

Manual for xgs Basic

Copyright 2009, Kenneth G. Goutal. All rights reserved.

This manual describes the XGS Basic system, version 1.0.

This system consists of the following parts:

- the compiler (xgsbasic.exe)
- the serial downloader to the XGS PIC (sdwrite.exe)
- a simulator for simple debugging on your PC (basicvm.exe)
- the virtual machine

The compiler program is called **XGSBASIC.EXE**. It is an MS-Windows console application; that is, it runs on MS-DOS, or under MS-Windows but only using the COMMAND or CMD application. You will need to use one of those methods to navigate to the folder and run command line arguments. A full description of the command line syntax is given below.

The microSD downloader program is called **SDWRITE.EXE**. It likewise is an MS-Windows console application. A full description of the command line syntax is given below.

The simulator program is called **BASICVM.EXE**. It, too, is an MS-Windows console application. It does not support graphics or sound or, indeed, any input/output other than a PRINT statement. A full description of the command line syntax is given below.

The virtual machine runs on the XGS PIC. You can either choose to compile this or use one of the already compiled **BASICVM_PIC_TILE.HEX** or **BASICVM_PIC_BITMAP.HEX** files. If you do choose to compile it make sure you follow all instructions in the main source file.

Downloading the hex file is the easiest method and you may use the PICKit 2 standalone gui or go to the File->Import menu in MPLAB lab to load the hex into MPLAB and then program using the PICKit2 like normal.

Support for the XGS PIC 16-Bit Development Kit may be obtained from Nurve Networks LLC, support@nurve.net, www.xgamestation.com.

1.1 Command Line Syntax

To compile a Basic source file to a file that can be downloaded:

```
XGSBASIC [ -o output-file ] [ -p port ] [ -c ] input-file
```

The default for output-file is the same name as the input file but with the ".bai" extension.

The default for port is "COM4".

Use -c to compile a file without downloading it.

So, to compile and run the tile graphics demo:

1. Flash the basicvm_pic_tile.hex file into the XGS PIC
2. Compile the lifetile.bas file and download it to the XGS PIC
3. Reset the XGS PIC to cause it to load and run the downloaded file XGSBASIC lifetile.bas

The Basic runtime code automatically loads and starts the file named "autorun.bai" on reset so that would be a good choice for the download-file. There is no default for the download-file. Instead, no download is performed and only the output file is written.

The default for the port argument is "COM4". This should be set to the COM port that the XGS PIC board is connected to. For example, if your XGS PIC board is connected to COM6, use the string "-p COM6" on the command line. The port will be configured to 9600 baud, 8 bits no parity.

To run Basic programs, you'll need to download the Basic runtime into your XGS PIC board. The runtime can be built from the files included in this directory. The main file is called db_vmmain_v010.c and it lists all of the other files needed at the start. Don't forget to change the receive buffer size in the UART driver before compiling and flashing the runtime.

To load the sample program, connect your XGS PIC to your PC using the USB2Serial adapter and determine the COM port that gets assigned to the adapter. Insert a formatted microSD card into the XGS PIC and type the following command:

```
XGSBASIC -p com2 life.bas
```

This should compile the life.bas demo program and download it into the microSD flash card. After that, just reset the XGS PIC and the program should start. A random pattern will be generated and then the "life" algorithm will be run for 30 generations. After that, a new random pattern will be generated and the process starts over. If you like what is unfolding with a particular pattern, you can press the yellow button on the gamepad and that will extend that pattern beyond the normal 30 generations. Just keep holding the button down until you're tired of watching that pattern. When you release the button, a new pattern will be generated.

1.2 Language Syntax

Names:

Before going further, a discussion of *names* is in order. Several kinds of things are identified by names: *variables* (both scalars and arrays); statement *labels*; and *subroutines*.

A name can be at most 32 characters long. A name must start with a letter but can contain letters of either case, digits, "\$", "%" and "_". Also, a variable name that ends with "\$" is assumed to be a string variable unless otherwise specified; similarly, a variable name that ends with "%" is assumed to be an integer variable.

it should be noted that xgsBasic is not case-sensitive: the names "foo", "Foo", and "FOO" are all the same name as far as it is concerned.

Programs in xgsBasic consist of statements. Each statement occupies a single line, and each line consists of a single statement.

Expressions:

Expressions are used in many of the statements of this language. While there are some statements that are so simple that they do not require any expressions, expressions are so fundamental that we will discuss them before discussing the actual statements.

Statements:

Any statement may be preceded by a *label*. Doing so is required for some purposes, but most lines do not require them, and should not have them. This dialect of BASIC does not support the concept of *line numbers*. The use of labels will be discussed later, as necessary.

Some statements are not complete in and of themselves, and must be used in groups, or at least in pairs. For example, the **SUB** statement begins the definition of a subroutine. The **END SUB** statement ends the definition. All statements in between the two statements are a part of that subroutine.

1.2.1 Expressions

An expression is either a constant, a variable, or some combination of one or more of those using various operators. There are so many that it is helpful to consider them in groups or categories:

- Constant Expressions
decimal-constant 0xhex-constant string-constant
- Arithmetic Expressions
*+ - * / MOD -*
- Logical Expressions
NOT OR AND = <> < <= >= >
- Bitwise Expressions
~ & | ^ ~& ~| << >>
- Other Expressions
(...) variable array-reference function-call

Below are descriptions of each of them, by category. We examine constant expressions first, because it we will see them in examples of all the other expressions.

