BrightSign®

# BRIGHTSCRIPT REFERENCE MANUAL

BrightScript version 3.0
Compatible with firmware versions 3.8.x and later

BrightSign, LLC. 16780 Lark Ave., Suite B Los Gatos, CA 95032 | 408-852-9263 | www.brightsign.biz

# TABLE OF CONTENTS

# INTRODUCTION

BrightScript is a powerful scripting language for building media and networked applications for embedded devices. This language features integrated support for a lightweight library of BrightScript objects, which are used to expose the API of the platform (device) that is running BrightScript. The BrightScript language connects generalized script functionality with underlying components for networking, media playback, UI screens, and interactive interfaces; BrightScript is optimized for generating user-friendly applications with minimal programmer effort.

This reference manual specifies the syntax of BrightScript. For a detailed listing of BrightScript objects, refer to the BrightScript Object Reference Manual. This manual is intended as a reference for those who have some level of programming experience—it is not a tutorial for those who are new to programming.

For a quick flavor of BrightScript code, see Appendix C for a game of "snake" created using BrightScript.

## BrightScript Characteristics

The following are some general characteristics of BrightScript, as compared to other common scripting languages:

- BrightScript is not case sensitive.
- Statement syntax is similar to Python, Basic, Ruby, and Lua (and dissimilar to C).
- Like JavaScript and Lua, objects and named data-entry structures are associative arrays.
- BrightScript supports dynamic typing (like JavaScript) and declared types (like C and Java).
- Similar to .Net and Java, BrightScript uses "interfaces" and "components" (i.e. objects).

## BrightScript Operation

BrightScript code is compiled into bytecode that is run by an interpreter. The compilation step occurs every time a script is loaded and run. Similar to JavaScript, there is no separate compilation step that results in a saved binary file.

BrightScript and its component architecture are written in 100% C for speed, efficiency, and portability. Since many embedded processors do not have floating-point units, BrightScript makes extensive use of the "integer" type. Unlike some languages (including JavaScript), BrightScript only uses floating point numbers when necessary.

# VARIABLES, LITERALS, AND TYPES

## Identifiers

Identifiers are names of variables, functions, and labels. They also apply to BrightScript object methods (i.e. functions) and interfaces (which appear after a "." Dot Operator). Identifiers have the following rules:

- Must start with an alphabetic character (a-z).
- May consist of alphabetic characters, numbers, or the underscore symbol ("_").
- Are not case sensitive.
- May be of any length.
- May not be a *reserved word* (see the Appendix B for a list of reserved words).
- (**variables only**) May end with an optional type declaration ("`$`" for a string, "`%`" for an integer, "`!`" for a float, "`#`" for a double).

**Examples**

```
a
boy5
super_man$
42%
```

## Types

BrightScript supports both dynamic typing and declared types. This means that every value has a type determined at runtime, but variables can also be instructed to always contain a value of a specified type. If a value is assigned to a variable that has a specified type, the type of the value will be converted to the variable type if possible. If conversion is impossible, a runtime error will occur.

A variable that does not end in a type declaration may change its type dynamically. For example, the statement `a=4` will create an integer, while a following statement specifying that `a="hello"` will change the type of the variable `a` to a string.

BrightScript supports the following types:

- **Boolean**: True or False
- **Integer**: A 32-bit signed integer number
- **Float**: The smallest floating point number format supported by either the hardware or software
- **Double**: The largest floating point number format supported by either the hardware or software. Although Double is an intrinsically understood type, it is implemented internally with the *roIntrinsicDouble* object. As a general rule, this type is hidden from developers.
- **String**: A sequence of ASCII (not UTF-8) characters. BrightScript uses two intrinsic string states:
    - **Constant strings**: A statement such as `s="astring"` will create an intrinsic constant string.
    - **roString instances**: Once a string is used in an expression, it becomes an *roString* instance. For example, the statement `s = s + "bstring"` will cause the intrinsic string `s` to convert to an *roString* instance. If this is followed by the statement `s2 = s`, the `s2` value will be a reference to `s`, not a copy of it. The behavior of reference counting strings is new to BrightScript version 3.0.
- **Object**: A reference to a BrightScript object (i.e. a native component). Note that the `type()` function will not return "Object" but the type of object instead (e.g. *roList*, *roVideoPlayer*). Also note that there is no separate type for intrinsic BrightScript Objects. All intrinsic BrightScript Objects are built on the *roAssociativeArray* object type.
- **Interface**: An interface in a BrightScript Object. If a "." Dot Operator is used on an interface type, the member must be static (since there is no object context).
- **Invalid**: A type that can have only one value: `Invalid`. This type is returned in various instances when no other type is valid (for example, when indexing an array that has never been sent).

**Example**

The following are examples of different types. The `?` statement is a shortcut for `print`, while the `type()` function returns a string that identifies the type of the passed expression.

```
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
```

## Type Declaration Characters

A type declaration 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, defining `a$` and `a%` would create two independent variables.

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

## Literals (Constants)

The following are valid literal types:

- Type Boolean: Either `True` or `False`
- Type Invalid: `Invalid` only
- Type String: A string in quotes (e.g. `"This is a string"`)
- Type Integer: An integer in hex (e.g. `HFF`) or decimal (e.g. `255`) format

- Type Float: A number with a decimal (e.g. `2.01`), in scientific notation (e.g. `1.23456E+30`), or with a Float type designator (e.g. `2!`)
- Type Double: A number in scientific notation containing a double-precision exponent symbol (e.g. `1.23456789D-12`) or with a Double type declaration (e.g. `2.3#`)
- Type Function: Similar to variable formatting (e.g. `MyFunction`)
- Type Integer: LINE_NUM – The current source line number

## Array Literals

The Array Operator (`[]`)can be used to declare an array. It can contain literals (constants) or expressions.

