MMBasic DOS/Windows Version

User Manual

MMBasic Ver 5.05.02

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

http://geoffg.net/micromite.html

and http://mmbasic.com

Copyright

The Micromite firmware including MMBasic and this manual are Copyright 2011-2019 by Geoff Graham.

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

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

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

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

Contents

Introduction 3
Serial Communications 7
File Input/Output 8
Full Screen Editor 11
MMBasic Characteristics 13
Predefined Read Only Variables 15
Commands 16
Functions 27
Obsolete Commands and Functions 31

Introduction

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

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

This manual covers the essentials of programming for the DOS/Windows version of MMBasic but for a more detailed explanation it is recommended that you read chapters 3 and 4 of the tutorial *Getting Started with the Micromite* which can be downloaded from: http://geoffg.net/Micromite#Downloads

Limitations

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

Installing and Running DOS MMBasic

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

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

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

Developing Programs

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

To run the program you have four choices:

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

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

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

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

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

Differences to the Micromite Version of MMBasic

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

This means that the following facilities are not supported:

- LCD display panels and associated features (touch, fonts, etc).
- Any interrupts including timing interrupts.
- The communications protocols I2C, SPI and 1-wire. **However, serial is implemented.**
- Serial console and commands which operate on the serial console such as AUTOSAVE and XMODEM

DOS MMBasic has a number of extra commands and functions:

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

Double Precision Floating Point

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

Environment Variables

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

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

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

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

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

The sequence is:

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

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

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

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

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

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

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

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

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

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

In Windows the environment variables can be set using:

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

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

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

Shortcut Keys

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

F2	RUN
F3	LIST
F4	EDIT
F5	WEDIT

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

Character Set

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

Note that the character values are decimal numbers:

			33 = !	34 = "	35 = #	36 = $	37 = %	38 = &	39 = '
40 = (	41 =)	42 = *	43 = +	44 = ,	45 = -	46 = .	47 = /	48 = 0	49 = 1
50 = 2	51 = 3	52 = 4	53 = 5	54 = 6	55 = 7	56 = 8	57 = 9	58 = :	59 = ;
60 = <	61 = =	62 = >	63 = ?	64 = @	65 = A	66 = B	67 = C	68 = D	69 = E
70 = F	71 = G	72 = H	73 = I	74 = J	75 = K	76 = L	77 = M	78 = N	79 = O
80 = P	81 = Q	82 = R	83 = S	84 = T	85 = U	86 = V	87 = W	88 = X	89 = Y
90 = Z	91 = [	92 = \	93 =]	94 = ^	95 = _	96 = `	97 = a	98 = b	99 = c
100 = d	101 = e	102 = f	103 = g	104 = h	105 = i	106 = j	107 = k	108 = l	109 = m
110 = n	111 = o	112 = p	113 = q	114 = r	115 = s	116 = t	117 = u	118 = v	119 = w
120 = x	121 = y	122 = z	123 = {	124 = |	125 = }	126 = ~	127 = ⌂	128 = Ç	129 = ü
130 = é	131 = â	132 = ä	133 = à	134 = å	135 = ç	136 = ê	137 = ë	138 = è	139 = ï
140 = î	141 = ì	142 = Ä	143 = Å	144 = É	145 = æ	146 = Æ	147 = ô	148 = ö	149 = ò
150 = û	151 = ù	152 = ÿ	153 = Ö	154 = Ü	155 = ø	156 = £	157 = Ø	158 = ×	159 = ƒ
160 = á	161 = í	162 = ó	163 = ú	164 = ñ	165 = Ñ	166 = ª	167 = º	168 = ¿	169 = ®
170 = ¬	171 = ½	172 = ¼	173 = ¡	174 = «	175 = »	176 = ░	177 = ▒	178 = ▓	179 = |
180 = ┤	181 = Á	182 = Â	183 = À	184 = ©	185 = ╣	186 = ║	187 = ╗	188 = ╝	189 = ¢
190 = ¥	191 = ┐	192 = └	193 = ┴	194 = ┬	195 = ├	196 = ─	197 = ┼	198 = ã	199 = Ã
200 = ╚	201 = ╔	202 = ╩	203 = ╦	204 = ╠	205 = ═	206 = ╬	207 = ¤	208 = ð	209 = Ð
210 = Ê	211 = Ë	212 = È	213 = ı	214 = Í	215 = Î	216 = Ï	217 = ┘	218 = ┌	219 = █
220 = ▄	221 = ¦	222 = Ì	223 = ▀	224 = Ó	225 = ß	226 = Ô	227 = Ò	228 = õ	229 = Õ
230 = µ	231 = þ	232 = Þ	233 = Ú	234 = Û	235 = Ù	236 = ý	237 = Ý	238 = ¯	239 = ´
240 = -	241 = ±	242 = ‗	243 = ¾	244 = ¶	245 = §	246 = ÷	247 = ¸	248 = °	249 = ¨
250 = ·	251 = ¹	252 = ³	253 = ²	254 = ■	255 =				

