start at index 1.

Here are some examples using numeric arrays:

Example 1:

```
myArray = {255,0,255};
Dialog.Message("First Number", myArray[1]);
```

This would display a dialog message containing the number "255."

Example 2:

```
alphabet = {"a","b","c","d","e","f","g","h","i","j","k",
"l","m","n","o","p","q","r","s","t","u","v","w","x","y","z"};
Dialog.Message("Seventh Letter", alphabet[7]);
```

This would display a dialog message containing the letter "g."

Example 3:

```
myArray = {};
myArray[1] = "Option One";
myArray[2] = "Option Two";
myArray[3] = "Option Three";
```

This is exactly the same as the following:

```
myArray = {"Option One", "Option Two", "Option Three"};
```

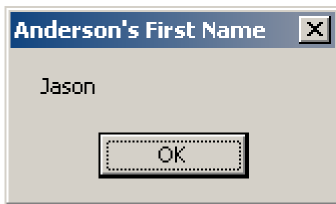
Associative Arrays

Associative arrays are the same as numerical arrays except that the indexes can be numbers, strings or even functions.

Here is an example of an associative array that uses a last name as an index and a first name as the value:

```
arrNames = {Anderson="Jason",
            Clemens="Roger",
            Contreras="Jose",
            Hammond="Chris",
            Hitchcock="Alfred"};

Dialog.Message("Anderson's First Name", arrNames.Anderson);
```

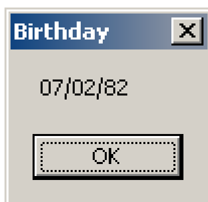
The resulting dialog message would look like this:



Here is an example of a simple employee database that keeps track of employee names and birth dates indexed by employee numbers:

```
Employees = {}; -- Construct an empty table for the employee numbers

-- store each employee's information in its own table
Employee1 = {Name="Jason Anderson", Birthday="07/02/82"};
Employee2 = {Name="Roger Clemens", Birthday="12/25/79"};

-- store each employee's information table
-- at the appropriate number in the Employees table
Employees[100099] = Employee1;
Employees[137637] = Employee2;

-- now typing "Employees[100099]" is the same as typing "Employee1"
Dialog.Message("Birthday",Employees[100099].Birthday);
```

The resulting dialog message would look like this:



Using For to Enumerate Tables

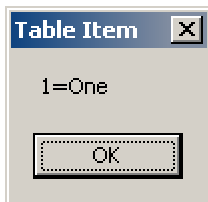
There is a special version of the for statement that allows you to quickly and easily enumerate the contents of an array. The syntax is:

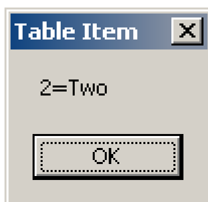
> for *index,value* in *table* do
>   *operate on index and value*
> end

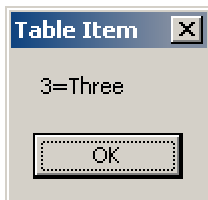
For example:

```
mytable = {"One","Two","Three"};

-- display a message for every table item
for j,k in mytable do
     Dialog.Message("Table Item", j .. "=" .. k);
end
```

The result would be three dialog messages in a row, one for each of the elements in mytable, like so:







Remember the above for statement, because it is a quick and easy way to inspect the values in a table. If you just want the indexes of a table, you can leave out the *value* part of the for statement:

```
a = {One=1,Two=2,Three=3};

for k in a do
    Dialog.Message("Table Index",k);
end
```

The above script will display three message boxes in a row, with the text “One,” “Three,” and then “Two.”

Whoa there—why aren’t the table elements in order? The reason for this is that internally the scripting engine doesn’t store tables as arrays, but in a super-efficient structure known as a hash table. (Don’t worry, I get confused about hash tables too.) The important thing to know is that when you define table elements, they are not necessarily stored in the order that you define or add them, unless you use a numeric array (i.e. a table indexed with numbers from 1 to whatever).

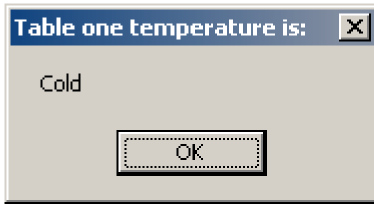
Copying Tables

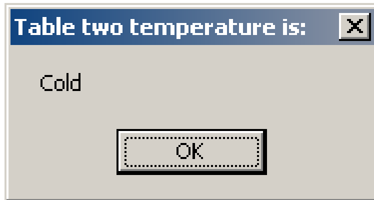
Copying tables is a bit different from copying other types of values. Unlike variables, you can't just use the assignment operator to copy the contents of one table into another. This is because the name of a table actually refers to an address in memory where the data within the table is stored. If you try to copy one table to another using the assignment operator, you end up copying the address, and not the actual data.

For example, if you wanted to copy a table, and then modify the copy, you might try something like this:

```
table_one = { mood="Happy", temperature="Warm" };

-- create a copy
table_two = table_one;

-- modify the copy
table_two.temperature = "Cold";

Dialog.Message("Table one temperature is:", table_one.temperature);
Dialog.Message("Table two temperature is:", table_two.temperature);
```

If you ran this script, you would see the following two dialogs:





Wait a minute...changing the "temperature" element in table_two also changed it in table_one. Why would they both change?

