GALACTIC STUDIOS BASIC 1.25

User Manual

Bob Alexander

7/30/2020

Contents

1 INTRODUCTION

Galactic Studios BASIC (GSBASIC) is an interactive compiling interpreter for a dialect of BASIC. While it mostly imitates other BASICs, it includes some features inspired by C. It is intended for use in embedded systems. It can execute statements entered at a console or it can load BASIC files and execute them. It has built-in debugging features that allow a running program to be interrupted. The state of that program can then be examined and modified before resuming execution.

GSBASIC contains many built-in functions, and they are documented in section 6. However, when GSBASIC is embedded in a system, that system may provide additional built-in functions. Refer to the system's documentation for a list of its built-in functions.

This manual assumes that you are familiar with programming in a BASIC-like or C-like language.

Galactic Studios BASIC was written by me, Bob Alexander. http://GalacticStudios.org is a web site that describes some of my other projects.

The implementation and documentation for GSBASIC are Copyright 2018 by Robert E. Alexander.

1.1 Acknowledgements

Many thanks to Dave Lindbergh for reviewing this document and providing valuable feedback.

Thanks also to the members of the teams at Digital Equipment Corporation that developed RSTS/11 and BASIC-Plus.

2 PROGRAM ELEMENTS

2.1 Comments

Comments are preceded by a single quote ('), the REM statement, or the C-like double slash (//) and extend to the end of the line. Comments may appear after a statement on the same line.

```
REM This is a comment
' This is another comment
A = 3 ' This comment is on the same line as a statement
A = 4 // And a comment style for you C programmers
```

2.2 Statements

A GSBASIC program is composed of a series of statements. Each statement is terminated by a newline (carriage return or line feed). A statement can be continued onto additional lines by typing an underscore (_) as the last non-space character on the line.

Statement keywords are case-insensitive, so that PRINT, print, Print, and pRiNt are equivalent.

```
print "Hello, World!"   ' Legal statement

PRINT _
"Hello, World"          ' Multiline statement using underscore

Rem The following statement is illegal because the FOR statement
Rem must be on a new line
Print "Hello, World" For x = 1 to 3
```

2.3 Data Types

GSBASIC supports three data types: 32 bit integers, 32 bit floating point numbers, and strings. The maximum length of strings is 64KB. Strings contain single-byte characters; multi-byte encodings (e.g. UTF-8) are not supported.

2.4 Literals

GSBASIC recognizes integers, floating point ("float") numbers, and string literals.

An integer literal may be written in decimal or hexadecimal. Hexadecimal numbers are written as in C, e.g. 0x3, 0XAF, or 0x0b, or with a preceding dollar sign, e.g. $abcd or $1B. Decimal literals are written as you would expect.

Floating point numbers are written with an optional exponent. Examples of valid floats are 3.14159, 6.02e23, or 6.626E-34. Floats must start with a digit, so you cannot write .123, you must write 0.123.

String literals are written with double quotes, e.g. "Hello, world!". If you want to include a double quote or a control character in the string, use string concatenation with the CHR() function.

2.5 Variables

Variables may store integers, floats, strings, arrays, or handles to objects. They are variants, so even if a variable is initialized with a value of one type, it can be assigned a value of another type later. This even allows a variable that holds an integer to be assigned an array, at which point the

variable becomes an array as if it had been in a DIM statement. Conversely, an array defined in a DIM can be assigned to an integer, float or string, and it will no longer be an array.

```
DIM a[10]
a = 1 ' 'a' is no longer an array; it's an integer
DIM a[5, 3] ' it's an array again
```

Variable names must begin with a letter or underscore, and then may contain letters, digits or underscores. A variable name may be as up to 32 characters long. Variable names are case-sensitive.

Variables do not need to be declared before use, but they must be assigned a value before they are used in an expression, i.e. variables do not have a default initialization value.

When a variable holds a string value, array, or handle to an object, the variable is actually holding a reference to the string, array, or object. If you assign that variable to another variable, it does not copy the string, array, or object; it only copies the reference.

2.6 Predefined Constants

GSBASIC defines certain identifiers as predefined constants. Examples are PI (defined as 3.14159) and NIL (defined as an uninitialized value). The names of predefined constants are case-insensitive and are reserved words (i.e. you cannot use those names for your own variables, subroutines, or functions).

```
a = PI
PRINT a ' 3.14159 will be printed
a = nil
PRINT a ' This will cause the error "Variable is used before
being assigned a value"
```

2.7 Arrays

Arrays can be defined with as many as 3 dimensions. Each element of an array is variable, so it can contain an integer, float, string, struct, or even another array (an array element that holds an array is not equivalent to adding dimension(s) to the array). Each element in an array is allowed to hold a different data type.

By default, arrays are one-based, i.e. the first element in an array has an index of 1. The OPTION BASE statement allows you to change all arrays in your program to be zero-based.

Arrays are created with the DIM statement or with an array initializer. Array elements are accessed by enclosing a list of indexes in square brackets (note that this is different from other BASICs, which typically use parentheses). Indexes are integers; if a float is used as an index, it will be cast to integer (i.e. rounded down).

```
DIM a[10], b[6, 2]
a[1] = 1
a[10] = 1
PRINT a[1.9] ' The 1.9 will be rounded down to 1
b[3, 1] = 5 ' An example of multi-dimensional arrays
```

The largest valid index in any dimension of an array can be found using the UBound built-in function:

```
DIM a[2, 7, 11]
PRINT UBound(a) ' Prints 2
PRINT UBound(a, 1) ' Also prints 2
PRINT UBound(a, 2) ' Prints 7
PRINT UBound(a, 3) ' Prints 11
```

Note that UBound returns the largest valid index. If you have left the Option Base at its default value of 1, the UBound is the same and the number of elements in the dimension of the array. But if you have used OPTION BASE 0 to make arrays zero-based, the largest valid index is one less than the size of the array. E.g. if an array has three elements and the Option Base is 1, its largest valid index is 3. But if Option Base is 0, the array's largest valid index is 2.

The number of items in any dimension of an array can be found using the Size built-in function. This example shows the difference between UBound and Size when the Option base is 0:

```
OPTION BASE 0
DIM a[2, 7, 11]
PRINT UBound(a) ' Prints 1
PRINT UBound(a, 1) ' Also prints 1
PRINT UBound(a, 2) ' Prints 6
PRINT UBound(a, 3) ' Prints 10
PRINT Size(a) ' Prints 2
PRINT Size(a, 1) ' Also prints 2
PRINT Size(a, 2) ' Prints 7
PRINT Size(a, 3) ' Prints 11
```

A variable that is an array is actually a reference to an array. If you assign it to another variable, e.g.:

```
DIM a[3]
a[1] = 1
a[2] = 2
a[3] = 3
b = a
```

b is now referring to the same values that *a* is. If you change a value, e.g. a[1] = 5, b[1] will also equal 5.

When you pass an array to a function or subroutine, it is passed by reference. So if you pass *a* to a function and the function changes any of the values of the argument, *a*'s values will be changed when you return from the function.

```
SUB S(x)
   x[1] = 2
ENDSUB

DIM a[3]
```

```
a[1] = 100
CALL S(a)
PRINT a[1] ' Prints 2dim b
```

In short, if you're familiar with C, *a* is really a pointer to an array. Assigning *b* to *a* copies the pointer, not the contents of the array.

If you want to make a copy of an array, use the CopyArray function or the newer DeepCopy function. (CopyArray does a shallow copy, so that if an element of the array is a struct or another array, only the reference to that struct or array is copied. DeepCopy does a recursive copy, so that if an element of the array is a struct or another array, its data is copied.)

There's more you should know about the DIM statement, so read about it in section 4.5.

2.7.1 Array Initializers

Arrays can be initialized much as they are in C, with a list of values enclosed in curly braces. Elements in an array initializer can be literals, variables, or expressions.

Array initializers can be nested to make multi-dimensional arrays. The magnitudes of the dimensions are determined by the largest magnitude of any nested array initializer. Unspecified elements in the initializer are set to integer 0. If this is confusing, look at the following example.

```
c = {0, 1, 2} ' c is now an array with three initialized elements

' The following statement creates a two dimensional array with
' a size of [3, 2]. Notice that no DIM statement was used.
' After this statement is executed, d[1, 1] equals 3; d[1, 2]
' was unspecified in the initializer, so it is set to integer
' zero; d[2, 1] equals "Hello"; d[2, 2] equals "world"; d[3, 1]
' equals 1.5; and d[3, 2] is integer zero.
d = {{3}, {"Hello", "world"}, {1 + 0.5}}
```

As mentioned earlier, a single element of an array can hold an array. This can be specified in an array initializer by adding parentheses around a nested initializer. For example, the code:

```
a = {{11, 12, 13, 14, 15}, {21, 22, 23}}
```

creates a 2x4 array that looks like this:

11	12	13	14	15
21	22	23	0	0

Since the array is 2x4, but the second row had only three values in the initializer, the values in the rest of the second row are zero.

