Micromite eXtreme

User Manual
Micromite eXtreme
Ver 0.20
for
MMBasic Ver 5.07.00b4

Draft Ver:0.20

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

About

The Micromite eXtreme was conceived and developed by Peter Mather (matherp on the Back Shed Forum).

It is based on the Micromite 2/Micromite Plus developed by Geoff Graham and uses the MMBasic interpreter written by Geoff Graham (http://geoffg.net).

Support

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

Copyright and Acknowledgments

This Manual

The assembler of this manual is Doug Pankhurst. It is a compilation of manuals for the Micromite 2/Micromite Plus/Micromite eXtreme/Maximite 2/ArmMite, all created by Geoff Graham, Peter Mather and Gerry Allerdice.

Conventions used throughout this manual are:

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

Additional information or qualifier about the current subject.

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

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

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

Contents

The Micromite eXtreme

Introduction

This section provides an introduction for users who are familiar with the Micromite and the Micromite Plus and need a summary of the extra features in the Micromite eXtreme and Micromite eXtreme64.

The Micromite eXtreme is an extension of the standard Micromite and the Micromite Plus; all features of these two versions are also in the Micromite eXtreme. This includes features of the BASIC language, input/output, communications, etc. Some commands have changed slightly (for example the CPU command) but for the main part Micromite programs will run unchanged on the Micromite eXtreme. The following summarises additional features in the Micromite eXtreme as compared to the standard Micromite and the Micromite Plus:

PIC32 MZ Processor

The Micromite eXtreme is based on the Microchip PIC32MZ 32 bit microcontroller. This chip is available in 64, 100 and 144-pin surface mount packages and is up to five times faster and has up to ten times the program space of the MX series used in the standard Micromite.

High Speed Double Precision Floating Point

The Micromite eXtreme uses the built in hardware floating point capability of the PIC32MZ which is much faster than floating point on the standard Micromite and uses double precision floating point.

I/O Pins

The 64-pin Micromite eXtreme has 46 free I/O pins with 24 analogue capable. The 100-pin Micromite eXtreme has up to 71 free I/O pins with 40 analog capable and the 144-pin chip has up to 115 free I/O pins with 50 analogue capable. All analogue pins use a 12-bit analogue to digital conversion rather than 10-bit on the standard Micromite.
The Micromite eXtreme has two I^{2}C ports, three SPI ports, six PWM channels and up to four serial COM ports. All serial COM ports are high speed (over 1,000,000 baud).
The Micromite eXtreme64 has one I^{2}C and two SPI ports but is otherwise the same.

High Speed LCD Panels

Like the Micromite Plus the Micromite eXtreme supports ten different sized LCD display panels from 1.44" to 9". In addition it can drive displays using the SSD1963 controller in 16-bit parallel mode to achieve an even greater display update speed. There is also a range of drivers for displays that use a memory based framebuffer. This allows complex updates to be written whilst minimising screen flashing and tearing.

VGA Output

The Micromite eXtreme can drive a VGA display in 640 x 480 pixels or 640 x 400 (widescreen) with eight colours. All the graphics commands and GUI controls available in the Micromite Plus will also work on the VGA output. This capability is not available on the Micromite eXtreme64.

Mouse Input

The Micromite eXtreme can support a PS2 mouse which can be used to activate on screen GUI controls. This feature will work with touch sensitive LCD displays (it works in parallel with the touch sensitivity) but it is especially useful with VGA monitors that do not normally incorporate a touch sensitive surface.

USB Keyboard Input

The Micromite eXtreme can support a USB keyboard. The software supports UK and US layouts (see OPTION USBKEYBOARD) and is automatically enabled when a keyboard is plugged in.

Sprites

The Micromite eXtreme supports a complete implementation of sprites including screen scrolling and collision detection

High Speed Frequency Counter

The Micromite eXtreme supports a high speed frequency counter tested up to 10MHz.

Micromite Family Summary

The Micromite Family consists of three major types, the standard Micromite, the Micromite Plus and the Micromite eXtreme. All use the same BASIC interpreter and have the same basic capabilities however they differ in the number of I/O pins, the amount of memory, the displays that they support and their intended use.

Standard Micromite Comes in a 28-pin or 44-pin package and is designed for small embedded controller applications and supports small LCD display panels. The 28-pin version is particularly easy to use as it is easy to solder and can be plugged into a standard 28-pin IC socket.

Micromite Plus This uses a 64-pin and 100-pin TQFP surface mount package and supports a wide range of touch sensitive LCD display panels from 1.44" to 8" in addition to the standard features of the Micromite. It is intended as a sophisticated controller with easy to create on-screen controls such as buttons, switches, etc.

Micromite eXtreme This comes in 64, 100-pin and 144-pin TQFP surface mount packages. The eXtreme version has all the features of the other two Micromites but is faster and has a larger memory capacity plus the ability to drive a VGA monitor for a large screen display. It works as a powerful, self contained computer with its own BASIC interpreter and instant start-up.

	Micromite		Micromite Plus		Micromite eXtreme		
	28-pin DIP	44-pin SMD	64-pin SMD	100-pin SMD	100-pin SMD	144-pin SMD	64-pin SMD
Maximum CPU Speed	48 MHz	48 MHz	120 MHz	120 MHz	252MHz	252 MHz	252 MHz
Maximum BASIC Program Size	59 KB	59 KB	100 KB	100 KB	540 KB	540 KB	540 KB
RAM Memory Size	52 KB	52 KB	108 KB	108 KB	460 KB	460 KB	460 KB
Clock Speed (MHz)	5 to 48	5 to 48	5 to 120	5 to 120	200 to 252	200 to 252	200 to 252
Total Number of I/O pins	19	33	45	77	75	115	46
Number of Analog Inputs	10	13	28	28	40	48	24
Number of Serial I/O ports	2	2	3 or 4	3 or 4	3 or 4	3 or 4	3 or 4
Number of SPI Channels	1	1	2	2	3	3	2
Number of I^2C Channels	1	1	1 + RTC	1 + RTC	2 + RTC	2 + RTC	1 + RTC
Number of 1-Wire I/O pins	19	33	45	77	75	115	46
PWM or Servo Channels	5	5	5	5	6	6	6
Serial Console	✓	✓	✓	✓	✓	✓	✓
USB Console			✓	✓	✓	✓	✓
PS2 Keyboard and LCD Console			✓	✓	✓	✓	✓
SD Card Interface			✓	✓	✓	✓	✓
Supports ILI9341 LCD Displays	✓	✓	✓	✓	✓	✓	✓
Supports Ten LCD Panels from 1.44" to 8" (diameter)			✓	✓	✓	✓	✓
Supports VGA Displays					✓	✓	
Sound Output (WAV/tones)			✓	✓	✓	✓	✓
Supports PS2 Mouse Input					✓	✓	✓
Floating Point Precision	Single	Single	Double S/W	Double S/W	Double H/W	Double H/W	Double H/W
Power Requirements	3.3V 30 mA	3.3V 30 mA	3.3V 80 mA	3.3V 80 mA	3.3V 160 mA	3.3V 160 mA	3.3V 160 mA

Micromite eXtreme Hardware

Suitable Microcontrollers

The microcontroller used in the Micromite eXtreme is the PIC32MZ EF series manufactured by Microchip. There are two chip sizes (100-pin and 144-pin) with two frequency specifications (200 MHz and 252 MHz).

The default clock speed of the Micromite eXtreme is 200 MHz however this can be changed with a configuration option to 252 MHz if required. The firmware will automatically adjust for either the 100 or 144 pin version.

The recommended chips are:

PIC32MZ2048EF064I/PT	64-pin TQFP package (0.5 mm pin pitch) – maximum speed 200 MHz
PIC32MZ2048EFH064-250I/PT	64-pin TQFP package (0.5 mm pin pitch) – maximum speed 252 MHz
PIC32MZ2048EFG100-I/PF	100-pin TQFP package (0.5 mm pin pitch) – maximum speed 200 MHz
PIC32MZ2048EFG144-I/PL	144-pin LQFP package (0.5 mm pin pitch) – maximum speed 200 MHz
PIC32MZ2048EFH100-250I/PF	100-pin TQFP package (0.5 mm pin pitch) – maximum speed 252 MHz
PIC32MZ2048EFH144-250I/PL	144-pin LQFP package (0.5 mm pin pitch) – maximum speed 252 MHz

The chips PIC32MZ2048EFM064-I/PT , PIC32MZ2048EFM100-I/PF and PIC32MZ2048EFM144-I/PL can also be used at 200 MHz. In addition the 0.4mm chips may be used but are harder to solder by hand (PT for 100-pin and PH for 144-pin)

See http://microchip.com for the data sheets.

Suitable Platforms

100-pin Test and Development Board

The best development board for the 100-pin 200 MHz chip is the SnadPIC MZ, PIC32MZ EF MCU Starter Kit. This can be ordered with either the PIC32MZ2048EFG100-I/PF or PIC32MZ2048EFH100-I/PF processor (both are similar, the latter has a CAN facility but that is not supported by MMBasic).

If you are developing your own board it would be worth using the SnadPIC board for guidance.

SnadPIC boards can be purchased from:

http://www.microcontroller-board.com/snadpic-board-32-bit/24-snadpic-mz-pic32mz-ef-mcu-starter-kit-pic32mz2048efg100.html
or
http://www.ebay.com.au/itm/PIC32MZ-USB-OTG-Microchip-Development-Board-Starter-kit-SD-Card-SnadPIC-MZ-/181805050475

64, 100 and 144 Pin Backpack PCBs

Backpack boards have been developed by Peter Mather for the 64, 100 and 144 pin versions.

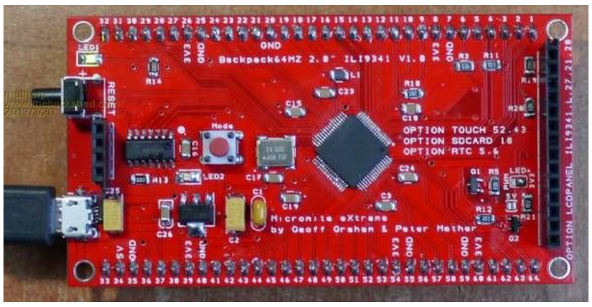
These are described in this post on TheBackshed Forum together with schematics and gerbers for those wishing to build their own versions.

64-pin Backpack Board

This board, developed by Peter Mather is configured as a backpack for the 2.8 inch ILI9341 LCD display.

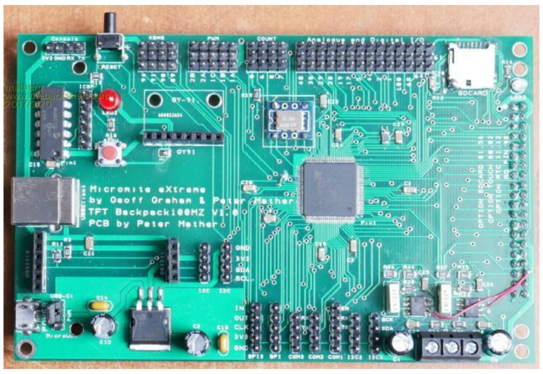
The main difference between this version and the 100/144 pin versions below are:-

- Only one I2C port
- Only two SPI ports
- No VGA support
- No 16-bit parallel SSD1963 support



100-pin Backpack Board

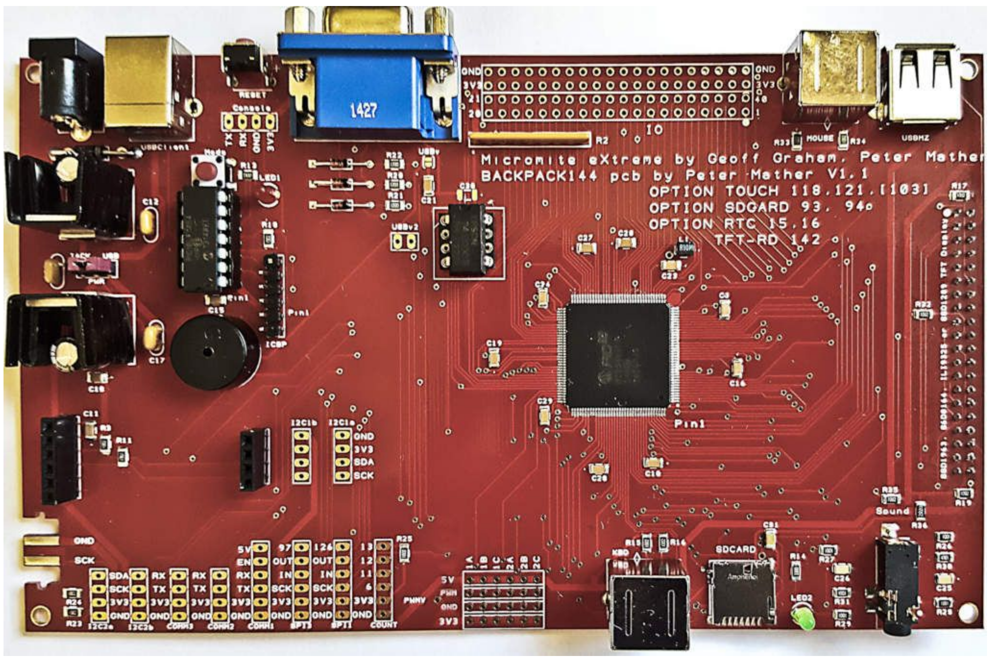
There are 2 versions that Peter developed – both have provision for the SSD1963 LCD. Additionally, the black PCB shown has provision for a GPS module and an accelerometer module for use as a navigational platform.



144-pin Backpack Board

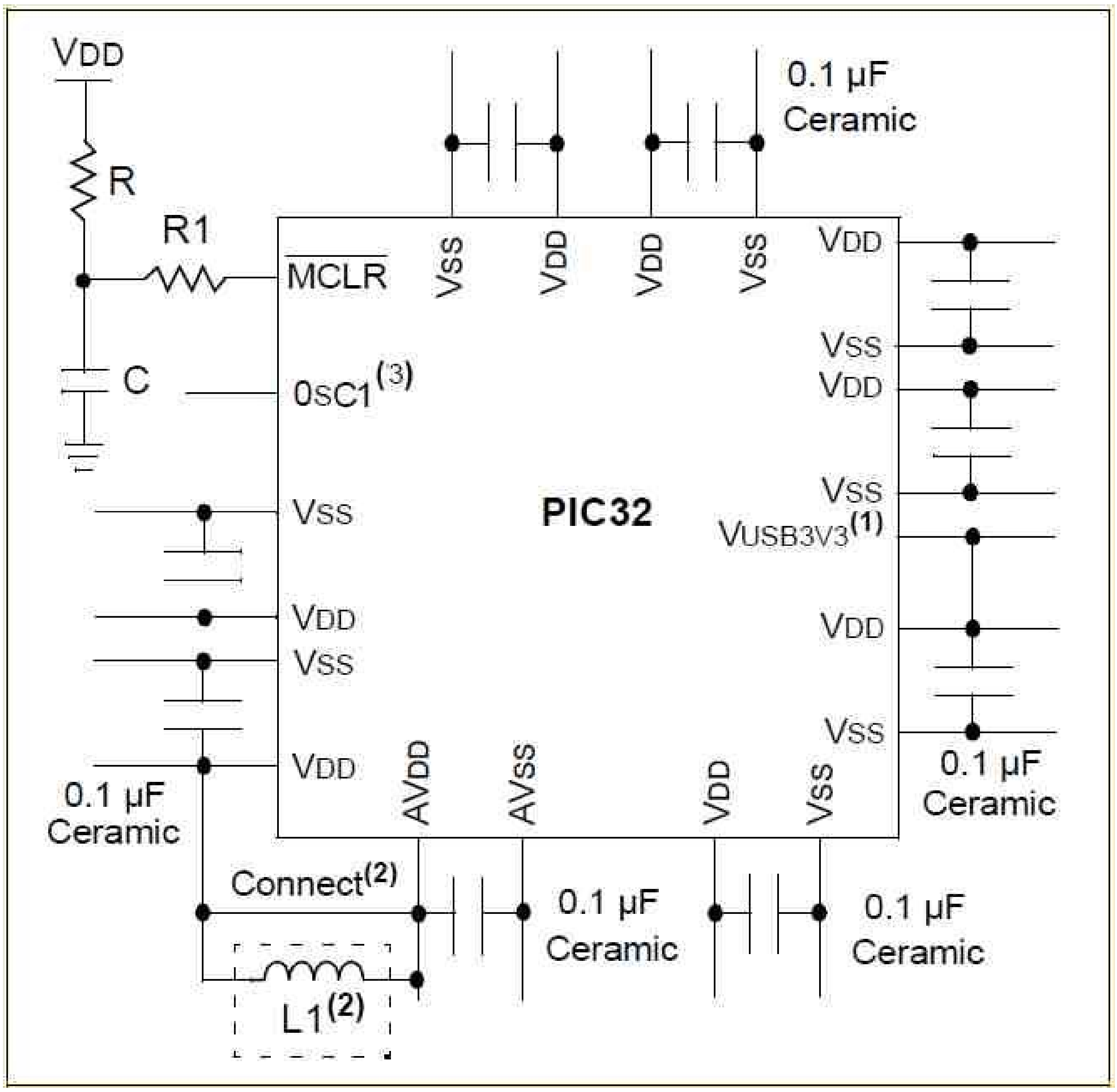
This board was also developed by Peter Mather. Schematic is shown in Annex H. and can also be downloaded from TheBackshed Forum It is a complete module and includes all the connectors for: VGA, TFT, PS2 keyboard, PS2 mouse, NunChuck, 3.5mm stereo sound, and USB. It has an onboard USB-to-UART (PIC16F1654), and sockets for an RTC module. It can be used standalone, or can be mounted directly onto the back of a 7" LCD display panel.

Descriptions of use and interfacing of the Micromite eXtreme in this manual are based on this 144pin Backpack PCB.



Typical Circuit

An example of the required circuit for a Micromite eXtreme chip is given below:



Notes:

1. If the USB module is not used, this pin may be connected to VSS.
2. As an option, instead of a hard-wired connection, an inductor (L1) can be substituted between VDD and AVDD to improve ADC noise rejection. The inductor impedance should be less than 1ohm and the inductor capacity greater than 10 mA. Alternatively a 10ohm resistor could be substituted for L1. Typical values for R, R1, and C would be 10Kohm, 1Kohm, and 0.1uF
3. A 24MHz crystal oscillator must be connected to the OSC1 pin. e.g. Epson SG8002DCPHB24MHZ. See the pinout below for the pin number

Programming the Firmware

Programming the 64, 100 and 144-pin Micromite eXtreme is similar to programming the 28-pin standard Micromite described in the Micromite User Manual.

Refer to the following table for the pin connections to a PICkit 3 programmer:

PICkit 3 Pins	Description	64-pin Micromite eXtreme pin numbers	100-pin Micromite eXtreme pin numbers	144-pin Micromite eXtreme pin numbers
1 - MCLR	Master Reset (active low)	9	15	20
2 - Vcc	Power Supply (3.3V)	8, 26, 39, 54, 60, 19 (AVDD), 34 (VUSB3V3)	14, 37, 46, 62, 74, 83, 93, 30 (AVDD), 52 (VUSB3V3)	18, 33, 55, 64, 88, 107, 122, 137, 41 (AVDD), 74 (VUSB3V3)
3 - GND	Ground	7, 25, 35, 40, 55, 59	13, 36, 45, 53, 63, 75, 84, 92, 31 (AVSS)	17, 32, 54, 63, 75, 89, 108, 123, 136, 42(AVSS)
4 - PGD	Programming Data	16 or 18	25 or 27	36 or 38
5 - PGC	Programming Clock	15 or 17	24 or 26	35 or 37
6 - NC	Not used			

Notes:

- PDD/PGC must be used in matched pairs as aligned vertically in the table
- A pullup resistor of 10K is required between MCLR and Vcc.
- An oscillator is not required to program these chips and will be ignored if present
- The microcontroller being programmed can be powered by the PICkit 3 but it is recommended that a separate power supply be used. When the PICkit 3 supplies the power pin 2 (Vcc) on the PICkit 3 will become an output supplying power to the chip being programmed

For all Backpack PCBs developed by Peter Mather, an onboard USB-to-UART Microbridge (PIC16F1455) is used to program the Micromite eXtreme using the PICPROG32 program.

Details can be found Geoff Grahams website here .

Micromite eXtreme Pinoutouts

64-pin Pinout

Pin	Features				
1	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D5	OV7670-D5
2	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D6	OV7670-D6
3	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D7	OV7670-D7
4	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SPI2-CLK	
5	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	I2C-SDA	
6	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	I2C-CLK	
7	VSS				
8	VDD				
9	MCLR				
10	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-1C	
11	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	OV7670-HR/RR
12	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	KBD-CLK	OV7670-VSYNC
13	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM2-RX	I2S-MCLK
14	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	
15	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	IR
16	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	OV7670-XC/WR
17	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM1-EN	OV7670-PC/RC
18	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-2B	
19	AVDD				
20	AVSS				
21	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-RESET	
22	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM1-RX	
23	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM2-TX	
24	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	KBD-DAT	
25	VSS				
26	VDD				
27	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-RS	
28	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-WR	
29	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM3-TX	I2S-BITCLK
30	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM3-RX	I2S-WORDCLK
31	OSC1				
32	DIGITAL_IN	DIGITAL_OUT	HEARTBEAT		
33	VBUS				
34	VDD				
35	VSS				
36	D-				

37	D+				
38	USBID				
39	VDD				
40	VSS				
41	DIGITAL_IN	DIGITAL_OUT	SPI2-OUT		
42	DIGITAL_IN	DIGITAL_OUT	COM1-TX		
43	DIGITAL_IN	DIGITAL_OUT	MOUSE-CLK		
44	DIGITAL_IN	DIGITAL_OUT	PWM-2A		
45	DIGITAL_IN	DIGITAL_OUT	PWM-2C	SOUND-LEFT	I2S-DATA
46	DIGITAL_IN	DIGITAL_OUT	PWM-1B		
47	DIGITAL_IN	DIGITAL_OUT	SPI2-IN		
48	DIGITAL_IN	DIGITAL_OUT	PWM-1A		
49	DIGITAL_IN	DIGITAL_OUT	SPI-CLK		
50	DIGITAL_IN	DIGITAL_OUT	SPI-IN		
51	DIGITAL_IN	DIGITAL_OUT	SPI-OUT		
52	DIGITAL_IN	DIGITAL_OUT	MOUSE-DAT	COUNT	
53	DIGITAL_IN	DIGITAL_OUT	SOUND-RIGHT		
54	VDD				
55	VSS				
56	DIGITAL_IN	DIGITAL_OUT	COM4-TX	CONSOLE-TX	
57	DIGITAL_IN	DIGITAL_OUT	COM4-RX	CONSOLE-RX	
58	DIGITAL_IN	DIGITAL_OUT	SSD1963-D0	OV7670-D0	
59	VSS				
60	VDD				
61	DIGITAL_IN	DIGITAL_OUT	SSD1963-D1	OV7670-D1	
62	DIGITAL_IN	DIGITAL_OUT	SSD1963-D2	OV7670-D2	
63	DIGITAL_IN	DIGITAL_OUT	SSD1963-D3	OV7670-D3	
64	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D4	OV7670-D4

100-pin Pinout

Pin	Features				
1	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
2	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
3	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-2C	SOUND-LEFT
4	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
5	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
6	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	PWM-2A
7	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	
8	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	
9	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT	IR
10	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SPI2-CLK	
11	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	I2C-SDA	
12	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	I2C-CLK	
13	VSS				
14	VDD				
15	MCLR				
16	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-1C	
17	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
18	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SOUND-RIGHT	
19	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-BLU-SS	
20	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D5	
21	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D4	VGA-VSYNC
22	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D3	
23	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D2	
24	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D1	
25	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D0	
26	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D6	
27	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D7	
28	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
29	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
30	AVDD				
31	AVSS				
32	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D8	VGA-GRN-OUT
33	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D9	VGA-BLU-OUT
34	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D10	VGA-RED-OUT
35	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D11	
36	VSS				

37	VDD				
38	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT		
39	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM1-EN	VGA-BLU-CLK
40	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-2B	
41	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D12	
42	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D13	
43	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D14	VGA-RED-CLK
44	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-D15	VGA-RED-SS
45	VSS				
46	VDD				
47	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM1-RX	
48	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-GRN-CLK	
49	OSC1				
50	OSC2 - unused				
51	VBUS				
52	VDD				
53	VSS				
54	D-				
55	D+				
56	USBID				
57	DIGITAL_IN	DIGITAL_OUT	COM3-TX		
58	DIGITAL_IN	DIGITAL_OUT	COM3-RX	COUNT	
59	DIGITAL_IN	DIGITAL_OUT	Snadpic-SD-CD	I2C2-CLK	
60	DIGITAL_IN	DIGITAL_OUT	I2C2-SDA		
61	DIGITAL_IN	DIGITAL_OUT			
62	VDD				
63	VSS				
64	DIGITAL_IN	DIGITAL_OUT	VGA-GRN-SS		
65	DIGITAL_IN	DIGITAL_OUT	COM1-TX		
66	DIGITAL_IN	DIGITAL_OUT	SPI2-OUT		
67	DIGITAL_IN	DIGITAL_OUT	SPI3-OUT	I2S-DATA	
68	DIGITAL_IN	DIGITAL_OUT	I2S-WORDCLK		
69	DIGITAL_IN	DIGITAL_OUT	SPI3-CLK	I2S-BITCLK	
70	DIGITAL_IN	DIGITAL_OUT	SPI3-IN	I2S-MCLK	
71	DIGITAL_IN	DIGITAL_OUT	PWM-1B		
72	DIGITAL_IN	DIGITAL_OUT	SPI2-IN		
73	DIGITAL_IN	DIGITAL_OUT	PWM-1A		
74	VDD				
75	VSS				
76	DIGITAL_IN	DIGITAL_OUT	SPI-CLK		

77	DIGITAL_IN	DIGITAL_OUT	SPI-IN	
78	DIGITAL_IN	DIGITAL_OUT	SPI-OUT	
79	DIGITAL_IN	DIGITAL_OUT	VGA-HSYNC	
80	DIGITAL_IN	DIGITAL_OUT		
81	DIGITAL_IN	DIGITAL_OUT	Snadpic-SD-CS	
82	DIGITAL_IN	DIGITAL_OUT		
83	VDD			
84	VSS			
85	DIGITAL_IN	DIGITAL_OUT	COM4-TX	CONSOLE-TX
86	DIGITAL_IN	DIGITAL_OUT	COM4-RX	CONSOLE-RX
87	DIGITAL_IN	DIGITAL_OUT	COM2-TX	
88	DIGITAL_IN	DIGITAL_OUT	COM2-RX	
89	DIGITAL_IN	DIGITAL_OUT	KBD-CLK	
90	DIGITAL_IN	DIGITAL_OUT	KBD-DAT	
91	DIGITAL_IN	DIGITAL_OUT	MOUSE-CLK	
92	VSS			
93	VDD			
94	DIGITAL_IN	DIGITAL_OUT	MOUSE-DAT	
95	DIGITAL_IN	DIGITAL_OUT	SSD1963-RESET	
96	DIGITAL_IN	DIGITAL_OUT	SSD1963-RS	
97	DIGITAL_IN	DIGITAL_OUT	SSD1963-WR	
98	DIGITAL_IN	DIGITAL_OUT		
99	DIGITAL_IN	DIGITAL_OUT	HEARTBEAT	
100	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	

144-pin Pinout

Pin	Features			
1	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
2	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
3	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-2C
4	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
5	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
6	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT3
7	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB8
8	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB9
9	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB12	
10	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB10	
11	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT1
12	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT2
13	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COUNT4-IR
14	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SPI2 CLK
15	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	I2C-SDA
16	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	I2C-CLK
17	GND			
18	VCC			
19	DIGITAL_IN	DIGITAL_OUT	HEARTBEAT	
20	RESET			
21	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
22	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
23	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM1-EN
24	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-HSYNC
25	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
26	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-VSYNC
27	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB11
28	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB13	
29	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB14	
30	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB15	
31	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
32	GND			
33	VCC			
34	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SOUND-LEFT
35	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	SOUND-RIGHT
36	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	CAMERA-XCLK

37	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
38	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
39	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
40	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
41	AVDD			
42	AVSS			
43	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	CAMERA-D0
44	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	CAMERA-D1
45	DIGITAL_IN	DIGITAL_OUT	CAMERA-D2	
46	DIGITAL_IN	DIGITAL_OUT	CAMERA-D3	
47	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-GRN-OUT
48	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-BLU-OUT
49	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-RED-OUT
50	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
51	DIGITAL_IN	DIGITAL_OUT		
52	DIGITAL_IN	DIGITAL_OUT		
53	DIGITAL_IN	DIGITAL_OUT		
54	GND			
55	VCC			
56	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
57	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-CLK
58	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	PWM-2B
59	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
60	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	
61	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-CLK
62	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-HSYNC
63	GND			
64	VCC			
65	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	CAMERA-D4
66	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	CAMERA-D5
67	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	CAMERA-D6
68	DIGITAL_IN	DIGITAL_OUT		CAMERA-D7
69	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	COM1-RX
70	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	VGA-CLK
71	OSC1			
72	OSC2 - unused			
73	VBUS			
74	VCC			
75	GND			
76	USBD-			

77	USBD+			
78	USBID			
79	DIGITAL_IN	DIGITAL_OUT	COM3-TX	
80	DIGITAL_IN	DIGITAL_OUT	COM3-RX	
81	DIGITAL_IN	DIGITAL_OUT	CAMERA-HSYNC	CAMERA-RCK
82	DIGITAL_IN	DIGITAL_OUT	CAMERA-VSYNC	COUNT
83	DIGITAL_IN	DIGITAL_OUT	CAMERA-PCLK	CAMERA-RRST
84	DIGITAL_IN	DIGITAL_OUT	CAMERA-WR	
85	DIGITAL_IN	DIGITAL_OUT	I2C2-SCK	
86	DIGITAL_IN	DIGITAL_OUT	I2C2-SDA	
87	DIGITAL_IN	DIGITAL_OUT		
88	VCC			
89	GND			
90	DIGITAL_IN	DIGITAL_OUT	VGA-HSYNC	
91	DIGITAL_IN	DIGITAL_OUT	COM1-TX	
92	DIGITAL_IN	DIGITAL_OUT		
93	DIGITAL_IN	DIGITAL_OUT		
94	DIGITAL_IN	DIGITAL_OUT		
95	DIGITAL_IN	DIGITAL_OUT	SPI2-OUT	
96	DIGITAL_IN	DIGITAL_OUT	SPI3-OUT	I2S-DATA
97	DIGITAL_IN	DIGITAL_OUT	I2S-WORDCLK	
98	DIGITAL_IN	DIGITAL_OUT	SPI3-CLK	I2S-BITCLK
99	DIGITAL_IN	DIGITAL_OUT	SPI3-IN	I2S-MCLK
100	DIGITAL_IN	DIGITAL_OUT		
101	DIGITAL_IN	DIGITAL_OUT		
102	DIGITAL_IN	DIGITAL_OUT		
103	DIGITAL_IN	DIGITAL_OUT		
104	DIGITAL_IN	DIGITAL_OUT	PWM-1B	
105	DIGITAL_IN	DIGITAL_OUT	SPI2-IN	
106	DIGITAL_IN	DIGITAL_OUT	PWM-1A	
107	VCC			
108	GND			
109	DIGITAL_IN	DIGITAL_OUT	SPI-CLK	
110	DIGITAL_IN	DIGITAL_OUT	SPI-IN	
111	DIGITAL_IN	DIGITAL_OUT	SPI-OUT	
112	DIGITAL_IN	DIGITAL_OUT	VGA-HSYNC	
113	DIGITAL_IN	DIGITAL_OUT		
114	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB0	
115	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB1	
116	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB2	

117	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB3	
118	DIGITAL_IN	DIGITAL_OUT		
119	DIGITAL_IN	DIGITAL_OUT	PWM-1C	
120	DIGITAL_IN	DIGITAL_OUT	PWM-2A	
121	DIGITAL_IN	DIGITAL_OUT		
122	VCC			
123	GND			
124	DIGITAL_IN	DIGITAL_OUT	Console-TX	COM4-TX
125	DIGITAL_IN	DIGITAL_OUT	Console-RX	COM4-RX
126	DIGITAL_IN	DIGITAL_OUT		
127	DIGITAL_IN	DIGITAL_OUT	COM2-TX	
128	DIGITAL_IN	DIGITAL_OUT	COM2-RX	
129	DIGITAL_IN	DIGITAL_OUT	KB-CLK	
130	DIGITAL_IN	DIGITAL_OUT	KB-DAT	
131	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB4	
132	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB5	
133	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB6	
134	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB7	
135	DIGITAL_IN	DIGITAL_OUT	MOUSE-CLK	
136	GND			
137	VCC			
138	DIGITAL_IN	DIGITAL_OUT	MOUSE-DAT	
139	DIGITAL_IN	DIGITAL_OUT	SSD1963-RESET	
140	DIGITAL_IN	DIGITAL_OUT	SSD1963-RS	
141	DIGITAL_IN	DIGITAL_OUT	SSD1963-WR	
142	DIGITAL_IN	DIGITAL_OUT		
143	DIGITAL_IN	DIGITAL_OUT		
144	ANALOG_IN	DIGITAL_IN	DIGITAL_OUT	

Unique Micromite eXtreme Features

I2S Audio Output

The Micromite can output high quality audio output on an I2S output. It supports FLAC file playback (which is an open-source lossless format), MOD, WAV and MP3playback.

FLAC, MP3 and WAV playback (both variants) all support playing an entire directory, just use a directory name instead of a filename. PLAY NEXT, PLAY PREVIOUS allow you to step through tracks

Playback over I2S

Supported frequencies are:

- 44100Hz 16-bit(CD quality) and 24-bit
- 48000Hz 16-bit and 24-bit
- 88200Hz 16-bit and 24-bit
- 96000Hz 24-bit
- 192000Hz 24-bit (252MHz only)

The command syntax is simple:

```
PLAY FLAC_I2S flacfilename$ [,interruptwhenfinished]
PLAY MP3_I2S mp3name$ [,interruptwhenfinished]
PLAY WAV_I2S wavname$ [,interruptwhenfinished]
PLAY MOD_I2S modname$ [,interruptwhenfinished]
```

The command is non-blocking and PLAY PAUSE, PLAY RESUME, and PLAY STOP work as expected. Volume control is available using

```
PLAY VOLUME leftlevel, rightlevel.
```

Playback over PWM

```
PLAY FLAC flacfilename$ [,interruptwhenfinished]
PLAY MP3 mp3name$ [,interruptwhenfinished]
PLAY WAV wavname$ [,interruptwhenfinished]
PLAY MODFILE modname$ [,interruptwhenfinished]
```

Connections to the DAC

I2S DACs are cheap (<£6.00) and readily available (ebay is a good source). Tested versions include those with ES9023 chips and those with PCM5102 chips. For DACs with an inbuilt MCLK source, the MCLK connection can be omitted.

