

MMBasic DOS Version

User Manual MMBasic Ver 5.4

For updates to this manual and more details on MMBasic go to

<http://geoffg.net/micromite.html>

and <http://mmbasic.com>

Copyright

The Micromite firmware including MMBasic and this manual are Copyright 2011-2017 by Geoff Graham.

The compiled object code (the MMBasic.exe file) is free software: you can use or redistribute it as you please. The source code is available via subscription (free of charge) to individuals for personal use or under a negotiated license for commercial use. In both cases go to <http://mmbasic.com> for details.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

This manual is distributed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0)

Contents

Introduction.....	3
File Input/Output.....	5
MMBasic Characteristics.....	8
Predefined Read Only Variables	10
Commands	11
Functions.....	20
Obsolete Commands and Functions	24

Introduction

MMBasic is an implementation of the BASIC language with floating point and string variables, long variable names, arrays of floats or strings with multiple dimensions and powerful string handling. It is generally compatible with Microsoft BASIC so it is easy to learn and run.

This version can run quite large and complex programs so it is useful for learning the BASIC language or running BASIC programs in a DOS/Windows environment. It uses the same syntax and basic commands as the Micromite version of MMBasic and can be used for testing programs in a convenient environment.

This manual covers the essentials of programming for the DOS version of MMBasic but for a more detailed explanation it is recommended that you read chapters 2 and 3 of the tutorial *Getting Started with the Micromite* which can be downloaded from: <http://geoffg.net/Micromite#Downloads>

Limitations

If you are familiar with the Micromite please note that this version does not attempt to emulate the full Micromite environment. Due to limitations of the DOS window this version does not support the full screen editor, graphics, external input/output, etc.

Installing and Running DOS MMBasic

The DOS version of MMBasic does not need installation. All you need do is copy the executable file (MMBasic.exe) to the directory of your choice. The executable is fully self contained; there are no libraries or other files required.

To run MMBasic just double click on the executable file (MMBasic.exe) and MMBasic will start running in a DOS box.



You can start MMBasic running a BASIC program by including the program's file name on the Windows command line.

Developing Programs

To prepare a BASIC program you should use a Windows text editor like Notepad to edit your program as a file within Windows. Do not use a word processing editor like WordPad or Word as they will insert formatting commands in the file causing errors when run in MMBasic.

To run the program you have four choices:

- Use RUN "filename" at the MMBasic command prompt (the greater than symbol '>'). Note that the double quotes are required (for example, RUN "MYFILE.BAS")
- Drag and drop the BASIC program file onto the MMBasic icon in Windows – this will cause Windows to start up MMBasic which will automatically run your program.

- If you associate the file extension of .BAS to MMBasic.exe you can run your BASIC program simply by double clicking on the program file (with the .BAS extension).
- Many editors like MMEdit, Notepad++ or Twistpad will let you define a single key that will save the file and run MMBasic with the file's name on the command line. This causes MMBasic to immediately run your program and results in a very fast edit/save/run cycle.

Differences to the Maximite Version of MMBasic

The main difference between the DOS version of MMBasic and the version running on the Micromite family is that the DOS version does not support any hardware related features of the Micromite.

This means that the following facilities are not supported:

- LCD display panels and associated features (touch, fonts, etc).
- Any interrupts including timing interrupts.
- External I/O and communications protocols (serial, I2C, SPI and 1-wire).
- Serial console and commands which operate on the serial console such as AUTOSAVE and XMODEM
- The full screen editor.

DOS MMBasic has two extra commands:

- QUIT which will close MMBasic and return to the operating system.
- SYSTEM which will issue a DOS command to the operating system.

When MMBasic is running it is possible to copy and paste text to and from MMBasic using the standard copy and paste facilities of the DOS box.

Double Precision Floating Point

All floating point numbers in the DOS version are double precision (the Maximite version uses single precision). This means that calculations will have a far greater range and will be accurate to about 16 decimal digits. Also when printing a floating point number MMBasic will display up to 9 digits.

File Input/Output

The DOS version of MMBasic has full support for accessing files and directories. This includes opening files for reading, writing or random access and loading and saving programs.

Note that:

- Long file/directory names are supported in addition to the old 8.3 format.
 - Upper/lowercase characters and spaces are allowed although the file system is not case sensitive.
 - Directory paths are allowed in file/directory strings. (ie, OPEN "\dir1\dir2\file.txt" FOR ...).
 - Back slashes must be used in paths. Eg \dir\file.txt.
 - Up to ten files can be simultaneously open.
 - In the following commands/functions the # in #fnbr is optional and may be omitted.
- ☐ OPEN fname\$ FOR mode AS #fnbr
Opens a file for reading or writing. 'fname\$' is the file name. 'mode' can be INPUT, OUTPUT, APPEND or RANDOM. '#fnbr' is the file number (1 to 10).
 - ☐ PRINT #fnbr, expression [[,;]expression] ... etc
Outputs text to the file opened as #fnbr.
 - ☐ INPUT #fnbr, list of variables
Read a list of comma separated data into the variables specified from the file previously opened as #fnbr.
 - ☐ LINE INPUT #fnbr, variable\$
Read a complete line into the string variable specified from the file previously opened as #fnbr.
 - ☐ CLOSE #fnbr [,#fnbr] ...
Close the file(s) previously opened with the file number '#fnbr'.

Programs can be loaded from a file.

- ☐ LOAD fname\$ [, R]
Load a BASIC program from disc. The optional suffix ",R" will cause the program to be run after it has been loaded.
- ☐ RUN fname\$
Load and run a BASIC program from disc.

Basic file and directory manipulation can be done from within a BASIC program.

