

GameScript SDK

Copyright (C) 2005 John Judnich

Overview

GameScript SDK provides you with the power to compile and execute embedded script programs within your BlitzBasic application. Embedded scripts can be used as plug-ins, mod-able content, and much more. GameScript supports multi threading scripts, which allows you to run more than one script *at the same time* in your application.

GameScript scripts are normally filed as “.gs” files and will be automatically compiled when necessary. When distributing your application, you can hide your script source code by simply deleting the “.gs” files, leaving the remaining compiled GameScript executables (.gsx). The “.gsx” files will exist for each “.gs” file after they have been run once (which automatically calls the GameScript compiler) or have been compiled manually with the GameScript compiler. This is extremely handy also if you have developed a plug-in for an application that utilizes GameScript, and you would like to protect your source code.

GameScript has very few limitations and is a full-featured programming language, including features such as custom data types, function overloading, standard flow control, and an optimizing compiler. Some of the features GameScript includes are:

- Easy to use, structured, programmer-friendly language syntax
- Easy-to-use programmer interface
- Custom data types
- Functions with recursion support
- Local, Global, Constant, and Static variables
- A fast compiler
- Ultra-fast multi-threading virtual machine
- Protect your scripts – Since GameScript natively runs only GameScript executables (.gsx), there's no need for your script source code (.gs) to be included with your final distribution
- Automatic compile management – GameScript source code (.gs) is compiled to GameScript executable (.gsx) only when changes have been made.

Before you begin learning the details of GameScript's syntax, you may want to try out the examples which are provided along with the GameScript library. They are located in the “\examples” folder under GameScript's main folder, for your convenience.

This reference manual provides a basic overview of the BASIC language, as well as reference tables containing data on keywords, etc. If you are already familiar with BASIC, you may only need to read a small portion of this manual to learn the slight differences between

GameScript and traditional BASIC. This manual should provide enough information you need to understand how GameScript works. This manual also provides documentation on integrating GameScript into your application (in the section labeled “Integrating GameScript In Your Application”).

Here is a short example partially demonstrating GameScript's syntax usage.

```
Type Vehicle
    Field Speed:Float
EndType

Local Car:Vehicle

Car = New Vehicle 'Create a new instance of Vehicle and store it's pointer in Car

Car.Speed = GetSpeed(Car) + 10

Function GetSpeed:Float(v:Vehicle)
    Return v.Speed
End Function
```

Several aspects in the GameScript language are significantly different from “traditional” BASIC. In addition to variable declaration being a requirement, variable types are defined in a different way. Instead of appending a symbol that represents a specific data type to the end of the variable name (such as # for float, % for int, \$ for string, etc.), a colon symbol is appended (:), followed by the *name* of the data type. Please read the following sections of this manual to learn more about GameScript's syntax.

Variables

All variables used must be declared with **Local**, **Global**, **Const**, or **Static** (or **Field** when within a type definition). Following the variable name must be a colon (:), and the variable's data type. The data type may be **Integer**, **Float**, **String**, or the name of a custom type definition. Variables may be given an initial value when defined. Initial values must be either a number or a string (expressions are not allowed). **Const** variables *must* be given a initial value when defined. See *Table 1* for examples.

While **Global** variables should generally be declared *outside* of any functions, GameScript still allows it to be declared anywhere, although it makes no difference whether it's declared inside or outside of a function. Remember: Don't declare a **Global** variable when you'll only use it in one function; declare a **Static** variable within the function instead. **Static** variables (be careful not to confuse “static” with “constant”) are a cross between Global and Local in a way; they can only be accessed from within the function they were declared (like Local variables), but their value is preserved the next time the function is called (like Global variables).

Note: “Integer”, “Float”, and “String” may be abbreviated to “Int”, “Flt”, and “Str” respectively, if you prefer.

