Roku BrightScript Reference

Matches HD600 Software Version: 1.1.28
Matches HD2000 Software Version: 1.1.20

Palo Alto, CA USA

Table of Contents

Introduction

Roku BrightScript is a scripting language based on Basic. It is easy to learn, and when combined with Roku Objects, it allows developers to create the logic needed for interactive signs or kiosks. This document specifies the syntax and built-in functions of the scripting language. To write useful applications, you should also refer to the Roku Object reference.

This manual is designed for people that have some experience programming software. It is a reference guide, not a tutorial.

BrightScript is an interpreted language, and runs on the BrightSign. To write BrightScript you just need a text editor on your PC or Mac. You can create it using any text editor. I use the editor that comes with Microsoft Visual Studio, which conveniently does color coding for Basic. Once a script is created, you save it, copy it to a Compact Flash card, put it in BrightSign, turn on the power, and use a terminal program to enter the "script <filename>" command. See the BrightSign User Guide for more information on this.

Quick Example

This example will give you a quick flavor of the Roku BrightScript scripting language with a script that plays a video file.

```
REM Defaults to 640x480x60p VGA Out
video=CreateObject("roVideoPlayer")
p=CreateObject("roMessagePort")
ok=video.SetLoopMode(1)
ok=video.PlayFile("/autoplay.vob")
msg=wait(0, p)
```

Let's go through each line:

```
REM Defaults to 640x480x60p VGA Out
```
This line does nothing, it is a "remark". You use REM or ' (apostrophe) to indicate a remark or comment.

```
video=CreateObject("roVideoPlayer")
```
The CreateObject function creates a "Roku Object". See the BrightScript Object Reference for more information on this. The variable "video" now holds the reference to this object, and has the type "roVideoPlayer"

```
p=CreateObject("roMessagePort")
```
This creates an object of type roMessagePort that is used to receive events.

```
ok=video.SetLoopMode(1)
```
This calls the method SetLoopMode of the object roVideoPlayer. This causes the roVideoPlayer to always loop any video it plays back (forever). The "ok" variable now holds the result of this method call, which indicates if there was an error or not. There is almost never an error in this case and this script doesn't actually check the return code.

```
ok=video.PlayFile("/autoplay.vob")
```
Tell the video object to play the file "/autoplay.vob" (autoplay.vob in the root of the Compact Flash card).

```
msg=wait(0, p)
```
Wait for an event on the message port p. There will never be one, since the "SetPort" member was never called. See the discussion on the event architecture in the BrightScript Roku Objects reference manual for more info. The "0" in "wait" is that timeout value. 0 means never time out. So the net result of this "wait" is to wait forever, since it will never time out, and p never receives events.

Line Numbers and Statement Separator

BrightScript lines may have optional line numbers. For example, the following two examples of code are the same:

```
10 hello=-1
20 if hello=-1 then print “hello there!!”
30 goto 20  ‘ loop and print forever!

hello=-1
loop_here:
if hello=-1 then print “hello there!!”
goto loop_here ‘ loop and print forever!
```

In BrightScript, if you wish to put multiple statements on a single line, separate them with a : (colon).

```
For example:
Hello=”hello there!”:print hello
```

Expressions, Variables, and Constants

Expression Summary

An expression is a combination of variables, constants, and functions that can evaluate to a value. The value will also have a type.

For example:

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

BrightScript> ?5+1
 6

BrightScript> ? 2.1 - 2
 0.0999999

BrightScript> days=5
BrightScript> ?days+2
 7

BrightScript> ?"hello"+" "+"there"
hello there
```

Variable Names

Variable names

- must start with an alphabetic character (a – z)
- may consist of alphabetic characters, numbers, or the symbol “_” (underscore)
- may end with an optional type designator.
- are not case sensitive
- may be of any reasonable length
- may not use a “reserved word” as the name (see appendix for list of reserved words).

For example:

```
a
boy5
```

```
super_man$
```

Types

All variables and values have a "type". The type of a variable is stored along with its value. The following types are used by Roku BrightScript:

- rotINT32– 32 bit signed number
- rotFLOAT – 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.
- rotSTRING. – a sequence of ASCII characters. Currently strings are ASCII, not UTF-8.
- rotOBJECT– See the Roku Object section. 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.
- rotINTERFACE- See the Roku Object Section.
- typeOmatic – This means that the type is determined 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 typeOmatic. It will take on the type of the expression assigned to it, and may change its type. For example: a=4 creates a as int, then a = "hello", changes a to a string. NOTE: Arrays that are typeOmatic can not change their type once assigned an initial type. All array elements must be of the same type.

Variables without declaration characters are assumed to be typeOmatic; this assumption can be changed with DEFine statements.

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.

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

BrightScript> ?type(1)
rotINT32

BrightScript> ?type(1.0)
rotFLOAT

BrightScript> ?type("hello")
rotSTRING

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

BrightScript> ?type(1%)
rotINT32

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

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

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

BrightScript> d=1
BrightScript> ?type(d)
rotINT32

BrightScript> d=1.0
```

```
BrightScript> ?type(d)
rotFLOAT
```

Type Declaration Characters

Character	Type	Examples
$	String	A$, ZZ$
%	Integer	A1%, SUM%
!	Single-Precision (float)	B!, N1!
#	Double-Precision (double)	A#, 1/3#
D	Double-Precision (exponential notation)	1.23456789D-12
E	Single-Precision (exponential notation)	1.23456E+30

Operators

+	Add
-	Subtract
*	Multiply
/	Divide
^	exponentiation (e.g., 2^3 = 8)
+ concatenate (string together)	"2" + "2"= "22"
<	is less than
>	is greater than
=	is equal to
< = or = <	is less than or equal to
> = or = >	is greater than or equal to
< >	does not equal
NOT	Not equal
AND	Both must be true
OR	Either one must be true

Order of Operations

Operations in the innermost level of parentheses are performed first, then evaluation proceeds to the next level out, etc. Operations on the same nesting level are performed according to the following hierarchy:

Exponentiation: A ^ B Negation: –X *, / (left to right) +, – (left to right) <, > , =, <=, >=, <> (left to right) NOT AND OR .

( ) Parentheses
^ (Exponentiation)
– (Negation)
*, /
+, -
<, >, =
NOT
AND
OR

String Operators

Symbol	Meaning	Example
<	precedes alphabetically	"A" < "B"
>	follows alphabetically	"JOE" > "JIM"
=	equals	B$= "WIN"
<>	does not equal	IF A$<>B$ THEN PRINT A$
<=	precedes or equals	IF A$<=AZ$ PRINT "DONE"
>=	follows or equals	IF L1$>="SMITH" PRINT L1$
+	concatenate the two strings	A$ = C$+C1$ A$ = "TRS-"+"80"

Determining the type of a constant

The following rules determine how a constant and typeOmatic variables are typed:

I.	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.
II.	If the number is not double-precision, and if it contains a decimal point , then the number is float. If number is expressed in exponential notation with E preceding the exponent, the number is float
III.	If neither I nor II is true of the constant, then it is an integer.

Examples:

```
1.234567  - float
5  - int
1.0  - float
1.0#    - double
12345678900    - double
1%  - int
1!  - float
1#  - double
1E0 – float
1D0 – double
1.1% -- syntax error
```

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 single-precision, 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 in single precision float with a single-precision 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 integer form. The result of a logical operation is always an integer.

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.
Examples:

```
10 A!=1.3
20 A#=A!
30 PRINT A#

RUN
1.299999952316284

10 A#=2/3
20 PRINT A#
```

```
RUN
.6666666865348816
```

Assigning Double-Precision Values

Here are three ways to be sure double-precision values are stored without any trailing "garbage digits". The first two (lines 10 and 20) are for entering constants; the third (line 30) is for converting from single precision to double-precision values.

```
A#=0.1D0
B#=0.1000000
C#=VAL(STR$(0.1))
```

BrightScript Objects do not support double precision, and double precision runs slower than float (single precision). “integer” math runs the fastest.

Arrays

An array is simply an ordered list of values. In Roku BrightScript these values may be of any type, depending on how the array is defined or typed.

.

Arrays start at zero. For example, after DIM A(4), array A contains 5 elements: A(0), A(1), A(2), A(3), A(4).

