MMBasic DOS/Windows Version

User Manual

MMBasic Ver 5.05.04

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

http://geoffg.net/micromite.html

and http://mmbasic.com

Copyright

Big thanks to the members of the Back Shed forum who beta tested this version of MMBasic and found many, many bugs. Thanks guys.

Contents

Introduction

MMBasic is an implementation of the BASIC language with floating point, integers and string variables, long variable names, arrays of floats/integers/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/Windows version of MMBasic but for a more detailed explanation it is recommended that you read chapters 3 and 4 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, graphics, specialised 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 console window.

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.

It is also possible to edit the program from within MMBasic. The EDIT command will invoke the internal MMBasic editor. This provides a colour coded display with keywords in one colour, comments in another, etc. With a single keystroke it is possible to save and run the program. If an error occurs a single keystroke at the command prompt will restart the editor with the cursor positioned at the error, so the edit/run/test cycle is very fast.

Another editing option is the WEDIT command which will use a Windows editor to edit the last file supplied on the command line or used in RUN, LOAD or SAVE. When the editor is closed MMBasic will automatically reload the program from the file. By default the editor used is Notepad but another editor can be specified by setting the DOS environment variable MMEDITOR to the path of the alternative editor.

When MMBasic is running you can also copy and paste text to and from MMBasic using the standard copy and paste facilities of the console window.

Differences to the Micromite 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.
- The communications protocols I2C, SPI and 1-wire. **However, serial is implemented.**
- Serial console and commands which operate on the serial console such as AUTOSAVE and XMODEM

DOS MMBasic has a number of extra commands and functions:

- QUIT which will close MMBasic and return to the operating system.
- SYSTEM which will issue a DOS command to the operating system.
- SETTITLE which will change the title in the console window.
- The read only variable MM.CMDLINE$ will return the DOS command line used to start MMBasic.
- The read only variables MM.VRES and MM.HRES will return the size of the window (in characters).
- CURSOR which will position the cursor on the console window.
- COLOUR which will set the foreground and background colours for subsequent character output.
- CLS will clear the console window.
- WEDIT will invoke an external editor (the standard MMBasic EDIT command is also included).

Double Precision Floating Point

All floating point numbers in this version of MMBasic are double precision (the Micromite version uses single precision while the Micromite Plus also uses double precision). This means that calculations will have a far greater range and will be accurate to about 16 decimal digits. When printing a double precision floating point number MMBasic will display up to 9 digits (this can be changed with the Str$() function).

Environment Variables

Windows environment variables can be used to change some characteristics of MMBasic.

MMDIR can be set to the default directory that MMBasic is to start running in. For example:

```
SET MMDIR=C:\Temp
```

will start MMBasic running in the directory C:\Temp. Without this the starting directory will be specified by the operating system and will vary depending on how MMBasic is started. The directory path must be quoted if it includes spaces. eg: `SET MMDIR="C:\Program Files"`

MMCOLOURS can be used to change the default colours used by the internal editor (the EDIT command). The colours are specified as a sequence of 7 numbers separated by commas (,). See the COLOUR command for the valid numbers and the corresponding colours.

The sequence is:

NormalText,Background,Comment,Keyword,QuotedText,Number,StatusLine

Only decimal digits and comma characters are allowed (no spaces).

NormalText is the colour used for non specific text and *Background* is the colour used for the background. The others are used to indicate special features of the program - comments, keywords, text constants (within double quotes), numeric constants and finally the status line at the bottom of the screen.

For example, the following uses an alternate colour scheme with black text on a white background:

```
SET MMCOLOURS=0,7,6,4,2,0,5
```

This will remove all colour coding from the editor's display except for the status line which will be purple:

```
SET MMCOLOURS=7,0,7,7,7,7,5
```

MMEDITOR can be used to change the default editor used in the WEDIT command. For example:

```
SET MMEDITOR= C:\Edt.exe
```

The path must be quoted if it includes spaces. eg: `SET MMEDITOR= "C:\Program Files\Edt.exe"`

In Windows the environment variables can be set using:

Control Panel => System => Advanced system settings => Environment Variables…

There are also other ways of accomplishing the same thing… see the Windows documentation for details.