Function Keys

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

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

AUTORUN.BAS

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

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

Serial Communications

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

The following is an example:

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

The OPEN Command

A serial port is opened using the command:

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

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

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

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

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

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

These include:

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

For example:

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

Examples

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

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

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

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

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

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

Reading and Writing

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

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

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

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

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

File Input/Output

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

Note that:

- Long file/directory names are supported in addition to the old 8.3 format.
- Upper/lowercase characters and spaces are allowed although the file system is not case sensitive.
- Directory paths are allowed in file/directory strings. (ie, OPEN "\dir1\dir2\file.txt" FOR …).
- Back slashes must be used in paths. Eg \dir\file.txt.
- Up to ten files can be simultaneously open.
- Except for PRINT, INPUT and LINE INPUT the # in #fnbr is optional and may be omitted.

□ OPEN fname$ FOR mode AS #fnbr
Opens a file for reading or writing. 'fname$' is the file name. 'mode' can be INPUT, OUTPUT, APPEND or RANDOM. '#fnbr' is the file number (1 to 10).

□ PRINT #fnbr, expression [[,;]expression] … etc
Outputs text to the file opened as #fnbr.

□ INPUT #fnbr, list of variables
Read a list of comma separated data into the variables specified from the file previously opened as #fnbr.

□ LINE INPUT #fnbr, variable$
Read a complete line into the string variable specified from the file previously opened as #fnbr.

□ CLOSE #fnbr [,#fnbr] …
Close the file(s) previously opened with the file number '#fnbr'.

Programs can be loaded from a file and saved. Alternatively the program can be edited from within MMBasic (it is automatically reloaded following the edit).

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

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

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

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

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

□ FILES [dname $]
Search the current or specified directory and list the files/directories found.

□ KILL fname$
Delete a file in the current directory.

□ MKDIR dname$
Make a sub directory in the current directory.

□ CHDIR dname$
Change into to the directory $dname. $dname can also be ".." (dot dot) for up one directory.

□ RMDIR dir$
Remove, or delete, the directory 'dir$'.

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

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

- INPUT$(nbr, #fnbr)
  Will return a string composed of ‘nbr’ characters read from a file previously opened for INPUT with the file number ‘#fnbr’. If less than ‘nbr’ characters are available the function will return with what it has (including an empty string if no characters are available).

- EOF(#fnbr)
  Will return true if the file opened for input as ‘#fnbr’ is positioned at the end of the file.

- LOC(#fnbr)
  For a file opened as RANDOM this will return the current position of the read/write pointer in the file.

- LOF(#fnbr)
  Will return the current length of the file in bytes.

Example of Sequential I/O

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

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

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

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

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

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

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

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

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

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

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

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

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

Random File I/O

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

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

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

```
SEEK #1, 257
```

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

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

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

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

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

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

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

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

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

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

Full Screen Editor

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

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

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

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

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

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

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

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

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

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

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

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

MMBasic Characteristics

Naming Conventions

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

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

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

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

Constants

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

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

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

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

Operators and Precedence

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

Arithmetic operators:

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

Shift operators:

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

Logical operators:

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

The operators AND, OR and XOR are integer bitwise operators. For example PRINT (3 AND 6) will output 2.