The number of dimensions an array can have (and the size or depth of the array), is limited only by the amount of memory available.

For example:

```
DIM TWO-D(100,100)
FOR I=0 TO 100:FOR J=0 TO 100
TWO-D(I,J) = 500
NEXT:NEXT
```

String arrays can be used. For example, C$(X) would automatically be interpreted as a string array. And if you use DEFSTR A at the beginning of your program, any array whose name begins with A would also be a string array.

Strings

String constants are contained in quotes. For Example: “This is a string!”

String variables end with $, or are defined with the DEFSTR statement. For example:
A$=”fred”

Strings may be compared for equality or alphabetic precedence. When they are checked for equality, every character, including any leading or trailing blanks, must be the same or the test fails.

IF Z$="END" THEN end_now

Strings are compared character-for-character from left to right. Actually, the ASCII codes for the characters are compared, and the character with the lower code number is considered to precede the other character.

For example, the constant "A!" precedes the constant "A#", because "!" (ASCII code: decimal 33) precedes "#" (ASCII code: decimal 35). When strings of differing lengths are compared, the shorter string is precedent if its characters are the same as those in the longer string. For example, "A" precedes "A ".

Not including the built-in string functions, there is only one string operation - concatenation, represented by the plus symbol +.

Example Programs:

```
A$="A ROSE"
B$=" IS A ROSE"
C$=A$+B$+B$+B$+".'
PRINT C$
```

RUN
A ROSE IS A ROSE IS A ROSE IS A ROSE.

Logical Operators

We described how AND, OR and NOT can be used with relational expressions. For example,

100 IF A=C AND NOT(B>40) THEN 60 ELSE 50

AND, OR and NOT can also be used for bit manipulation, bitwise comparisons, and Boolean operations. AND, OR and NOT convert their arguments to 32-bit, signed two's-complement integers. They then perform the specified logical operation on them and return a result within the same range

The operations are performed in bitwise fashion; this means that each bit of the result is obtained by examining the bit in the same position for each argument.

```
63 AND 16 = 16
-1 OR -2 = -1
```

Hex and Octal

Hex and octal constants use the following prefixes:
&H (hex constants)
&O (octal constants)

Example:

```
PRINT &H0ABC
RUN
2748
```

Program Statements

Type Definition	Assignment & Allocation	Sequence of Execution	(Conditional Statements)
DEFINT	CLEAR	END	IF
DEFSNG	DIM	STOP	THEN
DEFDBL	LET	GOTO	ELSE
DEFSTR	DATA	GOSUB	
	READ	RETURN	
	RESTORE	ON ... GOTO	
		ON ... GOSUB	
		FOR-NEXT-STEP	
		ON ERROR GOTO	
		RESUME	
		REM	

DEFtype

- DEFINT *letter or range*
- DEFSNG *letter or range*
- DEFDBL *letter or range*
- DEFSTR *letter or range*

Examples:
```
DEFINT A,I,N
```

All variables beginning with A, I or N will be treated as integers. For example, Al, AA, 13 and NN will be integer variables. However, A1#, AA#, I3# would still be double precision variables, because of the type declaration characters, which always override DEF statements.

```
DEFINT I-N
```

Causes variables beginning with I, J, K, L, M or N to be treated as integer variables.
DEFINT may be placed anywhere in a program, but it is normally placed at the beginning of a program.

CLEAR

Deletes all variables values and types, and array dimensions. Clears any DEFtypes.

DIM name (dim1, dim2, …, dimK)

Lets you set the "depth" (number of elements allowed per dimension) of an array or list of arrays. If no DIM statement is used, a depth of 11 (subscripts 0-10) is allowed for each dimension of each array used.
Example:
```
DIM A(5),B(2,3),C$(20)
```

To re-dimension an array, you must first use a CLEAR statement

LET variable = expression

May be used when assigning values to variables. Roku BrightScript does not require LET with assignment statements, but you can use it if you wish.

