MMBasic for Windows

User Manual

MMBasic Ver 5.07.03b24

Draft Ver:0.97

For more details on MMBasic go to

http://geoffg.net/maximite.html

and http://mmbasic.com

About

The port of MMBasic to Microsoft Windows was conceived and developed by Peter Mather (matherp on TheBackShed Forum) who also led the development project.

It is based on MMBasic developed by Geoff Graham and uses the MMBasic interpreter written by Geoff Graham (http://geoffg.net).

Support

Support questions should be raised on the Back Shed forum (http://www.thebackshed.com/forum/Microcontrollers) where there are many enthusiastic MMBasic users who would be only too happy to help.

Copyright and Acknowledgments

The MMBasic software is copyright 2011-2021 by Geoff Graham and Peter Mather 2016-2021.

MOD file support was written by Jean François DEL NERO (hxcmod.c).

WAV, MP3, and FLAC file support are copyright 2019 David Reid.

PNG support is copyright 2005-2010 Lode Vandevenne and 2010 Sean Middleditch.

The editor and file manager are based on code copyright 2016 Salvatore Sanfilippo and documentation from paileyq@gmail.com

The turtle graphics support including the polygon fill algorithm are copyright Mike Lam 2015.

Marcel Rodrigues wrote the GIF decoder.

Maury Quijada wrote the image resize and image rotate code.

CRC code based on work by Rob Tillaart.

The compilation of development posts on TheBackShed Forum by Mick (Mixtel90).

Implementation using Linux/Wine by Volhout on TheBackShed Forum (see Annex E.)

Microsoft and MS Windows are trademarks of Microsoft Corporation.

Graphics engine by One Lone Coder and the olcPixelGameEngine. https://github.com/OneLoneCoder

The compiled object code (the .exe file) for MMBasic for Windows 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

This manual relies heavily on content from the following manuals be Geoff Graham and Peter Mather.

Micromite User Manual
Micromite Plus User Manual
MMBasic for DOS User Manual
Colour Maximite 2 User Manual

Much information is also gleaned from posts (mainly by Peter Mather and collated by Mixtel90) in various threads relating to MMBasic for Windows (also referred to as MMB4W) on TheBackShed Forum. Many contributors may recognise their work within this document and are thanked for their contributions.

The assembler of this manual is Doug Pankhurst (panky on TheBackShed Forum). It is distributed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0)

Conventions used in this document:

This icon is used to indicate some special idea or tip about the current subject.

Additional information or qualifier about the current subject.

*** Warning or caveat applying to the current subject ***

```
Command line examples may be shown this way or as below
```

```
' Text like this with the blue background is MMBasic code
' which you can copy and paste
' into a program such as MMEdit (by Jim Hiley) for upload to MMB4W.
' Code examples have been tested but no guarantees are made regarding
' possible errors.
```

Table of Contents

Introduction 7
Limitations 7
Quick Start Tutorial 8
Installing and Running MMBasic for Windows 8
Developing Programs 9
Differences to the Micromite version of MMBasic 10
Double Precision Floating Point 10
Graphics Modes and the Page Command 10
Keyboard Layout 12
Commands and Functions 12
Timer and Settick 12
Command Prompt 13
Your First Program 14
Something More Complicated 14
Using MMBasic 15
Commands and Program Input 15
Editing the Command Line 15
Console Keyboard/Display 15
Keyboard Shortcuts 15
Line Numbers and Program Structure 16
All Programs Are Run From The Inbuilt File System 16
Running Programs 17
Expressions 17
Standards and Compatibility 17
Timing 17
Watchdog Timer 18
The Console Window 18
Configuring the Console window 18
Full Screen Editor 19
Mouse Support 20
Colour Coded Editor Display 21
Variables and Expressions 22
Variables 22
Constants 23
Option Default 23
Option Explicit 23
Dim and Local 23
Static 24
Const 25
Expressions and Operators 26
Strings 26
Mixing Floating Point and Integers 27
64-bit Unsigned Integers 27
Subroutines and Functions 28
Subroutines 28
Local Variables 28
Functions 28
Passing Arguments by Reference 29
Passing Arguments by Value 29
Passing Arrays 30
Early Exit 30
Recursion 31
Basic Graphics 32
Screen Coordinates 32
Read Only Variables 32
Colours 32
Fonts 35
Drawing Commands 36
Example of Basic Graphics 37
Rotated Text 38
Transparent Text 38
Displaying Images 38

Advanced Graphics....39
Defining Controls....39
Interacting with Controls....45
GUI Sub-commands....45
Example Program....47
Advanced Graphics Programming Techniques....50
The User Should Be In Control....50
Program Structure....50
Disable Invalid Controls....51
Use Constants for Control Reference Numbers....51
The Main Program Is Still Running....52
Use Interrupts and SELECT CASE Statements....52
Mouse Button Interrupt....53
Keep Interrupts Very Short....53
Multiple Screens....53
Multiple Interrupts....54
Using Basic Drawing Commands....55
Overlapping Controls....55
File System Support....56
Program Management Commands....56
File Access Within a Program....56
File and Directory Management....57
Play Audio Files....57
Load and Save Images....58
Sequential File Access....58
Random File Access....59
Audio Output....60
Playing WAV, MP3 and FLAC Files....60
Background Music....60
Generating Sine Waves....60
Specialised Audio Output....61
Using PLAY....61
Interrupts....62
Game Playing Features....63
Screen Resolution, Colour Depth and Pages....63
Scrolling and Sprites....63
Displaying Images....64
Fonts....64
BLIT Command....64
GAMEPAD Use....64
Porting Programs....67
Variables....67
Floating Point....67
Graphic Commands....67
Fonts....67
BLIT....67
Sprites....67
SOUND and TONE....68
File Related Commands....68
Special Devices....68
CONFIG Commands....68
Error Handling....68
Touch....68
Random Number Generator....69
MMBasic Implementation Characteristics....69
MMBasic Language Reference....70
Predefined Read Only Variables....70
Operators....72
MMBasic Configuration Options....73
Commands....76
Functions....108
Obsolete Commands and Functions....119
Appendix A. Serial Communications....120
The OPEN Command....120

Examples....121
Reading and Writing....121
Interrupts....121
Appendix B. Connecting to the Internet....122
Appendix C. Sprites....124
Appendix D. Special Keyboard Keys....126
Appendix E. Serial Ports in Linux/Wine....127
- By Volhout....127
Appendix F. Cyclic Redundancy Check (CRC)....129
The MMBasic CRC function:....130

Introduction

MMBasic for Windows (sometimes referred to as MMB4W) is an implementation of the BASIC language with floating point, integer 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 an Microsoft Windows environment. It uses the same syntax and basic commands as the Colour Maximite 2 (CMM2) version of MMBasic and can be used for testing programs in a convenient environment.

MMBasic has been tested on Microsoft Windows versions 7 through 11. It will also run under Linux using the Wine environment (see Annex E. for details). Some systems running MS Windows 8 fail to run MMBasic for Windows due to audio driver issues – see details of starting without audio below.

This manual covers the essentials of programming for the Windows 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

The graphics implemented in MMBasic are based on those in the Colour Maximite 2. For further understanding, see the Graphics Programming On The CMM2 which is included with the CMM2 documents package and can be downloaded from: http://geoffg.net/Micromite#Downloads

The main features of MMBasic are:

- **Full featured BASIC interpreter** with double precision floating point, 64-bit integers and string variables, long variable names, arrays of floats, integers or strings with multiple dimensions, extensive string handling and user defined subroutines and functions. Typically it will execute a program at over 1,0300,000 lines per second (this is processor dependant and can be much faster)..
- **Full Colour output** to a dedicated window under the Microsoft Windows OS with 20 program selectable video resolutions from 1920x1080 pixels to 240x216 pixels and 32-bit colour (ARGB8888 -16 million colours with 256 levels of transparency).
- **Stereo audio output** can play WAV, FLAC and MP3 files, computer generated music (MOD format) and robot speech and sound effects as well as generate precise sine wave tones.
- **A full screen editor** is built into the software. It includes advanced features such as colour coded syntax, search and copy, cut and paste to and from a clipboard with full mouse support. With one key press the program can be saved and run. If an error occurs another key press will return to the editor with the cursor placed on the line that caused the error.
- **Full support for file storage inbuilt** Files can be edited and run from any storage devices supported by the underlying Windows or Linux operating systems.
- **Extensive features for creating computer games.** These include multiple video planes, support for Blits and Sprites, the ability to create computer generated music, sound effects.

Limitations

If you are familiar with the Micromite/Maximite etc., please note that this version does not attempt to emulate the full hardware I/O environment. Due to limitations of the operating systems (MS Windows, or Linux) there is no support for GPIOs, I2C or SPI interfacing, however, serial I/O is fully implimented.

Quick Start Tutorial

Installing and Running MMBasic for Windows

The Windows version of MMBasic does not need installation. All you need do is copy the executable file (MMBasic.exe) to the directory of your choice. An example would be:

- create a directory C: \MMB4W
- setup MS Windows Path Environment Variable
- Open the Settings using any of the following ways. ...
- Under System, click on "About".
- Click on "Advanced System Settings".
- Click "Environment Variables...".
- The environment variables panel shows up on the screen.
- Click the New button to add new paths or edit to modify the existing path.
- and copy the executable MMBasic.exe into that directory
- this provides MS Windows with a search path to the MMBasic executable. The executable is fully self contained; there are no libraries or other files required. In order to easily access MMBasic, you can create a desktop shortcut that points to the executable MMBasic.exe in the directory set up above.

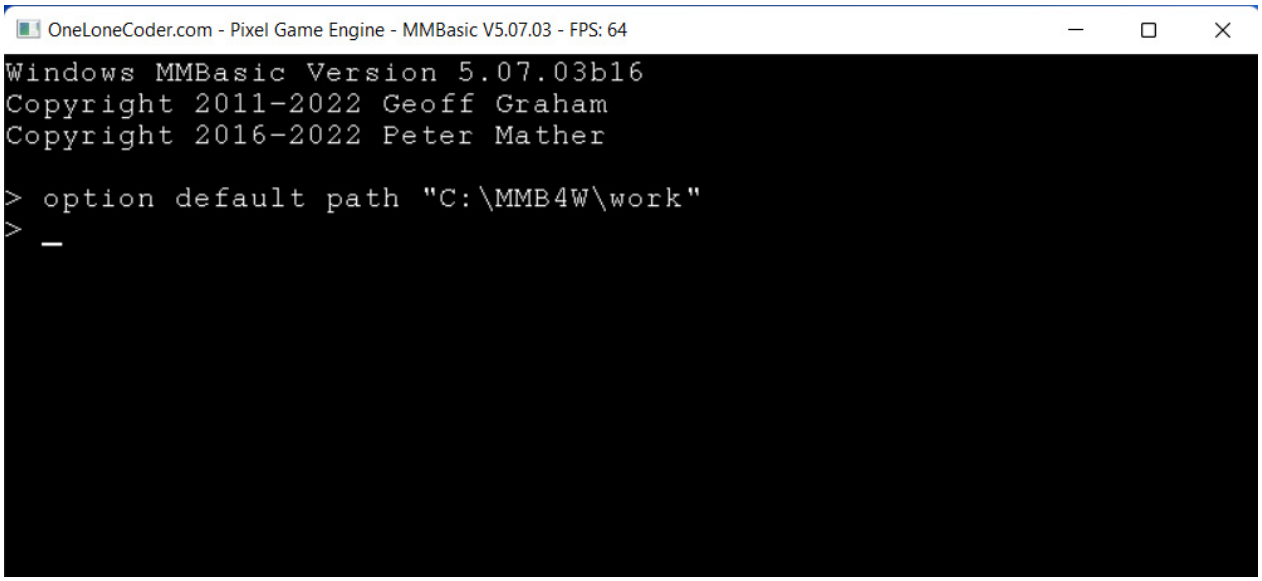
In addition, it is good practice to create a working directory for your programs and that you set the default path within MMBasic to point to this directory. An example would be C:\MMB4W\work.

Note: All code examples below assume you have MMBasic.exe into C:\MMB4W and that you have created a working directory called C:\MMB4W \work in which you have read and write rights.

***** An important note below *****

A note on terminology: MMBasic for Windows is started by first opening a command line DOS box style window, hereafter referred to as the *'console'* and typing MMBasic then press ENTER.. This then starts the actual MMBasic for Windows interpreter in it's own window and is referred to as the *'MMB4W window'* (see the image below). The *'console'* window must stay open (although it can be 'minimised') and can be used to log error messages. It can not, however be used for graphics output.

Alternatively, using Windows Explorer you can just double click on the executable file (MMBasic.exe) or double click on the desktop shortcut created above. MMBasic will open the *console* window which then in turn, opens another window for MMBasic as shown below. You should now set up the default path that MMBasic itself uses. This done by the following from the MMBasic prompt.



Note: MMBasic uses a command line style *console* window to start up the actual BASIC interpreter. You should NOT close this console window as this will also close the MMBasic window. You can however, shrink it down to the status bar.

Developing Programs

To prepare a BASIC program you can create/edit a program from within MMBasic. Refer to the section on the Full Screen Editor. The Editor 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, the command EDIT with no argument at the command prompt will restart the editor with the cursor positioned at the error, so the edit/run/test cycle is very fast.

Alternatively, you can use an external program such as MMEdit created by Jim Hiley (tassyjim on TheBackShed Forum) – details can be found here.

***** 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. *****

From a command line prompt in Windows, you can start MMBasic running a BASIC program in a number of ways as follows:-

- MMBASIC 'opens MMBasic
- MMBASIC "program.bas" ' opens MMBasic and immediately runs program.bas
- MMBASIC "program" ' opens MMBasic and immediately runs program.bas
- MMBASIC "program.bas",text ' opens MMBasic and immediately runs program.bas, passes text to the program as MM.CMDLINE$
- MMBASIC "program",text ' opens MMBasic and immediately runs program.bas, passes text to the program as MM.CMDLINE$
- MMBASIC 0 'opens MMBasic with audio disabled
- MMBASIC 0 "program.bas" ' opens MMBasic and immediately runs program.bas with audio disabled
- MMBASIC 0 "program" ' opens MMBasic and immediately runs program.bas with audio disabled
- MMBASIC 0 "program.bas",text ' opens MMBasic and immediately runs program.bas, passes text to the program as MM.CMDLINE$ with audio disabled
- MMBASIC 0 "program",text ' opens MMBasic and immediately runs program.bas, passes text to the program as MM.CMDLINE$ with audio disabled.

Note the comma after the filename if passing data to the basic program. This is essential if the data is to be passed. To pass data with spaces in it, enclose it in quotes.
Note the space after the 0. This is essential if you want to run a program immediately
If audio is disabled, the PLAY command and GUI BEEP will return immediately without doing anything

When using the mouse in MS Windows, you can start MMBasic as follows:-

- Drag and drop the BASIC program file onto the MMBasic shortcut icon in MS Windows – this will cause MS Windows to start up MMBasic and 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.

To exit MMBasic back to MS Windows, enter QUIT at the MMBasic prompt or press SHIFT-CTRL X

Differences to the Micromite version of MMBasic

Of all the different versions of MMBasic (for all the different hardware platforms), MMBasic for Windows is most similar to the Colour Maximite Computer 2. The main difference between the Windows version of MMBasic and the version running on the Micromite/Maximite family is that the Windows version does not support any hardware related features of the 'mite family.

This means that the following facilities are not supported:

- LCD display panels and associated features (touch, fonts, etc);
- any GPIO related functionality;
- the communications protocols I2C, SPI and 1-wire. **However, serial is implemented.**

MMBasic for Windows has a number of extra commands and functions:

- SYSTEM which will issue a system command to the operating system;
- the read only variable MM.CMDLINE$ will return the MS Windows command line used to start MMBasic;
- the read only variables MM.VRES and MM.HRES will return the size of the window (in pixels);
- COLOUR which will set the foreground and background colours for subsequent character output;
- extended MM.INFO() information.

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).

Graphics Modes and the Page Command

For a more detailed explanation of how the graphic modes and pages interact, see *Graphics Programming On The CMM2* which is included with the CMM2 documents package and can be downloaded from: http://geoffg.net/Micromite#Downloads

MMBasic for Windows starts with the default MODE set to 9 (1024 x 768 pixels) and the default font set to 3 with scale of 1. All graphics modes operate as ARGB8888 for 16 millions colours and 255 levels of transparency. Note that, in the CMM2 version of MMBasic, -1 = &HFFFFFFFF which was a coded value for the text command whereas in ARGB8888 &HFFFFFFFF is non-transparent white. This meant all colour values in the entire code needed changing to 64 bit integers from 32 and there are hundreds of them. The CMM2 didn't have this issue as there are only 15 transparency levels so &HFFFFFFF (7 F's) isn't the same as &HFFFFFFFF (8 F's)

Use OPTION DEFAULT MODE to select the screen format on running. This automatically chooses a sensible font for the screen size. You can't select the small screen sizes which make editing silly.

Use OPTION DEFAULT FONT number, scale to select the font on running. This overrides the automatic font selection caused by DEFAULT MODE

Valid Default modes are:

- Mode 1 800x600
- Mode 8 640x480
- Mode 9 1024x768
- Mode 10 848x480
- Mode 11 1280x720
- Mode 12 960x540
- Mode 15 1280x1024
- Mode 16 1920x1080
- Mode 18 1024x600

The mode is set by the command:-

```
MODE modeno [,alphaenabled] [,background colour]
```

By default alphaenabled is 0 and the background colour is BLACK To use the second layer set alphaenabled to 1 and optionally choose a background colour.

If you use use a negative value for the mode, you will get a *fullscreen display* with no border e.g. MODE -16 will fill a 1080p monitor perfectly.

When you enable transparency with alphaenable, by default, anything written to PAGE 0 will use the default foreground colour (WHITE) and background colour (BLACK). Either of these can be changed to include the desired level of transparency by including a value for A.

If you want to see the background colour, pixels on PAGE 0 must be written with a transparency of less than 255. Use CLS RGB(BLANK) to clear a page to completely transparent.

If you set to write to page 1 this will overlay page 0 and the background to the extent that the alpha value of the pixels on PAGE 1 are set (between 0=transparent and 255=opaque).

Note when playing with this at the command line you can get very confused so it is easier to understand what is happening with a very simple program.

```
mode 14,1,rgb(magenta) 'set into 2 layer mode with the background set to magenta
font 3
print "Click to get focus"
mouseclick
print "Now we will clear page 0 to blank - click to continue"
mouseclick
cls rgb(blank)
colour rgb(black),rgb(blank)
print "This text is written on page 0 with transparent backgound"
print "click to continue"
mouseclick
print "Now we will write to the top page - click to continue"
mouseclick
page write 1
cls rgb(blank)
sprite loadpng 1,"apple"
sprite write 1,50,10,1
sprite transparency 1,128 'change the transparency of the sprite
sprite write 1,200,10,1
page write 0
print @(0,mm.info(fontheight)*4)"You can see the page 0 text through the right
hand apple"
print
print "Click to exit and remember to click again to get focus"
mouseclick
sprite close all
mode 14
colour rgb(white),rgb(black)
end

sub mouseclick
  do
  loop until mouse(l)
  do
  loop until not mouse(l)
end sub
```

All colours are now ARGB8888. If you use the direct hex code then you must set the A.
The RGB function can take 1, 3, or 4 parameters. In the case of 3 parameters it assumes A=255 and of course defined colours like "RED" have A set to 255 which is totally opaque.

- RGB(COLOURNAME) will return a fully opaque representation of that colour.
- RGB(red, green, blue) will return a fully opaque representation of that colour.
- RGB(red, green, blue, trans) to set a partially transparent colour with values for red, green, blue and transparent being between 0 and 255.

If you must, you can use numerical values directly as colours e.g CLS &H80607080. This will set the transparency to &H80, the blue level to &H60, the green level to &H70 and the red level to &H80. Note that red and blue are reversed compared to the CMM2. If a direct numerical value is to be used in this way, it changes order to ABRG8888.

Keyboard Layout

Default keyboard layout is UK. Other versions supported include US, FR, GR, BE, IT, DE, ES and SW.

MMBasic for Windows uses the system time and date thus they can not be set from within MMBasic. They can be accessed via the functions TIME$ and DATE$

Commands and Functions

A list of all the command and functions can be display by the commands:-

```
LIST COMMANDS
LIST FUNCTIONS
```

Timer and Settick

TIMER function now accurate to 0.1uS and reported as a floating point number of milliseconds. SETTICK and PAUSE remain and have an accuracy of better than +/- 2mSec. If you need accurate timing you can have a tight loop in MMBasic with the TIMER function. The following demonstrates timing accuracy:-

```
option milliseconds on
settick 1000,myint
timer=0
do
  pause 5000
  print time$
loop

sub myint
  static int lasttime=timer
  print timer-lasttime
 l asttime=timer
end sub
```

Command Prompt

The following assumes that you have installed the MMBasic.exe program and tested that it runs. At that point you should have the command prompt (a greater than symbol ">") displayed on the screen.

Most interaction with MMBasic is done via the console at the command prompt . On startup,MMBasic will issue the command prompt and wait for some command to be entered. It will also return to the command prompt if your program ends or if it generated an error message.

When the command prompt is displayed, you have a range of commands that you can execute. Typically these would be to list a program (LIST) or edit it (EDIT) or set some options (the OPTION command). Most times the command is just RUN which instructs MMBasic to run the program currently in memory.

When entering text at the command prompt the text can be edited using the arrow keys to move along the line, the Delete key to delete a character and the Insert key to switch between insert and overwrite. The up and down arrow keys will move through a list of previously entered commands which you can edit and reuse.

Finally the “Enter” key will cause MMBasic to execute whatever is showing at the command prompt.

Almost any command can be entered at the command prompt and this is often used to test a command to see how it works. A simple example is the `PRINT` command, which you can test by entering the following at the command prompt:

```
PRINT 2 + 2 then ENTER
```

and not surprisingly, MMBasic will print out the number 4 before returning to the command prompt.

Here are a few more things that you can try out. What you type is shown in bold and MMBasic for Windows' output is shown in normal text.

Try a simple calculation:

```
> PRINT 1/7
0.1428571429
```

See how much memory you have:

```
> MEMORY
Program:
   0K ( 0%) Program (0 lines)
1024K (100%) Free

RAM:
   0K ( 0%) 0 Variables
   0K ( 0%) General
131200K (100%) Free
```

What is the current time?

```
> PRINT TIME$
10:04:01
```

What is the current date?

```
> PRINT DATE$
25/04/2020
```

Count to 20:

```
> FOR a = 1 to 20 : PRINT a; : NEXT a
 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

Note the colon character separating commands.

Your First Program

To enter a program you can use the EDIT command which is described later in this manual. To get a quick feel for how it works, try this sequence:

- At the command prompt type `EDIT "hello.bas"` followed by the ENTER key.
- The editor should start up and you can enter this line: `PRINT "Hello World"`
- Press the F1 key in your keyboard. This tells the editor to save your program and exit to the command prompt.
- At the command prompt type `RUN` followed by the ENTER key.
- You should see the message: `Hello World`

Congratulations. You have just written and run your first program on MMBasic for Windows. If you type EDIT again you will be back in the editor where you can change or add to your program.

Something More Complicated

A more interesting program would be to fill the screen with coloured bubbles.

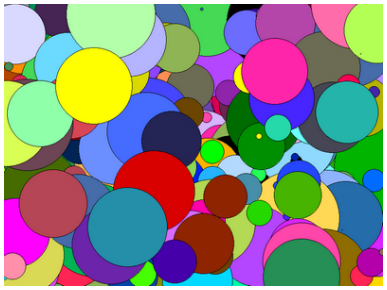
For this you need to know a little more about the editor. If you have used any full screen text editor in the past you will find the operation of this editor familiar. The arrow keys will move your cursor around in the text while the home and end keys will take you to the beginning or end of the line. The delete key will delete the character at the cursor and backspace will delete the character before the cursor.

To enter the bubbles program you should use the command `EDIT "bubbles.bas"` at the command prompt.

Then type in this short program:

```
DO
  r = RND * 255
  g = RND * 255
  b = RND *255
  CIRCLE RND * 800, RND * 600, RND * 100,,, 0, RGB(r,g,b)
  PAUSE 10
LOOP
```

Press the F2 key which will save your program and automatically run it. You should see the screen continuously fill with hundreds of coloured bubbles as shown on the right.



If there was an error you will get a message with the line number and a description of the error. If you then re-enter the command EDIT you will be taken back into the editor with the cursor positioned on the line that caused the error. Correct the error and then save/run the program by `pressing` F2 again.

In this program we first set three variables (r, g and b) to random numbers in the range of zero to 255. The random number generator is called RND and it returns a random number in the range of zero to 0.999999. We multiply it by 255 to give us a random number from 0 to 255.

Then we draw a circle at a random position (again using the random number generator) with a random radius using the three colours previously calculated (ie, r, g and b). This code is contained within a DO…LOOP which instructs MMBasic to keep repeating this code (and drawing bubbles) forever.

You will notice that while this program is running you will not get the command prompt back. This is because MMBasic is now busy executing your program and drawing coloured bubbles. You can stop the program whenever you want by entering CTRL-C at the console and you should get the command prompt back again.

The purpose of the PAUSE 5 command in the program is to slow down the program so that you have time to see the bubbles. To see how fast MMBasic for Windows can really go, you could go back into the editor and change that line to PAUSE 0 and then rerun the program.

For a more in-depth tutorial and a description of programming in BASIC you should download and read *"Programming with the Colour Maximite 2*" which can be found at http://geoffg.net/maximite.html (scroll to the bottom of the page).

Using MMBasic

Commands and Program Input

At the command prompt you can enter a command and it will be immediately run. Most of the time you will do this to run a program or set an option. But this feature also allows you to test out commands at the command prompt.

To enter a program the easiest method is to use the EDIT command. This will invoke the full screen editor which is built into MMBasic for Windows and is described later in this manual. It includes advanced features such as search and copy, cut and paste to and from a clipboard.

Another convenient method of writing and debugging a program is to use MMEdit. This is a program running on your Windows computer which allows you to edit your program on your computer then transfer it via the serial console with a single click of the mouse. MMEdit was written by Jim Hiley and can be downloaded for free from https://www.c-com.com.au/MMedit.htm

One thing that you cannot do is use the old BASIC method of entering a program which was to prefix each line with a line number. Line numbers are optional in MMBasic so you can still use them if you wish but if you enter a line with a line number at the prompt MMBasic will simply execute it immediately.

Editing the Command Line

When entering a line at the command prompt the line can be edited using the arrow keys to move along the line, the Delete key to delete a character and the Insert key to switch between insert and overwrite. The up and down arrow keys will move through a history of previously entered commands which can be edited and reused.

Console Keyboard/Display

The text output from MMBasic can be simultaneously sent to the MMBasic window and the command prompt console window however the graphics commands operate on the main window only.

This behaviour can be changed with the OPTION CONSOLE command. Using this it is possible to turn either console off or on and save the setting so that it will be automatically applied on reboot.

Keyboard Shortcuts

The function keys on the keyboard can be used at the command prompt to automatically enter common commands. The first four function keys (F1 to F4) run the following commands:

- F1 FILES
- F2 RUN
- F3 LIST
- F4 EDIT

Function keys F5 to F12 can be programmed with custom text. See the OPTION FNKey command.

To set the keyboard repeat use

```
OPTION KEYBOARD lang, repeatstart, repeatrate
```

Where *repeatstart* is the delay in milliseconds to commence repeating and *repeatrate* is the delay between repeats.

Line Numbers and Program Structure

The structure of a program line is:

```
[line-number]  [label:]  command arguments [: command arguments] …
```

A label or line number can be used to mark a line of code.

A label has the same specifications (length, character set, etc) as a variable name but it cannot be the same as a command name. When used to label a line, the label must appear at the beginning of a line but after a line number (if used) and be terminated with a colon character (:).

Commands such as GOTO can use labels or line numbers to identify the destination (in that case the label does not need to be followed by the colon character). For example:

```
GOTO xxxx
  - - -
xxxx:  PRINT "We have jumped to here"
```

Multiple commands separated by a colon can be entered on the one line (as in INPUT A : PRINT B).

All Programs Are Run From The Inbuilt File System

In MMBasic for Windows, all programs reside on the underlying file storage system which acts as the "disk drive" for the computer. This could be hard disks, SD cards, USB Memory sticks etc.

When you edit a program you are editing the program on the built in MS Windows file system (hard disk, SD card, Memory stick etc.); when you run a program you will run it from the MS Windows file system (hard disk, SD card, Memory stick etc.). The reason for this arrangement is that when a program is loaded into memory for execution, MMBasic will do a lot of pre-processing to speed up execution. This includes inserting any include files specified in the source, stripping out all comments, removing unnecessary spaces and so on. The resultant program is then saved in program memory but it cannot be edited or listed because after pre-processing the executable program is not easily human readable.

The benefit of this arrangement is a marked improvement in the speed of execution and that much larger programs can fit into program memory and be run.

The main commands used to manage a program are:

- RUN "*prog*" Run the program called *prog* located on the hard disk, SD card, USB stick etc.
- LIST "*prog*" List the program called *prog* on the MMB4W window. This will pause every screen full and any key press will continue the listing.
- EDIT "*prog*" Edit the program called *prog* located on the hard disk, SD card, USB stick etc.

For example: `RUN "hello.bas"`

Note that the file name must be surrounded by double quotes as shown above. This is because the file name is a string and in MMBasic all string constants (ie, not a variable) must be quoted. In all cases the file extension ".BAS" will be automatically added if an extension was not specified in the command line.

When RUN or EDIT are used they set what is known as the *current program name*. This is the file name that will be used if the commands RUN, EDIT and LIST are used without specifying a file name. For example, you could use the command EDIT "MyProg.bas" and that will set the current program name to "MyProg.bas". From then on you could use RUN, EDIT and LIST without a file name and they will refer to "MyProg.bas" on the file system.

To clear the current program name and erase the processed program held in program memory you can use the command NEW. This also clears all variables, closes all files, etc (ie. resets MMBasic).

There are three other commands that operate on program files. These are AUTOSAVE and LIST ALL. These are used for sending/receiving programs via the serial console to or from the file system.

```
AUTOSAVE fname$
```

The filename is now mandatory as on CMM2. You can now paste into AUTOSAVE using CTRL-V. When you exit AUTOSAVE with F1 or CTRL-Z. the file is saved and the program is loaded ready to run. The "last file edited" is updated so you can edit the file without specifying the filename. In addition if you exit with F2 the file is immediately run; if you exit with CTRL-C no file will be created.

Finally, all commands referred to above (with the exception of RUN) can be used with a different file extension to operate on files that are not programs. For example, EDIT "data.txt" will edit the text file in the current working directory called "data.txt". In this case the current program name will not be changed.

Running Programs

A program is set running by the RUN command. You can interrupt MMBasic and the running program at any time by typing CTRL-C on the console input and MMBasic will return to the command prompt.

The running program is run from disk. This means that it will not be lost if MMBasic for Windows is closed or the computer powered off. With the AUTORUN feature turned on, the program will automatically run when is MMBasic for Windows is restarted (use the OPTION command to turn AUTORUN on). Normally the working directory holding the original program must be present in order to run a program but this is one of the exceptions and allows you to change the working directory for recording data, etc.

Expressions

In most cases where a number or string is required you can also use an expression. For example:

```
FNAME$ = "TEST"
LIST FNAME$ + ".BAS"
```

The RUN command is the only exception, in this case the filename argument must be a string constant surrounded by double quotes (ie, not an expression).

Standards and Compatibility

MMBasic implements a large subset of Microsoft's GW-BASIC. There are numerous small differences due to physical and practical considerations but most ordinary 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 SELECT CASE, SUB/END SUB, the DO WHILE … LOOP and structured IF .. THEN … ELSE … ENDIF statements.

The SELECT CASE commands allow the programmer to create a clear and structured decision tree that is more flexible and easier to understand when multiple decisions must be made. The DO WHILE … LOOP command make it easy to build loops without using the GOTO statement. User defined subroutines and functions make it easy to add your own commands to MMBasic.

The IF… THEN command can span many lines with ELSEIF … THEN, ELSE and ENDIF statements as required and also spaced over many lines.

Timing

You can get the current date and time using the DATE$ and TIME$ functions however as they are an intrinsic part of the underlying OS, they can NOT be set from MMBasic.

You can freeze program execution for a number of milliseconds using PAUSE. MMBasic also maintains an internal stopwatch function (the TIMER function) which counts up in microseconds. You can reset this timer to zero or any other number by assigning a value to the TIMER.

Using SETTICK you can setup up to four "ticks" which will generate regular interrupts with a period from one millisecond to over a month.

Watchdog Timer

It is possible to have MMBasic for Windows set up so that the main MMBasic window and the console window are minimised . With OPTION AUTORUN ON MMBasic for Windows will run the program in memory automatically on startup or the RESTART command.

However there is the possibility that a fault in the program could cause MMBasic to generate an error and return to the command prompt. This would be of little use in this situation as there would be nothing connected to the console. Another possibility is that the BASIC program could get itself stuck in an endless loop for some reason. In both cases the visible effect would be the same, the program would stop running until the power was cycled.

To guard against this the watchdog timer can be used. This is a timer that counts down to zero and when it reaches zero the processor will be automatically restarted (the same as when power was first applied), this will occur even if MMBasic is sitting at the command prompt. Following the restart the automatic variable MM.WATCHDOG will be set to true to indicate that the restart was caused by a watchdog timeout.

The WATCHDOG command should be placed in strategic locations in the program to keep resetting the timer and therefore preventing it from counting down to zero. Then, if a fault occurs, the timer will not be reset, it will count down to zero and the program will be restarted (assuming the AUTORUN option is set).

The Console Window

MMBasic for Windows is started via a terminal window called the *'Command Line Console window'*. Once started, MMBasic then opens its own *'MMBasic Console window'* (graphics window) to which all output is now directed. This is also referred to as the *MMBasic window*, the *MMBasic Console* or just *console.*

Configuring the Console window