The answer is simply because the two are in fact the same table.

Internally, the name of a table just refers to a memory location. When table_one is created, a portion of memory is set aside to hold its contents. The location (or "address") of this memory is what gets assigned to the variable named table_one.

Assigning table_one to table_two just copies that memory address—not the actual memory itself.

It's like writing down the address of a library on a piece of paper, and then handing that paper to your friend. You aren't handing the entire library over, shelves of books and all...only the location where it can be found.

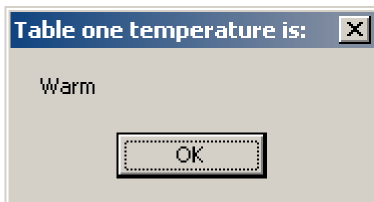
If you wanted to actually copy the library, you would have to create a new building, photocopy each book individually, and then store the photocopies in the new location.

That's pretty much how it is with tables, too. In order to create a full copy of a table, contents and all, you need to create a new table and then copy over all of the elements, one element at a time.

Luckily, the for statement makes this really easy to do. For example, here's a modified version of our earlier example, that creates a "true" copy of table_one.

```
table_one = { mood="Happy", temperature="Warm" };

-- create a copy
table_two = {};
for index, value in table_one do
    table_two[index] = value;
end

-- modify the copy
table_two.temperature = "Cold";

Dialog.Message("Table one temperature is:", table_one.temperature);
Dialog.Message("Table two temperature is:", table_two.temperature);
```

This time, the dialogs show that modifying table_two doesn't affect table_one at all:



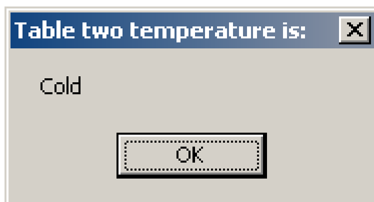


Table Functions

There are a number of built-in table functions at your disposal, which you can use to do such things as inserting elements into a table, removing elements from a table, and counting the number of elements in a table. For more information on these table functions, please see *Program Reference / Actions / Table* in the online help.

Functions

By far the coolest and most powerful feature of the scripting engine is functions. You have already seen a lot of functions used throughout this document, such as "Dialog.Message." Functions are simply portions of script that you can define, name and then call from anywhere else.

Although there are a lot of built-in AutoPlay functions, you can also make your own custom functions to suit your specific needs. In general, functions are defined as follows:

> function *function_name* (*arguments*)
>   *function script here*
>   return *return_value*;
> end

The first part is the keyword "function." This tells the scripting engine that what follows is a function definition. The *function_name* is simply a unique name for your function. The *arguments* are parameters (or values) that will be passed to the function every time it is called. A function can receive any number of arguments from 0 to infinity (well, not infinity, but don't get technical on me). The "return" keyword tells the function to return one or more values back to the script that called it.

The easiest way to learn about functions is to look at some examples. In this first example, we will make a simple function that shows a message box. It does not take any arguments and does not return anything.

```
function HelloWorld()
    Dialog.Message("Welcome","Hello World");
end
```

Notice that if you put the above script into an event and preview your application, nothing happens. Well, that is true and not true. It is true that nothing visible happens but the magic is in what you don't see. When the event is fired and the function script is executed, the function called "HelloWorld" becomes part of the scripting engine. That means it is now available to the rest of the application in any other script.

This brings up an important point about scripting in AutoPlay Media Studio. When making a function, the function does not get "into" the engine until the script is executed. That means that if you define HelloWorld() in a button's On Click event, but that event never gets triggered (because the user doesn't click on the button), the

HelloWorld() function will never exist. That is, you will not be able to call it from anywhere else.
That is why, in general, it is best to define your global functions in the global script of the project. (To access the global script, choose Project > Globals from the menu.)

Now back to the good stuff. Let's add a line to actually call the function:

```
function HelloWorld()
     Dialog.Message("Welcome","Hello World");
end

HelloWorld();
```

The "HelloWorld();" line tells the scripting engine to "go perform the function named HelloWorld." When that line gets executed, you would see a welcome message with the text "Hello World" in it.

Function Arguments

Let's take this a bit further and tell the message box which text to display by adding an argument to the function.

```
function HelloWorld(Message)
    Dialog.Message("Welcome", Message);
end

HelloWorld("This is an argument");
```

Now the message box shows the text that was "passed" to the function.

In the function definition, "Message" is a variable that will automatically receive whatever argument is passed to the function. In the function call, we pass the string "This is an argument" as the first (and only) argument for the HelloWorld function.

Here is an example of using multiple arguments.

```
function HelloWorld(Title, Message)
    Dialog.Message(Title, Message);
end

HelloWorld("This is argument one", "This is argument two");
HelloWorld("Welcome", "Hi there");
```

This time, the function definition uses two variables, one for each of its two arguments...and each function call passes two strings to the HelloWorld function.

