

Colour Maximite 2

User Manual

MMBasic Ver 7.00.00

Matches firmware 7.00.00b2

Geoff Graham

(21 October 2026)

For updates to this manual and more details on MMBasic go to
<http://geoffg.net/maximite.html>
and <http://mmbasic.com>

About

The Colour Maximite 2 was conceived and developed by Peter Mather (matherp on the Back Shed Forum) who also led the development project. It is based on the Colour Maximite developed by Geoff Graham and uses the MMBasic interpreter written by Geoff Graham (<http://geoffg.net>).

A team of people from around the world assisted with testing, advice and developing some initial games and programs. These are Phil Boyce, Jim Hiley, Graeme Rixon, Robert Severson and Mauro Xavier.

Version 6.00.00 of the firmware was developed by Gerry Allardice. Version 7.00 brings the MMBasic of the PicoMite version 7 to the Colour Maximite 2.

Support

Support questions should be raised on the Back Shed forum (<http://www.thebackshed.com/forum/Microcontrollers>) where there are many enthusiastic Maximite, Micromite and PicoMite users who would be only too happy to help. The developers of both the Colour Maximite 2 and MMBasic are also regulars on this forum.

Copyright and Acknowledgments

The Colour Maximite 2 firmware and MMBasic is copyright 2011-2026 by Geoff Graham, Peter Mather and Gerry Allardice.

1-Wire Support is copyright 1999-2006 Dallas Semiconductor Corporation and 2012 Gerard Sexton.

FatFs (SD card and USB drive) driver is copyright ChaN.

TinyUSB (the USB host) is copyright tinyusb.org.

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

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

JPG support is thanks to Rich Geldreich.

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

Marcel Rodrigues wrote the GIF decoder.

Maury Quijada wrote the image resize and image rotate code.

cJSON is copyright Dave Gamble and cJSON contributors.

The CRC calculations are copyright Rob Tillaart.

Thomas Williams and Gerry Allardice for MMBasic enhancements.

STM32 drivers and software components are copyright STMicroelectronics.

The compiled object code (the .bin file) for the Colour Maximite 2 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

Copyright 2026 Geoff Graham and Peter Mather

The author of this manual is Geoff Graham. Version 6.00.00 updates are by Gerry Allardice and version 7.00 updates by Peter Mather, using the PicoMite manual for the features the two computers share. It is distributed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0).

Contents

Introduction	10
What is New in Version 7.00.....	11
Quick Start Tutorial	12
Command Prompt	12
Your First Program.....	12
Something More Complicated.....	13
Colour Maximite 2 Generation 1	15
External Oscillator.....	16
Mouse Interface.....	16
Colour Maximite 2 Generation 2	17
Mouse Interface.....	17
Power Switch Orientation.....	18
High Accuracy Real Time Clock.....	18
ESP-01 Module	18
Comparing Page Buffers between Gen 1 and Gen 2.....	18
Hardware Features	19
STM32H743IIT6 Cortex-M7 32-bit RISC CPU.....	19
VGA Monitor Connector.....	19
USB Connector	19
Power & Serial Console Connector.....	19
Mouse Support	20
Audio Connector	20
Power & SD Card Activity LEDs.....	20
SD Card Connector	20
External I/O Connector.....	21
Wii Nunchuk and Wii Classic Connector.....	22
Temperature Sensor.....	22
IR Receiver.....	23
Firmware Upload Select.....	23
Backup Battery	23
Reset Switch.....	23
Serial Console.....	24
USB Serial Port	24
Terminal Emulator	24
The Console	24
Apple Macintosh	25

Linux	25
Android	25
Using MMBasic	26
Commands and Program Input	26
Editing the Command Line.....	26
Console Keyboard/Display.....	26
Shortcut Keys	26
Line Numbers and Program Structure	27
All Programs Are Run From the SD Card	27
Status Line.....	28
Running Programs.....	28
Expressions	28
Standards and Compatibility	28
Setting Options	28
Saved Variables.....	29
Timing.....	29
Watchdog Timer.....	29
PIN Security	29
The Serial Console	30
Resetting MMBasic.....	30
The Library.....	30
Program Initialisation	31
MM.STARTUP	31
MM.PROMPT.....	32
MM.END	32
Full Screen Editor	33
Edit Functions	33
Additional Keys.....	34
Saving	34
Mark Mode.....	35
Alternate Keys.....	35
Long Lines	35
Using a Mouse.....	35
Colour Coded Editor Display	35
File Manager	37
Variables and Expressions.....	39
Variables	39
Constants.....	39
OPTION DEFAULT	39
OPTION EXPLICIT.....	40
DIM and LOCAL	40
STATIC.....	41
CONST.....	41
Special Characters in Strings	42
Expressions and Operators	42
Mixing Floating Point and Integers	43
Structures	43

64-bit Unsigned Integers	44
Subroutines and Functions.....	45
Subroutines.....	45
Functions.....	45
Passing Arguments by Reference	46
Passing Arrays.....	46
Early Exit	47
Recursion	47
Examples.....	47
Video Modes and Pages	49
Resolutions.....	49
Colour Depths	49
Pages	50
Video Memory	50
The PicoMite's 16 Colour Display (MODE n,4).....	51
Image Slots.....	51
Running PicoMite Programs	51
Keyboard, Mouse and Gamepad.....	52
USB Hub.....	52
USB Keyboard	52
USB Mouse.....	52
PS2 and I2C Mice	53
USB Gamepads	53
USB Serial Devices (COM4 to COM7).....	53
USB Flash Drive	53
Wii Nunchuk and Classic Controllers	54
Program and Data Storage	55
Drives.....	55
Program Management Commands	55
File Access Within a Program	55
File and Directory Management	56
Play Audio Files	56
Load and Save Images.....	57
Sequential File Access.....	57
Random File Access	58
Sound Output.....	60
Playing WAV, FLAC, MP3 and MOD Files	60
Background Music	60
Generating Sine Waves	60
Specialised Audio Output.....	60
BBC Micro Sound.....	61
Using PLAY.....	61
Utility Commands	62
Using the I/O Pins	63
Digital Inputs.....	63
Analog Inputs	63

Counting Inputs	63
Digital Outputs	64
Pulse Width Modulation.....	64
Communications Interfaces (Serial, SPI, I2C and CAN).....	64
Interrupts	64
Special Device Support	67
Infrared Remote Control Decoder	67
Infrared Remote Control Transmitter	67
Measuring Temperature.....	68
Measuring Humidity and Temperature	68
Measuring Distance	69
WS2812 Support	69
Hobbytronic Mouse Support	70
CAN Support.....	70
Graphics Functions	71
Screen Coordinates.....	71
Read Only Variables	71
Colours	71
Fonts.....	72
Embedded Fonts	72
Drawing Commands.....	73
Example of Basic Graphics	74
Rotated Text	75
Transparent Text	75
Displaying Images.....	75
Advanced Graphics Tutorial.....	75
3D Engine	75
GUI Controls	77
Using a Mouse.....	77
Game Playing Features	78
VGA Resolution, Colour Depth and Pages.....	78
Scrolling, Sprites and Tile Maps	78
Displaying Images.....	78
Fonts.....	79
Playing Audio.....	79
Keyboard Keys.....	79
Wii Nunchuk and Classic Controllers	79
USB Gamepads	79
Long Strings.....	80
Long String Variables	80
Long String Commands.....	80
Long String Functions	81
Porting Programs	82
Running PicoMite Programs	82
Porting from the Original Colour Maximite	82
Variables	82
Floating Point.....	82

Graphic Commands.....	82
Fonts.....	83
Sprites	83
SOUND and TONE.....	84
SD Card and File Related Commands	84
Special Devices	84
CONFIG Commands.....	84
Error Handling.....	84
Random Number Generator.....	84
MMBasic Characteristics	85
Naming Conventions.....	85
Constants.....	85
Implementation Characteristics	85
MMBasic Firmware Memory Map.....	86
Compatibility.....	87
Predefined Read Only Variables	88
Detailed Listing	88
Operators	94
Numeric Operators (Float or Integer)	94
String Operators	94
Options	95
Detailed Listing.....	95
Commands	106
Detailed Listing.....	106
Functions	186
Detailed Listing	186
Obsolete Commands and Functions	210
Detailed Listing	210
Appendix A – Serial Communications	211
Serial Communications	211
The OPEN Command.....	211
Input/Output Pin Allocation	212
Examples.....	212
Reading and Writing	212
Interrupts	213
Low Cost RS-232 Interface	213
Appendix B – I2C Communications	214
I2C Communications.....	214
Keeping the Bus	214
7-Bit Addressing	215
I/O Pins	215
Example	215
Appendix C – 1-Wire Communications.....	216
1-Wire Communications	216

Appendix D – SPI Communications	217
SPI Communications	217
I/O Pins	217
SPI Open	217
Transmission Format	217
Standard Send/Receive	217
Bulk Send/Receive	218
SPI Close	218
Examples	218
Appendix E – Regular Expressions	219
Using Regular expressions with OPTION ESCAPE	220
Appendix F – Sprites	221
Sprites	221
Static Objects and the Background	222
Appendix G – Turtle Graphics	224
Turtle Graphics	224
Appendix H – Cyclic Redundancy Check (CRC)	226
Cyclic Redundancy Check (CRC)	226
Using a CRC	226
The MMBasic CRC Function	227
Demonstration Program	227
Appendix I – Special Keyboard Keys	229
Special Keyboard Keys	229
Appendix J – CAN Support	230
CAN Support	230
CAN OPEN index, speed, mode	230
CAN FILTER GLOBAL id, eid, idr, eidr	231
CAN FILTER index, eid, type, config, id1, id2	232
CAN START	232
CAN STOP	232
CAN SEND	232
CAN READ	232
CAN CLOSE	233
Identifying a Node	233
Appendix K – Loading the Firmware	234
Loading the Firmware	234
Alternative Method	236
Third Method	237
Linux and the Raspberry Pi	237
Appendix L – Programming in BASIC - A Tutorial	238
Command Prompt	238
Structure of a BASIC Program	238
Comments	239
The PRINT Command	239
Variables	240
Expressions	241

The IF Statement	241
FOR Loops	243
Multiplication Table	243
DO Loops	244
Console Input	245
GOTO and Labels	246
Testing for Prime Numbers	247
Arrays	248
Integers	249
Strings	250
Manipulating Strings	251
Scientific Notation	252
DIM Command	252
Constants	253
Subroutines	253
Functions	255
Local Variables	256
Static Variables	256
Calculate Days	257
Appendix M – Firmware Applicability	260
Appendix N – Where Each Keyword Is Documented	261
Appendix O – Companion Manuals	268

Introduction

The Colour Maximize 2 is a small self contained computer inspired by the home computers of the early 80's such as the Tandy TRS-80, Commodore 64 and Apple II. It includes its own BASIC interpreter and powers up in under a second into the BASIC interpreter (there is no operating system to boot). The emphasis is on ease of use and, as a result, a first time user could enter a small program and have it running within minutes.

While the concept is borrowed from computers of the 80's the technology used is very much up to date. The CPU that powers the Colour Maximize 2 is an ARM Cortex-M7 32-bit RISC processor running at up to 480MHz with 2MB flash memory and 1MB RAM. This processor includes its own video controller and generates a VGA output at resolutions up to 1920x1080 pixels and with up to 24-bit colour.

The Colour Maximize 2 is designed to be simple and fun. The assembly instructions and the firmware are completely free and the parts can be found from multiple sources. It can be assembled in an hour or two and will provide endless hours of fun.

The basic features of the Colour Maximize 2 are:

- **Low cost affordable fun.** The firmware (including the BASIC interpreter) is completely free. The main PCB is easy to assemble with thru hole components. The CPU and support circuits are contained on a fully assembled plug in board costing US\$30. The firmware can be loaded using free software so a programmer or special equipment is not required to get started.
- **Instant startup into the BASIC interpreter.** Program space is 516KB, enough for huge and complex programs (typically 25,000 lines or more), while the general RAM used for arrays and buffers is over 5MB on the Generation 1 and over 24MB on the Generation 2 (enough for enormous arrays).
- **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, user defined structures, extensive string handling and user defined subroutines and functions. Typically it will execute a program at 270,000 lines per second.
- **Rock solid VGA output** (or HDMI with an inexpensive converter), with 18 program selectable video modes from 1920x1080 pixels to 240x216 pixels and up to 24-bit colour (16 million colours).
- **USB keyboard, mouse and gamepads.** The keyboard can be wireless (with a USB dongle) or wired and have UK, US, DE, FR, ES or BE key mappings. With a USB hub the same socket also takes a mouse, gamepads, USB serial devices and a USB flash drive. A PS2 mouse or a dual USB/PS2 mouse can be used as well.
- **Stereo audio output** can play WAV, FLAC and MP3 files, computer generated music (MOD format), BBC Micro style sound, and robot speech and sound effects as well as generate precise sine wave tones.
- **A full screen editor** is built into the firmware. It includes advanced features such as colour coded syntax, search and replace, cut, copy and paste with a clipboard, lines longer than the screen, automatic indenting and full mouse support. With one key press the program can be saved and run. If an error occurs the EDIT command will return to the editor with the cursor placed on the line that caused the error.
- **Full support for SD cards** including editing and running programs on the SD card as well as opening files for reading, writing or random access. Cards up to 128GB formatted in FAT16, FAT32 or exFAT are supported and the files can also be read and written on personal computers running Windows, Linux or the Mac operating system. A USB flash drive can be used as a second drive (C:). A two-panel file manager (FM) is included in MMBasic.
- **Programs can be easily transferred** from another computer (Windows, Mac or Linux) using the SD card, a USB flash drive, the XModem or ZModem protocols or by streaming the program over the serial console input.
- **Extensive features for creating computer games.** These include multiple video planes, support for Blits, Sprites and tile maps, the ability to create computer generated music, sound effects and computer generated speech. The Colour Maximize 2 includes full support for the Wii Nunchuk and Wii Classic games controllers and for USB gamepads.
- **Battery backed clock** will keep the correct time, even with the power disconnected.
- **Twenty eight input/output pins** with 12 capable of analog input. Built in support for an IR remote control and temperature and humidity sensors. Communications protocols include I2C, asynchronous

serial, RS232, IEEE 485, SPI, 1-Wire and CAN. These can be used to communicate with many sensors (temperature, humidity, acceleration, etc) as well as for sending data to test equipment.

- **Power** is 5 volts at 220mA typical from a USB port or charger.

What is New in Version 7.00

Version 7.00 brings MMBasic on the Colour Maximite 2 up to date with MMBasic version 7 on the PicoMite. The language is the same on both computers, apart from the hardware that one of them does not have, so most programs written for the PicoMite can be run on the Colour Maximite 2 with few changes (see the chapter Porting Programs). In particular:

- **The interpreter** is the PicoMite's version 7. It runs programs faster and adds user defined structures (TYPE), a program library (LIBRARY), CHAIN, SAVE CONTEXT and LOAD CONTEXT, single element arrays, the full MATH command and many other commands and functions.
- **USB devices** use the PicoMite's USB support: a hub, mice, gamepads (including PlayStation, Xbox and Nintendo Switch Pro controllers), USB serial devices as COM4 to COM7 and a USB flash drive as the C: drive.
- **The file manager (FM)** is the PicoMite's two-panel file manager, which copies between the SD card and a USB flash drive.
- **The full screen editor** is the PicoMite's, with the Colour Maximite 2's mouse support and extra keys.
- **Files** can be transferred with the ZModem protocol as well as XModem.
- **Graphics** gain the PicoMite's tile maps (TILEMAP), image slots and its 16 colour display (MODE n,4), so that games written for the PicoMite need few changes.

This manual is for version 7.00.00b2, the second beta of version 7.00. Chapters that have not yet been revised line by line for this version say so at their start.

Quick Start Tutorial

The following assumes that you have built or purchased the Colour Maximite 2, loaded the firmware, attached a keyboard and VGA monitor, powered it up and tested that it runs. At that point you should have the command prompt (a greater than symbol ">") displayed on the monitor.

Command Prompt

Most interaction with MMBasic is done via the console at the command prompt (ie, the greater than symbol (>) on the console). 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 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.

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 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
```

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 the Colour Maximite's 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)  
 516K (100%) Free  
  
Data:  
  0K ( 0%) 0 Variables  
  0K ( 0%) General  
24288K (100%) Free
```

(The free data memory depends on the Colour Maximite 2: this is a Generation 2.)

What is the current time?

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

What is the current date?

```
> PRINT DATE$  
20-10-2026
```

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
```

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:

- Make sure that you have a properly formatted SD card inserted into the SD card slot.
- 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 on your keyboard. This tells the editor to save your program and exit to the command prompt.
- At the command prompt type RUN "hello.bas" followed by the Enter key.
- You should see the message: Hello World

Congratulations. You have just written and run your first program on the Colour Maximite 2. 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 5
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 below.



If there was an error you will get a message with the line number and a description of the error. If you then 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 and 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 the Colour Maximite 2 can really go you could go back into the editor and change that line to PAUSE 0 and then rerun the program.

The program is written for the default video mode, MODE 1, which is 800 x 600 pixels.

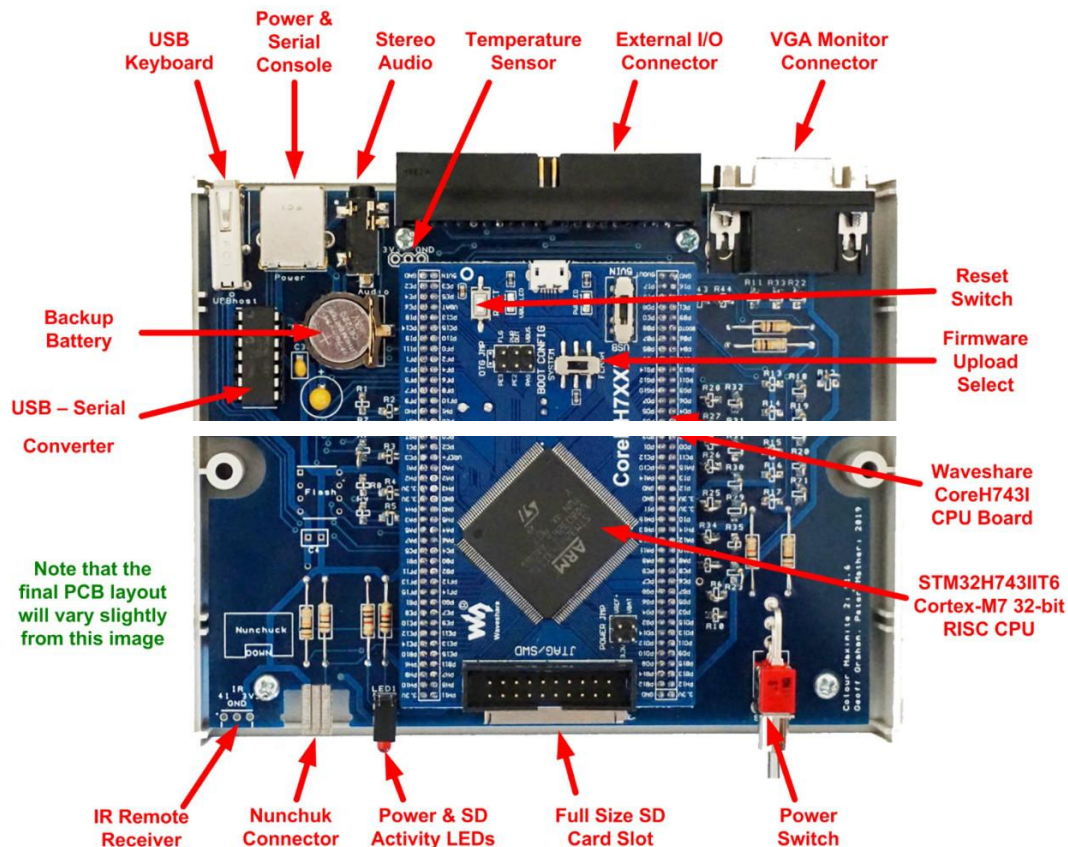
For a more in-depth tutorial and a description of programming in BASIC you should read Appendix L (Programming in BASIC - A Tutorial) at the end of this manual, and download Programming with the Colour Maximate 2 which can be found at <http://geoffg.net/maximate.html> (scroll to the bottom of the page).

Colour Maximite 2 Generation 1

This version uses a plug in CPU module made by Waveshare. The module carries the ARM Cortex-M7 32-bit RISC processor, 8MB SDRAM and various support components. The plug in board concept eliminates the need to solder the ARM CPU with its 176 pins and 0.2mm gap between pins and is therefore suitable for home construction.

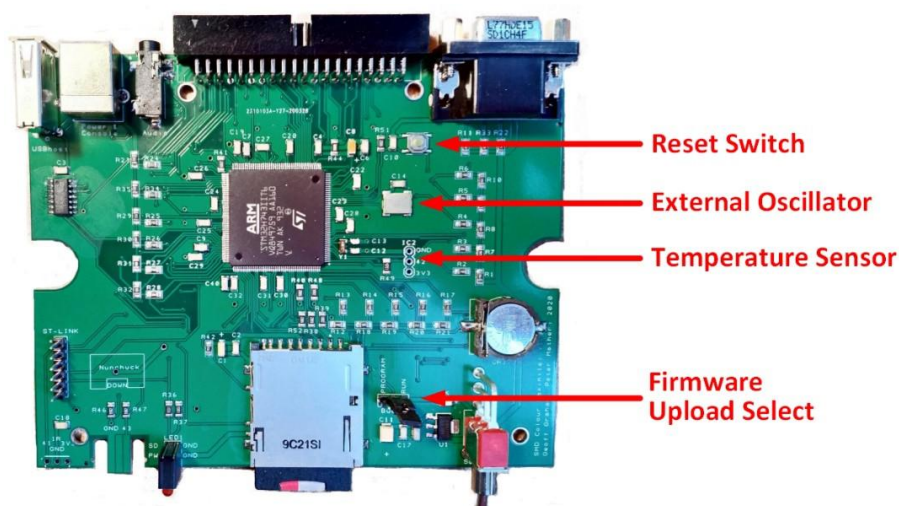
The CPU module has two connectors on its top, one a micro USB the other a multi pin header. Both of these are only needed for firmware development and are not used in normal operation as a BASIC computer. For normal operation all the jumpers on this board should be removed, the power switch should be set to "5VIN" and the BOOT CONFIG switch should be set to "Flash".

The following image illustrates the major connectors and features of this design:



Many suppliers are offering a fully assembled version of this design. These do not use the plug in module concept, instead the ARM Cortex-M7 processor and its supporting components are soldered directly to the main PCB. This makes sense for a machine assembled board and is cheaper to supply.

The layout of this board is essentially the same except that there are no jumpers as on the Waveshare board. Some items are positioned differently as noted in the image below.



External Oscillator

The main processor clock comes from an 8 MHz crystal on the Waveshare board which is driven by the on chip oscillator in the ARM processor. This arrangement can have some jitter on the clock which in turn may cause jitter or instability in the VGA image. However, in most cases, this design works well with the maximum 800x600 pixel VGA video resolution provided by the early firmware releases.

However, with firmware upgrades, it is now possible to generate a VGA image at much higher resolutions and these may display an instability caused by this jitter. The solution is to fit an external 8MHz oscillator to replace the crystal and oscillator used by the Waveshare board. If the higher resolutions are not required then, in most cases, this modification will not be required. This issue does not affect the single board versions (ie, fully assembled without the Waveshare module) as they use an external oscillator module.

Note that the following will require a level of skill in soldering surface mount parts. The following parts are required:

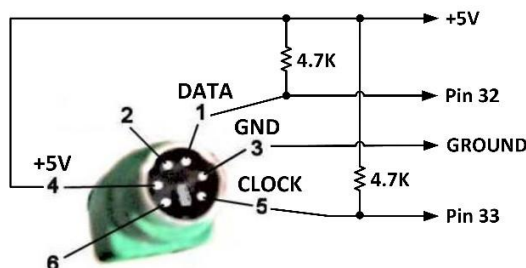
- 1 x SMD 1206 100nF ceramic capacitor
- 1 x 8MHz oscillator in 5.0 x 7.0 mm SMD package (QX7 XO 25ppm). Eg, RS Stock Nbr 813-6194.

The solder pads for these are located under the Waveshare board near the left hand 80 pin connector. The oscillator has a dot identifying pin 1 and this needs to be aligned with a small dot on the PCB silkscreen. Take care to not damage the 80-pin connector nearby and use the minimum of solder so that the solder does not short to the case of the oscillator. The capacitor should then be installed near the oscillator.

Nothing more is required and it is not necessary to remove the 8 MHz crystal from the Waveshare board. On power up the Colour Maximite 2 should operate as normal.

Mouse Interface

A PS2 mouse or a dual USB/PS2 mouse with a PS2 adapter can be connected via the rear I/O connector. The mouse CLOCK (pin 5) connects to pin 33 of the I/O connector and mouse DATA (pin 1) connects to pin 32. Both must have 4.7K Ω pullup resistors to +5V. The mouse GND and +5V pins should be connected to the ground and 5V pins on the I/O connector.



A USB mouse can also be plugged into the USB socket on the front panel (see USB Connector in the chapter Hardware Features).

MMBasic has built in support for the mouse via the `MOUSE()` function. The mouse can also be used in the editor to select text, etc. A full description is provided in the chapter Hardware Features.

The other features of this board are common with the Colour Maximite 2 Generation 2 and are described in the chapter Hardware Features.

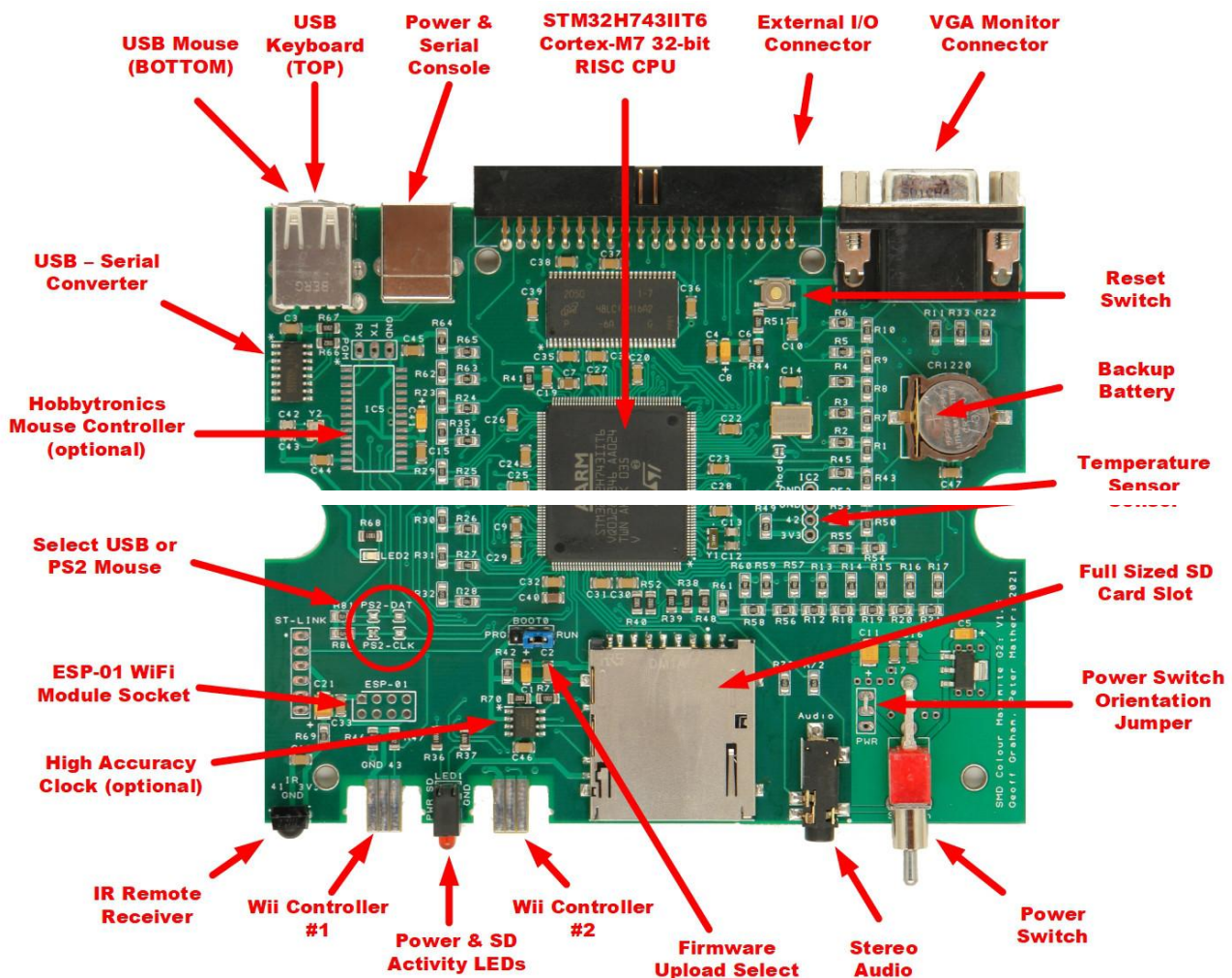
Colour Maximite 2 Generation 2

This is an upgraded design with three main improvements over the first generation:

- It uses a larger RAM chip (32MB vs 8MB) which is used to support the top VGA resolution of 1920x1080 pixels, 24-bit colour depth and provide more RAM for BASIC programs.
- An 8-bit digital to analog converter is used for each colour channel allowing the Generation 2 computer to display true colour 24-bit images.
- The PCB is a 4 layer design with all components mounted on the top side. This makes it more suitable for machine assembly.

This version also has connectors for a second Wii controller and a PS2 mouse and the rear I/O connector is inverted to make it more convenient to use connectors and cables designed to suit the Raspberry Pi Hats.

This diagram lists the key hardware features:



Mouse Interface

The Generation 2 design supports two types of mouse interface on its rear USB mouse socket (the lower socket). The first is a full USB mouse and for this to be implemented a Hobbytronics mouse controller chip (<https://www.hobbytronics.co.uk/usb-host-soic>) needs to be installed at the position for IC5 along with the associated supporting components. The mouse then connects to the USB mouse socket on the rear of the Colour Maximite 2.

The second type of support is for a dual USB/PS2 mouse. Most wired mice will automatically switch between USB and PS2 modes and many come with a USB to PS2 adapter. This adapter simply signals the mouse to switch to PS2 mode via a pull-up resistor and provides the physical PS2 connector. A typical example that has been tested on the Colour Maximite 2 is the Microsoft Basic Optical Mouse which is low cost and widely available.

To use a dual USB/PS2 mouse IC5 must not be populated, and all four solder jumpers marked "select USB or PS2 mouse" (diagram above) must be joined with a solder blob. The mouse then plugs into the USB mouse socket (the lower socket) on the rear of the Colour Maximite 2. A USB to PS2 adapter is not required (the Colour Maximite 2 will automatically switch the mouse into PS2 mode).

A USB mouse can also be plugged into the USB keyboard socket on the front panel, without the Hobbytronics chip (see USB Connector in the chapter Hardware Features).

MMBasic has built in support for a mouse via the MOUSE() function. The mouse can also be used in the editor to select text, etc. A full description is provided in the chapter Hardware Features.

Power Switch Orientation

Near the power switch there is a three pin jumper marked PWR which is used to select the orientation of the power switch. If the two pins towards the back of the Colour Maximite 2 are connected the power switch will operate as expected in the UK, Australia and New Zealand (down is on). If the two pins towards the front of the computer are connected the power switch will operate as expected in North America (up is on).

High Accuracy Real Time Clock

The basic design uses a miniature 32KHz crystal and the real time clock hardware built onto the ARM Cortex-M7 processor to keep the date and time. Optionally you can add an external DS3231MZ+ real time clock to the PCB (illustrated in the above photo) and it will be kept alive by the coin battery on the PCB. This clock will typically lose or gain only a couple of seconds in a week.

To enable this time source the command is: OPTION DS3231 ON

Once enabled the time and date can be set and retrieved from the DS3231MZ+ using the normal MMBasic DATE/TIME commands and functions.

ESP-01 Module

There is a footprint on the PCB for connecting an ESP-01 WiFi module to COM2. The firmware allows COM2 to be used as a serial console (OPTION CONSOLE PORT) so if the correct USB/UART code is loaded into the ESP-01 it will be possible to access the console of the Colour Maximite 2 remotely.

MMBasic does not include any Internet functions using this device but communications can be established using the normal serial communications protocol included in MMBasic.

Comparing Page Buffers between Gen 1 and Gen 2

Mode	Resolution	Ratio	Gen 1 (8MB) pages			Gen 2 (32MB) pages			
			8 bit	12 bit	16 bit	8 bit	12 bit	16 bit	32 bit
1 (def)	800x600	4:3	6	2	2	15	6	6	2
2	640x400	4:3	13	6	6	29	14	14	6
3	320x200	4:3	53	24	25	60	56	57	27
4	480x432	4:3	15	7	7	35	17	17	7
5	240x216	4:3	60	29	30	60	60	60	33
6	256x240	4:3	53	25	26	60	59	60	28
7	320x240	4:3	41	18	19	60	45	46	20
8	640x480	4:3	10	4	4	23	10	10	4
9	1024x768	4:3	3		1	8		3	
10	848x480	16:9	7	2	2	17	7	7	3
11	1280x720	16:9	2		0	6		2	
12	950x540	16:9	3		0	11		4	
13	400x300	4:3	26	10	11	60	27	28	12
15	1280x1024		1			4			
16	1920x1080	16:9				2			
17	384x240		34	14	15	60	37	38	16

The other features of the Generation 2 board are common with the first generation Colour Maximite 2 and are described in the chapter Hardware Features.

Hardware Features

The Colour Maximite 2 is based on an ARM Cortex-M7 32-bit processor which provides nearly all of the services for the user. This includes the flash memory (where the BASIC interpreter is installed), RAM and a sophisticated video display controller which generates the high quality VGA image with many advanced features useful to games programmers.

STM32H743IIT6 Cortex-M7 32-bit RISC CPU

This is a high performance 32-bit ARM processor with 2MB of flash and 1MB of RAM. It includes a sophisticated video display controller which is used to generate a stable VGA output. The MMBasic firmware is loaded onto the flash memory of this chip and provides the Colour Maximite 2 with its personality.

There are two versions of this chip, Rev Y with a maximum speed of 400MHz and Rev V with a speed of 480MHz. Most vendors do not allow you to specify the version in advance so, depending on the supply pipeline, you may get either. MMBasic will automatically accommodate either revision and you can tell which one you have with the MM.INFO(CPUSPEED) function which will return 400000000 for the Rev Y chip and 480000000 for the Rev V chip. Other than this the only difference in the Colour Maximite 2 is that MODE 16 (1920 x 1080 pixels) needs a Generation 2 with the Rev V chip.

VGA Monitor Connector

This is the main video output and it generates standard VGA signals in a variety of resolutions and number of colours as determined by the MODE command. These range from 800x600 pixels (the default at power up) up to 1280x1024 and down to 240x216 pixels with 256, 4096 or 65536 colours. Most modes 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). The Generation 2 Colour Maximite 2 can go to 1920x1080 pixels and 24-bit colour depth at some resolutions.

Note that on first setup some monitors may truncate the text on the margins or show an image that seems to shimmer or flicker. In most cases this can be fixed by pressing the auto setup button on the monitor or, failing that, using the monitor's image setup mode to adjust parameters such as the clock, phase and position.

If an HDMI output is required it is recommended that an inexpensive VGA to HDMI converter be used. These cost about US\$10+ on eBay and will also encode the audio from the computer so that it can play through the monitor's speakers.

USB Connector

This is the Type A USB connector on the front panel. It accepts a standard USB keyboard, including most that have a wireless dongle for the USB connection and keyboards with a built-in mouse or hub. With a USB hub it also accepts:

- a USB mouse, which is mouse channel 2 (see Mouse Support below);
- up to two USB gamepads, which are game controller channels 3 and 4 (see the chapter Keyboard, Mouse and Gamepad);
- USB serial devices, such as USB to serial adapters and GPS receivers, which appear as the serial ports COM4 to COM7;
- a USB flash drive, which appears as drive C:.

The processor's USB controller cannot work with a low-speed USB device through a hub. Many inexpensive keyboards and mice are low-speed devices. Such a device works when it is plugged straight into the socket; through a hub MMBasic refuses it with the message "USB low-speed device behind a hub is not supported: plug it in directly". Full-speed keyboards and mice work through a hub.

When a keyboard is connected the SD card activity LED will be illuminated. On the first access of the SD card the LED will revert to its normal action (illuminated during SD card access).

The firmware can accommodate a number of different keyboard layouts: UK, US, German (DE), French (FR), Spanish (ES) and Belgian (BE). The layout is asked for when MMBasic is first loaded and can be changed with the OPTION USBKEYBOARD command.

Power & Serial Console Connector

This USB B connector is for power and the serial console over USB. The power requirement is 5V at 160mA to 250mA (typical). This is within the capabilities of most USB chargers however some PCs (especially older

laptops) may have trouble supplying this. If your Colour Maximite 2 is suffering from intermittent issues such as reboots, errors reading the SD card, etc then it would be worth changing the power supply to one with a much higher capacity (for example, 2 amps or more).

The serial console is available if the Colour Maximite 2 is connected via this port to a personal computer: it appears there as a normal serial port, at 115200 baud. The console is used to write and debug BASIC programs and configure the computer. How to connect to it and set up a terminal emulator is described in the chapter Serial Console.

Mouse Support

A USB mouse plugged into the USB socket on the front panel (directly, or through a hub if it is a full-speed mouse) is mouse channel 2. It needs no configuration: the editor uses it whenever it is connected. `OPTION MOUSE SENSITIVITY n` divides its movement by `n`: a number above 1 slows the mouse down and a number below 1 speeds it up (0 or 1 leaves it as it is).

A PS2 mouse or the Hobbytronic USB mouse interface is configured for the editor with this command (use channel 0 for a PS2 mouse):

```
OPTION MOUSE channel [,sensitivity]
```

The optional sensitivity parameter allows you to tune the sensitivity of the Hobbytronic interface by specifying a number between 0 and 10. Zero indicates that the default automatic sensitivity should be used while other values will explicitly set the sensitivity as per the Hobbytronic documentation.

For a PS2 mouse the sensitivity can be set between 0 and 8. Values of 1-4 set the resolution of the mouse to 1, 2, 4 or 8 counts/mm. Values 5-8 do the same but in addition enable mouse scaling which gives a non-linear relationship between speed and count (note that not all PS2 mice implement these settings).

`OPTION MOUSE OFF` stops the editor using a PS2 or Hobbytronic mouse.

Within the editor the mouse works as you would expect. For example, you can use the mouse to point and select text, scroll using the scroll wheel, and so on. For the details see the chapter Full Screen Editor.

Within a MMBasic program the mouse can be enabled for access with various sensitivity settings and interrupts on left and/or right clicks. Specialised built in functions can be used to determine the current mouse position in pixels and detect single or double clicks.

Audio Connector

This is a 3.5mm stereo phono socket. The tip is the left channel, the ring is the right channel while the sleeve is ground. The signal level at full volume is about 1V RMS (approx 3V peak to peak). MMBasic can generate audio in many formats ranging from simple sine wave tones through to playing WAV, FLAC, and MP3 files.

Note:

- The output is high impedance suitable for feeding into an amplifier. It cannot directly drive a loudspeaker, headphones or any low impedance load and might be damaged if that was attempted.
- If the audio is garbled and/or lacks normal bass it might indicate that your amplifier has a low input impedance. Try adding a 4.7K resistor in series with the output (the Generation 2 board has these permanently installed).
- There is a DC offset on the output. Most amplifiers have an input capacitor so this has no effect but, if not, it may be necessary to add a 4.7µF capacitor in series with the output (positive leg to the Colour Maximite 2).

Power & SD Card Activity LEDs

The power indicator (green) is illuminated whenever power is applied.

The SD Card activity indicator (red) will illuminate briefly when reading or writing from/to the SD card. The SD card should not be removed when this LED shows activity. The red LED will also illuminate when a USB keyboard is plugged in and successfully communicates with MMBasic - it then reverts to normal activity on the first read/write to the SD card.

SD Card Connector

The SD card connector on the front panel is the main storage for the Colour Maximite 2. All programs reside on the SD card so it must be present for most operations. This is different from the original Colour Maximite where the SD card was not necessarily required.

The Colour Maximite 2 will support cards up to 128GB. Cards larger than 32GB should be formatted as exFAT and cards 32GB or less formatted as FAT32. Note that very small cards may not be reliable and exFAT is much slower than FAT32. BASIC programs are quite small so an 8GB or 16GB card formatted as FAT32 is ample and also readily available at a cheap price.

The SD card is drive A:. It can also be called B: (as the PicoMite names its SD card). A USB flash drive is drive C:.

External I/O Connector

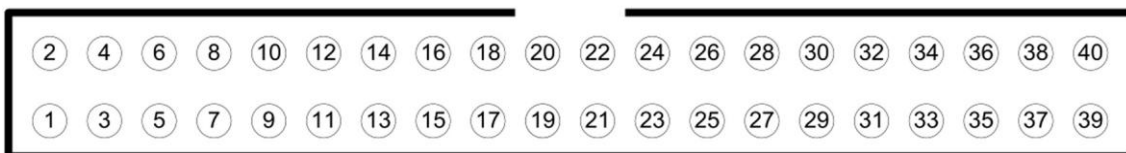
This 40-pin ribbon connector provides 28 input/output pins which can be controlled from within the BASIC program plus 3.3V and 5.0V outputs for powering external circuitry plus a number of ground pins.

The total current drawn from all 3.3V pins should be limited to 100mA. Exceeding this may cause the voltage regulator on the CPU module to enter thermal shutdown. The capacity of the 5V supply is limited only by your USB power supply.

If more input/output pins are required an I/O expander such as the Microchip MCP23017 can be used.

Colour Maximite 2 Generation 1

This is the image of the connector as seen from the rear of the first generation Colour Maximite 2:



These numbers are used as reference in the table below and by MMBasic to refer to an I/O pin. They are not the connector manufacturer's standard pin numbers.

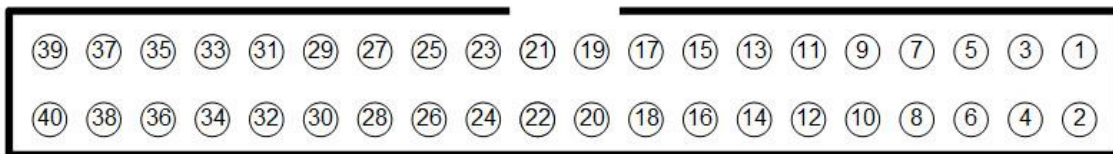
The pin layout and positioning of special functions on the external I/O connector is compatible with the Raspberry Pi allowing Pi HATs to be connected if required.

Note that most Raspberry Pi cables for use with solderless breadboards need to be inverted when plugged into the Colour Maximite 2's rear I/O connector (ie, the key tab on the connector needs to be on the bottom). If this is not done the pin legends on the breadboard connector will be wrong. This is not a problem if the Maximite connector is unshrouded otherwise you may need to cut a new keyway on the bottom of the shroud with side cutters (not hard to do).

Colour Maximite 2 Generation 2

The second generation Colour Maximite 2 has these pins reversed so that the issue described above (inverted connector) is avoided. On this version Raspberry Pi cables can plug directly in - there is no need to cut a second keyway.

This is the image of the connector as seen from the rear of the second generation Colour Maximite 2:



These are the connector manufacturer's standard pin numbers. These numbers are used as reference in the table below and by MMBasic to refer to an I/O pin.

I/O Pin Capabilities

The capabilities of each pin are (5V: the pin tolerates inputs up to 5.3V):

Pin	Features	5V	Pin	Features	5V
1	3.3 Volt Power		2	5.0 Volt Power	
3	I2C SDA / CAN1 Tx	5V	4	5.0 Volt Power	
5	I2C SCK / CAN1 Rx	5V	6	Ground	
7	Analog Input or COUNT 1		8	Analog Input or COM1: TX	
9	Ground		10	Analog Input or COM1: RX	
11	COM2: RX	5V	12	Analog Input or PWM 1A	

13	Analog Input or COUNT 2		14	Ground	
15	Analog Input or COUNT 3		16	Analog Input or COM2: TX	
17	3.3 Volt Power		18	FAST COUNT	5V
19	SPI MOSI / CAN2 Rx	5V	20	Ground	
21	SPI MISO	5V	22	Analog Input or PWM 1B	
23	SPI CLOCK	5V	24	Analog Input or COUNT 4	
25	Ground		26	Analog Input	
27	I2C2 SDA	5V	28	I2C2 SCK	5V
29	Analog Input or PWM 1C		30	Ground	
31	PWM 2B	5V	32	General I/O	5V
33	General I/O	5V	34	Ground	
35	SPI2 MISO	5V	36	PWM 2A	5V
37	Analog Input or COM1 DE		38	SPI2 MOSI	5V
39	Ground		40	SPI2 CLOCK / CAN2 Tx	5V

Except for the power and ground pins all pins can be used for digital I/O using the PIN() function and command and using the pin number as the reference. For example pin 3 can be set to an output using SETPIN 3, DOUT and then the pin set high (ie, to 3.3V) using the command PIN(3) = 1.

All pins marked 5V can tolerate inputs up to 5.3V; all other pins can tolerate up to 3.6V.

Pin 40 can be used to completely reset the Colour Maximite 2 to its "factory default" condition. If that pin is connected to ground (ie, pin 39) on power up all options will be reset to their defaults and any program in program memory erased. Note that external circuitry connected to this pin (eg, a capacitor) must not look like a short circuit at power up as this might trigger a reset.

There are 12 pins marked as supporting analog input and these can be used for measuring voltages. The other special capabilities (eg, COM2: RX, etc) are described in the relevant section of this manual.

Both the data line (SDA) and clock (SCL) for both I2C ports have 10K pullup resistors (to 3.3V) installed on the motherboard so external pullup resistors are not required. Note that these may interfere with the operation of these pins if they are used as general purpose inputs.

Wii Nunchuk and Wii Classic Connector

The Colour Maximite 2 includes connectors on the front panel for either the Wii Nunchuk (illustrated below) or the Wii Classic game controllers and MMBasic includes commands and functions to work with both of them. Many games written for the Colour Maximite 2 will use either controller to control the game play so they are a useful addition if you plan on playing some games.



For more details on the Nunchuk and Classic search the Internet for "Nintendo Nunchuk" or "Nintendo Classic". Clones of both of these can be purchased cheaply for US\$10 to US\$40.

The connector used by the Wii controllers has a large plastic clip on top and this must be uppermost when it is plugged in. Take care as it is possible to insert the connector upside down and that may damage the controller or the Colour Maximite 2.

Temperature Sensor

These solder pads are for an optional Dallas DS18B20 temperature sensor. Support for the DS18B20 is built into MMBasic - see the chapter Special Device Support in this manual for the details. The signal line for this

sensor is pin 42. If this sensor is installed the associated pullup resistor (4.7K Ω) on the motherboard must also be installed.

IR Receiver

These solder pads are for an IR remote control receiver which will allow the Colour Maximite 2 to be controlled via a standard NEC or Sony infra red remote control transmitter. See the chapter Special Device Support for details of the receivers that can be fitted and how to use them in your program. The signal pin for the receiver can also be used as a digital I/O pin as pin 41.

Firmware Upload Select

This is either a switch or a jumper. Normally the switch should be set in the "Flash" position or, if a jumper is used, it should be in the "RUN" position.

When uploading new firmware the switch is set in the "System" position and the jumper should be moved to the "PRG" position. See Appendix K (Loading the Firmware) at the end of this manual for the details.

Backup Battery

The CR1220 coin cell battery on the motherboard keeps the internal ARM Cortex-M7 real time clock running while the power is off and also keeps a bank of 4KB RAM alive at the same time. The real time clock is used to provide the correct time to MMBasic on startup and the battery backed RAM is used to store saved variables (see VAR SAVE). The options are kept in the flash memory and do not need the battery. The life of this battery will be about 3 to 4 years of normal use.

Reset Switch

This is used to reset the Colour Maximite 2 and start the bootup sequence as if the power had been cycled.

Serial Console

The serial console is a method of connecting your desktop or laptop computer to the Colour Maximite 2 and the MMBasic console. With access to the console you can enter and edit programs, run them, transfer files, etc.

Normally the VGA screen and the USB keyboard are used as the console, but the serial console works alongside them and just as well, including when using the File Manager and the Full Screen Editor. The only thing that the serial console does not show is graphics. It is particularly useful for transferring programs to and from your desktop computer (with the XModem and ZModem protocols), and for working on the Colour Maximite 2 from your desktop computer.

USB Serial Port

The serial console uses the USB Type-B connector on the rear of the Colour Maximite 2, which is also its power connector. A USB to serial converter chip on the board (a Microchip MCP2221A on the first generation design, a CH340 on the second) makes it appear to your desktop computer as a normal serial port.

Windows 10 and later include the required USB device driver. For other operating systems (ie, Linux, Mac, Windows 8 and earlier) and the MCP2221A go to the Microchip website <https://www.microchip.com/wwwproducts/en/MCP2221A> for instructions (select the Documents tab). Support for the CH340 is included in most operating systems.

In Windows the Colour Maximite 2 will appear in Device Manager as "USB Serial Port" as illustrated below (the COM number will probably be different).



The MCP2221A converter chip tells the host by default that it only needs 100mA on the 5V pin and in some rare cases this can cause trouble (ie, failure to power up, random restarts, etc). The fix is to go to <https://www.microchip.com/wwwproducts/en/MCP2221A> and download the "MCP2221 Utility" (it is under the Documents tab) and use that to change the MCP2221A to request 500mA by default.

Terminal Emulator

You also need a terminal emulator running on your desktop or laptop computer. This is a program that acts like an old fashioned computer terminal where it will display text received from a remote computer and any key presses will be sent to the remote computer over the serial link. The terminal emulator that you use should support VT100 emulation as this is required by the editor and file manager built into MMBasic.

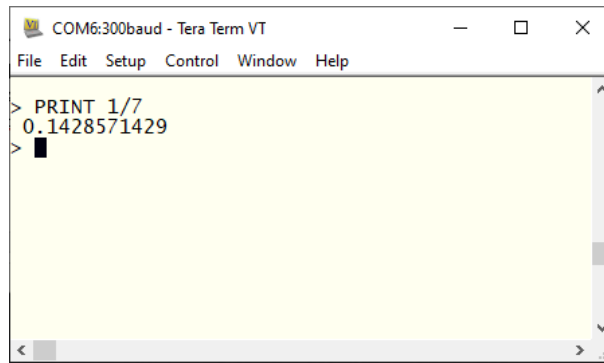
For Windows users it is recommended that you use Tera Term as this has a good VT100 emulator and is known to work with the XModem and ZModem protocols which you can use to transfer programs to and from the Colour Maximite 2. Tera Term can be downloaded from: <http://tera-term.en.lo4d.com>.

The terminal emulator and the serial port that it is using should be set to the Colour Maximite 2 standard of 115200 baud, 8 data bits, no parity and one stop bit. The speed can be changed with the OPTION BAUDRATE command (the terminal emulator must then be set to match). A Generation 2 runs well at 460800 baud, which makes the editor and file transfers much faster; the USB to serial converter of a Generation 1 cannot run at that speed.

If you are using Tera Term do not set a delay between characters and if you are using Putty set the backspace key to generate the backspace character.

The Console

Once you have connected your terminal emulator to the serial port you should be able to press return on your keyboard and see the MMBasic prompt, which is the greater than symbol (eg, ">"). If you reset the Colour Maximite 2 you will also see the MMBasic banner.



This is the console and you use it to issue commands to configure MMBasic, load the BASIC program, edit and run it. MMBasic also uses the console to display any error messages.

The OPTION CONSOLE command sets where the console is: BOTH (the screen and the serial console, the default), SCREEN or SERIAL. See the chapter Options for the details.

Apple Macintosh

The Apple Macintosh (OS X) is somewhat easier as it has the device driver and terminal emulator built in. First start the application 'Terminal' and at the prompt list the connected serial devices by typing in:

```
ls /dev/tty.*
```

The USB to serial converter will be listed as something like /dev/tty.usbmodem12345. While still at the Terminal prompt you can run the terminal emulator at 115200 baud by using the command:

```
screen /dev/tty.usbmodem12345 115200
```

By default the function keys will not be correctly defined for use in the built in program editor so you will have to use the control sequences as defined in the chapter Full Screen Editor of this manual. To avoid this you can configure the terminal emulator to generate these codes when the appropriate function keys are pressed.

Documentation for the screen command is here: <https://www.systutorials.com/docs/linux/man/1-screen/>

Linux

For Linux see these posts:

- <https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12171>
- <https://www.thebackshed.com/forum/ViewTopic.php?TID=14157&PID=175474#175474#175466>

Android

For Android devices see this post:

<https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=17476&LastEntry=Y#230521#230517>

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 the Colour Maximite 2 and is described later in this manual. It includes advanced features such as search and replace, copy, cut and paste to and from a clipboard.

You could also compose the program on your desktop computer using something like Notepad and then transfer it via the XModem or ZModem protocols (see the XMODEM and ZMODEM commands), by streaming it up the console serial link (see the AUTOSAVE command) or by saving it to an SD card or a USB flash drive and transferring that to the Colour Maximite 2.

A fourth and 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://geoffg.net/mmedit.html>

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 left and right arrow keys to move along the line, the Delete key to delete a character, the Backspace key to delete the character before the cursor and the Insert key to switch between insert and overwrite. Home and End move to the beginning and end of the line. At any point the Enter key will send the line to MMBasic which will execute it.

The up and down arrow keys will move through a history of previously entered commands which can be edited and reused.

Console Keyboard/Display

Input can come from either the USB keyboard or from a computer using a terminal emulator via the USB serial interface (see the chapter Serial Console). Both the keyboard and the serial over USB can be used simultaneously and can be detached or attached at any time without affecting a running program.

The text output from MMBasic will be simultaneously sent to the VGA monitor and the serial over USB interface however the graphics commands operate on the video output 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.

Shortcut Keys

The function keys on the keyboard or the serial console can be used at the command prompt to automatically enter common commands. The first four function keys (F1 to F4) will insert the text followed by the Enter key so that the command is immediately executed:

F1	FM (the file manager)
F2	RUN
F3	LIST
F4	EDIT

Function keys F5 to F10 will insert the text then position the cursor between the quote marks at the end so that the file name can be directly entered. Pressing Enter will then execute the command:

F5	AUTOSAVE ""
F6	XMODEM RECEIVE ""
F7	XMODEM SEND ""
F8	EDIT ""
F9	LIST ""
F10	RUN ""

Function keys F11 and F12, and Shift-F3, Shift-F4, Shift-F7 and Shift-F8, can be programmed with custom text: see the commands OPTION F11, OPTION F12, OPTION F15, OPTION F16, OPTION F19 and OPTION F20 (F15 is Shift-F3 and so on). Shift-F5 clears the console and the connected VT100 screen, and Shift-F6 sets the size of the terminal emulator's window to that of the screen.

Line Numbers and Program Structure

A BASIC program starts at the first line and continues until it runs off the end of the program or hits an END command - at which point MMBasic will display the command prompt (>) on the console and wait for something to be entered.

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 A = 24.6 : PRINT A).

Appendix L (Programming in BASIC - A Tutorial) at the rear of this manual provides a comprehensive tutorial on the language which will take you through the fundamentals in an easy to read format with lots of examples.

All Programs Are Run From the SD Card

On the Colour Maximite 2 all programs reside on the SD card (or a USB flash drive) which acts as the "disk drive" for the computer. As a result the SD card must be present for most operations. This is different from the original Maximite where the SD card was not necessarily required.

When you edit a program you are editing the program on the SD card, when you run a program you will run it from the SD card, 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:

Command	Description
RUN "prog"	Run the program called prog located on the SD card.
LIST "prog"	List the program called prog on the console screen. This will pause every screen full and any key press will continue the listing.
EDIT "prog"	Edit the program called prog located on the SD card.

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. (See also the * (asterisk) shortcut which allows *hello to be used without quotes as an alias for RUN "hello".)

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 SD card. (* can also be used to run the current program without supplying the program name.)

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 other commands that operate on program files: AUTOSAVE, LIST ALL, XMODEM and ZMODEM. These are used for sending/receiving programs via the serial console to or from the SD card. AUTOSAVE will also update the current file name so that after a file has been transferred the RUN command without a file name will run that program. Note that the Colour Maximite 2 does not need LOAD or SAVE commands for programs (LOAD and SAVE are used for images and other data).

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 on the SD card called "data.txt". In this case the current program name will not be changed.

Status Line

When at the command prompt MMBasic will display a status line at the bottom of the VGA screen. On the left side this will show the current directory on the SD card while the text in the centre is the current program name (ie, the file that will be used if the commands RUN, EDIT and LIST are used without specifying a file name). On the right side it shows the current time and date.

The status line can be turned off with the OPTION STATUS OFF command.

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 key used to interrupt a program can be changed with the OPTION BREAK command.

The running program is normally held in non volatile flash memory. This means that it will not be lost if the power is removed and, if you have the AUTORUN feature turned on, the program will automatically run when power is restored (use the OPTION AUTORUN command to turn it on). Normally an SD card 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 SD card 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 makes 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.

Setting Options

Many options can be set by using commands that start with the keyword OPTION. They are listed in their own section of this manual. For example, you can set the video mode that the Colour Maximite 2 starts in with the command:

```
OPTION DEFAULT MODE 8
```

The options set at the command prompt are saved in the flash memory and are remembered when the power is removed.

Saved Variables

Data is normally saved by the program to the SD card but sometimes there is a need to save a small amount of data which is independent of the SD card and will survive a power failure, reboot, etc. This data might include menu choices, calibration data and configuration information.

This can be done with the VAR SAVE command which will save the variables listed on its command line in non volatile memory. These variables can be restored with the VAR RESTORE command which will add all the saved variables to the variable table of the running program. Normally VAR RESTORE is placed near the start of a program so that the variables are ready for use by the program.

The space reserved for saved variables is 4KB. This is a RAM memory which is kept alive by the battery on the motherboard. Writing to this memory is near instantaneous and data can be written an unlimited number of times without degradation.

Saved variables are cleared by the NEW and VAR CLEAR commands, by a firmware upgrade and when MMBasic is reset to its original configuration (see below).

Timing

You can get the current date and time using the DATE\$ and TIME\$ functions and you can set them by assigning the new date and time to them. The Colour Maximite 2 has a battery backed clock so it will not lose the time even when powered off. If you find that the time drifts while the power is off you can use the OPTION RTC CALIBRATE command to correct for any inaccuracies.

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 milliseconds. 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, and SETTICK FAST sets up one more, from a hardware timer, at up to tens of thousands a second.

Watchdog Timer

It is possible to use the Colour Maximite 2 without a VGA monitor and also have nothing connected to the serial console. With OPTION AUTORUN ON set the program will run automatically on power up without human intervention.

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).

PIN Security

Sometimes it is important to keep the data and program confidential. In the Colour Maximite 2 this can be done by using the OPTION PIN command. This command will set a pin number (which is stored in non volatile memory) and whenever MMBasic returns to the command prompt (for whatever reason) the user at the console will be prompted to enter the PIN number. Without the correct PIN the user cannot get to the command prompt and their only option is to enter the correct PIN or reboot the computer. When it is rebooted the user will still need the correct PIN to access the command prompt.

Because an intruder cannot reach the command prompt they cannot list or copy the program held in the program memory, they cannot change the program or change any aspect of MMBasic. Once set the PIN can

only be removed by providing the correct PIN as set in the first place. If the number is lost the only method of recovery is to reset MMBasic as described below (which will erase the program).

Note that this is not a complete protection as it is possible to connect a debugger to the ARM Cortex-M7 CPU and access all areas of memory using that facility but it does deter casual access.

The Serial Console

The default settings for the serial over USB console are 115200 baud, 8 bits, no parity and one stop bit. Using the `OPTION BAUDRATE` command the baud rate of the serial console can be changed to any other speed. Changing the console baud rate to a higher speed makes the full screen editor faster in redrawing the screen: a Generation 2 runs well at 460800 baud, but the USB to serial converter of a Generation 1 cannot run at that speed.

Once changed the console baud rate will be permanently remembered unless another `OPTION BAUDRATE` command is used to change it. Using this command it is possible to accidentally set the baud rate to an invalid speed and in that case the only recovery is to connect a VGA monitor and USB keyboard to change the baudrate or reset MMBasic as described below.

If the serial console is not required it can be disabled with the command `OPTION CONSOLE SCREEN`. This is reset when the program ends. If you want to permanently disable the serial console then the above command should be followed with `OPTION CONSOLE SAVE`. Disabling the serial console has two advantages:

- The built in editor will operate faster as it does not need to echo the edited text on the slow serial console.
- The serial port used by the serial console can be opened as COM3:.

Resetting MMBasic

MMBasic can be reset to its original configuration by placing a link between pins 39 and 40 on the rear I/O connector while power is applied. This will result in the program memory and saved variables being completely erased and all options (security PIN, console baud rate, etc) reset to their initial defaults. The `OPTION RESET` command resets the options to their defaults from the command prompt (it does not erase the program or the saved variables).

When using pin 40 for general I/O care must be taken so that it does not look like it is shorted to ground on power up. The symptom of this is unexplained sudden losses of all options that have been changed and the erasure of the program held in flash.

The Library

Using the `LIBRARY` feature it is possible to create BASIC functions, subroutines and embedded fonts and add them to MMBasic to make them permanent and part of the language. For example, you might have written a series of subroutines and functions that perform sophisticated bit manipulation; these could be stored as a library and become part of MMBasic and perform the same as other built-in functions that are already part of the language. An embedded font can also be added the same way and used just like a normal font.

To install components into the library you need to write and test the routines as you would with any normal BASIC routines. When they are working correctly you can use the `LIBRARY SAVE` command. This will transfer the routines (as many as you like) to a non-visible part of flash memory where they will be available to any BASIC program but will not show when the `LIST` command is used and will not be deleted when a new program is loaded or `NEW` is used. However, the saved subroutines and functions can be called from within the main program and can even be run at the command prompt (just like a built-in command or function).

Some points to note:

- Library routines act exactly like normal BASIC code and can consist of any number of subroutines, functions, embedded C routines and fonts. The only difference is that they do not show when a program is listed and are not deleted when a new program is loaded.
- Library routines can create and access global variables and are subject to the same rules as the main program – for example, respecting `OPTION EXPLICIT` if it is set.
- When the routines are transferred to the library MMBasic will compress them by removing comments, extra spaces, blank lines and the hex codes in embedded C routines and fonts. This makes the library space efficient, especially when loading large fonts. Following the save the program area is cleared.
- You can use the `LIBRARY SAVE` command multiple times. With each save the new contents of the program space are appended to the already existing code in the library.

- You can use line numbers in the library but you cannot use a line number on an otherwise empty line as the target for a GOTO, etc. This is because the LIBRARY SAVE command will remove any blank lines.
- You can use READ commands in the library. DATA statements are searched as a single sequence: the main program first and then the library, so when the program's DATA statements are exhausted (or the program has none) reading continues with the DATA statements in the library. To read the library's DATA from the beginning use the RESTORE command before the first READ; used within library code this resets the pointer to the library space.
- Labels belong to the library and the program together: a label must not be defined in both, and the program must not have a subroutine or function with the name of one in the library. Either is reported when the program is run (Duplicate label or Duplicate name).
- The library is saved in the last 128KB of the program flash memory (flash slot 6), which cannot then be used for a program (see OPTION FLASH).
- You can see what is in the library by using the LIBRARY LIST command which will list the contents of the library space.
- The library can be saved to a file using LIBRARY DISK SAVE fname\$ and restored using LIBRARY DISK LOAD fname\$.

To delete the routines in the library space you use the LIBRARY DELETE command. This will clear the space and return the flash used by the library back to being available for normal programs. OPTION RESET does not delete the library.

A program can install its own library with LIBRARY LOAD fname\$, which reads one or more BASIC source files (up to eight) and makes them the library. This is intended for programs that are distributed with a library of their own: the library becomes part of the program rather than something that must be installed first. The command must be the first statement in the program and the program is restarted once the library has been installed, so that the library's initialisation code runs before the program proper. Because the command remembers a hash of the source files it will do nothing at all when the library already matches, so it is safe to leave at the top of a program permanently.

With the RAM option (LIBRARY LOAD fname\$, RAM) the library is kept in RAM instead of the flash memory. A RAM library lasts until the next RUN, NEW or program load, so the program loads it every time it starts, and the flash library is left as it is.

Program Initialisation

The library can also include code that is not contained within a subroutine or function. This code (if it exists) will be run automatically before a program starts running (ie, via the RUN command). This feature can be used to initialise constants or setup MMBasic in some way. For example, if you wanted to set some constants you could include the following lines in the library code:

```
CONST TRUE = 1
CONST FALSE = 0
```

For all intents and purposes, the identifiers TRUE and FALSE have been added to the language and will be available to any program that is run.

MM.STARTUP

There may be a need to execute some code on initial power up, perhaps to initialise some hardware, set some options or print a custom start-up banner. This can be accomplished by creating a subroutine with the name MM.STARTUP in the program. When the Colour Maximite 2 is first powered up or reset it will search the program held in program memory (the program that was last run) for this subroutine and, if found, it will be run once.

For example, the program could contain the following code:

```
SUB MM.STARTUP
  PRINT "Hello from " MM.DEVICE$
END SUB
```

This would print a greeting on every power up or reset.

After the code in MM.STARTUP has been run MMBasic will run the program if OPTION AUTORUN is on, otherwise it will return to the command prompt.

Note that you should not use MM.STARTUP for general setup of MMBasic (like dimensioning arrays, opening communication channels, etc) before running a program. The reason is that when you use the RUN command MMBasic will first clear the interpreter's state ready for a fresh start.

MM.PROMPT

If a subroutine with this name exists it will be automatically executed by MMBasic instead of displaying the command prompt. This can be used to display a custom prompt, set colours, define variables, etc all of which will be active at the command prompt.

Note that MMBasic will clear all variables and I/O pin settings when a program is run so anything set in this subroutine will only be valid for commands typed at the command prompt (i.e. in immediate mode).

As an example the following will display a custom prompt:

```
SUB MM.PROMPT
  PRINT TIMES$ "> ";
END SUB
```

Note that while constants can be defined, they will not be visible because a constant defined inside a subroutine is local to a subroutine. However, DIM will create variables that are global and that should be used instead.

MM.END

If a subroutine named MM.END exists in the program it will be executed whenever the program ends with an END command or by running off its last line (an implied END). It is not executed if the program ends with an error or a Ctrl-C.

The optional parameter 'noend' to the END command can be used to block execution of the MM.END subroutine if needed (see the END command for more information).

Full Screen Editor

An important productivity feature is the built-in full screen editor. When running it looks like this:



```
' Bubbles: coloured circles drawn at random
' Press any key to stop
Const MAXR = 60          ' the largest radius
Dim INTEGER count = 0

Cls
Do
  r = Rnd * 255 : g = Rnd * 255 : b = Rnd * 255
  x = Rnd * MM.HRes : y = Rnd * MM.VRes
  Circle x, y, Rnd * MAXR + 2,,, 0, RGB(r, g, b)
  count = count + 1
  If count Mod 100 = 0 Then Status count
Loop Until Inkey$ <> ""
End

' show how many bubbles have been drawn
Sub Status n As INTEGER
  Text 0, 0, "Bubbles: " + Str$(n), "LT", 1, 1, RGB(white), 0
End Sub

