BrightScript 2 Reference

Feb 24, 2009

Table of Contents

Introduction

Roku BrightScript is a powerful scripting language that makes it easy and quick to build media and networked applications for embedded devices. The language has integrated support for Roku Objects, a library of lightweight components. The APIs of the platform (device) the BrightScript is running on are all exposed to BrightScript as Roku Objects.

This document specifies the syntax of the language. To write useful applications, you should also refer to the Roku Object Reference Manual for the device you are targeting code for. This manual is designed for people that have some experience programming software. It is a reference guide, not a tutorial.

BrightScript compiles code into bytecode that is run by an interpreter. This compilation step happens every time a script is loaded and run. There is no separate compile step that results in a binary file being saved. In this way it is similar to JavaScript.

BrightScript statement syntax is not C-like; in this way it is similar to Python or Basic or Ruby. BrightScript Objects and named entry data structures are Associative Arrays; in this way it is similar to JavaScript or Lua. BrightScript supports dynamic typing (like JavaScript), or declared types (like C or Java). BrightScript uses “interfaces” and “components” for its APIs; similar to “.Net” or Java.

BrightScript is a powerful bytecode interpreted scripting language optimized for embedded devices; in this way it is unique. For example, BrightScript and the Roku Object architecture are written in 100% C for speed, efficiency, and portability. BrightScript makes extensive use of the “integer” type (since many embedded processors don’t have floating point units). This is different from languages like JavaScript where a number is always a float. BrightScript numbers are only floats when necessary.

If you want to get a quick flavor of BrightScript code, see the Appendix of this manual for the game “snake”.

BrightScript is optimized to be the “glue” that connects underling components for network connectivity, media playback, and UI screens into user friendly applications with minimal programmer effort.

Statement Summary

BrightScript supports the following familiar looking statement types:

- If / Then / Else If / Else / End If
- For / To / Next / Step / Exit For
- For Each / In / Next / Exit For
- While / End While / Exit While
- Function / End Function / As / Return
- Sub / End Sub
- Print
- Rem (or ‘)
- Goto
- Dim
- End
- Stop

BrightScript is not case sensitive.

Each statement’s syntax is documented precisely later in the manual.

Here is an example:

```
Function Main() As Void

    dim cavemen[10]

    cavemen.push("fred")
    cavemen.push("barney")
    cavemen.push("wilma")
    cavemen.push("betty")

    for each caveman in cavemen
        print caveman
    next

End Function
```

Each line may contain a single statement, or a colon (:) may be used to separate multiple statements on a single line.

```
myname = “fred”
if myname=”fred” then yourname = “barney”:print yourname
```

Expressions, Variables, and Types

Identifiers

Identifiers (names of variables, functions, labels, or object member functions or interfaces (appear after a ".")) have the following rules.

- must start with an alphabetic character (a – z)
- may consist of alphabetic characters, numbers, or the symbol "_" (underscore)
- are not case sensitive
- may be of any length
- may not use a "reserved word" as the name (see appendix for list of reserved words).
- if a variable: may end with an optional type designator character ($ for string, % for integer, ! for float, # for double) (functions do not support this).

For example:

```
a
boy5
super_man$
```

Types

BrightScript uses dynamic typing. This means that every value also has a type determined at run time. However, BrightScript also supports declared types. This means that a variable can be made to always contain a value of a specific type. If a value is assigned to a variable (which has a specific type), the type of the value assigned will be converted to the variables type, if possible. If not possible, a runtime error will result.

The following types are supported in BrightScript:

- **Boolean** – either true or false
- **Integer**– 32 bit signed integer number
- **Float** – the smallest floating point number format supported by the hardware or software
- **Double** - the largest floating point number format supported by the hardware or software. Note that although BrightScript supports Double, Roku Objects do not.
- **String**. – a sequence of ASCII characters. Currently strings are ASCII, not UTF-8.
- **Object** – a reference to a Roku Object (native component). Note that if you use the "type()" function, you will not get "rotOBJECT". Instead you will get the type of object. E.g.: "roList", "roVideoPlayer", etc. Also note that there is no special type for "intrinsic" BrightScript objects. BrightScript objects are all built on the Roku Object type "roAssociatiaveArray".
- **Interface**- An interface in a Roku Object. If a "dot operator" is used on an interface type, the member must be static (since there is no object context).
- **Invalid** – the type invalid has only one value – invalid. It is returned in various cases, for example, when indexing an array that has never been set.

- **Dynamic typing** – Unless otherwise specified, a variable is dynamically typed. This means that the type is determined by the value assigned to it at evaluation time. For example "1" is an int, "2.3" is a float, "hello" is a string, etc. A variable that does not end in a type specifier character is dynamically typed. It will take on the type of the expression assigned to it, and may change its type. For example: a=4 creates a as integer, then a = "hello", changes the variable a to a string.

Here are some examples of types. ? is a short cut for the "print" statement. The "type()" function returns a string that identifies the type of the expression passed in.

```
BrightScript Micro Debugger.
Enter any BrightScript statement, debug commands, or HELP.

BrightScript> ?type(1)
Integer

BrightScript> ?type(1.0)
Float

BrightScript> ?type("hello")
String

BrightScript> ?type(CreateObject("roList"))
roList

BrightScript> ?type(1%)
Integer

BrightScript> b!=1
BrightScript> ?type(b!)
Float

BrightScript> c$="hello"
BrightScript> ?type(c$)
String

BrightScript> d="hello again"
BrightScript> ?type(d)
String

BrightScript> d=1
BrightScript> ?type(d)
Integer

BrightScript> d=1.0
BrightScript> ?type(d)
Float
```

Literals (Constants)

Type Boolean: true, false
Type Invalid: invalid

Type String: String in quotes, eg “this is a string”
Type Integer: Hex integer, eg. &HFF, or decimal integer, eg. 255
Type Float: e.g., 2.01 or 1.23456E+30 or 2!
Type Double: eg, 1.23456789D-12, or .2.3#
Type BrSub , eg: MyFunction
Type Integer: LINE_NUM – the current source line number.

The following rules determine how integers, doubles, and floats are determined:

1. If a constant contains 10 or more digits, or if D is used in the exponent, that number is double precision. Adding a # declaration character also forces a constant to be double precision.

2. If the number is not double-precision, and if it contains a decimal point, then the number is float. If the number is expressed in exponential notation with E preceding the exponent, the number is float.

3. If neither of the above is true of the constant, then it is an integer.

Array “literal”
The Array Operator [ ] can be used to declare an array. It may contain literals (constants), or expressions. E.g:

```
Myarray = []
Myarray = [ 1, 2, 3]
Myarray = [ x+5, true, 1<>2, [“a”,”b”]]
```

Associative Array Literal
The { } operator can be used to define an Associative Array. It can contain literals or expressions. E.g:

```
aa={ }
aa={key1:”value”, key2: 55, key3: 5+3 }
```

Both Arrays and Associative Arrays can also have this form:

```
aa = {
   Myfunc1: aFunction
   Myval1 : “the value”
}
```

Note on Invalid vs. Object
Certain functions that return objects can also return invalid (for example, in the case when there is no object to return). In which case, the variable accepting the result must be dynamic, since it may get “invalid” or it may get an “object”.

```
l=[]
a$=l.pop()
```

This example will return a type mismatch (a$ is a string, and can not contain “invalid”). Many functions that return objects can return invalid as well

Type Declaration Characters

A type declaration character may be used at the end of a variable or literal to fix its type. Variables with the same identifier but separate types are separate variables. For example, a, a$, and a% are all independent.

Character	Type	Examples
$	String	A$, ZZ$
%	Integer	A1%, SUM%
!	Single-Precision (float)	B!, N1!
#	Double-Precision (double)	A#, 1/3#, 2#

Type Conversion (Promotion)

When operations are performed on one or two numbers, the result must be typed as integer, double or single-precision (float). When a +, -, or * operation is performed, the result will have the same degree of precision as the most precise operand. For example, if one operand is integer, and the other double-precision, the result will be double precision. Only when both operands are integers will a result be integer. If the result of an integer *, -, or + operation is outside the integer range, the operation will be done in double precision and the result will be double precision.

Division follows the same rules as +, * and -, except that it is never done at the integer level: when both operators are integers, the operation is done as float with a float result
.
During a compare operation (< , >,=,etc.) the operands are converted to the same type before they are compared. The less precise type will always be converted to the more precise type.

The logical operators AND, OR and NOT first convert their operands to Boolean. The result of a logical operation is always a Boolean.

Effects of Type Conversions on Accuracy

When a number is converted to integer type, it is "rounded down"; i.e., the largest integer, which is not greater than the number is used. (This is the same thing that happens when the INT function is applied to the number.)

When a number is converted from double to single precision, it is "4/5 rounded" (the least significant digit is rounded up if the fractional part > =5. Otherwise, it is left unchanged).

When a single precision number is converted to double precision, only the seven most significant digits will be accurate.