Note that by changing the content of those strings, you can send different arguments to the function, and achieve different results.

Returning Values

The next step is to make the function return values back to the calling script. Here is a function that accepts a number as its single argument, and then returns a string containing all of the numbers from one to that number.

```
function Count(n)

    -- start out with a blank return string
    ReturnString = "";

    for num = 1,n do
     -- add the current number (num) to the end of the return string
     ReturnString = ReturnString..num;

     -- if this isn't the last number, then add a comma and a space
     -- to separate the numbers a bit in the return string
     if (num ~= n) then
        ReturnString = ReturnString..", ";
     end
    end

    -- return the string that we built
    return ReturnString;
end

CountString = Count(10);
Dialog.Message("Count", CountString);
```

The last two lines of the above script uses the Count function to build a string counting from 1 to 10, stores it in a variable named CountString, and then displays the contents of that variable in a dialog message box.

Returning Multiple Values

You can return multiple values from functions as well:

```
function SortNumbers(Number1, Number2)
     if Number1 <= Number2 then
       return Number1, Number2
     else
       return Number2, Number1
     end
end

firstNum, secondNum = SortNumbers(102, 100);
Dialog.Message("Sorted", firstNum ..", ".. secondNum);
```

The above script creates a function called SortNumbers that takes two arguments and then returns two values. The first value returned is the smaller number, and the second value returned is the larger one. Note that we specified two variables to receive the return values from the function call on the second last line. The last line of the script displays the two numbers in the order they were sorted into by the function.

Redefining Functions

Another interesting thing about functions is that you can override a previous function definition simply by re-defining it.

```
function HelloWorld()
     Dialog.Message("Message","Hello World");
end

function HelloWorld()
     Dialog.Message("Message","Hello Earth");
end

HelloWorld();
```

The script above shows a message box that says “Hello Earth,” and not “Hello World.” That is because the second version of the HelloWorld() function overrides the first one.

Putting Functions in Tables

One really powerful thing about tables is that they can be used to hold functions as well as other values. This is significant because it allows you to make sure that your functions have unique names and are logically grouped. (This is how all of the AutoPlay Media Studio functions are implemented.) Here is an example:

```
-- Make the functions:
function HelloEarth()
     Dialog.Message("Message","Hello Earth");
end

function HelloMoon()
     Dialog.Message("Message","Hello Moon");
end

-- Define an empty table:
Hello = {};

-- Assign the functions to the table:
Hello.Earth = HelloEarth;
Hello.Moon = HelloMoon;

-- Now call the functions:
Hello.Earth();
Hello.Moon();
```

It is also interesting to note that you can define functions right in your table definition:

```
Hello = {
Earth = function () Dialog.Message("Message","Hello Earth") end,
 Moon = function () Dialog.Message("Message","Hello Moon") end   };

-- Now call the functions:
Hello.Earth();
Hello.Moon();
```

String Manipulation

In this section we will briefly cover some of the most common string manipulation techniques, such as string concatenation and comparisons.

(For more information on the string functions available to you in AutoPlay Media Studio, see *Program Reference / Actions / String* in the online help.)

Concatenating Strings

We have already covered string concatenation, but it is well worth repeating. The string concatenation operator is two periods in a row (..). For example:

```
FullName = "Bo".." Derek";  -- FullName is now "Bo Derek"

-- You can also concatenate numbers into strings
DaysInYear = 365;
YearString = "There are "..DaysInYear.." days in a year.";
```

Note that you can put spaces on either side of the dots, or on one side, or not put any spaces at all. For example, the following four lines will accomplish the same thing:

```
foo = "Hello " .. user_name;
foo = "Hello ".. user_name;
foo = "Hello " ..user_name;
foo = "Hello "..user_name;
```

Comparing Strings

Next to concatenation, one of the most common things you will want to do with strings is compare one string to another. Depending on what constitutes a “match,” this can either be very simple, or just a bit tricky.

If you want to perform a case-sensitive comparison, then all you have to do is use the equals operator (==). For example:

```
strOne = "Strongbad";
strTwo = "Strongbad";

if strOne == strTwo then
    Dialog.Message("Guess what?", "The two strings are equal!");
else
    Dialog.Message("Hmmm", "The two strings are different.");
end
```

Since the == operator performs a case-sensitive comparison when applied to strings, the above script will display a message box proclaiming that the two strings are equal.

If you want to perform a case-*insensitive* comparison, then you need to take advantage of either the String.Upper or String.Lower function, to ensure that both strings have the same case before you compare them. The String.Upper function returns an all-

uppercase version of the string it is given, and the String.Lower function returns an all-lowercase version. Note that it doesn't matter which function you use in your comparison, so long as you use the same function on both sides of the == operator in your if statement.