```
OPTION CONSOLE [SCREEN | SERIAL | BOTH]
```

There are two possible destinations for MMBasic output. The main MMBasic window which acts as the console and can display text and graphics and, the *Command Line Console* (the window you get when starting MMBasic initially).

OPTION CONSOLE SCREEN is the default value and will direct all output (both GUI and text) to the open MMBasic window whether in command line mode or running a program.

OPTION CONSOLE SERIAL will disable text output to the main MMBasic window. All text will be redirected to the *Command Line Console* window. This is useful for debugging graphics applications as diagnostic PRINT statements will not corrupt the screen display. This can be only enabled in a program and reverts to the previous value when the program ends.

Note: This has NOTHING to do with any serial ports-you cannot connect a VT100 terminal via a serial port as for other MMBasic implementations.

OPTION CONSOLE BOTH will enable both the Command Line Console window and the main MMBasic window for output of text with graphics only going to the MMBasic window.

A big advantage of enabling the Command Line Console window is that it is 'scrollable' thus enabling viewing of past commands/error messages.

*** Generally, within most MMBasic documentation *'console'* refers to the primary output window where all print, graphics, errors etc. are displayed. In MMB4W this is called the *'MMBasic console'* with the title 'OneLoneCoder Pixel Game Engine' as distinct from the *'Command Line Console window'* with the title 'MMBasic for Windows' that is used to start MMBasic. Yes, confusing, but that's the way it is! ***

Full Screen Editor

An important productivity feature is the built-in full screen editor. This will only work with the MMBasic windows screen. To run the editor you use the command EDIT at the command prompt. For example:

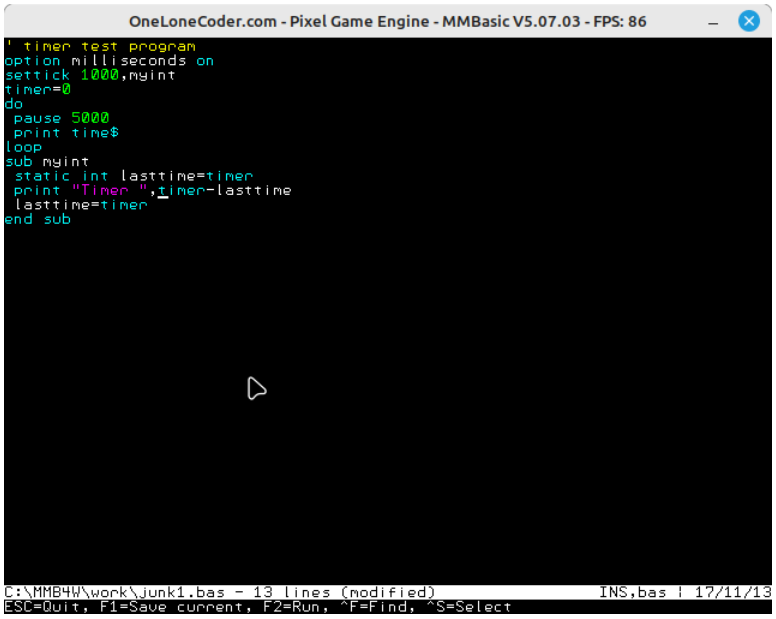
```
EDIT "filename"
```

Note that double quote marks must be used around the file name. If the file's extension is not specified MMBasic will automatically add the extension ".BAS". If the file does not exist it will be created when you save and exit the editor. Non program files can be edited by specifying an extension other than ".BAS".

You can also use EDIT without a file name and in that case the last program that was edited or RUN will be edited. After editing the file it can be run using the RUN command without specifying a file name or you could use the F2 function key within the editor to save and run the program.

On the screen the editor looks like this:

When the editor starts up the cursor will be automatically positioned at the last place that you were editing or, if your program had just been stopped by an error, the cursor will be positioned at the line that caused the error. At the bottom of the screen the status line lists details such as the current cursor position and the common functions supported by the editor.



If you have used an editor like Windows Notepad previously you will find that the operation of this editor is familiar. The arrow keys will move the 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. If a mouse is fitted it can also be used to position the cursor, select text, etc.

The full selection of commands is:

ESC	This will cause the editor to abandon all changes and return to the command prompt with the file unchanged. If you have changed the text you will be asked to press ESC twice more to confirm this action.
F1 : SAVE	This will save the file and return to the command prompt.
F2 : RUN	This will save the file and immediately run it.
F3 or CTRL-F	This will enter the **find mode**. You will be prompted for the search text and as you type this in the editor will automatically position the cursor at the first text found. You can then use the down arrow key to search for the next occurrence or the up arrow key for the previous occurrence. The Enter key will leave the cursor at this position and return to normal editing mode. F5 or CTRL-V will replace the searched text with whatever is in the clipboard (see below). Escape will abort the search.
F4 or CTRL-S	This will enter the **select mode**. In this mode you can use the arrow keys, HOME or END to select text and copy it to the clipboard. It will be highlighted on the screen as you select it. Then F5 or CTRL-C will copy the selection to the clipboard, F4 or CTRL-X will copy and delete the selection. DELETE will simply delete the selection and ESCAPE will return to the normal editing mode without changing anything. Note: you can also enter selection mode when using a keyboard by holding the shift key and pressing right-arrow or down-arrow.
F5 or CTRL-V	This will insert (at the current cursor position) the text that had been previously cut or copied in the select mode (see above). You can also paste from thr clipboard but only a single line.

F6	This will save the edited text and exit the editor similar to the F1 key. The difference is that F6 will not update the "current program name" which is used when the RUN, LIST and EDIT commands are entered without specifying a filename.
END or CTRL-K	Move the cursor to the end of the line.
CTRL-W	Will allow you to save a backup copy of the edited file to a different file. The editor will continue to edit the original file.
F7	Will prompt for a file name and will insert the text from that file into the editor at the current cursor position.
F8 or CTRL-B	Will prompt for a file name and will write to the file the contents of the text that had been previously cut or copied in the select mode (see above). This together with F7 is an easy mechanism for moving blocks of text beween files.
F11	Will paste at the current cursor position the top command last viewed in the help dialogue. It is inactive until the help facility has been used.
F12	Enters the help dialogue and automatically sets any text under the cursor as the match string for help. Use F12 or ESC to exit the help dialogue. On exiting help the top line of the help dialogue will be available to paste into the current program using F11. See the HELP command for more details.
TAB	Will move the cursor to the next tab position as defined by OPTION TAB.
SHIFT TAB	Deletes a number of spaces (defined by OPTION TAB) if the cursor is on a space character. Useful for changing the indenting of a line. USB keyboard only.
SHIFT DELETE	if used at the beginning of a line deletes all leading spaces. Anywhere else in the line and it acts like DELETE.
HOME	Double press to move to start of the file.
END	Double press to move to end of the file.

For security all save commands will create a backup file by appending ".bak" to the filename and renaming the original file before saving the file. This ensures that in the event of any sort of error writing to storage, the worst case is that only the edited version is lost.

The best way to learn how to use the 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 enter your program then, by pressing the F2 key, you can save and run the program. If your program stops with an error pressing the function key F4 at the command prompt 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.

Mouse Support

The mouse works as you would expect, you can:

- Left click to position the edit cursor.
- Scroll wheel to move up and down.
- Left click and hold down then move cursor to a new position and release the left button. The enclosed area will be selected for cut and paste.
- Within cut and paste mode you can use the cursor and wheel to change the selection.
- To exit cut and paste without doing anything right click.
- Right click in top area of screen to scroll up (except in select mode).
- Right click in bottom of screen to scroll down (except in select mode).
- Left click at far left to scroll horizontally one character if the screen is scrolled horizontally.
- Left click at far right to scroll horizontally one character if the line is too long for the display.
- Double left click at the far left to scroll to the beginning of line if the screen is scrolled horizontally.
- Double left click at the far right to scroll horizontally to the end of line if the line is too long to display.

Colour Coded Editor Display

The editor will automatically colour code the edited program with keywords, numbers and comments displayed in different colours. If necessary this feature can be disabled with the command:

```
OPTION COLOURCODE OFF or OPTION COLOURCODE REVERSE
```

and re enabled with:

```
OPTION COLOURCODE ON
```

This setting is saved in an options file, normally in the MyDocuments directory, and automatically applied on startup.

Variables and Expressions

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

Variables

Variables 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 31 characters long.

A variable name or a label must not be the same as a function or one of the following keywords: THEN, ELSE, GOTO, GOSUB, TO, STEP, FOR, WHILE, UNTIL, LOAD, MOD, NOT, AND, OR, XOR, AS.
Eg, step = 5 is illegal as STEP is a keyword.

MMBasic supports three different types of variables:

1. Double Precision Floating Point.
   These can store a number with a decimal point and fraction (eg, 45.386) however they will lose accuracy when more than 14 digits of precision are used. Floating point variables are specified by adding the suffix '!' to a variable's name (eg, i!, nbr!, etc). They are also the default when a variable is created without a suffix (eg, i, nbr, etc).
2. 64-bit Signed Integer.
   These can store positive or negative numbers with up to 19 decimal digits without losing accuracy but they cannot store fractions (ie, the part following the decimal point). These are specified by adding the suffix '%' to a variable's name. For example, i%, nbr%, etc.
3. A String.
   A string will store a sequence of characters (eg, "Tom"). Each character in the string is stored as an eight bit number and can therefore have a decimal value of 0 to 255. String variable names are terminated with a '$' symbol (eg, name$, s$, etc). Strings can be up to 255 characters long. If OPTION ESCAPE ON is set, a string can include escape sequences that are converted on-the-fly. Valid escape sequences are:-

Escape sequence	hex value	ASCII character/code represented
\a	07	Alert (Beep, Bell)
\b	08	Backspace
\e	1B	Escape character
\f	0C	Formfeed Page Break
\n	0A	Newline (Line Feed)
\r	0D	Carriage Return
\q	22	Quote symbol
\t	09	Horizontal Tab
\v	0B	Vertical Tab
\\	5C	Backslash
\nnn	any	The byte whose numerical value is given by nnn - interpreted as a decimal number
\&hh	any	The byte whose numerical value is given by hh - interpreted as a hexadecimal number

For example, if you want to print a quoted string, you could do the following:-:

```
PRINT CHR$(&22)+”Hello World”+CHR$(&22)
```

alternatively,

```
PRINT “\qHello World\q”
```

Note that it is illegal to use the same variable name with different types. Eg, using `nbr!` and `nbr%` in the same program would cause an error. This is different from the original Colour Maximite which allowed this.

Most programs use floating point variables for arithmetic as these can deal with the numbers used in typical situations and are more intuitive than integers when dealing with division and fractions. So, if you are not bothered with the details, always use floating point.

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 unsigned 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.

When a constant number is used it will be assumed that it is an integer if a decimal point or exponent is not used. For example, 1234 will be interpreted as an integer while 1234.0 will be interpreted as a floating point number. String constants are surrounded by double quote marks ("). Eg, "Hello World".

Option Default

A variable can be used without a suffix (ie, !, % or $) and in that case MMBasic will use the default type of floating point. For example, the following will create a floating point variable:

```
Nbr = 1234
```

However the default can be changed with the OPTION DEFAULT command. For example, OPTION DEFAULT INTEGER will specify that all variables without a specific type will be integer. So, the following will create an integer variable:

```
OPTION DEFAULT INTEGER
Nbr = 1234
```

The default can be set to FLOAT (which is the default when a program is run), INTEGER, STRING or NONE. In the latter all variables must be specifically typed otherwise an error will occur.

The OPTION DEFAULT command can be placed anywhere in the program and changed at any time but good practice dictates that if it is used it should be placed at the start of the program and left unchanged.

Option Explicit

By default MMBasic will automatically create a variable when it is first referenced. So, Nbr = 1234 will create the variable and set it to the number 1234 at the same time. This is convenient for short and quick programs but it can lead to subtle and difficult to find bugs in large programs. For example, in the third line of this fragment the variable Nbr has been misspelt as Nbrs. As a consequence the variable Nbrs would be created with a value of zero and the value of Total would be wrong.

```
Nbr = 1234
Incr = 2
Total = Nbrs + Incr
```

The OPTION EXPLICIT command tells MMBasic to not automatically create variables. Instead they must be explicitly defined using the DIM, LOCAL or STATIC commands (see below) before they are used. The use of this command is recommended to support good programming practice. If it is used it should be placed at the start of the program before any variables are used.

Dim and Local

The DIM and LOCAL commands can be used to define a variable and set its type and are mandatory when the OPTION EXPLICIT command is used.

The DIM command will create a global variable that can be seen and used throughout the program including inside subroutines and functions. However, if you require the definition to be visible only within a subroutine or function, you should use the LOCAL command at the start of the subroutine or function. LOCAL has exactly the same syntax as DIM.

If LOCAL is used to specify a variable with the same name as a global variable then the global variable will be hidden to the subroutine or function and any references to the variable will only refer to the variable defined by the LOCAL command. Any variable created by LOCAL will vanish when the program leaves the subroutine.

At its simplest level DIM and LOCAL can be used to define one or more variables based on their type suffix or the OPTION DEFAULT in force at the time. For example:

```
DIM nbr%, s$
```

But it can also be used to define one or more variables with a specific type when the type suffix is not used:

```
DIM INTEGER nbr, nbr2, nbr3, etc
```

In this case nbr, nbr2, nbr3, etc are all created as integers. When you use the variable within a program you do not need to specify the type suffix. For example, MyStr in the following works perfectly as a string variable:

```
DIM STRING MyStr
MyStr = "Hello"
```

The DIM and LOCAL commands will also accept the Microsoft practice of specifying the variable's type after the variable with the keyword "AS". For example:

```
DIM nbr AS INTEGER, s AS STRING
```

In this case the type of each variable is set individually (not as a group as when the type is placed before the list of variables).

The variables can also be initialised while being defined. For example:

```
DIM INTEGER a = 5, b = 4, c = 3
DIM s$ = "World", i% = &H8FF8F
DIM msg AS STRING = "Hello" + " " + s$
```

The value used to initialise the variable can be an expression including user defined functions.

The DIM or LOCAL commands are also used to define an array and all the rules listed above apply when defining an array.

For example, you can use:

```
DIM INTEGER nbr(10), nbr2, nbr3(5,8)
```

When initialising an array the values are listed as comma separated values with the whole list surrounded by brackets. For example:

```
DIM INTEGER nbr(5) = (11, 12, 13, 14, 15, 16)
or
DIM days(7) AS STRING = ("","Sun","Mon","Tue","Wed","Thu","Fri","Sat")
```

Static

Inside a subroutine or function it is sometimes useful to create a variable which is only visible within the subroutine or function (like a LOCAL variable) but retains its value between calls to the subroutine or function.

You can do this by using the STATIC command. STATIC can only be used inside a subroutine or function and uses the same syntax as LOCAL and DIM. The difference is that its value will be retained between calls to the subroutine or function (ie, it will not be initialised on the second and subsequent calls).

For example, if you had the following subroutine and repeatedly called it, the first call would print 5, the second 6, the third 7 and so on.

```
SUB Foo
  STATIC var = 5
  PRINT var
  var = var + 1
END SUB
```

Note that the initialisation of the static variable to 5 (as in the above example) will only take effect on the first call to the subroutine. On subsequent calls the initialisation will be ignored as the variable had already been created on the first call.

As with DIM and LOCAL the variables created with STATIC can be float, integers or strings and arrays of these with or without initialisation. The length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 31 characters.

Const

Often it is useful to define an identifier that represents a value without the risk of the value being accidently changed - which can happen if variables were used for this purpose (this practice encourages another class of difficult to find bugs).

Using the CONST command you can create an identifier that acts like a variable but is set to a value that cannot be changed. For example:

```
CONST InputVoltagePin = 26
CONST MaxValue = 2.4
```

The identifiers can then be used in a program where they make more sense to the casual reader than simple numbers. For example:

```
IF PIN(InputVoltagePin) > MaxValue THEN SoundAlarm
```

A number of constants can be created on the one line:

```
CONST InputVoltagePin = 26, MaxValue = 2.4, MinValue = 1.5
```

The value used to initialise the constant is evaluated when the constant is created and can be an expression including user defined functions.

The type of the constant is derived from the value assigned to it; so for example, MaxValue above will be a floating point constant because 2.4 is a floating point number. The type of a constant can also be explicitly set by using a type suffix (ie, !, % or $) but it must agree with its assigned value.

Expressions and Operators

MMBasic will evaluate a mathematical expression using the standard mathematical rules. For example, multiplication and division are performed first followed by addition and subtraction. These are called the rules of precedence and are detailed below.

This means that 2 + 3 * 6 will resolve to 20, so will 5 * 4 and also 10 + 4 * 3 – 2.

If you want to force the interpreter to evaluate parts of the expression first you can surround that part of the expression with brackets. For example, (10 + 4) * (3 – 2) will resolve to 14 not 20 as would have been the case if the brackets were not used. Using brackets does not appreciably slow down the program so you should use them liberally if there is a chance that MMBasic will misinterpret your intension.

The following operators, in order of precedence, are implemented in MMBasic. 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:

^	Exponentiation (eg, `b^n` means b^n)
* / \ 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. For a right shift any bits introduced are set to the value of the top bit (bit 63). For a left shift any bits introduced are set to zero.

Logical operators:

NOT INV	invert the logical value on the right (eg, `NOT a=b` is `a<>b`) or bitwise inversion of the value on the right (eg, `a = INV b`)
<> < > <= = >= =>	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

For Microsoft compatibility the operators AND, OR and XOR are integer bitwise operators. For example, PRINT (3 AND 6) will output the number 2. Because these operators can act as both logical operators (for example, IF a=5 AND b=8 THEN …) and as a bitwise operators (eg, y% = x% AND &B1010) the interpreter will be confused if they are mixed in the same expression. So, always evaluate logical and bitwise expressions in separate expressions.

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 will invert the logical value on its right (it is not a bitwise invert) while the INV operator will perform a bitwise invert. Both of these have the highest precedence so they will bind tightly to the next value. For normal use of NOT or INV the expression to be operated on should be placed in brackets. Eg:

```
IF NOT (A = 3 OR A = 8) THEN
```

Strings

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".

Mixing Floating Point and Integers

MMBasic automatically handles conversion of numbers between floating point and integers. If an operation mixes both floating point and integers (eg, PRINT A% + B!) the integer will be converted to a floating point number first, then the operation performed and a floating point number returned. If both sides of the operator are integers then an integer operation will be performed and an integer returned.

The one exception is the normal division ("/") which will always convert both sides of the expression to a floating point number and then returns a floating point number. For integer division you should use the integer division operator "\".

MMBasic functions will return a float or integer depending on their characteristics. For example, PIN() will return an integer when the pin is configured as a digital input but a float when configured as an analog input.

If necessary you can convert a float to an integer with the INT() function. It is not necessary to specifically convert an integer to a float but if it was needed the integer value could be assigned to a floating point variable and it will be automatically converted in the assignment.

** Real numbers can not always be exactly represented by floating point numbers. A number can have finite accurate decimal representation eg. 0.1 but can NOT be exactly represented in binary.

64-bit Unsigned Integers

MMBasic for Windows supports 64-bit signed integers. This means that there are 63 bits for holding the number and one bit (the most significant bit) which is used to indicate the sign (positive or negative). However it is possible to use full 64-bit unsigned numbers as long as you do not do any arithmetic on the numbers.

64-bit unsigned numbers can be created using the &H, &O or &B prefixes to a number and these numbers can be stored in an integer variable. You then have a limited range of operations that you can perform on these. They are << (shift left), >> (shift right), AND (bitwise and), OR (bitwise or), XOR (bitwise exclusive or), INV (bitwise inversion), = (equal to) and <> (not equal to). Arithmetic operators such as division or addition may be confused by a 64-bit unsigned number and could return nonsense results.

Note that shift right is a signed operation. This means that if the top bit is a one (a negative signed number) and you shift right then it will shift in ones to maintain the sign.

To display 64-bit unsigned numbers you should use the HEX$(), OCT$() or BIN$() functions.

For example, the following 64-bit unsigned operation will return the expected results:

```
X% = &HFFFF0000FFFF0044
Y% = &H800FFFFFFFFFFFF
X% = X% AND Y%
PRINT HEX$(X%, 16)
```

Will display "800F0000FFFF0044"

Subroutines and Functions

A program defined subroutine or function is simply a block of programming code that is contained within a module and can be called from anywhere within your program. It is the same as if you have added your own command or function to the language.

Subroutines

A subroutine acts like a command and it can have arguments (sometimes called a parameter list). In the definition of the subroutine they look like this:

```
SUB MYSUB arg1, arg2$, arg3
  <statements>
END SUB
```

And when you call the subroutine you can assign values to the arguments. For example:

```
MYSUB  23, "Cat", 55
```

Inside the subroutine `arg1` will have the value `23`, `arg2$` the value of `"Cat"`, and so on. The arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be hidden by the arguments defined for the subroutine.

When calling a subroutine you can supply less than the required number of values and in that case the missing values will be assumed to be either zero or an empty string. You can also leave out a value in the middle of the list and the same will happen. For example:

```
MYSUB  23,, 55
```

Will result in `arg2$` being set to the empty string `""`.

Rather than using the type suffix (eg, the $ in arg2$) you can use the suffix AS <type> in the definition of the subroutine argument and then the argument will be known as the specified type, even when the suffix is not used. For example:

```
SUB MYSUB arg1, arg2 AS STRING, arg3
  IF arg2 = "Cat" THEN …
END SUB
```

Local Variables

Inside a subroutine you can define a variable using LOCAL (which has the same syntax as DIM). This variable will only exist within the subroutine and will vanish when the subroutine exits. You can have a variable in your main program with the same name but it will be hidden and the local variable used while the subroutine is executed.

If you do not declare the variable as LOCAL within the subroutine and OPTION EXPLICIT is not in force it will be created as a global variable and be visible in your main program and subroutines, just like a normal variable declared outside a subroutine or function.

Functions

Functions are similar to subroutines with the main difference being that the function is used to return a value in an expression. The rules for the argument list in a function are similar to subroutines. The only difference is that brackets are required around the argument list when you are calling a function, even if there are no arguments (they are optional when calling a subroutine).

To return a value from the function you assign a value to the function's name within the function. If the function's name is terminated with a $, a % or a ! the function will return that type, otherwise it will return whatever the OPTION DEFAULT is set to. You can also specify the type of the function by adding AS <type> to the end of the function definition.

For example:

```
FUNCTION Fahrenheit(C) AS FLOAT
  Fahrenheit = C * 1.8 + 32
END FUNCTION
```

Passing Arguments by Reference

If you use an ordinary variable (ie. not an expression) as the value when calling a subroutine or a function, the argument within the subroutine/function will point back to the variable used in the call and any changes to the argument will also be made to the supplied variable. This is called passing arguments by reference.

For example, you might define a subroutine to swap two values, as follows:

```
SUB Swap a, b
  LOCAL t
  t = a
  a = b
  b = t
END SUB
```

In your calling program you would use variables for both arguments:

```
Swap nbr1, nbr2
```

And the result will be that the values of `nbr1` and `nbr2` will be swapped.

For this to work the type of the variable passed (eg, `nbr1`) and the defined argument (eg, `a`) must be the same (in the above example both default to float).

Unless you need to return a value via the argument you should not use an argument as a general purpose variable inside a subroutine or function. This is because another user of your routine may unwittingly use a variable in their call and that variable could be "magically" changed by your routine. It is much safer to assign the argument to a local variable or pass an argument by value (see below) and manipulate that instead.

Passing Arguments by Value

Where you need to ensure that the argument being passed is not altered in any way, you can pass a *value* to a subroutine. When the parameter being passed is an expression, the result of that expression is passed as a value. The expression could be the result of simple maths or the return value of a function. It can also be as simple as enclosing a variable in brackets, causing the interpreter to treat it as an expression.

In this case the value could be used or even changed in the sub routine without having any effect on the passed value. The same could be achieved by assigning a *passed by reference* variable and assigning it to a local variable in the subroutine and using/changing the local variable as desired.

The advantage of *passing by value* is that the argument passed in the calling statement is safe from any changes in the called routine and additionally, saves you having to use LOCAL in the sub routine.

```
a=4
b=4
c=4
testsub((a),b,c)
print a,b,c

sub testsub(arg1,arg2,arg3)
  local k
  arg1=arg1+1
  arg2=arg2+1
  k=arg3
  k=k+1
end sub

4  5  4
```

The result for both a and c is no change globally; a being *passed by value* and c using LOCAL

Passing Arrays

Single elements of an array can be passed to a subroutine or function and they will be treated the same as a normal variable. For example, this is a valid way of calling the Swap subroutine (discussed above):

```
Swap dat(i), dat(i + 1)
```

This type of construct is often used in sorting arrays.

You can also pass one or more complete arrays to a subroutine or function by specifying the array with empty brackets instead of the normal dimensions. For example, `a()`. In the subroutine or function definition the associated parameter must also be specified with empty brackets. The type (ie, float, integer or string) of the argument supplied and the parameter in the definition must be the same.

In the subroutine or function the array will inherit the dimensions of the array passed and these must be respected when indexing into the array. If required the dimensions of the array could be passed as additional arguments to the subroutine or function so it could correctly manipulate the array. The array is passed by reference which means that any changes made to the array within the subroutine or function will also apply to the supplied array.

For example, when the following is run the words "Hello World" will be printed out:

```
DIM MyStr$(5, 5)
MyStr$(4, 4) = "Hello" : MyStr$(4, 5) = "World"
Concat MyStr$()
PRINT MyStr$(0, 0)

SUB Concat arg$()
  arg$(0,0) = arg$(4, 4) + " " + arg$(4, 5)
END SUB
```

Early Exit

There can be only one END SUB or END FUNCTION for each definition of a subroutine or function. To exit early from a subroutine (ie, before the END SUB command has been reached) you can use the EXIT SUB command. This has the same effect as if the program reached the END SUB statement. Similarly you can use EXIT FUNCTION to exit early from a function.

Recursion

Recursion is where a subroutine or function calls itself. You can do recursion in MMBasic but there are a number of issues (these are a direct consequence of the limitations of the BASIC language):

- There is a fixed limit to the depth of recursion. In MMBasic for Windows this is 50 levels.
- If you have many arguments to the subroutine or function and many LOCAL variables (especially strings) you could easily run out of memory before reaching the 50 level limit.
- Any FOR…NEXT loops and DO…LOOPs will be corrupted if the subroutine or function is recursively called from within these loops.

There is often the need for a special command or function to be implemented in MMBasic but in many cases these can be constructed using an ordinary subroutine or function which will then act exactly the same as a built in command or function.

For example, sometimes there is a requirement for a TRIM function which will trim specified characters from the start and end of a string. The following provides an example of how to construct such a simple function in MMBasic.

The first argument to the function is the string to be trimmed and the second is a string containing the characters to trim from the first string. RTrim$() will trim the specified characters from the end of the string, LTrim$() from the beginning and Trim$() from both ends.

```
' trim any characters in c$ from the start and end of s$
Function Trim$(s$, c$)
  Trim$ = RTrim$(LTrim$(s$, c$), c$)
End Function

' trim any characters in c$ from the end of s$
Function RTrim$(s$, c$)
  RTrim$ = s$
  Do While Instr(c$, Right$(RTrim$, 1))
    RTrim$ = Mid$(RTrim$, 1, Len(RTrim$) - 1)
  Loop
End Function

' trim any characters in c$ from the start of s$
Function LTrim$(s$, c$)
  LTrim$ = s$
  Do While Instr(c$, Left$(LTrim$, 1))
    LTrim$ = Mid$(LTrim$, 2)
  Loop
End Function
```

As an example of using these functions:

```
S$ = "   ****23.56700  "
PRINT Trim$(s$, " ")
```

Will give "****23.56700"

```
PRINT Trim$(s$, " *0")
```

Will give "23.567"

```
PRINT LTrim$(s$, " *0")
```

Will give "23.56700"

Basic Graphics

The graphics subsystem in MMBasic for Windows is based on that of the Colour Maximite 2 and has many advanced features such as multiple video pages, layered images, sprites and sophisticated methods of manipulating images. These and other features of the graphics subsystem are explained in detail in the document Graphics Programming on the CMM2 that is included in the Colour Maximite documentation available from Geoff Graham's website. A PDF version of this is included in the Colour Maximite 2 firmware zip file.

There are ten basic drawing commands that you can use within MMBasic to draw images on MMBasic's main window (none of these apply to the console window).

Screen Coordinates

All screen coordinates and measurements on the screen are done in terms of pixels with the `X` coordinate being the horizontal position and `Y` the vertical position. The top left corner of the screen has the coordinates `X = 0` and `Y = 0` and the values increase as you move down and to the right of the screen. This can be changed by using the OPTION Y_AXIS DOWN|UP setting

By default on startup the window resolution will be set to `1024 x 768` pixels (`MODE 9`) with each pixel supporting 256 different colours. At this resolution the bottom right pixel will be at `X=1023` and `Y=767`.

Read Only Variables

There are six read only variables which provide useful information about the current display window:

- MM. HRES

Returns the width of the display (the X axis) in pixels.

- MM. VRES

Returns the height of the display (the Y axis) in pixels.

- MM.INFO(FONTHEIGHT)

Returns the height of the current font (in pixels). All characters in a font have the same height.

- MM.INFO(FONTWIDTH)

Returns the width of a character in the current font (in pixels). All characters in a font have the same width.

- MM.INFO(HPOS)

Returns the X coordinate of the text cursor (ie, the horizontal location (in pixels) of where the next character will be printed on the MMBasic window)

- MM.INFO(VPOS)

Returns the Y coordinate of the text cursor (ie, the vertical location (in pixels) of where the next character will be printed on the MMBasic window).

Colours

Colour is specified as a true colour 32 bit number in the form `ARGB8888`. Where the top eight bits represent the level of transparency, the next eight bits, the intensity of the red colour, the next eight bits the green intensity and the bottom eight bits the blue. For example the colour red is &HFFFF0000 and yellow is &HFFFFFF00.

An easier way to generate a colour value is to use the RGB() function which has the form:

```
RGB(red, green, blue)
```

A value of zero for a colour represents black and 255 represents full intensity.

The RGB() function also supports a shortcut where you can specify common colours by naming them. For example, RGB(red) or RGB(cyan). The colours that can be named using the shortcut form are white black, blue, green, cyan, red, magenta, yellow, brown, white, orange, pink, gold, salmon, beige, lightgrey and grey (or USA spelling gray/lightgray).

The following program by Jim Hiley (tassyjim on TheBackShed) demonstrates

```
' test card for MMB4W by TassyJim January 2022
OPTION EXPLICIT:OPTION DEFAULT NONE
DIM INTEGER wd, ht, wbox, sh, x, w, n, nn, m, maxMode, mk, cir
DIM FLOAT a
DIM k$, imgtitle$, fname$, imgRes$
DIM INTEGER c(8)
c(0) = RGB(BLACK):c(1) = RGB(YELLOW):c(2) = RGB(CYAN)
c(3) = RGB(GREEN):c(4) = RGB(MAGENTA):c(5) = RGB(RED)
c(6) = RGB(BLUE):c(7) = RGB(WHITE):c(8) = RGB(64,64,64)
maxMode = 16:a = 1:cir = 1
CLS
DO
  IF m = 0 THEN
    MODE 1 ',8
    CLS
    TEXT 400,100, "Video mode test",cm,5,1
    TEXT 400,180, "Ratio = aspect ratio used in the circle command",cm,1,1
    TEXT 400,220, "Q to quit, P to save page as a BMP",cm,3,1
    TEXT 400,260, "Up Down arrow or mouse wheel to change mode",cm,2,1
    TEXT 400,300, "+ - to change circle aspect ratio",cm,2,1
    TEXT 400,340, "C to toggle circle",cm,2,1
  ELSE
    CLS
    ' erase video memory before mode change
    MODE m
    CLS
    PAUSE 100
    wd = MM.HRES : ht = MM.VRES
    nn = INT(wd/80)
    imgtitle$ =" MODE "+STR$(m)+" Ratio "+STR$(a,1,3)+" "
    imgRes$ = " "+STR$(MM.HRES)+" x "+STR$(MM.VRES)+" "
    'pages$ = " Maximum page number = "+STR$(mp)+" "
    wbox = wd / 8
    FOR x = 0 TO 7
      BOX x*wbox,ht/4,wbox,ht/2,0,c(x), c(x)
    NEXT x
    ' full gradient for each primary colour and greyscale
    FOR x = 0 TO wd-1
      sh = 255*x/wd
      LINE x,0,x,ht/12,1,RGB(sh,0,0)
      LINE x,ht/12,x,ht/6,1,RGB(0,sh,0)
      LINE x,ht/6,x,ht/4,1,RGB(0,0,sh)
      LINE x,ht*9/12,x,ht*10/12,1,RGB(0,sh,sh)
      LINE x,ht*10/12,x,ht*11/12,1,RGB(sh,0,sh)
      LINE x,ht*11/12,x,ht,1,RGB(sh,sh,0)
      LINE x,ht/2,x,ht*3/4,1,RGB(sh,sh,sh) ' greyscale
    NEXT x     ' circle to check aspect ratio
    IF cir THEN CIRCLE wd/2,ht/2, ht*15/32,3,a,c(7)
    sh = 0
    x = wd/2 - 55*nn/2
    ' black white bars to check monitor bandwidth
    FOR w = 10 TO 1 STEP -1
      FOR n = 1 TO nn
        sh = 255 - sh
        LINE x,ht*3/8,x,ht*5/8,w,RGB(sh,sh,sh)
        x = x + w
      NEXT n
    NEXT w
    ' white and red border to check that image fits on monitor
    BOX 0,0,wd,ht,3,c(7)
    BOX 1,1,wd-2,ht-2,1,c(5)
    IF wd > 600 THEN  'title
      TEXT wd/2,ht/2-15, imgtitle$,cm,4,1
      'TEXT wd/2,ht/2, pages$,cm,4,1
      TEXT wd/2,ht/2+15, imgRes$,cm,4,1
    ELSE
```

```
      TEXT wd/2,ht/2-11, imgtitle$,cm,1,1
      'TEXT wd/2,ht/2, pages$,cm,1,1
      TEXT wd/2,ht/2+11, imgRes$,cm,1,1
    ENDIF
    ' show the new image
  ENDIF
  PAUSE 100  ' wait for keypress or mouse wheel
  DO
    k$ = INKEY$
    mk = MOUSE(w)
    IF mk < 0 THEN
      k$ = CHR$(129)
    ELSEIF mk > 0 THEN
      k$ = CHR$(128)
    ENDIF
  LOOP UNTIL k$<>""
  SELECT CASE k$
  CASE "C","c" ' toggle circle
    cir = 1 - cir
  CASE "Q","q"
    EXIT DO
  CASE "P","p"
    fname$ = MID$(imgtitle$,2)+".bmp"
    TIMER = 0
    SAVE IMAGE fname$
    ' PAGE WRITE 0
    TEXT wd/2,ht/2,"Saved as "+fname$+" in "+STR$(TIMER/1000,3,2)+" Sec",cm,1,1
    DO
      k$ = INKEY$
    LOOP UNTIL k$<>""
  CASE CHR$(128) ' up arrow
    m = m - 1
    IF m < 1 THEN m = maxMode
  CASE CHR$(129) ' down arrow
    m = m + 1
    IF m > maxMode THEN m = 1
  CASE "+" ' ratio plus
    IF a < 1.4 THEN a = a + 0.01
  CASE "-" ' ratio minus
    IF a > 0.75 THEN a = a - 0.01
  CASE ELSE ' same as down arrow
    m = m + 1
    IF m > maxMode THEN m = 1
  END SELECT
LOOP
CLS:MODE 1:END
```

In addition there is a special colour NOTBLACK. For any mode this will be the darkest colour that can be displayed that will not act as transparent when manipulated by graphics commands that support transparency.

Because MMBasic for Windows uses double precision floating point it can store the 32 bit number representing colour (i.e. returned by the RGB() function) in either a floating point variable or an integer variable.

The default colour for commands that require a colour parameter can be set with the COLOUR command. This is handy if your program uses a consistent colour scheme, you can then set the defaults and use the short version of the drawing commands throughout your program (the USA spelling COLOR is also accepted).

The `COLOUR` command takes the format:

```
COLOUR foreground-colour, background-colour
```

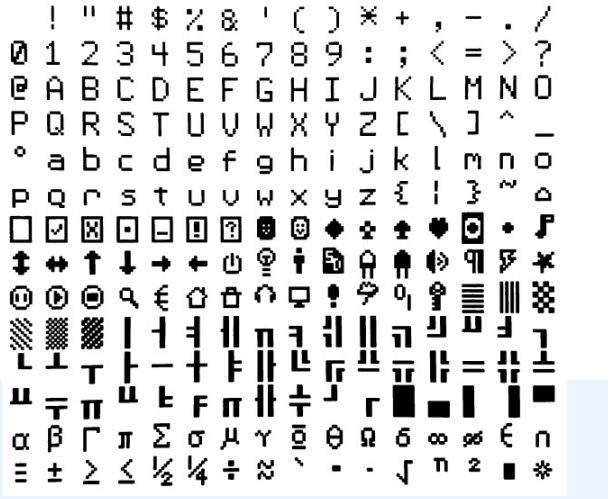
Fonts

There are seven built in fonts. These are:

Font Number	Size (width x height)	Character Set	Description
1	8 x 12	All 95 ASCII characters plus 7F to FF (hex)	Standard font (default on startup). Default font for the editor
2	12 x 20	All 95 ASCII characters	Medium sized font
3	16 x 24	All 95 ASCII characters	A larger font useful for the 800 x 600 display mode
4	10x16	All 95 ASCII characters plus 7F to FF (hex)	A useful font for improved clarity in high resolution modes
5	24 x 32	All 95 ASCII characters	Large font, very clear
6	32 x 50	0 to 9 plus some symbols	Numbers plus decimal point, positive, negative, equals, degree and colon symbols. Very clear.
7	6 x 8	All 95 ASCII characters	A small font useful when low resolutions are used.

In all fonts (including font #6) the back quote character (60 hex or 96 decimal) has been replaced with the degree symbol (º).

Font #1 (the default font) and font #4 have an extended character set covering all characters from CHR$(32) to CHR$(255) or 20 to FF (hex) as illustrated on the right.



If required, additional fonts can be embedded in a BASIC program. These fonts work exactly same as the built in font (ie, selected using the FONT command or specified in the TEXT command).

The format of an embedded font is:

```
DefineFont #Nbr
  hex [[ hex[…]
  hex [[ hex[…]
END DefineFont
```

It must start with the keyword "DefineFont" followed by the font number (which may be preceded by an optional # character). Any font number in the range of 2 to 5 and 8 to 16 can be specified and if it is the same as a built in font it will replace that font. The body of the font is a sequence of 8-digit hex words with each word separated by one or more spaces or a new line. The font definition is terminated by an "End DefineFont " keyword. These can be placed anywhere in a program and MMBasic will skip over it. This format is the same as that used by the Micromite.

Additional fonts and information can be found in the Embedded Fonts folder in the Colour Maximite 2 firmware download. These fonts cover a wide range of character sets including a symbol font (Dingbats) which is handy for creating on screen icons, etc.

In addition to using embedded fonts a program can dynamically load one font from storage using the LOAD FONT command. A program can load many fonts using this method during the course of its execution but each new font will overwrite the previously loaded font.

The format of fonts loaded using LOAD FONT have a similar format as the embedded fonts described above except that no comments or blank lines are allowed, the font number must always be #8, the first word must be on a line on its own and the following lines (except the last) must have exactly eight words per line.

As an example, the following is a tiny (6x4 pixel) font that is useful in the 320x200 display mode. This can be either loaded using LOAD FONT or embedded in the BASIC program:

```
DefineFont #8
60200604
44000000 00A04040 A0AEAE00 82406C6C EACC2048 00004460 84204424 E4A48044
00E404A0 00800400 040000E0 00480240 4CE0AAEA 48C24044 C062C2E0 E820E2AA
EA68E0E2 8048E2E0 EAE0EAEA 0404C0E2 80040400 0E208424 2484000E 4040E280
4A60E84A CACAA0EA 608868C0 E8C0AACA E8E8E0E8 60EA6880 E4A0EAAA 2A22E044
A0CAAA40 AEE08888 EEAEA0EA 40AA4AA0 4A80C8CA ECCA60AE C04268A0 AA4044E4
A4AA60AA A0EEAA40 AAA04AAA 48E24044 E088E8E0 E2004208 004AE022 F0000000
0C000084 AA8CE06A 608806C0 0660AA26 E42460AC 24AE0640 40A0CA88 22204044
A0CC8AA4 0EE044C4 AA0CA0EE 40AA04A0 06C8AA0C 880662AA C0C60680 0A60444E
AE0A60AA E0AE0A40 0AA0440A 6C0E24A6 608464E0 C4400444 006CC024 E0EEEE00
End DefineFont
```

You can convert the original Colour Maximite's font files to this format using the program FontTweak from: https://www.c-com.com.au/MMedit.htm

Drawing Commands

The drawing commands have optional parameters. You can completely leave these off the end of a command or you can use two commas in sequence to indicate a missing parameter. For example, the fifth parameter of the LINE command is optional so you can use this format:

```
LINE 0, 0, 100, 100,, rgb(red)
```

Optional parameters are indicated below by italics, for example: *font*.

In the following commands C is the drawing colour and defaults to the current foreground colour. FILL is the fill colour which defaults to -1 which indicates that no fill is to be used.

The drawing commands are described below; see the COMMANDS section for more detail.:

- CLS *C*

Clears the screen to the colour C. If C is not specified the current default background colour will be used.

- PIXEL X, Y, *C*

Illuminates a pixel. If C is not specified the current default foreground colour will be used.

- LINE X1, Y1, X2, Y2, *LW, C*

Draws a line starting at X1 and Y1 and ending at X2 and Y2.
LW is the line's width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or is changed to 1 if the line is a diagonal.

- LINE AA, X1, Y1, X2, Y2, *LW, C*

Draws a line with anti-aliasing . The parameters are as per the LINE command above. However this version will use variable intensity values of the specified colour to reduce the "staggered" quality of diagonal lines. In addition this version can draw diagonal lines of any width.

- BOX X, Y1, W, H, *LW, C, FILL*

Draws a box starting at X and Y1 which is W pixels wide and H pixels high.
LW is the width of the sides of the box and can be zero. It defaults to 1.

- RBOX X, Y1, W, H, *R, C, FILL*

Draws a box with rounded corners starting at X and Y1 which is W pixels wide and H pixels high.
R is the radius of the corners of the box. It defaults to 10.

- TRIANGLE X1, Y1, X2, Y2, X3, Y3, *C, FILL*

Draws a triangle with the corners at X1, Y1 and X2, Y2 and X3, Y3. C is the colour of the triangle and FILL is the fill colour. FILL can omitted or be -1 for no fill.

- CIRCLE X, Y, R, *LW, A, C, FILL*

Draws a circle with X and Y as the centre and a radius R. LW is the width of the line used for the circumference and can be zero (defaults to 1). A is the aspect ratio which is a floating point number and defaults to 1. For example, an aspect of 0.5 will draw an oval where the width is half the height.

- ARC x, y, r1, r2, a1, a2, c

Draws an arc with the centre at x and y, r1 and r2 are the inner and outer radius defining the thickness of the arc (if they are the same the arc will be one pixel thick), a1 and a2 are the start and end angles in degrees and c is the colour.

- POLYGON n, xarray%(), yarray%(), *C, FILL*

Draws a outline or filled polygon defined by the x, y coordinate pairs in xarray%() and yarray%(). 'n' is the number of points to use in drawing the polygon. If the last xy-coordinate pair is not the same as the first the firmware will automatically create an additional xy-coordinate pair to complete the polygon.

- TEXT X, Y, STRING*, ALIGNMENT, FONT, SCALE, C, BC*

Displays a string starting at X and Y. ALIGNMENT is 0, 1 or 2 characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER or RIGHT aligned text and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE or BOTTOM aligned text. The default alignment is left/top. FONT and SCALE are optional and default to that set by the FONT command. C is the drawing colour and BC is the background colour. They are optional and default to that set by the COLOUR command.

Most graphics commands allow the use of arrays as parameters so that you can draw multiple graphic objects with the one command. In this case the array is passed as the array name followed by empty brackets (eg arr()). Drawing multiple graphic elements this way is *much* faster than drawing them one by one using separate commands.

For example, the PIXEL command allows arrays to be specified for the x and y coordinates (in this case both must be arrays). The firmware will then plot the number of pixels as determined by the dimensions of the smallest array. For the PIXEL command 'c' can also be an array or a single variable/constant.

This is demonstrated with the following example which will draw three pixels in different colours:

```
DIM xx(2) = (10, 20, 30)
DIM yy(2) = (100, 150, 200)
DIM cc(2) = (RGB(red), RGB(green), RGB(blue))
PIXEL xx(), yy(), cc()
```

Example of Basic Graphics

As an example, the following program will draw a simple digital clock on the MMBasic main window.

```
CLS
CONST DBlue = RGB(0, 0, 128)              ' A dark blue colour
COLOUR RGB(GREEN), RGB(BLACK)             ' Set the default colours
FONT 6                                    ' Set the default font
BOX 0, 0, MM.HRes-1, MM.VRes/2, 3, RGB(RED), DBlue
DO
  TEXT MM.HRes/2, MM.VRes/4, TIME$, "CM", 6, 1, RGB(CYAN), DBlue
  TEXT MM.HRes/2, MM.VRes*3/4, DATE$, "CM"
LOOP
```

The program starts by defining a constant with a value corresponding to a dark blue colour and then sets the defaults for the colours and the font. It then draws a box with red walls and a dark blue interior. Following this the program enters a continuous loop where it performs two functions:

- Displays the current time inside the previously drawn box. The string is drawn centred both horizontally and vertically (the CM in the TEXT commands) in the middle of the box. Note that the TEXT command overrides both the default font and colours to set its own parameters.
- Draws the date centred in the lower half of the screen. In this case the TEXT command uses the default font and colours previously set.

The screenshot on the right shows the result.

Rotated Text

As described above the alignment of the text in the TEXT command can be specified by using one or two characters. In addition a third character can be used to indicate the rotation of the text. This character can be one of:

- N for normal orientation
- V for vertical text with each character under the previous running from top to bottom.
- I the text will be inverted (ie, upside down)
- U the text will be rotated counter clockwise by 90º
- D the text will be rotated clockwise by 90º

As an example, the following will display the words "Vertical Text" vertically down the left hand margin of the monitor and centred vertically:

```
TEXT 0, 250, "Vertical Text", "LMV", 5
```

Positioning is relative to the top left corner of the character when viewed normally so inverted 100,100 will have the top left pixel of the first character at 100,100 and the text will then be above y=101 and to the left of x=101. Similarly “R” in the alignment string is viewed from the perspective of the character in whatever orientation it is in.

Transparent Text

The TEXT command will allow the use of -1 for the background colour. This means that the text is drawn over the background with the background image showing through the gaps in the letters.

Displaying Images

Using the LOAD command you can load an image from the file system and display it on the MMBasic window. Supported formats are BMP, GIF, JPG and PNG and the image can be positioned anywhere on the screen.

There are some limitations on the format of the images and these are detailed in the commands later in this manual. The most flexible is the LOAD BMP command which supports all types of the BMP format including black and white and true colour 24-bit images. The image can be positioned anywhere on the MMBasic window and be of any size (pixels that end up being positioned off the MMBasic window will be ignored).

Advanced Graphics

MMBasic for Windows incorporates a suite of advanced graphic controls (**G**raphical **U**ser **I**nput) that respond to mouse click, these include on screen switches, buttons, indicator lights, keyboard, etc. MMBasic will draw the control and animate it (i.e. a switch will appear to depress when clicked with the mouse). All that the BASIC program needs to do is invoke a single command to specify the basic details of the control.

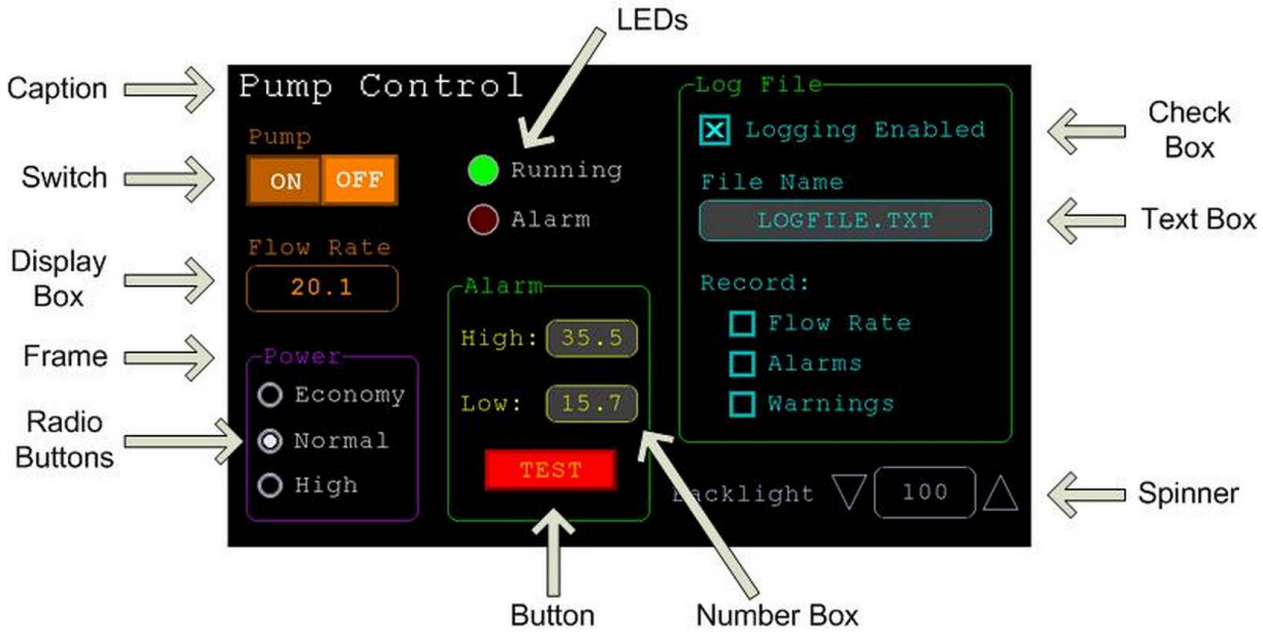
To use the GUI controls in MMBasic 4 Windows the memory required for the GUI controls must be allocated first by using the command OPTION GUI CONTROLS. Typically you would use the command like this:

```
OPTION GUI CONTROLS 75
```

This will set the maximum number of controls that you can define to 75. This option is permanent (i.e. it will be remembered on power down). By default the maximum number of controls is set to zero and in this case the GUI features will not be available and no memory will be used.

Defining Controls

These are some of the advanced GUI controls that you can use:



Each control has a reference number called '#ref' in the description of the control. This can be any number between 1 and the upper limit set by the OPTION CONTROL command. This reference number is used to identify a control. For example, a check box can be created with a reference number of #10:

```
GUI CHECKBOX #10, "Test", 100, 100, 50, rgb(BLUE)
```

Once created the user can check and uncheck the box using the touch feature of the LCD panel without the running BASIC program being involved. When needed the program can determine the check box value by using its reference number in the CtrlVal() function:

```
IF CtrlVal(#10) THEN ..
```

The # character is optional but serves to remind the programmer that this is not an ordinary number. In the following commands any arguments that are in italic font (e.g. Width, Height) are optional and if not specified will take the value of the previous command that did specify them. This means for example, that a number of radio buttons with the same size and colour can be specified with only the first button having to list all the details. Note that with the colour specification this is different to the basic drawing commands which default to the last COLOUR command.

All strings used in GUI controls and the MsgBox can display multiple lines by using the tilde character (~) to separate each line in the string. For example, a push button's caption can be "ALARM~TEST" and this would be displayed as two lines. For all controls the font used for the caption will be whatever is set with the FONT command and the colours will be whatever was set by the last COLOUR command.

If the display is capable of transparent text these commands will allow the use of -1 for the background colour. This means that the text is drawn over the background with the background image showing through the gaps in the letters.

The advanced graphics controls are:

Frame

```
GUI FRAME #ref, caption$, StartX, StartY, Width, Height, Colour
```

This will draw a frame which is a box with round corners and a caption. A frame does not respond to touch but is useful when a group of controls need to be visually brought together. It can also be used to surround a group of radio buttons and MMBasic will arrange for the radio buttons surrounded by the frame to be exclusive – that is, when one radio button is selected any other button that was selected and within the frame will be deselected.

LED

```
GUI LED #ref, caption$, CenterX, CenterY, Diameter, Colour
```

This will draw an indicator light (it looks like a panel mounted LED). When its value is set to one it will be illuminated and when it is set to zero it will be off (a dull version of its colour attribute). The LED can be made to flash by setting its value to the number of milliseconds that it should remain on before turning off. The caption will be drawn to the right of the LED and will use the colours set by the COLOUR command. The LED control is not animated when touched but its reference number can be found using TOUCH(REF) and TOUCH(LASTREF) in the touch interrupts and any required animation can be done in MMBasic.

Check Box

```
GUI CHECKBOX #ref, caption$, StartX, StartY, Size, Colour
```

This will draw a check box which is a small box with a caption. Both the height and width are specified with the 'Size' parameter. When touched an X will be drawn inside the box to indicate that this option has been selected and the control's value will be set to 1. When touched a second time the check mark will be removed and the control's value will be zero. The caption will be drawn to the right of the Check Box and will use the colours set by the COLOUR command.

Push Button

```
GUI BUTTON #ref, caption$, StartX, StartY, Width, Height, FColour, BColour
```

This will draw a momentary button which is a square switch with the caption on its face. When touched the visual image of the button will appear to be depressed and the control's value will be 1. When the touch is removed the value will revert to zero. Caption can be a single string with two captions separated by a vertical bar (|) character (e.g. "UP|DOWN"). When the button is up the first string will be used and when pressed the second will be used.

Switch

```
GUI SWITCH #ref, caption$, StartX, StartY, Width, Height, FColour, BColour
```

This will draw a latching switch with the caption on its face. When touched the visual image of the button will appear to be depressed and the control's value will be 1. When touched a second time the switch will be released and the value will revert to zero. Caption can be a single string with two captions separated by a | character (e.g. "ON|OFF"). When this is used the switch will appear to be a toggle switch with each half of the caption used to label each half of the toggle switch.

Radio Button

```
GUI RADIO #ref, caption$, CenterX, CenterY, Radius, Colour
```

This will draw a radio button with a caption. When touched the centre of the button will be illuminated to indicate that this option has been selected and the control's value will be 1. When another radio button is selected the mark on this button will be removed and its value will be zero.

Radio buttons are grouped together when surrounded by a frame and when one button in the group is selected all others in the group will be deselected. If a frame is not used all buttons on the screen will be grouped together.

The caption will be drawn to the right of the button and will use the colours set by the COLOUR command.

Display Box

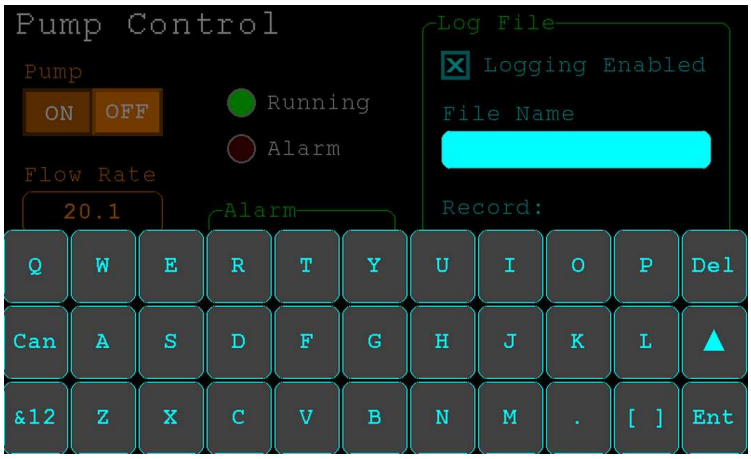
```
GUI DISPLAYBOX #ref, StartX, StartY, Width, Height, FColour, BColour
```

This will draw a box with rounded corners. Any string can be displayed in the box by using the CtrlVal(r) = command. This is useful for displaying text, numbers and messages. This control is not animated when touched but its reference number can be found using TOUCH(REF) and TOUCH(LASTREF) in the touch interrupts and any required animation can be done in MMBasic.

Text Box

```
GUI TEXTBOX #ref, StartX, StartY, Width, Height, FColour, BColour
```

This will draw a box with rounded corners. When the box is touched a QWERTY keyboard will appear on the screen as shown on the right. Using this virtual keyboard any text can be entered into the box including upper/lower case letters, numbers and any other characters in the ASCII character set. The new text will replace any text previously in the box.



Ent is the enter key, **Can** is the cancel key and will close the text box and return it to its original state, the triangle is the shift key, the [ ] key will insert a space and the &12 key will select an alternate key selection with numbers and special characters (there are two sets of special characters and the shift key will switch between them).

The displayed string can be set by assigning a string to the box using the CtrlVal(r) = command. The value of the control can also be set to a string starting with two hash characters (##) and in that case the string (without the leading two hash characters) will be displayed in the box with reduced brightness. This can be used to give the user a hint as to what should be entered (called "ghost text"). Reading the value of the control displaying ghost text will return an empty string. When a key is pressed the ghost text will vanish and be replaced with the entered text.

MMBasic will try to position the virtual keyboard on the screen to not obscure the text box that caused it to appear. A pen down interrupt will be generated just before the keyboard is deployed and a key up interrupt will be generated when the Enter or Cancel keys are touched and the keyboard is hidden.

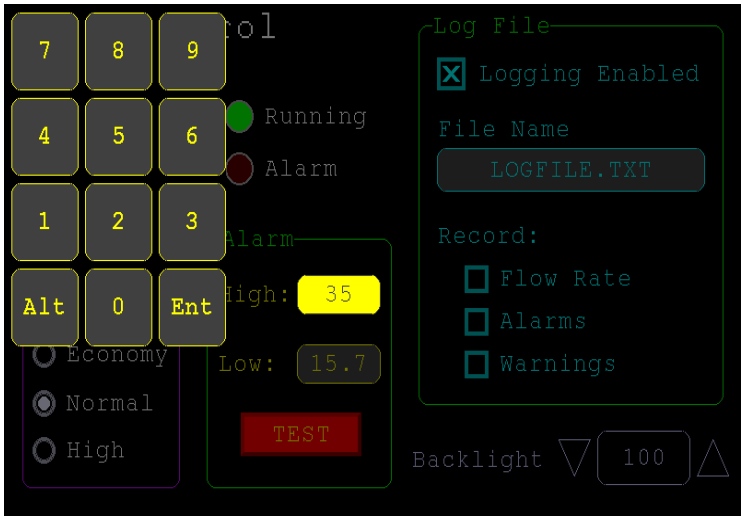
If necessary the virtual keyboard can be dismissed by the program (same as touching the cancel button) with the command:

```
GUI TEXTBOX CANCEL
```

Number Box

```
GUI NUMBERBOX #ref, StartX, StartY, Width, Height, FColour, BColour
```

This will draw a box with rounded corners. When the box is touched a numeric keypad will appear on the screen as shown on the right. Using this virtual keypad any number can be entered into the box including a floating point number in exponential format. The new number will replace the number previously in the box.



The Alt key will select an alternative key selection and the other special keys are the same as with the text box.

The displayed number can also be set by assigning a number (float or integer) to the box using the CtrlVal(r) = command. Similar to the Text Box, the value of the control can set to a literal string with two leading hash characters (e.g. "##Hint") and in that case the string (without the leading two characters) will be displayed in the box with reduced brightness.

Reading this will return zero and when a key is pressed the ghost text will vanish.

MMBasic will try to position the virtual keypad on the screen to not obscure the number box that caused it to appear. A pen down interrupt will be generated just before the keypad is deployed and a key up interrupt will be generated when the Enter key is touched and the keypad is hidden. Also, when the Enter key is touched the entered text will be evaluated as a number and the NUMBERBOX control redrawn to display this number.

Formatted Number Box

```
GUI FORMATBOX #ref, Format, StartX, StartY, Width, Height, FColour, BColour
```

This will draw a box with rounded corners. When the box is touched a numeric keypad will appear similar to a Number Box. The difference is that the Formatted Number Box will require the user to enter numbers according to a specific format for dates, time, etc. Invalid keys on the keypad will be disabled and the user will guided in their entry with guide text. This means that the programmer can be assured that the entry made by the user will always be in a fixed format.

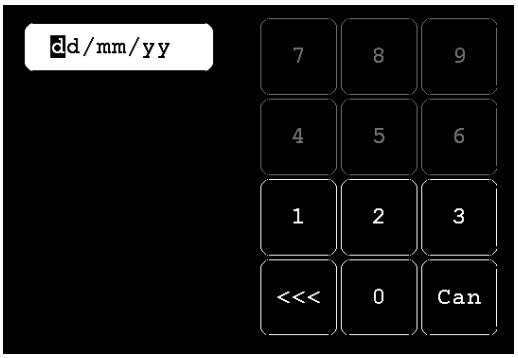
The type of entry is controlled by the 'Format' argument as follows:

- DATE1
  Date in UK/Aust/NZ format (dd/mm/yy)
- DATE2
  Date in USA format (mm/dd/yy)
- DATE3
  Date in international format (yyyy/mm/dd)
- TIME1
  Time in 24 hour notation (hh:mm)
- TIME2
  Time in 24 hour notation with seconds (hh:mm:ss)
- TIME3
  Time in 12 hour notation (hh:mm AM/PM)
- TIME4
  Time in 12 hour notation with seconds (hh:mm:ss AM/PM)
- DATETIME1
  Both date (UK fmt) and time (12 hour) (dd/mm/yy hh:mm AM/PM)
- DATETIME2
  Both date (UK fmt) and time (24 hour) (dd/mm/yy hh:mm)
- DATETIME3
  Both date (USA fmt) and time (12 hour) (mm/dd/yy hh:mm AM/PM)
- DATETIME4
  Both date (USA fmt) and time (24 hour) (mm/dd/yy hh:mm)
- LAT1
  Latitude in degrees, minutes and seconds (d°` mm' ss" N/S)
- LAT2
  Latitude with seconds to one decimal place (dd° mm' ss.s" N/S)
- LONG1
  Longitude in degrees, minutes and seconds (ddd° mm' ss" E/W)
- LONG2
  Longitude with seconds to one decimal place (ddd° mm' ss.s" E/W)
- ANGLE1
  Angle in degrees and minutes (ddd° mm')

For example:

```
GUI FORMATBOX #1, DATE1, 300, 150, 200, 50
```

would create a data entry box and when it is touched a keypad will appear as shown on the right . Note that:-



- The display box is filled with a guide string to prompt the user as to the data required.
- Because the day of the month can only start with a digit from 0 to 3 all other keys are disabled. This also happens with other numbers that have a limited range.
- The value of the control retrieved via CtrlVal(#1) is a string.

As an example, if the user entered the date for the 8th of May 2020 the returned string would be "08/05/20" (i.e. the UK/Aust/NZ format as specified by DATE1).

The value of the control can be pulled apart using the string functions or, in some cases, the string can be used directly. For example, if using the above format box to get a date from the user the Micromite eXtreme's internal clock could then be directly set as follows:

```
DATE$ = CtrlVal(#1)
```

The RTC SETTIME command will accept a single string argument in the format of dd/mm/yy hh:mm so similarly the RTC time could be set as follows if the formatted box used DATETIME2 for 'Format':

```
RTC SETTIME CtrlVal(#1)
```

You can use the USA style DATETIME4 to get the date/time. In that case you would use this to set the RTC:

```
RTC SETTIME MID$(CtrlVal(#1),4,3) + LEFT$(CtrlVal(#1),2) + RIGHT$((CtrlVal(#1),9)
```

MMBasic will try to position the virtual keypad on the screen so as to not obscure the format box that caused it to appear. A pen down interrupt will be generated when the keypad is deployed and a key up interrupt will be generated when all the required data has been entered and the keypad is hidden.

- Spin Box

```
GUI SPINBOX #ref, StartX, StartY, Width, Height, FColour, BColour, Step,Minimum,
Maximum
```

This will draw a box with up/down icons on either end. When these icons are touched the number in the box will be incremented or decremented by the 'StepValue', holding down the touch will repeat at a fast rate.

'Minimum' and 'Maximum' set a limit on the value that can be entered.

'StepValue', 'Minimum' and 'Maximum' are optional and if not specified 'StepValue' will be 1 and there will be no limit on the number entered. A pen down interrupt will be generated every time up/down is touched or when automatic repeat occurs.

Caption

```
GUI CAPTION #ref, text$, StartX, StartY, Alignment, FColour, BColour
```

This will draw a text string on the screen. It is similar to the basic drawing command TEXT, the difference being that MMBasic will automatically dim this control if a keyboard or number pad is displayed.

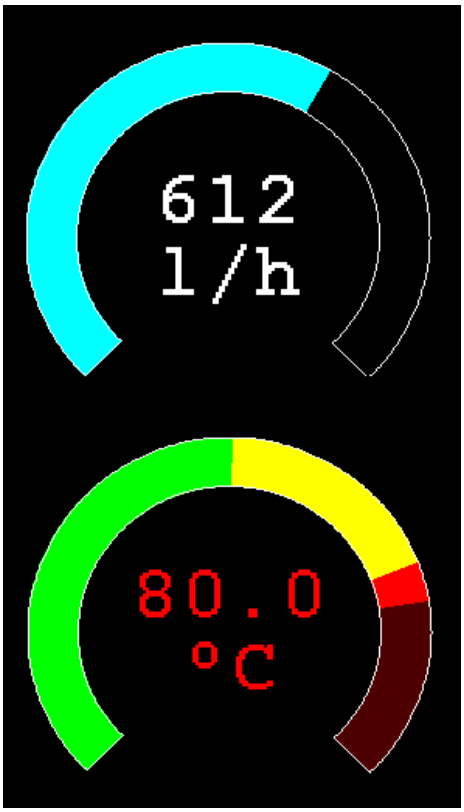
'Alignment' is zero to three characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE, BOTTOM.

A third character can be used to indicate the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (i.e. upside down), 'U' the text will be rotated counter clockwise by 90o and 'D' the text will be rotated clockwise by 90o. The default alignment is left/top with no rotation.

If the colours are not specified this control will use the colours set by the COLOUR command.

Circular Gauge

```
GUI GAUGE #ref, StartX, StartY, Radius, FColour, BColour, min, max, nbrdec,
units$, c1, ta, c2, tb, c3, tc, c4
```



This will define a graphical circular analog gauge with a digital display in the centre showing the value and units. If specified the gauge will be coloured to provide a graphical indication of the signal level (e.g. green for OK, yellow for warning, etc).

'StartX' and 'StartY' are the coordinates of the centre of the gauge while 'Radius' is the distance from the centre to the outer edge.

'min' is the value associated with the minimum value of the gauge and 'max' is the maximum value. When CtrlVal() is used to assign a value (floating point or integer) to the gauge the analogue portion of the gauge will be drawn to a length proportional to the range between 'min' and 'max'.

At the same time the digital value will be drawn in the centre of the gauge using the current font settings (set with the FONT command). 'nbrdec' specifies the number of decimal places to be used in this display. Under the digital value the 'units$' will be displayed (this can be skipped or a zero length string used if not required).

Normally the analog graph is drawn using the colour specified in 'Fcolour' however a multi colour gauge can be created using 'c1' to 'c4' for the colours and 'ta' to 'tc' for the thresholds used to determine when the colour will change.

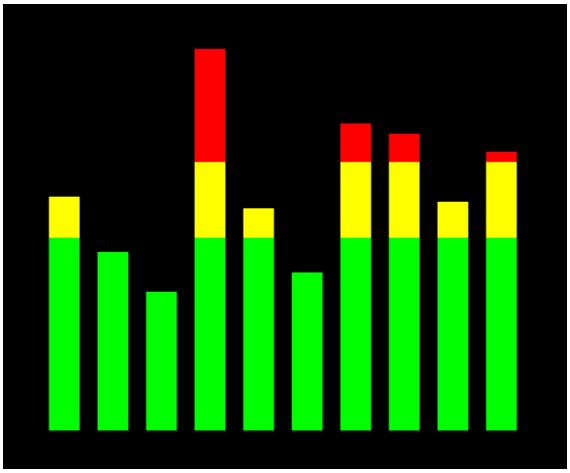
Specifically, 'c1' is the colour to be used for values up to 'ta'. 'c2' is the colour to be used for values between 'ta' and 'tb', 'c3' is used for values between 'tb' and 'tc' and c4 is used for values above 'tc'. Colours and thresholds not required can be left off then list. For example, for a two colour gauge only 'c1', 'ta' and 'c2' need to be specified.

When colours and thresholds are specified the background of the gauge will be drawn with a dull version of the gauge colour at that level ("ghost colouring") so that the user can appreciate how close to the various thresholds the actual value is. Also the digital value displayed in the centre will also change to the colour specified by the current value.

If only one colour is required for the whole analogue graph it can be specified by just using 'c1' and leaving all the following parameters off.

Bar Gauge

```
GUI BARGAUGE #ref, StartX, StartY, width, height, FColour, BColour, min, max, c1,
ta, c2, tb, c3, tc, c4
```



This will define either a horizontal or vertical bar gauge. The gauge can be coloured to provide a graphical indication of the signal level (e.g. green for OK, yellow for warning, etc) and many bar graphs can be packed close together so that a number of values can be displayed simultaneously using a small amount of screen space (as shown in the image which consists of ten bar gauges).

If the width is less that the height the bar gauge will be drawn vertically with the analogue graph growing from the bottom towards the top. Otherwise, if the width is more that the height, it will be drawn horizontally with the analogue graph growing from the left towards the right.

In both cases 'StartX' and 'StartY' reference the top left coordinate of the bar graph while 'width' is the horizontal width and 'height' the vertical height.

The bar graph does not have a digital display of its value but other than that the parameters are the same as for the circular gauge (described above).

'min' and 'max' specify the range of values for the bar and, if specified, 'c1' to 'c4' and 'ta' to 'tc' specify the colours and thresholds for the analogue bar image. Note that unlike the circular bar gauge a "ghost image" of the colours is not shown in the background.

As with the circular gauge, if only one colour is required for the whole gauge it can be specified by just using 'c1' and leaving all the following parameters off.

Area

```
GUI AREA #ref, StartX, StartY, Width, Height
```

This will define an invisible area of the screen that is sensitive to touch and will set TOUCH(REF) and TOUCH(LASTREF) accordingly when touched or released. It can be used as the basis for creating a custom control which is defined and managed by the BASIC program.

Interacting with Controls

Using the following commands and functions the characteristics of the on screen controls can be changed and their value retrieved.

```
a= CTRLVAL(#ref)
```

This is a function that will return the current value of a control. For controls like check boxes or switches it will be the number one (true) indicating that the control has been selected by the user or zero (false) if not.

For controls that hold a number (e.g. a SPINBOX) the value will be the number (normally a floating point number). For controls that hold a string (e.g. TEXTBOX) the value will be a string. For example:

```
PRINT "The number in the spin box is: " CTRLVAL(#10)
CTRLVAL(#ref) =1
```

This command will set the value of a control. For off/on controls like check boxes it will override any touch input and can be used to depress/release switches, tick/untick check boxes, etc. A value of zero is off or unchecked and non zero will turn the control on. For a LED it will cause the LED to be illuminated or turned off. It can also be used to set the initial value of spin boxes, text boxes, etc. For example:

```
CTRLVAL(#10) = 12.4
```

GUI Sub-commands

```
GUI FCOLOUR colour, #ref1 [, #ref2, #ref3, etc]
```

This will change the foreground colour of the specified controls to 'colour'. This is especially handy for a LED which can change colour.

```
GUI BCOLOUR colour, #ref1 [, #ref2, #ref3, etc]
```

This will change the background colour of the specified controls to 'colour'.

```
a= TOUCH(REF)
```

This is a function that will return the reference number of the control currently being touched. If no control is currently being touched it will return zero.

```
a= TOUCH(LASTREF)
```

This is a function that will return the reference number of the control that was last touched.

```
GUI DISABLE #ref1 [, #ref2, #ref3, etc]
```

This will disable the controls in the list. Disabled controls do not respond to touch and will be displayed

dimmed.

The keyword ALL can be used as the argument and that will disable all controls on the currently

displayed page. For example:

```
GUI DISABLE ALL

GUI ENABLE #ref1 [, #ref2, #ref3, etc]
```

This will undo the effects of GUI DISABLE and restore the controls in the list to normal operation. The keyword ALL can be used as the argument for all controls on the currently displayed page.

```
GUI HIDE #ref1 [, #ref2, #ref3, etc]
```

This will hide the controls in the list. Hidden controls will not respond to touch and will be replaced on the screen with the current background colour. The keyword ALL can be used as the argument.

```
GUI SHOW #ref1 [, #ref2, #ref3, etc]
```

This will undo the effects of GUI HIDE and restore the controls in the list to being visible and capable of normal operation. The keyword ALL can be used as the argument for all controls.