/E/bubbles.bas - 19 lines                               INS.bas | 6/8/19
F1:Save F2:Run F3:Find F4:Mark F5:Paste F7/8:Rpl/r F11:Help F12/^A:Btfg
```

The editor operates with both consoles: the VGA screen with the USB keyboard, and the serial console using a VT100 terminal emulator (such as Tera Term).

The editor is a very productive method of writing a program. With the command EDIT "file" you can enter and edit 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 (EDIT) at the command prompt will load the editor and position the cursor at the line that caused the error. This edit/run/edit cycle is very fast.

The editor works on a file on the SD card (A: or B:) or a USB flash drive (C:). Files of up to about 1MB can be edited.

If you have previously used an editor like Windows Notepad 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 overwrite 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.

The best way to learn how to use the editor is to simply fire it up and experiment.

Edit Functions

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

In more detail these functions are:

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 if you really want to abandon your changes.
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: FIND	This will prompt for the text that you want to search for, offering the text last searched for (Enter accepts it, typing replaces it). When you press Enter the cursor

	will be placed at the start of the first match found, which is brought into view. The search ignores case and wraps round at the end.
F4: MARK	This is described in detail below.
F5: PASTE	This will insert (at the current cursor position) the text that had been previously cut or copied (see below).
F6: FIND AGAIN	Once you have used the search function you can repeat the search by pressing F6 (Shift-F3 is also valid).
F7: REPLACE	If the cursor is positioned after F3 or F6, this will open a dialog allowing you to directly enter a string into the paste buffer and on pressing Enter, replace the search target string.
F8: REPLACE AGAIN	If the cursor is positioned after F3 or F6 the system will replace the search target string with the contents of the paste buffer.
F9: INSERT	You will be prompted for a file name and when Enter is pressed that file will be inserted at the current cursor position.
F10: SAVE CLIPBOARD	You will be prompted for a file name and when Enter is pressed the contents of the clipboard (if any) will be saved to that file. See the mark mode below.
F11: HELP	This will show the help for the keyword at the cursor (or just before it). The help is taken from the file A:/help.txt if the SD card has one, otherwise from the help built into MMBasic. Use the arrow keys and Page Up/Page Down to scroll through it, and Esc or F11 to go back to the text.
F12: BEAUTIFY	This will re-indent the program using a consistent two space indent for the body of multiline structures such as IF...END IF, FOR...NEXT, DO...LOOP, SELECT CASE...END SELECT, SUB...END SUB and FUNCTION...END FUNCTION. Single line IF statements are not changed, comment lines are indented at the current level and in programs that use line numbers the statements are aligned in a column after the longest line number. A single blank line is always placed after each END SUB and END FUNCTION. If the program contains any other blank lines you will be asked "KEEP BLANK LINES? (Y/N)" - answer Y (or press Enter) to keep them, N to remove them or press ESC to cancel. When finished the cursor will be placed at the start of the program. Ctrl-A can also be used to start the beautify.

Additional Keys

These keys are particular to the Colour Maximite 2's editor:

Shift-Tab	Delete up to a tab stop's worth of spaces at the cursor (at the start of a line, the indent goes back by one tab stop).
Shift-Del	At the start of a line, delete its leading spaces; elsewhere the same as Delete.
Shift-arrow	Start mark mode, with the arrow as its first move (Shift and the arrows also work in mark mode).
Page Up/Down	The first press moves the cursor to the top (or bottom) row of the screen, the next moves a page.
Print Screen	Save a picture of the screen to the SD card as A:/SCREENnnn.BMP (the next free number). It takes a few seconds.

The Tab key inserts spaces up to the next tab stop (see OPTION TAB).

Saving

When the editor saves a file it first writes the text to a temporary file (name.\$\$\$), then renames the old file to name.bak (replacing any older .bak) and finally renames the temporary file to name. Whatever goes wrong, the file or its .bak is whole. If the save fails (for example because the SD card has been removed) the editor says "SAVE FAILED: " with the reason and stays open with your text, so that you can correct the problem and save again.

In a program (a .bas file) a line can be at most 255 characters long, and F1 and F2 refuse to save a longer line with a message. Other files (for example .txt files) can have longer lines.

Mark Mode

Pressing the mark key (F4) will put the editor into mark mode which will allow you to cut or copy text to the clipboard or delete or save text.

When in the mark mode the arrow keys are used to mark the text which will be highlighted in reverse video. If you reach the beginning or end of the screen the arrow keys will continue to mark text as the screen scrolls. This also works with Page Up and Page Down.

In the mark mode the status line will change to show the functions available in this mode. These are:

ESC	Exit mark mode without changing anything.
F4: CUT	Copy the marked text to the clipboard and remove it from the program.
F5: COPY	Just copy the marked text to the clipboard.
F10: SAVE	Prompt for a filename and when Enter is pressed, save the marked text to the file.
DELETE	Delete the marked text leaving the clipboard unchanged.

The clipboard can hold as much text as there is memory for, and the status line shows how many characters it holds. A paste inserts the clipboard's text exactly as it is, tabs included.

Alternate Keys

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

Key	Control	Key	Control	Key	Control	Key	Control
LEFT	Ctrl-S	RIGHT	Ctrl-D	UP	Ctrl-E	DOWN	Ctrl-X
HOME	Ctrl-U	END	Ctrl-K	PageUp	Ctrl-P	PageDn	Ctrl-L
DEL	Ctrl-]	INSERT	Ctrl-N	F1	Ctrl-Q	F2	Ctrl-W
F3	Ctrl-R	F4	Ctrl-T	F5	Ctrl-Y or Ctrl-V	F6	Ctrl-G
F7	Ctrl-F	F8	Shift-F5	F9	Ctrl-O	F10	Ctrl-B
F12	Ctrl-A						

Long Lines

Long lines are handled by horizontal scrolling. A line that is longer than the display width will initially show just its first part, however, when the cursor is moved beyond the right hand margin, the line that the cursor is on will scroll horizontally so that any part of it can be seen and edited. The other lines on the screen are not affected and continue to show their first part.

Using a Mouse

The editor can be used with a mouse: a USB mouse (which needs no configuration), or a PS2 mouse or Hobbytronic mouse interface configured with OPTION MOUSE (see Mouse Support in the chapter Hardware Features). The mouse pointer is shown on the screen while the editor is running.

- Clicking the left button moves the edit cursor to the pointer (ie, the same as using the cursor keys). Double clicking at the left or right edge of the screen is the same as Home or End.
- Holding down the left button (for half a second, or while moving the mouse) starts mark mode at the pointer, and the text is marked as the mouse is moved.
- Clicking the right button moves the cursor to the pointer and starts mark mode (the same as F4). In mark mode it cuts the marked text (F4).
- Clicking the scroll wheel (the middle button) pastes the clipboard at the pointer (the same as F5). In mark mode it copies the marked text (F5).
- Turning the scroll wheel moves up and down through the text.

The buttons work only inside the editing area: on the status line they do nothing.

Colour Coded Editor Display

The editor can colour code the edited program with keywords, numbers, strings and comments displayed in different colours. This feature can be turned on or off at the command prompt with the commands:

```
OPTION COLOURCODE ON
```

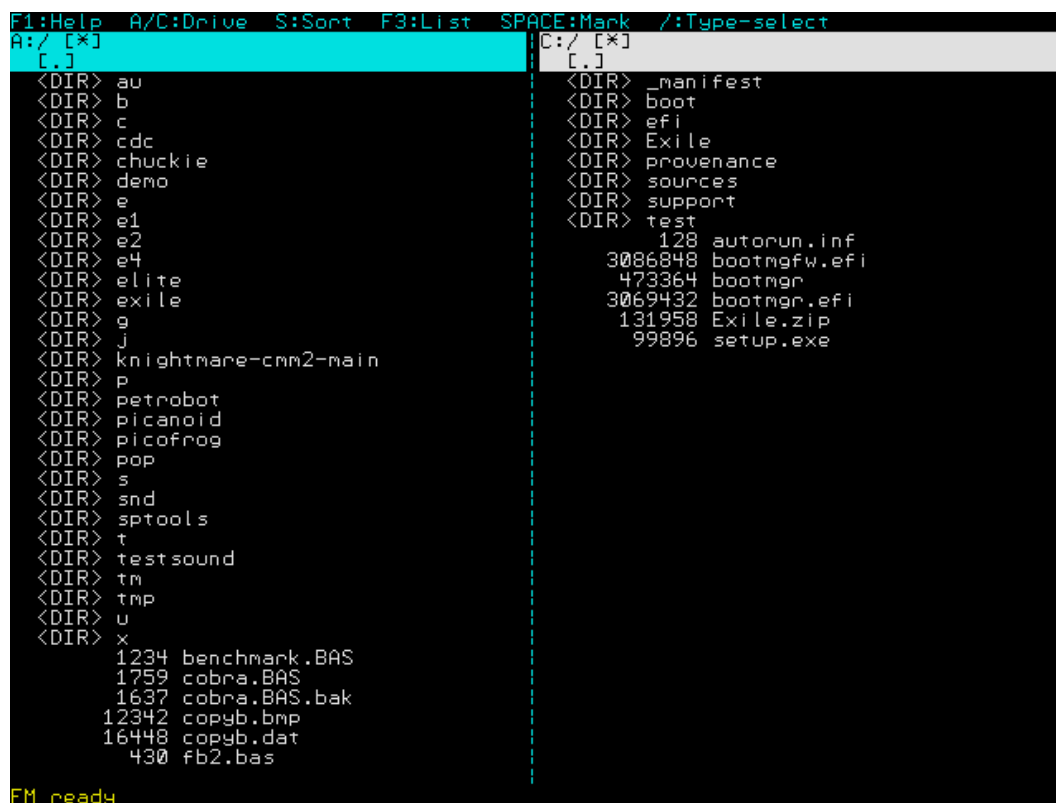
or

OPTION COLOURCODE OFF

This option is saved in non-volatile memory and will be automatically applied on start-up. Only BASIC programs (.bas files) are colour coded.

File Manager

The FM command opens a full screen, two panel file manager in the style of the classic Norton Commander. It can also be started by pressing F1 at the command prompt. It works on the VGA screen and on the serial console using a VT100 terminal emulator. On the VGA screen it sets up the display as the editor does, in the default video mode with the editor's font.



Each panel shows the contents of one directory. Pressing A, B or C switches the active panel between the drives: A: (or B:) is the SD card and C: is a USB flash drive. The panel header shows the drive, the current path and the active filter, and the highlighted panel is the active one - all file operations apply to it. Files can be marked with the space bar (shown with a leading) so that copy and delete operations work on several files at once, and each panel can have a wildcard filter (eg, .BAS) and its own sort order. The status line at the bottom shows the result of the last operation.

Pressing ENTER on a directory opens it; on a .BAS file it runs the program, and the file manager returns with its state when the program ends (if the program stops with an error, the error is shown on the file manager's status line); on an audio file (WAV, FLAC, MP3 or MOD) it plays the file in the background while the file manager remains usable; and on an image file (BMP, JPG or PNG) it displays a full screen preview on a graphical display.

Marked audio files play as a track list. Mark them with the space bar and press ENTER: when every marked item in the panel is an audio file, the file manager plays the marked files in panel order, as the PLAY WAV, FLAC and MP3 commands play a directory, with each track starting when the one before it ends. A list holds up to 20 tracks and may mix WAV, FLAC and MP3 files; MOD files cannot be in a list. The status line shows the track that is playing (for example "Playing 2/5 name") and reports when the list has finished. The . or > key skips to the next track and , or < returns to the previous one, as PLAY NEXT and PLAY PREVIOUS do, and F6 or T stops the list. With any other marks (a directory, a program, an image) ENTER acts on the selected item as usual.

Pressing F4 opens the selected file in the full screen editor and exiting the editor returns to the file manager. If a program run from the file manager stops with an error, pressing F4 (or E) on that file opens the editor with the cursor on the line that caused the error. Pressing F2 within the editor saves and runs the program as normal.

The file manager remembers its state (paths, selection, sort order and filters) between invocations, until the Colour Maximite 2 is restarted. The full set of keys is (press F1 inside the file manager for a quick reminder):

Action	Key(s)
Exit the file manager	ESC

Help screen	F1, H or ?
Switch the active panel	TAB, LEFT or RIGHT (Ctrl-S = left, Ctrl-D = right)
Move the selection up / down	UP, DOWN (Ctrl-E, Ctrl-X)
Page up / page down	PgUp, PgDn (Ctrl-P, Ctrl-L)
Top of the drive / end of the list	HOME, END (Ctrl-U, Ctrl-K)
Go to the parent directory	Backspace
Open / run the selected item	ENTER or F2
Change the drive	A, B or C
Go to a path	G
Cycle the sort mode (name / datetime / type)	S
Type-select by filename prefix	/ then type (ENTER opens, ESC cancels)
Set / clear the file filter	F9 (F) / F10 (W)
Mark or unmark the current item	SPACE
Mark all / clear all marks	* / \
Play the marked audio files as a track list	ENTER or F2 (when every marked item is an audio file)
Copy the selected or marked items to the other panel	F5 or Y
Move the selected item to the other panel	M
Delete the selected or marked items	DEL
Recursive delete (including directories)	X
Rename the selected item	R
Duplicate the selected item	D
New file (create and edit)	N
Make a directory	F11 or K
Edit the selected file	F4 or E
List the selected file (paged)	F3 or L
Stop audio playback	F6 or T
Volume down / volume up	F7 or - / F8 or +
Previous / next track of a track list	, or < / . or >

Notes: a file copy overwrites the destination without asking and a directory copy is recursive and works across drives. DEL only deletes empty directories - use X (which asks for two confirmations) to remove a directory and its contents. M moves within the same drive only; to move between drives use copy (F5) then delete (DEL). CTRL-C is ignored while the file manager is active, except in the LIST viewer where it exits the listing. FM starts by clearing the variables and the other memory left by the last program to run, as EDIT does, so the file manager and its editor always have the memory they need.

The file manager does not yet use the mouse.

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 32 characters long.

A variable name or a label must not be the same as a command, function, one of the nine PIO instructions (IN, OUT, JMP, WAIT, PUSH, PULL, MOV, IRQ, SET) 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.

A label must be unique: the same label defined twice, in the program or in both the program and the library, is reported as Duplicate label when the program is run.

MMBasic supports three types of variables:

- **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).
- **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 (i.e. the part following the decimal point). These are specified by adding the suffix `%` to a variable's name. For example, `i%`, `nbr%`, etc.
- **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.

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.

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 must be surrounded by double quote marks ("). Eg, "Hello World".

OPTION DEFAULT

A variable can be used without a suffix (i.e. `!`, `%` 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)
DIM days(7) AS STRING = ("","Sun","Mon","Tue","Wed","Thu","Fri","Sat")
```

Once created, a complete array can be copied into another array in a single assignment by using the array names with empty brackets. For example:

```
b%() = a%()
```

Both arrays must be the same type (for strings, also the same maximum length) and must hold the same total number of elements. The dimensions may differ so long as the totals match; for example, a 2 x 3 array can be copied into a six element array.

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 (i.e. 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 accidentally 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 = 31
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 = 31, 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 (i.e. !, % or \$) but it must agree with its assigned value.

A constant defined outside of any subroutine or function is global and can be used throughout the program. A constant defined inside a subroutine or function is local to that routine - it is created when the routine is called, disappears when the routine exits and will hide a global constant of the same name.

Special Characters in Strings

Special, non-printable characters can be inserted in string constants using the backslash (ie, \) as an escape symbol. To enable this facility the command `OPTION ESCAPE` must be placed at the start of the program.

These are the valid escape sequences:

	Hex Value	Description
\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 value is given by nnn interpreted as a decimal number
\&hh	any	The byte whose value is given by hh... interpreted as a hexadecimal number

For example, the following will print the words Hello and World on separate lines:

```
OPTION ESCAPE
PRINT "Hello\r\nWorld"
```

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 intention.

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 and any bits introduced are set to zero.
-------------------------------------	---

Logical operators:

NOT INV	invert the logical value on the right (eg, NOT (a=b) is $a \neq b$) or bitwise inversion of the value on the right (eg, $a = \text{INV } b$)
< > <= >= <> =>	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 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 ...

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) in order of ascii value.
=	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.

Structures

Structures (also known as User-Defined Types) allow you to group related variables of different types together under a single name. This is useful for organizing complex data such as coordinates, records, or any collection of related values. This section provides a brief overview but structures are described in full in the file *MMBasic_Structures_Manual.pdf* included in the firmware download zip file.

As an example of a structure, assume that you need to track a number of alarms. Each alarm has a number of properties; the time of the alarm, its level (1, 2, 3, etc) and the action taken. Using structures enables you to treat the data for an alarm as a single "thing".

For example, to create a structure for an alarm you might use:

```
TYPE StructAlarm
  time AS INTEGER
  level AS INTEGER
  action AS STRING
END TYPE
```

The types of the members can be any of the normal data types in MMBasic (FLOAT, INTEGER or STRING). This new type that you have defined can then be used to define a variable. For example you could define a new variable called alarm using:

```
DIM alarm AS StructAlarm
```

You can also define a structure as LOCAL or STATIC within a subroutine or function.

You can access members of the structure using a period (.) between the variable's name and the member's name. For example: `alarm.time`.

You can assign values to members of the structure using normal assignment. ie:

```
alarm.time = EPOCH(NOW)
alarm.level = 3
```

And you can access members of the structure using the same notation:

```
IF alarm.level > 4 THEN ...
```

You can assign one structure to another of the same type:

```
DIM alarm AS StructAlarm
DIM a2 AS StructAlarm
...
alarm = a2
```

You can call a subroutine or function using a structure as one or more of its parameters:

```
MySub alarm, a2
```

And a function can return a structure:

```
FUNCTION MyFun() AS StructAlarm
    MyFun.time = EPOCH(NOW)
    MyFun.level = 1
END FUNCTION
...
alarm = MyFun()
```

You can define an array of a structures as you would any normal array:

```
DIM alarm(nbr) AS StructAlarm
```

And you can easily access the members of each element in the array:

```
IF alarm(i).time < x AND alarm(i).level < y THEN CallSomeSub
```

There are many commands within MMBasic for special handling of structures including saving and loading them from a file, sorting a structure array, resetting the members of a structure and so on. The full details are covered the file *MMBasic_Structures_Manual.pdf* included in the firmware download zip file.

64-bit Unsigned Integers

MMBasic 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.

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% = &H800FFFFFFFFFFFFFFF
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>
    <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
```

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.

Functions

Functions are similar to subroutines with the main difference being that a 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 (brackets 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
```

A function can also return a complete array by declaring the array dimensions with the type at the end of the definition (`INTEGER` and `FLOAT` arrays only). The result is set by assigning to elements of the function's name. For example:

```
FUNCTION Multiples(n%) AS INTEGER(5)
    LOCAL i%
    FOR i% = 0 TO 5 : Multiples(i%) = n% * i% : NEXT i%
END FUNCTION
```

Such a function can only be used as the source of a whole-array assignment; the destination array must have the same type and total number of elements. For example:

```
DIM m%(5)
m%() = Multiples(3)
```

It cannot be used anywhere else in an expression. Note that reading an element of the function's name inside the function (eg, `x = Multiples(i%)`) will call the function recursively; if intermediate values must be read back, build the result in a LOCAL array and copy it to the function's name with a whole-array assignment (eg, `Multiples() = tmp%()`).

The dimensions can be expressions and are evaluated each time the function is called, after its arguments have been set up, so they can depend on the arguments. The result array can also be filled by passing it to any command that takes an array (eg, the MATH commands) rather than by assigning to its elements. For example, this returns the inverse of whatever size matrix it is given:

```
FUNCTION Invr(N()) AS FLOAT(BOUND(N(),1), BOUND(N(),2))
  MATH M_INVERSE N(), Invr()
END FUNCTION
DIM a(3,3), b(3,3) : b() = Invr(a())
```

Passing Arguments by Reference

If you use an ordinary variable (i.e., 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).

If you want to use an argument as a general purpose variable inside a subroutine or function (ie, change its value) you should prefix its definition with the keyword `BYVAL`. This instructs MMBasic to always use the value of the argument, even if it is a variable, and to never point back to the variable used in the call. 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 if you did not use `BYVAL`.

Note that `BYVAL` only works with simple variables (not arrays or structures).

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 (i.e., 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 or it could find these via the `BOUND()` function. 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

```

A subroutine or function can also return a complete array to its caller. This is described under Functions earlier in this chapter.

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 (i.e., 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 microcontrollers and the BASIC language):

- There is a fixed limit to the depth of recursion: 100 levels of subroutine and function calls together. A recursive function is also limited by the processor's stack, as each level is in the middle of evaluating an expression: a simple recursive function such as $f = f(n - 1) + 1$ reaches about 40 levels (about 37 with OPTION COMPILE ON) before the error "Stack overflow, at depth n, stack ..., heap ...". A recursive subroutine reaches the 100 levels. When a program has come within 1K of the stack's limit without passing it, the command prompt warns of it ("Warning: the stack came within n bytes of its limit, at depth n"). MM.INFO(STACK) is where the stack is now and MM.INFO(HEAPEND) its limit.
- 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 100 level limit.
- FOR...NEXT loops work correctly when the subroutine or function is called recursively from within them, but any DO...LOOPs will be corrupted.

Examples

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"

Video Modes and Pages

The video output of the Colour Maximite 2 is set by the MODE command:

```
MODE r [, bits [, bg [, int]]]
```

'r' is the screen resolution, a number from 1 to 18; 'bits' is the colour depth (see below); 'bg' is the background colour in the 12-bit mode; and 'int' is a subroutine that will be called at the start of the frame blanking (useful for updating the screen without tearing). The default at power up is MODE 1 with an 8-bit colour depth, or the mode set with OPTION DEFAULT MODE.

Resolutions

Mode	Resolution	Mode	Resolution
1	800 x 600 pixels	10	848 x 480 pixels (widescreen)
2	640 x 400 pixels	11	1280 x 720 pixels (widescreen)
3	320 x 200 pixels	12	960 x 540 pixels (widescreen, lines duplicated)
4	480 x 432 pixels	13	400 x 300 pixels
5	240 x 216 pixels	14	960 x 540 pixels (widescreen, lines not duplicated - not supported by all monitors)
6	256 x 240 pixels	15	1280 x 1024 pixels
7	320 x 240 pixels	16	1920 x 1080 pixels (widescreen, Generation 2 only)
8	640 x 480 pixels	17	384 x 240 pixels
9	1024 x 768 pixels (no 12-bit mode)	18	1024 x 600 pixels

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).

The monitor will see these signals: modes 1 and 13, 800 x 600 at 60Hz; modes 2, 3, 4, 5, 6, 7, 8 and 17, 640 x 480 at 75Hz; mode 9, 1024 x 768 at 60Hz; mode 10, 848 x 480 at 60Hz; mode 11, 1280 x 720 at 60Hz; mode 14, 1920 x 540 at 60Hz; modes 12 and 16, 1920 x 1080 at 60Hz; mode 15, 1280 x 1024 at 60Hz.

The GETSCANLINE function returns the line of the monitor that is being drawn. Its maximum value in each mode is: mode 1, 627; modes 2, 4 and 8, 499; modes 3, 5, 6, 7 and 17, 249; mode 9, 805; mode 10, 516; mode 11, 749; mode 12, 562; mode 13, 313; mode 14, 568; mode 15, 1065; mode 16, 1124; mode 18, 637.

Colour Depths

	32-bit (G2)	16-bit	12-bit	8-bit	4-bit
Pixel format	ARGB8888	RGB565	ARGB4444	RGB332	RGB121
Bits per pixel	32 (4 bytes)	16 (2 bytes)	16 (2 bytes)	8 (1 byte)	4 (two pixels a byte)
Colours	16,777,216	65,536	4,096	256	16
Transparency	None	None	16 levels	None	In the layers
Pages displayed	1	1	2 + background	1	1 to 3 (the layers)

The 32-bit depth is available on the Generation 2 only.

The 12-bit depth is particularly useful for games: it supports 4096 colours with an additional 4-bit alpha channel allowing 16 levels of transparency. It has two image layers (pages 0 and 1) plus a background colour ('bg' in the MODE command). Page 0 is the lower layer and page 1 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.

In the 8-bit depth each of the 256 colours can be changed with the MAP command.

The 4-bit depth is the PicoMite's 16 colour display: see The PicoMite's 16 Colour Display below.

Pages

In addition to the page that is displayed there are many extra video pages that are available to the programmer for building images. The display always shows the contents of page 0 (and pages 0 and 1 in the 12-bit depth) unless overridden with the PAGE DISPLAY command. PAGE WRITE sets the page that the graphics commands draw on, and PAGE COPY copies one page to another, which can be done during the video blanking period to update the display without flashing and tearing. PAGE SCROLL scrolls a page, PAGE STITCH and PAGE MERGE combine pages, and PAGE RESIZE changes the size of a page.

The pages available in each mode and colour depth are (Generation 1 / Generation 2):

Mode	32-bit (G2)	16-bit	12-bit	8-bit	4-bit (G2)
1	0 to 2	0-2 / 0-6	0-2 / 0-6	0-6 / 0-15	0 to 28
2	0 to 6	0-6 / 0-14	0-6 / 0-14	0-13 / 0-29	0 to 57
3	0 to 27	0-25 / 0-57	0-24 / 0-56	0-53 / 0-60	0 to 60
4	0 to 7	0-7 / 0-17	0-7 / 0-17	0-15 / 0-35	0 to 60
5	0 to 33	0-30 / 0-60	0-29 / 0-60	0-60 / 0-60	0 to 60
6	0 to 28	0-26 / 0-60	0-25 / 0-59	0-53 / 0-60	0 to 60
7	0 to 20	0-19 / 0-46	0-18 / 0-45	0-41 / 0-60	0 to 60
8	0 to 4	0-4 / 0-10	0-4 / 0-10	0-10 / 0-23	0 to 47
9	-	0-1 / 0-3	-	0-3 / 0-8	-
10	0 to 3	0-2 / 0-7	0-2 / 0-7	0-7 / 0-17	0 to 34
11	-	0 / 0-2	-	0-2 / 0-6	-
12	-	0 / 0-4	-	0-3 / 0-11	-
13	0 to 12	0-11 / 0-28	0-10 / 0-27	0-26 / 0-60	0 to 60
14	0 to 2	0-2 / 0-6	0-2 / 0-6	0-6 / 0-14	0 to 27
15	-	-	-	0-1 / 0-4	-
16	-	-	-	- / 0-2	-
17	0 to 16	0-15 / 0-38	0-14 / 0-37	0-34 / 0-60	0 to 60
18	0 to 1	0-1 / 0-4	0-1 / 0-4	0-6 / 0-10	-

A "-" means that the combination is not available. The 4-bit figures are for the Generation 2 (the Generation 1, with less memory, has fewer). MM.INFO(MAX PAGES) returns the highest page number in the current mode.

Video Memory

MM.INFO(PAGE ADDRESS n) and MM.INFO(MAX PAGES) 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:

- With an 8-bit colour depth you would use: $\text{add\%} = \text{MM.INFO(PAGE ADDRESS n)} + y * \text{MM.HRES} + x$
- For the 12 or 16-bit colour depth you would use: $\text{add\%} = \text{MM.INFO(PAGE ADDRESS n)} + (y * \text{MM.HRES} + x) * 2$
- With the 4-bit colour depth two pixels share a byte, the left pixel in the low four bits: $\text{add\%} = \text{MM.INFO(PAGE ADDRESS n)} + y * ((\text{MM.HRES} + 1) \setminus 2) + x \setminus 2$

For modes 3, 5, 6, 7, 12, 13 and 17 in the 8, 12, 16 and 32-bit depths 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 (4-bit pages hold each line once). For example, 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
```

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.

The PicoMite's 16 Colour Display (MODE n,4)

A colour depth of 4 bits gives the display that the PicoMite's VGA and HDMI versions have in their 16 colour modes, so that programs written for the PicoMite (and their PEEKs and POKEs of the screen) work as they do there. MODE 7,4 (320 x 240) is the PicoMite's MODE 2 and MODE 8,4 (640 x 480) is its MODE 3. The 4-bit depth is available in the modes that the table above shows.

- The 16 colours are the PicoMite's RGB121 colours. The MAP command changes them (MAP(n) returns the PicoMite's value for colour n), and MAP RESET restores them.
- The pixels are held two to a byte, the left one in the low four bits, at MM.INFO(WRITEBUFF) as on the PicoMite.
- The FRAMEBUFFER command works as it does on the PicoMite: FRAMEBUFFER CREATE makes the framebuffer F, LAYER the layer L (shown over the display, with a transparent colour), LAYER TOP the top layer T, and CREATE 2 a second framebuffer; WRITE, COPY, MERGE, WAIT and CLOSE act on them. The framebuffers are pages 0 to 4 (N, L, F, 2 and T) and the pages after them are more 4-bit pages, for PAGE WRITE, PAGE COPY, BLIT and the like.
- BLIT, SPRITE and TILEMAP work on 4-bit pages, with colour 0 (black) as the transparent colour of a sprite.

In the other colour depths FRAMEBUFFER is the Colour Maximize 2's own: a working area larger than the screen, with FRAMEBUFFER CREATE width, height, WRITE, BACKUP, RESTORE, WINDOW and CLOSE (see the FRAMEBUFFER command). While a framebuffer exists, a MODE command that changes the colour depth is refused with the error "Colour mismatch with framebuffer": use FRAMEBUFFER CLOSE first.

Image Slots

FLASH LOAD IMAGE n, file\$ loads an image into image slot n (1 to 8), as the PicoMite's games do. Slots 4 to 8, the PicoMite's RAM slots, are held in RAM until RAM ERASE, an overwrite or a restart, as the PicoMite's are, so a program that loads its images every time it starts uses FLASH LOAD IMAGE's O (OVERWRITE). Slots 1 to 3, the PicoMite's flash slots, are kept on the SD card and read into RAM when they are first used (see FLASH SAVE). MM.INFO(FLASH ADDRESS n) returns a slot's address and BLIT FLASH draws from it. The image is held in the PicoMite's format (an 8 byte width and height, then two pixels a byte), so programs that PEEK it work as they do on the PicoMite.

Running PicoMite Programs

A program written for the PicoMite selects its display with the PicoMite's MODE numbers, which mean different resolutions on the Colour Maximize 2 (MODE 1 here is 800 x 600). Change its MODE statements as follows so that it has the display it expects:

PicoMite	Colour Maximize 2	Display
MODE 1	MODE 8,8	640 x 480, 2 colours on the PicoMite (8-bit here)
MODE 2	MODE 7,4	320 x 240, 16 colours
MODE 3	MODE 8,4	640 x 480, 16 colours
MODE 4	MODE 7,16	320 x 240, RGB555 on the PicoMite (16-bit here)
MODE 5	MODE 7,8	320 x 240, 256 colours

See also the chapter Porting Programs.

Keyboard, Mouse and Gamepad

The USB connector on the front panel of the Colour Maximite 2 is a USB host. It takes a keyboard and, through a USB hub, a mouse, up to two gamepads, up to four USB serial devices and a USB flash drive, all at the same time. A PS2 mouse can also be connected (see Mouse Support in the chapter Hardware Features), and the Wii Nunchuk and Wii Classic controllers plug into the connectors on the front panel.

The USB support is the PicoMite's, so programs written for the PicoMite read these devices the same way here.

USB Hub

A USB hub can be plugged into the USB connector so that several devices can be used at once. The keyboard, the mouse and the gamepads are given a channel number: by default a keyboard is channel 1, a mouse channel 2 and the first and second gamepads channels 3 and 4. USB serial devices are numbered separately, as the serial ports COM4 to COM7.

The processor's USB controller cannot work with a low-speed USB device through a hub. Many inexpensive keyboards, mice and gamepads are low-speed devices. Such a device works when it is plugged straight into the USB connector; through a hub MMBasic refuses it with the message "USB low-speed device behind a hub is not supported: plug it in directly". Full-speed devices work through a hub.

When a device is connected or removed a message is shown on the console, for example "USB Keyboard Connected on channel 1" or "USB Mouse Disconnected". A device that MMBasic does not support is reported with its USB identity: "USB device (vid &H..., pid &H...) connected: not supported".

USB Keyboard

When a USB keyboard is connected it will be immediately recognised (no configuration required). Keyboards with a wireless dongle work, and so do keyboards with a built-in hub or a built-in mouse.

The keyboard's layout is set with the command:

```
OPTION USBKEYBOARD layout
```

Where 'layout' is US, UK, DE (German), FR (French), ES (Spanish) or BE (Belgian). OPTION KEYBOARD REPEAT start, rate sets the time before a held key starts to repeat and the time between its repeats (in milliseconds).

Note that the non US layouts map some of the special keys present on these keyboards but the corresponding special character may not display if it is not in the font used. Instead a standard ASCII character will be used.

The codes that the special keys generate (the function keys, the arrows and so on) are listed in Appendix I.

KEYDOWN() returns the keys that are being held down, which is useful in games.

USB Mouse

When a USB mouse is connected it will be immediately recognised (no configuration required) and MMBasic will allocate it to channel 2. It can be used in the editor (see the chapter Full Screen Editor) and read by a program.

Within a program the mouse is read with the DEVICE() function, as on the PicoMite:

```
' print any movement of the mouse on the console
Do
  mx = DEVICE(MOUSE 2, X)
  my = DEVICE(MOUSE 2, Y)
  If mx <> tx Or my <> ty Then Print mx, my
  tx = mx : ty = my
Loop
```

DEVICE(MOUSE 2, L), R and M return the state of the left, right and middle buttons, W the scroll wheel, and D a double click. The MOUSE INTERRUPT ENABLE command calls a subroutine when a button is clicked, and MOUSE SET sets the position of the mouse. The MOUSE() function (the Colour Maximite 2's) also reads it, as channel 2. OPTION MOUSE SENSITIVITY n divides the mouse's movement by n.

A program can also display a cursor on the screen and link it to the mouse using the GUI CURSOR command, and mouse clicks can be read with the CLICK() function - see the command and function listings for the details.

PS2 and I2C Mice

A PS2 mouse (connected to the rear I/O connector on the Generation 1, or to the mouse socket on the Generation 2) is mouse channel 0, and a Hobbytronic USB mouse interface is opened on an I2C channel. Both are opened in a program with `CONTROLLER MOUSE OPEN` and read with the `MOUSE()` function; for the editor they are set up with `OPTION MOUSE`. See Mouse Support in the chapter Hardware Features.

USB Gamepads

Up to two USB gamepads can be connected through a hub (or one straight into the USB connector). The first is allocated to channel 3 and the second to channel 4. Within a program the gamepad is read using the `DEVICE(GAMEPAD n, ...)` function: LX, LY, RX and RY return the sticks (0 to 255), L and R the analog triggers, B the buttons as a bitmap, H the direction of the d-pad, T the type of gamepad, and on a PlayStation 4 controller GX, GY and GZ the gyroscope and AX, AY and AZ the accelerometer.

These gamepads are recognised:

- the PlayStation 4 DualShock 4 and its clones, and the PlayStation 3 DualShock 3 (which sends only its buttons until its PS button is pressed);
- the Nintendo Switch Pro Controller, and the gamepads that present themselves as one in their Switch mode;
- the Xbox 360, Xbox One and Xbox Series controllers;
- a number of generic gamepads, such as the USB versions of the Super Nintendo SNES controller.

The buttons of every gamepad are mapped onto `DEVICE(GAMEPAD n, B)` by their position, so a program works with any of them. On a Switch Pro Controller X, the top face button, is bit 10, A (right) bit 11, Y (left) bit 12 and B (bottom) bit 13; + is START, - SELECT and Capture the touchpad bit; ZL and ZR are L2 and R2 and read 255 or 0 as the analog L and R.

The `GAMEPAD` command sets up a gamepad: `GAMEPAD INTERRUPT ENABLE` calls a subroutine when a button is pressed, `GAMEPAD HAPTIC` drives a controller's vibration motors and `GAMEPAD COLOUR` sets the colour of a PlayStation 4 controller's light bar. `GAMEPAD MONITOR` shows what a gamepad sends and `GAMEPAD CONFIGURE` describes a generic gamepad that MMBasic does not recognise.

USB Serial Devices (COM4 to COM7)

MMBasic can also communicate with USB serial devices. This includes USB to serial adapters based on the CH340, CP210x, FTDI and PL2303 chips as well as devices that present a USB serial interface of their own such as an Arduino, a Raspberry Pi Pico or a GPS receiver.

Up to four USB serial devices can be connected (via a hub) and they are allocated to the serial ports COM4, COM5, COM6 and COM7 in the order that they are plugged in. These are in addition to the three hardware serial ports COM1, COM2 and COM3. When a device is connected a message is shown on the console, eg, "USB serial device (CH34x) Connected as COM4".

The port is opened and used exactly like the hardware serial ports, eg:

```
OPEN "COM4:115200" AS #1
```

The baud rate and the 7BIT, EVEN, ODD and S2 options are passed to the USB device (where the device supports them); the DE, INV and OC options do not apply. A receive buffer size, an interrupt subroutine and an interrupt trigger level can be specified exactly as for COM1 to COM3 - see Appendix A (Serial Communications). The DTR and RTS signals are asserted while the port is open. A USB GPS receiver can be opened AS GPS like a GPS receiver on a hardware port.

If the USB device is unplugged while the port is open the port will remain open and communications will resume when the device is plugged back in. The `CLOSE` command closes the port and frees its receive buffer.

USB Flash Drive

A USB flash drive (memory stick) can be connected, either directly or via a hub. The drive will be automatically recognised and the message "USB Flash Drive Connected" will be shown on the console.

The drive is accessed as drive C: and works alongside the SD card (A:, also called B:). All of the normal file commands can be used with it (FILES, RUN, EDIT, OPEN, COPY, etc) and it can be made the default drive with the command `DRIVE "C:"` or by simply typing C: at the command prompt. The file manager (FM) can copy files and directories between the SD card and the drive. The drive can be formatted as FAT16, FAT32 or exFAT.

Wii Nunchuk and Classic Controllers

The Wii Nunchuk and the Wii Classic controllers plug into the connectors on the front panel, and up to three can be used (on the I2C channels 1 to 3). The Nunchuk controller is opened with the `CONTROLLER NUNCHUK OPEN` command and its status is queried using the `NUNCHUK()` function. The Classic controller is opened with the `CONTROLLER CLASSIC OPEN` command and its status is queried using the `CLASSIC()` function. See the chapter Game Playing Features.

Program and Data Storage

The Colour Maximite 2 has full support for programs, files and directories on the SD card and on a USB flash drive. This includes opening files for reading, writing or random access and editing and running programs.

Drives

- **A:** is the SD card in the front panel connector. It can also be called **B:**, as the PicoMite calls its SD card, so that PicoMite programs that use B: work unchanged.
- **C:** is a USB flash drive, plugged into the USB connector (directly or through a hub - see the chapter Keyboard, Mouse and Gamepad).

The active drive (the default) is A: on startup. It can be changed with the DRIVE command (eg, DRIVE "C:") or by typing A:, B: or C: at the command prompt, and then the drive prefix is not needed.

MMBasic will work with SD cards up to 128GB in capacity. Cards larger than 32GB should be formatted as exFAT and cards 32GB or less formatted as FAT32. The recommended size is 8GB formatted as FAT32. A USB flash drive can be formatted as FAT16, FAT32 or exFAT.

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 ...). A path that includes the drive letter must be a full path from the root (eg, "C:/mypath/myfile.txt").
- 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, on any of the drives.
- Input and output commands and functions can also use file #0 which refers to the console.
- Except for INPUT, LINE INPUT and PRINT the # in #fnbr is optional and may be omitted.

Program Management Commands

All programs reside on the SD card (or a USB flash drive) and so it must be present when running, editing and listing programs (see All Programs Are Run From the SD Card in the chapter Using MMBasic).

- 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.
- ZMODEM RECEIVE [fname\$] - Receive one or more files using the ZModem protocol, which is faster and more reliable than XModem.
- ZMODEM SEND [fname\$] - Send a file using the ZModem protocol.

File Access Within a Program

- OPEN fname\$ FOR mode AS #fnbr - Opens a file for reading or writing. 'fname\$' is the file name, 'mode' can be INPUT, OUTPUT, APPEND or RANDOM, 'fnbr' is the file number (1 to 10).
- PRINT #fnbr, expression [[,;]expression] ... etc - Outputs text to the file opened as #fnbr.

- INPUT #fnbr, list of variables - Read a list of comma separated data into the variables specified from the file previously opened as #fnbr.
- LINE INPUT #fnbr, variable\$ - Read a complete line into the string variable specified from the file previously opened as #fnbr.
- SEEK #fnbr, pos - Will position the read/write pointer in a file that has been opened for RANDOM access to the 'pos' byte.
- FLUSH #fnbr - Forces any buffered writes to be written to the drive. It is recommended that this command be used regularly where data loss could occur in the event of power loss.
- 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 'nbr' characters read from a file previously opened for INPUT or RANDOM. If less than 'nbr' characters are available the function will return with what it has.
- 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 an open file 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

- FILES [wildcard] [,sortorder] (or LIST FILES) - Search the current directory and list the files/directories found. These can only be used at the command prompt, not within a program. The file manager (FM) is a full screen alternative - see the chapter File Manager.
- KILL fname\$ - Delete a file. A wildcard (eg, KILL "*.BAK") deletes all the matching files - see the KILL command for the details.
- COPY fromname\$ TO toname\$ - Copy a file, on the same drive or between drives. At the command prompt a wildcard copies all the matching files to a directory.
- RENAME fromname\$ AS toname\$ - Rename a file.
- MKDIR dname\$ - Make a sub directory.
- CHDIR dname\$ - Change into the directory dname\$. It can also be ".." (dot dot) for up one directory or "/" for the root directory.
- RMDIR dir\$ - Remove, or delete, the directory dir\$.
- DRIVE drive\$ - Sets the active drive to drive\$ ("A:", "B:" or "C:").

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

- DIR\$(fspec, type) - Will search for files and return the names of the entries found.
- CWD\$ - Will return the current working directory.
- MM.INFO(DRIVE) - The active drive.
- MM.INFO(FREE SPACE) and MM.INFO(DISK SIZE) - The free space on and the size of the active drive.
- MM.INFO(EXISTS FILE fname\$) and MM.INFO(EXISTS DIR dirname\$) - True if the file or directory exists.
- MM.INFO(FILESIZE fname\$) and MM.INFO(MODIFIED fname\$) - The size of a file and the date and time it was last modified.
- MM.INFO(PATH) - The path of the program being run. This allows a program to find the resource files it needs relative to itself.

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.

See the chapter Sound Output for the details.

Load and Save Images

An image can be loaded from a drive and displayed on the VGA monitor. This can be used to draw a logo or add a background on the display.

```
LOAD BMP filename$ [, StartX, StartY]
LOAD GIF filename$ [, StartX, StartY]
LOAD JPG filename$ [, StartX] [, StartY] [, dither] [, xOffset] [,
yOffset] [, scale]
LOAD PNG filename$ [, StartX] [, StartY] [, transparent] [, alphacut]
[, xOffset] [, yOffset]
```

Where 'filename\$' is the image to load and 'StartX'/'StartY' are the coordinates of the top left corner of the image (these are optional and will default to the top left corner of the display if not specified). `LOAD IMAGE` is another name for `LOAD BMP` (the PicoMite's name for it).

`LOAD JPG` accepts four additional optional parameters. 'dither' selects a dithering mode (0 to 7) which can improve the appearance of the image in modes with a limited colour depth. 'xOffset' and 'yOffset' specify a pixel offset into the source image so that only the lower right portion of a large image is displayed. 'scale' reduces the displayed size of the image by pixel averaging: use 2 to display at 50% size, 4 at 25% and 8 at 12.5%. The default is 1 which displays the image at full size.

`LOAD PNG` accepts 'xOffset' and 'yOffset' in the same way, after its 'transparent' and 'alphacut' parameters (see `LOAD PNG` in the commands listing). With these formats an image larger than the display is cropped to fit.

The image must be in the appropriate format and MMBasic will add the extension to the file name if it is not specified.

The current image on the screen can be saved to a file using the following command:

```
SAVE IMAGE filename$ [, StartX, StartY, width, height]
```

This will save the image, or part of the image, as a 24-bit true colour BMP file (the extension .BMP will be added if an extension is not supplied). The Print Screen key does this at the command prompt, in the editor and in the file manager, saving the screen as A:/SCREENnnnn.BMP.

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
        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

```

Sound Output

The Colour Maximite 2 can play WAV, FLAC and MP3 files, generate synthesised music in the MOD format and the BBC Micro's sound, create robot speech and sound effects as well as generate precise sine wave tones. All these are output on the audio socket (see Audio Connector in the chapter Hardware Features). The ARM Cortex-M7 chip includes its own DAC (digital to analog converter) so an output filter network is not needed.

Playing WAV, FLAC, MP3 and MOD Files

The PLAY command will play an audio file residing on the SD card or a USB flash drive 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 on the format of the file:

```
PLAY WAV file$ [, interrupt]
PLAY FLAC file$ [, interrupt]
PLAY MP3 file$ [, interrupt]
PLAY MODFILE file$ [, interrupt]
```

'file\$' is the name of the audio file to play 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, mono and stereo (see the PLAY command in the command listing for the details).

A MOD file plays in a loop until it is stopped, unless an interrupt is given: then it plays once and the interrupt is called at its end.

Background Music

If file\$ in the PLAY WAV, FLAC or MP3 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. PLAY NEXT and PLAY PREVIOUS skip to the next or previous file, and MM.INFO(TRACK) returns the name of the file that is playing.

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 the Colour Maximite 2 to be used as a music player while programming or doing other tasks. The file manager (FM) can also play a list of marked audio files (see the chapter File Manager).

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, right [, duration] [, interrupt]
```

'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 'duration' 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 (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command. Note that the sine wave is generated by stepping through a lookup table so to reduce the distortion the audio output should be passed through a low pass filter.

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 EFFECT** which will play a WAV file at the same time as a MOD file is playing, with its own interrupt.
- **PLAY TTS** which will generate robotic speech.
- **PLAY SOUND** which will generate an output based on a mixture of sine, square, triangular, sawtooth and noise waveforms, on up to four sound channels. **PLAY LOAD SOUND** loads a wave of your own for it.
- **PLAY ARRAY** which plays 16-bit samples held in integer arrays.
- **PLAY SAMPLE** which plays one cycle of a wave held in an array, looped under ADSR envelopes, and **PLAY RELEASE** which starts the release of its envelopes.
- **PLAY BBC SOUND** and **PLAY BBC ENVELOPE**, described below.

See the details in the command listing.

BBC Micro Sound

The **PLAY BBC SOUND** and **PLAY BBC ENVELOPE** commands are the BBC Micro's **SOUND** and **ENVELOPE** statements, with the same parameters, units and behaviour, so the tunes and sound effects of a BBC Micro program carry across with only the **PLAY BBC** prefix added. They give MMBasic a four channel sequencer: notes are queued on each channel with a duration and play in order without the program having to time them, chords can be started together, and an envelope can shape both the volume and the pitch of a note while it plays. Channels 1 to 3 are square wave tones and channel 0 is noise, as on the BBC.

Pitch is in quarter semitones with 89 being A4 (440Hz) and 53 middle C, so each octave is 48 and the notes of the C major scale from middle C are 53, 61, 69, 73, 81, 89, 97 and 101. Durations are in twentieths of a second. For example, a scale, a chord and a note shaped by an envelope:

```
FOR p = 53 TO 101 STEP 4 : PLAY BBC SOUND 1, -15, p, 5 : NEXT p
PLAY BBC SOUND &H201, -10, 53, 20 : PLAY BBC SOUND &H202, -10, 69, 20
: PLAY BBC SOUND &H203, -10, 81, 20
PLAY BBC ENVELOPE 1, 1, 4, -4, 4, 10, 10, 10, 126, -2, 0, -10, 120,
100
PLAY BBC SOUND 1, 1, 100, 30
```

Each channel queues up to 16 notes and, when a channel's queue is full, **PLAY BBC SOUND** waits for a slot, so a loop of **PLAY BBC SOUND** commands paces itself to the music. **MM.INFO(BBC BUFFER)** returns how many slots are free, so a program that must not be held up can check before it queues. Adding **&H10** to the channel number flushes the queue and stops the current note so that a sound effect is heard at once, which is how a game fires a laser over whatever is playing:

```
PLAY BBC ENVELOPE 2, 1, -8, 0, 0, 20, 0, 0, 126, -8, 0, -20, 126, 100
PLAY BBC SOUND &H11, 2, 220, 4
```

The audio output is released automatically when every channel is silent and every queue is empty, so a tune can be followed by **PLAY WAV** or **PLAY MP3** without a **PLAY STOP** in between, and **MM.INFO(SOUND)** returns BBC while it is playing. Envelope definitions survive **PLAY STOP** and are cleared with the program's variables by **RUN**.

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 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
PLAY STOP
```

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.
PLAY NEXT	Skip to the next file when playing a directory.
PLAY PREVIOUS	Skip to the previous file when playing a directory.

Changes in volume, stopping and pausing are smoothed over a few milliseconds so that they do not click.

MM.INFO(SOUND) returns what the audio output is doing: OFF, WAV, FLAC, MP3, MOD, TONE, SOUND, TTS, DAC (the DAC command), ARRAY, SAMPLE or BBC, and PAUSED followed by what was playing (eg, PAUSED WAV) while it is paused.

Using the I/O Pins

The 40-pin ribbon connector on the rear panel provides 28 input/output pins which can be controlled from within the BASIC program with 12 of these supporting the measurement of voltage. An I/O pin is referred to by its pin number and these, and their capabilities, are listed in the section External I/O Connector of the chapter Hardware Features.

Digital Inputs

A digital input is the simplest type of input configuration. If the input voltage is higher than 2.5V the logic level will be true (numeric value of 1) and anything below 0.65V will be false (numeric value of 0). The inputs use a Schmitt trigger input so anything in between these levels will retain the previous logic level. Pins marked as 5V are 5V tolerant and can be directly connected to a circuit that generates up to 5.5V without the need for voltage dropping resistors.

In your BASIC program you would set the input as a digital input and use the PIN() function to get its level. For example:

```
SETPIN 4, DIN
IF PIN(4) = 1 THEN PRINT "High"
```

The SETPIN command configures pin 4 as a digital input and the PIN() function will return the value of that pin (the number 1 if the pin is high). The IF command will then execute the command after the THEN statement if the input was high. If the input pin was low the program would just continue with the next line in the program.

The SETPIN command also recognises a couple of options that will connect an internal resistor from the input to either the supply or ground. This is called a "pullup" or "pulldown" resistor and is handy when connecting to a switch as it saves having to install an external resistor to place a voltage across the contacts.

Analog Inputs

Pins marked as Analog Input can be configured to measure the voltage on the pin. The input range is from zero to 3.3V and the PIN() function will return the voltage. For example:

```
> SETPIN 7, AIN
> PRINT PIN(7)
2.345
>
```

You will need a voltage divider if you want to measure voltages greater than 3.3V. For small voltages you may need an amplifier to bring the input voltage into a reasonable range for measurement.

The measurement uses the 3.3V power supply to the CPU as its reference and it is assumed that this is exactly 3.3V. This value can be changed with the OPTION VCC command.

The ADC command can capture up to three channels of analog data in the background at up to 500KHz per channel, into arrays (see the ADC command).

Counting Inputs

The pins marked as COUNT can be configured as counting inputs to measure frequency, period or just count pulses on the input. For example, the following will print the frequency of the signal on pin 7:

```
> SETPIN 7, FIN
> PRINT PIN(7)
110374
>
```

In this case the frequency is 110.374 kHz.

By default the gate time is one second which is the length of time that MMBasic will use to count the number of cycles on the input and this means that the reading is updated once a second with a resolution of 1 Hz. By specifying a third argument to the SETPIN command it is possible to specify an alternative gate time between 10 ms and 100000 ms. Shorter times will result in the readings being updated more frequently but the value returned will have a lower resolution. The PIN() function will always return the frequency in Hz regardless of the gate time used. For example, the following will set the gate time to 10ms with a corresponding loss of resolution:

```
> SETPIN 7, FIN, 10
```

```
> PRINT PIN(7)
110300
>
```

For accurate measurement of signals less than 10 Hz it is generally better to measure the period of the signal. When set to this mode the Colour Maximite 2 will measure the number of milliseconds between sequential rising edges of the input signal. The value is updated on the low to high transition so if your signal has a period of (say) 100 seconds you should be prepared to wait that amount of time before the PIN() function will return an updated value.

The COUNT pins can also count the number of pulses on their input. When a pin is configured as a counter (for example, SETPIN 7,CIN) the counter will be reset to zero and the Colour Maximite 2 will then count every transition from a low to high voltage. The counter can be reset to zero again by executing the SETPIN command a second time (even though the input was already configured as a counter).

There is also a fast counting input (pin 18) which has been tested up to 40MHz for frequency measurement and counting. Note that period measurement is not available on this pin.

Digital Outputs

All I/O pins can be configured as a standard digital output. This means that when an output pin is set to logic low it will pull its output to zero and when set high it will pull its output to 3.3V. In MMBasic this is done with the PIN command. For example PIN(15) = 0 will set pin 15 to low while PIN(15) = 1 will set it high.

The "OC" option on the SETPIN command makes the output pin open collector. This means that the output driver will pull the output low (to zero volts) when the output is set to a logic low but will go to a high impedance state when set to logic high. If you then connect a pull-up resistor to 5V (on pins that are 5V tolerant) the logic high level will be 5V (instead of 3.3V using the standard output mode). The maximum pull-up voltage in this mode is 5.5V.

Pulse Width Modulation

The PWM (Pulse Width Modulation) command allows the Colour Maximite 2 to generate square waves with a program controlled duty cycle. By varying the duty cycle you can generate a program controlled voltage output for use in controlling external devices that require an analog input (power supplies, motor controllers, etc). The PWM outputs are also useful for driving servos and for generating a sound output via a small transducer.

There are two PWM controllers; the first has three outputs and the second two to give a total of five PWM outputs.

When the Colour Maximite 2 is powered up or the PWM OFF command is used the PWM outputs will be set to high impedance (they are neither off nor on). So, if you want the PWM output to be low by default (zero power in most applications) you should use a resistor to pull the output to ground when it is set to high impedance. Similarly, if you want the default to be high (full power) you should connect the resistor to 3.3V.

Communications Interfaces (Serial, SPI, I2C and CAN)

These are described in the appendices at the rear of this manual: serial in Appendix A, I2C in Appendix B, 1-Wire in Appendix C, SPI in Appendix D and CAN in Appendix J. The OPEN, I2C OPEN, SPI OPEN and CAN OPEN commands configure the pins that they use.

Interrupts

Interrupts are a handy way of dealing with an event that can occur at an unpredictable time. An example is when the user presses a button. In your program you could insert code after each statement to check to see if the button has been pressed 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.

Any I/O pin that can be used as a digital input can be configured to generate an interrupt using the SETPIN command with up to ten interrupts active at any one time. Interrupts can be set up to occur on a rising or falling digital input signal (or both) and will cause an immediate branch to the specified user defined subroutine. The target can be the same or different for each interrupt. 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 the order given below. During the processing of an interrupt all other interrupts are disabled until the interrupt subroutine returns. During an interrupt (and at all times) the value of the interrupt pin can be accessed using the PIN() function.

Interrupts can occur at any time but they are disabled during INPUT statements. Also interrupts are not recognised during some long hardware related operations (eg, the TEMPR() function) although they will be recognised if they are still present when the operation has finished. 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. Most commands will execute in under 15µs however some commands such as the TEMPR() function can take up to 200ms so it is possible for an interrupt to occur and vanish within this window and thus not be recognised.
- 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.

In addition to interrupts generated by the change in state of an I/O pin, an interrupt can also be generated by other sections of MMBasic including timers and communications ports and the above notes also apply to them. The list of all these interrupts (in high to low priority ranking) is:

- PID control loops
- SETTICK FAST
- ON KEY for any key
- COM1:, COM2: and COM3: serial ports (receive, then transmit)
- COM4: to COM7: USB serial ports
- USB gamepads (GAMEPAD INTERRUPT)
- GUI touch down and up
- ON KEY for an individual key
- PLAY finished, then PLAY EFFECT finished
- Sprite collision
- ADC completion
- DAC completion
- IR receive
- The MODE frame interrupt
- The fast counting input (pin 18)
- Wii Nunchuk and Classic controllers (channels 1 to 3)
- Mice (channels 0 to 3, and MOUSE INTERRUPT)
- INTERRUPT command and CSub interrupts
- I/O pin interrupts in order of definition
- Tick interrupts (SETTICK 1 to 4 in that order)

As an example: if an ON KEY interrupt (for any key) occurred at the same time as a COM1: interrupt, the ON KEY interrupt subroutine would be executed first and then, when the interrupt subroutine finished, the COM1: interrupt subroutine would then be executed.

Special Device Support

To make it easier for a program to interact with the external world the Colour Maximite 2 includes drivers for a number of common peripheral devices.

Infrared Remote Control Decoder

You can easily add a remote control to the Colour Maximite 2. The solder pads for the IR receiver are on the motherboard (near the Wii connector) and it is only a matter of soldering in a suitable receiver. With this done you can use the IR command to handle the device and key codes within your program.

It will work with any NEC or Sony compatible remote controls including ones that generate extended messages. Most cheap programmable remote controls will generate either protocol and using one of these you can add a sophisticated flair to your programs. The NEC protocol is also used by many other manufacturers including Apple, Pioneer, Sanyo, Akai and Toshiba so their branded remotes can be used.

NEC remotes use a 38kHz modulation of the IR signal and suitable receivers tuned to this frequency include the Vishay TSOP4838, Jaycar ZD1952 and Altronics Z1611A.

Sony remotes use a 40 kHz modulation and receivers for this frequency can be hard to find. Generally 38 kHz receivers will work but maximum sensitivity will be achieved with a 40 kHz receiver.

To setup the decoder you use the command:

```
IR dev, key, interrupt
```

Where dev is a variable that will be updated with the device code and key is the variable to be updated with the key code. Interrupt is the interrupt subroutine to call when a new key press has been detected. The IR decoding is done in the background and the program will continue after this command without interruption. This is an example of using the IR decoder:

```
IR DevCode, KeyCode, IR_Int      ' start the IR decoder
DO
  ' < body of the program >
LOOP

SUB IR_Int                        ' a key press has been detected
  PRINT "Received device = " DevCode " key = " KeyCode
END SUB
```

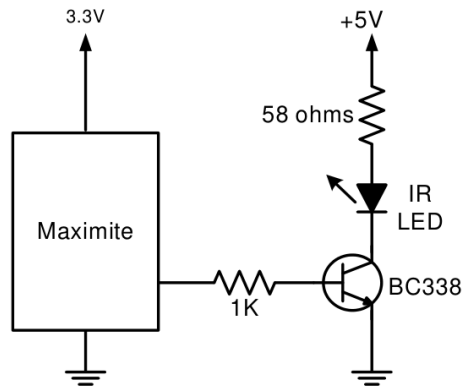
IR remote controls can address many different devices (VCR, TV, etc) so the program would normally examine the device code first to determine if the signal was intended for the program and, if it was, then take action based on the key pressed. There are many different devices and key codes so the best method of determining what codes your remote generates is to use the above program to discover the codes.

The decoder is stopped with IR CLOSE.

Infrared Remote Control Transmitter

Using the IR SEND command you can transmit a 12 bit Sony infrared remote control signal. This is intended for communications with another Colour Maximite 2 or a Micromite but it will also work with Sony equipment that uses 12 bit codes. Note that all Sony products require that the message be sent three times with a 26mS delay between each message.

The circuit below illustrates what is required. The transistor is used to drive the infrared LED because the output of the I/O pin is limited to about 10mA. This circuit provides about 50mA to the LED.



To send a signal you use the command:

```
IR SEND pin, dev, key
```

Where pin is the I/O pin used, dev is the device code to send and key is the key code. Any I/O pin can be used and you do not have to set it up beforehand (the IR SEND command will automatically do that).

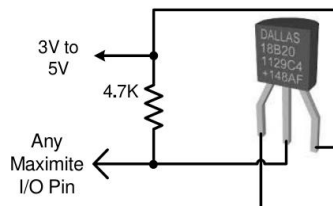
Note that the modulation frequency used is 38KHz and this matches the common IR receivers (described above) for maximum sensitivity when communicating to another Maximite or a Micromite.

Measuring Temperature

The TEMPR() function will get the temperature from a DS18B20 temperature sensor. This device can be purchased on eBay for about \$5 in a variety of packages including a waterproof probe version.

There is a position on the Colour Maximite 2's mother board for a TO-92 version of this sensor and this can be mounted so that the body of the sensor pokes out of a hole in the back panel to sense the ambient temperature. If this is installed the associated pullup resistor (4.7K Ω) on the motherboard must also be installed and pin 42 used in the function and command described below.

Sensors can also be connected to any of the pins on the external I/O connector. The DS18B20 should be powered separately by a 3V to 5V supply as shown below. Multiple sensors can be used but a separate I/O pin and a 4.7K pullup resistor is required for each one.



To get the current temperature you just use the TEMPR() function in an expression. For example:

```
PRINT "Temperature: " TEMPR(pin)
```

Where 'pin' is the I/O pin to which the sensor is connected. You do not have to configure the I/O pin, that is handled by MMBasic.

The returned value is in degrees C with a resolution of 0.25 $^{\circ}\text{C}$ and is accurate to ± 0.5 $^{\circ}\text{C}$. If there is an error during the measurement the returned value will be 1000.

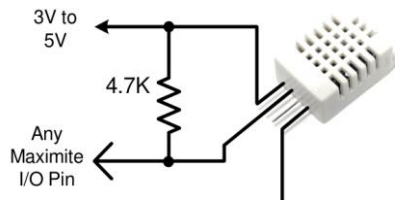
The time required for the overall measurement is 200ms and the running program will halt for this period while the measurement is being made. This also means that interrupts will be disabled for this period. If you do not want this you can separately trigger the conversion using the TEMPR START command then later use the TEMPR() function to retrieve the temperature reading. The TEMPR() function will always wait if the sensor is still making the measurement. For example:

```
TEMPR START 15
< do other tasks >
PRINT "Temperature: " TEMPR(15)
```

Measuring Humidity and Temperature

The HUMID command will read the humidity and temperature from a DHT22 humidity/temperature sensor. This device is also sold as the RHT03 or AM2302 but all are compatible and can be purchased on eBay for under \$5. The DHT11 sensor is also supported.

The DHT22 can be powered from 3.3V or 5V (5V is recommended) and it should have a pullup resistor on the data line as shown. This is suitable for long cable runs (up to 20 meters).



To get the temperature or humidity you use the HUMID command with three or four arguments as follows:

```
HUMID pin, tVar, hVar [,DHT11]
```

Where 'pin' is the I/O pin to which the sensor is connected. You can use any I/O pin but if the DHT22 is powered from 5V it must be 5V capable (ie, NOT a pin that supports analog input). The I/O pin will be automatically configured by MMBasic.

'tVar' is a floating point variable in which the temperature is returned and 'hVar' is a second variable for the humidity. Both of these variables must be declared first as floats (using DIM). The temperature is returned as degrees C with a resolution of one decimal place (eg, 23.4) and the humidity is returned as a percentage relative humidity (eg, 54.3).

If the optional DHT11 parameter is set to 1 then the command will use device timings suitable for that device. In this case the results will be returned with a resolution of 1 degree and 1% humidity. For example:

```
DIM FLOAT temp, humidity
HUMID pin, temp, humidity
PRINT "The temperature is" temp " and the humidity is" humidity
```

HUMID was called BITBANG DHT22 (or DHT22) in earlier versions of MMBasic for the Colour Maximite 2, and both are still accepted.

Measuring Distance

Using a HC-SR04 ultrasonic sensor (illustrated below) and the built in DISTANCE() function you can measure the distance to a target.



This device can be found on eBay for about \$4 and it will measure the distance to a target from 3cm to 3m. It works by sending an ultrasonic sound pulse and measuring the time it takes for the echo to be returned.

Compatible sensors are the SRF05, SRF06, Parallax PING and the DYP-ME007 (which is waterproof and therefore good for monitoring the level of a water tank).

You use the DISTANCE function as follows:

```
d = DISTANCE(trig, echo)
```

Where trig is the I/O pin connected to the "trig" input of the sensor and echo is the pin connected the "echo" output of the sensor. You can also use 3-pin devices and in that case only one pin number is specified.

The value returned is the distance in centimetres to the target. The I/O pins are automatically configured by this function but note that they should be 5V capable as the HC-SR04 is a 5V device.

WS2812 Support

The Colour Maximite 2 has built in support for the WS2812 multicolour LED chip. This chip needs a very specific timing to work properly and with the WS2812 command it is easy to control these devices with minimal effort.

This command will output the required signals needed to drive a chain of WS2812 LED chips connected to the pin specified and set the colours of each LED in the chain. The syntax of the command is:

```
WS2812 type, pin, nbr, colours%()
```

'type' is a single character specifying the type of chip being driven as follows:

- O = original WS2812
- B = WS2812B
- S = SK6812
- W = SK6812W (RGBW)

'nbr' is the number of LEDs in the chain (1 to 512). The colours%() array should be an integer array with at least as many elements as there are LEDs to be driven. For the first three types each element in the array should contain the colour in the normal RGB888 format (0 to &HFFFFFF); for type W it should contain an RGBW value (0 to &HFFFFFFF). If only one LED is connected then a single integer can be used instead of the array, with the same range.

The pin is made a digital output if it is not one already. For example:

```
DIM b%(4)=(RGB(red), RGB(green), RGB(blue), RGB(yellow), RGB(cyan))
SETPIN 5, DOUT
WS2812 O, 5, 5, b%()
```

will output the specified colours to an array of five WS2812 LEDs daisy chained off pin 5.

This command was called BITBANG WS2812 in earlier versions of MMBasic for the Colour Maximite 2, and that spelling is still accepted. Note that the number of LEDs ('nbr') is now required.

Hobbytronic Mouse Support

The Colour Maximite 2 has built in support for the Hobbytronic USB mouse interface (<https://www.hobbytronics.co.uk/usb-host-mouse>). This is enabled using the CONTROLLER MOUSE OPEN command and then the mouse position and buttons can be interrogated using the MOUSE function.

CAN Support

MMBasic on the Colour Maximite 2 exposes the native support for CAN provided by the STM32H743 chip as the MMBasic CAN command. A CAN transceiver such as the SN65HVD230 CAN Transceiver is required to interface to the actual physical CAN bus. It connects to 3.3V and GND plus the nominated CAN Tx and CAN Rx pins on the Colour Maximite 2. The CANL and CANH connectors go to the physical CAN bus. You can exercise the CAN command in loopback mode without having the transceiver attached.



See Appendix J (CAN Support) for the details.

Graphics Functions

These are the commands and functions that you can use within MMBasic to draw images on the VGA monitor (none of these apply to the serial console). There are also a number of advanced commands designed for programmers writing games (such as MODE, PAGE, BLIT, SPRITE and TILEMAP: see the chapters Video Modes and Pages and Game Playing Features) however this section will focus on the standard commands used by most programmers.

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.

By default on startup the VGA output will be set to 800x600 pixels with each pixel supporting 256 different colours. At this resolution the bottom right pixel will be at X = 799 and Y = 599.

Read Only Variables

There are six read only variables which provide useful information about the VGA video output:

- 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 VGA monitor).
- 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 VGA monitor).

Colours

Colour is specified as a true colour 24 bit number where the top eight bits represent the intensity of the red colour, the middle eight bits the green intensity and the bottom eight bits the blue. For example the colour red is &HFF0000 and yellow is &HFFFF00.

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, grey (or gray), lightgrey (or lightgray), orange, pink, gold, salmon, beige, and the PicoMite's lilac, fuchsia, rust, cerulean, midgreen, cobalt and myrtle.

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 the Colour Maximite 2 uses double precision floating point it can store the 24 bit number representing colour (i.e. returned by the RGB() function) in either a floating point variable or an integer variable.

MMBasic will automatically convert colours to the format required for the current colour depth set by the MODE command. So, for example, at startup the Colour Maximite 2's VGA output defaults to 8-bit colour and in this case the 16777215 colours that can be represented by a 24-bit colour specification will be translated as best as possible to the 256 colours supported by the 8-bit colour mode. The 32-bit colour depth (on the Generation 2) displays the 24-bit colours without loss of information.

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	10 x 16	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 (°).

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 below.