Note that by right clicking in the title bar of a console window and selecting Properties it is possible to set many default parameters for the console including the startup colours, font, window size, etc..

Shortcut Keys

At the command prompt you can use a function key to insert the following commands:

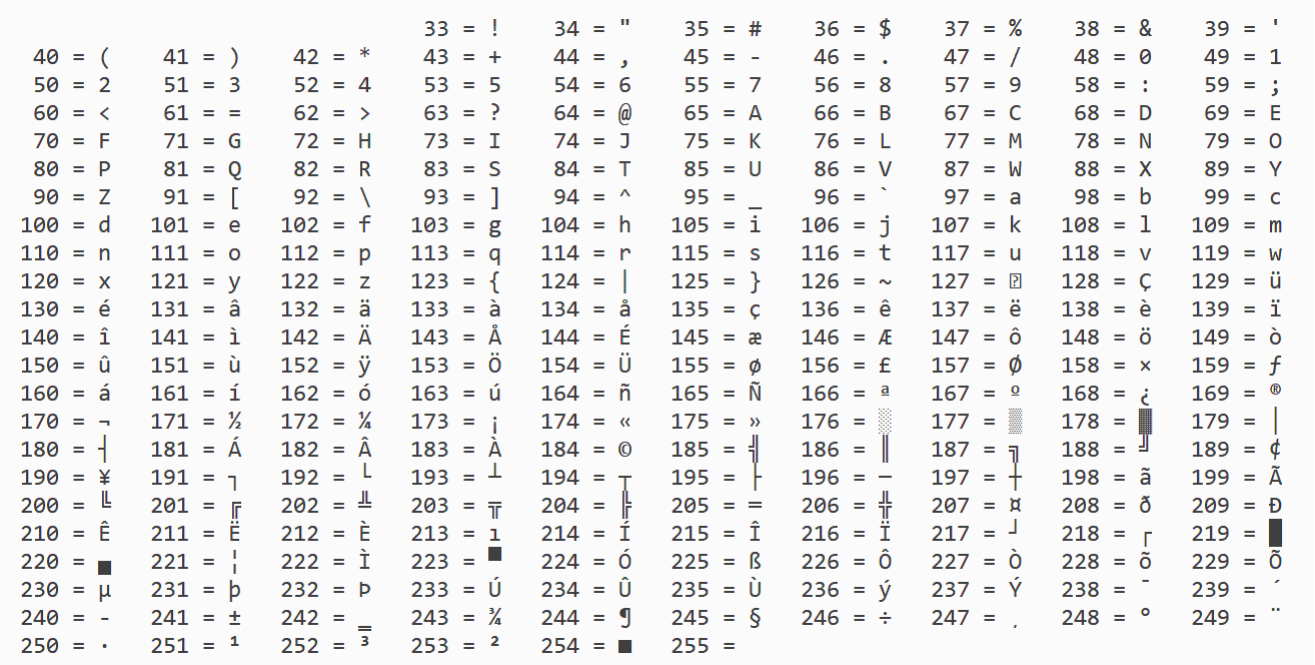
F2	RUN
F3	LIST
F4	EDIT
F5	WEDIT

Pressing the key will insert the text at the command prompt, just as if it had been typed on the keyboard.

Character Set

Special characters can be printed by specifying the number of the character using the CHR$() function. The range of characters includes line drawing characters and accented characters and the full list (for a Windows 10 computer) is illustrated below.