For example:

```
strOne = "Mooohahahaha";
strTwo = "MOOohaHAHAha";

if String.Upper(strOne) == String.Upper(strTwo) then
    Dialog.Message("Guess what?", "The two strings are equal!");
else
    Dialog.Message("Hmmm", "The two strings are different.");
end
```

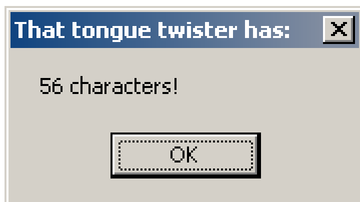
In the example above, the String.Upper function converts strOne to "MOOOHAHAHAHA" and strTwo to "MOOOHAHAHAHA" and then the if statement compares the results. (Note: the two original strings remain unchanged.) That way, it doesn't matter what case the original strings had; all that matters is whether the letters are the same.

Counting Characters

If you ever want to know how long a string is, you can easily count the number of characters it contains. Just use the String.Length function, like so:

```
twister = "If a wood chuck could chuck wood, how much would...um...";
num_chars = String.Length(twister);
Dialog.Message("That tongue twister has:", num_chars .. " characters!");
```

...which would produce the following dialog message:



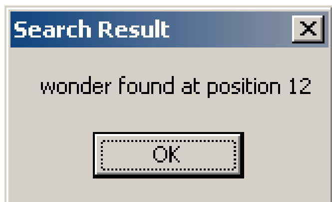
Finding Strings

Another common thing you'll want to do with strings is to search for one string within another. This is very simple to do using the String.Find action.

For example:

```
strSearchIn = "Isn't it a wonderful day outside?";
strSearchFor = "wonder";

-- search for strSearchIn inside strSearchFor
nFoundPos = String.Find(strSearchIn, strSearchFor);

if nFoundPos ~= nil then
     -- found it!
     Dialog.Message("Search Result", strSearchFor ..
                    " found at position " .. nFoundPos);
else
     -- no luck
     Dialog.Message("Search Result", strSearchFor .. " not found!");
end
```

...would cause the following message to be displayed:



Tip: Try experimenting with different values for strSearchFor and strSearchIn.

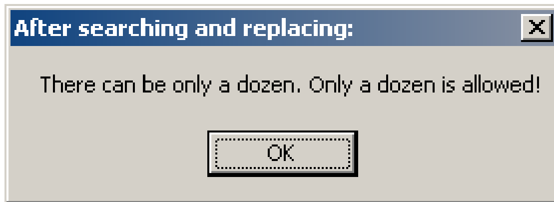
Replacing Strings

One of the most powerful things you can do with strings is to perform a search and replace operation on them. The following example shows how you can use the String.Replace action to replace every occurrence of a string with another inside a target string.

```
strTarget = "There can be only one. Only one is allowed!";
strSearchFor = "one";
strReplaceWith = "a dozen";
```

```
strNewString = String.Replace(strTarget, strSearchFor, strReplaceWith);
Dialog.Message("After searching and replacing:", strNewString);

-- create a copy of the target string with no spaces in it
strNoSpaces = String.Replace(strTarget, " ", "");
Dialog.Message("After removing spaces:", strNoSpaces);
```

The above example would display the following two messages:



Extracting Strings

There are three string functions that allow you to “extract” a portion of a string, rather than copying the entire string itself. These functions are String.Left, String.Right, and String.Mid.

String.Left copies a number of characters from the beginning of the string. String.Right does the same, but counting from the right end of the string instead. String.Mid allows you to copy a number of characters starting from any position in the string.

You can use these functions to perform all kinds of advanced operations on strings.

Here’s a basic example showing how they work:

```
strOriginal = "It really is good to see you again.";

-- copy the first 13 characters into strLeft
strLeft = String.Left(strOriginal, 13);
```

```
-- copy the last 18 characters into strRight
strRight = String.Right(strOriginal, 18);

-- create a new string with the two pieces
strNeo = String.Left .. "awesome" .. strRight .. " Whoa.";

-- copy the word "good" into strMiddle
strMiddle = String.Mid(strOriginal, 13, 4);
```

Converting Numeric Strings into Numbers

There may be times when you have a numeric string, and you need to convert it to a number.

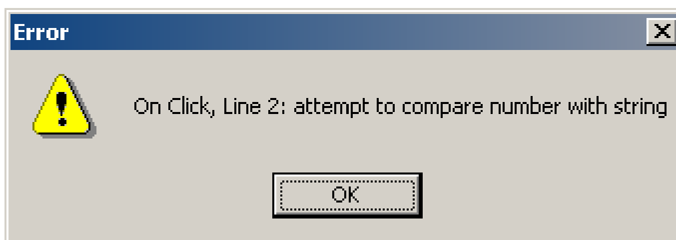
For example, if you have an input field where the user can enter their age, and you read in the text that they typed, you might get a value like “31”. Because they typed it in, though, this value is actually a string consisting of the characters “3” and “1”.

If you tried to compare this value to a number, you would get a syntax error saying that you attempted to compare a number with a string.

For example, the following script:

```
age = "31";
if age > 18 then
     Dialog.Message("", "You're older than 18.");
end
```

...would produce the following error message:



The problem in this case is the line that compares the contents of the variable “age” with the number 18:

```
if age > 18 then
```

This generates an error because age contains a string, and not a number. The script engine doesn't allow you to compare numbers with strings in this way. It has no way of knowing whether you wanted to treat age as a number, or treat 18 as a string.