1.2.1.1 Constant Expressions

decimal-constant *0xhex-constant* *string-constant*

1.2.1.1.1 decimal constant

The value of this expression is a specific integer value represented as a signed decimal number.

Syntax:

[{ **+** | **-** }] *decimal-digit-string*

where *decimal-digit-string* is from 1 to 5 decimal digits with no intervening characters of any kind. The lower bound is -32768, and the upper bound is 32767.

A “minus” (negative-value) symbol may precede the *decimal-digit-string*, and space is allowed between the sign and the *decimal-digit-string*.

Examples:

<u>This expression:</u>	<u>has this value:</u>
0	0
-0	0
000	0
9	9
09	9
-9	-9
-09	-9
32767	32767
- 32768	-32768

Counterexamples:

32768 (*too large a positive value*) -32768
-32769 (*too large a negative value*) 32767

1.2.1.1.2 hexadecimal constant

The value of this expression is a specific integer value represented as a unsigned hexadecimal number.

Syntax:

[{ **+** | **-** }] **0***hex-digit-string*

where *hex-digit-string* is from 1 to 4 hexadecimal digits with no intervening characters of any kind. The lower bound is **0000**, and the upper bound is **FFFF**.

The two-character string **0x** prefixes the hexadecimal number in order to let the compiler (and a subsequent human reader) know that any decimal digits are actually part of a base-sixteen number. No space is allowed between the two characters **0** and **x** or between the **x** and the *hex-digit-string*. The letter **x** must be in lower case; it must not be a capital or upper-case **X**.

A “minus” (negative-value) symbol may precede the **0x**, and space is allowed between the sign and the **0x**.

Examples:

<u>This expression:</u>	<u>has this value:</u>
0x0	0
0x00000	0
0x9	9
0x00009	9
0xF	15
0x0000F	15
0xFF	255
0xFFF	4095
0x7FFF	32767
0x8000	-32768
0xFFFF	-1
0xFFFFF	-1
0xf	15
0x0000f	5
-0xf	-15
0Xf (<i>upper-case X</i>)	(none)
0xG (<i>invalid hex digit</i>)	(none)

1.2.1.1.3 string constant

The value of this expression is a specific sequence of printable characters.

Syntax:

" *printable-characters* "

The printable characters include the blank (0x20) as well as punctuation, digits, and upper-case and lower-case letters.

The characters of the string must be enclosed in a pair of double-quotes (**"..."**); hence, the double-quote character itself may not be included as part of a string constant.

Examples:

This statement:

PRINT "ABcd 09 ,,:!?"

PRINT "'ABcd 09 ,,:!?"

produces this output:

ABcd 09 ,,:!?

'ABcd 09 ,,:!?'

1.2.1.2 Arithmetic Expressions

Arithmetic expressions include those with the following operators:

+ - * / MOD -

1.2.1.2.1 addition

Syntax:

$expr_1 + expr_2$

This expression adds **$expr_2$** to **$expr_1$** .

Example:

The value of the following expression is 11:

$6 + 5$

1.2.1.2.2 subtraction

Syntax:

$expr_1 - expr_2$

This expression subtracts **$expr_2$** from **$expr_1$** .

Example:

The value of the following expression is 6:

$11 - 5$

1.2.1.2.3 multiplication

Syntax:

$expr1 * expr2$

This expression subtracts **$expr_2$** from **$expr_1$** .

Example:

The value of the following expression is 30:

$6 * 5$

1.2.1.2.4 division

Syntax:

$expr_1 / expr_2$

This expression divides **$expr_1$** by **$expr_2$** .

Example:

The value of the following expression is 6:

$30 / 5$

1.2.1.2.5 modulo

Syntax:

expr₁* MOD *expr₂

This expression divides ***expr₁*** by ***expr₂***, and returns the remainder.

Example:

The value of the following expression is 0:

30 MOD 5

The value of the following expression is also 0:

30 MOD 6

The value of the following expression is 1:

31 MOD 5

The value of the following expression is 2:

30 MOD 4

1.2.1.2.6 negation

Syntax:

- *expr*

This expression negates, or returns the negative value of, ***expr***.

Example:

The value of the following expression is negative three:

- 3

1.2.1.3 Relational Expressions

= <> < <= > >=

Relational expressions make arithmetic comparisons between numbers. They return 0 (zero) to represent FALSE and 1 (one) to represent TRUE.

1.2.1.3.1 equality

The value of this expression is 1 (one) if the specified expressions are equal to each other; otherwise, the value is 0 (zero).

Note: *This expression should not be confused with the assignment statement!*

Syntax:

$expr_1 = expr_2$

Examples:

The value of the following expression is 0 (representing FALSE):

$3 = 2$

The value of the following expression is 1 (representing TRUE):

$4 = 4$

1.2.1.3.2 inequality

The value of this expression is 1 (one) if the specified expressions are *not* equal to each other; otherwise, the value is 0 (zero).

Syntax:

$expr_1 <> expr_2$

Examples:

The value of the following expression is 0 (representing FALSE):

$3 <> 3$

The value of the following expression is 1 (representing TRUE):

$3 <> 4$

1.2.1.3.3 less-than

The value of this expression is the 1 (one) if the value of ***expr*₁** is strictly less than the value of ***expr*₂**; otherwise, the value is 0 (zero).