```

! " # $ % & ' ( ) * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [ \ ] ^ _
` a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~ ¢
□ ▣ ▤ ▥ ▦ ▧ ▨ ▩ ♠ ♡ ♢ ♣ ♤ ♥ ♦ ♧
⬆ ⬇ ⬈ ⬉ ⬊ ⬋ ⬌ ⬍ ⬎ ⬏ ⬐ ⬑ ⬒ ⬓ ⬔ ⬕
⊖ ⊗ ⊘ ⊙ ⊚ ⊛ ⊜ ⊝ ⊞ ⊟ ⊠ ⊡ ⊢ ⊣ ⊤ ⊥
⊦ ⊧ ⊨ ⊩ ⊪ ⊫ ⊬ ⊭ ⊮ ⊯ ⊰ ⊱ ⊲ ⊳ ⊴
⊵ ⊶ ⊷ ⊸ ⊹ ⊺ ⊻ ⊼ ⊽ ⊾ ⊿ ⊿ ⊿ ⊿
α β γ δ ε ζ η θ ι κ λ μ ν ξ ο π ρ σ τ
υ φ χ ψ ω Ω Ω Ω Ω Ω Ω Ω Ω Ω Ω Ω Ω
≡ ± ≥ ≤ ½ ¼ ÷ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞ ∞

```

Embedded Fonts

If required, additional fonts can be embedded in a BASIC program. These fonts work exactly same as the built in fonts (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 and the PicoMite.

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 the SD card 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, which is included in the Colour Maximite 2 download zip.

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:

- 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 draws an anti-aliased line, and LINE GRAPH and LINE PLOT draw a series of lines from arrays (see the LINE command).
- BOX X, Y, W, H, *LW*, *C*, *FILL* - Draws a box starting at X and Y 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, Y, W, H, *R*, *C*, *FILL* - Draws a box with rounded corners 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.

- **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 be 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 R2 is omitted 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 an 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.
- **BEZIER** XARRAY%(), YARRAY%(), N, *C* - Draws a Bezier curve through the control points held in the arrays (the first N elements, or all of them).
- **FILL** X, Y, *C*, *BORDER* - Flood fills the area around X, Y with the colour *C*, up to the colour *BORDER* or, without it, over the area of the colour at X, Y.
- **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 VGA monitor.

```
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 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 below 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 90°
D	the text will be rotated clockwise by 90°

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 SD card (or a USB flash drive) and display it on the VGA monitor. 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 (see also Load and Save Images in the chapter Program and Data Storage). 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 screen and be of any size (pixels that end up being positioned off the screen will be ignored).

Advanced Graphics Tutorial

The graphics subsystem in the Colour Maximize 2 has many advanced features such as multiple video pages, layered images, sprites and sophisticated methods of manipulating images. Peter Mather, who developed the graphics subsystem, has written a tutorial for games developers and advanced users. This is presented as a series of posts on the Back Shed Forum and is recommended reading for users who need to get into the details: <https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12125>.

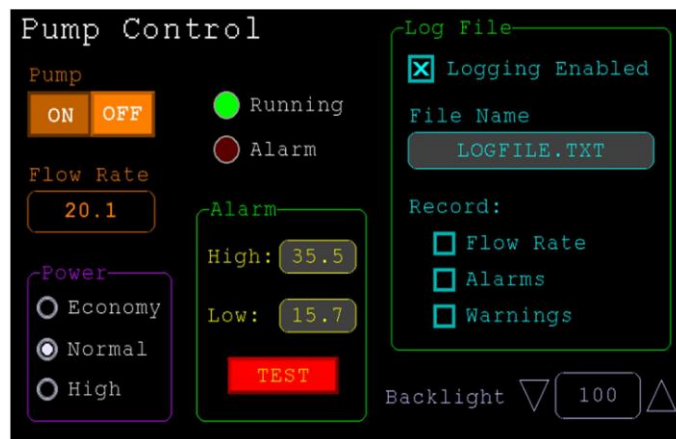
3D Engine

The 3D Engine includes commands for manipulating 3D images including setting the camera, creating, hiding, rotating, etc (see the DRAW3D command). The document The 3D engine.pdf in the Colour Maximize 2

firmware download has a full description of these commands and how to use them. This thread on the Back Shed Forum gave the initial explanation of the 3D Engine and may have later updates or examples:
<https://www.thebackshed.com/forum/ViewTopic.php?TID=13139&P=1>

GUI Controls

The Colour Maximite 2 incorporates a suite of advanced graphic controls that respond to input from a mouse. These include on screen switches, buttons, indicator lights, number and text boxes, gauges, etc. MMBasic will draw the control and animate it (i.e. a switch will appear to depress when activated). All that the BASIC program needs to do is invoke a single line command to specify the basic details of the control. These functions make it easy to create a control panel to manage any control functions like a lathe, motor controller, heating system, small industrial process and so on.



The controls are created with the GUI command: GUI BUTTON, SWITCH, CHECKBOX, RADIO, LED, FRAME, NUMBERBOX, TEXTBOX, FORMATBOX, SPINBOX, DISPLAYBOX, CAPTION, GAUGE, BARGAUGE, AREA, LISTBOX and SLIDER, and managed with GUI PAGE, SETUP, DELETE, DISABLE, ENABLE, HIDE, SHOW, RESTORE, REDRAW, FCOLOUR and BCOLOUR. The value of a control is read or set with the CTRLVAL() function and command, and MSGBOX() displays a message box and returns the button pressed. The GUI controls are described in detail in the document GUI Controls and Programming.pdf which is included in the firmware download file; GUI LISTBOX (a list to choose from), GUI SLIDER (a value set with a thumb on a track) and GUI CLICK, which came later from the PicoMite, are described with the GUI command.

Before the controls can be used, memory must be reserved for them with the command OPTION MAXCTRLS n, where 'n' is the number of controls that the program will use (up to 2000). Without it a program that creates a control stops with the error "No memory allocated for GUI controls".

Using a Mouse

The controls respond to a click of the left mouse button. A USB mouse is used as soon as it is plugged in, as the editor uses it, with no option to set; a PS2 or I²C mouse is set with OPTION MOUSE. The first GUI command a program runs opens the mouse and shows the cursor, which then tracks the mouse, and the left mouse button acts on the control under the cursor. A program can create its controls before a mouse is there: a USB mouse plugged in later is used at once. GUI CURSOR COLOUR and GUI CURSOR LOAD set the cursor's colour and its shape. In the 4-bit colour depth there is no cursor, and the controls are used without one. GUI CLICK clicks the controls without a mouse: from the program (at a position, or held down and released), or with a button on a pin.

The current state of the mouse button can be read at any time with the CLICK() function (or TOUCH(), its PicoMite name): CLICK(X) and CLICK(Y) return where it was clicked, CLICK(DOWN) and CLICK(UP) whether the button has been pressed or released, CLICK(REF) the control under the cursor, and CLICK(LASTX), CLICK(LASTY) and CLICK(LASTREF) where and on what the last click was.

The GUI INTERRUPT command arms interrupt routines that are called when the button is pressed or released:

```
GUI INTERRUPT Down [, Up]
```

'Down' is the subroutine called when the button is pressed and the optional 'Up' subroutine is called on release. GUI INTERRUPT 0 disables the interrupts.

Game Playing Features

The Colour Maximite 2 has many features designed to help programmers in writing computer games. One of the most important features of this computer is its speed (approx 10 times faster than the original 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 moveable images, a selection of text drawing methods and fonts, displaying images, playing audio and getting input from a keyboard or games controller.

VGA Resolution, Colour Depth and Pages

The video output to the VGA monitor is controlled by the MODE command. With this you can select various resolutions from 1920x1080 (on the Generation 2) and 800x600 pixels (the normal default on startup) down to 240x216, and colour depths of 8, 12, 16 and (on the Generation 2) 32 bits. For the 8-bit colour mode each individual colour can be specified from a 16-bit palette using the MAP command. A colour depth of 4 bits gives the PicoMite's 16 colour display, so that games written for the PicoMite can draw as they do there.

Particularly useful is the 12-bit colour mode which supports 4096 colours with an additional 4-bit alpha channel allowing 16 levels of transparency. This mode has provision for two image layers plus a background colour. Images on the top layer will cover the lower layer except where the alpha channel is set to allow transparency thereby allowing various degrees of the lower layer and/or background to show through.

In addition there are many extra video pages that are available to the programmer for building images (see the chapter Video Modes and Pages for how many there are in each mode). 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 the programmer can use the GETSCANLINE function to report on the line that is currently being drawn on the VGA monitor. Using this to time updates to the screen can avoid screen glitches caused by updates while the screen is being updated. A similar outcome can be achieved using an optional feature of the MODE command which will call a user defined subroutine at the start of the VGA frame blanking.

This and other features of the graphics subsystem are explained in detail in a tutorial presented on the Back Shed forum: <https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12125>.

Scrolling, Sprites and Tile Maps

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.

The Colour Maximite 2 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.

The TILEMAP command draws a background made of tiles (small images from one tile sheet) laid out as a map, and scrolls a view of it. It has sprites of its own, which move over the map and can test which tiles they touch. TILEMAP is the PicoMite's, so games written with it there run here too.

Displaying Images

The Colour Maximite 2 can load and display images stored on the SD card in a variety of formats including BMP, GIF, PNG and JPG. These are read 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.

Images can also be loaded into memory with FLASH LOAD IMAGE, into the image slots 1 to 8 that the PicoMite's games use, and drawn from there with BLIT FLASH.

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 SD card. All fonts can be displayed in various colours, scaled and rotated.

Playing Audio

The Colour Maximite 2 can play audio on the stereo audio output in a variety of formats using the PLAY command. These include sound effects or music encoded as WAV, FLAC or MP3 files. Also supported is the playing of MOD files which is a standard for computer generated music. The PLAY EFFECT command will allow the simultaneous playing of a WAV file over a MOD file so that sound effects can be superimposed on the computer generated music playing in the background.

Other audio formats available to the programmer include pure sine wave tones or sound effects consisting of a mixture of sine wave, square wave, triangle wave, sawtooth wave or random noise (the PLAY TONE and PLAY SOUND commands), the BBC Micro's sound and envelopes (PLAY BBC SOUND and PLAY BBC ENVELOPE), and samples played from memory (PLAY SAMPLE and PLAY ARRAY). The programmer can also change the volume, pause any audio output, resume the playing or step forward or back to the next audio file.

To round off the list of audio features the PLAY TTS command will generate a robotic speech output based on a string which defines the words and sounds to be generated.

Keyboard Keys

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.

Wii Nunchuk and Classic Controllers

The Wii Nunchuk (also spelt Nunchuck) and the Wii Classic are inexpensive games controllers which include a joystick and various buttons. MMBasic on the Colour Maximite 2 provides full support for up to three of these controllers including querying the state of all user inputs and their calibration constants.

The Nunchuk controller is opened with the CONTROLLER NUNCHUK OPEN command and its status is queried using the NUNCHUK() function. The Classic controller is opened with the CONTROLLER CLASSIC OPEN command and its status is queried using the CLASSIC() function.

USB Gamepads

Up to two USB gamepads can be plugged into the USB socket, through a hub (see the chapter Keyboard, Mouse and Gamepad). They include the PlayStation 3 and 4 controllers, Xbox 360, Xbox One and Xbox Series controllers, the Nintendo Switch Pro controller and many generic gamepads. A gamepad is read with the DEVICE(GAMEPAD n, ...) function and set up with the GAMEPAD command, as on the PicoMite.

Long Strings

Long Strings are a set of commands and functions that allow MMBasic to manipulate strings of unlimited length and are particularly useful when dealing with data sent via WiFi and the Internet. Standard strings in MMBasic are limited to a maximum length of 255 characters. Long strings duplicate these functions but will work with strings of any length limited only by the amount of available RAM.

Long String Variables

Variables for holding long strings must be defined as integer arrays. The long string routines do not keep numbers in these arrays but just use them as blocks of memory for holding long strings.

When creating these arrays they should 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.

The following is an example of declaring three long string variables which will be used to hold up to 2048 characters in each:

```
CONST MaxLen = 2048
DIM INTEGER Str1(MaxLen/8), Str2(MaxLen/8), Str3(MaxLen/8)
```

These will contain empty strings when created (ie, their length will be zero). When these variables are passed to the long string functions they should be entered as the variable name followed by empty brackets. For example:

```
LONGSTRING COPY Str1(), Str2()
```

Long string variables can be passed as arguments to user defined subroutines and functions. For example:

```
Sub MySub longarg() AS INTEGER
    PRINT "Long string length is" LLEN(longarg())
END SUB
```

And it could be called like this:

```
MySub str1()
```

Long String Commands

These are documented in detail in the *Commands* and *Functions* sections of this manual. The commands are:

LONGSTRING AES128 ENCRYPT/DECRYPT	Encrypts or decrypts a long string
LONGSTRING APPEND array%(), string\$ string	Append an ordinary string to a long string
LONGSTRING BASE64 ENCODE/DECODE using base 64	Encodes or decodes a long string
LONGSTRING CLEAR array%()	Clear (ie, set to empty) a long string
LONGSTRING COPY dest%(), src%()	Copy a long string
LONGSTRING CONCAT dest%(), src%()	Concatenate two long strings
LONGSTRING LCASE array%()	Convert a long string to lowercase
LONGSTRING LEFT dest%(), src%(), nbr long string	Get the left nbr characters from a long string
LONGSTRING LOAD array%(), nbr, string\$	Copy characters to a long string
LONGSTRING MID dest%(), src%(), start, nbr long string	Get characters from the middle of a long string
LONGSTRING PRINT [#n,] src%() [;]	Print a long string
LONGSTRING REPLACE array%(), string\$, start	Replace characters in a long string
LONGSTRING RESIZE addr%(), nbr	Set the length of a long string
LONGSTRING RIGHT dest%(), src%(), nbr long string	Get the right nbr characters from a long string
LONGSTRING SETBYTE addr%(), nbr, data	Set a byte in a long string
LONGSTRING TRIM array%(), nbr	Trim characters from the left of a long string
LONGSTRING UCASE array%()	Convert a long string to uppercase
LMID(array%(),start [,num])=string\$	Insert/replace text in a longstring

Long String Functions

<code>r = LGETBYTE(array%(), n)</code>	Return the value of a byte in a long string
<code>r\$ = LGETSTR\$(array%(), start, length)</code>	Returns part of a long string as a normal string.
<code>r = LINSTR(array%(), search\$ [,start] [,size])</code>	Returns the position of a string in a long string
<code>r = LLEN(array%())</code>	Returns the length of a long string
<code>r= LINPUT(array%(),fnbr,nbr)</code>	Reads nbr bytes from a file returning the number read
<code>MATH(BASE64 ENCODE/DECODE)</code>	Encodes or decodes data using base 64

Porting Programs

This chapter covers some of the considerations involved in porting programs to the Colour Maximite 2: first from the PicoMite, whose version of MMBasic the Colour Maximite 2 now shares, and then from the original Colour Maximite.

Running PicoMite Programs

The language of MMBasic version 7 is the same on the PicoMite and the Colour Maximite 2, so most PicoMite programs run with few or no changes. The differences come from the hardware:

- **The display.** The PicoMite's MODE numbers mean other resolutions on the Colour Maximite 2 (MODE 1 here is 800 x 600). Change a program's MODE statements as the table in the chapter Video Modes and Pages shows: the PicoMite's MODE 2 is MODE 7,4 here and its MODE 3 is MODE 8,4, with the same 16 colours, framebuffers and layers (FRAMEBUFFER) and the same screen memory for PEEK and POKE. TILEMAP, SPRITE and BLIT work on these modes as on the PicoMite.
- **Image slots.** FLASH LOAD IMAGE can load images into the slots 4 to 8 only, which are held in RAM; slots 1 to 3 (the PicoMite's flash slots) are not provided. A program loads its images every time it starts.
- **Drives.** On the PicoMite A: is its internal flash filesystem and B: its SD card. On the Colour Maximite 2 both A: and B: are the SD card, and a USB flash drive is C: on both. A program that uses B: works unchanged; one that uses A: uses the SD card.
- **I/O pins.** The PicoMite's pin names (GP0, GP1 and so on) are not accepted: use the pin numbers of the Colour Maximite 2's rear connector (see the chapter Hardware Features). The I/O devices are on different pins too, and the PicoMite's SETPIN commands that assign pins to COM ports, I2C and SPI are not needed here (each port has fixed pins).
- **Hardware the Colour Maximite 2 does not have:** WiFi and the WEB commands, Bluetooth, PIO, LCD display panels and touch, cameras, the RTC and keypad commands, the PicoMite's flash slots and filesystem. Programs that use these cannot run without changes.
- **CSUBs.** A CSUB built for the PicoMite runs unaltered if it uses only MMBasic's call table and none of the PicoMite's hardware (the Colour Maximite 2 provides the PicoMite's call table as well as its own). A CSUB can also be built for the Colour Maximite 2 with mmb2csub using --target cmm2.
- **MM.DEVICES** returns "Colour Maximite 2" (or "Colour Maximite 2 G2") so a program can tell which computer it is running on.

Porting from the Original Colour Maximite

There is a high degree of backwards compatibility in the Colour Maximite 2 and most programs written for the original Colour Maximite 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 the Colour Maximite 2, variables must be unique regardless of their type.

Floating Point

In the Colour Maximite 2 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. However, using the command `OPTION LEGACY ON` you can change these commands back to the same syntax as used in the original Colour Maximite. This option is not saved so it needs to be placed in the program before any graphic commands are used.

This option will also cause the drawing commands to accept colours in the range of 0 to 7. However, the colour shortcuts (red, blue, etc) are not defined as they were on the original Colour Maximite so, if these are required, they will need to be defined in the program as follows:

```
CONST Black = 0
CONST Blue = 1
CONST Green = 2
CONST Cyan = 3
CONST Red = 4
CONST Purple = 5
CONST Yellow = 6
CONST White = 7
```

You can also use `OPTION LEGACY OFF` to switch back to the Colour Maximite 2 syntax, so it is possible to mix both forms of the graphics commands in the one program.

The VGA display mode is not changed with this option so it should be specifically set to MODE 4, 8 for emulating the 480x432 resolution of the original Colour Maximite or MODE 5, 8 for emulating its 240x216 resolution.

The syntax of the MODE command has changed. The Scan Line Colour Override function is not supported in the Colour Maximize 2.

Fonts

The FONT command has a completely different purpose and syntax in the Colour Maximite 2. The Colour Maximite 2 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 included in the Colour Maximite 2 download zip.

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 the Colour Maximite 2 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.

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).

[illegible]

```

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 in 8 and 16-bit colour modes and transparent BLACK in 12-bit colour. A "0" is interpreted as NOTBLACK in 8 and 16-bit colour modes and solid BLACK in 12-bit colour.

SOUND and TONE

The SOUND command has a different purpose and syntax in the Colour Maximite 2 and should be replaced with the PWM command. The TONE command has been replaced by PLAY TONE.

SD Card and File Related Commands

The SD card is drive A: (also called B:). The commands PLAYMOD, LOADBMP and SAVEBMP have changed names: see PLAY MODFILE, LOAD BMP and SAVE IMAGE. CHAIN, the LIBRARY commands and DRIVE are available, as described in this manual.

Special Devices

LCD displays, Real Time Clocks (RTC) and keypads are not supported in the Colour Maximite 2. These can all be implemented in BASIC.

CONFIG Commands

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

Error Handling

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

Random Number Generator

The Colour Maximite 2 has an advanced random number generator based on an analog circuit that generates a sequence of true random numbers which will never repeat. For this reason RND cannot be seeded to give a repeatable sequence (RANDOMIZE is accepted, for programs written for the PicoMite, but it does not change the numbers that RND returns).

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 Characteristics

Naming Conventions

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

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

- Floating point. These can store a number with a decimal point and fraction (eg, 45.386) and also very large numbers. However, they will lose accuracy when more than 14 significant digits are stored or manipulated. The suffix is '!' and floating point is the default when a variable is created without a suffix
- 64-bit integer. These can store numbers with up to 19 decimal digits without losing accuracy but they cannot store fractions (i.e. the part following the decimal point). The suffix for an integer is '%'
- Strings. These will store a string of characters (eg, "Tom"). The suffix for a string is the '\$' symbol (eg, name\$, s\$, etc) Strings can be up to 255 characters long.

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

It is important to note that the alphanumeric name of the variable must be unique so you cannot have both a variable s\$ and s%. As a result, if a variable is declared without a type suffix, it can be used with the relevant suffix in a program and will still reference the same variable.

For file names see the chapter Program and Data Storage.

Constants

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

Decimal constants may be preceded with a minus (-) or plus (+) and may be terminated with 'E' followed by an exponent number to denote exponential notation. For example 1.6E+4 is the same as 16000. If the decimal constant contains a decimal point or an exponent, it will be treated as a floating point constant; otherwise it will be treated as a 64-bit integer constant.

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

Implementation Characteristics

- Maximum program size is 516KB of program memory. Note that MMBasic tokenises the program when it is loaded into program memory so the final size might vary from the plain text size.
- Maximum length of a command line, and of a line in a program, is 255 characters.
- Maximum length of a variable name or a label is 32 characters.
- Maximum number of variables is 1536: 1024 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, or of values in a DATA statement, is 75.
- Maximum number of nested FOR...NEXT loops is 100.
- Maximum number of nested DO...LOOP commands is 100.
- Maximum number of nested GOSUBs, subroutines and functions (combined) is 100.
- Maximum number of nested multiline IF...ELSE...ENDIF commands is 20.
- Maximum number of user defined subroutines and functions (combined) is 512. Labels are limited only by memory.

- Maximum number of structure types is 32, with up to 16 members each and nested up to 8 deep.
- Maximum number of interrupt pins that can be configured is 10.
- Numbers are stored and manipulated as double precision floating point numbers or 64-bit signed integers. 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 string length is 255 characters.
- Maximum line number is 65000.
- Maximum number of background pulses launched by the PULSE command is 5.
- Maximum number of sprites is 64 (#1 to #64) and of sprite collisions 8.
- Maximum number of files that can be open at once is 10, and the FILES command lists up to 2048.
- The maximum length of a filename/path is 127 characters.

MMBasic Firmware Memory Map

Below is a summary of how the MMBasic firmware makes use of the memory of the STM32H743 chip. This detail is not needed to use MMBasic but may be of interest if you want to dig deeper.

Address range (hex)	Size	Use
0000 0000 - 0000 FFFF	64K	ITCM RAM: the interrupt routines and some of the interpreter's code, run from here for speed.
0800 0000 - 080F FFFF	1024K	Flash bank 1: the MMBasic firmware and its constant data.
0810 0000 - 0811 FFFF	128K	Flash bank 2, sector 0: firmware too (the file manager and the editor).
0812 0000 - 081D FFFF	768K	Flash bank 2, sectors 1 to 6: the program in program memory (see OPTION FLASH) and the library (sector 6, 081C 0000 - 081D FFFF, when one is saved).
081E 0000 - 081F FFFF	128K	Flash bank 2, sector 7: the saved options.
2000 0000 - 2001 FFFF	128K	DTCM RAM: the firmware's own variables and, above them, the C stack (MM.INFO(HEAPEND) is the lowest it may go).
2400 0000 - 2407 FFFF	512K	AXI RAM: display page 0 (see the MODE command) and the display of the 4-bit modes.
3000 0000 - 3003 FFFF	256K	D2 RAM: the MMBasic heap in RAM, MM.INFO(HEAPI).
3004 0000 - 3004 7FFF	32K	D2 RAM: the firmware's buffers (sound's noise table, BLIT's buffers, the buffers for copies and lines).
3800 0000 - 3800 FFFF	64K	D3 RAM: MMBasic's variable records.
3880 0000 - 3880 0FFF	4K	Battery backed RAM: saved variables (VAR SAVE, VAR RESTORE and VAR CLEAR).
4000 0000 - 5FFF FFFF		The registers of the peripherals (GPIO, ADC, timers, SPI, DAC, USART, I2C...). MMBasic takes care of all this, however it is possible to PEEK and POKE these registers.
D000 0000 - D06F FFFF	7M	SDRAM (Generation 2): the display pages (see the MODE command).
D070 0000 - D077 FFFF	0.5M	SDRAM (Generation 2): program memory with OPTION RAM.
D078 0000 - D087 FFFF	1M	SDRAM (Generation 2): the compiled program (OPTION COMPILE).
D088 0000 - D1FF FFFF	23.5M	SDRAM (Generation 2): the MMBasic heap, MM.INFO(HEAPS): variables, arrays and buffers. Its top 512K (D1F8 0000 - D1FF FFFF) is where the code of a RAM library's CSUBs is kept.
D000 0000 - D02F FFFF	3M	SDRAM (Generation 1): the display pages.

D030 0000 - D037 FFFF	0.5M	SDRAM (Generation 1): program memory with OPTION RAM.
D030 0000 - D07F FFFF	5M	SDRAM (Generation 1): the MMBasic heap (from D038 0000 with OPTION RAM). While OPTION COMPILE is ON or SHADOW its bottom 1M is the compiled program, and the heap starts above it. Its top 512K (D078 0000 - D07F FFFF) is where the code of a RAM library's CSUBs is kept.

Compatibility

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

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

Predefined Read Only Variables

These variables are set by MMBasic and cannot be changed by the running program. Note that they do not do anything on their own, they must be printed or assigned to a variable.

For example:

```
> PRINT MM.VER
6.0002
>
```

or, in a program:

```
If MM.VER < 6.0002 Then Error "Needs version 6.00.02 or greater"
```

MM.CMDLINES\$	A string holding the command line arguments passed to the program: the text after the file name and its comma in the RUN command (eg, RUN "myprog", -f 3), the text after the file name in the * shortcut (eg, *myprog -f 3) or the 'cmdline\$' of CHAIN. The text given to RUN is not processed by MMBasic, so it can contain numbers, commas, quoted strings, etc: it is for the program to decode. It is an empty string when the program was run without arguments, from the editor (F2) or by OPTION AUTORUN.
MM.DEVICES\$	A string identifying the computer that MMBasic is running on: "Colour Maximize 2" on a Generation 1 and "Colour Maximize 2 G2" on a Generation 2. Other versions of MMBasic return their own name (eg, a PicoMite's), so a program can use it to tell which computer it is running on.
MM.ERRNO MM.ERRMSG\$ MM.ERRLINE	If a statement caused an error which was ignored (see ON ERROR) 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 holding the error message that would normally have been displayed on the console (without the "Error in line n:" in front of it). MM.ERRLINE is the number of the line that caused the error, as the error message gives it: a line of the program's file or, when the message names one, of an #INCLUDE file or of the library's file. It is zero if the error did not occur in a program line (eg, at the command prompt). They are reset to zero and an empty string by RUN and by ON ERROR IGNORE, SKIP, CLEAR and RESTART.
MM.FLAGS	The value of the system flag register, which the FLAG and FLAGS commands set (see also the FLAG() function).
MM.FONTHEIGHT MM.FONTWIDTH	Integers holding the height and the width, in pixels, of the characters of the current font, including its scaling (see FONT).
MM.HEIGHT MM.WIDTH	MM.HEIGHT is the number of lines of characters down the display and MM.WIDTH the number of characters across it, with the current font. When PAGE WRITE has selected another page they are for that page.
MM.HPOS MM.VPOS	The current horizontal and vertical position (in pixels) following the last graphics or PRINT command.
MM.HRES MM.VRES	Integers holding the horizontal and vertical resolution, in pixels, of the display in the current video mode (see MODE), or of the page or framebuffer being written to when PAGE WRITE or FRAMEBUFFER WRITE has selected another.

MM.I2C	Following an I²C write or read command this integer variable will be set to indicate the result of the operation as follows: 0 = The command completed without error. 1 = Received a NACK response 2 = Command timed out After I2C CHECK it is 0 if a device responded and non zero if none did. See Appendix B.
MM.INFO() MM.INFO\$()	MM.INFO returns information about the system: the keyword (or keywords) in the brackets says which, as listed in the rows below. The two versions can be used interchangeably (MM.INFO\$ is turned into MM.INFO when the line is entered) but good programming practice would require that you use the one corresponding to the returned datatype: MM.INFO\$ where the value is a string.
MM.INFO\$(DEVICE) MM.INFO(VERSION) MM.INFO(ERRNO) MM.INFO\$(ERRMSG) MM.INFO(I2C) MM.INFO(ONEWIRE) MM.INFO(FONTHEIGHT) MM.INFO(FONTWIDTH) MM.INFO(HPOS) MM.INFO(VPOS)	The same as MM.DEVICE\$, MM.VER, MM.ERRNO, MM.ERRMSG\$, MM.I2C, MM.ONEWIRE, MM.FONTHEIGHT, MM.FONTWIDTH, MM.HPOS and MM.VPOS.
MM.INFO(BBC BUFFER [n])	The number of free slots in the PLAY BBC SOUND queue of channel 'n' (0 to 3), or in the emptiest of the four channels if 'n' is omitted. Each channel queues up to 16 notes.
MM.INFO(BCOLOUR) MM.INFO(FCOLOUR)	The current background and foreground colours (see COLOUR). BCOLOR and FCOLOR are accepted too.
MM.INFO(CODE SIZE)	The size, in bytes, of the program in program memory, as MMBasic prepared it when it was loaded (see LIST FLASH).
MM.INFO\$(COMPILE)	What OPTION COMPILE did at the last RUN: OFF; "TEXT: " and why the program ran as text (eg, too big for the area, with the bytes it needed of those there are); or COMPILED followed by the number of compiles since the Colour Maximite 2 started (REUSED: the times the program had not changed), the statements in the compiled program (STMTS), those the last run executed from it (RAN), a count for testing the compiler (MISS), the statements run as compiled code (CODE), then its size of the area's (SIZE n OF m). Once a command has been typed at the command prompt it begins NONE instead of COMPILED.
MM.INFO(CPUSPEED)	The speed of the processor's clock in Hz, as a number: 400000000 for the Rev Y version of the STM32H743 and 480000000 for the Rev V (see the chapter Hardware Features), or 504000000 with OPTION OVERCLOCK ON. (The PicoMite returns the speed as a string: a PicoMite program that reads MM.INFO\$(CPUSPEED) into a string needs STR\$() here.)

MM.INFO\$(CURRENT) MM.INFO\$(PATH) MM.INFO\$(DIRECTORY)	MM.INFO\$(CURRENT) is the full name of the file that the program in program memory was loaded from, with its drive and directory (eg, "A:/GAMES/PACMAN.BAS"), or "NONE" if there is no program (eg, after NEW). This is the current program, which RUN, EDIT and LIST use when they are given no file name. MM.INFO\$(PATH) is the drive and directory of that file, ending with a "/" (eg, "A:/GAMES/"), or "NONE". A program can use it to find the files it needs relative to itself. MM.INFO\$(DIRECTORY) is the current directory, with its drive and ending with a "/" (eg, "A:/GAMES/"). See also CWD\$.
MM.INFO(DISK SIZE) MM.INFO(FREE SPACE) MM.INFO\$(DRIVE)	The size and the free space, in bytes, of the active drive (the SD card, or the USB flash drive when C: is the active drive), and the active drive itself: "A:" or "C:" (B:, the SD card's other name, is reported as "A:"). See DRIVE.
MM.INFO(EXISTS FILE file\$) MM.INFO(EXISTS DIR dir\$)	1 if the file 'file\$' exists and 0 if it does not (or if 'file\$' is a directory), and 1 if the directory 'dir\$' exists and 0 if it does not. The names can include a drive and a path. A directory on C: is reported as not existing when no USB flash drive is plugged in.
MM.INFO(FILESIZE file\$) MM.INFO\$(MODIFIED file\$)	The size of the file 'file\$' in bytes: -1 if it is not found and -2 if it is a directory. The date and time that 'file\$' was last modified, as a string in the form "YYYY-MM-DD HH:MM:SS". 'file\$' can also be a directory. It is an empty string if the file or directory is not found.
MM.INFO(FLASH)	The slot whose program is running: a flash slot, 1 to 3 (see FLASH RUN and FLASH CHAIN), or a RAM slot, 1 to 5 (see RAM RUN and RAM CHAIN), or 0 for the program in program memory.
MM.INFO(FLASH ADDRESS n)	The address of slot 'n' (1 to 8), as FLASH LOAD IMAGE, FLASH DISK LOAD and FLASH SAVE fill it: an image's first two 32-bit words are its width and height. A flash slot (1 to 3) is read from the SD card into RAM when it is first used after a RUN or NEW. An empty flash slot reads as erased flash memory (every byte &HFF), and an empty RAM slot (4 to 8) as a header whose width and height are zero, as on the PicoMite.
MM.INFO(FONT) MM.INFO(FONT ADDRESS n) MM.INFO(FONT POINTER n)	The number of the current font (see FONT). The address in memory of the data of font 'n' (1 to 16, 0 if there is no such font), and (POINTER) the address of the entry of MMBasic's font table that holds that address.
MM.INFO(FRAMEBUFFER) MM.INFO(FRAMEH) MM.INFO(FRAMEV)	The address in memory of the framebuffer made by FRAMEBUFFER CREATE (0 if there is none), and its width and height in pixels. The address is useful if you need to PEEK or POKE its contents. In the 4-bit colour depth FRAMEBUFFER makes the PicoMite's framebuffers, which are pages: use MM.INFO(WRITEBUFF) or MM.INFO(PAGE ADDRESS n) for them (see FRAMEBUFFER).
MM.INFO(HARDWARE)	1 on a Generation 1 Colour Maximite 2 and 0 on a Generation 2.

MM.INFO(HEAP) MM.INFO(IHEAP) MM.INFO(SHEAP) MM.INFO(MINHEAP)	The free space, in bytes, of the MMBasic heap, which holds strings, arrays, framebuffers and other buffers. MM.INFO(HEAP) is the total of the heap in the processor's RAM, MM.INFO(IHEAP), and the heap in the SDRAM, MM.INFO(SHEAP) (which does not count the top 512KB of the SDRAM, kept for the code of a RAM library's CSUBs). HEAPI and HEAPS are accepted as other names for IHEAP and SHEAP. See the memory map in the chapter MMBasic Characteristics. MM.INFO(MINHEAP) is the lowest address that the heap in the SDRAM, which is used from the top down, has reached since it was last cleared (by RUN, NEW or loading a program).
MM.INFO\$(ID) MM.INFO(ID48)	The unique 12 byte ID of the processor as a string of 24 hexadecimal digits, and a 48-bit hash of it as an integer. The hash is not guaranteed to be unique world wide but most likely is: it is suitable as a serial number for the device and fits in a CAN message.
MM.INFO\$(KEYBOARD)	"Connected" if a USB keyboard is connected and working, otherwise "Not Connected".
MM.INFO(LINES)	The number of lines of source code in the program in program memory: the lines read from its file and its #INCLUDE files when it was loaded.
MM.INFO(MAX PAGES) MM.INFO(PAGE ADDRESS n) MM.INFO(WRITE PAGE) MM.INFO(WRITEBUFF)	The highest page number that can be selected in the current video mode (see MODE and PAGE), and the address in memory of page 'n' (0 to MM.INFO(MAX PAGES)), which is useful to PEEK or POKE the contents of a page (see Video Memory in the chapter Video Modes and Pages). MM.INFO(WRITE PAGE) and MM.INFO(WRITEBUFF) (the PicoMite's name) are the address of the page that the drawing commands write to (see PAGE WRITE and FRAMEBUFFER WRITE).
MM.INFO(MODE)	The current video mode and colour depth as a number: the mode, then the depth as its decimals (eg, 1.8 for MODE 1,8, 8.16 for MODE 8,16, 1.12 for MODE 1,12 and 7.4 for MODE 7,4).
MM.INFO\$(OPTION option)	The current setting of one of the options that affect how a program runs. 'option' is one of: ANGLE: "RADIANS" or "DEGREES" (see OPTION ANGLE) AUTORUN: "On" or "Off" BASE: 0 or 1 (a number) BREAK: the character code of the break key (a number: 3 for Ctrl-C, 0 when it is disabled) CONSOLE: "Serial", "Screen", "Both" or "None" CONSOLE PORT: the serial port the console uses, 1 to 3 (a number: see OPTION CONSOLE PORT) DEFAULT: "Integer", "Float", "String" or "None" EXPLICIT: "On" or "Off" LEGACY: "On" or "Off" MOUSE: the channel set by OPTION MOUSE (a number), or -1 when none is set PROFILING: "On" or "Off" USBKEYBOARD: the layout of the USB keyboard: "UK", "US", "DE", "FR", "ES", "BE" or "IT", or "Disabled" (OPTION USBKEYBOARD DISABLE) Y_AXIS: "UP" or "DOWN"

MM.INFO(OVERRUN)	The number of samples that the last ADC START could not take because the next one was due before the conversion of the one before had finished: zero unless the sampling frequency was too high (see ADC).
MM.INFO\$(PIN pinno)	The status of I/O pin 'pinno' of the rear connector: "Unused", "In Use", "Reserved" (used by the Colour Maximite 2 itself) or "Invalid" (not an I/O pin).
MM.INFO\$(PLATFORM) MM.INFO\$(SYSTEM I2C)	These are the PicoMite's. On the Colour Maximite 2 both return an empty string, as a PicoMite with no OPTION PLATFORM and no OPTION SYSTEM I2C does, so that PicoMite programs that check them run.
MM.INFO(PROGRAM) MM.INFO(STACK) MM.INFO(HEAPEND)	The address in memory of the start of program memory (in the flash memory, or in the SDRAM with OPTION RAM), the current value of the processor's stack pointer, and the lowest the stack may go (the PicoMite's name): below it a program stops with the error "Stack overflow".
MM.INFO(PSRAM SIZE)	The size of the SDRAM in bytes: 8388608 (8MB) on a Generation 1 and 33554432 (32MB) on a Generation 2. This is the PicoMite's key for its PSRAM, which PicoMite programs read to see whether they can use the image slots 4 to 8 and the RAM commands: on the Colour Maximite 2 they can (see FLASH LOAD IMAGE and RAM SAVE).
MM.INFO\$(RESET)	The cause of the last restart: "Power-On" (the power was applied), "Switch" (the reset switch was pressed) or "Software" (a restart by MMBasic: CPU RESTART, a WATCHDOG timeout, or an OPTION command that restarts the Colour Maximite 2). It can be read once: afterwards it is an empty string.
MM.INFO(RTC)	1 if an external DS3231 real time clock is in use (OPTION DS3231 ON), otherwise 0.
MM.INFO(SAMPLE PLAYING n)	1 if channel 'n' (1 to 4) of PLAY MODSAMPLE is playing a sample, otherwise 0.
MM.INFO\$(SDCARD)	The status of the SD card: "Ready", "Not present" or "Unused" (a card is in the slot but has not been used yet).
MM.INFO\$(SEARCH PATH)	The directory set by OPTION SEARCH PATH, in which RUN at the command prompt looks for a program that is not in the current directory.
MM.INFO\$(SOUND) MM.INFO\$(TRACK)	What the audio output is doing: "OFF", "TONE", "WAV", "FLAC", "MP3", "MOD", "SOUND", "TTS", "ARRAY", "SAMPLE", "BBC" or "DAC" (the DAC command), or "PAUSED" followed by a space and what is paused (eg, "PAUSED WAV"). The name of the WAV, FLAC, MP3 or MOD file that is playing (or paused), or "OFF".
MM.INFO(UPTIME)	The time since the Colour Maximite 2 was restarted, in seconds, as a floating point number.
MM.INFO(VARCNT)	The number of variables in use: the global variables and the local variables of the subroutines and functions that are running. ERASE, CLEAR and the end of a subroutine or function lower it.
MM.ONEWIRE	Following a 1-Wire reset (ONEWIRE RESET, or ONEWIRE READ or WRITE with a reset) this integer variable is set to 1 if a device responded and to 0 if none did. See Appendix C.

MM.VER	The version number of the firmware as a floating point number in the form aa.bbcccdd, where aa is the major version number, bb the minor version number, cc the revision number and dd the beta number (none for a release). For example version 6.03.01 returns 6.0301 and version 7.00.00b2 returns 7.000002. A release candidate's dd is its number plus 50 (6.04.00RC1 returns 6.040051), so it is above every beta of the same version.
MM.WATCHDOG	An integer which is true (1) if MMBasic was restarted as the result of a watchdog timeout (see the WATCHDOG command), otherwise false (0). It stays true until a program ends, so the program that OPTION AUTORUN runs after the restart can see it.

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)

Operator	Description
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 to the power of n)
- (negation)	A minus sign in front of a value (rather than between two values) negates it. It applies after any exponentiation that follows, as in mathematics: -2^2 is -(2^2), which is -4, while (-2)^2 is 4. A minus after ^ belongs to the exponent: 2^-2 is 0.25.
* / \ MOD	Multiplication, division, integer division and modulus (remainder)
+ -	Addition and subtraction
x << y 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

Operator	Description
+	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: they compare the characters' codes, so "A" is less than "a".

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 the PicoMite firmware is restarted. Options that are not permanent will be reset on start-up, reset and in many cases when a program is run and/or exited.

Many OPTION commands will force a restart of the PicoMite firmware and that will cause the USB console interface to be reset. The program in memory will not be lost as it is held in non-volatile flash memory.

		Permanent?
OPTION ANGLE RADIANS DEGREES		This command switches trig functions between degrees and radians. Acts on SIN, COS, TAN, ATN, ATAN2, ACOS and ASIN, and on MATH ATAN3, MATH V_ROTATE, MATH Q_EULER, MATH Q_CREATE, MATH(C_POLAR), MATH(C_PHASE) and MATH SENSORFUSION. This is a temporary option that is reset to the default (RADIANS) when a program is run, when it ends, after an error and after CTRL-C, so it should be set at the top of any program that uses angles in degrees. MM.INFO(OPTION ANGLE) returns the current setting.
OPTION AUTORUN ON or OPTION AUTORUN OFF	✓	Instructs MMBasic to automatically run the program in program memory on power up or restart (for example, by the WATCHDOG timer). OFF is the default. The program is not run after the restart that an OPTION command causes. If the Colour Maximite 2 is set to run programs from RAM (OPTION RAM) MMBasic instead looks for the file AUTORUN.BAS on the SD card on power up or restart and runs it if it is found. Entering the break key (default CTRL-C) at the console will interrupt the running program and return to the command prompt. The NEW command turns this option off, and so does a firmware upgrade.
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 (otherwise the error is "Must be before DIM or LOCAL") and is reset to the default of 0 when a program is run.
OPTION BASELINE ON or OPTION BASELINE OFF	✓	OPTION BASELINE ON makes the Colour Maximite 2 run as a 400MHz Generation 1 machine. This is intended to allow testing of code on the lowest common denominator. OPTION BASELINE and OPTION OVERCLOCK cannot both be on: turning one on turns the other off, and OPTION BASELINE OFF (or OPTION OVERCLOCK OFF) returns to the normal speed. This command will cause a reboot.

OPTION BAUDRATE nbr	✓	Set the baud rate for the serial console to 'nbr' (at least 100). The default is 115200. A Generation 2 runs well at 460800, but the USB to serial converter of a Generation 1 cannot run at that speed. The new rate is saved and is used from the next restart (MMBasic replies "Restart to activate"): the terminal emulator must then be set to match. A firmware upgrade returns it to 115200. Using this command it is possible to set the console to an unworkable baud rate. It can then be changed using a USB keyboard and VGA monitor, or returned to the default by resetting MMBasic (see Resetting MMBasic in the chapter Using MMBasic). This command can only be run from the command line.
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 break key is set to CTRL-C whenever MMBasic returns to the command prompt, but it can be changed to any keyboard key using this command in a program (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. MM.INFO(OPTION BREAK) returns the current value.
OPTION CASE LOWER UPPER TITLE	✓	Change the case used for listing command and function names when using the LIST command. The default is TITLE but the old standard of MMBasic can be restored using OPTION CASE UPPER.
OPTION COLOURCODE ON or OPTION COLOURCODE OFF or OPTION COLOURCODE REVERSE	✓	Turn on or off colour coding for the editor's output. Keywords will be in cyan, comments in yellow, etc. The default is ON. Only BASIC programs (.bas files) are colour coded. OPTION COLOURCODE REVERSE is the same as OPTION COLOURCODE OFF except that the editor and the file manager show black text on a white background. The keyword COLORCODE (USA spelling) can also be used. On the serial console colour coding requires a terminal emulator that can interpret the VT100 escape codes (for example, Tera Term).

OPTION COMPILE ON or OPTION COMPILE OFF or OPTION COMPILE SHADOW	✓	Runs programs from a compiled form of the program, as the PicoMite does. When a program is run (by RUN, CHAIN, FLASH RUN, RAM RUN or the library's restart), MMBasic compiles it into statement records, if it has changed since it was last compiled, and runs them in place of its text. Assignments and expressions on numbers and simple strings, FOR, DO, IF, SELECT CASE, calls of SUBs and FUNCTIONS, LOCAL and STATIC run several times faster (a tight loop up to ten times); any statement the compiler does not handle runs as it always has. The program is unchanged: LIST, EDIT, error messages and their lines, TRACE, interrupts and ON ERROR behave as without the option, and commands typed at the command prompt are not compiled. The default is OFF. OPTION COMPILE SHADOW also runs each compiled calculation through the interpreter, and stops with an error giving both values if they differ: it is a check of the compiler, and slower than OFF. The compiled program is kept in 1 MB of RAM reserved for it; a program too big for that area runs as text. MM.INFO\$(COMPILE) reports what the last RUN did. A program runs as text while OPTION PROFILING is on. On a Generation 1 Colour Maximite 2 the 1 MB is taken from the memory for variables, and only while the option is ON or SHADOW. The first OPTION COMPILE ON or SHADOW after a restart saves the option and restarts the CMM2 to make that area, so it is then not allowed in a program; OPTION COMPILE OFF keeps the area until the next restart.
OPTION CONSOLE BOTH [font [, fc [, bc]]] or OPTION CONSOLE SCREEN or OPTION CONSOLE SERIAL or OPTION CONSOLE NONE		OPTION CONSOLE SCREEN will disable the serial console for both input and output and direct all output to the VGA monitor. This will allow the VGA output to update much faster. With this option the serial port used for the console can be opened as COM3. OPTION CONSOLE SERIAL will disable the console output to the VGA screen and send all output to the serial console. This is useful for debugging graphics applications as diagnostic PRINT statements will not corrupt the screen display. OPTION CONSOLE BOTH will enable both the serial console and the VGA screen for console input and output. This is the default unless OPTION CONSOLE SAVE has saved another setting. The optional 'font' (1 to 8), 'fc' and 'bc' set the font and the foreground and background colours of the console's text on the screen (by default white on black); OPTION CONSOLE SAVE keeps them. OPTION CONSOLE NONE (the PicoMite's) sends the output to neither. The serial console, if it was running, still takes input, so CTRL-C still works. These settings are temporary: when a program ends or an error occurs the console returns to the saved setting. OPTION CONSOLE BOTH and OPTION CONSOLE SERIAL give the error "Console port in use" while COM3 is open. MM.INFO(OPTION CONSOLE) returns the current setting.

OPTION CONSOLE PORT n	✓	Selects the serial port used as the console: 1 (COM1), 2 (COM2) or 3 (the default: the serial over USB on the USB Type-B connector). With 1 or 2 all console input and output goes through that port, which allows easier remote operation using a WiFi to UART module or a radio link (see ESP-01 Module in the chapter Colour Maximite 2 Generation 2), and COM3 is available on the USB Type-B connector. This command will cause a reboot. This setting is kept through a firmware upgrade.
OPTION CONSOLE SAVE	✓	Saves the current console setting (OPTION CONSOLE BOTH, SCREEN, SERIAL or NONE) as the one used at power up and whenever a program ends. If you are using the Colour Maximite 2 as a stand alone computer it is recommended that you execute OPTION CONSOLE SCREEN, then OPTION CONSOLE SAVE, to permanently disable the serial console and thereby eliminate serial I/O overhead. This command will cause a reboot.
OPTION CRLF CR LF CRLF		Defines what the USB keyboard will send when the Enter key is pressed: CR (a carriage return), LF (a line feed) or CRLF (both, the default). This command can only be used in a program. The setting returns to CRLF when a program is loaded and when it ends.
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 or the error "Variable type not specified" will occur. When a program is run the default is set to FLOAT for compatibility with Microsoft BASIC and previous versions of MMBasic. MM.INFO(OPTION DEFAULT) returns the current setting.
OPTION DEFAULT MODE n	✓	Specifies the video mode for the command prompt, the editor and the file manager, which is also the mode a program starts in. The colour depth is 8 bits. 'n' can be: 1 = 800x600, 100 x 50 characters (the default) 8 = 640x480, 80 x 40 characters 9 = 1024x768, 102 x 48 characters 10 = 848x480, 106 x 40 characters (widescreen) 11 = 1280x720, 128 x 45 characters (widescreen) 12 = 960x540, 120 x 45 characters (widescreen, lines duplicated) 14 = 960x540, 120 x 45 characters (widescreen, lines not duplicated: preferred over 12 but not supported by all monitors) 15 = 1280x1024, 160 x 85 characters 16 = 1920x1080, 120 x 45 characters (widescreen, Generation 2 only) 18 = 1024x600, 102 x 37 characters The characters are the columns and lines of the screen in the editor's font, which this option also sets: NORMAL (8x12 pixels) in modes 1, 8, 10, 12, 14 and 15, MEDIUM (10x16) in modes 9, 11 and 18 and VERY LARGE (16x24) in mode 16. OPTION EDIT FONT can then change it. This command can only be run from the command line and will cause a reboot. This setting is kept through a firmware upgrade.

OPTION DS3231 ON or OPTION DS3231 OFF	✓	<u>GENERATION 2 ONLY</u> Enables and disables the use of the optional DS3231 real time clock (see High Accuracy Real Time Clock in the chapter Colour Maximize 2 Generation 2) for providing the time and date. If OPTION MILLISECONDS is ON the processor's internal clock is always used for the time. On a Generation 1 this command gives the error "DS3231 not available on this H/W". This command will cause a reboot. This setting is kept through a firmware upgrade.
OPTION EDIT FONT SMALL NORMAL MEDIUM LARGE VERY LARGE	✓	Sets the font to be used in the editor and the file manager: SMALL (6x8 pixels), NORMAL (8x12, the default), MEDIUM (10x16), LARGE (12x20) or VERY LARGE (16x24). OPTION DEFAULT MODE also sets it, to the font that suits the mode. This command can only be run from the command line. This setting is kept through a firmware upgrade.
OPTION ERROR ABORT or OPTION ERROR CONTINUE		Sets what happens when an operation on a file on the SD card or the USB drive fails (for example, opening a file that does not exist). With OPTION ERROR ABORT (the default) the program stops with an error message. With OPTION ERROR CONTINUE it continues, and MM.ERRNO and MM.ERRMSG\$ report the error. It is reset to ABORT when a program is run. See also the ON ERROR command, which handles all errors.
OPTION ESCAPE [OFF]		Enables the ability to insert escape sequences into string constants. See the section Special Characters in Strings in the chapter Variables and Expressions. OPTION ESCAPE OFF will disable it. It is turned off when a program is run, so OPTION ESCAPE should be placed at the start of the program.
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 should be placed before any variables are used. MM.INFO(OPTION EXPLICIT) returns the current setting.
OPTION F11 string\$ or OPTION F12 string\$ or OPTION F15 string\$ or OPTION F16 string\$ or OPTION F19 string\$ or OPTION F20 string\$	✓	Define the string that will be generated when a function key is pressed at the command prompt: F11, F12, F15 (Shift-F3), F16 (Shift-F4), F19 (Shift-F7) or F20 (Shift-F8). The maximum string length is 63 characters. Setting the string to "" removes it. If the string ends with a carriage return and line feed the command is run at once. Example: <pre>OPTION F11 "RUN " + CHR\$(34) + "myprog" + CHR\$(34) + CHR\$(13) + CHR\$(10)</pre> The other function keys are fixed (see Shortcut Keys in the chapter Using MMBasic).

OPTION FLASH [n]	✓	Specifies that programs are to be run from flash memory (the default), so that the program is preserved when the Colour Maximite 2 is powered off or reset. It undoes OPTION RAM. The flash is not written every time a program is run: it is only written if it does not match the newly tokenised program, ie, a different or altered program is being loaded. The optional parameter 'n' (1 to 6, default 1) specifies where the program starts, in 128KB sectors of the flash: the maximum program size is 512KB for 'n' of 1 to 3, 384KB for 4, 256KB for 5 and 128KB for 6. Moving the program can spread the wear on the flash, although this is unlikely ever to be needed. A library saved in flash uses sector 6 (see the LIBRARY command), which a program then cannot use (the error is "Library is using flash slot 6"). This command can only be run from the command line and will cause a reboot.
OPTION HORIZONTAL OFFSET mode, offset	✓	Since different VGA monitors and HDMI converters interpret the analog sync signals slightly differently, the screen image may appear off-centre (eg, a black bar on one side or the edge of the text being cut off). This command allows you to calibrate the Colour Maximite 2's signal to match your specific hardware without needing to use the monitor's adjustment buttons every time. For each 'mode' (1 to 18), 'offset' is an adjustment in pixels to the start of the horizontal lines on the VGA display. The default is 0 and the allowed range is -128 to 127. This command will cause a reboot.
OPTION KEYBOARD REPEAT firstchar, nextchar	✓	Define the repeat characteristics of the USB keyboard when a key is held down. 'firstchar' is the time in milliseconds before the key first repeats: the default is 500 mSec and the valid range is 100 to 2000 mSec. 'nextchar' is the time in milliseconds between its subsequent repeats: the default is 75 mSec and the valid range is 25 to 2000 mSec.
OPTION LEGACY ON or OPTION LEGACY OFF		This will turn on or off compatibility mode with the graphic commands used in the original Colour Maximite. The commands LINE, CIRCLE and PIXEL use the legacy syntax, all drawing commands (and COLOUR) will accept colours in the range of 0 to 7, and fonts 1 to 3 are those of the original Colour Maximite. Notes: - Keywords such as RED, BLUE, etc are not implemented so they should be defined as constants if needed. - The video mode is not changed by this option so it should be set to MODE 4 to emulate the 480x432 resolution of the original Colour Maximite or MODE 5 for its 240x216 resolution. The colour depth must be 8 bits (otherwise the error is "Display must be in 8-bit mode for legacy use"). - Refer to the Colour Maximite MMBasic Language Manual for the syntax of the legacy commands. This can be downloaded from https://geoffg.net/OriginalColourMaximite.html. This command can only be used in a program. The option is not saved: it is turned off when the program ends. MM.INFO(OPTION LEGACY) returns the current setting.

OPTION LIST		This will list the processor's speed and the firmware's version, the video mode in use (and the default mode, when OPTION DEFAULT MODE has set one other than mode 1), the size of the console in lines and characters, and then the settings of the permanent options that have been changed from their default setting and the current console setting.
OPTION LOCAL VARIABLES n		This command is accepted, for programs written for the PicoMite, and ignored. On the PicoMite it sets how many of the variables are kept for local variables; the Colour Maximite 2 already has room for more local variables than it can set. 'n' must be 1 to 100000.
OPTION MAXCTRLS n	✓	Sets the number of GUI controls that a program can create (see the chapter GUI Controls). 'n' can be 0 to 2000 and the controls are numbered 1 to 'n'. The default is 0, and then a program that creates a control stops with the error "No memory allocated for GUI controls". The controls are clicked with a USB mouse, a mouse set with OPTION MOUSE, or GUI CLICK. This command will cause a reboot. This setting is kept through a firmware upgrade.
OPTION MILLISECONDS ON or OPTION MILLISECONDS OFF		Specifies that the TIMES\$ function will, or will not, include milliseconds as a decimal fraction of the seconds in its output (HH:MM:SS.mmm). The default is OFF. The milliseconds come from the processor's internal clock, so while this option is ON the internal clock provides the time even if OPTION DS3231 is ON. The option is not saved: it returns to OFF when a program ends or an error occurs.
OPTION MOUSE channel [, sensitivity] or OPTION MOUSE OFF	✓	Sets the mouse used by the editor and the GUI controls. 'channel' is 0 for a PS2 mouse, or 1 to 3 for a Hobbytronic USB mouse interface on that I ² C channel. The optional 'sensitivity' can be 0 to 8 for a PS2 mouse and 0 to 10 for the Hobbytronic interface (see Mouse Support in the chapter Hardware Features). While this option is set, a USB mouse plugged into the USB connector is used by the editor and the GUI controls in place of the mouse set here. If a Hobbytronic interface is set but not connected the error "Mouse not connected" is reported when the editor is started. OPTION MOUSE OFF (the default) disables it. MM.INFO(OPTION MOUSE) returns the channel, or -1 when it is off. This command can only be run from the command line and will cause a reboot. This setting, with its sensitivity, is kept through a firmware upgrade.

OPTION MOUSE SENSITIVITY n	✓	Sets the speed of a USB mouse: its movement is divided by 'n', so a number above 1 slows the mouse down and a number below 1 speeds it up. 'n' can be 0 to 1000 and need not be a whole number; 0 or 1 (the default) leaves the mouse as it is. The setting applies at once, in the editor and in programs. The firmware always reads the full report of a USB mouse (its wheel and all its buttons), so unlike the PicoMite this option does not change how the mouse is read. This command can only be run from the command line. The setting is kept through a firmware upgrade; OPTION RESET returns it to the default.
OPTION OVERCLOCK ON or OPTION OVERCLOCK OFF	✓	OPTION OVERCLOCK ON runs the processor at 504MHz in place of 480MHz (or 400MHz on processors of revision Y). This is the fastest speed at which most of both the original and the Generation 2 versions of the Colour Maximite 2 run reliably. OPTION OVERCLOCK and OPTION BASELINE cannot both be on: turning one on turns the other off, and OPTION OVERCLOCK OFF (or OPTION BASELINE OFF) returns to the normal speed. This command will cause a reboot.
OPTION PIN nbr	✓	Set 'nbr' as the PIN (Personal Identification Number) for access to the console prompt. 'nbr' can be any non zero number of up to eight digits. Whenever a running program tries to exit to the command prompt for whatever reason MMBasic will request this number before the prompt is presented. This is a security feature as without access to the command prompt an intruder cannot list or change the program in memory or modify the operation of MMBasic in any way. To disable this feature enter zero for the PIN number (ie, OPTION PIN 0). A permanent lock can be applied by using 99999999 for the PIN number: the console then only offers a restart. If a permanent lock is applied or the PIN number is lost the only way to recover is to reset MMBasic to its original configuration, which erases the program (see Resetting MMBasic in the chapter Using MMBasic).
OPTION PROFILING ON or OPTION PROFILING OFF	✓	Turns the profiler on or off. While it is on, each time a program is run MMBasic counts how many times each line of the program runs and the time spent in it. After the program has run, LIST PROFILE lists the program with, in front of each line, the number of times it ran and the average time it took in microseconds, and LIST PROFILE CSV writes the same to a file named after the program with the extension .CSV. The counters take 8 bytes of the heap for each line of the program, and collecting them makes the program run a little slower. MM.INFO(OPTION PROFILING) returns the current setting. This command will cause a reboot.

OPTION RAM	✓	Causes programs to be loaded into RAM to run rather than into flash memory. This makes loading somewhat faster and avoids wearing the flash. Program performance is similar to running from flash. The program is lost when the power is removed, on a reset and when an option that restarts MMBasic is changed. On the Generation 1 the memory available for variables is reduced by 512KB; on the Generation 2 it is not. With OPTION AUTORUN ON, MMBasic runs the file AUTORUN.BAS from the SD card on power up. OPTION FLASH returns to running programs from flash. This command can only be run from the command line and will cause a reboot.
OPTION RESET	✓	Reset all saved options to their default values (those kept through a firmware upgrade too). The program in program memory and the saved variables are not erased. This command can only be run from the command line and will cause a reboot.
OPTION RTC CALIBRATE n	✓	Used to calibrate the battery backed real time clock that keeps time in the Colour Maximite 2. 'n' is a number between -511 and +512. A change of ± 1 should equate to about 0.0824 seconds per day. Negative numbers will slow the clock down, positive will speed it up (different from the Micromite). This setting is kept through a firmware upgrade.
OPTION RTC DRIVE n	✓	Sets the drive strength of the oscillator for the real time clock's 32.768kHz crystal: 0 (low), 1 (medium low), 2 (medium high, the default) or 3 (high). A higher drive can help a crystal that is slow to start; a lower one takes less current from the battery. This command can only be run from the command line and will cause a reboot. This setting is kept through a firmware upgrade.
OPTION SD TIMING NORMAL or OPTION SD TIMING FAST	✓	The fast option will speed up the timing for SD card access. This results in read/write speeds being about 20% faster where cards can accommodate the higher speed access. The default is NORMAL. This command can only be run from the command line and will cause a reboot. This setting is kept through a firmware upgrade.
OPTION SEARCH PATH pathname\$ or OPTION SEARCH PATH DISABLE	✓	Defines a directory which will be searched when a program is run from the command prompt with the RUN command or its short form (*), or with EXECUTE, and the file does not exist in the current directory. 'pathname\$' must start and end with a forward slash (eg, "/programs/") and can be up to 127 characters long. OPTION SEARCH PATH DISABLE removes it. MM.INFO(SEARCH PATH) returns it.
OPTION SERIAL PULLUP ENABLE or OPTION SERIAL PULLUP DISABLE	✓	Enable or disable the pullup resistor (about 100K) that MMBasic places on the receive (Rx) line of a serial port (COM1, COM2 or COM3) when the port is opened. The default is ENABLE. See Appendix A for when it should be disabled. This command can only be run from the command line and will cause a reboot.

OPTION SLEEP n	✓	Turns the monitor off (the VGA sync signals are stopped, so that the monitor goes into its power saving mode) when no key has been pressed for 'n' minutes (1 to 255), at the command prompt or while a program runs. Any key press will wake the monitor, and the key is still read by the prompt or the program. Set 'n' to 0 (the default) to disable this.
OPTION STATUS ON OFF	✓	Enable or disable the status line at the bottom of the VGA screen at the command prompt. The status line shows the current directory, the "current program filename" used by the commands RUN, LIST and EDIT when a file name is not specified, and the time and date. The default is ON. This command can only be run from the command line.
OPTION SYMBOLS OFF		This command is accepted, for programs written for the PicoMite, and changes nothing. On the PicoMite it makes a program run without the symbol tables that MMBasic normally builds, to save the memory they use. As on the PicoMite, it must be the first command in the program (only blank lines and comments can come before it) and it cannot be used at the command prompt; elsewhere it gives an error.
OPTION TAB 2 3 4 8	✓	Set the spacing for the tab key in the editor. The default is 2.
OPTION TRACECACHE ON n or OPTION CACHE SUB name [, name ...] or OPTION CACHE DEBUG		These commands are accepted and ignored, for programs written for PicoMite firmware before V6.04.00, which had a statement cache set up with them (the cache has since been removed from the PicoMite, where they are now errors). Anything after the keywords is ignored.
OPTION USB POLLING n	✓	Sets how often, in milliseconds, MMBasic services the devices on the USB connector (the keyboard, the mouse, the gamepads, USB serial ports and the USB drive) and does its other routine checks. 'n' can be 1 to 64; the default is 2. A smaller value makes the devices more responsive and a larger one leaves a little more time for the program. This command can only be run from the command line.

OPTION USBKEYBOARD nn [, noLED] or OPTION USBKEYBOARD DISABLE	✓	Set the layout of the USB keyboard. ‘nn’ is a two character code: US for the standard keyboard layout in the USA, Australia and New Zealand, UK for the United Kingdom (the default), DE for Germany, FR for France, ES for Spain and BE for Belgium. The layout is also asked for when MMBasic is first loaded. The optional ‘noLED’ parameter can be set to 1 to block sending the command to the keyboard that lights the LEDs relating to Caps Lock etc. This may be needed on some keyboards which do not process this command properly and may lock up. It defaults to 0 if not specified (ie, the LED commands are sent). OPTION USBKEYBOARD DISABLE stops MMBasic using a USB keyboard: one that is plugged in is ignored (a message on the console says so), and the console must then be used through the serial port. MM.INFO(OPTION USBKEYBOARD) returns the layout, or "Disabled". This command can only be run from the command line and will cause a reboot. This setting is kept through a firmware upgrade.
OPTION VCC voltage		Specifies the voltage (Vcc) supplied to the ARM Cortex-M7 chip, from 1.8 to 3.6. When using the analog inputs to measure voltage the chip uses its supply voltage (Vcc) as its reference. This voltage can be accurately measured using a DMM and set using this command for more accurate measurement. The parameter is not saved and should be initialised either on the command line or in a program. The default if not set is 3.3.
OPTION VGA OUTPUT HIGH MEDIUM LOW	✓	Selects the colour look up table (CLUT) used to drive the VGA output: HIGH (the default) drives each colour up to its full level, MEDIUM up to about 7/8 of it and LOW up to 3/4. MEDIUM and LOW make the picture darker, for a monitor that shows it too brightly. This command can only be run from the command line and will cause a reboot.
OPTION Y_AXIS DOWN UP		This command can only be used in a program and inverts the y axis for the drawing commands: with UP, 0,0 is at the bottom left as in a typical graphing application. This is a temporary option that is reset to the default (DOWN) when programs end, after an error and after CTRL-C. MM.INFO(OPTION Y_AXIS) returns the current setting.

Commands

Square brackets indicate that the parameter or characters are optional.

Where a keyword contains COLOUR or GREY the American spellings COLOR and GRAY are accepted as well - for example COLOR, MAP GRAYSCALE, OPTION DEFAULT COLORS and MM.INFO(FCOLOR). Both spellings mean the same thing throughout and only the British spelling is listed below.

' (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
[file] [cmdline]	The star/asterisk command is a shortcut for RUN that may only be used at the MMBasic prompt. eg, <pre> RUN * -f RUN -f *foo RUN "foo" *foo --wombat RUN "foo", --wombat *foo "wom" RUN "foo", "wom" *"foo bar" -x RUN "foo bar", -x</pre> The file name ends at the first space, or it is given in quotes, which allows a name with spaces. Anything after the name is passed to the program as its command line (MM.CMDLINE\$) exactly as it was typed: it is not evaluated. See the RUN command.
? (question mark)	Shortcut for the PRINT command.
/* */	Start and end of multi-line comments. /* and */ must be the first non-space characters on a line. When the program is loaded (by RUN, or by F1 or F2 in the editor or AUTOSAVE) the lines from the one starting with /* to the one starting with */ are dropped, so they take no memory and can be used anywhere in a program or an #INCLUDE file, inside subroutines and functions too (but not at the command prompt). Any characters after /* or */ on these lines are also treated as a comment, and a line that starts with /* and also contains */ is a comment on its own. A /* without a matching */ gives the error "No matching */". See also #COMMENT.
#COMMENT START or #COMMENT END	Directives for multi-line comments. When the program is loaded (by RUN, or by F1 or F2 in the editor or AUTOSAVE) the lines between #COMMENT START and #COMMENT END are dropped: they are not loaded into program memory, and a directive among them (an #INCLUDE or a #DEFINE) has no effect. The directives must be the first non-space characters on their lines. They can also be used in an #INCLUDE file. See also /* and */.
#DEFINE "before", "after"	This directive causes the text "before" to be replaced by the text "after" in the lines of the program that follow it, when the program is loaded (by RUN, or by F1 or F2 in the editor or AUTOSAVE). Both parameters must be quoted strings. Only whole words are replaced (#DEFINE "N", "M" does not change the N in PRINT), case is ignored, and quoted strings and comments in the program are not changed. The replacements are also made in the #INCLUDE files that follow. The #DEFINES are applied in the reverse order of their creation, so a symbol can be redefined and from that point on in the program the new definition will be active. The program can have up to 256 #DEFINE directives ("Too many #DEFINE statements"). A line that the replacements make longer than 255 characters gives the error "Line n is too long".

#INCLUDE file\$	This directive inserts the file ‘file\$’ into the program at that point when the program is loaded (by RUN, or by F1 or F2 in the editor or AUTOSAVE). ‘file\$’ is a quoted string; ".INC" is added if it has no extension, and the file must have the extension .INC. Unless ‘file\$’ starts with a drive or a / it is looked for relative to the directory holding the program, not the current directory, so CHDIR in the program does not affect it. It can be on the SD card or the USB flash drive. This acts exactly as if the included file was inserted into the code with an editor, but it is more convenient for libraries and other code that several programs share (see also LIBRARY). An included file cannot have an #INCLUDE of its own ("Can't import from an import"); #DEFINE, #COMMENT, #MMDEBUG and /* */ work in it. The program is loaded from its file each time it is run, so a change to an included file takes effect at the next RUN. Errors in an included file are reported with the file's name and the line number in it.
#MMDEBUG ON or #MMDEBUG OFF	Directives for the program loader, which can appear anywhere in the program. While #MMDEBUG is OFF (the default at the start of the program) the lines that start with the command MMDEBUG are dropped when the program is loaded, as comments are, so they cost nothing in the program's size or speed. After #MMDEBUG ON they are kept. An #INCLUDE file starts with the setting in force at its #INCLUDE. See the MMDEBUG command.
A: , B: or C:	Shortcut for DRIVE “A:” ,DRIVE “B:” or DRIVE “C:” at the command prompt
ADC OPEN freq, pin1 [, pin2] [, pin3] [, interrupt]	The ADC commands capture up to three channels of analog data in the background into arrays, at up to 500KHz per channel (480KHz on a 400MHz processor), with an optional trigger. See also the chapter <i>Using the I/O Pins</i>. ADC OPEN opens the channels. ‘freq’ is the sampling frequency in Hz. The number of bits in each conversion follows from it and is applied automatically: • Above 160KHz the conversion is 8 bits per channel • From 40KHz to 160KHz it is 10 bits • From 20KHz to 40KHz it is 12 bits • From 10KHz to 20KHz it is 14 bits • Below 10KHz it is 16 bits Each channel has its own converter. ‘pin1’, the first channel, can be one of the pins 7, 10, 16, 22, 24 or 37, ‘pin2’ (the second) one of 8, 12, 26 or 29 and ‘pin3’ (the third) 13 or 15. The pins are reserved for the ADC until ADC CLOSE. ‘interrupt’ is a normal MMBasic subroutine that will be called when the conversion completes. It is always the fifth argument, so the arguments of unused channels are left empty, for example: ADC OPEN 100000, 7, , , Done
ADC FREQUENCY freq	Changes the sampling frequency without closing and opening the ADC. The new frequency is used by the next ADC START (it does not change a capture already running). This command is only valid if the number of bits given by the table above does not change; otherwise use ADC CLOSE and ADC OPEN.
ADC TRIGGER channel, level [, timeout]	Sets up triggering of the ADC. This is given after ADC OPEN and before the ADC START command.

ADC START array1!() [, array2!() [, array3!() [, C1min, C1max] [, C2min, C2max] [, C3min, C3max]	The ‘channel’ can be a number between one and three depending on the number of pins specified in the ADC OPEN command. The ‘level’ can be between -VCC and VCC (VCC is OPTION VCC, normally 3.3). A positive number indicates that the trigger will be on a positive going transition through the specified voltage. A negative number indicates a negative going transition through the specified voltage (without its sign). ‘timeout’ is the number of ADC samples to take before abandoning the wait for the trigger condition and starting the capture anyway. Setting a value equal to the frequency of the sampling would give a timeout of 1 second. If it is 0 or omitted the ADC waits for ever. The trigger applies to every ADC START until the ADC is closed. This starts conversion into the specified arrays, one for each channel opened (in the order of their pins in ADC OPEN). The arrays must be floating point and the same size. The size of the arrays defines the number of conversions. Start can be called repeatedly once the ADC is OPEN. ‘C_xmin’ and ‘C_xmax’ will scale the readings of channel x. For example, C1min=200 and C1max=100 will create values ranging from 200 to 100 for equivalent voltages of 0 - 3.3. If the scaling is not used the results are returned as a voltage between 0 and OPTION VCC (defaults to 3.3V). The scaling values are always the fourth to the ninth arguments, so the arguments of the arrays not used are left empty, for example: ADC START a!() , , , 0, 100 If ADC OPEN did not specify an interrupt the command waits until the arrays are filled. If it did, the command returns at once and the interrupt is called when they are full: the arrays must then be global (or STATIC), not LOCAL, as they are filled after the command has returned.
ADC CLOSE	Closes the ADC and returns the pins to normal use.
ARRAY ADD in(), value ,out()	This adds (or appends for strings) the value ‘value’ to every element of the matrix in() and puts the answer in out(). Works for arrays of any dimensionality of strings and both integer and float (can convert between integer and float). Setting ‘value’ to 0 or “” is optimised and is a fast way of copying an entire array. in() and out() can be the same array.
ARRAY INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5] , sourcearray()	This is the opposite of ARRAY 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 or a one dimensional array of strings into a two dimensional array of strings. The arrays can be numerical or strings and ‘sourcearray’ and ‘targetarray’ must be the same (NB: can convert between integers and floats for numerical arrays). String arrays must have the same LENGTH. eg, <pre>OPTION BASE 1 DIM targetarray(3,4,5) DIM sourcearray(4)=(1,2,3,4) ARRAY INSERT targetarray(), 2, , 3, sourcearray()</pre> Will set elements 2,1,3 = 1 and 2,2,3 = 2 and 2,3,3 = 3 and 2,4,3 = 4
ARRAY SET value, array()	Sets all elements in array() to the value ‘value’. Value can be a number or a string and ‘array’ must be the same (NB: can convert between integers and floats). Note this is the fastest way of clearing an array by setting it to zero or an empty string.

ARRAY 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 indices and there should be as many indices in the command, including the blank one, as there are dimensions in the source array. The arrays can be numerical or strings and 'sourcearray' and 'destinationarray' must be the same (NB: can convert between integers and floats for numerical arrays). String arrays must have the same LENGTH. eg, <pre>OPTION BASE 1 DIM a (3, 4, 5) DIM b (4) ARRAY SLICE a () , 2, , 3, b ()</pre> Will copy the elements 2,1,3 and 2,2,3 and 2,3,3 and 2,4,3 into array b()
ARC x, y, r1, [r2], a1, a2 [, c]	Draws an arc of a circle with a given colour and width between two radials (defined in degrees). Parameters for the ARC command are: x: X coordinate of centre of arc y: Y coordinate of centre of arc r1: inner radius of arc r2: outer radius of arc - can be omitted if 1 pixel wide a1: start angle of arc in degrees a2: end angle of arc in degrees c: Colour of arc (if omitted it will default to the foreground colour) Zero degrees is at the 12 o'clock position.
AUTOSAVE file\$ or AUTOSAVE N file\$	Enters automatic program entry mode: the lines of text sent to the console are saved to the file 'file\$' on the SD card or USB flash drive (".BAS" is added if it has no extension). This is one way of transferring a BASIC program from a computer: the program is pasted into a terminal emulator and this command captures the text. It can also be used for typing a small program at the console. The text is echoed back as it arrives, and a tab becomes a space. The mode is ended with a function key: F1 saves the file and, if it is a .BAS file, loads it into program memory, where it becomes the current program (used by RUN, LIST and EDIT without a file name) F2 does the same and then runs the program F6 saves the file without loading it At any time Control-C abandons the command and nothing is saved. The text is held in memory until it is saved, up to 512KB: a longer text stops the command with the error "Not enough memory". AUTOSAVE N does not echo the text. After the first character, when there is a pause of more than 100 milliseconds between characters it shows the message "Enter F1 ,F2 or F6 to exit" and echoes what follows. The command can only be used at the command prompt.

BEZIER x%(), y%() [,n] [,colour]	Draws a Bezier curve with up to 16 control points. 'x%()' and 'y%()' are single dimension integer arrays holding the coordinates of the control points; they must have the same number of elements. The curve starts at the first control point and ends at the last one used. The optional parameter 'n' defines how many control points to use for the plot (the first 'n' elements of the arrays). If omitted the size of the arrays determines the number. 'n' must be from 2 to the array size, and no more than 16. The optional parameter 'colour' defines the colour to plot the curve (defaults to the current foreground colour).
BIT(var%, bitno) = value	Sets a specific bit (0-63) in an integer variable. 'value' can be 0 or 1. See also the BIT function
BITBANG WS2812 type, pin, nbr, colours%() or BITBANG DHT22 pin, tvar, hvar [,DHT11] or BITBANG BITSTREAM pin, n, array()	The older names of the commands WS2812, HUMID and BITSTREAM, still accepted for compatibility with earlier versions of MMBasic for the Colour Maximate 2. See those commands. The PicoMite's command DEVICE, which replaced BITBANG there, is not available.
BITSTREAM pin, n, array()	This command is used to generate an extremely accurate bit sequence on the pin specified. The pin must have previously been set up as an output (SETPIN pin, DOUT, with or without OC) and set to the required starting level. 'n' is the number of transitions (1 to 250000) and 'array()' is a numeric array (integer or floating point) with at least 'n' elements. Notes: • The array contains the length of each level in the bitstream in microseconds (0 to 65535, so the maximum period is 65.5 ms). • The first transition will occur immediately on executing the command. • The last period in the array is ignored other than defining the time before control returns to the program or command line. • The pin is left in the starting state if the number of transitions is even and the opposite state if the number of transitions is odd. • Interrupts are disabled while the sequence is generated, so everything else (sound, the USB devices, the timers) waits until it is finished. The PicoMite's second pin, output modes and logic functions are not available. BITBANG BITSTREAM, the older name, is still accepted.
BLIT READ [#]b, x, y, w, h [,page] or BLIT WRITE [#]b, x, y [,orientation] or BLIT LOAD[BMP] [#]b, fname\$ [,x] [,y] [,w] [,h] 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. If the buffer is already in use and the new area has the same width and height the new area overwrites the stored one (a different size is an error). The optional 'page' specifies which page is to be read; the default is the current write page. Set 'page' to FRAMEBUFFER to read from the framebuffer (see the FRAMEBUFFER command). BLIT WRITE will copy the memory buffer '#b' to the current write page. The destination coordinate is 'x' and 'y' using the width and height of the buffer. The optional 'orientation' parameter (0 to 7) defaults to 0, every pixel copied,

	and specifies how the stored image data is changed as it is written out. It is the sum of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels BLIT LOAD, or BLIT LOADBMP, loads the BMP file 'fname\$' into the buffer '#b', as SPRITE LOADBMP does (see there for 'x', 'y', 'w' and 'h'). 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. They are the sprites: BLIT and SPRITE are the same command, so a buffer read with BLIT READ can be shown with SPRITE SHOW and every SPRITE command can also be written as BLIT (see SPRITE). • Two commands differ with the name used, as the PicoMite's separate BLIT and SPRITE commands do: BLIT WRITE copies every pixel unless its 'orientation' says otherwise, where SPRITE WRITE leaves out the transparent ones, and BLIT LOAD loads a BMP file, where SPRITE LOAD loads a sprite file. • When specifying the buffer number the # symbol is optional. • All other arguments are in pixels. • A buffer's transparent pixels are those that the colour depth shows as the transparent colour: black, unless SPRITE SET TRANSPARENT has chosen another (see SPRITE). 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 command. If 'page' is omitted the data is read from the write page. Set '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 area is clipped to the source and destination pages. The optional 'orientation' parameter (0 to 7, default 0) specifies how the section of the screen is changed as it is copied. It is the sum of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels: here those whose stored value is zero, black in the 4, 8 and 16-bit colour depths, and the colour 0 (black with a transparency of 0, which RGB(0,0,0) is not) in the 12 and 32-bit colour depths BLIT FRAMEBUFFER from, to, xin, yin, xout, yout, width, height [,colour] <u>IN THE 4-BIT COLOUR DEPTH ONLY</u> Copies an area of the framebuffer 'from' to the framebuffer 'to', as on the PicoMite. 'from' and 'to' are each N, L, F, 2 or T (a framebuffer that has not been created is N, the display: see the FRAMEBUFFER command) and must be different. 'xin' and 'yin' define the top left of the area of 'width' and 'height' on the source framebuffer to be copied, which must be within it. 'xout' and 'yout' define the top left of the area on the target framebuffer to receive the copy; the copy is clipped to the target.
--	--

BLIT FLASH n, to, xin, yin, xout, yout, width, height [,colour]	The optional parameter 'colour' defines a pixel colour on the source which will not be copied. If omitted (or -1) all pixels are copied. The colour is specified as a number between 0 and 15 representing: Black, Blue, Myrtle, Cobalt, Midgreen, Cerulean, Green, Cyan, Red, Magenta, Rust, Fuchsia, Brown, Lilac, Yellow and White Copies an area of the image held in image slot 'n' to a framebuffer or a page, as the PicoMite's BLIT FLASH does. 'n' is 1 to 8: the image slots are loaded with FLASH LOAD IMAGE (see Image Slots in the chapter Video Modes and Pages). 'to' is a framebuffer N, L, F, 2 or T, or a page number. In a 4-bit mode a framebuffer that has not been created is N (but 2 is then an error); in the other colour depths N, L, F, 2 and T are the pages 0, 1, 2, 3 and 4. 'xin' and 'yin' define the top left of the area of 'width' and 'height' on the image to be copied, which must be within the image. 'xout' and 'yout' define the top left of the area on the target to receive the copy; the copy is clipped to the target. The image's pixels are the PicoMite's 16 colours, drawn in the colour depth of the target. The optional parameter 'colour' defines a pixel colour in the image which will not be copied. If omitted (or -1) all pixels are copied. The colour is specified as a number between 0 and 15 representing: Black, Blue, Myrtle, Cobalt, Midgreen, Cerulean, Green, Cyan, Red, Magenta, Rust, Fuchsia, Brown, Lilac, Yellow and White
BLIT MEMORY address, x, y [,colour]	Draws an image held in memory as a packed array of colour nibbles on the current write page, with its top left at 'x', 'y'; the image is clipped to the screen. Each nibble is one of the PicoMite's 16 colours, a number between 0 and 15 representing: Black, Blue, Myrtle, Cobalt, Midgreen, Cerulean, Green, Cyan, Red, Magenta, Rust, Fuchsia, Brown, Lilac, Yellow and White The first word of the area of memory starting at 'address' must contain the width and height of the image as 16-bit integers with the width as the bottom 16 bits. The pixels follow, two to a byte with the left pixel in the low nibble, each row following on from the one before. If the optional parameter 'colour' is specified then pixels of that colour are not drawn (-1 draws them all). If the top bit of either the width or height is set to 1 then the colour data is treated as compressed (the remaining 15 bits are used as the width and/or height). The compression algorithm is simple: each byte contains a count in the bottom nibble (1-15) and a colour in the top nibble (0-15). In the event that more than 15 pixels are the same colour additional bytes are used for that colour.
BOX x, y, w, h [,lw] [,c] [,fill]	Draws a box on the screen 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 (0 to 100). 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' and 'y' must both be arrays or both be single variables/constants otherwise an error will be generated. 'w', 'h', 'lw', 'c' and 'fill' can be either arrays or single variables/constants (a single value is used for every box). See the chapter <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.

BOX AND_PIXELS x, y, w, h, colour [,page] or BOX OR_PIXELS x, y, w, h, colour [,page] or BOX XOR_PIXELS x, y, w, h, colour [,page]	Executes the requested logical operation (AND, OR or exclusive OR) between each pixel in the area with its top left at 'x', 'y', 'w' pixels wide and 'h' pixels high, and the colour 'colour' (in the pixel format of the colour depth), and writes the result to the same area of the current write page. The pixels are read from page 'page' (the write page if it is omitted; FRAMEBUFFER for the framebuffer). The area is clipped to the page. In the 12-bit colour depth the operation includes the pixels' transparency, which RGB() sets to 15 (solid): eg, XOR_PIXELS with RGB(255,255,255) also makes the solid pixels transparent, and with &HFFFFFF changes only their colour.
BYTE(var\$, byteno)=value	Sets a specific byte in a string to an integer value. 'value' can be in the range 0-255. The byteno can be between 1 and the current length of the string variable. This is the equivalent of MID\$(var\$,byteno,1)=CHR\$(value) but executes much faster. See also the BYTE function.
CALL usersubname\$ [,usersubparameters,..]	This is an efficient way of programmatically calling user defined subroutines (see also the CALL() function). In many cases 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.
CAN OPEN index, speed, mode or CAN OPEN index, 0, mode, prescaler, seg1, seg2, sjw or CAN OPEN index, 0, mode, prescaler, seg1, seg2, sjw, brs, dsize, dprescaler, dseg1, dseg2, dsjw	More detail is in Appendix J (CAN Support). A CAN transceiver is needed between the pins and the CAN bus (see the chapter <i>Special Device Support</i>). Opens the CAN interface 'index' and allocates its pins: interface 1 uses pin 5 (CAN Rx) and pin 3 (CAN Tx), shared with the first I²C channel, and interface 2 uses pin 19 (CAN Rx) and pin 40 (CAN Tx), shared with SPI MOSI and SPI2 CLOCK. Only one interface can be open at a time. 'speed' is the bit rate: 125000, 250000, 500000 or 1000000 bits per second, with the sample point at 80% and the Classical CAN frame type. With a 'speed' of 0 the bit timing is given instead, from the 40MHz clock of the CAN controller: 'prescaler', 'seg1', 'seg2' and 'sjw' (the synchronisation jump width) alone for the Classical CAN frame type, or with the further six values for the CAN FD frame type: 'brs' is 1 for bit rate switching or 0 for none, 'dsize' the size of the data field (8, 12, 16, 20, 24, 32, 48 or 64 bytes) and 'dprescaler', 'dseg1', 'dseg2' and 'dsjw' the timing of the data phase. Appendix J gives the values for the common speeds. 'mode' is 0 for normal operation, 1 for the internal loop back (nothing is sent to or received from the bus and the transceiver is not needed), 2 for the external loop back (as 1, but the messages are also sent to the bus) or 3 for monitoring the bus (messages are received but nothing is sent). CAN OPEN sets the global filter to accept every message into FIFO 1 (as CAN FILTER GLOBAL 1, 1, 1, 1). Other filters are set after CAN OPEN, then CAN START starts the interface.

CAN FILTER index, eid, type, config, id1, id2	Sets the filter ‘index’ (0 to 31) for 11-bit identifiers (‘eid’ 0) or for 29-bit identifiers (‘eid’ 1). The filters are tested in order of ‘index’; a message that matches none is dealt with by the global filter (below). ‘type’ is 0 for a range (the identifiers from ‘id1’ to ‘id2’), 1 for two identifiers (‘id1’ or ‘id2’), 2 for a classic filter (‘id1’ is the identifier and ‘id2’ the mask) or 3 for a range without the extended ID mask. ‘config’ is 0 to disable the filter, 1 to put a matching message in FIFO 0, 2 to put it in FIFO 1 or 3 to reject it. ‘id1’ and ‘id2’ are 0 to &H7FF for 11-bit identifiers and 0 to &H1FFFFFFF for 29-bit identifiers. A filter can be set or changed while the interface is running.
CAN FILTER GLOBAL id, eid, idr, eidr	Sets what happens to the messages that match no filter. ‘id’ is for those with 11-bit identifiers and ‘eid’ for those with 29-bit identifiers: 0 accepts them into FIFO 0, 1 into FIFO 1 and 2 rejects them. ‘idr’ and ‘eidr’ are the same for remote frames: 0 passes them to the filters, 1 rejects them. The global filter can only be set while the interface is stopped: after CAN OPEN and before CAN START, or after CAN STOP. CAN OPEN sets it to 1, 1, 1, 1.
CAN START or CAN STOP	CAN START starts the interface receiving messages and sending the messages queued by CAN SEND. It must be given after CAN OPEN (and any filters). CAN STOP stops it, for example to pause the reception of messages or to change the global filter. CAN START starts it again.
CAN SEND id, eid, rtr, dlc, msg, ret or CAN SEND id, eid, rtr, dlc, msg(), ret	Places a message in the next free transmit buffer and it is sent in order (there are 32 buffers). ‘id’ is the message’s identifier: 0 to &H7FF when ‘eid’ is 0 or 0 to &H1FFFFFFF (a 29-bit identifier) when ‘eid’ is 1. ‘rtr’ is 1 for a remote frame (no data), otherwise 0. ‘dlc’ is the number of data bytes: 0 to 8, or with CAN FD also 12, 16, 20, 24, 32, 48 or 64, and no more than the ‘dsize’ of CAN OPEN. ‘msg’ holds the data. For Classical CAN it is an integer variable, sent from its most significant byte (with ‘dlc’ 2, &HAABB000000000000 sends &HAA then &HBB). For CAN FD it is an integer array large enough for ‘dlc’ bytes (eg, INTEGER msg(7) for 64 bytes). ‘ret’ is an integer variable which is set to 0 if the message was queued or 1 if no transmit buffer was free.
CAN READ fifo, id, eid, rtr, dlc, msg, fmi, ret or CAN READ fifo, id, eid, rtr, dlc, msg(), fmi, ret	Reads the next message from FIFO 0 or FIFO 1 (‘fifo’ is 0 or 1; each holds up to 32 messages). The other arguments are integer variables (‘msg()’ an integer array with CAN FD) which are set from the message: ‘id’ its identifier, ‘eid’ 1 for a 29-bit identifier (otherwise 0), ‘rtr’ 1 for a remote frame, ‘dlc’ the number of data bytes, ‘msg’ the data (in the same order as CAN SEND, with the bytes beyond ‘dlc’ set to zero) and ‘fmi’ the number of the filter that accepted it, or 99 if it matched no filter and was accepted by the global filter. ‘ret’ is set to 0 if there was no message (the other variables are then not changed), otherwise to the number of messages that were waiting, including the one read. Only one message is read each time and nothing interrupts the program when one arrives: the program reads them by calling CAN READ regularly (eg, in its main loop).
CAN CLOSE	Closes the CAN interface and releases the pins used for CAN Rx and CAN Tx.

CAT S\$, N\$	Concatenates the strings by appending N\$ to S\$. This functionally the same as $S\$ = S\$ + N\$$ but operates somewhat faster.
CHAIN file\$ [, cmdline\$]	Runs another program, keeping the global variables (and constants) with their values. The program in 'file\$' is loaded and run as by RUN, but the global variables of the program that used CHAIN stay, so a program too large for program memory can be split into parts that pass their data on. 'file\$' and 'cmdline\$' are string expressions; ".BAS" is added to 'file\$' if it has no extension. 'cmdline\$' is passed to the new program as MM.CMDLINE\$. Everything else is cleared as by RUN: the local variables (so use CHAIN at the top level of the program, not in a subroutine or function), files, interrupts, I/O pins, sprites, the sound and the video mode. The library's code outside its subroutines and functions is not run again, and a RAM library (LIBRARY LOAD ..., RAM) does not survive the CHAIN. If the programs use structures both must define the same TYPEs in the same order, as a variable of a structure type kept across the CHAIN refers to its type by its number.
CHDIR dir\$	Change the current working directory to 'dir\$'. 'dir\$' can start with a drive (eg, "C:/games"); the directory is then on that drive, which becomes the current drive (see the DRIVE command). 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 centred at 'x' and 'y' with a radius of 'r' on the display output. '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 and 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 <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.
CLEAR	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
CLOSE [#]fnbr [, [#]fnbr] ... or CLOSE GPS	Close the file(s) or serial communications port(s) previously opened with the file number '#fnbr'. The # is optional. Before a serial port is closed the data waiting in its output buffer is sent. CLOSE GPS closes the serial port opened for a GPS receiver (OPEN ... AS GPS); GPS can also be one of a list of file numbers. Also see the OPEN command.

CLS [colour]	Clears the screen (the page being written to: see PAGE WRITE) and moves the text position for PRINT to the top left. Optionally 'colour' can be specified which will be used for the background colour when clearing the screen; otherwise the background colour set by COLOUR is used. When the write page is page 0 the screen of the terminal emulator on the serial console is cleared too. In the 4-bit colour depth CLS without a colour clears a layer made by FRAMEBUFFER LAYER or FRAMEBUFFER LAYER TOP, while it is the write target, to its transparent colour.
COLOUR fore [, back] or COLOR fore [, back]	Sets the default colour for commands (PRINT, etc) that display on the screen. 'fore' is the foreground colour, 'back' is the background colour. The background is optional and if not specified will default to any previously set background colour or black if not previously changed.
COLOUR MAP inarray%(), outarray%() [,colourmap%()]	This command generates RGB888 colours in outarray% from colour codes (0-15) in inarray%. If the optional colourmap% parameter is used this must be 16 elements long). In this case the values in inarray% are mapped to the colours for that index value in colourmap%
CONST id = expression [, id = expression] ... etc	Create a constant identifier which cannot be changed once created. 'id' is the identifier which follows the same rules as for variables. The identifier can have a type suffix (!, %, or \$) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.
CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with graphics, 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.
CONTROLLER CLASSIC OPEN [channel] [, interrupt [, bitmask]] or CONTROLLER CLASSIC CLOSE [channel]	Opens a Wii Classic controller for communications. These commands can also be written with WII in place of CONTROLLER (eg, WII CLASSIC OPEN). 'channel' is the I²C channel that the Classic is connected to: 1 or 2 on the rear I/O connector, or 3 (the default if it is not specified) for the Wii connector on the front panel. 'interrupt' is an optional MMBasic interrupt routine that will be triggered when a button is pressed. 'bitmask' specifies which of the 15 buttons on the Wii Classic will trigger an interrupt. The default is that they all will. See the CLASSIC() function for which bits in the bitmask control which buttons. The firmware checks for the existence of a controller on open and will throw an error if not found. Once open the Classic is polled continuously in the background every 20 ms. The I²C transfers are under interrupt control and non-blocking for the rest of the firmware. The I²C channel is opened at 100KHz: other I²C devices can share the channel with the Classic, in which case I2C OPEN should not be used (I2C WRITE, READ and CHECK work on

CONTROLLER MOUSE OPEN [channel] [, LBinterrupt] [, RBinterrupt] [, sensitivity] [, LBUppinterrupt] or CONTROLLER MOUSE CLOSE [channel]	the channel without it). A Nunchuk and a Classic cannot be open on the same channel. CONTROLLER CLASSIC CLOSE will close the Classic on 'channel' (3 if it is omitted). Also see the CLASSIC() function for accessing the state of the device. Opens a mouse connected through the Hobbytronic USB mouse interface (see the chapter <i>Special Device Support</i>) for communications. 'channel' is the I²C channel that the interface is connected to: 1, 2 (the default: pins 27 and 28 on the rear I/O connector) or 3. 'LBinterrupt' and 'RBinterrupt' are optional MMBasic interrupt routines that will be triggered when the left or the right mouse button is pressed. The interrupts trigger once on the button press and will not re-interrupt until it is released. 'LBUppinterrupt' is an optional interrupt routine which is called when the left button is released. The sensitivity of the Hobbytronic interface can be set by specifying a number between 0 and 10 for 'sensitivity'. 0 (the default) sets it from the width of the screen; other values explicitly set the sensitivity as per the Hobbytronic documentation. The interface is told the size of the screen (the current write page) when it is opened, so the mouse should be opened after the MODE command. The firmware checks for the existence of the interface on open and will throw an error if not found. This error can be trapped with the ON ERROR command. Once open the interface is polled continuously in the background every 20 ms. The I²C transfers are under interrupt control and non-blocking for the rest of the firmware. The I²C channel is opened at 100KHz: other I²C devices (such as a Wii controller) can share it with the mouse, in which case I2C OPEN should not be used. Only one mouse can be open at a time. While a USB mouse is plugged in it is the mouse, on channel 2: CONTROLLER MOUSE OPEN 2 then only sets the interrupts, which work for the USB mouse too, and the other channels give an error. CONTROLLER MOUSE CLOSE will close the mouse on 'channel' (2 if it is omitted). Also see the MOUSE() function for accessing the state of the device.
CONTROLLER MOUSE OPEN 0 [, LBinterrupt] [, RBinterrupt] [, sensitivity] [, LBUppinterrupt] or CONTROLLER MOUSE CLOSE 0	Opens a PS2 mouse for communications. On the Generation 1 Colour Maximite 2 the mouse clock line is connected to pin 33 and the mouse data to pin 32 of the rear I/O connector. Both lines MUST be pulled up to 5V with 4.7K resistors. On the Generation 2 additional pins are allocated to the PS2 mouse and it is connected to the lower USB port: the pullup resistors are already fitted. 'LBinterrupt', 'RBinterrupt' and 'LBUppinterrupt' are optional interrupt routines as for the mouse above. The PS2 mouse sensitivity can be set between 0 and 8. Values of 1-4 set the resolution of the mouse to 1, 2, 4 or 8 counts/mm. Values 5-8 do the same but in addition enable mouse scaling which gives a non-linear relationship between speed and count. See https://isdaman.com/alsos/hardware/mouse/ps2interface.htm for more details. NB: not all PS2 mice seem to implement these commands. The firmware checks for the existence of a mouse on open and will throw an error if not found. This error can be trapped with the ON ERROR command. Once open the mouse operates under interrupt control so it has no performance impact unless the mouse is moving.

CONTROLLER NUNCHUK OPEN [channel] [, Zinterrupt [, Cinterrupt]] or CONTROLLER NUNCHUK CLOSE [channel]	The PS2 mouse shares a hardware interrupt with pin 15, so it cannot be opened while pin 15 is a frequency, period or counting input, and pin 15 cannot be made one while the mouse is open. CONTROLLER MOUSE CLOSE 0 will close the PS2 mouse. Also see the MOUSE() function for accessing the state of the device. Opens a Wii Nunchuk for communications. ‘channel’ is the I²C channel that the Nunchuk is connected to: 1 or 2 on the rear I/O connector, or 3 (the default if it is not specified) for the Wii connector on the front panel. ‘Zinterrupt’ and ‘Cinterrupt’ are optional MMBasic interrupt routines that will be triggered when the Z or the C button is pressed. The interrupts trigger once on the button press and will not re-interrupt until it is released. The firmware checks for the existence of a controller on open and will throw an error if not found. This error can be trapped with the ON ERROR command. If a Classic controller is plugged in instead, an error will not be thrown: you can check the ID code with NUNCHUK(T, channel). Once open the Nunchuk is polled continuously in the background every 20 ms. The I²C transfers are under interrupt control and non-blocking for the rest of the firmware. The I²C channel is opened at 100KHz: other I²C devices can share the channel with the Nunchuk, in which case I2C OPEN should not be used (I2C WRITE, READ and CHECK work on the channel without it). CONTROLLER NUNCHUK CLOSE will close the Nunchuk on ‘channel’ (3 if it is omitted). Also see the NUNCHUK() function for accessing the state of the device.
COPY fname1\$ TO fname2\$ or COPY fname\$ TO dirname\$	Copy a file from ‘fname1\$’ to ‘fname2\$’. Both are strings. A file ‘fname2\$’ that exists already is overwritten. A directory path can be used in both ‘fname1\$’ and ‘fname2\$’, and either can include a drive (A: or B: for the SD card, C: for the USB flash drive), so a file can be copied from one drive to the other (see the DRIVE command). If ‘fname2\$’ ends in a / it is a directory and the file is copied into it with its own name. Wildcard copy. The bulk copy is triggered if ‘fname\$’ contains a ‘*’ or a ‘?’ character: all the files that match it (not directories, or hidden or system files) are copied into the directory ‘dirname\$’, each with its own name, and each name is shown as it is copied. The wildcards can only be in the file name, not in its path. A wildcard copy can only be used at the command prompt. The firmware will block copying a file to itself, and a wildcard copy into the directory the files are in ("Source and destination are the same").
CPU RESTART	Forces a restart of the processor. This clears all variables and resets everything (eg, timers, COM ports, I2C, etc) as at power up, but without the power up banner. If OPTION AUTORUN is set the program is run again. MM.INFO\$(RESET) then returns "Software".
CSUB name [type [, type] ...] hex [[hex[...] hex [[hex[...] END CSUB	Defines the binary code for an embedded machine code program module written in C or ARM assembler. The module will appear in MMBasic as the command 'name' and can be used in the same manner as a built-in command. Multiple embedded routines can be used in a program with each defining a different module with a different 'name'. The first 'hex' word is a 32 bit word which is the offset in 32-bit words from the word after it (the start of the compiled code) to the entry point of the embedded routine (usually the function main()). The following hex words are the compiled binary code for the module. These are automatically programmed into MMBasic when the program is loaded (by RUN). Each 'hex' must be exactly eight hex digits representing the bits in a 32-bit word and be separated

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.
DATE\$ = "DD-MM-YY[YY]" or DATE\$ = "DD/MM/YY[YY]"	Set the date of the internal clock/calendar. DD, MM and YY are numbers, for example: DATE\$ = "28-7-2024". The year can be abbreviated to two digits (ie, 24), and the real time clock holds the years 2000 to 2099. The date is kept by the battery backed real time clock while the power is off (see Backup Battery in the chapter Hardware Features), and with OPTION DS3231 ON it is set in the DS3231 real time clock too. If the clock holds a date before 2018 when the power is applied MMBasic takes it to have stopped: it sets the date and time to 01-01-2000 and 00:00:00 and asks for them (see Appendix K). So the time (set with TIMES\$ =) is lost when the power is cycled unless a correct date has also been set.
DEFINEFONT #Nbr hex [[hex[...] hex [[hex[...] END DEFINEFONT	This will define an embedded font which can be used alongside or to replace the built in fonts. These work exactly same as the built in fonts (i.e. selected using the FONT command or specified in the TEXT command). See Embedded Fonts in the chapter <i>Graphics Functions</i>, and the <i>Embedded Fonts</i> folder in the Colour Maximite 2 firmware download for a selection of embedded fonts. '#Nbr' is the font's reference number: 2 to 5 or 8 to 16. It can be the same as a built in font and in that case it will replace the built in font (fonts 1, 6 and 7 cannot be replaced). 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. • Multiple embedded fonts can be used in a program with each defining a different font with a different font number. • During execution MMBasic will skip over any DEFINEFONT commands so they can be placed anywhere in the program. • Any errors in the data format will be reported when the program is loaded (by RUN).
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>	Declares one or more variables (i.e. 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 (i.e. string, float or integer) can be specified in one of three ways: By using a type suffix (i.e. !, % or \$ for float, integer or string). For example: <pre>DIM nbr%, amount!, name\$</pre> 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 (i.e. it does not have to be repeated). For example: <pre>DIM STRING first_name, last_name, city</pre> By using the Microsoft convention of using the keyword "AS" and the type keyword (i.e. FLOAT, INTEGER or STRING) after each variable. If you use

For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used. Examples: <pre> DIM nbr(50) DIM INTEGER nbr(50) DIM name AS STRING DIM a, b\$, nbr(100), strn\$(20) DIM a(5,5,5), b(1000) DIM strn\$(200) LENGTH 20 DIM STRING strn(200) LENGTH 20 DIM a = 1234, b = 345 DIM STRING strn = "text" DIM x%(3) = (11, 22, 33, 44) </pre>	this method the type must be specified for each variable and can be changed from variable to variable. For example: <pre> DIM amount AS FLOAT, name AS STRING </pre> Floating point or integer variables will be set to zero when created and strings will be set to an empty string (i.e. ""). You can initialise the value of the variable by using an equals symbol (=) and an expression following the variable definition. For example: <pre> DIM STRING city = "Perth", house = "Brick" </pre> The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter <i>Variables and Expressions</i> for more examples of the syntax. As well as declaring simple variables the DIM command will also declare arrayed variables (i.e. an indexed variable with a number of dimensions). Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example: <pre> DIM array(10, 20) </pre> Each number specifies the index range in each dimension. Normally the indexing of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1. The above example specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each floating point number requires 8 bytes a total of 1848 bytes of memory will be allocated. A dimension can hold a single element. With the default OPTION BASE 0 this is written as DIM a(0) and with OPTION BASE 1 as DIM a(1). This matters when the size of an array is calculated while the program is running and the calculation can produce one element, and it allows the MATH matrix commands to produce a single row, a single column, or a 1x1 result. Strings will default to allocating 255 bytes (i.e. 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: <pre> DIM STRING s(5, 10) LENGTH 20 </pre> 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 (see below). In the above example you can also use the Microsoft syntax of specifying the type <u>after</u> the length qualifier. For example: <pre> DIM s(5, 10) LENGTH 20 AS STRING </pre> With a simple (non arrayed) string the LENGTH keyword also sets the maximum length of the string, and if the length is 15 or less the string is held in the variable's own record, saving the 256 bytes of memory that a string variable
--	--

	otherwise takes. This is useful for short strings. A string constant (CONST) of up to 15 characters is held in the same way. 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: <pre>DIM INTEGER nbr(4) = (22, 44, 55, 66, 88) or DIM s\$(3) = ("foo", "boo", "doo", "zoo")</pre> 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). If, at the start, the expression is false the statements in the loop will not be executed, not even once.
DO UNTIL expression <statements> LOOP	Loops until 'expression' is true. If, at the start, the expression is true 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.
DO <statements> LOOP WHILE expression	Loops until the expression following WHILE is false. 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 false.
DRAW3D SHOW n, x, y, z [,nocull] [,usefurthest] or DRAW3D WRITE n, x, y, z [,nocull] [,usefurthest]	The 3D engine draws objects made of flat faces as a camera sees them, with the hidden faces removed. It has the sub-commands CREATE, CAMERA, LIGHT, ROTATE, SHOW, WRITE, HIDE, HIDE ALL, RESTORE, RESET, SET FLAGS, DIAGNOSE, CLOSE and CLOSE ALL, which are described with the DRAW3D() function in the document <i>The 3D engine.pdf</i> in the Colour Maximite 2 firmware download (see Appendix O). Up to 128 objects and 6 cameras can be used. DRAW3D LIGHT n, x, y, z, ambient sets the position of the light source for object 'n' and its ambient light level, 0 to 100 (percent). DRAW3D SET FLAGS n, flag, face, count [, face, count ...] sets the flags (0 to 255) of 'count' faces of object 'n', from face number 'face'. DRAW3D DIAGNOSE n, x, y, z [, sort] prints each face of object 'n' as it would be drawn at 'x', 'y', 'z': its distance, its dot product and whether it is hidden or showing (sorted by distance unless 'sort' is 0). DRAW3D RESET n [, n ...] makes the current orientation of each object its starting point: later ROTATE commands turn it from there. DRAW3D SHOW draws the 3D object 'n' at the position 'x', 'y', 'z', first erasing (to colour 0) the area where it was last drawn; DRAW3D WRITE draws it without erasing anything. Rendering of the 3D object works by the painter's algorithm: the faces are sorted and those further away are drawn first. In addition MMBasic supports back face culling: faces that point away from the camera are not drawn. Set 'nocull' to 1 to draw them as well (0, the default, culls them).

	To sort the faces MMBasic uses the position of each face's centroid. This works in many situations but in some 3D models it is better to use the furthest vertex of each face: set 'usefurthest' to 1 for that (0, the default, uses the centroid).
DRIVE drive\$	Sets the active disk drive as 'drive\$'. 'drive\$' can be "A:" or "B:", the SD card (B: is another name for A:), or "C:", a USB flash drive - see the heading USB Flash Drive in the chapter Keyboard, Mouse and Gamepad. Each drive keeps its own current directory, so DRIVE returns to the directory that was current on the drive when it was last used. A drive that is not there gives the error "SD card not present" or "USB drive not present". Typing A:, B: or C: at the command prompt does the same as this command.
EDIT [file\$]	Invokes the full screen editor (see the chapter Full Screen Editor). It can only be used at the command prompt. EDIT 'file\$' edits the file 'file\$' on the SD card or USB flash drive: a program or any other text file, up to 1MB long. If 'file\$' has no extension and no file of that name exists, ".BAS" is added. A file that does not exist is created when the editor saves it. EDIT on its own edits the current program's file (see MM.INFO\$(CURRENT)), with the cursor where it was when the editor last saved it or, if the program stopped with an error, at the line that caused the error. F1 saves the file and returns to the command prompt and F2 saves it and runs it; Esc leaves without saving. A .BAS file that is saved is also loaded into program memory, so that it becomes the current program. The file as it was before is kept with .bak added to its name. Only BASIC programs (.BAS files) are colour coded (see OPTION COLOURCODE).
ELSE	Introduces an optional default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF expression THEN or ELSE IF expression THEN	Introduces an optional secondary condition in a multiline IF statement. See the multiline IF statement for more details.
END [noend] or END cmd\$	End the running program and return to the command prompt. If a subroutine named MM.END exists in the program it will be executed when the program ends with an END command or by running past its last line. It is not executed if the program ends with an error or with the break character (ie, Ctrl-C). The optional parameter 'noend' can be used to block execution of the MM.END subroutine eg, "END noend" if 'cmd\$' is specified then it will be executed as though at the command prompt after the program finishes. Note: if "END cmd\$" is used but a subroutine MM.END exists it will be executed and cmd\$ ignored.
END CSUB	Marks the end of a C subroutine. See the CSUB command. Each CSUB must have one and only one matching END CSUB statement.
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.
END TYPE	Marks the end of a user defined structure. See the TYPE command..
ERASE variable [,variable]..	Deletes global 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. 'error_msg\$' is optional and is the message to display on the console.
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 of one second before executing the requested command. If control does not return to EXECUTE within that time, as with a GOTO or a command that takes longer (eg, PAUSE 2000), the processor restarts with the message "Command timeout". You cannot call EXECUTE from within code that has been EXECUTE'd. 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 [fspec\$] [,sort]	Lists the files and directories in the current directory of the current drive (see DRIVE), or in the directory given in 'fspec\$'. FILES can only be used at the command prompt. 'fspec\$' (if specified) can contain a path and search wildcards in the filename. 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 mydir/* Find all entries in directory mydir C:/*.BAS Find all entries with an extension of BAS in the root directory of the USB flash drive NB: putting wildcards in the pathname will result in an error 'sort' specifies the sort order as follows:

	size by ascending size time by descending time/date name by file name (default if not specified) type by file extension Directories are listed before files, which are shown with their date, time and size, followed by the number of each; hidden and system files are not listed. The listing pauses each time the screen is full, and up to 2048 entries can be listed. LIST FILES does the same, and FM is a full screen alternative (see the chapter <i>File Manager</i>).
FILL x, y, fillcolour [,bordercolour]	Implements a flood fill on the current write page, starting at 'x', 'y' (which must be on the screen). When used without the optional 'bordercolour' the command will read the colour at position 'x', 'y' and then fill the area from that point where the current colour is the same with the new colour 'fillcolour'. If the optional 'bordercolour' is specified, the 'fillcolour' will replace whatever colour is already there until it hits the 'bordercolour' specified. See also PIXEL FILL.
FLAG(n%)=value	Sets a bit in a system flag register. N% can be between 0 and 63 (i.e. 64 flag bits are available). Value can be 0 or 1. See also the FLAGS command and FLAG function and MM.FLAGS
FLAGS=value%	Sets all bits in the system flag register to the value specified. See also the FLAG command, the FLAG() function and MM.FLAGS
FLASH SAVE n or FLASH OVERWRITE n	Saves the program in program memory (the one RUN last loaded) into flash slot 'n' (1 to 3), as the PicoMite does, for FLASH RUN, FLASH CHAIN and FLASH LOAD. FLASH SAVE gives the error "Already programmed" if the slot holds anything; FLASH OVERWRITE replaces what it holds. They can only be used at the command prompt. The Colour Maximate 2 keeps its flash slots on the SD card, as the files slot1 to slot3 in the hidden directory A:/.flash, and reads a slot into RAM when it is first used after a RUN or NEW. No flash memory is written, and the slots are kept when the power is off, as the PicoMite's are. A slot holds the program as it was tokenised, not its source text, so a version of the firmware that tokenises programs differently will not run it: FLASH LIST then says that the slot was saved by another firmware version, and the program must be loaded from its file and saved again.
FLASH RUN n	Runs the program in flash slot 'n', as RUN runs a program. The program runs where it is, in the slot's copy in RAM: it is not loaded into program memory or tokenised again. A slot of 0 runs the program in program memory. MM.INFO(FLASH) returns the slot whose program is running, or 0. In a slot's program, RUN with no file name runs that program again. When the program ends, or stops with an error or Ctrl-C, the program in program memory is the current program again (for LIST, RUN and EDIT), and CONTINUE cannot resume the slot's program.
FLASH CHAIN n	Runs the program in flash slot 'n' as FLASH RUN does, but the global variables are kept, as with CHAIN. A slot of 0 runs the program in program memory. It can only be used in a program.

FLASH LOAD n	Loads the program in flash slot 'n' into program memory, replacing the program there. RUN then runs it as it is (not its file, which RUN otherwise loads afresh) until another program is loaded, NEW is used or the Colour Maximite 2 restarts. It can only be used at the command prompt.
FLASH LIST [n [, ALL]]	FLASH LIST reports what each flash slot holds: "available", the file name of a program, or the size of an image. FLASH LIST n lists the program in slot 'n' as it is held (as LIST FLASH lists program memory), pausing at each screenful unless ALL is given.
FLASH ERASE n or FLASH ERASE ALL	Empties flash slot 'n', or all three. FLASH ERASE ALL can only be used at the command prompt.
FLASH LOAD IMAGE n, file\$ [, O] or FLASH LOAD IMAGE n, file\$ [, OVERWRITE]	Loads the BMP file 'file\$' (".BMP" is added if it has no extension) into slot 'n' in the PicoMite's format, for the PicoMite's commands that use image slots (BLIT FLASH and TILEMAP). The slot holds an 8 byte header, the image's width and height as two 32-bit words, followed by the pixels in the RGB121 format, two to a byte with the left pixel in the low four bits; MM.INFO(FLASH ADDRESS n) gives its address. 'n' is 1 to 8. Slots 1 to 3 are the flash slots, kept on the SD card (see FLASH SAVE). Slots 4 to 8 are the PicoMite's RAM slots 1 to 5, held in RAM until RAM ERASE, an overwrite or a restart (see RAM SAVE), so a program that loads its images every time it starts gives O. A slot that already holds something gives the error "Already programmed" unless O (or OVERWRITE) is given. At the command prompt the command reports the size of the image.
FLASH DISK LOAD n, file\$ [, O] or FLASH DISK LOAD n, file\$ [, OVERWRITE]	Loads the bytes of the file 'file\$', as they are, into slot 'n' (1 to 8), for a program to read with PEEK or MEMORY COPY at MM.INFO(FLASH ADDRESS n) (eg, a PicoMite program's sprite data). A slot that already holds something gives the error "Already programmed" unless O (or OVERWRITE) is given.
FM	Opens the full screen, two panel file manager for the files on the SD card (A:, also called B:) and a USB flash drive (C:). Files can be browsed, sorted, filtered, copied, moved, renamed and deleted; files can be listed and edited and BASIC programs run and created; audio files can be played, singly or as a track list of marked files, and image files displayed. Press Esc to exit and F1 for a summary of the keys. It can only be used at the command prompt, where the F1 key also starts it. See the chapter File Manager for the details.