Operators

Operations in the innermost level of parentheses are performed first, then evaluation proceeds according to the precedence in the following table. Operations on the same precedence are right-to-left associative, except for exponentiation, which is right-to-left.

( ) Function call, or Parentheses
. , [] Array Operator
^ (Exponentiation)
–, + (Negation)
*, /
+, -
<, >, = , <>, <=, >=
NOT
AND
OR

String Operators

The following operators work with strings
<. >, =, <>, <=, >=, +

Function References

= , <> work on variables that contain function references and function literals

Logical and Bitwise Operators

Example:

```
if a=c and not(b>40) then print "success"
```

AND, OR and NOT can be used for logical (Boolean) or bit manipulation & bitwise comparisons. If the arguments to these operators are Boolean, then they perform a logical operation. If the arguments are numeric, they perform bitwise operations.

```
x = 1 and 2    ' x is zero
y = true and false  ' y is false
```

When AND and OR are used for logical operations, only the necessary amount of the expression is executed. For example:

```
print true or invalid
```

The above statement will print “true”, where as:

```
print false or invalid
```

Will cause a runtime error because “invalid” is not a valid expression.

“dot” Operator

The “.” Operator can be used on any Roku Object or any AssociativeArray. It also has special meaning when used on roXMLElement or roXMLList. When used on a Roku Object, it refers to an interface or a member function. For example:

```
i=CreateObject(“roInt”)
i.ifInt.SetInt(5)
i.SetInt(5)
```

“ifInt” is the interface, and “SetInt” is the member function. Every member function of a Roku Object is part of an interface. However, specifying the interface with the dot operator is optional. If it is left out, as in the last line of the example above, each interface in the object is searched for the member function. If there is a conflict (a member function with the same name appearing in two interfaces), then the interface should be specified.

When the “.” Operator is used on an Associative Array, it is the same as calling the Lookup() or AddReplace() member of the AssociativeArray Object.

```
aa=CreateObject(“roAssociativeArray”)
aa.newkey=”the value”
print aa.newkey
```

The “.” Operator’s parameters are set at compile time – they are not dynamic (unlike the Lookup() or AddReplace() functons).

See the section on XML support for details on using the dot operator on xml objects.

Array/Function Call Operator

The “[ ]” operator is used to access an Array (any Roku Object that has an “ifArray” interface, such as roArray). It can also be used to access an AssociativeArray.

The function call operator “( )” can be used to call a function. When used on a Function literal (or variable containing a function reference), it calls the Function.

Examples:

```
aa=CreateObject(“roAssociativeArray”)
aa[“newkey”]=”the value”
print aa[“newkey”]

array=CreateObject(“roArray”, 10, true)
array[2]=”two”
print array[2]

fivevar=five
print fivevar()

array[1]=fivevar
print array[1]()    ‘ print 5

function five() As Integer
   return 5
end function
```

The “[ ]” operator takes expressions that are evaluated at runtime and so is different that a “.” Operator in this way. The dot operator takes compile time identifiers.

Arrays in BrightScript are one dimension. Multi-dimension arrays are implemented as arrays of arrays. The “[ ]” operator will automatically map “multi-dimensionality”. IE, the following two expressions to fetch “item” are the same:

```
dim array[5,5,5]
item = array[1][2][3]
item = array[1,2,3]
```

(**NOTE: if a multi-dimension array grows beyond its hint size the new entries are not automatically set to roArray**)

= Operator

“=” is used for both assignment and comparison. Example:

```
a=5
If a=5 then print “a is 5”
```

BrightScript does not support the use of the “=”Assignment operator inside an expression (like C does). This is to eliminate the common class of bugs where a programmer meant “comparison”, not “assignment”.

When an assignment occurs, intrinsic types are copied, but Roku Objects are reference counted.

Roku Objects, Interfaces, and Language Integration

[note: the name of Roku Objects will change to BrightScript Components]

The Roku Object architecture and library are separate from BrightScript, but BrightScript requires them.

- All APIs exposed to BrightScript are exposed as Roku Objects. In other words, if a platform wants to expose APIs to be scripted, the platform must register a new Roku Object. The Roku Object will most likely be written in C or C++.
- BrightScript has language features that are designed to work with Roku Object Interfaces. These include: for each, print, the array operator, and intrinsic objects.
- Fundamental BrightScript building blocks are implemented as Roku Objects. For example: Lists, Vector Arrays, Associative Arrays, and Objects.

A Brief Summary of Roku Objects

Roku Objects are light weight components that are implemented in C (or a C compatible language such as C++). C++ templates exist to help C++ programmers implement the key C functions needed to implement a Roku Object.

Roku Objects can be used in BrightScript, and they can be used by a C compatible language.

Roku Objects are robust against version changes. In other words, scripts are generally backwards compatible with Objects that have undergone version improvements.

Roku Objects keep a reference count and delete themselves when the reference count goes to zero.

A key Roku Object concept is the Interface. The term Interface is used here as it is in Java or Microsoft COM. An interface is a known set of member functions that implement a set of logic. In some ways an Interface is like a virtual base class in C++. Any script or C-compatible program can use an object's interface without regard to what type of object it is a part of, as long as it is familiar with a particular interface.

For example, the standard BrightScript serial interface (RS-232) object implements three interfaces: "ifSerialControl", "ifStreamReceive", and "ifStreamSend". Since the BrightScript "print" statement sends its output to any object that has an "ifStreamSend" interface, it works with the serial object (and others).

BrightScript statements that work with Roku Object Interfaces

For each

The for-each statement works on any object that has an "ifEnum" interface. These include: Array, Associative Array, List, and Message Port.

Print
The print #*object*, "hello" format will print "into" any object that has an "ifStreamSend" interface. These include the TextField and SerialPort objects.

If the expression being printed evaluates to an object that has an "ifEnum" interface, print will print every item that can be enumerated.

In addition to printing the values of intrinsic types, "print" will also print any object that exposes one of these interfaces: ifString, ifInt, ifFloat..

Wait
The wait function will work on any object that has an "ifMessagePort" interface.

Array Operator –"[]"
The array operator works on any object that has an "ifArray" or "ifAssociativeArray" interface. This includes Array, Associative Array, and Lists.

Member access operator "."
The "." Operator works on any object that has an "ifAssociativeArray" interface (as well as on any Roku Object (when calling a member function)). It also has special meaning when used on roXMLElement or roXMLList.

Expression Parsing
Any expression that is expecting an Integer, Float, Double, Boolean or String, can take an object with the "ifInt", "ifFloat", "ifBoolean" or "ifString" interface.

Wrapper Objects and intrinsic type promotion

The intrinsic BrightScript types integer, float, double, string, invalid, boolean and function all have object equivalents. If one of these intrinsic types is passed to a function that expects an Object, the appropriate wrapper object will be created, assigned the correct value, and passed to the function. This is sometimes referred to as "autoboxing". This is how, for example, roArray can store integers and strings as well as objects.

Any expression that expects one of the above types will work with the corresponding wrapper object as well.

For example:

```
Dim array[10]
Array.push(5)
intobj = array.pop()
print intobj+2                ' prints 7
print intobj.GetInt()+2       ' prints 7
print type(intobj)            ' prints "roInt"
```

BrightScript XML Support

BrightScript supports XML via two Roku Objects, and some dedicated language features. The Roku Object roXMLElement provides support for parsing, generating, and containing XML. In addition, the roXMLList object is often used to hold lists of roXMLElement, and implements the BrightScript standard ifList interface as well as the ifXMLList interface. Language features are provided via the dot operator, and the @ operator.

Dot Operator on XML

1. When applied to an roXMLElement, the dot operator returns an roXMLList of children that match the dot operand. If no tags match, an empty list is returned
2. When applied to an roXMLList, the dot operator aggregates the results of performing the dot operator on each roXMLElement in the list.

Attribute Operator

The @ operator can be used on an roXMLElement to return a named attribute. When used on an roXMLList, the @ operator will return a value only if the list contains exactly one element.

For example, if the file “example.xml” contains the following:

```
<?xml version="1.0" encoding="utf-8" ?>
<rsp stat="ok">
  <photos page="1" pages="5" perpage="100" total="500">
    <photo id="3131875696" owner="21963906@N06" secret="f248c84625"
      server="3125" farm="4" title="VNY 16R" ispublic="1" isfriend="0"
      isfamily="0" />
    <photo id="3131137552" owner="8979045@N07" secret="b22cfde7c4"
      server="3078" farm="4" title="hoot" ispublic="1" isfriend="0"
      isfamily="0" />
    <photo id="3131040291" owner="27651538@N06" secret="ae25ff3942"
      server="3286" farm="4" title="172 • 365 :: Someone once told me..."
      ispublic="1" isfriend="0"
  </photos>
</rsp>
```

Then

```
rsp=CreateObject(“roXMLElement”)
rsp.Parse(ReadAsciiFile(“example.xml”))

? rsp.photos.photo
```

Will return an roXMLList with three entries.

```
? rsp.photos.photo[0]
```

Will return an roXMLElement reference to the first photo (id="**3131875696”).**

```
? rsp.photos
```
Will return an roXMLList reference containing the photos tag.

```
rsp.photos@perpage
```

Will return the string 100.