Note that the character values are decimal numbers:



Function Keys

With the console input MMBasic uses the following codes to identify function keys on the keyboard:

Keyboard Key	Key Code (Hex)	Key Code (Decimal)
Up Arrow	80	128
Down Arrow	81	129
Left Arrow	82	130
Right Arrow	83	131
Insert	84	132
Home	86	134
End	87	135
Page Up	88	136
Page Down	89	137
Delete	7F	127
F1	91	145
F2	92	146
F3	93	147
F4	94	148
F5	95	149
F6	96	150
F7	97	151
F8	98	152
F9	99	153
F10	9A	154
F11	9B	155
F12	9C	156

AUTORUN.BAS

If a valid program file name is not provided on the command line MMBasic will attempt to load a program called AUTORUN.BAS and run it. It will first look for this file in the default directory provided by the Windows operating system and if it was not found there it will look for it in the root directory of the C: drive.

This program can be used to setup MMBasic in some way (for example set the text colour) or perhaps display a menu. Within this program the LOAD command can be used to load and run another program. It is also possible to use the NEW command which will clear the program memory and return to the command prompt.

Serial Communications

Serial ports on the computer can be accessed using their Windows COM designations (eg, COMxx). After being opened they will have an associated file number and you can use any commands that operate with a file number to read and write to/from the serial port. A serial port is also closed using the CLOSE command.

The following is an example:

```
OPEN "COM12: baud=4800" AS #5 ' open COM12 with a speed of 4800 baud
PRINT #5, "Hello"             ' send the string "Hello" out of the serial port
dat$ = INPUT$(20, #5)         ' get up to 20 characters from the serial port
CLOSE #5                      ' close the serial port
```

The OPEN Command

A serial port is opened using the command:

```
OPEN comspec$ AS #fnbr
```

'fnbr' is the file number to be used. It must be in the range of 1 to 10. The # is optional.

'comspec$' is the communication specification and is a string (it can be a string variable) specifying the serial port to be opened and optional parameters.

The basic form is `"COMn: baud=nnn"` where:

- 'n' is the serial port number (eg, COM1:, COM4:, or COM41:).
- 'nnn' is the baud rate. This can be any standard baud rate from 110 to 256000 bits per second. The default is whatever the serial port has been configured for in the Windows Device Manager.

Other optional settings can also be used in 'comspec$' with each setting separated by a space.

These include:

```
parity=p            data=d              stop=d
to={on|off}         xon={on|off}        odsr={on|off}
octs={on|off}       dtr={on|off|hs}     rts={on|off|hs|tg}
idsr={on|off}
```

For example:

```
"COM8: baud=9600 parity=y data=7 stop=2"
```

Examples

Opening a serial port using all the defaults set for the port in Device Manager:

```
OPEN "COM2:" AS #2
```

Opening a serial port specifying only the baud rate (4800 bits per second):

```
OPEN "COM32: baud=4800" AS #1
```

Opening a serial port specifying the baud rate (9600 bits per second) and XON/XOFF flow control:

```
OPEN "COM1:9600 xon=on" AS #8
```

Reading and Writing

Once a serial port has been opened you can use any command or function that uses a file number to write and read from the port. Generally the PRINT command is the best method for transmitting data and the INPUT$() function is the most convenient way of getting data that has been received. When using the INPUT$() function the number of characters specified will be the maximum number of characters returned but it could be less if there are less characters in the receive buffer. In fact the INPUT$() function will immediately return an empty string if there are no characters available in the receive buffer.

The EOF() function will return true if there are no characters waiting.

Reads are buffered; this means that data will be collected even when a read is not being executed. Writes however are not buffered so any command which writes to a serial port will only return when the data is sent.

Serial ports can be closed with the CLOSE command. This will free up any resources (memory, etc) reserved for the serial port. A serial port is also automatically closed when commands such as RUN and NEW are issued.

These functions will communicate over hardware ports (typically COM1: and COM2:) and also over virtual serial ports such as created for USB to serial converters or bridges.

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.
- Except for PRINT, INPUT and LINE INPUT 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 and saved. Alternatively the program can be edited from within MMBasic (it is automatically reloaded following the edit).

□ LOAD fname$ [, R]
Load a BASIC program from disc. The optional suffix ",R" will cause the program to be run .

□ RUN fname$
Load and run a BASIC program from disc.