FLUSH [#]fnbr	Causes any buffered writes to a file previously opened with the file number 'fnbr' to be written to disk. The # is optional. Using this command ensures that no data is lost if there is a power cut after a write command.
FONT [#]font-number [, scaling]	This will set the default font for displaying text on the screen. Fonts are specified as a number. For example, #2 (the # is optional). See the chapter <i>Graphics Functions</i> for details of the available fonts. A number that is not a built in font or one loaded by DEFINEFONT or LOAD FONT gives the error "Invalid font number". 'scaling' can range from 1 to 15 (the default is 1) 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.

FRAMEBUFFER WINDOW x, y, page [, I or B]	Copies an area of the framebuffer with its top left corner at 'x', 'y' to the page specified. The area is the size of the page (normally MM.HRES by MM.VRES) and must lie within the framebuffer. The optional parameter specifies if the copy is done immediately (I, the default) or during the next frame blanking (B). Without a framebuffer the error is "World not created".
FRAMEBUFFER CLOSE	Releases the memory used by the framebuffer and its backup, allowing a new framebuffer to be created with a different size. The framebuffer cannot be closed while it is the write page ("Framebuffer is set for write"): use PAGE WRITE first.
FRAMEBUFFER CREATE	<u>IN THE 4-BIT COLOUR DEPTH ONLY</u> In the 4-bit colour depth (MODE r, 4) the FRAMEBUFFER command is the PicoMite's, so that programs written for the PicoMite's VGA and HDMI versions work as they do there (see the chapter Video Modes and Pages). Its framebuffers are pages 0 to 4: N (the display, page 0), L (the layer, page 1), F (the framebuffer, page 2), 2 (the second framebuffer, page 3) and T (the top layer, page 4). They are held in the video memory, not in the variable memory, and MODE closes them. Creating a buffer that exists already is an error. BLIT FRAMEBUFFER, BLIT FLASH and TILEMAP use the same letters. A colour given as a number from 0 to 15 is one of the 16 colours: 0 black, 1 blue, 2 myrtle, 3 cobalt, 4 midgreen, 5 cerulean, 6 green, 7 cyan, 8 red, 9 magenta, 10 rust, 11 fuchsia, 12 brown, 13 lilac, 14 yellow and 15 white (unless changed by MAP). FRAMEBUFFER CREATE creates the framebuffer F, cleared to black, with the resolution of the current mode. It takes no size (the Colour Maximite 2's FRAMEBUFFER CREATE w, h gives the error "Unexpected text" in this depth).
FRAMEBUFFER CREATE 2	Creates the second framebuffer "2", cleared to black, with the resolution of the current mode. It is intended for use with FRAMEBUFFER MERGE as the alternative to a live layer.
FRAMEBUFFER LAYER [colour]	Creates the layer buffer L with the resolution of the current mode. The layer is shown over the display N automatically: at the end of each frame the firmware puts the two onto the screen, the pixels of the layer that are of the transparent colour 'colour' (0 to 15, default 0) letting N show through. The layer is cleared to that colour, as CLS without a colour does while the layer is the write target.
FRAMEBUFFER LAYER TOP [colour]	Creates the top layer T with the resolution of the current mode. It is shown over the layer L (where that has been created) and N in the same way, its pixels of the transparent colour 'colour' (0 to 15, default 0) letting what is under them show through. It is cleared to that colour.
FRAMEBUFFER WRITE where or FRAMEBUFFER WRITE where\$	Specifies the target for subsequent graphics commands. 'where' can be N, F, 2, T or L, where N is the display, or a string expression giving one of them. The buffer must have been created ("Frame buffer not created", "Layer not created" and the like). MM.HRES and MM.VRES report its size. PAGE WRITE selects any 4-bit page, including the pages after page 4.
FRAMEBUFFER CLOSE [which]	Closes a framebuffer. The optional parameter 'which' can be F, L, T or 2. If it is omitted all four are closed and the drawing returns to N. A buffer that was the write target when it was closed is replaced by N.

FRAMEBUFFER COPY from, to [, B] FRAMEBUFFER WAIT FRAMEBUFFER MERGE [colour] [, B]	Does a highly optimised full screen copy of one framebuffer to another. 'from' and 'to' can be N, F, 2, T or L. A buffer that has not been created stands for N, as on the PicoMite; the two must differ ("Buffer not created"). If the optional parameter B is specified the copy waits until the next frame blanking starts. Pauses processing until the next frame blanking starts. Copies framebuffer "2" over framebuffer F into the display N in a single pass. Pixels in "2" that are of the transparent colour 'colour' (0 to 15, default 0) show the F pixel instead. Both must have been created. The layers L and T are not involved and continue to be applied over the result. If the optional parameter B is specified the merge waits until the next frame blanking starts before it begins, so that tearing is avoided. This is the alternative to a live layer for programs that draw sprites over a background: the display only changes when MERGE is commanded.
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 (i.e. xxx\$) or by specifying the type using AS <type> at the end of the functions definition. For example: <pre>FUNCTION xxx (arg1, arg2) AS STRING</pre> '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. i.e. arg3(). The type of the argument can be specified by using a type suffix (i.e. arg1\$) or by specifying the type using AS <type> (i.e. 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 set the return value of the function you assign the value to the function's name. For example: <pre>FUNCTION SQUARE (a) SQUARE = a * a END FUNCTION</pre> 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: <pre>PRINT SQUARE (56.8)</pre> 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. The argument can be prefixed with BYVAL which will prevent this mechanism and cause only the value to be used. Alternatively, the prefix BYREF instructs

	MMBasic that a reference is required and an error will be generated if that cannot be done. 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. Doing so will have undefined side effects including ruining your day. A function can also return a whole array by declaring the array dimensions with the type at the end of the definition (INTEGER and FLOAT arrays only). For example: <pre>FUNCTION fred (arg1, arg2) AS INTEGER(6)</pre> Inside the function the result is set by assigning to elements of the function's name (eg, fred(i) = 44). Note that reading an element of the function's name inside the function will call the function recursively - if intermediate values must be read back, build the result in a LOCAL array and copy it with fred() = tmp%(). Such a function can only be used as the source of a whole-array assignment and the destination array must have the same type and total number of elements. For example: <pre>DIM b%(6) : b%() = fred(1, 2)</pre> It cannot be used anywhere else in an expression, so something like PRINT fred(1, 2) will generate an error. The dimensions can be expressions and are evaluated each time the function is called, after its arguments have been set up, so they can depend on the arguments. The result array can also be filled by passing it to any command that takes an array (eg, the MATH commands) rather than by assigning to its elements. For example, this returns the inverse of whatever size matrix it is given: <pre>FUNCTION Invr(N()) AS FLOAT(BOUND(N(), 1), BOUND(N(), 2)) MATH M_INVERSE N(), Invr() END FUNCTION DIM a(3,3), b(3,3) : b() = Invr(a())</pre>
GAMEPAD COLOUR channel, colour	Changes the colour of the display panel on a PS4 controller on USB channel 'channel'. 'colour' is set as a standard RGB888 value e.g. RGB(RED)
GAMEPAD HAPTIC channel, left, right	Causes the left and right vibration motors to operate on a PS4 controller or an Xbox 360, Xbox One or Xbox Series controller on USB channel 'channel'. 'left' and 'right' must be a number between 0 (off) and 255 (maximum). Any other gamepad gives the error "PS4 or Xbox controller only".
GAMEPAD INTERRUPT ENABLE channel, int [,mask]	Enables interrupts on the button presses on a USB game controller. The optional parameter 'mask' defines which of the switches will trigger the interrupt (defaults to all). 'mask' is a bitmap corresponding to the output of the DEVICE(GAMEPAD channel,B) function.
GAMEPAD INTERRUPT DISABLE channel	Disables interrupts from the gamepad on the channel specified
GAMEPAD MONITOR or GAMEPAD MONITOR SILENT GAMEPAD MONITOR OFF	Use GAMEPAD MONITOR before plugging in a gamepad and when plugged in it will show the before and after report with each change of buttons. GAMEPAD MONITOR SILENT turns the same monitoring on but suppresses the printed report. While the monitoring is on a gamepad that MMBasic does not otherwise recognise is accepted as a generic gamepad, and every generic gamepad (one that MMBasic or GAMEPAD CONFIGURE describes, too) is only monitored, not decoded; the PS4, PS3, Xbox and Switch Pro controllers

GAMEPAD CONFIGURE vid,pid,i0,c0,i1,c1,i2,c2,i3,c3,i4,c4,i5,c5,i6,c6,i7,c7,i8,c8,i9,c9,i10,c10,i11,c11,i12,c12,i13,c13,i14,c14,i15,c15	are not affected. GAMEPAD MONITOR OFF turns the monitoring off; running a program (and NEW) turns it off too. Use to configure a gamepad that isn't supported by the firmware. Run the command before plugging in the gamepad. All 34 parameters are mandatory. In each case the i/c parameters define the index into the report and the bit number at that index for the data that corresponds to the relevant bit. See DEVICE(GAMEPAD n,B) for more information on bit usage (0-15). The code is a bit number 0-7 (the button is pressed when that bit is set), 128-135 (pressed when bit code-128 is clear), 64 (pressed when the byte is below 64) or 192 (pressed when it is above 192). An index of 255 marks a button the gamepad does not have. Any other code is an error
GOSUB target	Initiates a call to the subroutine at 'target', which can be a line number or a label. The subroutine must be terminated with RETURN. GOSUBs, subroutines and functions can be nested up to 100 deep in total. GOSUB is a legacy command; defined subroutines (SUB...END SUB) are recommended instead for new programs.
GOTO target	Branches program execution to the target, which can be a line number or a label.
GUI	The GUI commands create the GUI controls and manage them: GUI BUTTON, SWITCH, CHECKBOX, RADIO, LED, FRAME, NUMBERBOX, TEXTBOX, FORMATBOX, SPINBOX, DISPLAYBOX, CAPTION, GAUGE, BARGAUGE, AREA, LISTBOX and SLIDER create a control and GUI PAGE, SETUP, DELETE, DISABLE, ENABLE, HIDE, SHOW, RESTORE, REDRAW, FCOLOUR, BCOLOUR and INTERRUPT manage them. GUI CLICK clicks them from a program or a pin. See the chapter <i>GUI Controls</i>; the commands are described in full in the document <i>GUI Controls and Programming.pdf</i> (see Appendix O), and GUI LISTBOX, SLIDER and CLICK below. The controls need memory reserved for them with OPTION MAXCTRLS (otherwise the error is "No memory allocated for GUI controls"). They are clicked with the left button of a USB mouse, which is used as soon as it is plugged in, or of a PS2 or I²C mouse set with OPTION MOUSE. The first GUI command opens that mouse and displays the cursor, which then follows it; a USB mouse plugged in after that is used at once, and the cursor is then displayed. In the 4-bit colour depth there is no cursor (see GUI CURSOR) and the controls are used without one. The controls are those of the Micromite Plus with the mouse in place of its touch screen: its TOUCH() function is CLICK() here (TOUCH() is accepted as another name, as on the PicoMite), and GUI BEEP is not provided (use PLAY TONE in the interrupt routine if a click is wanted).
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 can be 1 to 15 (the default is 1). 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. 'bc' can be -1 and then the background is not drawn (the pixels of the zero bits are left as they are).

	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. An integer holds 64 bits (its least significant byte is the first) and a string up to 255 bytes: 'width' x 'height' must not be more bits than it holds ("Not enough data"). See the chapter <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.
GUI CLICK [x, y] or GUI CLICK DOWN or GUI CLICK UP or GUI CLICK PIN pin [, INV] or GUI CLICK PIN OFF	Clicks the GUI controls without a mouse, as on the PicoMite: from a program (a joystick read by it, say, or a test), or with a button on a pin. The clicks act as the left mouse button, for the controls and for CLICK(DOWN), CLICK(X) and the others. GUI CLICK presses and releases the button where the pointer is: the cursor's active point, or without the cursor the mouse, or without a mouse the last 'x', 'y' given. With 'x' and 'y' it moves the pointer (and the mouse) there first. It returns when the controls have taken the press and the release (in 130 mS to one second), and a GUI interrupt routine that the click calls has run by then. GUI CLICK DOWN presses the button where the pointer is and holds it until GUI CLICK UP: a slider then follows the pointer (moved by the mouse, or by MOUSE SET), and a spin box repeats. GUI CLICK PIN makes a button on 'pin' click where the pointer is: pressed when the pin is low, the pin set as a digital input with a pull-up, or with INV pressed when it is high, with a pull-down. The pin must not be in use. GUI CLICK PIN OFF releases it. GUI CLICK, DOWN and UP cannot be used in an interrupt routine.
GUI CURSOR ON [n [,x, y [,colour]]] GUI CURSOR x, y GUI CURSOR OFF GUI CURSOR HIDE GUI CURSOR SHOW GUI CURSOR COLOUR colour GUI CURSOR LOAD fname\$	Displays and manipulates a cursor (eg, a mouse pointer) on the screen. The cursor sits above all other graphics and nothing can overwrite it. It is drawn on page 0 (page 1 in the 12-bit colour depth) and it is not available in the 4-bit colour depth ("Invalid for this screen mode"). GUI CURSOR ON enables and displays the cursor. 'n' is the cursor number: 0 is a mouse type pointer (the default), 1 is a cross hair and 2 is a user defined cursor previously loaded with GUI CURSOR LOAD. 'x' and 'y' specify the initial position of the cursor's active point (the top left corner of the screen if omitted) and 'colour' can be used to override the default white colour of cursors 0 and 1. If the cursor is already on the command does nothing. GUI CURSOR x, y moves the cursor to 'x', 'y'. If the cursor is hidden it is not displayed, just its stored location is updated. GUI CURSOR OFF turns off the cursor. GUI CURSOR HIDE hides the cursor but maintains its position and GUI CURSOR SHOW will show a hidden cursor again at its stored position. GUI CURSOR COLOUR changes the colour of cursor numbers 0 and 1. It does not affect a loaded cursor which has its colours specified by the cursor designer. GUI CURSOR LOAD loads cursor number 2 from the file 'fname\$' (".CUR" is added if it has no extension), and GUI CURSOR ON 2 then displays it. The cursor must be off when it is loaded ("Cursor On"). The file uses the same layout and colour coding as a sprite file (see SPRITE LOAD) except that the header line is: width, height, xoffset, yoffset. The two offsets define the position of the cursor's active point within the image - the mouse pointer (cursor 0) has offsets 0,0 (the tip of the arrow) and the cross hair (cursor 1) has 7,7 (its centre). The cursor is held in the pixel format of the colour depth in use when it is loaded, so it should be loaded after the MODE command.

GUI CURSOR LINK MOUSE GUI CURSOR UNLINK MOUSE	GUI CURSOR LINK MOUSE links the cursor to the mouse so that it automatically tracks the mouse movement. GUI CURSOR UNLINK MOUSE breaks the link. A USB mouse must be connected, or a PS2 or I²C mouse opened with CONTROLLER MOUSE OPEN, before the cursor can be linked (otherwise the error is "Not open"). The cursor cannot be linked while the GUI controls are using the mouse ("Mouse in use for GUI"). The cursor's active point (see GUI CURSOR LOAD) follows the mouse, and GUI CURSOR OFF also breaks the link.
GUI LISTBOX [#]ref, array\$(), x, y, w, h [,fc] [,bc] [,maxrows] or GUI LISTBOX CANCEL	Creates a list box, as on the PicoMite: a box 'w' by 'h' pixels with its top left at 'x', 'y', showing the selected item of the string array 'array\$()' and a marker pointing down. 'fc' and 'bc' are its foreground and background colours (by default the last ones used). Clicking it opens a list of the items below it (or above it, if there is more room there), each row as high as the box; while the list is open the other controls are greyed and do not respond. Clicking an item selects it and closes the list, and clicking outside the list closes it with the selection unchanged. If there are more items than fit on the screen, or than 'maxrows' (1 to 100), arrows above and below the rows scroll the list, and repeat while they are held. GUI LISTBOX CANCEL closes an open list, with the selection unchanged. CTRLVAL(#ref) returns the index of the item selected, 0 for the array's first element (so with OPTION BASE 1 the item is array\$(CTRLVAL(#ref) + 1)); it is 0 when the control is created. CTRLVAL(#ref) = n selects item 'n' (kept within the items). The GUI up interrupt is called when an item is selected. The control reads the array's items each time it is drawn, so an item changed and the control drawn again (by CTRLVAL, or GUI REDRAW) shows the change. The array must therefore last as long as the control: it must be global or STATIC ("Must be a global variable" for a LOCAL one) and one-dimensional, and the number of items is its size when the control is created.
GUI SLIDER [#]ref, x, y, w, h [,fc] [,bc] [,min] [,max] [,inc]	Creates a slider, as on the PicoMite: a thumb on a track, 'w' by 'h' pixels with its top left at 'x', 'y', for a value from 'min' to 'max' (by default 0 to 100). It is horizontal if it is wider than it is tall, with 'min' at the left, and vertical otherwise, with 'min' at the bottom. 'fc' is the colour of the thumb and of the track up to it, and 'bc' is the background colour (by default the last ones used). Clicking the track moves the thumb there, and while the button is held the thumb follows the mouse, past the ends of the track too. If 'inc' is more than 0 (the default is 0) the value is a whole number of steps of 'inc' from 'min'. CTRLVAL(#ref) returns the value, 'min' when the control is created, and CTRLVAL(#ref) = v sets it (kept within 'min' and 'max'). The GUI down interrupt is called every 50 mS while the thumb is being dragged, and the up interrupt when the button is released.
HELP [searchtext]	Shows the syntax of a command or function. It can only be used at the command prompt (in the editor F11 shows the help for the keyword at the cursor). 'searchtext' need not be quoted. If the SD card has the file help.txt in its root directory (A:/help.txt) HELP looks up 'searchtext' in that file. This can be a user written or community developed file and must be in a particular format. For each help entry the first line must be a search string prefixed with a ~ character. This is used for the help functionality to find an entry and is not displayed. The 'searchtext' can contain ? for single character substitution or * for multiple character (or none) substitution. Following, the search string the next line would typically give the syntax of a particular command or function. Any subsequent lines would be further explanation.

	e.g. <code>~COLOR</code> <code>COLOR fore [, back]</code> <i>Sets the default color for commands (PRINT, etc) that display on the screen.</i> <i>'fore' is the foreground colour, 'back' is the background colour.</i> <i>The background is optional and if not specified will default to black.</i> The command will return all entries that match the 'searchtext' and these will be paged to match the console device. Various versions of help.txt are available on https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=17865 Without A:/help.txt the help built into MMBasic is used. HELP then shows up to three lines of it that start with the text typed so far ('searchtext' and any characters typed after it, which can be deleted with backspace), and the up and down arrow keys move through the matches. Esc or F12 ends the command.
HUMID pin, tvar, hvar [.DHT11] or DHT22 pin, tvar, hvar [.DHT11]	Returns the temperature and humidity using the DHT22 sensor. Alternative versions of the DHT22 are the AM2302 or the RHT03 (all are compatible). 'pin' is the I/O pin connected to the sensor. Any I/O pin may be used but it must not be configured: the command sets it up and returns it to the not configured state afterwards. 'tvar' is the variable that will hold the measured temperature and 'hvar' is the same for humidity. Both must be present and both must be floating point variables. For example: HUMID 3, TEMP!, HUMIDITY! Temperature is measured in °C and the humidity is percent relative humidity. Both will be measured with a resolution of 0.1. If an error occurs (sensor not connected or corrupt signal) both values will be 1000.0. Normally the signal pin of the DHT22 should be pulled up by a 1K to 10K resistor (4.7K recommended) to the supply voltage. If the sensor is powered from 5V the pin must be 5V tolerant (see the pin table in the chapter <i>Hardware Features</i>). The optional DHT11 parameter modifies the timings to work with the DHT11. Set to 1 for DHT11 and 0 or omit for DHT22. The DHT11 readings have a resolution of 1 degree and 1%. DHT22 and BITBANG DHT22 are the older names of this command and are still accepted. See the chapter <i>Special Device Support</i>.
I2C OPEN speed, timeout I2C WRITE addr, option, sendlen, senddata [,senddata ..]	More detail is in Appendix B. Enables the first I²C channel (on the rear I/O connector) in master mode. 'speed' is the clock speed (in KHz) to use and must be one of 100, 400 or 1000. 'timeout' is a value in milliseconds after which the master send and receive commands will be interrupted if they have not completed. The minimum value is 100. A value of zero will disable the timeout (though this is not recommended). The channel cannot be opened while a Wii controller or an I²C mouse uses it (CONTROLLER); WRITE, READ and CHECK can then be used on it without opening it. Send data to the I²C slave device. 'addr' is the slave's I²C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command: a stop condition will not be sent at the completion of the command, and the next command on the channel begins with a repeated start.

	IMAGE WARP_V warps vertically: the area written runs from 'y1' down to 'y2' (which must be below 'y1'), 'w1' and 'w2' are its widths at its top and bottom edges, and 'x2', 'y2' is the left end of its bottom edge (its bottom left corner).
INC var [,increment]	Increments the variable "var" by either 1 or, if specified, the value in increment. "increment" can be negative which will decrement. This is functionally the same as <code>var = var + increment</code> but is processed much faster If "var" is a string variable "increment" must be given and must be a string, which is appended to "var" (as CAT does). The error is "String too long" if the result is longer than the variable can hold.
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 above except that the input is read from a serial port or file previously opened for INPUT as 'nbr'. See the OPEN command.
INTERRUPT [myint]	This command triggers a software interrupt. The interrupt is set up using INTERRUPT 'myint' where 'myint' is the name of a subroutine that will be executed when the interrupt is triggered. Use INTERRUPT 0 to disable the interrupt Use INTERRUPT without parameters to trigger the interrupt. Note: the interrupt can also be triggered from within a CSUB
IR dev, key, int or IR CLOSE	Decodes NEC or Sony infrared remote control signals. An IR Receiver Module is required to sense the IR light and demodulate the signal. It is soldered to the pads for it on the motherboard (see IR Receiver in the chapter <i>Hardware Features</i>), which are connected to pin 41, and this command sets that pin up. The IR signal decode is done in the background and the program will continue after this command without interruption. 'dev' and 'key' should be numeric variables and their values will be updated whenever a new signal is received ('dev' is the device code transmitted by the remote and 'key' is the key pressed). They must be global (or STATIC), not LOCAL, as they are set in the background. 'int' is a user defined subroutine that will be called when a new key press is received or when the existing key is held down for auto repeat. In the interrupt subroutine the program can examine the variables 'dev' and 'key' and take appropriate action. The IR CLOSE command will terminate the IR decoder.

	Note that for the NEC protocol the bits in 'dev' and 'key' are reversed. For example, in 'key' bit 0 should be bit 7, bit 1 should be bit 6, etc. This does not affect normal use but if you are looking for a specific numerical code provided by a manufacturer you should reverse the bits. This describes how to do it: http://www.thebackshed.com/forum/forum_posts.asp?TID=8367 See the chapter <i>Special Device Support</i> for more details.
IR SEND pin, dev, key	Generate a 12-bit Sony Remote Control protocol infrared signal. 'pin' is the I/O pin to use. This can be any I/O pin which will be automatically configured as an output and should be connected to an infrared LED. Idle is low with high levels indicating when the LED should be turned on. 'dev' is the device being controlled and is a number from 0 to 31, 'key' is the simulated key press and is a number from 0 to 127. The IR signal is modulated at about 38KHz and sending the signal takes about 25mS during which program execution is paused.
IRETURN	Returns from an interrupt routine. An interrupt routine is normally a defined subroutine which is terminated with END SUB, however an interrupt can also be specified as a line number or a label and in that case the interrupt code must be terminated with IRETURN.
KILL file\$ [, ALL]	Deletes the file (or empty directory) specified by 'file\$' on the SD card or USB flash drive. Any extension must be specified. Bulk erase is triggered if 'file\$' contains a '*' or a '?' character: every file that matches is deleted (directories, hidden and system files are left alone), and this can only be done at the command prompt. If ALL is added you will be prompted for a single confirmation. If ALL is not specified you will be prompted on each file.
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: <pre>Var = 56</pre> A whole array can be copied into another array by using the array names with empty brackets. For example: <pre>b%() = a%()</pre> Both arrays must be the same type (for strings, also the same maximum length) and must hold the same total number of elements; the dimensions may differ (eg, a 2 x 3 array can be copied into a six element array). The source can also be a user defined function that returns an array (see the FUNCTION command).
LIBRARY	The library is a second program whose subroutines, functions, embedded C routines (CSUBs), fonts, labels and DATA statements are available to every program, as if they were part of MMBasic: they do not show when the program is listed or edited and are not deleted when a new program is loaded or by NEW. Any code in the library that is not in a subroutine or function is run before each program, which can be used to set constants, options, etc. See The Library in the chapter Using MMBasic. The library is kept in the last 128KB of the program flash memory (flash slot 6, see OPTION FLASH), which cannot then be used for a program, and it can be up to 128KB long. A program can also load a library of its own into RAM (LIBRARY LOAD with RAM).
LIBRARY SAVE	Adds the program in program memory (the program last run, or loaded from the editor or AUTOSAVE with F1) to the library, compressed: its comments, blank lines and unnecessary spaces are removed and the hex codes of its CSUBs and fonts are kept as binary. Program memory is then cleared, as by

LIBRARY DELETE	NEW. LIBRARY SAVE can be used again to add more routines to the library. It can only be used at the command prompt.
LIBRARY LIST or LIBRARY LIST ALL	Deletes the library and returns its flash memory for normal use (OPTION RESET does not delete it). It can only be used at the command prompt.
LIBRARY DISK SAVE file\$ or LIBRARY DISK LOAD file\$	Lists the library (the RAM library while there is one), with a pause at every screen full. ALL lists without pauses. It can only be used at the command prompt.
LIBRARY LOAD file\$ [, file\$...] [, O] [, RAM]	LIBRARY DISK SAVE saves the library (the RAM library while there is one) to the file 'file\$' as a binary image, and LIBRARY DISK LOAD loads such a file into the library in the flash memory, replacing the library there. ".LIB" is added to 'file\$' if it has no extension. Together these allow a library to be kept with the programs that use it, and to be distributed. They can only be used at the command prompt. LIBRARY LOAD file\$ [, file\$...] [, O] [, RAM] Reads one or more BASIC source files (up to eight, joined in the order given; ".BAS" is added to a name without an extension), compresses them and installs them as the library, replacing whatever was there. Unlike LIBRARY SAVE it can be used in a program, so that a program can bring its own library rather than requiring the user to install it first. It must then be the first statement in the program: once the library is installed the program is restarted, so that the library's own initialisation code runs before the program proper. A hash of the source files is remembered, so if the library already holds those exact files the command does nothing and no flash is written: a program can therefore use it on every run at no cost. If a different library is already in the flash memory MMBasic asks for confirmation before replacing it; the option O (or OVERWRITE) suppresses the question. A bare O, OVERWRITE or RAM in the list is the option of that name rather than a file name: a file name is always a string expression, so a quoted "O" is a file name. Nothing is written until every file has been read, so a mistyped name leaves the existing library untouched. With RAM the library goes into RAM instead of the flash memory, which is left exactly as it was: the RAM library takes the place of the flash library until the next RUN, NEW, CHAIN or program load, so a program loads it every time it starts. No question is asked and O is not needed.
LINE x1, y1, x2, y2 [, [-] LW [, C]]	Draws a line on the display starting at the coordinates 'x1' and 'y1' and ending at 'x2' and 'y2'. 'LW' is the line's width. It defaults to 1 if not specified. For lines that have a defined width and the x1 and y1 coordinate define the top-left pixel of the thick line. i.e. the line is to the right of the specified position or below it on the screen. If width is given as a -ve value then width will apply to lines in all directions and they are centred on the given origin and destination coordinates. 'C' is an integer representing the colour and defaults to the current foreground colour. All parameters can 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. Note that it does not accept arrays as parameters.

LINE GRAPH x(), y() [, colour]	This command generates a line graph of the coordinate pairs specified in “x()” and “y()”. The graph will have n-1 segments where there are n elements in the x array; the y array must have at least as many. ‘colour’ defaults to the current foreground colour.
LINE INPUT [prompt\$,] string-variable\$	Reads an entire line from the console input into ‘string-variable\$’. ‘prompt\$’ is a string expression and if specified it will be printed first. It is followed by a comma or a semicolon (they act the same). A question mark is not printed unless it is part of ‘prompt\$’. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items.
LINE INPUT #nbr, string-variable\$	Same as above except that the input is read from a serial communications port or a file previously opened for INPUT as ‘nbr’. See the OPEN command.
LINE PLOT ydata() [,nbr] [,xstart] [,xinc] [,ystart] [,yinc] [,colour]	Plots a line graph from an array of y-axis data points. ‘ydata()’ is an array of floats or integers to be plotted ‘nbr’ (1 to MM.HRES-1) limits the plot to the first ‘nbr’ elements of the array (all of them if it is omitted): the points plotted are the elements from ‘ystart’ on, every ‘yinc’ elements, that lie within them, so with the other defaults ‘nbr’ points are joined by ‘nbr’-1 line segments. ‘xstart’ is the x-coordinate to start plotting - defaults to 0 ‘xinc’ is the increment along the x-axis to plot each coordinate - defaults to 1 ‘ystart’ is the location in ydata to start the plot - defaults to the array start. NB: respects OPTION BASE The values in ‘ydata()’ are the y coordinates of the points; they are clipped to the screen and the plot stops at its right edge. ‘yinc’ is the increment to the index into ydata to add for each point to be plotted - defaults to 1 ‘colour’ is the colour to draw the line - defaults to the current foreground colour
LIST [file\$] or LIST ALL [file\$] LIST FLASH or LIST FLASH ALL LIST FILES [fspec\$] [, sort] LIST PAGES LIST PROFILE or LIST PROFILE CSV	Lists a program or other text file on the console. LIST on its own lists the current program: the file that the program in program memory was loaded from (see MM.INFO\$(CURRENT)), as it is on the SD card or USB flash drive, with its comments. With ‘file\$’ that file is listed (“.BAS” is added if it has no extension). LIST pauses at every screen full (press any key to go on). LIST ALL lists without pauses, which is useful to capture the listing with a terminal emulator that can save what it receives to a file. Lists the program in program memory as MMBasic prepared it when it was loaded: without its comments, blank lines and unnecessary spaces, with the #INCLUDE files inserted and the #DEFINE substitutions made. ALL lists without pauses. The same as FILES. Lists the video pages of the current mode (see MODE and PAGE): for each its number, address, width, height and size in bytes (in hexadecimal), and whether its lines are each shown twice on the screen (2) or once (1). In the 4-bit colour depth it also gives the address of the display that MMBasic builds from the pages. Lists the profile of the program last run with OPTION PROFILING ON: how many times each of its lines ran and how long it took. Each line of the program, as it is held in program memory (see LIST FLASH), is listed with five fields separated by commas:

	the number of times the line ran the average time it took, in microseconds the line, in quotes (making it easy to load into a spreadsheet) the name of the #INCLUDE file that the line comes from, or nothing for the program's own file the number of the line in that file. LIST PROFILE lists on the console and can only be used at the command prompt. LIST PROFILE CSV writes the same to a file in the program's directory, named as the program's file with .CSV in place of .BAS. It can be used in a program, eg to save the profile of each part of a run in its own file (rename the file before the next). The errors "Profiling not enabled" and "Nothing profiled: RUN the program first" say why there is nothing to list.
LIST TYPE [name]	Lists the structure types (see TYPE) that the program has defined, or only the type 'name': each member with its type and its offset, and the size of the structure in bytes.
LIST COMMANDS [V] or LIST FUNCTIONS [V]	Lists all the commands, or all the functions and operators, that MMBasic recognises, in columns, and how many there are. With V it also gives the number of tokens they use.
LMID(array%(),start [,num])=string\$	inserts 'string\$' into the longstring 'array%()' at position 'start' replacing 'num' existing characters. If 'num' isn't specified then it is calculated from the length of 'string\$'. 'num' can be 0 in which case 'string\$' is inserted at the position specified.
LOAD BMP file\$ [,x] [,y] [,mode] [,ximage] [,yimage] or LOAD IMAGE file\$ [,x] [,y] [,mode] [,ximage] [,yimage]	Loads a BMP image from 'file\$' and draws it on the current write page with its top left corner at 'x', 'y' (0, 0 if they are not specified). If an extension is not specified ".BMP" will be added to the file name. All the usual types of the BMP format are supported including black and white and true colour 24-bit images: 1, 4, 8, 16 and 24 bits a pixel, uncompressed or RLE encoded. The image can be up to 3840 x 2160 pixels and positioned anywhere: pixels that end up off the screen are ignored. 'ximage' and 'yimage' specify the pixel of the image that is drawn at 'x', 'y' (0, 0 if they are not specified), so that a part of a large image can be shown. 'mode' (-1 to 7) is accepted so that the PicoMite's programs run: there it chooses a dither, here the image is drawn in the colour depth of the screen and 'mode' is not used. LOAD IMAGE is the PicoMite's name for LOAD BMP.
LOAD CONTEXT [KEEP]	Restores the variable space to the previously saved state and optionally preserves the stored variables to allow a second LOAD if required. See also SAVE CONTEXT
LOAD DATA fname\$, address	Loads the file 'fname\$' as binary data and writes it to the memory address specified. Note that there is no validation of the address so the command must be used with care.

LOAD FONT file\$	Loads the font contained in ‘file\$’ and installs it as font #8. If an extension is not specified “.FNT” will be added to the file name. A program can load many fonts this way during the course of its execution but each new font replaces the one loaded before. The file has the format of an embedded font (see DEFINEFONT) except that no comments or blank lines are allowed, the first line must be DefineFont #8 (spelt so), 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. See Embedded Fonts in the chapter <i>Graphics Functions</i>. You can convert font files designed for the original Colour Maximite using FontTweak, included in the Colour Maximite 2 download.
LOAD GIF [file\$ [,x [,y]]]	Loads a GIF image from ‘file\$’ and draws it with its top left corner at ‘x’, ‘y’ (0, 0 if they are not specified). If an extension is not specified “.GIF” will be added to the file name. 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) and repeat until it is stopped. 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.
LOAD JPG file\$ [,x] [,y] [,mode] [,ximage] [,yimage] [,scale]	Loads a JPG image from ‘file\$’ and draws it on the current write page with its top left corner at ‘x’, ‘y’ (0, 0 if they are not specified), which must be on the screen; the image is cropped at the right and bottom edges of the screen. If an extension is not specified “.JPG” will be added to the file name. Progressive JPG images are not supported. ‘ximage’ and ‘yimage’ specify the pixel of the image that is drawn at ‘x’, ‘y’ (0, 0 if they are not specified), so that a part of a large image can be shown. They must be within the image. ‘scale’ reduces the displayed size of the image by pixel averaging: use 2 for 50% (each output pixel is the average of a 2x2 block of source pixels), 4 for 25% (a 4x4 block) or 8 for 12.5%. The default is 1 which displays the image at full size. The optional ‘mode’ parameter dithers the image to fewer colours, which can give a better picture in the 4 and 8-bit colour depths. Bits 0 and 1 specify the colours and bit 2 the type of dithering: -1 = no dithering (the default) 0 = Floyd-Steinberg to RGB121 (the colours of the 4-bit colour depth) 1 = Floyd-Steinberg to RGB222 2 = Floyd-Steinberg to RGB332 (the colours of the 8-bit colour depth) 4 = Atkinson to RGB121 5 = Atkinson to RGB222 6 = Atkinson to RGB332
LOAD PNG file\$ [,x] [,y] [,t] [,cutoff] [,ximage] [,yimage]	Loads a PNG image from ‘file\$’ and draws it on the current write page with its top left corner at ‘x’, ‘y’ (0, 0 if they are not specified). If an extension is not specified “.PNG” will be added to the file name. The image can be of any size and positioned anywhere: pixels off the screen are ignored, and as the image is decoded a line at a time a large image needs little memory. The file can be an 8-bit RGB, RGBA (ARGB8888), grey or grey with alpha PNG, or an indexed (palette) one; interlaced files are not supported. ‘ximage’ and ‘yimage’ specify the pixel of the image that is drawn at ‘x’, ‘y’ (0, 0 if they are not specified), so that a part of a large image can be shown. They must be within the image.

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 be 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 a 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%()' (all of it if it is shorter). 'src%()' and 'dest%()' must be long string variables.
LONGSTRING LOAD array%(), nbr, string\$	Will copy the first 'nbr' characters of 'string\$' (all of them if it is shorter) 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.
LONGSTRING PRINT [#n,] src%() [;]	Prints the long string 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. A semicolon (;) at the end suppresses the carriage return and line feed that are otherwise printed after it.
LONGSTRING REPLACE array%(), string\$, start	Will substitute the characters of the normal MMBasic string 'string\$' into an existing long string 'array%()', starting at position 'start' in the long string. The replaced characters must lie within the long string (its length is not changed).
LONGSTRING RESIZE array%(), nbr	Sets the length of the long string to 'nbr' (0 to the number of characters its array can hold). This overrides the length set by the other long string commands so should be used with caution. Typical use would be in using a long string 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%()' (all of it if it is shorter). 'src%()' and 'dest%()' must be long string variables.
LONGSTRING SETBYTE array%(), nbr, data	Sets byte 'nbr' of the long string to the value 'data' (0 to 255). 'nbr' respects OPTION BASE: the first byte is 0, or 1 under OPTION BASE 1. The length of the string is not changed.
LONGSTRING TRIM array%(), nbr	Will trim 'nbr' characters (1 to its length) from the left of a long string. 'array%()' must be a long string variable.
LONGSTRING UCASE array%()	Will convert any lowercase characters in 'array%()' to uppercase. 'array%()' must be a long string variable.
LOOP [WHILE expression UNTIL expression]	Terminates a program loop: see DO.

MAP	The MAP commands set the colours used in the 8-bit and 4-bit colour depths. Each value in the palette (0 to 255 in the 8-bit depth, 0 to 15 in the 4-bit depth) can be set to any 24-bit colour. They are refused in the other colour depths.
MAP(n) = rgb%	Assigns the 24-bit colour ‘rgb%’ to all pixels with the palette value ‘n’. The change is made by the next MAP SET.
MAP SET	Updates the colour map (set with MAP(n) = rgb%) during the next frame blanking interval. The command waits for it.
MAP RESET	Resets the colour map to the default colours.
MAP GREYSCALE or MAP GRAYSCALE	Sets a colour map of shades and applies it at the next frame blanking interval. In the 8-bit depth it makes eight ramps of 32 levels: grey (values 0 to 31), blue (32 to 63), green, cyan, red, magenta, yellow, and grey again (224 to 255). In the 4-bit depth it makes 16 levels of grey.
MAP MAXIMITE	In the 8-bit depth, sets the colour map to the colours of the original Colour Maximite (palette values 0 to 7 and 48 to 55) and applies it at the next frame blanking interval.
MATH	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 RANDOMIZE [n]	Seeds the Mersenne Twister algorithm used by MATH(RAND). If n is not specified the seed is a number from the hardware random number generator. Otherwise ‘n’ is 0 to 4294967295, and the same ‘n’ always gives the same sequence of numbers. The Mersenne Twister algorithm gives a much better random number than the C-library inbuilt function.
Simple array arithmetic	
MATH SET nbr, array()	See ARRAY SET
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 ‘scale’ to 1 is optimised and is the fastest way of copying an entire array.
MATH ADD in(), num ,out()	See ARRAY ADD
MATH INTERPOLATE in1(), in2(), ratio, out()	This command implements the following equation on every array element: $\text{out} = (\text{in2} - \text{in1}) * \text{ratio} + \text{in1}$ Arrays can have any number of dimensions and must be distinct and have the same number of total elements. The command works with both integer and floating point arrays in any mixture
MATH WINDOW in(), minout, maxout, out() [,minin!, maxin!]	This command takes the “in” array and scales it between “minout” and “maxout” returning the answer in “out”. Optionally, it can also return the minimum and maximum floating point values found in the original data

	(“minin!” and “maxin!”). Note: “minout” can be greater than “maxout” and in this case the data will be both scaled and inverted. This command can massively simplify scaling data for plotting etc. Example: DIM IN(2)=(1,2,3) DIM OUT(2) MATH WINDOW IN(),7,3,OUT(),LOW,HIGH Will return OUT(0)=7, OUT(1)=5,OUT(2)=3,LOW=1,HIGH=3
MATH CLAMP in(), lo, hi, out()	Limits every element of ‘in()’ to the range ‘lo’ to ‘hi’ and puts the result in ‘out()’. Anything below ‘lo’ becomes ‘lo’, anything above ‘hi’ becomes ‘hi’ and everything else is copied unchanged. The two arrays must have the same number of elements but can be integer or floating point in any combination; when the output array is an integer array the limits are rounded to whole numbers. An error is reported if ‘lo’ is greater than ‘hi’. Note that this is not the same as MATH WINDOW, which rescales the data to fit the range given rather than clipping it. Example: DIM v(3) = (-8,2,40,11) DIM w(3) MATH CLAMP v(),0,10,w() Will return w(0)=0, w(1)=2, w(2)=10, w(3)=10
MATH SLICE sourcearray(), [d1] [,d2] [,d3] [,d4] [,d5] , destinationarray()	See ARRAY SLICE
MATH INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5] , sourcearray()	See ARRAY INSERT
MATH POWER inarray(), power, outarray()	Raises each element in ‘inarray()’ to the ‘power’ defined and puts the output in ‘outarray()’
MATH SHIFT inarray%(), nbr, outarray%() [,U]	This command does a bit shift on all elements of inarray%() and places the result in outarray%() (may be the same as inarray%()). nbr can be between -63 and 63. Positive numbers are a left shift (multiply by power of 2). Negative number are a right shift. The optional parameter ,U will force an unsigned shift.
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(). All arrays must be 2D floating point arrays but need not be square, and out() must be a different array from in1() and in2(). If not square then the arrays must be dimensioned in1(m,n) in2(p,m) ,out(p,n)

Vector arithmetic MATH V_PRINT array() [,HEX] MATH V_NORMALISE inV(), outV() MATH V_MULT matrix(), inV(), outV() MATH V_CROSS inV1(), inV2(), outV() MATH V_ROTATE x, y, a, xin(), yin(), xout(), yout()	Quick mechanism to print a small one-dimensional array on a single line. HEX prints the values in hexadecimal (integer arrays only). Converts a vector inV() to unit scale and puts the answer in outV() ($\text{sqr}(x*x + y*y + \dots) = 1$) There is no limit on number of elements in the vector Multiplies matrix() and vector inV() returning vector outV(). The vectors and the 2D matrix can be any size but must have the same cardinality. Calculates the cross product of two three element vectors inV1() and inV2() and puts the answer in outV() This command rotates the coordinate pairs in “xin()” and “yin()” around the centre point defined by “x” and “y” by the angle “a” and puts the results in “xout()” and “yout()”. NB: the input and output arrays can be the same and the rotation angle is, by default, in radians but this can be changed using the OPTION ANGLE command.
Quaternion arithmetic MATH Q_INVERT inQ(), outQ() MATH Q_VECTOR x, y, z, outVQ() MATH Q_CREATE theta, x, y, z, outRQ() MATH Q_EULER yaw, pitch, roll, outRQ() MATH Q_MULT inQ1(), inQ2(), outQ() MATH Q_ROTATE RQ(), inVQ(), outVQ()	Invert the quaternion in inQ() and put the answer in outQ() Converts a vector specified by x , y, and z to a normalised quaternion vector outVQ() with the original magnitude stored 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. 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 vector 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 Multiplies two quaternions inQ1() and inQ2() and puts the answer in outQ() Rotates the source quaternion vector inVQ() by the rotate quaternion RQ() and puts the answer in outVQ()

MATH C_ADD array1(), array2(), array3() MATH C_SUB array1(), array2(), array3() MATH C_MUL array1(), array2(), array3() MATH C_DIV array1(), array2(), array3() MATH C_AND array1(), array2(), array3() MATH C_OR array1(), array2(), array3() MATH C_XOR array1(), array2(), array3()	These commands do cell by cell operations on arrays. array1 and array2 are the parameters and array3 is the output. All arrays must be the same size and the same type (float or integer). There are no restrictions on the number of dimensions and no restrictions on using the same array twice or even three times in the parameters. eg, MATH C_MUL a%(),a%(),a%() will square all the values in the array a%() C_MULT is another name for C_MUL. C_DIV of integer arrays gives whole numbers (the remainder is dropped) and an error if a divisor is zero. C_AND, C_OR and C_XOR of floating point arrays work on the whole number part of each value.
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 from 2 to 32768 (eg, 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 (eg, 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 from 2 to 32768 (eg, 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 from 2 to 32768 (eg, 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 from 2 to 32768 (eg, 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 (in radians) of the signal at various frequencies according to the formula above.
MATH SENSORFUSION type ax, ay, az, gx, gy, gz, [mx], [my], [mz], pitch, roll, yaw [,p1] [,p2]	Type can be MAHONY or MADGWICK Ax, ay, and az are the accelerations in the three directions and should be specified in units of standard gravitational acceleration. Gx, gy, and gz are the instantaneous values of rotational speed which should be specified in radians per second, or in degrees per second under OPTION ANGLE DEGREES. Mx, my, and mz are the magnetic fields in the three directions and should be specified in nano-Tesla (nT). They can be omitted, keeping their commas (eg, gz, , , pitch), in which case the fusion uses only the accelerometer and

	gyroscope and the yaw may slowly drift, but this allows sensors like the MPU-6050 to be used. The magnetometer is used when mx is given. Care must be taken to ensure that the x, y and z components are consistent between the three inputs. So , for example, using the MPU-9250 the correct input will be ax, ay,az, gx, gy, gz, my, mx, -mz based on the reading from the sensor. Pitch, roll and yaw should be floating point variables and will contain the outputs from the sensor fusion, in radians, or in degrees under OPTION ANGLE DEGREES. The SENSORFUSION routine will automatically measure the time between consecutive calls (in milliseconds, and at most one second) and will use this in its internal calculations. The Madgwick algorithm takes an optional parameter p1. This is used as beta in the calculation. It defaults to 0.5 if not specified The Mahony algorithm takes two optional parameters p1, and p2. These are used as Kp and Ki in the calculation. If not specified these default to 10.0 and 0.0 respectively. A fully worked example of using the code is given on the BackShed forum at: https://www.thebackshed.com/forum/ViewTopic.php?TID=13459&PID=166962#166962												
MATH SINC x_in(), y_in(), n, m, window, freq, x_out(), y_out() MATH SINC x_in(), y_in(), n, window, freq, x_out(), y_out()	Apply a windowed sinc filter to smooth or interpolate coordinate data. A sinc filter is an ideal low-pass filter that removes high-frequency noise while preserving the signal shape. It is particularly good for resampling (interpolation). Parameters: <table> <tr> <td>x_in(), y_in()</td><td>Input coordinate arrays (float, one-dimensional, with at least n elements)</td></tr> <tr> <td>n</td><td>Number of input points, 1 to 10000</td></tr> <tr> <td>m</td><td>Number of output points (optional), 1 to 20000. If omitted or m=n: Smoothing mode (output size n). If m!=n: Interpolation/Resampling mode (output size m); n must then be at least 2, the input points are taken to be evenly spaced in x, and x_out() is set to m evenly spaced values from the first x_in() to the last.</td></tr> <tr> <td>window</td><td>Filter kernel size, 3 to 101 (must be odd; even values are incremented). Typical values: 15 to 101. Larger values give a sharper cutoff but are slower.</td></tr> <tr> <td>freq</td><td>Normalized cutoff frequency (0.0 < freq <= 0.5). 0.5 is the Nyquist frequency (no filtering). Typical values: 0.1 (heavy smoothing) to 0.4 (light smoothing).</td></tr> <tr> <td>x_out(), y_out()</td><td>Output coordinate arrays (float, one-dimensional, with at least as many elements as the output)</td></tr> </table>	x_in(), y_in()	Input coordinate arrays (float, one-dimensional, with at least n elements)	n	Number of input points, 1 to 10000	m	Number of output points (optional), 1 to 20000. If omitted or m=n: Smoothing mode (output size n). If m!=n: Interpolation/Resampling mode (output size m); n must then be at least 2, the input points are taken to be evenly spaced in x, and x_out() is set to m evenly spaced values from the first x_in() to the last.	window	Filter kernel size, 3 to 101 (must be odd; even values are incremented). Typical values: 15 to 101. Larger values give a sharper cutoff but are slower.	freq	Normalized cutoff frequency (0.0 < freq <= 0.5). 0.5 is the Nyquist frequency (no filtering). Typical values: 0.1 (heavy smoothing) to 0.4 (light smoothing).	x_out(), y_out()	Output coordinate arrays (float, one-dimensional, with at least as many elements as the output)
x_in(), y_in()	Input coordinate arrays (float, one-dimensional, with at least n elements)												
n	Number of input points, 1 to 10000												
m	Number of output points (optional), 1 to 20000. If omitted or m=n: Smoothing mode (output size n). If m!=n: Interpolation/Resampling mode (output size m); n must then be at least 2, the input points are taken to be evenly spaced in x, and x_out() is set to m evenly spaced values from the first x_in() to the last.												
window	Filter kernel size, 3 to 101 (must be odd; even values are incremented). Typical values: 15 to 101. Larger values give a sharper cutoff but are slower.												
freq	Normalized cutoff frequency (0.0 < freq <= 0.5). 0.5 is the Nyquist frequency (no filtering). Typical values: 0.1 (heavy smoothing) to 0.4 (light smoothing).												
x_out(), y_out()	Output coordinate arrays (float, one-dimensional, with at least as many elements as the output)												
MATH PID INIT channel, pid_params!(), callback	This command sets up a PID controller that can work automatically in the background. Up to 16 PID controllers can run simultaneously (channels 1 to 16). ‘callback’ is a MMbasic subroutine which is called at the rate defined by the sample time. See the MATH(PID ...) function for details of what should be included in the subroutine. The pid_params!() array must be dimensioned for all the listed elements, including the controller memory parameters (14 elements, eg, DIM pid_params!(13) with OPTION BASE 0) and be initialised with the required settings. The sample time T must be at least 0.001 seconds.												

MATH PID START channel MATH PID STOP channel	'pid_params!()' must be global (or STATIC), not LOCAL, as it is used and updated in the background. PID configuration Element 0 = Kp Element 1 = Ki Element 2 = Kd Element 3 = tau 'Derivative low-pass filter time constant Element 4 = limMin 'Output limits Element 5 = limMax Element 6 = limMinInt 'Integrator limits Element 7 = limMaxInt Element 8 = T 'Sample time (in seconds) Controller "memory" Element 9 = integrator Element 10 = prevError Element 11 = differentiator Element 12 = prevMeasurement Element 13 = out Starts a previously initialised PID controller on the channel specified Stops a previously initialised PID controller on the channel specified and deletes the internal data structures See https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=17263 For an example of setting up and running a PID controller
MATH AES128 ENCRYPT/DECRYPT CBC/ECB/CTR key\$/key(), in\$/in(), out\$/out() [,iv\$/iv()]	This command encrypts or decrypts the data in 'in' and puts the answer in 'out' using the AES128 encryption method specified The parameters can each be either a string, integer array, or float array with any mix possible The key must be 16 elements long (16*8=128bits), in and out must be a multiple of 16 elements long. In the case of out being specified as a string (e.g. out\$), the string variable must exist and should be set to empty (DIM out\$="") The maximum number of elements in 'in' and 'out' is limited by memory when defined as arrays. Strings for encrypting are limited to 240bytes (ECB) and 224bytes (CTR and CBC). For CBC and CTR encryption you can optionally specify an initialisation vector 'iv'. 'iv' must be 16 elements long (16*8=128bits). If an initialisation vector is not specified encryption will generate a random initialisation vector. In both cases the output is prepended with the IV. For CBC and CTR, decryption requires that the first 16 elements of the input are the initialisation vector. In the case where you want to transmit a message without IV you should remove the IV before sending the message using standard MMBasic manipulations. In this case the recipient must know the IV as well as the key and create a complete message before using DECRYPT by prepending the IV to the incoming message.
MEMORY	List the amount of memory currently in use. For example: Program: 12K (2%) Program (475 lines)

	504K (98%) Free Data: 1K (1%) 4 Variables 2K (0%) General 24285K (99%) Free Notes: • Memory usage is rounded to the nearest 1K byte. • ‘Program’ is the program memory: the program last run (see the chapter Using MMBasic) and, when there are any, its embedded C routines and fonts and the saved variables (see VAR SAVE), each on a line of its own. When there is a library (see LIBRARY) it is listed under Library. • General memory (RAM) is used by arrays, strings, serial I/O buffers, etc. • The amount of RAM depends on the generation: the example is from a Generation 2 (see the memory map in the chapter MMBasic Characteristics).
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 ,numberoffloats [,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 and FLOAT = Eight bytes per memory address. ‘increment’ (INTEGER and FLOAT only, 0 to 65535) 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, destinationaddress, numberofbytes MEMORY COPY INTEGER sourceaddress, destinationaddress, numberofintegers [,sourceincrement][,destination increment] MEMORY COPY FLOAT sourceaddress, destinationaddress,	This command will copy one region of memory to another. COPY INTEGER and FLOAT will copy eight bytes per operation. ‘sourceincrement’ (INTEGER and FLOAT only, 0 to 65535) 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.

numberoffloats [,sourceincrement][,destination increment]	
MEMORY PRINT [#]fnbr , nbr, address%/array() MEMORY INPUT [#]fnbr , nbr, address%/array()	These commands save or read ‘nbr’ of data bytes from or to memory from or to an open disk file. The memory to be saved can be specified as an integer array in which case the nbr of bytes to be saved or read is checked against the array size. Alternatively, a memory address can be used in which case no checking can take place and user errors could result in a crash of the firmware..
MEMORY PACK source%()/sourceaddress%, dest%()/destaddress%, number, size MEMORY UNPACK source%()/sourceaddress%, dest%()/destaddress%, number, size	Memory pack and unpack allow integer values from one array to be compressed into another or uncompressed from one to the other. The two arrays are always normal integer arrays but the packed array can have 2, 4, 8, 16 or 64 values “packed into them. Thus a single integer array element could store 2 off 32-bit words, 4 off 16 bit values, 8 bytes, 16 nibbles, or 64 booleans (bits). “number specifies the number of values to be packed or unpacked and “size” specifies the number of bits (1,4,8,16,or 32) Alternatively, memory address(es) can be used in which case no checking can take place and user errors could result in a crash of the firmware.
MKDIR dir\$	Make, or create, the directory ‘dir\$’ on the SD card or USB flash drive (on the current drive unless ‘dir\$’ names another).
MID\$(str\$, start [, num]) = str2\$	The ‘num’ characters in 'str\$', beginning at position 'start', are replaced by the characters in 'str2\$'. ‘num’ can be set be 0 i.e. insert the new string at the position specified. If num isn’t specified it defaults to the length of ‘str2\$’. The characters replaced must lie within 'str\$' ("Selection exceeds length of string") and the result must fit in the variable ("String too long").
MMDEBUG [expression [[,;]expression] ...] or MMDEBUG BREAK	A command for debugging a program that costs nothing when it is not wanted. When a program is loaded (by RUN, or by F1 or F2 in the editor or AUTOSAVE), a line that starts with MMDEBUG is dropped, as a comment is, unless the directive #MMDEBUG ON has come before it in the program; #MMDEBUG OFF drops the lines that follow it again. So MMDEBUG commands can be left throughout a program while it is developed and turned on or off with one line. An #INCLUDE file starts with the setting in force at its #INCLUDE. MMDEBUG on its own or with a list of expressions acts exactly like PRINT. MMDEBUG BREAK stops the program and shows the prompt DEBUG>. Any command can then be entered as at the command prompt, to print or change variables, call subroutines and so on: commands that make sense at the command prompt (not GOTO, for example). The command CONTINUE leaves this mode and the program carries on after the MMDEBUG BREAK.
MODE r [, bits [, bg [, int]]]	Sets the resolution and the colour depth of the VGA video output. ‘r’ is the screen resolution. It is a number from 1 to 18 as follows: 1 = 800 x 600 pixels (the default) 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

	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, lines not duplicated: not supported by all monitors) 15 = 1280 x 1024 pixels 16 = 1920 x 1080 pixels (widescreen format, Generation 2 with the Rev V processor only) 17 = 384 x 240 pixels 18 = 1024 x 600 pixels 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). ‘bits’ is the colour depth. It can be 4, 8, 12, 16 or 32 and is 8 if it is left out: 4 = RGB121, 16 colours, two pixels to a byte: the PicoMite's 16 colour display (see below) 8 = RGB332, 256 colours, each of which can be changed with the MAP command 12 = ARGB4444, 4096 colours with 16 levels of transparency, in two layers over a background colour 16 = RGB565, 65,536 colours 32 = ARGB8888, 16,777,216 colours (Generation 2 only) Not every combination is available: modes 9, 11 and 12 have no 12 or 32-bit depth, modes 15 and 16 have the 8-bit depth only, and the 4-bit depth is not available in modes 9, 11, 12, 15, 16 and 18. With OPTION LEGACY ON only the 8-bit depth can be used. ‘bg’ is the background colour and can be used in the 12-bit depth. If the pixels in page 0 (the lower layer) are not set to solid (transparency = 15) then the background will show through as determined by the transparency value of the pixel. This parameter is ignored in the other depths. ‘int’ is a subroutine that will be called at the start of each frame blanking (useful for updating the screen without tearing). A MODE command without it cancels the interrupt. To give ‘int’ without a background colour leave ‘bg’ out (MODE r, bits, , int) or give 0. MODE sets the colours to white on black and the default font for the resolution: font 7 in modes 3, 5, 6, 7, 13 and 17, font 4 in modes 9, 11, 15 and 18, font 3 in mode 16 and font 1 in the others. When it changes the resolution or the colour depth it also clears the screen and makes page 0 the write page. A change of colour depth also closes the sprites and deletes a cursor loaded by GUI CURSOR LOAD, as they are held in the pixel format of the old depth. A program is started (RUN) in the default mode, MODE 1 or the mode set by OPTION DEFAULT MODE with the 8-bit depth, and the display returns to it when the program ends. The display always shows the contents of page 0 (pages 0 and 1 in the 12-bit depth, page 1 being the upper layer) unless overridden with the PAGE DISPLAY command. Use PAGE WRITE and PAGE COPY to avoid the artefacts of flashing and tearing. The pages available in each mode and colour depth, and how to PEEK and POKE them, are listed in the chapter Video Modes and Pages. MM.INFO(MAX PAGES) returns the highest page number in the current mode. The 4-bit depth gives the display of the PicoMite's VGA and HDMI versions in their 16 colour modes: MODE 7,4 (320 x 240) is the PicoMite's MODE 2 and MODE 8,4 (640 x 480) is its MODE 3. Its 16 colours can be changed with MAP and its FRAMEBUFFER command is the PicoMite's (see
--	--

	FRAMEBUFFER). PAGE DISPLAY and the IMAGE commands are not available in it.
MOUSE INTERRUPT ENABLE channel, int or MOUSE INTERRUPT DISABLE channel or MOUSE SET channel, x, y [, wheel]	These commands act on the mouse in use, which a program reads as channel 2 (see the chapter <i>Keyboard, Mouse and Gamepad</i> and the DEVICE(MOUSE ...) function). 'channel' can be 1 to 4 for MOUSE INTERRUPT, where it is not used, and must be 2 for MOUSE SET. MOUSE INTERRUPT ENABLE sets 'int', a user defined subroutine that will be called whenever a button of a USB mouse is pressed or released, and when the left button is double clicked. MOUSE INTERRUPT DISABLE disables it. For a PS2 or I²C mouse see CONTROLLER MOUSE OPEN. MOUSE SET sets the position that the mouse will report: 'x' (0 to MM.HRES-1), 'y' (0 to MM.VRES-1) and optionally the wheel count (-1000000 to 1000000). The error is "No mouse" if no mouse is connected or open.
NEW	Clears the program memory and all variables including saved variables.
NEXT [counter-variable] [, counter-variable], etc	NEXT comes at the end of a FOR-NEXT loop; see FOR. The 'counter-variable' specifies exactly which loop is being operated on. If no 'counter-variable' is specified the NEXT will default to the innermost loop. It is also possible to specify multiple counter-variables as in: <pre>NEXT x, y, z</pre>
ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR or ON ERROR RESTART	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, 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 RESTART will cause MMBasic to restart the Colour Maximite 2 when an error occurs and, of course, if you have OPTION AUTORUN set the program will then restart cleanly 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 (1 to 10000), 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, MM.ERRMSG\$ will be set to the error message that would normally be generated and MM.ERRLINE to the number of the line. 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, SKIP and RESTART 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 (typed on the USB keyboard or received from the serial console). 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 ('ASCIIcode' is 1 to 255). This operates at a low level for both the

	USB keyboard and the 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
ON nbr GOTO GOSUB target [, target, target...]	ON either branches (GOTO) or calls a subroutine (GOSUB) depending on the rounded value of 'nbr': if it is 1 the first target is used, if it is 2 the second target, and so on. Each target can be a line number or a label. If 'nbr' is zero or greater than the number of targets the program carries on with the next command. 'nbr' must be between 0 and 255. ON ... GOSUB nests as GOSUB does and the subroutine must be terminated with RETURN. This is a legacy command; SELECT CASE and defined subroutines are recommended for new programs.
ONEWIRE RESET pin or ONEWIRE WRITE pin, flag, length, data [, data...] or ONEWIRE READ pin, flag, length, data [, data...]	Commands for communicating with 1-Wire devices. ONEWIRE RESET will reset the 1-Wire bus ONEWIRE WRITE will send a number of bytes ONEWIRE READ will read a number of bytes 'pin' is the I/O pin (located in the rear connector) to use. It can be any pin capable of digital I/O. 'flag' is a combination of the following options: 1 - Send reset before command 2 - Send reset after command 4 - Only send/recv a bit instead of a byte of data 8 - Invoke a strong pullup after the command (the pin will be set high and open drain disabled) 'length' is the length of data to send or receive 'data' is the data to send or variable to receive. The data to send (WRITE) or the destination for received data (READ) may be supplied in any one of three forms: • A list of individual items. The number of items must equal the length parameter. For WRITE each item is a constant, variable or expression; for READ each item is a numeric variable to receive one value. Array elements such as a(3) are allowed. • An array, specified with empty brackets, e.g. array(). It may be a floating-point or integer array and must be large enough to supply (WRITE) or hold (READ) the specified number of elements, which are taken from / stored into consecutive elements starting at the first. • A string variable. For WRITE the first length characters are sent (the string must be at least that long); for READ the received bytes are stored as sequential characters and the string is set to that length. See also <i>Appendix C</i>.
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. The file systems on both the SD Card and the flash drives are NOT case sensitive. A directory path can be specified with the backslash as directory separators. The parent of the current directory can be specified by using a directory name of ".." (two dots) and the current directory with "." (a single dot). For example: OPEN ".\dir1\dir2\filename.txt" FOR INPUT AS #1

	'mode' is INPUT, OUTPUT, APPEND or RANDOM. INPUT will open the file for reading and throw an error if the file does not exist. OUTPUT will open the file for writing and will automatically overwrite any existing file with the same name. APPEND will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the APPEND mode will act the same as the OUTPUT mode (i.e. the file is created then opened for writing). RANDOM will open the file for both read and write and will allow random access using the SEEK command. When opened the read/write pointer is positioned at the end of the file. If the file does not exist, it will be created. 'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously and can be on either or both the A: and B: drives. 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. COM1: and COM2: are on the rear I/O connector. COM3: is the port of the serial console (on the USB connector used for power) and can be opened when the serial console is disabled with OPTION CONSOLE SCREEN. COM4: to COM7: are USB serial devices plugged into the USB connector (see the chapter <i>Keyboard, Mouse and Gamepad</i>). 'comspec\$' is the communication specification, a string of the form "COMn: baud, buf, int, int-trigger, tx-int" followed by any of the options DEP or DEN, 7BIT, ODD or EVEN, INV, OC and S2. The default is 9600 baud (the minimum is 1200), a receive buffer of 1024 bytes, 8 data bits, no parity and one stop bit. DEP and DEN (RS485 mode, using the COM1 DE pin, pin 37) are available on COM1: only, and INV, OC, DEP and DEN do not apply to COM4: to COM7:.. For a full description with examples see <i>Appendix A</i>. 'fnbr' is the file number (1 to 10). Using 'fnbr' the port can be written to and read from using any command or function that uses a file number, and it is closed with CLOSE.
OPEN comspec\$ AS GPS [, timezone_offset] [, monitor]	Will open a serial communications port (COM1: to COM7:) for reading from a GPS receiver. 'comspec\$' is as for the OPEN command above. The data can then be retrieved using the GPS() function. The sentences interpreted are GPRMC, GNRMC, GPGGA and GNGGA (those with a correct checksum). The 'timezone_offset' parameter is used to convert UTC as received from the GPS to the local timezone. If omitted the timezone will default to UTC. The 'timezone_offset' is in hours and can be any number between -12 and 14 allowing the time to be set correctly even for the Chatham Islands in New Zealand (UTC +12:45, an offset of 12.75). If the 'monitor' parameter is set to 1 then all GPS input is also directed to the console. This can be stopped by closing the GPS channel. The port takes the first free file number. It is closed with CLOSE GPS, and PRINT #GPS sends a sentence to the receiver.
OPTION	See the section <i>Options</i> earlier in this manual.

PAGE	The PAGE commands manage the video pages. Building an image on a page that is not displayed and then copying it to the display page avoids flashing and tearing. The pages available in each mode and colour depth are listed in the chapter Video Modes and Pages, and MM.INFO(MAX PAGES) returns the highest page number in the current mode. In the 4-bit colour depth only PAGE WRITE, PAGE COPY and PAGE SCROLL are available (the others give the error "Invalid for this screen mode"). There pages 0 to 4 are the PicoMite's framebuffers N, L, F, 2 and T (see the FRAMEBUFFER command).
PAGE WRITE n or PAGE WRITE FRAMEBUFFER	Instructs MMBasic to make all graphics commands write to page 'n'. If not used page 0 is the default. MM.HRES and MM.VRES report the size of the write page. Set the page to FRAMEBUFFER to write to the framebuffer: see the FRAMEBUFFER command.
PAGE DISPLAY n [, layer] or PAGE DISPLAY -1	Selects the page that is displayed on the screen (this does not change the write page). For example: <pre>PAGE DISPLAY 3</pre> will display the current contents of page 3. In the 12-bit depth 'layer' can be 1 to select the page that is to be displayed as the top layer, for example: <pre>PAGE DISPLAY 4, 1</pre> 'layer' 0, the default, is the lower layer. The command executes in the next frame blanking period. When a program ends for any reason page 0 is displayed again. In modes 3, 5, 6, 7, 12, 13 and 17 only pages 0 and 1 (pages 0, 1 and 2 in the 12-bit depth) can be displayed, as only they hold each line twice. A page must also lie in the first 3 MB of the video memory (MM.INFO(PAGE ADDRESS n) plus the size of the page no higher than &HD0300000), so on the Generation 2 the higher pages cannot be displayed (the error "Display page address > &HD0300000 - use a lower page number"). PAGE DISPLAY -1 turns off the display (the monitor no longer receives the synchronisation signals). It is only valid in a program. To turn the display back on select a valid display page; the display is also turned back on when the program ends.
PAGE COPY n TO m [, when] [, T]	Copies the contents of one video page to another. 'n' is the source and 'm' is the destination (a comma can be used instead of TO). The two pages must be the same size ("Page size mismatch - use BLIT"). 'when' can be one of the letters I, B or D. If omitted it will default to I. I means do the copy immediately. It is the most efficient but risks causing screen artefacts. B means wait until the next frame blanking and then do the copy. It is the least efficient but is absolutely determinate in its effect and no screen artefacts will ever be seen. D means carry on processing the next command and do the copy in the background when the next frame blanking occurs. This is efficient but must be used with care as subsequent drawing commands may or may not be included in the copy depending on the timing of the next frame blanking. Only one such copy is held: another D copy before the frame blanking replaces it. If T (or 1) is added then only the non-black pixels are copied, for example <pre>PAGE COPY 1 TO 0, B, T.</pre>

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 (negative values scroll it left and down). 'x' can be from -w/2-1 to w/2 and 'y' from -h/2-1 to h/2, where 'w' and 'h' are the width and height of the page (MM.HRES and MM.VRES unless it has been resized). By default the area scrolled off the page 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 RESIZE pageno, w, h	Changes the width and height of a given page in memory (its contents are not rescaled). 'w' can be set between 1 and the normal width for the current display mode and 'h' between 1 and its normal height. Page 0 (pages 0 and 1 in the 12-bit depth) cannot be resized. 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. The next MODE command, and the end of the program, restore the normal size.
PAGE STITCH frompage1, frompage2, topage, offset	Will take the last MM.HRES minus 'offset' columns from 'frompage1' and the first 'offset' columns from 'frompage2' and copies them, side by side, to 'topage'. Stepping 'offset' from 0 to MM.HRES moves the view smoothly from the first page to the second. The three pages must be the same size, and 'frompage1' and 'frompage2' cannot be page 0 (pages 0 and 1 in the 12-bit depth).
PAGE MERGE frompage1, frompage2, topage [, colour]	Copies 'frompage1' to 'topage' with the image on 'frompage2' over it: the pixels of 'frompage2' that are of the transparent colour 'colour' (default 0, black) let 'frompage1' show through. The three pages must be the same size, and 'frompage1' and 'frompage2' cannot be page 0 (pages 0 and 1 in the 12-bit depth).
PAGE AND_PIXELS sourcepage1, sourcepage2, destinationpage or PAGE OR_PIXELS sourcepage1, sourcepage2, destinationpage or PAGE XOR_PIXELS sourcepage1, sourcepage2, destinationpage	These commands combine the pixels on 'sourcepage1' and 'sourcepage2' by ANDing, ORing or XORing them and write the result to 'destinationpage', which can be the same as either of the source pages if required. The three pages must be the same size.
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.

PIN(pin) = value	For a 'pin' configured as digital output this will set the output to low ('value' is zero) or high ('value' non-zero). You can set an output high or low before it is configured as an output and that setting will be the default output when the SETPIN command takes effect. See the function PIN() for reading from a pin and the command SETPIN for configuring it. 'pin' is the number of the pin on the rear I/O connector (see External I/O Connector in the chapter <i>Hardware Features</i>). Refer to the chapter <i>Using the I/O Pins</i> for a general description of the Colour Maximize 2's input/output capabilities.
PIXEL x, y [,c]	Set a pixel on the screen to a colour. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. 'c' is a 24 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 array or a single variable or constant. With OPTION LEGACY ON the original Colour Maximize's form PIXEL(x, y) = c is used instead. See the chapter <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.
PIXEL FAST x, y [,c]	A quicker PIXEL for a single point: 'x' and 'y' are single values (not arrays) and 'c' is the colour (the current foreground colour if it is omitted). PIXEL FAST is another name for CLS with two or three arguments, which draws a pixel in the same way, and a program lists it as CLS.
PIXEL FILL x, y, c	Implements a flood fill by reading the colour of the pixel at coordinates 'x', 'y' on the screen and replacing it and the entire area of connected pixels having the same colour with the new colour 'c'. See also FILL.
PLAY	The PLAY commands generate sound on the audio output: the stereo audio socket on the front panel, driven by the processor's DAC (see the chapter Sound Output). The sound plays in the background, so the program continues while it plays. Only one kind of sound can play at a time. A command that would start another kind while a sound is playing (or paused) gives the error "Sound output in use", so use PLAY STOP first. A PLAY TONE, PLAY SOUND, PLAY ARRAY, PLAY SAMPLE or PLAY BBC SOUND while a sound of its own kind plays changes that sound, and PLAY EFFECT and PLAY MODSAMPLE play over a MOD file. The sound is stopped when a program ends, and RUN sets the volume to the maximum.
PLAY TONE left, right [, dur] [, interrupt]	Generates two separate sine waves on the left and right channels of the audio output. 'left' and 'right' are the frequencies in Hz, from 0 (silence) to 20000. The frequencies are very accurate (they are based on a crystal oscillator). 'dur' specifies the number of milliseconds that the tone will sound for. MMBasic rounds the time to the next complete cycle of the left frequency (when it is 10Hz or more) so that the tone always finishes with the output at its middle level, without a click. If 'dur' is not specified the tone will continue until explicitly stopped or the program terminates. 'interrupt' is an optional subroutine which will be called when the tone has finished. The frequencies can be changed at any time by issuing a new PLAY TONE command. A paused tone takes the new frequencies and stays paused.

PLAY WAV file\$ [, interrupt] or PLAY FLAC file\$ [, interrupt] or PLAY MP3 file\$ [, interrupt]	Plays an audio file on the audio output. 'file\$' is the file to play, on the SD card or USB flash drive; the appropriate extension (.WAV, .FLAC or .MP3) is added if it has none. 'interrupt' is optional and is the name of a subroutine which will be called when the file has finished playing. For WAV and FLAC files MMBasic adjusts to the file's sample rate, number of bits and number of channels: they can be mono or stereo, with sample rates up to 96kHz (eg, 44.1kHz 16-bit CD quality or 96kHz 24-bit). MP3 files can have the MP3 format's sample rates (up to 48kHz). If 'file\$' is a directory MMBasic finds the files of the type in it and plays them one after another (up to 20 of them); an empty string ("") is the current directory. The files found are listed when the command is used at the command prompt but not in a program. The file names are kept with their full path, so CHDIR can be used while they play. The interrupt is called when the last file has played. PLAY NEXT and PLAY PREVIOUS skip between the files and MM.INFO\$(TRACK) returns the name of the file that is playing.
PLAY MODFILE file\$ [, samplerate] [, interrupt]	Plays a MOD file (music made with a tracker) on the audio output. 'file\$' is the MOD file (.MOD is added if it has no extension); it is read into memory and played in the background. The MOD player supports up to 32 channels and 32 samples, with 16-bit resolution. Without 'interrupt' the MOD file plays continuously, in a loop, until PLAY STOP. With 'interrupt' it plays once through the sequence and then the subroutine is called (in a program only). PLAY MODFILE file\$, interrupt (with no sample rate) is accepted, as on the PicoMite. 'samplerate' is the number of samples a second that the MOD player generates: 8000, 16000, 22050, 44100 (the default) or 48000. A lower rate reduces the processor's work. A WAV file played with PLAY EFFECT while the MOD file plays must have the same sample rate.
PLAY MODSAMPLE samplerate, channel [, volume] [, samplerate]	Plays one of the samples of the MOD file that is playing, at the same time as the MOD file, so that the MOD file's samples can be used as sound effects. 'samplerate' is the sample's number, 1 to 32. Up to four samples can play at once on independent channels: 'channel' is 1 to 4, and a sample started on a channel that is playing replaces the sample there (MM.INFO\$(SAMPLE PLAYING n) says whether channel n is playing). 'volume' is 0 to 64 (the default). 'samplerate' is the rate at which the sample is played, 150 to 500000 (default 16000): changing it changes the pitch and the duration, and it should be the sample's own rate to play it as it was recorded. A MOD file must be playing (otherwise the error is "Samples play over MOD file").
PLAY EFFECT file\$ [, interrupt]	Plays the WAV file 'file\$' (.WAV is added if it has no extension) at the same time as the MOD file that is playing, for sound effects over the music. If a previous effect is still playing it is stopped and the new one starts. The file can be mono or stereo but must have the same sample rate as the MOD file's output (see PLAY MODFILE): 44100Hz unless another rate was given. 'interrupt' is optional and is the name of a subroutine which will be called when the file has finished playing. A MOD file must be playing (otherwise the error is "Effects play over MOD file").
PLAY TTS text\$ [, speed] [, pitch] [, mouth] [, throat] [, interrupt] or	Speaks the English text 'text\$' on the audio output with the voice of SAM (the Software Automatic Mouth). With PHONETIC 'text\$' is SAM's phonemes instead of English: see http://www.retrobits.net/atari/sam.shtml for them and for the voice's parameters. 'speed', 'pitch', 'mouth' and 'throat' set the voice: each is 1 to 255 and they default to 72, 64, 128 and 128. The speech plays in the background and

PLAY TTS PHONETIC text\$ [, speed] [, pitch] [, mouth] [, throat] [, interrupt]	'interrupt' is optional: a subroutine which will be called when it has finished. The speech cannot be paused.
PLAY SOUND soundno, channelno, type [, frequency] [, volume]	Plays up to four sounds at once on each channel of the audio output, made from simple waveforms. 'soundno' is the sound's number, 1 to 4. 'channelno' is the output channel: L (left speaker), R (right speaker) or B (both speakers). 'type' is the waveform: S (sine wave), Q (square wave), T (triangle wave), W (sawtooth), P (periodic noise), N (white noise), U (the wave loaded by PLAY LOAD SOUND) or O (off). 'channelno' and 'type' can also be string expressions (eg, "L"; "M" is the same as "B"). 'frequency' is from 1 to 20000 (Hz) and can have decimals; it must be specified except when 'type' is O. For type N it is the number of periods of 1/44100 second that the output stays at each random value. For type U the 4096 values of the wave are played 'frequency' times a second. 'volume' is optional and must be between 0 and 25. It defaults to 25. Calling PLAY SOUND on a sound that is playing will immediately replace it; individual sounds are turned off using type O. Output can be stopped temporarily using PLAY PAUSE and PLAY RESUME, and PLAY STOP stops all the sounds.
PLAY LOAD SOUND array%()	Loads the wave that PLAY SOUND plays for type U. 'array%()' is an integer array of 1024 elements holding 4096 16-bit values between 0 and 4095, four to an element: MEMORY PACK can make it from a normal 4096 element integer array. The array must be global (or STATIC), not LOCAL, as it is read while sounds play. The wave is forgotten by PLAY STOP and when the program ends. While it is loaded another cannot be loaded (the error is "Already loaded").
PLAY ARRAY left%(), right%(), freq [, start] [, end] [, interrupt]	Plays sound held in two integer arrays. 'left%()' and 'right%()' (which can be the same array) hold the samples for the left and right channels as 16-bit values (-32768 to 32767), four to an element (see MEMORY PACK); they must have the same number of elements. 'freq' is the rate at which the samples are played: 10 to 48000 a second. If 'start' is specified the arrays are played from that sample (0 is the first) and if 'end' is specified until that sample is reached. 'interrupt' (in a program only) is the name of a subroutine which will be called once the last sample has been played. It is important to understand the 'freq' parameter: it is the rate at which each sample is output. So, for example, if the arrays held 10 cycles of a sine wave in 18000 samples then a 'freq' of 1800 would output a 1Hz sine wave for 10 seconds, and a 'freq' of 18000 a 10Hz sine wave for 1 second. The arrays must be global (or STATIC), not LOCAL, as they are read while the sound plays.
PLAY SAMPLE left%(), right%(), freq, attack, decay, sustain, release [, freqR, attackR, decayR, sustainR, releaseR] [, interrupt] or PLAY RELEASE	Plays a looped single cycle waveform under an ADSR envelope, as a synthesiser plays a note. 'left%()' and 'right%()' hold one cycle of the wave for the left and right channels, one 16-bit value (-32768 to 32767) in each element; they must have the same number of elements. The cycle is repeated 'freq' times a second (10 to 48000). 'attack', 'decay' and 'release' are times in milliseconds (0 to 30000) and 'sustain' is a level, 0 to 100 percent of the full volume. The sound rises to the full volume over the attack time, falls to the sustain level over the decay time

	and stays there until PLAY RELEASE, after which it falls to silence over the release time and ends. The optional second set of parameters gives the right channel its own frequency and envelope (otherwise both use the first). 'interrupt' (in a program only) is the name of a subroutine which will be called when the sound has ended. PLAY RELEASE starts the release of the envelopes. The arrays must be global (or STATIC), not LOCAL, as they are read while the sound plays.
PLAY BBC SOUND channel, amplitude, pitch, duration	The BBC Micro's SOUND statement. Queues a note on one of four channels: channels 1 to 3 are square-wave tones and channel 0 is a noise source. The notes queued on a channel play in order, each for its duration, so a tune can be written as a run of PLAY BBC SOUND commands and the program carries on while it plays. 'channel' is the BBC channel word &HSFC. C (bits 0 and 1) is the channel, 0 to 3. Adding &H10 flushes the channel's queue and stops its current note so that the new note is heard at once. Adding &H100, &H200 or &H300 holds the note until that many other channels are waiting with the same value at the head of their queues, and then they all start together, so a three note chord uses &H2xx on all three channels. 'amplitude' is -15 (loudest) to 0 (silent, a rest), or 1 to 16 to use an envelope defined with PLAY BBC ENVELOPE. 'pitch' is 0 to 255 in quarter semitones, 89 being A4 (440Hz) and 53 middle C; on channel 0 only its low three bits are used and they select one of eight noises: 0 to 2 are periodic (a buzzy, pitched tone) at high, medium and low rates, 4 to 6 are white noise at the same three rates, and 3 and 7 are periodic and white clocked by channel 1's tone rather than at a fixed rate, so a note on channel 1 tunes the noise and a pitch envelope on channel 1 bends it (channel 1 need not be audible; an amplitude of 0 still sets its pitch). 'duration' is 0 to 255 in twentieths of a second and 255 means until the channel is flushed or PLAY STOP is used. Each channel queues up to 16 notes. MM.INFO(BBC BUFFER) returns the number of free slots, so a program can keep ahead of the music without ever waiting. When a channel's queue is full the command waits for a slot, as the BBC did, so a loop of PLAY BBC SOUND commands paces itself to the music (Ctrl-C works while it waits but MMBasic interrupts are held off until it returns). The audio output is released automatically when every channel is silent and every queue is empty, as it is when a PLAY TONE ends, and the next PLAY BBC SOUND starts it again. The output is mono and PLAY VOLUME applies. The engine's state (608 bytes) is taken from the MMBasic heap by the first PLAY BBC command and freed when the program's variables are cleared. See the Sound Output chapter for the pitch table and examples.

PLAY BBC ENVELOPE n, T, PI1, PI2, PI3, PN1, PN2, PN3, AA, AD, AS, AR, ALA, ALD	The BBC Micro's ENVELOPE statement. Defines envelope 'n' (1 to 16) for use as the amplitude of a PLAY BBC SOUND. It shapes the volume of the note (attack, decay, sustain and release) and can bend its pitch while it plays. Definitions survive PLAY STOP and are cleared with the program's variables by RUN, NEW and loading a program, so a program defines its envelopes at the start. 'T' is the length of each step in hundredths of a second (bits 0 to 6, 0 is treated as 1); add 128 to stop the pitch envelope repeating. 'PI1', 'PI2' and 'PI3' (-128 to 127) are the pitch change per step, in quarter semitones, in each of three pitch sections and 'PN1', 'PN2' and 'PN3' (0 to 255) are the number of steps in each section. When the three sections are done the pitch returns to the note's own and the sections repeat, unless 128 was added to T. 'AA', 'AD', 'AS' and 'AR' (-127 to 127) are the volume change per step during the attack, decay, sustain and release phases and 'ALA' and 'ALD' (0 to 126) are the levels the attack rises to and the decay falls to (126 is as loud as amplitude -15). The attack runs from 0 to ALA, the decay from there to ALD, the sustain lasts until the note's duration ends and the release then runs until the level reaches 0, at which point the note ends. A release of 0 never ends the note, so it will sound until the channel is flushed.
PLAY PAUSE PLAY RESUME PLAY STOP PLAY CLOSE PLAY VOLUME left [, right] PLAY NEXT PLAY PREVIOUS	PLAY PAUSE will temporarily halt the sound that is playing (a file, a tone or any of the others but TTS speech) and PLAY RESUME will resume it. PLAY STOP will terminate the sound; PLAY CLOSE (the PicoMite's name) does the same. When the program terminates for whatever reason the sound output will also be automatically stopped. Pausing, resuming and stopping are smoothed over a few milliseconds so that they do not click. 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. Either can be omitted to leave that channel as it is: PLAY VOLUME ,50 changes the right channel only, and PLAY VOLUME 50 the left only. A change is smoothed over a few milliseconds so that it does not click. The volume is set to the maximum when a program is run. When a directory of files is playing (PLAY WAV, FLAC or MP3 with a directory) these stop the file that is playing and start the next one or the previous one. At the command prompt a message says when the last (or the first) file is already playing.
POKE BYTE addr%, byte or POKE SHORT addr%, short% or POKE WORD addr%, word% or POKE INTEGER addr%, int% or POKE FLOAT addr%, float! or POKE VAR var, offset, byte or	Will set a byte or a word within the memory space. Where more than one byte is written the address must be exactly divisible by the number of bytes: 2, 4, or 8 otherwise an error will be reported. POKE BYTE will set the byte (i.e. 8 bits) at the memory location 'addr%' to 'byte'. 'addr%' should be an integer. POKE SHORT will set the short integer (i.e. 16 bits) at the memory location 'addr%' to 'short%'. 'addr%' and 'short%' should be integers. POKE WORD will set the word (i.e. 32 bits) at the memory location 'addr%' to 'word%'. 'addr%' and 'word%' should be integers. POKE INTEGER will set the MMBasic integer (i.e. 64 bits) at the memory location 'addr%' to int%. 'addr%' and int%' should be integers. POKE FLOAT will set the word (i.e. 64 bits) at the memory location 'addr%' to 'float!'. 'addr%' should be an integer and 'float!' a floating point number. 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 was removed in V7.00.00: the variables are no longer kept in one table, so an offset into it means nothing. Use POKE VAR to change a variable's value, and PEEK(VARHEADER var) for the address of its header.
POLYGON n, xarray%(), yarray%() [, bordercolour] [, fillcolour] POLYGON n(), xarray%(), yarray%() [, bordercolour()] [, fillcolour()] 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. 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] The elements of 'array n()' define the number of xy-coordinate pairs in each of the polygons. eg, 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: <pre> 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%() </pre>
PORT(start, nbr [,start, nbr]...) = value	Set a number of I/O pins simultaneously (i.e. with one command). 'start' is an I/O pin number and the lowest bit in 'value' (bit 0) will be used to set that pin. Bit 1 will be used to set the pin 'start' plus 1, bit 2 will set pin 'start'+2 and so on for 'nbr' number of bits. I/O pins used must be numbered consecutively and any I/O pin that is invalid or not configured as an output will cause an error. The start/nbr pair can be repeated if an additional group of output pins needed to be added. The pins are numbered on the rear I/O connector, where the power and ground pins (which are invalid) break the runs of consecutive I/O pins: see External I/O Connector in the chapter <i>Hardware Features</i>. For example; PORT(10, 4, 21, 4) = &B10000011 Will set eight I/O pins. Pins 10 and 11 will be set high while 12, 13, 21, 22 and 23 will be set to a low and finally 24 will be set high. This command can be used to conveniently communicate with parallel devices like LCD displays. Up to 32 I/O pins (and therefore bits) can be set by one command, as 'value' is used as a 32-bit number. See the PORT function to simultaneously read from a number of pins.