Connections to the DAC are as follows:

- MCLK - SPI3-IN (this is re-configured as an output of the clock signal and runs at 128 to 512 times the frame clock depending on frequency allowing the DAC to oversample.
- DATA - SPI3-OUT
- SCK (Bit-Clock) - SPI3-CLK
- LRCK (Frame-Clock) - pin-97 on the 144-pin, pin 68 on the 100-pin

All these pins are available on the SPI3 header on the Backpack144 and the Backpack 100 Nav.

Connections for the 64-pin chip are as follows:

- MCLK - SPI3-CLK Pin-13
- DATA - SPI3-OUT Pin-45
- LRCK - SPI3SS Pin-30
- SCK – SPI3-IN Pin 29

VGA Driver

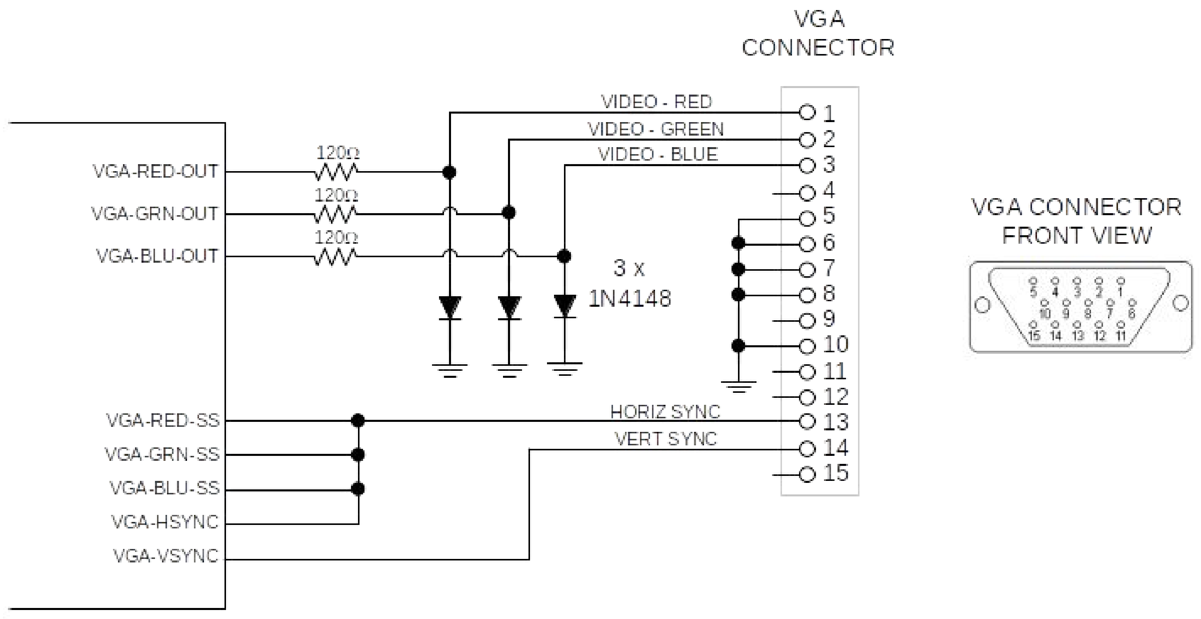
The Micromite eXtreme (144 and 100 pin versions but not the Micromite eXtreme64) can drive a standard VGA monitor by internally generating the necessary VGA signals (red, green, sync, etc). When a VGA monitor is connected and configured the VGA output works exactly the same as a connected LCD display panel – this means that all graphics commands, GUI objects, etc can be used as described later in this manual.

The features of the VGA driver are:

- 640 x 480 pixel output in RGB111 mode
- Optional 640 x 400 widescreen output
- Eight colours (red, blue, green, yellow, cyan, magenta, black, white)
- Works with all graphics and GUI commands.
- Supports any command/features that use transparency (transparent text and the BLIT command).
- Works as the console with 80 characters x 36 lines and will work with the EDIT command

VGA Connections

The connections for the VGA monitor are shown below:



The I/O pins VGA-RED-CLK, VGA-GRN-CLK, VGA-BLU-CLK must be left unconnected. For the actual pin numbers refer to the pinout tables earlier in this manual.

Note that the diodes must be high speed signal types like the 1N4148 (not general purpose power diodes).

Configuring VGA Output

The command to enable the standard VGA 640 x 480 pixel output is:

```
OPTION LCDPANEL VGA
```

and to enable the widescreen format (640 x 400 pixels) the command is:

```
OPTION LCDPANEL VGA, 16
```

These commands only need to be run once as the parameters are stored in non volatile memory. Every time the Micromite is restarted MMBasic will automatically initialise the display ready for use. If the VGA output is no longer required the command OPTION LCDPANEL DISABLE can be used which will disable the VGA feature and return the I/O pins for general use.

The default output for the console is the USB interface. If the VGA monitor or the LCDPANEL are to be used as the console in addition to the USB interface, the command to enable this is:

```
OPTION LCDPANEL CONSOLE
```

To disable console output to the LCDPANEL or the VGA monitor:

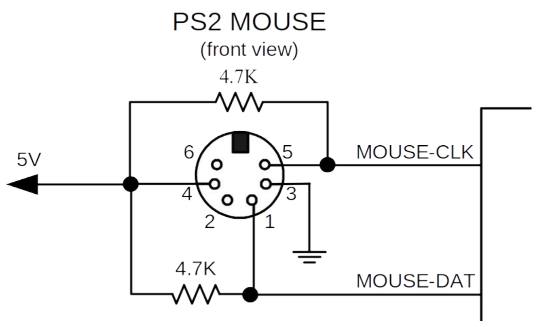
```
OPTION LCDPANEL NOCONSOLE
```

Mouse Support

The Micromite eXtreme supports a PS2 mouse which will act like a touch input on an LCD screen (it also works with the VGA output). MMBasic will automatically display a mouse pointer on the display which is moved by the mouse. When the left button the mouse is clicked it will act like a touch at the location pointed to by the cursor. This feature will also work with GUI controls.

Connecting the Mouse

The PS2 mouse uses a 6-pin DIN connector which should be connected to the Micromite eXtreme as illustrated.



To enable the mouse the command is:

```
    OPTION MOUSE ENABLE
```

and to disable it:

```
    OPTION MOUSE DISABLE
```

These commands only need to be run once as the parameters are stored in non volatile memory. Every time the Micromite is restarted MMBasic will automatically initialise the mouse input ready for use.

Cursor Commands

The cursor (ie, mouse pointer) can be turned off/on, its colour can be set and its status can be overridden using the CURSOR command. It will only work with displays that support transparency. Displays capable of transparent text are a VGA monitor or any LCD panels that use the ILI9341 controller or an SSD1963 controller. The latter must have the RD pin specified in the OPTION LCDPANEL command. The cursor command can be used without a mouse to allow, for example, a joystick to move the cursor.

The cursor commands are as follows:

CURSOR ON
Enables the display of the cursor (this is the default when MOUSE is first enabled). If it is used after a previous CURSOR OFF command it will also restore the previous position of the cursor.

CURSOR OFF
Hides the cursor.

CURSOR X, Y [, LEFT] [, RIGHT] [, MID]
Positions the cursor to the screen location X and Y(in pixels) and sets the left-click (0 or 1) and optionally the right-click and mid-click status.

CURSOR COLOUR colour
Sets the cursor colour (this is a standard 24-bit colour value).
(CURSOR COLOR (US spelling) is also valid).

Cursor Functions

CURSOR(x)	Returns the current x coordinate (in pixels) of the cursor
CURSOR(y)	Returns the current y coordinate (in pixels) of the cursor
CURSOR(left)	Returns the current state of the left button
CURSOR(right)	Returns the current state of the right button
CURSOR(middle)	Returns the current state of the middle button

Clock Speed Control

MMBasic can work with chips rated for 200 MHz or 252 MHz operation. By default the firmware will start running at 200 MHz however the 252 MHz clock speed can be selected with the command:

```
        OPTION CPU 252
```

or the speed can be returned to 200 MHz with the command:

```
        OPTION CPU 200
```

These commands change how MMBasic starts up and will cause a restart of the processor. The clock speed is saved in flash memory so the command only needs to be used once and will be automatically applied on startup.

***** OPTION CPU 252 must only be used on chips specifically rated for 250Mhz operation. *****

Use of this command on a 200MHz rated chip will cause MMBasic to stop running. Apart from the increased processing speed the only difference with a chip running at 252 MHz is the ability to use WAV files recorded at 24 KHz and 48 KHz.

16-bit Interface to SSD1963 Based LCD Displays

The Micromite eXtreme (but not the Micromite eXtreme64) can drive an SSD1963 display using a 16-bit parallel bus for extra speed. The extra I/O pins for this are listed as SSD1963-DB8 to SSD1963-DB15 on the pinout tables in this manual and they must be connected to the pins labelled DB8 to DB15 on the I/O connector on the SSD1963 display.

Note that in this mode the SSD1963 controller runs with a reduce colour range (65 thousand colours) compared to 16 million colours with the normal 8-bit interface.

To select the 16-bit bus the following controller names must be used with the OPTION LCDPANEL command when configuring the display:

•	SSD1963_4_16	For a 4.3 inch display.
•	SSD1963_5_16	For a 5 inch display.
•	SSD1963_5A_16	For an alternative version of the 5 inch display if SSD1963_5 does not work.
•	SSD1963_7_16	For a 7 inch display.
•	SSD1963_7A_16	For a different version of the 7 inch display if SSD1963_7 does not work.
•	SSD1963_8_16	For an 8 inch display.

Two I^2C Channels

The Micromite eXtreme (but not the Micromite eXtreme64) supports two I^2C channels. The second channel operates the same as the first, the only difference is that the commands use the notation I2C2 (for example I2C2 OPEN, etc).

Six PWM Channels

The second PWM controller (ie, PWM 2) supports three channels (the other versions of the Micromite only support two). The command to use all three channels is:

```
        PWM 2, freq, 2A, 2B, 2C
```

Similarly the SERVO command can also control six channels with the extra channel available on controller 2:

```
        SERVO 2 [, freq], 2A, 2B, 2C
```

Three SPI Channels

The Micromite eXtreme (but not the Micromite eXtreme64) supports three SPI channels. The second and third channels operate the same as the first, the only difference is that the commands use the notation SPI2 and SPI3 (for example SPI3 WRITE, etc).

Note that by default, if the Micromite eXtreme is configured for a SPI based LCD panel, touch or an SD card then SPI2 will be unavailable to BASIC programs as these functions will use that channel.

Alternate SPI Channel for the SD Card

The SPI channel used for the SD Card (if configured) defaults to the second channel (SPI2) however this can be changed by appending the SPI channel number to the end of the OPTION SDCARD command this:

```
      OPTION SDCARD CSPIN [,CDPIN] [,WPPIN] [,SPIno]
```

'SPIno' is the SPI controller to use and can be 1, 2 or 3.

This is particularly useful with the SnadPIC MZ, PIC32MZ EF MCU Starter Kit is as it has the SD Card hardwired to controller 3. In that case the MMBasic command to configure the SD Card would be:

```
      OPTION SDCARD 81, 59, , 3
```

Heartbeat

The heartbeat is an I/O pin which is pulsed off and on at a 1Hz rate. It is normally used to drive a LED to show that MMBasic is alive and running on the Micromite eXtreme.

The default is for it to be enabled however it can be disabled with: OPTION HEARTBEAT DISABLE

If necessary it can be re enabled with:

```
      OPTION HEARTBEAT ENABLE
```

These commands only needs to be run once as the parameters are stored in non volatile memory. Every time the Micromite is restarted MMBasic will automatically initialise the heartbeat feature.

Random Number Generation

The Micromite eXtreme uses the hardware random number generator in the MZ series of chips to deliver true random numbers. This means that the RANDOMIZE command is no longer needed and is not supported.

MM.DEVICE$

On the Micromite eXtreme the read only variable MM.DEVICE$ will return "Micromite eXtreme, Microchip ID 0xhhhhhhhh".

OPTION VCC command

The Micromite eXtreme supports the OPTION VCC command. This allows the user to precisely set the supply voltage to the chip and is used in the calculation of voltages when using analog inputs e.g.

```
      OPTION VCC 3.15.
```

The parameter is not saved and should be initialised either on the command line or in a program.

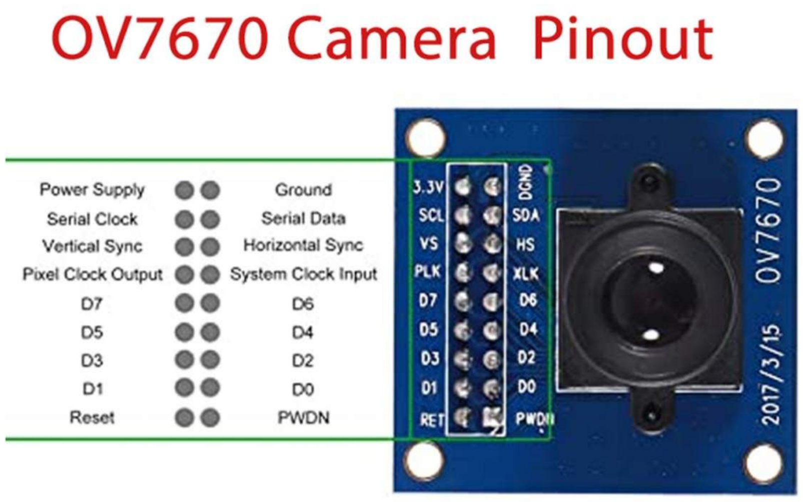
CPU command

The Micromite eXtreme does not support dynamically changing the CPU speed or the sleep function. Accordingly the commands CPU speed and CPU SLEEP are not available. However the eXtreme does support “CPU SLEEP time” where time is specified in seconds.

The CPU speed of the Micromite eXtreme can be permanently set to 200 MHz or 252 MHz using the OPTION CPU command.

OV7670 Camera Support

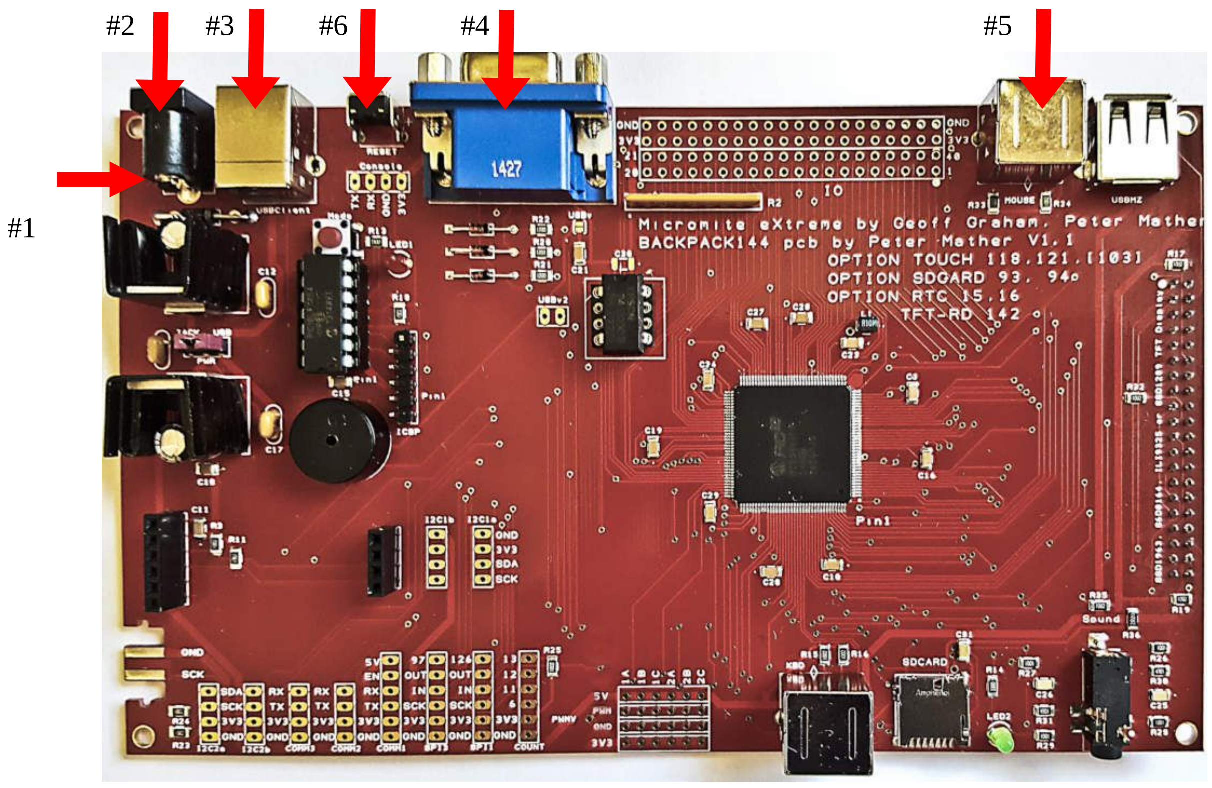
The Micromite eXtreme 144 pin version supports connection of an OV7670 camera



FUNCTION	PIN NUMBER	OV7670 PIN	Backpack 144 PIN	FUNCTION	PIN NUMBER	OV7670 PIN	Backpack 144 PIN
+3v3 Power	VCC	1		Pixel Data Out D6	D6	10	67
Ground	GND	2		Pixel Data Out D5	D5	11	66
I2C Serial Clock	SCL	3		Pixel Data Out D4	D4	12	65
I2C Data	SDA	4		Pixel Data Out D3	D3	13	46
Vertical Sync-active high active frame	VSYNC	5	82	Pixel Data Out D2	D2	14	45
Horizontal reference- active high	HREF	6	81	Pixel Data Out D1	D1	15	44
Pixel clock from sensor	PCLK	7	83	Pixel Data Out D0(LSB)	D0(LSB)	16	43
Master clock to sensor	XCLK	8	36	Reset	RESET	17	83
Pixel Data Out (MSB)	D7	9	68	Power Down	PWDN	18	

Backpack144 Quick Start Tutorial

The following assumes that you have built or purchased the Micromite eXtreme 144 pin Backpack PCB (see the picture below), set jumper PWR (#1) pins 2-3 (power from external source to power jack (#2) – it can be 8VDC to 12VDC at 1A – ideally 9VDC), attached a USB-B connection to a PC (#3) running a serial interface program such as TerraTerm (8-N-1 at 115200 baud).
You could alternatively connect up a mouse and keyboard for use with an external VGA monitor - #4 and #5 – see Annex F. for details.4.



You can now load the MMBasic firmware – see Annex E. for details. Upon applying power, press the RESET button (#6). At that point you should see the following displayed on the console:-

```
> Micromite eXtreme MMBasic Ver 5.07.01b1 @200MHz on 144 pin chip
Copyright 2011-2023 Geoff Graham
Copyright 2016-2023 Peter Mather

>
```

Now would be a good time to configure MMBasic that is running in the Micromite eXtreme. Assuming the above platform with an SSD1963 7" LCD and a DS3231 RTC module installed, enter the following OPTIONs:-

```
OPTION COLOURCODE ON
OPTION LCDPANEL SSD1963_7A_16,RLANDSCAPE,,142
```

Tip! Note: some 7" LCDPANELS may not need the A in 7A above – try either

…. test with

```
GUI TEST LCDPANEL
```

…. you should see a LCD PANEL full of coloured circles, press any key to terminate test.

```
OPTION TOUCH 118,121,103
GUI CALIBRATE
```

…. you should see a target on the LCD PANEL, press each target to set which should result in a calibration successful message. Now ...

```
OPTION SDCARD 93
OPTION RTC 15,16
OPTION COLOURCODE ON
```

Enter the command OPTION LIST at the command prompt should should show the following:-

```
> option list
Micromite eXtreme MMBasic 5.07.01b1
OPTION CPU 200
OPTION COLOURCODE ON
OPTION LCDPANEL SSD1963_7a_16,RLANDSCAPE,,142
OPTION TOUCH 118,121,103
OPTION SDCARD 93
OPTION RTC 15,16
OPTION HEARTBEAT ON

>
```

Tip! Note: The OPTIONs above are for the 144pin Backpack – see pinout lists for alternate configurations.

MMBasic in the Micromite eXtreme

Command Prompt

Experienced users of MMBasic can skip this section.

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 that is currently in memory.

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 Micromite eXtreme's output is shown in normal text.
Try a simple calculation:

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

See how much memory you have:

```
> MEMORY
Flash:
   1K ( 0%) Program (0 lines)
 566K (100%) Free

RAM:
   0K ( 0%) 0 Variables
   0K ( 0%) General
 362K (100%) Free
```

What is the current time?

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

What is the current date?

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

Count to 20:

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

Your First Program

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

- At the command prompt type `EDIT "hello.bas"` followed by the ENTER key.
- The editor should start up and you can enter this line: `PRINT "Hello World"`
- Press the F1 key in your keyboard. This tells the editor to save your program and exit to the command prompt.
- At the command prompt type `RUN "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 Micromite eXtreme. 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 (as we saw in the GUI TEST LCDPANEL above).

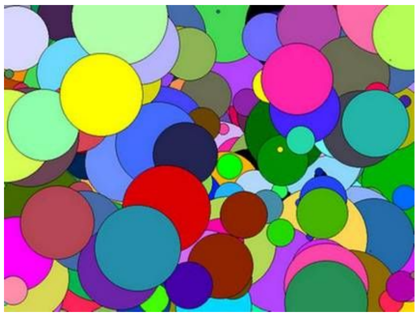
For this you need to know a little more about the editor. If you have used any full screen text editor in the past you will find the operation of this editor familiar. The arrow keys will move your cursor around in the text while the home and end keys will take you to the beginning or end of the line.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



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

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

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

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

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

For newcomers to MMBasic, a more in-depth tutorial and a description of programming in BASIC see the document *"Getting Started with the Micromite" and "GUI Controls and Programming*" which can be found at https://geoffg.net/micromite.html (scroll to the bottom of the page).

Using MMBasic

Commands and Program Input

At the command prompt you can enter a command and it will be immediately run. Most of the time you will do this to tell the Micromite eXtreme to do something like 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 program editor which is built into the Micromite eXtreme and is described later in this manual. It includes advanced features such as search and 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 to the Micromite eXtreme via the XModem protocol (see the XMODEM command) or by streaming it up the console serial link (see the AUTOSAVE command).

A third 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 to the Micromite eXtreme with a single click of the mouse. MMEdit was written by Jim Hiley and can be downloaded for free from https://www.c-com.com.au/MMedit.htm.

One thing that you cannot do is use the old BASIC way 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.

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.

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 interchangeably). 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. For example (the label name is JmpBack):

```
JmpBack: A = A + 1
PRINT A
GOTO JmpBack
```

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 (:).

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 and the Insert key to switch between insert and overwrite. 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 command lines which can be edited and reused.

Shortcut Keys

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

F2	RUN
F3	LIST
F4	EDIT
F10	AUTOSAVE
F11	XMODEM RECEIVE
F12	XMODEM SEND

Function keys F1, and F5 to F9 can be programmed with custom text. See the OPTION FNKey command.

Interrupting A Running Program

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.

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 change the CPU clock speed with the command:

```
   OPTION CPUSPEED speed
```

Saved Variables

Because the Micromite eXtreme does not necessarily have a normal storage system it needs to save data that can be recovered when power is restored. This can be done with the VAR SAVE command which will save the variables listed on its command line in non-volatile flash memory. The space reserved for saved variables is 16KB.

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 this command is placed near the start of a program so that the variables are ready for use by the program.

This facility is intended for saving calibration data, user selected options and other items which change infrequently. It should not be used for high speed saves as you may wear out the flash memory. The flash used for the Raspberry Pi Pico has a high endurance but this can be exceeded by a program that repeatedly saves variables. If you do want to save data often you should add a real time clock chip. The RTC commands can then be used to store and retrieve data from the RTC's battery backed memory. See the RTC command for more details.

PIN Security

Sometimes it is important to keep the data and program in an embedded controller confidential. In the Micromite eXtreme this can be done by using the OPTION PIN command. This command will set a pin number (which is stored in flash) and whenever the Micromite eXtreme 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 Micromite eXtreme. 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 a program, they cannot change the program or change any aspect of MMBasic or the Micromite eXtreme. 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 reload the Micromite eXtreme firmware (which will erase the program and all options).

There are other time consuming ways of accessing the data (such as using a programmer to examine the flash memory) so this should not be regarded as the ultimate security but it does act as a significant deterrent.

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 but they will default to reading DATA statements in the main program memory. If you want to read from DATA statements in the library you must use the
- RESTORE command before the first READ command. This will reset the pointer to the library space.
- The library is saved to program flash memory Slot 4 and this will not be available for storing a program if LIBRARY SAVE is used.
- You can see what is in the library by using the LIBRARY LIST command which will list the contents of the library space.

To delete the routines in the library space you use the LIBRARY DELETE command. This will clear the space and return the Flash Slot 4 used by the library back to being available for storage for normal programs. The only other way to delete a library is to use OPTION RESET.

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 on the Micromite.

```
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. When the Micromite eXtreme is first powered up or reset it will search for this subroutine and, if found, it will be run once.

For example, if the Micromite eXtreme has a real time clock attached, the program could contain the following code:

```
SUB MM.STARTUP
  RTC GETTIME
END SUB
```

This would cause the internal clock within MMBasic to be set to the current time on every power up or reset. After the code in MM.STARTUP has been run MMBasic will continue with running the rest of the program in program memory. If there is no other code MMBasic 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 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 TIME$ "> ";
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 that that should be used instead.

Line Numbers and Program Structure

The structure of a program line is:

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

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

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

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

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

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

The main commands used to manage a program are:

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.

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.

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

There are three other commands that operate on program files. These are AUTOSAVE, LIST ALL and XMODEM. These are used for sending/receiving programs via the serial console to or from the SDcard. 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 Micromite eXtreme does not have the commands LOAD or SAVE as they are not required.

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.

Running Programs

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

The running program is 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 command to turn AUTORUN 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 make it easy to build loops without using the GOTO statement. User defined subroutines and functions make it easy to add your own commands to MMBasic.

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

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 (unlike with the Micromite).

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. If the Micromite eXtreme has the RTC module DS3231 installed, it will not lose the time even when powered off.

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

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

Watchdog Timer

It is possible to use the Micromite eXtreme without a VGA monitor or LCD PANEL 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).

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.

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.

Full Screen Editor

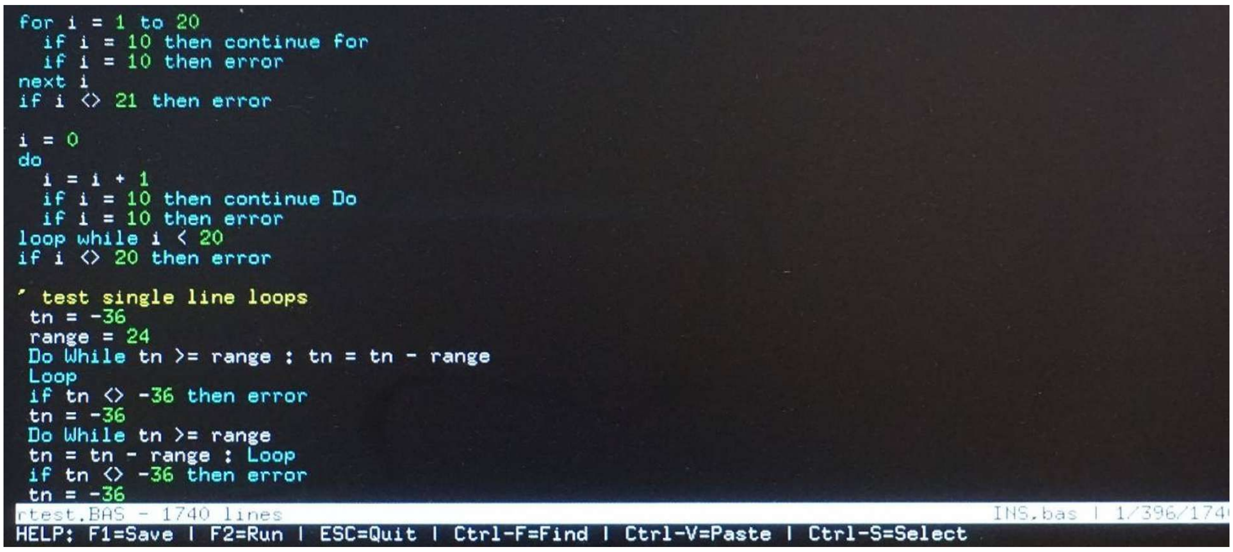
An important productivity feature is the built-in full screen editor. This will work with both the VGA or LCD PANEL video output and the serial console. To run the editor you use the command EDIT at the command prompt. For example:

```
        EDIT "filename"
```

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

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

On the VGA screen the editor looks like this:



```
for i = 1 to 20
  if i = 10 then continue for
  if i = 10 then error
next i
if i <> 21 then error

i = 0
do
  i = i + 1
  if i = 10 then continue Do
  if i = 10 then error
loop while i < 20
if i <> 20 then error

' test single line loops
 tn = -36
 range = 24
 Do While tn >= range : tn = tn - range
 Loop
 if tn <> -36 then error
 tn = -36
 Do While tn >= range
 tn = tn - range : Loop
 if tn <> -36 then error
 tn = -36
rtest.BAS - 1740 lines                                                    INS.bas | 1/396/174
HELP: F1=Save | F2=Run | ESC=Quit | Ctrl-F=Find | Ctrl-V=Paste | Ctrl-S=Select
```

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

If you have used an editor like Windows Notepad previously you will find that the operation of this editor is familiar. The arrow keys will move the cursor around in the text, home and end will take you to the beginning or end of the line. Page up and page down will do what their titles suggest. The delete key will delete the character at the cursor and backspace will delete the character before the cursor. The insert key will toggle between insert and overtype modes. About the only unusual key combination is that two home key presses will take you to the start of the program and two end key presses will take you to the end. If a mouse is fitted it can also be used to position the cursor, select text, etc.At the bottom of the screen the status line will list the various function keys used by the editor and their action.

In more details these are:

ESC	This will cause the editor to abandon all changes and return to the command prompt with the program memory unchanged. If you have changed the text you will be asked if you really what want to abandon your changes.
F1: SAVE	This will save the program to program memory and return to the command prompt.
F2: RUN	This will save the program to program memory and immediately run it.
F3: FIND	This will prompt for the text that you want to search for. When you press enter the cursor will be placed at the start of the first entry found.
SHIFT-F3	Once you have used the search function you can repeat the search by pressing SHIFT-F3.
F4: MARK	This is described in detail below.
F5: PASTE	This will insert (at the current cursor position) the text that had been previously cut or copied (see below).

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

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

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

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

If you are using Tera Term, Putty, MMEdit or GFXterm as the terminal emulator it is also possible to position the cursor by left clicking the PC's mouse in the terminal emulator's window.

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

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

Mouse Support

To use the mouse in the editor it needs to be configured using the OPTION MOUSE command (see the previous *Hardware Features* section. When configured the mouse works as you would expect, you can:

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

Colour Coded Editor Display

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

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

and re enabled with:

```
OPTION COLOURCODE ON
```

This setting is saved in non-volatile memory and automatically applied on startup. It applies to both the VGA output and the serial console.

Variables and Expressions

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

Variables

Variables can start with an alphabetic character or underscore and can contain any alphabetic or numeric character, the period (.) and the underscore (_). They may be up to 31 characters long.

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

MMBasic supports four different types of variables:

1. Double Precision Floating Point.
   These can store a number with a decimal point and fraction (eg, 45.386) however they will lose accuracy when more than 14 digits of precision are used. Floating point variables are specified by adding the suffix '!' to a variable's name (eg, i!, nbr!, etc). They are also the default when a variable is created without a suffix (eg, i, nbr, etc).
2. 64-bit Signed Integer.
   These can store positive or negative numbers with up to 19 decimal digits without losing accuracy but they cannot store fractions (ie, the part following the decimal point). These are specified by adding the suffix '%' to a variable's name. For example, i%, nbr%, etc.
3. A String.
   A string will store a sequence of characters (eg, "Tom"). Each character in the string is stored as an eight bit number and can therefore have a decimal value of 0 to 255. String variable names are terminated with a '$' symbol (eg, name$, s$, etc). Strings can be up to 255 characters long.
4. Long Strings.
   The LONGSTRING commands allow for the manipulation of strings longer than the normal MMBasic limit of 255 characters. Variables for holding long strings must be defined as single dimensioned integer arrays with the number of elements set to the number of characters required for the maximum string length divided by eight. The reason for dividing by eight is that each integer in an MMBasic array occupies eight bytes.
   Note that the long string routines do not check for overflow in the length of the strings. If an attempt is made to create a string longer than a long string variable's size the outcome will be undefined.

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

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

ESCAPE Sequences

MMbasic in the MicroMite eXtreme supports Escape sequences inside strings for normal string input and quoted DATA statements (but not unquoted DATA strings). These need to be enabled with OPTION ESCAPE.

ESCAPE sequence	Hex value in ASCII	Character represented
\a	07	Alert (Beep, Bell)
\b	08	Backspace
\e	1B	ESCAPE character
\f	0C	Formfeed Page Break
\n	0A	Newline (Line Feed); see notes below
\r	0D	Carriage Return
\q	22	Quote symbol
\t	09	Horizontal Tab
\v	0B	Vertical Tab
\\	5C	Backslash
\nnn	any	The byte whose numerical value is given by nnn interpreted as a decimal number
\xhh	any	The byte whose numerical value is given by hh…interpreted as a hexadecimal number

For example:

```
    PRINT “This is an example of a \qquoted\q string can be embedded”
    This is an example of how a “quoted” string can be embedded
```

OPTION DEFAULT

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

```
Nbr = 1234
```

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

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

The default can be set to FLOAT (which is the default when a program is run), INTEGER, STRING or NONE. In the latter all variables must be specifically typed otherwise an error will occur. The OPTION DEFAULT command can be placed anywhere in the program and changed at any time but good practice dictates that if it is used it should be placed at the start of the program and left unchanged.

OPTION EXPLICIT

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

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

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

DIM and LOCAL

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

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

If LOCAL is used to specify a variable with the same name as a global variable then the global variable will be hidden to the subroutine or function and any references to the variable will only refer to the variable defined by the LOCAL command. Any variable created by LOCAL will vanish when the program leaves the subroutine.
At its simplest level DIM and LOCAL can be used to define one or more variables based on their type suffix or the OPTION DEFAULT in force at the time. For example:

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

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

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

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

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

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

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

In this case the type of each variable is set individually (not as a group as when the type is placed before the list of variables). The variables can also be initialised while being defined. For example:

```
DIM INTEGER a = 5, b = 4, c = 3
DIM s$ = "World", i% = &H8FF8F
DIM msg AS STRING = "Hello" + " " + s$The value used to initialise the
variable can be an expression including user defined functions.
```

The DIM or LOCAL commands are also used to define an array and all the rules listed above apply when defining an array. For example, you can use:

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

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

```
DIM INTEGER nbr(5) = (11, 12, 13, 14, 15, 16)
```

or

```
DIM days(7) AS STRING = ("","Sun","Mon","Tue","Wed","Thu","Fri","Sat")
```

STATIC

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

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

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

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

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

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

Constants

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 = 26
CONST MaxValue = 2.4
```

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

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

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

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

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

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

Expressions and Operators

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

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

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

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

Arithmetic operators:

^	Exponentiation (eg, `b^n` means b^n)
* / \ MOD	Multiplication, division, integer division and modulus (remainder)
+ -	Addition and subtraction

Shift operators:

x << y x >> y	These operate in a special way. << means that the value returned will be the value of x shifted by y bits to the left while >> means the same only right shifted. They are integer functions and any bits shifted off are discarded. For a right shift any bits introduced are set to the value of the top bit (bit 63). For a left shift any bits introduced are set to zero.