□ SAVE fname$
Save the current program to a file.

□ EDIT
Edit the current program using Notepad and reload when finished.

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

□ FILES [dname $]
Search the current or specified 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.

□ 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 opened for input as '#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.

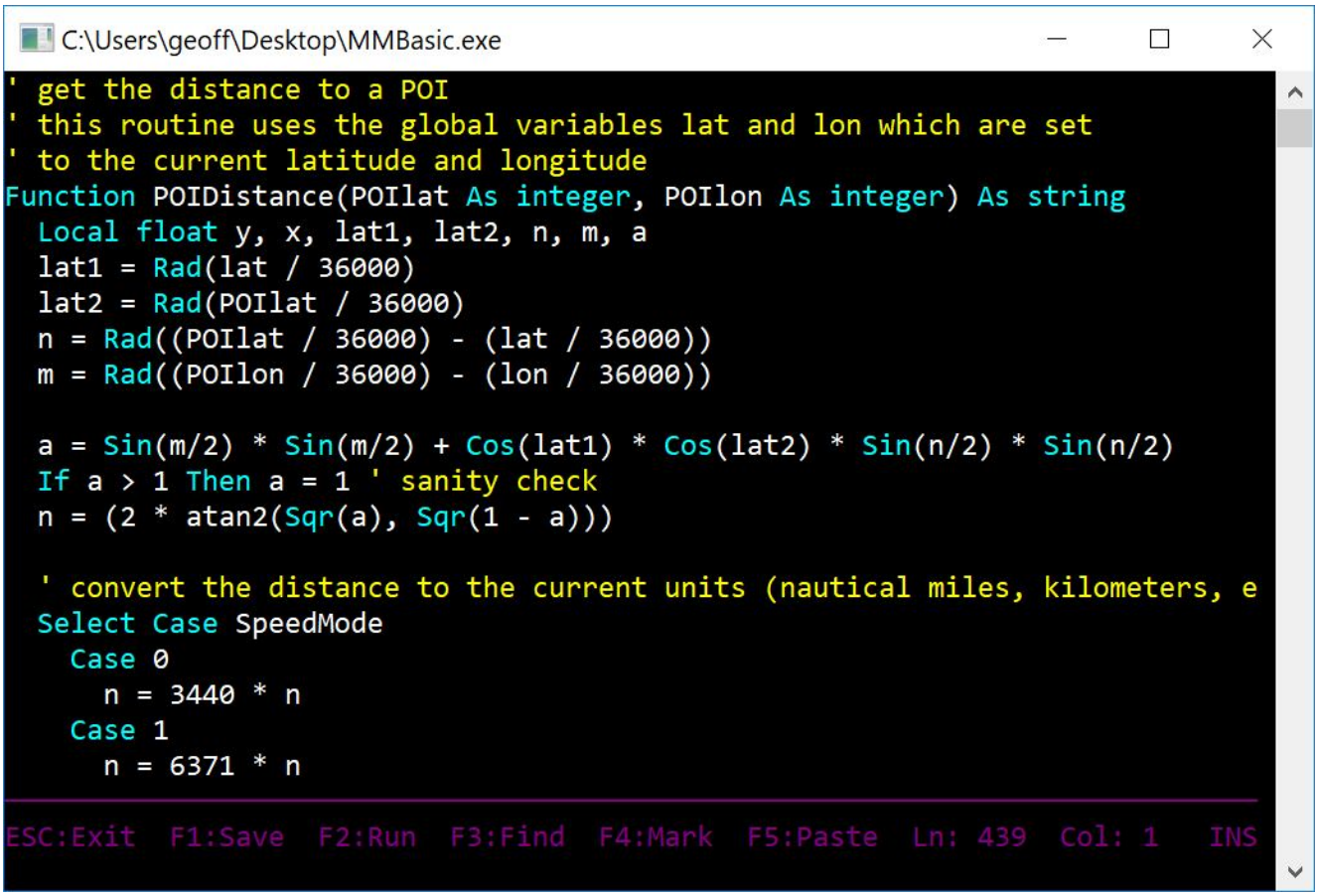
```
RecLen = 64
OPEN "test.dat" FOR RANDOM AS #1
DO
   abort: PRINT
   PRINT "Number of records in the file =" LOF(#1)/RecLen
   INPUT "Command (r = read,w = write, a = append, q = quit): ", cmd$
   IF cmd$ = "q" THEN CLOSE #1 : END
   IF cmd$ = "a" THEN
      SEEK #1, LOF(#1) + 1
   ELSE
      INPUT "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
```