PRINT expression [[,;]expression] ... etc	Outputs text to the console (the screen and the serial console, as set by OPTION CONSOLE) followed by a carriage return/newline pair. 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 (;) or a comma (,) at the end of the expression list will suppress the output of the carriage return/newline pair 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 or small floating point numbers are automatically printed in scientific number format. The function TAB() can be used to space to a certain column and the STR\$() function can be used to justify or otherwise format strings.
PRINT #nbr, expression [[,;]expression] ... etc	Same as above except that the output is directed to a serial communications port or a file opened for OUTPUT or APPEND with a file number of 'nbr'. See the OPEN command.
PRINT #GPS, string\$	Outputs a NMEA string to the GPS receiver, opened with OPEN ... AS GPS. Only one string can be given. The string must start with a \$ character and end with a * character. The checksum is automatically calculated and appended to the string together with the CR/LF characters.
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. A VT100 terminal emulator on the serial console is sent the escape sequence that moves its cursor to the same character position (the pixel coordinates divided by the size of the font's characters). 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. As with any PRINT, text that reaches the right edge of the screen continues on the next line, and the carriage return and line feed at the end are printed unless the list ends with a semicolon or a comma. 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 The text is drawn in the background colour and its background is not drawn

PULSE pin, width	Will generate a pulse on 'pin' with duration of 'width' ms. 'width' can be a fraction. For example, 0.01 is equal to 10µs and this enables the generation of very narrow pulses. The generated pulse is of the opposite polarity to the state of the I/O pin when the command is executed. For example, if the output is set high the PULSE command will generate a negative going pulse. Notes: • 'pin' must be configured as an output. • For a pulse of less than 3 ms the accuracy is $\pm 1 \mu\text{s}$. • For a pulse of 3 ms or more the accuracy is $\pm 0.5 \text{ ms}$. • A pulse of 3 ms or more will run in the background. Up to five different and concurrent pulses can be running in the background and each can have its time changed by issuing a new PULSE command or it can be terminated by issuing a PULSE command with zero for 'width'.
PWM 1, freq, 1A [, 1B] [, 1C] [, centre] or PWM 2, freq, 2A [, 2B] [, , centre] or PWM channel, STOP	Generate a pulse width modulated (PWM) output for driving analog circuits, sound output, etc. There are a total of five outputs designated as PWM in the table of the external I/O connector in the chapter <i>Hardware Features</i> (they are also used for the SERVO command). Controller 1 can have one, two or three outputs (1A on pin 12, 1B on pin 22 and 1C on pin 29) while controller 2 can have one or two outputs (2A on pin 36 and 2B on pin 31). Both controllers are independent and can be turned on and off and have different frequencies. '1' or '2' is the controller number and 'freq' is the output frequency in Hz (1 to 20000000). 1A, 1B and 1C are the duty cycles for each of the controller 1 outputs while 2A and 2B are the duty cycles for the controller 2 outputs. 1B and 1C can be left out or left empty. The specified I/O pins will be automatically configured as outputs while any others will be unaffected and can be used for other duties. The duty cycle for each output is independent of the others and is specified as a percentage. If it is close to zero the output will be a narrow positive pulse, if 50 a square wave will be generated and if close to 100 it will be a very wide positive pulse. A negative duty cycle (down to -100) inverts the output, giving negative going pulses. 'centre' is 1 for centre aligned outputs: the pulses of the outputs are centred in the period, so that a shorter pulse starts and ends equal times from a longer one. 0 (or omitted) starts all the pulses at the beginning of the period. The frequency is derived from a 240MHz clock (200MHz on a 400MHz processor), so it can be any value of $240,000,000/n$ and the duty cycle and frequency accuracy will depend on the frequency (eg, at 1MHz the duty cycle has 240 steps). The output will run continuously in the background while the program is running and can be stopped using the STOP command. The frequency and duty cycle can be changed at any time (without stopping the output) by issuing a new PWM command. The PWM function will take control of any specified outputs and when stopped the pins will be returned to a high impedance "not configured" state.

RAM SAVE n or RAM OVERWRITE n	Saves the program in program memory (in a program, the program that is running) into RAM slot 'n' (1 to 5), as the PicoMite does, for RAM RUN, RAM CHAIN and RAM LOAD. RAM SAVE gives the error "Already programmed" if the slot holds anything; RAM OVERWRITE replaces what it holds. The RAM slots are the image slots 4 to 8 (RAM slot 1 is image slot 4), held in RAM. They are kept through RUN, CHAIN, NEW and loading a program, and go only with RAM ERASE, an overwrite or a restart, as the PicoMite's slots in its PSRAM do. A program in a RAM slot is held as it was tokenised, and runs where it is.
RAM FILE LOAD n, file\$ [, O] or RAM FILE LOAD n, file\$ [, OVERWRITE]	Loads the program in the file 'file\$' (".BAS" is added if it has no extension) into RAM slot 'n', as RUN loads a program (with its #INCLUDE files and #DEFINE substitutions), but without disturbing the program that is running. So a program can load the parts that it will run with RAM CHAIN. A slot that already holds something gives the error "Already programmed" unless O (or OVERWRITE) is given.
RAM RUN n RAM CHAIN n	Runs the program in RAM slot 'n', as FLASH RUN and FLASH CHAIN run the program in a flash slot: RAM CHAIN keeps the global variables, and can only be used in a program. A slot of 0 runs the program in program memory. MM.INFO(FLASH) returns the number of the RAM slot whose program is running.
RAM LOAD n	Loads the program in RAM slot 'n' into program memory, as FLASH LOAD does. It can only be used at the command prompt.
RAM LIST [n [, ALL]]	RAM LIST reports what each RAM slot holds: "available", the file name of a program, or the size of an image. RAM LIST n lists the program in slot 'n' as it is held, pausing at each screenful unless ALL is given.
RAM ERASE n or RAM ERASE ALL	Empties RAM slot 'n', or all five.
RANDOMIZE [nbr]	Seeds the C library's random number generator with 'nbr' (0 or more) or, if 'nbr' is omitted, from the microsecond timer. This does not affect the RND function: RND uses the hardware random number generator of the STM32H743, which needs no seeding and does not repeat a sequence (see Random Number Generator in the chapter <i>Porting Programs</i>). RANDOMIZE is accepted so that programs written for other versions of MMBasic run. For a repeatable sequence of random numbers use MATH RANDOMIZE and MATH(RAND).
RBOX x, y, w, h [, r] [,c] [,fill]	Draws a box with rounded corners on the screen starting at 'x' and 'y' which is 'w' pixels wide and 'h' pixels high. 'r' is the radius of the corners of the box (0 to 100). 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 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 <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.
REDIM [PRESERVE] array1(dimensions) [, array2(dimensions)...[,arrayn(dimensions]	This resizes the array(s) specified with the dimensions given. If the optional sub-command PRESERVE is specified the existing data will be copied into the new array. The new array can be bigger or smaller than the original. In the case of string arrays, the original LENGTH specified is preserved. Note that for multi-dimensional arrays only the last dimension can be changed if PRESERVE is used. When using PRESERVE there must be enough memory available for both the original array and its changed version to exist simultaneously. The memory allocated to the original array is freed as the command exits.
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. DATA statements are searched in the main program first and then in the library, so when the program's DATA is exhausted (or the program has none) reading continues with any DATA statements in the library. See also DATA and RESTORE.
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.
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. Both must be on the same drive ("Cannot rename across drives").
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 a label or a variable with these values. If 'line' is not specified the counters will be reset to the start of the program, or to the start of the library when RESTORE is executed within library code.
RETURN	Concludes a subroutine called by GOSUB and returns to the statement after the GOSUB. GOSUB/RETURN are legacy commands; defined subroutines (SUB...END SUB) are recommended instead for new programs.
RMDIR dir\$	Remove, or delete, the directory 'dir\$' on the default selected drive.

<pre> <statements> <statements> CASE test-n [[, test-n] ...] <statements> <statements> CASE ELSE <statements> <statements> END SELECT </pre>	• A single expression (i.e. 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" (i.e. 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: <pre> SELECT CASE nbr% CASE 4, 9, 22, 33 TO 88 statements CASE IS < 4, IS > 88, 5 TO 8 statements CASE ELSE statements END SELECT </pre> 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.
<pre> SERVO 1 [, freq], 1A [, 1B [, 1C]] or SERVO 2 [, freq], 2A [, 2B] or SERVO channel, STOP </pre>	Generate a constant stream of positive going pulses for driving a servo. The Colour Maximite 2 has two servo controllers with the first being able to control up to three servos and the second two servos. Both controllers are independent and can be turned on and off and have different frequencies. This command uses the I/O pins that are designated as PWM in the table of the external I/O connector (the two commands are very similar): 1A is pin 12, 1B pin 22, 1C pin 29, 2A pin 36 and 2B pin 31. '1' or '2' is the controller number. 'freq' is the output frequency (between 20Hz and 1000Hz) and is optional. If not specified it will default to 50Hz. (A first value of 20 or more is taken as the frequency.) 1A, 1B and 1C are the pulse widths for each of the controller 1 outputs while 2A and 2B are the pulse widths for the controller 2 outputs. The specified I/O pins will be automatically configured as outputs while any others will be unaffected and can be used for other duties. The pulse width for each output is independent of the others and is specified in milliseconds, which can be a fractional number (i.e. 1.536). For accurate positioning the output resolution is about 0.005 ms. The minimum value is 0.01ms while the maximum is 18.9ms (and less than the period). Most servos will accept a range of 0.8ms to 2.2ms. The output will run continuously in the background while the program is running and can be stopped using the STOP command. The pulse widths of the outputs can be changed at any time (without stopping the output) by issuing a new SERVO command. The SERVO function will take control of any specified outputs and when stopped the pins will be returned to a high impedance "not configured" state.
<pre> SETPIN pin, cfg [, option] </pre>	Will configure an external I/O pin. Refer to the chapter <i>Using the I/O Pins</i> for a general description of the Colour Maximite 2's input/output capabilities, and

	to the table of the external I/O connector in the chapter <i>Hardware Features</i> for the capabilities of each pin. 'pin' is the I/O pin to configure (its number in that table), 'cfg' is the mode that the pin is to be set to and 'option' is an optional parameter. 'cfg' is a keyword and can be any one of the following: OFF Not configured or inactive (0 can also be used). AIN Analog input (i.e. measure the voltage on the input). Only the pins marked Analog Input can be used. 'option' can be used to specify the number of bits in the conversion. Valid values are 8, 10, 12, 14 and 16. The default (if not specified) is 16 bits. The more bits the longer the conversion will take. DIN Digital input. If 'option' is omitted the input will be high impedance. If 'option' is the keyword PULLUP or PULLDOWN an internal resistor will pull the input up to 3.3V or down to zero volts. FIN Frequency input, on one of the counting pins 7, 13, 15 and 24 or the fast counting pin 18 (see below). 'option' can be used to specify the gate time (the length of time used to count the input cycles). It can be any number between 10 ms and 100000 ms. The PIN() function will always return the frequency correctly scaled in Hz regardless of the gate time used. If 'option' is omitted the gate time will be 1 second. PIN Period input, on one of the counting pins 7, 13, 15 and 24. 'option' can be used to specify the number of input cycles to average the period measurement over. It can be any number between 1 and 10000. The PIN() function will always return the average period of one cycle correctly scaled in ms regardless of the number of cycles used for the average. If 'option' is omitted the period of just one cycle will be used. CIN Counting input, on one of the counting pins 7, 13, 15 and 24 (for pin 18 see below). 'option' can be used to specify which edge triggers the count and if any pullup or pulldown is enabled: 1 specifies a rising edge with pulldown, 2 a falling edge with pullup, 3 both edges with no pullup or pulldown, 4 both edges with pulldown and 5 both edges with pullup. If 'option' is omitted a rising edge will trigger the count and a pulldown is enabled. The count is reset to zero whenever the pin is configured as a counting input (even if it is already one). DOUT Digital output. 'option' can be OC in which case the output will be open collector (or more correctly open drain). Previous versions of MMBasic used numbers for 'cfg' (1 AIN, 2 DIN, 3 FIN, 4 PIN, 5 CIN, 6 INTL, 7 INTL, 8 DOUT and 9 OOUT) and the mode OOUT (an open collector output). For backwards compatibility they will still be recognised. Pin 15 shares its hardware interrupt with the PS2 mouse: it cannot be a frequency, period or counting input while the PS2 mouse is open (CONTROLLER MOUSE OPEN 0). On the counting pins 7, 13, 15 and 24 a CSub can have its own routine called on each edge in place of the count (CFuncInt1 to CFuncInt4). The functions PIN() and PORT() can also be used to return the value on one or more output pins. See the function PIN() for reading inputs and the statement
--	---

	PIN()= for setting an output. See the command below if an interrupt is configured.
SETPIN pin, cfg, target [, option]	Will configure 'pin' to generate an interrupt according to 'cfg'. Any I/O pin capable of digital input can be configured to generate an interrupt with a maximum of ten interrupts configured at any one time. 'cfg' is a keyword and can be any one of the following: INTH Interrupt on low to high input INTL Interrupt on high to low input INTB Interrupt on both (i.e. any change to the input) 'target' is a user defined subroutine which will be called when the event happens. Return from the interrupt is via the END SUB or EXIT SUB commands. 'option' can be the keyword PULLUP or PULLDOWN as specified for a normal input pin (SETPIN pin, DIN). If 'option' is omitted the input will be high impedance. This mode also configures the pin as a digital input so the value of the pin can always be retrieved using the function PIN(). Configuring the pin again (eg, SETPIN pin, OFF) removes the interrupt. Refer to the chapter <i>Using the I/O Pins</i> for a general description.
SETPIN 18, FIN [, option] or SETPIN 18, CIN [, target]	Pin 18 is the fast counting pin: it is counted by a hardware counter and has been tested up to 40MHz. It cannot be used for period measurement (PIN). It can also be used as an ordinary digital input or output. With FIN 'option' is the gate time, as above (10 ms to 100000 ms, 1 second if it is omitted), and PIN(18) returns the frequency in Hz. With CIN every rising edge is counted and PIN(18) returns the count. 'target' is an optional user defined subroutine which will be called each time the count passes a multiple of 4294967296 (2^{32}, when the 32-bit hardware counter wraps round). Return from the interrupt is via the END SUB or EXIT SUB commands. The count is reset to zero whenever the pin is configured as a counting input.
SETTICK period, target [, nbr] SETTICK PAUSE, target [, nbr] or SETTICK RESUME, target [, nbr]	This will setup a periodic interrupt (or "tick"). Four tick timers are available ('nbr' is 1, 2, 3 or 4). 'nbr' is optional and if not specified timer number 1 will be used. 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 (i.e. SETTICK 0, 0, 3 will disable tick timer number 3). Pause or resume the specified timer. When paused the interrupt is delayed but the current count is maintained.

SETTICK FAST frequency, target	An interrupt faster than the tick timers above allow (their shortest period is 1 ms), from a hardware timer. 'frequency' is in Hz (it has been tested up to 50 kHz) and 'target' is the interrupt subroutine. SETTICK FAST 0, 0 stops it, as does the end of the program. Ticks that arrive while the subroutine is running, or during a statement that takes longer than the time between them, are counted, and the subroutine is called once for each. If the subroutine takes longer than the time between ticks the program will do nothing else.
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. 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 (strings only). - bit2: 0 (default) normal sort - 1 empty strings go to the end of the array 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: <pre>SORT array() , , , 50</pre> Example: The array city\$() might contain the names of world cities and can be easily sorted into increasing alphabetical order with the command: SORT city\$() The SORT command will work with strings, floats and integers however the array to be sorted must be single dimensioned. Often data is held in multiple arrays, for example, the name of each city might be held in the array city\$(), the population held in the array pop%() and the size of the city held in area!(). The same index would refer to the name, population and the area of the city. Sorting and accessing this data is a little more complex but it can be done relatively easily using an optional parameter to the sort command as follows: <pre>SORT array(), indexarray%()</pre> indexarray%() must be a single dimension integer array of the same size as the array being sorted. Following the sort indexarray%() will contain the corresponding index to the original data before it was sorted. (anything previously in indexarray%() will be overwritten). To access the sorted data you would first copy the array holding the main key to a temporary array and sort that while specifying indexarray%(). After the sort indexarray%() can be used to index the original arrays. For example:

	<pre> DIM city\$(100),pop%(100),area!(100),sindex%(100),t\$(100) FOR i = 0 to 100 t\$(i) = city\$(i) ` temporary copy of the keys NEXT i SORT t\$(), sindex%()         ` sort the temporary array, FOR i = 0 to 100 k = sindex%(i) ` index to the original array     PRINT city\$(k),pop%(k),area!(k) ` print in sorted order NEXT i </pre>
SPI OPEN speed, mode [, bits] or SPI READ nbr, data [, data...] or SPI WRITE nbr, data [, data...] or SPI CLOSE	Communications via the first SPI channel, on the rear I/O connector. See Appendix D for the details. ‘speed’ is the clock speed in Hz, ‘mode’ the transmission format (0 to 3) and ‘bits’ the number of bits in each item (4 to 32, 8 if it is left out). ‘nbr’ is the number of data items to send or receive. The data to send (WRITE) or the destination for the received data (READ) may be supplied in any one of three forms: • A list of individual items. The number of items must equal ‘nbr’. For WRITE each item is a constant, variable or expression; for READ each item is a numeric variable to receive one value. Array elements such as a(3) are allowed. • An array specified with empty brackets, eg, array(). It may be a floating point or integer array and must be large enough to supply (WRITE) or hold (READ) ‘nbr’ elements, which are taken from / stored into consecutive elements starting at the first. • A string variable. For WRITE the first ‘nbr’ characters are sent (the string must be at least that long); for READ the received bytes are stored as sequential characters and the string is set to that length. SPI WRITE discards any data returned by the slave and SPI READ sends zeros while reading the data from the slave. SPI CLOSE returns the I/O pins to a "not configured" state.
SPI2	The same set of commands as for SPI (above) but applying to the second SPI channel. It shares its pins with COM2: the two cannot be open at the same time.
SPRITE	The SPRITE commands are used to manipulate small graphic images on the screen. These are useful when writing games. See Appendix F for how sprites, their layers and their collisions work, and the SPRITE() functions for interrogating them. SPRITE and BLIT are one command: every SPRITE command can also be written as BLIT and every BLIT command as SPRITE. The sprites are the buffers #1 to #64 that BLIT READ also uses, and the # before a sprite number is optional. WRITE and LOAD differ with the name used, as on the PicoMite (see BLIT). A sprite can be up to the size of the screen. Its image is held in full colour whatever the colour depth, so a sprite can be loaded before the MODE command and it is kept through a change of mode. A MODE command clears the screen, and with it the sprites that were showing: they are no longer displayed and can be shown again.

	A sprite's transparent pixels are not drawn, so that the background shows through them. A pixel is transparent when the colour depth shows it as the transparent colour, which is black unless SPRITE SET TRANSPARENT has set another. So in every colour depth but the 32-bit one the colours too dark to be told from black are transparent too: in the 8-bit depth, say, any colour with red and green below 32 and blue below 64.
SPRITE CLOSE [#]n	Closes sprite 'n' and releases its memory resources allowing the sprite number to be re-used. If the sprite is displayed it is first removed and the stored background replaced. 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, removes the static objects (see SPRITE STATIC) and makes black the transparent colour again. The screen is not changed. RUN does the same.
SPRITE COPY [#]n, [#]m, nbr	Makes a copy of sprite 'n' to 'nbr' of new sprites starting at number 'm'. Copied sprites share the same loaded image as the original to save memory. The new sprite numbers must not be in use, and a copy cannot itself be copied. While copies exist the original cannot be closed or read into.
SPRITE HIDE [#]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").
SPRITE HIDE ALL	Hides all the sprites allowing the background to be manipulated. SPRITE RESTORE shows them again. While the sprites are hidden SPRITE SHOW, SHOW SAFE, HIDE, HIDE SAFE, HIDE ALL, SWAP, MOVE, SCROLL, SCROLLR and CLOSE must not be used (they give the error "Sprites are hidden").
SPRITE HIDE SAFE [#]n	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 is less performant than the simple version and should only be used if 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 F for how to use the SPRITE() functions to interrogate details of what has collided.
SPRITE LOAD fname\$ [,start_sprite_number] [,palette]	Loads the file 'fname\$' which must be formatted as an original Colour Maximite sprite file. If an extension is not specified ".SPR" will be added to the file name. The first line of the file holds the width of the sprites and their number, and optionally their height (the width if it is omitted), separated by commas: eg, 16, 8 or 16, 8, 24. Each sprite follows as 'height' lines of 'width' characters, a character for each pixel: space = transparent 0 = black, 1 = blue, 2 = green, 3 = cyan, 4 = red, 5 = magenta, 6 = yellow, 7 = white 8 = light grey, 9 = grey, A = orange, B = pink, C = gold, D = salmon, E = beige, F = brown Black is the transparent colour (unless SPRITE SET TRANSPARENT has set another), so a 0 is not drawn when the sprite is shown, as on the PicoMite.

With 'palette' 1 the characters 0 to F are the PicoMite's 16 colours in order instead (its palette 1):

0 = black, 1 = blue, 2 = myrtle, 3 = cobalt, 4 = midgreen, 5 = cerulean, 6 = green, 7 = cyan

8 = red, 9 = magenta, A = rust, B = fuchsia, C = brown, D = lilac, E = yellow, F = white

'palette' 0, the list above it, is the default.

Any other character is also transparent, a short line is filled out with spaces, and lines starting with an apostrophe (') are comments.

The sprites are loaded as the sprite numbers from 'start_sprite_number' (1 if it is omitted) on. 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 sprite numbers used by the files do not overlap.

A sprite number already in use is loaded again, as SPRITE READ reads into one: a sprite that is showing takes the new image and stays where it is, if it is the same size ("Sprite n is showing, at another size" otherwise), and a copy, or a sprite that has copies, cannot be loaded into ("Sprite n is a copy", "Sprite n has copies"). The file's first line must give a width of 1 to 255 and at least one sprite ("Invalid sprite file").

Written as BLIT LOAD the command loads a BMP file instead (see BLIT).

SPRITE LOADARRAY [#]n,
w, h, array%()

Creates the sprite 'n' with width 'w' and height 'h' by reading $w \times h$ RGB888 colour values from 'array%()' (an integer or floating point array). The values must be stored in order of columns across and then rows down starting at the top left. Black pixels are transparent.

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, or if sprite 'n' is already in use.

SPRITE LOADBMP [#]n,
fname\$ [,x] [,y] [,w] [,h]

Loads the BMP image 'fname\$' as sprite number 'n', which must not be in use. If an extension is not specified ".BMP" will be added to the file name. The file can be of any of the types that LOAD BMP reads.

'x', 'y', 'w' and 'h' load a part of the image: 'w' by 'h' pixels with its top left at 'x', 'y' in the image (0, 0 is the image's top left). 'x' and 'y' default to 0, and 'w' and 'h' to the rest of the image. The part must be within the image (the error "Coordinates") and no bigger than the screen ("Image too large").

Every pixel of the image is loaded as it is, so its black pixels are transparent when the sprite is shown (see SPRITE). BLIT LOAD and BLIT LOADBMP are other names for this command.

SPRITE LOADPNG [#]n,
fname\$ [,t] [,cutoff] [,x] [,y]
[,w] [,h]

Loads the PNG image 'fname\$' as sprite number 'n', which must not be in use. If an extension is not specified ".PNG" will be added to the file name. The file can be of any of the types that LOAD PNG reads, and it is decoded a line at a time.

A pixel whose alpha value (its opacity) is above 'cutoff' (1 to 254, default 30) is solid. The others take the colour 't', one of the PicoMite's 16 colours (0 to 15, default 0, black: the transparent colour, so that they are not drawn), as on the PicoMite. The colours are numbered from 0:

Black, Blue, Myrtle, Cobalt, Midgreen, Cerulean, Green, Cyan, Red, Magenta, Rust, Fuchsia, Brown, Lilac, Yellow and White

A negative 't' (-1 to -15) instead gives the colour '-t' to the solid pixels that the colour depth shows as black, which would be transparent, so that the image's black can be drawn; the others are then transparent.

A file without transparency (RGB or grey, or indexed with no transparency chunk) has every pixel solid.

	‘x’, ‘y’, ‘w’ and ‘h’ load a part of the image, as with SPRITE LOADBMP. The part must be within the image (the error "Coordinates") and no bigger than the screen ("Image too large").
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 NOSTINTERRUPT	Disables the static object interrupt (see SPRITE STINTERRUPT).
SPRITE READ [#]n, x, y, w, h [,page]	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’ specifies which page is to be read to create the sprite. The default is the current write page. Set ‘page’ to FRAMEBUFFER to read from the framebuffer (see the FRAMEBUFFER command).
SPRITE RESTORE	Restores all the sprites hidden by SPRITE HIDE ALL. Any position changes previously requested using SPRITE NEXT for sprites on layers other than 0 will be actioned by the RESTORE and collision detection will be run.
SPRITE SCROLL x, y [,colour]	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/2-1 and MM.HRES, ‘y’ any number between -MM.VRES/2-1 and MM.VRES. Sprites on any layer other than zero will remain fixed in position on the screen (or move to the position set by SPRITE NEXT). By default the scroll wraps the image round. If ‘colour’ is specified the colour will replace the area behind the scrolled image. If ‘colour’ is set to -1 the scrolled area will be left untouched.
SPRITE SCROLLR x, y, w, h, delta_x, delta_y [,colour]	Scrolls the region of the screen defined by the top left coordinates ‘x’ and ‘y’ and width and height ‘w’ and ‘h’ by ‘delta_x’ pixels to the right and ‘delta_y’ pixels up (each less than the width or height of the region). By default the scroll wraps the background round. If ‘colour’ is specified the colour will replace the area behind the scrolled image (-1 wraps it as the default does). Sprites on any layer other than zero will remain fixed in position on the screen (or move to the position set by SPRITE NEXT). 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 SET TRANSPARENT colour	Sets the transparent colour of the sprites to ‘colour’, one of the PicoMite's 16 colours (0 to 15, default 0, black). They are numbered from 0: Black, Blue, Myrtle, Cobalt, Midgreen, Cerulean, Green, Cyan, Red, Magenta, Rust, Fuchsia, Brown, Lilac, Yellow and White A pixel of a sprite is then transparent when the colour depth shows it as that colour. It applies to every sprite from its next draw on, and to SPRITE(B). SPRITE CLOSE ALL, and RUN, make black the transparent colour again.

SPRITE SHOW [#]n, x, y, layer [,orientation]	Displays sprite 'n' on the screen with the top left at coordinates 'x', 'y' (part of the sprite can be off the screen: 'x' can be from 1 - its width to MM.HRES-1 and 'y' from 1 - its height to MM.VRES-1). 'layer' is a number from 0 to 10. Sprites will only collide with other sprites on the same layer, layer zero, the static objects or 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 (and layer). 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) 4 to 7 - the same with every pixel drawn, the transparent ones too, as on the PicoMite The orientation stays with the sprite: SPRITE MOVE, SCROLL, SCROLLR and RESTORE draw it the same way.
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. ie, 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 STATIC [#]n, x, y, w, h or SPRITE STATIC [#]n, OFF	Declares static object 'n' (1 to 64): an invisible rectangle 'w' by 'h' pixels with its top left at 'x', 'y', as on the PicoMite. OFF removes it. A static object has no pixels and is never drawn: it marks where the scenery that the program has drawn (a wall, a platform) is solid. Each collision check tests the sprites displayed against the static objects as well as the other sprites. A sprite that overlaps static object 'n' (touching is not overlapping) has &H80 + 'n' in its list of collisions (see SPRITE(C ...)), and the collision interrupt is called. Unlike a collision between sprites, which is reported once when it begins, a static object is reported at each check while the sprite overlaps it. SPRITE SCROLL moves the static objects with the background, wrapping them round as it does, and SPRITE SCROLLR moves those whose centre is within its region. See also SPRITE STINTERRUPT and SPRITE(ST ...), and SPRITE(B) for a sprite's collision with the drawn background itself.
SPRITE STATIC CLEAR	Removes every static object. SPRITE CLOSE ALL does too.
SPRITE STINTERRUPT sub	Specifies the name of the subroutine that will be called when a sprite overlaps a static object (see SPRITE STATIC). SPRITE(ST, COLLISION) and SPRITE(ST, OBJECT) give the sprite and the static object of the last such collision.
SPRITE SWAP [#]n1, [#]n2 [,orientation]	Replaces the sprite 'n1' with the sprite 'n2'. The sprites must have the same width and height, 'n1' must be displayed and 'n2' must not be, 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 SPRITE WRITE [#]n, x, y [,orientation]	<u>IN THE 12-BIT COLOUR DEPTH ONLY</u> Changes all pixels with a non-zero transparency in the stored sprite 'n' to the new level 'transparency', which can be between 1 and 15 (15 is solid). Overwrites the display with the contents of sprite buffer 'n' with the top left at coordinates 'x', 'y'. 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 (0 to 7) defaults to 4 and specifies how the stored image data is changed as it is written out. It is the sum of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels Written as BLIT WRITE the default is 0, every pixel copied (see BLIT). A sprite that is displayed can be written too: the copy is drawn, and the sprite shown, its orientation and its stored background are left as they were.
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).
STRUCT	This command allows the manipulation of structures (also known as a User-Defined Types) which allow related variables of different types to be grouped under a single name. This is described in full in the file <i>MMBasic_Structures_Manual.pdf</i> included in the firmware download zip file.
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. i.e. arg3(). The type of the argument can be specified by using a type suffix (i.e. arg1\$) or by specifying the type using AS <type> (i.e. arg1 AS 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. The argument can be prefixed with BYVAL which will prevent this mechanism and cause only the value to be used. Alternatively, the prefix BYREF instructs MMBasic that a reference is required and an error will be generated if that cannot be done. 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. 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: <code>MySub a1, a2</code> 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. Brackets around the argument list in both the caller and the definition are optional.
SYNC time% [,period] or SYNC	The SYNC command allows the user to implement very precisely timed repeated actions (1-2 microseconds accuracy). To enable this the command is first called with the parameter time%. This sets up a repeating clock for time% microseconds. The optional parameter 'period' modifies the time and can be "U" for microseconds, "M" for milliseconds or "S" for seconds. Once the clock is set up the program is synchronised to it using the SYNC command without parameters. This waits for the clock period to expire. For periods below 2 ms this is non-interruptible. Above 2 ms the program will respond to Ctrl-C but not any MMBasic interrupts. SYNC 0 stops the clock. Typical use is to set the clock outside of a loop and then at the top of the loop call the SYNC command without parameters. This means the contents of the loop will be executed exactly once for each clock period set. For example, the following would drive a servo with the required precise 50Hz timing: <pre> SETPIN 7, DOUT SYNC 20, M DO SYNC PULSE 7, n LOOP </pre>
TEMPR START pin [,precision] [,timeout]	This command can be used to start a conversion running on a DS18B20 temperature sensor connected to 'pin'. Normally the TEMPR() function alone is sufficient to make a temperature measurement so usage of this command is optional. For more detail see the section <i>Measuring Temperature</i>. This command will start the measurement on the temperature sensor. The program can then attend to other duties while the measurement is running and later use the TEMPR() function to get the reading. If the TEMPR() function is used before the conversion time has completed the function will wait for the remaining conversion time before returning the value. Any number of these conversions (on different pins) can be started and be running simultaneously. 'precision' is the resolution of the measurement and is optional. It is a number between 0 and 3 meaning: 0 = 0.5°C resolution, 100 ms conversion time. 1 = 0.25°C resolution, 200 ms conversion time (this is the default). 2 = 0.125°C resolution, 400 ms conversion time. 3 = 0.0625°C resolution, 800 ms conversion time. The optional timeout parameter (100 to 2000 ms) overrides the conversion times above to allow for slow devices. 'pin' can be any I/O pin, or pin 42 for a sensor fitted on the motherboard (see the chapter <i>Hardware Features</i>).

TEXT x, y, string\$ [,alignment\$] [, font] [, scale] [, c] [, bc]	Displays a string on the screen 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. For example. "CM" will centre the text vertically and horizontally. The 'alignment\$' string can be a constant (eg, "CM") or it can be a string variable. For backwards compatibility with earlier versions of MMBasic the string can also be unquoted (eg, CM). 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 (i.e. upside down), 'U' the text will be rotated counter clockwise by 90° and 'D' the text will be rotated clockwise by 90° '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. 'bc' can be -1, and then the background of the characters is not drawn (see Transparent Text in the chapter <i>Graphics Functions</i>). See the chapter <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.
TIME\$ = "HH:MM:SS" or TIME\$ = "HH:MM" or TIME\$ = "HH" or TIME\$ = sec	Sets the time of the internal clock. MM and SS are optional and will default to zero if not specified. For example TIME\$ = "14:30" will set the clock to 14:30 with zero seconds. TIME\$ = sec, where 'sec' is a number rather than a string, adds 'sec' seconds to the current time (or subtracts them when it is negative), which makes it easy to fine tune the clock; the date changes too if the adjustment crosses midnight. The clock is the battery backed real time clock of the processor, or the DS3231 of a Generation 2 with OPTION DS3231 ON, so the time is kept while the power is off (see Backup Battery in the chapter <i>Hardware Features</i>). The time is "00:00:00" when the clock is first powered.
TILEMAP	TILEMAP is a powerful command for writing games. See the separate TILEMAP_User_Manual.pdf for more details
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 number. 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 (a line of an #INCLUDE file or of the library as [file:n]). This is useful in debugging programs. TRACE LIST will list the last 'nn' lines executed (0 to 127, or 127 if 'nn' is omitted) 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 screen 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 firmware implements a full turtle graphics engine. See Appendix G.
TYPE struct-name struct-member1, struct-member2 ... END TYPE	Define a structure (also known as a User-Defined Type) which allows related variables of different types to be grouped under a single name. This is described in full in the file <i>MMBasic_Structures_Manual.pdf</i> included in the firmware download zip file.
UPDATE FIRMWARE	Restarts the ARM Cortex-M7 processor in its firmware update mode, as setting the BOOT CONFIG switch on the Waveshare CPU board to SYSTEM or moving the jumper to PGM does, so that new firmware can be loaded without opening the case (see Appendix K). Cycle the power after the update to return the processor to normal operation. Cycling the power also cancels the update mode if the command was used by mistake.
VAR SAVE var [, var]... or VAR RESTORE or VAR CLEAR	VAR SAVE will save one or more variables to non-volatile memory where they can be restored later (normally after a power interruption). 'var' can be any number of numeric or string variables and/or arrays. Arrays are specified by using empty brackets. For example: var() The VAR SAVE command can be used repeatedly. Variables that had been previously saved will be updated with their new value and any new variables (not previously saved) will be added to the saved list for later restoration. VAR RESTORE will retrieve the previously saved variables and insert them (and their values) into the variable table. VAR CLEAR will erase all saved variables. This command is normally used to save calibration data, options, and other data which does not change often but needs to be retained across a power interruption. Normally the VAR RESTORE command is placed at the start of the program so that previously saved variables are restored and immediately available to the program when it starts. Notes: • The storage space available to this command is 4KB of RAM kept alive by the battery on the motherboard (see Saved Variables in the chapter <i>Using MMBasic</i>). It can be written to an unlimited number of times. The error is "Not enough memory" when it is full. • Using VAR RESTORE without a previous save will have no effect and will not generate an error. • If, when using RESTORE, a variable with the same name already exists its value will be overwritten. • Saved arrays must be declared (using DIM) before they can be restored. • Only global variables can be saved: not LOCAL or STATIC ones, constants or structures. • Be aware that string arrays can rapidly use up all the memory allocated to this command. The LENGTH qualifier can be used when a string array is declared to reduce the size of the array (see the DIM command). This is not needed for ordinary string variables.

	The saved variables will be automatically cleared by a firmware upgrade, by the NEW command and when MMBasic is reset to its original configuration. Running or loading a program does not clear them.
WATCHDOG timeout or WATCHDOG OFF	Starts the watchdog timer, which restarts the Colour Maximite 2 when it times out. This can be used to recover from some event that stopped the running program (such as an endless loop, or an error that returned to the command prompt), which can be important in an unattended control situation. ‘timeout’ is the time in milliseconds (1 or more) before a restart is forced. This command should be placed in strategic locations in the running BASIC program to constantly reset the watchdog timer (to ‘timeout’) 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 Colour Maximite 2 will be restarted, "Watchdog timeout" will be displayed and the automatic variable MM.WATCHDOG will be set to true (ie, 1) indicating that an error occurred. This happens even when MMBasic is at the command prompt. OPTION AUTORUN must be set for the program to run again. On a normal startup MM.WATCHDOG will be set to false (ie, 0). WATCHDOG OFF disables the watchdog timer (this is the default on a reset or power up). The timer is also turned off when the break key (normally Ctrl-C) is used on the console to interrupt a running program.
WII	WII is another name for CONTROLLER: for example WII NUNCHUK OPEN is the same as CONTROLLER NUNCHUK OPEN. See CONTROLLER. The PicoMite's spellings WII OPEN (for a Classic) and WII NUNCHUCK OPEN are not accepted: use WII CLASSIC OPEN and WII NUNCHUK OPEN, and the CLASSIC() and NUNCHUK() functions in place of the PicoMite's DEVICE(WII ...) and DEVICE(NUNCHUCK ...).
WS2812 type, pin, nbr, colours%() or WS2812 type, pin, 1, colour%	This command will drive one or more WS2812 LED chips connected to 'pin', which is made a digital output if it is not one already. 'type' is a single character specifying the type of chip being driven: - O = original WS2812 - B = WS2812B - S = SK6812 - W = SK6812W (RGBW) ‘nbr’ is the number of LEDs in the chain (1 to 512). The 'colours%()' array should be an integer array, specified with empty brackets, with at least as many elements as the number of LEDs to be driven. For the first three variants each element in the array should contain the colour in the normal RGB888 format (i.e. 0 to &HFFFFFF). For type W it holds the white level above the red: &HWWRRGGBB (0 to &HFFFFFFF). If only one LED is connected then a single integer value (‘colour%’) is used instead of the array, with the same range. See the chapter Special Devices. BITBANG WS2812, the older name, is still accepted.
XMODEM SEND file\$ [, port] or XMODEM RECEIVE file\$ [, port]	Transfers a file to or from a remote computer using the XModem protocol. XMODEM SEND sends the file ‘file\$’ on the SD card or USB flash drive to the remote computer and XMODEM RECEIVE accepts a file sent by the remote computer and saves it as ‘file\$’, overwriting a file of that name. Any file can be transferred, not only programs (a program received is run with RUN). SEND and RECEIVE can be abbreviated to S and R.

	The transfer is made over the serial console, which must be enabled (see OPTION CONSOLE), unless 'port' is given: 1, 2 or 3 for the serial port COM1, COM2 or COM3, which must have been opened (see OPEN) with a suitable baud rate. The XModem protocol requires a cooperating software program running on the remote computer. It has been tested on Tera Term running on Windows and it is recommended that this be used. After running the XMODEM command in MMBasic select: File -> Transfer -> XMODEM -> Receive/Send from the Tera Term menu to start the transfer. Download Tera Term from http://ttssh2.sourceforge.jp/ If the remote computer does not respond within about 60 seconds the command returns to the MMBasic prompt with the error "Remote did not respond". The break key is disabled while the transfer runs. XModem sends the data in blocks of 128 bytes, so a file received is padded at its end to a multiple of 128 bytes (with the character &H1A). The ZMODEM command does the same job faster, keeps a file's exact length and recovers from transmission errors. The YModem protocol is not available.
ZMODEM SEND [file\$] or ZMODEM RECEIVE [file\$]	Transfers files to or from a remote computer over the serial console using the ZModem protocol. It does the job of XMODEM but is much faster, and it recovers from transmission errors: every block of data is checked and if any is lost or damaged the receiver asks for it again from that point, so a poor connection slows a transfer down rather than stopping it. SEND and RECEIVE can be abbreviated to S and R. The serial console must be enabled (see OPTION CONSOLE). ZMODEM SEND 'file\$' sends the file 'file\$' from the SD card or USB flash drive; the remote computer is given its name without the drive or directory. ZMODEM SEND with no file name sends the file of the current program (see MM.INFO\$(CURRENT)). ZMODEM RECEIVE 'file\$' saves the file sent by the remote computer as 'file\$', overwriting any file of that name; the name sent by the remote computer is ignored. If the remote computer sends several files only the first is received, unless 'file\$' is "". ZMODEM RECEIVE "", or ZMODEM RECEIVE with no file name, will accept any number of files and save each one in the current directory under the name given by the remote computer (without any directory), overwriting any file of that name. A directory can be put in front of the *, for example ZMODEM RECEIVE "C:/downloads/*". In Tera Term select all the files to be sent in the ZMODEM Send dialog; with lrzsz list them all, for example sz *.bas. When the transfer is complete the number of files and bytes received is reported. If an error stops the transfer part way through, the files received before it are complete and the error message says how many there were. ZMODEM SEND and RECEIVE with no file name can only be used at the command prompt; with a file name they can also be used in a program. Programs are transferred as files: a program received is run with RUN. The PicoMite's ZMODEM CRUNCH is not available. The ZModem protocol requires a cooperating program on the remote computer. On Windows Tera Term is recommended: after running the ZMODEM command select File -> Transfer -> ZMODEM -> Send or Receive from the Tera Term menu. On Linux and macOS use sz (to send) and rz (to receive) from the lrzsz package, run with the serial port as their input and output, for example from a terminal program such as minicom. If the remote computer does not start the transfer within 60 seconds the command returns to the MMBasic prompt with the error "Sender did not respond" (or "Receiver did not respond"). If the remote computer cancels the transfer the error is "Cancelled by remote" (or "Receiver cancelled"). The

	break key is disabled while the transfer runs. The speed of the transfer is set by the console's baud rate (see OPTION BAUDRATE).
--	---

Functions

Note that the functions related to communications functions (I²C, 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' (i.e. any negative sign is removed and a positive number is returned).
ACOS(number)	Returns the inverse cosine of the argument 'number' in radians (or in degrees after OPTION ANGLE DEGREES). 'number' must be -1 to 1.
ASC(string\$)	Returns the ASCII code (i.e. byte value) for the first letter in 'string\$'.
ASIN(number)	Returns the inverse sine value of the argument 'number' in radians (or in degrees after OPTION ANGLE DEGREES). 'number' must be -1 to 1.
ATN(number)	Returns the arctangent of the argument 'number' in radians (or in degrees after OPTION ANGLE DEGREES).
ATAN2(y, x)	Returns the arc tangent of the two numbers x and y as an angle expressed in radians (or in degrees after OPTION ANGLE DEGREES). 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.
BASE\$(base, number [, chars])	Returns a string giving the value of 'number' in the number base 'base'. 'base' can be from 2 to 36 with the digits above nine represented by the letters A to Z. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s). BIN\$(), OCT\$() and HEX\$() are shortcuts for BASE\$() with a base of 2, 8 and 16 respectively. Use BASE\$() for any other base, for example BASE\$(12, 100) will return 84.
BAUDRATE(comm [, timeout])	Returns the baud rate of any data received on the serial communications port 'comm', which is 1 for COM1: or 2 for COM2: (the ports on the rear I/O connector). The port must not be open. This will sample the port over the period of 'timeout' seconds (1 to 10). 'timeout' will default to one second if not specified. The result is the nearest standard baud rate, from 110 to 460800. Returns zero if no activity on the port within the timeout period.
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 number converted to a 4 byte string DOUBLE double precision floating point number converted to a 8 byte string By default the string contains the number in little-endian format (i.e. 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 (i.e. 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
BIT(var%, bitno)	Returns the value of a specific bit (0-63) in an integer variable (0 or 1). See also the BIT command
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. Requesting a dimension that the array does not have will generate an error, as will using BOUND() on a variable that is not an array. Note that zero is a valid upper limit: an array declared with a single element under OPTION BASE 0 (for example DIM a(0)) has an upper limit of zero. For example: DIM myarray(44,45) BOUND(myarray(),2) will return 45
BYTE(var\$, byteno)	Returns the integer value of a specific byte in a string (0-255). This is the equivalent of ASC(MID\$(var\$,byteno,1)) but operates much faster. See also the BYTE command
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 (i.e. byte value) 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().
CLASSIC(funcnt [, channel])	Returns data from a Wii Classic controller opened with CONTROLLER CLASSIC OPEN. ‘channel’ is optional and is the I²C channel for the controller (1 to 3, defaults to 3, the front panel Wii connector). ‘funcnt’ is a 1 or 2 letter code indicating the information to return as follows: - LX the position of the analog left joystick x axis - LY the position of the analog left joystick y axis - RX the position of the analog right joystick x axis - RY the position of the analog right joystick y axis - L the position of the analog left button - R the position of the analog right button - B a bitmap of the state of all the buttons. A bit will be set to 1 if the button is pressed. - T the ID code of the controller, &HA4200101 The positions are from 0 to 255: the left joystick has 64 steps and the right joystick and the L and R buttons 32. The button bitmap is as follows: - BIT 0 Button R - BIT 1 Button start - BIT 2 Button home - BIT 3 Button select - BIT 4 Button L - BIT 5 Button down cursor - BIT 6 Button right cursor - BIT 7 Button up cursor - BIT 8 Button left cursor - BIT 9 Button ZR - BIT 10 Button x - BIT 11 Button a - BIT 12 Button y - BIT 13 Button b - BIT 14 Button ZL These bit positions are also used in the interrupt bitmask specified in the CONTROLLER CLASSIC OPEN command.
CLICK(DOWN) or CLICK(UP) or CLICK(X)	Reports the clicks of the left mouse button, for the GUI controls (see the chapter GUI Controls): a USB mouse's as soon as it is plugged in, or a mouse's set with OPTION MOUSE. A click made with GUI CLICK counts as the button too. TOUCH() is another name for CLICK(), the PicoMite's. CLICK(DOWN) returns 1 (true) if the left mouse button is currently down. CLICK(UP) returns 1 (true) if it is not.

or CLICK(Y) or CLICK(REF) or CLICK(LASTREF) or CLICK(LASTX) or CLICK(LASTY)	CLICK(X) and CLICK(Y) return the coordinates of the mouse while the button is down, or -1 if it is not. CLICK(LASTX) and CLICK(LASTY) return the coordinates of the last location that was clicked on. CLICK(REF) returns the reference number of the GUI control that is currently being clicked on, or zero if none. CLICK(LASTREF) returns the reference number of the last control that was clicked on. LASTX, LASTY, REF and LASTREF are kept by the GUI controls: they change only when OPTION MAXCTRLS is set and the program has used the GUI commands (for example, created a control or set GUI INTERRUPT). The controls are described in the document GUI Controls and Programming.pdf (see Appendix O).
COS(number)	Returns the cosine of the argument 'number' in radians (or in degrees after OPTION ANGLE DEGREES).
CTRLVAL([#]ref)	Returns the value of the GUI control number 'ref': a number for most controls (for a list box the index of the item selected, for a slider its value), and a string for a text box, a format box, a display box or a caption. The GUI controls need OPTION MAXCTRLS, and are clicked with a mouse or with GUI CLICK. They are described in the chapter GUI Controls and in the document GUI Controls and Programming.pdf (see Appendix O).
CWD\$	Returns the current working directory, with its drive, as a string. A: is the SD card and C: the USB flash drive. The format is: A:/dir1/dir2. See also MM.INFO(DIRECTORY), which always ends with a '/'.
DATE\$	Returns the current date based on MMBasic'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), from - 2147483648 to 2147483647 (13 December 1901 to 19 January 2038). The format of the returned string is "dd-mm-yyyy hh:mm:ss". Use the text NOW to get the current datetime string, ie, DATETIME\$(NOW)
DAY\$(date\$)	Returns the day of the week for a given date as a string. For example, "Monday", "Tuesday" etc. 'date\$' is a string and its format can be DD-MM-YY or DD-MM-YYYY or YYYY-MM-DD. You can also use NOW to get the day for the current date, eg, PRINT DAY\$(NOW)
DEG(radians)	Converts 'radians' to degrees.
DEVICE(GAMEPAD channel, funct)	Returns data from a USB gamepad: a PS3 or PS4 controller, a Nintendo Switch Pro Controller, an Xbox 360, Xbox One or Xbox Series controller, or a generic gamepad that the firmware supports or GAMEPAD CONFIGURE describes (see USB Gamepads in the chapter Keyboard, Mouse and Gamepad). 'channel' is 1 to 4: the first gamepad is normally channel 3 and the second channel 4. The values are 0 when there is no gamepad on the channel. 'funct' is a 1 to 3 letter code indicating the information to return as follows: LX the position of the analog left joystick x axis (0 to 255)

LY	the position of the analog left joystick y axis (0 to 255)
RX	the position of the analog right joystick x axis (0 to 255)
RY	the position of the analog right joystick y axis (0 to 255)
GX	the reading from the X axis gyro (a PS4 controller only)
GY	the reading from the Y axis gyro (a PS4 controller only)
GZ	the reading from the Z axis gyro (a PS4 controller only)
AX	the reading from the X axis accelerometer (a PS4 controller only)
AY	the reading from the Y axis accelerometer (a PS4 controller only)
AZ	the reading from the Z axis accelerometer (a PS4 controller only)
L	the position of the analog left button (0 to 255)
R	the position of the analog right button (0 to 255)
B	a bitmap of the state of all the buttons. A bit will be set to 1 if the button is pressed.
H	the direction of the d-pad: 0 for up, then clockwise in steps of 45 degrees to 7 for up and left, or 255 when it is not pressed (where supported: other gamepads always give 255)
T	the type of the controller: 128 for a PS4, 129 for a PS3, 130 for a generic gamepad, 131 for an Xbox controller and 133 for a Switch Pro Controller (0 if there is none)
RAW	the last report that the gamepad sent, as a string of its bytes (for working out a GAMEPAD CONFIGURE with GAMEPAD MONITOR SILENT)

The button bitmap is as follows:

BIT 0	Button R/R1
BIT 1	Button start/options
BIT 2	Button home
BIT 3	Button select/share
BIT 4	Button L/L1
BIT 5	Button down cursor
BIT 6	Button right cursor
BIT 7	Button up cursor
BIT 8	Button left cursor
BIT 9	Right shoulder button 2/R2
BIT 10	Button x/triangle
BIT 11	Button a/circle
BIT 12	Button y/square
BIT 13	Button b/cross
BIT 14	Left shoulder button 2/L2
BIT 15	Touchpad

DEVICE(MOUSE channel, funct)	Returns data from the mouse in use: a USB mouse, or else a mouse opened with CONTROLLER MOUSE OPEN (a PS2 mouse or a Hobbytronic interface). This is the PicoMite's function; the MOUSE() function reads the same mice. 'channel' is 1 to 4 and is normally 2, the channel of a USB mouse, but the function reads the mouse in use whatever the channel. It returns 0 if there is no mouse. 'funct' is a 1 letter code indicating the information to return as follows: - X the X coordinate (0 to MM.HRES-1) - Y the Y coordinate (0 to MM.VRES-1) - L the state of the left mouse button - R the state of the right mouse button - M the state of the middle mouse button (wheel click) - D 1 if there has been a double click of the left mouse button (cleared when it is read) - W the count of the scroll wheel (as MOUSE(S)) - B the state of all buttons as a bitmap: bit 0 is the left button, bit 1 the right and bit 2 the middle; with a USB mouse the other bits are the mouse's further buttons in its own order - T 3 if a PS2 mouse has a scroll wheel, otherwise 0
DIR\$(fspec, type) or DIR\$(fspec) or DIR\$()	Will search the current drive (the SD card or the USB flash drive) 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. Note that the wildcard *.* does not find files or folders without an extension. 'fspec' can include a path (eg, "C:/data/*.TXT") and then that directory is searched instead of the current one. 'type' is the type of entry to return and can be one of: - ALL Search for all 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. i.e. 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: <pre>f\$ = DIR\$ ("*", FILE) DO WHILE f\$ <> "" PRINT f\$ f\$ = DIR\$ () LOOP</pre>
DISTANCE(trigger, echo) or DISTANCE(trig-echo)	Measure the distance to a target using the HC-SR04 ultrasonic distance sensor. Four pin sensors have separate trigger and echo connections. 'trigger' is the I/O pin connected to the "trig" input of the sensor and 'echo' is the pin connected to the "echo" output of the sensor. Three pin sensors have a combined trigger and echo connection and in that case you only need to specify one I/O pin to interface to the sensor. Note that the HC-SR04 is a 5V device so the I/O pins used with it should be 5V tolerant (the pins marked 5V under I/O Pin Capabilities in the chapter Hardware Features).

	The I/O pins are automatically configured by this function and multiple sensors can be used on different I/O pins. The value returned is the distance in centimetres to the target or -1 if no target was detected or -2 if there was an error (i.e. sensor not connected).
DRAW3D(XMIN n) or DRAW3D(XMAX n) or DRAW3D(YMIN n) or DRAW3D(YMAX n) or DRAW3D(X n) or DRAW3D(Y n) or DRAW3D(Z n) or DRAW3D(DISTANCE n)	Returns information about the 3D object 'n' (see the DRAW3D command). There is no comma between the word and the object's number. XMIN, XMAX, YMIN and YMAX return the boundaries on the screen of the object as it was last drawn. These can be used to define an area to be erased before re-drawing it. X, Y and Z return the position at which the object was last drawn, and DISTANCE the average distance of its faces from the camera. The document <i>The 3D engine.pdf</i> in the Colour Maximite 2 firmware download has the details (see Appendix O).
EOF([#]fnbr)	Will return true if the file previously opened on the SD card or USB flash drive for INPUT with the file number '#fnbr' is positioned at the end of the file. For a serial communications port opened as 'fnbr' 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', i.e. e^x where x is 'number'.
FIELD\$(string1, nbr [, string2 [, string3]])	Returns a particular field in a string with the fields separated by delimiters. Note that a space character cannot be used as a delimiter. '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); if it is omitted the delimiter is a comma. The space character may not be used as a delimiter. '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). The

	quote characters are returned as part of the field. Spaces at the start and end of the field are removed. 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 (i.e. does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .
FLAG(n%)	Returns the value (0 or 1) of the bit n% (0-63) in the system flag register. See also MM.FLAGS and the FLAG and FLAGS commands
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 and defaults to 6. 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 E is used the exponential output will use an uppercase E. If the format specification is not specified "%g" is assumed. Only one format specification is allowed, and the result can be at most 255 characters long ("String too long"). 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
GETSCANLINE	Reports the line that is currently being drawn on the VGA monitor. Using this to time updates to the screen can avoid timing effects caused by updates while the screen is being updated. The first visible line returns 0 and the last MM.VRES - 1; a larger value is in the frame blanking period. In the modes whose lines are drawn twice (3, 5, 6, 7, 12, 13 and 17) the count is in the mode's own lines.
GPS()	The GPS functions return data from a GPS receiver on a serial port opened with OPEN comspec\$ AS GPS (an error is given if there is none). The values are updated in the background as the receiver sends them. The function GPS(VALID) should be checked before any of these functions are used to ensure that the returned value is valid.
GPS(ALTITUDE)	Returns the current altitude in metres (if the receiver sends the GGA sentence).
GPS(DATE)	Returns the normal date string corrected for local time, e.g. "12-01-2020".
GPS(DOP)	Returns the DOP (horizontal dilution of precision) value (if the receiver sends the GGA sentence).
GPS(FIX)	Returns the quality of the fix (if the receiver sends the GGA sentence): 0 for no fix, 1 for a GPS fix, 2 for a differential GPS fix, or another value as the receiver gives it.
GPS(GEOID)	Returns the geoid-ellipsoid separation in metres (if the receiver sends the GGA sentence).
GPS(LATITUDE)	Returns the latitude in degrees as a floating point number, values are negative for South of the equator.
GPS(LONGITUDE)	Returns the longitude in degrees as a floating point number, values are negative for West of the meridian.
GPS(SATELLITES)	Returns the number of satellites in use (if the receiver sends the GGA sentence).
GPS(SPEED)	Returns the ground speed in knots as a floating point number.
GPS(TIME)	Returns the normal time string corrected for local time, e.g. "12:09:33".
GPS(TRACK)	Returns the track over the ground (degrees true) as a floating point number.
GPS(VALID)	Returns 0 for invalid data and 1 for valid data. It returns 0 when nothing has been received from the receiver for two seconds.
HEX\$(number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
INKEY\$	Checks the console input 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 the input buffer is empty this function will immediately return with an empty string (i.e. "").

INPUT\$(nbr, [#]fnbr)	Will return a string composed of 'nbr' characters read from a file or serial communications port opened as 'fnbr'. This function will return as many characters as are in the file or receive buffer up to 'nbr'. If there are no characters available 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\$ [,size])	Returns the position at which 'string-pattern\$' occurs in 'string-searched\$', beginning at 'start-position'. If 'start-position' is not provided it will default to 1. 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. If the optional parameter "size" is specified the "string-pattern" is treated as a regular expression. See <i>Appendix E</i> for the details.
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%(). Note that many JSON data sets are quite large and may be too big to parse with the memory available. 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 USB 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 I. 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. LGETBYTE treats the LONGSTRING as a plain array of bytes and therefore honours OPTION BASE when interpreting 'n': with OPTION BASE 0 the first byte is number 0, while with OPTION BASE 1 the first byte is number 1. Note that LINSTR and LGETSTR\$ always work with 1-based positions (the first character is position 1) irrespective of OPTION BASE. Consequently, when OPTION BASE 0 is in effect, a position returned by LINSTR (or passed to LGETSTR\$) refers to the byte that LGETBYTE would read at that position minus one. Subtract 1 from a LINSTR result before passing it to LGETBYTE under OPTION BASE 0 (under OPTION BASE 1 the values line up directly).
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. The 'start' position is always measured from 1 (the first character is position 1) regardless of the OPTION BASE setting, and so matches the position returned by LINSTR. Unlike LGETBYTE, LGETSTR\$ does not honour OPTION BASE.
LINPUT(array%(),fnbr,nbr)	This reads 'nbr' bytes from a file open as 'fnbr' into the LONGSTRING 'array%()'. The function returns the number of bytes actually read so if you are near the end of the file the number may be less than the number requested. It only works with file I/O and not serial or console I/O
LINSTR(array%(), search\$ [,start] [,size])	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 ' array%()' but the optional third parameter allows the start position of the search to be specified. If the optional parameter 'size' is specified the 'search\$' is treated as a regular expression. See <i>Appendix E</i> for the details. The position returned is always 1-based (the first character is position 1) and is not affected by OPTION BASE. It can be passed directly to LGETSTR\$, but note that LGETBYTE indexes bytes according to OPTION BASE (see LGETBYTE).
LLEN(array%())	Returns the length of a long string stored in 'array%()'.

LOC([#]fnbr)	For a file on the SD card or USB flash drive opened as 'fnbr' 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 SD card or USB flash drive opened as 'fnbr' 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 so this function can be used to avoid this. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.
MAP(n)	Returns the colour that is drawn with the palette value 'n' in the 8-bit or 4-bit colour depth: 'n' is 0 to 255 (RGB332) in the 8-bit depth and 0 to 15 (RGB121) in the 4-bit depth. For example, in the 8-bit depth MAP(255) always returns &HE0E0C0, which the firmware will then convert to 255. This is a way of selecting the colour that will use the place in the colour map that you want. It is not a window into the map: MAP(n) = rgb% changes the colour shown, not the value returned. See 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 y, x)	Returns the arc tangent of y / x, using the signs of both to find the quadrant as ATAN2(y, x) does, but as an angle from 0 to 2*PI radians (0 to 360 degrees under OPTION ANGLE DEGREES) instead of -PI to PI.
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
MATH(CRCn data [,length] [,polynome] [,startmask] [,endmask] [,reverseIn] [,reverseOut])	Calculates the CRC to n bits (8, 12, 16, 32) of "data". "data" 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 result is reversed, if reverseOut says so, before it is XORed with endmask. The defaults for polynomes are CRC8=&H07, CRC12=&H80D, CRC16=&H1021,

	crc32=&H04C11DB7. polynome, startmask and endmask must fit in n bits. An array must be one-dimensional and each element must be a byte value (0 to 255). "Length" is 1 to 65535 and must not be more than the number of elements or characters in "data". eg, for crc16_CCITT use MATH(CRC16 array(), n,, &HFFFF)
MATH(RAND)	Returns a random number $0.0 \leq n < 1.0$ using the Mersenne Twister algorithm. If not seeded with MATH RANDOMIZE the first use in a program seeds it from the hardware random number generator.
Simple Statistics	
MATH(CHI a())	Returns the Pearson's chi-squared value of the two dimensional array a(). The array must have at least 2 rows and 2 columns, and no more than 50 degrees of freedom $((\text{rows} - 1) * (\text{columns} - 1))$.
MATH(CHI_p a())	Returns the associated probability in % of the Pearson's chi-squared value of the two dimensional array a(). The array is as for MATH(CHI).
MATH(CROSSING array() [,level] [,direction] [,confirm])	This returns the array index of the first sample at which the values in the array cross the "level" in the direction specified, where the crossing is sustained for "confirm" consecutive samples. level defaults to 0. direction defaults to 1 (valid values are -1 or 1). confirm defaults to 1; values greater than 1 reject brief noise spikes by requiring that many consecutive samples to remain past the level. Returns -1 if no qualifying crossing is found; a crossing within the final (confirm-1) samples cannot be confirmed and is not reported.
MATH(CORREL a(), b())	Returns the Pearson's correlation coefficient between arrays a() and b(), which must have the same number of elements
MATH(MAX a() [,index%])	Returns the maximum of all values in the a() array, a() can have any number of dimensions. If the integer variable is specified then it will be updated with the index of the maximum value in the array. This is only available on one-dimensional arrays
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() [,index%])	Returns the minimum of all values in the a() array, a() can have any number of dimensions. If the integer variable is specified then it will be updated with the index of the minimum value in the array. This is only available on one-dimensional arrays.
MATH(SD a())	Returns the sample 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()) MATH(DOTPRODUCT v1(), v2()) Matrix Arithmetic MATH(M_DETERMINANT array!())	Returns the magnitude of the vector v(). The vector can have any number of elements Returns the dot product of two vectors v1() and v2(). The vectors can have any number of elements but must have the same cardinality Returns the determinant of the array. The array must be square.
Creation complex% = MATH(C_CPLX r!, i!) complex% = MATH(C_POLAR radius!, angle!) Floating returns real! = MATH(C_REAL complex%) imag! = MATH(C_IMAG complex%) arg! = MATH(C_CARG complex%) mod! = MATH(C_MOD complex%) phase! = MATH(C_PHASE complex%) Unary functions complex1% = MATH(C_CONJ complex2%) complex1% = MATH(C_SIN complex2%) complex1% = MATH(C_COS complex2%) complex1% = MATH(C_TAN complex2%) complex1% = MATH(C_ASIN complex2%) complex1% = MATH(C_ACOS complex2%) complex1% = MATH(C_ATAN complex2%) complex1% = MATH(C_SINH complex2%) complex1% = MATH(C_COSH complex2%) complex1% = MATH(C_TANH complex2%) complex1% = MATH(C_ASINH complex2%) complex1% = MATH(C_ACOSH complex2%) complex1% = MATH(C_ATANH complex2%) complex1% = MATH(C_PROJ complex2%) complex1% = MATH(C_EXP complex2%) complex1% = MATH(C_LOG complex2%) complex1% = MATH(C_SQRT complex2%) complex1% = MATH(C_ABS complex2%) Basic Arithmetic complex1% = MATH(C_ADD complex2%,complex3%) complex1% = MATH(C_SUB complex2%,complex3%) complex1% = MATH(C_MUL complex2%,complex3%) complex1% = MATH(C_DIV complex2%,complex3%) complex1% = MATH(C_POW complex2%,complex3%)	MMBasic supports a full range of functions to allow the manipulation of complex numbers. In this implementation complex numbers have a 32-bit real and 32-bit imaginary part and to make this work in MMBasic, it uses integers (64-bit) to hold these. The angle given to C_POLAR and the phase returned by C_PHASE respect OPTION ANGLE; C_CARG returns the same angle in radians (C_ARG is another name for C_CARG). C_ABS returns the modulus as a complex number (with an imaginary part of zero), and C_MOD as a floating point number.

MATH(PID channel, setpoint!, measurement!)	This function must be called in the PID callback subroutine for the 'channel' specified and returns the output of the controller function. The 'setpoint' value is the desired state that the controller is trying to achieve. The 'measurement' is the current value of the real world. https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=17263 For an example of setting up and running a PID controller
MATH(BASE64 ENCODE/DECODE in\$/in(), out\$/out())	Returns the length of out\$/out(). This base64 encodes or decodes the data in 'in' and puts the result in 'out'. Where arrays are used as the output they must be big enough relative to the input and the direction. Encoding increases the length by 4/3 and decoding decreases it by 3/4. A string output is limited to 255 characters.
MAX(arg1 [, arg2 [, ...]]) or MIN(arg1 [, arg2 [, ...]])	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.
MID\$(string\$, start) or MID\$(string\$, start, nbr)	Returns a substring of 'string\$' beginning at 'start' and continuing for 'nbr' characters. The first character in the string is number 1. If 'nbr' is omitted the returned string will extend to the end of 'string\$'
MOUSE(funcnt [, channel])	Returns data from a mouse opened with CONTROLLER MOUSE OPEN, or from a USB mouse. 'channel' is optional: 0 for a PS2 mouse, or 1 to 3 for a mouse on the Hobbytronic interface on that I²C channel. It defaults to 2, which is also the channel of a USB mouse. An error is given if no mouse is open on the channel. 'funcnt' is a 1 letter code indicating the information to return as follows: - X the X coordinate (0 to MM.HRES-1) - Y the Y coordinate (0 to MM.VRES-1) - L the state of the left mouse button (1 if pressed) - R the state of the right mouse button (1 if pressed) - W the state of the scroll wheel mouse button (1 if pressed) - D 1 if there has been a double click of the left mouse button. The two clicks must occur between 100 and 500 milliseconds apart (within 500 milliseconds with a USB mouse). The report is then valid for 500mSec before it times out or until it is read. - S the current count of the scroll wheel - Z the count of the scroll wheel, which is then reset to 0 - T 3 if a PS2 mouse has a scroll wheel, otherwise 0 A USB mouse can also be read with the PicoMite's DEVICE(MOUSE 2, funcnt) function.
MSGBOX(message\$, button\$ [, button\$ [, button\$ [, button\$]]])	Displays a message box in the centre of the screen with one to four buttons and waits for one of them to be clicked with the mouse. It returns the number of the button that was clicked (1 for the first). A ~ character in 'message\$' starts a new line. The message box is one of the GUI controls, which need OPTION MAXCTRLS and OPTION MOUSE. They are described in the chapter GUI Controls and in the document GUI Controls and Programming.pdf (see Appendix O).
NUNCHUK(funcnt [, channel])	Returns data from a Wii Nunchuk opened with CONTROLLER NUNCHUK OPEN. 'channel' is optional and is the I²C channel for the controller (1 to 3, defaults to 3, the front panel Wii connector).

	'funct' is a 1, 2 or 3 letter code indicating the information to return as follows: JX the value of the joystick X-axis (0 to 255) JY the value of the joystick Y-axis (0 to 255) AX the value of the x acceleration (0 to 1023) AY the value of the y acceleration (0 to 1023) AZ the value of the z acceleration (0 to 1023) Z the state of the Z button (1 if pressed) C the state of the C button (1 if pressed) T the ID code of the Wii device: &HA4200000 for the original Nunchuk, &HA4200101 for a Classic and &HA4200402 for a Balance Board JXL the calibrated X value of the joystick in the far left position JXC the calibrated X value of the joystick in the centre position JXR the calibrated X value of the joystick in the far right position JYT the calibrated Y value of the joystick in the top position JYC the calibrated Y value of the joystick in the centre position JYB the calibrated Y value of the joystick in the bottom position AX0 the calibrated zero gravity value of the x-axis accelerometer AX1 the calibrated one gravity value of the x-axis accelerometer AY0 the calibrated zero gravity value of the y-axis accelerometer AY1 the calibrated one gravity value of the y-axis accelerometer AZ0 the calibrated zero gravity value of the z-axis accelerometer AZ1 the calibrated one gravity value of the z-axis accelerometer
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).
PEEK(BYTE addr%) or PEEK(SHORT addr%) or PEEK(WORD addr%) or PEEK(INTEGER addr%) or PEEK(FLOAT addr%) or PEEK(VARADDR var) or PEEK(VARHEADER var) or	Will return a byte or a word within the processor's virtual memory space. NB: the address is rounded down to a multiple of the size of the data requested (2 for SHORT, 4 for WORD and 8 for INTEGER and FLOAT). An address outside the processor's memory gives the error "Address". BYTE will return the byte (8-bits) located at 'addr%' SHORT will return the short integer (16-bits) located at 'addr%' WORD will return the word (32-bits) located at 'addr%' INTEGER will return the integer (64-bits) located at 'addr%' FLOAT will return the floating point number (64-bits) located at 'addr%' VARADDR will return the address (32-bits) of the variable 'var' in memory. An array is specified as var(). VARHEADER will return the address (32-bits) of the variable 'var' header in memory. An array is specified as var(). This will be the address of the first byte of the variable name.

PEEK(CFUNADDR cfun) or PEEK(VAR var, ±offset) or PEEK(VARTBL, ±offset) or PEEK(PROGMEM, ±offset)	CFUNADDR will return the address (32-bits) of the CFunction 'cfun' in memory. This address can be passed to another CFunction which can then call it to perform some common process. VAR, will return a byte in the memory allocated to 'var'. An array is specified as var(). VARTBL was removed in V7.00.00: the variables are no longer kept in one table. Use PEEK(VARHEADER var) for the address of a variable's header and PEEK(VAR var, ±offset) to read its bytes. 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.
PIN(pin)	Returns the value on the external I/O 'pin'. Zero means digital low, 1 means digital high and for analogue inputs it will return the measured voltage as a floating point number. Frequency inputs will return the frequency in Hz. A period input will return the period in milliseconds while a count input will return the count since reset (counting is done on the positive rising edge). The count input can be reset to zero by resetting the pin to counting input (even if it is already so configured). When a pin is configured as an output this function will return the value of the output setting (ie, high or low). Also see the SETPIN and PIN() = commands. Refer to the chapter <i>Using the I/O pins</i> for a general description of the PicoMite's input/output capabilities.
PIN(function\$)	Returns the value of a special function. 'function\$' is a string (a string variable or constant), for example PRINT PIN("BAT"). It can be one of: "BAT": the voltage of the backup battery. "TEMP": the temperature of the ARM Cortex-M7 processor's core, in °C. "DAC1" and "DAC2": the output voltage of DAC1 or DAC2 (on the audio output). "SREF": the stored calibrated value of the internal reference voltage, measured with a supply of exactly 3.3V. This is programmed into the chip during production. "IREF": the measured value of the internal reference voltage. The actual value of VREF+ can be calculated as $3.3 * \text{PIN}(\text{"SREF"}) / \text{PIN}(\text{"IREF"})$, and this can be used to set OPTION VCC.

PIXEL(x, y [,page])	Returns the colour of a pixel on the screen. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. The optional 'page' specifies which page is to be read; the default is the current write page. Set 'page' to FRAMEBUFFER to read from the framebuffer (see the FRAMEBUFFER command). The colour is returned as a 24-bit RGB number: in the 12-bit colour depth with its transparency as well, and in the 4-bit colour depth as one of the PicoMite's 16 colours (which MAP does not change). The colour is the one the colour depth holds, which is not the value RGB() gave when the pixel was drawn. RGB() includes the transparency (15, opaque) in every colour depth but the 32-bit one, and each depth keeps only part of the 24 bits. For example RGB(200, 100, 50) reads back as &HCD6531 in the 16-bit colour depth, &HFCC6633 in the 12-bit one and &HFF4000 in the 4-bit one. In the 8-bit colour depth the colour comes back through the palette (see MAP and OPTION VGA OUTPUT), so with the default palette RGB(red) reads back as &HE00000. Only the 32-bit colour depth returns the colour exactly. So, to test what is on the screen, do not compare PIXEL(with RGB(). Compare it with a colour read by PIXEL(from a pixel drawn in that colour: drawing the value PIXEL(returned and reading it again gives the same value in every colour depth. Black reads as 0 in every colour depth but the 12-bit one, where opaque black is &HF000000. See the chapter <i>Graphics Functions</i> for a definition of the colours and graphics coordinates.
PORT(start, nbr [,start, nbr]...)	Returns the value of a number of I/O pins in one operation. 'start' is an I/O pin number and its value will be returned as bit 0. 'start'+1 will be returned as bit 1, 'start'+2 will be returned as bit 2, and so on for 'nbr' number of bits. I/O pins used must be numbered consecutively and any I/O pin that is invalid or not configured as a digital input or output (DIN, INTL, INTB or DOUT) will cause the error "Invalid input pin". An output pin returns its state. The start/nbr pair can be repeated if additional groups of input pins need to be added: the first pair gives the lowest bits. This can be used to conveniently communicate with parallel devices like memory chips. See the PORT command to simultaneously output to a number of pins.
PULSIN(pin, polarity) or PULSIN(pin, polarity, t1) or PULSIN(pin, polarity, t1, t2)	Measures the width of an input pulse from 1µs to 10 seconds with 1µs resolution. 'pin' is the I/O pin to use for the measurement, it must be previously configured as a digital input. 'polarity' is the type of pulse to measure, if zero the function will return the width of the next negative pulse, if non zero it will measure the next positive pulse. 't1' is the timeout applied while waiting for the pulse to arrive, 't2' is the timeout used while measuring the pulse. Both are in microseconds (µs), 5 to 10000000 (10 seconds), and are optional. If 't2' is omitted the value of 't1' will be used for both timeouts. If both 't1' and 't2' are omitted then the timeouts will be set at 100000 (i.e. 100ms). This function returns the width of the pulse in microseconds (µs) or -1 if a timeout has occurred. The measurement is accurate to ±1µs. Note that this function will cause the running program to pause while the measurement is made and interrupts will be ignored during this period.
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, grey (or gray), lightgrey (or lightgray), orange, pink, gold, salmon, beige, lilac, fuchsia, rust, cerulean, midgreen, cobalt and myrtle. For example, RGB(red) or RGB(cyan). There is also one special colour, notblack: in any colour depth this is the darkest colour that is not treated as black (transparent) by the graphics commands that support transparency. 'trans' is the level of transparency: 0 (transparent) to 15 (opaque, the default) in the 12-bit colour depth, and 0 to 255 in the 32-bit colour depth, where a colour holds 16 levels of transparency as in the 12-bit depth (the top 4 bits of 'trans' are kept: 255 is opaque). In the other colour depths it gives the error "Transparency not valid for this mode".
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 random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The Colour Maximite 2 uses the hardware random number generator in its STM32H743 processor to deliver true random numbers, so they need no seeding and the RANDOMIZE command does not affect them. For a sequence that can be repeated use MATH(RAND) with MATH RANDOMIZE.
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 (or in degrees after OPTION ANGLE DEGREES).
SPACE\$(number)	Returns a string of blank spaces 'number' characters long.
SPI (data) or SPI2 (data)	Send and receive data using an SPI channel. A single SPI transaction will send data while simultaneously receiving data from the slave. 'data' is the data to send and the function will return the data received during the transaction. 'data' can be an integer or a floating point variable or a constant.
SPRITE() SPRITE(A, [#]n) SPRITE(B, [#]n)	The SPRITE functions return information regarding sprites, which are small graphic images on the screen. These are useful when writing games. See also the SPRITE commands and Appendix F. In each of them the # before a sprite number is optional. Returns the address in memory of the image of sprite 'n', or 0 if the sprite is not loaded. The pixels are held row after row, four bytes each whatever the colour depth: the red, green and blue in bits 0-23 and the transparency (0 to 15, 15 solid) in bits 24-27. Returns how the background under displayed sprite 'n' meets the sprite, pixel by pixel, as on the PicoMite: what is drawn on the screen, scenery for example, against the sprite's shape. The background is the one stored when the sprite was last drawn (any sprites under it included). The shape is each row's and each column's span of solid pixels, from the first to the last, so a gap inside a span counts as the sprite.

	It returns: 0 - nothing is there: every background pixel is the transparent colour (in the 12-bit colour depth, or transparent) 1 - something is within the sprite's rectangle but outside its shape 2 - something is within its shape It returns 0 if the sprite is not displayed.
SPRITE(B, [#]n, side)	Returns more about the background found by the last SPRITE(B, n), for 'side': 0 - how far in from the sprite's left edge the background reaches (the offset of the furthest pixel + 1) 1 - how far in from its right edge 2 - how far in from its top edge 3 - how far in from its bottom edge 4 - how deep the background within the shape is, from the left ends of the row spans 5 - from their right ends 6 - from the top ends of the column spans 7 - from their bottom ends Sides 0 to 3 are 0 if no background is there, or if it reaches all the way across. Of 4 and 5 only the smaller, the depth from the nearer end, is given (the other is 0), and so of 6 and 7; both are 0 if they are equal. So, after a step to the right, SPRITE(B, n, 5) > 0 says that the sprite has run into something on its right.
SPRITE(C, [#]n)	Returns the number of new collisions (contacts that began) of sprite 'n' found when it was last moved, or -1 if there were none. Up to seven are recorded. If n = 0 it returns the number of sprites that had a new collision following the last SPRITE SCROLL, SCROLLR, MOVE or RESTORE command. A sprite that stays in contact with another is not counted again: SPRITE(T, n) returns all the sprites it is touching.
SPRITE(C, [#]n, m)	Returns the number of the sprite which caused the 'm'th collision of sprite 'n' ('m' from 1 to SPRITE(C, n)). If n = 0 then returns the sprite number of the 'm'th sprite that had a new collision following a SPRITE SCROLL, SCROLLR, MOVE or RESTORE command. If the collision was with a static object (see SPRITE STATIC) the return value is &H80 + its number, and if it was with the edge of the screen it is: &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 displayed).
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. Returns 0 if the sprite is not displayed.
SPRITE(H, [#]n)	Returns the height of sprite 'n'. This function is active whether or not the sprite is currently displayed (active). Returns -1 if the sprite is not loaded.
SPRITE(L, [#]n)	Returns the layer number of sprite 'n' if it is displayed, -1 if it is not.
SPRITE(N)	Returns the number of displayed (active) sprites.

SPRITE(N, n)	Returns the number of displayed (active) sprites on layer 'n' (0 to 10).
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, SCROLLR, MOVE or RESTORE command and the SPRITE(C...) function should be used to find how many and which sprites collided. Returns -1 if the last command that checks for collisions found none.
SPRITE(ST, [#]n, X Y W H A)	Returns the left (X), top (Y), width (W) or height (H) of static object 'n' (1 to 64), or -1 if it is not declared, or (A) 1 if it is declared and 0 if not. See SPRITE STATIC.
SPRITE(ST, COLLISION)	Returns the number of the sprite in the last collision with a static object, or -1 if there has been none.
SPRITE(ST, OBJECT)	Returns the number of the static object in the last collision with one, or -1 if there has been none.
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. Returns 0 if the sprite is not displayed.
SPRITE(V, [#]s1, [#]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 (and it increases clockwise). 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). Returns -1 if the sprite is not loaded.
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.
SQR(number)	Returns the square root of the argument 'number'.
STR\$(number) or STR\$(number, m) or STR\$(number, m, n) or STR\$(number, m, n, c\$)	Returns a string in the decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative 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"
STR2BIN(type, string\$ [,BIG])	Returns a number equal to the binary representation in 'string\$'. 'type' can be: - INT64 converts an 8 byte string representing a signed 64-bit integer to an integer - UINT64 converts an 8 byte string representing an unsigned 64-bit integer to an integer - INT32 converts a 4 byte string representing a signed 32-bit integer to an integer - UINT32 converts a 4 byte string representing an unsigned 32-bit integer to an integer - INT16 converts a 2 byte string representing a signed 16-bit integer to an integer - UINT16 converts a 2 byte string representing an unsigned 16-bit integer to an integer - INT8 converts a 1 byte string representing a signed 8-bit integer to an integer - UINT8 converts a 1 byte string representing an unsigned 8-bit integer to an integer - SINGLE converts a 4 byte string representing a single precision float to a float - DOUBLE converts an 8 byte string representing a double precision float to a float By default the string must contain the number in little-endian format (i.e. 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 (i.e. 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 ("String length") will be generated if the string is the incorrect length for the conversion requested. See also the function BIN2STR\$
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 an integer or float number in the range of 0 to 255.
STRUCT(arg, ...)	Returns data from a structure (also known as a User-Defined Type) which allows related variables of different types to be grouped under a single name. This is described in full in the file <i>MMBasic_Structures_Manual.pdf</i> included in the firmware download zip file.

TAB(number)	Outputs spaces until the column indicated by 'number' has been reached on the console output.
TAN(number)	Returns the tangent of the argument 'number' in radians (or in degrees after OPTION ANGLE DEGREES).
TEMPR(pin [,timeout])	Return the temperature measured by a DS18B20 temperature sensor connected to 'pin' (which does not have to be configured). The returned value is degrees C with a default resolution of 0.25°C. If there is an error during the measurement the returned value will be 1000. The time required for the overall measurement is 200ms and interrupts will be ignored during this period. The optional parameter timeout (100 to 2000 milliseconds) can be used to override the default (200mSec) to allow for slow devices. Alternatively the TEMPR START command can be used to start the measurement and your program can do other things while the conversion is progressing. When this function is called the value will then be returned instantly assuming the conversion period has expired. If it has not, this function will wait out the remainder of the conversion time before returning the value. The DS18B20 can be powered separately by a 3V to 5V supply or it can operate on parasitic power from the I/O pin. See the chapter <i>Special Device Support</i> for more details.
TILEMAP(arg, ...)	Returns information about a tile map created with the TILEMAP command, or about its sprites: TILEMAP(TILE id, x, y) returns the tile at a position, TILEMAP(COLLISION id, x, y, w, h [, mask]) the first tile in an area, TILEMAP(ATTR id, tile) a tile's attributes, TILEMAP(VIEWX id) and TILEMAP(VIEWY id) the position of the view, TILEMAP(COLS id) and TILEMAP(ROWS id) the size of the map, and TILEMAP(SPRITE X id), SPRITE Y, SPRITE TILE, SPRITE W, SPRITE H and TILEMAP(SPRITE HIT id1, id2) its sprites. This is described in full in the file <i>TILEMAP_User_Manual.pdf</i> (see Appendix O).
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 OPTION MILLISECOND ON has been used this function will return the time including milliseconds as a decimal fraction of the seconds. For example: "14:35:06.239". To set the current time use the command TIME\$ = .
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 (or set it to any value) by using TIMER as a command. If not specifically reset it will continue to count up forever (it counts microseconds in a 64-bit number and therefore will only roll over after more than 500,000 years).
TRIM\$(source\$ [,mask\$] [,where/where\$])	This function can remove characters from the beginning or end of a string or both ‘source\$’ is the input string ‘mask\$’ is a string containing a list of character to be removed. If omitted it defaults to space ‘where/where\$’ can be L, R, or B or a string beginning with L, R, or B to

	specify characters should be removed from the left of the source, the right of the source or both. If omitted defaults to L
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 and functions are deprecated and should be avoided in new programs, but they are a permanent part of MMBasic and will not be removed - existing programs that use them will continue to run.

BITBANG WS2812 type, pin, nbr, colours%() or BITBANG DHT22 pin, tvar, hvar [,DHT11] or BITBANG BITSTREAM pin, n, array()	The older names of the commands WS2812, HUMID and BITSTREAM, still accepted. New programs should use WS2812, HUMID and BITSTREAM.
DHT22 pin, tvar, hvar [,DHT11]	The older name of the command HUMID, still accepted. New programs should use HUMID.
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 (i.e. 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 (1 for the first column). New programs can use MM.HPOS, the horizontal position of the cursor on the screen in pixels.
RETURN	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB. New programs should use defined subroutines (i.e. SUB...END SUB), which return with END SUB or EXIT SUB.

Appendix A – Serial Communications

Serial Communications

Two serial ports are available on the rear I/O connector for asynchronous serial communications, labelled COM1: and COM2:. In addition, if the serial console is disabled then its port is available as COM3:. USB serial devices plugged into the USB connector are COM4: to COM7: (see USB Serial Devices in the chapter Keyboard, Mouse and Gamepad); they are opened and used in the same way.

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 "COM1:4800" AS #5      ' open the first 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
CLOSE #5                   ' close the serial port
```