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

## Assoiative Array Literals

The Associative Array Operator (`{}`) can be used to define an associative array. It can contain literals (constants) or expressions.

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

Arrays and associative arrays can also be defined with the following format:

```
 aa = {
   Myfunc1: aFunction
   Myval1 : "the value"
}
```

## Invalid Object Return

Many methods (i.e. functions) that return objects can also return Invalid (for example, in cases where there is no object to return). In these cases, the variable accepting the result must be dynamically typed since it may be assigned either type.

Example: The following code will return a type mismatch: `a$` is a string that has a string type declaration, and thus it cannot contain Invalid.

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

## Dynamic Typing for Numbers

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

1. If a constant contains 10 or more digits, or if `D` is used in the exponent, the number is Double. Adding a `#` type declaration also forces a constant to be a Double.
2. If the number is not double precision and it contains a decimal point, the number is a Float. Expressing a number in scientific notation using the `E` exponent also forces a constant to be a Float.
3. If neither of the above conditions is true for a constant, the number is an Integer.

## Number Type Conversion

When operations are performed on one or two numbers, the result must be typed as an Integer, Float, or Double. When an addition (+), subtraction (-), or multiplication (*) operation is performed, the result will have the same degree of precision as the most precise operand: For example, multiplying an Integer by a Double will return a number that is a Double.

Only when both operands are Integers will the result be an Integer number. If the result of two Integer operands is outside the 32-bit range, the operation and return will be carried out with Doubles.

Division (/) operates using the same rules as above, except that it can never be carried out at the Integer level: When both operators are Integers, the operation and return will be carried out with Floats.

Comparison operations (e.g. <, >, =) will convert the numbers to the same type before they are compared. The less precise type will always be converted to the more precise type.

## Type Conversion and Accuracy

When a Float or Double number is converted to the Integer type, it is *rounded down*: The largest integer that is not greater than the number is used. This also happens when the `INT` function is called on a number.

When a Double number is converted to the Float type, it is *4/5 rounded*: The least significant digit is rounded up if the fractional part is >=5 (otherwise, it is left unchanged).

When a Float number is converted to the Double type, only the seven most significant digits will be accurate.

# OPERATORS

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

| | Description | Symbol(s) |
|---|---|---|
| 1. | Function Calls or Parentheses | `()` |
| 2. | Array Operators | `.` , `[]` |
| 3. | Exponentiation | `^` |
| 4. | Negation | `-`, `+` |
| 5. | Multiplication, Division, Modulus | `*`, `/`, `MOD` |
| 6. | Addition, Subtraction | `+`, `-` |
| 7. | Comparison | `<`, `>`, `=` , `<>`, `<=`, `>=` |
| 8. | Logical Negation | `NOT` |
| 9. | Logical Conjunction | `AND` |
| 10. | Logical OR | `OR` |

### String Operators

The following operators work with strings: `<`, `>`, `=`, `<>`, `<=`, `>=`, `+`

### Function References

The `=` and `<>` operators work on variables that contain function references and function literals.

## Logical and Bitwise Operators

The `AND`, `OR`, and `NOT` operators are used for logical (Boolean) comparisons if the arguments for these operators are Boolean:

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

On the other hand, if the arguments for these operators are numeric, they will perform bitwise operations:

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

When the `AND` or `OR` operator is used for a logical operation, only the necessary amount of the expression is executed. For example, the first statement below will print "True", while the second statement will cause a runtime error (because "invalid" is not a valid operand for `OR`):

```
print true or invalid
print false or invalid
```

## Dot Operator

The "`.`" Dot Operator can be used on any BrightScript object. It also has special meaning when used on an *roAssociativeArray* object, as well as *roXMLElement* and *roXMLList* objects. When used on a BrightScript Object, it refers to an interface or method associated with that object. In the following example, `IfInt` refers to the interface and `SetInt()` refers to a method that is part of that interface:

```
i=CreateObject("roInt")
i.ifInt.SetInt(5)
i.SetInt(5)
```

Every object method is part of an interface. However, specifying the interface with the "." Dot Operator is optional. If the interface is omitted, as in the third line of the above example, each interface that is part of the object will be searched for the specified member. If there is a naming conflict (i.e. a method with the same name appears in two interfaces), then the interface should be specified.

**Associative Arrays**

When the "." Dot Operator is used on an Associative Array, it is the same as calling the `Lookup()` or `AddReplace()` methods, which are member functions of the *roAssociativeArray* object:

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

Note that the parameters of the "." Dot Operator are set at compile time; they are not dynamic, unlike the `Lookup()` and `AddReplace()` methods.

The "." Dot Operator is always case insensitive: For example, the following statement will create the entry "newkey" in the associative array:

```
aa.NewKey=55
```

## Array and Function-Call Operators

The `[]` operator is used to access an array (i.e. any BrightScript object that has an *ifArray* interface, such as *roArray* and *roList* objects). It can also be used to access an associative array. The `[]` operator takes expressions that are evaluated at runtime, while the "." Dot Operator takes identifiers at compile time.

The `()` operator can be used to call a function. When used on a function literal (or variable containing a function reference), that function will be called.

Example: This code snippet demstrates the use of both array and function-call operators.

```
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
```

**Array Dimensions**

Arrays in BrightScript are one dimensional. Multi-dimensional arrays are implemented as arrays of arrays. The `[]` operator will automatically map multi-dimensionality. For example, the following two fetching expressions are the same:

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

**Note**: *If a multi-dimensional array grows beyond its hint size, the new entries are not automatically set to* roArray.