Full Screen Editor

The full screen program editor is invoked with the EDIT command. The cursor will be automatically positioned at the last place that you were editing at or, if your program had just been stopped by an error, the cursor will be positioned at the line that caused the error.



If you are used to an editor like Notepad you will find that the operation of this editor is familiar. The arrow keys will move your cursor around in the text, home and end will take you to the beginning or end of the line. Page up and page down will do what their titles suggest. The delete key will delete the character at the cursor and backspace will delete the character before the cursor. The insert key will toggle between insert and overtype modes.

About the only unusual key combination is that two home key presses will take you to the start of the program and two end key presses will take you to the end.

At the bottom of the screen the status line will list the various function keys used by the editor and their action. In more details these are:

ESC	This will cause the editor to abandon all changes and return to the command prompt with the program memory unchanged. If you have changed the text you will be asked if you really want to abandon your changes.
F1: SAVE	This will save the program to program memory and return to the command prompt. If the program had been previously loaded from, or saved to a file, MMBasic will also update the file on disc (the file name is shown in the title bar of the window).
F2: RUN	This will save the program (as above) and immediately run it.
F3: FIND	This will prompt for the text that you want to search for. When you press enter the cursor will be placed at the start of the first entry found.
F6	Once you have used the search function you can repeatedly search for the same text by pressing F6.
F4: MARK	This is described in detail below.
F5: PASTE	This will insert (at the current cursor position) the text that had been previously cut or copied (see below).

If you pressed the mark key (F4) the editor will change to the *mark mode*. In this mode you can use the arrow keys to mark a section of text which will be highlighted in reverse video. You can then delete, cut or copy the marked text. In this mode the status line will change to show the functions of the function keys in the mark mode. These keys are:

ESC	Will exit mark mode without changing anything.
F4: CUT	Will copy the marked text to the clipboard and remove it from the program.
F5: COPY	Will just copy the marked text to the clipboard.
DELETE	Will delete the marked text leaving the clipboard unchanged.

You can also use control keys instead of the functions keys listed above. These control keystrokes are:

LEFT	Ctrl-S	RIGHT	Ctrl-D	UP	Ctrl-E	DOWN	Ctrl-X
HOME	Ctrl-U	END	Ctrl-K	PageUp	Ctrl-P	PageDn	Ctrl-L
DEL	Ctrl-]	INSERT	Ctrl-N	F1	Ctrl-Q	F2	Ctrl-W
F3	Ctrl-R	ShiftF3	Ctrl-G	F4	Ctrl-T	F5	Ctrl-Y

The best way to learn the full screen editor is to simply fire it up and experiment.

The editor is a very productive method of writing a program. With the command EDIT you can quickly load and edit your program. Then, by pressing the F2 key, you can save and run the program. If your program stops with an error you can press the function key F4 which will run the command EDIT and place you back in the editor with the cursor positioned at the line that caused the error. This edit/run/edit cycle is very fast.

The editor will colour code the edited program with keywords, numbers and comments displayed in different colours. You can change the colours by using the environment variable MMCOLOURS (see the full description at the start of this manual).

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/gwbasman/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.CMDLINE$	The DOS command line used to start MMBasic.
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.VER	The version number of the firmware as a floating point number in the form aa.bbcc 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 5.03 and version 5.03.01 will return 5.0301.
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.
MM.HRES MM.VRES	The current height and width of the console window in characters (not pixels). During execution of a program the window size can be adjusted by the user and these variables will be automatically updated.