The OPEN Command

A serial port is opened using the command:

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

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

'comspec\$' is the communication specification and is a string (it can be a string variable) specifying the serial port to be opened and optional parameters. The default is 9600 baud, 8 data bits, no parity and one stop bit. It has the form "COMn: baud, buf, int, int-trigger, tx-int, 7BIT, (ODD or EVEN), INV, OC, S2" where:

- 'n' is the serial port number, 1 to 7.
- 'baud' is the baud rate. This can be any value from 1200 (the minimum). Default is 9600.
- 'buf' is the receive buffer size in bytes (default size is 1024). 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.
- 'tx-int' is a user defined subroutine which will be called when everything in the transmit buffer has been sent. The default is no interrupt.

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\$':

- 'INV' specifies that the transmit and receive polarity is inverted. Default is non inverted.
- 'OC' will force the transmit pin to be open collector. The default is normal (0 to 3.3V) output.
- 'S2' specifies that two stop bits will be sent following each character transmitted. Default is one stop bit.
- '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).
- 'DEP' will enable RS485 mode with a positive output on the COM1-DE pin.
- 'DEN' will enable RS485 mode with a negative output on the COM1-DE pin.

On COM4: to COM7: (USB serial devices) the baud rate, 7BIT, ODD, EVEN and S2 are passed to the device; INV, OC, DEP and DEN do not apply.

Input/Output Pin Allocation

When a serial port is opened the pins used by the port will be automatically set to input or output as required and the SETPIN and PIN commands will be disabled for the pins. When the port is closed (using the CLOSE command) all pins used by the serial port will be set to a not-configured state and the SETPIN command can then be used to reconfigure them.

The connections for each COM port are shown in the I/O connector pinout tables in the chapter Hardware Features. Note that Tx means an output from the Colour Maximite 2 and Rx means an input to it.

The signal polarity is standard for devices running at TTL voltages (for RS232 voltages see below). Idle is voltage high, the start bit is voltage low, data uses a high voltage for logic 1 and the stop bit is voltage high. These signal levels allow you to directly connect to devices like GPS modules (which generally use TTL voltage levels).

When a serial port is opened MMBasic will enable an internal pullup resistor (to Vdd) on the Rx (receive data) pin. This has a value of about 100K and its purpose is to prevent the input from floating if it is left unconnected. Normally this is fine but it can cause a problem if you have an external resistor in series with the Rx pin, in that case this resistor and the pullup resistor will form a voltage divider limiting how high or low the voltage on the Rx pin can swing and that in turn might mean that the input signal is not recognised. The solution is to use the command OPTION SERIAL PULLUP DISABLE (at the command prompt) to disable it.

Examples

Opening a serial port using all the defaults:

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

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

```
OPEN "COM2:4800" AS #1
```

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

```
OPEN "COM1:9600, 1024" AS #8
```

The same as above but with two stop bits enabled:

```
OPEN "COM1:9600, 1024, S2" AS #8
```

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

```
OPEN "COM1:19200, 1024, ComIntLabel, 256, INV, 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. 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.

Low Cost RS-232 Interface

The RS-232 signalling system is used by modems, hardwired serial ports on a PC, test equipment, etc. It is the same as the serial TTL system used on the Colour Maximite 2 with two exceptions:

- The voltage levels of RS-232 are +12V and -12V where TTL serial uses +3.3V and zero volts.
- The signalling is inverted (the idle voltage is -12V, the start bit is +12V, etc).

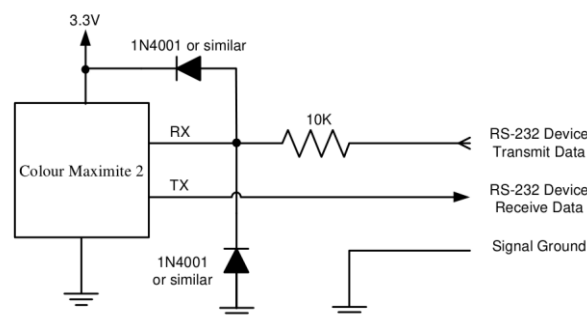
It is possible to purchase cheap RS-232 to TTL converters on the Internet but it would be handy if it was possible to directly interface to RS-232.

The first issue is that the signalling polarity is inverted with respect to TTL. On the Colour Maximite 2 COM1: can be specified to invert the transmit and receive signal (the 'INV' option) so that is an easy fix.

For the receive data (that is the $\pm 12V$ signal from the remote RS-232 device) it is easy to limit the voltage using a series resistor of (say) $10K\Omega$ and two diodes that will clamp the input voltage to the 3.3V rail and ground. The input impedance of the I/O pin is very high so the resistor will not cause a voltage drop but it does mean that when the signal swings to the maximum $\pm 12V$ it will be safely clipped by the diodes.

For the transmit signal (from the Colour Maximite 2 to the RS-232 device) you can connect this directly to the input of the remote device. The output will only swing the signal from zero to 3.3V but most RS-232 inputs have a threshold of about +1V so the signal will still be interpreted as a valid signal.

These measures break the rules for RS-232 signalling, but if you only want to use it over a short distance (a metre or two) it should work fine. Use this circuit:



And open COM1: with the invert option. For example:

```
OPEN "COM1: 4800, INV" AS #1
```

Appendix B – I2C Communications

I2C Communications

The Colour Maximite 2 implements three I2C channels, two on the rear I/O connector and the third dedicated to the front panel Wii connector. All operate in master mode (slave mode is not available).

The I2C command refers to channel 1 while the commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax. There are five commands that can be used:

Command	Description
I2C OPEN speed, timeout	Enables the I2C module in master mode. 'speed' is the clock speed (in KHz) to use and must be one of 100, 400 or 1000. 'timeout' is a value in milliseconds after which the master send and receive commands will be interrupted if they have not completed. The minimum value is 100. A value of zero will disable the timeout (though this is not recommended).
I2C WRITE addr, option, sendlen, senddata [,senddata]	Send data to the I2C slave device. 'addr' is the slave's I2C address. 'option' can be 0 or 1 (see Keeping the Bus below). 'sendlen' is the number of bytes to send (1 to 256). 'senddata' is the data to be sent (all values sent will be between 0 and 255) in any one of three forms: a list of constants, variables or expressions, one for each byte, where array elements such as a(3) are allowed (example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25); a one dimensional array specified with empty brackets, when 'sendlen' elements of the array will be sent starting with the first (example: I2C WRITE &H6F, 0, 3, ARRAY()); or a string variable (not a constant) at least 'sendlen' characters long, when its first 'sendlen' characters are sent (example: I2C WRITE &H6F, 0, 3, STRING\$).
I2C READ addr, option, rcvlen, rcvbuf [,rcvbuf]	Get data from the I2C slave device. 'addr' is the slave's I2C address. 'option' can be 0 or 1 (see Keeping the Bus below). 'rcvlen' is the number of bytes to receive. 'rcvbuf' is where the received data goes, in any one of three forms: a list of numeric variables, one for each byte, where array elements such as a(3) are allowed (a single variable is a list of one, and 'rcvlen' must then be 1); a one dimensional array of numbers specified with empty brackets, when received bytes will be stored in sequential elements of the array starting with the first (example: I2C READ &H6F, 0, 3, ARRAY()); or a string variable, when bytes will be stored as sequential characters and the string set to 'rcvlen' characters ('rcvlen' can then be at most 255, and no more than a LENGTH given to the string).
I2C CHECK addr	Checks whether a device answers at the I2C address 'addr'. MM.I2C is set to 0 if a device answers and to a non zero value if none does. The channel must be open (by I2C OPEN or by a Wii controller or mouse using it). A bus kept by option 1 is released first.
I2C CLOSE	Disables the master I2C module and returns the I/O pins to a "not configured" state. They can then be configured using SETPIN. A bus kept by option 1 is released first.

Following an I2C write or read command the automatic variable MM.I2C will be set to indicate the result of the operation as follows:

- 0 = The command completed without error.
- 1 = Received a NACK response
- 2 = Command timed out

The Wii controllers (CONTROLLER NUNCHUK, CONTROLLER CLASSIC) and an I2C mouse (CONTROLLER MOUSE) use the same channels in the background. A program's I2C commands on a channel that one of them is using wait for the controller's transfer under way to finish, so the two do not collide.

Keeping the Bus

An 'option' of 0 ends each I2C WRITE and READ with a stop condition, which frees the bus. An 'option' of 1 keeps control of the bus: no stop condition is sent, and the next I2C WRITE or READ on the channel begins

with a repeated start instead of a start. Many devices need this to read a register: its number is written with option 1, then the register is read with option 0, which ends with the stop. For example, to read the seconds register (0) of a DS3231 real time clock on channel 2:

```
I2C2 OPEN 100, 1000
I2C2 WRITE &H68, 1, 1, 0      ' the register number, keeping the bus
I2C2 READ &H68, 0, 1, secs    ' a repeated start, the read, and the
stop
I2C2 CLOSE
PRINT HEX$(secs)
```

A bus that is kept is released by the next WRITE or READ with option 0, by I2C CHECK or I2C CLOSE, and at the end of the program. A Wii controller or mouse on the channel is not read while a program keeps its bus.

7-Bit Addressing

The standard addresses used in these commands are 7-bit addresses (without the read/write bit). MMBasic will add the read/write bit and manipulate it accordingly during transfers.

Some vendors provide 8-bit addresses which include the read/write bit. You can determine if this is the case because they will provide one address for writing to the slave device and another for reading from the slave. In these situations you should only use the top seven bits of the address. For example: If the read address is 9B (hex) and the write address is 9A (hex) then using only the top seven bits will give you an address of 4D (hex).

Another indicator that a vendor is using 8-bit addresses instead of 7-bit addresses is to check the address range. All 7-bit addresses should be in the range of 08 to 77 (hex). If your slave address is greater than this range then probably your vendor has specified an 8-bit address.

I/O Pins

Refer to the rear panel I/O connector diagram in the chapter Hardware Features for the pin numbers used for the I2C channels 1 and 2. Their signals are marked as data line (SDA) and clock (SCL). I2C channel 3 is routed to the Wii connector on the front panel. When the I2C CLOSE command is used the I/O pins are reset to a "not configured" state. They can then be configured as per normal using SETPIN.

Both the data line (SDA) and clock (SCL) for all three I2C ports have 10K pullup resistors installed on the motherboard so external pullup resistors are not required. These should be considered if these pins are to be used as general purpose I/O pins.

When running the I2C bus at above 100 kHz the cabling between the devices becomes important. Ideally the cables should be as short as possible (to reduce capacitance) and also the data and clock lines should not run next to each other but have a ground wire between them (to reduce crosstalk).

If the data line is not stable when the clock is high, or the clock line is jittery, the I2C peripherals can get "confused" and end up locking the bus (normally by holding the clock line low). If you do not need the higher speeds then operating at 100 kHz is the safest choice.

Example

As an example, the following program will read and display the current time (hours and minutes) maintained by a PCF8563 real time clock chip connected to I2C channel 2:

```
DIM AS INTEGER RData(2)      ' this will hold received data
I2C2 OPEN 100, 1000          ' open the I2C channel
I2C2 WRITE &H51, 0, 1, 3     ' set the first register to 3
I2C2 READ &H51, 0, 2, RData() ' read two registers
I2C2 CLOSE                   ' close the I2C channel
PRINT "Time is " RData(1) ":" RData(0)
```

Appendix C – 1-Wire Communications

1-Wire Communications

The 1-Wire protocol was developed by Dallas Semiconductor to communicate with chips using a single signalling line. This implementation was written for MMBasic by Gerard Sexton.

There are three commands that you can use:

Command	Description
ONEWIRE RESET pin	Reset the 1-Wire bus
ONEWIRE WRITE pin, flag, length, data [, data...]	Send a number of bytes
ONEWIRE READ pin, flag, length, data [, data...]	Get a number of bytes

Where:

- pin - The I/O pin (located in the rear connector) to use. It can be any pin capable of digital I/O.
- flag - A combination of the following options: 1 - Send reset before command; 2 - Send reset after command; 4 - Only send/recv a bit instead of a byte of data; 8 - Invoke a strong pullup after the command (the pin will be set high and open drain disabled)
- length - Length of data to send or receive
- data - The data to send (WRITE) or where the received data goes (READ), in any one of three forms: a list of items, one for each byte (or bit), whose number must agree with the length parameter (for WRITE each item is a constant, variable or expression; for READ each is a numeric variable; array elements such as a(3) are allowed); an array, floating point or integer, specified with empty brackets (eg, array()), used from its first element; or a string variable (for WRITE its first 'length' characters are sent; for READ the bytes received are stored as its characters and its length is set to 'length').

And the automatic variable:

- MM.ONEWIRE - Returns true if a device was found

After the command is executed, the I/O pin will be set to the not configured state unless flag option 8 is used.

When a reset is requested the automatic variable MM.ONEWIRE will return true if a device was found. This will occur with the ONEWIRE RESET command and the ONEWIRE READ and ONEWIRE WRITE commands if a reset was requested (flag = 1 or 2).

The 1-Wire protocol is often used in communicating with the DS18B20 temperature measuring sensor and to help in that regard MMBasic includes the TEMPR() function which provides a convenient method of directly reading the temperature of a DS18B20 without using these functions.

Appendix D – SPI Communications

SPI Communications

The Serial Peripheral Interface (SPI) communications protocol is used to send and receive data between integrated circuits. The command SPI refers to channel 1 and SPI2 refers to channel 2. SPI2 is not listed below however it is available on the Colour Maximite 2 and has an identical syntax (its function is SPI2()).

I/O Pins

The SPI OPEN command will automatically configure the relevant I/O pins on the rear I/O connector (listed in the chapter Hardware Features). MISO stands for Master In Slave Out and because the Colour Maximite 2 is always the master that pin will be configured as an input. Similarly MOSI stands for Master Out Slave In and that pin will be configured as an output.

When the SPI CLOSE command is used these pins will be returned to a "not configured" state. They can then be configured as per normal using SETPIN.

SPI Open

To use the SPI function the SPI channel must be first opened. The syntax for opening the SPI channel is:

```
SPI OPEN speed, mode [, bits]
```

Where:

- 'speed' is the speed of the clock. This can be 25000000, 12500000, 6250000, 3125000, 1562500, 781250, 390625 or 195315 (ie, 25MHz, 12.5MHz, 6.25MHz, 3.125MHz, 1562.5KHz, 781.25KHz, 390.625KHz or 195.3125KHz). For any other values the firmware will select the next valid speed that is equal or slower than the speed requested.
- 'mode' is a single numeric digit representing the transmission mode – see Transmission Format below.
- 'bits' is the number of bits to send/receive. This can be from 4 to 32 and is 8 if it is left out.

It is the responsibility of the program to separately manipulate the CS (chip select) pin if required.

Transmission Format

The most significant bit is sent and received first. The format of the transmission can be specified by the 'mode' as shown below. Mode 0 is the most common format.

Mode	Description	CPOL	CPHA
0	Clock is active high, data is captured on the rising edge and output on the falling edge	0	0
1	Clock is active high, data is captured on the falling edge and output on the rising edge	0	1
2	Clock is active low, data is captured on the falling edge and output on the rising edge	1	0
3	Clock is active low, data is captured on the rising edge and output on the falling edge	1	1

For a more complete explanation see: http://en.wikipedia.org/wiki/Serial_Peripheral_Interface_Bus

Standard Send/Receive

When the SPI channel is open data can be sent and received using the SPI function. The syntax is:

```
received_data = SPI(data_to_send)
```

Note that a single SPI transaction will send data while simultaneously receiving data from the slave. 'data_to_send' is the data to send and the function will return the data received during the transaction. 'data_to_send' can be an integer or a floating point variable or a constant.

If you do not want to send any data (ie, you wish to receive only) any number (eg, zero) can be used for the data to send. Similarly if you do not want to use the data received it can be assigned to a variable and ignored.

Bulk Send/Receive

Data can also be sent in bulk:

```
SPI WRITE nbr, data1, data2, data3, ... etc
SPI WRITE nbr, string$
SPI WRITE nbr, array()
```

In the first method 'nbr' is the number of data items to send and the data is the expressions in the argument list (ie, 'data1', 'data2' etc). The data can be an integer or a floating point variable or a constant, and array elements such as a(3) are allowed.

In the second or third method listed above the data to be sent is contained in the 'string\$' or the contents of 'array()' (which must be a single dimension array of integer or floating point numbers, specified with empty brackets, and is sent from its first element). The string length, or the size of the array must be the same or greater than nbr. Any data returned from the slave is discarded.

Data can also be received in bulk, in the same three forms:

```
SPI READ nbr, var1, var2, var3, ... etc
SPI READ nbr, string$
SPI READ nbr, array()
```

Where 'nbr' is the number of data items to be received. In the first method each item is stored in a numeric variable of the list (array elements such as a(3) are allowed). In the second the items are stored as the characters of 'string\$', whose length is set to 'nbr' (so 'nbr' can be no more than 255). In the third they are stored in sequential elements of 'array()', integer or floating point, starting with the first. This command sends zeros while reading the data from the slave.

SPI Close

If required the SPI channel can be closed as follows (the I/O pins will be set to inactive):

```
SPI CLOSE
```

Examples

The following example shows how to use the SPI port for general I/O. It will send a command 80 (hex) and receive two bytes from the slave SPI device using the standard send/receive function:

```
PIN(10) = 1 : SETPIN 10, DOUT      ' pin 10 will be used as the enable
signal
SPI OPEN 5000000, 3, 8              ' speed is 5 MHz and the data size is
8 bits
PIN(10) = 0                        ' assert the enable line (active low)
junk = SPI(&H80)                    ' send the command and ignore the
return
byte1 = SPI(0)                      ' get the first byte from the slave
byte2 = SPI(0)                      ' get the second byte from the slave
PIN(10) = 1                        ' deselect the slave
SPI CLOSE                          ' and close the channel
```

The following is similar to the example given above but this time the transfer is made using the bulk send/receive commands:

```
OPTION BASE 1                      ' our array will start with the index
1
DIM data%(2)                       ' define the array for receiving the
data
PIN(10) = 1 : SETPIN 10, DOUT      ' pin 10 will be used as the enable
signal
SPI OPEN 5000000, 3, 8              ' speed is 5 MHz, 8 bits data
PIN(10) = 0                        ' assert the enable line (active low)
SPI WRITE 1, &H80                   ' send the command
SPI READ 2, data%()                 ' get two bytes from the slave
PIN(10) = 1                        ' deselect the slave
SPI CLOSE                          ' and close the channel
```

Appendix E – Regular Expressions

Regex Syntax

The alternate forms of the INSTR() and LINSTR() functions can take a regular expression as the search pattern. The alternate form of the functions are:

INSTR([start],text\$, search\$ [,size])

LINSTR(text%(),search\$ [,start] [,size])

In both cases specifying the size parameter causes the firmware to interpret the search string as a regular expression. The size parameter is a floating point or integer variable that is used by the firmware to return the size of a matching string. As implemented in MMBasic you need to apply the returned *start* and *size* values to the MID\$ function to extract the matched string. e.g.

IF start THEN match\$=MID\$(text\$,start,size) ELSE match\$="" ENDIF

The syntax of regular expressions can vary slightly with the various implementations. This appendix is a summary of the syntax and supported operations used in the MMBasic implementation.

NB: the use of regular expression syntax with OPTION ESCAPE is best avoided otherwise confusion is very likely!

Anchors

- ^ Start of string
- \$ End of string
- \b Word Boundary
- \B Not a word boundary

Quantifiers

- * 0 or more (not escaped)
- + 1 or more
- ? 0 or 1
- {3} Exactly 3
- {3,} 3 or more
- {,5} 5 or less
- {3,5} 3,4 or 5

Groups and Ranges

- (a|b|c) a or b or c
- (...) group
- [abc] Range (a or b or c)
- [^abc] Not (a or b or c)
- [a-q] lower case letters a to q
- [A-Q] upper case letters A to Q
- [0-7] Digits from 0 to 7

Character Classes

- \w digits, and letters plus _
- \W non alpha
- \s white space \t \f \r \n \v spaces
- \S non white space
- \d digits
- \D non digits
- \xXX hex encoded byte

Escapes Required to Match Normal Characters

- \^ to match ^ (caret)
- \. to match . (dot)
- * to match * (asterisk)
- \\$ to match \$ (dollar)
- \[to match [(left bracket)
- \\ to match \ (backslash)
- \? to match ? (question mark)
- \{ to match { (left curly bracket)
- \} to match } (right curly bracket)
- \| to match | (pipe)
- \(to match ((left bracket)
- \) to match) (right bracket)
- \+ to match + (plus)

Matching rules

- Non special characters match themselves
- . matches any character
- A word boundary is at start or end of a string or where a \w Character has a \W Character adjacent to it.

Limitations

- Anchors within a group are not supported. e.g. (^hello) or (hello\$) will not match "hello" at beginning or end of the line as expected.
- However, anchors outside the group are ok. e.g. ^(hello) or (hello)\$ will match

Example expression to match an IP Address.

```
"[\d]+\.[\d]+\.[\d]+\.[\d]+"
```

Using Regular expressions with OPTION ESCAPE

When a regular expression is literally embedded in the INSTR or L INSTR function then any syntax using the ‘\’ escape character is handled correctly without further escaping the backslash, whether OPTION ESCAPE is in use or not. e.g.

```
? instr("test123", "\d", size)
```

The INSTR / L INSTR functions temporarily turn off OPTION ESCAPE while the regex expression is read.

However, if the Regex expression is passed as a string variable and OPTION ESCAPE is enabled when you assign the regex expression to the variable, you must escape any backslash in the regex expression. e.g.

```
s$="\d"  
? instr("test123", s$, size)
```

Appendix F – Sprites

Sprites

The concept of the sprite implementation is as follows:

- Sprites are full colour, up to the size of the screen, and kept through a change of mode. 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, the static objects, 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.
- Up to seven collisions are recorded for each sprite by a SHOW or SCROLL command.
- 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).

SPRITE and BLIT are one command: every SPRITE command can also be written as BLIT. Only WRITE and LOAD differ with the name used, as on the PicoMite (see BLIT).

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

```

(A collision with static object k returns $\&H80 + k$: see below.)

Static Objects and the Background

A game's scenery is usually drawn on the screen rather than made of sprites. Two features of the PicoMite let sprites collide with it:

- **Static objects** (SPRITE STATIC) are up to 64 invisible rectangles that the program declares where its scenery is solid: a wall, a platform, the ground. They have no pixels and are never drawn. Every collision check tests the sprites displayed against them, whatever their layer; a sprite overlapping static object k has $\&H80 + k$ in its list of collisions (SPRITE(C, n, m)), and SPRITE STINTERRUPT can name a routine of its own for them. Unlike a collision between sprites, which is reported once when it begins, a static object is reported at every check while the sprite overlaps it. SPRITE SCROLL moves the static objects with the background.
- **SPRITE(B, n)** compares the background stored under sprite n with the sprite's shape, pixel by pixel: 0 if nothing is there, 1 if something is within its rectangle but not its shape, 2 if something touches its shape. SPRITE(B, n, side) then says from which side and how deep.

So a sprite can land on a platform declared as a static object, and be stopped by anything drawn. This is the heart of a platform game's step, a frame at a time (the player is sprite 1, PH pixels high, at px, py and moving vx across and vy down):

```

' across: into the scenery, the step is taken back
SPRITE SHOW #1, px + vx, py, 1
IF SPRITE(B, 1) = 2 THEN SPRITE SHOW #1, px, py, 1 ELSE px = px + vx
' down: a platform met from above is landed on
SPRITE SHOW #1, px, py + vy, 1
FOR m = 1 TO SPRITE(C, 1)
    e = SPRITE(C, 1, m)
    IF e >= &H80 AND e < &HF0 AND vy >= 0 THEN
        IF py + PH <= SPRITE(ST, e - &H80, Y) THEN vy = 0 : py =
        SPRITE(ST, e - &H80, Y) - PH
    ENDIF
NEXT m

```

```
NEXT m
SPRITE SHOW #1, px, py + vy, 1
py = py + vy
```

Appendix G – Turtle Graphics

Turtle Graphics

MMBasic includes a very comprehensive turtle graphics implementation (the same as the PicoMite's). The supported commands, each with "TURTLE " in front of it, are listed below. A short form is shown in brackets after a command's description.

Several commands may also be written in other ways: BACK as BACKWARD, LEFT and RIGHT as TURN LEFT and TURN RIGHT, SET XY as SETXY or MOVE, SET X as SETX, SET Y as SETY, SET HEADING as SETHEADING or HEADING, PEN UP as PENUP, PEN DOWN as PENDOWN, PEN COLOUR as PEN COLOR, PENCOLOR or PENCOLOR, PEN WIDTH as PENWIDTH, ARCLEFT as ARC LEFT, ARCRIGHT as ARC RIGHT, FILL COLOUR as FILL COLOR, FILLCOLOUR or FILLCOLOR, FILL PATTERN as FILLPATTERN, NO FILL as NOFILL, BEGIN FILL as BEGIN_FILL, END FILL as END_FILL, SHOW TURTLE as SHOWTURTLE, HIDE TURTLE as HIDETURTLE, CURSOR SIZE as CURSORSIZE and CURSOR COLOUR as CURSOR COLOR, CURSORCOLOUR or CURSORCOLOR.

Command	Description
Movement Commands	
FORWARD distance	Move forward by distance pixels (FD)
BACK distance	Move backward by distance pixels (BK)
LEFT [angle]	Turn left by angle degrees, 90 degrees if not specified (LT)
RIGHT [angle]	Turn right by angle degrees, 90 degrees if not specified (RT)
Position Commands	
SET XY x, y	Move to absolute position (x,y)
SET X x	Set X coordinate, keep Y
SET Y y	Set Y coordinate, keep X
SET HEADING angle	Set absolute heading (0=up, 90=right) (SETH)
HOME	Return to center (MM.HRES\2, MM.VRES\2) heading 0
Pen Control Commands	
PEN UP	Lift pen (stop drawing) (PU)
PEN DOWN	Lower pen (start drawing) (PD)
PEN COLOUR colour	Set pen colour (PC)
PEN WIDTH width	Set pen line width (PW)
Arc and Curve Commands	
ARC radius, angle	Draw arc with given radius and angle
ARCLEFT radius, angle	Draw arc turning left (ARCL)
ARCRIGHT radius, angle	Draw arc turning right (ARCR)
BEZIER cp1, cp1angle, cp2, cp2angle, end, endangle	Draw Bezier curve with control points
Basic Shape Commands	
CIRCLE radius	Draw circle at current position
DOT [size]	Draw filled dot (default size=5)
FCIRCLE radius	Draw filled circle
FRECTANGLE width, height	Draw filled rectangle (FRECT)
ARECTANGLE width, height	Draw rectangle outline (ARECT)
WEDGE radius, start, end	Draw filled wedge/pie slice
Fill Commands	
FILL COLOUR colour	Set fill colour and enable filling (FC)
FILL PATTERN pattern	Set fill pattern (0-31) (FP)

NO FILL	Disable filling
BEGIN FILL	Start recording polygon for fill (BF)
END FILL	End recording and fill polygon (EF)
Cursor Commands	
SHOW TURTLE	Show turtle cursor (ST)
HIDE TURTLE	Hide turtle cursor (HT)
CURSOR SIZE size	Set cursor size (CS)
CURSOR COLOUR colour	Set cursor colour (CC)
STAMP	Draw a turtle at the current x,y position
State Management Commands	
RESET [show]	Clear screen and reset everything, show the turtle if show = 1
PUSH	Save current position and heading to stack
POP	Restore position and heading from stack

The code includes the ability to fill circles, rectangles and polygons with a textured fill, using these patterns:

Nbr	Pattern	Nbr	Pattern
0	Solid fill	16	Grid
1	Checkerboard	17	Weave pattern
2	Vertical lines	18	Diamond
3	Horizontal lines	19	Gradient diagonal
4	Diagonal cross	20	Gradient diagonal reverse
5	Diagonal stripes	21	Border/frame
6	Crosshatch	22	Vertical split
7	Fine diagonal	23	Woven
8	Dense checkerboard	24	Sparse dots
9	Diagonal right medium	25	Diagonal very fine
10	Diagonal left medium	26	Arrow up
11	Vertical lines medium	27	Dense dots
12	Horizontal lines medium	28	Chevron
13	Large checkerboard	29	Diamond hollow
14	Dotted vertical	30	Circle
15	Horizontal stripes tight	31	Circle filled

Appendix H – Cyclic Redundancy Check (CRC)

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$)
' check the received CRC and data
c% = MATH(CRC16 txd$, , &h8005, &hffff, 0, 1, 1)
check$ = CHR$(c% AND &HFF) + CHR$((c% >> 8) AND &HFF)
PRINT
PRINT "The CRC of the received data including the"
PRINT "received 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])
```

'n' is 8, 12, 16 or 32. Please see the entry for MATH() in the Functions table. 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%)
```

On the Colour Maximite 2 this prints:

```
CRC16-CCITT  29B1
CRC16-MODBUS 4B37
CRC16-XMODEM 31C3
CRC8-MAXIM   A1
CRC32        CBF43926
```

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 and https://en.wikipedia.org/wiki/Cyclic_redundancy_check

- On line CRC calculators: <http://zorc.breitbandkatze.de/crc.html>, <https://crccalc.com> and <https://www.1ddgo.net/en/encrypt/crc>

Appendix I – Special Keyboard Keys

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. Where a key gives a different code when the Shift key is held down, that code is shown too.

Keyboard Key	Key Code (Hex)	Key Code (Decimal)	With Shift (Hex)
Backspace	08	8	
Tab	09	9	9F
Escape	1B	27	
Delete	7F	127	A0
Up Arrow	80	128	A4
Down Arrow	81	129	A1
Left Arrow	82	130	A2
Right Arrow	83	131	A3
Insert	84	132	
Home	86	134	
End	87	135	
Page Up	88	136	
Page Down	89	137	
F1	91	145	B1
F2	92	146	B2
F3	93	147	B3
F4	94	148	B4
F5	95	149	B5
F6	96	150	B6
F7	97	151	B7
F8	98	152	B8
F9	99	153	B9
F10	9A	154	BA
F11	9B	155	BB
F12	9C	156	BC
Print Screen	9D	157	BD
Pause/Break	9E	158	

For the function keys and Print Screen the Shift key adds 20 (hex) to the code (this is the equivalent of setting bit 5): for example Shift-F3 generates B3 (hex) and Shift-F10 generates BA (hex). The Shift key also gives the codes shown for Tab, Delete and the arrow keys, which the editor uses for its Shift-Tab, Shift-Delete and selection keys. It is ignored for the other keys in the table.

The Alt keys and the Control keys are modifiers: they do not generate a code of their own (with Control held down a letter key gives its control code, eg Ctrl-C gives 3). The state of the modifier keys is returned by KEYDOWN(7) and that of the lock keys by KEYDOWN(8).

MMBasic will translate the VT100 escape codes generated by terminal emulators such as Tera Term and Putty for the arrow keys, Insert, Delete, Home, End, Page Up, Page Down and F1 to F12 to these codes. Of the shifted keys only Shift-F3 is translated. 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 J – CAN Support

CAN Support

MMBasic on the Colour Maximite 2 exposes the native support for CAN provided by the STM32H743 chip as the MMBasic CAN command.

A CAN transceiver such as the SN65HVD230 CAN Transceiver is required to interface to the actual physical CAN bus. It connects to 3.3V and GND plus the nominated CAN Tx and CAN Rx pins on the Colour Maximite 2. The CANL and CANH connectors go to the physical CAN bus. You can exercise the CAN command in loopback mode without having the transceiver attached.



The STM32H743 chip supports both the Classical CAN frame type and also the newer CAN FD (CAN Flexible Data-rate) frame type. The improvements are:

- CAN FD supports up to 64 data bytes per data frame vs. 8 data bytes for Classical CAN.
- CAN FD supports dual bit rates: the nominal (arbitration) bit-rate limited to 1 Mbit/s as in Classical CAN, and the data bit-rate, which depends on the network topology/transceivers.
- CAN FD uses an improved cyclic redundancy check (CRC) and the "protected stuff-bit counter", which lower the risk of undetected errors.

The following MMBasic commands provide access to the CAN controller in the STM32H743 chip. There are some variations to the syntax between Classical CAN and CAN FD. These common parameters apply to their use in the following commands:

Parameter	Description
id, id1, id2	These refer to either an 11bit (0-7FF) or 29bit (0-1FFFFFFF) identifier for a CAN Frame. id2 may also refer to a bit mask in the CAN FILTER command.
eid	0 indicates 11bit id (STDID) is used/expected, 1 indicates a 29bit id (EXTID) is used/expected
rtr	Indicates a Remote Frame. i.e. dlc is 0 and no data. These are not used much any more.
dlc	Indicates the size of the data in the CAN message. (0-8) (Classical Frame Type)
msg	Contains 8 bytes of data, only those indicated by dlc are valid, the rest are set to 00.
ret	Is the return value for the command.

CAN OPEN index, speed, mode

This command opens the CAN interface indicated by index. This configures the CAN based on predefined pin allocations as shown below.

Index	CAN Rx	CAN Tx	Notes
1	pin 5	pin 3	Rx shares I2C SCK, Tx shares I2C SDA
2	pin 19	pin 40	Rx shares SPI MOSI, Tx shares SPI2 CLK

The speed indicates the desired CAN speed. Allowed values are 0 (special case), 125000, 250000, 500000 and 1000000. The sampling point is set to 80%. The speed is not directly given to the CAN hardware, a set of parameters is provided that will achieve the desired CAN speed and sample point. This short form of the CAN OPEN opens the CAN with a Frame Type of Classical.

To set a different speed for Classical CAN or use CAN FD the special case 0 must be used and the syntax as described below is required to specify the parameters to obtain the desired speed, frame type and sample points.

The mode parameter determines the mode to be configured:

Mode	Operation	Notes
0	Normal	Normal operation

1	Internal Loop Back	This mode can be used for a loopback test. No data is placed on or received from the physical CAN Bus, however internally the transmitted data is also seen as received data. The transceiver does not need to be attached.
2	External Loop Back	This mode can be used for a loopback test. The transmitted data is also placed on the CAN Bus and seen by other nodes if the transceiver is attached and connected to the CAN Bus.
3	Bus Monitoring	This mode does not place data on the physical CAN Bus, but does receive data from the CAN Bus. Can be used to monitor a live system.

In Normal mode the CAN sends and reads messages via the CAN bus using the attached CAN transceiver. It does not see its own messages. In Loop Back mode the CAN sees its own messages as if they were received from the CAN bus. The CAN transceiver does not need to be attached. This mode is useful for testing/developing and checking your filters work as expected. In this mode no messages are placed on or received from the CAN bus if it is attached.

The CAN OPEN command also sets a default Global Filter that accepts all messages into the FIFO1 buffer (see CAN FILTER GLOBAL below). This will start as soon as a CAN START command is issued. If you want to apply alternate filters to restrict what messages are accepted you enter them after the CAN OPEN command and before the CAN START command.

The following long forms of the CAN OPEN command, used when the speed is set to 0, allow the user to provide the prescaler, segment1, segment2 and SJW (SyncJumpWidth) values that allow the detailed selection of the speed and sampling point. The base clock speed is 40MHz and an online calculator (search for "Bit Timing Calculator for CAN FD") allows the determination of the values for various CAN speeds and sampling points.

The syntax below is used to set a Frame Type of Classical CAN to a desired speed and sample point:

```
CAN OPEN index,0,mode,prescaler,seg1,seg2,sjw
```

The syntax below is used to set a Frame Type of FD CAN to a desired speed and sample point. It also defines additional parameters to set the speed and sample point of the data segment, sets whether Bit Rate Switching is used and the data size to be used in the FD CAN frame (8, 12, 16, 20, 24, 32, 48 or 64):

```
CAN OPEN
index,0,mode,prescaler,seg1,seg2,sjw,brs,dsiz,e,dprescaler,dseg1,dseg2,dsjw
```

The parameters prescaler, seg1, seg2, sjw, dprescaler, dseg1, dseg2 and dsjw are the values obtained from the Bit Timing Calculator for CAN FD. brs indicates whether Bit Rate Switching is used (0 or 1). dsiz indicates the maximum data size (8 to 64). These values give the common speeds:

```

/**** Bus Timing Parameters for FDCAN Clock Frequency 40MHz on H7 *****/
*
Nominal sample point 80%
DATA sample point at 60%
Speed   Pre  Seg1 Seg2 JSW Sample  Data-> Pre TSeg1 TSeg2 SJW Sample
KHz     Scale  --- --- ---   Point  Scale  --- --- ---   point
-----
125/125  1    255  64  64  80%    1    191  128  128  60%
250/250  1    127  32  32  80%    1    95   64   64   60%
500/500  1    63   16  16  80%    1    47   32   32   60%
1000/1000 1    31    8   8  80%    1    23   16   16   60%
500/2000 1    63   16  16  80%    1    11    8    8   60%
1000/2000 1    31    8   8  80%    1    11    8    8   60%

```

CAN FILTER GLOBAL id, eid, idr, eidr

The Global filter determines how each of the possible four types of CAN messages are handled. id and eid respectively apply to STDID and EXTID messages. They determine what happens to messages that do not match any filter after all the filters are checked. idr and eidr determine whether remote frames (i.e. rtr=1) are accepted. They can have values as below.

id and eid (normal frame rtr=0)	idr and eidr (remote frames i.e. rtr=1)
0 = Accept unmatched and pass to RxFIFO0	0 = Pass to filters for possible matching.
1 = Accept unmatched and pass to RxFIFO1	1 = Reject message, don't attempt matching.

2 = Reject message	
--------------------	--

When a CAN OPEN command is issued a default Global Filter is set so that all messages are accepted to RxFIFO1. i.e. the equivalent of CAN FILTER GLOBAL 1,1,1,1

CAN FILTER index, eid, type, config, id1, id2

Filters are used to let the CAN hardware do the work to discard messages you are not interested in, without using the MMBasic resources to analyse them. There are 32 filters configured for each of STDID and EXTID. They are tested in order of the index.

- index is the filter number 0-31, these are applied in order.
- eid is 0 for STDID filters and 1 for EXTID filters.
- type is the filter type: 0 is a range filter matching ids in the range id1 to id2; 1 is a filter containing two IDs (id1, id2) to be matched; 2 is a classic filter where id1 is the filter and id2 is the mask; 3 is a range filter for EXTIDs (eid 1) in which the extended ID mask is not applied.
- config values: 0 will disable the filter, i.e. it is ignored; 1 will accept a matched message into Rx FIFO0; 2 will accept a matched message into Rx FIFO1; 3 will reject the message.

The following filter will accept STDID messages with ids 235-300 into FIFO0:

```
CAN FILTER 0, 0, 0, 1, &H235, &H300
```

CAN START

After the CAN is configured and the filters are set the CAN START command tells the CAN to start accepting messages and to accept messages to send. The CAN START command MUST be issued to start operation.

CAN STOP

The CAN STOP command can be issued to stop sending and receiving messages. You could issue this command if you wanted to pause the reception of messages.

CAN SEND

```
CAN SEND id, eid, rtr, dlc, msg, ret      ' Classical CAN
CAN SEND id, eid, rtr, dlc, msg(), ret    ' CAN FD
```

This command places a message in the next available transmit buffer and it is sent in order. There are 32 transmit buffers configured, so there is no need to configure a queue in MMBasic. If no free buffer is available then the ret value is 1.

If Frame Type of CAN FD is used then:

- If brs=1 in the CAN OPEN command Bit Rate Switching will be used.
- msg() must be a suitably dimensioned array for the data e.g. INTEGER msg(7) for 64 bytes.
- dlc must not exceed the datasize set in the CAN OPEN command.
- dlc can have values 0-8, 12, 16, 20, 24, 32, 48 or 64.

CAN READ

```
CAN READ fifo, id, eid, rtr, dlc, msg, fmi, ret    ' Classical CAN
CAN READ fifo, id, eid, rtr, dlc, msg(), fmi, ret  ' CAN FD
```

This command reads a message from either FIFO0 or FIFO1 depending on the value of fifo and populates id, eid, rtr, dlc, msg and fmi with the details from the received CAN Frame. Each of the FIFO queues is configured with 32 buffers, so there is no need to configure a queue within MMBasic.

If Frame Type of CAN FD is used then:

- msg() must be a suitably dimensioned array. e.g. INTEGER msg(7) for 64 bytes.
- dlc must not exceed the datasize set in the CAN OPEN command.
- dlc can expect values 0-8, 12, 16, 20, 24, 32, 48 or 64.

ret is 0 if no messages are available else the number of messages available. The application needs to continually call this command in the main DO:LOOP as this implementation does not use interrupts and expects that messages are polled. Only one message is retrieved for each CAN READ command.

fmi is the Filter Match Index. It indicates which filter matched the message. The fmi is the filter number used when adding the filter. There are two numbering sequences numbered sequentially starting from 0 for each of STDID and EXTID filters. Using the fmi is an alternate method of analysing the incoming message, i.e. know which filter matched it to see what to do with it. The typical method is to analyse the ID to identify a particular message.

CAN CLOSE

This command closes the CAN and releases the pins.

Identifying a Node

If you are implementing a CAN system it is useful to be able to uniquely identify a node e.g. serial no.

MM.INFO(ID) will return a unique 96bit ID for the processor as a hexadecimal string. This, whilst guaranteed to be unique, is too long to easily pass in a single CAN message.

MM.INFO(ID48) will return an integer with a 48 bit hash of this 96 bit ID which while not guaranteed to be unique will most probably be unique within any MMBasic systems you have. This can be placed in 6 bytes of a CAN message to identify a node and have two bytes left for the return of an allocated shorter node number to be used within the system.

Appendix K – Loading the Firmware

Loading the Firmware

The ARM Cortex-M7 processor includes its own programmer/bootloader so the Colour Maximite 2 firmware can be easily loaded via USB using a personal computer or laptop (special hardware is not needed). Just follow these steps. Note that the same firmware file will work on either the Generation 1 or 2 versions of the Colour Maximite 2 (the hardware is automatically detected).

If you have a version using the Waveshare CPU board:

- Plug the Waveshare CPU board into its position on the motherboard.
- Remove all the jumpers on the Waveshare CPU board.
- Set the power switch to "5VIN".
- Set the BOOT CONFIG switch to "SYSTEM".

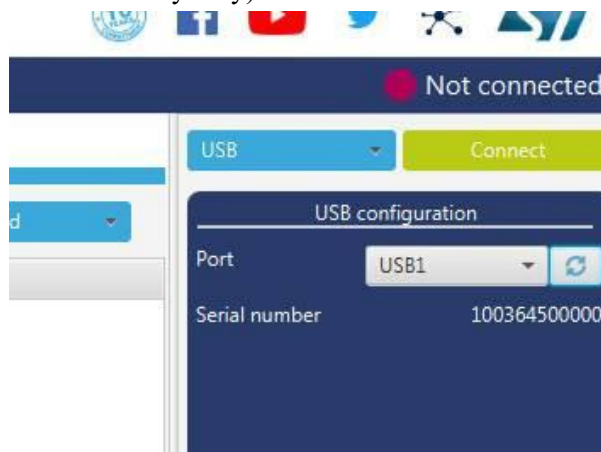
If you have a version without the Waveshare module (ie, the Generation 2 design) the programming function is provided by a set of three jumper pins labelled "PGM" and "RUN". Place a jumper on the common and "PGM" pins.

Make sure that the Type-B USB power cable to the Colour Maximite 2 is disconnected.

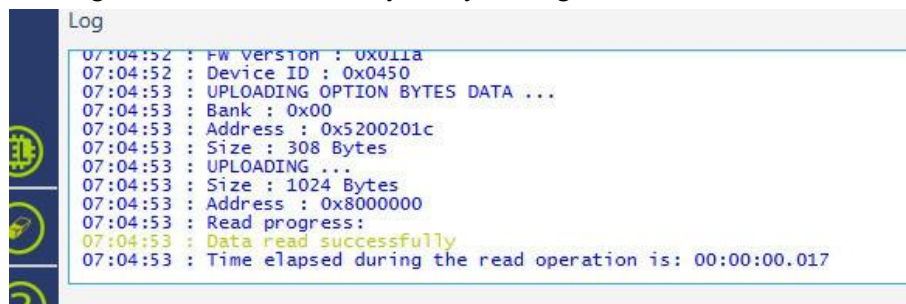
Using a USB Type-A to Type-A cable connect the USB Keyboard port on the Colour Maximite 2 to a USB port on your desktop computer. This will power up the Colour Maximite 2 regardless of the position of the power switch. You should also hear a sound from your computer as it connects to the Colour Maximite 2.

Go to <https://www.st.com/en/development-tools/stm32cubeprog.html> and download the STM32CubeProgrammer software. This is free software but STM do require you to have an STM account or provide your name and email address. They will email you a link to download the software. Then install this software on your computer (Windows, Linux and macOS are supported).

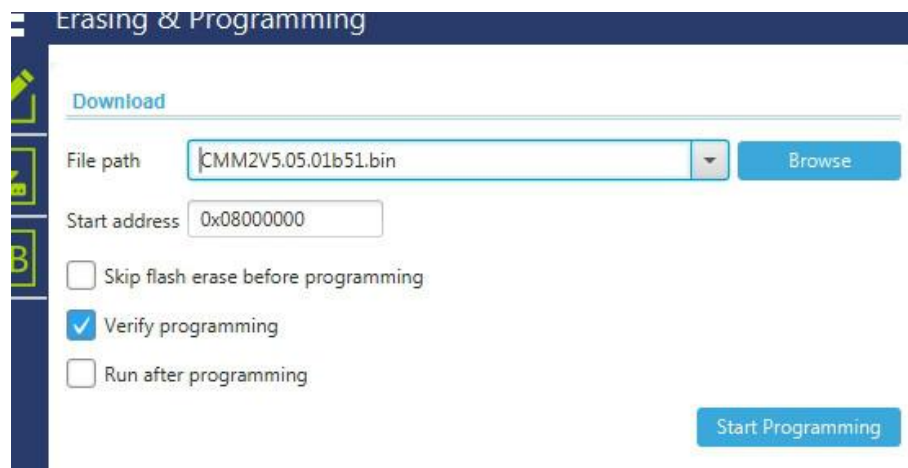
Run the STM32CubeProgrammer software on your computer. On the top right of the program window select USB as the communications method. If the program does not recognise the USB connection click on the small blue circle to the right of the Port drop down list to refresh the entry. Your screen should look like the illustration below (the USB port number may vary).



Click on the "Connect" button. You should then see a series of messages as shown in the screenshot below finishing with the message "Data read successfully". Any messages in red will indicate an error.

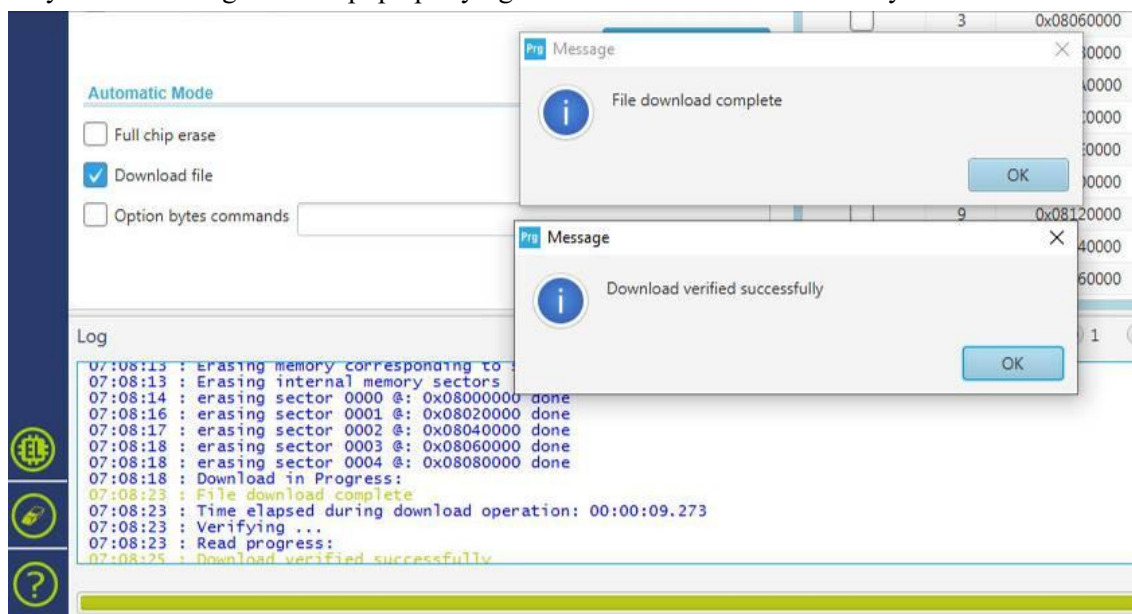


Click on the download button on the left side of the STM32CubeProgrammer window and the software will switch to the "Erasing and Programming" mode as shown below.



Use the "Browse" button to select the firmware file (it will have an extension of .bin). Tick the "Verify programming" checkbox. Finally, click on the "Start Programming" button.

The STM32CubeProgrammer software will then program the firmware into the flash memory on the ARM Cortex-M7 in the Colour Maximite 2 (the STM32CubeProgrammer software calls this "downloading"). After a short time a dialog box will pop up saying that "File download completed". Do not do anything at this point as the software will then start reading back the firmware programmed into the flash. When this has completed successfully another dialog box will pop up saying "Download verified successfully" as shown below.



The whole operation will take under a minute and any messages in red will indicate an error. Then:

- Dismiss all the dialog boxes and close the STM32CubeProgrammer software.
- On the Waveshare module set the BOOT CONFIG switch to "Flash" or on versions without the Waveshare module the jumper should be between the common and the "RUN" pins.
- Remove the USB Type-A to Type-A cable from the USB Keyboard port.
- Plug in the VGA monitor and USB keyboard.
- Plug the Colour Maximite 2 into power and set the front panel power switch to ON.

On power up you should now see on the VGA monitor the Maximite logo and the version number of the firmware that you have just loaded.

If your monitor remains blank check that you are not still in programming mode (ie, reset the BOOT CONFIG switch or the programming jumper). If you do not have a VGA monitor you can use the serial console (over USB) to check the firmware installation. This is described in the section Power & Serial Console Connector in the chapter Hardware Features.

When MMBasic is first loaded (and whenever the clock has stopped, for example because its battery has failed) it will ask for the keyboard type, the time and the date. On a firmware upgrade MMBasic keeps these options

and does not ask for them again: the keyboard type, the clock's settings (OPTION RTC CALIBRATE, OPTION RTC DRIVE and OPTION DS3231), the console port, the SD card speed, the default video mode and the editor's font, the mouse and OPTION MAXCTRLS (the number of GUI controls). The other options return to their defaults (the console's baud rate among them) and can be set again with the relevant OPTION commands. Saved variables (VAR SAVE) are cleared by an upgrade, and so is the program held in flash: a program that ran at power-up (OPTION AUTORUN) must be run again from its file, and OPTION AUTORUN set again. Programs on the SD card or the USB drive are not affected.

If you wish to load another version of the firmware (either earlier or later) this can be done by repeating the steps above.

Alternative Method

An alternative method of loading the firmware is to use serial transfer over USB. To do this you must first install the serial over USB driver on your desktop computer as described in the section Power & Serial Console Connector in the chapter Hardware Features.

If you have a version using the Waveshare CPU board:

- Remove all the jumpers on the Waveshare CPU board.
- Set the power switch to "5VIN".
- Set the BOOT CONFIG switch to "System".

If your version does not use the Waveshare CPU board:

- Place a jumper between the common and "PGM" pins on the mainboard.

Then:

- Connect your desktop computer to the USB Type-B connector on the Colour Maximite 2.
- Set the front panel power switch to ON and/or press the reset button on the Waveshare CPU board.

This should power up the Colour Maximite 2 which will then connect to your desktop computer via USB. In Windows the connection will appear in Device Manager as "USB Serial Port" (the COM number will probably be different):



Install the STM32CubeProgrammer software on your computer as described above. Run the software and select UART in the top right corner (as illustrated below). Then select the correct COM port number as reported in the Windows Device Manager. Finally make sure that the baudrate is set to 115200 baud and the parity set to even:



From then on the process is the same as that described above when using a direct USB connection via the keyboard port:

- Click on "Connect".
- Select "Erase & Programming" mode.
- Browse for the firmware file.
- Tick the "Verify programming" checkbox.
- Click on "Program".

The whole operation will take about 5 minutes.

Note that after the initial firmware has been loaded you can use the UPDATE FIRMWARE command to place the ARM Cortex-M7 into download mode (equivalent to setting the BOOT CONFIG switch to "SYSTEM" or the jumper to "PGM") thereby avoiding opening the case.

When the programming/verify has completed set the BOOT CONFIG switch to "Flash" on the CPU board (or return the jumpers to the "RUN" position) and press the RESET button.

Third Method

If you have a version using the Waveshare board it is also possible to program the firmware using the micro USB connector on this board. This method is not recommended as the Waveshare module must be removed to access this connector and doing this too many times will inevitably damage the module's pins. However, this is a handy method of testing the Waveshare board - if you can load the firmware without error it is a good indication that the module is working correctly.

To use this method remove the Waveshare CPU module from the motherboard, place shorting jumpers on all header pins except PA9-VBUS, set the power switch to "USB" and the BOOT CONFIG switch to "SYSTEM". Plug a USB cable into the micro USB connector on the top of the module and the other end into your desktop computer - both LEDs on the module should illuminate and it should connect to your computer.

Then, using the steps listed for loading the firmware via the USB keyboard port, load and verify the Colour Maximite 2 firmware using this USB cable and your desktop computer.

Linux and the Raspberry Pi

Loading the firmware from a Linux computer and/or the Raspberry Pi has some special considerations and these are explained here: <https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12171>

Appendix L – Programming in BASIC - A Tutorial

Programming in BASIC - A Tutorial

The BASIC language was introduced in 1964 by Dartmouth College in the USA as a computer language for teaching programming and accordingly it is easy to use and learn. At the same time, it has proved to be a competent and powerful programming language and as a result it became very popular in the late 70s and early 80s. Even today some large commercial data systems are still written in the BASIC language (primarily Pick Basic).

The BASIC interpreter used in the Colour Maximite 2 is called MMBasic and is a modern version of the BASIC language which loosely emulates the Microsoft BASIC interpreter that was popular years ago.

For a programmer the greatest advantage of BASIC is its ease of use. Some more modern languages such as C and C++ can be truly mind bending but with BASIC you can start with a one line program and get something sensible out of it. MMBasic is also powerful in that you can draw sophisticated graphics, manipulate the external I/O pins to control other devices and communicate with other devices using a range of built-in communications protocols.

Command Prompt

Interaction with MMBasic is done via the console at the command prompt (ie, the greater than symbol (>) on the console). 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 wide range of commands that you can enter and execute. Typically they would list the program held in memory (LIST) or edit it (EDIT) or perhaps set some options (the OPTION command). Most times the command is just RUN which instructs MMBasic to run the program held in program memory.

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 (more on this later), which you can test by entering the following at the command prompt:

```
PRINT 2 + 2
```

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

This ability to test a command at the command prompt is useful when you are learning to program in BASIC, so it would be worthwhile having your Colour Maximite 2 handy for the occasional test while you are working through this tutorial.

Structure of a BASIC Program

A BASIC program starts at the first line and continues until it runs off the end or hits an END command - at which point MMBasic will display the command prompt (>) on the console and wait for something to be entered.

A program consists of a number of statements or commands, each of which will cause the BASIC interpreter to do something (the words *statement* and *command* generally mean the same and are used interchangeable in this tutorial).