Use the Text() method to return an element's text.

For example, if the variable booklist contains this roXMLElement:

```
    <booklist>
       <book lang=eng>The Dawn of Man</book>
    </booklist>

Print booklist.book.gettext()
```

Will print "The Dawn of Man".

```
print booklist.book@lang
```

Will print "eng".

example flikr code clip

```
REM
REM Interestingness
REM pass an (optional) page of value 1 - 5 to get 100 photos
REM starting at 0/100/200/300/400
REM
REM returns a list of "Interestingness" photos with 100 entries
REM

Function GetInterestingnessPhotoList(http as Object, page=1 As Integer) As Object

       print "page=";page


http.SetUrl("http://api.flickr.com/services/rest/?method=flickr.interestingness
.getList&api_key=YOURKEYGOESHERE&page="+mid(stri(page),2))

    xml=http.GetToString()

    rsp=CreateObject("roXMLElement")
    if not rsp.Parse(xml) then stop

       return helperPhotoListFromXML(http, rsp.photos.photo)
'rsp.GetBody().Peek().GetBody())


End Function

Function helperPhotoListFromXML(http As Object, xmllist As Object,
          owner=invalid As dynamic) As Object

    photolist=CreateObject("roList")
    for each photo in xmllist
```

```
       photolist.Push(newPhotoFromXML(http, photo, owner))
    next
    return photolist

End Function

REM
REM newPhotoFromXML
REM
REM    Takes an roXMLElement Object that is an <photo> ... </photo>
REM    Returns an brs object of type Photo
REM       photo.GetTitle()
REM       photo.GetID()
REM       photo.GetURL()
REM       photo.GetOwner()
REM

Function newPhotoFromXML(http As Object, xml As Object, owner As dynamic) As Object
    photo = CreateObject("roAssociativeArray")
    photo.http=http
    photo.xml=xml
    photo.owner=owner
    photo.GetTitle=function():return m.xml@title:end function
    photo.GetID=function():return m.xml@id:end function
    photo.GetOwner=pGetOwner
    photo.GetURL=pGetURL
    return photo
End Function

Function pGetOwner() As String
       if m.owner<>invalid return m.owner
       return m.xml@owner
End Function

Function pGetURL() As String
       a=m.xml.GetAttributes()
       url="http://farm"+a.farm+".static.flickr.com/"+a.server+"/"+a.id+"_"+a.se
cret+".jpg"
       return url
End Function
```

Garbage Collection

BrightScript will automatically free strings when they are no longer used, and it will free objects when their reference count goes to zero. This is done at the time the object or string is no longer used; there is no background garbage collection task. This results in very predictable "garbage collection" -- there are no unexpected stalls in execution.

A "mark and sweep" garbage collection is run after a script executes, or can be manually forced to run via the debug console. There is currently no way to run it from a script, and it doesn't currently run in the background. Its purpose is to clean up objects that refer to themselves or have other circular references (which are not managed by the normal reference counting garbage collection).

```
i=roCreateObject("roInt")
j=I      ' reference incremented
i=invalid    ' reference decremented
j=0   ' roInt just free'd.
```

Events

Events in BrightScript center around an event loop and the “roMessagePort” Roku object. Any RokuObject can be posted to a message port. Typically these will be Objects that are designed to post events. For example, the “roTimer” class posts events of type “roTimerEvent”.

Example:

```
print "BrightSign Button-LED Test Running"
p =   CreateObject("roMessagePort")
gpio =  CreateObject("roGpioControlPort")
gpio.SetPort(p)

while true
   msg=wait(0, p)
   if type(msg)="roGpioButton" then
       butn = msg.GetInt()
       if butn <=5 then
           gpio.SetOutputState(butn+17,1)
           print "Button Pressed: ";butn
           sleep(500)
           gpio.SetOutputState(butn+17,0)
        end if
    end if

   REM ignore buttons pressed while flashing led above
   while p.GetMessage()<>invalid
   end while
end while
```

Note that the following two lines:

```
while true
   msg=wait(0, p)
```

Could be replaced with:

```
For each msg in p
```

And then the last “end while” would become a “next”.

Threading Model

A BrightScript script runs in a single thread. The general rule of thumb is that Roku Object calls are synchronous if they return quickly, or asynchronous if they take a long time to complete. For example, class roArray methods are all synchronous. But if "roVideoPlayer" is used to play a video, the play method returns immediately (it is asynchronous). As the video plays, it will post events to a script created message port. Typical events would be "media finished" or "frame x reached".

Whether a Roku Object launches a background thread to perform asynchronous operations is a decision made by the Object implementer. Often an object will have synchronous and asynchronous versions of the same method.

This threading model means that the script writer does not have to deal with mutexes and other synchronization objects. It is always single threaded and the message port is polled or waited upon to receive events into the thread. However, Roku Object implementers have to think about threading issues. For example, roList and roMessagePort are thread safe internally so that they can be used by multiple threads.

Scope

BrightScript uses the following scoping rules:

- Currently all related BrightScript code must reside in one file. This set of code is known as a “module”.
- BrightScript does not support global variables. Except, there is one hard-coded global variable “global” that is an interface to the global object. The global object contains all global library functions.
- Functions declared with the FUNCTION statement are at global scope, unless they are anonymous, in which case they are local scope.
- Local variables exist with function Scope. If a function calls another function, that new function has its own scope.
- Labels exist in function scope.
- Block Statements (like FOR-NEXT or WHILE-END WHILE) do not create a separate scope

Creating and Using Intrinsic Objects

In most of this manual we use the term “object” to refer to a “Roku Object”. These are C or C++ components with interfaces and member functions that BrightScript uses directly. Other than a few core objects that BrightScript relies upon (roArray, roAssociativeArray, roInt, etc.) Roku Objects are platform specific.

You can create “intrinsic” objects in BrightScript itself to use in your scripts. However, to be clear, these are not Roku Objects. There is currently no way to create a Roku Object in BrightScript, or to create intrinsic objects that have interfaces (they only contain member functions, properties, or other objects).

A BrightScript object is built upon an Associative Array. In fact, it is an roAssociativeArray. When a member function is called “from” an AssociativeArray, a “this” pointer is set. The “this” pointer is “m”. “m” is accessible inside the FUNCTION code to access object keys.

A “constructor” in BrightScript is a normal function at global scope that creates the AssociativeArray and fills in its member functions and properties. There is nothing “special” about it.

For an example, see the “snake” game at the end of this manual.

Program Statements

DIM name (dim1, dim2, …, dimK)

DIM ("dimension") is a statement that provides a short cut to creating roArray objects. It sets variable *name* to type "roArray", and creates Array's of Array's as needed for multi-dimensional arrays. The dimension passed to Dim is the index of the maximum entry to be allocated (the array initial size = dimension+1); the array will be resized larger automatically if needed.

```
Dim array[5]
```

Is the same as:

```
array=CreateObject("roArray",6,true)
```

```
Note that x[a,b] is the same as x[a][b]

Another example:

Dim c[5, 4, 6]

For x = 1 To 5
   For y = 1 To 4
        For z = 1 To 6
           c[x, y, z] = k
           k = k + 1
       Next
   Next
Next

k=0
For x = 1 To 5
    For y = 1 To 4
       For z = 1 To 6
          If c[x, y, z] <> k Then print"error" : Stop
          if c[x][y][z] <> k then print "error":stop
          k = k + 1
          Next
   Next
Next
```

variable = expression

Assigns a variable to a new value.

Examples:

```
a$="a rose is a rose"
b1=1.23
x=x-z1
```

In each case, the variable on the left side of the equals sign is assigned the value of the constant or expression on the right side.

END

Terminates execution normally.

STOP

Interrupts execution return a STOP error. Invokes the debugger. Use “cont” at the debug prompt to continue execution, or “step” to single step.

GOTO label

Transfers program control to the specified line number. GOTO *label* results in an branch. A label is an identifier terminated with a colon, on a line by itself. Example:

```
mylabel:
print “Anthony was here!”
goto mylabel
```

RETURN expression

Used to return from a function back to the caller. If the function is not of type Void, return can return a value to the caller.

FOR counter = exp TO exp STEP exp NEXT counter

Creates an iterative (repetitive) loop so that a sequence of program statements may be executed over and over a specified number of times. The general form is (brackets indicate optional material):

FOR *counter-variable* = *initial value* TO *final value* [STEP *increment*]
[*program statements*]
NEXT [*counter-variable*]

In the FOR statement, *initial value, final value* and *increment* can be any expression. The first time the FOR statement is executed, these three are evaluated and the values are saved; if the variables are changed by the loop, it will have no effect on the loop's operation. However, the counter variable must not be changed or the loop will not operate normally. The first time the FOR statement is executed the counter is set to the "initial value” and to the type of “initial value”.