Examples:
```
LET A$="A ROSE IS A ROSE"
LET B1=1.23
LET 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.

Example:
```
10 INPUT S1,S2
20 GOSUB 100
.
.
.
99 END
100 H=SQR(S1*S1+S2*S2)
110 RETURN
```

The END statement in line 99 prevents program control from "crashing" into the subroutine. Now line 100 can only be accessed by a branching statement such as 20 GOSUB 100.

STOP

Interrupts execution and prints a BREAK IN *line number* message. STOP is primarily a debugging aid.

GOTO line number or label or run-time variable

Transfers program control to the specified line number. GOTO *line number/label/run-time variable* results in an unconditional (or automatic) branch.

GOSUB line number or label or run-time variable

Transfers program control to the subroutine beginning at the specified line number and stores an address to RETURN to after the subroutine is complete. When the Interpreter encounters a RETURN statement in the subroutine, it will then return control to the statement, which follows GOSUB.

Example:
```
GOSUB ["king"]
a$="queen"
GOTO [a$]
PRINT "ERROR!":STOP
king:
  PRINT "king!"
```

```
  RETURN
PRINT "ERROR!":STOP
queen:
PRINT "queen!"
```

RETURN

Ends a subroutine and returns control to the statement immediately following the most recently executed GOSUB.

ON n GOTO line number or label, …, line number or label or run-time variable

This is a multi-way branching statement that is controlled by a test variable or expression. The general format for ON *n* GOTO is:

ON *expression* GOTO *1st line number, 2nd line number, …, Kth line number*

When ON … GOTO is executed, first the expression is evaluated and the integer portion … INT(expression) … is obtained. We'll refer to this integer portion as J. The Computer counts over to the Jth element in the line-number list, and then branches to the line number specified by that element. If there is no Jth element (that is, if J > K or J=0 in the general format above), then control passes to the next statement in the program. If the test expression or number is less than zero, or greater than 255, an error will occur. The line-number list may contain any number of items. For example:

```
ON MI GOTO 150, 160, 170, 150, 180
ON MI GOTO LABEL1, LABEL2, LABEL3, LABEL4, LABEL5
```

FOR counter = exp TO exp STEP exp NEXT counter

Opens 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):

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

In the FOR statement, *initial value, final value* and *increment* can be constants, variables or expressions. 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 FOR-NEXT-STEP loop works as follows: the first time the FOR statement is executed, the counter is set to the "initial value." Execution proceeds until a NEXT statement is encountered. At this point, 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.

Then 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.

Example Programs:

```
    FOR I=10 TO 1 STEP -1
    PRINT I;
    NEXT
```

RUN
10 9 8 7 6 5 4 3 2 1

```
    FOR I=1TO3
    PRINT"OUTER LOOP"
    FOR J=1 TO 2
    PRINT" INNER LOOP"
50  NEXT J
60  NEXT I
```

RUN

OUTER LOOP
INNER LOOP
INNER LOOP
OUTER LOOP
INNER LOOP
INNER LOOP
OUTER LOOP
INNER LOOP
INNER LOOP

Note that each NEXT statement 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. It is also advisable to specify the counter variable with NEXT statements when your program allows branching to program lines outside the FOR-NEXT loop. Another option with nested NEXT statements is to use a counter variable list.
Delete line 50 from the above program and change line 60:

60 NEXT J,I

ON ERROR GOTO line number

When the Interpreter encounters any kind of error in your program, it normally breaks out of execution and prints an error message. With ON ERROR GOTO, you can set up an error-trapping routine, which will allow your program to "recover" from an error and continue, without any break in execution. Normally you have a particular type of error in mind when you use the ON ERROR GOTO statement. For example, suppose your program performs some division operations and you have not ruled out the possibility of division by zero. You might want to write a routine to handle a division-by-zero error, and then use ON ERROR GOTO to branch to that routine when such an error occurs.

Example:

```
5 ON ERROR GOTO 100
10 C = 1/0
```

The error handling routine must be terminated by a RESUME statement. See RESUME.

Use ON ERROR GOTO 0 to deactivate the ON ERROR.

RESUME line number

Terminates an error handling routine by specifying where normal execution is to resume.

RESUME without a line number and RESUME 0 cause the Interpreter to return to the statement in which the error occurred.

RESUME followed by a line number causes the Interpreter to branch to the specified line number. RESUME NEXT causes the Computer to branch to the statement following the point at which the error occurred.

Sample Program with an Error Handling Routine