If we add parentheses to the initializer:

```
a = { ( {11, 12, 13, 14, 15} ), ( {21, 22, 23} ) }
```

we create a one dimensional array with two elements, each of which is an array:

{11, 12, 13, 14, 15}

{21, 22, 23}

In C, an array initializer is a mechanism for initializing arrays and nothing more. In GSBASIC, an array initializer is actually an expression, which means it can be used within a larger expression. For example, if there is a subroutine s that takes an array as an argument, you can write:

```
call s({1, 2, 3})
```

In short, an array initializer creates an array, and that array can then be assigned to a variable or used as an argument to a function or subroutine that takes an array as a parameter.

2.8 Structs

GSBASIC supports C-like structs. You define a structure with the Struct statement, giving the structure a name and specifying its member variables:

```
Struct ShipPos {x, y, z}
Struct ShipAngle {pitch, roll, yaw}
Struct Ship {name, pos, angle}
```

You can then create an instance of the struct using its name. Optionally, you can initialize the members of the new structure:

```
' Creates a Ship structure where all the members are
' uninitialized
myShip = Ship()

' Creates a Ship structure and initializes its first member.
' The remaining members are set to zero
kirksShip = Ship("Enterprise")

' Creates a Ship structure where all the members are set.
' Notice that we're creating structures for some of  the
' members
khansShip = Ship("Botany Bay", ShipPos(0, 0, 0), ShipAngle())
```

You can access the members of a structure using the same dot notation that C uses:

```
khansShip.name = "Reliant"
print khansShip.pos
```

When you assign a variable that holds a struct to another variable, it is not copied. Instead, both variables will point to the same data. This means that if you change the data for one variable, it will change in the other too:

```
' deckersShip and kirksShip now point to the same data
deckersShip = kirksShip ' i.e. Star Trek: The Motion Picture

deckerShip.pos = ShipPos(5, 6, 7)
' kirksShip.pos is also {5, 6, 7} now
```

The good news is that this means structures are passed to functions by reference, so it's fast. But there are times you'll want to make a copy of the data. You can do that with the DeepCopy function:

```
deckersShip = DeepCopy(kirksShip)
' deckersShip is now a separate copy of the data.
' Changes made to one will not affect the other
```

This is the same DeepCopy function that works with arrays, and it does a recursive copy of each of the struct's member variables. So if the member variables are arrays or structs, their data will be copied.

2.9 Objects and Handles

An object is a collection of data. For example, to store the coordinates of a point on a graph, you need an X value and a Y value. Instead of using two variables (or two items of an array), an object can combine the two values into one variable which could be called a "coordinate object". Then, you need a way to read and write the individual components (the X and Y values) of that object.

An object can take up a large amount of memory, depending on how much information it holds. A handle is a small amount of data, usually an integer that identifies the object. C programmers know it as a "pointer".

GSBASIC does not allow you to define objects in your program. But an application that embeds GSBASIC might use objects internally and make them available to GSBASIC. For example, The Vectrex32 provides a GSBASIC function called LinesSprite, which takes an array containing coordinates and returns a handle to an object that represents a drawing on the screen. In your GSBASIC program, you can then pass that handle to functions that rotate the drawing or hide it or change its size.

2.10 Expressions

Expressions consist of literals, variables, function calls and operators. Supported operators are (in priority order):

(,)	Left and right parentheses
+, -, NOT, ~, IsNil	Unary plus, unary minus, logical NOT, bitwise NOT, test for uninitialized variable
*, /, MOD	Multiply, divide, modulo
+, -	Add or string concatenate, subtract
<=, <, >=, >	Relational operators
=, <>, !=	Equals, and two ways to express Not Equal
&	Bitwise AND for integers
^	Bitwise exclusive OR for integers (Note: in versions before 1.21, ^ was the exponentiation operator. Now it's bitwise exclusive OR. Use the POW() function for exponentiation.)
|	Bitwise inclusive OR for integers
AND, OR, XOR	Logical operators. AND and OR use "short circuit evaluation" (see below).

Operators are case-insensitive, e.g. MOD is the same as mod and IsNil is the same as isnil.

The AND, OR, XOR, and NOT operators return a 0 for false or 1 for true. However, when evaluating their arguments, they treat any non-zero value as true.

Like C, but unlike many BASICs, the AND and OR operators use "short circuit evaluation". This means that the right hand side is evaluated only if necessary. If the left hand side of an AND evaluates to 0, then the AND will return 0 regardless of the value on the right side, so it does not evaluate the right side. Similarly, if the left side of an OR is non-zero, then the OR will return a 1 regardless of the right side's value, so the right side is not evaluated.

This can be very useful. Consider the following code:

```
DIM a[3]
i = 4
if i <= 3 AND a[i] = 0 THEN
   ' do something
ENDIF
```

If i is greater than 3, it is important not to evaluate the right side of the AND because it would cause an error. Fortunately, with GSBASIC's short circuit evaluation, the right side will not be evaluated when i is greater than 3.

The IsNil operator returns 1 if its operand is uninitialized:

```
PRINT IsNil a   ' Prints 1
b = 2
PRINT IsNil b   ' Prints 0
b = nil
PRINT IsNil b   ' Prints 1

IF IsNil b THEN
      b = 3
ENDIF
```

2.10.1 Integer Arithmetic

Numbers in GSBASIC are either integers or floating point numbers ("floats"). Integer arithmetic is faster than floating point arithmetic, and only integers can be used as array indexes, but it's important to understand their peculiar behavior.

The integer 5 is different from the floating point value 5.0. If you divide 5.0 by 2, you get 2.5; if you divide the integer 5 by 2, you get 2. An integer divided by an integer produces an integer.

Numbers can be converted to floats using the Float function; they can be converted to integers using the Int function. Dividing Float(5) by 2 would produce 2.5.

2.10.2 Operand Type Promotion Rules

Arithmetic operators (+, -, *, /, MOD) operate on numeric operands. If one of the operands is an integer and the other is a float, the integer is promoted to a float.

As mentioned before, the integer expression 5 / 2 equals 2. But 5 / 2.0 would produce 2.5, because the integer 5 would be promoted to a float. Likewise, 5.0 / 2 would equal 2.5.

The logical operators (NOT, AND, OR, and XOR) operate on integers, floats and strings. They cast their operands to integer ones or zeros (true or false), apply the logical operation, and produce an integer one or zero. If an operand is an integer or a float, it is cast to a 0 if the integer or float is zero. If the integer or float is non-zero, it is cast to a one. If an operand is a string, it is cast to a one if it is non-empty; an empty string is cast to a zero.

The string concatenation operator (+) must have a string as one of its operands. If the other operand is an integer or float, it is converted to a string and the concatenation is done.

```
x = 3.14159
print "x equals " + x ' Prints "x equals 3.14159"
```

2.11 String Manipulation

Strings can be concatenated with the + operator and compared with =, !=, <>, <=, and >=. There are several functions that operate on them: Asc, Chr, Instr, Left, Len, Mid, Right, StrComp (case-insensitive compare), ToLower, ToUpper, and Val.

Strings can also be used as byte arrays. For example:

```
str = "abcde"
PRINT s[1] ' Prints 97, the ASCII code for 'a'
s[1] = 65 ' Change the first byte in the array
PRINT str ' Prints "Abcde"

' Create a string initialized to 100 nul bytes.
data = ByteArray(100)
```

Strings are stored as a 16 bit length followed by the bytes. They are not nul terminated the way C strings are, so you can treat them as a byte array and store zeroes in them. However, the PRINT statement will only print them up to a nul byte.

2.12 Vector and Matrix Arithmetic

Arrays can be treated as mathematical vectors or matrices. BASIC allows you to add, subtract, and multiply matrices. They can also be divided by a number or negated with a unary minus sign. The built-in functions Transpose and Norm are also available.

A matrix and a number can be added together:

```
matrix1 = {{1, 2, 3}, {4, 5, 6}}
matrix1 = a + 10
matrix1 = 10 + a
```

In the above example, each element of matrix1 has 10 added to it. You can also add two matrices together. The matrices must be the same size. The corresponding elements of each matrix are added together:

```
matrix2 = {{7,8,9}, {10, 11, 12}}
d = matrix1 + matrix2
```

```
' d now equals {{8, 10, 12}, {14, 16, 18}}
```

Likewise, a number can be subtracted from a matrix (the number gets subtracted from each element in the matrix) and one matrix can be subtracted from another (the elements of the second matrix are subtracted from the corresponding elements in the first).