- ☐ FILES
Search the current directory and list the files/directories found.
- ☐ KILL fname\$
Delete a file in the current directory.
- ☐ MKDIR dname\$
Make a sub directory in the current directory.
- ☐ CHDIR dname\$
Change into to the directory \$dname. \$dname can also be ".." (dot dot) for up one directory or "\" for the root directory.
- ☐ RMDIR dir\$
Remove, or delete, the directory 'dir\$'.
- ☐ SEEK #fnbr, pos
Will position the read/write pointer in a file that has been opened for RANDOM access to the 'pos' byte.

Also there are a number of functions that support the above commands.

- ☐ INPUT\$(nbr, #fnbr)
Will return a string composed of 'nbr' characters read from a file previously opened for INPUT with the

file number '#fnbr'. If less than 'nbr' characters are available the function will return with what it has (including an empty string if no characters are available).

- EOF(#fnbr)
Will return true if the file previously opened for INPUT with the file number '#fnbr' is positioned at the end of the file.
- LOC(#fnbr)
For a file opened as RANDOM this will return the current position of the read/write pointer in the file.
- LOF(#fnbr)
Will return the current length of the file in bytes.

Example of Sequential I/O

In the example below a file is created and two lines are written to the file (using the PRINT command). The file is then closed.

```
OPEN "fox.txt" FOR OUTPUT AS #1
PRINT #1, "The quick brown fox"
PRINT #1, "jumps over the lazy dog"
CLOSE #1
```

You can read the contents of the file using the LINE INPUT command. For example:

```
OPEN "fox.txt" FOR INPUT AS #1
LINE INPUT #1, a$
LINE INPUT #1, b$
CLOSE #1
```

LINE INPUT reads one line at a time so the variable a\$ will contain the text "The quick brown fox" and b\$ will contain "jumps over the lazy dog".

Another way of reading from a file is to use the INPUT\$() function. This will read a specified number of characters. For example:

```
OPEN "fox.txt" FOR INPUT AS #1
ta$ = INPUT$(12, #1)
tb$ = INPUT$(3, #1)
CLOSE #1
```

The first INPUT\$() will read 12 characters and the second three characters. So the variable ta\$ will contain "The quick br" and the variable tb\$ will contain "own".

Files normally contain just text and the print command will convert numbers to text. So in the following example the first line will contain the line "123" and the second "56789".

```
nbr1 = 123 : nbr2 = 56789
OPEN "numbers.txt" FOR OUTPUT AS #1
PRINT #1, nbr1
PRINT #1, nbr2
CLOSE #1
```

Again you can read the contents of the file using the LINE INPUT command but then you would need to convert the text to a number using VAL(). For example:

```
OPEN "numbers.txt" FOR INPUT AS #1
LINE INPUT #1, a$
LINE INPUT #1, b$
CLOSE #1
x = VAL(a$) : y = VAL(b$)
```

Following this the variable x would have the value 123 and y the value 56789.

Random File I/O

For random access the file should be opened with the keyword RANDOM. For example:

```
OPEN "filename" FOR RANDOM AS #1
```

To seek to a record within the file you would use the SEEK command which will position the read/write pointer to a specific byte. The first byte in a file is numbered one so, for example, the fifth record in a file that uses 64 byte records would start at byte 257. In that case you would use the following to point to it:

```
SEEK #1, 257
```

When reading from a random access file the INPUT\$() function should be used as this will read a fixed number of bytes (ie, a complete record) from the file. For example, to read a record of 64 bytes you would use:

```
dat$ = INPUT$(64, #1)
```

When writing to the file a fixed record size should be used and this can be easily accomplished by adding sufficient padding characters (normally spaces) to the data to be written. For example:

```
PRINT #1, dat$ + SPACE$(64 - LEN(dat$));
```

The SPACE\$() function is used to add enough spaces to ensure that the data written is an exact length (64bytes in this example). The semicolon at the end of the print command suppresses the addition of the carriage return and line feed characters which would make the record longer than intended.

Two other functions can help when using random file access. The LOC() function will return the current byte position of the read/write pointer and the LOF() function will return the total length of the file in bytes.

The following program demonstrates random file access. Using it you can append to the file (to add some data in the first place) then read/write records using random record numbers. The first record in the file is record number 1, the second is 2, etc.

```
RecLen = 64
OPEN "test.dat" FOR RANDOM AS #1
DO
  abort: PRINT
  PRINT "Number of records in the file =" LOF(#1)/RecLen
  INPUT "Command (r = read,w = write, a = append, q = quit): ", cmd$
  IF cmd$ = "q" THEN CLOSE #1 : END
  IF cmd$ = "a" THEN
    SEEK #1, LOF(#1) + 1
  ELSE
    INPUT "Record Number: ", nbr
    IF nbr < 1 or nbr > LOF(#1)/RecLen THEN PRINT "Invalid record" : GOTO abort
    SEEK #1, RecLen * (nbr - 1) + 1
  ENDIF
  IF cmd$ = "r" THEN
    PRINT "The record = " INPUT$(RecLen, #1)
  ELSE
    LINE INPUT "Enter the data to be written: ", dat$
    PRINT #1,dat$ + SPACE$(RecLen - LEN(dat$));
  ENDIF
LOOP
```

Random access can also be used on a normal text file. For example, this will print out a file backwards:

```
OPEN "file.txt" FOR RANDOM AS #1
FOR i = LOF(#1) TO 1 STEP -1
  SEEK #1, i
  PRINT INPUT$(1, #1);
NEXT i
CLOSE #1
```

MMBasic Characteristics

Naming Conventions

Command names, function names, labels, variable names,, etc are not case sensitive, so that "Run" and "RUN" are equivalent and "dOO" and "Doo" refer to the same variable.