```
5 ON ERROR GOTO 100
10 INPUT"SEEKING SQUARE ROOT OF";X
20 PRINT SQR(X)
30 GOTO 10
100 PRINT "IMAGINARY ROOT:";SQR(-X);"*I"
110 RESUME 10
```

REM

Instructs the Interpreter 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 true/false expression THEN action-clause

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 logical or relational expression. If the expression is True, control will proceed to the "action" clause 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.

In numerical terms, if the expression has a non-zero value, it is always equivalent to a logical True. Examples:

```
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 an ambiguity. For example:

```
IF Y=M THEN M=O won't work without THEN.

INPUT A$: IF A$="YES" THEN yes_here
INPUT A$: IF A$="YES" GOTO yes_here
```

The two statements have the same effect.

```
IF A>0 AND B>0 PRINT "BOTH POSITIVE"
```

THEN statement or line number or line label

Initiates the "action clause" of an IF-THEN type statement. THEN is optional except when it is required to eliminate an ambiguity, as in IF A<0 THEN 100. THEN should also be used in IF-THEN-ELSE statements.

ELSE statement or line number or line label

Used after IF to specify an alternative action in case the IF test fails. (When no ELSE statement is used, control falls through to the next program line after a test fails.)

Examples:

```
INPUT A$: IF A$="YES" THEN 300 ELSE END
IF A<B PRINT"A<B"ELSE PRINT"B<=A"
IF 1=1 THEN LABEL
```

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)="rotint32" then
            if debug then print "timeout"
            retcode=6
            return
        endif

        goto vp_msg_loop
```

DATA item list

Lets you store data inside your program to be accessed by READ statements. The data items will be read sequentially, starting with the first item in the first DATA statement, and ending with the last item in the last DATA statement. Items in a DATA list may be string or numeric constants - no expressions are allowed. If your string values include leading blanks, colons or commas, you must enclose these values in quotes.
It is important that the data types in a DATA statement match up with the variable types in the corresponding READ statement (unless you use typeOmatic variables). DATA statements may appear anywhere it is convenient in a program. Generally, they are placed consecutively, but this is not required.

Examples:

```
READ N1$,N2$,N1,N2
DATA "SMITH, J.R.","WILSON, T.M."
DATA 150,175
```

READ item list

Instructs the Interpreter to read a value from a DATA statement and assign that value to the specified variable. The first time a READ is executed, the first value in the first DATA statement will be used; the second time, the second value in the DATA statement will be read. When all the items in the first DATA statement have been read, the next READ will use the first value in the second DATA statement; etc. (An Out-of-Data error occurs if there are more attempts to READ than there are DATA items.) The following program illustrates a common application for READ/DATA statements.

```
50 PRINT "NAME","AGE"
100 READ N$
110 IF N$="END" PRINT "END OF LIST":END
120 READ AGE
130 IF AGE < 18 PRINT N$,AGE
140 GOTO100
150 DATA "SMITH, JOHN",30,"ANDERSON,T.M.",20
160 DATA "JONES, BILL", 15,"DOE,SALLY",21
170 DATA "COLLINS,W.P.",17,END

RUN
NAME            AGE
JONES, BILL     15
COLLINS,W.P.    17
END OF LIST
```

The program locates and prints all the minors' names from the data supplied. Note the use of an END string to allow READing lists of unknown length.

The same rule regarding commas, colons and leading blanks applies to values input via DATA statements and INPUT # statements.

```
READ T$,N$,D$
PRINT T$;N$;D$
DATA "TOTAL IS: ","ONE THOUSAND,TWO HUNDRED "
DATA DOLLARS.
```

T$ requires quotes because of the colon; N$ requires quotes because of the comma.

RESTORE

Causes the next READ statement executed to start over with the first item in the first DATA statement. This lets your program re-use the same DATA lines.
Example:

```
READ X
RESTORE
READ Y
PRINT X,Y
DATA 50,60
```

RUN
50 50

Because of the RESTORE statement, the second READ statement starts over with the first DATA item.

Console Statements

The statements described in this section let you send and receive strings and character to consoles. Currently the only console is the BrightSign serial port Shell.

Statements covered in this section:

TTY Console
PRINT
@ (PRINT modifier)
TAB (PRINT modifier)
INPUT
LINEINPUT

PRINT item list

Prints an item or a list of items on the console. The items may be either string constants (character sequences enclosed in quotes), string variables, numeric constants (numbers), variables, or expressions involving all of the preceding items. 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

10 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.