At the top of the loop, the counter is compared with the *final value* specified in the FOR statement. If the counter is greater than the *final value*, the loop is completed and execution

continues with the statement following the NEXT statement. (If *increment* was a negative number, loop ends when counter is less than *final value*.) If the counter has not yet exceeded the *final value*, control passes to the first statement after the FOR statement.

When program flow reaches the NEXT statement, the counter is incremented by the amount specified in the STEP *increment*. (If the *increment* has a negative value, then the counter is actually decremented.) If STEP *increment* is not used, an increment of 1 is assumed.

.Example:

```
for i=10 to 1 step -1
   print i
next
```

Note that each NEXT statement optionally specifies the appropriate counter variable; however, this is just a programmer's convenience to help keep track of the nesting order. The counter variable may be omitted from the NEXT statements. But if you do use the counter variables, you must use them in the right order; i.e., the counter variable for the innermost loop must come first.

The counter variable must be “simple”; eg, not an array.

“EXIT FOR” is used to exit a FOR block prematurely.

FOR EACH item IN object

The FOR EACH statement iterates through each item in any object that has an “ifEnum” interface (enumerator). The For block is terminated with a NEXT statement. The variable *item* is set at the top of the loop to the next item in the object. Objects that are intrinsically ordered (like a List) are enumerated in order. Objects that have no intrinsic order (like AssociativeArray) are enumerated in apparent random order. It is okay to delete entries as you enumerate them.

“EXIT FOR” is used to exit a FOR block prematurely.

The following example objects can be enumerated: roList, roArray, roAsscoiativeArray, roMessagePort.

Example:
NOTE: this example does not yet work because literal arrays are not yet implemented

```
aa={joe: 10, fred: 11, sue:9}
For each n in ar
   Print n;aa(n)
   aa.delete(n)
next
```

While expression / Exit While

The While loop executes until expression is false. The “exit while” statement can be used to terminate a while loop prematurely.

Example:

```
k=0
while k<>0
   k=1
   Print “loop once”.
end while


while true
   Print “loop once”
   Exit while
End while
```

REM

Instructs the compiler to ignore the rest of the program line. This allows you to insert comments (REMarks) into your program for documentation. An ‘ (apostrophe) may be used instead of REM.

Examples Program:

```
rem ** this remark introduces the program **
'this too is a remark
```

IF expression THEN statements [ELSE statements]

There are two forms of the IF THEN ELSE statement. The single line form (this one), and the multi-line or block form (see next section). The IF instructs the Interpreter to test the following *expression*. If the *expression* is true, control will proceed to the statements immediately following the expression. If the expression is False, control will jump to the matching ELSE statement (if there is one) or down to the next program line.

Examples:

```
if x>127 then print "out of range" : end
if caveman=”fred” then print “flintsone” else print “rubble”
```

NOTE: THEN is optional in the above and similar statements. However, THEN is sometimes required to eliminate an ambiguity. For example:

```
if y=m then m=o ‘won't work without THEN.
```

BLOCK IF, ELSEIF, THEN, ENDIF

The multi-line or block form of IF THEN ELSE is more flexible. It has the form:

If BooleanExpression [Then]
[Block]
[ElseIfStatement+]
[ElseStatement]
End If

ElseIfStatement ::=
ElseIf BooleanExpression [Then]
[Block]

ElseStatement ::=
Else
[Block]

For example:

```
    vp_msg_loop:
        msg=wait(tiut, p)
        if type(msg)="rovideoevent" then

            if debug then print "video event";msg.getint()
            if lm=0 and msg.getint() = meden then
                if debug then print "videofinished"
                retcode=5
                return
            endif
        else if type(msg)="rogpiobutton" then
            if debug then print "button press";msg
            if esc0 and msg=b0 then retcode=1:return
            if esc1 and msg=b1 then retcode=2:return
            if esc2 and msg=b2 then retcode=3:return
            if esc3 and msg=b3 then retcode=4:return
        else if type(msg)=" Invalid" then
            if debug then print "timeout"
            retcode=6
            return
        endif

        goto vp_msg_loop
```

PRINT [#output_object], [@location], item list

Prints an item or a list of items on the console, or if *output_object* is specified, to an object that has an "ifStreamSend" interface. . The items may be either strings, number, variables, or expressions. Objects that have an ifInt, ifFloat, or ifString interface may also be printed.

The items to be PRINTed may be separated by commas or semi-colons. If commas are used, the cursor automatically advances to the next print zone before printing the next item. If semi-colons are used, no space is inserted between the items printed.

Positive numbers are printed with a leading blank (instead of a plus sign); all numbers are printed with a trailing blank; and no blanks are inserted before or after strings.

Examples:

```
x=5:print 25; "is equal to"; x ^2
run
 25 is equal to 25

a$="string"
print a$;a$,a$;" ";a$
run
stringstring string string

 print "zone 1","zone 2","zone 3","zone 4"
run
zone 1          zone 2          zone 3          zone 4
```

each print zone is 16 char wide. the cursor moves to the next print zone each time a comma is encountered.

```
 print "print statement #1 ";
 print "print statement #2"
run
print statement #1 print statement #2
```

Semi-colon's can be dropped in some cases. For example, this is legal:

```
      Print "this is a five "5"!!"
```

A trailing semi-colon over-rides the cursor-return so that the next PRINT begins where the last one left off .

If no trailing punctuation is used with PRINT, the cursor drops down to the beginning of the next line.

If the console you are printing to has the interface "ifTextField" then @ specifies exactly where printing is to begin. For example:

```
print #m.text_field,@width*(height/2-1)+(width-len(msg$))/2,msg$;
```

Whenever you PRINT @ on the bottom line of the Display, there is an automatic line-feed, causing everything displayed to move up one line. To suppress this, use a trailing semi-colon at the end of the statement.

TAB (expression)
Moves the cursor to the specified position on the current line (modulo the width of your console if you specify TAB positions greater than the console width). TAB may be used several times in a PRINT list.

Example:
print tab(5)"tabbed 5";tab(25)"tabbed 25"

No punctuation is required after a TAB modifier. Numerical expressions may be used to specify a TAB position. TAB cannot be used to move the cursor to the left. If the cursor is beyond the specified position, the TAB is ignored.

POS(x)
Returns a number from 0 to window width, indicating the current cursor position on the cursor. Requires a "dummy argument" (any numeric expression).

print tab(40) pos(0) ‘prints 40 at position 40

print "these" tab(pos(0)+5)"words" tab(pos(0)+5)"are";
print tab(pos(0)+5)"evenly" tab(pos(0)+5)"spaced"

Function([parameter As Type, …]) As Type / End Function

A function is declared using the function statement. In parentheses, one or more optional parameters to be passed may be declared. The return type of the function may also be declared. If the parameter or return type are not declared, they are assumed to be “dynamic”

Intrinsic types are passed by value (a copy is made). Objects are passed by reference.

Parameters can be of type:

- Integer
- Float
- Double
- String
- Object
- Dynamic

In addition to the above types, the return type can be:

- Void

Parameters can have default values and expressions.

For example:

```
Function cat(a, b)
        Return a+b    'a, b could be numbers or strings
End Function
```

```
Function five() As Integer
    Return 5
End function


Function add(a As Integer, b As Integer) As Integer
   Return a+b
End function

Function add2(a As Integer, b=5 as Integer) As Integer
      Return a+b
End Function

Function add3(a As Integer, b=a+5 as Integer) As Integer
      Return a+b
End Function
```

Functions have their own scope.

The statement "Sub" can be used instead of "function" as a shortcut to a function of Void return Type.

If a function is called from an associative array, then a local variable "m" is set to the AssociatiaveArray that the function is stored in.

For example:

```
sub main()
   obj={
       add: add
       a: 5
       b: 10
       }

   obj.add()
   print obj.result
end sub

function add() As void
    m.result=m.a+m.b
```

```
    end function
```

If a function is not called from an AssociatiaveArray, then its “m” is set to an AssocicateArray that is global to the module, and persists across calls.

Anonymous Functions

A function is anonymous if it does not have a name. It can be declared like this;

```
myfunc=function (a, b)
   Return a+b
end function

print myfunc(1,2)
```

They can be used with AA literals like this:

```
q = {

starring : function(o, e)
     str = e.GetBody()
     print "Starring: " + str
     toks = box(str).tokenize(",")
     for each act in tok
          actx = box(act).trim()
          if actx <> "" then
               print "Actor: [" + actx + "]"
                o.Actors.Push(actx)
          endif
     next
     return 0
     end function
}

q.starring(myobj, myxml)
```

Built-In Functions

BrightScript has a small number of built-in, module scope, intrinsic functions. They are the following.

