GLBasic Programmers Reference Guide

(Second Edition – First Revision)

A programmers guide to using GLBasic commands. Also contains 11 tutorials.

By Nicholas Kingsley

Tutorials by BigSofty and Gernot Frisch.
Updated and revised by Cliff3D and Nicholas Kingsley
Android signing section originally posted by Dabz in the GLBasic forums

Based on the contents of the GLBasic Wiki
(http://wiki.just-do-i-t.com/index.php?title=GLBasic_Wiki)
by Nicholas Kingsley, Cliff3D et al

ISBN #: 978-1-4477-2272-4

Designed By Nicholas Kingsley

Revision : 4.0.0.4

For errors and omissions, I can be contacted at njk@un-map.com

Last revised on : 16/02/12

Introduction

Welcome to the GLBasic Programmers Reference guide. This weighty tome is designed to help the beginner and more experienced GLBasic user get the most out of the language.

Covering topics such as sound, music, sprites, 3D objects as well as revised tutorials this should help everyone get to grips with the language.

Tutorials have been revised from those that come with the language, which take into account the improvements that GLBasic has had since it was originally written.

The language details have been revised from the Wiki entries (at **http://wiki.just-do-i-t.com/index.php?title=GLBasic_Wiki**)

V9 of GLBasic has introduced a new platform to the fold : The Palm Pre. It has also updated the Windows input system, updated the editor and added a few new commands

PART #1

GLBasic Tutorials

Tutorial #1 (General)

Hello and welcome to GLBasic, your new programming language.

Here you will learn how to convert your ideas into real programs.

A little word of warning: GLBasic cannot do wonders, it has its strengths and weaknesses. A good programmer excels by utilising the strengths of a language and working around its weaknesses.

Having said that, there are many areas where GLBasic excels - sprites for example. You will find sprite rotations, alpha blending and zooming, while difficult in some languages, are simple to implement in GLBasic. To reach a stable frame rate, GLBasic can easily be slowed down to a certain number of frames per second to match any target platform.

A program is made up of very simple commands that individually do almost nothing. Combining these simple commands to make them do something powerful is the difficult thing about programming.

Programs can only:

- Declare variables (named containers for data)
- Make jumps to other parts of the program
- Manipulate the data held in variables
- Transfer variables to functions
-

A compiler takes these commands and converts them into a language the computer can understand.

General

GLBasic is a variant of the BASIC programming language and a very easy one at that. It has its own syntax and if you don't keep within this syntax 100% of the time, the GLBasic compiler will display an error asking you to correct your syntax before it will compile your code.

GLBasic finishes every command with a ';' (semicolon) or a new line, although this is usually optional, unless you want to put more than one command on a line.

Some examples of using a semicolon include :

```
a%=1;
test(); PRINT “Hello”,0,0; PRINT “More text”,0,100,TRUE
```

Semicolons should not be used to separate variable declarations in functions, although they can be used when defining variables in a type.

All commands in GLBasic must be written in capital letters. The editor will automatically convert the GLBasic commands you type to upper-case even if you do not. If it fails to convert a command, check that you have spelt it correctly.

Every character to the right of '//' on a line will be seen as a comment within the code. Comments will be ignored when the compiler converts your code into a program. This is handy for adding notes within your source. You can uncomment or comment several lines at once within the GLBasic IDE by selecting the lines and then pressing the Comments toolbar button:

GLBasic uses hidden surfaces (Double Buffering) for graphical display. If you code something in GLBasic you will see nothing before your program calls the `SHOWSCREEN` command.

`SHOWSCREEN` takes the current back buffer, makes it visible, then creates a new back buffer with a predefined image or a cleared image. If you don't see any result from your program, the first thing to check is whether you have forgotten to add the `SHOWSCREEN` command to it.

Think of Double Buffering as two sheets of paper, one on top of the other where you draw on the bottom sheet. Once you're finished drawing, you swap the sheets and in the process erase the new bottom sheet. This hides the actual drawing of the sheet from the user and in the process eliminates screen flicker.

Our First Program

Start the editor and select "Create a new project" in the wizard. Type "Hello World" within the "Name of Project" box and press OK.