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

The NOT operator is highest in precedence so it will bind tightly to the next value. For normal use the expression to be negated should be placed in brackets. For example, IF NOT (A = 3 OR A = 8) THEN …

String operators:

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

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

Implementation Characteristics

Maximum program size is 512KB.

Maximum number of variables is 500.

Maximum length of a command line is 255 characters.

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

Maximum number of dimensions to an array is 8.

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

Maximum number of nested FOR…NEXT loops is 50.

Maximum number of nested DO…LOOP commands is 50.

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

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

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

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

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

Maximum string length is 255 characters.

Maximum line number is 65000.

Compatibility

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

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

Predefined Read Only Variables

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

MM.CMDLINE$	The DOS command line used to start MMBasic.
MM.DEVICE$	A string representing the device or platform that MMBasic is running on. Currently this variable will contain one of the following: "Maximite" on the standard Maximite and compatibles. "Colour Maximite" on the Colour Maximite and UBW32. "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "Generic PIC32" for the generic version of MMBasic on a PIC32. "Micromite" on the PIC32MX150/250 "Micromite MkII" on the PIC32MX170/270 "Micromite Plus" on the PIC32MX470 "Micromite Extreme" on the PIC32MZ series
MM.VER	The version number of the firmware as a floating point number in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number. For example version 5.03.00 will return 5.03 and version 5.03.01 will return 5.0301.
MM.ERRNO MM.ERRMSG$	If a statement caused an error which was ignored these variables will be set accordingly. MM.ERRNO is a number where non zero means that there was an error and MM.ERRMSG$ is a string representing the error message that would have normally been displayed on the console. They are reset to zero and an empty string by RUN, ON ERROR IGNORE or ON ERROR SKIP.
MM.HRES MM.VRES	The current height and width of the console window in characters (not pixels). During execution of a program the window size can be adjusted by the user and these variables will be automatically updated.

Commands

Square brackets indicate that the parameter or characters are optional.

‘ (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
? (question mark)	Shortcut for the PRINT command.
CHDIR dir$	Change the current working directory to ‘dir$’ The special entry “..” represents the parent of the current directory and “.” represents the current directory.
CLOSE [#]fnbr [,[#]fnbr] …	Close the file(s) or COM port previously opened with the file number ‘#fnbr’. The # is optional. Also see the OPEN command.
CLS	Clears all text in the console window.
CLEAR	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
COLOUR fc, bc	Sets the colours for subsequent characters written to the console window. 'fc' is the foreground colour and 'bg' is the background colour. These values can be any one of the following: 0 = Black 8 = Gray 1 = Blue 9 = Bright Blue 2 = Green 10 = Bright Green 3 = Cyan 11 = Bright Cyan 4 = Red 12 = Bright Red 5 = Purple 13 = Bright Purple 6 = Yellow 14 = Bright Yellow 7 = White 15 = Bright White This command can also be spelt as COLOR.
CONST id = expression [, id = expression] … etc	Create a constant identifier which cannot be changed once created. 'id' is the identifier which follows the same rules as for variables. The identifier can have a type suffix (!, %, or $) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.
CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.

CURSOR x, y	Positions the cursor in the console window. 'x' and 'y' are the horizontal and vertical screen coordinates (in characters). The top left hand corner has the coordinates x = 0 and y = 0. Note that the coordinates are in characters (not pixels) and if the coordinates are outside of the window area the command will be silently ignored.
DATA constant [, constant]...	Stores numerical and string constants to be accessed by READ. In general string constants should be surrounded by double quotes ("). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as THEN, WHILE, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.

DIM [type] decl [,decl]...
where 'decl' is:
var [length] [type] [init]
'var' is a variable name with optional dimensions
'length' is used to set the maximum size of the string to 'n' as in LENGTH n
'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword *AS* - as in AS FLOAT)
'init' is the value to initialise the variable and consists of:
= <expression>
For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.
 	