Syntax:

***expr*₁ < *expr*₂**

Examples:

The value of the following expression is 0 (representing FALSE):

4 < 3

The value of the following expression is 0 (representing FALSE):

4 < 4

The value of the following expression is 1 (representing TRUE):

4 < 5

1.2.1.3.4 less-than-or-equal-to

The value of this expression is the 1 (one) if the value of ***expr*₁** is less than *or equal to* the value of ***expr*₂**; otherwise, the value is 0 (zero).

Syntax:

***expr*₁ <= *expr*₂**

Examples:

The value of the following expression is 0 (representing FALSE):

4 <= 3

The value of the following expression is 1 (representing TRUE):

4 <= 4

The value of the following expression is 1 (representing TRUE):

4 <= 5

1.2.1.3.5 greater-than

The value of this expression is the 1 (one) if the value of ***expr₁*** is strictly greater than the value of ***expr₂***; otherwise, the value is 0 (zero).

Syntax:

expr₁ > ***expr₂***

Examples:

The value of the following expression is 0 (representing FALSE):

6 > 7

The value of the following expression is 0 (representing FALSE):

7 > 7

The value of the following expression is 1 (representing TRUE):

7 > 6

1.2.1.3.6 greater-than-or-equal-to

The value of this expression is the 1 (one) if the value of ***expr₁*** is greater than *or equal to* the value of ***expr₂***; otherwise, the value is 0 (zero).

Syntax:

expr₁ >= ***expr₂***

Examples:

The value of the following expression is 0 (representing FALSE):

6 >= 7

The value of the following expression is 1 (representing TRUE):

7 >= 7

The value of the following expression is 1 (representing TRUE):

7 >= 6

1.2.1.4 Logical Expressions

NOT OR AND

Logical expressions treat 0 (zero) as FALSE and *any non-zero value* as TRUE. Similarly, they return 0 (zero) to represent FALSE and 1 (one) to represent TRUE.

Note: *These are not the same as bitwise operations with the same or similar names.* Logical operators perform their operations on the whole value of each expression, and return either an integer 0 (zero) or an integer 1 (one); bitwise operators (see below) perform their operations on corresponding bits in each of the expressions, and return a new integer representing those result of those operations.

1.2.1.4.1 logical NOT

The value of this expression is TRUE if the specified expression is FALSE , and is FALSE if the specified expression is TRUE.

Syntax:

NOT expr

Examples:

The value of the following expression is 1 (representing TRUE):

NOT 0

The value of the following expression is 0 (representing FALSE):

NOT 3

1.2.1.4.2 logical OR

The value of this expression is TRUE if the values of either (or both) of the specified expressions is (or are) TRUE.

Syntax:

expr₁ OR expr₂

Examples:

The value of the following expression is 0 (representing FALSE):

0 OR 0

The value of the following expression is 1 (representing TRUE):

0 OR 3

The value of the following expression is 1 (representing TRUE):

-12 OR 0

The value of the following expression is 1 (representing TRUE):

-11 OR 1

1.2.1.4.3 logical AND

The value of this expression is TRUE if the values of both of the specified expressions are TRUE.

Syntax:

expr₁ AND expr₂

Examples:

The value of the following expression is 0 (representing FALSE):

0 AND 0

The value of the following expression is 0 (representing FALSE):

0 AND 3

The value of the following expression is 0 (representing FALSE):

-12 AND 0

The value of the following expression is 1 (representing TRUE):

-11 AND 1

1.2.1.5 Bitwise Expressions

~ & | ^ << >>

1.2.1.5.1 bitwise NOT

The value of this expression is the integer representation of the inversion, or ones-complement, of the bits of the specified expression.

Syntax:

~ *expr*

Examples:

This expression:

has this value:

~ 0	-1
~ 0x0000	0xFFFF
~ -1	0
~ 0xFFFF	0x0000
~ -2	1
~ 0xFFFE	0x0001
~ -256	255
~ 0xFF00	0x00FF
~ -275	274
~ 0xFEED	0x0112

1.2.1.5.2 bitwise inclusive OR

The value of this expression is the integer representation of the inclusive OR of the corresponding bits of the specified expressions. That is, if a given bit in *expr₁* is set to 1 *or* the corresponding bit in *expr₂* is set to 1, or *both bits are set*, then the corresponding bit in the result is set to 1; otherwise, it is set to 0 (zero).

Syntax:

expr₁ | *expr₂*

Examples:

This expression:

has this value:

0 | 1
0x0000 | 0x0001
1 | 2
0x0001 | 0x0002
2 | 3
0x0002 | 0x0003
-256 | 255
0xFF00 | 0x00FF

1
0x1
3
0x0003
3
0x0003
-1
0xFFFF

1.2.1.5.3 bitwise exclusive OR

The value of this expression is the integer representation of the exclusive OR of the corresponding bits of the specified expressions. That is, if a given bit in *expr₁* is set to 1 *or* the corresponding bit in *expr₂* is set to 1, *but not both bits are set*, then the corresponding bit in the result is set to 1; otherwise, it is set to 0 (zero).