A matrix can be multiplied by a number:

```
e = matrix1 * 5
' e now equals {{5, 10, 15}, {20, 25, 30}}
```

And matrices can be multiplied together, which follows the normal rules of dot products. The number of columns of the first matrix must equal the number of rows of the second matrix:

```
matrix3 = {{7, 8}, {9, 10}, {11, 12}}
f = matrix1 * matrix3
' f now equals {{58, 64}, {139, 154}}
```

Vectors, or one dimensional arrays, can be multiplied together. Again, it's done as a dot product but the result is a scalar:

```
vector1 = {1, 2, 3}
vector2 = {4, 5, 6}
' Dot product will be 32 (1 * 4 + 2 * 5 + 3 * 6)
dotProduct = vector1 *  vector2
```

The Transpose() function swaps the columns and rows of a matrix:

```
g = Transpose(matrix1)
' g is now {{1,4},{2,5},{3,6}}
h = Transpose({1, 2, 3})
' h is now {{1}, {2}, {3}}
```

The Norm() function calculates the Euclidean norm of a matrix or the magnitude of a vector. Like the Pythagorean theorem, it adds the squares of all the elements in the matrix and returns the square root:

```
v = {3, 4}
n = norm(v) ' n is set to sqrt(3 * 3 + 4 * 4), or 5
```

The multiplication operator (*) does a dot product; there is a Cross() function to calculate the cross product of two 3 dimensional vectors:

```
' cp is set to the cross product of two vectors, in this case
' {-3, 6, -3}
cp = Cross({2, 3, 4}, {5, 6, 7})
```

2.13 Control Structures

GSBASIC offers IF...THEN...ELSEIF...ELSE...ENDIF, FOR...NEXT, WHILE...ENDWHILE, and REPEAT...UNTIL control structures.

There are two statements for altering the execution of the FOR, WHILE and REPEAT loops: CONTINUE <loop type> and EXIT <loop type>. The CONTINUE statement jumps to the end of the loop, so that no more code is executed in the current iteration of the loop, and the next iteration of the loop begins (assuming the loop's exit condition is not met). It is similar to C's *continue* statement.

The EXIT statement terminates the loop. It is similar to C's *break* statement.

CONTINUE and EXIT specify which loop they apply to. For example, if a FOR loop contains a WHILE loop, and the WHILE loop contains a CONTINUE FOR statement, then the CONTINUE applies to the FOR loop, not the WHILE. In this way, CONTINUE and EXIT are more versatile than C's *continue* and *break*.

(If a FOR loop contains another FOR loop, and the inner FOR loop contains a CONTINUE FOR or an EXIT FOR, the CONTINUE or EXIT apply to the inner FOR loop. There is no way to have them apply to the outer FOR loop.)

2.14 Labels

GSBASIC does not currently support labels, and therefore does not support GOTO statements. However, its looping control structures (FOR...NEXT, WHILE...[DO]...ENDWHILE, REPEAT...UNTIL) support EXIT and CONTINUE to alter their flow.

2.15 DATA and READ Statements

GSBASIC does not support DATA or READ statements. However, you can use Array Initializers as a source of hard-coded data.

2.16 Functions and Subroutines

GSBASIC provides built-in functions and subroutines, which are described in section 6. You can also define your own functions and subroutines; see the statements FUNCTION and SUB. Functions are different from subroutines in that they return a value.

Subroutines are invoked by the CALL statement. Functions can also be invoked by CALL, but are more typically called by using them in expressions (e.g. c = SQRT(a * a + b * b)), where their return value can be used.

By default, parameters are passed to functions and subroutines by value. However, variables that have arrays or structures are actually pointers to the array or structure data. So the pointers are passed by value, but if you modify the contents of the array or struct, those changes will be visible when you return from the subroutine or function.

The variable itself, however, can't be changed unless you pass it by reference. The ByRef keyword specifies that a parameter is to be passed by reference. A couple of examples will help clarify all this:

```
myInt = 5
myInt = 5

myArray1 = {4, 5, 6}
myArray2 = myArray1
```

```
struct Struct {a, b, c}
myStruct1 = Struct(1, 2, 3)
myStruct2 = myStruct1

call AlterValue(myInt, myArray1, myStruct1)

sub AlterValue(scalarArg, arrayArg, structArg)
   scalarArg = scalarArg + 1
   arrayArg[1] = arrayArg[1] + 1
   arrayArg = arrayArg * 2
   structArg.c = structArg.c + 1
   structArg = "Replaced by string"
endsub
```

In the above example, myArray1 and myArray2 are pointing to the same array data, and myStruct1 and myStruct2 are both pointing to the same structure data. After the call to AlterValue, myInt will still equal 5, since it was passed by value. myArray1 and myArray2 will still point to the same array data (despite the fact that AlterValue modified arrayArg) but the first element of that array will have been incremented. Likewise, myStruct1 and myStruct2 will continue to point to the same data (despite the fact that AlterValue modified structArg), and that structure's c member will have been incremented by one.

Now let's look at an example where all the arguments are passed by reference:

```
myInt = 5

myArray1 = {4, 5, 6}
myArray2 = myArray1

struct Struct {a, b, c}
myStruct1 = Struct(1, 2, 3)
myStruct2 = myStruct1

call AlterRef(myInt, myArray1, myStruct1)

sub AlterRef(ByRef scalarArg, ByRef arrayArg, ByRef structArg)
   scalarArg = scalarArg + 1
   arrayArg[1] = arrayArg[1] + 1
   arrayArg = arrayArg * 2
   structArg.c = structArg.c + 1
   structArg = "Replaced by string"
endsub
```

When AlterRef returns, myInt will have been changed to 6. myArray2[1] will have been incremented, but myArray1 will no longer point to the array; instead it will point to the array {8, 10, 12} (because AlterRef created a new array when it multiplied ArrayArg by 2). Likewise, myStruct2.c will have been incremented, but myStruct1 will be set to the string "Replaced by string".

If a subroutine or function expects an array as one of its arguments, you can write an array initializer in its place.

```
SUB PrintArray(array)
      FOR x = 1 TO UBOUND(array)
```

```
            PRINT array[x]
      NEXT x
ENDSUB

CALL PrintArray({1, 2, 3})
```

2.17 Include Files

You might want to separate some of your code into separate file(s), for example if you've written a set of subroutines that are used by multiple programs. BASIC supports C-like include files to allow this.

You might, for example, have a game written in the file foo.bas. It could make use of subroutines that are in the file bar.bai ("bai" meaning "BASIC include"). So in foo.bas, you would have a line of code:

```
INCLUDE "bar.bai"
```

When BASIC is loading foo.bas and it encounters the INCLUDE statement, it will read in the contents of bar.bai. It's almost as if you copied and pasted the contents of bar.bai into foo.bas.

Include files can, themselves, have include statements in them. Bar.bai, for example, could include baz.bai. Each file will be included only once. If, for example, foo.bas includes bar.bai and baz.bai, but bar.bai also includes baz.bai, baz.bai will be included only once, and it will be included at the first INCLUDE statement that BASIC encounters.

Including a file is almost like copying and pasting it in, but not entirely. Line numbers are not affected by an INCLUDE statement. Errors that occur in foo.bas will show the foo.bas line number. Errors that occur in an include file will show the file name and line number of the error.

It is possible to set breakpoints in an include file. For example,

```
BP "baz.bai" 123
```

Will set a breakpoint at line 123 in the baz.bai include file. Likewise, breakpoints can be cleared:

```
BP C "baz.bai" 123
```

A program can have no more than 8 include files (either directly or included by other include files).

2.18 Variable Scope

A function or subroutine in your program has a local scope. A variable's scope is determined when it is first assigned a value (i.e. when it is created): if the variable is first assigned a value inside a function or subroutine, its scope is limited to the function or subroutine; if a variable is first assigned a value in the main program, it has global scope.

If a variable name that exists in the global scope is used within a function or subroutine, the global variable will be used. If you would like to ensure that a variable accessed within a function or subroutine is a local variable, you can explicitly DIM it:

```
foo = 1
CALL MySubroutine

SUB MySubroutine()
      DIM foo
      ' The following PRINT statement will cause an
      ' uninitialized variable error, since this foo is local
      PRINT foo
ENDSUB
```

2.19 Case Sensitivity

Only variable names are case-sensitive. All other program elements are case insensitive (e.g. statements, logical operators like AND, OR and NOT, built-in function names, user function names, predefined constants, and file names).