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) or COM port previously opened with the file number ‘#fnbr’. The # is optional. Also see the OPEN command.
CLS	Clears all text in the console window.
CLEAR	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
COLOUR fc, bc	Sets the colours for subsequent characters written to the console window. 'fc' is the foreground colour and 'bg' is the background colour. These values can be any one of the following: 0 = Black 8 = Gray 1 = Blue 9 = Bright Blue 2 = Green 10 = Bright Green 3 = Cyan 11 = Bright Cyan 4 = Red 12 = Bright Red 5 = Purple 13 = Bright Purple 6 = Yellow 14 = Bright Yellow 7 = White 15 = Bright White This command can also be spelt as COLOR.
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.

CURSOR x, y	Positions the cursor in the console window. 'x' and 'y' are the horizontal and vertical screen coordinates (in characters). The top left hand corner has the coordinates x = 0 and y = 0. Note that the coordinates are in characters (not pixels) and if the coordinates are outside of the window area the command will be silently ignored.
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 FLOAT or INTEGER or STRING (the type can be prefixed by the keyword *AS* - as in AS FLOAT)
'init' is the value to initialise the variable and consists of:
= <expression>
For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.
 	
Examples:
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 STRING strn(200) LENGTH 20
DIM a = 1234, b = 345
DIM STRING strn = "text"
DIM x%(3) = (11, 22, 33, 44)
	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, LOCAL or STATIC 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:
	`DIM nbr%, amount!, name$`
	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:
	`DIM STRING first_name, last_name, city`
	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.
	For example:
	`DIM amount AS FLOAT, name AS STRING`
	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:
	`DIM STRING city = "Perth", house = "Brick"`
	The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter "Defining and Using Variables" for more examples of the syntax.
	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:
	`DIM array(10, 20)`
	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.
	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).
	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.
	For example: `DIM STRING s(5, 10)` will declare a string array with 66 elements consuming 16,896 bytes of memory while:
	`DIM STRING s(5, 10) LENGTH 20`
	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.
	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. This keyword can also be used with non array string variables but it will not save any memory.
	In the above example you can also use the Microsoft syntax of specifying the type after the length qualifier. For example:
	`DIM s(5, 10) LENGTH 20 AS STRING`
	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:
	`DIM INTEGER nbr(4) = (22, 44, 55, 66, 88)`
	or `DIM s$(3) = ("foo", "boo", "doo", "zoo")`
	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.

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.
EDIT	Invoke the full screen editor. On entry the cursor will be automatically positioned at the last line edited or, if there was an error when running the program, the line that caused the error. The editing keys are: Left/Right arrows — Moves the cursor within the line. Up/Down arrows — Moves the cursor up or down a line. Page Up/Down — Move up or down a page of the program. Home/End — Moves the cursor to the start or end of the line. A second Home/End will move to the start or end of the program. Delete — Delete the character over the cursor. This can be the line separator character and thus join two lines. Backspace — Delete the character before the cursor. Insert — Will switch between insert and overtype mode. Escape Key — Will close the editor without saving (confirms first).

	F1 Save the edited text and exit. F2 Save, exit and run the program. F3 Invoke the search function. F6 Repeat the search using the text entered with F3. F4 Mark text for cut or copy (see below). F5 Paste text previously cut or copied. When in the mark text mode (entered with F4) the editor will allow you to use the arrow keys to highlight text which can be deleted, cut to the clipboard or simply copied to the clipboard. The status line will change to indicate the new functions of the function keys. The editor will work with lines wider than the screen but characters beyond the screen edge will not be visible. You can split such a line by inserting a new line character and the two lines can be later rejoined by deleting the inserted new line character. When a save is executed MMBasic will also update the file that originally held the program (derived from RUN, LOAD or from the command line and shown in the title bar of the window). Otherwise it will prompt for the file name to save to. The editor will colour code the program to highlight keywords, comments, etc. These colours can be changed by specifying an environment variable – see the description at the start of this manual.
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.
END	End the running program and return to the command prompt.