```
GUI DELETE #ref1 [, #ref2, #ref3, etc]
```

This will delete the controls in the list. This includes removing the image of the control from the screen using the current background colour and freeing the memory used by the control. The keyword ALL can be used as the argument and that will cause all controls to be deleted.

- MsgBox()

The MsgBox() function will display a message box on the screen and wait for user input. While the message box is displayed all controls will be disabled so that the message box has the complete focus.

The syntax is:

```
r = MsgBox(message$, button1$ [, button2$ [, button3$ [, button4$]]])
```

All arguments are strings. 'message$' is the message to display. This can contain one or more tilde characters (~) which indicate a line break. Up to 10 lines can be displayed inside the box. 'button1$' is the caption for the first button, 'button2$' is the caption for the second button, etc. At least one button must be specified and four is the maximum. Any buttons not included in the argument list will not be displayed.

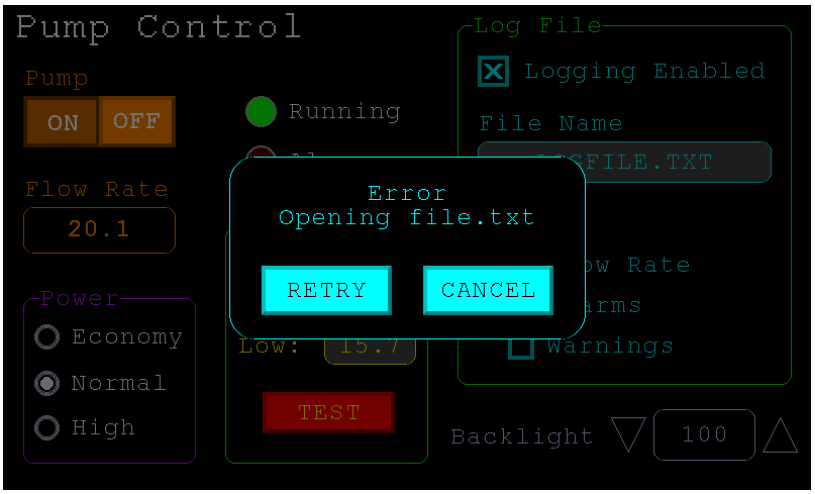
The font used will be the default font set using the FONT command and the colours used will be the defaults set by the COLOUR command. The box will be automatically sized taking into account the dimensions of the default font, the number of lines to display and the number of buttons specified.

When the user touches a button the message box will erase itself, restore the display (e.g. re enable all controls) and return the number of the button that was touched (the first button will return 1, the second 2, etc). Note that, unlike all other GUI controls the BASIC program will stop running while the message box is displayed, interrupts however will be honoured and acted upon.

To illustrate the usage of a message box will the following program fragment will attempt to open a file and if an error occurs the program will display an error message using the MsgBox() function. The message has two lines and the box has two buttons for retry and cancel.

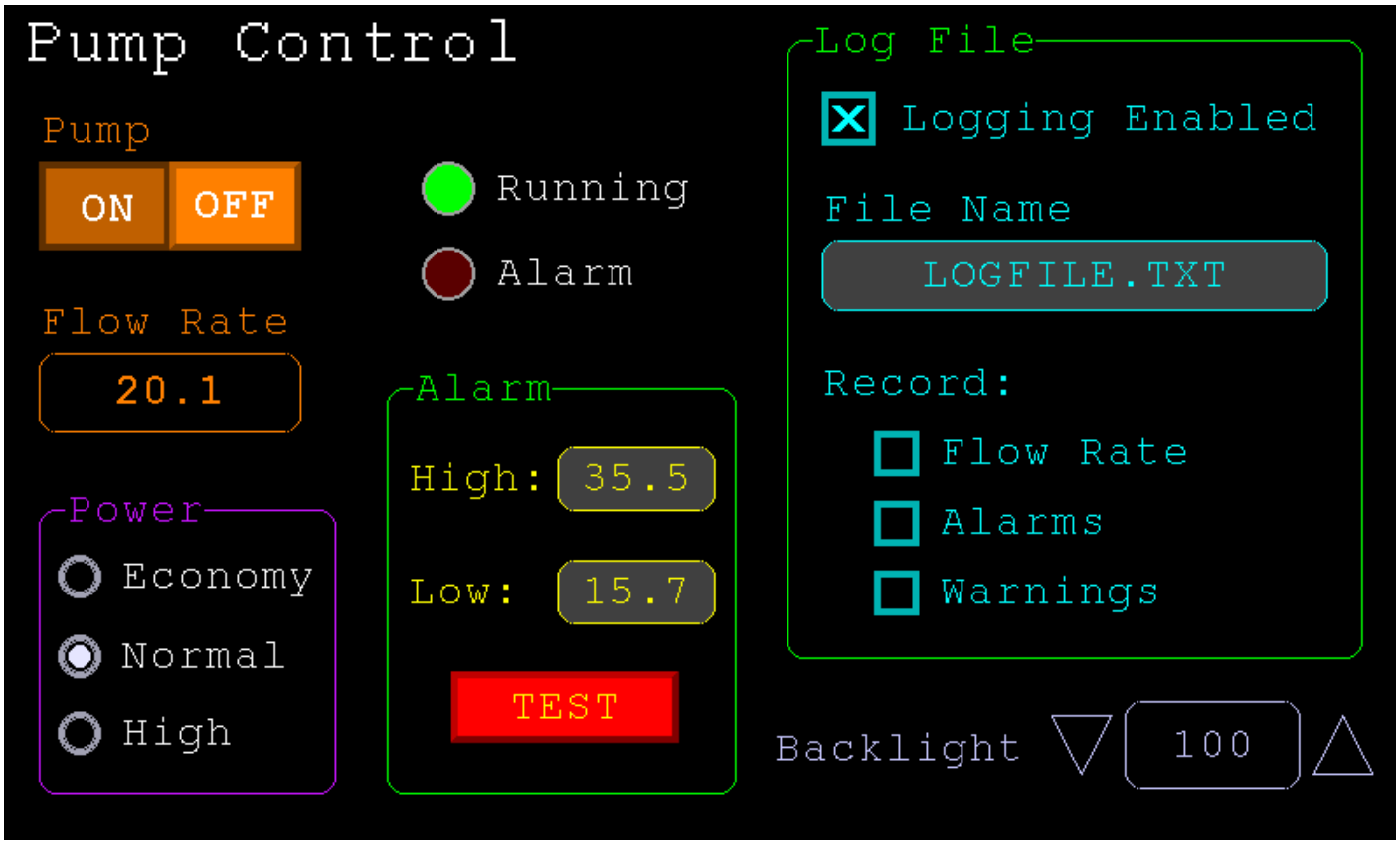
```
Do
On Error Skip
Open "file.txt" For Input As #1
If MM.ErrNo <> 0 Then
if MsgBox("Error~Opening file.txt","RETRY","CANCEL") = 2 Then Exit Sub
EndIf
Loop While MM.ErrNo <> 0
```

This would be the result if the file "file.txt" did not exist:



Example Program

As a test you can enter the following "Pump Control" demonstration program. It will draw a selection of advanced controls as shown below. These controls are active so that you can test how they work.




```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Demonstration program for MMBasic for Windows
' It does not do anything useful except demo the various controls
'
' Geoff Graham, October 2015
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
Option Explicit
Dim ledsY
Colour RGB(white), RGB(black)
' reference numbers for the controls are defined as constants
Const c_head = 1, c_pmp = 2, sw_pmp = 3, c_flow = 4, tb_flow = 5
Const led_run = 6, led_alarm = 7
Const frm_alarm = 20, nbr_hi = 21, nbr_lo = 22, pb_test =23
Const c_hi = 24, c_lo = 25
Const frm_pump = 30, r_econ = 31, r_norm = 32, r_hi = 33
Const frm_log = 40, cb_enabled = 41, c_fname = 42, tb_fname = 43
Const c_log = 44, cb_flow = 45, cb_pwr = 46, cb_warn = 47
Const cb_alarm = 48, c_bright = 49, sb_bright = 50
' now draw the "Pump Control" display
CLS
GUI Interrupt TouchDown, TouchUp
' display the heading
Font 2,2 : GUI Caption c_head, "Pump Control", 10, 0
Font 3 : GUI Caption c_pmp, "Pump", 20, 60, , RGB(brown)
' now, define and display the controls
' first display the switch
Font 4
GUI Switch sw_pmp, "ON|OFF", 20, 90, 150, 50, RGB(white),RGB(brown)
CtrlVal(sw_pmp) = 1
' the flow rate display box
Font 3 : GUI Caption c_flow, "Flow Rate", 20, 170,, RGB(brown),0
Font 4 : GUI Displaybox tb_flow, 20, 200, 150, 45
CtrlVal(tb_flow) = "20.1"
```

```
' the radio buttons and their frame
Font 3 : GUI Frame frm_pump, "Power", 20, 290, 170, 163,RGB(200,20,255)
GUI Radio r_econ, "Economy", 43, 328, 12, RGB(230, 230, 255)
GUI Radio r_norm, "Normal", 43, 374
GUI Radio r_hi, "High", 43, 418
CtrlVal(r_norm) = 1
' start with the "normal" button selected
' the alarm frame with two number boxes and a push button switch
Font 3 : GUI Frame frm_alarm, "Alarm", 220, 220, 200, 233,RGB(green)
GUI Caption c_hi, "High:", 232, 260, "LT", RGB(yellow)
GUI Numberbox nbr_hi, 318,MM.Info(VPos)-
6,90,MM.FontHeight+12,RGB(yellow),RGB(64,64,64)
GUI Caption c_lo, "Low:", 232, 325, LT, RGB(yellow),0
GUI Numberbox nbr_lo, 318,MM.Info(VPos)-
6,90,MM.FontHeight+12,RGB(yellow),RGB(64,64,64)
GUI Button pb_test, "TEST", 257, 383, 130, 40,RGB(yellow), RGB(red)
CtrlVal(nbr_lo) = 15.7 : CtrlVal(nbr_hi) = 35.5
' draw the two LEDs
Const ledsX = 255, coff = 50
' define their position
ledsY = 105 : GUI LED led_run, "Running", ledsX, ledsY, 15, RGB(green)
ledsY = ledsY+49 : GUI LED led_alarm, "Alarm", ledsX, ledsY, 15, RGB(red)
CtrlVal(led_run) = 1
' the switch defaults to on so set the LED on
' the logging frame with check boxes and a text box
Colour RGB(cyan), 0
GUI Frame frm_log, "Log File", 450, 20, 330, 355, RGB(green)',0
GUI Checkbox cb_enabled, "Logging Enabled", 470, 50, 30, RGB(cyan)
GUI Caption c_fname, "File Name", 470, 105
GUI Textbox tb_fname, 470, 135, 290, 40, RGB(cyan), RGB(64,64,64)
GUI Caption c_log, "Record:", 470, 205, , RGB(cyan), 0
GUI Checkbox cb_flow, "Flow Rate", 500, 245, 25
GUI Checkbox cb_alarm, "Alarms", 500, 285, 25
GUI Checkbox cb_warn, "Warnings", 500, 325, 25
CtrlVal(cb_enabled) = 1
CtrlVal(tb_fname) = "LOGFILE.TXT"
' define and display the spinbox for controlling the backlight
GUI Caption c_bright, "Backlight", 442, 415, ,RGB(200,200,255),0
GUI Spinbox sb_bright, MM.Info(HPos) + 8, 400, 200, 50,,,10, 10, 100
CtrlVal(sb_bright) = 100
' All the controls have been defined and displayed. At this point
' the program could do some real work but because this is just a
' demo there is nothing to do. So it just sits in a loop.
Do : Loop
' the interrupt routine for touch down
' using a select case command it has a different process for each control
Sub TouchDown
Select Case Touch(REF)
' find out the control touched
Case cb_enabled
' the enable check box
If CtrlVal(cb_enabled) Then
GUI ENABLE c_fname, tb_fname, c_log, cb_flow, cb_alarm, cb_warn
Else
GUI Disable c_fname, tb_fname, c_log, cb_flow, cb_alarm, cb_warn
EndIf
Case sb_bright
' the brightness spin box only - no brightness control in MMB4W
'BackLight CtrlVal(sb_bright)
Case sw_pmp
' the pump on/off switch
CtrlVal(led_run) = CtrlVal(sw_pmp)
CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 20.1)
CtrlVal(r_norm) = 1
Case pb_test
' the alarm test button
CtrlVal(led_alarm) = 1
GUI beep 250
```

```
Case r_econ
' the economy radio button
CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 18.3)
Case r_norm
' the normal radio button
CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 20.1)
Case r_hi
' the high radio button
CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 23.7)
End Select
End Sub
' interrupt routine when the touch is removed
Sub TouchUp
Select Case Touch(LASTREF)
' use the last reference
Case pb_test
' was it the test button
CtrlVal(led_alarm) = 0
' turn off the LED
End Select
End Sub
```

Advanced Graphics Programming Techniques

When programming using the advanced GUI commands there are a number of hints and techniques to consider that will make it easier to develop and maintain your program.

The User Should Be In Control

Traditional character based programs are normally in control of the interaction with the user. For example, the program may display a menu and prompt the user to select an action. If the user selects an invalid option the program would display an error message and display the menu options again.

However graphical based programs such as that created using the advanced GUI commands are different. Usually the program just starts running doing what it normally does (e.g. control temperature, speed, etc) and it is the user's job to select and change parameters without being prompted. This is a different way of programming and is often hard for the traditional programmer to get used to this different technique.

As an example, consider a program that is to control a cutting device. The traditional program would prompt the user for the speed and cutting time. When both have been entered the program would prompt to start the cutting cycle. However, a graphical based program would display two number boxes where the user could enter the speed and time along with a run button. The number boxes could be filled with default values and the run button would be disabled if the user entered an invalid speed or time. When the run button is touched the cutting cycle would start.

A good example of this type of graphical interface is the dialogue box used on a Windows/IOS/Android computer to set the time and date. It displays a number of boxes where the user can enter the date/time along with an OK button that tells the program to accept the data entered. At no time is the user forced to make a selection from a menu. Also, the current time/date is already displayed in the entry boxes so the user can accept them as the default if they wanted to do so.

If you need some inspiration as to how your graphical program should look and feel check your nearest GUI based operating system to see how they operate.

Program Structure

Typically a program would start by defining the controls (which MMBasic will draw on the screen), then it would set the defaults and finally it would drop into a continuous loop where it would do whatever job it was design to do. For example, take the case of a simple controller for a motor where the user could select the speed and cause the motor to run by pressing an on screen button.

To implement this function the program would look something like this:

```
' Program snippit to show numberbox
' first, set default value for the speed
' then define and draw the number box
' then define and draw the RUN button
DIM MotorSpeed as Integer = 100
DIM RunSwitch as Integer = 0 ' 0 = off, 1 = on
GUI CAPTION #1, "Speed (rpm)", 200, 20
GUI NUMBERBOX #2, 200, 60, 150, 40
CtrlVal(#2) = MotorSpeed ' pre-set speed to 100
GUI BUTTON #3, "RUN", 200, 350, 60, 60, RGB(red),RGB(white)
CtrlVal(#3) = RunSwitch ' pre-set the run button to off
' now the main program loop

DO
  IF CtrlVal(#2) <10 OR CtrlVal(#2) >200 THEN
    GUI DISABLE #3 ' check the speed setting, disable RUN if it is invalid
  ELSE
    GUI ENABLE #3 ' otherwise, enable the RUN button
  ENDIF
    IF CtrlVal(#3) = 0 THEN
  ' note the switch is off "ie. not pressed"
    MotorSpeed = 0
  ELSE ' on
    MotorSpeed = CtrlVal(#2)
  ENDIF
  SetMotorSpeed (MotorSpeed) ' call the sub to set the speed
LOOP   ' this runs in a loop forever

' if the button was pressed, make the motor run
```

```
' otherwise the button is up therefore set motor speed to zero
`
SUB SetMotorSpeed arg1
  ` dummy subroutine - would normally set the speed of the motor
  ` just print out the speed
  TEXT 10,250,"                                                  " ` clear line
  TEXT  10,250, "This sub sets the motor to speed "+str$(arg1)
END SUB
```

Note that the user is not prompted to do anything; the program just sits in a loop reacting to the changes that the user has made to the controls (i.e. the user is in control).

Disable Invalid Controls

As in the above example, disabling a control will prevent a user from using it and MMBasic will redraw it in a dull colour to indicate that it is not available. This is the equivalent of an error message in a traditional text based program and is more user friendly than popping up a message box which must be dismissed before anything else can be done.

There are many times that a control could be invalid, for example when an input is not ready or simply when an option or action does not apply. Later, when the control becomes valid you can use the GUI ENABLE command to return it to use. Another example is when a GUI NUMBERBOX keypad is displayed MMBasic will automatically disable all other controls on the screen so that it is obvious to the user where their input is required.

Disabling a control still leaves it on the screen, so that the user knows that it is there but it will be dimmed and will not respond to a mouse click. Not responding to mouse clicks also means that the user cannot change it and an interrupt will not be generated when it is clicked. This is handy for you the programmer because you do not have to check if the control is valid before acting on it.

Use Constants for Control Reference Numbers

The advanced controls use a reference number to identify the control. To make it easy to read and maintain your program you should define these numbers as constants with easy to recognise names. For example, in the following program fragment MAIN_SWITCH is defined as a constant and this constant is used wherever the reference number for that control is required:

```
CONST MAIN_SWITCH = 5
CONST ALARM_LED = 6
'...
GUI SWITCH MAIN_SWITCH, "ON|OFF", 330, 50, 140, 50, RGB(white), RGB(blue)
GUI LED ALARM_LED, 215, 220,30, RGB(red)
'...
IF CtrlVal(MAIN_SWITCH) = 0 THEN ...
' for example turn the pump off
IF ALARM THEN CtrlVal(ALARM_LED) = 1
```

It is much easier to remember what MAIN_SWITCH does than remembering what control the number 5 refers to. Also, when you have a lot of controls it is much easier to renumber the controls when all their numbers are defined at the one place at the start of the program.

The reference number must be a number between 1 and the value set with the OPTION CONTROL command. Increasing this number will consume more RAM and decreasing it will recover some RAM.

The Main Program Is Still Running

It is important to realise that your main BASIC program is still running while the user is interacting with the GUI controls. For example, it will continue running even while a user holds down an on screen switch and it will keep running while the virtual keyboard is displayed as a result of touching a TEXTBOX control.

For this reason your main program should not arbitrarily update click sensitive screen controls, because they might change the on screen image while the user is using them (with undefined results). Normally when a BASIC program using GUI controls starts it will initialise controls such as a SPINBOX, NUMBERBOX and TEXTBOX to some initial value but from then on the main program should just read the value of these controls – it is the responsibility of the user to change these, not your program.

However, if you do want to change the value of such an on-screen control you need some mechanism to prevent both the program and the user making a change at the same time. One method is to set a flag within the key down interrupt to indicate that the control should not be updated during this time. This flag can then be cleared in the key up interrupt to allow the main program to resume updating the control.

Note that this discussion only applies to controls that respond to mouse clicks. Controls such as CAPTION and LED can be changed at any time by the main program and often are.

Use Interrupts and SELECT CASE Statements

Everything that happens on a screen using the advanced controls will be signalled by an interrupt, either left mouse click up or down or right mouse button click up or down. The mouse wheel click can be detected with the MOUSE function but it does not generate an interrupt. So, if you want to do something immediately when a control is changed, you should do it in an interrupt. Mostly you will be interested in when the mouse click is down but in some cases you might also want to know when it is released.

Because the interrupt is triggered when the mouse click occurs on any control or part of the screen you need to discover what control was being activated. This is best performed using the MOUSE(REF) function and the SELECT CASE statement. For example, in the following fragment the subroutine PenDown will be called when there is a mouse click and the function MOUSE(REF) will return the reference number of the control being clicked.

Using the SELECT CASE the alarm LED will be turned on or off depending on which button is clicked. The action could be any number of things like selecting a menu item or printing a message on the console.

```
CONST ALARM_ON = 15
CONST ALARM_OFF = 16
CONST ALARM_LED = 33
GUI INTERRUPT MouseLeftDown
'...
GUI BUTTON ALARM_ON, "ALARM ON ", 330, 50, 140, 50, RGB(white), RGB(blue)
GUI BUTTON ALARM_OFF, "ALARM OFF ", 330, 150, 140, 50, RGB(white), RGB(blue)
GUI LED ALARM_LED, 215, 220, 30, RGB(red)
DO : LOOP  ' the main program is doing something
' this sub is called when touch is detected
SUB MouseLeftDown
  SELECT CASE MOUSE(REF)
    CASE ALARM_ON
      CtrlVal(ALARM_LED) = 1
    CASE ALARM_OFF
      CtrlVal(ALARM_LED) = 0
  END SELECT
END SUB
```

The SELECT CASE can also test for other controls and perform whatever actions are required for them in their own section of the CASE statement. The important point is that the maintenance of the controls (e.g. responding to the buttons and turning the alarm LED off or on) is done automatically without the main program being involved – it can continue doing something useful like calculating some control response, etc.

Mouse Button Interrupt

In most cases you can process all user input in the mouse button down interrupt. But there are exceptions and a typical example is when you need to change the characteristics of the control that is being clicked. For example, if you wanted to change the foreground colour of a button from white to red when it is down. When it is returned to the up state the colour should revert to white.

Setting the colour on the click down is easy. Just define a mouse button down interrupt and change the colour in the interrupt when that control is clicked. However, to return the colour to white you need to detect when the mouse buttonh has been released (i.e. mouse button up up). This can be done with a mouse button up interrupt.

To specify a mouse button up interrupt you add the name of the subroutine for this interrupt to the end of the GUI INTERRUPT command. For example:

```
GUI INTERRUPT MouseLeftDown, MouseLeftUp
```

Within the mouse button up subroutine you can use the same structure as in the mouse button down sub but you need to find the reference number of the last control that was clicked. This is because the control where the mouse button was clicked may have already changed/moved and no control is currently being activated. To get the number of the last control clicked you need to use the function MOUSE(LASTREF).

The following example shows how you could meet the above requirement and implement both a mouse button down and a mouse button up interrupt:

```
SUB MouseLeftDown
  SELECT CASE MOUSE(REF)
    CASE ButtonRef
      GUI FCOLOUR RGB(RED), ButtonRef
  END SELECT
END SUB
SUB MouseLeftUp
SELECT CASE MOUSE(LASTREF)
  CASE ButtonRef
    GUI FCOLOUR RGB(WHITE), ButtonRef
  END SELECT
END SUB
```

Keep Interrupts Very Short

Because a mouse button interrupt indicates a request by the user it is tempting to do some extensive programming within an interrupt. For example, if the mouse button click indicates that the user wants to send a message to another controller it sounds logical to put all that code within the interrupt. But this is not a good idea because MMBasic cannot do anything else while your program is processing the interrupt and sending a message could take many milliseconds.

Instead your program should update a global variable to indicate what is requested and leave the actual execution to the main program. For example, if the user did click on the "send a message" button, your program could simply set a global variable to true. Then the main program can monitor this variable and if it changes perform the logic and communications required to satisfy the request.

Remember the commandment ***"Thou shalt not hang around in an interrupt"***.

Multiple Screens

Your program might need a number of screens with differing controls on each screen. This could be implemented by deleting the old controls and creating new ones when the screen is switched. But another way to do this is to use the GUI SETUP and GUI PAGE commands. These allow you to organise the controls onto pages and with one simple command you can switch pages. All controls on the old page will be automatically hidden and controls on the new page will be automatically shown.

To allocate controls to a page you use the GUI SETUP nn command where nn refers to the page in the range of 1 to 32. When you have used this command any newly created controls will be assigned to that page. You can use GUI SETUP as many times that you want.

For example, in the program fragment below the first two controls will be assigned to page 1, the second to page 2, etc.

```
GUI SETUP 1
GUI Caption #1, "Flow Rate", 20, 170,, RGB(brown),0
GUI Displaybox #2, 20, 200, 150, 45
GUI SETUP 2
GUI Caption #3, "High:", 232, 260, LT, RGB(yellow)
GUI Numberbox #4, 318, 6,90, 12, RGB(yellow), RGB(64,64,64)
GUI SETUP 3
GUI Checkbox #5, "Alarms", 500, 285, 25
GUI Checkbox #6, "Warnings", 500, 325, 25
```

By default only the controls setup as page 1 will be displayed and the others will be hidden. To switch the screen to page 3 all you need do is use the command GUI PAGE 3. This will cause controls #1 and #2 to be automatically hidden and controls #5 and #6 to be displayed. Similarly GUI PAGE 2 will hide all except #3 and #4 which will be displayed.

You can specify multiple pages to display at the one time, for example, GUI PAGE 1,3 will display both pages 1 and 3 while hiding page 2. This can be useful if you have a set of controls that must be visible all the time. For example, GUI PAGE 1,2 and GUI PAGE 1,3 will leave the controls on page 1 visible while the others are switched on and off.

It is perfectly legal for a program to modify controls on other pages even though they are not displayed at the time. This includes changing the value and colours as well as disabling or hiding them. When the display is switched to their page the controls will be displayed with their new attributes.

It is possible to place the GUI PAGE commands in the touch down interrupt so that pressing a certain control or part of the screen will switch to another page. Note that when ALL is used for the list of controls in commands such as GUI ENABLE ALL this only refers to the controls on the pages that are currently selected for display. Controls on other pages will be unaffected.

All programs start with the equivalent of the commands GUI SETUP 1 and GUI PAGE 1 in force. This means that if the GUI SETUP and GUI PAGE commands are not used the program will run as you would expect with all controls displayed. A typical usage of the GUI PAGE command is shown below.

Two buttons (which are always displayed) allow the user to select between the first page and the second page. The switch is done in the touch down interrupt.

```
GUI SETUP 1
GUI Button #10, "SELECT PAGE ONE", 50, 100, 150, 30, RGB(yellow), RGB(blue)
GUI Button #11, "SELECT PAGE TWO", 50, 140, 150, 30, RGB(yellow), RGB(blue)
GUI SETUP 2
GUI Caption #1, "Displaying First Page", 20, 20
GUI SETUP 3
GUI Caption #2, "Displaying Second Page", 20, 50
Page 1, 2
GUI INTERRUPT MouseLeftDown
Do  ' the main program loop
Loop
Sub MouseLeftDown
  If MOUSE(REF) = 10 Then GUI Page 1, 2
  If MOUSEh(REF) = 11 Then GUI Page 1, 3
End Sub
```

Multiple Interrupts

With many screen pages the interrupt subroutine could get long and complicated. To work around that it is possible to have multiple interrupt subroutines and switch dynamically between them as you wish (normally after switching pages). This is done by redefining the current interrupt routines using the GUI INTERRUPT command.

For example, this program fragment uses different interrupt routines for pages 4 and 5 and they are specified immediately after switching the pages.

```
GUI PAGE 4
GUI INTERRUPT P4mouseleftdown, P4mouseleftup
..
GUI PAGE 5
GUI INTERRUPT P5mouseleftdown, P5mouseleftup
..
```

Using Basic Drawing Commands

There are two types of objects that can be on the screen. These are the GUI controls and the basic drawing objects (PIXEL, LINE, TEXT, etc). Mixing the two on the screen is not a good idea because MMBasic does not track the position of the basic drawing objects and they can clash with the GUI controls.

As a result, unless you are prepared to do some extra programming, you should use either the GUI controls or the basic drawing objects – but you should not use both. So, for example, do not use TEXT but use GUI CAPTION instead. If you only use GUI controls MMBasic will manage the screen for you including erasing and redrawing it as required, for example when a virtual keyboard is displayed.

Note that the CLS command (used to clear the screen) will automatically set any GUI controls on the screen to hidden (i.e. it does a GUI HIDE ALL before clearing the screen). The main problem with mixing basic graphics and GUI controls occurs with the Text Box, Formatted Box and Number Box controls which display a virtual keyboard. This can erase any basic graphics and MMBasic will not know to restore them when the keyboard is removed. If you want to mix basic graphics with GUI controls you should:

- Intercept the mouse button down interrupt for the Text Box, Formatted Box and Number Box controls as that indicates that a virtual keyboard is about to be displayed and that will give you the opportunity to redraw your non GUI basic graphics in anticipation of this event (for example, draw them in a dimmed state to appear as if they are disabled).
- Intercept the mouse button up interrupt for the same controls as that indicates that the virtual keyboard has been removed and you could then redraw any non GUI graphics in their original state.

Overlapping Controls

Controls can be defined to overlap on the display, this mostly occurs with GUI AREA which, as an example, you might want to capture a touch that was intended for (say) a GUI BUTTON. This will allow you to create your own animation for the button rather than that provided by MMBasic. In this case the control that you wish to respond to the mouse button (i.e. GUI AREA) should have a lower reference number (ie. #ref) than the control that it is covering (ie. the GUI BUTTON).

This is because when the mouse button is clicked on an area on the screen, MMBasic will check the current list of active controls starting with control number 1 and working upwards. When a match is made MMBasic will take the appropriate action and terminate the search. This results in the lower numbered control effectively masking out a higher numbered control covering the same screen area as that clicked with the mouse button location.

File System Support

MMBasic for Windows has full support for programs, files and directories on the data storage system of the OS. This includes opening files for reading, writing or random access and editing and running programs.

In the following note that:

- The filename can be a string expression, variable or constant. If it is a constant the string must be quoted (eg, KILL "MYPROG.BAS"). The exception is the RUN command where only a constant is allowed.
- Long file/directory names are supported in addition to the old 8.3 format.
- The maximum file/path length is 127 characters.
- 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 …).
- Forward slashes or back slashes are valid in paths between directories. Eg /dir/file.txt or \dir\file.txt.
- The current MMBasic time is used for file create and last access times.
- Up to ten files can be simultaneously open.
- Input and output commands and functions can also use file #0 which refers to the console.

There are many commands and functions related to the file system. See the Commands/Functions section later in this manual for their full description):

Program Management Commands

All programs reside on the built in storage (eg. hard disk, SD card, memory stick etc.) and they must be present when running, editing and listing programs.

- RUN “prog”

Run a program. ‘prog’ must be a string constant (ie, not a variable).

- EDIT fname$

Edit a program or text file.

- LIST fname$

List on the console a program or text file.

- AUTOSAVE fname$

Receive a file streamed by a computer connected to the serial console.

- XMODEM RECEIVE fname$

Receive a file from a computer connected to the serial console using the XModem protocol.

- XMODEM SEND fname$

Send a file to a computer connected to the serial console using the XModem protocol.

File Access Within a Program

Except for INPUT, LINE INPUT and PRINT 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.

- SEEK #fnbr, pos

Will position the read/write pointer in a file that has been opened for RANDOM access to the 'pos' byte.

- 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'.

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

- INPUT$(nbr, #fnbr)

Will return a string composed of a number of characters read from a file previously opened for INPUT.

- 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

File and Directory Management

- LIST FILES [wildcard] [,sortorder]

Search the current directory and list the files/directories found.

- KILL fname$

Delete a file.

- COPY oldfile$ TO newfile$

Copy a file.

- RENAME oldfile$ AS newfile$

Rename a file.

- MKDIR dname$

Make a sub 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$'.

And there are two functions that are handy for searching and managing files/directories.

- DIR$(fspec, type)

Will search an SD card for files and return the names of the entries found.

- MM.INFO(function)

Returns many types of information related to the storage system (size, free space, file size, etc). Refer to the section Predefined Read Only Variables for more information.

Play Audio Files

- PLAY WAV | FLAC | MP3 file$ [, interrupt]

Play a WAV, FLAC or MP3 audio file on the stereo audio output.

- PLAY MODFILE file$

Play a MOD file on the stereo audio output.

- PLAY EFFECT filename$ [,interrupt]

Play a WAV file at the same time as a MOD file is playing.

Load and Save Images

- LOAD BMP | GIF | JPG | PNG fname$

Load a BMP, GIF, JPG or PNG image and display it on the MMBasic window.

- SAVE IMAGE fname$

Save the current MMBasic window's screen image as a BMP file.

Sequential File Access

Sequential input/output is the standard method of reading or writing to a file and the easiest to understand. When a file is opened it is read from the beginning character by character or line by line. Similarly, when a file is opened for writing the output is sequentially added to the end of the file. This method is often used for recording data or saving temporary information.

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$() will read 12 characters and the second 3 characters. So the variable `ta$` 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 Access

Random access allows the program to jump around within a file so that sections in the middle (ie, not at the end) can be read or written. This method is often used for database type applications where the file consists of many records which have the same fixed length.

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 (64 bytes 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
    ENDIF
    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
```

Audio Output

MMBasic for Windows can play WAV, FLAC and MP3 files from system storage, generate synthesised music in the MOD format, create sound effects as well as generate precise sine wave tones. All these are outputted on the computer audio socket (if available).

Playing WAV, MP3 and FLAC Files

The PLAY command will play an audio file residing on the file system to the sound output. It can be used to provide background music, add sound effects to programs and provide informative announcements.

The syntax of the command is one of the following depending of the format of the file:

```
PLAY WAV file$, interrupt
```

or

```
PLAY MP3 file$, interrupt
```

or

```
PLAY FLAC file$, interrupt
```

file$ is the name of the audio file to play. It must be on the file system and the appropriate extension (eg .WAV) will be appended if missing. The audio will play in the background (ie, the program will continue without pause). *interrupt* is optional and is the name of a subroutine which will be called when the file has finished playing. Most variations in encoding are supported (see the PLAY command in the command listing for the details).

Background Music

If *fname$* in the PLAY WAV/MP3/FLAC command is a directory then the firmware will list all the files of the relevant type in that directory and start playing them one-by-one. To play files in the current directory use an empty string (ie, ""). Each file listed will play in turn and the optional interrupt will fire when all files have been played. The filenames are stored with full path so you can use CHDIR while tracks are playing without causing problems. All files in the directory are listed if the command is executed at the command prompt but the listing is suppressed in a program.

While playing in this background mode the user can edit programs, run programs, etc without interrupting the playing of the music. Amongst other things this allows MMBasic for Windows to be used as a music player while programming or doing other tasks.

Generating Sine Waves

The PLAY TONE command also uses the audio output and will generate sine waves with selectable frequencies for the left and right channels. This feature is intended for generating attention catching sounds but, because the frequency is very accurate, it can be used for many other applications. For example, signalling DTMF tones down a telephone line or testing the frequency response of loudspeakers.

The syntax of the command is:

```
PLAY TONE left, riupt
```

left and *right* are the frequencies in Hz to use for the left and right channels. The tone plays in the background (the program will continue running after this command) and 'dur' specifies the number of milliseconds that the tone will sound for.

duration is optional and if not specified the tone will continue until explicitly stopped or the program terminates. *interrupt* (if specified) will be triggered when the duration has finished.

The frequency can be from 1 Hz to 20 KHz and is very accurate. The frequency can be changed at any time by issuing a new PLAY TONE command. The sine wave generated by the PC is generally clean although some noise is visible – a low pass filter will help to *clean up* the sine wave signal.

Specialised Audio Output

There are a number of specialised audio commands that are mostly used in computer games.

These are:

- PLAY MODFILE which will play synthesised music using the MOD format.
- PLAY SOUND which will generate an output based on a mixture of sine, square, noise, etc waveforms.
- PLAY EFFECT command which will play a WAV at the same time as a MOD file is playing.

Using PLAY

It is important to realise that the PLAY command will generate the audio in the background. This allows a program (for example) to play the sound of an explosion while still animating the visual of the explosion on the screen. Without the background facility the whole computer would freeze while the sound was heard.

However, generating the audio in the background has some subtle inferences which can trip up newcomers. For example, take the following program:

```
PLAY TONE 500, 500, 2000
END
```

You may expect the 500Hz tone to sound for 2 seconds but in practice it will not make any sound at all. This is because MMBasic will execute the PLAY TONE command (which will start generating the sound in the background) and then it will immediately continue and execute the END command which will terminate the program and the background sound. This happens so fast that nothing is heard.

Similarly the following program will not work either:

```
PLAY TONE 500, 500, 2000
PLAY TONE 300, 300, 5000
```

This is because the first command will set a 500Hz the tone playing but then the second PLAY command will immediately replace that with a 300Hz tone and following that the program will run off the end terminating the program and the background audio resulting in nothing being heard.

If you want MMBasic to wait while the PLAY command is doing its thing you should use suitable PAUSE commands. For example:

```
PLAY TONE 500, 500
PAUSE 2000
PLAY TONE 300, 300
PAUSE 5000
```

This applies to all versions of the PLAY command (eg, PLAY WAV/MP3/FLAC, etc).

Utility Commands

There are a number of commands that can be used to manage the sound output:

- PLAY PAUSE Temporarily halt (pause) the currently playing file or tone.
- PLAY RESUME Resume playing a file or tone that was previously paused.
- PLAY STOP Terminate the playing of the file or tone. The sound output will also be automatically stopped when the program ends.
- PLAY VOLUME L, R Set the volume to between 0 and 100 with 100 being the maximum volume. The volume will reset to the maximum level when a program is run.