Valid Variable Declaration	Invalid Variable Declaration
Local dist:Int	Local dist% = 3 '% is invalid
Static length:Float = 10	Static length# '# is invalid

Valid Variable Declaration	Invalid Variable Declaration
Global a:String = "Hello" Global b:String	Global a:String = Hello 'No quotes around text Global b\$ = "Hello" '\$ is invalid
Const txt:String = "Hi" Const num:Float = 10.2	Const txt:String 'No value is given Const num:Float = 5 + 10.2 'Init. expressions

Table 1 – Examples of valid and invalid variable declaration styles

When assigning variables and calculating mathematical expressions with varying data types, remember that each individual variable type has a set priority. An expression adding a string to a float, for example, will result in a string. Similarly, adding a int to a float will result in a float. Here is a table listing variable priorities:

Variable Type	Priority
Integer	<i>Float+Int = Float, Int+Int = Int, Str+Int = Str</i>
Float	<i>Int+Float = Float, Float + Float = Float, Str+Float = Str</i>
String	<i>Int+Str = Str, Str + Str = Str, Float+Str = Str</i>

Custom Type Definitions

Custom types are extremely important in *any* language to maintain structure. Custom type support (the equivalent of C structs) is one step further toward object oriented programming. To define a custom type, use the **Type**, **Field**, and **EndType** keywords, as shown:

```
Type Vehicle
  Field Speed:Float
  Field Width:Float, Height:Float
End Type
```

"*Type Vehicle*" marks the beginning of the *Vehicle* type definition. Between **Type** and **EndType** are **Field** keywords, similar to the **Local** keyword in that it defines a "local" variable, except **Field** variables are relative to their custom type (*Vehicle*, in this case). Like the **Local** variable declarations, custom type variable declarations may not be assigned an initial value.

Now, defining a variable which makes use of a custom type is as simple as defining any variable. Instead of an Int, Float, or String being specified as the variable type, simply use the custom type name. In the example above, that would be *Vehicle*. Example:

```
Local car:Vehicle
```

However, custom type variables are not *exactly* like standard Int, Float, or String variables. While a Float variable may be used appropriately as long as it is defined, custom types must be created first. In reality, a custom type variable is a pointer (or reference) of a custom type. Until the variable is assigned to a instance of the custom type, it is "empty", and it's fields may not be accessed. For example:

```
Type Vehicle
  Field Speed:Float
  Field Width:Float, Height:Float
```

```

End Type

Local car:Vehicle

car = New Vehicle
car.Speed = 120

```

When *car* is first initialized, it is pointing (referring) to *nothing* (**Null**). "*car* = New Vehicle" points *car* to a newly created Vehicle instance created with the **New** keyword. If you are familiar with C/C++, think of custom type variables as pointers to structs, and the "." as a "->" operator.

When using custom type variables alone (not accessing it's fields), custom types act exactly like a normal variable. You can assign type objects to them, and even test two custom type variables to see if they're pointing to the same object. You can't, however, use operators such as +, -, /, etc. on them, which makes no sense because they really don't have a value that you should be concerned about – just a reference to an actual type object.

Keywords

GameScript supports a number of keywords, each of which perform basic calculations/manipulations and control program flow. Table 2 lists every keyword supported by GameScript* and it's description. For more information that what is provided here, please refer a book or other source on BASIC. *Note: When specifying keyword syntax usage in Table 2, text enclosed in angled brackets "<...>" are places to "insert" text of the type specified. Square brackets "[...]" indicate that their contents is optional.*

Keyword	Description
Global	Initializes a variable, as a "Global", which means that it will be accessible from any point in the program.
Const	Initializes a variable, as a "Constant", which means that it will be accessible from any point in the program, and may not be modified.
Local	Initializes a variable, as a "Local", which means that it will be accessible from only the function which contains it.
Static	Initializes a variable exactly as "Local" does, except the value of the variable will remain as it is then next time it's containing function is called.
Type	Syntax: "Type <custom type name>". The marker of the beginning of a type definition. Between "Type" and "EndType" keywords should be only "Field" statements containing variable definitions. See the section labeled "Variables" for more information.
EndType	The marker of the end of a type definition. Between "Type" and "EndType" keywords should be only "Field" statements containing variable definitions. See the section labeled "Variables" for more information.
Field	Defines variables which belong to a custom type. See the section labeled "Variables" for more information.
If	Part of an If .. Then statement. If .. Then statements control program flow by branching program flow depending on a conditional expression. See the section labeled "Conditional Statements" for more information.
Then	Part of an If .. Then statement. If .. Then statements control program flow by