ENDIF
or
END IF
	Terminates a multiline IF statement.
	See the multiline IF statement for more details.

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).
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 [fspec]	Lists files. Same as the DOS DIR command. If 'fspec' is specified that will be added to the DIR command line.
FOR counter = start TO finish [STEP increment]	Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' greater than '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.

FUNCTION xxx (arg1 [,arg2, …]) [AS <type>}
<statements>
<statements>
xxx = <return value>
END FUNCTION
	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 and have the correct type 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 and therefore may be accessed after the function has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference and must be the correct type.
	You must not jump into or out of a function using commands like GOTO, GOSUB, etc. Doing so will have undefined side effects including ruining your day.

GOTO target	Branches program execution to the target, which can be a line number or a label.

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 or COM port previously opened as ‘#fnbr’. See the OPEN command.
KILL file$	Deletes the file specified by ‘file$’. If there is an extension it must be specified.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command.
LINE INPUT [prompt$,] 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$’.
LINE INPUT #fnbr, string-variable$	Same as the LINE INPUT command except that the input is read from a file or COM port previously opened as ‘#fnbr’. See the OPEN command.
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.
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.
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$’ which is a string variable or constant.
NAME old$ AS new$	Rename a file or a directory from ‘old$’ to ‘new$’. Both are strings. A directory path can be used in both 'old$' and 'new$'.
NEW	Deletes the program and clears all variables.
NEXT [counter-variable] [, counter-variable], etc	NEXT comes at the end of a FOR-NEXT loop; see FOR. 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: NEXT x, y, z
ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR	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. 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. ON ERROR IGNORE will cause any error to be ignored. 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. If an error occurs and is ignored or 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. ON ERROR IGNORE can make it very difficult to debug a program so it is strongly recommended that only ON ERROR SKIP be used.
OPEN fname$ FOR mode AS [#]fnbr	Opens a file for reading or writing. ‘fname’ is the filename with an optional extension separated by a dot (.). Long file names with upper and lower case characters are supported. 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). For example OPEN "..\dir1\dir2\filename.txt" FOR INPUT AS #1 ‘mode’ is INPUT, OUTPUT, APPEND or RANDOM. 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. 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). 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. ‘fnbr’ is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT and CLOSE commands as well as the EOF() and INPUT$() functions all use ‘fnbr’ to identify the file being operated on. See also ON ERROR and MM.ERRNO for error handling.

OPEN comspec$ AS [#]fnbr	Opens a serial COM port for reading or writing.
	'comspec$' is the communication specification and is a string variable or constant.
	The basic form is `"COMn baud=nnn"` where:
	• 'n' is the serial port number (eg, COM1:, COM4:, or COM41:).
	• 'nnn' is the baud rate. This can be any standard baud rate from 110 to 256000 bits per second. The default is set in Device Manager.
	Other optional settings can also be used in 'comspec$' with each setting separated by a space.
	These include:
	`parity=p data=d stop=d`
	`to={on|off} xon={on|off} odsr={on|off}`
	`octs={on|off} dtr={on|off|hs} rts={on|off|hs|tg}`
	`idsr={on|off}`
	For example:
	`"COM8: baud=9600 parity=y data=7 stop=2"`
	'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT and CLOSE commands as well as the EOF() and INPUT$() functions all use 'fnbr' to identify the file being operated on.
	See also ON ERROR and MM.ERRNO for error handling.