Examples:
DIM nbr(50)
DIM INTEGER nbr(50)
DIM name AS STRING
DIM a, b$, nbr(100), strn$(20)
DIM a(5,5,5), b(1000)
DIM strn$(200) LENGTH 20
DIM STRING strn(200) LENGTH 20
DIM a = 1234, b = 345
DIM STRING strn = "text"
DIM x%(3) = (11, 22, 33, 44)
	Declares one or more variables (ie, makes the variable name and its characteristics known to the interpreter).
	When OPTION EXPLICIT is used (as recommended) the DIM, LOCAL or STATIC commands are the only way that a variable can be created. If this option is not used then using the DIM command is optional and if not used the variable will be created automatically when first referenced.
	The type of the variable (ie, string, float or integer) can be specified in one of three ways:
	By using a type suffix (ie, !, % or $ for float, integer or string). For example:
	`DIM nbr%, amount!, name$`
	By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (ie, it does not have to be repeated). For example:
	`DIM STRING first_name, last_name, city`
	By using the Microsoft convention of using the keyword "AS" and the type keyword (ie, FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable.
	For example:
	`DIM amount AS FLOAT, name AS STRING`
	Floating point or integer variables will be set to zero when created and strings will be set to an empty string (ie, ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example:
	`DIM STRING city = "Perth", house = "Brick"`
	The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter "Defining and Using Variables" for more examples of the syntax.
	As well as declaring simple variables the DIM command will also declare arrayed variables (ie, an indexed variable with a number of dimensions). Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example:
	`DIM array(10, 20)`
	Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1.
	The above example specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each floating point number requires 4 bytes a total of 924 bytes of memory will be allocated (integers

	are different and require 8 bytes per element).
	Strings will default to allocating 255 bytes (eg, characters) of memory for each element and this can quickly use up memory. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters.
	For example: `DIM STRING s(5, 10)` will declare a string array with 66 elements consuming 16,896 bytes of memory while:
	`DIM STRING s(5, 10) LENGTH 20`
	Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.
	If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this, string arrays created with the LENGTH keyword act exactly the same as other string arrays. This keyword can also be used with non array string variables but it will not save any memory.
	In the above example you can also use the Microsoft syntax of specifying the type after the length qualifier. For example:
	`DIM s(5, 10) LENGTH 20 AS STRING`
	Arrays can also be initialised when they are declared by adding an equals symbol (=) followed by a bracketed list of values at the end of the declaration. For example:
	`DIM INTEGER nbr(4) = (22, 44, 55, 66, 88)`
	or `DIM s$(3) = ("foo", "boo", "doo", "zoo")`
	Note that the number of initialising values must match the number of elements in the array including the base value set by OPTION BASE. If a multi dimensioned array is initialised then the first dimension will be initialised first followed by the second, etc.

DO <statements> LOOP	This structure will loop forever; the EXIT DO command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like GOTO or EXIT SUB (if in a subroutine).
DO WHILE expression <statements> LOOP	Loops while "expression" is true (this is equivalent to the older WHILE-WEND loop, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, even once.
DO <statements> LOOP UNTIL expression	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is true.
EDIT	Invoke the full screen editor. On entry the cursor will be automatically positioned at the last line edited or, if there was an error when running the program, the line that caused the error. The editing keys are: Left/Right arrows — Moves the cursor within the line. Up/Down arrows — Moves the cursor up or down a line. Page Up/Down — Move up or down a page of the program. Home/End — Moves the cursor to the start or end of the line. A second Home/End will move to the start or end of the program. Delete — Delete the character over the cursor. This can be the line separator character and thus join two lines. Backspace — Delete the character before the cursor. Insert — Will switch between insert and overtype mode. Escape Key — Will close the editor without saving (confirms first).