Keyword	Description
	branching program flow depending on a conditional expression. See the section labeled “Conditional Statements” for more information.
Else	Part of an If .. Then statement. If .. Then statements control program flow by branching program flow depending on a conditional expression. See the section labeled “Conditional Statements” for more information.
EndIf	Part of an If .. Then statement. If .. Then statements control program flow by branching program flow depending on a conditional expression. See the section labeled “Conditional Statements” for more information.
New	Syntax: “<custom type variable> = New <custom type name>”. Creates a new instance of the given type and puts the pointer (reference) to it in <custom type variable>. After calling this command, <custom type variable> will be Null .
Delete	Syntax: “Delete <custom type variable>”. Deletes an instance of a custom type, specified by <custom type variable>.
Let	A useless command only for compatibility with other BASIC language styles. Syntax: “Let <variable> = <expression>”. This is <i>exactly</i> the same thing as: “<variable> = <expression>”.
End	This simply terminates (ends) the script, halting all execution.
And	Syntax: “<expression> And <expression>”. Applies the logical operator “And” to two expressions. Example: “If <expression> And <expression> Then ...”
Or	Syntax: “<expression> Or <expression>”. Applies the logical operator “Or” to two expressions. Example: “If <expression> Or <expression> Then ...”
XOr	Syntax: “<expression> Xor <expression>”. Applies the logical operator “And” to two expressions. Example: “If <expression> Xor <expression> Then ...”. Refer to a book or other source on general programming for more information.
Not	Syntax: “Not <expression>”. Inverts the True/False state of an expression. Example: “If Not <expression> Then ...”
Mod	Syntax: “<int expression> Mod <int expression>”. Mod is the BASIC equivalent of the C <i>modulus</i> operator “%” (also supported by GameScript). This performs an integer division, calculating the remainder rather than the quotient.
Shr	Syntax: “<expression> Shr <expression>”. Performs a right binary shift operation on the expression on the left of “Shr”, repeated the number of times specified by the expression on the right of “Shr”. Example: “a = b Shr 8”. This example shifts “b” right 8 times and stores the result in “a”.
Shl	Syntax: “<expression> Shl <expression>”. Performs a left binary shift operation on the expression on the left of “Shr”, repeated the number of times specified by the expression on the right of “Shr”. Example: “a = b Shl 8”. This example shifts “b” left 8 times and stores the result in “a”.
True	This is actually not a keyword, but a reserved constant. This constant indicates a <i>true</i> expression or value (1).
False	This is actually not a keyword, but a reserved constant. This constant indicates a <i>false</i> expression or value (0).
Null	This is actually not a keyword, but a reserved constant. This constant indicates an unassigned custom type variable which has not been assigned any instance of a custom type.
Function	Syntax: “Function <function name><variable type>(<variable