Then type the following program into the editor's window.

```
// ------------------------------- //
// Project: Hello World
// My First Program
// ------------------------------- //
PRINT "HELLO WORLD!" ,100,100
SHOWSCREEN
MOUSEWAIT
END
```

Save all your work with the Double-disk toolbar button:

or use the menu option "File/Source code file/Save all".

Compile the program using the Sand-bucket toolbar button:

or the menu option "Compiler/Build".

Finally, start the program with either the "Start" toolbar button:

or the menu option "Compiler/Start".

For more detailed information on the editor, please refer to the *GLBasic User Guide* book by Nicholas Kingsley, which is available from http://www.lulu.com/product/paperback/glbasic-user-guide/12674927

You should see a black screen with "HELLO WORLD" printed on it. You may have trouble reading the text – this is due to the fact that there is no proper font to use. For more details, see the *2D Graphics* section below.

After clicking your mouse button the program will exit. You have now made your first step into the great new world of programming!

Enough of the simple stuff, let's get started with 'variables'!

Declaring variables. There are 3 'Types' of variables native to GLBasic (As of version 8, GLBasic can also use some C/C++ variable types – this is discussed much later on). These native variable types are Floating-point numbers, Integers and Strings.

You can think of Variables as desk-drawers. On the front of the drawer there's the name of the variable and within the drawer you'll find the value of the variable. Variables should be declared, as GLOBAL or LOCAL, before their first use, but de-activating the "Explicit Declarations" options in the Project → Options menu will allow you to use variables without the need to declare them.

For brevity, these declarations will be kept to a minimum throughout this tutorial, only being SHOWN once per data type - but variables will always have to be declared in your programs!

Let's start with numbers and see how it works.

Numeric Variable Types

Please note : With the release of version 9, any variables defined by LET need to be defined before use.

```
LOCAL a
LET a=5
```

We use the command 'LET' to tell the compiler we'd like to have the variable 'a' store the value 5 within it. To make variable declarations somewhat faster to type, the 'LET' command can be left out. The resulting line would look like this:

```
a=5
```

and

```
LET a=5
LET a=a+1
```

will look like this:

```
a=5
a=a+1
```

What are we doing here? Firstly we are taking the variable 'a' and assigning ('=') the value '5' to it. We then ask the compiler to take the variable with the name 'a' again and assign it the value ('=') of 'a+1'. As 'a' was already set to the value 5 at this point, 'a' will now be given the value of 'a+1' - i.e. '5+1', which is '6'. We can also increase a by a number using:

```
INC a,1
```

which can itself be abbreviated to

```
INC a
```

as the default amount to INCrease by is one.

There are other operations that can be done with numbers. GLBasic uses the standard computer symbols for mathematical calculations.

There are :

- '+' = Addition,
- '-' = Subtraction,
- '*' = Multiplication
- '/' = Division

with INC (increase) and DEC (decrease) being useful shorthand for simple additions and subtractions.

ATTENTION:

```
LET a= 3+4*5
```

would be processed as 3+(4*5) = 23. This has changed from older versions of GLBasic which would not evaluate it this way. GLBasic now has the following hierarchy for evaluating terms:
(brackets first) * / + - AND OR < > =
There are some special mathematical functions in GLBasic that work differently to the rest: `SIN()`, `COS()`, `TAN()` -These functions return a value.

So, for example, `a=COS(10)`, will put the mathematical result of Cosine 10 into the variable 'a'.

Another example of a numeric function is the `RND()` command. `RND()` returns a random number in the range 0 to argument.

For example:

```
LET a=RND(50) // a= random number from 0 to 50
```

When you create a variable for the first time (e.g. using LOCAL/GLOBAL command) you can append a '%' character to its name to indicate that this variable is an integer number. (An integer is a number without a fraction part).

Appending an optional '#' will make the variable a floating point number (one with a fractional part). The appendices ('%' or '#') are optional for further use of the variable.

Note : If you wish to perform integer only calculations, you can ensure this by using the function INTEGER().

String Variable Types

String variables are specified differently to numeric variables by having a '$' at the end of their name, which must be present whenever you want to use the variable.