2.20 Error Handling

Normally, when an error occurs in a program, execution stops and an error message is printed. However, you can avoid that by setting up your own error handling subroutine.

To set this up, write a subroutine to do the error handling and use the ON ERROR CALL statement to tell GSBASIC to call your subroutine in the event of an error. Once in the subroutine, you can call GetLastError() to retrieve information about what happened. For example:

```
‘ Create a global variable that will be set in the error handler
errorFlag = false

SUB MyErrorHandler
      errorFlag = true
      err = GetLastError()
      ‘ Print the error information
      PRINT “Error code = “, err[1]
      PRINT “Error message = “, err[2]
      PRINT “Line number = “, err[3]
      PRINT “Column number = “, err[4]
      PRINT “Source file = “, err[5]
      CALL ClearLastError()
ENDSUB

ON ERROR CALL MyErrorHandler
a = MyErrorProneFunction()
ON ERROR CALL 0
IF errorFlag = true THEN
      ‘ Do something different
ENDIF
```

If an error occurs in MyErrorProneFunction, the MyErrorHandler subroutine will be called. When MyErrorHandler returns, execution will resume at the line following the one where the error occurred. Notice that since if the error occurs in MyErrorProneFunction(), execution of that line of code will stop and the variable ‘a’ will not be assigned a value.

ON ERROR CALL can be used multiple times in a program to set up different error handlers for different blocks of code. The error handler can be removed with the statement ON ERROR CALL 0. Error handler settings are scoped to the main program or the subroutine/function they are

configured in. I.e. if you have an ON ERROR CALL in a subroutine, it only applies while the program is executing that subroutine, or functions/subroutines called from that subroutine. When the program returns from the subroutine, any error handler that was configured before is automatically re-instated as the error handler.

The function GetLastError() returns a five-element array with the error code (an integer), the text of the error message, the line number where the error occurred, the column number where it occurred, and the file name (e.g. the main BASIC file or the include file). The ClearLastError() routine resets the last error to zero.

3 USING GSBASIC

GSBASIC accepts statements entered at the command line. Typically, one statement is entered per line, and the statement is executed when you hit Enter. There are two exceptions. First, if the last non-space character on a line is an underscore (_), GSBASIC will prompt for the rest of the statement to be entered. Second, if the statement begins a block (e.g. a WHILE statement that must be followed by a block of code and an ENDWHILE, a FUNCTION statement that must be followed by a block of code and an ENDFUNCTION) GSBASIC will prompt you to enter additional statements until the block is terminated.

Lines are limited to 200 characters in length.

When GSBASIC is accepting commands, it can be in one of two modes: Immediate Mode or Debug Mode. They are the same except that Debug Mode means that you have interrupted a running BASIC program.

3.1 Immediate Mode

GSBASIC uses the prompt "Ready" followed by a newline to show that it's ready for you to enter a statement. If you are continuing a single statement on a new line by using the underscore, GSBASIC will prompt with "_>". If you are entering a block of code, GSBASIC will prompt you with "&>".

GSBASIC offers a few line-editing features. Their correct functioning depends on the specific console you are using. Under Microsoft Windows, GSBASIC uses the operating system's console. If GSBASIC is embedded in another system, refer to that system's manual for information on the console.

The backspace key works as you would expect, deleting the character at the end of the line.

GSBASIC retains a history of the last 10 unique lines you entered. Assuming the terminal emulator supports arrow keys, you can move through the history list by pressing the Up and Down Arrow keys; otherwise, you can use Ctrl+P and Ctrl+N. The history list is circular.

Ctrl+C causes GSBASIC to discard any statement(s) you're in the process of entering.

3.2 Running Programs

GSBASIC has access to a single directory on the host system. On Windows, it's the working directory - typically the directory GSBASIC.exe was run from. When embedded in another system, it's whatever that system provides to GSBASIC. Subdirectories are not currently supported.

If the directory has a file called autorun.bas, GSBASIC will load and run it at startup.

GSBASIC has the LOAD statement to load a BASIC program from the directory and a RUN statement to run it. The LOAD statement takes a string as an argument, so the file name must be enclosed in quotes. If the file is not found, GSBASIC adds a ".BAS" extension to it and tries again. File names are case-insensitive.

```
' Tries to load the file Hello. If not found, tries Hello.bas
load "Hello"
run
```

GSBASIC does not allow you to edit or save programs, so your workflow will be to write the program in an editor of your choice, save it into GSBASIC's working directory, then use GSBASIC's console to load and run it.

3.3 Debugging

While a program is running, it can break into Debug Mode either by executing a BP (Breakpoint) statement or in response to your hitting Ctrl+C. Debug Mode is very similar to Immediate Mode; while in Debug Mode, you can execute statements, print or modify variables, and call subroutines and functions in the program. To single-step through your program, enter the STEP statement with an optional count of the number of statements to execute. To resume normal execution, enter the RUN statement.

While in Debug mode, the console prompt is "Debug (depth = *n*)" instead of "Ready". The depth count is important for keeping track of how deeply nested in Debug modes you are. For example, consider this program with a subroutine that contains an infinite loop:

```

call loop()

sub loop
      i = 0
      repeat
            print i
            i = i + 1
      until 1 = 0
endsub
```

When run, this program will start printing out numbers. If you hit Ctrl+C, execution will break somewhere inside the loop and you will be at Debug depth 1.

In Debug mode, you can call functions and subroutines in your program. So let's say you enter "CALL LOOP". It will start printing out numbers again and you will need to break it by hitting Ctrl+C. You will again enter Debug mode but you will be at depth 2. This is because you did not exit Debug (depth = 1) mode; you called a subroutine from Debug mode and then had to break out of it into a new Debug mode.

Normally, you would exit Debug mode by entering RUN. In this perverse example, RUN will just put you into the infinite loop again, and you can only break out of it by hitting Ctrl+C again. The only way out is to STOP the program. In a more typical scenario, where you call a well-behaved function from Debug mode, and the function returns in a reasonable amount of time, you will stay

in Debug (depth = 1) mode and will be able to resume running your program with the RUN command.

3.4 Breakpoints

Breakpoints cause your program to pause execution and enter Debug mode. Breakpoints can be hard-coded into your program or set at the command line.

To hard-code a breakpoint into your program, place the BP statement in your program. When the program executes that line, GSBASIC will print the line number of the BP statement, then enter Debug mode. You may have any number of BP statements in the program.

Alternatively, you can set, clear, and display soft breakpoints from Immediate or Debug mode. For details, see the section labeled BP (Breakpoint) Statement

3.5 Tracing

In Immediate or Debug Mode, you can give the commands TRON (Trace On) or TROFF (Trace Off). They can even be embedded in your program. When tracing is on, the line number, in brackets, of each statement will be printed before the statement is executed. GSBASIC does not have line numbers like old versions of BASIC do, so the line number that is printed is the actual position of the statement in the program file.

3.6 Autorun.bas

When GSBASIC starts up, it looks for a file named autorun.bas in its working directory. If it finds it, GSBASIC loads and runs the autorun.bas program. If autorun.bas exits, GSBASIC enters immediate mode.

4 STATEMENTS

4.1 BP (Breakpoint)

Description

Breaks execution of the program and enters Debug mode

Example

```
BP ' In a program, breaks to the debugger.
BP ' At the command line, prints the "soft" breakpoints.
BP 15 ' Sets a soft breakpoint at line 15
BP CLEAR ' Removes all soft breakpoints
BP CLEAR 15 ' Removes the soft breakpoint at line 15
BP C ' Same as BP CLEAR
BP C 15 ' Same as BP CLEAR 15

BP "baz.bai" 123 ' Sets a breakpoint at line 123 in
                 ' the include file baz.bai
BP C "baz.bai" 123 'Clears the breakpoint in the include file
```

More Information

This statement can be used at the command line or in a program. When hard-coded into a program, it takes no arguments, and when executed, causes GSBASIC to enter Debug mode.

When used at the command line, in Immediate or Debug mode, it sets, removes, or displays a list of "soft" breakpoints (as opposed to hard-coded breakpoints). Before executing a line of code, GSBASIC will check whether a soft breakpoint has been set there. If it has, GSBASIC will enter Debug mode before the line of code is executed.

Once in Debug mode, GSBASIC will allow you to examine and modify the program's state. Use the RUN statement to leave Debug mode and continue executing the program.

See Also
RUN, Debugging, Breakpoints

4.2 CALL

Description
Calls a subroutine or function.