```
10 PRINT "ZONE 1",,"ZONE 3"
RUN
ZONE 1                          ZONE 3
```

```
10 PRINT "PRINT STATEMENT #10 ";
20 PRINT "PRINT STATEMENT #20"
RUN
PRINT STATEMENT #10 PRINT STATEMENT #20
```

A trailing semi-colon over-rides the cursor-return so that the next PRINT begins where the last one left off (see line 10).

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 is “windowed” (not TTY), then @ Specifies exactly where printing is to begin. The @ modifier must follow PRINT immediately, and the location specified must be a number from 0 to 1023.

```
PRINT @ 550, "LOCATION 550"
```

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.

Example:

```
PRINT @ 1000, 1000;
```

PRINT 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.

INPUT item list

LINEINPUT string variable

Causes the Interpreter to stop execution until, you enter the specified number of values via the keyboard. The INPUT statement may specify a list of string or numeric variables to be input. The items in the list must be separated by commas.

```
INPUT X$, X1, Z$, Z1
```

This statement calls for you to input a string-literal, a number, another string literal, and another number, in that order. When the statement is encountered, the Interpreter will display a

```
?_
```

You may then enter the values all at once or one at a time. To enter values all at once, separate them by commas. (If your string literal includes leading blanks, colons, or commas, you must enclose the string in quotes.)

To input a string with no ? displayed, use LINEINPUT.
100 LINEINPUT A$

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"

Built-in Roku Object and Miscellaneous Functions

Roku BrightScript has integrated support for Roku Objects. These objects are how the Roku system exposes blocks of functionality to the scripting language. For example, there are objects for receiving events (aka messages), accessing consoles (such as the serial port), playing back video, playing audio, etc. See the separate Roku Object specification for the details.

The following built-in functions are for manipulating Roku Objects and Miscellaneous functions.

WAIT
LISTDIR
CREATEOBJECT
GETINTERFACE
OBJFUN
TYPE
SLEEP

wait (timeout, object)

This function waits on objects that are “waitable” (those that have a MessagePort interface). It returns the message object. 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 “rotINT32”.

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
```

ListDir(path)

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

```
BrightScript> l=ListDir("/")
BrightScript> for i=1 to l.Count():print l.RemoveHead():next
test_movie_3.vob
test_movie_4.vob
test_movie_1.vob
test_movie_2.vob
```

CreateObject(string)

Creates a roku object of the name specified by string. Example:

```
sw = CreateObject("roGpioControlPort")
```

GetInterface(object, ifname)

Each Roku Object has one or more interfaces. This function returns a value of type “rotINTERFACE”.

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.

ObjFun(interface, name, param1, …, paramN)

This function executes an object function. It is not normally used. Instead the shortcut “.” syntax is used.

```
ser = CreateObject(“roGpioControlPort”)
if = GetInterface(ser, “ifMessagePort”)
print ObjFun(if, “GetValue”)
```

Is the same as:

```
ser = CreateObject(“roSerialConsole”)
print ser.ifMessagePort.GetValue()
```

or simply:

```
print ser.GetValue()
```

Type(variable)

Returns the type of a variable and/or object. See the Roku Object specification for a list of types. To check the type of an array, use one of the array elements (they will all be the same type). For example: print type(q(0))

sleep(milliseconds)

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
```

Built-in String Functions

ASC	MID
CHR	RIGHT
INSTR	STR
INKEY	STRING
LEN	VAL
LEFT	READASCIIFILE

ASC (string)

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

```
100 PRINT ASC("A")
```

CHR (expression)

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, text-to-search, substring-to-find)

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 (string, n)

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

```
PRINT LEFT$(“Timothy”, 3)  ‘ displays Tim
```

LEN (string)

Returns the character length of the specified string. Example:

```
PRINT LEN(“Timothy”)  ‘ prints 7
```

MID (string, p, [n])

Returns a substring of *string* 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 (string, n)

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

RIGHT$(ST$,4) returns the last 4 characters of ST$.

ReadAsciiFile(filepath)

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

text=DoReadAsciiFile(“/config.txt”)

STR (expression)

Converts a numeric expression or constant 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, "character" or number)

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

STRING $(30,"*")
returns "******************************"

VAL (string)

Performs the inverse of the STR$ function: returns the number represented by the characters in a string argument. The numerical type of the result can be integer, float, or double precision, as determined by the rules for the typing of constants. For example, if A$="12" and B$="34" then VAL(A$+ "."+B$) returns the value 12.34.