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

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

END FUNCTION	Marks the end of a user defined function. See the FUNCTION command. Each function must have one and only one matching END FUNCTION statement. Use EXIT FUNCTION if you need to return from a function from within its body.
END SUB	Marks the end of a user defined subroutine. See the SUB command. Each sub must have one and only one matching END SUB statement. Use EXIT SUB if you need to return from a subroutine from within its body.
ERASE variable [,variable]...	Deletes variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (eg, `dat()`) or just by specifying the variable's name (eg, `dat`). Use CLEAR to delete all variables at the same time (including arrays).
ERROR [error_msg$]	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.

FILES [fspec]	Lists files. Same as the DOS DIR command. If 'fspec' is specified that will be added to the DIR command line.
FOR counter = start TO finish [STEP increment]	Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' greater than 'finish'. The 'increment' can be an integer or floating point number. Note that using a floating point fractional number for 'increment' can accumulate rounding errors in 'counter' which could cause the loop to terminate early or late. 'increment' can be negative in which case 'finish' should be less than 'start' and the loop will count downwards. See also the NEXT command.

FUNCTION xxx (arg1 [,arg2, …]) [AS <type>}
<statements>
<statements>
xxx = <return value>
END FUNCTION
	Defines a callable function. This is the same as adding a new function to MMBasic while it is running your program.
	'xxx' is the function name and it must meet the specifications for naming a variable. The type of the function can be specified by using a type suffix (ie, xxx$) or by specifying the type using AS <type> at the end of the functions definition. For example:
	`FUNCTION xxx (arg1, arg2) AS STRING`
	'arg1', 'arg2', etc are the arguments or parameters to the function. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using `AS <type> (ie, arg1 AS STRING).`
	The argument can also be another defined function or the same function if recursion is to be used (the recursion stack is limited to 1000 nested calls).
	To set the return value of the function you assign the value to the function's name. For example:
	`FUNCTION SQUARE(a)`
	`  SQUARE = a * a`
	`END FUNCTION`
	Every definition must have one END FUNCTION statement. When this is reached the function will return its value to the expression from which it was called. The command EXIT FUNCTION can be used for an early exit.
	You use the function by using its name and arguments in a program just as you would a normal MMBasic function. For example:
	`PRINT SQUARE(56.8)`
	When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function.
	Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string.
	Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference.
	You must not jump into or out of a function using commands like GOTO, GOSUB, etc. Doing so will have undefined side effects including ruining your day.

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

IF expr THEN statement or IF expr THEN stmt ELSE stmt	Evaluates the expression ‘expr' and performs the THEN statement if it is true or skips to the next line if false. The optional ELSE statement is the reverse of the THEN test. This type of IF statement is all on one line. The ‘THEN statement’ construct can be also replaced with: GOTO linenumber | label’.

IF expression THEN
<statements>
[ELSEIF expression THEN
<statements>]
[ELSE
<statements>]
ENDIF
	Multiline IF statement with optional ELSE and ELSEIF cases and ending with ENDIF. Each component is on a separate line.
	Evaluates 'expression' and performs the statement(s) following THEN if the expression is true or optionally the statement(s) following the ELSE statement if false.
	The ELSEIF statement (if present) is executed if the previous condition is false and it starts a new IF chain with further ELSE and/or ELSEIF statements as required.
	One ENDIF is used to terminate the multiline IF.