The solution is simply to convert the value of age to a number before comparing it. There are two ways to do this. One way is to use the String.ToNumber function.

The String.ToNumber function translates a numeric string into the equivalent number, so it can be used in a numeric comparison.

```
age = "31";
if String.ToNumber(age) > 18 then
    Dialog.Message("", "You're older than 18.");
end
```

The other way takes advantage of the scripting engine's ability to convert numbers into strings when it knows what your intentions are. For example, if you're performing an arithmetic operation (such as adding two numbers), the engine will automatically convert any numeric strings to numbers for you:

```
age = "26" + 5; -- result is a numeric value
```

The above example would not generate any errors, because the scripting engine understands that the only way the statement makes sense is if you meant to use the numeric string as a number. As a result, the engine automatically converts the numeric string to a number so it can perform the calculation.

Knowing this, we can convert a numeric string to a number without changing its value by simply adding 0 to it, like so:

```
age = "31";
if (age + 0) > 18 then
    Dialog.Message("", "You're older than 18.");
end
```

In the preceding example, adding zero to the variable gets the engine to convert the value to a number, and the result is then compared with 18. No more error.

Other Built-in Functions

Script Functions

There are three other built-in functions that may prove useful to you: dofile, require, and type.

dofile

Loads and executes a script file. The contents of the file will be executed as though it was typed directly into the script. The syntax is:

dofile(*file_path*);

For example, say we typed the following script into a file called MyScript.lua (just a text file containing this script, created with notepad or some other text editor):

```
Dialog.Message("Hello", "World");
```

Now we drag and drop the file onto AutoPlay Media Studio's main window. (This will copy the file into the project's Modules folder.) A dialog will appear that asks if we want to add a require line to our global script. Click "No" for now. We will explain the require statement later.

Now, wherever we add the following line of script to an event:

```
dofile(_SourceFolder.."\\AutoPlay\\Modules\\MyScript.lua");
```

...that script file will be read in and executed immediately. In this case, you would see a message box with the friendly "hello world" message.

Tip: Use the dofile function to save yourself from having to re-type or re-paste a script into your projects over and over again.

require

Loads and runs a script file into the scripting engine. It is similar to dofile except that it will only load a given file once per session, whereas dofile will re-load and re-run the file each time it is used. The syntax is:

require(*file_path*);

So, for example, even if you do two requires in a row:

```
require("foo.lua");
require("foo.lua"); -- this line won't do anything
```

...only the first one will ever get executed. After that, the scripting engine knows that the file has been loaded and run, and future calls to require that file will have no effect.

Note that as long as you put the .lua file into your project's Modules folder, you don't even have to provide a full path to the file. For example:

```
require("MyScript.lua");
```

...is the same as:

```
require(_SourceFolder.."\\AutoPlay\\Modules\\MyScript.lua");
```

Since require will only load a given script file once per session, it is best suited for loading scripts that contain only variables and functions. Since variables and functions are global by default, you only need to "load" them once; repeatedly loading the same function definition would just be a waste of time.

This makes the require function a great way to load external script libraries. Every script that needs a function from an external file can safely require() it, and the file will only actually be loaded the first time it's needed.

type

This function will tell you the type of value contained in a variable. It returns the string name of the variable type. Valid return values are "nil," "number," "string," "boolean," "table," or "function." For example:

```
a = 989;
strType = type(a);  -- sets strType to "number"

a = "Hi there";
strType = type(a);  -- sets strType to "string"
```

The type function is especially useful when writing your own functions that need certain data types in order to operate. For example, the following function uses type() to make sure that both of its arguments are numbers:

```
-- find the maximum of two numbers
function Max(Number1, Number2)

    -- make sure both arguments are numeric
    if (type(Number1) ~= "number") or (type(Number2) ~= "number") then
        Dialog.Message("Error", "Please enter numbers");
        return nil -- we're using nil to indicate an error condition
    else
        if Number1 >= Number2 then
            return Number1;
        else
            return Number2;
        end
    end
end
```

Actions

AutoPlay Media Studio comes with a large number of built-in functions. In the program interface, these built-in functions are commonly referred to as *actions*. For scripting purposes, actions and functions are essentially the same; however, the term “actions” is generally reserved for those functions that are built into the program and are included in the alphabetical list of actions in the online help. When referring to functions that have been created by other users or yourself, the term “functions” is preferred.

Debugging Your Scripts

Scripting (or any kind of programming) is relatively easy once you get used to it. However, even the best programmers make mistakes, and need to iron the occasional wrinkle out of their code. Being good at debugging scripts will reduce the time to market for your projects and increase the amount of sleep you get at night. Please read this section for tips on using Auto Play Media Studio as smartly and effectively as possible!

This section will explain AutoPlay Media Studio’s error handling methods as well as cover a number of debugging techniques.

Error Handling

All of the built-in AutoPlay Media Studio actions use the same basic error handling techniques. However, this is not necessarily true of any third-party functions, modules, or scripts—even scripts developed by Indigo Rose Corporation that are not built into the product. Although these externally developed scripts can certainly make use of AutoPlay's error handling system, they may not necessarily do so. Therefore, you should always consult a script or module's author or documentation in order to find out how error handling is, well, handled.