Example
```
CALL FOO
CALL BAR(1, a, SIN(3.14159))
```

More Information
Subroutines may have any number of parameters. If a subroutine takes no arguments, it is not necessary to have parentheses. (This is not true of function calls in expressions: they must have parentheses even when they have no arguments.)

A function may be called with the CALL statement. The function's return value will be discarded.

See Also
SUB...ENDSUB, FUNCTION...ENDFUNCTION,

Functions, Variable Scope

4.3 CHAIN

Description
Ends the currently running program, loads and runs another.

Example
```
CHAIN "lunar.bas"
CHAIN "lunar" RERUN
```

More Information
The CHAIN statement is the equivalent of a STOP, LOAD, and RUN. The RERUN keyword is optional. In the second example above, the current program chains to lunar.bas and specified RERUN. When lunar.bas exits the calling program will be loaded and run from the beginning. (Some other BASICs have CHAIN ... RETURN which resumes execution at the statement after the CHAIN. GSBASIC does not support that.)

4.4 CONTINUE {FOR | REPEAT | WHILE}

Description
Passes control to the next iteration of the nearest enclosing loop of the specified type.

Example

```
a = 0
b = 10
WHILE a < b
      a = a + 1
      PRINT "This text will be printed 10 times"
      CONTINUE WHILE
      PRINT "This text will not be printed"
ENDWHILE

FOR x = 1 TO 10
      PRINT "This text will be printed 10 times"
      REPEAT
            CONTINUE FOR ' Exits REPEAT loop and CONTINUEs FOR
            PRINT "This text will not be printed"
      UNTIL a < b
      PRINT "This text will also not be printed"
NEXT x

FOR x = 1 TO 10
      PRINT "This text will be printed 10 times"
      FOR y = 1 TO 10
            PRINT "This text will be printed 100 (10*10) times"
            CONTINUE FOR
            PRINT "This text will not be printed"
      NEXT y
      PRINT "This text will be printed 10 times"
NEXT x
```

More Information

The CONTINUE statement must specify a FOR, REPEAT, or WHILE loop type. It skips the remainder of the code in the specified loop type and continues with the next iteration of that loop. As shown in the examples, this allows the CONTINUE statement to exit out of inner loops on its way to continuing an outer loop.

See Also

EXIT {FOR | REPEAT | WHILE}

4.5 DIM

Description

Create or re-create an array. Alternatively, create a local variable.

Example

```
DIM a[10]
' Destroy the previous 'a' array and create a new one
DIM a[2,3]

b = 1 ' Create a global variable
c = 2 ' Create another local variable

CALL MySubroutine
```

```
SUB MySubroutine
      DIM b ' Create a local variable
      b = 3 ' Initialize the local variable
      PRINT c ' print the global variable
ENDSUB

DIM d[b + c], e[3, 4, b] ' Dimensions can be expressions
```

More Information

DIM typically allocates an array. If an array of that name already exists within the same scope, that array is destroyed and a new one is created. Data in the old array is not preserved.

When crating a multi-dimensional array, the numbers of elements in each dimension are separated by commas.

DIM creates variables in the current scope, so when used inside a subroutine or function, DIM creates local variables, both arrays and scalars.

The DIM statement is executable, unlike an array declaration in C. Since it is executed, the dimensions specified for the array can be expressions, as shown in the example above.

See Also

Arrays, Variables, Variable Scope

4.6 DIR

Description

Prints out a list of files in the working directory.

Example

```
DIR
```

See Also

Running Programs, Dir Function

4.7 EXIT {FOR | REPEAT | WHILE}

Description

Terminate execution of the nearest enclosing loop of the specified type.

Example

```
WHILE a < b
      PRINT "This text will be printed once"
      EXIT WHILE
      PRINT "This text will not be printed"
ENDWHILE

FOR x = 1 TO 10
      PRINT "This text will be printed once"
      REPEAT
            PRINT "This text will be printed once"
```

```
            EXIT FOR
            PRINT "This text will not be printed"
      UNTIL a < b
      PRINT "This text will also not be printed"
NEXT x

FOR x = 1 TO 10
      PRINT "This text will be printed 10 times"
      FOR y = 1 TO 10
            PRINT "This text will be printed 10 times"
            EXIT FOR
            PRINT "This text will not be printed"
      NEXT y
      PRINT "This text will be printed 10 times"
NEXT x
```

More Information

The EXIT statement specifies a FOR, REPEAT, or WHILE loop type. It skips the remainder of the code in the specified loop type and exits the loop. As shown in the examples, this allows the EXIT statement to exit out of inner loops on its way to exiting an outer loop.

See Also

CONTINUE {FOR | REPEAT | WHILE}

4.8 FOR...NEXT

Description

Repeats a group of statements while a variable counts from a start value to an end value.

Example

```
last = 100
FOR x = 1 TO last
      PRINT "This will be printed 100 times"
NEXT x

FOR y = last TO last + 10
      PRINT "This will be printed 10 times"
NEXT

FOR z = 10 TO 1 STEP -2.5
      PRINT "This will be printed 4 times"
NEXT z
```

More Information

The start, end, and optional step values of the loop must be numbers. They are calculated only once, when the loop is entered (i.e. if the end or step value is an expression, the expression is evaluated only once). If no step is specified, the loop variable is incremented by 1 on each iteration.

When a FOR ... NEXT loop starts, BASIC evaluates the start, end, and step values. BASIC evaluates these values only at this time and then assigns start to the counter variable. Before the

statement block runs, BASIC compares the counter to end. If counter is already larger than the end value (or smaller if the step is negative), the FOR loop ends and control passes to the statement that follows the NEXT statement. Otherwise, the statement block runs.

Each time BASIC encounters the NEXT statement, it increments the counter by the step and returns to the FOR statement. Again it compares counter to end, and again it either runs the block or exits the loop, depending on the result. This process continues until the counter passes end or an EXIT FOR statement is encountered.

The loop doesn't stop until the counter has passed end. If the counter is equal to end, the loop continues. The comparison that determines whether to run the block is counter <= end if step is positive and counter >= end if step is negative.

Changing the value of start, end, or step doesn't affect the iteration values that were determined when the loop was first entered. The NEXT statement does not need to specify the loop variable.

See Also

CONTINUE {FOR | REPEAT | WHILE}, EXIT {FOR | REPEAT | WHILE}

4.9 FUNCTION...ENDFUNCTION

Description

Defines a user-defined function.

Example

```
FUNCTION MyFunction(a, b)
      RETURN a + b
ENDFUNCTION

FUNCTION PI()
      RETURN 3.14159
ENDFUNCTION

PRINT MyFunction(1, 2) + PI()
```

More Information

A function definition must have a parameter list, even if it is empty (as in the example of Pi(), above). A function must return a value. A function is typically called from an expression. It may also be called from a CALL statement, but if it is, the return value is discarded.

Function names are case-insensitive.

Variables used within a function are global unless they are explicitly created in a DIM statement. Functions may call other subroutines and functions, and may be recursive.

See Also

SUB...ENDSUB,

Functions, Variable Scope

4.10 IF...[THEN]...[ELSEIF...[THEN]...][ELSE...]ENDIF

Description

Conditional execution.

Example

```
IF a = b AND c <> d THEN
      PRINT "Hello "
ELSEIF e = f THEN
      PRINT "Goodbye "
ELSE
      PRINT "world"
ENDIF
```

More Information

The THEN keywords on the IF and ELSEIF lines are optional. There may be any number of ELSEIFs.

Unlike other versions of BASIC, there is no single-line IF...THEN statement, e.g. you cannot write

```
IF a = b THEN PRINT "Hi" ' This will cause an error
```

4.11 INCLUDE

Description

Includes a source file in the program, similar to C's #include.

Example

```
INCLUDE "bar.bai"
```

More Information

The INCLUDE statement allows you to incorporate the code in another file into the current program. Details can be found in section 2.17.

See Also

Include Files

4.12 LET

Description

Assign a value to a variable.

Example

```
LET a = 1
b = 2 + a
c = {{a, b}, {4, 5}} ' Using an array initializer
```

More Information

The LET keyword is optional.

4.13 LOAD

Description

Loads and compiles a program from the working directory into memory.

Example

```
LOAD hello.bas
LOAD hello
LOAD "01_hello" ' Quotes are required for file names that don't
begin with a letter
```

More Information

File names are case-insensitive. LOAD looks for the file you specified and, if it does not find it, appends ".bas" to the string and looks again.

If the file is found, it is loaded, compiled to an intermediate form and prepared for execution. If there are compilation errors, the LOAD will fail.