The type of a variable can be specified in the DIM command or by adding a suffix to the end of the variable's name. For example the suffix for an integer is '%' so if a variable called nbr% is automatically created it will be an integer. There are three types of variables:

1. Floating point. These can store a number with a decimal point and fraction (eg, 45.386) and also very large numbers. The suffix is '!' and floating point is the default when a variable is created without a suffix
2. 64-bit integer. These can store numbers with up to 19 decimal digits without losing accuracy but they cannot store fractions (ie, the part following the decimal point). The suffix for an integer is '%'
3. Strings. These will store a string of characters (eg, "Tom"). The suffix for a string is the '\$' symbol (eg, name\$, s\$, etc) Strings can be up to 255 characters long.

Variable names and labels can start with an alphabetic character or underscore and can contain any alphabetic or numeric character, the period (.) and the underscore (_). They may be up to 32 characters long. A variable name or a label must not be the same as a command or a function or one of the following keywords: THEN, ELSE, TO, STEP, FOR, WHILE, UNTIL, MOD, NOT, AND, OR, XOR, AS. Eg, step = 5 is illegal.

Constants

Numeric constants may begin with a numeric digit (0-9) for a decimal constant, &H for a hexadecimal constant, &O for an octal constant or &B for a binary constant. For example &B1000 is the same as the decimal constant 8. Constants that start with &H, &O or &B are always treated as 64-bit integer constants.

Decimal constants may be preceded with a minus (-) or plus (+) and may be terminated with 'E' followed by an exponent number to denote exponential notation. For example 1.6E+4 is the same as 16000.

If the decimal constant contains a decimal point or an exponent, it will be treated as a floating point constant; otherwise it will be treated as a 64-bit integer constant.

String constants are surrounded by double quote marks ("). Eg, "Hello World".

Operators and Precedence

The following operators, in order of precedence, are recognised. Operators that are on the same level (for example + and -) are processed with a left to right precedence as they occur on the program line.

Arithmetic operators:

^	
* / \ MOD	Multiplication, division, integer division and modulus (remainder)
+ -	Addition and subtraction

Shift operators:

x << y x >> y	These operate in a special way. << means that the value returned will be the value of x shifted by y bits to the left while >> means the same only right shifted. They are integer functions and any bits shifted off are discarded and any bits introduced are set to zero.
------------------	--

Logical operators:

NOT	logical inverse of the value on the right
<> < > <= >= >= <=	Inequality, less than, greater than, less than or equal to, less than or equal to (alternative version), greater than or equal to, greater than or equal to (alternative version)
=	equality
AND OR XOR	Conjunction, disjunction, exclusive or

The operators AND, OR and XOR are integer bitwise operators. For example PRINT (3 AND 6) will output 2. The other logical operations result in the integer 0 (zero) for false and 1 for true. For example the statement PRINT 4 >= 5 will print the number zero on the output and the expression A = 3 > 2 will store +1 in A. The NOT operator is highest in precedence so it will bind tightly to the next value. For normal use the expression to be negated should be placed in brackets. For example, IF NOT (A = 3 OR A = 8) THEN ...

String operators:

+	Join two strings
<> < > <= =< >= =>	Inequality, less than, greater than, less than or equal to, less than or equal to (alternative version), greater than or equal to, greater than or equal to (alternative version)
=	Equality

String comparisons respect case. For example "A" is greater than "a".

Implementation Characteristics

Maximum program size is 512KB.

Maximum number of variables is 500.

Maximum length of a command line is 255 characters.

Maximum length of a variable name or a label is 32 characters.

Maximum number of dimensions to an array is 8.

Maximum number of arguments to commands that accept a variable number of arguments is 50.

Maximum number of nested FOR...NEXT loops is 50.

Maximum number of nested DO...LOOP commands is 50.

Maximum number of nested GOSUBs, subroutines and functions (combined) is 1000.

Maximum number of nested multiline IF...ELSE...ENDIF commands is 20.

Maximum number of user defined subroutines and functions (combined): 512

Numbers are stored and manipulated as double precision floating point numbers or 64-bit signed integers. The maximum floating point number allowable is 1.7976931348623157e+308 and the minimum is 2.2250738585072014e-308.

The range of 64-bit integers (whole numbers) that can be manipulated is ± 9223372036854775807 .

Maximum string length is 255 characters.

Maximum line number is 65000.

Compatibility

MMBasic implements a large subset of Microsoft's GW-BASIC. There are numerous differences due to physical and practical considerations but most standard BASIC commands and functions are essentially the same. An online manual for GW-BASIC is available at <http://www.antonis.de/qbebooks/gwasman/index.html> and this provides a more detailed description of the commands and functions.

MMBasic also implements a number of modern programming structures documented in the ANSI Standard for Full BASIC (X3.113-1987) or ISO/IEC 10279:1991. These include SUB/END SUB, the DO WHILE ... LOOP, the SELECT...CASE statements and structured IF .. THEN ... ELSE ... ENDIF statements.

Predefined Read Only Variables

These variables are set by MMBasic and cannot be changed by the running program.