## Equals Operator

The = operator is used for both assignment and comparison:

```
a=5
If a=5 then print "a is 5"
```

Unlike the C language, BrightScript does not support use of the = assignment operator inside an expression. This is meant to eliminate a common class of bugs caused by confusion between assignment and comparison.

When assignment occurs, intrinsic types are copied, while BrightScript Objects are reference counted.

# OBJECTS, INTERFACES, AND LANGUAGE INTEGRATION

## BrightScript Objects

Though BrightScript operates independently of its object architecture and library, they are both required for programming BrightScript applications. The API of a BrightSign platform is exposed to BrightScript as a library objects: Platforms must register a new BrightScript object to expose some part of its API.

BrightScript objects are written in C (or a compatible language such as C++), and are robust against version changes: Scripts are generally backwards compatible with objects that have undergone revisions.

BrightScript objects keep a reference count; they delete themselves when the reference count reaches zero.

## Wrapper Objects

All intrinsic BrightScript types (Boolean, Integer, Float, Double, String, and Invalid) 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 allows, for example, *roArray* objects to store values (e.g. 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: *roBoolean*, *roInt*, *roFloat*, *roDouble*, *roString*.

Example: The following examples illustrate how wrapper objects work.

```
Print 5.tostr()+"th"
Print "5".toint()+5

-5.tostr()    'This will cause an error. Instead, use the following:
```

```
(-5).tostr()

if type(5.tostr())<> "String" then stop
if (-5).tostr()<>"-5" then stop
if (1+2).tostr()<>"3" then stop
i=-55
if i.tostr()<>"-55" then stop
if 100%.tostr()<>"100" then stop
if (-100%).tostr()<>"-100" then stop
y%=10
if y%.tostr()<>"10" then stop

if "5".toint()<>5 or type("5".toint())<>"Integer" then stop
if "5".tofloat()<>5.0 or type("5".tofloat())<>"Float" then stop
fs="-1.1"
if fs.tofloat()<>-1.1 or fs.toint()<>-1 then stop

if "01234567".left(3)<>"012" then stop
if "01234567".right(4)<>"4567" then stop
if "01234567".mid(3)<>"34567" then stop
if "01234567".mid(3,1)<>"3" then stop
if "01234567".instr("56")<>5 then stop
if "01234567".instr(6,"56")<>-1 then stop
if "01234567".instr(0,"0")<>0 then stop
```

## Interfaces

Interfaces in BrightScript operate similarly to 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 similar to a virtual base class in C++; any script or program that is compatible with C can use an object interface without regards to the type of object it belongs to: For example, the *roSerialPort* object, which controls the standard serial interface (RS-232), implements three interfaces: *ifSerialControl*, *ifStreamReceive*, and *ifStreamSend*. Since the `print` statement sends its output to any object that has an *ifStreamSend* interface, it works with the *roSerialPort* object, as well as any other object with the *ifStreamSend* interface.

## Statement and Interface Integration

Some BrightScript statements have integrated functionality with interfaces. This section describes how to use statements with interfaces.

### PRINT

Using the `PRINT` statement in the following format will print into an object that has an *ifStreamSend* interface, including the *roTextField* and *roSerialPort* objects:

```
print #object, "string"
```

If the expression being printed evaluates to an object with an *ifEnum* interface, the `PRINT` statement will print every item that can be enumerated.

In addition to printing the values of intrinsic types, the `PRINT` statement can also be used to print any object that exposes one of the following interfaces: *ifString*, *ifInt*, *ifFloat*.

### WAIT

The `WAIT` statement can work in conjunction with any object that has an *ifMessagePort* interface.

### Expression Parsing

Any expression that expects a certain type of variable—including Integer, Float, Double, Boolean, or String—can accept an object with an interface equivalent of that type: *ifInt*, *ifFloat*, *ifDouble*, *ifBoolean*, *ifString*.

### Array Operator

The array "`[]`" operator works with any object that has an *ifArray* or *ifAssociativeArray* interface, including arrays, associative arrays, and lists.

### Member Access Operator

The member access operator (i.e. Dot Operator) works with any object that has an *ifAssociativeArray* interface. It also works with any object when used to call a member function (i.e. method). It also has special meaning when used on an *roXMLElement* or *roXMLList* object.

## XML Support in BrightScript

BrightScript provides XML support with two BrightScript objects and a set of dedicated language features:

### roXMLElement

This object provides support for parsing, generating, and containing XML.

### roXMLList

This object is used to contain a list of *roXMLElement* instances.

### Dot operator

The "." Dot Operator has the following features when used with XML objects:

- When used with an *roXMLElement* instance, the "." Dot Operator returns an *roXMLList* instance of the child tags that match the dot operand. If no tags match the operand, an empty list is returned.
- When applied to an *roXMLList* instance, the "." Dot Operator aggregates the results of performing the above operation on each *roXMLElement* in the list.

- When applied to XML, which is technically case sensitive, the "." Dot Operator is still case insensitive. If you wish to perform a case-sensitive XML operation, use the member functions of the *roXMLElement*/*roXMLList* objects.

**Attribute Operator**

The "@" Attribute Operator can be used with an *roXMLElement* instance to return a named attribute. Though XML is case sensitive, the Attribute Operator is always case insensitive. If the Attribute Operator is used with an *roXMLList* instance, it will only return a value if that list contains exactly one element.

**Examples**

```
<?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>
```

Given the XML in the above *example.xml* file, then the following code will return an *roXMLList* instance with three entries:

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

? rsp.photos.photo
```

The following will return an *roXMLElement* reference to the first photo (id="**3131875696**"):

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

The following will return an *roXMLList* reference containing the `<photos>` tag:

```
? rsp.photos
```

The following will return the string “100”:

```
rsp.photos@perpage
```

You can use the *roXMLElement.GetText()* method to return an element’s text: For example, if the variable `<booklist>` contains the element `<book lang=eng>The Dawn of Man</book>`, then the following code will print the string “The Dawn of Man”.