Syntax:

expr₁ ^ *expr₂*

Examples:

This expression:

has this value:

0 ^ 1	1
0x0000 ^ 0x0001	0x0001
1 ^ 2	3
0x0001 ^ 0x0002	0x0003
2 ^ 3	1
0x0002 ^ 0x0003	0x0001
65280 ^ 255	-1
0xFF00 ^ 0x00FF	0xFFFF
43690 ^ 21845	-1
0xAAAA ^ 0x5555	0xFFFF
43690 ^ 65280	21930
0xAAAA ^ 0xFF00	0x55AA

1.2.1.5.4 bitwise AND

The value of this expression is the integer representation of the AND of the corresponding bits of the specified expressions. That is, if a given bit in ***expr₁*** is set to 1 *and* the corresponding bit in ***expr₂*** is set to 1, then the corresponding bit in the result is set to 1; otherwise, it is set to 0 (zero).

Syntax:

expr₁* & *expr₂

Examples:

<u>This expression:</u>	<u>has this value:</u>
0 & 1	0
0x0000 & 0x0001	0x0000
1 & 2	0
0x0001 & 0x0002	0x0000
2 & 3	2
0x0002 & 0x0003	0x0002
-256 & 255	0
0xFF00 & 0x00FF	0x0000
-21846 & 21845	0
0xAAAA & 0x5555	0x0000
-21846 & -256	-22016
0xAAAA & 0xFF00	0xAA00

1.2.1.5.5 bitwise shift left

The value of this expression is the integer representation of shifting *expr₁* left by the number of bits specified by *expr₂*.

Syntax:

expr₁ << *expr₂*

Examples:

This expression:

has this value:

1 << 1	2
0x0001 << 0x0001	0x0002
1 << 2	4
0x0001 << 0x0002	0x0004
1 << 8	256
0x0001 << 0x0008	0x0100
15 << 4	240
0x000F << 0x0004	0x00F0
15 << 8	3840
0x000F << 0x0008	0x0F00
255 << 8	-256
0x00FF << 0x0008	0xFF00
255 << 16	0
0x00FF << 0x0010	0x0000

1.2.1.5.6 bitwise shift right

The value of this expression is the integer representation of shifting *expr₁* right by the number of bits specified by *expr₂*.

Note: This is an *arithmetic* shift. Hence, the sign bit (the most-significant bit) is preserved, and is also copied to the next bit to its right, for as many bits as specified by *expr₂*.

Syntax:

expr₁ >> *expr₂*

Examples:

This expression:

has this value:

1 >> 1	0
0x0001 >> 0x0001	0x0000
1 >> 2	0
0x0001 >> 0x0002	0x0000
2 >> 1	1
0x0002 >> 0x0001	0x0001
15 >> 1	7
0x000F >> 0x0001	0x0007
240 >> 4	15
0x00F0 >> 0x0004	0x000F
-256 >> 8	-1
0xFF00 >> 0x0008	0xFFFF
-256 >> 16	-1
0xFF00 >> 0x0010	0xFFFF

1.2.1.6 Other Expressions

(...) *variable array-reference function-call*

1.2.1.6.1 parentheses

The value of this expression is the expressions inside the matched pair of parentheses.

Syntax:

(*expr*)

Parentheses simply provide the traditional way of grouping expressions together, particularly for the purpose of over-riding operator precedence.

Examples:

This expression:

has this value:

6 / 2 + 4

7

(6 / 2) + 4

7

6 / (2 + 4)

1

4 + 6 / 2

7

(4 + 6) / 2

5

1.2.1.6.2 variable

A variable is simply a value that changes, while the variable *name* remains the same.

Syntax:

variable

Examples:

<u>This expression or statement:</u>	<u>has this value or does this:</u>
x	(undefined!)
LET x = 6	assigns x the value 6
LET y=2	assigns y the value 2
z=4	assigns z the value 4
x	6
y	2
z	4
x / y	3
y + z	6
x / y + z	7
(x / y) + z	7
x / (y + z)	1
z + x / y	7
(z + x) / y	5

1.2.1.6.3 array reference or element

An *array* is simply a variable that can contain or represent more than one value simultaneously, each one distinguished from the others by its index (or subscript). The index may be any expression whose value is an integer; that is, it may not be a floating-point value or a string.

Generally, an array is used to group together two or more values that are in some sense alike, for instance, the highest temperature on each day of the year, or the wave frequency of each note in a scale or tune.

Syntax:

***variable* (*index* [, *index*])**

Example:

Suppose your program includes the following statements:

```
LET piano[40] = 261      REM C4
LET piano[41] = 277      REM C#4 or Db4
LET piano[42] = 293      REM D4
LET piano[43] = 311      REM D#4 or Eb4
LET piano[44] = 329      REM E4
LET piano[45] = 349      REM F4
LET piano[46] = 369      REM F#4 or Gb4
LET piano[47] = 391      REM G4
LET piano[48] = 415      REM G#4 or Ab4
LET piano[49] = 440      REM A4
LET piano[50] = 466      REM A#4 or Bb4
LET piano[51] = 493      REM B4
LET piano[52] = 523      REM C5
```

This stores the frequencies of the musical pitches noted in the comments into a set of array elements. (Yes, those frequencies are approximate.) The index of each array element is the piano key corresponding to that pitch.