Keyword	Description
	declaration>, <variable declaration>, ...)” Declares the beginning of a function. Example: “Function GetSpeed:Float(car:Vehicle)”. For more information, see the section labeled “Functions”.
Return	Syntax: “Return <expression>”. Returns out of a function and resumes code execution before it was called, returning the value specified. For more information, see the section labeled “Functions”.
EndFunction	Marks the end of a function. If the execution point reaches an “EndFunction” before it reaches a “Return” statement, it returns 0 (if the function is a Float or Int type) or a blank string (if the function is a String type). For more information, see the section labeled “Functions”.
While	Syntax: “While <condition>”. When used, the code between a “While” and “WhileEnd” will be repeatedly executed while <expression> is True. For more information, see the section labeled “Loops”.
Wend, WhileEnd	These two synonymous keywords, “Wend” and “WhileEnd”, do exactly the same thing; they marks the end of a “While” loop. For more information, see the section labeled “Loops”.
Repeat	This keyword marks the beginning of a “Repeat..Until” or “Repeat..Forever” loop. For more information, see the section labeled “Loops”.
Forever	This marks the end of a “Repeat..Forever” loop. Any code between “Repeat” and “Forever” will be repeatedly executed for ever unless an Exit command is used somewhere in the loop to break out. For more information, see the section labeled “Loops”.
Until	Syntax: “Until <condition>” This marks the end of a “Repeat..Until” loop. Any code between “Repeat” and “While” will be repeatedly executed until <expression> is True. For more information, see the section labeled “Loops”.
For	Syntax: “For <variable> = <number1> To <number2> [Step <number3>]”. This begins a “For..Next” loop. A “For..Next” loop executes the code between “For” and “Next” a specific number of times, each with <variable> assigned a different value which changes from <number1> to <number2> incrementing <number3> units each time. For more information, see the section labeled “Loops”.
To	Part of a “For..Next” loop specifying the range of incremental counting. For more information, see the section labeled “Loops”.
Step	Part of a “For..Next” loop specifying the range of incremental counting. For more information, see the section labeled “Loops”.
Next	Ends a “For..Next” loop. For more information, see the section labeled “Loops”.
Exit	Breaks out of a loop earlier than normal, or prematurely exits out of a “Case” statement. For more information, see the section labeled “Loops” and “Conditional Statements”. <i>Note: If “Exit” is used outside of loops and “Select..Case” keywords, it will cause an error (“Cannot exit here”).</i>
Select	Syntax: “Select <variable>”. Begins “Select .. Case” conditional statements. See the section labeled “Conditional Statements” for more information.
Case	Syntax: “Case <value>, [<value>, ...]”. See the section labeled “Conditional Statements” for more information.
Default	Syntax: “Default”. See the section labeled “Conditional Statements” for more information.
EndSelect	Syntax: “EndSelect”. Ends “Select .. Case” conditional statements. See the section labeled “Conditional Statements” for more information.

Keyword	Description
Goto	Syntax: "Goto <label>". Relocates the program execution point to another position in the code. Goto only allows relocations within the same function (you cannot "goto" a place outside of the current function). <label> specifies the goto label to jump to. To insert a label somewhere in your code, type the name of the label, preceded with the "at" symbol (@). Note: Extensive use of Goto is not recommended, as it can quite easily turn your code into unstructured "spaghetti" code. Use only when absolutely necessary.

Table 2 – Descriptions of keywords supported by GameScript

Operators

GameScript includes support for most standard BASIC operators, as well as a few (optional) operators from C/C++. Operators supported by GameScript are:

<, <=, >=, >, =, <>, !=, ==, <<, >>, *, /, +, -, (,), %

Either "==" (C/C++) or "=" (BASIC) may be used for equality checks, and either "!=" (C/C++) or "<>" (BASIC) may be used for inequality checks. The "<<" and ">>" operators are the C/C++ equivalent of **Shl** and **Shr** (also supported). The "%" operator is the C/C++ equivalent of **Mod** (also supported). Most of the other operators are used to form mathematical expressions and conditional expressions (parenthesis are used to group expressions, which contain +, -, /, *, etc.)

Most operators work only with integer values or float values, although the "+" operator supports the addition of all types except custom types (not their fields – a custom type field is treated just like a float, int, or whatever it is declared as). Using the "+" operator, you could, for example, add a integer value to a string of text. The resulting value would be a string, since strings are higher priority than ints or floats. The same is true about float values being above integer values; a float plus an integer results in a float, so that no precision is lost. For more information, see the "Variables" section of this manual.

Operators also have their priority levels. For example, multiplication and division overrides addition and subtraction. Here is a table listing the standard operator priorities GameScript uses:

Operator	Priority
Boolean compare <, <=, >=, >, =, <>, !=, ==	1
Addition and subtraction +, -	2
Multiplication and division *, /, %	3
Binary shift <<, >>	4

For example, the expression "a + b * c / d << e" could mean one of many different translations, when no parenthesis are given. It could mean "(a + b) * (c / d) << e", "a + (b * c) / (d << e)", or almost any other grouping. The operator priorities instruct the compiler what to do in a situation like this. Since * and / operators are higher than + and -, it is interpreted as "(a + ((b * c) / d)) << e". Since "<<" is highest in priority, everything left of the operator is used, not just "d", or "c / d". "*" is higher than "+", so it is grouped accordingly. Generally, operators should be grouped whenever you're in doubt of what will happen.