```
Print booklist.book.gettext()
```

Alternatively, using the Attribute Operator will print the string “eng”.

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

**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&a
pi_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))
    end for
    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.secret+".jpg
"
      return url
End Function
```

# GARBAGE COLLECTION

BrightScript automatically frees strings when they are no longer used, and it will free objects when their reference count goes to zero. This is carried out at the time the object or string is no longer used; there is no background garbage collection task. The result is a predictable garbage-collection process, with no unexpected stalls in execution.

Objects may enter a state of circular reference counting: Objects that reference each other will never reach a reference count of zero and will need to be freed manually using the `RunGarbageCollector()` method. This method is useful when destroying old presentation data structures and creating a new presentation.

# EVENTS

Events in BrightScript center around an event loop and the *roMessagePort* object. Most BrightScript objects can post to a message port in the form of an event object: For example, the *roTimer* object posts events of the type *roTimerEvent*.

Example: The following script sets the destination message port using the `SetPort()` method, waits for an event in the form of an *roGpioButton* object, and then processes the event.

```
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 these two lines,

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

Can be replaced using the following (and substituting `end while` with `end for`):

```
For each msg in p
```

# THREADING MODEL

BrightScript runs in a single thread. In general, BrightScript object calls are synchronous if they return quickly, and asynchronous if they take a substantial amount of time to complete. For example, methods belonging to the *roArray* object are all synchronous, while the `Play()` method that is part of the *roVideoPlayer* object will return immediately (it is asynchronous). As a video plays, the *roVideoPlayer* object will post messages to the message port, indicating such events as “media playback finished” or “frame x reached”.

The object implementer decides whether a BrightScript object should launch a background thread to perform a synchronous operation. Sometimes, an object will feature synchronous and asynchronous versions of the same method.

This threading model ensures that the script writer does not have to deal with mutexes and other synchronization objects. The script is always single threaded, and the message port is polled or waited on to receive events into the thread. On the other hand, those implementing BrightScript objects have to consider threading issues: For example, the *roList* and *roMessagePort* objects are thread-safe internally, allowing them to be used by multiple threads.

# SCOPE

BrightScript uses the following scoping rules:

- Global variables are not supported; however, there is a single hard-coded global variable (“global”) that is an interface to the global BrightScript component, which contains all global library functions.
- Functions declared with the `Function` statement are global in scope; however, if the function is anonymous, it will still be local in scope.
- Local variables exist within the function scope. If a function calls another function, that new function has its own scope.
- Labels exist within the function scope.
- Block statements such as `For / End For` and `While / End While` do not create a separate scope.

# INTRINSIC OBJECTS

In general, this manual uses the term “object” to refer to “BrightScript components”, which are C or C++ components with interfaces and member functions that BrightScript uses directly. With the exception of some core objects (*roArray*, *roAssociativeArray*, *roInt*, *roMessagePort*, etc.), BrightScript objects are platform specific.

You can create intrinsic objects in BrightScript, but these objects are not BrightScript components. There is currently no way to create a BrightScript component in BrightScript or to create intrinsic objects that have interfaces (intrinsic objects can only contain member functions, properties, and other objects).

A BrightScript object is simply an *roAssociativeArray*: When a member function is called from an associative array, a “this” pointer is set to “m”, and “m” is accessible inside the Function code to access object keys. A “constructor” in BrightScript is simply a normal function at a global scope that creates an *roAssociativeArray* instance and fills in its member functions and properties.

See the “snake” game in the appendix for examples of creating intrinsic objects.

# PROGRAM STATEMENTS

BrightScript supports the following statement types. The syntax of each statement is documented in more detail later in this chapter.

**Note**: *BrightScript is not case sensitive.*

- `Library`
- `Dim`
- `=` (assignment)
- `End`
- `Stop`
- `Goto`
- `Rem` (or `'`)
- `Print`
- `For / To / End For / Step / Exit For`
- `For Each / In / End For / Exit For`
- `While / End While / Exit While`
- `If / Then / Else If / Else / End If`
- `Function / End Function / As / Return`

**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
	end for

End Function
```

**Statement Syntax**

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

Example:

```
myname = "fred"
if myname="fred" then yourname = "barney":print yourname
```

## LIBRARY

`LIBRARY Filename.brs`

BrightScript 3.0 allows you to add your own BrightScript libraries (*.brs* files), which can then be utilized by your script. To include a library, use the `LIBRARY` statement in your script or at the BrightScript shell prompt. The `LIBRARY` statement(s) must occur at the beginning of a script, before any other statements, functions, operators, etc.

The system locates a library by searching the directory containing the current script, as well as the `SYS:/script-lib/` directory. Note that the `Run()` function does not currently change the path of a `LIBRARY` statement to that of the called script (i.e. the system will continue searching the directory of the caller script). On the other hand, running a script directly from the BrightSign shell does modify the library search path to that of the called script.

Example: The first statement will include a library in the same folder as the script, while the second will include a library in a sub-folder.

```
LIBRARY "myBSL1.brs"
LIBRARY "new_lib/myBSL2.brs"
```

Example: The following statement will include the *bslCore.brs* library, which has some useful BrightScript features, from the `SYS:/script-lib/` directory.

```
LIBRARY "v30/bslCore.brs"
```

## DIM

```
DIM Name (dim1, dim2, …, dimK)
```

The `DIM` ("dimension") statement provides a shortcut for creating *roArray* objects. It sets the variable `Name` to type "roArray". It can create arrays of arrays as needed for multi-dimensionality. The dimension passed to `DIM` is the index of the maximum entry to be allocated (i.e. the array initial size = dimension+1), though the array will be resized larger automatically if needed.