```
20 INPUT "ENTER MESSAGE"; M$
30 FOR K=1 TO LEN(M$)
40 T$=MID(M$, K, 1 )
60 CD=ASC(T$)+5: IF CD>255 CD=CD-255
70 NU$=NU$ + CHR(CD)
80 NEXT
90 PRINT "THE CODED MESSAGE IS"
100 PRINT NU$
110 FOR K=1 TO LEN(NU$)
120 T$=MID(NU$, K, 1)
130 CD=ASC(T$)-5: IF CD < 0 CD=CD+255
```

```
140 OLDS=OLD$+CHR$(CD)
150 NEXT
160 PRINT "THE DECODED MESSAGE IS"
170 PRINT OLD$
```

Built-in Arithmetic Functions

BrightScript offers a wide variety of intrinsic ("built-in") functions for performing arithmetic and special operations.

All the common math functions use type rotFLOAT (not Double).

Trig functions use or return radians, not degrees.

For all the functions, the argument must be enclosed in parentheses. The argument may be a numeric variable, expression or constant.

Functions described in this section:

ABS	COS	INT	SGN	ERR
ATN	CSNG	LOG	SIN	ERL
CDBL	EXP	RANDOM	SQR	
	FIX	RND	TAN	

ABS (x)

Returns the absolute value of the argument.

ATN (x)

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. Returns a double-precision representation of the argument.

COS (x)

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)

Returns a single-precision float representation of the argument. When the argument is a double-precision value, it is returned as six significant digits with "4/5 rounding" in the least significant digit. So CSNG(.6666666666666667) is returned as .666667; CSNG(.3333333333333333) is returned as .333333.

EXP (x)

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

FIX (x)

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)

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)

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.

```
PRINT LOG(3.3*X)
```

RANDOM

is actually a complete statement rather than a function. It reseeds the random number generator. If a program uses the RND function, you may want to put RANDOM at the beginning of the program. This will ensure that you get an unpredictable sequence of pseudo-random numbers each time you run the program.

```
RANDOM
C=RND(6)
```

RND(x)

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 single-precision 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.

SGN(x)

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

SIN(x)

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)

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

TAN(x)

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).

ERL

Returns the line number in which an error has occurred. This function is primarily used inside an error-handling routine accessed by an ON ERROR GOTO statement. If no error has occurred when ERL is called, line number 0 is returned. However, if an error has occurred since power-up, ERL returns the line number in which the error occurred.

ERR

Similar to ERL, except ERR returns a value related to the code of the error rather than the line in which the error occurred. Commonly used inside an error handling routine accessed by an ON ERROR GOTO statement.

ERR/2+1 = true error code (true error code -1)*2=ERR

Example Program

```
10 ON ERROR GOTO 1000
20 DIM A(15): I=1
30 READ A(1)
40 I=I+1: GOTO 30
50 REM REST OF PROGRAM
100 DATA 2,3,5,7,1,13
999 END
1000 IF ERR/2+1=4 RESUME 50
1010 ON ERROR GOTO 0
```

Note line 1000: 4 is the error code for Out of Data.

Reserved Words

@	ERL	ERR
AND	FOR	POS
CLEAR	CLS	PRINT
CMD	GOSUB	RANDOM
CONT	GOTO	READ
DATA	IF	REM
DEFDBL	INPUT	RESET
DEFFN	INT	RESTORE
DEFINT	LET	RESUME
DEFSNG	LINE	RETURN
DEFUSR	NEXT	RND
DEFSTR	NOT	STEP
DIM	ON	STOP
ELSE	TO	TAB
END	USING	THEN