The following commands can be used when playing a sequence of files (ie, "background music"):

- PLAY NEXT Skip to the next file.
- PLAY PREVIOUS Skip to the previous file.

Interrupts

Interrupts are a handy way of dealing with an event that can occur at an unpredictable time. Examples are timers or communications ports. In your program you could insert code after each statement to check to see if the timer has timed out or a character received on the serial port but an interrupt makes for a cleaner and more readable program.

When an interrupt occurs MMBasic will execute a special section of code and when finished return to the main program. The main program is completely unaware of the interrupt and will carry on as normal.

Return from an interrupt is via the END SUB or EXIT SUB commands. Note that no parameters can be passed to the subroutine however within the interrupt subroutine calls to other subroutines are allowed.

If two or more interrupts occur at the same time they will be processed in order of the interrupts as defined. During the processing of an interrupt all other interrupts are disabled until the interrupt subroutine returns.

Interrupts can occur at any time but they are disabled during INPUT statements. When using interrupts, the main program is completely unaffected by the interrupt activity unless a variable used by the main program is changed during the interrupt.

Because interrupts run in the background they can cause difficult to diagnose bugs. Keep in mind the following factors when using interrupts:

- Interrupts are only checked by MMBasic at the completion of each command, and they are not latched by the hardware. This means that an interrupt that lasts for a short time can be missed, especially when the program is executing commands that take some time to execute.
- When inside an interrupt all other interrupts are blocked so your interrupts should be short and exit as soon as possible. For example, never use PAUSE inside an interrupt. If you have some lengthy processing to do you should simply set a flag and immediately exit the interrupt, then your main program loop can detect the flag and do whatever is required.
- The subroutine that the interrupt calls (and any other subroutines called by it) should always be exclusive to the interrupt. If you must call a subroutine that is also used by an interrupt you must disable the interrupt first (you can reinstate it after you have finished with the subroutine).
- Remember to disable an interrupt when you have finished needing it – background interrupts can cause strange and non-intuitive bugs.

Game Playing Features

MMBasic for Windows has many features designed to help programmers in writing computer games. One of the most important features of this version is its speed (approx 10 times faster than the Colour Maximite) and its large program memory. These alone make it possible to write large and complex programs without the issues of optimising the program for speed or space.

Other useful features include managing the video output and drawing movable images, a selection of text drawing methods and fonts, displaying images, playing audio and getting input from a keyboard or games controller.

Screen Resolution, Colour Depth and Pages

The resolution of the MMBasic window is controlled by the MODE command. With this you can select various resolutions from 1280x720 to 240x216. The default startup resolution is MODE 9 - 1024 x 768.

All MODES are ARGB8888 and offer 256 levels of intensity for red, green and blue respectively plus 256 levels of transparency. For red, green and blue, 1 is no or zero intensity, 255 is full intensity. For the alpha channel, 0 is fully transparent while 255 is fully opaque (ie. zero transparency).

All modes have provision for two image layers plus a background layer. The background layer can only be set by the MODE command and is the lowest layer. The next two layers are image layers and are generally referred to as PAGE 0 and PAGE 1. Each of the two image layers – PAGE 0 and PAGE 1 have both a foreground and background colour setting. If either of the foreground or background colours have transparency level set, the layer below will 'show through" at a level determined by the alpha value (0-full transparency to 255- totally opaque). Images on the top layer (PAGE 1) will cover or overlay the lower layer (PAGE 0) and PAGE 0 images will cover or overlay the background except where the alpha channel is set to allow transparency thereby allowing various degrees of the lower layer and/or background to show through.

It is important to note that PAGE 0 is the ONLY page displayed! Both PAGE 1 and the background layer are "flattened' or 'merged' into PAGE 0 'on-the'fly' during display, roughly 60 times a second, as determined by the alpha channel.

In addition there are many extra video pages that are available to the programmer for building images. These pages can then be copied at high speed during the video blanking period to the main display page providing an instantaneous display update without any tearing artefacts. This is managed by the PAGE WRITE command which specifies the video page to be used for the output of subsequent graphics commands and the PAGE COPY command which will copy one page to another.

When drawing to the main page being displayed on the monitor (PAGE 0 only remember!), the programmer can use the GETSCANLINE function to report on the line that is currently being drawn on the display. Using this to time updates to the screen can avoid screen glitches caused by updates while the screen is being updated.

Scrolling and Sprites

The PAGE SCROLL command will scroll a page horizontally or vertically by a specified number of pixels allowing the games programmer to create a smoothly scrolling background for platform games and the like.

MMBasic for Windows has extensive support for sprites which is an image that can be moved over the background without disturbing the background. This feature is far more sophisticated than that available on the original Colour Maximite. The sprite can be a PNG image or an image defined in a text file and can be of any size up to the video horizontal and vertical resolution. The transparency colour can be either black or defined in the image depending on the colour depth specified by the MODE command.

Multiple sprites can be loaded and they can be moved around the screen, hidden or displayed as a group or individually. MMBasic also includes a versatile mechanism for detecting when two sprites collide allowing (for example) the programmer to create a realistic bounce effect.

Displaying Images

MMBasic for Windows can load and display images stored on the file system in a variety of formats including BMP, GIF, PNG and JPG. These are read from the file system by the LOAD command and can be of any size and positioned anywhere on the screen. In addition on-screen images can be scaled and rotated under control of the program.

Fonts

Built in to MMBasic is support for seven fonts ranging from small to large. In addition user supported fonts can be defined by the BASIC program or dynamically loaded from the file system. All fonts can be displayed in various colours, scaled and rotated.

In games that use the keyboard for input the user will often hold down a number of keys simultaneously. This condition can be detected using the KEYDOWN() function. This will return the number of keys held down and their values in the order that they were depressed. MMBasic will track up to six simultaneous key depressions.

BLIT Command

If the display is capable of transparent text the BLIT command allows a portion of the image currently showing on the display to be copied to a memory buffer and later copied back to the display. This is useful when something needs to be drawn over the background and later removed without damaging the image in the background. Examples include a game where a character is moving about in front of a landscape or the moving needle of a photorealistic gauge.
The available commands are:

```
BLIT READ #b, x, y, w, h
BLIT WRITE #b, x, y, w, h
BLIT LOAD #b, f$, x, y, w, h
BLIT CLOSE #b
```

#b is the buffer number in the range of 1 to 32. x and y are the coordinates of the top left corner and w and h are the width and height of the image. READ will copy the display image to the buffer, WRITE will copy the buffer to the display and CLOSE will free up the buffer and reclaim the memory used. LOAD will load an image file into the buffer.
BLIT LOAD, BLIT WRITE, BLIT and BLIT READ commands can be used to copy a portion of the display to another location (by copying to a buffer then writing somewhere else) but a simpler method is to use an alternative version of the BLIT command as follows:

```
BLIT x1, y1, x2, y2, w, h
```

This will copy a portion of the image at x1/y1 to the location x2/y2. w and h specify the width and height of the image to be copied. The source and destination areas can overlap and the BLIT command will perform the copy correctly. This form of the BLIT command is particularly useful for creating graphs that can scroll horizontally or vertically as new data is added.

GAMEPAD Use

Support has been added for GAMEPADs. Usage may vary between different models/makes of gamepads – please feed back any issues to TheBackShed forum.

GAMEPAD ON [interrupt] [,MASK] ' mask specifies which buttons should trigger an interrupt - defaults to all
GAMEPAD OFF
GAMEPAD VIBRATE n ' (n=1 to 100)
GAMEPAD STOP 'Stops vibration

NB: the vibrate commands appear to be coded correctly but don't work on my cheap second-hand DualShock4. The code including vibrate isn't tested on a XBox controller which is supposed to be supported but I don't have one.
After the GAMEPAD ON command the first button pressed on the controller activates the system and will report that the controller is connected if all is well. The command can be issued at the command line.

NB: The gamepad must be connected to the computer when MMBasic is started

A new GAMEPAD function has been added -

GAMEPAD(func)

func is the same as the PicoMite and CMM2 i.e. LX, LY, RX, RY for the joysticks and L and R for the analogue buttons. B returns the digital buttons as a bitmap identical to the Wii Classic on the CMM2 and PicoMite. See the example code below:-

```
gamepad on myint
Pause 1000
Do
Loop

Sub myint
  Static integer l%=0
  m%=1
  a%=gamepad(b)
  t%=l% Xor a%
  If t% And m% Then
    If a% And m% Then : Print "R_ON":Else : Print "R_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "Start_ON":Else : Print "Start_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "Home_ON":Else : Print "Home_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "Select_ON":Else : Print "Select_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "L_ON":Else : Print "L_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "DOWN_ON":Else : Print "DOWN_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "RIGHT_ON":Else : Print "RIGHT_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "UP_ON":Else : Print "UP_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "LEFT_ON":Else : Print "LEFT_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "ZR_ON":Else : Print "ZR_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "X_ON":Else : Print "X_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "A_ON":Else : Print "A_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "Y_ON":Else : Print "Y_OFF":EndIf
```

```
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "B_ON":Else : Print "B_OFF":EndIf
  EndIf
  Inc m%,m%
  If t% And m% Then
    If a% And m% Then : Print "ZL_ON":Else : Print "ZL_OFF":EndIf
  EndIf
  l%=a%
End Sub
```

Porting Programs

This chapter covers some of the considerations involved in porting programs from the original Colour Maximite and the Colour Maximite 2. There is a high degree of backwards compatibility in MMBasic for Windows and most programs will run with little effort, however, as can be expected, there are some differences that need to be addressed.

Most of these differences involve the more specialised functions such as graphics, input/output and some functions like the random number generator. Note that not all differences are listed here, just the more important ones that are likely to cause problems when porting programs.

Variables

In the original Colour Maximite it was possible to define variables with the same name but using a different type. For example, it was possible to use the following: `v=3.4512` and `v$="abcdefg"` in the same program (ie, the variable was defined as both a float and a string).

This is not allowed in MMBasic for Windows, variables must be unique regardless of their type.

Floating Point

In MMBasic for Windows floating point is double precision. This means that if a number is printed without formatting it will contain more significant digits than the same number on the original Colour Maximite. Generally, this will not cause an issue but it might mess up numbers that need to be printed in neat columns.

Graphic Commands

The syntax of the graphic commands PIXEL, LINE, BOX and CIRCLE have completely changed.

The syntax of the MODE command has changed.

The Scan Line Colour Override function is not supported in MMBasic for Windows.

Fonts

The FONT command has a completely different purpose and syntax in MMBasic for Windows.

MMBasic for Windows has the ability to load fonts using the LOAD FONT command and you can convert the original Colour Maximite's font files to this format using the program FontTweak from: https://www.c-com.com.au/MMedit.htm

BLIT

The BLIT command is compatible with the exception that it does not implement the optional parameter specifying which colour planes to copy.

Sprites

As with the original Colour Maximite sprites can be loaded from a Maximite sprite file using the command:

```
SPRITE LOAD filename$ [,start_sprite_number]
```

This is compatible with the original Colour Maximite however MMBasic for Windows has a few improvements:

- An optional third parameter is available in the first line of the sprite file in which case the first parameter becomes the width and the third parameter the height. Both width and height can be set to any size.
- If only two parameters are specified the first parameter sets both the width and height and can be a number other than 16.
- Multiple sprite files can be loaded by specifying the optional 'start_sprite_number' to ensure that the sprites from different files do not overlap. This parameter defaults to 1 if not specified.

A detailed description of sprites and their usage can be found in the CMM2 User Manual. See the *commands* and *functions* sections for listings of all the sprite instructions.

The following example shows a red mouse pointer. The first line of the file holds the parameters for the sprite (ie, the width, the number of sprites in the file, and the height in that order).

```
13,1,19
4
44
4 4
4  4
4   4
4    4
4     4
4      4
4       4
4        4
4         4
4          4
4       444444
4   4  4
4  44  4
4 4  4  4
44   4  4
       4  4
       4444
```

The above sprite can be saved as "mouse.spr". Then the following commands will load and show it:

```
SPRITE LOAD "mouse.spr", 5
SPRITE SHOW 5, 100, 100, 1
```

Note that a space character is interpreted as BLACK transparent BLACK. A “0” is interpreted as NOTBLACK solid BLACK.

SOUND and TONE

The TONE command has been replaced by PLAY TONE.

File Related Commands

All file system storage is as for MS Windows.

The command CHAIN is not implemented. MMBasic for Windows has an exceptionally large program memory space so this feature is not necessary. The library and the LIBRARY commands are not implemented and not needed on MMBasic for Windows. Instead of using these commands you can use #INCLUDE to insert files into the program at run time.

The commands PLAYMOD, LOADBMP and SAVEBMP have changed names. See PLAY MODFILE, LOAD BMP and SAVE IMAGE.

The DRIVE command and the predefined read-only variable MM.DRIVE$ are not supported.

Special Devices

LCD displays, Real Time Clocks (RTC) and keypads are not supported in MMBasic for Windows.

CONFIG Commands

Relevant CONFIG commands in the original Colour Maximite have been converted to OPTION commands where applicable.

Error Handling

Error handling using the OPTION ERROR commands has changed (see ON ERROR).

Touch

Touch is emulated by the cursor on the MMBasic screen and the mouse. See the MOUSE commands and functions for detail.

Random Number Generator

MMBasic for Windows has an advanced random number generator based on the internal MS Windows function that generates a sequence of true random numbers which will never repeat. For this reason MMBasic for Windows does not require (or allow) the programmer to seed the random generator to get different sequences.

The original Colour Maximite generated a pseudo random number sequence that always repeated with the same seed and in some rare cases this is what the programmer requires. If you need this behaviour you can use the following to generate a repeatable set of random numbers:

```
function pseudo() as float
  static seed%=7
  static a%=1103515245, c%=12345, m%=2^31
  seed%=(a% * seed% + c%) mod m%
  pseudo = seed%/m%
end function
```

Change the assignment to seed% to change the seeding number.

MMBasic Implementation Characteristics

- Maximum program size (as plain text) is 1MB. Note that MMBasic tokenises the program when it is stored in program memory so the final size in program memory might vary from the plain text size
- Maximum length of a command line is 255 characters
- Maximum length of a variable name or a label is 31 characters
- Maximum number of variables 1024: 512 global and 512 local
- Maximum number of dimensions to an array is 5
- Maximum number of arguments to commands that accept a variable number of arguments is 32
- Maximum number of nested FOR…NEXT loops is 128
- Maximum number of nested DO…LOOP commands is 128
- Maximum number of nested GOSUBs, subroutines and functions (combined) is 512
- Maximum number of nested multiline IF…ELSE…ENDIF commands is 128
- Maximum number of SELECT CASE statements is unlimited
- Maximum number of user defined subroutines and functions (combined): 500
- The range of floating point numbers is 1.797693134862316e+308 to 2.225073858507201e-308
- The range of 64-bit integers (whole numbers) that can be manipulated is ±9223372036854775807
- Maximum normal string length is 255 characters (longstrings only limited by memory)
- Maximum line number is 65000
- Maximum number of sprites is 64 (#1 to #64)
- Maximum number of sprite collisions is 8
- Maximum length of a line in a program is 240 characters

Predefined Read Only Variables

These variables are set by MMBasic and cannot be changed by the running program. They can be accessed from the command line with the PRINT command, eg.

```
PRINT MM.DEVICE$  ' this will output to the screen the current platform/version of
MMBasic
MMBasic for Windows
>
```

They can also be used in a program as a function; that is, they will return a value to be used as desired, eg.

```
PRINT "Demo using pre-defined Read Only variables"
PRINT MM.INFO(VPOS) ' will print the vertical position of the cursor
a$ = MM.DEVICE$
PRINT a$  ' will print out MMBasic for Windows
```

Detailed Listing

MM.CMDLINE$	A string representing the arguments on the command line when the program was run.
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. "Colour Maximite 2" on the Colour Maximite 2. "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "MMBasic for Windows" when running in Windows 10 or later "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.
MM.HRES MM.VRES	Integers representing the horizontal and vertical resolution of the MMBasic window in pixels.
MM.INFO() MM.INFO$()	These two versions can be used interchangeably but good programming practice would require that you use the one corresponding to the returned datatype.
MM.INFO(BCOLOUR)	Returns the current background colour
MM.INFO$(CPUSPEED)	Returns the CPU speed as a string.
MM.INFO$(CURRENT)	Returns the name of the current program or NONE if called after a NEW command.
MM.INFO$(DIRECTORY)	Returns the current working directory. This will always end with a '\ character.
MM.INFO(DEFAULT PATH)	Returns the full path to the current working directory ending with a '\ character.
MM.INFO$(ENVVAR name$)	Will return a Windows environment variable that has been set from within Windows. Refer to Windows SYSTEM, SETTINGS, ADVANCED SYSTEM SETTINGS

MM.INFO(EXISTS FILE filename$	Returns 1 if the file filename$ exists otherwise returns 0.
MM.INFO(EXISTS DIR dirname$)	Returns 1 if the directory dirname$ exists otherwise returns 0.
MM.INFO(FCOLOUR)	Returns the current foreground colour
MM.INFO(FILESIZE file$)	Returns the size of 'file$' in bytes. Returns -1 if the file is not found. Returns -2 if file$ is the name of a valid directory
MM.INFO(FONT ADDRESS n)	Returns the address of the memory location containing the address of FONT n
MM.INFO(FONT POINTER n)	Returns a POINTER to the start of FONT n in memory
MM.INFO(FONTHEIGHT)	Integers representing the height and width of the current font (in pixels).
MM.INFO(FONTWIDTH)	
MM.INFO(FRAMEBUFFER)	Returns the physical memory location of the framebuffer. This is useful if you need to POKE/PEEK the contents of the page.
MM.INFO(FRAMEH)	Returns the horizontal size of the frame buffer in pixels.
MM.INFO(FRAMEV)	Returns the vertical size of the framebuffer in pixels
MM.INFO(HPOS)	The current horizontal and vertical position (in pixels) following the last graphics or print command.
MM.INFO(VPOS)	
MM.INFO(MAX PAGES)	Returns the maximum page number that can be selected in the current graphics mode.
MM.INFO(OPTION option)	Returns the current value of a range of options that affect how a program will run. "option" can be one of ANGLE, AUTORUN, BASE, BREAK, CONSOLE, DEFAULT, EXPLICIT, Y_AXIS, SEARCH PATH,
MM.INFO(PAGE ADDRESS n)	Returns the physical memory location of page 'n'. This is useful if you need to POKE/PEEK the contents of the page.
MM.INFO(PATH)	Returns the path of the current program or NONE if called after a NEW command
MM.INFO(PROGRAM)	Returns the address in memory of the start of the program
MM.INFO$(SOUND)	Returns the status of the sound output device. Valid returns are: OFF, PAUSED, TONE, WAV, MP3, MODFILE, FLAC, SOUND
MM.INFO$(TRACK)	The name of the current audio track playing. This returns "OFF" if nothing is playing.
MM.INFO(VERSION)	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.INFO(WRITE PAGE)	Returns the address in memory of the page to which writes will take place
MM.WATCHDOG	An integer which is true (ie, 1) if MMBasic was restarted as the result of a Watchdog timeout (see the WATCHDOG command). False (ie, 0) if MMBasic started up normally.

Operators

The following operators are listed in order of precedence. 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.

Numeric Operators (Float or Integer)

NOT INV	NOT will invert the logical value on the right. INV will perform a bitwise inversion of the value on the right. Both of these have the highest precedence so if the value being operated on is an expression it should be surrounded by brackets. For example, `IF NOT (A = 3 OR A = 8) THEN …`
^	Exponentiation (eg, `b^n` means b^n)
* / \ MOD	Multiplication, division, integer division and modulus (remainder)
+ -	Addition and subtraction
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. For a left shift any bits introduced are set to zero. >> is an unsigned right shift where the top bit is set to 0 >>> is a signed right shift and any bits introduced are set to the value of the top bit (bit 63).
<> < > <= =< >= =>	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 (also used in assignment to a variable, eg implied LET).
AND OR XOR	Conjunction, disjunction, exclusive or. These are bitwise operators and can be used on 64-bit unsigned integers.

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.

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 the case of the characters (ie "A" is greater than "a").

MMBasic Configuration Options

This table lists the various option commands which can be used to configure MMBasic and change the way it operates. Options that are marked as permanent will be saved in non volatile memory and automatically restored when MMBasic for Windows is restarted. Options that are not permanent will be reset on startup.

	Permanent	
OPTION ANGLE RADIANS | DEGREES		This command switches trig functions between degrees and radians. Acts on SIN, COS, TAN, ATN, ATAN2, MATH ATAN3, ACOS, ASIN This is a temporary option that is cleared to default (RADIANS) when programs end, after an error, or after Ctrl-C so should be set at the top of any program that requires to use angles in degrees.
OPTION AUTORUN OFF | ON	✔	Instructs MMBasic to automatically run the program that was previously loaded on power up or restart (eg, by the WATCHDOG timer). This is turned off by the NEW command but other commands that might change program memory (EDIT, etc) do not change this setting. Entering the break key (default CTRL-C) at the console will interrupt the running program and return to the command prompt.
OPTION Y_AXIS DOWN | UP		This command can only be used in a program and inverts the y axis for drawing commands. Ie, with UP set 0,0 is at the bottom left as in a typical graphing application. You can not use PRINT sensibly when set to UP – use TEXT to place characters on the screen. This is a temporary option that is cleared to default (DOWN) when programs end, after an error, or after Ctrl-C.
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 BREAK nn		Set the value of the break key to the ASCII value 'nn'. This key is used to interrupt a running program. The value of the break key is set to CTRL-C key at power up but it can be changed to any keyboard key using this command (for example, OPTION BREAK 4 will set the break key to the CTRL-D key). Setting this option to zero will disable the break function entirely.
OPTION COLOURCODE ON | OFF	✔	Turn on or off colour coding for the editor's output. Keywords will be in cyan, comments in yellow, etc. The default is ON.
OPTION CONSOLE SCREEN or OPTION CONSOLE SERIAL or OPTION CONSOLE BOTH		This is the default value and will direct all output (both GUI and text) to the open MMBasic window whether in command line mode or running a program. OPTION CONSOLE SERIAL will disable text output to the main MMBasic window. All text will be redirected to the Command Line Console window. This is useful for debugging graphics applications as diagnostic PRINT statements will not corrupt the screen display. This can be only enabled in a program and is reverted to the previous value when the program ends. Note: This has NOTHING to do with any serial ports. OPTION CONSOLE BOTH will enable both the command line console window and current MMBasic widow for console output of text. GUI goes only to the MMBasic window. This is the default on power up or RESTART. A big advantage to enabling the command line console window is that it is 'scrollable' thus enabling viewing of past commands/error msgs.

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 DEFAULT MODE n	✔	Specifies the video mode for the command prompt, the editor and the file manager and is the default mode when a program is run. n can be: 1 = 800x600 (default) 8 = 640x480 9 = 1024x768 10 = 848x480 (widescreen) 11 = 1280x720 (widescreen) 12 = 960x540 (widescreen) 14 = 960x540 (widescreen 15 = 1280x1024 16 = 1920x1080 (widescreen)
OPTION EXPLICIT		Placing this command at the start of a program will require that every variable be explicitly declared using the DIM, LOCAL or STATIC commands 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 ESCAPE ON | OFF		Enable escape sequence processing in string data. Escape sequence / hex value / ASCII character/code represented \a 07 Alert (Beep, Bell) \b 08 Backspace \e 1B Escape character \f 0C Formfeed Page Break \n 0A Newline (Line Feed) \r 0D Carriage Return \q 22 Quote symbol \t 09 Horizontal Tab \v 0B Vertical Tab \\ 5C Backslash \nnn any The byte whose numerical value is given by nnn – interpreted as a decimal number \&hh any The byte whose numerical value is given by hh – interpreted as a hexadecimal number Can be used at the command prompt but like OPTION EXPLICIT it will be disabled again by the RUN or NEW commands so will need to be included in a program if required.
OPTION FNKey string$	✔	Define the string that will be generated when a function key is pressed at the command prompt. FNKey can be F5 through F12. Example: OPTION F8 “RUN”+chr$(34)+”myprog”+chr$(34)+chr$(13)+chr$(10) This command must be run at the command prompt (not in a program).
OPTION KEYBOARD lang, repeatstart, repeatrate	✔	Where lang is one of UK(default), US, DE, BE, GR,IT, ES, SW, repeatstart is the delay to commence repeat and repeatrate is the delay between repeats, both in milliseconds.
OPTION LIST		This will list the settings of any options that have been changed from their default setting and are the permanent type.
OPTION MILLISECONDS ON | OFF		Specifies that the TIME$ function will, or will not, include milliseconds as a decimal fraction of seconds in its output. The default is OFF.

OPTION RESET	✔	Reset all saved options to their default values.
OPTION SEARCH PATH pathname$	✔	This defines a path which will be searched when you use the existing RUN command or the short form RUN command (*) if the file does not exist in the current directory
OPTION TAB 2 | 3 | 4 | 8	✔	Set the spacing for the tab key. Default is 2.

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.
* (asterix)	Synonym for the RUN command at the command prompt e.g. *myprog any test string Will run the program myprog.bas and pass it the command line “any test string” in MM.CMDLINE$
#COMMENT START #COMMENT END	Directive to allow multi-line comments. The command must be in capitals. Any lines between the two commands are completely ignored and not loaded into memory
#DEFINE “before”, “after”	This will cause all copies of the string “before” in a program to be replaced by the string “after”. Both parameters must be literal quoted strings. Matches within quoted strings in the program are ignored. DEFINEs are executed in reverse order of creation so a symbol can be redefined and from that point on in the program the new definition will be active. Case is ignored in the strings in the DEFINE directive. The program can support up to 256 #define statements.
#INCLUDE file$	This will insert the file 'file$' into the program at that point. This file must be resident on the file system and must have the extension ".INC". Inserting the text is performed by the pre-processor when the program is loaded into program memory by the RUN command or on exiting EDIT or AUTOSAVE using F2. Because this operation is performed before the program is run it is recommended that include files are specified relative to the directory holding the program or with full pathnames. Within the program the command CHDIR will be executed at runtime so will not affect MMBasic's ability to locate include files. This command acts exactly as if the included file was manually inserted into the code using an editor but it is more convenient for loading libraries and other static code fragments. It essentially replaces the LIBRARY command on the original Maximite. Runtime errors in the included file are reported with the file name and line number in the file. The firmware will automatically check for changes in include files when a program is RUN and update the program if required.
ARC x, y, r1, [r2], rad1, rad2, colour	Draws an arc of a circle or a given colour and width between two radials (defined in degrees). Parameters for the ARC command are: 'x' is the X coordinate of the centre of arc. 'y' is the Y coordinate of the centre of arc. 'r1' is the inner radius of the arc. 'r2' is the outer radius of the arc - can be omitted if 1 pixel wide. 'rad1' is the start radial of the arc in degrees. 'rad2' is the end radial of the arc in degrees. 'colour' is the colour of the arc.

AUTOSAVE file$	Enter automatic program entry mode. This command will take lines of text from the console serial input and save them to a file on the file system specified as 'file$'. This mode is terminated by pressing F1 on the console keyboard which will then cause the received data to be saved to the file system. Terminating the transfer by pressing F2 will cause a similar save but then the saved program will be immediately loaded into program memory and run. Both F1 and F2 update the "current program name" which is used by RUN, LIST and EDIT when a file is not specified. The transfer can also be terminated using F6 which acts the same as F1 without updating the current program name. At any time this command can be aborted by Control-C which will leave program memory untouched. You can also transfer a program into tMMBasic by pasting directly into the file from the clipboard. This is done with CTRL+v which takes previously saved text from the clipboard directly into the opened file. It needs to be terminated with CTRL+z to complete the transfer and close the file.
BLIT READ [#]b, x, y, w, h [,pagenumber] or BLIT WRITE [#]b, x, y [,orientation] or BLIT CLOSE [#]b	Copy one section of the display screen to or from a memory buffer. BLIT READ will copy a portion of the display to the memory buffer '#b'. The source coordinate is 'x' and 'y' and the width of the display area to copy is 'w' and the height is 'h'. When this command is used the memory buffer is automatically created and sufficient memory allocated. The optional parameter page number specifies which page is to be read. The default is the current write page. This buffer can be freed and the memory recovered with the BLIT CLOSE command. Set the pagenumber to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command BLIT WRITE will copy the memory buffer '#b' to the display. The destination coordinate is 'x' and 'y' using the width/height of the buffer. The optional 'orientation' parameter defaults to 4 and specifies how the stored image data is changed as it is written out. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels BLIT CLOSE will close the memory buffer '#b' to allow it to be used for another BLIT READ operation and recover the memory used. Notes: • Sixty four buffers are available ranging from #1 to #64. • When specifying the buffer number the # symbol is optional. • All other arguments are in pixels.
BLIT x1, y1, x2, y2, w, h [, page] [,orientation]	Copy one section of the display screen to another part of the display. The source coordinate is 'x1' and 'y1'. The destination coordinate is 'x2' and 'y2'. The width of the screen area to copy is 'w' and the height is 'h'. 'page' is the page number that the image data is read from; it is then written to the current write page as specified by the PAGE WRITE n command. If 'page' is omitted the data is read from the write page. Set the page to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command. All arguments are in pixels and the source and destination can overlap. The optional 'orientation' parameter specifies how the section of the screen is changed as it is copied. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels

BOX x, y, w, h [, lw] [,c] [,fill]	Draws a box on the MMBasic window with the top left hand corner at 'x' and 'y' with a width of 'w' pixels and a height of 'h' pixels. 'lw' is the width of the sides of the box and can be zero. It defaults to 1. 'c' is the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'c', and fill can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
BOX AND_PIXELS x, y, w, h, colour [,pageno] BOX OR_PIXELS x, y, w, h, colour [,pageno] BOX XOR_PIXELS x, y, w, h, colour [,pageno]	Executes the requested logical operation between the pixels in the area defined on the page specified (defaults to the write page) with the colour specified
CALL usersubname$ [,usersubparameters,....]	This is an efficient way of programmatically calling user defined subroutines (see also the CALL() function). In many case it can allow you to get rid of complex SELECT and IF THEN ELSEIF ENDIF clauses and is processed in a much more efficient way. The “usersubname$” can be any string or variable or function that resolves to the name of a normal user subroutine (not an in-built command). The “usersubparameters” are the same parameters that would be used to call the subroutine directly. A typical use could be writing any sort of emulator where one of a large number of subroutines should be called depending on some variable. It also allows a way of passing a subroutine name to another subroutine or function as a variable.
CAT S$, N$	Concatenates the strings by appending N$ to S$. This is functionally the same a S$ = S$ + N$ but operates faster.
CHDIR dir$	Change the current working directory on the file system to ‘dir$’ The special entry “..” represents the parent of the current directory and “.” represents the current directory. "/" is the root directory.
CIRCLE x, y, r [,lw] [, a] [, c] [, fill]	Draw a circle on the video output centred at 'x' and 'y' with a radius of 'r' on the MMBasic window ‘lw’ is optional and is the line width (defaults to 1). 'c' is the optional colour and defaults to the current foreground colour if not specified. The optional 'a' is a floating point number which will define the aspect ratio. If the aspect is not specified the default is 1.0 which gives a standard circle 'fill' is the fill colour. It can be omitted or set to -1 in which case the circle will not be filled. All parameters can be expressed as arrays and the software will plot the number of circles as determined by the dimensions of the smallest array. 'x', 'y' and 'r' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'a', 'c', and fill can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
CLEAR	Delete all variables and recover the memory used by them.
CLOSE [#]nbr [,[#]nbr] …	Close the file(s) previously opened with the file number ‘#fnbr’ Close the serial communications port(s) previously opened with the file number ‘nbr’. The # is optional. See Appendix A for caveats using serial I/O, also see the OPEN command. The text “GPS” can be substituted for [#]nbr to close a communications port used for a GPS receiver.

CLS [colour]	Clears the current active PAGE. Optionally 'colour' can be specified using RGB(n,n,n) which will be used for the current active PAGE background when clearing the screen. If transparency has been enabled via the MODE command, colour can be RGB(BLANK) which sets the background of the current active PAGE to fully transparent.
COLOUR fore [, back] or COLOR fore [, back]	Sets the default colour for commands (PRINT, etc) that display on the on the MMBasic window. 'fore' is the foreground colour, 'back' is the background colour. The background is optional and if not specified will default to black.
CONSOLE string$	This is the same as print but puts output to the Command Line Console window - perfect for debugging
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 $) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.
CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The command is also used to exit a BREAK state and resume program execution following a MMDEBUG BREAK command. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with graphics, music, nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.
COPY fname1$ TO fname2$	Copy a file from ‘fname1$’ to ‘fname2$’. Both are strings. A directory path can be used in both 'fname$' and 'fname$'. If the paths differ the file specified in 'fname$' will be copied to the path specified in 'fname2$' with the file name as specified.
CTRLVAL(#ref) =	This command will set the value of an advanced control. '#ref' is the control's reference number. For off/on controls like check boxes it will override any touch input and can be used to depress/release switches, tick/untick check boxes, etc. A value of zero is off or unchecked and non-zero will turn the control on. For a LED it will cause the LED to be illuminated or turned off. It can also be used to set the initial value of spin boxes, text boxes, etc. For example: CTRLVAL(#10) = 12.4 All controls expect to be assigned a number (float or integer) except Frame, Caption, Display Box, Text Box and Format Box which expect a string.
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.
DEFINEFONT #n hex [[hex[…] hex [[hex[…] END DEFINEFONT	This will define an embedded font which can be used exactly same as the built in fonts (ie, selected using the FONT command or specified in the TEXT command). MMBasic must execute the font in order for it to be loaded. '#n' is the font's reference number (1 to 16). It can be the same as an existing font (except fonts 1, 6 and 7) and in that case it will replace that font. Each 'hex' must be exactly eight hex digits and be separated by spaces or new lines from the next. Multiple lines of 'hex' words can be used with the command terminated by a matching END DEFINEFONT.

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 up to five dimensions). Note that this is different from the original Colour Maximite and Micromite versions of MMBasic which supported up to eight 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 on the Colour Maximite 2 requires 8 bytes a total of 1848 bytes of memory will be allocated.
Strings will default to allocating 255 bytes (ie, characters) of memory for each element and this can quickly use up memory when defining arrays of strings. 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 unless the length is less than 16 when it will both save memory and improve performance.
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. Also note that the initialising values must be after the LENGTH qualifier (if used) and after the type declaration (if used).
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, not 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.
DRAW3D	V5.07.03b15 and later includes a 3D engine. See the document “The CMM2 3D engine” for full details.
EDIT	Invoke the full screen editor. See the section *Full Screen Editor* for details of how to use the editor.
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.
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.

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

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.

EXECUTE command$	This executes the Basic command "command$". Use should be limited to basic commands that execute sequentially for example the GOTO statement will not work properly Things that are tested and work OK include GOSUB, Subroutine calls, other simple statements (like PRINT and simple assignments) Multiple statements separated by : are not allowed and will error The command sets an internal watchdog before executing the requested command and if control does not return to the command, like in a GOTO statement, the timer will expire. In this case you will get the message "Command timeout". RUN is a special case and will cancel the timer allowing you to use the command to chain programs if required.
EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.
FILES	Lists files in the current working directory 'fspec$' (if specified) can contain search wildcards. Question marks (?) will match any character and an asterisk (*) will match any number of characters. If omitted, all files will be listed. For example: * Find all entries *.TXT Find all entries with an extension of TXT E*.* Find all entries starting with E X?X.* Find all three letter file names starting and ending with X 'sort' specifies the sort order as follows: size by ascending size time by ascending time/date name by file name (default if not specified) type by file extension. Can be invoked at the command line by pressing F1.
FONT [#]font-number, scaling	This will set the default font for displaying text on the MMBasic window. Fonts are specified as a number. For example, #2 (the # is optional) See the chapter "Basic Graphics" for details of the available fonts. 'scaling' can range from 1 to 15 and will multiply the size of the pixels making the displayed character correspondingly wider and higher. Eg, a scale of 2 will double the height and width.
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' is 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.