Example: The following two lines create identical arrays.

```
Dim array[5]
array = CreateObject("roArray", 6, true)
```

**Note**: *The expression* `x[a,b]` *is identical to* `x[a][b]`.

Example: The following script demonstrates useful operations on a `DIM` array.

```
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
       End for
   End for
End for

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
          End for
   End for
End for
```

## Assignment (“=”)

```
variable = expression
```

The assignment statement (“=”) assigns a variable to a new value.

Example: In each of the following lines, the variable on the left side of the equals operator is assigned the value of the constant or expression on the right side of the equals operator.

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

## END

The `END` statement terminates script execution normally.

## STOP

The `STOP` statement interrupts script execution, returns a “STOP” error, and invokes the debugger. Use the `cont` command at the debugger prompt to continue execution of the script or the `step` command to execute a single step in the script.

## GOTO

```
GOTO label
```

The `GOTO` statement transfers program control to the line number specified by `Label`. The `GOTO label` statement results in a branching operation. A label is an identifier terminated with a colon on a line that contains no other statements or expressions.

Example:

```
mylabel:
print "Hello World"
goto mylabel
```

## RETURN

`RETURN expression`

The `RETURN` statement returns from a function back to its caller. If the function is not type Void, `RETURN` can also return a value to the caller.

## REM

The `REM` statement instructs the compiler to ignore the remainder of the program line. This allows you to insert comments into your script for documentation. An apostrophe (`'`) may be used instead of `REM`.

Example:

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

## PRINT

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

The `PRINT` statement prints an item or list of items in the console. The item(s) may be strings, integers, floats, variables, or expressions. An object with an *ifInt*, *ifFloat*, or *ifString* interface may also be printed. If the `output_object` is specified, this statement will print to an object with an *ifStreamSend* interface.

If the statement is printing a list of items, the items must be separated with semicolons or commas. If semicolons are used, spaces are not inserted between printed items; if commas are used, the cursor will automatically advance to the next print zone before printing the next item.

Positive numbers are printed with a leading blank (without a plus sign). All numbers are printed with a trailing blank, and no blanks are inserted before or after strings.

Example:

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

Example:

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

Example: Each print zone is 16 characters wide. The cursor moves to the next print zone each time a comma is encountered.

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

Example:

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

Example: In some cases, semicolons can be dropped. The following statement is legal.

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

A trailing semicolon overrides the cursor-return so that the next `PRINT` statement begins where the last left off. If no trailing punctuation is used with a `PRINT` statement, the cursor drops to the beginning of the next line.

**[@location]**

If the console you are printing to has the *ifTextField* interface, you can use the `@` character to specify where printing will begin.

Example:

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

Whenever you use `PRINT @` on the bottom line of the display, an automatic line-feed causes all displayed lines to move up one line. To prevent this from happening, use a trailing semicolon at the end of the statement.

**TAB (expression)**

This statement moves the cursor to the specified position on the current line (modulo the width of the console if the `TAB` position is greater than the console width).

Example:

```
print tab(5)"tabbed 5";tab(25)"tabbed 25"
```

Note the following about the `TAB` statement:

- The `TAB` statement may be used several times in a `PRINT` list.
- No punctuation is required after a `TAB` statement.
- Numerical expressions may be used to specify a `TAB` position.
- The `TAB` statement cannot be used to move the cursor to the left.
- If the cursor is beyond the specified position, the `TAB` statement is ignored.

**POS(x)**

This statement returns an integer that indicates the current cursor position from 0 to the maximum width of the window. This statement requires a dummy argument in the form of any numeric expression.

Example:

```
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"
```

## FOR / END FOR

```
FOR counter_variable = initial_value TO final_value STEP increment / END FOR
```

The `FOR` statement creates an iterative loop that allows a sequence of program statements to be executed a specified number of times.

The `initial_value`, `final_value`, and `increment` can be any expression. The first time the `FOR` statement is executed, these three variables are evaluated and their values are saved; changing the variables during the loop will have no affect on the operation of the loop. 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 both the value and type of the `initial_value`.

At the beginning of each loop, the value of the `counter_variable` is compared with the `final_value`. If the value of the `counter_variable` is greater than the `final_value`, the loop will complete and execution will continue with the statement following the `END FOR` statement. If, on the other hand, the counter has not yet exceeded the `final_value`, control passes to the first statement after the `FOR` statement. If `increment` is a negative number, the loop will complete when the value of the `counter_variable` is less than the `final_value`.

When program flow reaches the `END FOR` statement, the counter is incremented by the specified `increment` amount (or decremented if `increment` is a negative value). If the `STEP [increment]` language is not included in the `FOR` statement, the `increment` defaults to 1.

Use `EXIT FOR` to exit a `FOR` block prematurely.

Example: The following script decrements `i` at the beginning of each loop until it is less than 1.

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

## FOR EACH IN / END FOR

`FOR EACH item IN object / END FOR`

The `FOR EACH` statement can iterate through a set of items in any object that has an *ifEnum* interface (i.e. an enumerator). The `FOR` block is terminated with the `END FOR` statement. Objects that are ordered intrinsically (such as *roList*) are enumerated in order, while objects that have no intrinsic order (such as *roAssociativeArray*) are enumerated in apparent random order. It is possible to delete entries as they are enumerated.

Use `EXIT FOR` to exit a `FOR` block prematurely.

The following objects can be enumerated: *roList*, *roArray*, *roAssociativeArray*, *roMessagePort*.