MM.VER	The version number of the firmware as a floating point number in the form aa.bb.cc where aa is the major version number, bb is the minor version number and cc is the revision number. For example version 5.03.00 will return 5.3 and version 5.03.01 will return 5.0301.
MM.DEVICE\$	A string representing the device or platform that MMBasic is running on. Currently this variable will contain one of the following: "Maximite" on the standard Maximite and compatibles. "Colour Maximite" on the Colour Maximite and UBW32. "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "Generic PIC32" for the generic version of MMBasic on a PIC32. "Micromite" on the PIC32MX150/250 "Micromite MkII" on the PIC32MX170/270 "Micromite Plus" on the PIC32MX470 "Micromite Extreme" on the PIC32MZ series
MM.ERRNO MM.ERRMSG\$	If a statement caused an error which was ignored these variables will be set accordingly. MM.ERRNO is a number where non zero means that there was an error and MM.ERRMSG\$ is a string representing the error message that would have normally been displayed on the console. They are reset to zero and an empty string by RUN, ON ERROR IGNORE or ON ERROR SKIP.

Commands

Square brackets indicate that the parameter or characters are optional.

' (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
? (question mark)	Shortcut for the PRINT command.
CHDIR dir\$	Change the current working directory to 'dir\$' The special entry ".." represents the parent of the current directory and "." represents the current directory.
CLOSE [#]fnbr [, [#]fnbr] ...	Close the file(s) previously opened with the file number '#fnbr'. The # is optional. Also see the OPEN command.
CLS	Clears all text in the DOS box.
CLEAR	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
CONST id = expression [, id = expression] ... etc	Create a constant identifier which cannot be changed once created. 'id' is the identifier which follows the same rules as for variables. The identifier can have a type suffix (!, %, or \$) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.
CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.
DATA constant [,constant]...	Stores numerical and string constants to be accessed by READ. In general string constants should be surrounded by double quotes ("). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as THEN, WHILE, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.
DIM [type] decl [,decl]... where 'decl' is: var [length] [type] [init] 'var' is a variable name with optional dimensions 'length' is used to set the maximum size of the string to 'n' as in LENGTH n 'type' is one of AS FLOAT or	Declares one or more variables (ie, makes the variable name and its characteristics known to the interpreter). When OPTION EXPLICIT is used (as recommended) the DIM or LOCAL commands are the only way that a variable can be created. If this option is not used then using the DIM command is optional and if not used the variable will be created automatically when first referenced. The type of the variable (ie, string, float or integer) can be specified in one of three ways: By using a type suffix (ie, !, % or \$ for float, integer or string). For example:

<p>AS INTEGER or AS STRING</p> <p>'init' is the value to initialise the variable and consists of: = <expression></p> <p>For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.</p> <p>Examples:</p> <pre>DIM nbr(50) DIM INTEGER nbr(50) DIM name AS STRING DIM a, b\$, nbr(100), strn\$(20) DIM a(5,5,5), b(1000) DIM strn\$(200) LENGTH 20 DIM AS STRING strn(200) LENGTH 20 DIM a = 1234, b = 345 DIM STRING strn = "text" DIM x%(3) = (11, 22, 33, 44)</pre>	<pre>DIM nbr%, amount!, name\$</pre> <p>By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (ie, it does not have to be repeated). For example:</p> <pre>DIM AS STRING first_name, last_name, city</pre> <p>By using the Microsoft convention of using the keyword "AS" and the type keyword (ie, FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable.</p> <p>For example:</p> <pre>DIM amount AS FLOAT, name AS STRING</pre> <p>Floating point or integer variables will be set to zero when created and strings will be set to an empty string (ie, ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example:</p> <pre>DIM AS STRING city = "Perth", house = "Brick"</pre> <p>The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed.</p> <p>As well as declaring simple variables the DIM command will also declare arrayed variables (ie, an indexed variable with a number of dimensions). Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example:</p> <pre>DIM array(10, 20)</pre> <p>Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1.</p> <p>The above example specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each floating point number requires 4 bytes a total of 924 bytes of memory will be allocated (integers are different and require 8 bytes per element).</p> <p>Strings will default to allocating 255 bytes (eg, characters) of memory for each element and this can quickly use up memory. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters.</p> <p>For example: DIM str\$(5, 10) will declare a string array with 66 elements consuming 16,896 bytes of memory while:</p> <pre>DIM AS STRING str (5, 10) LENGTH 20</pre> <p>Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.</p> <p>If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this string arrays created with the LENGTH keyword act exactly the same as other string arrays. Note that the LENGTH keyword can also be used when defining non array string variables.</p> <p>In the above example you can also use the Microsoft syntax of specifying the type after the length specifier. For example:</p> <pre>DIM str (5, 10) LENGTH 20 AS STRING</pre> <p>Arrays can also be initialised when they are declared by adding an equals symbol (=) followed by a bracketed list of values at the end of the declaration. For example:</p> <pre>DIM INTEGER nbr(4) = (22, 44, 55, 66, 88)</pre>
--	--

	<p>or DIM str\$(3) = ("foo", "boo", "doo", "zoo")</p> <p>Note that the number of initialising values must match the number of elements in the array including the base value set by OPTION BASE. If a multi dimensioned array is initialised then the first dimension will be initialised first followed by the second, etc.</p>
DO <statements> LOOP	This structure will loop forever; the EXIT DO command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like GOTO or EXIT SUB (if in a subroutine).
DO WHILE expression <statements> LOOP	Loops while "expression" is true (this is equivalent to the older WHILE-WEND loop, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, even once.
DO <statements> LOOP UNTIL expression	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is true.
ELSE	Introduces a default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF expression THEN or ELSE IF expression THEN	Introduces a secondary condition in a multiline IF statement. See the multiline IF statement for more details.
ENDIF or END IF	Terminates a multiline IF statement. See the multiline IF statement for more details.
END	End the running program and return to the command prompt.
END FUNCTION	Marks the end of a user defined function. See the FUNCTION command. Each function must have one and only one matching END FUNCTION statement. Use EXIT FUNCTION if you need to return from a function from within its body.
END SUB	Marks the end of a user defined subroutine. See the SUB command. Each sub must have one and only one matching END SUB statement. Use EXIT SUB if you need to return from a subroutine from within its body.
ERASE variable [,variable]...	Deletes variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (eg, dat ()) or just by specifying the variable's name (eg, dat). Use CLEAR to delete all variables at the same time (including arrays).
ERASE variable [,variable]...	Deletes arrayed variables and frees up the memory. Use CLEAR to delete all variables including all arrayed variables.
ERROR [error_msg\$]	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.
FILES	Lists files in the current directory. Same as the DOS DIR.