Note : Numeric variable can swap their values with String variables and vice-versa, this process is known as a "type conversion" (i.e. converting a variable from one type - float, integer, string - to one of the others). All type conversions are handled automatically. More on this later.
String Variable Sample :

```
LOCAL a$
LET a$="HELLO"
```

We know that strings names end with '$', but how do you define the text that is to be stored within the String variable?

To allocate a string of text to a variable you surround the text with quotation marks (" ") - a$="HELLO" for example will store the text "HELLO" within the variable 'a$'.

The command LET allows you to do some useful things with strings.

```
LET a$="My"
LET b$="favourite number:"
LET c= 7
// Now look!!
LET c$=a$ + " " + b$ + " : " + c
PRINT c$,0,20
SHOWSCREEN
```

Output: My favourite number: 7

The 'LET' command, as always, can be left out if you like. You can see from this example that numbers can be converted to strings ('c$=c'), and strings can be joined together ('a$=a$+b$').

Why is this useful? Assume within a program you want to load an image from a file where the image's file-name is "image5.dat". The following is an example of how the automatic type conversion can make programming easier for the coder.

```
LET imagenumber=5 // This happens somewhere in the program...
LET image$="image" + imagenumber + ".dat"
```

Yes, you could just write `LET image$="image5.dat"` in this basic example, but as you learn GLBasic you will see where this sort of thing can come in really handy.

GLBasic offers some special functions for handling Strings.

```
INPUT name$, x, y
```

This command will cause a blinking cursor to appear at the screen position (x, y) and wait for the user to input something on the keyboard. After the user hits the return key, the variable 'name$' will contain the string of characters that was just typed. This works with numbers too (i.e. `INPUT number, x,y`).

```
dest$=MID$(source$, start, length)
```

You can use this function to obtain any part of a string. The string **dest$** will contain the resulting text from the string **source$** starting at the character indicated by **start** and running for the number of characters indicated by **length**. The location (index) of the first letter is 0, not 1.

Example :

```
name$="My house is blue"
m$=MID$(name$, 3, 5)
PRINT m$ , 20, 20
SHOWSCREEN
```

Output: house

Say you had a person typing their name in for a high score list but only wanted the first 7 letters. To achieve this you can trim the names by simply copying just the first 7 letters of the original string into a sub-string.

```
INPUT enter$, 100, 100
name$ = MID$(enter$, 0, 7)
PRINT name$, 100, 100; SHOWSCREEN; MOUSEWAIT
```

The `FORMAT$` command can convert numbers to strings more efficiently. Keep this in mind as you might need it sometime.

```
fmt$=FORMAT$(numLetters, numAfterComma, number)
```

Data Arrays Types

Now you have an overview of variables, but how would you store and manipulate large blocks of numbers or text? How would you store 1000 names in your program for example? 1000 separate variables? You could do this but it is not a practical solution - for this you would use something called

an "array".

An array can be imagined as a checker-board At each location on the checker-board is a value. To create an array in GLBasic that is 8 fields in its x-dimension and 8 fields in its y-dimension, you would define it like this:

```
LOCAL checker[] // we do not specify how many dimensions the array has when declaring the variable
DIM checker[8][8]
```

The computer will allocate memory (DIM) to store an 8x8 array, set each field to 0 and allow you to reference it via the name 'checker'.

If we want the field in location 3 across, 4 down to have a value of 7, we would write this:

```
LET checker[2][3]=7
```

Why "[2][3]" and not "[3][4]"? The array is indexed starting at 0 rather than one - i.e. the fields are numbered from 0 to 7 across and 0 to 7 down. 0,0 = top left of array, 7,7 = bottom right of array - just like map references. This is somewhat confusing at the beginning, but you'll get used to it very soon.