Logical operators:

NOT INV	invert the logical value on the right (eg, `NOT a=b` is `a<>b`) or bitwise inversion of the value on the right (eg, `a = INV b`)
<> < > <= =< >= =>	Inequality, less than, greater than, less than or equal to, less than or equal to (alternative version), greater than or equal to, greater than or equal to (alternative version)
=	equality
AND OR XOR	Conjunction, disjunction, exclusive or

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

The other logical operations result in the integer 0 (zero) for false and 1 for true. For example the statement PRINT 4 >= 5 will print the number zero on the output and the expression A = 3 > 2 will store +1 in A.

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

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

String operators:

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

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

MMBasic Regex Syntax used in INSTR and LINSTR functions.

The alternate forms of the INSTR() and LINSTR() functions can take a regular expression as the search pattern. The alternate form of the commands 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 variable that is by the firmware to return the size of a matching string. If the variable doesn't exist it is created.. The size parameter is a floatingpoint variable that is used by the firmware to return the size of a matching string. If the variable doesn't exist it is created.

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 document is a summary of the syntax and supported operations used in the MMBasic implementation.

Anchors

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

Qualifiers

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

Groups and Ranges

(a|b)	a or b
\(…\)	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

Escapes Required to Match Normal Characters

\^	to match ^ (caret)
\.	to match . (dot)
*	to match * (asterix)
\$	to match $ (dollar)
\[	to match [(left bracket)
\\	to match \ (backslash)

Escapes with Special Functions

\+	See Qualifiers
\?	See Qualifiers
\{	See Qualifiers
\}	See Qualifiers
|	See Groups and Ranges
\(	See Groups and Ranges
\)	See Groups and Ranges
\w	See Character Classes

Character Classes

\w	digits,letters and _
[:word:]	digits,letters and _
[:upper:]	Upper case letters_
[:upper:]	Upper case letters_
[:lower:]	Lower case letters_
[:alpha:]	All letters
[:alnum:]	Digits and letters
[:digit:]	Digits
[:xdigit:]	Hexidecimal digits
[:punct:]	Puntuation
[:blank:]	Space and tab
[:space:]	Blank charaters
[:cntrl:]	Control charaters
[:graph:]	Printed characters
[:print:]	Printed chars and spaces

Example expression to match an IP Address which is contained within a word boundary.

```
"\<[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\.[0-9]\{1,3\}\>"
```

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.

64-bit Unsigned Integers

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

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

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

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

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

```
X% = &HFFFF0000FFFF0044
Y% = &H800FFFFFFFFFFFFF
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
```

Local Variables

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

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

Functions

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

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

For example:

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

Passing Arguments by Reference

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

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

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

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

```
Swap nbr1, nbr2
```

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

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

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

Passing Arrays

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

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

This type of construct is often used in sorting arrays.

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

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

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

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

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

Early Exit

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

Recursion

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

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

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"

Creating CSUBs

It is possible to write C code and have it compiled as inline MMBasic commands or functions. This can then be loaded into MMBasic as a CSUB which can be called just like it was another MMBasic command or function. Writing CSUBs is beyond the scope of this document other than to document the CSUB command which loads the actual CSUB once it is developed.

This thread on TBS forum is a starting point for further information.

CSUBs in the ArmMite F4

Basic Graphics

There are ten basic drawing commands that you can use within MMBasic to draw images on the VGA monitor or LCD PANEL (none of these apply to the serial console). There are also a number of advanced commands designed for programmers writing games (such as MODE, FRAMEBUFFER, BLIT and SPRITE) 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 640x480 pixels in RGB111 format giving 8 different colours. At this resolution the bottom right pixel will be at X = 639 and Y = 479.

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, white, orange, pink, gold, salmon, beige, lightgrey and grey (or USA spelling gray/lightgray).

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 Micromite eXtreme 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.

So, for example, at startup the Micromite eXtreme'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 8 colours supported by the RGB(111) colour mode.

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

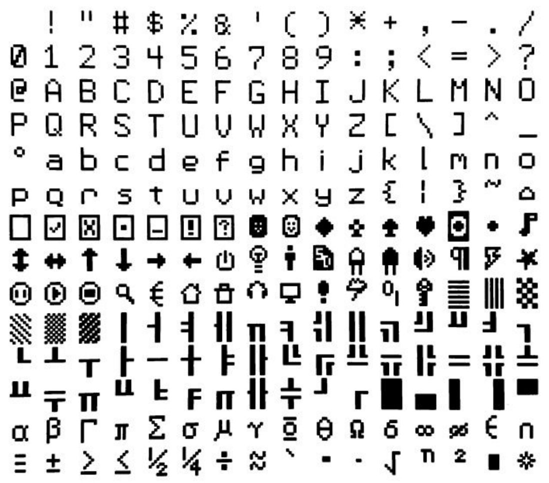
The COLOUR command takes the format:

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

Fonts

There are seven built in fonts. These are:

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



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

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

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

The format of an embedded font is:

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

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

Additional fonts and information can be found in the Embedded Fonts folder in the Micromite eXtreme 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.

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

Drawing Commands

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

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

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

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

The drawing commands are:

- `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.
- `BOX X, Y1, W, H,` *`LW, C, FILL`*
  Draws a box starting at X and Y1 which is W pixels wide and H pixels high. LW is the width of the sides of the box and can be zero. It defaults to 1.
- `RBOX X, Y1, W, H,` *`R, C, FILL`*
  Draws a box with rounded corners starting at X and Y1 which is W pixels wide and H pixels high. R is the radius of the corners of the box. It defaults to 10.
- `TRIANGLE X1, Y1, X2, Y2, X3, Y3,` *`C, FILL`*
  Draws a triangle with the corners at X1, Y1 and X2, Y2 and X3, Y3. C is the colour of the triangle and FILL is the fill colour. FILL can omitted or be -1 for no fill.
- `CIRCLE X, Y, R,` *`LW, A, C, FILL`*
  Draws a circle with X and Y as the centre and a radius R. LW is the width of the line used for the circumference and can be zero (defaults to 1). A is the aspect ratio which is a floating point number and defaults to 1. For example, an aspect of 0.5 will draw an oval where the width is half the height.
- `ARC x, y, r1, r2, a1, a2, c`
  Draws an arc with the centre at x and y, r1 and r2 are the inner and outer radius defining the thickness of the arc (if they are the same the arc will be one pixel thick), a1 and a2 are the start and end angles in degrees and c is the colour.
- `POLYGON n, xarray%(), yarray%(),` *`C , FILL`*
  Draws a outline or filled polygon defined by the x, y coordinate pairs in xarray%() and yarray%(). 'n' is the number of points to use in drawing the polygon. If the last xy-coordinate pair is not the same as the first the firmware will automatically create an additional xy-coordinate pair to complete the polygon.
- `TEXT X, Y, STRING,` *`ALIGNMENT, FONT, SCALE, C, BC`*
  Displays a string starting at X and Y. ALIGNMENT is 0, 1 or 2 characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER or RIGHT aligned text and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE or BOTTOM aligned text. The default alignment is left/top. FONT and SCALE are optional and default to that set by the FONT command. C is the drawing colour and BC is the background colour. They are optional and default to that set by the COLOUR command.

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

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

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

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

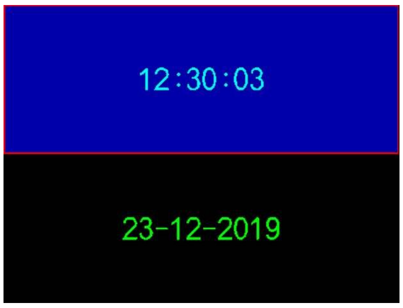
Example of Basic Graphics

As an example, the following program will draw a simple digital clock on the 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:

1. 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.
2. Draws the date centred in the lower half of the screen. In this case the TEXT command uses the default font and colours previously set. The screenshot on the right shows the result.



Rotated Text

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

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

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

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

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

Transparent Text

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

BLIT Command

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

The available commands are:

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

#b is the buffer number in the range of 1 to 32. x and y are the coordinates of the top left corner and w and h are the width and height of the image. READ will copy the display image to the buffer, WRITE will copy the buffer to the display and CLOSE will free up the buffer and reclaim the memory used. LOAD will load an image file into the buffer.

BLIT LOAD and BLIT WRITE will work on any display while BLIT and BLIT READ will only work on displays capable of transparent text (i.e. using the SSD1963, ILI9341, ST7789_320, or ILI9488 with MISO connected).

These commands can be used to copy a portion of the display to another location (by copying to a buffer then writing somewhere else) but a simpler method is to use an alternative version of the BLIT command as follows:
BLIT x1, y1, x2, y2, w, h

This will copy a portion of the image at x1/y1 to the location x2/y2. w and h specify the width and height of the image to be copied. The source and destination areas can overlap and the BLIT command will perform the copy correctly.

This form of the BLIT command is particularly useful for creating graphs that can scroll horizontally or vertically as new data is added.

Displaying Images

Using the LOAD command you can load an image from the SD card 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. 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 and will be ignored).

Load Image

The LOAD IMAGE and LOAD JPG commands can be used to load an image from the Flash Filesystem or SD Card and display it on the LCD display. This can be used to draw a logo or add an ornate background to the graphics drawn on the display.

Example

As an example the following program will draw a simple digital clock on an ILI9341 based LCD display. The program will terminate and return to the command prompt if the display screen is touched. First the display and touch options must be configured by entering the commands listed at the beginning of this section.

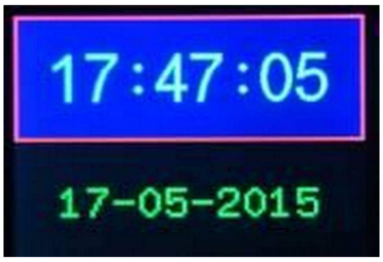
The exact format of these will depend on how you have connected the display panel. Then enter and run the program:

```
CONST DBlue = RGB(0, 0, 128)
COLOUR RGB(GREEN), RGB(BLACK)
FONT 1, 3
' A dark blue colour
' Set the default colours
' 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$, "CM", 1, 4, RGB(CYAN), DBlue
TEXT MM.HRes/2, MM.VRes*3/4, DATE$, "CM"
IF TOUCH(X) <> -1 THEN END
LOOP
```

This 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 three functions:

1. 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.
2. 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.
3. Checks for a touch on the screen. This is indicated when the TOUCH(X) function returns something other than -1. In that case the program will terminate.



The screen display should look like this (the font used in this illustration is different):

Advanced Graphics

The Micromite eXtreme incorporates a suite of advanced graphic controls that respond to touch on LCD screens and mouse clicks on a VGA monitor (144 pin version). These include on screen switches, buttons, indicator lights, keyboard, etc. MMBasic will draw the control and animate it (i.e. a switch will appear to depress when touched). All that the BASIC program needs to do is invoke a single command to specify the basic details of the control.

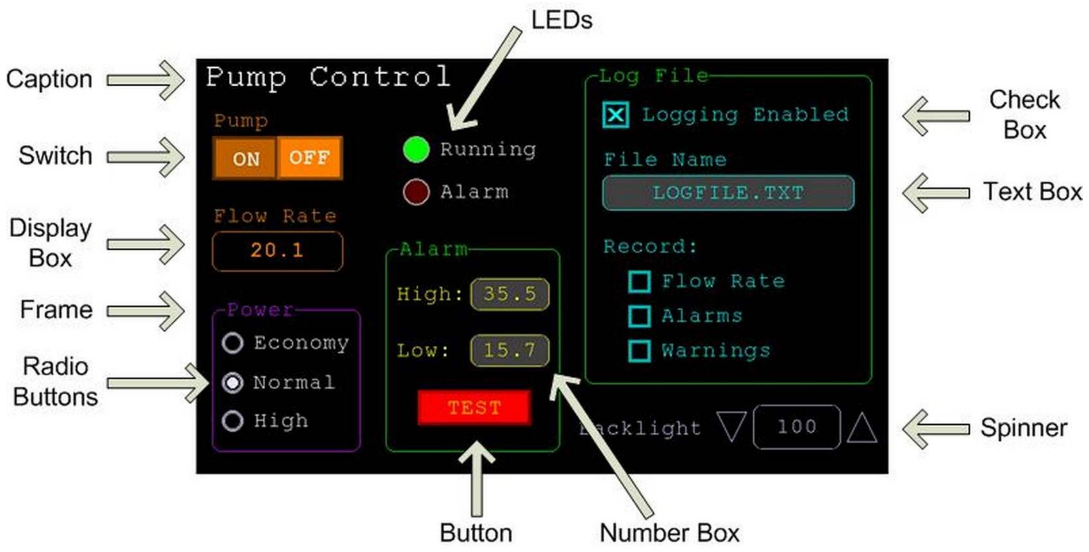
To use the GUI controls in the Micromite eXtreme the memory required for the GUI controls must be allocated first by using the command OPTION GUI CONTROLS. Typically you would use the command like this:

```
      OPTION GUI CONTROLS 75
```

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

Defining Controls

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



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

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

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

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

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

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

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

The advanced graphics controls are:

Frame

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

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

LED

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

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

Check Box

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

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

Push Button

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

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

Switch

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

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

Radio Button

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

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

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

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

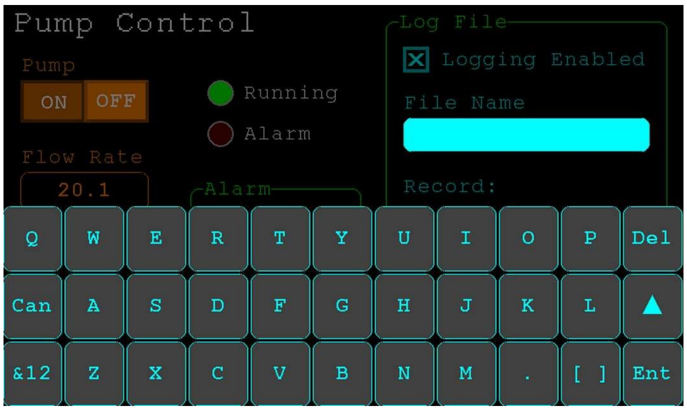
Display Box

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

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

Text Box

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



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

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

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

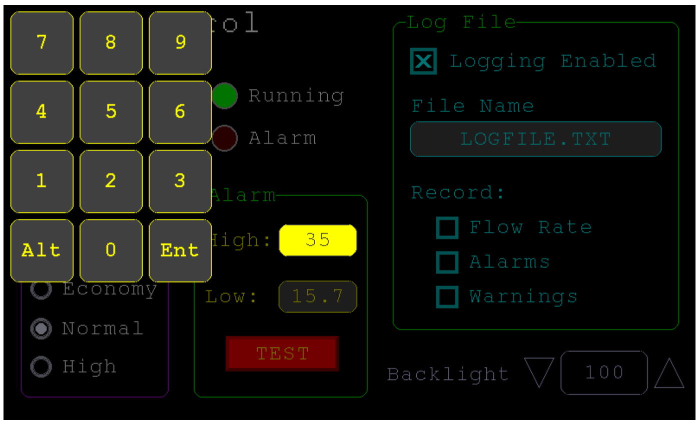
MMBasic will try to position the virtual keyboard on the screen to not obscure the text box that caused it to appear. A pen down interrupt will be generated just before the keyboard is deployed and a key up interrupt will be generated when the Enter or Cancel keys are touched and the keyboard is hidden.

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

```
GUI TEXTBOX CANCEL.
```

Number Box

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



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

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

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

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

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

Formatted Number Box

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

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

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

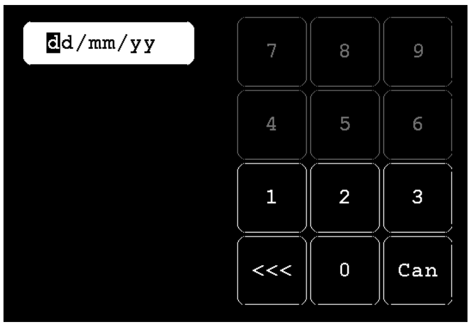
- DATE1
  Date in UK/Aust/NZ format (dd/mm/yy)
- DATE2
  Date in USA format (mm/dd/yy)
- DATE3
  Date in international format (yyyy/mm/dd)
- TIME1
  Time in 24 hour notation (hh:mm)
- TIME2
  Time in 24 hour notation with seconds (hh:mm:ss)
- TIME3
  Time in 12 hour notation (hh:mm AM/PM)
- TIME4
  Time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

For example:

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

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

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



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

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

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

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

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

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

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

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

Spin Box

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

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

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

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

Caption

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

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

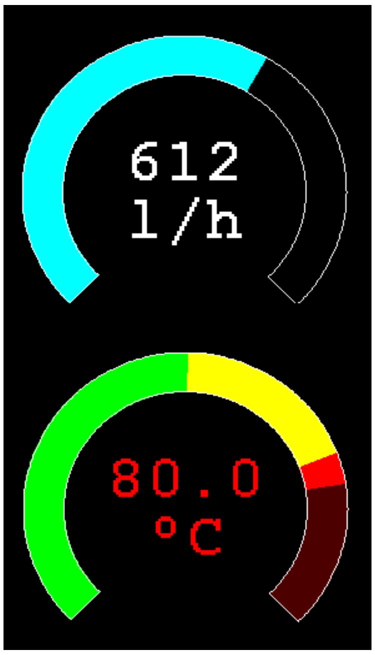
'Alignment' is zero to three characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE, BOTTOM.

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

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

Circular Gauge

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



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

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

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

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

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

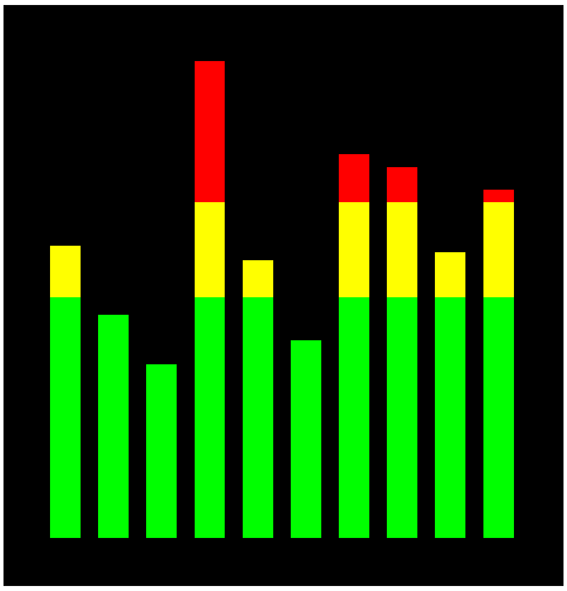
Specifically, 'c1' is the colour to be used for values up to 'ta'. 'c2' is the colour to be used for values between 'ta' and 'tb', 'c3' is used for values between 'tb' and 'tc' and c4 is used for values above 'tc'. Colours and thresholds not required can be left off then list. For example, for a two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

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

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

Bar Gauge

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



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

If the width is less that the height the bar gauge will be drawn vertically with the analogue graph growing from the bottom towards the top. Otherwise, if the width is more that the height, it will be drawn horizontally with the analogue graph growing from the left towards the right. In both cases 'StartX' and 'StartY' reference the top left coordinate of the bar graph while 'width' is the horizontal width and 'height' the vertical height.

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

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

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

Area

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

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

Interacting with Controls

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

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

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

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

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

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

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

GUI Sub-commands

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

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

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

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

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

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

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

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

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

This will disable the controls in the list. Disabled controls do not respond to touch and will be displayed dimmed. The keyword ALL can be used as the argument and that will disable all controls on the currently displayed page. For example:

```
GUI DISABLE ALL
```

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

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

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

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

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

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

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

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

MsgBox()

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

The syntax is:

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

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

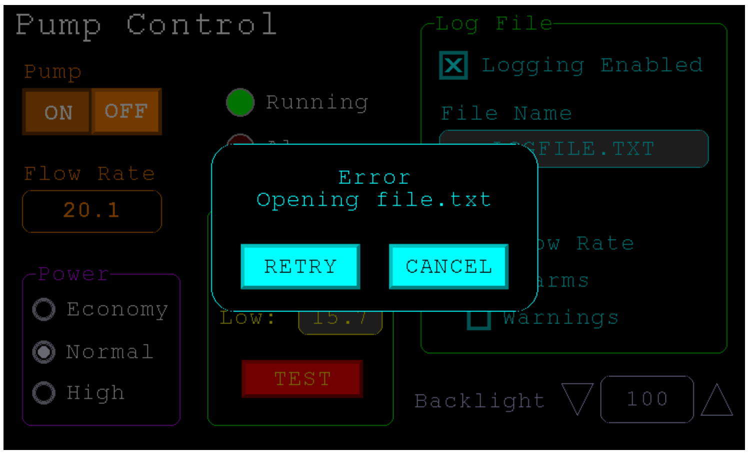
The font used will be the default font set using the FONT command and the colours used will be the defaults set by the COLOUR command. The box will be automatically sized taking into account the dimensions of the default font, the number of lines to display and the number of buttons specified.

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

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

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

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



Error
Opening file.txt

Advanced Graphics Programming Techniques

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

The User Should Be In Control

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

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

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

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

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

Program Structure

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

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

```
GUI CAPTION #1, "Speed (rpm)", 200, 50
GUI NUMBERBOX #2, 200, 100, 150, 40
CtrlVal(#2) = 100
GUI BUTTON #3, "RUN", 200, 350, 0, RGB(red)' label the number box
' define and draw the number box
' default value for the speed
' define and draw the RUN button
DO
IF CtrlVal(#3)<10 OR CtrlVal(#3)>200 THEN
GUI DISABLE #3
ELSE
GUI ENABLE #3
ENDIF' this runs in a loop forever
' check the speed setting
' disable RUN if it is invalid
' otherwise
' enable the RUN button
IF CtrlVal(#3) = 1 THEN
SetMotorSpeed CtrlVal(#2)
ELSE
SetMotorSpeed 0
ENDIF
LOOP' if the button is pressed
' make the motor run
' otherwise the button is up
' therefore set motor speed to zero
```

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

Disable Invalid Controls

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

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

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

Use Constants for Control Reference Numbers

The advanced controls use a reference number to identify the control. To make it easy to read and maintain your program you should define these numbers as constants with easy to recognise names.

For example, in the following program fragment MAIN_SWITCH is defined as a constant and this constant is used wherever the reference number for that control is required:

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

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

The reference number must be a number between 1 and the value set with the OPTION CONTROL command.

Increasing this number will consume more RAM and decreasing it will recover some RAM.

The Main Program Is Still Running

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

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

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

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

Use Interrupts and SELECT CASE Statements

Everything that happens on a screen using the advanced controls will be signalled by an interrupt, either touch down or touch up. So, if you want to do something immediately when a control is changed, you should do it in an interrupt. Mostly you will be interested in when the touch (or pen) is down but in some cases you might also want to know when it is released.

Because the interrupt is triggered when the pen touches any control or part of the screen you need to discover what control was being touched. This is best performed using the TOUCH(REF) function and the SELECT CASE statement.

For example, in the following fragment the subroutine PenDown will be called when there is a touch and the function TOUCH(REF) will return the reference number of the control being touched. Using the SELECT CASE the alarm LED will be turned on or off depending on which button is touched. The action could be any number of things like raising an I/O pin to turn on a physical siren or printing a message on the console.

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

The SELECT CASE can also test for other controls and perform whatever actions are required for them in their own section of the CASE statement.

The important point is that the maintenance of the controls (e.g. responding to the buttons and turning the alarm LED off or on) is done automatically without the main program being involved – it can continue doing something useful like calculating some control response, etc.

Touch Up Interrupt

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

Setting the colour on the touch down is easy. Just define a touch down interrupt and change the colour in the interrupt when that control is touched. However, to return the colour to white you need to detect when the touch has been removed from the control (i.e. touch up). This can be done with a touch up interrupt.

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

```
GUI INTERRUPT IntTouchDown, IntTouchUp
```

Within the touch up subroutine you can use the same structure as in the touch down sub but you need to find the reference number of the last control that was touched. This is because the touch has already left the screen and no control is currently being touched. To get the number of the last control touched you need to use the function TOUCH(LASTREF)

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

```
SUB IntTouchDown
SELECT CASE TOUCH(REF)
CASE ButtonRef
GUI FCOLOUR RGB(RED), ButtonRef
END SELECT
END SUB
SUB IntTouchUp
SELECT CASE TOUCH(LASTREF)
CASE ButtonRef
GUI FCOLOUR RGB(WHITE), ButtonRef
END SELECT
END SUB
```

Keep Interrupts Very Short

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

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

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

Multiple Screens

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

To allocate controls to a page you use the GUI SETUP nn command where nn refers to the page in the range of 1 to 32. When you have used this command any newly created controls will be assigned to that page. You can use GUI SETUP as many times that you want. For example, in the program fragment below the first two controls will be assigned to page 1, the second to page 2, etc.

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

By default only the controls setup as page 1 will be displayed and the others will be hidden.

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

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

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

It is possible to place the GUI PAGE commands in the touch down interrupt so that pressing a certain control or part of the screen will switch to another page.

Note that when ALL is used for the list of controls in commands such as GUI ENABLE ALL this only refers to the controls on the pages that are currently selected for display. Controls on other pages will be unaffected.

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

A typical usage of the GUI PAGE command is shown below.

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

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

Multiple Interrupts

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

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

```
GUI PAGE 4
GUI INTERRUPT P4keydown, P4keyup
..
GUI PAGE 5
GUI INTERRUPT P5keydown, P5keyup
..
```

Using Drawing Commands – GUI or Basic Drawing

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

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

Note that the CLS command (used to clear the screen) will automatically set any GUI controls on the screen to hidden (i.e. it does a GUI HIDE ALL before clearing the screen).

The main problem with mixing basic graphics and GUI controls occurs with the Text Box, Formatted Box and Number Box controls which display a virtual keyboard. This can erase any basic graphics and MMBasic will not know to restore them when the keyboard is removed. If you want to mix basic graphics with GUI controls you should:

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

Overlapping Controls

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

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

SD Card Support

The Micromite eXtreme has full support for programs, files and directories on the SD card. This includes opening files for reading, writing or random access and editing and running programs.

MMBasic will work with 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.

In the following note that:

- The filename can be a string expression, variable or constant. If it is a constant the string must be quoted (eg, KILL "MYPROG.BAS"). The exception is the RUN command where only a constant is allowed.
- Long file/directory names are supported in addition to the old 8.3 format.
- The maximum file/path length is 127 characters.
- Upper/lowercase characters and spaces are allowed although the file system is not case sensitive.
- Directory paths are allowed in file/directory strings. (ie, OPEN "/dir1/dir2/file.txt" FOR …).
- Forward slashes or back slashes are valid in paths between directories. Eg /dir/file.txt or \dir\file.txt.
- The current MMBasic time is used for file create and last access times.
- Up to ten files can be simultaneously open.
- Input and output commands and functions can also use file #0 which refers to the console.

There are 34 commands and functions related to the SD card. See the Commands/Functions section later in this manual for their full description).

Program Management Commands

Programs can reside on the SD card as .BAS files but must be loaded into memory with the LOAD, RUN, EDIT or LIST commands. Once in memory, a program can be directly operated on by the RUN, EDIT or LIST commands without the fname$. If a program in memory is changed, it can be written back to the SD Card with the SAVE command:

- ☐ LOAD fname$
- ☐ RUN “prog”
  Run a program. ‘prog’ must be a string constant (ie, not a variable).
- ☐ EDIT fname$
  Edit a program or text file.
- ☐ LIST fname$
  List on the console a program or text file.
- ☐ AUTOSAVE fname$
  Receive a file streamed by a computer connected to the serial console.
- ☐ XMODEM RECEIVE fname$
  Receive a file from a computer connected to the serial console using the XModem protocol.
- ☐ XMODEM SEND fname$
  Send a file to a computer connected to the serial console using the XModem protocol.

File Access Within a Program

Except for INPUT, LINE INPUT and PRINT the # in #fnbr is optional and may be omitted.

- OPEN fname$ FOR mode AS #fnbr
  Opens a file for reading or writing. 'fname$' is the file name, 'mode' can be INPUT, OUTPUT, APPEND or RANDOM, ‘fnbr’ is the file number (1 to 10).
- PRINT #fnbr, expression [[,;]expression] … etc
  Outputs text to the file opened as #fnbr.
- INPUT #fnbr, list of variables
  Read a list of comma separated data into the variables specified from the file previously opened as #fnbr.
- SEEK #fnbr, pos
  Will position the read/write pointer in a file that has been opened for RANDOM access to the 'pos' byte.
- LINE INPUT #fnbr, variable$
  Read a complete line into the string variable specified from the file previously opened as #fnbr.
- CLOSE #fnbr [,#fnbr] …
  Close the file(s) previously opened with the file number ‘#fnbr’.

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

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

File and Directory Management

- LIST FILES [wildcard] [,sortorder]
  Search the current directory and list the files/directories found.
- KILL fname$
  Delete a file.
- COPY oldfile$ TO newfile$
  Copy a file.
- RENAME oldfile$ AS newfile$
  Rename a file.
- MKDIR dname$
  Make a sub directory.
- CHDIR dname$
  Change into to the directory $dname. $dname can also be ".." (dot dot) for up one directory or "/" for the root directory.
- RMDIR dir$
  Remove, or delete, the directory ‘dir$’.
- And there are two functions that are handy for searching and managing files/directories.
- DIR$(fspec, type)
  Will search an SD card for files and return the names of the entries found.
- MM.INFO(function)
  Returns many types of information related to the SD card (size, free space, file size, etc).

Play Audio Files

- PLAY WAV | FLAC | MP3 file$ [, interrupt]
  Play a WAV, FLAC or MP3 audio file on the stereo audio output.
- PLAY MODFILE file$
  Play a MOD file on the stereo audio output.
- PLAY EFFECT filename$ [,interrupt]
  Play a WAV file at the same time as a MOD file is playing.

Load and Save Images

- LOAD BMP | GIF | JPG | PNG fname$
  Load a BMP, GIF, JPG or PNG image and display it on the VGA monitor.
- SAVE IMAGE fname$
  Save the current VGA monitor's screen image as a BMP file.

Sequential File Access

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

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

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

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

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

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

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

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

The first INPUT$() will read 12 characters and the second 3 characters. So the variable `ta$` will contain "The quick br" and the variable `tb$` will contain "own".

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

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

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

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

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

Random File Access

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

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

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

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

```
      SEEK #1, 257
```

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

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

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

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

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

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

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

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

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

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

Audio Output

The Micromite eXtreme can play WAV, FLAC and MP3 files from the SD card, generate synthesised music in the MOD format, create robot speech and sound effects as well as generate precise sine wave tones. All these are outputted on the audio socket. The ARM Cortex-M7 chip includes its own DAC (digital to analog converter) so an output filter network is not needed (as on the original Colour Maximite).

Playing WAV, MP3 and FLAC Files

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

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

```
     PLAY WAV file$, interrupt
                                                  or
     PLAY MP3 file$, interrupt
                                                  or
     PLAY FLAC file$, interrupt
```

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

Background Music

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

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

Generating Sine Waves

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

The syntax of the command is:

```
     PLAY TONE left, 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 'dur' specifies the number of milliseconds that the tone will sound for.

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

The frequency can be from 1 Hz to 20 KHz and is very accurate (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 TTS command which will generate robotic speech.
- PLAY SOUND which will generate an output based on a mixture of sine, square, noise, etc waveforms.
- PLAY EFFECT command which will play a WAV at the same time as a MOD file is playing.

Using PLAY

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

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

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

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

Similarly the following program will not work either:

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

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

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

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

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

Utility Commands

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

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

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

```
PLAY NEXT                     ‘ Skip to the next file.
PLAY PREVIOUS                 ‘ Skip to the previous file.
```

Using the I/O pins

The 40-pin ribbon connector on the 144 pin Backpack PCB 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 Appendix B.

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 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 23, AIN
> PRINT PIN(23)
 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 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 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 Micromite eXtreme 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 COUNTING 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 Micromite eXtreme 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 Micromite eXtreme 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 Micromite eXtreme 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.

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 order of the interrupts as defined with SETPIN. 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.

Special Device Support

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

Infrared Remote Control Decoder

You can easily add a remote control to the Micromite eXtreme. 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:

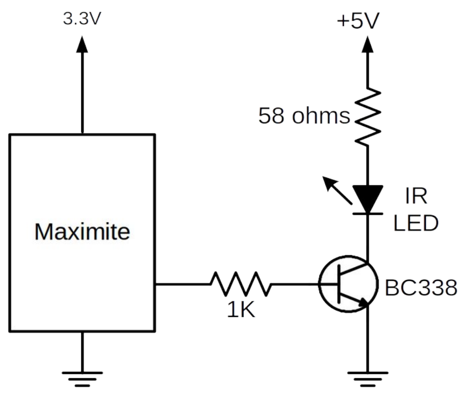
```
IR DevCode, KeyCode, IR_Int        ' start the IR decoder
DO
  ' < body of the program >
LOOP

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.