There are two kinds of errors that you can have in your scripts when calling AutoPlay Media Studio actions: syntax errors, and functional errors.

Syntax Errors

Syntax errors occur when the syntax (or “grammar”) of a script is incorrect, or a function receives arguments that are not appropriate. Some syntax errors are caught by AutoPlay Media Studio when you build or preview your application.

For example, consider the following script:

```
foo =
```

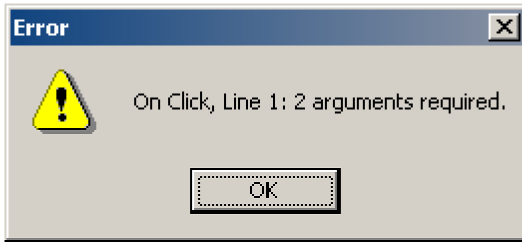
This is incorrect because we have not assigned anything to the variable foo—the script is incomplete. This is a pretty obvious syntax error, and would be caught by the scripting engine at build time (when you build your project).

Another type of syntax error is when you do not pass the correct type or number of arguments to a function. For example, if you try and run this script:

```
Dialog.Message("Hi There");
```

...the project will build fine, because there are no *obvious* syntax errors in the script. As far as the scripting engine can tell, the function call is well formed. The name is valid, the open and closed parentheses match, the quotes are in the right places, and there’s even a terminating semi-colon at the end. Looks good!

However, at run time you would see something like the following:



Looks like it wasn't so good after all. Note that the message says two arguments are required for the Dialog.Message function. Ah. Our script only provided one argument.

According to the function prototype for Dialog.Message, it looks like the function can actually accept up to *five* arguments:

```
number Dialog.Message ( string Title,
                        string Text,
                        number Type = MB_OK,
                        number Icon = MB_ICONNONE,
                        number DefaultButton = MB_DEFBUTTON1 )
```

Looking closely at the function prototype, we see that the last three arguments have default values which will be used if those arguments are omitted from the function call. The first two arguments—Title and Text—don't have default values, so they cannot be omitted without generating an error. To make a long story short, it's okay to call the Dialog.Message action with anywhere from 2 to 5 arguments...but 1 argument isn't enough.

Fortunately, syntax errors like these are usually caught at build time or when you test your application. The error messages are usually quite clear, making it easy for you to locate and identify the problem.

Functional Errors

Functional errors are those that occur because the functionality of the action itself fails. They occur when an action is given incorrect information, such as the path to a file that doesn't exist. For example, the following code will produce a functional error:

```
filecontents = TextFile.ReadToString("this_file_don't exist.txt");
```

If you put that script into an event right now and try it, you will see that nothing appears to happen. This is because AutoPlay Media Studio's functional errors are not automatically displayed the way syntax errors are. We leave it up to you to handle (or to not handle) such functional errors yourself.

The reason for this is that there may be times when you don't care if a function fails. In fact, you may expect it to. For example, the following code tries to remove a folder called C:\My Temp Folder:

```
Folder.Delete("C:\\My Temp Folder");
```

However, in this case you don't care if it really gets deleted, or if the folder didn't exist in the first place. You just want to make sure that if that particular folder exists, it will be removed. If the folder isn't there, the Folder.Delete action causes a functional error, because it can't find the folder you told it to delete...but since the end result is exactly what you wanted, you don't need to do anything about it. And you certainly don't want the user to see any error messages.

Conversely, there may be times when it is very important for you to know if an action fails. Say for instance that you want to copy a very important file:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");
```

In this case, you really want to know if it fails and may even want to exit the program or inform the user. This is where the Debug actions come in handy. Read on.

Debug Actions

AutoPlay Media Studio comes with some very useful functions for debugging your applications. This section will look at a number of them.

Application.GetLastError

This is the most important action to use when trying to find out if a problem has occurred. At run time there is always an internal value that stores the status of the last action that was executed. At the start of an action, this value is set to 0 (the number zero). This means that everything is OK. If a functional error occurs inside the action, the value is changed to some non-zero value instead.

This last error value can be accessed at any time by using the Application.GetLastError action.

The syntax is:

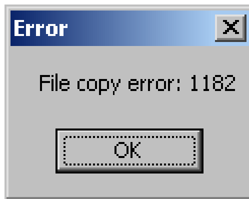
last_error_code = Application.GetLastError();

Here is an example that uses this action:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");

error_code = Application.GetLastError();
if (error_code ~= 0) then
     -- some kind of error has occurred!
     Dialog.Message("Error", "File copy error: "..error_code);
     Application.Exit();
end
```

The above script will inform the user that an error occurred and then exit the application. This is not necessarily how all errors should be handled, but it illustrates the point. You can do anything you want when an error occurs, like calling a different function or anything else you can dream up.