You might then define a two-dimensional array to contain a sequence of notes. Each element of this array would have two parts – the frequency, and the duration. Something like this:

```
tada[1,1] = piano[52]      REM "TA" on High C ...
tada[1,2] = 483            REM for almost half a second.
tada[2,2] = 0              REM Silence ...
tada[2,2] = 17             REM for just a jiffy.
tada[3,1] = piano[52]      REM "DA" on High C ...
tada[3,2] = 1500           REM for 1.5 seconds.
```

1.2.1.6.4 function call

The value of a function call is the value of the name of the function immediately prior to ending (or returning, or exiting). See the section later in this document regarding how to define a function..

Syntax:

name ([***arg*** [, ***arg***] ...])

The ***name*** is just the name of the function.

There can be any number of ***arguments***, even none at all, as long as they match they number of arguments with which the function was defined.

Each argument can be any expression, as long as it matches the type of expression of the corresponding argument with which the function was defined.

Example:

Suppose your program contains the following statements, which define a function that computes the area of a right triangle, given the two orthogonal sides.

```
DEF rightTriangleArea ( side1 , side2 )  
  rightTriangleArea = side1 * side2 / 2  
END DEF
```

This function could then be called as follows:

```
LET A = rightTriangleArea ( 3, 4 )
```

which would set the variable “A” to the value $3*4/2$, or 6. Or it could be called this way:

```
PRINT rightTriangleArea(9,8)
```

which would display the number 36 (that is, $9*8/2$) on a line by itself.

Now we are ready to consider the statements that use all these expressions.

1.2.2 Simple Statements

Here is a list of statements that stand by themselves:

REM
OPTION
DEF¹
DIM
IF²
LET
GOTO
CALL
PRINT
STOP
END

Here are descriptions of each of them:

1.2.2.1 REM

Syntax:

REM [*comment text to end of line*]

1.2.2.2 OPTION

Syntax:

OPTION TARGET = { "tile" | "bitmap" }

This statement is the way to set various compiler options. At the moment the only one that is implemented is **TARGET**, which selects either the "tile" or "bitmap" graphics runtime environments.

-
- 1 There are two forms of the **DEF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in this section. The other form requires a matching **END DEF** statement, and is described in the **Compound Statements** section, below.
 - 2 There are two forms of the **IF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in this section. The other form requires a matching **END IF** statement, and may also include **ELSE** or **ELSE IF** statements, and is described in the **Compound Statements** section, below.

1.2.2.3 DEF

Syntax:

```
DEF name ( [ arg [ , arg ] ... ] )  
...  
END DEF
```

This form of the **DEF** statement is self-contained, and merely defines a constant; that is, it defines a name to have an unchangeable value.

Example

The following defines “hundredpi” to be a constant whose value is always (roughly) 100 times the value of π .

```
DEF hundredpi = 314
```

1.2.2.4 DIM

Syntax:

```
DIM variable-defs
```

This statement is the way to declare one or more variables to be arrays.

1.2.2.5 LET

Syntax:

```
[ LET ] l-value = expr
```

This is the assignment statement. It assigns the expression to the right of the “equals” sign to the l-value on the left. An l-value is just a way of saying something that can have a value assigned to it, i.e. either a scalar (one-dimensional) variable or a single element of an array.

Note that the word **LET** is optional. However, if present, it must be the first word of the statement, and no other word may be there instead.

Example:

```
LET A = 7  
pixels_per_brick = 47  
let ballWidth=15
```

1.2.2.6 IF

Syntax:

IF *expr* THEN *statement*

This statement is a way for a program to do a thing or not do a thing.

Examples:

If a value is zero, set it to some specific (default) value:

```
IF number_of_monsters = 0 THEN LET number_of_monsters = 111
```

Similarly, if some counter has reached a predetermined maximum, set it back to one.

```
IF N >= 24 THEN N = 1
```

1.2.2.7 GOTO

Syntax:

GOTO *label*

This statement causes the program execute the statement at “label” instead of executing the statement immediately following the **GOTO** statement. The **GOTO** statement seems obvious and innocent at first, but has generally been found to cause complexity and confusion if used more than sparingly. The xgsBasic language has many ways to organize sequences of statements in an orderly way, so the **GOTO** statement should be easy to avoid in most cases.

Example:

```
LET x=1
abc: LET x=x+1
GOTO hijk
efg: LET x=x-5
GOTO efg
hijk: LET x=x+2
STOP
END
```

Two questions immediately arise: (1) Does this program ever finish? (2) What is the value of x if and when it does?

1.2.2.8 CALL

Syntax:

CALL *name* ([*arg* [, *arg*] ...])

This statement calls the specified subroutine with the specified arguments (if any). Even if there are no arguments, the parentheses are required. If there is more than one argument, each argument must be separated from the next by using a comma. Note that the ellipsis (three periods in a row) just means that there may be an arbitrary number of arguments, each (except the first) preceded by a comma. See subsection **SUB**, under **Compound Statements**, below.

Example:

```
SUB sayhi
PRINT "Hello, World!"
END SUB
CALL sayhi
CALL sayhi
CALL sayhi
```

This example defines a subroutine (see below) called "sayhi", which merely displays the text string "Hello, World!". Following the definition, the subroutine is *called* three times in a row; each time, it displays the same message.

1.2.2.9 PRINT

Syntax:

PRINT [expr [[{ , | ; } expr] ...]]

This statement displays text on the screen. The text will represent zero or more expressions, as specified in the statement. Each expression may be a string or decimal or hexadecimal constant, or a scalar variable, or an array element. If no expressions are included, a blank line is displayed. If only one expression is included, no other syntax is required. If more than one expression is included, each must be separated from the next by either a comma or a semicolon.