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 Micromite eXtreme 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 on the right 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:

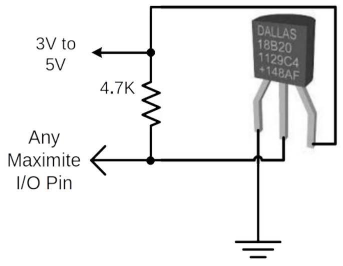
```
      IR SEND pin, dev, cmd
```

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 Micromite eXtreme'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 (4.7KΩ) 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 eternal I/O connector.
The DS18B20 should be powered separately by a 3V to 5V supply as shown on the right. 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 ºC and is accurate to ±0.5 ºC. If there is an error during the measurement the returned value will be 1000.

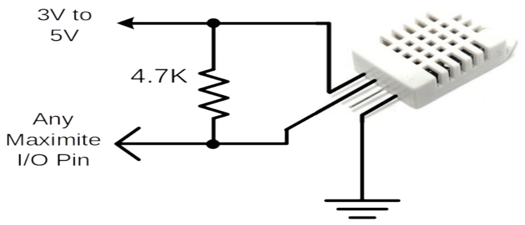
The time required for the overall measurement is 200ms and the running program will halt for this period while the measurement is being made. This also means that interrupts will be disabled for this period. If you do not want this you can separately trigger the conversion using the TEMPR START command 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:

```
      DHD22 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
```

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.

LCD Display

The LCD command will display text on a standard LCD module with the minimum of programming effort.

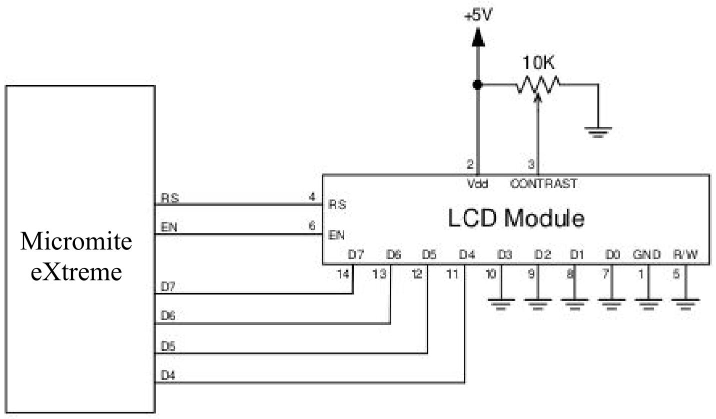
This command will work with LCD modules that use the KS0066, HD44780 or SPLC780 controller chip and have 1, 2 or 4 lines. Typical displays include the LCD16X2 (futurlec.com), the Z7001 (altronics.com.au) and the QP5512 (jaycar.com.au). eBay is another good source where prices can range from $10 to $50.

To setup the display you use the BITBANG LCD INIT command:

```
      BITBANG LCD INIT d4, d5, d6, d7, rs, en
```

d4, d5, d6 and d7 are the numbers of the I/O pins that connect to inputs D4, D5, D6 and D7 on the LCD module (inputs D0 to D3 and R/W on the module should be connected to ground). 'rs' is the pin connected to the register select input on the module (sometimes called CMD or DAT). 'en' is the pin connected to the enable or chip select input on the module.

Any I/O pins on the Micromite eXtreme can be used and you do not have to set them up beforehand (the LCD command automatically does that for you). The following shows a typical set up.



To display characters on the module you use the LCD command:

```
      BITBANG LCD line, pos, data$
```

Where line is the line on the display (1 to 4) and pos is the position on the line where the data is to be written (the first position on the line is 1). data$ is a string containing the data to write to the LCD display. The characters in data$ will overwrite whatever was on that part of the LCD.

The following shows a typical usage where d4 to d7 are connected to pins GP2 to GP4 on the Micromite eXtreme, rs is connected to pin GP23 and en to pin GP24.

```
      BITBANG LCD INIT GP2, GP3, GP4, GP5, GP23, GP24
      BITBANG LCD 1, 2, "Temperature"
      BITBANG LCD 2, 6, STR$(TEMPR(GP15))
      ' DS18B20 connected to pin GP15
```

Note that this example also uses the TEMPR() function to get the temperature (described above).

Keypad Interface

A keypad is a low tech method of entering data into a Micromite eXtreme based system. The Micromite eXtreme supports either a 4x3 keypad or a 4x4 keypad and the monitoring and decoding of key presses is done in the background. When a key press is detected an interrupt will be issued where the program can deal with it.

Examples of a 4x3 keypad and a 4x4 keypad are the Altronics S5381 and S5383 (go to www.altronics.com).

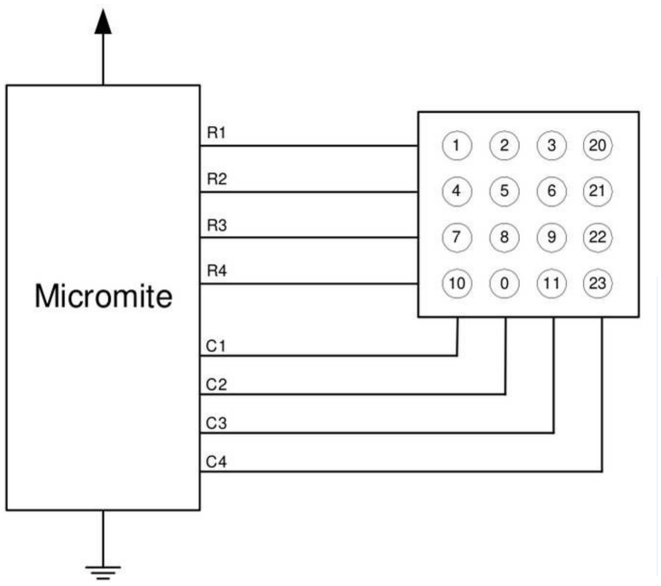
To enable the keypad feature you use the command:

```
      KEYPAD var, int, r1, r2, r3, r4, c1, c2, c3, c4
```

Where var is a variable that will be updated with the key code and int is the name of the interrupt subroutine to call when a new key press has been detected. r1, r2, r3 and r4 are the pin numbers used for the four row connections to the keypad (see the diagram below) and c1, c2, c3 and c4 are the column connections. c4 is only used with 4x4 keypads and should be omitted if you are using a 4x3 keypad.

Any I/O pins on the Micromite eXtreme can be used and you do not have to set them up beforehand, the KEYPAD command will automatically do that for you.

The detection and decoding of key presses is done in the background and the program will continue after this command without interruption. When a key press is detected the value of the variable var will be set to the number representing the key (this is the number inside the circles in the diagram to the right). Then the interrupt will be called.



For example:

```
Keypad KeyCode,
KP_Int,GP2,GP3,GP4,GP5,GP21,GP22,GP26
' 4x3 keybd
DO
< body of the program >
LOOP
SUB KP_Int
PRINT "Key press = " KeyCode
END SUB

' a key press has been detected
```

WS2812 Support

The Micromite eXtreme has built in support for the WS2812 multicolour LED chip. This chip needs a very specific timing to work properly and with the BITBANG WS8212 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:

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

Note that the pin must be set to a digital output before this command is used.

The colours%() array should be sized to have exactly the same number of elements as the number of LEDs to be driven. Each element in the array should contain the colour in the normal RGB888 format (0 - &HFFFFFF). There is no limit to the size of the WS2812 string supported.

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

O = original WS2812
B = WS2812B
S = SK6812

As an example:

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

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

LCD Display Panels

The Micromite eXtreme includes support for many LCD display panels using an SPI, I2C or parallel interface. These commands must be entered at the command prompt (not in a program) and will cause the Micromite eXtreme to restart. This has the side effect of disconnecting the USB console interface which will need to be reconnected.

Note that the maximum voltage on all the Micromite eXtreme I/O pins is 3.3V. Level shifting will be required if a display uses 5V levels for signalling.

SPI Based Display Panels

The SPI based display controllers share the SYSTEM SPI channel interface on the Micromite eXtreme with the touch controller (if present). An SD Card can also be configured to use the same pins. When this is done the pins allocated to the SYSTEM SPI will not be available to other MMBasic commands. The speed of drawing to SPI based displays will be largely unaffected by the CPU speed.

These panels are configured using the following commands. In all commands the parameters are:

- OR = This is the orientation of the display and it can be LANDSCAPE, PORTRAIT, RLANDSCAPE or RPORTRAIT. These can be abbreviated to L, P, RL or RP. The R prefix indicates the reverse or "upside down" orientation.
- DC = Display Data/Command control pin.
- RESET = Display Reset pin (when pulled low).
- CS = Display Chip Select pin.

Any free pins can be used.

```
OPTION LCDPANEL ILI9341, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a TFT display using the ILI9341 controller. This supports 320 * 240 resolution. Displays using this controller are capable of transparent text and will work with the BLIT and BLIT READ commands.

```
OPTION LCDPANEL ILI9163, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a TFT display using the ILI9163 controller. This supports 128 * 128 resolution.

```
OPTION LCDPANEL ILI9481, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a TFT display using the ILI9481 controller (Pi HAT version). This supports 480 * 320 resolution.

```
OPTION LCDPANEL ILI9481IPS, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises an IPS display using the ILI9481 controller. This supports 480 * 320 resolution.

```
OPTION LCDPANEL ILI9488, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a TFT display using the ILI9488 controller. This supports 480 * 320 resolution.
Note that this controller has an issue with the MISO pin which interferes with the touch controller. For this display to work **the MISO pin must not be connected.**

```
OPTION LCDPANEL ILI9488W, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a TFT display using the ILI9488 controller. This supports the Waveshare 3.5" display as used on their Pico Eval board and the normal 3.5" display adapter.

```
OPTION LCDPANEL N5110, OR, DC, RESET, CS [,contrast]
```

Initialises a LCD display using the Nokia 5110 controller. This supports 84 * 48 resolution. An additional parameter LCDVOP may be specified to control the contrast of the display. Try contrast values between &HA8 and &HD0 to suit your display, default if omitted is &HB1

```
OPTION LCDPANEL SSD1306SPI, OR, DC, RESET, CS [,offset]
```

Initialises a OLED display using the SSD1306 controller with an SPI interface. This supports 128 * 64 resolution. An additional parameter offset may be specified to control the position of the display. 0.96" displays typically need a value of 0. 1.3" displays typically need a value of 2. Default if omitted is 0.

```
OPTION LCDPANEL SSD1331, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a colour OLED display using the SSD1331 controller. This supports 96 * 64 resolution.

```
OPTION LCDPANEL ST7735, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a TFT display using the ST7735 controller. This supports 160 * 128 resolution.

```
OPTION LCDPANEL ST7735S, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a IPS display using the ST7735S controller. This supports 160 * 80 resolution.

```
OPTION LCDPANEL ST7735S_W, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a Waveshare 128x128 ST7735S display. This supports 128 * 128 resolution.

```
OPTION LCDPANEL ST7789, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a IPS display using the 7789 controller. This supports 240 * 240 resolution. NOTE: display boards without a CS pin are not currently supported on the Micromite eXtreme unless modified.

```
OPTION LCDPANEL ST7789_135, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a IPS display using the 7789 controller. This supports 240 * 135 resolution. NOTE: display boards without a CS pin are not currently supported on the Micromite eXtreme unless modified.

```
OPTION LCDPANEL ST7789_320, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a IPS display using the 7789 controller. This type supports the 320 * 240 resolution display from Waveshare (https://www.waveshare.com/wiki/Pico-ResTouch-LCD-2.8).
These are capable of transparent text and will work with the BLIT and BLIT READ commands. NOTE: display boards without a CS pin are not currently supported on the Micromite eXtreme unless modified.

```
OPTION LCDPANEL GC9A01, OR, DC, RESET, CS [,BACKLIGHTPIN]
```

Initialises a IPS display using the GC9A01 controller. This supports 240 * 240 resolution.

```
OPTION LCDPANEL ST7920, OR, DC, RESET
```

Initialises a LCD display using the ST7920 controller. This supports 128 * 64 resolution. Note this display does not support a chip select so the SPI bus cannot be shared if this display is used.

I2C Based LCD Panels

The I2C based display controllers use the SYSTEM I2C pins as per the pinout for the specific device. Other I2C devices can share the bus subject to their addresses being unique.

If an I2C display is configured it will not be necessary to "open" the I2C port for an additional device (I2C OPEN), I2C CLOSE is blocked, and all I2C devices must be capable of 100KHz operation. The I2C bus speed is not affected by changes to the CPU clock speed.

In all commands the parameters OR is the orientation of the display and it can be LANDSCAPE, PORTRAIT, RLANDSCAPE or RPORTRAIT. These can be abbreviated to L, P, RL or RP. The R prefix indicates the reverse or "upside down" orientation.

These panels are configured using the following commands.

```
OPTION LCDPANEL SSD1306I2C, OR [,offset]
```

Initialises a OLED display using the SSD1306 controller with an I2C interface. This supports 128 * 64 resolution. An additional parameter offset may be specified to control the position of the display. 0.96" displays typically need a value of 0. 1.3" displays typically need a value of 2. Default if omitted is 0. NOTE: many cheap I2C versions of SSD1306 displays do not implement I2C properly due to a wiring error. This seems to be particularly the case with 1.3" variants

```
OPTION LCDPANEL SSD1306I2C32, OR
```

Initialises a OLED display using the SSD1306 controller with an I2C interface. This supports 128 * 32 resolution.

SSD1963 Based LCD Panels

In addition to the SPI and I2C based controllers, the Micromite eXtreme supports LCD displays using the SSD1963 controller as illustrated below.

These use a parallel interface, are available in sizes from 4.3" to 9" and have better specifications than the smaller displays. All these displays have an SD Card socket which is fully supported by MMBasic on the Micromite eXtreme.

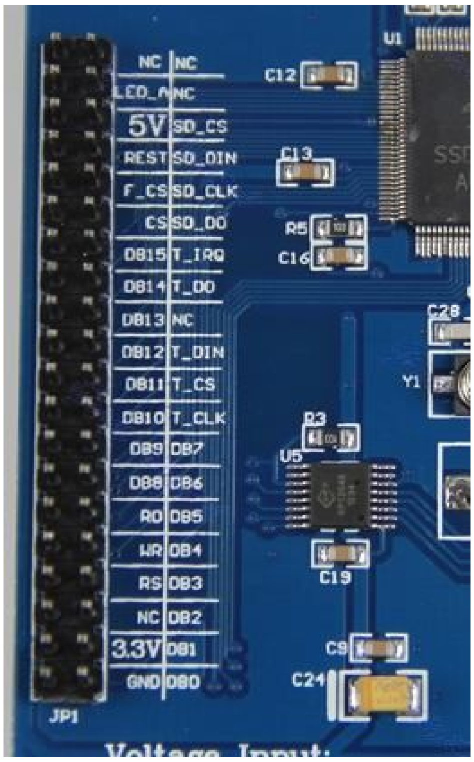
On eBay you can find suitable displays by searching for the controller’s name (SSD1963).

Because the SSD1963 controller uses a parallel interface the Micromite eXtreme can transfer data much faster than an SPI interface resulting in a very quick screen update.

These displays are also much larger, have more pixels and are brighter. MMBasic drives them using 24-bit true colour for a full colour rendition (16 million colours).

The characteristics of these displays are:

- A 4.3, 5, 7, 8 or 9 inch display
- Resolution of 480 x 272 pixels (4.3" version) or 800 x 480 pixels (5”, 7", 8" or 9" versions).
- An SSD1963 display controller with a parallel interface (8080 format).
- A touch controller (SPI interface).
- A full sized SD Card socket.



There are a number of different designs using the SSD1963 controller but fortunately most Chinese suppliers have standardised on a single connector as illustrated on the right.

It is strongly recommended that any display purchased has a matching connector – this provides some confidence that the manufacturer has followed the standard that the Micromite eXtreme is designed to use.

The SSD1963 can operate in either 8 bit or 16 bit paralled mode – see details below for OPTION configurations.

Connecting an SSD1963 Based LCD Panel

The SSD1963 controller uses a parallel interface while the touch controller and SD Card use an SPI interface. The touch and SD Card features are optional but if they are used they will use the second SPI port (SPI2).

The following table lists the connections required between the display board and the Micromite eXtreme to support the SSD1963 interface and the LCD display. The touch controller and SD Card interfaces are listed further below.

SSD1963 Display	Description	64 Pin Backpack Micromite eXtreme	100 Pin Backpack Micromite eXtreme	144 Pin Backpack Micromite eXtreme
DB0	Parallel data bus bit 0	58	25	114
DB1	Parallel data bus bit 1	61	24	115
DB2	Parallel data bus bit 2	62	23	116
DB3	Parallel data bus bit 3	63	22	117
DB4	Parallel data bus bit 4	64	21	131
DB5	Parallel data bus bit 5	1	20	132
DB6	Parallel data bus bit 6	2	26	133
DB7	Parallel data bus bit 7	3	27	7
DB8	Parallel data bus bit 8	N/A	32	8
DB9	Parallel data bus bit 9	N/A	33	
DB10	Parallel data bus bit 10	N/A	34	10
DB11	Parallel data bus bit 11	N/A	35	27
DB12	Parallel data bus bit 12	N/A	41	9
DB13	Parallel data bus bit 13	N/A	42	28
DB14	Parallel data bus bit 14	N/A	43	29
DB15	Parallel data bus bit 15	N/A	44	30
CS	LCD Chip select (active low)	Configurable – usually permanently low		

WR	Write	28	97	141
RD	Read	Configurable		
RS	Command/Data	27	96	140
RESET	Reset the SSD1963	21	95	139
LED_A	Backlight control for an unmodified display panel	Configurable		
5V	5V power for the backlight on some displays (most use 3.3V for this)			
3.3V	Power supply			
GND	Ground			

The following table lists the connections required to support the touch controller interface:

SSD1963 Display	**Description**	**64 Pin Backpack Micromite eXtreme**	**100 Pin Backpack Micromite eXtreme**	**144 Pin Backpack Micromite eXtreme**
T_CS	Touch Chip Select	52	19	118
T_IRQ	Touch Interrupt	17	17	121
T_DIN	Touch Data In (SPI2 MOSI)	41	66	95
T_CLK	Touch SPI2 Clock	4	10	14
T_DO	Touch Data Out (SPI2 MISO)	47	72	105

The following table lists the connections required to support the SD Card connector:

SSD1963 Display	**Description**	**64 Pin Backpack Micromite eXtreme**	**100 Pin Backpack Micromite eXtreme**	**144 Pin Backpack Micromite eXtreme**
SD_CS	SD Card Chip Select	4	4	93
SD_DIN	SD Card Data In (SPI2 MOSI)	41	66	95
SD_CLK	SD Card SPI2 Clock	4	10	14
SD_DO	SD Card Data Out (SPI2 MISO)	47	72	105

Where a Micromite eXtreme connection is listed as "Recommend" the specific pin should be specified in the appropriate OPTION command (see below).

Generally 7 inch and larger displays have a separate pin on the connector (marked 5V) for powering the backlight from a 5V supply. If this pin is not provided the backlight power will be drawn from the 3.3V pin.

Note that the power drawn by the backlight can be considerable. For example, a 7 inch display will typically draw 330 mA from the 5V pin. The current drawn by the backlight can cause a voltage drop on the LCD display panel’s ground pin which can in turn shift the logic levels as seen by the display controller resulting in corrupted colours or text. An easy way of diagnosing this effect is to reduce the CPU speed to (say) 48MHz. If this fixes the problem it is a strong indication that this is the cause.

Soldering power and ground wires direct to the LCD display panel’s PCB is one workaround. Care must be taken with display panels that share the SPI port between a number of devices (SD Card, touch, etc). In this case all the Chip Select signals must configured in MMBasic or disabled by a permanent connection to 3.3V. If this is not done the pin will float causing the wrong controller to respond to commands on the SPI bus.

On the Micromite eXtreme either SPI channel can used to communicate with the touch controller and the SD Card interface as defined by the OPTION SYSTEM SPI setting. If this is set, that SPI channel will be unavailable to BASIC programs (which can use the other SPI channel).

Configuring an SSD1963 Based LCD Panel

To use the display MMBasic must be configured using the OPTION LCDPANEL command which is normally entered at the command prompt. Every time the Micromite eXtreme is restarted MMBasic will automatically initialise the display.

The syntax is:

```
OPTION LCDPANEL controller, orientation
```

Where: '`controller`' for 8 bit parallel mode can be:

- SSD1963_4
  For a 4.3 inch display
- SSD1963_5
  For a 5 inch display
- SSD1963_5A
  For an alternative version of the 5 inch display if SSD1963_5 does not work
- SSD1963_7
  For a 7 inch display
- SSD1963_7A
  For an alternative version of the 7 inch display if SSD1963_7 does not work.
- SSD1963_8
  For 8 inch or 9 inch displays.

For 16 bit paralled mode, ‘`controller`’ can be:

- SSD1963_4_16
  For a 4.3 inch display
- SSD1963_5_16
  For a 5 inch display
- SSD1963_5A_16
  For an alternative version of the 5 inch display if SSD1963_5 does not work
- SSD1963_7_16
  For a 7 inch display
- SSD1963_7A_16
  For an alternative version of the 7 inch display if SSD1963_7 does not work.
- SSD1963_8_16
  For 8 inch or 9 inch displays.

Additionally, there are 16 bit versions of the ILI9341, OTM8009A and NT35510 LCD Panel Controllers:

- ILI9341_16
- IPS_4_16 (supports OTM8009A and NT35510)

'orientation' can be LANDSCAPE, PORTRAIT, RLANDSCAPE or RPORTRAIT. These can be abbreviated to L, P, RL or RP. The R prefix indicates the reverse or "upside down" orientation.

This command only needs to be run once. From then on MMBasic will automatically initialise the display on startup or reset. In some circumstances it may be necessary to interrupt power to the LCD panel while the Micromite eXtreme is running (eg, to save battery power) and in that case the GUI RESET LCDPANEL command can be used to reinitialise the display.

If the LCD panel is no longer required the command OPTION LCDPANEL DISABLE can be used which will return the I/O pins for general use.

To verify the configuration you can use the command OPTION LIST to list all options that have been set including the configuration of the LCD panel.

To test the display you can enter the command GUI TEST LCDPANEL. You should see an animated display of colour circles being rapidly drawn on top of each other. Press the space bar on the console's keyboard to stop the test.

8 and 9 inch Displays

The controller configuration SSD1963_8 has only been tested with the 8 and 9 inch displays made by EastRising (available at www.buydisplay.com). These must be purchased as a TFT LCD panel with 8080 interface, 800x480 pixel LCD, SSD1963 display controller and XPT2046 touch controller. Note that the EastRising panels use a non-standard interface connector pin-out so you will need to refer to their data sheets when connecting these to the Micromite eXtreme. A suitable adaptor to convert to the standard 40-pin connection can be purchased from:

https://www.rictech.nz/micromite-products

Backlight Control

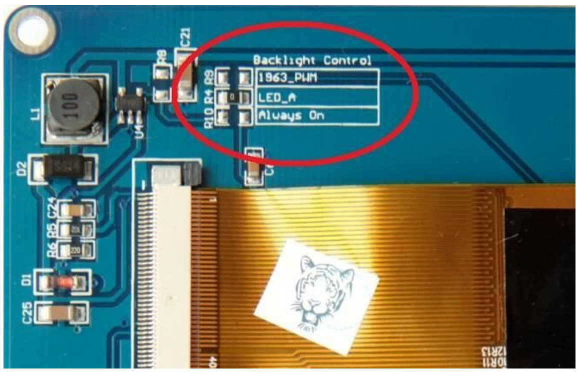
For the ILI9163, ILI9341, ST7735, ST7735S, SSD1331, ST7789, ILI9481, ILI9488, ILI9488W, ST7789_135 and ST7789_320 displays an optional parameter ', backlight' can be added to the end of the configuration parameters which specifies a pin to use to control the brightness of the backlight. This will setup a PWM output on that pin with a frequency of 1KHz and an initial duty cycle of 99%.

You can then use the BACKLIGHT command to change the brightness between 0 and 100%. The PWM channel is blocked for normal PWM use and must not conflict with the PWM channel that may be set up for audio. For example:

```
    OPTION LCDPANEL ILI9341, OR, DC, RESET, CS, GP11
```

The backlight can then be set to 40% with this command:

```
    BACKLIGHT 40
```

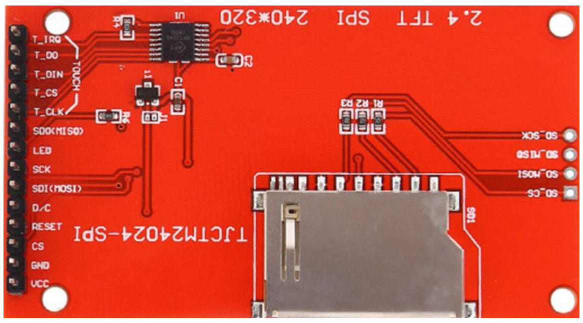


Most SSD1963 based LCD panels have three pairs of solder pads on the PCB which are grouped under the heading "Backlight Control" as illustrated on the right.

Normally the pair marked "LED-A" are shorted together with a zero ohm resistor and this allows control of the backlight's brightness with a PWM (pulse width modulated) signal on the LED-A pin of the display panel's main connector.

However, it is better to use the SSD1963 controller to generate this signal as it frees up one I/O pin. To use the SSD1963 for brightness control the zero ohm resistor should be removed from the pair marked "LED-A" and used to short the nearby pair of solder pads marked "1963-PWM". The Micromite eXtreme can then control the brightness via the SSD1963 controller using the BACKLIGHT command.

Example SPI LCD Panel Configuration



The following is a summary of how a typical LCD panel using an ILI9341 controller can be connected. This example supports the SD Card socket, the LCD display and the touch interface.

Typical panels can be found on ebay.com and similar sites by searching for the keyword "ILI9341". Make sure that the connections on the rear of the panel resemble that shown below: The panel should be connected to the Micromite eXtreme as illustrated:

ILI9341 Display	**Description**	**64 pin Micromite eXtreme**	**100 pin Micromite eXtreme**	**144 pin Micromite eXtreme**
T_IRQ	Touch Interrupt	Configurable		
T_DO	Touch Data Out (MISO)			
T_DIN	Touch Data In (MOSI)			
T_CS	Touch Chip Select	Configurable		
T_CLK	Touch SPI Clock			
SDO (MISO)	Display Data Out (MISO)			
LED	Power supply for backlight (see below)			
SCK	Display SPI Clock			
SDI (MOSI)	Display Data In (MOSI)			
D/C	Display Data/Command Control			
RESET	Display Reset (active low)			
CS	Display Chip Select			
GND	Ground			
VCC	5V supply (the display typically draws less then 10mA)			

Note: Be careful to ground yourself when handling the display as the ILI9341 controller is sensitive to static discharge and can be easily destroyed. Where a Micromite connection is listed as "configurable" the specific pin should be specified with the OPTION

```
      LCDPANEL or OPTION TOUCH commands (see below).
```

The connector to the right of the picture of the ILI9341 LCD Display above is the SD Card interface. Connections are as follows:

SSD1963 Display	**Description**	**64 pin Micromite eXtreme**	**100 pin Micromite eXtreme**	**144 pin Micromite eXtreme**
SD_CS	SD Card Chip Select			
SD_DIN	SD Card Data In (MOSI)			
SD_CLK	SD Card SPI Clock			
SD_DO	SD Card Data Out (MISO)			

The backlight power (the LED connection) should be supplied from the main 5V supply via a current limiting resistor. A typical value for this resistor is 18Ω which will result in a LED current of about 63 mA. The value of this resistor can be varied to reduce the power consumption or to provide a brighter display.

Important: Care must be taken with display panels that share the SPI port between a number of devices (display controller, touch, etc). In this case all the Chip Select signals must be configured in MMBasic or disabled by a permanent connection to 3.3V. If this is not done any unconnected Chip Select pins will float causing the wrong controller to respond to commands on the SPI bus.

To match the above connections the following configuration commands should be entered, one by one at the command prompt:

```
OPTION SYSTEM SPI GP18, GP19, GP16
OPTION SDCARD GP22
OPTION LCDPANEL ILI9341, L, GP15, GP14, GP13
OPTION TOUCH GP12, GP11
```

These commands will be remembered and automatically applied on power up. Note that after each command is entered the Micromite eXtreme will restart, and the USB connection will be lost and must be reconnected.
Next the touch screen should be calibrated:

```
GUI CALIBRATE
```

You can then test the various components. The following will list the files on the SD Card, if it executes without error you can be assured that the SD Card interface is good.

```
FILES
```

The following will draw multiple colourful overlapping circles on the LCD screen which will confirm that the LCD is connected correctly:

```
GUI TEST LCDPANEL
```

Finally, the following will test the touch interface. When you touch the LCD screen a dot should appear on the screen at the exact point of the touch.