Type(variable) As String

Returns the type of a variable and/or object. See the Roku Object specification for a list of types. For example:

```
      Print type (5)
```

Rnd(0) As Float
Rnd(range As Integer) As Integer

Generates a pseudo-random number using the current pseudo-random "seed number" (generated internally and not accessible to user). RND may be used to produce random numbers between 0 and 1, or random integers greater than 0, depending on the argument.

RND(0) returns a float value between 0 and 1.
RND(integer) returns an integer between 1 and *integer* inclusive . For example, RND(55) returns a pseudo-random integer greater than zero and less than 56.

Box(x as Dynamic) as Object

Box() will return an object version of an intrinsic type, or pass through an object if given one. For example, in places where you want to use string object functions on a string, you can:

```
str="   this is a string   "
print box(str).trim()
bo = box("string")
bo=box(bo)  ' no change to bo
print bo.md5()
```

Run(filename As String, Optional Args…) As dynamic

The run command will run a script from a script. Args may be passed to the scripts Main() function, and the called script may return arguments.

Example:

```
Sub Main()
      Run("test.brs")
      BreakIfRunError(LINE_NUM)
      Print Run("test2.brs", "arg 1", "arg 2")
      stop
End Sub

Sub BreakIfRunError(ln)
```

```
   el=GetLastRunCompileError()
   if el=invalid then
      el=GetLastRunRuntimeError()
      if el=&hFC or el=&hE2 then return
      'FC==ERR_NORMAL_END, E2=ERR_VALUE_RETURN
      print "Runtime Error (line ";ln;"): ";el
      stop
   else
      print "compile error (line ";ln;")"
      for each e in el
         for each i in e
            print i;": ";e[i]
         next
      next

      stop
   end if

End Sub
```

GetLastRunCompileError() As Object

Returns an roList of compile errors, or invalid if no errors. Each list entry is an roAssociativeArray with the keys: ERRNO, ERRLN, ERRSTR.

GetLastRunRuntimeError() As Integer

Returns an error code result after the last script Run().

These two are normal:

```
  &hFC==ERR_NORMAL_END
  &hE2==ERR_VALUE_RETURN
```

Global Functions

BrightScript has a set of standard, module scope, functions. These functions are stored in the global object. If the compiler sees a reference to one of the global functions, it directs the runtime to call the appropriate global object member.

Sleep(milliseconds As Integer) As Void

This function causes the script to pause for the specified time, without wasting CPU cycles. There are 1000 milliseconds in one second.
Example:

```
sleep(1000)   ' sleep for 1 second
sleep(200)    ' sleep 2/10 of a second
sleep(3000)  ' sleep three seconds
```

Wait (timeout As Integer, port As Object) As Object

This function waits on objects that are "waitable" (those that have a MessagePort interface). Wait() returns the event object that was posted to the message port. If timeout is zero, "wait" will wait for ever. Otherwise, Wait will return after timeout milliseconds if no messages are received. In this case, Wait returns a type "invalid".

Example:

```
p =   CreateObject("roMessagePort")
sw =  CreateObject("roGpioControlPort")
sw.SetPort(p)
msg=wait(0, p)
print type(msg)      ' should be roGpioButton
print msg.GetInt()   ' button number
```

CreateObject(name As String) As Object

Creates a Roku Object of class *name* specified. Return invalid if the object creation fails. Some Objects have optional parameters in their constructor that are passed after *name*.
Example:

```
sw = CreateObject("roGpioControlPort")
serial = CreateObject("roSerialPort", 0, 9600)
```

GetInterface(object As Object, ifname As String) As Interface

Each Roku Object has one or more interfaces. This function returns a value of type "Interface".

Note that generally Roku Objects allow you to skip the interface specification. In which case, the appropriate interface within the object is used. This works as long as the function names within the interfaces are unique.

UpTime(dummy As Integer) As Float

Returns the uptime of the system since the last reboot.

RebootSystem() As Void

Causes a soft reboot.

ListDir(path As String) As Object

Returns a List object containing the contents of the directory path specified. All files names are converted to all lowercase For example:

```
BrightScript> l=ListDir("/")
BrightScript> print l
test_movie_3.vob
test_movie_4.vob
test_movie_1.vob
test_movie_2.vob
```

ReadAsciiFile(filepath As String) As String

This function reads the specified file and returns it as a string. For example:

```
text=ReadAsciiFile(“/config.txt”)
```

WriteAsciiFile(filepath As String, buffer As String) As Boolean

This function reads the specified file and returns it as a string. For example:

```
WriteAsciiFile(“/config.txt”, “the text to write”)
```

CopyFile(source As String, destination As String) As Bool

Make a copy of a file.

MatchFiles(path As String, pattern_in As String) As Object

Search a directory for filenames that match a certain pattern. *Pattern* is a wildmat expression. Returns a List object.

This function checks all the files in the directory specified against the pattern specified and places any matches in the returned roList. The returned list contains only the part of the filename that is matched against the pattern not the full path.

The pattern may contain certain special characters:

A '?' matches any single character.

A '*' matches zero or more arbitrary characters.

The character class '[...]' matches any single character specified within the brackets. The closing bracket is treated as a member of the character class if it immediately follows the opening bracket. i.e. '[]]' matches a single close bracket. Within the class '-' can be used to specify a range unless it is the first or last character. e.g. '[A-Cf-h]' is equivalent to '[ABCfgh]'. A character class can be negated by specifying '^' as the first character. To match a literal '^' place it elsewhere within the class.

The characters '?', '*' and '[' lose their special meaning if preceded by a single '\'. A single '\' can be matched as '\\'.

Example:

```
l=MatchFiles(".", "*.mpg")
```

DeleteFile(file As String) As Boolean

Delete the specified file from the current directory.

DeleteDirectory(dir As String) As Boolean

It is only possible to delete an empty directory.

CreateDirectory(dir As String) As Boolean

Creates the specified Directory. Only one directory can be created at a time

FormatDrive(drive As String , fs_type As String) As Boolean

Formats a specified drive using the specified filesystem.

Global String Functions

UCase(s As String) As String

Converts the string to all upper case.

LCase(s As String) As String

Converts the string to all lower case.

Asc (letter As String) As Integer

Returns the ASCII code for the first character of the specified string. . A null-string argument will cause an error to occur. Example:

```
print asc("a")
```

Chr (ch As Integer) As String

Performs the inverse of the ASC function: returns a one-character string whose character has the specified ASCII, or control. Example:

```
print chr(35)  'prints a number-sign #
```

Using CHR, you can assign quote-marks (normally used as string-delimiters) to strings. The ASCII code for quotes - is 34. So A$=CHR(34) assigns the value " to A$.

Instr(position to start As Integer, text-to-search As String, substring-to-find As String) As Integer

Returns the position of a substring within a string. Returns 0 if the substring is not found. The first position is 1. For example:

```
print instr(1, "this is a test", "is")
```

will print 3

Left (s As String, n As Integer) As String

Returns the first *n* characters of *s*.

```
print left("timothy", 3)    ' displays tim
```

Len (s As String) As Integer

Returns the character length of the specified string. Example:

```
print len("timothy")   ' prints 7
```

Mid (s As String, p As Integer, [n As Integer]) As String

Returns a substring of *s* with length *n* and starting at position *p*. n may be omitted, in which case the string starting at p and ending at the end of the string is returned. The first character in the string is poison 1. Example:

```
print mid(“timothy”, 4,3)     ‘prints oth
```

Right (s As String, n As Integer) As String

Returns the last n characters of *string*. Example*:*

```
right$(st$,4) returns the last 4 characters of st$.
```

Str (value As Float) As String
Stri(value as Integer) As String

Converts a *value* to a string. STR$(A), for example, returns a string equal to the character representation of the value of A. For example, if A=58.5, then STR$(A) equals the string " 58.5". (Note that a leading blank is inserted before "58.5" to allow for the sign of A).

String (n As Integer, character As String) As String
Stringi (n As Integer, character As Integer) As String

Returns a string composed of n *character*-symbols. For example,

```
string(30,"*")
 returns "******************************"
```

Val (s As String) As Float

Performs the inverse of the STR function: returns the number represented by the characters in a string argument. For example, if A$="12" and B$="34" then VAL(A$+ "."+B$) returns the number 12.34.

Global Math Functions

The following math functions are part of global. Trig functions use or return radians, not degrees.

Abs (x As Float) As Float

Returns the absolute value of the argument.

Atn (x As Float) As Float

Returns the arctangent (in radians) of the argument; that is, ATN(X) returns "the angle whose tangent is X". To get arctangent in degrees, multiply ATN(X) by 57.29578.

Cos (x As Float) As Float

Returns the cosine of the argument (argument must be in radians). To obtain the cosine of X when X is in degrees, use CGS(X*.01745329).

Csng (x As Integer) As Float

Returns a single-precision float representation of the argument.