Conditional Statements

GameScript supports **If .. Then**, and **Select .. Case** conditional statements. A standard one-line "If" statement should look like: (square brackets indicate optional text)

```
If <condition> Then <statement> [Else <statement>]
```

In many cases, more than one statement is needed. In this case, the *multi-line* "If .. Then" statement format is used. The format of a multi-line "If .. Then" statement is:

```
If <condition> Then
    <multiple statements>
Else
    <multiple statements>
EndIf
```

Or:

```
If <condition> Then
    <multiple statements>
EndIf
```

Note: GameScript does not support "Elseif" "If" statement behavior. Try using multiple "If" statements or "Select .. Case" statements instead.

The **Select .. Case** conditional statements are best suited for branching program flow to many different sections of code based on the value of one variable. Example:

```
Select <variable>
Case <expression>
    <statement>
Case <expression>, <expression>, ...
    <statement>
...
Default
    <statement>
EndSelect
```

*Note: The "Default" case must always be after all other "Case" keywords. You may also use the **Exit** keyword to exit out of a "Case" (or "Default") prematurely. Example:*

```
Select <variable>
Case <expression>
    <statement>
    Exit 'Jumps down to EndSelect
    <statement>
Case <expression>, <expression>, ...
    <statement>
...
Default
    <statement>
EndSelect
```

Note: If "Exit" is used outside of loops and "Select..Case" keywords, it will cause an

error ("Cannot exit here"). For more information about "If .. Then" and "Select .. Case" statements, please refer to a book or other source on BASIC syntax.

Loops

Conditional (and unconditional) loops are essential to all programming languages. Without some way to repeat procedures under certain circumstances, a program would be of little use; it would simply run quickly and end. Conditional loops provide a way to repeatedly execute a group of code until a certain condition is met. GameScript supports several loops.

The **While .. WhileEnd** loop executes the code between the "While" keyword and the "WhileEnd" keyword *while* a condition is met. For example:

```
While done = False
    <statement>
    <statement>
    ...
WhileEnd
```

The example above would execute the code between the "While" and "WhileEnd" as long as "Done" equals "False". *Note: Optionally, "Wend" may be used as an abbreviation of "WhileEnd", if you prefer.*

The **Repeat .. Until** loop executes the code between both keywords *until* the condition is met. For example:

```
Repeat
    <statement>
    <statement>
    ...
Until done = True
```

The example shown above executes the code between "Repeat" and "Until" constantly *until* "done" equals "True".

GameScript supports only one unconditional loop: **Repeat .. Forever**. This loop executes the code between "Repeat" and "Forever" indefinitely. The only way a "Repeat .. Forever" loop can stop is with the "Exit" keyword.

The **Exit** keyword may be used anywhere within a loop. "Exit" breaks out of the current conditional or unconditional loop immediately. *Note: If "Exit" is used outside of loops and "Select..Case" keywords, it will cause an error ("Cannot exit here").*

Another conditional loop GameScript supports is the **For .. Next** loop. This loop repeats the code between the "For" and "Next" for a specified number of times. Example:

```
For count = 1 To 100 Step 2
    <statement>
    <statement>
    ...
Next
```

That example runs the code between the "For" and "Next", each time "count" is set to a different number. The first time, "count" is set to 1. The next time it steps ahead 2 (specified

by “Step 2”), and then runs the code with “count” at 3. This continues until “count” reaches 100. *Note: The “Step” option is optional. When left out, the step value defaults to 1.*

Functions

GameScript functions are basically the same as standard BASIC functions, differing in type declaration. Like all GameScript type declarations, a colon (:) followed by the name of the data-type is used instead of cryptic symbols specifying a data-type (GameScript also doesn't require you to redundantly specify the data-type every time you use the variable). Here's an example of a GameScript function:

```
Function Dist:Float(x:Float, y:Float)
  Local d:Float
  d = Sqr(x*x + y*y)
  Return d
End Function
```