The above script has one possible problem. Imagine the user seeing a message like this:

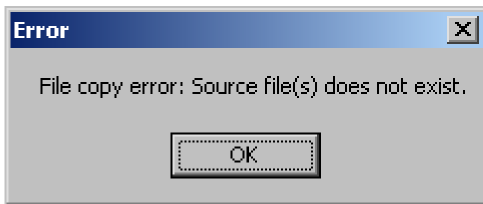


It would be much nicer to actually tell them some information about the exact problem. Well, you are in luck! At run time there is a table called _tblErrorMessages that contains all of the possible error messages, indexed by the error codes. You can easily use the last error number to get an actual error message that will make more sense to the user than a number like "1182."

For example, here is a modified script to show the actual error string:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");

error_code = Application.GetLastError();
if (error_code ~= 0) then
     -- some kind of error has occurred!
     Dialog.Message("Error", "File copy error: " ..
                             _tblErrorMessages[error_code]);
     Application.Exit();
end
```

Now the script will produce the following error message:



Much better information!

Just remember that the value of the last error gets reset every time an action is executed. For example, the following script would not produce an error message:

```
File.Copy("C:\\Temp\\My File.dat","C:\\Temp\\My File.bak");

-- At this point Application.GetLastError() could be non-zero, but...

Dialog.Message("Hi There","Hello World");

-- Oops, now the last error number will be for the Dialog.Message action,
-- and not the File.Copy action. The Dialog.Message action will reset the
-- last error number to 0, and the following lines will not catch any
-- error that happened in the File.Copy action.

error_code = Application.GetLastError();

if (error_code ~= 0) then
     -- some kind of error has occurred!
     Dialog.Message("Error",
                    "File copy error: "..
                    _tblErrorMessages[error_code]);
     Application.Exit();
end
```

Debug.ShowWindow

The AutoPlay Media Studio runtime has the ability to show a debug window that can be used to display debug messages. This window exists throughout the execution of your application, but is only visible when you tell it to be.

The syntax is:

Debug.ShowWindow(*show_window*);

...where *show_window* is a Boolean value. If true, the debug window is displayed, if false, the window is hidden. For example:

```
-- show the debug window
Debug.ShowWindow(true);
```

If you call this script, the debug window will appear on top of your application, but nothing else will really happen. That's where the following Debug actions come in.

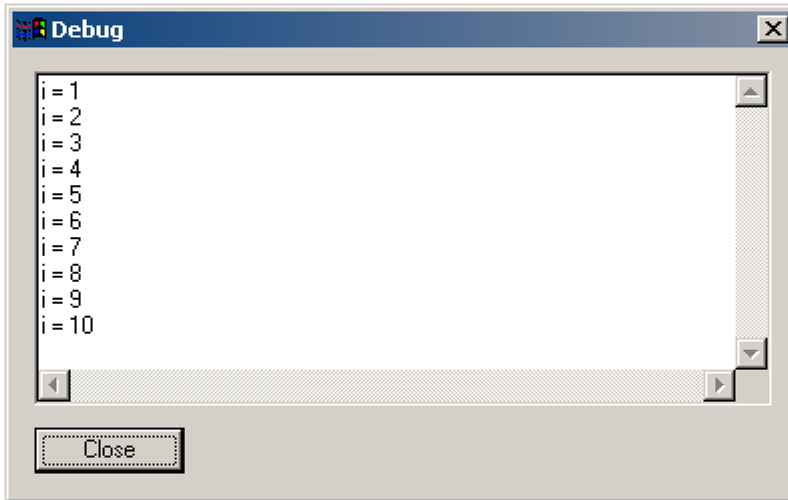
Debug.Print

This action prints the text of your choosing in the debug window. For example, try the following script:

```
Debug.ShowWindow(true);

for i = 1, 10 do
    Debug.Print("i = " .. i .. "\r\n");
end
```

The "\r\n" part is actually two escape sequences that are being used to start a new line. (This is technically called a "carriage return/linefeed" pair.) You can use \r\n in the debug window whenever you want to insert a new line.

The above script will produce the following output in the debug window:



You can use this method to print all kinds of information to the debug window. Some typical uses are to print the contents of a variable so you can see what it contains at run time, or to print your own debug messages like “inside outer for loop” or “foo() function started.” Such messages form a trail like bread crumbs that you can trace in order to understand what’s happening behind the scenes in your project. They can be invaluable when trying to debug your scripts or test your latest algorithm.

Debug.SetTraceMode

AutoPlay Media Studio can run in a special “trace” mode at run time that will print information about every line of script that gets executed to the debug window, including the value of Application.GetLastError() if the line involves calling a built-in action. You can turn this trace mode on or off by using the Debug.SetTraceMode action:

Debug.SetTraceMode(*turn_on*);

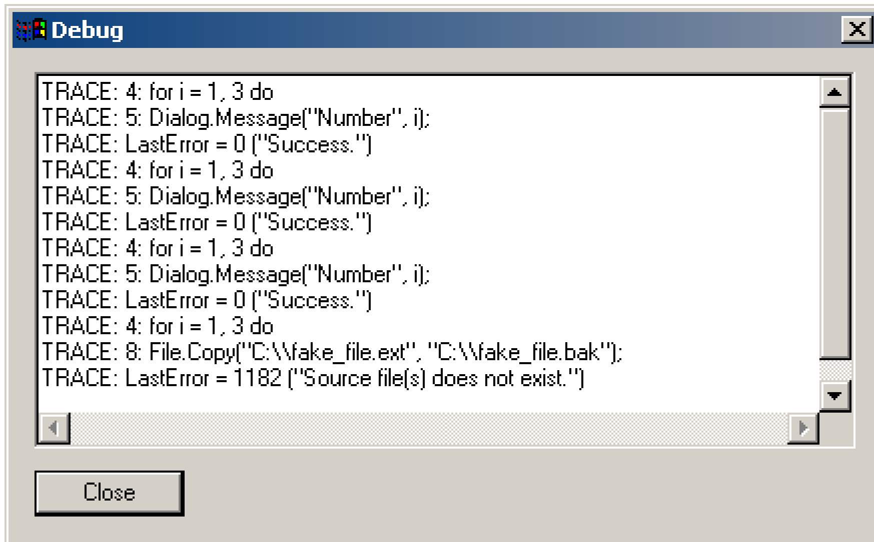
...where *turn_on* is a Boolean value that tells the program whether to turn the trace mode on or off.

Here is an example:

```
Debug.ShowWindow(true);
Debug.SetTraceMode(true);
```

```
for i = 1, 3 do
     Dialog.Message("Number", i);
end

File.Copy("C:\\fake_file.ext", "C:\\fake_file.bak");
```

Running that script will produce the following output in the debug window:



Notice that every line produced by the trace mode starts with "TRACE:" This is so you can tell them apart from any lines you send to the debug window with Debug.Print. The number after the "TRACE:" part is the line number that is currently being executed in the script.

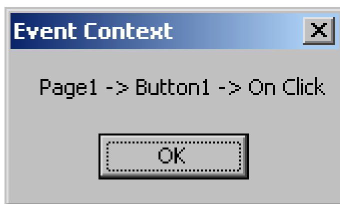
Turning trace mode on is something that you will not likely want to do in your final, distributable application, but it can really help find problems during development. In fact, there is an option in the AutoPlay Media Studio build preferences that allows you to automatically turn on trace mode every time you preview. (Choose Tools > Preferences and click on the Build category. In the Preview section, turn on the option called "Show Debug Window.") However, even with this option enabled, trace mode will not be turned on in the version that gets built when you publish your project, unless you specifically turn it on in your script.

Debug.GetEventContext

This action is used to get a descriptive string about the event that is currently being executed. This can be useful if you define a function in one place but call it somewhere else, and you want to be able to tell where the function is being called from at any given time. For example, if you execute this script from a button's On Click event on Page1:

```
Dialog.Message("Event Context", Debug.GetEventContext());
```

...you will see something like this:



Dialog.Message

This brings us to good ole' Dialog.Message. You have seen this action used throughout this document, and for good reason. This is a great action to use throughout your code when you are trying to track down a problem.

Final Thoughts

Hopefully this document has helped you to understand scripting in AutoPlay Media Studio 5.0. Once you get the hang of it, it is a really fun, powerful way to get things done.

Other Resources

Here is a list of other places that you can go for help with scripting in AutoPlay Media Studio 5.0.

Help File

The AutoPlay Media Studio help file is packed with good reference material for all of the actions and events supported by AutoPlay Media Studio, and for the design

environment itself. You can access the help file at any time by choosing Help > AutoPlay Media Studio Help from the menu.

Another useful tip: if you are in the script editor and you want to learn more about an action, simply click on the action and hit the F1 key on your keyboard.

User's Guide

The user's guide is a fantastic way to get started with AutoPlay Media Studio 5.0. It is written in an easy-to-follow tutorial format, teaching you about events, actions and scripts. You'll be off and running in no time! You can access the user's guide by choosing Help > Getting Started from the menu.

AutoPlay Media Studio Web Site

The AutoPlay Media Studio Web site is located at http://www.autoplaystudio.com. Be sure to check out the user forums where you can read questions and answers by fellow users and Indigo Rose staff as well as ask questions of your own.

A quick way to access the online forums is to choose Help > Online Forums from the menu.

Indigo Rose Technical Support

If you need help with any scripting concepts or have a mental block to push through, feel free to email us at support@indigorose.com. Although we can't write scripts for you or debug your specific scripts, we will be happy to answer any general scripting questions that you have.

The Lua Web Site

AutoPlay's scripting engine is based on a popular scripting language called *Lua*. Lua is designed and implemented by a team at Tecgraf, the Computer Graphics Technology Group of PUC-Rio (the Pontifical Catholic University of Rio de Janeiro in Brazil). You can learn more about Lua and its history at the official Lua web site:

http://www.lua.org

This is also where you can find the latest documentation on the Lua language, along with tutorials and a really friendly community of Lua developers.

Note that there may be other built-in functions that exist in Lua and in AutoPlay Media Studio that are not officially supported in AutoPlay. These functions, if any, are documented in the Lua 5.0 Reference Manual.

Only the functions listed in the online help are supported by Indigo Rose Software. Any other “undocumented” functions that you may find in the Lua documentation are not supported. Although these functions may work, you must use them entirely on your own.