Cdbl(x As Integer) As Float

Also returns a single precision float representation of the argument. Someday may return double.

Exp (x As Float) As Float

Returns the "natural exponential" of X, that is, *ex.* This is the inverse of the LOG function, so X=EXP(LOG(X)).

Fix (x as Float) As Integer

Returns a truncated representation of the argument. All digits to the right of the decimal point are simply chopped off, so the resultant value is an integer. For non-negative X, FIX(X)=lNT(X). For negative values of X, FIX(X)=INT(X)+1. For example, FIX(2.2) returns 2, and FIX(-2.2) returns -2.

Int(x As Float) As Integer

Returns an integer representation of the argument, using the largest whole number that is not greater than the argument.. INT(2.5) returns 2; INT(-2.5) returns -3; and INT(1000101.23) returns 10000101.

Log(x As Float) As Float

Returns the natural logarithm of the argument, that is, *$\log_e$(argument). This* is the inverse of the EXP function, so X=LOG(EXP (X)). To find the logarithm of a number to another base b, use the formula $\log_b(X) = \log_e(X)/\log_e(b)$. For example, LOG(32767)/LOG(2) returns the logarithm to base 2 of 32767.

Sgn(x As Float) As Integer
Sgn(x As Integer) As Integer

The "sign" function: returns -1 for X negative, 0 for X zero, and +l for X positive.

Sin(x As Float) As Float

Returns the sine of the argument (argument must be in radians). To obtain the sine of X when X is in degrees, use SIN(X*.01745329).

Sqr(x As Float) As Float

Returns the square root of the argument. SQR(X) is the same as X^(1/2), only faster.

Tan(x As Float) As Float

Returns the tangent of the argument (argument must be in radians). To obtain the tangent of X when X is in degrees, use TAN(X*.01745329).

Core Roku Objects

The following set of core Roku Objects are used extensively by BrightScript, and are therefore incorporated into the language definition.

- roArray (interfaces: ifArray, ifEnum)
- roAssociativeArray (interfaces: ifAssociativeArray, ifEnum)
- roList (interfaces: ifList, ifArray, ifEnum) **NOTE: Will add ifArray**
- roMessagePort (interfaces: ifMessagePort, ifEnum)
- roGlobal (interfaces: ifGlobal; all static member functions)
- roInt (interfaces: ifInt)
- roFloat (interfaces: ifFloat)
- roString (interfaces: ifString, if StringOps)
- roBrSub (interfaces: ifBrSub) **NOTE: Will likely change to ro/ifFunction**
- roBoolean (interface: ifBoolean)
- roInvalid (no interfaces)
- roXMLElement (interfaces: ifXMLElement)
- roXMLList (interfaces: ifList, ifXMLList)

ifList

```
ResetIndex() As Boolean
AddTail(ro As Object) As Void
AddHead(ro As Object) As Void
RemoveIndex() As Object
GetIndex() As Object
RemoveTail() As Object
RemoveHead() As Object
GetTail() As Object
GetHead() As Object
Count() As Integer
Clear() As Void
```

ifEnum

```
Reset() As Void
Next() As Object
IsNext() As Boolean
IsEmpty() As Boolean
```

ifMessagePort

```
GetMessage () As Object
WaitMessage(timeout As Integer) As Object
PostMessage(msg As Object) As Void
```

roInt, roFloat, roString, roBoolean, roBrSub, roInvalid

The intrinsic types Integer (“Integer”), Float (“Float”), “rotBrSub”, Boolean (“Boolean”), Invalid (“Invalid”) and String “String” have an object and interface equivalents, with the following interfaces:

ifInt

```
GetInt() As Integer
SetInt(value As Integer) As Void
```

ifFloat

```
GetFloat() As Float
SetFloat(value As Float) As Void
```

ifString

```
GetString() As String
SetString(value As String) As Void
```

ifStringOps

```
SetString(s As String, strlen As Integer) As Void
AppendString(s As String, strlen As Integeter) As void
Len() As Integer
GetEntityEncode() As String
Tokenize(delim as String) As Object
Trim() As String
MD5() As String
```

ifBrSub

```
GetSub() As String
SetSub(value As BrSub) As Void
```

ifBoolean

```
GetBolean() As Boolean
SetBoolean(value As Boolean) As Void
```

. These are useful in the following situations:

- When an object is needed, instead of an intrinsic value. For example, “roList” maintains a list of objects. If an Integer is added to roList, for example, it will be automatically wrapped in an roInt by the language interpreter. When a function that expects a Roku Object as a parameter is passed an int, float, function, or string,

BrightScript automatically creates the equivalent Roku object.

- If any object exposes the ifInt, ifFloat, ifBoolean or ifString interfaces, that object can be used in any expression that expects an intrinsic value. For example, in this way an roTouchEvent can be used as an integer whose value is the userid of the roTouchEvent.

Notes:

- If o is an roInt, then the following statements have the following effects
  1. print o ‘ prints o.GetInt()
  2. i%=o ‘ assigns the integer i% the value of o.GetInt()
  3. k=o ‘presumably k is dynamic typed, so it becomes another reference to the roInt o
  4. o=5 ‘this is NOT the same as o.SetInt(5). Instead it releases o, and ‘changes the type of o to Integer (o is dynamically typed). And assigns it to 5.

Example:

```
BrightScript> o=CreateObject("roInt")
BrightScript> o.SetInt(555)
BrightScript> print o
 555
BrightScript> print o.GetInt()
 555
BrightScript> print o-55
 500
```

Example:

```
BrightScript> list=CreateObject("roList")
BrightScript> list.AddTail(5)
BrightScript> print type(list.GetTail())
  roInt
```

Note that an integer value of "5" is converted to type "roInt" automatically, because list.AddTail() expects an Roku Object as its parameter.

roAssociativeArray

An associative array (also knows as a map, dictionary or hash table) allows objects to be associated with string keys. The roAssociativeArray class implements the ifAssociativeArray and ifEnum interfaces.

This object is created with no parameters:

- `CreateObject("roAssociativeArray")`

The ifAssociativeArray interface provides:

- `AddReplace(key As String, value As Object) As Void`
  - Add a new entry to the array associating the supplied object with the supplied string. Only one object may be associated with a string so any existing object is discarded.
- `Lookup(key As String) As Object`
  - Look for an object in the array associated with the specified key. If there is no object associated with the key then type "invalid" is returned.
- `DoesExist(key As String) As Boolean`
  - Look for an object in the array associated with the specified key. If there is no associated object then false is returned. If there is such an object then true is returned.
- `Delete(key As String) As Boolean`
  - Look for an object in the array associated with the specified key. If there is such an object then it is deleted and true is returned. If not then false is returned.
- `Clear() As Void`
  - Remove all objects from the associative array.
- `SetModeCaseSensitive() As void`
  - Associative Array lookups are case insensitive by default. This call makes all subsequent actions case sensitive.

Example:

```
aa = CreateObject("roAssociativeArray")

aa.AddReplace("Bright", "Sign")
aa.AddReplace("TMOL", 42)

print aa.Lookup("TMOL")
print aa.Lookup("Bright")

Produces:

 42
Sign
```

roArray

An array stores objects in a continuous array of memory location. Since an roArray contains Roku Objects, and there are object wrappers for most intrinsic data types, each entry of an array can be a different type (or all of the same type).

The roArray class implements the ifArray and ifEnum interfaces.

This object is created with two parameters:

- `CreateObject("roArray", size As Integer, resize As Boolean)`
  - *Size* is the initial number of elements allocated for the array. If *resize* is true, the array will be resized larger if needed to accommodate more elements. If the array is large, this process might be slow.
  - The "dim" statement may be used instead of CreateObject to create a new array. Dim has the advantage in that it automatically creates arrays of arrays for multi-dimensional arrays.