INPUT ["prompt string$";] list of variables	Allows input from the console to a list of variables. The input command will prompt with a question mark (?). The input must contain commas to separate each data item if there is more than one variable. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 If the "prompt string$" is specified it will be printed before the question mark. If the prompt string is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #fnbr, list of variables	Same as the normal INPUT command except that the input is read from a file or COM port previously opened as ‘#fnbr’. See the OPEN command.
KILL file$	Deletes the file specified by ‘file$’. If there is an extension it must be specified.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command.
LINE INPUT [prompt$,] string-variable$	Reads an entire line from the console input into ‘string-variable$’. If specified the ‘prompt$’ will be printed first. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of ‘prompt$’.
LINE INPUT #fnbr, string-variable$	Same as the LINE INPUT command except that the input is read from a file or COM port previously opened as ‘#fnbr’. See the OPEN command.
LIST or LIST ALL	List a program on the serial console. LIST on its own will list the program with a pause at every screen full. LIST ALL will list the program without pauses.
LOAD file$ [,R]	Loads a program called ‘file$’ from the current drive into program memory. If the optional suffix ,R is added the program will be immediately run without prompting. If an extension is not specified “.BAS” will be added to the file name.
LOCAL variable [, variables] See DIM for the full syntax.	Defines a list of variable names as local to the subroutine or function. This command uses exactly the same syntax as DIM and will create variables that will only be visible within the subroutine or function. They will be automatically discarded when the subroutine or function exits.
LOOP [UNTIL expression]	Terminates a program loop: see DO.
MEMORY	List the amount of memory currently in use.

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

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

OPTION BASE 0 | 1	Set the lowest value for array subscripts to either 0 or 1. This must be used before any arrays are declared and is reset to the default of 0 when a program is run.
OPTION CASE UPPER | LOWER | TITLE	Change the case used for listing command and function names when using the LIST command. The default is TITLE but the old standard of MMBasic can be restored using OPTION CASE UPPER.
OPTION DEFAULT FLOAT | INTEGER | STRING | NONE	Used to set the default type for a variable which is not explicitly defined. If OPTION DEFAULT NONE is used then all variables must have their type explicitly defined. When a program is run the default is set to FLOAT for compatibility with previous versions of MMBasic.
OPTION EXPLICIT	Placing this command at the start of a program will require that every variable be explicitly declared using the DIM command before it can be used in the program. This option is disabled by default when a program is run. If it is used it must be specified before any variables are used.
OPTION TAB 2 | 4 | 8	Set the spacing for the tab key. Default is 2.
PAUSE delay	Halt execution of the running program for 'delay' ms. Note that unlike the Micromite MMBasic a fractional number will not be recognised and will be rounded to the nearest integer.
PRINT expression [[,;]expression] … etc	Outputs text to the serial console. Multiple expressions can be used and must be separated by either a: • Comma (,) which will output the tab character • Semicolon (;) which will not output anything (it is just used to separate expressions). • Nothing or a space which will act the same as a semicolon. A semicolon (;) at the end of the expression list will suppress the automatic output of a carriage return/ newline at the end of a print statement. When printed, a number is preceded with a space if positive or a minus (-) if

	negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large floating point numbers (greater than six digits) are printed in scientific number format. The function TAB() can be used to space to a certain column and the string functions can be used to justify or otherwise format strings.
PRINT #fnbr, expression [[,;]expression] … etc	Same as the normal PRINT command except that the output is directed to a file or COM port previously opened as '#fnbr'. See the OPEN command.
QUIT	Terminate the running program, exit MMBasic and close the DOS box.
RANDOMIZE nbr	Seed the random number generator with 'nbr'. When MMBasic is started the random number generator is seeded with zero and will generate the same sequence of random numbers each time. To generate a different random sequence each time you must use a different value for 'nbr' (the TIMER function is handy for that).
READ variable[, variable]...	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. See also DATA and RESTORE.
REM string	REM allows remarks to be included in a program. Note the Microsoft use of the single quotation mark to denote remarks is also supported and is preferred.
RESTORE [line]	Resets the line and position counters for the READ statement. If 'line' is specified the counters will be reset to the beginning of the specified line. 'line' can be a line number or label. If 'line' is not specified the counters will be reset to the start of the program.
RMDIR dir$	Remove, or delete, the directory 'dir$'.
RUN [file$]	Run the program held in memory. Optionally 'file$' can be specified used and this will load and run a file (the extension .BAS is added if an extension is not specified).
SAVE file$	Save the program held in memory to the file 'file$'. The extension .BAS will be added if an extension is not specified.
SELECT CASE value CASE testexp [[, testexp] …] <statements> <statements> CASE ELSE <statements> <statements> END SELECT	Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression. 'testexp' is the value that 'exp' is to be compared against. It can be: • A single expression (ie, 34, "string" or var*5) to which it may equal • A range of values in the form of two single expressions separated by the keyword "TO" (ie, 5 TO 9 or "aa" TO "cc") • A comparison starting with the keyword "IS" (which is optional). For example: IS > 5, IS <= 10. When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true. If 'value' cannot be matched with a 'testexp' it will be automatically matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any <statements> and continue with the code following the END SELECT. When a match is made the <statements> following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT. An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT. Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT…CASE statements can be nested inside