FRAMEBUFFER	This command allows you to create, use and remove a variable size framebuffer which should make many applications which have a working area bigger than the screen easier to program. While using a framebuffer setting a different graphics mode which changes the colour depth will cause an error. JPG files cannot be loaded to the framebuffer and will error if tried. The framebuffer is deleted by Ctrl-C and by running a new program.
FRAMEBUFFER CREATE HorizontalSize%, VerticalSize%	This command creates a framebuffer with the width and height specified in pixels. HorizontalSize>=MM.HRES and <=1600: VerticalSize>=MM.VRES and <=1200
FRAMEBUFFER WRITE	This command sets all drawing commands to write to the framebuffer and inherit the width and height defined.
FRAMEBUFFER BACKUP	This command creates a backup copy of the framebuffer. If a backup already exists it is overwritten. This allows the programmer to save the background before he/she starts writing non-static data to it. NB: It won't be possible to use this command if a very large framebuffer is specified in 12 or 16-bit colour depth. A sensible error will be given in this case.
FRAMEBUFFER RESTORE [x, y, w, h]	This command restores all or part of the framebuffer from the backup. This allows the programmer to “clean” all or part of the framebuffer before adding new non-static items
FRAMEBUFFER WINDOW x, y, page [,I or B]	This command copies an area MM.HRES by MM.VRES from the framebuffer with top left at x,y to the page specified, The optional parameter specifies if the copy is Immediate or during frame Blanking
FRAMEBUFFER CLOSE	This command releases the memory resources used by the framebuffer and backup allowing a new framebuffer to be created with a different size

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$) or by specifying the type using AS <type> at the end of the functions definition.
	Eg. FUNCTION xxx (arg1, arg2) AS STRING
	'arg1', 'arg2', etc are the arguments or parameters to the function (the brackets are always required, even if there are no arguments). An array is specified by using empty brackets. ie, arg3().
	The type of the argument can be specified by using a type suffix (ie, arg1$) 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 50 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 the possibility of ruining your day.
GOTO target	Branches program execution to the target, which can be a line number or a label.
GUI	This is a full implementation of the GUI controls as popularised on the Micromite Plus. This is a suite of advanced graphic controls that respond to input from a mouse. These include on screen switches, buttons, indicator lights, keyboard, etc. The GUI controls have their own manual: *GUI Controls and Programming.pdf* Note: • A mouse must be connected and working for the GUI commands to work. • The number of controls allowed is defined using the OPTION MAXCTRLS n command where "n" is the maximum number that can be used to identify a control. By default this is set to zero so must be configured before GUI controls can be used (recommended is 100). Differences compared to the Micromite Plus implementation: • MMBasic for Windows uses the mouse as the user interface rather than touch. • The TOUCH function is renamed MOUSE(REF).
GUI CURSOR	The GUI CURSOR command provides a mechanism for displaying and manipulating a cursor on the screen. The cursor sits above all other graphics and nothing can overwrite it. BLIT, page copy, text, sprite, box etc. can all be used and the cursor will stay in view unless deliberately hidden. The cursor is always on PAGE 0 for colours 8 and 16 and page 1 for 12-bit colour and it can be moved even if the write page is somewhere else.
GUI CURSOR ON [cursorno [, x, y [,cursorcolour]]]	Cursor ON can be 0 (Default: mouse type pointer) or 1 (cross), in addition the user can load his own cursor using a SPRITE look-alike file in which case this is cursor no. 2 For cursor numbers 0 and 1 the programmer can override the default white cursor by specifying the colour in the open command.
GUI CURSOR x, y	Moves the cursor to x, y. Does not display the cursor if hidden but just updates the location.
GUI CURSOR OFF	Turns off the cursor
GUI CURSOR HIDE	Hides the cursor but maintains its position
GUI CURSOR SHOW	Shows a hidden cursor in its stored position
GUI CURSOR COLOUR cursorcolour	Changes the colour of cursor number 0 or 1. Does not impact loaded cursors where its colours are specified by the cursor designer
GUI CURSOR LOAD "fname"	Loads a user cursor from a file in the Maximite sprite format with a minor change. The header is now Width, Height, Xoffset, Yoffset. The two offsets determine where on the cursor the pointer is defined to be. So the mouse cursor has offsets 0,0 and the cross has offsets 7,7

GUI BITMAP x, y, bits [, width] [, height] [, scale] [, c] [, bc]	Displays the bits in a bitmap on the screen starting at 'x' and 'y' 'height' and 'width' are the dimensions of the bitmap as displayed on the screen and default to 8x8. 'scale' is optional and defaults to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. The bitmap can be an integer or a string variable or constant and is drawn using the first byte as the first bits of the top line (bit 7 first, then bit 6, etc) followed by the next byte, etc. When the top line has been filled the next line of the displayed bitmap will start with the next bit in the integer or string. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
IF expr THEN stmt [: stmt] or IF expr THEN stmt ELSE stmt	Evaluates the expression ‘expr' and performs the statement following the THEN keyword if it is true or skips to the next line if false. If there are more statements on the line (separated by colons :) they will also be executed if true or skipped if false. The ELSE keyword is optional and if present only one true statement is allowed following the THEN keyword. If 'expr' is resolved to be false the single statement following the ELSE keyword will be executed. The ‘THEN statement’ construct can be also replaced with: GOTO linenumber | label’. This type of IF statement is all on one line.

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.

IMAGE RESIZE x, y, width, height, new_x, new_y, new_width, new_height [,page_number] IMAGE RESIZE_FAST x, y, width, height, new_x, new_y, new_width, new_height [,page_number] [,flag]	This takes the part of the image with a top corner at 'x', 'y' and of specified 'width' and 'height' and resizes it writing it back to the area specified by 'new_x', 'new_y', 'new_width', new_height. The command will both increase and decrease the size of the part of the image chosen. It uses a bi-linear interpolation to generate the new pixels. The 'page number' is the page that the image data is read from it is then written to the current write page as specified by the PAGE WRITE n command. If 'page_number' is omitted the data is read from the write page. Use the text FRAMEBUFFER as the page no. to read from the framebuffer. IMAGE RESIZE uses bi-linear interpolation to resize the image. IMAGE RESIZE_FAST uses a nearest neighbour technique and is much faster but the resulting image quality will not be as good. If flag is set to 1 then black pixels are not written in the resized image.
IMAGE ROTATE x, y, width, height, new_x, new_y, angle! [,page_number] IMAGE ROTATE_FAST x, y, width, height, new_x, new_y, angle! [,page_number] [,flag]	Takes the part of the image with a top corner at 'x', 'y' and of specified 'width' and 'height' and rotates it about its centre in a clockwise direction by 'angle' (in degrees). Areas of the image that after rotation are outside of the area specified are cropped. The image is then drawn with the top left corner specified by new_x and new_y The 'page number' is the page that the image data is read from and it is written to the current write page as specified by the PAGE WRITE n command. If 'page_number' is omitted the data is read from the write page. Use the text FRAMEBUFFER as the page no. to read from the framebuffer. IMAGE ROTATE uses bi-linear interpolation to resize the image. IMAGE ROTATE_FAST uses a nearest neighbour technique and is much faster but the resulting image quality will not be as good. If flag is set to 1 then black pixels are not written in the rotated image.

IMAGE WARP_H x, y, w, h, x1, y1, h1, x2, y2, h2 [,readpage] [,flag] IMAGE WARP_V x, y, w, h, x1, y1, w1, x2, y2, w2 [,readpage] [,flag]	These commands allow you to modify an image by translating and/or stretching or compressing one of the axis. x, y, w, h define the top left coordinates and the width and height of the image to read from the current write page or optional read page In both cases x1 and x1 define the top left corner of the area to write In both cases x2 and y2 define the top right corner of the area to write When warping horizontally h1 and h2 define the height of the transformed area at the left edge and right edge When warping vertically w1 and w2 define the width of the transformed area at the top edge and bottom edge. If flag is set to 1 then black pixels are not written in the warped image.
INC var [,increment]	Increments the variable “var” by either 1 or, if specified, the value in increment. “increment” can have a negative. This is functionally the same as var = var + increment but is processed much faster
INPUT ["prompt$";] var1 [,var2 [, var3 [, etc]]]	Will take a list of values separated by commas (,) entered at the console and will assign them to a sequential list of variables. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 The list of variables can be a mix of float, integer or string variables. The values entered at the console must correspond to the type of variable. If a single value is entered a comma is not required (however that value cannot contain a comma). ‘prompt$’ is a string constant (not a variable or expression) and if specified it will be printed first. Normally the prompt is terminated with a semicolon (;) and in that case a question mark will be printed following the prompt. If the prompt is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #nbr, list of variables	Same as the normal INPUT command except that the input is read from a file previously opened for INPUT as ‘#fnbr’ or a serial port previously opened for INPUT as ‘nbr’. See the OPEN command. #0 can be used which refers to the console.
KILL file$	Deletes the file or empty directory specified by ‘file$’. If there is an extension it must be specified.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command. For example: Var = 56
LINE x1, y1, x2, y2 [, LW [, C]]	Draws a line starting at the coordinates ‘x1’ and ‘y1’ and ending at ‘x2’ and ‘y2’. ‘LW’ is the line’s width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal. ‘C’ is an integer representing the colour and defaults to the current foreground colour. All parameters can now be expressed as arrays and the software will plot the number of lines as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', and 'y2' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw' and 'c' can be either arrays or single variables/constants.
LINE AA x1, y1, x2, y2 [, LW [, C]]	Draws a line with anti-aliasing . The parameters are as per the LINE command above. However this version will use variable intensity values of the specified colour to reduce the “staggered” quality of diagonal lines. In addition this version can draw diagonal lines of any width.
LINE INPUT [prompt$,] string-variable$	Reads an entire line from the console input into ‘string-variable$’. ‘prompt$’ is a string constant (not a variable or expression) and if specified it 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 #nbr, string-variable$	Same as the LINE INPUT command except that the input is read from a file previously opened for INPUT as '#fnbr' or a serial communications port previously opened for INPUT as 'nbr'. See the OPEN command. #0 can be used which refers to the console. The # character is required.
LIST [file$] or LIST ALL [file$]	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. This is useful if you wish to transfer the program in the Maximite to a terminal emulator on a PC that has the ability to capture its input stream to a file. In most cases the filename 'file$' is required however if EDIT file$ or RUN file$ has been used previously the "current program name" will have been set and in that case LIST will default to using that filename.
LIST COMMANDS or LIST FUNCTIONS	Lists all valid commands or functions
LIST COM PORTS	Show currently available serial ports.
LIST PAGES	Lists the start address, width, height, and size of all the video pages for the current mode. In addition it shows whether for specific modes lines are duplicated in order to support the video output format.
LOAD DATA fname$, address	Loads the binary contents of file "fname$" and stores it in CMM2 memory starting at "address". See also SAVE DATA
LOAD FONT file$	Load the font contained in 'file$' on the SD card and install it as font #8. See the section *Basic Graphics* earlier in this manual. You can convert font files designed for the original Colour Maximite using FontTweak from: https://www.c-com.com.au/MMedit.htm
LOAD BMP file$ [, x, y] or LOAD GIF [file$ [, x, y]] or LOAD JPG file$ [, x, y] or LOAD PNG file$ [, x, y] [, transparency_cut_off]	Load an image from the SD card and display it on the MMBasic window. "file$' is the name of the file and 'x' and 'y' are the screen coordinates for the top left hand corner of the image. If the coordinates are not specified the image will be drawn at the top left hand position on the screen. If an extension is not specified the appropriate extension will be added to the file name. All types of the BMP format are supported including black and white and true colour 24-bit images. The image can be of any size and pixels off the screen will be ignored. GIFs can be a single image or animated. If it is animated it will start playing in the background (ie, program execution will continue while it is playing). If an animated GIF is already running it will be replaced by the new one. If LOAD GIF is used without any parameters it will stop the currently playing animated GIF. JPG images cannot use progressive encoding and are limited to being completely within the screen resolution (ie, pixels cannot extend beyond the screen limits). MODE 2,16 is the optimum for displaying JPG images as the hardware decoder can write RGB565 pixels directly into the frame buffer. For all other modes, the firmware has to adjust the image by duplicating lines (mode 3) and/or converting from RGB565 to RGB332. PNG files must be in the RGB888 or ARGB8888 format and can be sized up to the current resolution of the screen. If the x & y start coordinates are specified pixels off the screen will be ignored. If the transparency level is specified and none-zero then: • If a PNG file is in ARGB8888 format the 'transparency_cut_off' parameter is used to determine whether the pixel should be solid or missing/transparent. Valid values are 1 to 15, no default. MMBasic compares the 4 most significant bits of the transparency data in the file with the cut off value and assigns a transparency of 0 or 15 depending on the comparison. This allows RGB(0,0,0) to be a valid solid colour. • If the file is in RGB888 format then an RGB level of 0,0,0 is used to determine transparency as there is no other information to use. If the 'transparency_cut_off' level is not specified all pixels will be loaded as solid colours as with any other image load.

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.
LONGSTRING	The LONGSTRING commands allow for the manipulation of strings longer than the normal MMBasic limit of 255 characters. Variables for holding long strings must be defined as single dimensioned integer arrays with the number of elements set to the number of characters required for the maximum string length divided by eight. The reason for dividing by eight is that each integer in an MMBasic array occupies eight bytes. Note that the long string routines do not check for overflow in the length of the strings. If an attempt is made to create a string longer than a long string variable's size the outcome will be undefined.
LONGSTRING APPEND array%(), string$	Append a normal MMBasic string to a long string variable. array%() is a long string variable while string$ is a normal MMBasic string expression.
LONGSTRING CLEAR array%()	Will clear the long string variable array%(). ie, it will be set to an empty string.
LONGSTRING COPY dest%(), src%()	Copy one long string to another. dest%() is the destination variable and src%() is the source variable. Whatever was in dest%() will be overwritten.
LONGSTRING CONCAT dest%(), src%()	Concatenate one long string to another. dest%() is the destination variable and src%() is the source variable. src%() will the added to the end of dest%() (the destination will not be overwritten).
LONGSTRING LCASE array%()	Will convert any uppercase characters in array%() to lowercase. array%() must be long string variable.
LONGSTRING LEFT dest%(), src%(), nbr	Will copy the left hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the beginning of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING LOAD array%(), nbr, string$	Will copy 'nbr' characters from string$ to the long string variable array%() overwriting whatever was in array%().
LONGSTRING MID dest%(), src%(), start, nbr	Will copy 'nbr' characters from src%() to dest%() starting at character position 'start' overwriting whatever was in dest%(). ie, copy from the middle of src%(). 'nbr' is optional and if omitted the characters from 'start' to the end of the string will be copied src%() and dest%() must be long string variables. 'start' and 'nbr' must be an integer constants or expressions.
LONGSTRING PRINT [#n,] src%()	Prints the longstring stored in ‘src%()’ to the file or COM port opened as ‘#n’. If ‘#n’ is not specified the output will be sent to the console.
LONGSTRING REPLACE array%() , string$, start	Will substitute characters in the normal MMBasic string string$ into an existing long string array%() starting at position ‘start’ in the long string.
LONGSTRING RESIZE array%(), nbr	Sets the size of the longstring to nbr. This overrides the size set by other longstring commands so should be used with caution. Typical use would be in using a longstring as a byte array.
LONGSTRING RIGHT dest%(), src%(), nbr	Will copy the right hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the end of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING SETBYTE array%(), nbr, data	sets byte nbr to the value “data”, nbr respects OPTION BASE
LONGSTRING TRIM array%(), nbr	Will trim ‘nbr’ characters from the left of a long string. array%() must be a long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING UCASE array%()	Will convert any lowercase characters in array%() to uppercase. array%() must be long string variable.
LOOP [UNTIL expression]	Terminates a program loop: see DO.

MAP	The MAP commands allow the programmer to set the colours used in 8-bit colour modes. Each value in the 8-bit colour pallet can be set to an independent 24-bit colour.
MAP(n) = rgb%	his will assign the 24-bit colour 'rgb% to all pixels with the 8-bit colour value of 'n'. The change is activated after the MAP SET command.
MAP MAXIMITE	This will set the colour map to the colours implemented in the original Colour Maximite.
MAP SET	This will cause MMBasic to update the colour map (set using MAP(n)=rgb%) during the next frame blanking interval.
MAP RESET	This will reset the colour map to the default colours. This map is used to assign 24-bit colours to individual values in the 8-bit colour space.
MATH **Simple array arithmetic**	The math command performs many simple mathematical calculations that can be programmed in BASIC but there are speed advantages to coding looping structures in the firmware and there is the advantage that once debugged they are there for everyone without re-inventing the wheel. Note: 2 dimensional maths matrices are always specified DIM matrix(n_columns, n_rows) and of course the dimensions respect OPTION BASE. Quaternions are stored as a 5 element array w, x, y, z, magnitude.
MATH SET nbr, array()	Sets all elements in array() to the value nbr. Note this is the fastest way of clearing an array by setting it to zero.
MATH SCALE in(), scale,out()	This scales the matrix in() by the scalar scale and puts the answer in out(). Works for arrays of any dimensionality of both integer and float and can convert between. Setting b to 1 is optimised and is the fastest way of copying an entire array.
MATH ADD in(), num,out()	This adds the value 'num' to every element of the matrix in() and puts the answer in out(). Works for arrays of any dimensionality of both integer and float and can convert between. Setting num to 0 is optimised and is a fast way of copying an entire array. in() and out() can be the same array.
MATH INTERPOLATE in1(), in2(), ratio, out()	This command implements the following equation on every array element: out = (in2 - in1) * ratio + in1 Arrays can have any number of dimensions and must be distinct and have the same number of total elements. The command works with bot integer and floating point arrays in any mixture

MATH SLICE sourcearray(), [d1] [,d2] [,d3] [,d4] [,d5], destinationarray()	This command copies a specified set of values from a multi-dimensional array into a single dimensional array. It is much faster than using a FOR loop. The slice is specified by giving a value for all but one of the source array indicies and there should be as many indicies in the command, including the blank one, as there are dimensions in the source array
	e.g.
	`OPTION BASE 1`
	`DIM a(3,4,5)`
	`DIM b(4)`
	`MATH SLICE a(), 2,, 3, b()`
	Will copy the elements 2,1,3 and 2,2,3 and 2,3,3 and 2,4,3 into array b()

MATH INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5], sourcearray()	This is the opposite of MATH SLICE, has a very similar syntax, and allows you, for example, to substitute a single vector into an array of vectors with a single instruction
	e.g.
	`OPTION BASE 1`
	`DIM targetarray(3,4,5)`
	`DIM sourcearray(4)=(1,2,3,4)`
	`MATH INSERT targetarray(), 2,, 3, sourcearray()`
	Will set elements 2,1,3 = 1 and 2,2,3 = 2 and 2,3,3 = 3 and 2,4,3 = 4

Matrix arithmetic	
MATH M_INVERSE array!(), inversearray!()	This returns the inverse of array!() in inversearray!(). The array must be square and you will get an error if the array cannot be inverted (determinant=0). array!() and inversearray!() cannot be the same.
MATH M_PRINT array()	Quick mechanism to print a 2D matrix one row per line.
MATH M_TRANSPOSE in(), out()	Transpose matrix in() and put the answer in matrix out(), both arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in(m,n) out(n,m)
MATH M_MULT in1(), in2(), out()	Multiply the arrays in1() and in2() and put the answer in out()c. All arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in1(m,n) in2(p,m), out(p,n)
Vector arithmetic	
MATH V_PRINT array()	Quick mechanism to print a small array on a single line
MATH V_NORMALISE inV(), outV()	Converts a vector inV() to unit scale and puts the answer in outV() (sqr(x*x + y*y +.....)=1 There is no limit on number of elements in the vector
MATH V_MULT matrix(), inV(), outV()	Multiplies matrix() and vector inV() returning vector outV(). The vectors and the 2D matrix can be any size but must have the same cardinality.
MATH V_CROSS inV1(), inV2(), outV()	Calculates the cross product of two three element vectors inV1() and inV2() and puts the answer in outV()
Quaternion arithmetic	
MATH Q_INVERT inQ(), outQ()	Invert the quaternion in inQ() and put the answer in outQ()
MATH Q_VECTOR x, y, z, outVQ()	Converts a vector specified by x, y, and z to a normalised quaternion vector outVQ() with the original magnitude stored
MATH Q_CREATE theta, x, y, z, outRQ()	Generates a normalised rotation quaternion outRQ() to rotate quaternion vectors around axis x,y,z by an angle of theta. Theta is specified in radians but respects the setting of OPTION ANGLE
MATH Q_EULER yaw, pitch, roll, outRQ()	Generates a normalised rotation quaternion outRQ() to rotate quaternion vectors as defined by the yaw, pitch and roll angles With the vector in front of the “viewer” yaw is looking from the top of the ector and rotates clockwise, pitch rotates the top away from the camera and roll rotates around the z-axis clockwise. The yaw, pitch and roll angles default to radians but respect the setting of OPTION ANGLE
MATH Q_MULT inQ1(), inQ2(), outQ()	Multiplies two quaternions inQ1() and inQ2() and puts the answer in outQ()
MATH Q_ROTATE, RQ(), inVQ(), outVQ()	Rotates the source quaternion vector inVQ() by the rotate quaternion RQ() and puts the answer in outVQ()
MATH FFT signalarray!(), FFTarray!()	Performs a fast fourier transform of the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "FFTarray" must be floating point and have dimension 2*N where N is the same as the signal array (e.g. f(1,1023) assuming OPTION BASE is zero) The command will return the FFT as complex numbers with the real part in f(0,n) and the imaginary part in f(1,n)

MATH FFT INVERSE FFTarray!(), signalarray!()	Performs an inverse fast fourier transform of the data in “FFTarray!”. "FFTarray" must be floating point and have dimension 2*N where N must be a power of 2 (e.g. f(1,1023) assuming OPTION BASE is zero) with the real part in f(0,n) and the imaginary part in f(1,n). "signalarray" must be floating point and the single dimension must be the same as the FFT array. The command will return the real part of the inverse transform in "signalarray".
MATH FFT MAGNITUDE signalarray!(),magnitudearray!()	Generates magnitudes for frequencies for the data in “signalarray!” "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "magnitudearray" must be floating point and the size must be the same as the signal array The command will return the magnitude of the signal at various frequencies according to the formula: frequency at array position N = N * sample_frequency / number_of_samples
MATH FFT PHASE signalarray!(), phasearray!()	Generates phases for frequencies for the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "phasearray" must be floating point and the size must be the same as the signal array The command will return the phase angle of the signal at various frequencies according to the formula above.
MEMORY	List the amount of memory currently in use. For example: Program: 39K (6%) Program (1450 lines) 985K (94%) Free RAM: 0K (0%) 0 Variables 0K (0%) General 131200K (100%) Free Notes: • General memory is used by serial I/O buffers, etc. • Memory usage is rounded to the nearest 1K byte.
MEMORY SET address, byte, numberofbytes MEMORY SET BYTE address, byte, numberofbytes MEMORY SET SHORT address, short, numberofshorts MEMORY SET WORD address, word, numberofwords MEMORY SET INTEGER address, integervalue, numberofintegers [,increment] MEMORY SET FLOAT address, floatingvalue, numberofloats [,increment]	This command will set a region of memory to a value. BYTE = One byte per memory address. SHORT = Two bytes per memory address. WORD = Four bytes per memory address. INTEGER = Four bytes per memory address. FLOAT = Four bytes per memory address. ‘increment’ is optional and controls the increment of the ‘address’ pointer as the operation is executed. For example, if increment=3 then only every third element of the target is set. The default is 1.

MEMORY COPY sourceaddress, destinationaddres, numberofbytes MEMORY COPY INTEGER sourceaddress, destinationaddress, numberofintegers [,sourceincrement] [,destinationincrement] MEMORY COPY FLOAT sourceaddress, destinationaddress, numberoffloats [,sourceincrement] [,destinationincrement]	This command will copy one region of memory to another. COPY INTEGER and FLOAT will copy four bytes per operation. 'sourceincrement' is optional and controls the increment of the 'sourceaddress' pointer as the operation is executed. For example, if sourceincrement=3 then only every third element of the source will be copied. The default is 1. 'destinationincrement' is similar and operates on the 'destinationaddress' pointer.
MID$(str$, start [, num]) = str2$	The characters in 'str$', beginning at position 'start', are replaced by the characters in 'str2$'. The optional 'num' refers to the number of characters from 'str2' that are used in the replacement. If 'num' is omitted, all of 'str2' is used. Whether 'num' is omitted or included, the replacement of characters never goes beyond the original length of 'str$'.
MKDIR dir$	Make, or create, the directory 'dir$' on the SD card.
MODE [-]r, alphaenable [, bg]	Set the format for the MMBasic graphics window 'r' is the screen resolution and is a number from 1 to 20 as follows: 1 = 800 x 600 pixels 2 = 640 x 400 pixels 3 = 320 x 200 pixels 4 = 480 x 432 pixels 5 = 240 x 216 pixels 6 = 256 x 240 pixels 7 = 320 x 240 pixels 8 = 640 x 480 pixels 9 = 1024 x 768 pixels (default) 10 = 848 x 480 pixels (widescreen format) 11 = 1280 x 720 pixels (widescreen format) 12 = 960 x 540 pixels (widescreen format, lines duplicated) 13 = 400 x 300 pixels 14 = 960 x 540 pixels (widescreen format, 2 screen pixels per logical pixel 15 = 1200x1024 16 = 1920x1080 (widescreen format) 17 = 384 x 240 pixels 18 = 1024 x 600 19 = 1850 x 980 20 = 480 x 320 All resolutions except 10, 11, 12, 14, and 16 work perfectly with monitors that have an aspect ratio of 4:3 or widescreen monitors that can switch to that ratio (most widescreen monitors will do this automatically). If 'r' is negative, the display will open in 'full screen' mode ie. no border You can not set a full screen mode as the default.. 'alphaenable' flags whether 2 layer (transparency) is enabled or not 1 = enabled, 0 = not enabled 'bg' is the background colour and can be used in all mode.s If pixels in layer 0 are not set to solid (transparency = 255) then the background will show through as determined by the transparency value of the pixel.

The pages available in the various modes are:

Mode	Resolution	Pages
1	800 x 600	0 to 68
2	640 x 400	0 to 128
3	320 x 200	0 to 517
4	480 x 432	0 to 159
5	240 x 216	0 to 639
6	256 x 240	0 to 539
7	320 x 240	0 to 431
8	640 x 480	0 to 107
9	1024 x 768	0 to 41
10	848 x 480	0 to 80
11	1280 x 720	0 to 35
12	960 x 540	0 to 63
13	400 x 300	0 to 275
14	960 x 540	0 to 63
15	1200 x 1024	0 to 24
16	1920 x 1080	0 to 15
17	384 x 240	0 to 359
18	1024 x 600	0 to 53
19	1850 x 980	0 to 16
20	480 x 320	0 to 215

The display always shows the contents of PAGE 0 unless overridden with the PAGE DISPLAY command. Use PAGE WRITE and PAGE COPY to avoid artefacts of flashing and tearing. PAGE 0 is the lower level and PAGE1 the upper so the stack is: background, PAGE 0, PAGE 1 with each one overwriting the previous in turn as defined by the transparency values of each individual pixel.

MM.INFO(MAX PAGES) and MM.INFO(PAGE ADDRESS n) are useful if you wish to PEEK or POKE the video memory. In all cases the memory is arranged as a two dimensional array x,y so, to get the address of a specific pixel on a specific page n.
Notes:
For modes 3, 5, 6, 7, 12, and 13 each line in PAGE 0 is duplicated to get square pixels so Y needs to be multiplied by 2 for PEEK and both lines y*2 and Y*2+1 need to be POKEd. e.g, for an 8-bit colour depth:
add1% = MM.INFO(page address n)+ (y * 2) * MM.HRES + x
add2% = MM.INFO(page address n)+ (y * 2 + 1) * MM.HRES + x
NB: this duplication will also apply to page 1 in 12-bit colour modes.
For the 12 and 16-bit modes you can use POKE SHORT and PEEK(SHORT) which are designed for this purpose.

MOUSE LEFTDOWNint [, RIGHTDOWNint] [,LEFTUPint]

Generates an interrupt for left mouse button down or up and right mouse button down. Eg.

```
MOUSE ld_int, rd_int, lu_int
sub ld_int
  print "left down @ ",mouse(x),mouse(y)
end sub
sub lu_int
  print "left up @ ",mouse(x),mouse(y)
end sub
sub rd_int
  print "right down @ ",mouse(x),mouse(y)
end sub
```

NEW

Deletes the program in program memory, clears all variables including saved variables and resets the interpreter (ie, closes files, serial ports, etc).

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 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, missing hardware, file system access, 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 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/skipped the read only variable `MM.ERRNO` will be set to non zero and `MM.ERRMSG$` will be set to the error message that would normally be generated. These are reset to zero and an empty string by `ON ERROR CLEAR`. They are also cleared when the program is run and when `ON ERROR IGNORE` and `ON ERROR SKIP` are used. `ON ERROR IGNORE` can make it very difficult to debug a program so it is strongly recommended that only `ON ERROR SKIP` be used.
ON KEY target or ON KEY ASCIIcode, target	The first version of the command sets an interrupt which will call 'target' user defined subroutine whenever there is one or more characters waiting in the console input buffer. Note that all characters waiting in the input buffer should be read in the interrupt subroutine otherwise another interrupt will be automatically generated as soon as the program returns from the interrupt. The second version allows you to associate an interrupt routine with a specific key press. This operates at a low level for both the USB keyboard and a serial console and if activated the key does not get put into the input buffer but merely triggers the interrupt. It uses a separate interrupt from the simple ON KEY command so can be used at the same time if required. In both variants, to disable the interrupt use numeric zero for the target, i.e.: ON KEY 0. or ON KEY ASCIIcode, 0
OPEN fname$ 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 forward or backwards slash as a directory separator. 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, WRITE 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	Will open a serial communications port for reading and writing. Any serial port managed by the OS is available. To find out what ports are available, use `LIST COM PORTS`. Using 'fnbr' the port can be written to and read from using any command or function that uses a file number. 'comspec$' is the communication specification and is a string (it can be a string variable) specifying the serial port to be opened and optional parameters. The default is 9600 baud, 8 data bits, no parity and one stop bit. It has the form `"COMn: baud, buf, int, int-trigger, (DEN or DEP), 7BIT, (ODD or EVEN), INV, OC, S2"` Where: • 'n' is the serial port number. • 'baud' is the baud rate. This can be any value between 1200 (the minimum) and 1000000 (1MHz). Default is 9600. • 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes. • 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt. • 'int-trigger' sets the trigger condition for calling the interrupt subroutine. If it is a normal number the interrupt subroutine will be called when this number of characters has arrived in the receive queue. All parameters except the serial port name (COMn:) are optional. If any one parameter is left out then all the following parameters must also be left out and the defaults will be used. These options can be added to the end of 'comspec$' • 'INV' specifies that the transmit and receive polarity is inverted. • 'S2' specifies that two stop bits will be sent following each character transmitted. • '7BIT' will specify that 7 bit transmit and receive is to be used. • 'ODD' will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'EVEN' will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)
OPTION	See the section *Options* earlier in this manual.
PAGE COPY n TO m [,t]	Copy the contents of one video page to another. 'n' is the source and 'm' is the destination. If the optional field ', t' is used then only non-black pixels are copied.
PAGE RESIZE pageno, w, h	This command changes the width and height of a given page in memory. 'w' can be set between 1 and the normal width for the current display mode. 'h' can be set between 1 and the normal height of the display mode. When PAGE WRITE is set to a resized page all graphics commands will respect the new values of width and height, including print output scrolling, and MM.HRES and MM.VRES will report the revised size.
PAGE SCROLL pageno, x, y [,fillcolour]	Will scroll the image on the page specified by 'pageno' by the amount defined by 'x' pixels to the right and 'y' pixels up. By default the area scrolled off the screen appears on the other side. If 'fillcolour' is specified it will replace the area left behind by the scroll with the colour specified. If 'fillcolour' is set to -1 then the area left behind by the scroll is left untouched. This is the most efficient version and is suitable if there is a black background.
PAGE STITCH frompage1, from_page_2, topage, offset	Will take the last horizontal resolution minus 'offset' columns from 'frompage1' and the first 'offset' columns from 'frompage2' and copies them to 'topage'

PAGE AND_PIXELS sourcepage1, sourcepage2, destinationpage PAGE OR_PIXELS sourcepage1, sourcepage2, destinationpage PAGE XOR_PIXELS sourcepage1, sourcepage2, destinationpage	These commands combine the pixels on sourcepage1 and sourcepage2 by ANDing, ORing, or XORing them. destinationpage can be the same as either of the sourcepages if required.
PAGE WRITE n	Instructs MMBasic to make all graphics commands write to page 'n'. If not used page 0 is the default. Set the page to FRAMEBUFFER to write to the framebuffer – see the FRAMEBUFFER command.
PAUSE delay	Halt execution of the running program for 'delay' ms. This can be a fraction. For example, 0.2 is equal to 200 µs. The maximum delay is 2147483647 ms (about 24 days). Note that interrupts will be recognised and processed during a pause.
PIXEL x, y [,c]	Set a pixel in the currently display ing window to a colour. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. 'c' is a 32 bit number specifying the colour. 'c' is optional and if omitted the current foreground colour will be used. All parameters can be expressed as arrays and the software will plot the number of pixels as determined by the dimensions of the smallest array. 'x' and 'y' must both be arrays or both be single variables /constants otherwise an error will be generated. 'c' can be either an arrays or a single variable or constant. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
PIXEL FILL x, y, c	Implements a flood fill by reading the colour of the pixel at coordinates x,y and replacing it and the entire area of connected pixels having the same color with the new colour "c"
PLAY EFFECT file$ [,interrupt]	This will play the WAV file ' file$' at the same time as a MOD file is playing. If a previous EFFECT file is playing this command will immediately terminate it and commence playing the new file. The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. Note: wav files played using PLAY EFFECT during mod file playback must have the same sample rate as the modfile output. Files can be mono or stereo.
PLAY TONE left, right [, dur [, interrupt]]	Generates two separate sine waves on the sound output left and right channels. The tone plays in the background (the program will continue running after this command). 'left' and 'right' are the frequencies in Hz to use for the left and right channels. 'dur' specifies the number of milliseconds that the tone will sound for. MMBasic will round the time to the next nearest complete waveform of the first frequency specified so that the tone will always finish with the DC level in the middle and no discontinuity. If the duration is not specified the tone will continue until explicitly stopped or the program terminates. 'interrupt' is optional and is an interrupt subroutine to call when the tone has completed. The frequency can be from 1Hz to 20KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command.