The ifArray interface provides:

- `GetEntry(index As Integer) As Object`
  - Returns an Array entry of a given index. Entries start at zero. If an entry is fetched that has not been set, "invalid" is returned.
- `SetEntry(index As Integer, value As Object) As Void`
  - Sets an entry of a given index
- `Peek() As Object`
  - Returns the last (highest index) array entry without removing it.
- `Pop() As Object`
  - Returns the last (highest index) array entry and removes it from the array.
- `Push(value As Object) As Object`
  - Adds a new highest index entry into an array (adds to the end of the array)
- `Shift() As Object`
  - Removes index zero from the array and shifts every other entry down one. This is like a "pop" from the bottom of the array instead of the top.
- `Unshift(value As Object) As Integer`
  - Adds a new index zero to the array and shifts ever other index up one to accomidate. This is like a Push to the bottom of the array.
- `Delete(index as Integer) As Boolean`
  - Deletes the indicated array entry, and shifts down all entries above to fill the hole. The array length is decreased by one.
- `Count() As Integer`
  - Returns the index of highest entry in the array+1 (the length of the array).
- `Clear() As Void`
  - Deletes every entry in the array.
- `Append(As Object) As Void`
  - Appends one object to another. The two objects must be of the same type.

roXMLElement

Also see the section "BrightScript XML Support" for details of using the dot and @ shortcuts for many of these member functions.

roXMLElement is used to contain an XML tree. It has one interface: ifXMLElement

`GetBody() As Object`

```
GetAttributes() As Object
GetName() As String
GetText() As String
GetChildElements() As Object
GetNamedElements(As String) As Object
Parse(s As String) As Boolean
SetBody(As Object) As Void
AddBodyElement() As Oject
AddElement(As String) As Void
AddElementWithBody(As String, As Object) As Object
AddAttribute(As String, As String) As Void
SetName(As String) As Void
Parse(As String) As Boolean
GenXML(gen_header As String) As String
Clear() As Void
GenXMLHeader()
IsName(As String) Ad Boolean
HasAttribute(As String) As Boolean
```

Example:

```
<tag1>this is some text</tag1>
```

Would parse such that:

Name = tag1
Attributes = invalid
Body = roString containing “this is some text”

```
<emptytag caveman=”barney” />
```

Would parse such that:

Name= emptytag
Attributes = roAssociatveArray, with one entry {caveman, barney)
Body = invalid

If the tag contains other tags, body will by of type “roXMLList”.

To generate XML, create an roXMLElement, then use the “Set” functions to build it. Then call GenXML().

GenXML() takes one parameter (boolean) that indicates whether the generated xml should have the <?xml …> tag at the top.

AddBody() will create an roXMLList for body, if needed, then add the passed item (which should be an roXMLElement tag).

Example subroutine to print out the contents of an roXMLElement tree:

```
PrintXML(root, 0)


Sub PrintXML(element As Object, depth As Integer)
    print tab(depth*3);"Name: ";element.GetName()
    if not element.GetAttributes().IsEmpty() then
        print tab(depth*3);"Attributes: ";
        for each a in element.GetAttributes()
            print a;"=";left(element.GetAttributes()[a], 20);
            if element.GetAttributes().IsNext() then print ", ";
        next
        print
    end if

    if element.GetText()<>invalid then
        print tab(depth*3);"Contains Text: ";left(element.GetText(), 40)
    end if

    if element.GetChildElements()<>invalid
        print tab(depth*3);"Contains roXMLList:"
        for each e in element.GetChildElements()
            PrintXML(e, depth+1)
        next
    end if
    print
end sub
```

Example of generating XML:

```
    root.SetName("myroot")
    root.AddAttribute("key1","value1")
    root.AddAttribute("key2","value2")
    ne=root.AddBodyElement()
    ne.SetName("sub")
    ne.SetBody("this is the sub1 text")
    ne=root.AddBodyElement()
    ne.SetName("subelement2")
    ne.SetBody("more sub text")
    ne.AddAttribute("k","v")
    ne=root.AddElement("subelement3")
    ne.SetBody("more sub text 3")
    root.AddElementWithBody("sub","another sub (#4)")

    PrintXML(root, 0)
    print root.GenXML(false)
```

Another Example

```
    xml = CreateObject("roXMLElement")
    xml.SetName("root")
    subel1 = xml.AddBodyElement()
    subel1.SetName("subelement1")
    subel2 = xml.AddBodyElement()
    subel2.SetName("subelement2")
```

Is the same as:

```
xml = CreateObject("roXMLElement")
xml.SetName("root")
subel1 = xml.AddElement("subelement1")
subel2 = xml.AddElement("subelement2")
```

roXMLList

Interfaces:

- ifList (documented elsewhere)
- ifXMLList
  - GetAttributes() As Object
  - GetText() As String
  - GetChildElements() As Object
  - GetNamedElements(As String) As Object
  - Simplify() As Object

GetNamedElements() is used to return a new XMLList that contains all roXMLElements that matched the passed in name. This is the same as using the dot operator on an roXMLList.

Simplify will

- Otherwise, Return an roXMLElement if the list contains exactly one
- Otherwise, will return itself

GetAttributes() and GetText() are similar to calling xmllist.Simplify().GetText(), xmllist.Simplify().GetAttributes().

Appendix – BrightScript Debug Console

When a script is running, if a runtime error is encountered, or the "stop" statement is encountered, the BrightScript debug console is entered. Access to the console is device specific. For example, in WinBrightScript, it is a separate window. On BrightSigns, it is the main serial port (connect it to a PC with a null-modem cable and use a terminal program).

Use the "help" command to get a list of commands you can use. For example, "var" will list all in scope variables and their types and values. The Scope is set to the function that was running when the error was encountered.

One of the most powerful things you can do at the debug console is to type in any BrighScript statement. It will be compiled and execute in the current context.

Typically the default device (or window) for the “print” statement, and the debug console, are the same.

As of this writing, the following debugger commands are available:

```
bt          Print backtrace of call function context frames
bytecode    Show bytecode for this function
classes     List public classes
cont or c   Continue Script Execution
down or d   Move down the function context chain one
gc          Run garbage collector and show stats
exit        Exit debug shell
last        Show last line that executed
list        List current function's source
next        Show the next line to execute
objects     List all allocated Roku Object instances
stats       Show statistics
step or s   Step one program statement
up or u     Move up the function context chain one
var         Display local variables and their types/values
print or ?  Print variable value or expression
```

Appendix – Planned Improvements

- Iterators for reflection
- Ability to use interfaces with intrinsic objects
- Teleportation
- Switch statement and/or message port map
- Ability to split script into more than one file
- libraries

Appendix – BrightScript Versions

BrightScript Version Matrix

9-Jan-09

	HD20000 1.3 Branch	HD2000 2.0 Branch	Compact Main Line
SnapShot Date	1/7/2008	7/16/2008	1/9/2009
Defxxx, on, gosub, clear, random, data, read, restore, err, errl, let, clear, line numbers	**X**	**X**	
Intrinsic Arrays	**X**	**X**	
Compiler		**X**	**X**
AA & dot Op & m reference		**X**	**X**
Sub/Functions		**X**	**X**
ifEnum & For Each		**X**	**X**
For/Next Does Not Always Execute At Least Once		**X**	**X**
Exit For		**X**	**X**
Invalid Type. Errors that used to be Int Zero are now Invalid. Added roInvalid; Invalid Autoboxing			**X**
Array's use roArray; Added ifArray			**X**
Uninit Var Usage No Longer Allowed			**X**
Sub can have "As" (like Function)			**X**
roXML Element & XML Ops dot and @			**X**
Type() Change: Now matches declaration names (eg. Integer not roINT32)			**X**
Added roBoolean			**X**
Added dynamic Type; Type now optional on Sub/Functions			**X**
And/Or Don't Eval un-needed Terms			**X**
Sub/Fun Default Parameter Values e.g. Sub (x=5 As Integer)			**X**
AA declaration Op { }			**X**
Array Declaration Op [ ]			**X**
Change Array Op from ( ) to []			**X**
Anonymous Functions			**X**
Added Circ. Ref. Garbage Collector			**X**
Add Eval(), Run(), and Box()			**X**