You can also reference an array's field via variables:

```
DIM checker[8][8]
LET checker[2][3]=7
LET x=2
LET y=3
LET a=checker[x][y]
PRINT "Checkerboard[2][3] has the value:", 100, 80
PRINT a, 100, 100
```

Output:

Checkerboard[2][3] has the value: 7

Note : Arrays may not have more than 4 dimensions.

```
DIM a[1][2][3][4]; // just OK
Data arrays can contain Strings too. Samples:
DIM space$[10][10][10] // A space with x, y and z (10 checkerboards each)
// each 10 fields (0-9)
LET space$[5][7][3]="Green"

DIM name$[5] // 5 strings
LET name$[0]="Peggy"
LET name$[1]="Tim"

DIM cinemavisitor$[30][10]
LET cinemavisitor$[seat][row]=" Tom"
```

With the command REDIM you can change the dimensions of existing arrays but keep the data held within. If you resize the array smaller than it was originally, the data outside the new array is lost.

```
DIM name$[5] // 5 strings
LET name$[0]="Peggy"
LET name$[4]="Tim"
```

```
REDIM name$[2] // 2 Words
```

The value at position 4 ("Tim") would be lost when the REDIM command is executed.
If you want to copy an array :

```
LOCAL a[]
LOCAL b[]
DIM a[5]
b[] = a[]
```

This re-dimensions a new array b[5] and copies the data from a[] to it. Note that it does not CREATE the array = as with a[], you must have previously declared the b[] array.

You can also remove rows from an array with the DIMDEL command. This is very convenient for managing a dynamic number of elements.

The last value of an array (I'll use an array called "a" in this example) is `a[ LEN(a[])-1 ]` or simply `a[-1]`. The value prior to the last value is `a[-2]`.
You can easily clear the contents of an array with the command REDIM *arrayname*[0].

Jump-marks

Problem: The program runs once and then exits but we want it to keep running.
Solution: We put a command in the program (in this case, at the end) to make it jump back to an earlier point within the program (in this case the start).
Example :

```
// A GOTO sample
beginning:
LET a=RND(600)
LET b=RND(400)
PRINT "Wow",a ,b
SHOWSCREEN
MOUSEWAIT
GOTO beginning
```

What is this doing? Firstly it defines a jump-mark with the name 'beginning' (indicated by the ":" at the end of the word). The program prints "Wow" to a random position and displays it on the screen until a mouse button is pressed. The GOTO command at the end makes the program jump to the line labelled 'beginning'. The example here shows what is called an "infinite loop" - i.e. the program will never end. To break the program out of the loop you push the "ESC" key. Keep this in mind - pushing ESC will exit your program whenever you want to.

Program jumps can also be used to write 'blocks' of code called subroutines. These are a great way of organising and debugging your code. To define a subroutine block use the menu command Project → New SUB, then use the wizard to create a new subroutine block (SUB) within your project. The GOSUB command can be used at any time to call your new subroutine block (SUB) of code after which time it will continue where it left off. An example:

```
// A GOSUB Sample
PRINT "Start",0,0
```

```
GOSUB middle
PRINT "End",0 ,30
END

// A SUB Function called 'middle'
SUB middle: // Define the Sub
    PRINT "Middle",0,20
    RETURN // Jumps back to next command after the GOSUB call
ENDSUB // End Sub Marker
```

Output: Start Middle End

A good example of a SUB would be a function for loading the input data for a program.
Splitting up your code into blocks makes them easier to debug and to share common code with other programs.

Note : SUBS are always placed at the end of the main program. Between the commands ENDSUB and SUB no code is allowed, only comments are allowed. You are not allowed to use GOTO to jump your program to a jump mark not inside the SUB you are in, nor are you allowed to call a GOTO from the main program to a jump mark inside a SUB. These simple rules, once gotten used to, force a clean programming style.

FOR-Loops

Assuming you want to fill an array with '1's, how would you code this when the array is 10 fields long? 10 LET commands? This would work, but it's not very elegant, and not practical if your array contains hundreds or thousands of fields. We need a set of commands for repeatedly calling a section of code to set an array value, and to increment a counter (pointing at the new array location) at the same time.

We use the FOR and NEXT commands to define the code section to be repeated, and a variable to store the 'loop' counter. Lets see an example with this For-Next Sample:

```
FOR x=0 TO 9
    LET array[x]=1
NEXT
```

In this example, the code between the FOR and NEXT commands is executed while x is counting up from 0 to 9. After each call of the NEXT command, x will automatically be incremented by 1.