Example: The following script iterates over an associative array in random order, prints each key/value pair, then deletes it.

```
aa={joe: 10, fred: 11, sue:9}
```

```
For each n in aa
   Print n;aa[n]
   aa.delete[n]
end for
```

## WHILE / EXIT WHILE

`WHILE expression / EXIT WHILE`

A `WHILE` loop executes until the specified `expression` is false. Use the `EXIT WHILE` statement to exit a `WHILE` block prematurely.

Example:

```
k=0
while k<>0
   k=1
   Print "loop once"
end while


while true
   Print "loop once"
   Exit while
End while
```

## IF / THEN / ELSE

`IF expression THEN statements [ELSE statements]`

**Note**: *This is the single-line form of the `IF THEN ELSE` statement; see the next section for more details about the block form of the `IF THEN ELSE` statement.*

The `IF` statement 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 either the matching `ELSE` statement (if there is one) or to the next program line after the block.

Example:

```
if x>127 then print "out of range" : end
```

**Note**: *`THEN` is optional in the above and similar statements. However, `THEN` is sometimes required to eliminate ambiguity, as in the following example.*

Example:

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

## Block IF / ELSEIF / THEN / ENDIF

The block (i.e. multi-line) form of IF / THEN / ELSE has the following syntax:

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

ElseIfStatement ::=
```

```
   ElseIf BooleanExpression [ Then ]
   [ Block ]

ElseStatement ::=
   Else
   [ Block ]
```

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
```

## Function() As Type / End Function

`Function name(parameter As Type, …) As Type`

**Note**: *Each function has its own scope.*

A function is declared using the `Function()` statement. The parentheses may contain one or more optional parameters; parameters can also have default values and expressions.

The type of each parameter may be declared. The return type of the function may also be declared. If a parameter type or return type is not declared, it is Dynamic by default. Intrinsic types are passed by value (and a copy is made), while objects are passed by reference. The `Sub` statement can be used instead of `Function` as a shortcut for creating a function with return type Void.

A parameter can be one of the following types:

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

The function return can be one of the following types:

- Void

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

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
```

**“m” Identifier**

If a function is called from an associative array, then the local variable `m` is set to the associative array in which the function is stored. If the function is not called from an associative array, then its `m` variable is set to an associative array that is global to the module and persists across calls.

The `m` identifier should only be used for the purpose stated above: We do not recommend using `m` as a general-purpose identifier.

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
```

**Anonymous Functions**

A function without a name declaration is considered anonymous.

Example: The following is a simple anonymous function declaration.

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

print myfunc(1,2)
```

Example: Anonymous functions can also be used with associative-array literals.

```
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
end for
return 0
end function
}
```

```
q.starring(myobj, myxml)
```

# BUILT-IN FUNCTIONS

BrightScript features a set of built-in, module-scope, intrinsic functions. A number of file I/O, string, mathematics, and system functions are also available via the *roGlobal* object, which is documented in the Object Reference Manual.

## Type()

```
Type(a As Variable) As String
```

This function returns the type of the passed variable and/or object. See the Object Reference Manual for a list of available object types.

## GetGlobalAA()

```
GetGlobalAA() As Object
```

This function fetches the global associative array for the current script.

## Rnd()

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

If passed a positive, non-zero integer, this function returns a pseudo-random integer between 1 and the argument value. The range includes the argument value: For example, calling `Rnd(55)` will return a pseudo-random integer greater than 0 and less than 56.

If the argument is 0, this function returns a pseudo-random Float value between 0 and 1.

**Note**: *The `Rnd()` functions utilize a pseudo-random seed number that is generated internally and not accessible to the user.*

## Box()

```
Box(type As Dynamic) As Object:
```

This function returns an object version of the specified intrinsic type. Objects will be passed through.

Example:

```
b = box("string")
b = box(b)   ' b does not change
```

## Run()

```
Run(file_name As String, [optional_arg As Dynamic, …]) As Dynamic
Run(file_names As roArray, [optional_arg As Dynamic, …]) As Dynamic
```

This function runs one or more scripts from the current script. You may append optional arguments, which will be passed to the `Main()` function of the script(s). The called script may also return arguments to the caller script.

If a string file name is passed, the function will compile and run the corresponding file. If an array of files is passed, the function will compile each file, link them together, and run them.

Example:

```
Sub Main()
      Run("test.brs")
      BreakIfRunError(LINE_NUM)
```

```
	Print Run("test2.brs", "arg 1", "arg 2")
	if Run(["file1.brs","file2.brs"])<>4 then stop
	BreakIfRunError(LINE_NUM)
	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]
			end for
		end for

		stop
	end if
End Sub
```

## Eval()

```
Eval(code_snippet As String) As Dynamic
```

This function runs the passed code snippet in the context of the current function. The function compiles the snippet, then executes the byte-code. If the code compiles and runs successfully, it will return zero. If the code compiles successfully, but encounters a runtime error, it will return an integer indicating the error code (using the same codes as the `GetLastRunRuntimeError()` function). If compilation fails, it will return an *roList* object; the *roList* structure is identical to that of the `GetLastRunCompileError()` function.

The `Eval()` function can be useful in two cases:

- When you need to dynamically generate code at runtime.
- When you need to execute a statement that could result in a runtime error, but you don't want code execution to stop.

Example:

```
PRINT Eval("1/0") 'Returns a divide by zero error.
```

## GetLastRunCompileError()