See Also

RELOAD, Running Programs

4.14 MEM

Description

Prints the amount of memory being used by the program, as well as the free space available.

4.15 ON ERROR CALL

Description

Sets up or removes an error handler.

Example

```
SUB MyErrorHandler
      errorFlag = true
      err = GetLastError()
      ' Print the error information
      PRINT "Error code = ", err[1]
      PRINT "Error message = ", err[2]
      PRINT "Line number = ", err[3]
      PRINT "Column number = ", err[4]
      CALL ClearLastError()
ENDSUB

ON ERROR CALL MyErrorHandler
a = MyErrorProneFunction()
ON ERROR CALL 0
IF errorFlag = true THEN
      ' Do something different
ENDIF
```

More Information

Normally, when an error occurs in a program, execution stops and an error message is printed. However, you can avoid that by setting up your own error handling subroutine. You can also remove your error handling with ON ERROR CALL 0.

See Also

Error Handling

4.16 OPTION BASE

Description

Changes the array index base.

Example

```
OPTION BASE 0
OPTION BASE 1
```

More Information

By default, arrays a one-based (the index of their first element is 1). However, you might prefer to have your arrays zero-based (the index of their first element is 0). To do this, put OPTION BASE 0 anywhere in your program. When your program is loaded – even before it is run – the base for arrays will be set to zero.

Alternatively, you can also have OPTION BASE 1 in your program. This sets the array base to 1, which is done by default anyway. So OPTION BASE 1 is really just for documentation purposes.

OPTION BASE may appear only once in your program.

The Option Base will affect the behavior of the UBound function, but not the Size function.

See Also

Size, UBound

4.17 PRINT

Description

Outputs text to the console.

Example

```
PRINT "The variable 'a' equals ", a
```

More Information

The PRINT statement evaluates a comma-separated list of expressions and outputs them as text to the console, followed by a new line (carriage return and line feed). No spaces or tabs are inserted between expressions.

4.18 RELOAD

Description

Reloads the current program from the working directory.

Example

```
RELOAD
```

More Information

Reloads the program that was previously loaded. This is a convenient command to use when you are editing a program in a separate window, saving it, and testing the changes.

See Also

LOAD

4.19 REPEAT...UNTIL

Description

Executes a block of statements until a condition is true.

Example

```
REPEAT
      a = a + 1
UNTIL a = 100
```

More Information

The statements inside the REPEAT...UNTIL block will be executed at least once.

See Also

WHILE...[DO]...ENDWHILE, CONTINUE {FOR | REPEAT | WHILE}, EXIT {FOR | REPEAT | WHILE}

4.20 RETURN

Description

Returns from a user-defined function or subroutine.

Example

```
RETURN a + b ' Return from a function
RETURN ' Return from a subroutine
```

More Information

The RETURN statement may appear anywhere in a function or subroutine, and may even appear multiple times. The RETURN statement must appear in a function, along with an expression that will be the returned value.

The RETURN statement is optional in a subroutine - the subroutine will automatically return when execution reaches the ENDSUB statement - but if it appears in a subroutine, it must not specify a return value.

See Also

SUB...ENDSUB,

4.21 RUN

Description

Executes the program in memory.

Example

```
LOAD "Hello.bas"
RUN
```

More Information

This statement is allowed only in Immediate mode. The program must have been loaded with the LOAD or RELOAD statement. All variables are destroyed before beginning execution.

See Also

LOAD, Immediate Mode, Running Programs

4.22 STEP

Description

Execute the specified number of statements.

Example

```
STEP ' Executes one statement
STEP 10 ' Executes 10 statements
```

More Information

The STEP statement may be used in Immediate mode, after a program has been loaded, or in Debug mode. It executes the specified number of statements, then breaks into Debug mode. If no number of statements is specified, one statement will be executed. If a number is specified, it must be an integer literal.

See Also

Debugging

4.23 STOP

Description

Terminates the program.

Example

```
STOP
```

More Information

This statement can be used within or program or in Debug mode. It terminates the program and returns to Immediate mode.

4.24 STRUCT

Description

Defines a C-like struct

Example

```
STRUCT Person {Name, Address, Age}

STRUCT Address {Street, City }

fred = Person("Fred Flintstone", _
   Address("345 Cave Stone Road", "Bedrock"), 35)

print fred.Address.Street
```

More Information

The STRUCT statement defines the name and member variables for a structure. Variables can then be created with that structure type and used much a C structs are.

See Also

Structs

4.25 SUB...ENDSUB

Description

Defines a user-defined subroutine.

Example

```
SUB TowersOfHanoi(disks, source, destination, temp)
      IF disks = 1 THEN
            PRINT "Move disk from tower ", source, _
                  " to tower ", destination
      ELSE
            CALL Hanoi(disks - 1, source, temp, destination)
            CALL Hanoi(1, source, destination, temp)
            CALL Hanoi(disks - 1, temp, destination, source)
      ENDIF
ENDSUB
```

More Information

A subroutine definition may have a parameter list, but does not require one. A subroutine is invoked from a CALL statement. Subroutine names are case-insensitive.

Variables used within a subroutine are global unless they are explicitly created in a DIM statement. Subroutines may call other subroutines and functions, and may be recursive.

See Also

RETURN, FUNCTION...ENDFUNCTION,

Functions, Variable Scope

4.26 TRON and TROFF

Description

Turns program tracing on and off.

Example

```
TRON
RUN
[1][2][3][4][5][6] ...
TROFF
```

More Information

Tracing may be turned on in immediate mode, debug mode, or even from a line in a program. When tracing is on, the current line number will be printed before each line of the program is executed.

See Also

STEP, Tracing, Debugging

4.27 WHILE...[DO]...ENDWHILE

Description

Executes a block of statements while a condition is true.

Example

```
WHILE a < 100
      a = a + 1
ENDWHILE
WHILE true DO ' An infinite loop
ENDWHILE
```

More Information

The statements inside the WHILE...ENDWHILE will be executed as long as the condition is true.

The DO keyword is optional.

See Also

REPEAT...UNTIL, CONTINUE {FOR | REPEAT | WHILE}, EXIT {FOR | REPEAT | WHILE}

5 Built-In Constants

These are the constants built in to GSBASIC. There may be additional built-in constants provided by the system that GSBASIC is embedded in; see that system's documentation for a list.

Built-in constant names are case-insensitive.

5.1 False

Description

The integer 0.

Example

```
REPEAT
UNTIL false ' An infinite loop
```

5.2 Nil

Description

An uninitialized value.

Example

```
DIM a[10]
a[1] = 1
a[1] = nil
PRINT a[1] ' Causes an error, "Variable is used before being
           ' assigned a value"
IF IsNil a[1] THEN
      a[1] = 3
ENDIF
```

More Information

Nil can be used to uninitialize a value that was previously initialized.

5.3 Pi

Description

The ratio of a circle's circumference to its diameter; 3.14159.

5.4 True

Description

The integer 1.

Example

```
WHILE true ' An infinite loop
ENDWHILE
```

6 Built-In Functions

These are the functions built in to GSBASIC. There may be additional built-in functions provided by the system that GSBASIC is embedded in; see that system's documentation for a list.

Built-in function names are case-insensitive.

6.1 Abs

Description

Returns the absolute value of a number.

Example

```
PRINT Abs(-4.5) ' Prints 4.5
```

More Information

The argument must be an integer or a float. The return type is the same as the argument type.

6.2 Acos

Description

Returns the arccosine of a number in radians.

Example

```
PRINT ACos(-1) ' Prints 3.14159
```

See Also

Asin, Atan, Sin, Tan

6.3 AppendArrays

Description

Returns an array that has the values of two arrays appended to each other.

Example

```
a = {1, 2, 3}
b = {4, 5}
c = AppendArrays(a, b) ' c will be {1, 2, 3, 4, 5}
d = {{1, 2}}
e = {{3, 4}}
f = AppendArrays(d, e) ' f will be {{1, 2},{3, 4}}
```

More Information

AppendArrays creates a new array, copies the values of the first array argument into it then copies the values of the second array into it. The two array arguments must have the same number of columns but may have different numbers of rows. For example:

```
DIM a[5], b[2], c[2, 5], d[3, 5], e[5, 6], f[2, 3, 4], g[5, 3, 4]
x = AppendArrays(a, b)   ' Valid
x = AppendArrays(c, d)   ' Valid
x = AppendArrays(f, g)   ' Valid
x = AppendArrays(d, e)   ' Illegal
x = AppendArrays(a, c)   ' Illegal
```

It's sometimes useful to build an array completely with calls to AppendArray, e.g. you start with an empty array and append a row at time to it. This can be done by dimensioning the array with a zero rows. For example:

```
' Start with a 2D array with zero rows
DIM arrayToBuild[0, 2]

' Create newRow as a 2D array of dimension [1, 2]
newRow = {{"hello", "world"}}

' Build up the array by appending a row
arrayToBuild = AppendArrays(arrayToBuild, newRow)
```

When values are copied to the new array, it is a "shallow copy", i.e. if a value is an array, the copied value will point to the same underlying array.