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 when a program is run.
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 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.
PRINT expression [[,;]expression] … etc	Outputs text to the serial console. Multiple expressions can be used and must be separated by either a: • 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 or COM port previously opened 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'. When MMBasic is started 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.
RMDIR dir$	Remove, or delete, the directory 'dir$'.
RUN [file$]	Run the program held in memory. Optionally 'file$' can be specified used and this will load and run a file (the extension .BAS is added if an extension is not specified).
SAVE file$	Save the program held in memory to the file 'file$'. The extension .BAS will be 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: • 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:
	`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`

SETTITLE string$	Will set the title of the console window to 'string'.
STATIC variable [, variables] See DIM for the full syntax.	Defines a list of variable names which are local to the subroutine or function. These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command). This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters. Static variables can be initialised to a value. This initialisation will take effect only on the first call to the subroutine (not on subsequent calls).
SUB xxx (arg1 [,arg2, …]) <statements> <statements> END SUB	Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program. 'xxx' is the subroutine name and it must meet the specifications for naming a variable. '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$) or by specifying the type using `AS <type> (ie, arg1 AS STRING)`. Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: `MySub a1, a2` When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.
SYSTEM command-line$	This will exit to DOS, run the 'command-line$' as if typed in at the command prompt and then return to the running MMBasic program. This can be used to access features of the DOS window that are not available from within MMBasic. For example: `SYSTEM "DIR /b > flist.txt"` will run the DOS DIR command (with brief output) and redirect its output to the file flist.txt. The MMBasic program could then open this file and read the list of files and directories listed by the DIR command. Note that 'command-line$' must be a string constant (surrounded by double quotes) or a string expression.

TIMER = msec	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. See the TIMER function for more details.
TRACE ON or TRACE OFF or TRACE LIST nn	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. 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).
WEDIT	This will invoke the Notepad editor with the file that was last specified on the MMBasic command line, or loaded using RUN/LOAD or saved using the SAVE command. Notepad can then be used to edit the program. When Notepad is terminated MMBasic will automatically reload the file into program memory. If a file operation has not been previously used MMBasic will save the current program to a temporary file for editing. To change the editor used by MMBasic set the Windows environment variable MMEDITOR to the path of the editor to be used. This command is a convenient method of editing a program from within MMBasic. Note that at the command prompt the F5 key will trigger the WEDIT command so a program can be edited via a single keystroke.

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. For a serial COM port it will return true if there are no characters currently waiting in the input buffer to be read. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT$ function.

EVAL(string$)	Will evaluate 'string$' as if it is a BASIC expression and return the result. 'string$' can be a constant, a variable or a string expression. The expression can use any operators, functions, variables, subroutines, etc that are known at the time of execution. The returned value will be an integer, float or string depending on the result of the evaluation. For example: S$ = "`COS(RAD(30)) * 100" : PRINT EVAL(S$)` Will display: `86.6025`
EXP(number)	Returns the exponential value of 'number'.
FIX(number)	Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point. For example 9.89 will return 9 and -2.11 will return -2. The major difference between FIX and INT is that FIX provides a true integer function (ie, does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .
HEX$(number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
INKEY$	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. If the input buffer is empty this function will immediately return with an empty string (ie, "").
INPUT$(nbr, [#]fnbr)	Will return a string composed of up to ‘nbr’ characters read from a file or COM port previously with the file number ‘#fnbr’. This function will read all characters including carriage return and new line without translation. If there are less than 'nbr' characters waiting this function will return with as many that are waiting to be read, this also means that it could return with an empty string if there are no characters waiting. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, string-pattern$)	Returns the position at which 'string-pattern$' occurs in 'string-searched$', beginning at 'start-position'. Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern$' is not found.
INT(number)	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. This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function. See also CINT() .
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. For a serial COM port this will return the number of bytes waiting in the receive queue to be read. The # is optional.
LOF([#]fnbr)	Return the current length of a file opened with the file number '#fnbr' in bytes. For a serial COM port this function will return the number of characters waiting to be sent and in the current implementation this will always be zero as the serial output is unbuffered. 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.
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$(number, m, n) or STR$(number, m, n, c$)	Returns a string in the decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the minus symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' 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). Examples: 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.

GOSUB target	Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN. New programs should use defined subroutines (ie, SUB…END SUB).
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
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. New programs should use SELECT CASE.
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.
POS	For the console, returns the current cursor position in the line in characters.
RETURN	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB.
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.