```
GetLastRunCompileError() As roList
```

This function returns an *roList* object containing compile errors (or Invalid if no errors occurred). Each *roList* entry is an *roAssociativeArray* object containing the following keys:

- `ERRSTR`: The compile error type (as String)
- `FILESPEC`: The file URI of the script containing the error (as String)
- `ERRNO`: The error number (as Integer)
- `LINENO`: The line number where the error occurs (as Integer)

The following are possible ERRNO values:

| Error Code | | Description | Expanded Description |
|---|---|---|---|
| &hBF | 191 | ERR_NW | ENDWHILE statement occurs without WHILE statement. |
| &hBE | 190 | ERR_MISSING_ENDWHILE | WHILE statement occurs without ENDWHILE statement. |
| &hBC | 188 | ERR_MISSING_ENDIF | End of script reached without finding an ENDIF statement. |
| &hBB | 187 | ERR_NOLN | No line number found. |
| &hBA | 186 | ERR_LNSEQ | Line number sequence error. |
| &hB9 | 185 | ERR_LOADFILE | Error loading file. |
| &hB8 | 184 | ERR_NOMATCH | MATCH statement does not match. |
| &hB7 | 183 | ERR_UNEXPECTED_EOF | Unexpected end of string encountered during string compilation. |
| &hB6 | 182 | ERR_FOR_NEXT_MISMATCH | Variable on NEXT does not match FOR. |
| &hB5 | 181 | ERR_NO_BLOCK_END | |
| &hB4 | 180 | ERR_LABELTWICE | Label defined more than once. |
| &hB3 | 179 | ERR_UNTERMED_STRING | Literal string does not have end quote. |
| &hB2 | 178 | ERR_FUN_NOT_EXPECTED | |
| &hB1 | 177 | ERR_TOO_MANY_CONST | |
| &hB0 | 176 | ERR_TOO_MANY_VAR | |
| &hAF | 175 | ERR_EXIT_WHILE_NOT_IN_WHILE | |
| &hAE | 174 | ERR_INTERNAL_LIMIT_EXCEDED | |
| &hAD | 173 | ERR_SUB_DEFINED_TWICE | |
| &hAC | 172 | ERR_NOMAIN | |
| &hAB | 171 | ERR_FOREACH_INDEX_TM | |
| &hAA | 170 | ERR_RET_CANNOT_HAVE_VALUE | |
| &hA9 | 169 | ERR_RET_MUST_HAVE_VALUE | |
| &hA8 | 168 | ERR_FUN_MUST_HAVE_RET_TYPE | |
| &hA7 | 167 | ERR_INVALID_TYPE | |
| &hA6 | 166 | ERR_NOLONGER | Feature no longer supported. |

| | | | |
|---|---|---|---|
| &hA5 | 165 | ERR_EXIT_FOR_NOT_IN_FOR | |
| &hA4 | 164 | ERR_MISSING_INITILIZER | |
| &hA3 | 163 | ERR_IF_TOO_LARGE | |
| &hA2 | 162 | ERR_RO_NOT_FOUND | |
| &hA1 | 161 | ERR_TOO_MANY_LABELS | |
| &hA0 | 160 | ERR_VAR_CANNOT_BE_SUBNAME | |
| &h9F | 159 | ERR_INVALID_CONST_NAME | |
| &h9E | 158 | ERR_CONST_FOLDING | |

## GetLastRunRuntimeError()

```
GetLastRunRuntimeError() As Integer
```

This function returns the error code that resulted from the last `Run()` function.

These codes indicate a normal result:

| Error Code | | Description | Expanded Description |
|---|---|---|---|
| &hFF | 255 | ERR_OKAY | |
| &hFC | 252 | ERR_NORMAL_END | Execution ended normally, but with termination (e.g. END, shell "exit", window closed). |
| &hE2 | 226 | ERR_VALUE_RETURN | Return executed with value returned on the stack. |
| &hE0 | 224 | ERR_NO_VALUE_RETURN | Return executed without value returned on the stack. |

The following codes indicate runtime errors:

| Error Code | | Description | Expanded Description |
|---|---|---|---|
| &hFE | 254 | ERR_INTERNAL | Unexpected condition occurred. |
| &hFD | 253 | ERR_UNDEFINED_OPCD | Opcode could not be handled. |

| | | | |
|---|---|---|---|
| `&hFB` | 251 | ERR_UNDEFINED_OP | Expression operator could not be handled. |
| `&hFA` | 250 | ERR_MISSING_PARN | |
| `&hF9` | 249 | ERR_STACK_UNDER | No value to pop off the stack. |
| `&hF8` | 248 | ERR_BREAK | `scriptBreak()` function called. |
| `&hF7` | 247 | ERR_STOP | `STOP` statement executed. |
| `&hF6` | 246 | ERR_RO0 | bscNewComponent failed because object class not found. |
| `&hF5` | 245 | ERR_R01 | BrightScript member function call does not have right number of parameters. |
| `&hF4` | 244 | ERR_RO2 | BrightScript member function not found in object or interface. |
| `&hF3` | 243 | ERR_RO3 | BrightScript interface not a member of the object. |
| `&hF2` | 242 | ERR_TOO_MANY_PARAM | Too many function parameters to handle. |
| `&hF1` | 241 | ERR_WRONG_NUM_PARAM | Number of function parameters incorrect. |
| `&hF0` | 240 | ERR_RVIG | Function returns a value, but is ignored. |
| `&hEF` | 239 | ERR_NOTPRINTABLE | Value not printable. |
| `&hEE` | 238 | ERR_NOTWAITABLE | `WAIT` statement cannot be applied to object because object does not have an *roMessagePort* interface. |
| `&hED` | 237 | ERR_MUST_BE_STATIC | Interface calls from rotINTERFACE type must be static. |
| `&hEC` | 236 | ERR_RO4 | "." Dot Operator used on object that does not contain legal object or interface reference. |
| `&hEB` | 235 | ERR_NOTYPEOP | Operation attempted on two type-less operands. |
| `&hE9` | 233 | ERR_USE_OF_UNINIT_VAR | Uninitialized variable used illegally. |
| `&hE8` | 232 | ERR_TM2 | Non-numeric index applied to array. |
| `&hE7` | 231 | ERR_ARRAYNOTDIMMED | |
| `&hE6` | 230 | ERR_USE_OF_UNINIT_BRSUBREF | Reference to uninitialized SUB. |
| `&hE5` | 229 | ERR_MUST_HAVE_RETURN | |
| `&hE4` | 228 | ERR_INVALID_LVALUE | Left side of the expression is invalid. |
| `&hE3` | 227 | ERR_INVALID_NUM_ARRAY_IDX | Number of array indexes is invalid. |

| | | | |
|---|---|---|---|
| `&hE1` | 225 | ERR_UNICODE_NOT_SUPPORTED | |
| `&hE0` | 224 | ERR_NOTFUNOPABLE | |
| `&hDF` | 223 | ERR_STACK_OVERFLOW | |
| `&h20` | 32 | ERR_CN | Continue (`cont` or `c`) not allowed. |
| `&h1C` | 28 | ERR_STRINGTOLONG | |
| `&h1A` | 26 | ERR_OS | String space has run out. |
| `&h18` | 24 | ERR_TM | A Type Mismatch (string /number operation mismatch) has occurred. |
| `&h14` | 20 | ERR_DIV_ZERO | |
| `&h12` | 18 | ERR_DD | Attempted to re-dimension array. |
| `&h10` | 16 | ERR_BS | Array subscript out of bounds. |
| `&h0E` | 14 | ERR_MISSING_LN | |
| `&h0C` | 12 | ERR_OUTOFMEM | |
| `&h08` | 8 | ERR_FC | Invalid parameter passed to function/array (e.g. a negative matrix dim or square root). |
| `&h06` | 6 | ERR_OD | Out of data (READ). |
| `&h04` | 4 | ERR_RG | Return without Gosub. |
| `&h02` | 2 | ERR_SYNTAX | |
| `&h00` | 0 | ERR_NF | `Next` without `For`. |

# BRIGHTSCRIPT CORE LIBRARY EXTENSION

There are a number of built-in functions that are not part of the BrightScript Core Library. You can use the `LIBRARY` statement to include this subset of functions:

```
LIBRARY "v30/bslCore.brs"
```

- `bslBrightScriptErrorCodes() As roAssociativeArray`: Returns an associative array of name/value pairs corresponding to BrightScript error codes and their descriptions.
- `bslGeneralConstraints() As roAssociativeArray`: Returns an associative array of name/value pairs corresponding to system constants
- `bslUniversalControlEventCodes() As roAssociativeArray`: Returns an associative array of name/value pairs corresponding to the remote key code constraints.
- `AsciiToHex(ascii As String) As String`: Returns a hex-formatted version of the passed ASCII string.
- `HexToAscii(hex As String) As String`: Returns an ASCII-formatted version of the passed hex string.
- `HexToInteger(hex As String) As Integer`: Returns the integer value of the passed hex string.

# BRIGHTSCRIPT DEBUG CONSOLE

If, while a script is running, a runtime error occurs or a `STOP` statement is encountered, the BrightSign application will enter the BrightScript debug console. This console can be accessed from a terminal program using a null-modem cable connected to the RS-232, GPIO, or VGA port (depending on the player model). Networked players can also be accessed via Telnet or SSH.

The console scope is set to the function that was running when the runtime error or `STOP` statement occurred. While in the console, you can type in any BrightScript statement; it will then be compiled and executed in the current context.

Typically, the debug console is the default device for the `PRINT` statement.

## Console Commands

The following console commands are currently available:

| | |
|---|---|
| `bt` | Print a backtrace of call-function context frames. |
| `classes` | List all public classes. |
| `cont or c` | Continue script execution. |
| `counts` | List count of BrightScript Component instances. |
| `da` | Show disassembly and bytecode for this function. |
| `down or d` | Move one position down the function context chain. |
| `exit` | Exit the debug shell. |
| `gc` | Run the garbage collector and show collection statistics. |
| `hash` | Print the internal hash-table histograms. |
| `last` | Show the last line that executed. |
| `method <class>` | List methods provided by specified class. |
| `method <class>.<interface>` | List methods provided by the specified interface or class. |
| `list` | List the current source of the current function. |

| | |
|---|---|
| `ld` | Show line data (source records) |
| `next` | Show the next line to execute. |
| `bsc` | List all allocated BrightScript Component instances. |
| `stats` | Show statistics. |
| `step or s` | Step one program statement. |
| `t` | Step one statement and show each executed opcode. |
| `up or u` | Move one function up the context chain. |
| `var` | Display local variables and their types/values. |
| `print or p or ?` | Print variable value or expression. |

# APPENDIX A – BRIGHTSCRIPT VERSIONS

**BrightScript Version Matrix**

**January 9, 2009**

| | 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 B – RESERVED WORDS

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

*Although `m` is not strictly a reserved word, it should not be used as an identifier outside of its intended purpose.

# APPENDIX C – EXAMPLE SCRIPT

The following code uses GPIO buttons 1, 2, 3, 4 for controls. It will work on any BrightSign model that has a video output and a GPIO port.

```
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 BrightScript Component 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 BrightScript Component of type/class roAssociativeArray
    game_board.Init=gbInit                             ' Add an entry to the AA of type roFunction 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 BrightScript Component
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);
    end for

    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);
    end for

    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 BrightScript Component; 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
        end for
    end for
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
    end for

    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
```