See Also

CopyArray, Arrays

6.4 Asc

Returns the ASCII value of the first character in a string.

Example

```
PRINT Asc("ABC") ' prints 65
```

See Also

Chr

6.5 Asin

Description

Returns the arcsine of a number in radians.

Example

```
PRINT Asin(0) ' Prints 1.5708, which is PI / 2
```

See Also

Acos, Atan, Sin

6.6 Atan

Description

Returns the arctangent of a number in radians.

Example

```
PRINT Atan(1) ' Prints 0.7854, which is PI / 4
```

See Also

Acos, Asin, Atan2

6.7 Atan2

Description

Returns the arctangent of y / x in radians, taking into account the sign of both arguments in order to determine the quadrant.

Example

```
PRINT Atan2(1, 1) ' Prints 0.785398, which is 45 degrees
PRINT Atan2(1, -1) ' Prints 2.35619, which is 135 degrees
PRINT Atan2(-1, -1) ' Prints -2.35619, which is -135 degrees
PRINT Atan2(-1, 1) ' Prints -0.785398, which is -45 degrees
```

More Information

Atan2 returns values between $+\pi$ (inclusive) and $-\pi$.

See Also

Acos, Asin, Atan

6.8 ByteArray

Description

Creates a string initialized with nul bytes.

Example

```
' Create a string initialized to 100 nul bytes
data = ByteArray(100)
```

More Information

Individual bytes in a string can be accessed as if the string is an array, e.g:

```
data = "hello"
PRINT data[1] ' Prints 104, the ASCII code for h
data[1] = 72 ' Sets the first character to capital H
```

Aside from the convenience of accessing individual characters this way, it allows binary data to be stored as byte arrays, thus saving a lot of memory compared to regular arrays. The ByteArray() function lets you allocate a zero-initialized string of any length that you can then use as an array.

6.9 Chr

Description

Returns the character associated with the specified character code.

Example

```
PRINT Chr(33) ' Prints an exclamation mark (ASCII code 33)
```

More Information

The argument must be an integer between 0 and 255.

See Also

Asc

6.10 ClearLastError

Description

Sets the last error to {0, “”, 0, 0}

Example

```
CALL ClearLastError()
‘ GetLastError now returns {0, “”, 0, 0}
err = GetLastError()
```

See Also

GetLastError, Error Handling

6.11 CopyArray

Description

Returns an array that has a copy of the values of the input array.

Example

```
a = {1, 2, 3}
b = CopyArray(a) ' b will be {1, 2, 3}
```

More Information

CopyArray creates a new array and copies the values of the input array argument into it. This is different from assigning one array to another, because an assignment creates two references to the same underlying array. For example:

```
a = {1, 2, 3}
copy = CopyArray(a) ' Copies the values of a into copy
ref = a      ' ref is now referring to the same array data as a
a[1] = 5
PRINT copy[1]' Prints 1
PRINT ref[1] ' Prints 5, since a has the same array data as ref
```

When values are copied to the new array, it is a "shallow copy", i.e. if a value is an array or a struct, the copied value will point to the same underlying array or struct. It’s possible to recursively copy the data, instead of copying references, with the DeepCopy function.

See Also

Arrays, DeepCopy

6.12 Cos

Description

Returns the cosine of an angle, which is given in radians.

Example

```
PRINT Cos(3.14159) ' Prints -1
```

See Also

Acos, Asin, Atan, Sin

6.13 Cross

Description

Calculate the cross product of two vectors.

```
cp = Cross({2, 3, 4}, {5, 6, 7})
```

More Information

The arguments must be one-dimensional arrays with three elements each, i.e. vectors in three dimensions. The result is a vector perpendicular to the two arguments.

See Also

Vector and Matrix Arithmetic

6.14 DeepCopy

Description

Returns variable that is a deep copy of the input value.

Example

```
Struct myStruct {member1, member2, member3}

a = {1, 2, myStruct(3, 4, 5)}
b = DeepCopy(a) ' b will be {1, 2, {3, 4, 5}}
' This will change b[3].member2 but not a[3].member2
b[3].member2 = 100

' This just has c and a refer to the same data.
c = a
' This modifies c[3].member2, which will also change
' a[3].member2
c[3].member2 = 200
```

More Information

DeepCopy recursively copies the values of the input argument (typically a struct or array). This is different from assigning one array or struct to another, because an assignment creates two references to the same underlying array. It is different from CopyArray because (1) it can also copy structs and (2) if some of the elements of the array are structs or (nested) arrays, CopyArray only copies references to them whereas DeepCopy creates new structs or arrays and copies the data.

See Also

Arrays, CopyArray

6.15 Dir

Description

Returns the an array containing the names of all the files in the current directory.

Example

```
PRINT Dir() ' Prints an array of file names
```

More Information

The file names are sorted into alphabetic order.

See Also

DIR Statement

6.16 Float

Description

Converts an integer to a float.

Example

```
a = 5
a = Float(a) ' Converts a to a float (5.0)
```

See Also

Int

6.17 GetLastError

Description

Returns an array with information about the last error that occurred.

Example

```
err = GetLastError()
' Print the error information
PRINT "Error code = ", err[1]
PRINT "Error message = ", err[2]
PRINT "Line number = ", err[3]
PRINT "Column number = ", err[4]
```

More Information

Normally, an error during program execution stops the program and prints out the error, so there's little reason to call GetLastError(). But if you're using ON ERROR CALL to set up an error handler, calling GetLastError becomes very useful.

It returns an array with five elements: the error code (an integer), the error text (a string), the line number where the error occurred (and integer), the column number where the error occurred (also an integer), and the file name (e.g. the main BASIC file or the include file).

See Also

ClearLastError, Error Handling

6.18 Instr

Description

Finds the first occurrence of one string within another.

Example

```
str = "ABCDEABCDE"
print Instr(str, "C") ' Prints 3
print Instr(str, "X") ' Prints 0
print Instr(4, str, "C") ' Prints 8
```

More Information

Instr takes two or three arguments. If it is given two arguments, they must both be strings. Instr returns the 1-based position of the second string in the first string. If the second string is not found, Instr returns 0.

If Instr has three arguments, then the first is the 1-based position in the string at which to start the search.

This table summarizes the return values, assuming the call pos = Instr(start, string1, string2):

If	**Instr returns**
string1 is empty	0
string2 is empty	*start*
string2 is not found	0
string2 is found within *string1*	Position in string1 where match begins
start > length of *string1*	0

6.19 Int

Description

Convert a floating point number to an integer.

Example

```
a = 5.0
PRINT a / 2 ' Prints 2.5
a = Int(a)  ' Converts a to an integer
PRINT a / 2 ' Prints 2, because of integer arithmetic
```

More Information

The float is rounded towards zero and then its internal representation is converted to an integer.

See Also

Float

6.20 LBound

Description

Returns the smallest index of an array for a given dimension.

Example

```
OPTION BASE 1
DIM a[5, 7]
PRINT LBound(a) ' Prints 1
PRINT LBound(a, 1) ' Prints 1
PRINT LBound(a, 2) ' Prints 1
PRINT LBound(a, 3) ' Prints 0; there is no 3rd dimension
```

```
OPTION BASE 0
DIM a[5, 7]
PRINT LBound(a) ' Prints 0
PRINT LBound(a, 1) ' Prints 0
PRINT LBound(a, 2) ' Prints 0
PRINT LBound(a, 3) ' Prints 0; there is no 3rd dimension
```

More Information

LBound accepts one or two arguments. The first argument is an array. If there is no second argument, LBound returns the smallest index for the first dimension of the array. If there *is* a second argument, it specifies which dimension of the array to return the smallest index of.

The second argument is always 1-based, even if OPTION BASE has been set to 0.

Note that the smallest index is just the value of OPTION BASE. If the Option Base is 1, LBound will return 1 (for valid array and dimension arguments). If the Option Base is 0, LBound will always return 0.

See Also

UBound, OPTION BASE

6.21 Left

Description

Returns the leftmost n characters of a string.

Example

```
PRINT Left("Hello", 3) ' Prints Hel
PRINT Left("Hello", 6) ' Prints Hello
```

See Also

Len, Mid, Right

6.22 Len

Description

Returns the length of a string.