If the separator is a semicolon, the second expression will appear immediately adjacent to the previous expression; in effect, they will *appear* to be concatenated.

On the other hand, if the separator is a comma, the second expression will begin at the next 8th column on the line.

Examples:

```
a: PRINT
b: PRINT A
c: PRINT pixels_per_brick ; ballWidth
d: PRINT pixels_per_brick , ballWidth
```

The example labeled “a” will print an empty or blank line.

Example “b” will print the number “7” on a line by itself.

Example “c” will print “4715” on a line by itself.

Example “d” will print “47 15” (that is “47” followed by 6 blanks or spaces, followed by “15”) on a line.

1.2.2.10 STOP

Syntax:

STOP

This statement tells the program to stop altogether, regardless of where in the program it appears or how it was encountered.

1.2.2.11 END

Syntax:

END

This statement tells the compiler that it is the last statement of the program. It has no effect on the program at run time. It is optional, but its use is encouraged.

1.2.3 Compound Statements

Here is a list of statements that must appear in groups:

DEF³
END DEF
SUB
END SUB
IF⁴
ELSE IF
ELSE
END IF
FOR
NEXT
DO
LOOP

Here are descriptions of each of them:

-
- 3 There are two forms of the **DEF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in the **Simple Statements** section, above. The other form requires a matching **END DEF** statement, and is described in the this section.
- 4 There are two forms of the **IF** statement. One is a simple statement, requiring no other statements to be complete. That form is described in the **Simple Statements** section, above. The other form requires a matching **END DEF** statement, and may also include **ELSE** or **ELSE IF** statements, and is described in the this section.

1.2.3.1 DEF

This form of the **DEF** statement defines a function.

Syntax:

```
DEF name ( [ arg [ , arg ] ... ] )  
...  
END DEF
```

The statement itself (with the name and parentheses and arguments) specifies how the function will be called. It must be followed by a matching **END DEF** statement (as shown). All the statements in between specify what the function does to achieve the result that it returns. In this form, the **END DEF** statement is *required*.

Inside of the function, the function's name is used as a variable to which to assign the return value; the value of that variable at the time the function completes execution is the return value of the function. There is no **RETURN** statement, as in some other dialects of BASIC.

Examples:

The following defines a function that computes the area of a right triangle, given the two orthogonal sides. The “body” of the function consists of just one statement, which computes the area of the square and divides that by 2, and assigns that the name of the function.

```
DEF rightTriangleArea ( side1, side2 )  
    rightTriangleArea = side1 * side2  
    rightTriangleArea = rightTriangleArea / 2  
END DEF
```

The body of this function could just as easily be as a single line, as follows:

```
DEF rightTriangleArea ( side1, side2 )  
    rightTriangleArea = side1 * side2 / 2  
END DEF
```

This function could then be called as follows:

```
LET A = rightTriangleArea ( 3, 4 )
```

which would set the variable “A” to the value 6. Or it could be called this way:

```
PRINT rightTriangleArea(9,8)
```

which would display the number 36 on a line by itself.

1.2.3.2 SUB

This statement defines a subroutine.

Syntax:

```
SUB name ( [ arg [ , arg ] ... ] )  
END SUB
```

A subroutine is essentially the same as a function, except that it does not return a value (or “have a return value”). See subsection CALL, under Simple Statements, above.

Example:

The following defines a subroutine that chooses a “fortune” at random, and displays it. Note that it does not return the value of the fortune for any further processing.

```
SUB dpyRightTriangleArea ( side1, side2 )  
PRINT side1 * side2 / 2  
END DEF
```

This subroutine could be called as follows:

```
CALL dpyRightTriangleArea ( 9, 8 )
```

which would have exactly the same result as the second example of calling the function in the previous section: it would display the number 36 on a line by itself.

1.2.3.3 IF

Syntax:

```
IF expr THEN statement  
IF expr THEN  
    [ ELSE IF expr THEN ]  
    [ ELSE ]  
END IF
```

This statement is the way for a program to do different things instead of each other, depending on circumstances.

The simplest case provides the means to either do a thing or not do a thing. The second form provides a way to do several things, or not do them; or to do more than one alternative thing or set of things.

Examples:

If a value is zero, set it to some specific (default) value:

```
IF number_of_monsters = 0 THEN LET number_of_monsters = 111
```

Similarly, if some counter has reached a predetermined maximum, set it back to one.

```
IF N >= 24 THEN N = 1
```

If you need to do more than one thing (or not), use this form:

```
IF number_of_monsters = 0 THEN  
    LET level = level + 1  
    LET number_of_monsters = 111 * level  
END IF
```

If you need to do two different things depending on circumstances, use this form:

```
DEF furry = 1  
DEF flying = 2  
IF level MOD 2 = 1 THEN  
    monster_type = furry  
ELSE  
    monster_type = flying  
END IF
```

If you need to do more than two different things, the IF ... THEN ... ELSE IF chain may be your answer:

```
DEF Sunday = 1
DEF Monday = 2
...
DEF Saturday = 7
IF (dayOFweek = Saturday)
  PRINT "Have a nice weekend!"
ELSE IF (dayOFweek = Sunday)
  PRINT "Have a nice Sunday!"
ELSE
  PRINT "Have a nice day!"
```

(This example is based on one in the PHP section of the w3schools.com web site.)