```
GUI TEST TOUCH
```

If this is not accurate you may have to run the GUI CALIBRATE command a second time taking greater care.

If you run into trouble getting the display to work it is worth disconnecting everything and clear the options with the command OPTION RESET so that you can start with a clean slate. Then reconnect it one stage at a time and configure and test each new stage as you progress. First OPTION SYSTEM SPI, then the LCD display, the touch interface and finally the SD Card.

Also note that the ILI9341 controller is sensitive to static discharge so, if the panel will not respond, it could be damaged and it would be worth testing with another panel.

Touch Support

Many LCD panels are supplied with a resistive touch sensitive panel and associated controller chip. MMBasic fully supports this interface on the Micromite eXtreme and this allows many of the physical knobs and switches used in a project to be implemented as on screen controls activated by touch.

Note that the maximum voltage on all the Micromite eXtreme I/O pins is 3.3V. Level shifting will be required if a display uses 5V levels for signalling.

Configuring Touch

The touch controller on an LCD panel uses the SPI protocol for communications and this needs to be specifically configured before the panel can be configured. This is the “system” SPI port which is the port that will be used for system use (SD Card, LCD display and the touch controller on a LCD panel). This SPI port will then not be available to BASIC programs (i.e., it is reserved) There are a number of ports and pins that can be used but these are the same as the configuration used for the example LCD panel interface previously in this manual. This command does not need to be repeated if the system SPI has already been configured:

```
      OPTION SYSTEM SPI GP18, GP19, GP16
```

To use the touch facility MMBasic must be told that it is available using the OPTION TOUCH command. This should be done after the LCD display has been configured. This command tells MMBasic what pins are used for the Chip Select and Interrupt signals. For example this sets Chip Select to the GP12 pin and Interrupt to GP11:

```
      OPTION TOUCH GP12, GP11
```

These commands must be entered at the command prompt and will cause the Micromite eXtreme to restart. This has the side effect of disconnecting the USB console interface which will need to be reconnected.

When the Micromite eXtreme is restarted MMBasic will automatically initialise the touch controller. To verify the configuration you can use the command OPTION LIST to list all options that have been set including the configuration of the display panel and touch.

Note that you can use many different configurations using various pin allocations – this is just an example based on the configuration commands listed above.

Care must be taken when the SPI port is shared between a number of devices (SD Card, touch, etc). In this case all the Chip Select signals must configured in MMBasic or alternatively disabled.

Calibrating the Touch Screen

Before the touch facility can be used it must be calibrated using the GUI CALIBRATE command.

This command will present a target in the top left corner of the screen. Using a pointy but blunt object (such as a toothpick) press exactly on the centre of the target and hold it down for at least a second. MMBasic will record this location and then continue the calibration by sequentially displaying the target in the other three corners of the screen for touch and calibration.

The calibration routine may warn that the calibration was not accurate. This is just a warning and you can still use the touch feature if you wish but it would be better to repeat the calibration using more care.

Following calibration you can test the touch facility using the GUI TEST TOUCH command. This command will blank the screen and wait for a touch. When the screen is touched a white dot will be placed on the display marking the position on the screen. If the calibration was carried out successfully the dot should be displayed exactly under the location of the stylus on the screen.

To exit the test routine you can press the space bar on the console’s keyboard.

Touch Functions

To detect if and where the screen is touched you can use the following functions in a BASIC program:

- TOUCH(X)
- Returns the X coordinate of the currently touched location or -1 if the screen is not being touched.
- TOUCH(Y)
- Returns the Y coordinate of the currently touched location or -1 if the screen is not being touched.
-
- TOUCH(DOWN)
- Returns true if the screen is currently being touched (this is much faster than TOUCH(X or Y)).

- TOUCH(UP)
- Returns true if the screen is currently NOT being touched (also faster than TOUCH(X or Y))
- TOUCH(LASTX)
- Returns the X coordinate of the last location that was touched.
- TOUCH(LASTY)
- Returns the Y coordinate of the last location that was touched.
- TOUCH(REF)
- Returns the reference number of the control that is currently being touched or zero if no control is being
- touched. See the section Advanced Graphics for more details.
- TOUCH(LASTREF)
- Returns the reference number of the control that was last touched.

The GUI BEEP Command

The Piezo buzzer specified in the OPTION TOUCH command can also be driven by a BASIC program using the command:

GUI BEEP msec

Where 'msec' is the number of milliseconds that the beeper should be driven. A time of 3ms produces a click while 100ms produces a short beep.

Touch Interrupts with no Advanced GUI controls

An interrupt can be set on the IRQ pin number that was specified when the touch facility was configured. To detect touch down the interrupt should be configured as INTL (i.e., high to low).

For example, if the command OPTION TOUCH 7, 15 was used to configure touch the following program will print out the X and Y coordinates of any touch on the screen:

```
SETPIN 15, INTL, MyInt
DO : LOOP
SUB MyInt
PRINT TOUCH(X) TOUCH(Y)
END SUB
```

The interrupt can be cancelled with the command SETPIN pin, OFF.

Touch Interrupts with Advanced GUI controls

When the Advanced GUI controls are activated (by setting the number of GUI controls to a non-zero number using OPTION GUI CONTROLS) the GUI INTERRUPT command is used instead to setup a touch interrupt. The syntax is:

```
      GUI INTERRUPT down [, up]
```

Where 'down' is the subroutine to call when a touch down has been detected. And optionally 'up' is the subroutine to call when the touch has been lifted from the screen ('up' and 'down' can point to the same subroutine if required).

As an example, the following program will print out the X and Y coordinates of any touch on the screen:

```
GUI INTERRUPT MyInt
DO : LOOP
SUB MyInt
PRINT TOUCH(X) TOUCH(Y)
END SUB
```

Specifying the number zero (single digit) as the argument will cancel both up and down interrupts. Ie:

```
      GUI INTERRUPT 0
```

Porting Programs

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

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

Floating Point

In the Micromite eXtreme 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 Micromite. Generally, this will not cause an issue but it might mess up numbers that need to be printed in neat columns.

Editors note: BLIT and SPRITE are identical and use the same internal code.

BLIT

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

Sprites

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

SPRITE LOAD filename$ [,start_sprite_number]

This is compatible with the original Colour Maximite however the Micromite eXtreme 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).

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

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

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

Note that a space character is interpreted as BLACK 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.

Tip! The FRAMEBUFFER is similar in concept to Video RAM or memory. It can be a bank of internal memory within MMBasic for use with VGA style displays or the internal memory part of LCD controllers like SSD1963 or ILI9341.

There are additional SPRITE commands that use a CSUB as a convenient bank of memory used to hold SPRITE data. In conjunction with FRAMEBUFFER commands, SPRITEs (or more properly) TILES can be embedded within a program utilising the memory allocated to the CSUB. The structure of the SPRITE/TILE data in the CSUB is different to the original Maximite SPRITEs. For example:

SOUND and TONE

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

SD Card and File Related Commands

There is no drive B: in the Micromite eXtreme. The SD card is drive A:.

The command CHAIN is not implemented. The Micromite eXtreme has an exceptionally large program memory space so this feature is not necessary. The library and the LIBRARY commands are implemented.

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

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

Special Devices

LCD displays, Real Time Clocks (RTC) and keypads are all supported in the Micromite eXtreme.

Error Handling

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

MMBasic Implementation Characteristics

- Maximum program size (as plain text) is 512KB. Note that MMBasic tokenises the program when it is stored in program memory so the final size in program memory might vary from the plain text size
- Maximum length of a command line is 255 characters
- Maximum length of a variable name or a label is 31 characters
- Maximum number of variables 1024: 512 global and 512 local
- Maximum number of dimensions to an array is 5
- Maximum number of arguments to commands that accept a variable number of arguments is 32
- Maximum number of nested FOR…NEXT loops is 100
- Maximum number of nested DO…LOOP commands is 100
- Maximum number of nested GOSUBs, subroutines and functions (combined) is 50
- Maximum number of nested multi-line IF…ELSE…ENDIF commands is 20
- Maximum number of SELECT CASE statements is unlimited
- Maximum number of user defined subroutines and functions (combined): 500
- Maximum number of interrupt pins that can be configured: 10
- 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)
- Maximum number of sprite collisions is 8
- Maximum length of a line in a program is 240 characters

Predefined Read Only Variables

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

MM.CMDLINE$	A string representing the arguments on the command line when the program was run.
MM.DEVICE$	A string representing the device or platform that MMBasic is running on. Currently this variable will contain one of the following: "Maximite" on the standard Maximite and compatibles. "Colour Maximite" on the Colour Maximite and UBW32. "Micromite eXtreme" on the **Micromite eXtreme.** "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "Generic PIC32" for the generic version of MMBasic on a PIC32. "Micromite" on the PIC32MX150/250 "Micromite MkII" on the PIC32MX170/270 "Micromite Plus" on the PIC32MX470 "Micromite Extreme" on the PIC32MZ series
MM.ERRNO MM.ERRMSG$	If a statement caused an error which was ignored these variables will be set accordingly. MM.ERRNO is a number where non zero means that there was an error and MM.ERRMSG$ is a string representing the error message that would have normally been displayed on the console. They are reset to zero and an empty string by RUN, ON ERROR IGNORE or ON ERROR SKIP.
MM.HRES MM.VRES	Integers representing the horizontal and vertical resolution of the VGA display in pixels.
MM.I2C	Following an I^{2}C 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
MM.INFO() MM.INFO$()	These two versions can be used interchangeably but good programming practice would require that you use the one corresponding to the returned datatype.
MM.INFO(FCOLOUR)	Returns the current foreground colour
MM.INFO(FILESIZE file$)	Returns the size of 'file$' in bytes. Returns -1 if the file is not found. Returns -2 if file$ is the name of a valid directory

MM.INFO(FONT ADDRESS n)
 	
MM.INFO(FONT POINTER n)
MM.INFO(FONTHEIGHT)
MM.INFO(FONTWIDTH)
	Returns the address of the memory location containing the address of FONT n
	
	Returns a POINTER to the start of FONT n in memory
	
	Integers representing the height and width of the current font (in pixels).

MM.INFO(HPOS)
MM.INFO(VPOS)
	The current horizontal and vertical position (in pixels) following the last graphics or print command.

MM.INFO$(KEYBOARD)	Returns the string CONNECTED if a USB keyboard is connected and working. Otherwise returns "NOT CONNECTED"
MM.INFO(MODE)	Returns the video mode as a floating point number e.g. 1.8, 2.16, etc.
MM.INFO$(MODIFIED file$)	Returns the date/time that 'file$' was last modified. File$ can be a normal file or the name of a directory. Returns an empty string if the file or directory is not found.

MM.INFO(OPTION option)	Returns the current value of a range of options that affect how a program will run. “option” can be one of ANGLE, AUTORUN, BASE, BREAK, CONSOLE, CONSOLE PORT, DEFAULT, EXPLICIT, LEGACY, MOUSE, PROFILING, USBKEYBOARD, Y_AXIS
MM.INFO$(PIN pinno)	Returns the status of I/O pin 'pinno'. Valid returns are: INVALID, RESERVED, IN USE, and UNUSED
MM.INFO$(RESET)	Returns the cause of a firmware restart. The returned value will be one of “Switch” (i.e. pressing the reset switch), “Power-On”, “Software”, and “Watchdog” (NB the latter is the H/W watchdog and unrelated to the MMbasic version which causes a software reset)
MM.INFO$(SDCARD)	Returns the status of the SD card. Valid returns are: DISABLED, NOT PRESENT, READY, and UNUSED
MM.INFO$(SOUND)	Returns the status of the sound output device. Valid returns are: OFF, PAUSED, TONE, WAV, MP3, MODFILE, TTS, FLAC, DAC, SOUND
MM.INFO$(TRACK)	The name of the current audio track playing. This returns "OFF" if nothing is playing.
MM.ONEWIRE	Following a 1-Wire reset function this integer variable will be set to indicate the result of the operation as follows: 0 = Device not found. 1 = Device found
MM.WATCHDOG	An integer which is true (ie, 1) if MMBasic was restarted as the result of a Watchdog timeout (see the WATCHDOG command). False (ie, 0) if MMBasic started up normally.

Operators

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

Numeric Operators (Float or Integer)

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

The operators AND, OR and XOR are integer bitwise operators. For example PRINT (3 AND 6) will output 2. The other logical operations result in the integer 0 (zero) for false and 1 for true. For example the statement PRINT 4 >= 5 will print the number zero on the output and the expression A = 3 > 2 will store +1 in A.

String Operators

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

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

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 Micromite eXtreme is restarted. Options that are not permanent will be reset on startup.

	Permanent?	
OPTION ANGLE RADIANS | DEGREES		This command switches trig functions between degrees and radians. Acts on SIN, COS, TAN, ATN, ATAN2, MATH ATAN3, ACOS, ASIN This is a temporary option that is cleared to default (RADIANS) when programs end, after an error, or after Ctrl-C so should be set at the top of any program that requires to use angles in degrees.
OPTION AUTORUN OFF | ON	✓	Instructs MMBasic to automatically run the program in program flash memory on power up or restart (eg, by the WATCHDOG timer). This is turned off by the NEW command but other commands that might change program memory (EDIT, etc) do not change this setting. Entering the break key (default CTRL-C) at the console will interrupt the running program and return to the command prompt. If the CMM2 is set to run programs from RAM (OPTION RAM) then the firmware will look for the file AUTORUN.BAS on reset or power up and execute it if found.
OPTION BASE 0 | 1		Set the lowest value for array subscripts to either 0 or 1. This must be used before any arrays are declared and is reset to the default of 0 on power up.
OPTION BAUDRATE nbr	✓	Set the baud rate for the serial console to 'nbr'. This can be any value between 1200 (the minimum) and 1000000 (1MHz). This change is made immediately and will be remembered when the power is cycled. Using this command it is possible to set the console to an unworkable baud rate and in this case the baudrate should be reset using a USB keyboard and VGA monitor. If that is not available then reseting the firmware will reset the baudrate to the default of 115200.
OPTION BREAK nn		Set the value of the break key to the ASCII value 'nn'. This key is used to interrupt a running program. The value of the break key is set to CTRL-C key at power up but it can be changed to any keyboard key using this command (for example, OPTION BREAK 4 will set the break key to the CTRL-D key). Setting this option to zero will disable the break function entirely.
OPTION COLOURCODE ON 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. OPTION COLOURCODE REVERSE is the same as OPTION COLORCODE OFF except that the text will be in reverse video black on white. This will apply to the file manager and editor. The keyword COLORCODE (USA spelling) can also be used. On the serial console colour coding requires a terminal emulator that can interpret the appropriate escape codes.
OPTION CONSOLE PORT n	✓	Allow you to select the serial port to be used as the console. 'n' can be 1, 2, 3 or 4 (default). If this option is set all console output is directed to the specified com port. This allows easier remote operation using a wifi-uart or radio link. If the option is set to other than default, then COM4 is available on the USB-B connector.

OPTION CONSOLE SCREEN or OPTION CONSOLE SERIAL or OPTION CONSOLE BOTH		OPTION CONSOLE SCREEN will disable the serial console for both input and output and direct all output to the VGA monitor. This will allow theVGA output to update much faster. With this option enabled 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 and VGA screen for console input/output. This is the default on power up unless OPTION CONSOLE SAVE (see below) is used.
OPTION CONSOLE SAVE	✓	This will save the current console mode (see above) as the default stored mode. If you are using the Micromite eXtreme 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.
OPTION CRLF mode		Defines what the USB keyboard will send when the Enter key is pressed. 'mode' can be one of CR, LF or CRLF. The default is CRLF.
OPTION DEFAULT FLOAT | INTEGER | STRING | NONE		Used to set the default type for a variable which is not explicitly defined. If OPTION DEFAULT NONE is used then all variables must have their type explicitly defined. When a program is run the default is set to FLOAT for compatibility with previous versions of MMBasic.
OPTION EXPLICIT		Placing this command at the start of a program will require that every variable be explicitly declared using the DIM, LOCAL or STATIC commands before it can be used in the program. This option is disabled by default when a program is run. If it is used it must be specified before any variables are used.
OPTION EDIT FONT SMALL | NORMAL | MEDIUM | LARGE| VERY LARGE	✓	Sets the font to be used in the editor and file manager. The default is NORMAL which is a 8x12 pixel font.
OPTION F11 string$	✓	Define the string that will be generated when the F11 function key is pressed at the command prompt. Example: OPTION F11 “RUN “+chr$(34)+”myprog” +chr$(34)+chr$(13)+chr$(10). The maximum string length is 64 characters.
OPTION F12 string$	✓	Define the string that will be generated when the F12 function key is pressed at the command prompt. The maximum string length is 64 characters.
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 a new character repeats. Default is 600mSec, the valid range is 100 to 2000 mSec 'nextchars' is the time in milliseconds before subsequent character repeats. Default is 150mSec, the valid range is 25 to 2000 mSec
OPTION LIST		This will list the settings of any options that have been changed from their default setting and are the permanent type.

OPTION MILLISECONDS ON or OPTION MILLISECONDS OFF		Specifies that the TIME$ function will, or will not, include milliseconds as a decimal fraction of seconds in its output. The default is OFF.
OPTION MOUSE n [,sensitivity] OPTION MOUSE OFF	✓	Enables a mouse to be used in the filemanager and editor. Use channel 0 for a PS2 mouse, or 1 to 3 for a Hobbytronic connected mouse. Note if the mouse is enabled but not plugged in and working you will get an error when entering the filemanager or editor. Use OPTION MOUSE OFF to disable.
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. If a permanent lock is applied or the PIN number is lost the only way to recover is to reset the Micromite eXtreme firmware (as described in the section *Resetting MMBasic*).
OPTION RESET	✓	Reset all saved options to their default values.
OPTION RTC CALIBRATE ±n	✓	Used to calibrate the battery backed Real Time Clock that keeps time in the Micromite eXtreme. '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 remembered even after a firmware upgrade.
OPTION SLEEP n	✓	n is the number of minutes (0-255) before the monitor is turned off when at the command line. Set to 0 to disable. Any keypress will wake the monitor
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 setting is remembered even after a firmware upgrade.
OPTION SERIAL PULLUP ENABLE or OPTION SERIAL PULLUP DISABLE	✓	Enable or disable pullup resistors on the receive line of all serial ports including the serial console. The default is disabled.
OPTION STATUS ON | OFF	✓	Enable or disable the status line at the bottom of the VGA screen. The status line shows the date/time and the "current program filename" used by the commands RUN, LIST and EDIT when a file name is not specified. Default is ON.
OPTION TAB 2 | 3 | 4 | 8	✓	Set the spacing for the tab key. Default is 2.

OPTION VCC voltage		Specifies the voltage (Vcc) supplied to the MZ chip. When using the analog inputs to measure voltage the MZ2048 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.

Commands

Square brackets indicate that the parameter or characters are optional.

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

AUTOSAVE file$	Enter automatic program entry mode. This command will take lines of text from the console serial input and save them to a file on the SD card specified as 'file$'. This mode is terminated by pressing F1 on the console keyboard which will then cause the received data to be saved to the SD card. Terminating the transfer by pressing F2 will cause a similar save but then the saved program will be immediately loaded into program memory and run. Both F1 and F2 update the "current program name" which is used by RUN, LIST and EDIT when a file is not specified. The transfer can also be terminated using F6 which acts the same as F1 without updating the current program name. At any time this command can be aborted by Control-C which will leave program memory untouched. This is one way of transferring a BASIC program into the Maximite. The program to be transferred can be pasted into a terminal emulator and this command will capture the text stream and store it into program memory. It can also be used for entering a small program directly at the console input.
BITBANG HUMID pin,tvar,hvar[,DHT11]	Returns the temperature and humidity using the DHT22 sensor. Alternative versions of the DHT22 are the AM2303 or the RHT03 (all are compatible). 'pin' is the I/O pin connected to the sensor. Any I/O pin may be used. 'tvar' is the variable that will hold the measured temperature and 'hvar' is the same for humidity. Both must be present and both must be floating point variables. For example: HUMID 3, TEMP!, HUMIDITY! Temperature is measured in oC and the humidity is percent relative humidity. Both will be measured with a resolution of 0.1. If an error occurs (sensor not connected or corrupt signal) both values will be 1000.0. Normally the DHT22 should powered by 3.3V to keep its output below 3.6V for the Micromite eXtreme and the signal pin of should be pulled up by a 1K to 10K resistor (4.7K recommended) to 3.3V. The optional DHT11 parameter modifies the timings to work with the DHT11. Set to 1 for DHT11 and 0 or omit for DHT22.
BITBANG LCD INIT d4, d5, d6, d7, rs, en or BITBANG LCD line, pos, text$ or BITBANG LCD CLEAR or BITBANG LCD CLOSE	Display text on an LCD character display module. This command will work with most 1-line, 2-line or 4-line LCD modules that use the KS0066, HD44780 or SPLC780 controller (however this is not guaranteed). The LCD INIT command is used to initialise the LCD module for use. 'd4' to 'd7' are the I/O pins that connect to inputs D4 to D7 on the LCD module (inputs D0 to D3 should be connected to ground). 'rs' is the pin connected to the register select input on the module (sometimes called CMD). 'en' is the pin connected to the enable or chip select input on the module. The R/W input on the module should always be grounded. The above I/O pins are automatically set to outputs by this command. When the module has been initialised data can be written to it using the LCD command. 'line' is the line on the display (1 to 4) and 'pos' is the character location on the line (the first location is 1). 'text$' is a string containing the text to write to the LCD display. 'pos' can also be C8, C16, C20 or C40 in which case the line will be cleared and the text centred on a 8 or 16, 20 or 40 line display. For example: `    LCD 1, C16, "Hello"` LCD CLEAR will erase all data displayed on the LCD and LCD CLOSE will terminate the LCD function and return all I/O pins to the not configured state. See the section *Special Hardware Devices* for more details.

BITBANG WS2812 type, pin, nbr, value%[()]	This command outputs the required signals to drive one or more WS2812 LED chips connected to 'pin'. Note that the pin must be set to a digital output before this command is used.
	'type' is a single character specifying the type of chip being driven:
	O = original WS2812
	B = WS2812B
	S = SK6812
	‘nbr’ is the number of LEDs in the chain (1 to 256). The 'value%()' array should be an integer array sized to have exactly the same number of elements as the number of LEDs to be driven. Each element in the array should contain the colour in the normal RGB888 format (i.e. 0 to &HFFFFFF). If only one LED is connected then a single integer should be used for value% (ie, not an array).

BLIT READ [#]b, x, y, w, h [,pagenumber] or BLIT WRITE [#]b, x, y [,orientation] or BLIT CLOSE [#]b	Copy one section of the display screen to or from a memory buffer. BLIT READ will copy a portion of the display to the memory buffer '#b'. The source coordinate is 'x' and 'y' and the width of the display area to copy is 'w' and the height is 'h'. When this command is used the memory buffer is automatically created and sufficient memory allocated. The optional parameter page number specifies which page is to be read. The default is the current write page. This buffer can be freed and the memory recovered with the BLIT CLOSE command. Set the pagenumber to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command BLIT WRITE will copy the memory buffer '#b' to the display. The destination coordinate is 'x' and 'y' using the width/height of the buffer. The optional 'orientation' parameter defaults to 4 and specifies how the stored image data is changed as it is written out. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels BLIT CLOSE will close the memory buffer '#b' to allow it to be used for another BLIT READ operation and recover the memory used. Notes: • Sixty four buffers are available ranging from #1 to #64. • When specifying the buffer number the # symbol is optional. • All other arguments are in pixels.
BLIT x1, y1, x2, y2, w, h [, page] [,orientation] BLIT FRAMEBUFFER F/L/N, F/L/N, xfrom, yfrom, xto, yto, width, height [,col] BLIT MEMORY address%, x, y[,col] BLIT COMPRESSED address %, x, y[,col]	Copy one section of the display screen to another part of the display. The source coordinate is 'x1' and 'y1'. The destination coordinate is 'x2' and 'y2'. The width of the screen area to copy is 'w' and the height is 'h'. 'page' is the page number that the image data is read from; it is then written to the current write page as specified by the PAGE WRITE n command. If 'page' is omitted the data is read from the write page. The optional 'orientation' parameter specifies how the section of the screen is changed as it is copied. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels Set the page to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command. All arguments are in pixels and the source and destination can overlap. This copies a sprite (or block of csub memory) to the current framebuffer (or VGA screen). The size of the sprite is specified in the csub as two 16 bit integers as the first 4 bytes of each sprite. It can also be compressed. As before col is optional and specifies a colour that shouldn't be copied (-1 to 15) defaults to -1 if not specified indicating that all colours should be copied
BOX x, y, w, h [, lw] [,c] [,fill]	Draws a box on the VGA monitor with the top left hand corner at 'x' and 'y' with a width of 'w' pixels and a height of 'h' pixels.

	'lw' is the width of the sides of the box and can be zero. It defaults to 1. 'c' is the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'c', and fill can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
BOX AND_PIXELS x, y, w, h, colour [,pageno] BOX OR_PIXELS x, y, w, h, colour [,pageno] BOX XOR_PIXELS x, y, w, h, colour [,pageno]	Executes the requested logical operation between the pixels in the area defined on the page specified (defaults to the write page) with the colour specified
CALL usersubname$ [,usersubparameters,....]	This is an efficient way of programmatically calling user defined subroutines (see also the CALL() function). In many case it can allow you to get rid of complex SELECT and IF THEN ELSEIF ENDIF clauses and is processed in a much more efficient way. The “usersubname$” can be any string or variable or function that resolves to the name of a normal user subroutine (not an in-built command). The “usersubparameters” are the same parameters that would be used to call the subroutine directly. A typical use could be writing any sort of emulator where one of a large number of subroutines should be called depending on some variable. It also allows a way of passing a subroutine name to another subroutine or function as a variable.
CAMERA OPEN CAMERA OPEN FIFO CAMERA CAPTURE CAMERA SAVE “filename” CAMERA REGISTER register,value CAMERA CLOSE	Initialise an OV7670 camera ready for use Initialises an OV7670 camera with FIFO ready for use Captures an image from the camera to a connected 800x480 SSD1963 or 480x272 display. NB: the display must be set to 16-bit operation. Also available for the ILI9341 display on the 64-pin MMX saves the on-screen image to the SDcard. If the file extension is not specified then ".BMP" is appended. can be used to change the camera settings see the datasheet for details disables the camera and frees the allocated pins ILI9341 only on 64-PIN processor SSD1963 only on 144-PIN processor
CAT S$, N$	Concatenates the strings by appending N$ to S$. This is functionally the same a S$ = S$ + N$ but operates faster.
CHDIR dir$	Change the current working directory on the SD card to ‘dir$’ The special entry “..” represents the parent of the current directory and “.” represents the current directory. "/" is the root directory.

CIRCLE x, y, r [,lw] [, a] [, c] [, fill]	Draw a circle on the video output centred at 'x' and 'y' with a radius of 'r' on the VGA monitor. 'lw' is optional and is the line width (defaults to 1). 'c' is the optional colour and defaults to the current foreground colour if not specified. The optional 'a' is a floating point number which will define the aspect ratio. If the aspect is not specified the default is 1.0 which gives a standard circle 'fill' is the fill colour. It can be omitted or set to -1 in which case the circle will not be filled. All parameters can be expressed as arrays and the software will plot the number of circles as determined by the dimensions of the smallest array. 'x', 'y' and 'r' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'a', 'c', and fill can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
CLEAR	Delete all variables and recover the memory used by them.
CLOSE [#]nbr [,[#]nbr] …	Close the file(s) previously opened with the file number '#fnbr' Close the serial communications port(s) previously opened with the file number 'nbr'. The # is optional. Also see the OPEN command. The text "GPS" can be substituted for [#]nbr to close a communications port used for a GPS receiver.
CLS [colour]	Clears the VGA screen and the terminal emulator's screen. Optionally 'colour' can be specified which will be used for the VGA background when clearing the screen. Note: the terminal emulator screen is only cleared if the current write page is page 0
COLOUR fore [, back] or COLOR fore [, back]	Sets the default colour for commands (PRINT, etc) that display on the on the VGA monitor. 'fore' is the foreground colour, 'back' is the background colour. The background is optional and if not specified will default to black.
CONST id = expression [, id = expression] … etc	Create a constant identifier which cannot be changed once created. 'id' is the identifier which follows the same rules as for variables. The identifier can have a type suffix (!, %, or $) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.
CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The command is also used to exit a BREAK state and resume program execution following a MMDEBUG BREAK command. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with graphics, music, nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.
COPY fname1$ TO fname2$	Copy a file from 'fname1$' to 'fname2$'. Both are strings. A directory path can be used in both 'fname$' and 'fname$'. If the paths differ the file specified in 'fname$' will be copied to the path specified in 'fname2$' with the file name as specified.

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 bytes from the start of the CSUB 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 saved. Each 'hex' must be exactly eight hex digits representing the bits in a 32-bit word and be separated by one or more spaces or new lines. The command must be terminated by a matching END CSUB. Any errors in the data format will be reported when the program is run.
	During execution MMBasic will skip over any CSUB commands so they can be placed anywhere in the program.
	The type of each parameter can be specified in the definition. For example: CSub MySub integer, integer, string.
	This specifies that there will be three parameters, the first two being integers and the third a string. Note:
	• Up to ten arguments can be specified ('arg1', 'arg2', etc).
	• If a variable or array is specified as an argument the C routine will receive a pointer to the memory allocated to the variable or array and the C routine can change this memory to return a value to the caller. In the case of arrays, they should be passed with empty brackets e.g. arg(). In the CSUB the argument will be supplied as a pointer to the first element of the array.
	• Constants and expressions will be passed to the embedded C routine as pointers to a temporary memory space holding the value.
	• CSUBs must call routinechecks() every millisecond or so both to keep the USB keyboard active and also ensure the watchdog doesn't trigger. CSUBs that run to completion within a couple of milliseconds can ignore this.

CPU RESTART	Will force a restart of the processor. This will clear all variables and reset everything (eg, timers, COM ports, I^2C, etc) similar to a power up situation but without the power up banner. If OPTION AUTORUN has been set the program will restart.
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"
or
DATE$ = "DD/MM/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).
	The date is set to "01-01-2000" on first power up but the date will be remembered and kept updated as long as the battery is installed and can maintain a voltage of over 2.5V.
	Battery life should be 3 to 4 years even if the CMM2 is left powered off.
	Note that the time (set using the TIME$= command) will be lost when the power is cycled if a correct date is not set.

DEFINEFONT #n
  hex [[hex[…]
  hex [[hex[…]
END DEFINEFONT
	This will define an embedded font which can be used exactly same as the built in fonts (ie, selected using the FONT command or specified in the TEXT command).
	MMBasic must execute the font in order for it to be loaded. '#n' is the font's reference number (1 to 16). It can be the same as an existing font (except fonts 1, 6 and 7) and in that case it will replace that font.
	Each 'hex' must be exactly eight hex digits and be separated by spaces or new lines from the next. Multiple lines of 'hex' words can be used with the command terminated by a matching END DEFINEFONT.

DIM [type] decl [,decl]... where 'decl' is: var [length] [type] [init] 'var' is a variable name with optional dimensions 'length' is used to set the maximum size of the string to 'n' as in LENGTH n 'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword AS - as in AS FLOAT) 'init' is the value to initialise the variable and consists of: = <expression> DIM ….continued For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used. DIM Examples: DIM nbr(50) DIM INTEGER nbr(50) DIM name AS STRING DIM a, b$, nbr(100), strn$(20) DIM a(5,5,5), b(1000) DIM strn$(200) LENGTH 20 DIM STRING strn(200) LENGTH 20 DIM a = 1234, b = 345 DIM STRING strn = "text" DIM x%(3) = (11, 22, 33, 44)	Declares one or more variables (ie, makes the variable name and its characteristics known to the interpreter). When OPTION EXPLICIT is used (as recommended) the DIM, LOCAL or STATIC commands are the only way that a variable can be created. If this option is not used then using the DIM command is optional and if not used the variable will be created automatically when first referenced. The type of the variable (ie, string, float or integer) can be specified in one of three ways: By using a type suffix (ie, !, % or $ for float, integer or string). For example: DIM nbr%, amount!, name$ By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (ie, it does not have to be repeated). For example: DIM STRING first_name, last_name, city By using the Microsoft convention of using the keyword "AS" and the type keyword (ie, FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable. For example: DIM amount AS FLOAT, name AS STRING Floating point or integer variables will be set to zero when created and strings will be set to an empty string (ie, ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example: DIM STRING city = "Perth", house = "Brick" element and this can quickly use up memory when defining arrays of strings. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters. For example: DIM STRING s(5, 10) will declare a string array with 66 elements consuming 16,896 bytes of memory while: DIM STRING s(5, 10) LENGTH 20 Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element. If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this, string arrays created with the LENGTH keyword act exactly the same as other string arrays. This keyword can also be used with non array string variables but it will not save any memory unless the length is less than 16 when it will both save memory and improve performance. In the above example you can also use the Microsoft syntax of specifying the type after the length qualifier. For example: DIM s(5, 10) LENGTH 20 AS STRING The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter "Defining and Using Variables" for more examples of the syntax. As well as declaring simple variables the DIM command will also declare arrayed variables (ie, an indexed variable with up to five dimensions). Note that this is different from the original Colour Maximite and Micromite versions of MMBasic which supported up to eight dimensions. Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example: DIM array(10, 20) Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1. The above example specifies a two dimensional array with 11 elements

	(0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each floating point number on the Micromite eXtreme requires 8 bytes a total of 1848 bytes of memory will be allocated. Strings will default to allocating 255 bytes (ie, characters) of memory for eachArrays can also be initialised when they are declared by adding an equals symbol (=) followed by a bracketed list of values at the end of the declaration. For example: DIM INTEGER nbr(4) = (22, 44, 55, 66, 88) or DIM s$(3) = ("foo", "boo", "doo", "zoo") Note that the number of initialising values must match the number of elements in the array including the base value set by OPTION BASE. If a multi dimensioned array is initialised then the first dimension will be initialised first followed by the second, etc. Also note that the initialising values must be after the LENGTH qualifier (if used) and after the type declaration (if used).
DO <statements> LOOP	This structure will loop forever; the EXIT DO command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like GOTO or EXIT SUB (if in a subroutine).
DO WHILE expression <statements> LOOP	Loops while "expression" is true (this is equivalent to the older WHILE-WEND loop, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, not even once.
DO <statements> LOOP UNTIL expression	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is true.
EDIT	Invoke the full screen editor. See the section *Full Screen Editor* for details of how to use the editor.
ELSE	Introduces a default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF expression THEN or ELSE IF expression THEN	Introduces a secondary condition in a multiline IF statement. See the multiline IF statement for more details.
END	End the running program and return to the command prompt.
END 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.
ERASE variable [,variable]...	Deletes variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (eg, dat()) or just by specifying the variable's name (eg, dat). Use CLEAR to delete all variables at the same time (including arrays).
ERROR [error_msg$]	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXECUTE command$	This executes the Basic command "command$". Use should be limited to basic commands that execute sequentially for example the GOTO statement will not work properly Things that are tested and work OK include GOSUB, Subroutine calls, other simple statements (like PRINT and simple assignments) Multiple statements separated by : are not allowed and will error The command sets an internal watchdog before executing the requested command and if control does not return to the command, like in a GOTO statement, the timer will expire. In this case you will get the message

	"Command timeout". RUN is a special case and will cancel the timer allowing you to use the command to chain programs if required.
EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.
FILES [fspec$] [,sort]	List the files on the SD card.Lists files in any directories on the default Flash Filesystem or SD Card. 'fspec$' (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 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
FONT [#]font-number, scaling	This will set the default font for displaying text on the VGA screen. Fonts are specified as a number. For example, #2 (the # is optional) See the chapter "Basic Graphics" for details of the available fonts. 'scaling' can range from 1 to 15 and will multiply the size of the pixels making the displayed character correspondingly wider and higher. Eg, a scale of 2 will double the height and width.
FOR counter = start TO finish [STEP increment]	Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' is greater than 'finish'. The ‘increment’ can be an integer or floating point number. Note that using a floating point fractional number for 'increment' can accumulate rounding errors in 'counter' which could cause the loop to terminate early or late. 'increment' can be negative in which case 'finish' should be less than 'start' and the loop will count downwards. See also the NEXT command.
FRAMEBUFFER FRAMEBUFFER CREATE FRAMEBUFFER LAYER FRAMEBUFFER WRITE where FRAMEBUFFER CLOSE [which] FRAMEBUFFER COPY from,to FRAMEBUFFER BLIT from,to,x_from,y_from,x_to,y_	FRAMEBUFFER command for colour SPI displays. This command can be used to avoid screen artefacts when updating SPI displays with moving elements. Creates a framebuffer “F” with a RGB121 colour space and resolution to match the configured SPI colour display Creates a framebuffer “L” with a RGB121 colour space and resolution to match the configured SPI colour display Specifies the target for subsequent graphics commands. "where" can be N, F, or L where N is the actual display. Closes a framebuffer and releases the memory. The optional parameter "which" can be F or L. If omitted closes both. Does a highly optimised full screen copy of one framebuffer to another. "from" and "to" can be N, F, or L where N is the physical display. You can only copy from N on displays that support BLIT and transparent text. The firmware will automatically compress or expand the RGB resolution when copying to and from unmatched framebuffers. Of course copying from RGB565 to RGB121 loses information but for many applications (e.g. games) 16 colour levels is more than enough Moves a subsection of one framebuffer to another.

to,width,height	“x_from” and “y_from” define the coordinates of the top left of the area to be copied with "width" and "height" “x_to” and “y_to” define the coordinates of the top left of the area to receive the copy with "width" and "height" You can only copy from N on displays that support BLIT and transparent text. This command cannot be optimised as well as the full framebbuffer copy and if the area is greater than about 150x150 then a full screen copy may be preferred.

FUNCTION xxx (arg1 [,arg2, …]) [AS <type>}
  <statements>
  <statements>
  xxx = <return value>
END FUNCTION
	Defines a callable function. This is the same as adding a new function to MMBasic while it is running your program.
	'xxx' is the function name and it must meet the specifications for naming a variable. The type of the function can be specified by using a type suffix (ie, xxx$) or by specifying the type using AS <type> at the end of the functions definition. For example:
	FUNCTION xxx (arg1, arg2) AS STRING
	'arg1', 'arg2', etc are the arguments or parameters to the function (the brackets are always required, even if there are no arguments). An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using AS <type> (ie, arg1 AS STRING).
	The argument can also be another defined function or the same function if recursion is to be used (the recursion stack is limited to 50 nested calls).
	To set the return value of the function you assign the value to the function's name. For example:
	FUNCTION SQUARE(a)
	  SQUARE = a * a
	END FUNCTION
	Every definition must have one END FUNCTION statement. When this is reached the function will return its value to the expression from which it was called. The command EXIT FUNCTION can be used for an early exit.
	You use the function by using its name and arguments in a program just as you would a normal MMBasic function. For example:
	PRINT SQUARE(56.8)
	When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function.
	Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string.
	Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable and therefore may be accessed after the function has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference and must be the correct type.
	You must not jump into or out of a function using commands like GOTO, GOSUB, etc. Doing so will have undefined side effects including the possibility of ruining your day.

GUI	This is a full implementation of the GUI controls as popularised on the Micromite Plus. This is a suite of advanced graphic controls that respond to input from a mouse. These include on screen switches, buttons, indicator lights, keyboard, etc. The GUI controls have their own manual: *GUI Controls and Programming.pdf* Note: • The mouse must be connected and working for the GUI commands to work and has been set up using the OPTION MOUSE command • The number of controls allowed is defined using the OPTION MAXCTRLS n command where "n" is the maximum number that can be used to identify a control. By default this is set to zero so it must be configured before the GUI controls can be used (recommended is 100). Differences compared to the Micromite Plus implementation: • The Micromite eXtreme uses the mouse as the user interface rather than touch. • GUI BEEP not implemented. Use PLAY TONE or PULSE in the click interrupt routine if required • The TOUCH function is renamed CLICK.
GOTO target	Branches program execution to the target, which can be a line number or a label.
GUI CURSOR GUI CURSOR ON [cursorno [, x, y [,cursorcolour]]] GUI CURSOR x, y GUI CURSOR OFF GUI CURSOR HIDE GUI CURSOR SHOW GUI CURSOR COLOUR cursorcolour GUI CURSOR LOAD "fname" GUI CURSOR LINK MOUSE GUI CURSOUR UNLINK MOUSE	The GUI CURSOR command provides a mechanism for displaying and manipulating a cursor on the screen. The cursor sits above all other graphics and nothing can overwrite it. BLIT, page copy, text, sprite, box etc. can all be used and the cursor will stay in view unless deliberately hidden. The cursor is always on PAGE 0 for colours 8 and 16 and page 1 for 12-bit colour and it can be moved even if the write page is somewhere else. Cursor no can be 0 (Default: mouse type pointer) or 1 (cross), in addition the user can load his own cursor using a SPRITE look-alike file in which case this is cursor no. 2 For cursor numbers 0 and 1 the programmer can override the default white cursor by specifying the colour in the open command. Moves the cursor to x, y. Does not display the cursor if hidden but just updates the location. Turns off the cursor Hides the cursor but maintains its position Shows a hidden cursor in its stored position Changes the colour of cursor number 0 or 1. Does not impact loaded cursors where its colours are specified by the cursor designer Loads a user cursor from a file in the Maximite sprite format with a minor change. The header is now Width, Height, Xoffset, Yoffset. The two offsets determine where on the cursor the pointer is defined to be. So the mouse cursor has offsets 0,0 and the cross has offsets 7,7 These commands link and unlink mouse activity to the cursor. When linked the cursor will automatically track the mouse activity. The mouse must be opened using the CONTROLLER MOUSE OPEN command before the cursor is linked.

GUI BITMAP x, y, bits [, width] [, height] [, scale] [, c] [, bc]	Displays the bits in a bitmap on the screen starting at 'x' and 'y' 'height' and 'width' are the dimensions of the bitmap as displayed on the screen and default to 8x8. 'scale' is optional and defaults to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. The bitmap can be an integer or a string variable or constant and is drawn using the first byte as the first bits of the top line (bit 7 first, then bit 6, etc) followed by the next byte, etc. When the top line has been filled the next line of the displayed bitmap will start with the next bit in the integer or string. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
I2C	The I2C commands will send and receive data over an I^{2}C channel. I2C (no suffix) refers to channel 1 while commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax. Also see *Appendix B*.
I2C OPEN speed, timeout	Enables the I^{2}C 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 [,sendata ....]	Send data to the I^{2}C slave device. ‘addr’ is the slave’s I^{2}C 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) ‘sendlen’ is the number of bytes to send. ‘senddata’ is the data to be sent - this can be specified in various ways (all values sent will be between 0 and 255). Notes: • The data can be supplied as individual bytes on the command line. Example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25 • The data can be in a one dimensional array specified with empty brackets (ie, no dimensions). ‘sendlen’ bytes of the array will be sent starting with the first element. Example: I2C WRITE &H6F, 0, 3, ARRAY() The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 0, 3, STRING$
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I^{2}C slave device. ‘addr’ is the slave’s I^{2}C 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) ‘rcvlen’ is the number of bytes to receive. ‘rcvbuf’ is the variable or array used to save the received data - this can be: • A string variable. Bytes will be stored as sequential characters. • A one dimensional array of numbers specified with empty brackets. Received bytes will be stored in sequential elements of the array starting with the first. Example: I2C READ &H6F, 0, 3, ARRAY() A normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE	Disables the master I^{2}C module and returns the I/O pins to a "not configured" state. They can then be configured using SETPIN. This command will also send a stop if the bus is still held.

IF expr THEN stmt [: stmt]
or
IF expr THEN stmt ELSE stmt
	Evaluates the expression ‘expr' and performs the statement following the THEN keyword if it is true or skips to the next line if false. If there are more statements on the line (separated by colons (:) they will also be executed if true or skipped if false.
	The ELSE keyword is optional and if present only one true statement is allowed following the THEN keyword. If 'expr' is resolved to be false the single statement following the ELSE keyword will be executed.
	The ‘THEN statement’ construct can be also replaced with:
	GOTO linenumber | label’.
	This type of IF statement is all on one line.

IF expression THEN
  <statements>
[ELSEIF expression THEN
  <statements>]
[ELSE
  <statements>]
ENDIF
	Multiline IF statement with optional ELSE and ELSEIF cases and ending with ENDIF. Each component is on a separate line.
	Evaluates 'expression' and performs the statement(s) following THEN if the expression is true or optionally the statement(s) following the ELSE statement if false. The ELSEIF statement (if present) is executed if the previous condition is false and it starts a new IF chain with further ELSE and/or ELSEIF statements as required.
	One ENDIF is used to terminate the multiline IF.

INC var [,increment]	Increments the variable “var” by either 1 or, if specified, the value in increment. “increment” can have a negative. This is functionally the same as var = var + increment but is processed much faster
INPUT ["prompt$";] var1 [,var2 [, var3 [, etc]]]	Will take a list of values separated by commas (,) entered at the console and will assign them to a sequential list of variables. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 The list of variables can be a mix of float, integer or string variables. The values entered at the console must correspond to the type of variable. If a single value is entered a comma is not required (however that value cannot contain a comma). ‘prompt$’ is a string constant (not a variable or expression) and if specified it will be printed first. Normally the prompt is terminated with a semicolon (;) and in that case a question mark will be printed following the prompt. If the prompt is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #nbr, list of variables	Same as the normal INPUT command except that the input is read from a file previously opened for INPUT as ‘#fnbr’ or a serial port previously opened for INPUT as ‘nbr’. See the OPEN command. #0 can be used which refers to the console.
IR dev, key , int or IR CLOSE	Decodes NEC or Sony infrared remote control signals. An IR Receiver Module is used to sense the IR light and demodulate the signal. It should be connected to the IR pin (see the pinout tables). This command will automatically set that pin to an input. 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). '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 and return the I/O pin to a not configured state. 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 "Special Hardware Devices" 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.
KILL file$	Deletes the file specified by ‘file$’. Any extension must be specified. Bulk erase is triggered if fname$ contains a '*' or a '?' character If the

	optional 'all' parameter is used then you will be prompted for a single confirmation. If 'all' is not specified you will be prompted on a file-by-file basis
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command. For example: Var = 56
LINE x1, y1, x2, y2 [, LW [, C]]	Draws a line starting at the coordinates 'x1' and 'y1' and ending at 'x2' and 'y2'. 'LW' is the line's width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal. 'C' is an integer representing the colour and defaults to the current foreground colour. All parameters can now be expressed as arrays and the software will plot the number of lines as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', and 'y2' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw' and 'c' can be either arrays or single variables/constants.
LINE INPUT [prompt$,] string-variable$	Reads an entire line from the console input into 'string-variable$'. 'prompt$' is a string constant (not a variable or expression) and if specified it will be printed first. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of 'prompt$'.
LINE INPUT #nbr, string-variable$	Same as the LINE INPUT command except that the input is read from a file previously opened for INPUT as '#fnbr' or a serial communications port previously opened for INPUT as 'nbr'. See the OPEN command. #0 can be used which refers to the console. The # character is required.
LIST [file$] or LIST ALL [file$]	List a program on the serial console. LIST on its own will list the program with a pause at every screen full. LIST ALL will list the program without pauses. This is useful if you wish to transfer the program in the Maximite to a terminal emulator on a PC that has the ability to capture its input stream to a file. In most cases the filename 'file$' is required however if EDIT file$ or RUN file$ has been used previously the "current program name" will have been set and in that case LIST will default to using that filename.
LIST FILES [fspec$] [, sort]	Lists files in the current directory on the SD card. 'fspec$' (if specified) can contain search wildcards. Question marks (?) will match any character and an asterisk (*) will match any number of characters. If omitted, all files will be listed.For example: * Find all entries *.TXT Find all entries with an extension of TXT E*.* Find all entries starting with E X?X.* Find all three letter file names starting and ending with X 'sort' specifies the sort order as follows: size by ascending size time by ascending time/date name by file name (default if not specified) type by file extension
LIST COMMANDS or LIST FUNCTIONS	Lists all valid commands or functions
LOAD DATA fname$, address	Loads the binary contents of file "fname$" and stores it in CMM2 memory starting at "address". See also SAVE DATA
LOAD FONT file$	Load the font contained in 'file$' on the SD card and install it as font #8. See the section *Basic Graphics* earlier in this manual. You can convert font files designed for the original Colour Maximite using FontTweak from: https://www.c-com.com.au/MMedit.htm

LOAD BMP file$ [, x, y] or LOAD GIF [file$ [, x, y]] or LOAD JPG file$ [, x, y] or LOAD PNG file$ [, x, y] [, transparency_cut_off]	Load an image from the SD card and display it on the VGA monitor. "file$' is the name of the file and 'x' and 'y' are the screen coordinates for the top left hand corner of the image. If the coordinates are not specified the image will be drawn at the top left hand position on the screen. If an extension is not specified the appropriate extension will be added to the file name. All types of the BMP format are supported including black and white and true colour 24-bit images. The image can be of any size and pixels off the screen will be ignored. GIFs can be a single image or animated. If it is animated it will start playing in the background (ie, program execution will continue while it is playing). If an animated GIF is already running it will be replaced by the new one. If LOAD GIF is used without any parameters it will stop the currently playing animated GIF. JPG images cannot use progressive encoding and are limited to being completely within the screen resolution (ie, pixels cannot extend beyond the screen limits). MODE 2,16 is the optimum for displaying JPG images as the hardware decoder can write RGB565 pixels directly into the frame buffer. For all other modes, the firmware has to adjust the image by duplicating lines (mode 3) and/or converting from RGB565 to RGB332. PNG files must be in the RGB888 or ARGB8888 format and can be sized up to the current resolution of the screen. If the x & y start coordinates are specified pixels off the screen will be ignored. If the transparency level is specified and none-zero then: • If a PNG file is in ARGB8888 format the 'transparency_cut_off' parameter is used to determine whether the pixel should be solid or missing/transparent. Valid values are 1 to 15, no default. MMBasic compares the 4 most significant bits of the transparency data in the file with the cut off value and assigns a transparency of 0 or 15 depending on the comparison. This allows RGB(0,0,0) to be a valid solid colour. • If the file is in RGB888 format then an RGB level of 0,0,0 is used to determine transparency as there is no other information to use. If the 'transparency_cut_off' level is not specified all pixels will be loaded as solid colours as with any other image load.
LOCAL variable [, variables] See DIM for the full syntax.	Defines a list of variable names as local to the subroutine or function. This command uses exactly the same syntax as DIM and will create variables that will only be visible within the subroutine or function. They will be automatically discarded when the subroutine or function exits.
LONGSTRING	The LONGSTRING commands allow for the manipulation of strings longer than the normal MMBasic limit of 255 characters. Variables for holding long strings must be defined as single dimensioned integer arrays with the number of elements set to the number of characters required for the maximum string length divided by eight. The reason for dividing by eight is that each integer in an MMBasic array occupies eight bytes. Note that the long string routines do not check for overflow in the length of the strings. If an attempt is made to create a string longer than a long string variable's size the outcome will be undefined.
LONGSTRING APPEND array%(), string$	Append a normal MMBasic string to a long string variable. array%() is a long string variable while string$ is a normal MMBasic string expression.
LONGSTRING CLEAR array %()	Will clear the long string variable array%(). ie, it will be set to an empty string.
LONGSTRING COPY dest% (), src%()	Copy one long string to another. dest%() is the destination variable and src%() is the source variable. Whatever was in dest%() will be overwritten.
LONGSTRING CONCAT dest %(), src%()	Concatenate one long string to another. dest%() is the destination variable and src%() is the source variable. src%() will the added to the end of dest%() (the destination will not be overwritten).
LONGSTRING LCASE array %()	Will convert any uppercase characters in array%() to lowercase. array%() must be long string variable.
LONGSTRING LEFT dest%(), src%(), nbr	Will copy the left hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the beginning of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING LOAD array%	Will copy 'nbr' characters from string$ to the long string variable array%()

(), nbr, string$	overwriting whatever was in array%().
LONGSTRING MID dest%(), src%(), start, nbr	Will copy 'nbr' characters from src%() to dest%() starting at character position 'start' overwriting whatever was in dest%(). ie, copy from the middle of src%(). 'nbr' is optional and if omitted the characters from 'start' to the end of the string will be copied src%() and dest%() must be long string variables. 'start' and 'nbr' must be an integer constants or expressions.
LONGSTRING PRINT [#n,] src%()	Prints the longstring stored in ‘src%()’ to the file or COM port opened as ‘#n’. If ‘#n’ is not specified the output will be sent to the console.
LONGSTRING REPLACE array%() , string$, start	Will substitute characters in the normal MMBasic string string$ into an existing long string array%() starting at position ‘start’ in the long string.
LONGSTRING RESIZE array %(), nbr LONGSTRING RIGHT dest% (), src%(), nbr	Sets the size of the longstring to nbr. This overrides the size set by other longstring commands so should be used with caution. Typical use would be in using a longstring as a byte array. Will copy the right hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the end of src%(). src %() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING SETBYTE array%(), nbr, data LONGSTRING TRIM array% (), nbr	sets byte nbr to the value “data”, nbr respects OPTION BASE Will trim ‘nbr’ characters from the left of a long string. array%() must be a long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING UCASE array %()	Will convert any lowercase characters in array%() to uppercase. array%() must be long string variable.
LOOP [UNTIL expression]	Terminates a program loop: see DO.

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.
Simple array arithmetic	
MATH SET nbr, array()	Sets all elements in array() to the value nbr. Note this is the fastest way of clearing an array by setting it to zero.
MATH SCALE in(), scale ,out()	This scales the matrix in() by the scalar scale and puts the answer in out(). Works for arrays of any dimensionality of both integer and float and can convert between. Setting b to 1 is optimised and is the fastest way of copying an entire array.
MATH ADD in(), num ,out()	This adds the value 'num' to every element of the matrix in() and puts the answer in out(). Works for arrays of any dimensionality of both integer and float and can convert between. Setting num to 0 is optimised and is a fast way of copying an entire array. in() and out() can be the same array.
MATH INTERPOLATE in1(), in2(), ratio, out()	This command implements the following equation on every array element: out = (in2 - in1) * ratio + in1 Arrays can have any number of dimensions and must be distinct and have the same number of total elements. The command works with bot integer and floating point arrays in any mixture
MATH SLICE sourcearray(), [d1] [,d2] [,d3] [,d4] [,d5] , destinationarray()	This command copies a specified set of values from a multi-dimensional array into a single dimensional array. It is much faster than using a FOR loop. The slice is specified by giving a value for all but one of the source array indicies and there should be as many indicies in the command, including the blank one, as there are dimensions in the source array e.g. OPTION BASE 1 DIM a(3,4,5) DIM b(4) MATH SLICE a(), 2, , 3, b() Will copy the elements 2,1,3 and 2,2,3 and 2,3,3 and 2,4,3 into array b()
MATH INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5] , sourcearray()	This is the opposite of MATH SLICE, has a very similar syntax, and allows you, for example, to substitute a single vector into an array of vectors with a single instruction e.g. OPTION BASE 1 DIM targetarray(3,4,5) DIM sourcearray(4)=(1,2,3,4) MATH INSERT targetarray(), 2, , 3, sourcearray() Will set elements 2,1,3 = 1 and 2,2,3 = 2 and 2,3,3 = 3 and 2,4,3 = 4

Matrix arithmetic MATH M_INVERSE array!(), inversearray!() MATH M_PRINT array() MATH M_TRANSPOSE in(), out() MATH M_MULT in1(), in2(), out()	 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. Quick mechanism to print a 2D matrix one row per line. 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) Multiply the arrays in1() and in2() and put the answer in out()c. All arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in1(m,n) in2(p,m) ,out(p,n)
Vector arithmetic MATH V_PRINT array() MATH V_NORMALISE inV(), outV() MATH V_MULT matrix(), inV(), outV() MATH V_CROSS inV1(), inV2(), outV()	 Quick mechanism to print a small array on a single line Converts a vector inV() to unit scale and puts the answer in outV() (sqr(x*x + y*y +.......)=1 There is no limit on number of elements in the vector 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()
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 ector and rotates clockwise, pitch rotates the top away from the camera and roll rotates around the z-axis clockwise. The yaw, pitch and roll angles default to radians but respect the setting of OPTION ANGLE 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 FFT signalarray!(), FFTarray!()	Performs a fast fourier transform of the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "FFTarray" must be floating point and have dimension 2*N where N is the same as the signal array (e.g. f(1,1023) assuming OPTION BASE is zero) The command will return the FFT as complex numbers with the real part in f(0,n) and the imaginary part in f(1,n)
MATH FFT INVERSE FFTarray!(), signalarray!()	Performs an inverse fast fourier transform of the data in “FFTarray!”. "FFTarray" must be floating point and have dimension 2*N where N must be a power of 2 (e.g. f(1,1023) assuming OPTION BASE is zero) with the real part in f(0,n) and the imaginary part in f(1,n). "signalarray" must be floating point and the single dimension must be the same as the FFT array.

	The command will return the real part of the inverse transform in "signalarray".
MATH FFT MAGNITUDE signalarray!(),magnitudearray!()	Generates magnitudes for frequencies for the data in "signalarray!" "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "magnitudearray" must be floating point and the size must be the same as the signal array The command will return the magnitude of the signal at various frequencies according to the formula: frequency at array position N = N * sample_frequency / number_of_samples
MATH FFT PHASE signalarray!(), phasearray!() MATH SENSORFUSION type ax, ay, az, gx, gy, gz, mx, my, mz, pitch, roll, yaw [,p1] [,p2]	Generates phases for frequencies for the data in "signalarray!". "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "phasearray" must be floating point and the size must be the same as the signal array The command will return the phase angle of the signal at various frequencies according to the formula above. 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. Mx, my, and mz are the magnetic fields in the three directions and should be specified in nano-Tesla (nT) 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. The SENSORFUSION routine will automatically measure the time between consecutive calls and will use this in its internal calculations. The Madwick 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
MEMORY	List the amount of memory currently in use. For example: Flash: 39K (6%) Program (1450 lines) 527K (94%) Free RAM: 0K (0%) 0 Variables 1K (0%) General 24799K (100%) Free Notes: • General memory is used by serial I/O buffers, etc. • Memory usage is rounded to the nearest 1K byte.
MEMORY SET address, byte, numberofbytes MEMORY SET BYTE address, byte, numberofbytes MEMORY SET SHORT address, short, numberofshorts	This command will set a region of memory to a value. BYTE = One byte per memory address. SHORT = Two bytes per memory address. WORD = Four bytes per memory address. INTEGER = Four bytes per memory address. FLOAT = Four bytes per memory address. 'increment' is optional and controls the increment of the 'address' pointer as the operation is executed. For example, if increment=3 then only every third element of the target is set. The default is 1.

MEMORY SET WORD address, word, numberofwords MEMORY SET INTEGER address, integervalue ,numberofintegers [,increment] MEMORY SET FLOAT address, floatingvalue ,numberofloats [,increment]	
MEMORY COPY sourceaddress, destinationaddres, numberofbytes MEMORY COPY INTEGER sourceaddress, destinationaddress, numberofintegers [,sourceincrement] [,destinationincrement] MEMORY COPY FLOAT sourceaddress, destinationaddress, numberoffloats [,sourceincrement] [,destinationincrement]	This command will copy one region of memory to another. COPY INTEGER and FLOAT will copy four bytes per operation. 'sourceincrement' is optional and controls the increment of the 'sourceaddress' pointer as the operation is executed. For example, if sourceincrement=3 then only every third element of the source will be copied. The default is 1. 'destinationincrement' is similar and operates on the 'destinationaddress' pointer.
MID$(str$, start [, num]) = str2$	The characters in 'str$', beginning at position 'start', are replaced by the characters in 'str2$'. The optional 'num' refers to the number of characters from 'str2' that are used in the replacement. If 'num' is omitted, all of 'str2' is used.
MKDIR dir$	Make, or create, the directory 'dir$' on the SD card.
MMDEBUG MMDEBUG BREAK	By default the MMDEBUG command acts exactly like the PRINT command so you can use it liberally throughout your program during development and know that by simply setting #MMDEBUG OFF or removing any #MMDEBUG directives, your program will run normally without any impact on performance. The second form of the MMDEBUG command is MMDEBUG BREAK. in this case you get a command promptDEBUG> Here you can execute any normal MMBASIC commands as though you were at the normal command prompt allowing you to change any variables, print them, or execute any other user subroutines or built in commands. These should be commands that make sense at the command line (e.g. do not use GOTO). To exit this mode use the command CONTINUE
NEW	Deletes the program in program memory, clears all variables including saved variables and resets the interpreter (ie, closes files, serial ports, etc).
NEXT [counter-variable] [, counter-variable], etc	NEXT comes at the end of a FOR-NEXT loop; see FOR. The 'counter-variable' specifies exactly which loop is being operated on. If no 'counter-variable' is specified the NEXT will default to the innermost loop. It is also possible to specify multiple variables as in: NEXT x, y, z
ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR	This controls the action taken if an error occurs while running a program and applies to all errors discovered by MMBasic including syntax errors, wrong data, missing hardware, SD Card access, etc. ON ERROR ABORT will cause MMBasic to display an error message, abort the program and return to the command prompt. This is the normal behaviour and is the default when a program starts running. ON ERROR IGNORE will cause any error to be ignored. ON ERROR SKIP will ignore an error in a number of commands (specified by the number 'nn') executed following this command. 'nn' is

	optional, the default if not specified is one. After the number of commands has completed (with an error or not) the behaviour of MMBasic will revert to ON ERROR ABORT. If an error occurs and is ignored/skipped the read only variable MM.ERRNO will be set to non zero and MM.ERRMSG$ will be set to the error message that would normally be generated. These are reset to zero and an empty string by ON ERROR CLEAR. They are also cleared when the program is run and when ON ERROR IGNORE and ON ERROR SKIP are used. ON ERROR IGNORE can make it very difficult to debug a program so it is strongly recommended that only ON ERROR SKIP be used.
ON KEY target or ON KEY ASCIIcode, target	The first version of the command sets an interrupt which will call 'target' user defined subroutine whenever there is one or more characters waiting in the serial console input buffer. Note that all characters waiting in the input buffer should be read in the interrupt subroutine otherwise another interrupt will be automatically generated as soon as the program returns from the interrupt. The second version allows you to associate an interrupt routine with a specific key press. This operates at a low level for both the USB keyboard and a serial console and if activated the key does not get put into the input buffer but merely triggers the interrupt. It uses a separate interrupt from the simple ON KEY command so can be used at the same time if required. In both variants, to disable the interrupt use numeric zero for the target, i.e.: ON KEY 0. or ON KEY ASCIIcode, 0
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 number of data items must agree with the length parameter. See also *Appendix C*.
OPEN fname$ FOR mode AS [#]fnbr	Opens a file for reading or writing. 'fname' is the filename with an optional extension separated by a dot (.). Long file names with upper and lower case characters are supported. A directory path can be specified with the forward or backwards slash as a directory separator. The parent of the current directory can be specified by using a directory name of .. (two dots) and the current directory with . (a single dot). For example OPEN "../dir1/dir2/filename.txt" FOR INPUT AS #1 'mode' is INPUT, OUTPUT, APPEND or RANDOM. INPUT will open the file for reading and throw an error if the file does not exist. OUTPUT will open the file for writing and will automatically overwrite any existing file with the same name. APPEND will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the APPEND mode will act the same as the OUTPUT mode (i.e. the file is created then opened for writing). RANDOM will open the file for both read and write and will allow random access using the SEEK command. When opened the read/write pointer is positioned at the end of the file. 'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT, WRITE and CLOSE commands as well as the EOF() and INPUT$() functions all use 'fnbr' to identify the file being operated on. See also ON ERROR and MM.ERRNO for error handling.