<p>FOR counter = start TO finish [STEP increment]</p>	<p>Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' equals 'finish'. The 'increment' can be an integer or floating point number. Note that using a floating point fractional number for 'increment' can accumulate rounding errors in 'counter' which could cause the loop to terminate early or late. 'increment' can be negative in which case 'finish' should be less than 'start' and the loop will count downwards. See also the NEXT command.</p>
<p>FUNCTION xxx (arg1 [,arg2, ...]) [AS <type>] <statements> <statements> xxx = <return value> END FUNCTION</p>	<p>Defines a callable function. This is the same as adding a new function to MMBasic while it is running your program. 'xxx' is the function name and it must meet the specifications for naming a variable. The type of the function can be specified by using a type suffix (ie, xxx\$) or by specifying the type using AS <type> at the end of the functions definition. For example: FUNCTION xxx (arg1, arg2) AS STRING 'arg1', 'arg2', etc are the arguments or parameters to the function. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1\$) or by specifying the type using AS <type> (ie, arg1 AS STRING). The argument can also be another defined function or the same function if recursion is to be used (the recursion stack is limited to 1000 nested calls). To set the return value of the function you assign the value to the function's name. For example: FUNCTION SQUARE (a) SQUARE = a * a END FUNCTION Every definition must have one END FUNCTION statement. When this is reached the function will return its value to the expression from which it was called. The command EXIT FUNCTION can be used for an early exit. You use the function by using its name and arguments in a program just as you would a normal MMBasic function. For example: PRINT SQUARE (56.8) When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function. Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string. Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. You must not jump into or out of a function using commands like GOTO, GOSUB, etc. Doing so will have undefined side effects including the possibility of ruining your day.</p>
<p>GOSUB target</p>	<p>Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN.</p>
<p>GOTO target</p>	<p>Branches program execution to the target, which can be a line number or a label.</p>

IF expr THEN statement or IF expr THEN stmt ELSE stmt	Evaluates the expression 'expr' and performs the THEN statement if it is true or skips to the next line if false. The optional ELSE statement is the reverse of the THEN test. This type of IF statement is all on one line. The 'THEN statement' construct can be also replaced with: GOTO linenumber label'.
IF expression THEN <statements> [ELSEIF expression THEN <statements>] [ELSE <statements>] ENDIF	Multiline IF statement with optional ELSE and ELSEIF cases and ending with ENDIF. Each component is on a separate line. Evaluates 'expression' and performs the statement(s) following THEN if the expression is true or optionally the statement(s) following the ELSE statement if false. The ELSEIF statement (if present) is executed if the previous condition is false and it starts a new IF chain with further ELSE and/or ELSEIF statements as required. One ENDIF is used to terminate the multiline IF.
INPUT ["prompt string\$";] list of variables	Allows input from the console to a list of variables. The input command will prompt with a question mark (?). The input must contain commas to separate each data item if there is more than one variable. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 If the "prompt string\$" is specified it will be printed before the question mark. If the prompt string is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #fnbr, list of variables	Same as the normal INPUT command except that the input is read from a file previously opened for INPUT as '#fnbr'. See the OPEN command.
KILL file\$	Deletes the file specified by 'file\$'. If there is an extension it must be specified.
LINE INPUT #fnbr, string-variable\$	Same as the LINE INPUT command except that the input is read from a file previously opened for INPUT as '#fnbr'. See the OPEN command.
LOAD file\$ [,R]	Loads a program called 'file\$' from the current drive into program memory. If the optional suffix ,R is added the program will be immediately run without prompting. If an extension is not specified ".BAS" will be added to the file name.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command.
LINE INPUT [prompt\$,] string-variable\$	Reads an entire line from the console input into 'string-variable\$'. If specified the 'prompt\$' will be printed first. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of 'prompt\$'.
LIST or LIST ALL	List a program on the serial console. LIST on its own will list the program with a pause at every screen full. LIST ALL will list the program without pauses.
LOCAL variable [, variables] See DIM for the full syntax.	Defines a list of variable names as local to the subroutine or function. This command uses exactly the same syntax as DIM and will create variables that will only be visible within the subroutine or function. They will be automatically discarded when the subroutine or function exits.
LOOP [UNTIL expression]	Terminates a program loop: see DO.
MEMORY	List the amount of memory currently in use.