One IF statement can be “nested” inside another:

```
DEF furry = 1
DEF flying = 2
DEF slimy = 3
DEF arach = 4
IF level MOD 2 = 1 THEN
  IF LEVEL > 5 THEN
    monster_type = furry
  ELSE
    monster_type = slimy
  END IF
ELSE
  IF level > 5
    monster_type = arach
  ELSE
    monster_type = flying
  END IF
END IF
```

1.2.3.4 FOR

Syntax:

```
FOR variable1 = expr1 TO expr2 [ STEP expr3 ]  
  statements  
NEXT variable1
```

This statement is the way to do one or more statements over and over again, a certain number of times, each time setting the value of some variable to a new value.

First, the variable is set the value of the first expression. Then the statements in the middle are executed. The **NEXT** statement indicates that the variable (note that this is the same variable that is part of the **FOR** statement) should be set to the next value; if the new value of the variable is equal to or greater than the second expression, the statements in the middle are skipped, and the next statement to be executed will be the one immediately following the **NEXT** statement.

By default – i.e. if the STEP clause is omitted – the next value is always one (integer 1) greater than the previous value.

The variable may be used in the statements between the FOR and NEXT statements, or not; sometimes you only need it to control *how many times* a thing is done, not use it for anything else.

Examples:

Print out the numbers from 1 to 10:

```
FOR j = 1 TO 10  
  PRINT j  
NEXT j
```

Print out every 3rd number from 1 to 20 (1, 4, 7, 10, 13, 16, and 19):

```
FOR j = 1 TO 20 STEP 3  
  PRINT j  
NEXT j
```

1.2.3.5 DO

Syntax:

```
DO { UNTIL | WHILE } expr  
  statements  
LOOP
```

or

```
DO  
  statements  
LOOP { UNTIL | WHILE } expr
```

This statement is the way to do one or more statements over and over again, based on very general criteria. The two forms are equivalent, although the first format is generally clearer.

In either case, the use of **UNTIL** guarantees that the statements in the middle will be executed at least once, whereas with the use of **WHILE** there is no such guarantee – the statements inside the loop may not be executed at all.

In detail: In the case of **DO UNTIL *expr* ... LOOP** (and **DO ... LOOP UNTIL *expr***), the statement(s) inside the loop is (are) first executed once. The expression is then evaluated, and if it is *false*, then the statement(s) is (are) executed again, and so on, *until* the expression is *true*.

In the case of **DO WHILE *expr* ... LOOP** (and **DO ... LOOP WHILE *expr***), the expression is evaluated *first*, and *while* (or, as long as) the expression is *true*, the statement(s) in the middle is (are) executed over and over again.

IMPORTANT: Unlike the **FOR** statement, the **DO** statement in all its forms can very easily become an “infinite”, i.e. never-ending, loop! Specifically, if no statement(s) inside the loop alter any of the variables that make up the expression in the **DO** or **LOOP** statement, then the expression will never be altered, and can never become true (for **UNTIL**) or false (for **WHILE**). Even changing one or more variables that make up the expression doesn't guarantee that the expression will change from false to true or vice versa, so considerable care is required.

Examples:

Get 128 bytes of data from somewhere (using a user-defined function):

```
byteCount = 0
DO until byteCount = 128
  CALL loadByte()
  byteCount = byteCount + 1
LOOP
```

Get bytes of data from somewhere (using a user-defined function) until an EOF byte is encountered. As each byte comes in, store it in a buffer, and keep a count. Don't store the EOF in the buffer or include it in the count:

```
DEF EOF = 0x0F
i = 1
do until byte = EOF
  byte = getByte()
  if byte != EOF THEN
    buffer[i] = byte
    i = i + 1
  END IF
LOOP
byteCount = i - 1
```

1.3 Language Summary

This section summarizes the entire syntax of xgsBasic, using a format very similar to one known as Backus-Naur Form, or BNF. In each definition, or “production”, the first term is the one being defined, and it is shown in normal typeface.

The actual syntax is shown in **bold face**.

By contrast, the meta-syntax – those characters indicating denoting which pieces of actual syntax are optional or alternatives – are shown in normal case.

Keywords are shown in **ALL-UPPERCASE**, although (as noted above) this is not a requirement of the language; it's just used here to help distinguish keywords from things that are not keywords.

Terms that require further definition, and are defined below where they are used, are shown in *italics*.

As with BNF, brackets ('[' and ']') enclose optional pieces of syntax – you can include them, or leave them out, either at your whim or as appropriate to the situation. Braces ('{' and '}') enclose sets of alternatives, each alternative separated from its neighbor(s) by a vertical bar ('|'). A trio of dots or periods ('...') is used to indicate that the previous piece of syntax may be repeated any number of times.

1.3.1 Labels

Any statement may be preceded by an identifier followed by a colon. This is called a *label* and can be the target of a **GOTO** statement.

1.3.2 Statements

```
statements ::=
    statement
  |
    statement
    statements
```

Note: This definition is somewhat informal. It means that the word “statements” (plural) as used in the syntax descriptions above mean either a single statement or more than one statement, each on a line by itself.