As you can see, the above function declares two Float parameters: x, and y. The colon followed by “Float” after the function name (Dist) indicates that the function will return a Float value. You may omit this function data-type specification when your function will return no value. Optionally, you could use the special function type called “Void”, which specifies a “null” return value. For example, this:

```
Function DoStuff()
  '...
  '...
End Function
```

Is exactly the same as this:

```
Function DoStuff:Void()
  '...
  '...
End Function
```

Function calls can be performed in one of two styles. The first style uses the function in standard BASIC command format, with no parenthesis surrounding the parameters. Example:

```
DisplayText "Hello!"
```

The above example would call the function “DisplayText” (if one existed), passing one parameter: “Hello!”. This format may only be used as a stand-alone statement (not contained within an expression). The second function call style uses parenthesis surrounding the functions parameters, and may be used either as a single statement (like the previous style), or within an expression. Example:

```
DisplayText("Hello!") 'This does the same as the previous example (different style)
num = Dist(5, 3) + 2  'This style may be used within expressions
```

If you prefer, you can use just the second method. Or, you might want to use the first

method for single statement function calls, and the second style when used within expressions. Basically, the choice of style is up to you.

Integrating GameScript In Your Application

Now that you've learned how to program in GameScript, it's time to integrate it into your application. GameScript integration is really very easy; here are a few simple step-by-step instructions to install and prepare GameScript for use with your BlitzBasic applications:

1. Copy the file "*TimeStamp.decls*" (located in the main GameScript SDK folder) into your BlitzBasic userlib folder (look for a folder named "*userlibs*" under your BlitzBasic installation). This file is required to compile project(s) using the GameScript SDK.
2. Run "example1.bb", located in the "\Examples" folder under GameScript's main folder. If the above step has been completed properly, you should see a successful compile message, followed by some output from the sample script.
3. Copy "GameScript Compiler.bb", "*GameScript VM.bb*", and "*GameScript Functions.bb*" into the project folder of the application(s) you would like to integrate GameScript into.
4. Include all three of the above files into your project using BlitzBasic's "Include" command.

If you followed the above steps correctly, your application should be able to use all of GameScript's features. The three required include files have the following purposes:

- "GameScript Compiler.bb" contains functions that compile GameScripts (.gs) to GameScript Executables (.gsx), as well as functions returning helpful messages in case of errors in the script. The only functions you should use from this file are the error/warning message functions.
- "GameScript VM.bb" includes the GameScript Virtual Machine, which executes GameScript Executables (.gsx) produced by the GameScript Compiler. This contains functions for terminating, pausing, etc. execution threads.
- "GameScript Functions.bb" allows you to easily expand GameScript to suit your application's needs. You can add functions to this file which will be accessible from your GameScripts.

GameScript SDK uses two types of files; **.gs** files contain source code in the GameScript syntax which you type into the computer, while **.gsx** files contain binary executable data only the GameScript VM can understand.

The GameScript VM processes *only* **.gsx** files internally (although if it's given a .gs file it will compile it to .gsx automatically). The Virtual Machine is responsible for the actual execution of the script program. With the VM, you can issue runtime operations, such as terminating a script program (called a *thread*), etc.

The GameScript Compiler processes *only* **.gs** files. The Compiler converts the human-readable scripts (.gs) to GameScript Executables (.gsx) which the GameScript VM can use. To reduce loading times of your application, the GameScript Compiler automatically detects whether it really is necessary to recompile a script file. If the file has already been compiled to a .gsx file, and it's up-to-date, it doesn't recompile the script.

GameScript SDK comes with several examples (found in the "Examples" folder within the GameScript SDK installation folder) demonstrating the use of the GameScript SDK. You'll probably want to try these out to get a better understanding of how GameScript works.

Remember: When running GameScripts with your application in debug mode, execution will be about **10 times slower**. If scripts seem running unusually slow, make sure debug is off. A general GameScript should run around 3-5 times *slower* than a native blitzbasic EXE.

License

This software is provided 'as-is', without any express or implied warranty. In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely, subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.