OPEN comspec$ AS [#]fnbr	Will open a serial communications port for reading and writing. Two ports are available (COM1: and COM2:) and both can be open simultaneously. If OPTION CONSOLE SCREEN is used then the console serial port is available as COM3:. Using 'fnbr' the port can be written to and read from using any command or function that uses a file number. 'comspec$' is the communication specification and is a string (it can be a string variable) specifying the serial port to be opened and optional parameters. The default is 9600 baud, 8 data bits, no parity and one stop bit. It has the form "COMn: baud, buf, int, int-trigger, (DEN or DEP), 7BIT, (ODD or EVEN), INV, OC, S2" Where: • 'n' is the serial port number for either COM1:, COM2 or COM3:.:. • 'baud' is the baud rate. This can be any value between 1200 (the minimum) and 1000000 (1MHz). Default is 9600. • 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes. • 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt. • 'int-trigger' sets the trigger condition for calling the interrupt subroutine. If it is a normal number the interrupt subroutine will be called when this number of characters has arrived in the receive queue. All parameters except the serial port name (COMn:) are optional. If any one parameter is left out then all the following parameters must also be left out and the defaults will be used. These options can be added to the end of 'comspec$' • 'INV' specifies that the transmit and receive polarity is inverted. • 'OC' will force the transmit pin (and DE on COM1:) 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. • '7BIT' will specify that 7 bit transmit and receive is to be used. • 'ODD' will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'EVEN' will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'DEP' will enable RS485 mode with positive output on COM1-DE • 'DEN' will enable RS485 mode with negative output on COM1-DE
OPEN comspec$ AS GPS [,timezone_offset] [,monitor]	Will open a serial communications port for reading from a GPS receiver. See the GPS function for details. The sentences interpreted are GPRMC, GNRMC, GPCGA and GNCGA. 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 can be a any number between -12 and 14 allowing the time to be set correctly even for the Chatham Islands in New Zealand (UTC +12:45). If the monitor parameter is set to 1 then all GPS input is directed to the console. This can be stopped by closing the GPS channel.
OPTION	See the section *Options* earlier in this manual.
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.
PIXEL x, y [,c]	Set a pixel on an attached VGA monitor 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 arrays or a

	single variable or constant. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
PIXEL FILL x, y, c	Implements a flood fill by reading the colour of the pixel at coordinates x,y and replacing it and the entire area of connected pixels having the same color with the new colour “c”
PLAY EFFECT file$ [,interrupt]	This will play the WAV file ' file$' at the same time as a MOD file is playing. If a previous EFFECT file is playing this command will immediately terminate it and commence playing the new file. The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. Note: wav files played using PLAY EFFECT during mod file playback must have the same sample rate as the modfile output. Files can be mono or stereo.
PLAY TONE left , right [, dur [, interrupt]]	Generates two separate sine waves on the sound output left and right channels. The tone plays in the background (the program will continue running after this command). 'left' and 'right' are the frequencies in Hz to use for the left and right channels. 'dur' specifies the number of milliseconds that the tone will sound for. MMBasic will round the time to the next nearest complete waveform of the first frequency specified so that the tone will always finish with the DC level in the middle and no discontinuity. If the duration is not specified the tone will continue until explicitly stopped or the program terminates. 'interrupt' is optional and is an interrupt subroutine to call when the tone has completed. The frequency can be from 1Hz to 20KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command.
PLAY WAV file$ [, interrupt] or PLAY FLAC file$ [, interrupt] or PLAY MP3 file$ [, interrupt]	Play an audio file on the audio (DAC) output. 'file$' is the file to play (the appropriate extension will be appended if missing). The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. For WAV files MMBasic will automatically compensate for the frequency, number of bits and number of channels of the WAV file. For FLAC files the supported frequencies are: 44100Hz 16-bit(CD quality) and 24-bit 48000Hz 16-bit and 24-bit 88200Hz 16-bit and 24-bit 96000Hz 24-bit Maximums for FLAC and WAV file playback are 96KHz 24-bit. Both will auto-configure to the file provided. As an indication, 96KHz 24-bit FLAC uses just over 50% of the CPU's resources. If ‘file$’ is a directory then the firmware will list all the files of the relevant type in that directory and start playing them one-by-one. To play files in the current directory use an empty string (ie, ""). Each file listed will play in turn and the optional interrupt will fire when all files have been played. The filenames are stored with full path so you can use CHDIR while tracks are playing without causing problems. All files in the directory are listed if the command is executed at the command prompt but the listing is suppressed in a program
PLAY MODFILE file$ [,samplerate]	Will play a MOD file on the DAC outputs. 'file$' is the MOD file to play (the extension of .mod will be appended if missing). The MOD file is played in the background and will run continuously until PLAY STOP is called. The MOD encoder supports 32 channels, 16-bit resolution (but the DACs are only 12 bit), 32 samples, no fixed maximum size. The optional parameter samplerate specifies the number of samples per second generated by the modfile engine. The default is 44100. Processor overhead is reduced by decreasing this. Valid values are 8000, 16000, 22050, 44100, 48000 Note: wav files played using PLAY EFFECT during modfile playback must have the same sample rate as the modfile output.
PLAY MODSAMPLE sampleno, channelno	Plays one of the samples in the MOD file concurrently with the main MOD file playback. This allows sound effects to be incorporated in the MOD

[,volume] [,samplerate]	file. “sampleno” can be in the range 1 to 32. Up to 4 samples can be played simultaneously on independent channels using the specified “channelno” which must be in the range 1 to 4. The optional “volume” should be set in the range 0 to 64 (default 64). The optional “samplerate” specifies the update rate for the sample. The default is 16000. Changing this will change the pitch of the sample and the duration of playback and it should be set to the sample's original rate for playback as recorded.
PLAY PAUSE PLAY RESUME PLAY STOP	PLAY PAUSE will temporarily halt the currently playing file or tone. PLAY RESUME will resume playing a sound that was paused. PLAY STOP will terminate the playing of the file or tone. When the program terminates for whatever reason the sound output will also be automatically stopped.
PLAY NEXT PLAY PREVIOUS	When playing a sequence of audio tracks (by using PLAY MP3 on a directory holding multiple MP3 files) these commands can be used to skip forward or back a file. The commands PLAY PAUSE, RESUME, VOLUME can also be used.
PLAY TTS [PHONETIC] "text" [,speed] [,pitch] [,mouth] [,throat] [, interrupt]	Outputs text as speech on the DAC outputs. See http://www.retrobits.net/atari/sam.shtml for details of parameter usage. The command is non-blocking and the speech is played in the background. 'interrupt' is optional and is the name of a subroutine which will be called when the speech has finished playing.
PLAY VOLUME left, right	Will adjust the volume of the audio output. 'left' and 'right' are the levels to use for the left and right channels and can be between 0 and 100 with 100 being the maximum volume. There is a linear relationship between the specified level and the output. The volume defaults to maximum when a program is run.
POKE BYTE addr%, byte or 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 POKE VARTBL, offset, byte	Will set a byte or a word within the CPU’s virtual memory space. POKE BYTE will set the byte (ie, 8 bits) at the memory location 'addr%' to 'byte'. 'addr%' should be an integer. POKE SHORT will set the short integer (ie, 16 bits) at the memory location 'addr%' to 'word%'. 'addr%' and short%' should be integers. POKE WORD will set the word (ie, 32 bits) at the memory location 'addr %' to 'word%'. 'addr%' and 'word%' should be integers. POKE INTEGER will set the MMBasic integer (ie, 64 bits) at the memory location 'addr%' to int%'. 'addr%' and int%' should be integers. POKE FLOAT will set the word (ie, 64 bits) at the memory location 'addr %' to 'float!'. 'addr%' should be an integer and 'float!' a floating point number. POKE VAR 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 will set a byte in MMBasic's variable table. 'offset' is the ±offset from the start of the variable table. Note that a comma is required after the keyword VARTBL.
POLYGON n, xarray%(), yarray%() [, bordercolour] [, fillcolour] 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. e.g DIM n(1)=(3,3) would define that 2 polygons are to be drawn with three vertices each. The size of the n array determines the number of polygons that will be drawn unless an element is found with the value zero in which case the firmware only processes polygons up to that point. The x,y-coordinate pairs for all the polygons are stored in xarray%() and yarray%(). The xarray%() and yarray%() parameters must have at least as many elements as the total of the values in the n array. Each polygon can be closed with the first and last elements the same. If

	the last element is not the same as the first the firmware will automatically create an additional x,y-coordinate pair to complete the polygon. If fill colour is omitted then just the polygon outlines are drawn. The colour parameters can be a single value in which case all polygons are drawn in the same colour or they can be arrays with the same cardinality as n. In this case each polygon drawn can have a different colour of both border and/or fill. For example, this will draw 3 triangles in yellow, green and red: DIM c%(2)=(3,3,3) DIM x%(8)=(100,50,150,100,50,150,100,50,150) DIM y%(8)=(50,100,100,150,200,200,250,300,300) DIM fc%(2)=(rgb(yellow),rgb(green),rgb(red)) POLYGON c%(),x%(),y%(),fc%(),fc%()
PORT(start, nbr [,start, nbr]…) = value	Set a number of I/O pins simultaneously (ie, 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. For example; PORT(15, 4, 23, 4) = &B10000011 Will set eight I/O pins. Pins 15 and 16 will be set high while 17, 18, 23, 24 and 25 will be set to a low and finally 26 will be set high. This command can be used to conveniently communicate with parallel devices like LCD displays. Any number of I/O pins (and therefore bits) can be used from 1 to the number of I/O pins on the chip. See the PORT function to simultaneously read from a number of pins.
PRINT expression [[,;]expression] … etc	Outputs text to the console (either the VGA screen or the serial or both if they are available). Multiple expressions can be used and must be separated by either a: • Comma (,) which will output the tab character • Semicolon (;) which will not output anything (it is just used to separate expressions). • Nothing or a space which will act the same as a semicolon. A semicolon (;) at the end of the expression list will suppress the automatic output of a carriage return/ newline at the end of a print statement. When printed, a number is preceded with a space if positive or a minus (-) if negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large floating point numbers (greater than six digits) are printed in scientific number format. The function TAB() can be used to space to a certain column and the string functions can be used to justify or otherwise format strings.
PRINT #nbr, expression [[,;]expression] … etc	Same as the normal PRINT command except that the output is directed to a file previously opened for OUTPUT or APPEND as '#fnbr' or to a serial communications port previously opened as 'nbr'. See the OPEN command. #0 can be used which refers to the console.
PRINT #GPS, string$	Outputs a NMEA string to an opened GPS device. The string must start with a $ character and end with a * character. The checksum is calculated automatically by the firmware and is appended to the string together with the carriage return and line feed 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. Example: PRINT @(150, 45) "Hello World" The @ function can be used anywhere in a print command. Example: PRINT @(150, 45) "Hello" @(150, 55) "World" The @(x,y) function can be used to position the cursor anywhere on or off the screen. For example, PRINT @(-10, 0) "Hello" will only show "llo" as the first two characters could not be shown because they were off the screen. The @(x,y) function will automatically suppress the automatic line wrap normally performed when the cursor goes beyond the right screen margin. If 'm' is specified the mode of the video operation will be as follows: m = 0 Normal text (white letters, black background) m = 1 The background will not be drawn (ie, transparent) m = 2 The video will be inverted (black letters, white background) m = 5 Current pixels will be inverted (transparent background)
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 ± 1 µs. • For a pulse of 3 ms or more the accuracy is ± 0.5 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 or PWM 1, freq, 1A, 1B or PWM 1, freq, 1A, 1B, 1C or PWM 2, freq, 2A or PWM 2, freq, 2A, 2B 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 diagrams on pages 6 and 7 (they are also used for the SERVO command). Controller 1 can have one, two or three outputs while controller 2 can have one or two outputs. 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 . 1A, 1B and 1C are the duty cycle for each of the controller 1 outputs while 2A and 2B are the duty cycle 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 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 Minimum frequency is 1Hz, maximum is 24MHz. Duty cycle and frequency accuracy will depend on frequency. The frequency can be any value of 240,000,000/n. 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 stoping 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.
RBOX x, y, w, h [, r] [,c] [,fill]	Draws a box with rounded corners on the VGA monitor starting at 'x' and 'y' which is 'w' pixels wide and 'h' pixels high. 'r' is the radius of the corners of the box. It defaults to 10. 'c' specifies the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled.

	All parameters can now be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'r', 'c', and 'fill' can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
READ variable[, variable]...	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. Arrays can be used as variables (specified with empty brackets, eg, a()) and in that case the size of the array is used to determine how many elements are to be read. If the array is multidimensional then the leftmost dimension will be the fastest moving. See also DATA and RESTORE.
REM string	REM allows remarks to be included in a program. Note the Microsoft style use of the single quotation mark to denote remarks is also supported and is preferred.
RENAME old$ AS new$	Rename a file or a directory from ‘old$’ to ‘new$’. Both are strings. A directory path can be used in both 'old$' and 'new$'. If the paths differ the file specified in 'old$' will be moved to the path specified in 'new$' with the file name as specified.
RESTORE [line]	Resets the line and position counters for the READ statement. If ‘line’ is specified the counters will be reset to the beginning of the specified line. ‘line’ can be a line number or label. A variable can also be used as the parameter. In that case a numerical variable should be used for a line number and a string variable for a label If ‘line’ is not specified the counters will be reset to the start of the program.
RMDIR dir$	Remove, or delete, the directory ‘dir$’ on the SD card.
RUN file$ or RUN file$, cmdline	Run the program 'file$' held on the SD card. Note that 'file$' must be a string constant (ie, "MYPROG.BAS") including the quotes required around a string constant. It cannot be a variable or expression. If 'cmdline' is specified it will be available to the running program as the string returned by MM.CMDLINE$. 'cmdline' is not processed by MMBasic so it can contain numbers, commas, quoted strings, etc. It is the responsibility of the running program to decode this string of characters. 'file$' can be omitted and in that case MMBasic will run the "current program name" which is the file last used by RUN, EDIT or AUTOSAVE.
SAVE DATA fname$, address, size	Saves "size" bytes to file "fname$" starting from "address". This allows areas of the Micromite eXtreme memory to be saved as binary files. See also LOAD DATA
SAVE IMAGE file$ [,x, y, w, h]	Save the current image on the VGA screen as a 24-bit BMP file. 'file$' is the name of the file. If an extension is not specified “.BMP” will be added to the file name. ‘x’, ‘y’, ‘w’ and ‘h’ are optional and are the coordinates (x and y are the top left coordinate) and dimensions (width and height) of the area to be saved. If not specified the whole screen will be saved.
SEEK [#]fnbr, pos	Will position the read/write pointer in a file that has been opened on the SD card for RANDOM access to the 'pos' byte. The first byte in a file is numbered one so SEEK #5,1 will position the read/write pointer to the start of the file.

SELECT CASE value
  CASE testexp [[, testexp] …]
      <statements>
      <statements>
  CASE ELSE
      <statements>
      <statements>
END SELECT
	Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression. 'testexp' is the value that 'exp' is to be compared against. It can be:
	• A single expression (ie, 34, "string" or PIN(4)*5) to which it may equal
	• A range of values in the form of two single expressions separated by the keyword "TO" (ie, 5 TO 9 or "aa" TO "cc")
	• A comparison starting with the keyword "IS" (which is optional). For example: IS > 5, IS <= 10.
	When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true.
	If 'value' cannot be matched with a 'testexp' it will be automatically

	matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any <statements> and continue with the code following the END SELECT. When a match is made the <statements> following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT. An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT.

SELECT CASE ...examples	SELECT CASE nbr%
	  CASE 4, 9, 22, 33 TO 88
	      statements
	  CASE IS < 4, IS > 88, 5 TO 8
	      statements
	  CASE ELSE
	      statements
	END SELECT
	Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT…CASE statements can be nested inside the CASE statements of other SELECT…CASE statements.

SERVO 1 [, freq], 1A or SERVO 1 [, freq], 1A, 1B or SERVO 1 [, freq], 1A, 1B, 1C or SERVO 2 [, freq], 2A or SERVO 2 [, freq], 2A, 2B or SERVO channel, STOP	Generate a constant stream of positive going pulses for driving a servo. The Maximite 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 external I/O diagram (the two commands are very similar). '1' or '2' is the controller number. 'freq' is the output frequency (between 20Hz and 1000 Hz) and is optional. If not specified it will default to 50 Hz 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 (ie, 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. 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 stoping 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.
SETPIN pin, cfg [, option]	Will configure an external I/O pin. 'pin' is the I/O pin to configure, '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 AIN Analog input (ie, measure the voltage on the input). '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. A single conversion takes between 0.2mSec (8-bit) to 0.9mSec (16-bit). DIN Digital input If 'option' is omitted the input will be high impedance If 'option' is the keyword "PULLUP" a simulated resistor will be used to pull up the input pin to 3.3V If the keyword "PULLDOWN" is used the pin will be pulled down to zero volts. The pull up/down is a constant current of about 50µA. FIN Frequency input '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. Note that 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 '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. Note that 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
SETPIN pin, cfg [,option] …...continued	DOUT Digital output 'option' can be "OC" in which case the output will be open collector (or more correctly open drain). The functions PIN() and PORT() can also be used to return the value on one or more output pins . Previous versions of MMBasic used numbers for 'cfg' and the mode OOUT. For backwards compatibility they will still be recognised. 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: OFF Not configured or inactive INTH Interrupt on low to high input INTL Interrupt on high to low input INTB Interrupt on both (ie, 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 keywords "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().
SETTICK period, target [, nbr] SETTICK FAST frequency, target	This will setup a periodic interrupt (or "tick"). Four tick timers are available ('nbr' = 1, 2, 3 or 4). 'nbr' is optional and defaults to timer number 1. The time between interrupts is 'period' milliseconds and 'target' is the interrupt subroutine which will be called when the timed event occurs. The period can range from 1 to 2147483647 ms (about 24 days). These interrupts can be disabled by setting 'period' to zero This allows you to exceed the current maximum rate of 1 interrupt per millisecond (1000Hz) and has been tested up to 50KHz. If the interrupt routine overruns the time available then interrupts will be lost. If the program is executing a statement that takes longer than the time between interrupts the interrupts will be stacked.
SETTICK PAUSE, target [, nbr] or SETTICK RESUME, target [, nbr]	Pause or resume the specified tick timer. When paused the interrupt is delayed but the current count is maintained.
SORT array() [,indexarray()] [,flags] [,startposition] [,elementstosort]	This command takes an array of any type (integer, float or string) and sorts it into ascending order in place. It has an optional parameter 'indexarray%()'. If used this must be an integer array of the same size as the array to be sorted. After the sort this array will contain the original index position of each element in the array being sorted before it was sorted. Any data in the array will be overwritten.This allows connected arrays to be sorted. See the section *Sorting Data* in the tutorial Programming with the Micromite eXtreme for an example. The 'flag' parameter is optional and valid flag values are: bit0: 0 (default if omitted) normal sort - 1 reverse sort bit1: 0 (default) case dependent - 1 sort is case independent (string sorts only). The optional 'startposition' defines which element in the array to start the sort. Default is 0 (OPTION BASE 0) or 1 (OPTION BASE 1)

	The optional 'elementstosort' defines how many elements in the array should be sorted. The default is all elements after the startposition. Any of the optional parameters may be omitted so, for example, to sort just the first 50 elements of an array you could use: SORT array(), , , ,50
SPI OPEN speed, mode, bits or SPI READ nbr, array() or SPI WRITE nbr, data1, data2, data3, … etc or SPI WRITE nbr, string$ or SPI WRITE nbr, array() or SPI CLOSE	Communications via an SPI channel. The command SPI refers to channel 1. The command SPI2 refers to channel 2 and has an identical syntax. 'nbr' is the number of data items to send or receive 'data1', 'data2', etc can be float or integer and in the case of WRITE can be a constant or expression. If 'string$' is used 'nbr' characters will be sent. 'array' must be a single dimension float or integer array and 'nbr' elements will be sent or received. See Appendix D for the details.
SPRITE	The SPRITE commands are used to manipulate small graphic images. These are useful when writing games. The maximum size of a sprite is MM.HRES-1 and MM.VRES-1 See also the SPRITE() functions.
SPRITE CLOSE [#]n	Closes sprite "n" and releases its memory resources allowing the sprite number to be re-used. The command will give an error if other sprites are copied from this one unless they are closed first.
SPRITE CLOSE ALL	Closes all sprites and releases sprite memory. The screen is not changed.
SPRITE COPY [#]n, [#]m, nbr	Makes a copy of sprite "n" to "nbr" of new sprites starting a number "m". Copied sprites share the same loaded image as the original to save memory
SPRITE HIDE [#]n	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. The following commands cannot be used when all sprites are hidden: SPRITE SHOW (SAFE) SPRITE HIDE (SAFE, ALL) SPRITE SWAP SPRITE MOVE SPRITE SCROLLR SPRITE SCROLL
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 performance than the simple version and should only be used it there is a risk of the sprite being partially covered.
SPRITE INTERRUPT sub	Specifies the name of the subroutine that will be called when a sprite collision occurs. See Appendix E for how to use the function SPRITE to interrogate details of what has collided
SPRITE NOINTERRUPT	Disables collision interrupts.
SPRITE RESTORE	Restores all the sprites. NB that any position changes previously requested using SPRITE NEXT will be actioned by the RESTORE and collision detection will be run

SPRITE READ [#]n, x , y, w, h [,pagenumber]	Reads the display area specified by coordinates ‘x’ and ‘y’, width ‘w’ and height ‘h’ into buffer number ‘n’. If the buffer is already in use and the width and height of the new area are the same as the original then the new command will overwrite the stored area. The optional parameter page number specifies which page is to be read to create the sprite. The default is the current write page. Set the page to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command.
SPRITE WRITE [#]n, x y [,orientation]	Overwrites the display with the contents of sprite buffer ‘n’ with the top left at coordinates ‘x’, ‘y’. SPRITE WRITE overwrites the complete area of the display. The background that is overwritten is not stored so SPRITE WRITE is inherently higher performing than SPRITE SHOW but with greater functional limitations. The optional 'orientation' parameter defaults to 4 and specifies how the stored image data is changed as it is written out. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels
SPRITE LOAD fname$ [,start_sprite_number]	Loads the file ‘fname$’ which must be formatted as an original Colour Maximite sprite file. See the original Colour Maximite *MMBasic Language Manual* for the file format. Multiple sprite files can be loaded by specifying a different ‘start_sprite_number’ for each file. The programmer is responsible for making sure that the sprites do not overlap.
SPRITE LOADARRAY [#]n, w, h, array%()	Creates the sprite 'n' with width 'w' and height 'h' by reading w*h RGB888 values from 'array%()'. The RGB888 values must be stored in order of columns across and then rows down starting at the top left. This allows the programmer to create simple sprites in a program without needing to load them from disk or read them from the display. The firmware will generate an error if 'array%()' is not big enough to hold the number of values required.
SPRITE LOADPNG [#]n, fname$ [, transparency_cut_off]	Loads the PNG image ‘fname$’ as sprite number ‘n’. If the PNG file is in ARGB8888 format the ‘transparency_cut_off’ parameter is used to determine whether the pixel should be solid or missing/transparent. Valid values are 1 to 15, default is 8. MMBasic compares the 4 most significant bits of the transparency data in the file with the cut off value and assigns a transparency of 0 or 15 depending on the comparison. This allows RGB(0,0,0) to be a valid solid colour. If the file is in RGB888 format then an RGB level of 0,0,0 is used to determine transparency as there is no other information to use.
SPRITE LOADBMP [#]b, fname$ [,x] [,y] [,w] [,h]	SPRITE LOADBMP will load a blit buffer from a 24-bit bmp image file. x,y define the start position in the image to start loading and w,h specify the width and height of the area to be loaded. e.g. SPRITE LOAD #1,"image1", 50,50,100,100 will load an area of 100 pixels square with the top left had corner at 50,50 from the image *image1.bmp*
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 SCROLL x, y [,col]	Scrolls the background and any sprites on layer 0 'x' pixels to the right and 'y' pixels up. 'x' can be any number between -MM.HRES-1 and MM.HRES-1, 'y' can be any number between -MM.VRES-1 and MM.VRES-1. Sprites on any layer other than zero will remain fixed in position on the screen. By default the scroll wraps the image round. If 'col' is specified the colour will replace the area behind the scrolled image. If 'col' is set to -1 the scrolled area will be left untouched.
SPRITE SCROLLR x, y, w, h, delta_x, delta_y [,col]	Scrolls the region of the screen defined by top-right coordinates ‘x’ and ‘y’ and width and height ‘w’ and ‘h’ by ‘delta_x’ pixels to the right and ‘delta_y’ pixels up. By default the scroll wraps the background round. If ‘col’ is specified the colour will replace the area behind the scrolled image. Sprites on any layer other than zero will remain fixed in position on the screen. Sprites in layer zero where the centre of the sprite (x+ w/2, y+ h/2) falls within the scrolled region will move with the scroll and wrap round if the centre moves outside one of the boundaries of the scrolled region.
SPRITE SHOW [#]n, x,y, layer, [orientation]	Displays sprite ‘n’ on the screen with the top left at coordinates ‘x’, ‘y’. Sprites will only collide with other sprites on the same layer, layer zero, or with the screen edge. If a sprite is already displayed on the screen then the SPRITE SHOW command acts to move the sprite to the new location. The display background is stored as part of the command and will be replaced when the sprite is hidden or moved further. ‘orientation’ is optional and can be: 0 - normal display (default if omitted) 1 - mirrored left to right 2 - mirrored top to bottom 3 - rotated 180 degrees (= 1+2)
SPRITE SHOW SAFE [#]n, x,y, layer [,orientation] [,ontop]	Shows a sprite and automatically compensates for any other sprites that overlap it. If the sprite is not already being displayed the command acts exactly the same as SPRITE SHOW. If the sprite is already shown it is moved and remains in its position relative to other sprites based on the original order of writing. i.e. if sprite 1 was written before sprite 2 and it is moved to overlap sprite 2 it will display under sprite 2. If the optional "ontop" parameter is set to 1 then the sprite moved will become the newest sprite and will sit on top of any other sprite it overlaps. Refer to SPRITE SHOW for details of the orientation parameter.
SPRITE SWAP [#]n1, [#]n2 [,orientation]	Replaces the sprite ‘n1’ with the sprite ‘n2’. The sprites must have the same width and height and ‘n1’ must be displayed or an error will be generated. Refer to SPRITE SHOW for details of the orientation parameter. The replacement sprite inherits the background from the original as well as its position in the list of order drawn.
SPRITE TRANSPARENCY [#]n, transparency	Transparency can be between 1 and 15 and changes all pixels with a non-zero transparency in the stored sprite to the new level.
STATIC variable [, variables] See DIM for the full syntax.	Defines a list of variable names which are local to the subroutine or function. These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command). This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 31 characters. Static variables can be initialised to a value. This initialisation will take effect only on the first call to the subroutine (not on subsequent calls).

SUB xxx (arg1 [,arg2, …])
  <statements>
  <statements>
END SUB
	Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program.
	'xxx' is the subroutine name and it must meet the specifications for naming a variable. 'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. ie, arg3().
	The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using AS <type> (ie, arg1 AS STRING).
	Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit.

SUB … continued	You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: MySub a1, a2 When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.
TEMPR START pin [, precision]	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. 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 = 0.5ºC resolution, 100 ms conversion time. 1 = 0.25ºC resolution, 200 ms conversion time (this is the default). 2 = 0.125ºC resolution, 400 ms conversion time. 3 = 0.0625ºC resolution, 800 ms conversion time.
TEXT x, y, string$ [,alignment$] [, font] [, scale] [, c] [, bc]	Displays a string on the VGA monitor starting at 'x' and 'y'. 'string$' is the string to be displayed. Numeric data should be converted to a string and formatted using the Str$() function. ' alignment$' is a string expression or string variable consisting of 0, 1 or 2 letters where the first letter is the horizontal alignment around 'x' and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around 'y' and can be T, M or B for TOP, MIDDLE, BOTTOM. The default alignment is left/top. A third letter can be used in the alignment string to indicate the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (ie, upside down), 'U' the text will be rotated counter clockwise by 90º and 'D' the text will be rotated clockwise by 90º 'font' and 'scale' are optional and default to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.

TIME$ = "HH:MM:SS"
or
TIME$ = "HH:MM"
or
TIME$ = "HH"
	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.
	Note:
	• The time will be set to "00:00:00" on first power up.
	• The time will be remembered and kept updated as long as the battery is installed and can maintain a voltage of over 2.5V.
	• IMPORTANT: The date must also be set (using DATE$=) otherwise the correct time will be lost after the power is cycled.
	• Battery life should be 3 to 4 years even if the computer is powered off.

TIME$ = ±sec	Adds or subtracts 'sec' seconds from the current time being maintained by MMBasic. This makes it easier to fine tune the current time.
TIMER = msec	Resets the timer to a number of milliseconds. Normally this is just used to reset the timer to zero but you can set it to any positive integer. See the TIMER function for more details.
TRACE ON or TRACE OFF or TRACE LIST nn	TRACE ON/OFF will turn on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. TRACE LIST will list the last 'nn' lines executed in the format described above. MMBasic is always logging the lines executed so this facility is always available (ie, it does not have to be turned on).
TRIANGLE X1, Y1, X2, Y2, X3, Y3 [, C [, FILL]]	Draws a triangle on the VGA monitor 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.

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() VAR RESTORE will retrieve the previously saved variables and insert them (and their values) into the variable table. 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 CLEAR will erase all saved variables. Also, the saved variables will be automatically cleared by the NEW command or when a new program is loaded via AUTOSAVE, XMODEM, etc. This command is normally used to save calibration data, options, and other data which 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. The memory used is battery backed RAM which operates at high speed and can be written to an unlimited number of times without restriction (unlike the Micromite). • 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. • 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.
WATCHDOG timeout or WATCHDOG OFF	Starts the watchdog timer which will automatically restart the processor when it has timed out. This can be used to recover from some event that disabled the running program (such as an endless loop or a programming or other error that halts a running program). This can be important in an unattended control situation. 'timeout' is the time in milliseconds (ms) before a restart is forced. This command should be placed in strategic locations in the running BASIC program to constantly reset the watchdog timer and therefore prevent it from counting down to zero. If the timer count does reach zero (perhaps because the BASIC program has stopped running) the Maximite will be restarted and the automatic variable MM.WATCHDOG will be set to true (ie, 1) indicating that an error occurred. On a normal startup MM.WATCHDOG will be set to false (ie, 0). WATCHDOG OFF will disable the watchdog timer (this is the default on a reset or power up). The timer is also turned off when the break character (normally CTRL-C) is used on the console to interrupt a running program.

XMODEM SEND file$ [,comportno] or XMODEM RECEIVE file$ [,comportno]	Transfers a BASIC program to or from a remote computer using the XModem protocol. The transfer is done over the serial console connection. XMODEM SEND will send 'file$' held on the Colour Maximite's SD card to the remote device. SEND can be abbreviated to S. XMODEM RECEIVE will accept 'file$' sent by the remote device and save it on the Colour Maximite's SD card. If the file already exists it will be overwritten when receiving a file. RECEIVE can be abbreviated to R. The XModem protocol requires a cooperating software program running on the remote computer and connected to its serial port. 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. The transfer can take up to 15 seconds to start and if the XMODEM command fails to establish communications it will return to the MMBasic prompt after 60 seconds and leave the program memory untouched. If 'commportno' is specified the transfer will take place over the serial port specified (1 or 2). In this case the port must have been previously opened with an appropriate baudrate. Download Tera Term from http://ttssh2.sourceforge.jp/

Functions

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

Square brackets indicate that the parameter or characters are optional.

ABS(number)	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ACOS(number)	Returns the inverse cosine of the argument 'number' in radians.
ASC(string$)	Returns the ASCII code for the first letter in the argument 'string$'.
ASIN(number)	Returns the inverse sine value of the argument 'number' in radians.
ATAN2(y, x)	Returns the arc tangent of the two numbers x and y as an angle expressed in radians. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.
ATN(number)	Returns the arctangent of the argument 'number' in radians.
BASE$(base, number [, chars])	Returns a string giving the base value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s). Base can be between 2 and 36. Numbers greater than 9 are represented by a letter as per the HEX notation. Example: PRINT BASE$(36, 35) will display "Z" Internally the functions BIN$, OCT$, and HEX$ use this function.
BAUDRATE(comm [, timeout])	Returns the baudrate of any data received on the serial communications port 'comm'). This will sample the port over the period of 'timeout' seconds. 'timeout' will default to one second if not specified. 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 (ie, the least significant byte is the first one in the string). Setting the third parameter to 'BIG' will return the string in big-endian format (ie, the most significant byte is the first one in the string) In the case of the integer conversions, an error will be generated if the 'value' cannot fit into the 'type' (eg, an attempt to store the value 400 in a INT8). This function makes it easy to prepare data for efficient binary file I/O or for preparing numbers for output to sensors and saving to flash memory. See also the function STR2BIN
BOUND(array() [,dimension]	This returns the upper limit of the array for the dimension requested. The dimension defaults to one if not specified. Specifying a dimension value of 0 will return the current value of OPTION BASE. Unused dimensions will return a value of zero. For example: DIM myarray(44,45) BOUND(myarray(),2) will return 45

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

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

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 any I/O pins used with the HC-SR04 should be 5V capable as the HC-SR04 is a 5V device. 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 (ie, sensor not connected).

EOF([#]nbr)	Will return true if the file previously opened on the SD card for INPUT with the file number '#fnbr' is positioned at the end of the file. For a serial communications port this function will return true if there are no characters waiting in the receive buffer. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT$ function.
EPOCH(DATETIME$)	Returns the epoch number (number of seconds that have elapsed since midnight GMT on January 1, 1970) for the supplied DATETIME$ string. The format for DATETIME$ is "dd-mm-yyyy hh:mm:ss", "dd-mm-yy hh:mm:ss", or "yyyy-mm-dd hh:mm:ss",. Use NOW to get the epoch number for the current date and time, i.e. PRINT EPOCH(NOW)
EVAL(string$)	Will evaluate 'string$' as if it is a BASIC expression and return the result. 'string$' can be a constant, a variable or a string expression. The expression can use any operators, functions, variables, subroutines, etc that are known at the time of execution. The returned value will be an integer, float or string depending on the result of the evaluation. For example: S$ = `"COS(RAD(30)) * 100" : PRINT EVAL(S$)` Will display: `86.6025`
EXP(number)	Returns the exponential value of 'number', ie, e^x where x is 'number'.
FIELD$(string1, nbr, string2 [, string3])	Returns a particular field in a string with the fields separated by delimiters. 'nbr' is the field to return (the first is nbr 1). 'string1' is the string to search and 'string2' is a string holding the delimiters (more than one can be used). 'string3' is optional and if specified will include characters that are used to quote text in 'string1' (ie, quoted text will not be searched for a delimiter). For example: S$ = "foo, boo, zoo, doo" r$ = FIELD$(s$, 2, ",") will result in r$ = "boo". While: s$ = "foo, 'boo, zoo', doo" r$ = FIELD$(s$, 2, ",", "'") will result in r$ = "boo, zoo".
FIX(number)	Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point.

	For example 9.89 will return 9 and -2.11 will return -2. The major difference between FIX and INT is that FIX provides a true integer function (ie, does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .
FORMAT$(nbr [, fmt$])	Will return a string representing 'nbr' formatted according to the specifications in the string 'fmt$'. The format specification starts with a % character and ends with a letter. Anything outside of this construct is copied to the output as is. The structure of a format specification is: % [flags] [width] [.precision] type Where 'flags' can be: - Left justify the value within a given field width 0 Use 0 for the pad character instead of space + Forces the + sign to be shown for positive numbers space Causes a positive value to display a space for the sign. Negative values still show the – sign 'width' is the minimum number of characters to output, less than this the number will be padded, more than this the width will be expanded. 'precision' specifies the number of fraction digits to generate with an e, or f type or the maximum number of significant digits to generate with a g type. If specified, the precision must be preceded by a dot (.). 'type' can be one of: g Automatically format the number for the best presentation. f Format the number with the decimal point and following digits e Format the number in exponential format If uppercase G or F is used the exponential output will use an uppercase E. If the format specification is not specified "%g" is assumed. Examples: format$(45) will return 45 format$(45, "%g") will return 45 format$(24.1, "%g") will return 24.1 format$(24.1,"%f") will return 24.100000 format$(24.1, "%e") will return 2.410000e+01 format$(24.1,"%09.3f") will return 00024.100 format$(24.1,"%+.3f") will return +24.100 format$(24.1,"**%-9.3f**") will return **24.100 **
GETSCANLINE	This will report on the line that is currently being drawn on the VGA monitor. Using this to time updates to the screen can avoid timing effects caused by updates while the screen is being updated. The first visible line will return a value of 0. Any line number above MM.VRES is in the frame blanking period.
GPS()	The GPS functions are used to return data from a serial communications channel opened as GPS. The function GPS(VALID) should be checked before any of these functions are used to ensure that the returned value is valid.
GPS(ALTITUDE)	Returns current altitude (if sentence GGA is enabled).
GPS(DATE)	Returns the normal date string corrected for local time e.g. "12-01-2020".
GPS(DOP)	Returns DOP (dilution of precision) value (if sentence GGA is enabled).
GPS(FIX)	Returns DOP (dilution of precision) value (if sentence GGA is enabled).
GPS(GEOID)	Returns the geoid-ellipsoid separation (if sentence GGA is enabled).
GPS(LATITUDE)	Returns the latitude in degrees as a floating point number, values are negative for South of equator
GPS LONGITUDE)	Returns the longitude in degrees as a floating point number, values are negative for West of the meridian.
GPS(SATELLITES)	Returns number of satellites in view (if sentence GGA is enabled).
GPS(SPEED)	Returns the ground speed in knots as a floating point number.
GPS(TIME)	Returns the normal time string corrected for local time e.g. "12:09:33".
GPS(TRACK)	Returns the track over the ground (degrees true) as a floating point number.
GPS(VALID)	Returns: 0=invalid data, 1=valid data
HEX$(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 this is a carriage return, it is likely that there will be a line feed character following as often the enter key will produce a CR/LF pair. If the input buffer is empty this function will immediately return with an empty string (ie, "").
INPUT$(nbr, [#]fnbr)	Will return a string composed of 'nbr' characters read from a file on the SD card previously opened for INPUT with the file number '#fnbr'. This function will read all characters including carriage return and new line without translation. Will return a string composed of 'nbr' characters read from a serial communications port opened as 'fnbr'. This function will return as many characters as are waiting in the receive buffer up to 'nbr'. If there are no characters waiting it will immediately return with an empty string. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, string-pattern$)	Returns the position at which 'string-pattern$' occurs in 'string-searched$', beginning at 'start-position'. Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern$' is not found.
INT(number)	Truncate an expression to the next whole number less than or equal to the argument. For example 9.89 will return 9 and -2.11 will return -3. This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function. See also CINT() .
JSON$(array%(),string$)	Returns a string representing a specific item out of the JSON input stored in the longstring array%() Examples taken from api.openweathermap.org JSON$(a%(), "name") JSON$(a%(), "coord.lat") JSON$(a%(), "weather[0].description") JSON$(a%(),"list[4].weather[0].description
KEYDOWN(n)	Return the decimal ASCII value of the 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 F. This function will report multiple simultaneous key presses and the parameter 'n' is the number of the keypress to report. KEYDOWN(0) will return the number of keys being pressed For example, if "c", "g" and "p" are pressed simultaneously KEYDOWN(0) will return 3, KEYDOWN(1) will return 99, KEYDOWN(2) will return 103, etc. The keys do not need to be pressed simultaneously and will report in the order pressed. Taking a finger off a key will promote the next key pressed to #1. The first key ('n' = 1) is entered in the keyboard buffer (accessible using INKEY$) while keys 2 to 6 can only be accessed via this function. Using this function will clear the console input buffer. KEYDOWN(7) will give any modifier keys that are pressed. These keys do not add to the count in keydown(0) The return value is a bitmask as follows: lalt ? 1, lctrl ? 2, lgui ? 4, lshift ? 8, ralt ? 16, rctrl ? 32, rgui ? 64, rshift ? 128 KEYDOWN(8) will give the current status of the lock keys. These keys do not add to the count in keydown(0) The return value is a bitmask as follows: caps_lock ? 1, num_lock ? 2, scroll_lock ? 4 Note that some keyboards will limit the number of active keys that they can report on.
LCASE$(string$)	Returns 'string$' converted to lowercase characters.
LCOMPARE(array1%(), array2%())	Compare the contents of two long string variables array1%() and array2%(). The returned is an integer and will be -1 if array1%() is less than array2%(). It will be zero if they are equal in length and content and +1 if array1%() is greater than array2%(). The comparison uses the ASCII character set and is case sensitive.

LEFT$(string$, nbr)	Returns a substring of ‘string$’ with ‘nbr' of characters from the left (beginning) of the string.
LEN(string$)	Returns the number of characters in 'string$'.
LGETBYTE(array%(), n)	Returns the numerical value of the 'n'th byte in the LONGSTRING held in 'array%()'. This function respects the setting of OPTION BASE in determining which byte to return.
LGETSTR$(array%(), start, length)	Returns part of a long string stored in array%() as a normal MMBasic string. The parameters start and length define the part of the string to be returned.
LINSTR(array%(), search$ [,start])	Returns the position of a search string in a long string. The returned value is an integer and will be zero if the substring cannot be found. array%() is the string to be searched and must be a long string variable. Search$ is the substring to look for and it must be a normal MMBasic string or expression (not a long string). The search is case sensitive. Normally the search will start at the first character in 'str' but the optional third parameter allows the start position of the search to be specified.
LLEN(array%())	Returns the length of a long string stored in array%()
LOC([#]fnbr)	For a file on the SD card opened as RANDOM this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. For a serial communications port opened as 'fnbr' this function will return the number of bytes received and waiting in the receive buffer to be read. #0 can be used which refers to the console's input buffer. The # is optional.
LOF([#]fnbr)	For a file on the SD card this will return the current length of the file in bytes. For a serial communications port opened as 'fnbr' this function will return the space (in characters) remaining in the transmit buffer. Note that when the buffer is full MMBasic will pause when adding a new character and wait for some space to become available. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.
MATH	The math function performs many simple mathematical calculations that can be programmed in Basic but there are speed advantages to coding looping structures in C and there is the advantage that once debugged they are there for everyone without re-inventing the wheel.
Simple functions	
MATH(ATAN3 x,y)	Returns ATAN3 of x and y
MATH(COSH a)	Returns the hyperbolic cosine of a
MATH(LOG10 a)	Returns the base 10 logarithm of a
MATH(SINH a)	Returns the hyperbolic sine of a
MATH(TANH a)	Returns the hyperbolic tan of a
Simple Statistics	
MATH(CHI a())	Returns the Pearson's chi-squared value of the two dimensional array a())
MATH(CHI_p a())	Returns the associated probability in % of the Pearson's chi-squared value of the two dimensional array a())
MATH(CORREL a(), a())	Returns the Pearson’s correlation coefficient between arrays a() and b()
MATH(MAX a())	Returns the maximum of all values in the a() array, a() can have any number of dimensions
MATH(MEAN a())	Returns the average of all values in the a() array, a() can have any number of dimensions

MATH(MEDIAN a())	Returns the median of all values in the a() array, a() can have any number of dimensions
MATH(MIN a())	Returns the minimum of all values in the a() array, a() can have any number of dimensions
MATH(SD a())	Returns the standard deviation of all values in the a() array, a() can have any number of dimensions
MATH(SUM a())	Returns the sum of all values in the a() array, a() can have any number of dimensions
Vector Arithmetic	
MATH(MAGNITUDE v())	Returns the magnitude of the vector v(). The vector can have any number of elements
MATH(DOTPRODUCT v1(), v2())	Returns the dot product of two vectors v1() and v2(). The vectors can have any number of elements but must have the same cardinality
Matrix Arithmetic	
MATH(M_DETERMINANT array!())	Returns the determinant of the array. The array must be square.

MAX(arg1 [, arg2 [, …]])
or
MIN(arg1 [, arg2 [, …]])
	Returns the maximum or minimum number in the argument list.
	Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.

MID$(string$, start) or MID$(string$, start, nbr)	Returns a substring of ‘string$’ beginning at ‘start’ and continuing for ‘nbr’ characters. The first character in the string is number 1. If ‘nbr’ is omitted the returned string will extend to the end of ‘string$’
MOUSE(funct [, channel])	Returns data from a Mouse controller supporting the Hobbytronic protocol. 'channel' is optional and is the I^{2}C channel for the controller (defaults to 2, pins 27 and 28). 'funct' is a 1 letter code indicating the information to return as follows: X returns the value of the mouse X-position Y returns the value of the mouse Y-position L returns the value of the left mouse button (1 if pressed) R returns the value of the right mouse button (1 if pressed) W returns the value of the scroll wheel mouse button (1 if pressed) D This allows you to detect a double click of the left mouse button . The algorithm say the two clicks must occur between 100 and 500 milliseconds apart. The report via MOUSE(D) is then valid for 500mSec before it times out or until it is read. T returns 0
MOUSE(funct, 0)	Returns data from a PS2 mouse 'funct' is a 1 letter code indicating the information to return as follows: X returns the value of the mouse X-position Y returns the value of the mouse Y-position L returns the value of the left mouse button (1 if pressed) R returns the value of the right mouse button (1 if pressed) W returns the value of the scroll wheel mouse button (1 if pressed) D This allows you to detect a double click of the left mouse button. The algorithm requires that the two clicks must occur between 100 and 500 milliseconds apart. The report via MOUSE(D) is then valid for 500mSec before it times out or until it is read. T This returns 3 if a PS2 mouse has a scroll wheel or 0 if not.
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%)
PEEK(SHORT addr%)
PEEK(WORD addr%)
	Will return a byte or a word within the CPU’s virtual memory space.
	BYTE will return the byte (8-bits) located at 'addr%'
	SHORT will return the short integer (16-bits) located at 'addr%'

PEEK(INTEGER addr%)
PEEK(FLOAT addr%
PEEK(VARADDR var)
 	
PEEK(VARHEADER var)
PEEK(VAR var, ±offset)
 	
PEEK(VARTBL, ±offset)
 	
PEEK(PROGMEM, ±offset)
	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 descriptor of the variable var in memory. An array is specified as var()
	VAR, will return a byte in the memory allocated to 'var'. An array is specified as var().
	VARTBL, will return a byte in the memory allocated to the variable table maintained by MMBasic. Note that there is a comma after VARTBL.
	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 analog 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). This function will also return the state of a pin configured as an output. Also see the SETPIN and PIN() = commands.
PIN(function)	Returns the value of a special function. 'function' is a string (ie, it can be a string variable or string 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. "DAC1" The output voltage of DAC1 (on the audio output). "DAC2" The output voltage of 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 * PIN("SREF") / PIN("IREF") and this can be used to set OPTION VCC.
PIXEL(x, y [,page_number])	Returns the colour of a pixel on the VGA monitor. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates. The optional parameter page_number specifies which page is to be read. The default is the current write page. Set the page number to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command
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 an input will cause an error. The start/nbr pair can be repeated up to 25 times if additional groups of input pins need to be added. This function will also return the state of a pin configured as an output. It can be used to conveniently communicate with parallel devices like memory chips. Any number of I/O pins (and therefore bits) can be used from 1 to the number of I/O pins on the chip. 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 1 second with 0.1µs 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) 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 (ie, 100ms).
	This function returns the width of the pulse in microseconds (µs) or -1 if a timeout has occurred. The measurement is accurate to ±1 µs.
	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 and grey or gray (USA spelling). For example, RGB(red) or RGB(cyan).
	There is also one special colour 'notblack', For any video mode this is the darkest colour that is not treated as black by various graphics commands.
	'trans' is the level of transparency for colour depths 4 and 12. It is optional and defaults to 15 if not specified.

RIGHT$(string$, number-of-chars)	Returns a substring of 'string$' with 'number-of-chars' from the right (end) of the string.
RND(number) or RND	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The Micromite eXtreme uses the hardware random number generator in the ARM Cortex-M7 to deliver true random numbers. This means that the RANDOMIZE command is no longer needed and is not supported.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE$(number)	Returns a string of blank spaces 'number' characters long.

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()	The SPRITE functions return information regarding sprites which are small graphic images on the VGA screen. These are useful when writing games. See also the SPRITE commands.
SPRITE(C, [#]n)	Returns the number of currently active collisions for sprite n. If n=0 then returns the number of sprites that have a currently active collision following a SPRITE SCROLL command
SPRITE(C, [#]n, m)	Returns the number of the sprite which caused the "m"th collision of sprite n. If n=0 then returns the sprite number of "m"th sprite that has a currently active collision following a SPRITE SCROLL command. If the collision was with the edge of the screen then the return value will be: &HF1 collision with left of screen &HF2 collision with top of screen &HF4 collision with right of screen &HF8 collision with bottom of screen
SPRITE(D ,[#]s1, [#]s2) SPRITE(E, [#]n	Returns the distance between the centres of sprites 's1' and 's2' (returns -1 if either sprite is not active) 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,

SPRITE(H,[#]n)	8=bottom of screen Returns the height of sprite n. This function is active whether or not the sprite is currently displayed (active).
SPRITE(L, [#]n)	Returns the layer number of active sprites number n
SPRITE(N)	Returns the number of displayed (active) sprites
SPRITE(N,n)	Returns the number of displayed (active) sprites on layer n
SPRITE(S)	Returns the number of the sprite which last caused a collision. NB if the number returned is Zero then the collision is the result of a SPRITE SCROLL command and the SPRITE(C…) function should be used to find how many and which sprites collided.
SPRITE(V,spriteno1,spriteno2)	Returns the vector from 'spriteno1' to 'spriteno2' in radians. The angle is based on the clock so if 'spriteno2' is above 'spriteno1' on the screen then the answer will be zero. This can be used on any pair of sprites that are visible. If either sprite is not visible the function will return -1. This is particularly useful after a collision if the programmer wants to make some differential decision based on where the collision occurred. The angle is calculated between the centre of each of the sprites which may of course be different sizes.

SPRITE(T, [#]n)
 	
SPRITE(V,[#]so1, [#]s2)
 	
SPRITE(W, [#]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 the vector from sprite 's1' to 's2' in radians.
	The angle is based on the clock so if 's2' is above 's1' on the screen then the answer will be zero. This can be used on any pair of sprites that are visible. If either sprite is not visible the function will return -1.
	This is particularly useful after a collision if the programmer wants to make some differential decision based on where the collision occurred.
	The angle is calculated between the centre of each of the sprites which may of course be different sizes.
	
	Returns the width of sprite n. This function is active whether or not the sprite is currently displayed (active).

SPRITE(X, [#]n)	Returns the X-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.
SPRITE(Y, [#]n)	Returns the Y-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.

STR2BIN(type, string$ [,BIG])	Returns a number equal to the binary representation in ‘string$’. ‘type’ can be: INT64 converts 8 byte string representing a signed 64-bit integer to an integer UINT64 converts 8 byte string representing an unsigned 64-bit integer to an integer INT32 converts 4 byte string representing a signed 32-bit integer to an integer UINT32 converts 4 byte string representing an unsigned 32-bit integer to an integer INT16 converts 2 byte string representing a signed 16-bit integer to an integer UINT16 converts 2 byte string representing an unsigned 16-bit integer to an integer INT8 converts 1 byte string representing a signed 8-bit integer to an integer UINT8 converts 1 byte string representing an unsigned 8-bit integer to an integer SINGLE converts 4 byte string representing single precision float to a float DOUBLE converts 8 byte string representing single precision float to a float By default the string must contain the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to ‘BIG’ will interpret the string in big-endian format (ie, the most significant byte is the first one in the string). This function makes it easy to read data from binary data files, interpret numbers from sensors or efficiently read binary data from flash memory chips. An error will be generated if the string is the incorrect length for the conversion requested See also the function BIN2STR$
SQR(number)	Returns the square root of the argument 'number'.
STR$(number) or STR$(number, m) or STR$(number, m, n) or STR$(number, m, n, c$)	Returns a formatted string in decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR$(123.456) will return "123.456" STR$(-123.456) will return "-123.456" STR$(123.456, 1) will return "123.456" STR$(123.456, -1) will return "+123.456" STR$(123.456, 6) will return " 123.456" STR$(123.456, -6) will return " +123.456" STR$(-123.456, 6) will return " -123.456" STR$(-123.456, 6, 5) will return " -123.45600" STR$(-123.456, 6, -5) will return " -1.23456e+02" STR$(53, 6) will return " 53" STR$(53, 6, 2) will return " 53.00" STR$(53, 6, 2, "*") will return "****53.00"

STRING$(nbr, ascii) or STRING$(nbr, string$)	Returns a string 'nbr' bytes long consisting of either the first character of string$ or the character representing the ASCII value 'ascii' which is a decimal number in the range of 32 to 126.
TAB(number)	Outputs spaces until the column indicated by 'number' has been reached on the console output. The tab function will not work when printing to a file but will behave like the SPACE$ function.
TAN(number)	Returns the tangent of the argument 'number' in radians.
TEMPR(pin)	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 0.25ºC. 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. 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. See the chapter "Special Hardware Devices" for more details.
TIME$	Returns the current time based on MMBasic's internal clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00". If the OPTION MILLISECONDS ON command has been used this function will return the time including milliseconds as a decimal fraction of the seconds. For example: "14:35:06.239". 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 to any value by using TIMER as a command.
UCASE$(string$)	Returns ‘string$’ converted to uppercase characters.
VAL(string$)	Returns the numerical value of the ‘string$’. If 'string$' is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.

Obsolete Commands and Functions

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

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

DHT22	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 (ie, SUB…END SUB).
IF condition THEN linenbr	For Microsoft compatibility a GOTO is assumed if the THEN statement is followed by a number. A label is invalid in this construct. New programs should use: IF condition THEN GOTO linenbr | label
IRETURN	Returns from an interrupt when the interrupt destination was a line number or a label. New programs should use a user defined subroutine as an interrupt destination. In that case END SUB or EXIT SUB will cause a return from the interrupt.
ON nbr GOTO | GOSUB target[,target, target,...]	ON either branches (GOTO) or calls a subroutine (GOSUB) based on the rounded value of 'nbr'; if it is 1, the first target is called, if 2, the second target is called, etc. Target can be a line number or a label. New programs should use SELECT CASE.
POS	For the console, returns the current cursor position in the line in characters.
RETURN	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB.

Appendix A

Serial Communications

Three serial ports are available for asynchronous serial communications labelled COM1: , COM2: and COM3:. In addition, if the serial console is disabled then that port is available as COM4:.

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, 7BIT, (ODD or EVEN), INV, OC, S2"` where:

- 'n' is the serial port number for either COM1:, COM2:, COM3: or COM4:.
- 'baud' is the baud rate. This can be any value between 1200 (the minimum) and 1000000 (1MHz). Default is 9600.
- 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes.
- 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt.
- 'int-trigger' sets the trigger condition for calling the interrupt subroutine. It is an integer and the interrupt subroutine will be called when this number of characters has arrived in the receive queue.

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

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

- '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

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 diagrams in the beginning of this manual. Note that Tx means an output from the Micromite eXtreme and Rx means an input to the Micromite eXtreme.

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 SERIAL PULLUP DISABLE 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 Micromite eXtreme 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.

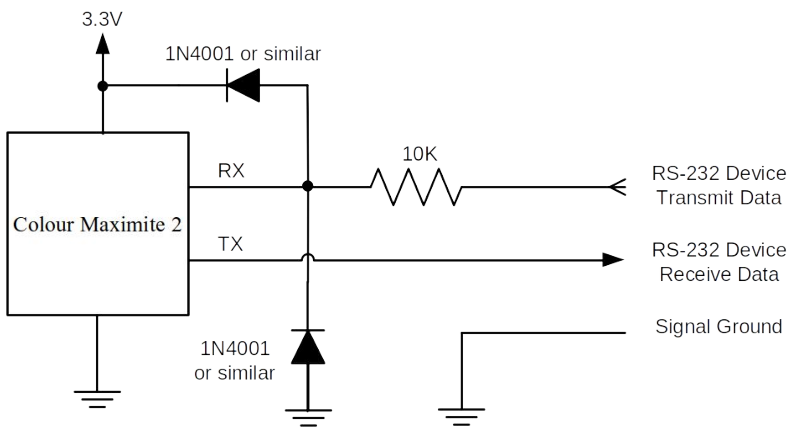
The first issue is that the signalling polarity is inverted with respect to TTL. On the Micromite eXtreme COM1: can be specified to invert the transmit and receive signal (the 'INV' option) so that is an easy fix.

For the receive data (that is the ±12V signal from the remote RS-232 device) it is easy to limit the voltage using a series resistor of (say) 10KΩ 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 ±12V it will be safely clipped by the diodes.

For the transmit signal (from the Micromite eXtreme 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
I^{2}C Communications

The Micromite eXtreme implements three I^{2}C 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).

There are four commands that can be used:

I2C OPEN speed, timeout	Enables the I^{2}C module in master mode. The I2C command refers to channel 1 while commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax. '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 [,sendata ....]	Send data to the I^{2}C slave device. The I2C command refers to channel 1 while commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax. 'addr' is the slave's I^{2}C 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) 'sendlen' is the number of bytes to send. 'senddata' is the data to be sent - this can be specified in various ways (all values sent will be between 0 and 255): • The data can be supplied as individual bytes on the command line. Example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25 • The data can be in a one dimensional array specified with empty brackets (ie, no dimensions). 'sendlen' bytes of the array will be sent starting with the first element. Example: I2C WRITE &H6F, 0, 3, ARRAY() • The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 0, 3, STRING$
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I^{2}C slave device. The I2C command refers to channel 1 while commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax. 'addr' is the slave's I^{2}C 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) 'rcvlen' is the number of bytes to receive. 'rcvbuf' is the variable or array used to save the received data - this can be: • A string variable. Bytes will be stored as sequential characters in the string. • A one dimensional array of numbers specified with empty brackets. Received bytes will be stored in sequential elements of the array starting with the first. Example: I2C READ &H6F, 0, 3, ARRAY() • A normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE	Disables the master I^{2}C module and returns the I/O pins to a "not configured" state. They can then be configured using SETPIN. This command will also send a stop if the bus is still held. The I2C command refers to channel 1 while commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax.

Following an I^{2}C 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

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 at the beginning of this manual for the pin numbers used for the I^{2}C channels 1 and 2. Their signals are marked as data line (SDA) and clock (SCL). I^{2}C 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. Then can then be configured as per normal using SETPIN.

Both the data line (SDA) and clock (SCL) for all three I^{2}C 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 I^{2}C 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 I^{2}C 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 I^{2}C 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

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:

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 - Data to send or variable to receive.
(the number of data items must agree with the length parameter).

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

The Serial Peripheral Interface (SPI) communications protocol is used to send and receive data between integrated circuits. The command SPI refers to channel 1, SPI2 refers to channel 2 and SPI3 refers to channel 3. SPI2 is normally reserved for Touch and SD Card.

I/O Pins

The SPI OPEN command will automatically configure the relevant I/O pins on the rear I/O connector (listed at the start of this manual). MISO stands for Master In Slave Out and because the Micromite eXtreme 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 8, 16 or 32.
- 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
or
SPI WRITE nbr, string$
or
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.

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). 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:

SPI READ nbr, array()

Where 'nbr' is the number of data items to be received and array() is a single dimension integer array where the received data items will be saved. 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
Sprites

The concept of the sprite implementation is as follows:

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

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

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