PLAY WAV file$ [, interrupt] or PLAY FLAC file$ [, interrupt] or PLAY MP3 file$ [, interrupt]	Play an audio file on the OS audio output (if available) 'file$' is the file to play (the appropriate extension will be appended if missing). The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. For WAV files MMBasic will automatically compensate for the frequency, number of bits and number of channels of the WAV file. For FLAC files the supported frequencies are: 44100Hz 16-bit(CD quality) and 24-bit If 'file$' is a directory then the firmware will list all the files of the relevant type in that directory and start playing them one-by-one. To play files in the current directory use an empty string (ie, ""). Each file listed will play in turn and the optional interrupt will fire when all files have been played. The filenames are stored with full path so you can use CHDIR while tracks are playing without causing problems. All files in the directory are listed if the command is executed at the command prompt but the listing is suppressed in a program
PLAY MODFILE file$ [,samplerate]	Will play a MOD file on the OS audio output. 'file$' is the MOD file to play (the extension of .mod will be appended if missing). The MOD file is played in the background and will run continuously until PLAY STOP is called. The optional parameter samplerate specifies the number of samples per second generated by the modfile engine. The default is 44100. Note: wav files played using PLAY EFFECT during modfile playback must have the same sample rate as the modfile output.
PLAY MODSAMPLE sampleno, channelno [,volume] [,samplerate]	Plays one of the samples in the MOD file concurrently with the main MOD file playback. This allows sound effects to be incorporated in the MOD file. "sampleno" can be in the range 1 to 32. Up to 4 samples can be played simultaneously on independent channels using the specified "channelno" which must be in the range 1 to 4. The optional "volume" should be set in the range 0 to 64 (default 64). The optional "samplerate" specifies the update rate for the sample. The default is 16000. Changing this will change the pitch of the sample and the duration of playback and it should be set to the sample's original rate for playback as recorded.
PLAY SOUND soundno, channelno, type [,frequency] [,volume]	Play a series of sounds simultaneously on the audio output. 'soundno' is the sound number and can be from 1 to 4 allowing for four simultaneous sounds on each channel. 'channelno' specifies the output channel. It can be L (left speaker), R (right speaker) or B (both speakers) 'type' is the type of waveform. It can be S (sine wave), Q (square wave), T (triangle wave), W (rising sawtooth), N (noise), P (periodic noise) or O (turn off sound). Type N is true white noise. In this case the frequency parameter specifies the number of periods of 1/70000 seconds that the output stays at a particular random value. Type P is periodic white noise. In this case the frequency is some sort of relationship to the periodic frequency of the noise 'frequency' is the frequency from 1 to 20000 (Hz) and it must be specified except when type is O. 'volume' is optional and must be between 1 and 25. It defaults to 25 The first time PLAY SOUND is called all other audio usage will be blocked and will remain blocked until PLAY STOP is called. Output can be stopped temporarily using PLAY PAUSE and PLAY RESUME. Calling SOUND on an already running 'soundno' will immediately replace the previous output. Individual sounds are turned off using type "O" Running 4 sounds simultaneously on both channels of the audio output consumes about 23% of the CPU.

PLAY PAUSE	PLAY PAUSE will temporarily halt the currently playing file or tone.
PLAY RESUME	PLAY RESUME will resume playing a sound that was paused.
PLAY STOP	PLAY STOP will terminate the playing of the file or tone. When the program terminates for whatever reason the sound output will also be automatically stopped.
PLAY NEXT PLAY PREVIOUS	When playing a sequence of audio tracks (by using PLAY MP3 on a directory holding multiple MP3 files) these commands can be used to skip forward or skip back a file. The commands PLAY PAUSE, RESUME, VOLUME can also be used.
PLAY VOLUME left, right	Will adjust the volume of the audio output. 'left' and 'right' are the levels to use for the left and right channels and can be between 0 and 100 with 100 being the maximum volume. There is a linear relationship between the specified level and the output. The volume defaults to maximum when a program is run.
POKE BYTE addr%, byte or	Will set a byte or a word within the CPU's virtual memory space. POKE BYTE will set the byte (ie, 8 bits) at the memory location 'addr%' to 'byte'. 'addr%' should be an integer.
POKE SHORT addr%, short% or	POKE SHORT will set the short integer (ie, 16 bits) at the memory location 'addr%' to 'word%'. 'addr%' and short%' should be integers.
POKE WORD addr%, word% or	POKE WORD will set the word (ie, 32 bits) at the memory location 'addr%' to 'word %'. 'addr%' and 'word%' should be integers.
POKE INTEGER addr%, int% or	POKE INTEGER will set the MMBasic integer (ie, 64 bits) at the memory location 'addr%' to int%'. 'addr%' and int%' should be integers.
POKE FLOAT addr%, float! or	POKE FLOAT will set the word (ie, 64 bits) at the memory location 'addr%' to 'float!'. 'addr%' should be an integer and 'float!' a floating point number.
POKE VAR var, offset, byte or	POKE VAR will set a byte in the memory address of 'var'. 'offset' is the ±offset from the address of the variable. An array is specified as var().
POKE VARTBL, offset, byte	POKE VARTBL will set a byte in MMBasic's variable table. 'offset' is the ±offset from the start of the variable table. Note that a comma is required after the keyword VARTBL.
POLYGON n, xarray%(), yarray%() [, bordercolour] [, fillcolour]	Draws a filled or outline polygon with n xy-coordinate pairs in xarray%() and yarray%(). If 'fillcolour' is omitted then just the polygon outline is drawn. If 'bordercolour' is omitted then it will default to the current default foreground colour.
POLYGON n(), xarray%(), yarray%() [, bordercolour()] [, fillcolour()]	If the last xy-coordinate pair is not the same as the first the firmware will automatically create an additional xy-coordinate pair to complete the polygon. The size of the arrays should be at least as big as the number of x,y coordinate pairs. 'n' can be an array and the colours can also optionally be arrays as follows:
POLYGON n(), xarray%(), yarray%() [, bordercolour] [, fillcolour]	POLYGON n(), xarray%(), yarray%() [, bordercolour()] [, fillcolour()] POLYGON n(), xarray%(), yarray%() [, bordercolour] [, fillcolour] The elements of array n() define the number of xy-coordinate pairs in each of the polygons. e.g DIM n(1)=(3,3) would define that 2 polygons are to be drawn with three vertices each. The size of the n array determines the number of polygons that will be drawn unless an element is found with the value zero in which case the firmware only processes polygons up to that point. The x,y-coordinate pairs for all the polygons are stored in xarray%() and yarray%(). The xarray%() and yarray%() parameters must have at least as many elements as the total of the values in the n array. Each polygon can be closed with the first and last elements the same. If the last element is not the same as the first the firmware will automatically create an additional x,y-coordinate pair to complete the polygon. If fill colour is omitted then just the polygon outlines are drawn.

	The colour parameters can be a single value in which case all polygons are drawn in the same colour or they can be arrays with the same cardinality as n. In this case each polygon drawn can have a different colour of both border and/or fill. For example, this will draw 3 triangles in yellow, green and red: DIM c%(2)=(3,3,3) DIM x%(8)=(100,50,150,100,50,150,100,50,150) DIM y%(8)=(50,100,100,150,200,200,250,300,300) DIM fc%(2)=(rgb(yellow),rgb(green),rgb(red)) POLYGON c%(),x%(),y%(),fc%(),fc%()
PRINT expression [[,;]expression] … etc	Outputs text to the console (either the MMBasic window or the serial or both if they are available). 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 #nbr, 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' or to a serial communications port previously opened as 'nbr'. See the OPEN command. #0 can be used which refers to the console.
PRINT #nbr, BREAK	It is possible to send a break out the serial port. This is a true BREAK holding the serial line in the space status for 20 bit periods.
PRINT @(x [, y]) expression or PRINT @(x, [y], m) expression	Same as the standard PRINT command except that the cursor is positioned at the coordinates x, y expressed in pixels. If y is omitted the cursor will be positioned at "x" on the current line. Example: PRINT @(150, 45) "Hello World" The @ function can be used anywhere in a print command. Example: PRINT @(150, 45) "Hello" @(150, 55) "World" The @(x,y) function can be used to position the cursor anywhere on or off the screen. For example, PRINT @(-10, 0) "Hello" will only show "llo" as the first two characters could not be shown because they were off the screen. The @(x,y) function will automatically suppress the automatic line wrap normally performed when the cursor goes beyond the right screen margin. If 'm' is specified the mode of the video operation will be as follows: m = 0 Normal text (white letters, black background) m = 1 The background will not be drawn (ie, transparent) m = 2 The video will be inverted (black letters, white background) m = 5 Current pixels will be inverted (transparent background)
QUIT or CTRL+SHIFT+x	Will close both the main MMBasic window and the command line console window. Handy if you want to reload from MMEdit which requires MMB4W to be closed down.
READ SAVE or READ RESTORE	READ SAVE will save the virtual pointer used by the READ command to point to the next DATA to be read. READ RESTORE will restore the pointer that was previously saved. This enables subroutines to READ data and then restore the read pointer so as not to disturb other parts of the program that may be reading the same data statements. These commands can be nested.

RBOX x, y, w, h [, r] [,c] [,fill]	Draws a box with rounded corners on the MMBasic window starting at 'x' and 'y' which is 'w' pixels wide and 'h' pixels high. 'r' is the radius of the corners of the box. It defaults to 10. 'c' specifies the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can now be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'r', 'c', and 'fill' can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
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. Arrays can be used as variables (specified with empty brackets, eg, a()) and in that case the size of the array is used to determine how many elements are to be read. If the array is multidimensional then the leftmost dimension will be the fastest moving. See also DATA and RESTORE.
REM string	REM allows remarks to be included in a program. Note the Microsoft style use of the single quotation mark to denote remarks is also supported and is preferred.
RENAME 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$' and 'new$'. If the paths differ the file specified in 'old$' will be moved to the path specified in 'new$' with the file name as specified.
RESTART	Restarts the MMBasic interpreter (same as reset on a Maximite/Micrimite).
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. A variable can also be used as the parameter. In that case a numerical variable should be used for a line number and a string variable for a 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$’ on the SD card.
RUN file$ or RUN file$, cmdline	Run the program 'file$' held on the SD card. Note that 'file$' must be a string constant (ie, "MYPROG.BAS") including the quotes required around a string constant. It cannot be a variable or expression. If 'cmdline' is specified it will be available to the running program as the string returned by MM.CMDLINE$. 'cmdline' is not processed by MMBasic so it can contain numbers, commas, quoted strings, etc. It is the responsibility of the running program to decode this string of characters. 'file$' can be omitted and in that case MMBasic will run the "current program name" which is the file last used by RUN, EDIT or AUTOSAVE.
SAVE DATA fname$, address, size	Saves "size" bytes to file "fname$" starting from "address". This allows areas of the Colour Maximite 2 memory to be saved as binary files. See also LOAD DATA
SAVE IMAGE file$ [,x, y, w, h]	Save the current image on the MMBasic window as a 24-bit BMP file. 'file$' is the name of the file. If an extension is not specified “.BMP” will be added to the file name. ‘x’, ‘y’, ‘w’ and ‘h’ are optional and are the coordinates (x and y are the top left coordinate) and dimensions (width and height) of the area to be saved. If not specified the whole screen will be saved.
SEEK [#]fnbr, pos	Will position the read/write pointer in a file that has been opened on the SD card for RANDOM access to the 'pos' byte. The first byte in a file is numbered one so SEEK #5,1 will position the read/write pointer to the start of the file.

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 PIN(4)*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.
	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
	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.
	Each CASE statement should not have multiple commands (ie, something : something) on the same line.

SETTICK period, target [, nbr] SETTICK FAST frequency, target	This will setup a periodic interrupt (or "tick"). Four tick timers are available ('nbr' = 1, 2, 3 or 4). 'nbr' is optional and defaults to timer number 1. The time between interrupts is ‘period’ milliseconds and ‘target' is the interrupt subroutine which will be called when the timed event occurs. The period can range from 1 to 2147483647 ms (about 24 days). These interrupts can be disabled by setting ‘period’ to zero This allows you to exceed the current maximum rate of 1 interrupt per millisecond (1000Hz) and has been tested up to 50KHz. If the interrupt routine overruns the time available then interrupts will be lost. If the program is executing a statement that takes longer than the time between interrupts the interrupts will be stacked.
SETTICK PAUSE, target [, nbr] or SETTICK RESUME, target [, nbr]	Pause or resume the specified tick timer. When paused the interrupt is delayed but the current count is maintained.

SORT array() [,indexarray()] [,flags] [,startposition] [,elementstosort]	This command takes an array of any type (integer, float or string) and sorts it into ascending order in place. It has an optional parameter 'indexarray%()'. If used this must be an integer array of the same size as the array to be sorted. After the sort this array will contain the original index position of each element in the array being sorted before it was sorted. Any data in the array will be overwritten.This allows connected arrays to be sorted. See the section *Sorting Data* in the tutorial Programming with the Colour Maximite 2 for an example. The 'flag' parameter is optional and valid flag values are: bit0: 0 (default if omitted) normal sort - 1 reverse sort bit1: 0 (default) case dependent - 1 sort is case independent (string sorts only). The optional 'startposition' defines which element in the array to start the sort. Default is 0 (OPTION BASE 0) or 1 (OPTION BASE 1) The optional 'elementstosort' defines how many elements in the array should be sorted. The default is all elements after the startposition. Any of the optional parameters may be omitted so, for example, to sort just the first 50 elements of an array you could use: SORT array(),,,,50
SPRITE	The SPRITE commands are used to manipulate small graphic images on the MMBasic window. These are useful when writing games. The maximum size of a sprite is MM.HRES-1 and MM.VRES-1 See also the SPRITE() functions.
SPRITE CLOSE [#]n	Closes sprite "n" and releases its memory resources allowing the sprite number to be re-used. The command will give an error if other sprites are copied from this one unless they are closed first.
SPRITE CLOSE ALL	Closes all sprites and releases all sprite memory. The screen is not changed.
SPRITE COPY [#]n, [#]m, nbr	Makes a copy of sprite "n" to "nbr" of new sprites starting a number "m". Copied sprites share the same loaded image as the original to save memory
SPRITE HIDE [#]n SPRITE HIDE ALL SPRITE HIDE SAFE [#]n	Removes sprite n from the display and replaces the stored background. To restore a screen to a previous state sprites should be hidden in the opposite order to which they were written "LIFO" Hides all the sprites allowing the background to be manipulated. The following commands cannot be used when all sprites are hidden: SPRITE SHOW (SAFE) SPRITE HIDE (SAFE, ALL) SPRITE SWAP SPRITE MOVE SPRITE SCROLLR SPRITE SCROLL Removes sprite n from the display and replaces the stored background. Automatically hides all more recent sprites as well as the requested one and then replaces them afterwards. This ensures that sprites that are covered by other sprites can be removed without the user tracking the write order. Of course this version has less performance than the simple version and should only be used it there is a risk of the sprite being partially covered.
SPRITE INTERRUPT sub	Specifies the name of the subroutine that will be called when a sprite collision occurs. See Appendix E for how to use the function SPRITE to interrogate details of what has collided
SPRITE LOAD fname$ [,start_sprite_number]	Loads the file 'fname$' which must be formatted as an original Colour Maximite sprite file. See the original Colour Maximite *MMBasic Language Manual* for the file format. Multiple sprite files can be loaded by specifying a different 'start_sprite_number' for each file. The programmer is responsible for making sure that the sprites do not overlap.

SPRITE LOADARRAY [#]n, w, h, array%()	Creates the sprite 'n' with width 'w' and height 'h' by reading w*h RGB888 values from 'array%()'. The RGB888 values must be stored in order of columns across and then rows down starting at the top left. This allows the programmer to create simple sprites in a program without needing to load them from disk or read them from the display. The firmware will generate an error if 'array%()' is not big enough to hold the number of values required.
SPRITE LOADPNG [#]n, fname$ [, transparency_cut_off]	Loads the PNG image ‘fname$’ as sprite number ‘n’. If the PNG file is in ARGB8888 format the ‘transparency_cut_off’ parameter is used to determine whether the pixel should be solid or missing/transparent. Valid values are 1 to 15, default is 8. MMBasic compares the 4 most significant bits of the transparency data in the file with the cut off value and assigns a transparency of 0 or 15 depending on the comparison. This allows RGB(0,0,0) to be a valid solid colour. If the file is in RGB888 format then an RGB level of 0,0,0 is used to determine transparency as there is no other information to use.
SPRITE MOVE	Actions a single atomic transaction that re-locates all sprites which have previously had a location change set up using the SPRITE NEXT command. Collisions are detected once all sprites are moved and reported in the same way as from a scroll
SPRITE NEXT [#]n, x, y	Sets the X and Y coordinate of the sprite to be used when the screen is next scrolled or the SPRITE MOVE command is executed. Using SPRITE NEXT rather than SPRITE SHOW allows multiple sprites to be moved as part of the same atomic transaction.
SPRITE NOINTERRUPT	Disables collision interrupts
SPRITE RESTORE	Restores all the sprites. NB that any position changes previously requested using SPRITE NEXT will be actioned by the RESTORE and collision detection will be run
SPRITE READ [#]n, x, y, w, h [,pagenumber]	Reads the display area specified by coordinates ‘x’ and ‘y’, width ‘w’ and height ‘h’ into buffer number ‘n’. If the buffer is already in use and the width and height of the new area are the same as the original then the new command will overwrite the stored area. The optional parameter page number specifies which page is to be read to create the sprite. The default is the current write page. Set the page to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command.
SPRITE SCROLL x, y [,col]	Scrolls the background and any sprites on layer 0 'x' pixels to the right and 'y' pixels up. 'x' can be any number between -MM.HRES-1 and MM.HRES-1, 'y' can be any number between -MM.VRES-1 and MM.VRES-1. Sprites on any layer other than zero will remain fixed in position on the screen. By default the scroll wraps the image round. If 'col' is specified the colour will replace the area behind the scrolled image. If 'col' is set to -1 the scrolled area will be left untouched.
SPRITE SCROLLR x, y, w, h, delta_x, delta_y [,col]	Scrolls the region of the screen defined by top-right coordinates ‘x’ and ‘y’ and width and height ‘w’ and ‘h’ by ‘delta_x’ pixels to the right and ‘delta_y’ pixels up. By default the scroll wraps the background round. If ‘col’ is specified the colour will replace the area behind the scrolled image. Sprites on any layer other than zero will remain fixed in position on the screen. Sprites in layer zero where the centre of the sprite (x+ w/2, y+ h/2) falls within the scrolled region will move with the scroll and wrap round if the centre moves outside one of the boundaries of the scrolled region.

SPRITE SHOW [#]n, x,y, layer, [orientation]	Displays sprite ‘n’ on the screen with the top left at coordinates ‘x’, ‘y’. Sprites will only collide with other sprites on the same layer, layer zero, or with the screen edge. If a sprite is already displayed on the screen then the SPRITE SHOW command acts to move the sprite to the new location. The display background is stored as part of the command and will be replaced when the sprite is hidden or moved further. ‘orientation’ is optional and can be: 0 - normal display (default if omitted) 1 - mirrored left to right 2 - mirrored top to bottom 3 - rotated 180 degrees (= 1+2)
SPRITE SHOW SAFE [#]n, x,y, layer [,orientation] [,ontop]	Shows a sprite and automatically compensates for any other sprites that overlap it. If the sprite is not already being displayed the command acts exactly the same as SPRITE SHOW. If the sprite is already shown it is moved and remains in its position relative to other sprites based on the original order of writing. i.e. if sprite 1 was written before sprite 2 and it is moved to overlap sprite 2 it will display under sprite 2. If the optional "ontop" parameter is set to 1 then the sprite moved will become the newest sprite and will sit on top of any other sprite it overlaps. Refer to SPRITE SHOW for details of the orientation parameter.
SPRITE SWAP [#]n1, [#]n2 [,orientation]	Replaces the sprite ‘n1’ with the sprite ‘n2’. The sprites must have the same width and height and ‘n1’ must be displayed or an error will be generated. Refer to SPRITE SHOW for details of the orientation parameter. The replacement sprite inherits the background from the original as well as its position in the list of order drawn.
SPRITE TRANSPARENCY [#]n, transparency	Transparency can be between 1 and 15 and changes all pixels with a non-zero transparency in the stored sprite to the new level.
SPRITE WRITE [#]n, x y [,orientation]	Overwrites the display with the contents of sprite buffer ‘n’ with the top left at coordinates ‘x’, ‘y’. SPRITE WRITE overwrites the complete area of the display. The background that is overwritten is not stored so SPRITE WRITE is inherently higher performing than SPRITE SHOW but with greater functional limitations. The optional 'orientation' parameter defaults to 4 and specifies how the stored image data is changed as it is written out. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels
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 31 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 string$ [,array%()]	Executes the operating system command in ‘string$’. If the optional parameter is specified, the output from the system command will be directed to the long string ‘array%()’ otherwise output will appear on the console (stdout). Output can also be directed to a file using standard MS Windows or Linux notation. See Appendix C for examples of usage
TCP CLIENT server$, serverport TCP CLOSE TCP SEND string$ TCP RECEIVE string$	Opens a TCPIP socket as a client to a designated server. The server IP address is specified as a string of the form “192.168.1.149”. The serverport is the port number the intended server is expected to be listening on e.g. 80 for an HTML server. Closes an open TCPIP socket Sends a string to a connected device via TCPIP to the server specified in the TCP CLIENT statement. Attempts to receive a message on an opened socket. If no message is waiting string$ be a blank string. If a message is waiting it will be returned in string$. If the message will not fit into a string then repeated calls can be used to read all the message. A blank string will then indicate end of message
TEXT x, y, string$ [,alignment$] [, font] [, scale] [, c] [, bc]	Displays a string on the MMBasic window starting at 'x' and 'y'. ‘string$’ is the string to be displayed. Numeric data should be converted to a string and formatted using the Str$() function. ' alignment$' is a string expression or string variable consisting of 0, 1 or 2 letters where the first letter is the horizontal alignment around 'x' and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around 'y' and can be T, M or B for TOP, MIDDLE, BOTTOM. The default alignment is left/top. A third letter can be used in the alignment string to indicate the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (ie, upside down), 'U' the text will be rotated counter clockwise by 90º and 'D' the text will be rotated clockwise by 90º 'font' and 'scale' are optional and default to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
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. MMBasic is always logging the lines executed so this facility is always available (ie, it does not have to be turned on).

TRIANGLE X1, Y1, X2, Y2, X3, Y3 [, C [, FILL]]	Draws a triangle on the MMBasic window with the corners at X1, Y1 and X2, Y2 and X3, Y3. 'C' is the colour of the triangle and defaults to the current foreground colour. 'FILL' is the fill colour and defaults to no fill (it can also be set to -1 for no fill). All parameters can be expressed as arrays and the software will plot the number of triangles as determined by the dimensions of the smallest array unless X1 = Y1 = X2 = Y2 = X3 = Y3 = -1 in which case processing will stop at that point 'x1', 'y1', 'x2', 'y2', 'x3',and 'y3' must all be arrays or all be single variables /constants otherwise an error will be generated 'c' and 'fill' can be either arrays or single variables/constants.
TURTLE	The TURTLE commands are a simple method of drawing graphic images on the MMBasic window. They are suitable for newcomers and students.
TURTLE RESET	Clears the screen and re-initialises the turtle system. The turtle is located at MM.HRES\2, MM.VRES\2. The default pen colour is white and the fill colour is green. The initial heading is up (0 degrees)
TURTLE DRAW TURTLE	Draws a turtle at the current location and in the current orientation
TURTLE DRAW PIXEL x, y	Draw a 1-pixel dot at the given location using the current draw colour, regardless of current turtle location or pen status.
TURTLE FILL PIXEL x, y	Draw a 1-pixel dot at the given location using the current fill colour, regardless of current turtle location or pen status.
TURTLE DRAW LINE x1, y1, x2, y2	Draw a straight line between the given coordinates, regardless of current turtle location or pen status.
TURTLE DRAW CIRCLE x, y, r	Draw a circle at the given coordinates with the given radius, regardless of current turtle location or pen status.
TURTLE PEN UP	Lifts the pen so moves do not write to the screen
TURTLE PEN DOWN	Lowers the pen so moves do write to the screen
TURTLE FORWARD n	Moves the turtle n pixels in the current heading
TURTLE BACKWARD n	Moves the turtle n pixels opposite the current heading
TURTLE DOT	Draw a 1-pixel dot at the current location, regardless of pen status
TURTLE TURN LEFT deg	Turn the turtle to the left (anti-clockwise) by the specified number of degrees.
TURTLE TURN RIGHT deg	Turn the turtle to the right (clockwise) by the specified number of degrees.
TURTLE BEGIN FILL	Start filling. Call this before drawing a polygon to activate the bookkeeping required to run the filling algorithm later.
TURTLE END FILL	End filling. Call this after drawing a polygon to trigger the fill algorithm. The filled polygon may have up to 128 sides.
TURTLE HEADING deg	Rotate the turtle to the given absolute heading (in degrees). 0 degrees means facing straight up. 90 degrees means facing to the right.
TURTLE PEN COLOUR col	Set the current drawing colour. Colours are specified as per normal drawing commands
TURTLE FILL COLOUR col	Set the current fill colour. Colours are specified as per normal drawing commands
TURTLE MOVE x, y	Move the turtle to the specified location, drawing a straight line if the pen is down.

WATCHDOG timeout or WATCHDOG OFF	Starts the watchdog timer which will automatically restart the processor when it has timed out. This can be used to recover from some event that disabled the running program (such as an endless loop or a programming or other error that halts a running program). This can be important in an unattended control situation. 'timeout' is the time in milliseconds (ms) before a restart is forced. This command should be placed in strategic locations in the running BASIC program to constantly reset the watchdog timer and therefore prevent it from counting down to zero. If the timer count does reach zero (perhaps because the BASIC program has stopped running) the Maximite will be restarted and the automatic variable MM.WATCHDOG will be set to true (ie, 1) indicating that an error occurred. On a normal startup MM.WATCHDOG will be set to false (ie, 0). WATCHDOG OFF will disable the watchdog timer (this is the default on a reset or power up). The timer is also turned off when the break character (normally CTRL-C) is used on the console to interrupt a running program.

Functions

Note that the functions related to communications functions (I^2C, 1-Wire, and SPI) are not listed here but are described in the appendices at the end of this document.

Square brackets indicate that the parameter or characters are optional.

ABS(number)	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ACOS(number)	Returns the inverse cosine of the argument 'number' in radians.
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.
ATAN2(y, x)	Returns the arc tangent of the two numbers x and y as an angle expressed in radians. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.
ATN(number)	Returns the arctangent of the argument 'number' in radians.
BASE$(base, number [, chars])	Returns a string giving the base value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s). Base can be between 2 and 36. Numbers greater than 9 are represented by a letter as per the HEX notation. Example: PRINT BASE$(36, 35) will display "Z" Internally the functions BIN$, OCT$, and HEX$ use this function.
BIN$(number [, chars])	Returns a string giving the binary (base 2) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
BIN2STR$(type, value [,BIG])	Returns a string containing the binary representation of 'value'. 'type' can be: INT64 signed 64-bit integer converted to an 8 byte string UINT64 unsigned 64-bit integer converted to an 8 byte string INT32 signed 32-bit integer converted to a 4 byte string UINT32 unsigned 32-bit integer converted to a 4 byte string INT16 signed 16-bit integer converted to a 2 byte string UINT16 unsigned 16-bit integer converted to a 2 byte string INT8 signed 8-bit integer converted to a 1 byte string UINT8 unsigned 8-bit integer converted to a 1 byte string SINGLE single precision floating point converted to a 4 byte string DOUBLE double precision floating point converted to a 8 byte string By default the string contains the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to ‘BIG’ will return the string in big-endian format (ie, the most significant byte is the first one in the string) In the case of the integer conversions, an error will be generated if the ‘value’ cannot fit into the ‘type’ (eg, an attempt to store the value 400 in a INT8). This function makes it easy to prepare data for efficient binary file I/O or for preparing numbers for output to sensors and saving to flash memory. See also the function STR2BIN
BOUND(array() [,dimension]	This returns the upper limit of the array for the dimension requested. The dimension defaults to one if not specified. Specifying a dimension value of 0 will return the current value of OPTION BASE. Unused dimensions will return a value of zero. For example: DIM myarray(44,45) BOUND(myarray(),2) will return 45

CALL(userfunname$, [,userfunparameters,....])	This is an efficient way of programmatically calling user defined functions. (See also the CALL command). In many cases it can be used to eliminate complex SELECT and IF THEN ELSEIF ENDIF clauses and is processed in a much more efficient manner. "userfunname$" can be any string or variable or function that resolves to the name of a normal user function (not an in-built command). "userfunparameters" are the same parameters that would be used to call the function directly. A typical use for this command could be writing any sort of emulator where one of a large number of functions should be called depending on a some variable. It also provides a method of passing a function name to another subroutine or function as a variable.
CHOICE(condition, ExpressionIfTrue, ExpressionIfFalse)	This function allows you to do simple either/or selections more efficiently and faster than using IF THEN ELSE ENDIF clauses. The condition is anything that will resolve to nonzero (true) or zero (false). The expressions are anything that you could normally assign to a variable or use in a command and can be integers, floats or strings. Examples: PRINT CHOICE(1, "hello","bye") will print "Hello" PRINT CHOICE (0, "hello","bye") will print "Bye" a=1 : b=1 : PRINT CHOICE (a=b, 4, 5) will print 4
CHR$(number)	Returns a one-character string consisting of the character corresponding to the ASCII code indicated by argument 'number'.
CINT(number)	Round numbers with fractional portions up or down to the next whole number or integer. For example, 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also INT() and FIX().
COMPORT(port number)	return 0 is port does not exist, 1 if exists. See command LIST COM PORTS
COS(number)	Returns the cosine of the argument 'number' in radians.
CTRLVAL(#ref)	Returns the current value of an advanced control. '#ref' is the control's reference. For controls like check boxes or switches it will be the number one (true) indicating that the control has been selected by the user or zero (false) if not. For controls that hold a number (e.g. a SPINBOX) the value will be the number (normally a floating point number). For controls that hold a string (e.g. TEXTBOX) the value will be a string.
CWD$	Returns the current working directory on the SD card as a string. The format is: `A:/dir1/dir2`. See also MM.INFO(DIRECTORY) which will return the same thing but will always have a '/' character at the end
DATE$	Returns the current date based on MS Window's internal clock as a string in the form "DD-MM-YYYY". For example, "28-07-2012". The internal clock/calendar will keep track of the time and date including leap years. To set the date use the command DATE$ =.
DATETIME$(n)	Returns the date and time corresponding to the epoch number n (number of seconds that have elapsed since midnight GMT on January 1, 1970). The format of the returned string is "dd-mm-yyyy hh:mm:ss". Use the text NOW to get the current datetime string, i.e. ? DATETIME$(NOW)
DAY$(date$)	Returns the day of the week for a given date as a string "Monday", "Tuesday" etc. The format for date$ is "DD-MM-YY", "DD-MM-YYYY", or "YYYY-MM-DD". Use NOW to get the day for the current date, e.g. PRINT DAY$(NOW)
DEG(radians)	Converts 'radians' to degrees.

DIR$(fspec, type)
or
DIR$(fspec)
or
DIR$( )
	Will search currently active directory for files and return the names of entries found.
	'fspec' is a file specification using wildcards the same as used by the FILES command. Eg, "*" will return all entries, "*.TXT" will return text files.
	'type' is the type of entry to return and can be one of:
	ALL Search for both files and directories
	DIR Search for directories only
	FILE Search for files only (the default if 'type' is not specified)
	The function will return the first entry found. To retrieve subsequent entries use the function with no arguments. ie, DIR$( ). The return of an empty string indicates that there are no more entries to retrieve.
	This example will print all the files in a directory:
	`f$ = DIR$("*", FILE)`
	`DO WHILE f$ <> ""`
	`  PRINT f$`
	`  f$ = DIR$()`
	`LOOP`
	You must change to the required directory before invoking this command.

DRAW3D()	V5.07.00 includes a 3D engine. See the document “The CMM2 3D engine” for full details
EOF([#]nbr)	Will return true if the file previously opened on the SD card for INPUT with the file number ‘#fnbr’ is positioned at the end of the file. For a serial communications port this function will return true if there are no characters waiting in the receive buffer. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT$ function.
EPOCH(DATETIME$)	Returns the epoch number (number of seconds that have elapsed since midnight GMT on January 1, 1970) for the supplied DATETIME$ string. The format for DATETIME$ is “dd-mm-yyyy hh:mm:ss”, “dd-mm-yy hh:mm:ss”, or “yyyy-mm-dd hh:mm:ss”,. Use NOW to get the epoch number for the current date and time, i.e. PRINT EPOCH(NOW)
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$)` Will display: `86.6025`
EXP(number)	Returns the exponential value of 'number', ie, e^x where x is 'number'.
FIELD$(string1, nbr, string2 [, string3])	Returns a particular field in a string with the fields separated by delimiters. 'nbr' is the field to return (the first is nbr 1). 'string1' is the string to search and 'string2' is a string holding the delimiters (more than one can be used). 'string3' is optional and if specified will include characters that are used to quote text in 'string1' (ie, quoted text will not be searched for a delimiter). For example: S$ = "foo, boo, zoo, doo" r$ = FIELD$(s$, 2, ",") will result in r$ = "boo". While: s$ = "foo, 'boo, zoo', doo" r$ = FIELD$(s$, 2, ",", "'") will result in r$ = "boo, zoo".

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() .
FORMAT$(nbr [, fmt$])	Will return a string representing 'nbr' formatted according to the specifications in the string 'fmt$'. The format specification starts with a % character and ends with a letter. Anything outside of this construct is copied to the output as is. The structure of a format specification is: % [flags] [width] [.precision] type Where 'flags' can be: - Left justify the value within a given field width 0 Use 0 for the pad character instead of space + Forces the + sign to be shown for positive numbers space Causes a positive value to display a space for the sign. Negative values still show the – sign 'width' is the minimum number of characters to output, less than this the number will be padded, more than this the width will be expanded. 'precision' specifies the number of fraction digits to generate with an e, or f type or the maximum number of significant digits to generate with a g type. If specified, the precision must be preceded by a dot (.). 'type' can be one of: g Automatically format the number for the best presentation. F Format the number with the decimal point and following digits e Format the number in exponential format If uppercase G or F is used the exponential output will use an uppercase E. If the format specification is not specified "%g" is assumed. Examples: format$(45) will return 45 format$(45, "%g") will return 45 format$(24.1, "%g") will return 24.1 format$(24.1,"%f") will return 24.100000 format$(24.1, "%e") will return 2.410000e+01 format$(24.1,"%09.3f") will return 00024.100 format$(24.1,"%+.3f") will return +24.100 format$(24.1,"**%-9.3f**") will return **24.100 **
GETIP$()	Translates a web address such as www.thebackshed.com into the Page 18IP address *nnn.nnn.nnn.nnn*. The IP address is returned as a string which can be used as an input to UDP CLIENT
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 MMBasic input buffer 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 this is a carriage return, it is likely that there will be a line feed character following as often the enter key will produce a CR/LF pair. If the input buffer is empty this function will immediately return with an empty string (ie, "").

INPUT$(nbr, [#]fnbr)	Will return a string composed of ‘nbr’ characters read from a file in the current directory previously opened for INPUT with the file number ‘#fnbr’. This function will read all characters including carriage return and new line without translation. Will return a string composed of ‘nbr’ characters read from a serial communications port opened as 'fnbr'. This function will return as many characters as are waiting in the receive buffer up to ‘nbr’. If there are no characters waiting it will immediately return with an empty string. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, 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() .
JSON$(array%(),string$)	Returns a string representing a specific item out of the JSON input stored in the longstring array%() Examples taken from api.openweathermap.org JSON$(a%(), “name”) JSON$(a%(), “coord.lat”) JSON$(a%(), “weather[0].description”) JSON$(a%(),”list[4].weather[0].description
KEYDOWN(n)	Return the decimal ASCII value of the keyboard key that is currently held down or zero if no key is down. The decimal values for the function and arrow keys are listed in Appendix D. This function will report multiple simultaneous key presses and the parameter 'n' is the number of the keypress to report. KEYDOWN(0) will return the number of keys being pressed For example, if "c", "g" and "p" are pressed simultaneously KEYDOWN(0) will return 3, KEYDOWN(1) will return 99, KEYDOWN(2) will return 103, etc. The keys do not need to be pressed simultaneously and will report in the order pressed. Taking a finger off a key will promote the next key pressed to #1. The first key ('n' = 1) is entered in the keyboard buffer (accessible using INKEY$) while keys 2 to 6 can only be accessed via this function. Using this function will clear the console input buffer. KEYDOWN(7) will give any modifier keys that are pressed. These keys do not add to the count in keydown(0) The return value is a bitmask as follows: lalt ? 1, lctrl ? 2, lgui ? 4, lshift ? 8, ralt ? 16, rctrl ? 32, rgui ? 64, rshift ? 128 KEYDOWN(8) will give the current status of the lock keys. These keys do not add to the count in keydown(0) The return value is a bitmask as follows: caps_lock ? 1, num_lock ? 2, scroll_lock ? 4 Note that some keyboards will limit the number of active keys that they can report on.
LCASE$(string$)	Returns ‘string$’ converted to lowercase characters.
LCOMPARE(array1%(), array2%())	Compare the contents of two long string variables array1%() and array2%(). The returned is an integer and will be -1 if array1%() is less than array2%(). It will be zero if they are equal in length and content and +1 if array1%() is greater than array2%(). The comparison uses the ASCII character set and is case sensitive.
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$'.