Line-by-line walk-through:

```
FOR x=0 TO 9
```

Assign a variable called x with the value of 0. The following code is to be repeated until x is bigger than 9.

```
LET array[x]=1
```

Assign the value 1 to the array 'array' at position 'x'

```
NEXT
```

x is increased by 1. The code between FOR and NEXT is repeated.

Now we fill an 100x100 array with the value 1.

```
FOR x=0 TO 99
    FOR y=0 TO 99
        LET array[x][y]=1
    NEXT
NEXT
```

Loops can also run backwards - e.g. `FOR x=9 to 0 STEP -1` to start the counter at 9 and go back to 0 decrementing by 1 every time. You can increment the loop counter by values greater than 1 (e.g. `FOR x=0 to 9 STEP 2` - to make it count up by 2's) as well.

The command `STEP` define the amount that will be added to / subtracted from the counter with every call to `NEXT`.

```
FOR x=24 TO 0 STEP -5
PRINT x, 0, (x*20)
NEXT
```

Output: 24 19 14 9 4

The commands `BREAK` and `CONTINUE` allow you to leave or jump to the top of a loop before the code within the loop has completed its execution.

WHILE Loops

A `WHILE` loop will be evaluated (and repeated) as long as its defining argument is **TRUE**. e.g. We want to obtain a random number in the range 3 to 10. To do this we can pick a random number from 0 to 10 and continue picking random numbers while the random number returned is less than 3.

```
LET z=0 // z starts < 3, otherwise WHILE loop
// won't ever be evaluated!
WHILE z<3 // As long as z is < 3,
z= RND(10) // z is a random number from 0-10
WEND // Repeat.
```

Another example : You want a number from 0 to 10, but not the 5. Keep picking random numbers while the last random number picked was 5.

```
LET z=5 // z=5, otherwise WHILE loop fails
WHILE z=5 // As long as z=5
z=RND(10) // z is a random number from 0-10
WEND // Repeat.
By the way, random numbers from 3 to 10 are better calculated this way:
z=RND(7)+3 // (0 to 7) +3 = (3 to 10)
```

REPEAT Loops

A `REPEAT` loop will be evaluated (an repeated) as long as its argument defined on the `UNTIL` line is **TRUE**. The following example repeats a loop until the counter reaches 10

```
LET z=0
REPEAT
DEBUG z+"\n"
INC z
UNTIL z>=10
```

Tutorial #2 (Graphics)

General

To create the final screen image which represents each frame of your application GLBasic uses three separate image buffers (or 'surfaces'). One is used to display the final image to the monitor, one is used to draw your GLBasic graphical objects (Sprites, Polygons etc.) and the last one is used to store the 'background' for your final image.

Let's take a look at these:

1. Primary Surface - This is what you see on the screen.
2. Back Buffer - This is where all graphical commands in GLBasic write to. It's invisible, that is it's not transferred to the monitor until you call the command SHOWSCREEN.

This command takes current the back buffer and transfers it to the primary surface (the monitor).
Why not have GLBasic draw directly to the monitor (Primary Surface) rather than to a Back Buffer? If you did this you would see GLBasic processing its graphical commands individually 'on-screen' rather than all-at-once, and this would result in screen flicker.

Remember: If you don't see anything you might have forgotten to use the SHOWSCREEN command.

3. Off-screen Surface This screen can be accessed with several commands - LOADBMP, BLACKSCREEN, CLEARSCREEN, CREATESCREEN and USEASBMP. Please see the Reference Section for details on these commands

When SHOWSCREEN is called, the Back Buffer and Primary Surface are swapped. The now unusable Back Buffer will be replaced (cleaned) with the Off-screen Surface which is black when no bitmap is loaded.

GLBasic uses hidden surfaces (Double Buffering) for graphical display. If you code something in GLBasic you will see nothing before your program calls the SHOWSCREEN command. If you don't see any result from your program, the first thing to check is whether you have forgotten to add the SHOWSCREEN command to it. Think of Double Buffering as two sheets of paper, one on top of the other where you draw on the bottom sheet. Once you're finished drawing, you swap the sheets and in the process erase the new bottom sheet. This hides the actual drawing of the sheet from the user and in the process eliminates screen flicker.