Appendix – Example Script - Snake

The following code will run on any BrightSign and uses GPIO buttons 1,2,3,4 for controls.

```
REM
REM The game of Snake
REM demonstrates BrightScript programming concepts
REM June 22, 2008

REM
REM Every BrightScript program must have a single Main()
REM

Sub Main()

    game_board=newGameBoard()

    While true
        game_board.SetSnake(newSnake(game_board.StartX(), game_board.StartY()))
        game_board.Draw()
        game_board.EventLoop()
        if game_board.GameOver() then ExitWhile
    End While
End Sub


REM ******************************************************
REM ******************************************************
REM ***************                  *********************
REM *************** GAME BOARD OBJECT *********************
REM ***************                  *********************
REM ******************************************************
REM ******************************************************


REM
REM An example BrightScript constructor.   "newGameBoard()" is regular Function of module scope
REM BrightScript Objects are "dynamic" and created at runtime.  They have no "class".
REM The object container is a Roku Object of type roAssocitiveArray (AA).
REM The AA is used to hold member data and member functions.
REM

Function newGameBoard() As Object
    game_board=CreateObject("roAssociativeArray")     ' Create a Roku Object of type/class roAssociativeArray
    game_board.Init=gbInit                            ' Add an entry to the AA of type roBrSub with value gbDraw
(a sub defined in this module)
    game_board.Draw=gbDraw
    game_board.SetSnake=gbSetSnake
    game_board.EventLoop=gbEventLoop
    game_board.GameOver=gbGameOver
    game_board.StartX=gbStartX
    game_board.StartY=gbStartY
    game_board.Init()                                    ' Call the Init member function (which is gbInit)

    return game_board

End Function


REM
REM gbInit() is a member function of the game_board BrightScript Object.
REM When it is called, the "this" pointer "m" is set to the appropriate instance by
REM the BrightScript bytecode interpreter
REM
Function gbInit() As Void
    REM
    REM button presses go to this message port
```

```
    REM
    m.buttons = CreateObject("roMessagePort")
    m.gpio =  CreateObject("roGpioControlPort")
    m.gpio.SetPort(m.buttons)

    REM
    REM determine optimal size and position for the snake gameboard
    REM
    CELLWID=16     ' each cell on game in pixels width
    CELLHI=16      ' each cell in pix height
    MAXWIDE=30    ' max width (in cells) of game board
    MAXHI=30      ' max height (in cells) of game board
    vidmode=CreateObject("roVideoMode")
    w=cint(vidmode.GetResX()/CELLWID)
    if w>MAXWIDE then w = MAXWIDE
    h=cint(vidmode.GetResY()/CELLHI)
    if h>MAXHI then h=MAXHI

    xpix = cint((vidmode.GetResX() - w*CELLWID)/2)     ' center game board on screen
    ypix = cint((vidmode.GetResY() - h*CELLHI)/2)      ' center game board on screen

    REM
    REM Create Text Field with square char cell size
    REM
    meta=CreateObject("roAssociativeArray")
    meta.AddReplace("CharWidth",CELLWID)
    meta.AddReplace("CharHeight",CELLHI)
    meta.AddReplace("BackgroundColor",&H202020)   'very dark grey
    meta.AddReplace("TextColor",&H00FF00)   ' Green
    m.text_field=CreateObject("roTextField",xpix,ypix,w,h,meta)
    if type(m.text_field)<>"roTextField" then
        print "unable to create roTextField 1"
        stop
    endif
End Function

REM
REM As Object refers to type Roku Object
REM m the "this" pointer
REM
Sub gbSetSnake(snake As Object)
    m.snake=snake
End Sub


Function gbStartX() As Integer
    return cint(m.text_field.GetWidth()/2)
End Function


Function gbStartY() As Integer
    return cint(m.text_field.GetHeight()/2)
End Function


Function gbEventLoop() As Void

    tick_count=0

    while true
        msg=wait(250, m.buttons)   ' wait for a button, or 250ms (1/4 a second) timeout
        if type(msg)="roGpioButton" then
            if msg.GetInt()=1 m.snake.TurnNorth()
            if msg.GetInt()=2 m.snake.TurnSouth()
            if msg.GetInt()=3 m.snake.TurnEast()
            if msg.GetInt()=4 m.snake.TurnWest()
        else  'here if time out happened, move snake forward
            tick_count=tick_count+1
            if tick_count=6 then
                tick_count=0
                if m.snake.MakeLonger(m.text_field) then return
```

```
            else
                if m.snake.MoveForward(m.text_field) then return
            endif
        endif
    end while

End Function


Sub gbDraw()
    REM
    REM given a roTextField Object in "m.text_field", draw a box around its edge
    REM

    solid=191   ' use asc("*") if graphics not enabled
    m.text_field.Cls()

    for w=0 to m.text_field.GetWidth()-1
        print #m.text_field,@w,chr(solid);
        print #m.text_field,@m.text_field.GetWidth()*(m.text_field.GetHeight()-1)+w,chr(solid);
    next

    for h=1 to m.text_field.GetHeight()-2
        print #m.text_field,@h*m.text_field.GetWidth(),chr(solid);
        print #m.text_field,@h*m.text_field.GetWidth()+m.text_field.GetWidth()-1,chr(solid);
    next

    m.snake.Draw(m.text_field)

End Sub

Function gbGameOver() As Boolean
    msg$= " G A M E    O V E R "
    msg0$="                    "
    width = m.text_field.GetWidth()
    height = m.text_field.GetHeight()

    while true
        print #m.text_field,@width*(height/2-1)+(width-len(msg$))/2,msg$;
        sleep(300)
        print #m.text_field,@width*(height/2-1)+(width-len(msg$))/2,msg0$;
        sleep(150)
        REM  GetMessage returns the message object, or an int 0 if no message available
        If m.buttons.GetMessage() <> invalid Then Return False
    endwhile

End Function


REM ******************************************************
REM ******************************************************
REM ******************             ***********************
REM ****************** SNAKE OBJECT ***********************
REM ******************             ***********************
REM ******************************************************
REM ******************************************************

REM
REM construct a new snake BrightScript object
REM
Function newSnake(x As Integer, y As Integer) As Object

    ' Create AA Roku Object; the container for a "BrightScript Object"
    snake=CreateObject("roAssociativeArray")
    snake.Draw=snkDraw
    snake.TurnNorth=snkTurnNorth
    snake.TurnSouth=snkTurnSouth
    snake.TurnEast=snkTurnEast
    snake.TurnWest=snkTurnWest
    snake.MoveForward=snkMoveForward
    snake.MakeLonger=snkMakeLonger
```

```
    snake.AddSegment=snkAddSegment
    snake.EraseEndBit=snkEraseEndBit

    REM
    REM a "snake" is a list of line segments
    REM a line segment is an roAssociativeArray that conains a length and direction (given by the x,y delta
needed to move as it is drawn)
    REM

    snake.seg_list = CreateObject("roList")
    snake.AddSegment(1,0,3)

    REM
    REM The X,Y pos is the position of the head of the snake
    REM
    snake.snake_X=x
    snake.snake_Y=y
    snake.body=191   ' use asc("*") if graphics not enabled.
    snake.dx=1      ' default snake direction / move offset
    snake.dy=0      ' default snake direction / move offset

    return snake

End Function


Sub snkDraw(text_field As Object)
    x=m.snake_X
    y=m.snake_Y
    for each seg in m.seg_list
        xdelta=seg.xDelta
        ydelta=seg.yDelta
        for j=1 to seg.Len
            text_field.SetCursorPos(x, y)
            text_field.SendByte(m.body)
            x=x+xdelta
            y=y+ydelta
        next
    next
End Sub


Sub snkEraseEndBit(text_field As Object)
    x=m.snake_X
    y=m.snake_Y
    for each seg in m.seg_list
        x=x+seg.Len*seg.xDelta
        y=y+seg.Len*seg.yDelta
    next

    text_field.SetCursorPos(x, y)
    text_field.SendByte(32)    ' 32 is ascii space, could use asc(" ")

End Sub


Function snkMoveForward(text_field As Object)As Boolean
    m.EraseEndBit(text_field)
    tail=m.seg_list.GetTail()
    REM
    REM the following shows how you can use an AA's member functions to perform the same
    REM functions the BrightScript . operator does behind the scenes for you (when used on an AA).
    REM there is not point to this longer method other than illustration
    REM
    len=tail.Lookup("Len")          ' same as len = tail.Len (or tail.len, BrightScript syntax is not case
sensative)
    len = len-1
    if len=0 then
        m.seg_list.RemoveTail()
    else
        tail.AddReplace("Len",len)  ' same as tail.Len=len
```

```
    endif

    return m.MakeLonger(text_field)

End Function

Function snkMakeLonger(text_field As Object) As Boolean
    m.snake_X=m.snake_X+m.dx
    m.snake_Y=m.snake_Y+m.dy
    text_field.SetCursorPos(m.snake_X, m.snake_Y)
    if text_field.GetValue()=m.body then return true
    text_field.SendByte(m.body)
    head = m.seg_list.GetHead()
    head.Len=head.Len+1
    return false
End Function

Sub snkAddSegment(dx As Integer, dy As Integer, len as Integer)

    aa=CreateObject("roAssociativeArray")
    aa.AddReplace("xDelta",-dx)  ' line segments draw from head to tail
    aa.AddReplace("yDelta",-dy)
    aa.AddReplace("Len",len)
    m.seg_list.AddHead(aa)

End Sub


Sub snkTurnNorth()
    if m.dx<>0 or m.dy<>-1 then m.dx=0:m.dy=-1:m.AddSegment(m.dx, m.dy, 0)      'north
End Sub

Sub snkTurnSouth()
     if m.dx<>0 or m.dy<>1 then m.dx=0:m.dy=1:m.AddSegment(m.dx, m.dy, 0)       'south
End Sub

Sub snkTurnEast()
     if m.dx<>-1 or m.dy<>0 then m.dx=-1:m.dy=0:m.AddSegment(m.dx, m.dy, 0)     'east
End Sub

Sub snkTurnWest()
     if m.dx<>1 or m.dy<>0 then m.dx=1:m.dy=0:m.AddSegment(m.dx, m.dy, 0)       'west
End Sub
```

Reserved Words

INVALID	FOR	POS
AND	PRINT	LINE_NUM
OR	GOTO	REM
EACH	IF	RETURN
NEXT	NOT	STEP
DIM	THEN	STOP
ELSE	TO	TAB
END	TAB	OBJFUN
TYPE	RND	TRUE
FALSE	CREATEOBJECT	WHILE
ENDWHILE	EXITWHILE	ENDSUB
SUB	FUNCTION	EACH
EXIT	ENDFUNCTION	ENDIF