LGETBYTE(array%(), n)	Returns the numerical value of the 'n'th byte in the LONGSTRING held in 'array%()'. This function respects the setting of OPTION BASE in determining which byte to return.
LGETSTR$(array%(), start, length)	Returns part of a long string stored in array%() as a normal MMBasic string. The parameters start and length define the part of the string to be returned.
LINSTR(array%(), search$ [,start])	Returns the position of a search string in a long string. The returned value is an integer and will be zero if the substring cannot be found. array%() is the string to be searched and must be a long string variable. Search$ is the substring to look for and it must be a normal MMBasic string or expression (not a long string). The search is case sensitive. Normally the search will start at the first character in 'str' but the optional third parameter allows the start position of the search to be specified.
LLEN(array%())	Returns the length of a long string stored in array%()
LOC([#]fnbr)	For a file on the system 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 communications port opened as 'fnbr' this function will return the number of bytes received and waiting in the receive buffer to be read. #0 can be used which refers to the console's input buffer. The # is optional.
LOF([#]fnbr)	For a file on the system, this will return the current length of the file in bytes. For a serial communications port opened as 'fnbr' this function will return the space (in characters) remaining in the transmit buffer. Note that when the buffer is full MMBasic will pause when adding a new character and wait for some space to become available. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.
MAP(number)	This function returns the RGB332 colour that is used internally to map to the colour lookup table (CLUT) when using 8-bit colour depths. For example MAP(255) will always return &HE0E0C0 which the firmware will then convert to 255. This is a way of selecting the normal colour that will find the place in the CLUT you want. It is not a window into the CLUT. See also the MAP command.

MATH	The math function performs many simple mathematical calculations that can be programmed in Basic but there are speed advantages to coding looping structures in C and there is the advantage that once debugged they are there for everyone without re-inventing the wheel.
Simple functions	
MATH(ATAN3 x,y)	Returns ATAN3 of x and y
MATH(COSH a)	Returns the hyperbolic cosine of a
MATH(LOG10 a)	Returns the base 10 logarithm of a
MATH(SINH a)	Returns the hyperbolic sine of a
MATH(TANH a)	Returns the hyperbolic tan of a
Simple Statistics	
MATH(CHI a())	Returns the Pearson's chi-squared value of the two dimensional array a())
MATH(CHI_p a())	Returns the associated probability in % of the Pearson's chi-squared value of the two dimensional array a())
MATH(CORREL a(), a())	Returns the Pearson's correlation coefficient between arrays a() and b()
MATH(MAX a())	Returns the maximum of all values in the a() array, a() can have any number of dimensions
MATH(MEAN a())	Returns the average of all values in the a() array, a() can have any number of dimensions
MATH(MEDIAN a())	Returns the median of all values in the a() array, a() can have any number of dimensions
MATH(MIN a())	Returns the minimum of all values in the a() array, a() can have any number of dimensions
MATH(SD a())	Returns the standard deviation of all values in the a() array, a() can have any number of dimensions
MATH(SUM a())	Returns the sum of all values in the a() array, a() can have any number of dimensions
Vector Arithmetic	
MATH(MAGNITUDE v())	Returns the magnitude of the vector v(). The vector can have any number of elements
MATH(DOTPRODUCT v1(), v2())	Returns the dot product of two vectors v1() and v2(). The vectors can have any number of elements but must have the same cardinality
Matrix Arithmetic	
MATH(M_DETERMINANT array!())	Returns the determinant of the array. The array must be square.

MATH(CRCn data [,length] [,polynome] [,startmask] [,endmask] [,reverseIn] [,reverseOut]	Calculates the CRC to n bits (8, 12, 16, 32) of "data" which can be an integer or floating point array or a string variable. "Length" is optional and if not specified the size of the array or string length is used.
	
	The defaults for startmask, endmask reverseIn, and reversOut are all zero. reverseIn, and reversOut are both Booleans and take the value 1 or 0.
	
	The defaults for polynomes are CRC8=&H07, CRC12=&H80D, CRC16=&H1021, and CRC32=&H04C11DB7
	Eg: For CRC16_CCITT use MATH(CRC16 array(), n,, &HFFFF)
	`DIM a%(8)=(49,50,51,52,53,54,55,56,57)`
	`PRINT HEX$(MATH(CRC16 a%(),9,,&HFFFF))`
	gives &H29B1
	See Appendix F for a detailed explanation of CRC. Thanks to Bill McKinley (turbo46 on TheBackShed Forum).

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$' beginning at 'start' and continuing for 'nbr' characters. The first character in the string is number 1. If 'nbr' is omitted the returned string will extend to the end of 'string$'
MOUSE(funct)	Returns data from a connected mouse 'funct' is a 1 letter code indicating the information to return as follows: X returns the value of the mouse X-position Y returns the value of the mouse Y-position L returns the value of the left mouse button (1 if pressed) R returns the value of the right mouse button (1 if pressed) W returns the value of the scroll wheel mouse button (1 if pressed) D This allows you to detect a double click of the left mouse button. The algorithm requires that the two clicks must occur between 100 and 500 milliseconds apart. The report via MOUSE(D) is then valid for 500mSec before it times out or until it is read.
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).
	Will return a byte or a word within the CPU's virtual memory space.
PEEK(BYTE addr%)	BYTE will return the byte (8-bits) located at 'addr%'
PEEK(SHORT addr%)	SHORT will return the short integer (16-bits) located at 'addr%'
PEEK(WORD addr%)	WORD will return the word (32-bits) located at 'addr%'
PEEK(INTEGER addr%)	INTEGER will return the integer (64-bits) located at 'addr%'
PEEK(FLOAT addr%	FLOAT will return the floating point number (64-bits) located at 'addr%'
PEEK(VARADDR var)	VARADDR will return the address (32-bits) of the variable 'var' in memory. An array is specified as var().
PEEK(VARHEADER var)	VARHEADER will return the address (32-bits) of the variable descriptor of the variable var in memory. An array is specified as var()
PEEK(VAR var, ±offset)	VAR, will return a byte in the memory allocated to 'var'. An array is specified as var().
PEEK(VARTBL, ±offset)	VARTBL, will return a byte in the memory allocated to the variable table maintained by MMBasic. Note that there is a comma after VARTBL.

PEEK(PROGMEM, ±offset)	PROGMEM, will return a byte in the memory allocated to the program. Note that there is a comma after the keyword PROGMEM. Note that 'addr%' should be an integer.
PI	Returns the value of pi.
PIXEL(x, y [,page_number])	Returns the colour of a pixel on the MMBasic window 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates. The optional parameter page_number specifies which page is to be read. The default is the current write page. Set the page number to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command
RAD(degrees)	Converts 'degrees' to radians.
RGB(red, green, blue [, trans]) or RGB(shortcut [, trans])	Generates an RGB true colour value. 'red', 'blue' and 'green' represent the intensity of each colour. A value of zero represents black and 255 represents full intensity. 'shortcut' allows common colours to be specified by naming them. The colours that can be named are white, black, blue, green, cyan, red, magenta, yellow, brown and grey or gray (USA spelling). For example, RGB(red) or RGB(cyan). There is also one special colour 'NOTBLACK', For any video mode this is the darkest colour that is not treated as black by various graphics commands. 'trans' is the level of transparency . It is optional and defaults to 255 if not specified.
RIGHT$(string$, number-of-chars)	Returns a substring of 'string$' with 'number-of-chars' from the right (end) of the string.
RND(number) or RND	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. MMBasic for Windows uses the random number generator in the underlying OS to deliver true random numbers. This means that the RANDOMIZE command is no longer needed and is not supported.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE$(number)	Returns a string of blank spaces 'number' characters long.
SPRITE()	The SPRITE functions return information regarding sprites which are small graphic images on the MMBasic window. These are useful when writing games. See also the SPRITE commands.
SPRITE(C, [#]n)	Returns the number of currently active collisions for sprite n. If n=0 then returns the number of sprites that have a currently active collision following a SPRITE SCROLL command
SPRITE(C, [#]n, m)	Returns the number of the sprite which caused the "m"th collision of sprite n. If n=0 then returns the sprite number of "m"th sprite that has a currently active collision following a SPRITE SCROLL command. If the collision was with the edge of the screen then the return value will be: &HF1 collision with left of screen &HF2 collision with top of screen &HF4 collision with right of screen &HF8 collision with bottom of screen
SPRITE(D, [#]s1, [#]s2)	Returns the distance between the centres of sprites 's1' and 's2' (returns -1 if either sprite is not active)
SPRITE(E, [#]n	Returns a bitmap indicating any edges of the screen the sprite is in contact with: 1 =left of screen, 2=top of screen, 4=right of screen, 8=bottom of screen

SPRITE(H,[#]n)	Returns the height of sprite n. This function is active whether or not the sprite is currently displayed (active).
SPRITE(L, [#]n)	Returns the layer number of active sprites number n
SPRITE(N)	Returns the number of displayed (active) sprites
SPRITE(N,n)	Returns the number of displayed (active) sprites on layer n
SPRITE(S)	Returns the number of the sprite which last caused a collision. NB if the number returned is Zero then the collision is the result of a SPRITE SCROLL command and the SPRITE(C…) function should be used to find how many and which sprites collided.
SPRITE(V,spriteno1,spriteno2)	Returns the vector from 'spriteno1' to 'spriteno2' in radians. The angle is based on the clock so if 'spriteno2' is above 'spriteno1' on the screen then the answer will be zero. This can be used on any pair of sprites that are visible. If either sprite is not visible the function will return -1. This is particularly useful after a collision if the programmer wants to make some differential decision based on where the collision occurred. The angle is calculated between the centre of each of the sprites which may of course be different sizes.
SPRITE(T, [#]n)	Returns a bitmap showing all the sprites currently touching the requested sprite Bits 0-63 in the returned integer represent a current collision with sprites 1 to 64 respectively
SPRITE(V,[#]so1, [#]s2)	Returns the vector from sprite 's1' to 's2' in radians. The angle is based on the clock so if 's2' is above 's1' on the screen then the answer will be zero. This can be used on any pair of sprites that are visible. If either sprite is not visible the function will return -1. This is particularly useful after a collision if the programmer wants to make some differential decision based on where the collision occurred. The angle is calculated between the centre of each of the sprites which may of course be different sizes.
SPRITE(W, [#]n)	Returns the width of sprite n. This function is active whether or not the sprite is currently displayed (active).
SPRITE(X, [#]n)	Returns the X-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.
SPRITE(Y, [#]n)	Returns the Y-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.
STR2BIN(type, string$ [,BIG])	Returns a number equal to the binary representation in ‘string$’. ‘type’ can be: INT64 converts 8 byte string representing a signed 64-bit integer to an integer UINT64 converts 8 byte string representing an unsigned 64-bit integer to an integer INT32 converts 4 byte string representing a signed 32-bit integer to an integer UINT32 converts 4 byte string representing an unsigned 32-bit integer to an integer INT16 converts 2 byte string representing a signed 16-bit integer to an integer UINT16 converts 2 byte string representing an unsigned 16-bit integer to an integer INT8 converts 1 byte string representing a signed 8-bit integer to an integer UINT8 converts 1 byte string representing an unsigned 8-bit integer to an integer SINGLE converts 4 byte string representing single precision float to a float DOUBLE converts 8 byte string representing single precision float to a float By default the string must contain the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to ‘BIG’ will interpret the string in big-endian format (ie, the most significant byte is the first one in the string). This function makes it easy to read data from binary data files, interpret numbers from sensors or efficiently read binary data from flash memory chips. An error will be generated if the string is the incorrect length for the conversion requested. See also the function BIN2STR$

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$)	Returns a formatted string in decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' 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) will return "-123.456" STR$(123.456, 1) will return "123.456" STR$(123.456, -1) 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 on the console output. The tab function will not work when printing to a file but will behave like the SPACE$ function.
TAN(number)	Returns the tangent of the argument 'number' in radians.
TIME$	Returns the current time based on MMBasic's internal clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00". If the OPTION MILLISECONDS ON command has been used this function will return the time including milliseconds as a decimal fraction of the seconds. For example: "14:35:06.239".
TIMER	Returns the elapsed time in milliseconds (eg, 1/1000 of a second) since reset. This is a fractional floating point number with a resolution of 1µs. The timer is reset to zero on power up or a CPU restart and you can also reset it to any value by using TIMER as a command.
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 modern commands in MMBasic should be used.

These commands may be removed in the future to recover memory for other features.

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
IRETURN	Returns from an interrupt when the interrupt destination was a line number or a label. New programs should use a user defined subroutine as an interrupt destination. In that case END SUB or EXIT SUB will cause a return from the interrupt.
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.
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.

Appendix A. Serial Communications

Any serial port provided by the underlying OS is available for asynchronous serial communications. Use LIST COM PORTS to show available serial ports. The function COMPORT(portno) will return 0 if the port does not exist, 1 if it exists.

If using a USB to Serial adaptor, the adaptor should be plugged in prior to starting MMB4W – in some cases, dynamic detection may be missed.

After being opened, the serial port will have an associated file number and you can use any commands that operate with a file number to read and write to/from it. A serial port is also closed using the CLOSE command.

The following is an example:

```
OPEN "COM3:4800" AS #5      ' open the serial port 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
PAUSE 1     ' allow background receive to complete prior to close
CLOSE #5    ' close the serial port
```

The serial functionality is a combination of MS Windows hardware and software, OneLoneCoder middleware and the MMBasic interpreter. This is a complex interaction that can have unpredictable results. In particular, after a serial read with data continuously being received, using either the INPUT command or the INPUT$ function immediately followed by a CLOSE command it is possible that MMBasic may hang. This is due to the asynchronous nature of all procedures in action (Windows hardware/software, OLC and MMB4W). Hence the PAUSE statement in the example above.

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 default is 9600 baud, 8 data bits, no parity and one stop bit.

It has the form "COMn: baud, buf, int, int-trigger, 7BIT, (ODD or EVEN), S2" where:

- 'n' is the serial port number starting at COM3 (for Linux under WINE, COM33:)
- 'baud' is the baud rate. This can be any value between 1200 (the minimum) and 4000000 (4MHz). Default is 9600.
- 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes.
- 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt.
- 'int-trigger' sets the trigger condition for calling the interrupt subroutine. It is an integer and the interrupt subroutine will be called when this number of characters has arrived in the receive queue.

All parameters except the serial port name (COMn:) are optional. If any parameter is left out then all the following parameters must also be left out and the defaults will be used.

The following options can be added to the end of 'comspec$'

- '7BIT' will specify that 7 bit transmit and receive is to be used. Default is 8 bits.
- 'ODD' will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)
- 'EVEN' will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)'
- 'S2' specifies that two stop bits will be sent following each character transmitted. Default is one stop bit.

Most modern computers/laptops do not have a built in serial port. Instead they rely on users connecting a USB to serial adaptor such as the FTDI232 or CP2014 modules.

It is possible to send a break out the serial port.

```
PRINT #1, BREAK
```

This is a true BREAK holding the serial line in the space status for 20 bit periods.

Examples

Opening a serial port using all the defaults:

```
OPEN "COM3:" AS #2
```

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

```
OPEN "COM3:4800" AS #1
```

Opening a serial port specifying the baud rate (9600 bits per second) and receive buffer size (1KB):

```
OPEN "COM3:9600, 1024" AS #8
```

The same as above but with two stop bits enabled:

```
OPEN "COM3:9600, 1024, S2" AS #8
```

An example specifying everything including an interrupt, an interrupt level and two stop bits:

```
OPEN "COM3:19200, 1024, ComIntLabel, 256, S2" AS #5
```

Reading and Writing

Once a serial port has been opened you can use any command or function that uses a file number to read from and write to the port. Data received by the serial port will be automatically buffered in memory by MMBasic until it is read by the program and the INPUT$() function is the most convenient way of doing that. When using the INPUT$() 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 LOC() function is also handy; it will return the number of characters waiting in the receive buffer (ie, the maximum number characters that can be retrieved by the INPUT$() function). Note that if the receive buffer overflows with incoming data the serial port will automatically discard the oldest data to make room for the new data.

The PRINT command is used for outputting to a serial port and any data to be sent will be held in a memory buffer while the serial port is sending it. This means that MMBasic will continue with executing the commands after the PRINT command while the data is being transmitted. The one exception is if the output buffer is full and in that case MMBasic will pause and wait until there is sufficient space before continuing. The LOF() function will return the amount of space left in the transmit buffer and you can use this to avoid stalling the program while waiting for space in the buffer to become available.

If you want to be sure that all the data has been sent (perhaps because you want to read the response from the remote device) you should wait until the LOF() function returns 256 (the transmit buffer size) indicating that there is nothing left to be sent.

Serial ports can be closed with the CLOSE command. This will wait for the transmit buffer to be emptied then free up the memory used by the buffers, cancel the interrupt (if set) and set all pins used by the port to the not configured state. Note however that if serial data is continuously being received, contention can arise between MMB4W if a CLOSE statement is immediately preceded with either an INPUT command or an INPUT$ function. In such a case, it is advisable to insert a PAUSE 1 (1mS) between the two statements or to ensure that the OPEN / CLOSE structure is not included within a tight loop reading serial input. Note that each instance of OPEN / CLOSE creates and releases the receive buffer thus having them in a loop just wastes system resources.

A serial port is also automatically closed when commands such as RUN and NEW are issued.

Interrupts

The interrupt subroutine (if specified) will operate the same as a general interrupt on an external I/O pin.

When using interrupts you need to be aware that it will take some time for MMBasic to respond to the interrupt and more characters could have arrived in the meantime, especially at high baud rates. For example, if you have specified the interrupt level as 250 characters and a buffer of 256 characters then quite easily the buffer will have overflowed by the time the interrupt subroutine can read the data. In this case the buffer should be increased to 512 characters or more. Note that serial output is currently blocking and the time taken will depend on the baudrate.

Appendix B. Connecting to the Internet

Below is a simple Telnet client in MMB4W. This allows you to talk to the Web version of the PicoMite (WebMite) over the internet all without leaving Basic.

Also implemented is OPTION ESCAPE to make things even easier.

This example just lists a file remotely but you could execute a program, and read the output or enter commands at the command prompt to do things like read an analogue voltage.

e.g.

```
option escape ' turn on escape codes
telnet client open "192.168.1.147",23 ' open a telnet connection to the Webmite
telnet send "list all \qsolar_eclipse.bas\q\r\n" 'send any sort of command to the
Webmite
timer=0
do
telnet receive s$ 'read anything the Webmite has returned
if len(s$) then
print s$;
timer=0
endif
loop until timer>100
telnet close 'close the telnet connection
```

An example of accessing OpenWeatherMap - refer to the Picomite/PicoWEB User Manual for more detail.

```
Option explicit
Option default none
Dim a$,b$,c$
Dim integer i,j,rlen
Dim m%(4000)
Dim PLACEID$="2638878"
Dim APIKEY$="myAPIkey"
'
' get the IP address of openweathermap and open a client connection to it
'
a$="api.openweathermap.org"
TCP client a$,80
'
' construct the request for a weather report and send it
'
b$= "GET /data/2.5/weather?id="+PLACEID$+"&APPID="+APIKEY$
b$=b$+" HTTP/1.0"+Chr$(13)+Chr$(10)+Chr$(13)+Chr$(10)
TCP send b$
'
'set up a longstring variable and read the return message into it
'
LongString clear m%()
Do
TCP receive c$
If c$<>"" Then LongString append m%(),c$
Loop While c$<>""
'
' trim off the HTTP header
'
LongString trim m%(),LInStr(m%(),"{")-1
rlen=LLen(m%())
'
' process the message, in this case just print it out
'
i=1
Do While rlen>0
If rlen>240 Then
j=240
Else
j=rlen
EndIf
```

```
rlen=rlen-j
Print LGetStr$(m%(),i,j);
i=i+j
Loop
Print ""
TCP close
Print "Weather for "+json$(m%(),"name")
Print "Temperature is ",val(json$(m%(),"main.temp"))-273
Print "Pressure is ",json$(m%(),"main.pressure")
Print json$(m%(),"weather[0].description")
```

Appendix C. Sprites

The concept of the sprite implementation is as follows:

- Sprites are full colour and of any size. The collision boundary is the enclosing rectangle.
- Sprites are loaded to a specific number (1 to 64).
- Sprites are displayed using the SPRITE SHOW command.
- For each SHOW command the user must select a "layer". This can be between 0 and 10.
- Sprites collide with sprites on the same layer, layer 0, or the screen edge.
- Layer 0 is a special case and sprites on all other layers will collide with it.
- The SCROLL commands leave sprites on all layers except layer 0 unmoved.
- Layer 0 sprites scroll with the background and this can cause collisions.
- There is no practical limit on the number of collisions caused by SHOW or SCROLL commands.
- The SPRITE() function allows the user to fully interrogate the details of a collision.
- A SHOW command will overwrite the details of any previous collisions for that sprite.
- A SCROLL command will overwrite details of previous collisions for ALL sprites.
- To restore a screen to a previous state sprites should be removed in the opposite order to which they were written (ie, last in first out).

Because moving a sprite or, particularly, scrolling the background can cause multiple sprite collisions it is important to understand how they can be interrogated.

The best way to deal with a sprite collision is using the interrupt facility. A collision interrupt routine is set up using the SPRITE INTERRUPT command. Eg:

```
SPRITE INTERRUPT collision
```

The following is an example program for identifying all collisions that have resulted from either a SPRITE SHOW command or a SCROLL command

```
' This routine demonstrates a complete interrogation of collisions
'
SUB collision
  LOCAL INTEGER i
' First use the SPRITE(S) function to see what caused the interrupt
  IF SPRITE(S) <> 0 THEN 'collision of specific individual sprite
    'SPRITE(S) returns the sprite that moved to cause the collision
    PRINT "Collision on sprite ", SPRITE(S)
    process_collision(SPRITE(S))
    PRINT
  ELSE     '0 means collision of one or more sprites caused by background move
           ' SPRITE(C, 0) will tell us how many sprites had a collision
    PRINT "Scroll caused a total of ", SPRITE(C,0)," sprites to have collisions"
    FOR I = 1 TO SPRITE(C, 0)
      ' SPRITE(C, 0, i) will tell us the sprite number of the "I"th sprite
      PRINT "Sprite ", SPRITE(C, 0, i)
      process_collision(SPRITE(C, 0, i))
    NEXT i
    PRINT
  ENDIF
END SUB

' get details of the specific collisions for a given sprite
SUB process_collision(S AS INTEGER)
  LOCAL INTEGER i, j
  ' SPRITE(C, #n) returns the number of current collisions for sprite n
  PRINT "Total of " SPRITE(C, S) " collisions"
   FOR I = 1 TO SPRITE(C, S)
    ' SPRITE(C, S, i) will tell us the sprite number of the "I"th sprite
    j = SPRITE(C, S, i)
    IF j = &HF1 THEN
```

```
      PRINT "collision with left of screen"
    ELSE IF j = &HF2 THEN
      PRINT "collision with top of screen"
    ELSE IF j = &HF4 THEN
      PRINT "collision with right of screen"
    ELSE IF j = &HF8 THEN
      PRINT "collision with bottom of screen"
    ELSE
      ' SPRITE(C, #n, #m) returns details of the mth collision
      PRINT "Collision with sprite ", SPRITE(C, S, i)
    ENDIF
  NEXT i
END SUB
```

Appendix D. Special Keyboard Keys

MMBasic generates a single unique character for the function keys and other special keys on the keyboard.

These are shown in this table as hexadecimal and decimal numbers:

Keyboard Key	Key Code (Hex)	Key Code (Decimal)
DEL	7F	127
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
Alt	8B	139
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
PrtScr/SysRq	9D	157
PAUSE/BREAK	9E	158
SHIFT_TAB	9F	159
SHIFT_DEL	A0	160
SHIFT_DOWN_ARROW	A1	161
SHIFT_RIGHT_ARROW	A3	163

If the shift key is simultaneously pressed with the function keys F1 to F12 then 40 (hex) is added to the code (this is the equivalent of setting bit 6). For example Shift-F10 will generate DA (hex).

The shift modifier works with the function keys F1 to F12; it is ignored for the other keys except TAB, DEL, DOWN_ARROW, and RIGHT_ARROW as identified above.

MMBasic will translate most VT100 escape codes generated by terminal emulators such as Tera Term and Putty to these codes (excluding the shift and control modifiers). This means that a terminal emulator operating over a USB or a serial port opened as console will generate the same key codes as a directly attached keyboard.

Appendix E. Serial Ports in Linux/Wine

- By Volhout

How to use serial ports in Linux when running MMB4W under Wine.

Tested on Ubuntu 20.04LTS (20.04.03).

- Install Wine (instructions from Ubuntu website, (this does not install the latest version of Wine, but that is not essential for this post).

Most of us will use USB serial converters using FTDI or CH340 chips. Good news: Linux does not require you to install specific drivers. All drivers (also for cheap Chinese copies) are in the Linux kernel.

Wine emulates MS Windows in Linux, and for serial ports it uses symbolic links to the Linux serial ports. In Linux these serial ports (devices) are visible as objects in the folder /dev/. The serial ports have a group identification (tty) extended with a type identification (i.e. S) and number (0...32).

The first serial port on a Linux system would be /dev/ttyS0.

If you open a terminal in Linux and type ls /dev/tty* you get an overview of all serial ports.

The type designator (the "S") can have different types

S = hardware serial port

USB = USB-serial convertor (this type is used by FTDI chips)

ACM = USB modem (this type is often used by Arduino, CH340 and the Microbridge/PIC32F1455)

Thus an Microbridge based converter would show up as /dev/ttyACM0 (if the first plugged in).

Linux automatically enumerates new serial ports when a USB-serial cable is plugged in. This is a convenience (easy) and an annoyance (2 USB serial ports are not always enumerated in the same order, so you can easily accidentally swap them). Note that they need to be plugged in PRIOR to starting MMB4W under Wine to be recognised by MMB4W.

Using Wine

Wine lives within the Linux system in the users space. In Ubuntu the user has a home folder, and in this folder you find folders like "Documents", "Downloads", etc.. very similar to Windows user space.

In this user space, Wine uses a hidden folder called ".wine" (in Linux, folders become hidden when the folder name starts with a period "."). You can view this folder in the file browser when enabling view of hidden folders (and then you will see many more hidden folders, but .wine is one of them once you have started Wine once.

In user space the folder .wine/dosdevices shows the locations where Wine interfaces with the Linux system. You can find a link to your C drive, and links to serial ports.

Every time you start Wine, the folder .wine/dosdevices is refreshed, so it is not advised to make changes to the links in this folder, they will get lost.

Wine re-writes the contents of this folder with links to serial ports in the /dev/ttyS* list, and these are defaulted to hardware serial ports (ttyS0...ttyS32).

It links COM1 to /dev/ttyS0, COM2 to /dev/ttyS1 etc...).

So how do we attach a USB serial port to MMBasic for Windows running under Wine? All are attached to non existing hardware ports. Wine will add the /dev/ttyUSB0 or /dev/ttyACM0 after the first 32 hardware serial ports. so it will become COM33. When your program cannot handle COM33 (I expect MMBasic for Windows will, but some programs can not, they are restricted to COM1..COM4) then you need to force Wine to use the USB serial interface as COM1.

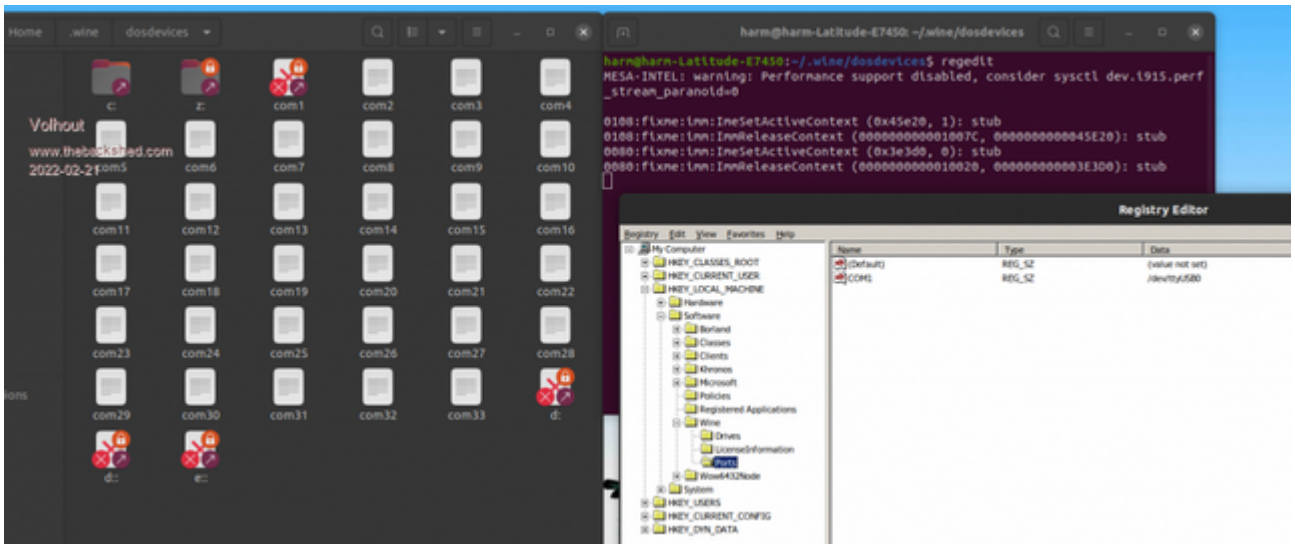
This is how:

Regedit !

In the Linux terminal, type regedit. You will start Wine's registry editor.

In HKEY_LOCAL_MACHINE/software/wine/ports you can find serial ports. This section is empty (only the default is there). When you add "COM1" with value "/dev/ttyUSB0" or “/dev/ttyACM0” it will be replacing the default com1 link to /dev/ttyS0 to /dev/ttyUSB0 or /dev/ttyACM0 respectively.

See below picture that shows (in Ubuntu 20.04) the folder .wine/dosdevices with many serial port links, and a link for the C drive. It shows the terminal where you start regedit, and it shows an example in regedit, where I define the com1 link to /dev/ttyUSB0



For basic usage, when only 1 or 2 USB-serial interfaces are attached, this may suffice. You simply add a second entry in regedit with com2 linking to /dev/ttyUSB1

For more complex setups, Linux can use rules, based on the manufacturer and serial number of the USB-serial converter chip to identify each of them uniquely. Search Linux information for details.

So does it work .... YES

The following simple terminal program can be used to communicate between MMBasic for Windows running under Wine, with a CMM2. In between is a FTDI TTL-232RG cable.

```
'very simple terminal program

open "com1:9600" as #1

do
 do
   if loc(#1) <> 0 then
     b$=input$(loc(#1),#1)
     print b$;
   end if
   a$=inkey$
 loop until a$<>""
 print #1,a$;
loop until a$="q"
```

The only quirk I found is that the CMM2 and MMBasic for Windows under Wine use a different terminator (<LF> and <CR><LF>) but that is easily overcome.

Appendix F. Cyclic Redundancy Check (CRC)

The purpose of this description is not to explain or examine the mathematics behind CRCs but merely to explain the benefits of using them and how they may be used in MMBasic.

A cyclic redundancy check (CRC) is a strong algebraic error-detecting code commonly used in digital networks and storage devices to detect accidental changes to the data. Blocks of data have a short check value attached, this is the CRC. A CRC is based on the remainder of a polynomial division of their contents. This technique was invented by W. Wesley Peterson in 1961 and further developed by the CCITT.

Note: A CRC is not an error correction code it is just for error detection.

The advantages of using a CRC when sending or saving data are that it is a fast and efficient method for detecting errors in data transmission and can detect errors that occur during transmission, and storage caused by things such as noise, interference or distortion.

There are simpler methods of error detection including the use of ODD/EVEN parity for ASCII transmission and the use of a checksum. A checksum is simply an addition of all of the bits transmitted in a block of data, usually the carry bit is ignored and the resultant value is truncated to 8 or 16 bits which is appended to the end of the block of data. Neither of these methods are particularly secure.

The more bits in the CRC, the more errors it will detect so a 16 bit CRC will be more secure than an 8 bit CRC and so on. While a CRC will not catch all errors, it is much more secure than a simple checksum.

Using a CRC

Suppose we want to send a string via some medium. It could be data from your weather station which is located up a pole which is sent via radio to your base station for example.

A simple example using the MODBUS CRC:

```
' A simple demonstration of using a CRC
' the data to send
a$="123456789"
' calculate the CRC
b% = math(crc16 a$,,&h8005,&hffff,0,1,1)
' convert the CRC to a string
acrc$ = CHR$(b% AND &HFF) + CHR$((b%>>8) AND &HFF)
' add the CRC to the data to send
txd$ = a$ + acrc$
' then transmit the data
' here the data is printed for demonstration
print "The data to be transmitted"
printstr(txd$)

' check the recieved CRC and data
 c% = math(crc16 txd$,,&h8005,&hffff,0,1,1)
check$ = CHR$(c% AND &HFF) + CHR$((c%>>8) AND &HFF)
print
print "The CRC of the recieved data including the"
print "recieved CRC - the result should be zero"
printstr(check$)

' Print a string as HEX numbers (for debug)
sub PrintStr(b$)
  for i = 1 to len(b$)
    print Hex$(asc(mid$(b$,i,1)),2); ", ";
  next i
  print
end sub
```

The CRC value is transmitted along with the data to the receiver. The receiver can verify the received data by removing the CRC then recalculate the CRC and compare that with the received CRC Or, more simply, recalculate the CRC for the whole received message and verify that the result is zero as demonstrated above. If the CRC does not check then the receiver should reply with a negative acknowledgement and request a re-transmission of the data.

The MMBasic CRC function:

MATH(CRCn data [,length] [,polynome] [,startmask] [,endmask] [,reverseIn] [,reverseOut]

Please see the entry in the Functions table. Some of the parameters used in the calculation of the CRC are not explained but their purpose may be made clear in the fullness of time. In the meantime here are some examples for commonly used CRC calculations from Volhout's demonstration program that may be useful.

```
' CCITT CRC16
CRC% = math (crc16 data$,,,&hffff)

' MODBUS CRC16
CRC% = math (crc16 data$,,&h8005,&hffff,0,1,1)

' XMODEM CRC16
CRC% = math (crc16 data$,)

' MAXIM CRC8
CRC% = math (crc8 data$,,&h31,,,1,1)

' standard CRC32
CRC% = math (crc32 data$,,,&hffffffff,&hffffffff,1,1)
```

Demonstration program

```
' MATH CRC Demonstration program
' Based on the program "MATH CRC evaluation"
' written by Volhout
option base 1

'test string
a$="123456789"
l%=len(a$)
print "Test string "; a$
print

'convert test string to array
dim b%(l%)
for i=1 to l% : b%(i)=asc(mid$(a$,i,1)) : next i

'perform CRC validation
dim CRC%

'check CCITT CRC16
CRC% = math (crc16 b%(),l%,,&hffff)

print "CRC16-CCITT  "; hex$(CRC%)

'check MODBUS CRC16
CRC% = math (crc16 b%(),l%,&h8005,&hffff,0,1,1)
print "CRC16-MODBUS "; hex$(CRC%)

'check XMODEM CRC16
CRC% = math (crc16 b%(),l%)
print "CRC16-XMODEM "; hex$(CRC%)

'check MAXIM CRC8 (used in DALLAS single wire devices)
CRC% = math (crc8 b%(),l%,&h31,,,1,1)
print "CRC8-MAXIM   "; hex$(CRC%)

'check standard CRC32
CRC% = math (crc32 b%(),l%,,&hffffffff,&hffffffff,1,1)
print "CRC32        "; hex$(CRC%)
```

Some useful links:

The author of this CRC code

https://github.com/RobTillaart/CRC

Explanations of CRCs

http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#ch1

https://en.wikipedia.org/wiki/Cyclic_redundancy_check

On line CRC calculators

http://zorc.breitbandkatze.de/crc.html

https://crccalc.com

https://www.lddgo.net/en/encrypt/crc