MKDIR dir\$	Make, or create, the directory 'dir\$'.
NAME old\$ AS new\$	Rename a file or a directory from 'old\$' to 'new\$'. Both are strings. A directory path can be used in both 'old\$' and 'new\$'.
OPEN fname\$ FOR mode AS [#]fnbr	<p>Opens a file for reading or writing.</p> <p>'fname' is the filename with an optional extension separated by a dot (.). Long file names with upper and lower case characters are supported.</p> <p>A directory path can be specified with the backslash as directory separators. The parent of the current directory can be specified by using a directory name of .. (two dots) and the current directory with . (a single dot).</p> <p>For example OPEN "..\dir1\dir2\filename.txt" FOR INPUT AS #1</p> <p>'mode' is INPUT, OUTPUT, APPEND or RANDOM.</p> <p>INPUT will open the file for reading and throw an error if the file does not exist. OUTPUT will open the file for writing and will automatically overwrite any existing file with the same name.</p> <p>APPEND will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the APPEND mode will act the same as the OUTPUT mode (i.e. the file is created then opened for writing).</p> <p>RANDOM will open the file for both read and write and will allow random access using the SEEK command. When opened the read/write pointer is positioned at the end of the file.</p> <p>'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT, WRITE and CLOSE commands as well as the EOF() and INPUT\$() functions all use 'fnbr' to identify the file being operated on.</p> <p>See also OPTION ERROR and MM.ERRNO for error handling.</p>
NEW	Deletes the program and clears all variables.
NEXT [counter-variable] [, counter-variable], etc	<p>NEXT comes at the end of a FOR-NEXT loop; see FOR.</p> <p>The 'counter-variable' specifies exactly which loop is being operated on. If no 'counter-variable' is specified the NEXT will default to the innermost loop. It is also possible to specify multiple counter-variables as in:</p> <p>NEXT x, y, z</p>
ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR	<p>This controls the action taken if an error occurs while running a program and applies to all errors discovered by MMBasic including syntax errors, wrong data, etc.</p> <p>ON ERROR ABORT will cause MMBasic to display an error message, abort the program and return to the command prompt. This is the normal behaviour of MMBasic and is the default when a program starts running.</p> <p>ON ERROR IGNORE will cause any error to be ignored.</p> <p>ON ERROR SKIP will ignore an error in a number of commands (specified by the number 'nn') executed following this command. 'nn' is optional, the default if not specified is one. After the number of commands has completed (with an error or not) the behaviour of MMBasic will revert to ON ERROR ABORT.</p> <p>If an error occurs and is ignored/skipped the read only variable MM.ERRNO will be set to non zero and MM.ERRMSG\$ will be set to the error message that would normally be generated. These are reset to zero and an empty string by ON ERROR CLEAR. They are also cleared when the program is run and when ON ERROR IGNORE and ON ERROR SKIP are used.</p> <p>ON ERROR IGNORE can make it very difficult to debug a program so it is strongly recommended that only ON ERROR SKIP be used.</p>

ON nbr GOTO GOSUB target[,target, target,...]	ON either branches (GOTO) or calls a subroutine (GOSUB) based on the rounded value of 'nbr'; if it is 1, the first target is called, if 2, the second target is called, etc. Target can be a line number or a label.
OPTION BASE 0 1	Set the lowest value for array subscripts to either 0 or 1. This must be used before any arrays are declared and is reset to the default of 0 on power up.
OPTION CASE UPPER LOWER TITLE	Change the case used for listing command and function names when using the LIST command. The default is TITLE but the old standard of MMBasic can be restored using OPTION CASE UPPER.
OPTION DEFAULT FLOAT INTEGER STRING NONE	Used to set the default type for a variable which is not explicitly defined. If OPTION DEFAULT NONE is used then all variables must have their type explicitly defined. When a program is run the default is set to FLOAT for compatibility with previous versions of MMBasic.
OPTION ERROR CONTINUE or OPTION ERROR ABORT	Set the treatment for errors in file input/output. The option CONTINUE will cause MMBasic to ignore file related errors. The program must check the variable MM.ERRNO to determine if and what error has occurred. The option ABORT sets the normal behaviour (ie, stop the program and print an error message). The default is ABORT. Note that this entry only relates to errors reading from or writing to a file. See the ON ERROR command for handling syntax and other program errors.
OPTION EXPLICIT	Placing this command at the start of a program will require that every variable be explicitly declared using the DIM command before it can be used in the program. This option is disabled by default when a program is run. If it is used it must be specified before any variables are used.
OPTION TAB 2 4 8	Set the spacing for the tab key. Default is 2.
PAUSE delay	Halt execution of the running program for 'delay' ms. Note that unlike the Micromite MMBasic a fractional number will not be recognised and will be rounded to the nearest integer. The maximum delay is 2147483647 ms (about 24 days).
PRINT expression [[,;]expression] ... etc	Outputs text to the serial console. Multiple expressions can be used and must be separated by either a: <ul style="list-style-type: none"> • Comma (,) which will output the tab character • Semicolon (;) which will not output anything (it is just used to separate expressions). • Nothing or a space which will act the same as a semicolon. A semicolon (;) at the end of the expression list will suppress the automatic output of a carriage return/ newline at the end of a print statement. When printed, a number is preceded with a space if positive or a minus (-) if negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large floating point numbers (greater than six digits) are printed in scientific number format. The function TAB() can be used to space to a certain column and the string functions can be used to justify or otherwise format strings.
PRINT #fnbr, expression [[,;]expression] ... etc	Same as the normal PRINT command except that the output is directed to a file previously opened for OUTPUT or APPEND as '#fnbr'. See the OPEN command.
QUIT	Terminate the running program, exit MMBasic and close the DOS box.