```
SPRITE INTERRUPT collision
```

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

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

' get details of the specific collisions for a given sprite
SUB process_collision(S AS INTEGER)
  LOCAL INTEGER i, j
  ' SPRITE(C, #n) returns the number of current collisions for sprite n
  PRINT "Total of " SPRITE(C, S) " collisions"
   FOR I = 1 TO SPRITE(C, S)
    ' SPRITE(C, S, i) will tell us the sprite number of the “I”th sprite
    j = SPRITE(C, S, i)
    IF j = &HF1 THEN
      PRINT "collision with left of screen"
    ELSE IF j = &HF2 THEN
      PRINT "collision with top of screen"
    ELSE IF j = &HF4 THEN
      PRINT "collision with right of screen"
    ELSE IF j = &HF8 THEN
```

```
      PRINT "collision with bottom of screen"
    ELSE
      ' SPRITE(C, #n, #m) returns details of the mth collision
      PRINT "Collision with sprite ", SPRITE(C, S, i)
    ENDIF
  NEXT i
END SUB
```

Appendix F

Special Keyboard Keys

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

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

Keyboard Key	Key Code (Hex)	Key Code (Decimal)
DEL	7F	127
Up Arrow	80	128
Down Arrow	81	129
Left Arrow	82	130
Right Arrow	83	131
Insert	84	132
Home	86	134
End	87	135
Page Up	88	136
Page Down	89	137
Alt	8B	139
F1	91	145
F2	92	146
F3	93	147
F4	94	148
F5	95	149
F6	96	150
F7	97	151
F8	98	152
F9	99	153
F10	9A	154
F11	9B	155
F12	9C	156
PrtScr/SysRq	9D	157
PAUSE/BREAK	9E	158
SHIFT_TAB	9F	159
SHIFT_DEL	A0	160
SHIFT_DOWN_ARROW	A1	161
SHIFT_RIGHT_ARROW	A3	163

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

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

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

Appendix G

144 Pin Backpack PCB GPIO

MMX144 GPIO PINOUT

(component side view)

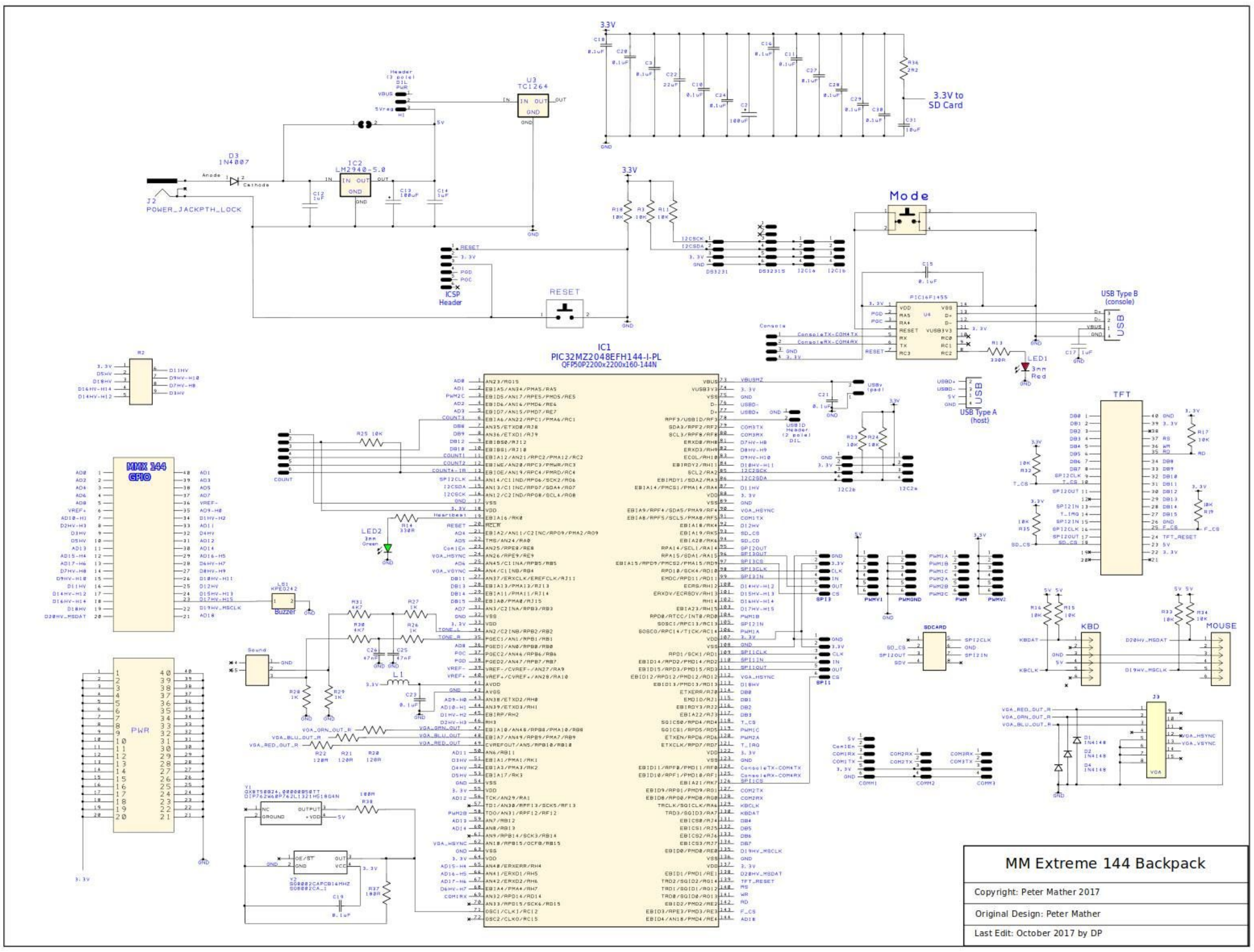
FUNCTION	MZ PIN	GPIO PINS				MZ PIN	FUNCTION
AD1	2	40	A D	A D	1	1	AD0
AD3	5	39	A D	A D	2	4	AD2
AD5	22	38	A D	A D	3	21	AD4
AD7	31	37	A D	A D	4	25	AD6
VREF- (also AIN, DIN, DOUT)	39	36	⬛🟥 A D	A D	5	36	AD8
AD9-H0 (also used by Camera)	43	35	A D	⬛🟥 A D	6	40	VREF+ (also AIN, DIN, DOUT)
D1HV-H2 (also used by Camera)	45	34	D5	A D	7	44	AD10-H1 (also used by Camera)
AD11	50	33	A D	D5	8	46	D2HV-H3 (also used by Camera)
D4HV	52	32	D5	10K PullUp D5	9	51	D3HV-10K pullup
AD12	56	31	A D	10K PullUp D5	10	53	D5HV-10K pullup
AD14	60	30	A D	A D	11	59	AD13
AD16-H5 (also used by Camera)	66	29	A D	A D	12	65	AD15-H4 (also used by Camera)
D6HV-H7 (also used by Camera)	68	28	D5	A D	13	67	AD17-H6 (also used by Camera)
D8HV-H9 (also used by Camera)	82	27	D5	10K PullUp D5	14	81	D7HV-H8-10K pullup (also used by Camera)
D10HV-H11 (also used by Camera)	84	26	D5	10K PullUp D5	15	83	D9HV-H10-10K pullup (also used by Camera)
D12HV	92	25	D5	10K PullUp D5	16	87	D11HV-10KPU
D15HV-H13 (also used by Camera)	101	24	D5	10K PullUp D5	17	100	D14HV-H12-10K pullup (also used by Camera)
D17HV-H15 (also piezzo buzzer, camera)	103	23	🟨🟪 D	10K PullUp D5	18	102	D16HV-H14-10K pullup (also used by Camera)
D19HV (also Mouse clock)	135	22	🟨🟪 D	10K PullUp D5	19	113	D18HV
AD18	144	21	A D	🟨🟪 D	20	138	D20HV-MSDAT (also mouse data)

🟨🟪	MSCLK - MZ PIN 135 MSDAT - MZ PIN 138 BUZZER - MZ PIN 102	D · D5 · 10K PullUp D · 10K PullUp D5	Digital input/output pins (D5 pins are 5V tolerent).
⬛🟥	VREF - and + (also AIN,DIN or DOUT pins)	A D · A D5	Analog input pins or digital I/O pins (D5 pins are 5V tolerent).

Appendix H.

144 Pin Backpack

Schematic



Appendix I – Sprites

Sprites

The concept of the sprite implementation is as follows:

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

Because moving a sprite or, particularly, scrolling the background can cause multiple sprite collisions it is important to understand how they can be interrogated. The best way to deal with a sprite collision is using the interrupt facility. A collision interrupt routine is set up using the SPRITE INTERRUPT command. Eg:

```
      SPRITE INTERRUPT collision
```

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

```
'
` This routine demonstrates a complete interrogation of collisions
'
SUB collision
  LOCAL INTEGER i
' First use the SPRITE(S) function to see what caused the interrupt
  IF SPRITE(S) <> 0 THEN 'collision of specific individual sprite
    'SPRITE(S) returns the sprite that moved to cause the collision
    PRINT "Collision on sprite ", SPRITE(S)
    process_collision(SPRITE(S))
    PRINT
  ELSE  ' 0 means collision of one or more sprites caused by background move
        ' SPRITE(C, 0) will tell us how many sprites had a collision
    PRINT "Scroll caused a total of ", SPRITE(C,0)," sprites to have collisions"
    FOR I = 1 TO SPRITE(C, 0)
     ' SPRITE(C, 0, i) will tell us the sprite number of the "I"th sprite
      PRINT "Sprite ", SPRITE(C, 0, i)
      process_collision(SPRITE(C, 0, i))
    NEXT i
    PRINT
  ENDIF
END SUB
`
' get details of the specific collisions for a given sprite
SUB process_collision(S AS INTEGER)
  LOCAL INTEGER i, j
' SPRITE(C, #n) returns the number of current collisions for sprite n
  PRINT "Total of " SPRITE(C, S) " collisions"
  FOR I = 1 TO SPRITE(C, S)
  ' SPRITE(C, S, i) will tell us the sprite number of the "I"th sprite
    j = SPRITE(C, S, i)
    IF j = &HF1 THEN
      PRINT "collision with left of screen"
    ELSE IF j = &HF2 THEN
      PRINT "collision with top of screen"
    ELSE IF j = &HF4 THEN
      PRINT "collision with right of screen"
    ELSE IF j = &HF8 THEN
      PRINT "collision with bottom of screen"
    ELSE
    ' SPRITE(C, #n, #m) returns details of the mth collision
      PRINT "Collision with sprite ", SPRITE(C, S, i)
    ENDIF
  NEXT i
END SUB
```

Appendix J.

Peter's Notes:

V5.07.00b4

100 and 144 pin

Micromite.X.production.zip

64 pin

Micromite.X.production.zip

POLYGON command brought up to latest syntax as per PicoMite
SPRITE MEMORY and SPRITE MEMORY COMPRESSED as per PicoMite
CSUB, CFUNCTION and DEFINE FONT functionality fixed

Major rework of all things file related
FATFS brought up to latest release
KILL command updated as per PicoMite
FILES command as per PicoMite
COPY command added

V5.07.00b3

100 and 144 pin
Micromite.X.production.zip

64 pin
Micromite.X.production.zip

This should be getting near finished. Just file handling to bring up-to-date

Changes:
BITBANG HUMID replaces HUMID
BITBANG LCD replaces LCD
BITBANG WS2812 introduced (same syntax as PicoMite)
FRAMEBUFFER command - same as PicoMite - works for all displays
All Sprite functionality reworked
Multiline comments (same as latest PicoMite beta)

Revised screen support

SPI
ILI9488, ILI9341, ILI9481 (Pi HAT version)

8-Bit parallel
SSD1963_4, SSD1963_5, SSD1963_5A, SSD1963_7, SSD1963_7A, SSD1963_8

16-Bit parallel
SSD1963_4_16, SSD1963_5_16, SSD1963_5A_16, SSD1963_7_16, SSD1963_7A_16, SSD1963_8_16
ILI9341_16, IPS_4_16 (supports OTM8009A and NT35510)

VGA
640x480 RGB111, 640x400 RGB111

Demo of sprites and framebuffer - should work on any SPI based display except ILI9481 (which doesn't support reading the controllers framebuffer). Also works on SSD1963 displays as well as the Micromite eXtreme Backpack144 in VGA mode.

```
Option explicit
Option default none
Framebuffer create
Framebuffer write f
CLS
'brownian motion demo using sprites
Dim integer x(64),y(64),c(64)
Dim float direction(64)
Dim integer i,j,k, collision=0
Dim string q$
For i=1 To 64
  direction(i)=Rnd*360 'establish the starting direction for each atom
  c(i)=RGB(Rnd*255,Rnd*255,Rnd*255) 'give each atom a colour
  Circle 10,10,4,1,,RGB(white),c(i) 'draw the atom
  SPRITE read i,6,6,9,9 'read it in as a sprite
Next i
CLS RGB(blue)
Box 0,0,MM.HRes,MM.VRes
k=1
For i=MM.HRes\9 To MM.HRes\9*8 Step MM.HRes\9
  For j=MM.VRes\9 To MM.VRes\9*8 Step MM.VRes\9
    SPRITE show k,i,j,1
    x(k)=i
    y(k)=j
    vector k,direction(k), 0, x(k), y(k) 'load up the vector move
    k=k+1
  Next j
Next i
'
Do
  For i=1 To 64
    vector i, direction(i), 1, x(i), y(i)
    SPRITE show i,x(i),y(i),1
    If sprite(S,i)<>-1 Then
      break_collision i
    EndIf
  Next i
Framebuffer copy f,n
Loop
'
Sub vector(myobj As integer, angle As float, distance As float, x_new As integer, y_new As integer)
  Static float y_move(64), x_move(64)
  Static float x_last(69), y_last(64)
  Static float last_angle(64)
  If distance=0 Then
    x_last(myobj)=x_new
    y_last(myobj)=y_new
  EndIf
  If angle<>last_angle(myobj) Then
    y_move(myobj)=-Cos(Rad(angle))
    x_move(myobj)=Sin(Rad(angle))
    last_angle(myobj)=angle
  EndIf
  x_last(myobj) = x_last(myobj) + distance * x_move(myobj)
  y_last(myobj) = y_last(myobj) + distance * y_move(myobj)
  x_new=Cint(x_last(myobj))
  y_new=Cint(y_last(myobj))
End Sub

' keep doing stuff until we break the collisions
Sub break_collision(atom As integer)
  Local integer j=1
  Local float current_angle=direction(atom)
  ' start by a simple bounce to break the collision
  If sprite(e,atom)=1 Then
    ' collision with left of screen
    current_angle=360-current_angle
  ElseIf sprite(e,atom)=2 Then
    ' collision with top of screen
   current_angle=((540-current_angle) Mod 360)
  ElseIf sprite(e,atom)=4 Then
    ' collision with right of screen
    current_angle=360-current_angle
  ElseIf sprite(e,atom)=8 Then
    ' collision with bottom of screen
    current_angle=((540-current_angle) Mod 360)
  Else
    ' collision with another sprite or with a corner
    current_angle = current_angle+180
  EndIf
  direction(atom)=current_angle
  vector atom,direction(atom),j,x(atom),y(atom) 'break the collision
  SPRITE show atom,x(atom),y(atom),1
  ' if the simple bounce didn't work try a random bounce
  Do While (sprite(t,atom) Or sprite(e,atom)) And j<10
    Do
      direction(atom)= Rnd*360
      vector atom,direction(atom),j,x(atom),y(atom) 'break the collision
      j=j+1
    Loop Until x(atom)>=0 And x(atom)<=MM.HRes-sprite(w,atom) And y(atom)>=0 And y(atom) <=MM.VRes-
sprite(h,atom)
    SPRITE show atom,x(atom),y(atom),1
  Loop
  ' if that didn't work then place the atom randomly
  Do While (sprite(t,atom) Or sprite(e,atom))
    direction(atom)= Rnd*360
    x(atom)=Rnd*(MM.HRes-sprite(w,atom))
    y(atom)=Rnd*(MM.VRes-sprite(h,atom))
    vector atom,direction(atom),0,x(atom),y(atom) 'break the collision
    SPRITE show atom,x(atom),y(atom),1
  Loop
End Sub
```

V5.07.00b2

100 and 144 pin
Micromite.X.production.zip

64 pin
Micromite.X.production.zip

Small changes
MM.INFO$(LINE)
LGETBYTE
MATH(CRCn....
MM.INFO(TRACK)
MM.INFO(SOUND)

Big changes

Complete re-write of the audio code. This is now the best platform for audio of all MMBasic ports

Supports via 12S,
Flac playback: PLAY FLAC_I2S fname$
mp3 playback: PLAY MP3_I2S fname$
wav playback: PLAY WAV_I2S fname$
mod file playback: PLAY MOD_I2S fname$

Supports via PWM
Flac playback: PLAY FLAC fname$
mp3 playback: PLAY MP3 fname$
wav playback: PLAY WAV fname$
mod file playback: PLAY MODFILE fname$
Tone command: PLAY TONE LFREQ, RFREQ [,DurationInMsec]
Sound command: As per PicoMite manual
SAM Text-to-speech PLAY TTS string$

Flac, wav and mp3 playback (both variants) all support playing an entire directory, just use a directory name instead of a filename. PLAY NEXT, PLAY PREVIOUS allow you to step through tracks

I2S playback is rated up to 24-bit 192KHz

MMX-144 wired for I2S playback.
Bit-clock SPI3-CLK (pin 98)
Data SPI3-OUT (pin 96)
L/R word clock SPI3-SS (pin 97)
Master clock (if required) SPI3-IN (pin 99)

MMX-100 wired for I2S playback.
Bit-clock SPI3-CLK (pin 69)
Data SPI3-OUT (pin 67)
L/R word clock SPI3-SS (pin 68)
Master clock (if required) SPI3-IN (pin 70)

MMX-64 wired for I2S playback
Bit-clock SPI3-CLK (pin 29)
Data SPI3-OUT (pin 45)
L/R word clock SPI3-SS (pin 30)
Master clock (if required) SPI3-IN (pin 11)

V5.07.00b0
100 and 144 pin
Micromite.X.production.zip

64 pin
Micromite.X.production.zip

Use LIST COMMANDS and LIST FUNCTIONS to see the complete range of what is available.

Highlights are:
Many bug fixes as found in CMM2 and PicoMite testing
Support for 16-bit parallel ILI9341, OTM8009A, and NT35510 displays (as per the ArmmiteF407
Previous command recall at the command prompt
Support for * shortcut to run programs at the command prompt
Support for MM.CMDLINE$
Support for EXECUTE, CALL, and CALL() commands and function
Huge performance improvements for more complex programs
MM.INFO(EXISTS FILE)
MM.INFO(EXISTS DIR)
Lots of other changes as per PicoMite (e.g. MID$ command functionality)
Still to-do
Major rework of audio to bring up to CMM2 standard
Major rework of file handling to bring up to CMM2 standard
Major rework of Sprites
Find a decent FFT algorithm as the Microchip IDE doesn't appear to support complex.h
Bug fixes as found