Example

```
PRINT Len("Hello") ' Prints 5
```

See Also

Left, Mid, Right

6.23 Max

Description

Returns the larger of two values.

Example

```
PRINT Max(5, 8.2) ' Prints 8.2
PRINT Max("Hello", "Goodbye") ' Prints Hello
```

See Also

Min

6.24 Mid

Description

Returns characters from the middle of a string.

Example

```
PRINT Mid("Hello", 1, 3) ' Prints Hel
PRINT Mid("Hello", 2, 3) ' Prints ell
PRINT Mid("Hello", 2) ' Prints ello
```

More Information

Mid can take two or three arguments. The first argument is the string. The second argument is the position in the string at which to start getting characters (the first character is position 1). The optional third argument is the number of characters to get. If the third argument is omitted, all characters from the starting position to the end of the string are returned.

See Also

Left, Len, Right

6.25 Min

Description

Returns the smaller of two values.

Example

```
PRINT Min(5, 8.2) ' Prints 5
PRINT Min("Hello", "Goodbye") ' Prints Goodbye
```

See Also

Max

6.26 Norm

Description

Returns the Euclidean norm of an array.

Example

```
v = {3, 4}
n = norm(v) ‘ n is set to sqrt(3 * 3 + 4 * 4), or 5
```

More Information

The Norm() function calculates the Euclidean norm of a matrix or the magnitude of a vector. Like the Pythagorean theorem, it adds the squares of all the elements in the matrix and returns the square root.

See Also

Vector and Matrix Arithmetic

6.27 Rand

Description

Returns a pseudo-random integer.

Example

```
PRINT Rand() MOD 10 ' Prints an integer between 0 and 9
```

More Information

The integer returned by Rand is between 0 and the maximum positive value an integer can hold. That maximum is platform-dependent, but will always be at least 32,767. The best way to get a random number in a given range is to take the modulo of the value returned by Rand().

Every time a program runs, the sequence of numbers returned by Rand() will be the same, i.e. they're pseudo-random, but reproducible. Usually, you will want the sequence to be different every time. In that case, call the Randomize() function once at the beginning of your program.

See Also

Randomize

6.28 Randomize

Description

Initializes the pseudo-random number generator.

Example

```
CALL Randomize
```

```
CALL Randomize(45)
```

More Information

Seeds the pseudo-random number generator (RNG). If no argument is given, the RNG is seeded with an unpredictable value and subsequent calls to Rand() will return unpredictable values. If an integer argument is given, the RNG will be seeded with that value. Two different initializations with the same seed will generate the same succession of results in subsequent calls to Rand.

See Also

Rand

6.29 Pow

Description

Calculates x^y.

Example

```
PRINT POW(2, 3) ' Prints 8
PRINT POW(2, 0.5) ' Prints 1.41421
```

More Information

Before version 1.21, the ^ operator calculated the exponent. ^ is now being used for bitwise XOR, so exponentiation is done with the POW() function

6.30 Right

Description

Returns the rightmost n characters of a string.

Example

```
PRINT Right("Hello", 3) ' Prints llo
PRINT Right("Hello", 6) ' Prints Hello
```

See Also

Left, Len, Mid

6.31 Round

Description

Rounds a number to the nearest integer.

Example

```
PRINT Round(5.6) ' Prints 6
PRINT Round(-3.5) ' Prints -4
```

More Information

The internal representation of the returned value is a float. Round() is different from Int() in two ways: first, it rounds the number whereas Int() discards the fractional part; second, it returns a float whereas Int() returns an integer.

See Also

Truncate

6.32 Sin

Description

Returns the sine of an angle, which is given in radians.

Example

```
PRINT Sin(3.14159 / 2) ' Prints 1
```

See Also

Acos, Asin, Atan, Cos, Tan

6.33 Size

Description

Returns the number of elements in an array for a given dimension.

Example

```
DIM a[5, 7]
PRINT Size(a) ' Prints 5
PRINT Size(a, 1) ' Prints 5
PRINT Size(a, 2) ' Prints 7
PRINT Size(a, 3) ' Prints 0; there is no 3rd dimension

b = 3
PRINT Size(b) ' Prints 0, since b is not an array
```

More Information

Size accepts one or two arguments. The first argument is an array. If there is no second argument, Size returns the number of elements for the first dimension of the array. If there *is* a second argument, it specifies which dimension of the array to return the number of elements of.

See Also

UBound

6.34 Sqrt

Description

Returns the square root of a number.

Example

```
PRINT Sqrt(5) ' Prints 2.23606
PRINT Sqrt(-3.5) ' Prints nan (Not a Number)
```

More Information

Returns a float.

6.35 StrComp

Description

Returns -1, 0, or 1, based on the result of a string comparison.

Example

```
PRINT StrComp("HELLO", "HELLo") ' Prints -1
PRINT StrComp("HELLO", "HELLo", 0) ' Prints -1
PRINT StrComp("HELLO", "HELLo", 1) ' Prints 0
```

More Information

The third argument is optional and indicates whether the string comparison is case insensitive. If there is no third argument, or if the third argument is 0, the comparison is case-sensitive. If the third argument is non-zero, the comparison is case-insensitive.

The return value is zero if the strings are equal, -1 if the first string is less than the second, and 1 if the first string is greater than the second.

6.36 Tan

Description

Returns the tangent of an angle, which is given in radians.

Example

```
PRINT Tan(3.14159 / 4) ' Prints 1
```

See Also

Acos, Asin, Atan, Cos, Sin

6.37 ToLower

Description

Returns a copy of a string converted to lower case.

Example

```
PRINT ToLower("HELLO") ' Prints hello
```

More Information

Returns a string that is a copy of the argument, except that letters are converted to lower case. The argument is unchanged.

See Also

ToUpper

6.38 ToUpper

Description

Returns a copy of a string converted to upper case.

Example

```
PRINT ToLower("hello") ' Prints HELLO
```

More Information

Returns a string that is a copy of the argument, except that letters are converted to upper case. The argument is unchanged.

See Also
ToLower

6.39 Transpose
Description
Swaps the rows and columns of an array.

Example
```
a = Transpose({{1, 2, 3}, {4, 5, 6}})
' a is now {{1,4},{2,5},{3,6}}
b = Transpose({1, 2, 3})
' b is now {{1}, {2}, {3}}
c = Transpose(b)
' c is now {1, 2, 3}
```

See Also
Vector and Matrix Arithmetic

6.40 Truncate
Description
Rounds a number towards zero.

Example
```
PRINT Truncate(5.6) ' Prints 5
PRINT Truncate(-3.5) ' Prints -3
```

More Information
Removes the fractional portion of a number. Returns a float. Truncate() is different from Int() in that it returns a float whereas Int() returns an integer.

See Also
Round

6.41 UBound
Description
Returns the largest index of an array for a given dimension.

Example
```
OPTION BASE 1
DIM a[5, 7]
PRINT UBound(a) ' Prints 5
PRINT UBound(a, 1) ' Prints 5
PRINT UBound(a, 2) ' Prints 7
PRINT UBound(a, 3) ' Prints 0; there is no 3rd dimension

b = 3
PRINT UBound(b) ' Prints 0, since b is not an array
```

```
OPTION BASE 0
DIM a[5, 7]
PRINT UBound(a) ' Prints 4
PRINT UBound(a, 1) ' Prints 4
PRINT UBound(a, 2) ' Prints 6
PRINT UBound(a, 3) ' Prints -1; there is no 3rd dimension

b = 3
PRINT UBound(b) ' Prints -1, since b is not an array
```

More Information

UBound accepts one or two arguments. The first argument is an array. If there is no second argument, UBound returns the largest index for the first dimension of the array. If there *is* a second argument, it specifies which dimension of the array to return the largest index of.

The second argument is always 1-based, even if OPTION BASE has been set to 0.

Note that the largest index is dependent on the OPTION BASE. If an array has 3 elements and the Option Base is 1, UBound will return 3. If the Option Base is 0, UBound will return 2. In the last two examples, when UBound returns 0: if the Option Base were 0, UBound would return -1. If you have set the Option Base to 0, you might find the Size function to be more useful than UBound.

See Also

LBound, Size, OPTION BASE

6.42 Val

Description

Converts a string to a number.

Example

```
a = Val("123.456")
```

More Information

Val ignores leading whitespace in the string. When it gets to the first digit or sign (+ or -) it begins converting the string to a floating point number and continues until it reaches the first character that cannot be parsed as part of a number. Val supports exponents (e.g. “1e-3”). If no valid number was found, Val returns 0.