A game mostly is structured like this:

```
// My game
start:
LOADBMP "Background.bmp"
// Load further graphics as sprites...
// load level, setup (lives=3, time=100...)
maingame:
//Keyboard input
//Move and draw player
```

```
//Move and draw enemies
SHOWSCREEN
GOTO maingame
```

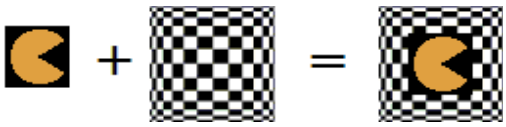
Sprites

What is a sprite? A sprite is usually a small graphic object that moves around the screen. As an example, in the game Pacman, Pacman and the Pac Ghosts are sprites and the maze is the background.

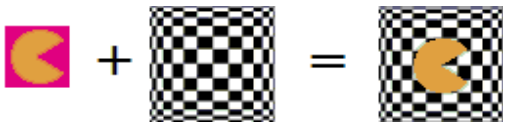
Here is a bitmap containing what could be sprites for a Pacman-style game :

If you know anything about computers you might know the truth - the PC doesn't have real sprites. Real 'sprites' in the days of the Commodore 64, Amiga, Atari XL or Archimedes, were little separate pieces of video memory that you could place an image in, these images were 'hardware overlayed' onto the display memory without damaging the background image. The PC does not have sprite hardware overlay technology but the good news is that GLBasic can emulate sprite hardware.
Sprites have to be what is referred to as "cookie cut" - i.e. the area around the shape of the sprite must be transparent. Without cookie cutting the sprite there would be a solid black rectangle around it when it was displayed.

Like this :



How does the program know which parts of the sprite are to be displayed transparently? There are many ways depending on the program you are using. GLBasic uses the following: Every pixel of an image loaded for a sprite that has the colour value red:255, green:0 and blue:128 (Hex: FF 00 80) (a shade of Pink) is considered to be transparent.



Note : All graphic files loaded are in the Windows-Bitmap format with 8 or 24 bit colour depths, uncompressed. You cannot load 16-colour graphics. You have to reload and save them as either 8-bit or

24-bit colour images.

You can also use PNG files. They use either `RGB(255,0,128)` for transparency or the alpha channel if it exists.

Your first sprite:

```
////////////////
// SPRITES
////////////////
// load sprite image from file "Bubble.bmp" and assign it to ID 0
LOADSPRITE "Bubble.bmp" ,0
// show sprite image with ID 0 at position x=300, y=150
DRAWSPRITE 0, 300, 150
SHOWSCREEN
END
```

As you can see with just 3 commands you produced a visible result. Short, sweet and simple, all the technical nitty gritty is hidden from the user by the simplicity of GLBasic. If you were to try to do this in C++ for example, the code would several pages long.

The `LOADSPRITE` command is fairly obvious. It loads a BMP file, which can be used as a sprite and assigns an ID (in this case 0) to it. Every sprite, needs a unique ID number, though the choice of ID is up to you. In the rest of the program this sprite will be referred to by its ID ('0' in this case). See also: the `GENSPRITE()` command.

Take a look at the .BMP file. You'll see there's a pink colour around the image. This part of the image will be the its transparent pixels. To demonstrate this, we'd better load an image for the background by adding...

```
LOADBMP "back.bmp"
```

...to the beginning of the program.

The command `DRAWSPRITE` needs 3 parameters. The first is the ID of the sprite to be shown. Parameters 2 and 3 are the screen position of the sprite (X,Y coordinates).

`ALPHAMODE` lets you specify an alpha value. GLBasic supports Alpha Blending. Never heard of 'Alpha Blending'? No problem...

Alpha Blending is when 2 images (namely the background and the sprite) are 'blended' together. GLBasic has two 'blending modes' for doing this.

Additive Alpha Blending

A value from 0.001 to 1.0 mixes the 2 colour values of each pixel by adding the red, green and blue values (a value over 255 (or FF in Hexadecimal) is considered to be 255). It sounds complicated, but all work is done by GLBasic. A value more than +1 for the Alpha Blending is for effects like fire, explosions and glass screens. If you want two images to get brighter when they are displayed over each