Normally each statement is on its own line but you can have multiple statements in the one line separated by the colon character (:).

For example;

```
A = 24.6 : PRINT A
```

Each line can start with a line number. Line numbers were mandatory in the early BASIC interpreters however modern implementations (such as MMBasic) do not need them. You can still use them if you wish but they have no benefit and generally just clutter up your programs.

This is an example of a program that uses line numbers:

```
50 A = 24.6  
60 PRINT A
```

A line can also start with a label which can be used as the target for a program jump using the GOTO command. This will be explained in more detail when we cover the GOTO command but this is an example (the label name is `JumpBack`):

```
JumpBack: A = A + 1  
PRINT A
```

Comments

A comment is any text that follows the single quote character ('). A comment can be placed anywhere and extends to the end of the line. If MMBasic runs into a comment it will just skip to the end of it (ie, it does not take any action regarding a comment).

Comments should be used to explain non obvious parts of the program and generally inform someone who is not familiar with the program how it works and what it is trying to do. Remember that after only a few months a program that you have written will have faded from your mind and will look strange when you pick it up again. For this reason you will thank yourself later if you use plenty of comments.

The following are some examples of comments:

```
' calculate the hypotenuse
PRINT SQR(a * a + b * b)
INPUT var      ' get the temperature
```

Older BASIC programs used the command REM to start a comment and you can also use that if you wish but the single quote character is easier to use and more convenient.

The PRINT Command

There are a number of common commands that are fundamental and we will cover them in this tutorial but arguably the most useful is the PRINT command. Its job is simple; to print something on the console. This is mostly used to output data for you to see (like the result of calculations) or provide informative messages.

PRINT is also useful when you are tracing a fault in your program; you can use it to print out the values of variables and display messages at key stages in the execution of the program.

In its simplest form the command will just print whatever is on its command line. So, for example:

```
PRINT 54
```

will display on the console the number 54 followed by a new line.

The data to be printed can be something simple like this or an expression, which means something to be calculated. We will cover expressions in more detail later but as an example the following:

```
> PRINT 3/21
0.1428571429
>
```

would calculate the result of three divided by twenty one and display it. Note that the greater than symbol (>) is the command prompt produced by MMBasic – you do not type that in.

Other examples of the PRINT command include:

```
> PRINT "Wonderful World"
Wonderful World
> PRINT (999 + 1) / 5
200
>
```

You can try these out at the command prompt.

The PRINT command will also work with multiple values at the same time, for example:

```
> PRINT "The first number is" 20+25 " and the second is" 18/3
The first number is 45 and the second is 6
>
```

Normally each value is separated by a space character as shown in the previous example but you can also separate values with a comma (.). The comma will cause a tab to be inserted between the two values. In MMBasic tabs in the PRINT command are eight characters apart.

To illustrate tabbing, the following command prints a tabbed list of numbers:

```
> PRINT 12, 34, 9.4, 1000
12      34      9.4      1000
>
```

Note that there is a space printed before each number. This space is a place holder for the minus symbol (-) in case the value is negative. You can see the difference with the numbers 12 and 9.4 in this example:

```
> PRINT -12, 34, -9.4, 1000
-12      34      -9.4      1000
>
```

The print statement can be terminated with a semicolon (;). This will prevent the PRINT command from moving to a new line when it has printed all the text. For example:

```
PRINT "This will be";
PRINT " printed on a single line."
```

Will result in this output:

```
This will be printed on a single line.
```

The message would be look like this without the semicolon at the end of the first line:

```
This will be
printed on a single line.
```

Variables

Before we go much further we need to define what a "variable" is as they are fundamental to the operation of the BASIC language (in fact, most programming languages). A variable is simply a place to store an item of data (ie, its "value"). This value can be changed as the program runs which why it is called a "variable".

Variables in MMBasic can be one of three types. The most common is floating point and this is automatically assumed if the type of the variable is not specified. The other two types are integer and string and we will cover them later. A floating point number is an ordinary number which can contain a decimal point. For example 3.45 or -0.023 or 100.00 are all floating point numbers.

A variable can be used to store a number and it can then be used in the same manner as the number itself, in which case it will represent the value of the last number assigned to it.

As a simple example:

```
A = 3
B = 4
PRINT A + B
```

will display the number 7. In this case both A and B are variables and MMBasic used their current values in the PRINT statement. MMBasic will automatically create a variable when it first encounters it, so the statement A = 3 both created a floating point variable (the default type) with the name of A and then it assigned the value of 3 to it.

The name of a variable must start with a letter while the remainder of the name can use letters, numbers, the underscore or the full stop (or period) characters. The name can be up to 31 characters long and the case (ie, capitals or not) is not important. Here are some examples:

```
Total_Count
ForeColour
temp3
count
x
ThisIsAVeryLongVariableName
increment.value
```

You can change the value of a variable anywhere in your program by using the assignment command, ie:

```
variable = expression
```

For example:

```
temp3 = 24.6
count = 5
CTemp = (FTemp - 32) * 0.5556
```

In the last example both CTemp and FTemp are variables and this line converts the value of FTemp (in degrees Fahrenheit) to degrees Celsius and stores the result in the variable CTemp.

Expressions

We have met the term 'expression' before in this tutorial and in programming it has a specific meaning. It is a formula which can be resolved by the BASIC interpreter to a single number or value.

MMBasic will evaluate numeric expressions using the same rules that we learnt at school. For example, multiplication and division are performed first followed by addition and subtraction. These are called the rules of precedence and are fully described previously in this manual (see the chapter *Expressions and Operators*).

This means that MMBasic will resolve $2 + 3 * 6$ by first multiplying 3 by 6 giving 18 then adding 2 resulting in a final value of 20. Similarly, both $5 * 4$ and $10 + 4 * 3 - 2$ will also resolve to 20.

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 intention.

As pointed out earlier, you can use variables in an expression exactly the same as straight numbers. For example, this will increment the value of the variable `temp` by one:

```
temp = temp + 1
```

You can also use functions in expressions. These are special operations provided by MMBasic, for example to calculate trigonometric values.

An example of using a function is the following which will print the length of the hypotenuse of a right angled triangle. This uses the `SQR()` function which returns the square root of a number (`a` and `b` are variables holding the lengths of the other sides):

```
PRINT SQR(a * a + b * b)
```

MMBasic will first evaluate this expression by multiplying `a` by `a`, then multiplying `b` by `b`, then adding the results together. The resulting number is then passed to the `SQR()` function which will calculate the square root of that number (ie, the hypotenuse) and return it for the `PRINT` command to display.

Some other mathematical functions provided by MMBasic include:

`SIN(r)` – the sine of `r`

`COS(r)` – the cosine of `r`

`TAN(r)` – the tangent of `r`

There are many more functions available to you and they are all listed earlier in this manual.

Note that in the above trigonometric functions the value passed to the function (ie, '`r`') is the angle in radians. In MMBasic you can use the function `RAD(d)` to convert an angle from degrees to radians ('`d`' is the angle in degrees).

Another feature of most programming languages (including BASIC) is that you can nest function calls within each other. For example, given the angle in degrees (ie, '`d`') the sine of that angle can be found with this expression:

```
PRINT SIN(RAD(d))
```

In this case MMBasic will first take the value of `d` and convert it to radians using the `RAD()` function. The output of this function then becomes the input to the `SIN()` function.

The IF Statement

Making decisions is at the core of most computer programs and in BASIC this is usually done with the `IF` statement. This is written almost like an English sentence:

```
IF condition THEN action
```

The *condition* is usually a comparison such as equals, less than, more than, etc.

For example:

```
IF Temp < 25 THEN PRINT "Cold"
```

`Temp` would be a variable holding the current temperature (in °C) and `PRINT "Cold"` the action to be done.

There are a range of tests that you can make:

=	equals	<>	not equal
<	less than	<=	less than or equals

> greater than >= greater than or equals

You can also add an ELSE clause which will be executed if the initial condition tested false:

```
IF condition THEN true-action ELSE false-action
```

For example, this will execute different actions when the temperature is under 25 or 25 or more:

```
IF Temp < 25 THEN PRINT "Cold" ELSE PRINT "Hot"
```

The previous examples all used single line IF statements but you can also use a multiline IF statement. They look like this:

```
IF condition THEN
    true-action
    true-action
ENDIF
IF condition THEN
    true-action
    true-action
ELSE
    false-action
    false-action
ENDIF
```

Unlike the single line IF statement you can have many true actions with each on their own line and similarly many false actions. Generally the single line IF statement is handy if you have a simple action that needs to be taken while the multiline version is much easier to understand if the actions are numerous and more complicated.

An example of a multiline IF statement with more than one action is:

```
IF Amount < 100 THEN
    PRINT "Too low"
    PRINT "Minimum value is 100"
ELSE
    PRINT "Input accepted"
    SaveToSDCard
    PRINT "Enter second amount"
ENDIF
```

Note that in the above example each action is indented to show what part of the IF structure it belongs to. Indenting is not mandatory but it makes a program much easier to understand for someone who is not familiar with it and therefore it is highly recommended.

In a multiline IF statement you can make additional tests using the ELSE IF command. This is best explained by using an example (the temperatures are all in °C):

```
IF Temp < 0 THEN
    PRINT "Freezing"
ELSE IF Temp < 20 THEN
    PRINT "Cold"
ELSE IF Temp < 35 THEN
    PRINT "Warm"
ELSE
    PRINT "Hot"
ENDIF
```

The ELSE IF uses the same tests as an ordinary IF (ie, <, <=, etc) but that test will only be made if the preceding test was false. So, for example, you will only get the message *Warm* if Temp < 0 failed, and Temp < 20 failed but Temp < 35 was true. The final ELSE will catch the case where all the tests were false.

An expression like Temp < 20 is evaluated by MMBasic as either true or false with true having a value of one and false zero. You can see this if you entered the following at the console:

```
PRINT 30 > 20
```

MMBasic will print 1 meaning that the value of the expression is true.

Similarly the following will print 0 meaning that the expression evaluated to false.

```
PRINT 30 < 20
```

The IF statement does not really care about what the condition actually is, it just evaluates the condition and if the result is zero it will take that as false and if non zero it will take it as true.

This allows for some handy shortcuts. For example, if `BalanceCorrect` is a variable that is true (non zero) when some feature of the program is correct then the following can be used to make a decision based on that value:

```
IF BalanceCorrect THEN ...do something...
```

FOR Loops

Another common requirement in programming is repeating a set of actions. For instance, you might want to step through all seven days in the week and perform the same function for each day. BASIC provides the FOR loop construct for this type of job and it works like this:

```
FOR day = 1 TO 7
    Do something based on the value of 'day'
NEXT day
```

This starts by creating the variable `day` and assigning the value of 1 to it. The program will then execute the following statements until it comes to the NEXT statement. This tells the BASIC interpreter to increment the value of `day`, go back to the previous FOR statement and re-execute the following statements a second time. This will continue looping around until the value of `day` exceeds 7 and the program will then exit the loop and continue with the statements following the NEXT statement.

As a simple example, you can print the numbers from one to ten like this:

```
FOR nbr = 1 TO 10
    PRINT nbr, ;
NEXT nbr
```

The comma at the end of the PRINT statement tells the interpreter to tab to the next tab column after printing the number and the semicolon will leave the cursor on this line rather than automatically moving to the next line. As a result, the numbers will be printed in neat columns across the page.

This is what you would see:

```
1         2         3         4         5         6         7         8         9
10
```

The FOR loop also has a couple of extra tricks up it sleeves. You can change the amount that the variable is incremented by using the STEP keyword. So, for example, the following will print just the odd numbers:

```
FOR nbr = 1 TO 10 STEP 2
    PRINT nbr, ;
NEXT nbr
```

The value of the step (or increment value) defaults to one if the STEP keyword is not used but you can set it to whatever number you want.

When MMBasic is incrementing the variable it will check to see if the variable has exceeded the TO value and, if it has, it will exit from the loop. So, in the above example, the value of `nbr` will reach nine and it will be printed but on the next loop `nbr` will be eleven and at that point execution will leave the loop. This test is also applied at the start of the loop. For example, if in the beginning the value of the variable exceeds the TO value, the loop will never be executed, not even once.

By setting the STEP value to a negative number you can use the FOR loop to step down from a high number to low. In that case the starting number must be greater than the TO number.

For example, the following will print the numbers from 1 to 10 in reverse:

```
FOR nbr = 10 TO 1 STEP -1
    PRINT nbr, ;
NEXT nbr
```

Multiplication Table

To further illustrate how loops work and how useful they can be, the following short program uses two FOR loops to print out the multiplication table that we all learnt at school. The program for this is not complicated:

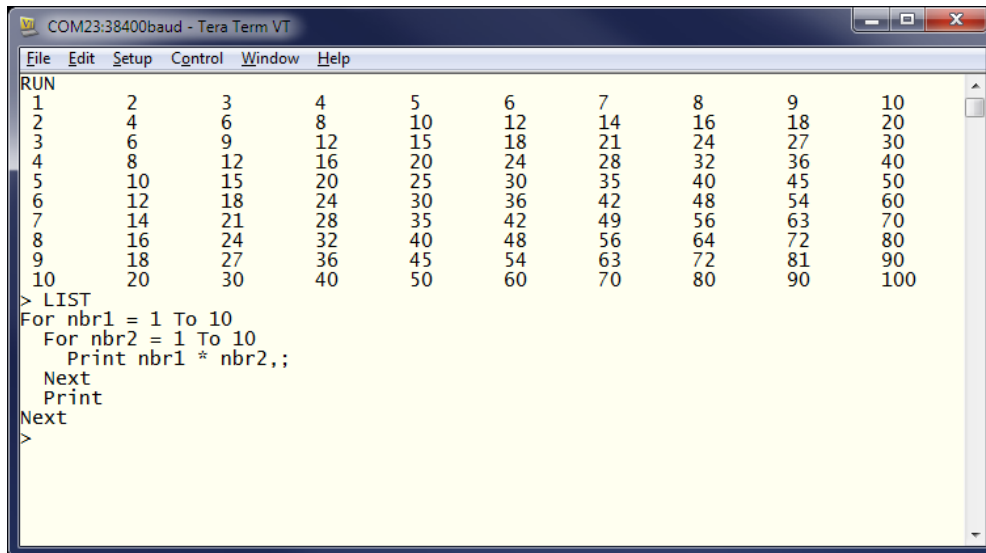
```
FOR nbr1 = 1 to 10
    FOR nbr2 = 1 to 10
```

```

    PRINT nbr1 * nbr2,;
NEXT nbr2
PRINT
NEXT nbr1

```

The output is shown in the following screen grab, which also shows a listing of the program.



```

COM23:38400baud - Tera Term VT
File Edit Setup Control Window Help
RUN
1      2      3      4      5      6      7      8      9      10
2      4      6      8      10     12     14     16     18     20
3      6      9      12     15     18     21     24     27     30
4      8      12     16     20     24     28     32     36     40
5      10     15     20     25     30     35     40     45     50
6      12     18     24     30     36     42     48     54     60
7      14     21     28     35     42     49     56     63     70
8      16     24     32     40     48     56     64     72     80
9      18     27     36     45     54     63     72     81     90
10     20     30     40     50     60     70     80     90     100
> LIST
For nbr1 = 1 To 10
  For nbr2 = 1 To 10
    Print nbr1 * nbr2,;
  Next
  Print
Next
>

```

You need to work through the logic of this example line by line to understand what it is doing. Essentially it consists of one loop inside another. The inner loop, which increments the variable `nbr2` prints one horizontal line of the table. When this loop has finished it will execute the following `PRINT` command which has nothing to print - so it will simply output a new line (ie, terminate the line printed by the inner loop).

The program will then execute another iteration of the outer loop by incrementing `nbr1` and reexecuting the inner loop again. Finally, when the outer loop is exhausted (when `nbr1` exceeds 10) the program will reach the end and terminate.

One last point, you can omit the variable name from the `NEXT` statement and MMBasic will guess which variable you are referring to. However, it is good practice to include the name to make it easier for someone else who is reading the program to understand it. You can also terminate multiple loops using a comma separated list of variables in the `NEXT` statement. For example:

```

FOR var1 = 1 TO 5
  FOR var2 = 10 to 13
    PRINT var1 * var2
  NEXT var1, var2

```

DO Loops

Another method of looping is the `DO...LOOP` structure which looks like this:

```

DO WHILE condition
  <statement>
  <statement>
LOOP

```

This will start by testing the *condition* and if it is true the statements will be executed until the `LOOP` command is reached, at which point the program will return to `DO` statement and the *condition* will be tested again, and if it is still true the loop will execute again. The condition is the same as in the `IF` command (ie, `X < Y`).

For example, the following will keep printing the word "Hello" on the console for 4 seconds then stop:

```

Timer = 0
DO WHILE Timer < 4000
  PRINT "Hello"
LOOP

```

Note that `Timer` is a function within MMBasic which will return the time in milliseconds since the timer was reset. A reset is done by assigning zero to `Timer` (as done above) or when powering up the Colour Maximite 2.

A variation on the `DO-LOOP` structure is the following:

```

DO
    <statement>
    <statement>
LOOP UNTIL condition

```

In this arrangement the loop is first executed once, the *condition* is then tested and if the condition is false, the loop will be repeatedly executed until the *condition* becomes true. Note that the test in LOOP UNTIL is the inverse of DO WHILE.

For example, similar to the previous example, the following will also print "Hello" for four seconds:

```

Timer = 0
DO
    PRINT "Hello"
LOOP UNTIL Timer >= 4000

```

Both forms of the DO-LOOP essentially do the same thing, so you can use whatever structure fits with the logic that you wish to implement.

Finally, it is possible to have a DO Loop that has no conditions at all - ie,

```

DO
    <statement>
    <statement>
LOOP

```

This construct will continue looping forever and you, as the programmer, will need to provide a way to explicitly exit the loop (the EXIT DO command will do this). For example:

```

Timer = 0
DO
    PRINT "Hello"
    IF Timer >= 4000 THEN EXIT DO
LOOP

```

Console Input

As well as printing data for the user to see your programs will also want to get input from the user. For that to work you need to capture keystrokes from the console and this can be done with the INPUT command. In its simplest form the command is:

```
INPUT var
```

This command will print a question mark on the console's screen and wait for a number to be entered followed by the Enter key. That number will then be assigned to the variable `var`.

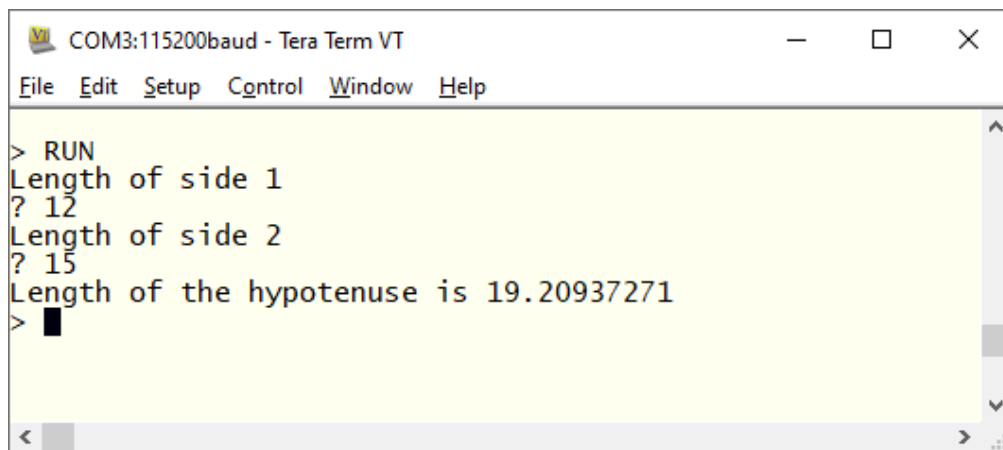
For example, the following program extends the expression for finding the hypotenuse of a triangle by allowing the user to enter the lengths of the other sides from the console.

```

PRINT "Length of side 1"
INPUT a
PRINT "Length of side 2"
INPUT b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)

```

This is a screen capture of a typical session:



The INPUT command can also print your prompt for you, so that you do not need a separate PRINT command. For example, this will work the same as the above program:

```
INPUT "Length of side 1"; a
INPUT "Length of side 2"; b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

Finally, the INPUT command will allow you to input a series of numbers separated by commas with each number being saved in different variables.

For example:

```
INPUT "Enter the length of the two sides: ", a, b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

If the user entered 12, 15 the number 12 would be saved in the variable a and 15 in b.

Another method of getting input from the console is the LINE INPUT command. This will get the whole line as typed by the user and allocate it to a string variable. Like the INPUT command you can also specify a prompt. This is a simple example:

```
LINE INPUT "What is your name? ", s$
PRINT "Hello " s$
```

We will cover string variables later in this tutorial but for the moment you can think of them as a variable that holds a sequence of characters. If you ran the above program and typed in John when prompted the program would respond with Hello John.

Sometimes you do not want to wait for the user to hit the enter key, you want to get each character as it is typed in. This can be done with the INKEY\$ function which will return the value of the character as a string consisting of just one character or an empty string (ie, contains no characters) if nothing has been entered.

GOTO and Labels

One method of controlling the flow of the program is the GOTO command. This essentially tells MMBasic to jump to another part of the program and continue executing from there. The target of the GOTO is a label and this needs to be explained first.

A label is an identifier that marks part of the program. It must be the first thing on the line and it must be terminated with the colon (:) character. The name that you use can be up to 31 characters long and must follow the same rules as for a variable's name. For example, in the following program line LoopBack is a label:

```
LoopBack: a = a + 1
```

When you use the GOTO command to jump to that particular part of the program you would use the command like this:

```
GOTO LoopBack
```

To put all this into context the following program will print out all the numbers from 1 to 10:

```
z = 0
LoopBack: z = z + 1
PRINT z
IF z < 10 THEN GOTO LoopBack
```

The program starts by setting the variable z to zero then incrementing it to 1 in the next line. The value of z is printed and then tested to see if it is less than 10. If it is less than 10 the program execution will jump back to

the label `LoopBack` where the process will repeat. Eventually the value of `z` will reach 10 and the program will run off the end and terminate.

Note that a `FOR` loop can do the same thing (and is simpler), so this example is purely designed to illustrate what the `GOTO` command can do.

In the past the `GOTO` command gained a bad reputation. This is because using `GOTO`s it is possible to create a program that continuously jumps from one point to another (often referred to as "spaghetti code") and that type of program is almost impossible for another programmer to understand. With constructs like the multiline `IF` statements the need for the `GOTO` statement has been reduced and it should be used only when there is no other way of changing the program's flow.

Testing for Prime Numbers

The following is a simple program which brings together many of the programming features previously discussed.

```
DO
  InpErr:
  PRINT
  INPUT "Enter a number: "; a
  IF a < 2 THEN
    PRINT "Number must be equal or greater than 2"
    GOTO InpErr
  ENDIF

  Divs = 0
  FOR x = 2 TO SQR(a)
    r = a/x
    IF r = FIX(r) THEN Divs = Divs + 1
  NEXT x

  PRINT a " is ";
  IF Divs > 0 THEN PRINT "not ";
  PRINT "a prime number."
LOOP
```

This will first prompt (on the console) for a number and, when it has been entered, it will test if that number is a prime number or not and display a suitable message.

It starts with a `DO` Loop that does not have a condition – so it will continue looping forever. This is what we want. It means that when the user has entered a number, it will report if it is a prime number or not and then loop around and ask for another number. The way that the user can exit the program (if they wanted to) is by typing the break character (normally `CTRL-C`).

The program then prints a prompt for the user which is terminated with a semicolon character. This means that the cursor is left at the end of the prompt for the `INPUT` command which will get the number and store it in the variable `a`.

Following this the number is tested. If it is less than 2 an error message will be printed and the program will jump backwards and ask for the number again.

We are now ready to test if the number is a prime number. The program uses a `FOR` loop to step through the possible divisors testing if each one can divide evenly into the entered number. Each time it does the program will increment the variable `Divs`.

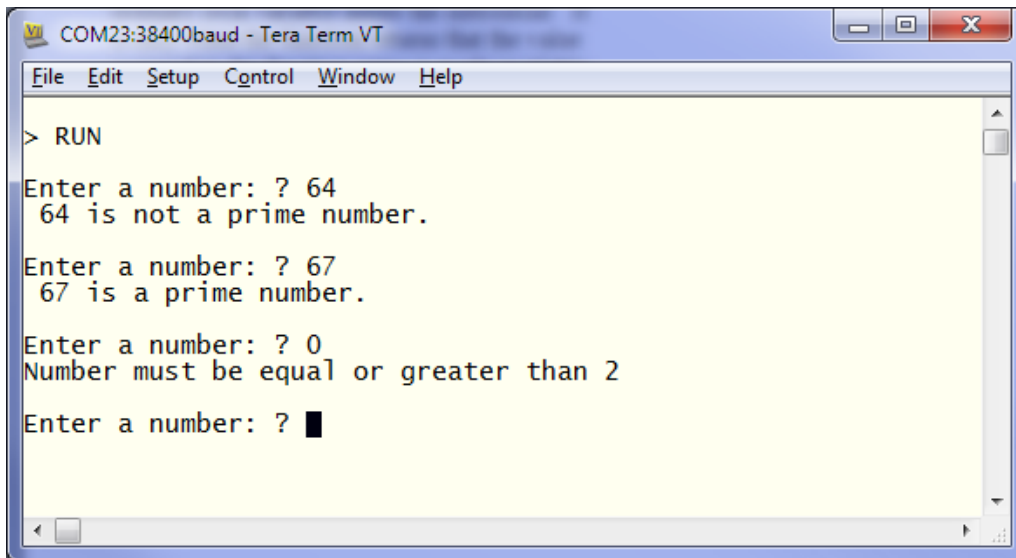
Note that the test is done with the function `FIX(r)` which simply strips off any digits after the decimal point. So, the condition `r = FIX(r)` will be true if `r` is an integer (ie, has no digits after the decimal point).

Finally, the program will construct the message for the user. The key part is that if the variable `Divs` is greater than zero it means that one or more numbers were found that could divide evenly into the test number. In that case the `IF` statement inserts the word "not" into the output message.

For example, if the entered number was 21 the user will see this response:

```
21 is not a prime number.
```

This is the result of running the program and some of the output:



You can test this program by using the editor (the EDIT command) to enter it.

Using your newly learnt skills you could then have a shot at making it more efficient. For example, because the program counts how many times a number can be divided into the test number it takes a lot longer than it should to detect a non prime number. The program would run much more efficiently if it jumped out of the FOR loop at the first number that divided evenly. You could use the GOTO command to do this or you could use the command EXIT FOR – that would cause the FOR loop to terminate immediately.

Other efficiencies include only testing the division with odd numbers (by using an initial test for an even number then starting the FOR loop at 3 and using STEP 2) or by only using prime numbers for the test (that would be much more complicated).

Arrays



Arrays are something which you will probably not think of as useful at first glance but when you do need to use them you will find them very handy indeed.

An array is best thought of as a row of letterboxes for a block of units or condos as shown on the right. The letterboxes are all located at the same address and each box represents a unit or condo at that address. You can place a letter in the box for unit one, or unit two, etc.

Similarly an array in BASIC is a single variable with multiple sub units (called *elements* in BASIC) which are numbered. You can place data in element one, or element two, etc. In BASIC an array is created by the DIM command, for example:

```
DIM numarr(300)
```

This creates an array with the name of `numarr` containing 301 elements (think of them as letterboxes) ranging from 0 to 300. By default an array will start from zero so this is why there is an extra element making the total 301. To specify a specific element in the array (ie, a specific letterbox) you use an index which is simply the

number of the array element that you wish to access. For example, if you want to set element number 100 in this array to (say) the number 876, you would do it this way:

```
numarr(100) = 876
```

Normally the index to an array is not a constant number as in this example (ie, 100) but a variable which can be changed to access different array elements.

As an example of how you might use an array, consider the case where you would like to record the maximum temperature for each day of the year and, at the end of the year, calculate the overall average. You could use ordinary variables to record the temperature for each day but you would need 365 of them and that would make your program quite unwieldy. Instead, you could define an array to hold the values like this:

```
DIM days(365)
```

Every day you would need to save the temperature in the correct location in the array. If the number of the day in the year was held in the variable `doy` and the maximum temperature was held in the variable `maxtemp` you would save the reading like this:

```
days(doy) = maxtemp
```

At the end of the year it would be simple to calculate the average for the year. For example:

```
total = 0
FOR i = 1 to 365
    total = total + days(i)
```



```
    NEXT i
PRINT "Average is:" total / 365
```

This is much easier than adding up and averaging 365 individual variables.

The above array was single dimensioned but you can have multiple dimensions. Reverting to our analogy of letterboxes, an array with two dimensions could be thought of as a block of flats with multiple floors. A block could have a row of four letter boxes for level one, another row of four boxes for level two, and so on. To place a letter in a letterbox you need to specify the floor number and the unit number on that floor.

In BASIC such an array is specified using two indices separated by a comma. For example:

```
LetterBox(floor, unit)
```

As a practical example, assume that you needed to record the maximum temperature for each day over five years. To do this you could dimension the array as follows:

```
DIM days(365, 5)
```

The first index is the day in the year and the second is a number representing the year. If you wanted to set day 100 in year 3 to 24 degrees you would do it like this:

```
days(100, 3) = 24
```

In MMBasic you can have up to five dimensions. The size of an array is limited only by the amount of free RAM that is available.

A complete array can be copied into another array with a single assignment by using the array names with empty brackets (eg, `b() = a()`). Both arrays must be the same type and hold the same total number of elements.

Integers

So far all the numbers and variables that we have been using have been floating point. As explained before, floating point is handy because it will track digits after the decimal point and when you use division it will

return a sensible result. So, if you just want to get things done and are not concerned with the details you should stick to floating point.

However, the limitation of floating point is that it stores numbers as an approximation with an accuracy of 14 digits in MMBasic. Most times this characteristic of floating point numbers is not a problem but there are some cases where you need to accurately store larger numbers.

As an example, let us say that you want to manipulate time accurately down to the microsecond so that you can compare two different date/times to work out which one is earlier. The easy way to do this is to convert the date/time to the number of microseconds since some date (say 1st Jan in year zero) - then finding the earliest of the two is just a matter of using an arithmetic compare in an IF statement.

The problem is that the number of microseconds since that date will exceed the accuracy range of floating point variables and this is where integer variables come in. An integer variable can accurately hold very large numbers up to nine million million million (or ± 9223372036854775807 to be precise).

The downside of using an integer is that it cannot store fractions (ie, numbers after the decimal point). Any calculation that produces a fractional result will be rounded up or down to the nearest whole number when assigned to an integer. However this characteristic can be handy when dealing with money – for example, you don't want to send someone a bill for \$100.133333333333.

It is easy to create an integer variable, just add the percent symbol (%) as a suffix to a variable name. For example, `sec%` is an integer variable. Within a program you can mix integers and floating point and MMBasic will make the necessary conversions but if you want to maintain the full accuracy of integers you should avoid mixing the two.

Just like floating point you can have arrays of integers, all you need to do is add the percent character as a suffix to the array name. For example: `days%(365, 5)`.

Beginners often get confused as to when they should use floating point or integers and the answer is simple... always use floating point unless you need an extremely high level of accuracy. This does not happen often but when you need them you will find that integers are quite useful.

Strings

Strings are another variable type (like floating point and integers). Strings are used to hold a sequence of characters. For example, in the command:

```
PRINT "Hello"
```

The string "Hello" is a string constant. Note that a constant is something that does not change (as against a variable, which can) and that string constants are always surrounded by double quotes.

String variables names use the dollar symbol (\$) as a suffix to identify them as a string instead of a normal floating point variable and you can use ordinary assignment to set their value. The following are examples (note that the second example uses an array of strings):

```
Car$ = "Holden"
Country$(12) = "India"
Name$ = "Fred"
```

You can also join strings using the plus operator:

```
Word1$ = "Hello"
Word2$ = "World"
Greeting$ = Word1$ + " " + Word2$
```

In which case the value of `Greeting$` will be "Hello World".

Strings can also be compared using operators such as = (equals), <> (not equals), < (less than), etc. For example:

```
IF Car$ = "Holden" THEN PRINT "Was an Aussie made car"
```

The comparison is made using the full ASCII character set so a space will come before a printable character. Also the comparison is case sensitive so 'holden' will not equal "Holden". Using the function `UCASE()` to convert the string to upper case you can then have a case insensitive comparison. For example:

```
IF UCASE$(Car$) = "HOLDEN" THEN PRINT "Was an Aussie made car"
```

You can have arrays of strings but you need to be careful when you declare them as you can rapidly run out of RAM (general memory used for storing variables, etc). This is because MMBasic will by default allocate 255 bytes of RAM for each element of the array. For example, a string array with 100 elements will by default use 25K of RAM.

To alleviate this you can use the LENGTH qualifier to limit the maximum size of each element. For instance, if you know that the maximum length of any string that will be stored in the array will be less than 20 characters you can use the following declaration to allocate just 20 bytes for each element:

```
DIM MyArray$(100) LENGTH 20
```

The resultant array will only use 2K of RAM.

You can use the LENGTH qualifier when declaring a normal (non array) string variable and it will save RAM: a string declared with a short length is held in the variable table itself instead of in a 256 byte block.

Manipulating Strings

String handling is one of MMBasic's strengths and using a few simple functions you can pull apart and generally manipulate strings. The basic string functions are:

LEFT\$(string\$, nbr)	Returns a substring of <i>string\$</i> with <i>nbr</i> of characters from the left (beginning) of the string.
RIGHT\$(string\$, nbr)	Same as the above but return <i>nbr</i> of characters from the right (end) of the string.
MID\$(string\$, pos, nbr)	Returns a substring of <i>string\$</i> with <i>nbr</i> of characters starting from the character <i>pos</i> in the string (ie, the middle of the string).

For example if S\$ = "This is a string"

then:	R\$ = LEFT\$(S\$, 7)	would result in the value of R\$ being set to: "This is"
and:	R\$ = RIGHT\$(S\$, 8)	would result in the value of R\$ being set to: "a string"
finally:	R\$ = MID\$(S\$, 6, 2)	would result in the value of R\$ being set to: "is"

Note that in MID\$() the first character position in a string is number 1, the second is number 2 and so on. So, counting the first character as one, the sixth position is the start of the word "is".

Another useful function is:

INSTR(string\$, pattern\$)	Returns a number representing the position at which <i>pattern\$</i> occurs in <i>string\$</i> .
-----------------------------	--

This can be used to search for a string inside another string. The number returned is the position of the substring inside the main string. Like with MID\$() the start of the string is position 1.

For example if S\$ = "This is a string"

Then: pos = INSTR(S\$, " ")

would result in pos being set to the position of the first space in S\$ (ie, 5).

INSTR() can be combined with other functions so this would return the first **word** in S\$:

```
R$ = LEFT$(S$, INSTR(S$, " ") - 1)
```

There is also an extended version of INSTR():

INSTR(pos, string\$, pattern\$)	Returns a number representing the position at which <i>pattern\$</i> occurs in <i>string\$</i> when starting the search at the character position <i>pos</i> .
----------------------------------	--

So we can find the second word in S\$ using the following:

```
pos1 = INSTR(S$, " ")
pos2 = INSTR(pos1 + 1, S$, " ")
r$ = MID$(S$, pos1 + 1, pos2 - pos1)
```

This last example is rather complicated so it might be worth working through it in detail so that you can understand how it works.

Note that INSTR() will return the number zero if the sub string is not found and that any string function will throw an error (and halt the program) if that is used as a character position. So, in a practical program you would first check for zero being returned by INSTR() before using that value.

For example:

```
r$ = ""
pos1 = INSTR(S$, " ")
```

```

if pos1 > 0 THEN
    pos2 = INSTR(pos1 + 1, S$, " ")
    if pos2 > 0 THEN r$ = MID$(S$, pos1 + 1, pos2 - pos1)
ENDIF

```

Scientific Notation

Before we finish discussing data types we need to cover off the subject of floating point numbers and scientific notation.

Most numbers can be written normally, for example 11 or 24.5, but very large or small numbers are more difficult. For example, it has been estimated that the number of grains of sand on planet Earth is 7500000000000000000. The problem with this number is that you can easily lose track of how many zeros there are in the number and consequently it is difficult to compare this with a similar sized number.

A scientist would write this number as 7.5×10^{18} which is called scientific notation and is much easier to comprehend.

When using the PRINT command MMBasic will automatically use scientific notation when printing very large or small floating point numbers. For example, if the above number was stored in a floating point variable the PRINT command would display it as 7.5E+18 (this is BASIC's way of representing 7.5×10^{18}). As another example, the number 0.000000456 would display as 4.56E-8 which is the same as 4.56×10^{-8} .

You can also use scientific notation when entering constant numbers in MMBasic. For example:

```
SandGrains = 7.5E+18
```

MMBasic only uses scientific notation for floating point numbers (not integers). For instance, if you assigned the number of grains of sand to an integer variable the PRINT command would display it as a normal number (with lots of zeros).

DIM Command

We have used the DIM command before for defining arrays but it can also be used to create ordinary variables. For example, you can simultaneously create four string variables like this:

```
DIM STRING Car, Name, Street, City
```

Note that because these variables have been defined as strings using the DIM command we do not need the \$ suffix, the definition alone is enough for MMBasic to identify their type. Similarly, when you use these variables in an expression you do not need the type suffix: For example:

```
City = "Sydney"
```

You can also use the keyword INTEGER to define a number of integer variables and FLOAT to do the same for floating point variables. This type of notation can similarly be used to define arrays.

For example:

```
DIM INTEGER seconds(200)
```

Another method of defining the variables type is to use the keyword AS. For example:

```
DIM Car AS STRING, Name AS STRING, Street AS STRING
```

This is the method used by Microsoft (MMBasic tries to maintain Microsoft compatibility) and it is useful if the variables have different types. For example:

```
DIM Car AS STRING, Age AS INTEGER, Value AS FLOAT
```

You can use any of these methods of defining a variable's type, they all act the same.

The advantage of defining variables using the DIM command is that they are clearly defined (preferably at the start of the program) and their type (float, integer or string) is not subject to misinterpretation.

You can strengthen this by using the following commands at the very top of your program:

```

OPTION EXPLICIT
OPTION DEFAULT NONE

```

The first specifies to MMBasic that all variables must be explicitly defined using DIM before they can be used. The second specifies that the type of all variables must be specified when they are created.

Why are these two commands important?

The first can help avoid a common programming error which is where you accidentally misspell a variable's name. For example, your program might have the current temperature saved in a variable called Temp but at

one point you accidentally misspell it as `Tmp`. This will cause MMBasic to automatically create a variable called `Tmp` and set its value to zero.

This is obviously not what you want and it will introduce a subtle error which could be hard to find, even if you were aware that something was not right. On the other hand, if you used the `OPTION EXPLICIT` command at the start of your program MMBasic would refuse to automatically create the variable and instead would display an error thereby saving you from a probable headache.

The command `OPTION DEFAULT NONE` further helps because it tells MMBasic that the programmer must specifically specify the type of every variable when they are declared. It is easy to forget to specify the type and allowing MMBasic to automatically assume the type can lead to unexpected consequences.

For small, quick and dirty programs, it is fine to allow MMBasic to automatically create variables but in larger programs you should always disable this feature with `OPTION EXPLICIT` and strengthen it with `OPTION DEFAULT NONE`.

When a variable is created it is set to zero for float and integers and an empty string (ie, contains no characters) for a string variable. You can set its initial value to something else when it is created using `DIM`.

For example:

```
DIM FLOAT nbr = 12.56
DIM STRING Car = "Ford", City = "Perth"
```

You can also initialise arrays by placing the initialising values inside brackets like this:

```
DIM s$(2) = ("zero", "one", "two")
```

Note that because arrays start from zero by default this array actually has three elements with the index numbers of 0, 1 and 2. This is why we needed three string constants to initialise it.

Constants

A common requirement in programming is to define an identifier that represents a value without the risk of the value being accidentally changed - which can happen if variables were used for this purpose. These are called constants and they can represent I/O pin numbers, signal limits, mathematical constants and so on.

You can create a constant using the `CONST` command. This defines an identifier that acts like a variable but is set to a value that cannot be changed.

For example, if you wanted to check the voltage of a battery connected to pin 31 you could define the relevant values thus:

```
CONST BatteryVoltagePin = 31
CONST BatteryMinimum = 1.5
```

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

```
SETPIN BatteryVoltagePin, AIN
IF PIN(BatteryVoltagePin) < BatteryMinimum THEN SoundAlarm
```

It is good programming practice to use constants for any fixed number that represents an important value. Normally they are defined at the start of a program where they are easy to see and conveniently located for another programmer to adjust (if necessary).

Subroutines

A subroutine is a block of programming code which is self contained (like a module) and can be called from anywhere within your program. To your program it looks like a built in MMBasic command and can be used the same. For example, assume that you need a command that would signal an error by printing a message on the console. You could define the subroutine like this:

```
SUB ErrMsg
  PRINT "Error detected"
END SUB
```

With this subroutine embedded in your program all you have to do is use the command `ErrMsg` whenever you want to display the message. For example:

```
IF A < B THEN ErrMsg
```

The definition of a subroutine can be anywhere in the program but typically it is at the end. If MMBasic runs into the definition while running your program it will simply skip over it.

The above example is fine enough but it would be better if a more useful message could be displayed, one that could be customised every time the subroutine was called. This can be done by passing a string to the subroutine as an argument (sometimes called a parameter).

In this case the definition of the subroutine would look like this:

```
SUB ErrMsg Msg$
  PRINT "Error: " + Msg$
END SUB
```

Then, when you call the subroutine, you can supply the string to be printed on the command line of the subroutine.

For example:

```
IF A < B THEN ErrMsg "Number too small"
```

When the subroutine is called like this the message "Error: Number too small" will be printed on the console. Inside the subroutine `Msg$` will have the value of "Number too small" when called like this and it will be concatenated in the `PRINT` statement to make the full error message.

A subroutine can have any number of arguments which can be float, integer or string with each argument separated by a comma.

Within the subroutine 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 within the subroutine and be different from arguments defined for the subroutine.

The type of the argument to be supplied can be specified with a type suffix (ie, \$, % or ! for string, integer and float). For example, in the following the first argument must be a string and the second an integer:

```
SUB MySub Msg$, Nbr%
...
END SUB
```

MMBasic will convert the supplied values if it can, so if your program supplied a floating point value as the second argument MMBasic will convert it to an integer. If MMBasic cannot convert the value it will display an error and return to the command prompt. For example, if you supplied a string for the second argument your program will stop with an error.

You do not have to use the type suffixes, you can instead define the type of the arguments using the `AS` keyword similar to the way it is used in the `DIM` command.

For example, the following is identical to the above example:

```
SUB MySub Msg AS STRING, Nbr AS INTEGER
...
END SUB
```

Of course, if you used only one variable type throughout the program and used `OPTION DEFAULT` to set that type you could ignore the question of variable types completely.

When a subroutine is called with an argument that is a variable (ie, not a constant or expression) MMBasic will create a corresponding variable within the subroutine *that points back to this variable*. Any changes to the variable representing the argument inside the subroutine will also change the variable used in the call. This is called passing arguments by reference.

This is best explained by example:

```
DIM MyNumber = 5      ' set the variable to 5
CalcSquare MyNumber    ' the subroutine will square its value
PRINT MyNumber         ' this will print the number 25
END

SUB CalcSquare nbr
  nbr = nbr * nbr      ' square the argument and pass it back
END SUB
```

The subroutine `CalcSquare` will take its argument, square it and write it back to the variable representing the argument (`nbr`). Because the subroutine was called with a variable (`MyNumber`) the variable `nbr` will point

back to `MyNumber` and any change to `nbr` will also change `MyNumber` accordingly. As a result the `PRINT` statement will output 25.

Passing arguments by reference is handy because it allows a subroutine to pass values back to the code that called it. However, it could lead to trouble if a subroutine used the variable representing an argument as a general purpose variable and changed its value. Then, if it were called with a variable as an argument, that variable would be inadvertently changed. To avoid this, you should prefix its definition with the keyword `BYVAL`. This instructs MMBasic to always use the value of the argument, even if it is a variable, and to never point back to the variable used in the call.

When you call a subroutine you can omit some (or all) of the parameters and they will take the value of zero (for floats or integers) or an empty string. This is handy as your subroutine can tell if a parameter is missing and act accordingly.

For example, here is our subroutine to generate an error message but this version can be used without specifying an error message as a parameter:

```
SUB ErrMsg Msg$
  IF Msg$ = "" THEN
    PRINT "Error detected"
  ELSE
    PRINT "Error: " + Msg$
  ENDIF
END SUB
```

Within a subroutine you can use most features of MMBasic including calling other subroutines, `IF...THEN` commands, `FOR...NEXT` loops and so on. However, one thing that you cannot do is jump out of a subroutine using `GOTO` (if you do the result will be undefined and may cause your hair to turn grey).

Normally the subroutine will exit when the `END SUB` command is reached but you can also terminate the subroutine early by using the `EXIT SUB` command.

Functions

Functions are similar to subroutines with the main difference being that a function is used to return a value in an expression. For example, if you wanted a function to convert a temperature from degrees Celsius to Fahrenheit you could define:

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

Then you could use it in an expression:

```
Input "Enter a temperature in Celsius: ", t
PRINT "That is the same as" Fahrenheit(t) "F"
```

Or as another example:

```
IF Fahrenheit(temp) <= 32 THEN PRINT "Freezing"
```

You could also define the reverse:

```
FUNCTION Celsius(F)
  Celsius = (F - 32) * 0.5556
END FUNCTION
```

As you can see, the function name is used as an ordinary local variable inside the subroutine. It is only when the function returns that the value is made available to the expression that called it.

The rules for the argument list in a function are similar to subroutines. The only difference is that parentheses are always required around the argument list when you are calling a function, even if there are no arguments (parentheses 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 type suffix (ie, \$, a % or a !) the function will return that type (string, integer or float), otherwise it will return whatever the `OPTION DEFAULT` is set to. For example, the following function will return a string:

```
FUNCTION LVal$(nbr)
  IF nbr = 0 THEN LVal$ = "False" ELSE LVal$ = "True"
```

```
END FUNCTION
```

You can explicitly specify the type of the function by using the AS keyword and then you do not need to use a type suffix (similar to defining a variable using DIM).

This is the above example rewritten to take advantage of this feature:

```
FUNCTION LVal(nbr) AS STRING
    IF nbr = 0 THEN LVal = "False" ELSE LVal = "True"
END FUNCTION
```

In this case the type returned by the function LVal will be a string.

A function is not limited to returning a single value - it can also return a whole array. This is described in the Subroutines and Functions chapter earlier in this manual.

As for subroutines you can use most features of MMBasic within functions. This includes FOR...NEXT loops, calling other functions and subroutines, etc. Also, the function will return to the expression that called it when the END FUNCTION command is reached but you can also return early by using the EXIT FUNCTION command.

Local Variables

Variables that are created using DIM or that are just automatically created are called *global* variables. This means that they can be seen and used anywhere in the program including within subroutines and functions. However, inside a subroutine or function you will often need to use variables for various tasks that are internal to the subroutine/function. In portable code you do not want the name you chose for such a variable to clash with a global variable of the same name. To this end you can define a variable using the LOCAL command within the subroutine/function.

The syntax for LOCAL is identical to the DIM command, this means that the variable can be an array, you can set the type of the variable and you can initialise it to some value.

For example, this is our ErrMsg subroutine but this time it has been extended to use a local variable for joining the error message strings.

```
SUB ErrMsg Msg$
    LOCAL STRING tstr
    tstr = "Error: " + Msg$
    PRINT tstr
END SUB
```

The variable tstr is declared as LOCAL within the subroutine, which means that (like the argument list) it will only exist within the subroutine and will vanish when the subroutine exits. You can have a global variable called tstr in your main program and it will be different from the variable tstr in the subroutine (in this case the global tstr will be hidden within the subroutine).

You should always use local variables for operations within your subroutine or function because they help make the module much more self contained and portable.

Static Variables

LOCAL variables are reset to their initial values (normally zero or an empty string) every time the subroutine or function starts, however there are times when you would like the variable to retain its value between calls. This type of variable is defined with the STATIC command.

We can demonstrate how STATIC variables are useful by extending the ErrMsg subroutine to prevent duplicated calls to the subroutine repeatedly displaying the same message. For example, our program might call this subroutine from multiple places but if the message is the same in a number of subsequent calls we would like to see the message just once.

This is our new subroutine:

```
SUB ErrMsg Msg$
    STATIC STRING lastmsg
    LOCAL STRING tstr
    IF Msg$ <> lastmsg THEN
        tstr = "Error: " + Msg$
        PRINT tstr
        lastmsg = Msg$
    ENDIF
END SUB
```

To keep track of the last message displayed we use a static variable called `lastmsg`. This will hold the text of the last message and we can compare it to the current message text to determine if it is different and therefore should be printed. This would give just one message every time a call is made with a duplicate message text. The `STATIC` command uses exactly the same syntax as `DIM`. This means that you can define different types of static variables including arrays and you can also initialise them to some value.

The static variable is created on the first time the `STATIC` command is encountered and it is automatically set to zero (if a float or integer) or an empty string. On subsequent calls to the subroutine or function `MMBasic` will recognise that the variable has already been created and it will leave its value untouched (ie, whatever it was in the previous call). As with `DIM` you can also initialise a static variable to some value. For example:

```
STATIC INTEGER var = 123
```

On the first call (when the variable is created) it will be initialised to 123 but on subsequent calls it will keep whatever its value was previously set to.

Mostly static variables are used to keep track of the *state* of something while inside a subroutine or function. A *state* is a record of something that has happened previously.

Examples include:

- Has the COM port already been opened?
- What steps in a sequence have we completed?
- What text has already been displayed?

Normally you will use global variables (created using `DIM`) to track a *state* but sometimes you want this to be contained within a module and this is where static variables are valuable. Just like `LOCAL` the use of `STATIC` helps to make your subroutines and functions more self contained and portable.

Calculate Days

We have covered a lot of programming commands and techniques so far in this tutorial and before we finish it would be worth giving an example of how they work together. The following is an example that uses many features of the BASIC language to calculate the number of days between two dates:

```
' Example program to calculate the number of days between two dates

OPTION EXPLICIT
OPTION DEFAULT NONE

DIM STRING s
DIM FLOAT d1, d2

DO
  ' main program loop
  PRINT : PRINT "Enter the date as  dd mmm yyyy"
  PRINT " First date";
  INPUT s
  d1 = GetDays(s)
  IF d1 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Second date";
  INPUT s
  d2 = GetDays(s)
  IF d2 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Difference is" ABS(d2 - d1) " days"
LOOP

' Calculate the number of days since 1/1/1900
FUNCTION GetDays(d$) AS FLOAT
  LOCAL STRING Month(11) =
  ("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov","dec")
  LOCAL FLOAT Days(11) = (0,31,59,90,120,151,181,212,243,273,304,334)
  LOCAL FLOAT day, mth, yr, s1, s2

  ' Find the separating space character within a date
  s1 = INSTR(d$, " ")
```

```

IF s1 = 0 THEN EXIT FUNCTION
s2 = INSTR(s1 + 1, d$, " ")
IF s2 = 0 THEN EXIT FUNCTION

' Get the day, month and year as numbers
day = VAL(MID$(d$, 1, s2 - 1)) - 1
IF day < 0 OR day > 30 THEN EXIT FUNCTION
FOR mth = 0 TO 11
    IF LCASE$(MID$(d$, s1 + 1, 3)) = Month(mth) THEN EXIT FOR
NEXT mth
IF mth > 11 THEN EXIT FUNCTION
yr = VAL(MID$(d$, s2 + 1)) - 1900
IF yr < 1 OR yr >= 200 THEN EXIT FUNCTION

' Calculate the number of days including adjustment for leap years
GetDays = (yr * 365) + FIX((yr - 1) / 4)
IF yr MOD 4 = 0 AND mth >= 2 THEN GetDays = GetDays + 1
GetDays = GetDays + Days(mth) + day
END FUNCTION

```

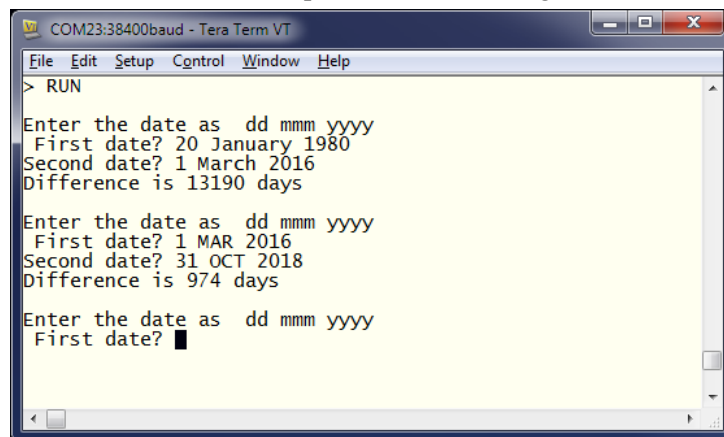
Note that the line starting `LOCAL STRING Month(11)` has been wrapped around because of the limited page width – it is one line as follows:

```

LOCAL STRING Month(11) =
("jan", "feb", "mar", "apr", "may", "jun", "jul", "aug", "sep", "oct", "nov", "dec")

```

This program works by getting two dates from the user at the console and then converting them to the number of days since 1900. With these two numbers a simple subtraction will give the number of days between them.



When this program is run it will ask for the two dates to be entered and you need to use the form of: `dd mmm YYYY`.

The screen capture on the right shows what the running program will look like.

The main feature of the program is the defined function `GetDays()` which takes a string entered at the console, splits it into its day, month and year components then calculates the number of days since 1st January 1900.

This function is called twice, once for the first date and then again for the second date. It is then just a matter of subtracting one date (in days) from the other to get the difference in days.

We will not go into the detail of how the calculations are made (ie, handling leap years) as that can be left as an exercise for the reader. However, it is appropriate to point out some features of MMBasic that are used by the program.

It demonstrates how local variables can be used and how they can be initialised. In the function `GetDays()` two arrays are declared and initialised at the same time. These are just a convenient method of looking up the names of the months and the cumulative number of days for each month. Later in the function (the FOR loop) you can see how they make dealing with twelve different months quite efficient.

Another feature highlighted by this program is the string handling features of MMBasic. The `INSTR()` function is used to locate the two space characters in the date string and then later the `MID$()` function uses these to

extract the day, month and year components of the date. The VAL() function is used to turn a string of digits (like the year) into a number that can be stored in a numeric variable.

Note that the value of a function is initialised to zero every time the function is called and unless it is set to some value it will return a zero value. This makes error handling easy because we can just exit the function if an error is discovered. It is then the responsibility of the calling program code to check for a return value of zero which signifies an error.

This program illustrates one of the benefits of using subroutines and functions which is that when written and fully tested they can be treated as a trusted "black box" that does not have to be opened. For this reason functions like this should be properly tested and then, if possible, left untouched (in case you add some error).

There are a few features of this program that we have not covered before. The first is the MOD operator which will calculate the remainder of dividing one number into another. For example, if you divided 4 into 15 you would have a remainder of 3 which is exactly what the expression 15 MOD 4 will return. The ABS() function is also new and will return its argument as a positive number (eg, ABS(-15) will return +15 as will ABS(15)).

The EXIT FOR command will exit a FOR loop even though it has not reached the end of its looping, EXIT FUNCTION will immediately exit a function even though execution has not reached the end of the function and CONTINUE DO will immediately cause the program to jump to the end of a DO loop and execute it again.

Why would this program be useful? Well some people like to count their age in days, that way every day is a birthday! You can calculate your age in days, just enter the date that you were born and today's date. That is not particularly useful but the program itself is valuable as it demonstrates many of the characteristics of programming in MMBasic. So, work your way through the program and review each section until you understand it – it should be a rewarding experience.

Appendix M – Firmware Applicability

The same firmware runs on both generations of the Colour Maximite 2. Where an entry in Commands, Functions, Options or Predefined Read Only Variables is not available everywhere, the restriction is written as an underlined label at the start of its description. An entry with no label is available on both generations.

These are the labels, and what each means:

Label	Meaning
GENERATION 2 ONLY	Available on the Generation 2 Colour Maximite 2 only (it needs its larger memory, its 24-bit colour or its extra hardware: for example the 32-bit colour depth, MODE 16 and OPTION DS3231).
IN THE 4-BIT COLOUR DEPTH ONLY	Available only in a 4-bit mode (MODE n,4: the PicoMite's 16 colour display).
IN THE 12-BIT COLOUR DEPTH ONLY	Available only in a 12-bit mode, the one with transparency.

No other wording is used for applicability, so that a reader can find every restriction by looking for these labels.

Appendix N – Where Each Keyword Is Documented

Every keyword of the language, and the section of this manual where it is described. Where a feature also has a manual of its own, that is named too (see Appendix O). The options are in the section Options, under OPTION.

Keyword	Where it is documented
#COMMENT END	Commands
#COMMENT START	Commands
#DEFINE	Commands
#INCLUDE	Commands
#MMDEBUG OFF	Commands
#MMDEBUG ON	Commands
*	the chapter Operators
+	the chapter Operators
-	the chapter Operators
/	the chapter Operators
<	the chapter Operators
<<	the chapter Operators
<=	the chapter Operators
<>	the chapter Operators
=	the chapter Operators
>	the chapter Operators
>=	the chapter Operators
>>	the chapter Operators
>>>	the chapter Operators
?	another name for PRINT (Commands)
@(	Commands, with PRINT
ABS(	Functions
ACOS(	Functions
ADC	Commands
AND	the chapter Operators
ARC	Commands
ARRAY ADD	Commands
ARRAY INSERT	Commands
ARRAY SET	Commands
ARRAY SLICE	Commands
AS	Commands, with DIM, LOCAL, OPEN and the other statements that use it
ASC(	Functions
ASIN(	Functions
ATAN2(	Functions
ATN(	Functions
AUTOSAVE	Commands
BASE\$(	Functions
BAUDRATE(	Functions
BEZIER	Commands

BIN\$(	Functions
BIN2STR\$(	Functions
BIT(	Commands, Functions
BITBANG	Commands, Obsolete Commands and Functions
BITSTREAM	Commands
BLIT	Commands
BOUND(	Functions
BOX	Commands
BYTE(	Commands, Functions
CALL	Commands
CALL(	Functions
CAN	Commands
CASE	Commands
CASE ELSE	Commands, with SELECT CASE
CAT	another name for INC (Commands)
CHAIN	Commands
CHDIR	Commands
CHOICE(	Functions
CHR\$(	Functions
CINT(	Functions
CIRCLE	Commands
CLASSIC(	Functions
CLEAR	Commands
CLICK(	Functions
CLOSE	Commands
CLS	Commands
COLOR	another name for COLOUR (Commands)
COLOUR	Commands
CONST	Commands
CONTINUE	Commands
CONTROLLER	Commands
COPY	Commands
COS(	Functions
CPU	Commands
CSUB	Commands
CTRLVAL(	Functions; GUI Controls and Programming.pdf
CWD\$	Functions
DAC	Commands
DATA	Commands
DATES	Commands, Functions
DATETIME\$(	Functions
DAYS\$(	Functions
DEFINEFONT	Commands
DEG(	Functions

DEVICE(	Functions
DHT22	Commands, Obsolete Commands and Functions
DIM	Commands
DIR\$(	Functions
DISTANCE(	Functions
DO	Commands
DRAW3D	Commands; The 3D engine.pdf
DRAW3D(	Functions; The 3D engine.pdf
DRIVE	Commands
EDIT	Commands
ELSE	Commands
ELSE IF	another name for ELSEIF (Commands)
ELSEIF	Commands
END	Commands
END CSUB	Commands
END DEFINEFONT	Commands
END FUNCTION	Commands
END IF	another name for ENDIF (Commands)
END SELECT	Commands, with SELECT CASE
END SUB	Commands
END TYPE	Commands
ENDIF	Commands
EOF(	Functions
EPOCH(	Functions
ERASE	Commands
ERROR	Commands
EVAL(	Functions
EXECUTE	Commands
EXIT	Commands
EXIT DO	another name for EXIT (Commands)
EXIT FOR	Commands
EXIT FUNCTION	Commands
EXIT SUB	Commands
EXP(	Functions
FIELD\$(	Functions
FILES	Commands
FILL	Commands
FIX(	Functions
FLAG(	Commands, Functions
FLAGS	Commands
FLASH	Commands
FLUSH	Commands
FM	Commands; FM_User_Manual.pdf

FONT	Commands
FOR	Commands
FORMAT\$(	Functions
FRAMEBUFFER	Commands
FUNCTION	Commands
GAMEPAD	Commands
GETSCANLINE	Functions
GOSUB	Commands, Obsolete Commands and Functions
GOTO	Commands
GPS(	Functions
GUI	Commands; GUI Controls and Programming.pdf
HELP	Commands
HEX\$(	Functions
HUMID	Commands
I2C	Commands
I2C2	Commands
I2C3	Commands
IF	Commands, Obsolete Commands and Functions
IMAGE	Commands
INC	Commands
INKEY\$	Functions
INPUT	Commands
INPUT\$(	Functions
INSTR(	Functions
INT(	Functions
INTERRUPT	Commands
INV	the chapter Operators
IR	Commands
IRETURN	Commands, Obsolete Commands and Functions
JSON\$(	Functions
KEYDOWN(	Functions
KILL	Commands
LCASE\$(	Functions
LCOMPARE(	Functions
LEFT\$(	Functions
LEN(	Functions
LET	Commands
LGETBYTE(	Functions
LGETSTR\$(	Functions
LIBRARY	Commands
LINE	Commands
LINE INPUT	Commands
LINPUT(	Functions
LINSTR(	Functions

LIST	Commands
LLEN(	Functions
LMID(	Commands
LOAD	Commands
LOC(	Functions
LOCAL	Commands
LOF(	Functions
LOG(	Functions
LONGSTRING	Commands
LOOP	Commands
MAP	Commands
MAP(	Commands, Functions
MATH	Commands, Functions
MATH(	Functions
MAX(	Functions
MEMORY	Commands
MID\$(	Commands, Functions
MIN(	Functions
MKDIR	Commands
MM.CMDLINES\$	Predefined Read Only Variables
MM.DEVICES\$	Predefined Read Only Variables
MM.END	the chapter Using MMBasic, and Commands, with END
MM.ERRLINE	Predefined Read Only Variables
MM.ERRMSG\$	Predefined Read Only Variables
MM.ERRNO	Predefined Read Only Variables
MM.FLAGS	Predefined Read Only Variables
MM.FONTHEIGHT	Predefined Read Only Variables
MM.FONTWIDTH	Predefined Read Only Variables
MM.HEIGHT	Predefined Read Only Variables
MM.HPOS	Predefined Read Only Variables
MM.HRES	Predefined Read Only Variables
MM.I2C	Predefined Read Only Variables
MM.INFO\$	another name for MM.INFO (Predefined Read Only Variables)
MM.INFO(	Predefined Read Only Variables
MM.ONEWIRE	Predefined Read Only Variables
MM.VER	Predefined Read Only Variables
MM.VPOS	Predefined Read Only Variables
MM.VRES	Predefined Read Only Variables
MM.WATCHDOG	Predefined Read Only Variables
MM.WIDTH	Predefined Read Only Variables
MMDEBUG	Commands
MOD	the chapter Operators
MODE	Commands

MOUSE	Commands
MOUSE(	Functions
MSGBOX(	Functions; GUI Controls and Programming.pdf
NEW	Commands
NEXT	Commands
NOT	the chapter Operators
NUNCHUK(	Functions
OCT\$(	Functions
ON	Commands, Obsolete Commands and Functions
ONEWIRE	Commands
OPEN	Commands
OPTION	Commands, Options
OR	the chapter Operators
PAGE	Commands
PAUSE	Commands
PEEK(	Functions
PI	Functions
PIN(	Commands, Functions
PIXEL	Commands
PIXEL FAST	another name for CLS (Commands)
PIXEL(	Functions
PLAY	Commands
POKE	Commands
POLYGON	Commands
PORT(	Commands, Functions
POS	Obsolete Commands and Functions
PRINT	Commands
PULSE	Commands
PULSIN(	Functions
PWM	Commands
RAD(	Functions
RAM	Commands
RANDOMIZE	Commands
RBOX	Commands
READ	Commands
REDIM	Commands
REM	Commands
RENAME	Commands
RESTORE	Commands
RETURN	Commands, Obsolete Commands and Functions
RGB(	Functions
RIGHT\$(	Functions
RMDIR	Commands
RND	Functions

RND(	Functions
RUN	Commands
SAVE	Commands
SEEK	Commands
SELECT CASE	Commands
SERVO	Commands
SETPIN	Commands
SETTICK	Commands
SGN(	Functions
SIN(	Functions
SORT	Commands
SPACE\$(	Functions
SPI	Commands, Functions
SPI(	Functions
SPI2	Commands, Functions
SPI2(	Functions
SPRITE	Commands (the sprite commands: SPRITE is another name for BLIT), and Functions, SPRITE(
SPRITE(	Functions
SQR(	Functions
STATIC	Commands
STEP	Commands, with FOR
STR\$(	Functions
STR2BIN(	Functions
STRING\$(	Functions
STRUCT	Commands; MMBasic_Structures_Manual.pdf
STRUCT(	Functions; MMBasic_Structures_Manual.pdf
SUB	Commands
SYNC	Commands
TAB(	Functions
TAN(	Functions
TEMPR START	Commands
TEMPR(	Functions
TEXT	Commands
THEN	Commands, with IF
TILEMAP	Commands; TILEMAP_User_Manual.pdf
TILEMAP(	Functions; TILEMAP_User_Manual.pdf
TIMES	Commands, Functions
TIMER	Commands, Functions
TO	Commands, with FOR
TRACE	Commands
TRIANGLE	Commands
TRIM\$(	Functions

TURTLE	Commands
TYPE	Commands; MMBasic_Structures_Manual.pdf
UCASE\$(	Functions
UNTIL	Commands, with DO
UPDATE FIRMWARE	Commands
VAL(	Functions
VAR	Commands
WATCHDOG	Commands
WHILE	Commands, with DO
WII	another name for CONTROLLER (Commands)
WS2812	Commands
XMODEM	Commands
XOR	the chapter Operators
ZMODEM	Commands
\	the chapter Operators
^	the chapter Operators

Appendix O – Companion Manuals

Some features are large enough to have a manual of their own rather than an entry in the listings. Some of these were written for the Colour Maximite 2 and are included in its firmware distribution zip file; others were written for the PicoMite and describe features that the Colour Maximite 2 shares with it. Appendix N says which keyword is described where.

Manual	Covers	Notes
GUI Controls and Programming.pdf	The GUI controls - buttons, switches, LEDs, check boxes, spin boxes, text and number boxes, gauges and the date, time and position controls - together with CTRLVAL() and MSGBOX().	The Colour Maximite 2's own.
The 3D engine.pdf	The DRAW3D command and function: cameras, objects, rotation and hidden line removal.	The Colour Maximite 2's own.
Graphics Programming on the CMM2	Pages, layers, sprites and the other advanced graphics features, written for games developers (also on the Back Shed forum).	The Colour Maximite 2's own.
MMBasic_Structures_Manual.pdf	User defined types: TYPE, END TYPE, STRUCT and the STRUCT() function.	The PicoMite's; the same on the Colour Maximite 2.
FM_User_Manual.pdf	The two panel file manager opened with the FM command.	The PicoMite's. The chapter File Manager says what differs here (the drives, and no mouse yet).
TILEMAP_User_Manual.pdf	Building scrolling backgrounds with the TILEMAP command and function.	The PicoMite's; the same on the Colour Maximite 2.
PLAY_SAMPLE_User_Manual.pdf	Playing sampled sound effects with PLAY SAMPLE.	The PicoMite's; the same on the Colour Maximite 2.

USB_CDC_Host_User_Manual.pdf	Using USB serial devices.	The PicoMite's: its COM3 to COM6 are COM4 to COM7 here.
mmb2csub.pdf	The mmb2csub tool, which turns a BASIC SUB or FUNCTION into a CSUB without writing any C, so that the busiest routines of a program run as machine code.	Use --target cmm2 to build CSUBs for the Colour Maximate 2.