RANDOMIZE nbr	Seed the random number generator with 'nbr'. On power up the random number generator is seeded with zero and will generate the same sequence of random numbers each time. To generate a different random sequence each time you must use a different value for 'nbr' (the TIMER function is handy for that).
READ variable[, variable]...	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. See also DATA and RESTORE.
REM string	REM allows remarks to be included in a program. Note the Microsoft use of the single quotation mark to denote remarks is also supported and is preferred.
RESTORE [line]	Resets the line and position counters for the READ statement. If 'line' is specified the counters will be reset to the beginning of the specified line. 'line' can be a line number or label. If 'line' is not specified the counters will be reset to the start of the program.
RETURN	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB.
RMDIR dir\$	Remove, or delete, the directory 'dir\$'.
RUN [file\$]	Run the program held in memory. Optionally 'file\$' can be specified used and this will load and run a file (the extension .BAS is added if an extension is not specified).
SELECT CASE value CASE testexp [[, testexp] ...] <statements> <statements> CASE ELSE <statements> <statements> END SELECT	Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression. 'testexp' is the value that 'exp' is to be compared against. It can be: <ul style="list-style-type: none"> • A single expression (ie, 34, "string" or var*5) to which it may equal • A range of values in the form of two single expressions separated by the keyword "TO" (ie, 5 TO 9 or "aa" TO "cc") • A comparison starting with the keyword "IS" (which is optional). For example: IS > 5, IS <= 10. When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true. If 'value' cannot be matched with a 'testexp' it will be automatically matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any <statements> and continue with the code following the END SELECT. When a match is made the <statements> following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT. An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT. Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT...CASE statements can be nested inside the CASE statements of other SELECT...CASE statements. Example: <pre> SELECT CASE nbr% CASE 4, 9, 22, 33 TO 88 statements CASE IS < 4, IS > 88, 5 TO 8 statements CASE ELSE statements END SELECT </pre>

<p>SUB xxx (arg1 [,arg2, ...]) <statements> <statements> END SUB</p>	<p>Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program.</p> <p>'xxx' is the subroutine name and it must meet the specifications for naming a variable.</p> <p>'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1\$) or by specifying the type using AS <type> (ie, arg1 AS STRING) .</p> <p>Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit.</p> <p>You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: MySub a1, a2</p> <p>When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string.</p> <p>Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.</p>
<p>SYSTEM command-line\$</p>	<p>This will exit to DOS, run the ' command-line\$' as if typed in at the command prompt and then return to the running MMBasic program.</p> <p>This can be used to access features of the DOS window that are not available from within MMBasic. For example:</p> <pre>SYSTEM "DIR /b > flist.txt"</pre> <p>will run the DOS DIR command (with brief output) and redirect its output to the file flist.txt. The MMBasic program could then open this file and read the list of files and directories listed by the DIR command.</p>
<p>TIMER = msec</p>	<p>Resets the timer to a number of milliseconds. Normally this is just used to reset the timer to zero but you can set it to any positive integer.</p> <p>See the TIMER function for more details.</p>
<p>TRACE ON or TRACE OFF or TRACE LIST nn</p>	<p>TRACE ON/OFF will turn on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs.</p> <p>TRACE LIST will list the last 'nn' lines executed in the format described above. Note that MMBasic is always logging the lines executed so this facility is always available (ie, it does not have to be turned on).</p>

Functions

Square brackets indicate that the parameter or characters are optional.

ACOS(number)	Returns the inverse cosine of the argument 'number' in radians.
ABS(number)	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ASC(string\$)	Returns the ASCII code for the first letter in the argument 'string\$'.
ASIN(number)	Returns the inverse sine value of the argument 'number' in radians.
ATN(number)	Returns the arctangent of the argument 'number' in radians.
BIN\$(number [, chars])	Returns a string giving the binary (base 2) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
CHR\$(number)	Returns a one-character string consisting of the character corresponding to the ASCII code indicated by argument 'number'.
CINT(number)	Round numbers with fractional portions up or down to the next whole number or integer. For example, 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also INT() and FIX().
COS(number)	Returns the cosine of the argument 'number' in radians.
CWD\$	Returns the current working directory as a string.
DATE\$	Returns the current date based on the internal DOS clock as a string in the form "DD-MM-YYYY". For example, "28-07-2012".
DEG(radians)	Converts 'radians' to degrees.
EOF([#]fnbr)	Will return true if the file previously opened for INPUT with the file number '#fnbr' is positioned at the end of the file. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT\$ function.

EVAL(string\$)	<p>Will evaluate 'string\$' as if it is a BASIC expression and return the result. 'string\$' can be a constant, a variable or a string expression. The expression can use any operators, functions, variables, subroutines, etc that are known at the time of execution. The returned value will be an integer, float or string depending on the result of the evaluation.</p> <p>For example: <code>S\$ = "COS (RAD (30)) * 100" : PRINT EVAL (S\$)</code></p> <p>Will display: 86.6025</p>
EXP(number)	Returns the exponential value of 'number'.
FIX(number)	<p>Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point.</p> <p>For example 9.89 will return 9 and -2.11 will return -2.</p> <p>The major difference between FIX and INT is that FIX provides a true integer function (ie, does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .</p>
HEX\$(number [, chars])	<p>Returns a string giving the hexadecimal (base 16) value for the 'number'.</p> <p>'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).</p>
INKEY\$	<p>Checks the console input buffers and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string.</p> <p>If the input buffer is empty this function will immediately return with an empty string (ie, "").</p>
INPUT\$(nbr, [#]fnbr)	<p>Will return a string composed of 'nbr' characters read from a file previously opened for INPUT with the file number '#fnbr'. This function will read all characters including carriage return and new line without translation.</p> <p>The # is optional. Also see the OPEN command.</p>
INSTR([start-position,] string-searched\$, string-pattern\$)	<p>Returns the position at which 'string-pattern\$' occurs in 'string-searched\$', beginning at 'start-position'.</p> <p>Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern\$' is not found.</p>
INT(number)	<p>Truncate an expression to the next whole number less than or equal to the argument. For example 9.89 will return 9 and -2.11 will return -3.</p> <p>This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function.</p> <p>See also CINT() .</p>
LEFT\$(string\$, nbr)	Returns a substring of 'string\$' with 'nbr' of characters from the left (beginning) of the string.
LEN(string\$)	Returns the number of characters in 'string\$'.