1.3.3 Statement

```
statement ::=
  | REM comment text to end of line
  | OPTION TARGET = { "tile" | "bitmap" }
  | DEF name = value
  | DEF name ( [ arg [ , arg ] ... ] )
  | END DEF
  | SUB name ( [ arg [ , arg ] ... ] )
  | END SUB
  | DIM variable-defs
  | [ LET ] 1-value = expr
  | IF expr THEN statement
  | IF expr THEN
  | ELSE IF expr THEN
  | ELSE
  | END IF
  | FOR var = expr TO expr [ STEP expr ]
  | NEXT var
  | DO
  | DO WHILE expr
  | DO UNTIL expr
  | LOOP
  | LOOP WHILE expr
  | LOOP UNTIL expr
  | GOTO label
  | CALL name ( [ arg [ , arg ] ... ] )
  | PRINT
  | STOP
  | END
```

1.3.4 variable-defs

```
variable-defs ::=  
    variable-def  
    | variable-defs , variable-def
```

1.3.5 variable-def

```
variable-def ::=  
    variable [ AS type ]  
    | variable ( size [ , size ] ) [ AS type ]
```

1.3.6 type

```
type ::=  
    BYTE  
    | FLOAT (not yet implemented)  
    | INTEGER  
    | STRING
```


1.3.7 expr

In what follows, it may not always be clear that the punctuation marks that either are between one *expr* and another, or precede the *expr*, or surround the *expr*, are in bold face. They are, just like the keywords OR, AND, MOD, and so forth. As such they are required. Likewise it may not be clear that “0x” is in bold face. It is, and is a required part of hexadecimal constant.

```
expr ::=
    expr OR expr
    |
    expr AND expr
    |
    expr ^ expr
    |
    expr | expr
    |
    expr & expr
    |
    expr = expr
    |
    expr <> expr
    |
    expr < expr
    |
    expr <= expr
    |
    expr >= expr
    |
    expr > expr
    |
    expr << expr
    |
    expr >> expr
    |
    expr + expr
    |
    expr - expr
    |
    expr * expr
    |
    expr / expr
    |
    expr MOD expr
    |
    - expr
    |
    NOT expr
    |
    BNOT expr
    |
    ( expr )
    |
    decimal-constant
    |
    0xhex-constant
    |
    string
    |
    variable
    |
    array-reference
    |
    function-call
```

1.3.8 l-value

l-value ::=
 array-reference
 | *variable*

1.3.9 array-reference

array-reference ::=
 variable (*index* [, *index*])

1.3.10 variable

variable ::=
 name

1.3.11 function-call

function-call ::=
 name ([*arg* [, *arg*] ...])

1.3.12 built-in bitmap graphics functions

PLOT (*x*, *y*, *color*)
LINE (*x1*, *y1*, *x2*, *y2*, *color*)
HLINE (*x1*, *x2*, *y*, *color*)
VLINE (*x*, *y1*, *y2*, *color*)
CLEAR (*color*)
FILLRECT (*x1*, *y1*, *x2*, *y2*, *color*)
CIRCLE (*x*, *y*, *radius*, *color*)
TEXT (*x*, *y*, *string*)

1.3.13 built-in tile graphics functions

SETTILEMAP (*tile-map-array*)
SETTILES (*tile-array*)
SETCOLORTABLE (*color-table-array*)
SETSPRITE (*n state x y tile-number*)

1.3.14 other built-in functions

RND (*limit*)
WAITFORVSYNC ()
GAMEPAD (*which*)
SOUND (*frequency*)

1.3.15 name

name ::=
 letter
 | *letter alphanums*

1.3.16 decimal-constant

decimal-constant ::=
 [*sign*] *digit-string*

Note: The value of a decimal-constant must be in the range -32768 through 32767, inclusive. Spaces are not allowed within a decimal-constant.

1.3.17 digit-string

digit-string ::=
 digit
 | *digit digit-string*

Note: Spaces are not allowed within a digit-string.

1.3.18 hex-constant

hex-constant ::=
 hex-digit
 | *hex-digit hex-constant*

Note: The value of a hex-constant must be in the range 0x0000 through 0xFFFF, inclusive. Spaces are not allowed within a hex-constant.

1.3.19 string-constant

string-constant ::=
 " *printable-characters* "

Note: There is no specific limit to the length of a string constant, only the practical limit of the available memory. The doublequotes, one at each end of the string constant, are required.

1.3.20 printable-characters

```
printable-characters ::=  
    printable-characters  
    | printable-character printable-characters
```

1.3.21 printable-character

```
printable-character ::=  
    letter  
    | digit  
    | punctuation-mark  
    | blank
```

1.3.22 alphanums

```
alphanums ::=  
    alphanum  
    | alphanum alphanums
```

1.3.23 alphanum

```
alphanum ::= letter | digit
```

The following define the which specific characters make up the syntactic items above.

1.3.24 letter

letter ::=

	A	B	C	D	E	F	G	H	I	J	K	L	M
	N	O	P	Q	R	S	T	U	V	W	X	Y	z
	a	b	c	d	e	f	g	h	i	j	k	l	m
	n	o	p	q	r	s	t	u	v	w	x	y	z

1.3.25 punctuation-mark

punctuation-mark ::=

	.	,	:	;	!	?	/	\	'		
	`	~	@	#	\$	%	^	&	*		
	_	+	-	=	()	{	}	[]	

1.3.26 hex-digit

hex-digit ::=

	A	B	C	D	E	F	a	b	c	d	e	f
	<i>digit</i>											

1.3.27 digit

digit ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9