LOC([#]fnbr)	For a file opened as RANDOM this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. The # is optional.
LOF([#]fnbr)	Return the current length of a file in bytes. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.
LCASE\$(string\$)	Returns 'string\$' converted to lowercase characters.
MAX(arg1 [, arg2 [, ...]]) or MIN(arg1 [, arg2 [, ...]])	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.
MID\$(string\$, start) or MID\$(string\$, start, nbr)	Returns a substring of 'string\$' beginning at 'start' and continuing for 'nbr' characters. The first character in the string is number 1. If 'nbr' is omitted the returned string will extend to the end of 'string\$'
OCT\$(number [, chars])	Returns a string giving the octal (base 8) representation of 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
PI	Returns the value of pi.
POS	Returns the current cursor position in the line in characters.
RAD(degrees)	Converts 'degrees' to radians.
RIGHT\$(string\$, number-of-chars)	Returns a substring of 'string\$' with 'number-of-chars' from the right (end) of the string.
RND(number)	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The RANDOMIZE command reseeds the random number generator.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE\$(number)	Returns a string of blank spaces 'number' bytes long.
SQR(number)	Returns the square root of the argument 'number'.
STR\$(number) or STR\$(number, m) or STR\$(number, m, n)	Returns a string in the decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and

or STR\$(number, m, n, c\$)	<p>negative numbers with the minus symbol. If 'm' is positive then only the negative symbol will be used.</p> <p>'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number.</p> <p>'c\$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument).</p> <p>Examples:</p> <table> <tr> <td>STR\$(123.456)</td><td>will return "123.456"</td></tr> <tr> <td>STR\$(123.456, 6)</td><td>will return " 123.456"</td></tr> <tr> <td>STR\$(123.456, -6)</td><td>will return " +123.456"</td></tr> <tr> <td>STR\$(-123.456, 6)</td><td>will return " -123.456"</td></tr> <tr> <td>STR\$(-123.456, 6, 5)</td><td>will return " -123.45600"</td></tr> <tr> <td>STR\$(-123.456, 6, -5)</td><td>will return " -1.23456e+02"</td></tr> <tr> <td>STR\$(53, 6)</td><td>will return " 53"</td></tr> <tr> <td>STR\$(53, 6, 2)</td><td>will return " 53.00"</td></tr> <tr> <td>STR\$(53, 6, 2, "*")</td><td>will return "*****53.00"</td></tr> </table>	STR\$(123.456)	will return "123.456"	STR\$(123.456, 6)	will return " 123.456"	STR\$(123.456, -6)	will return " +123.456"	STR\$(-123.456, 6)	will return " -123.456"	STR\$(-123.456, 6, 5)	will return " -123.45600"	STR\$(-123.456, 6, -5)	will return " -1.23456e+02"	STR\$(53, 6)	will return " 53"	STR\$(53, 6, 2)	will return " 53.00"	STR\$(53, 6, 2, "*")	will return "*****53.00"
STR\$(123.456)	will return "123.456"																		
STR\$(123.456, 6)	will return " 123.456"																		
STR\$(123.456, -6)	will return " +123.456"																		
STR\$(-123.456, 6)	will return " -123.456"																		
STR\$(-123.456, 6, 5)	will return " -123.45600"																		
STR\$(-123.456, 6, -5)	will return " -1.23456e+02"																		
STR\$(53, 6)	will return " 53"																		
STR\$(53, 6, 2)	will return " 53.00"																		
STR\$(53, 6, 2, "*")	will return "*****53.00"																		
STRING\$(nbr, ascii) or STRING\$(nbr, string\$)	Returns a string 'nbr' bytes long consisting of either the first character of string\$ or the character representing the ASCII value 'ascii' which is a decimal number in the range of 32 to 126.																		
TAB(number)	Outputs spaces until the column indicated by 'number' has been reached.																		
TAN(number)	Returns the tangent of the argument 'number' in radians.																		
TIME\$	Returns the current time based on the internal DOS clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00".																		
TIMER	Returns the elapsed time in milliseconds (eg, 1/1000 of a second) since reset. The timer is reset to zero when MMBasic is started and you can also reset it by using TIMER as a command. Note that under DOS the timer will reset to 0 for each subsequent 24 hour interval that elapses.																		
UCASE\$(string\$)	Returns 'string\$' converted to uppercase characters.																		
VAL(string\$)	Returns the numerical value of the 'string\$'. If 'string\$' is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.																		

Obsolete Commands and Functions

These commands and functions are mostly included to assist in converting programs written for Microsoft BASIC. For new programs the corresponding commands in MMBasic should be used.

Note that these commands may be removed in the future.

IF condition THEN linenbr	For Microsoft compatibility a GOTO is assumed if the THEN statement is followed by a number. A label is invalid in this construct. New programs should use: IF condition THEN GOTO linenbr label
SPC(number)	This function returns a string of blank spaces 'number' bytes long. It is similar to the SPACE\$() function and is only included for Microsoft compatibility.
TROFF	Turns the trace facility off; see TRON.
TRON	Turns on the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. New programs should use the TRACE command.
WHILE expression WEND	WHILE initiates a WHILE-WEND loop. The loop ends with WEND, and execution reiterates through the loop as long as the 'expression' is true. This construct is included for Microsoft compatibility. New programs should use the DO WHILE ... LOOP construct.