	the CASE statements of other SELECT…CASE statements.
	Example:
	`SELECT CASE nbr%`
	`  CASE 4, 9, 22, 33 TO 88`
	`  statements`
	`  CASE IS < 4, IS > 88, 5 TO 8`
	`  statements`
	`  CASE ELSE`
	`  statements`
	`END SELECT`

SETTITLE string$	Will set the title of the console window to 'string'.
STATIC variable [, variables] See DIM for the full syntax.	Defines a list of variable names which are local to the subroutine or function. These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command). This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters. Static variables can be initialised to a value. This initialisation will take effect only on the first call to the subroutine (not on subsequent calls).
SUB xxx (arg1 [,arg2, …]) <statements> <statements> END SUB	Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program. 'xxx' is the subroutine name and it must meet the specifications for naming a variable. 'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using `AS <type> (ie, arg1 AS STRING)`. Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: `MySub a1, a2` When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.
SYSTEM command-line$	This will exit to DOS, run the 'command-line$' as if typed in at the command prompt and then return to the running MMBasic program. This can be used to access features of the DOS window that are not available from within MMBasic. For example: `SYSTEM "DIR /b > flist.txt"` will run the DOS DIR command (with brief output) and redirect its output to the file flist.txt. The MMBasic program could then open this file and read the list of files and directories listed by the DIR command. Note that 'command-line$' must be a string constant (surrounded by double quotes) or a string expression.

TIMER = msec	Resets the timer to a number of milliseconds. Normally this is just used to reset the timer to zero but you can set it to any positive integer. See the TIMER function for more details.
TRACE ON or TRACE OFF or TRACE LIST nn	TRACE ON/OFF will turn on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. TRACE LIST will list the last 'nn' lines executed in the format described above. Note that MMBasic is always logging the lines executed so this facility is always available (ie, it does not have to be turned on).
WEDIT	This will invoke the Notepad editor with the file that was last specified on the MMBasic command line, or loaded using RUN/LOAD or saved using the SAVE command. Notepad can then be used to edit the program. When Notepad is terminated MMBasic will automatically reload the file into program memory. If a file operation has not been previously used MMBasic will save the current program to a temporary file for editing. To change the editor used by MMBasic set the Windows environment variable MMEDITOR to the path of the editor to be used. This command is a convenient method of editing a program from within MMBasic. Note that at the command prompt the F5 key will trigger the WEDIT command so a program can be edited via a single keystroke.

Functions

Square brackets indicate that the parameter or characters are optional.

ACOS(number)	Returns the inverse cosine of the argument 'number' in radians.
ABS(number)	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ASC(string$)	Returns the ASCII code for the first letter in the argument ‘string$’.
ASIN(number)	Returns the inverse sine value of the argument 'number' in radians.
ATN(number)	Returns the arctangent of the argument 'number' in radians.
BIN$(number [, chars])	Returns a string giving the binary (base 2) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
CHR$(number)	Returns a one-character string consisting of the character corresponding to the ASCII code indicated by argument 'number'.
CINT(number)	Round numbers with fractional portions up or down to the next whole number or integer. For example, 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also INT() and FIX().
COS(number)	Returns the cosine of the argument 'number' in radians.
CWD$	Returns the current working directory as a string.
DATE$	Returns the current date based on the internal DOS clock as a string in the form "DD-MM-YYYY". For example, "28-07-2012".
DEG(radians)	Converts 'radians' to degrees.
EOF([#]fnbr)	Will return true if the file previously opened for INPUT with the file number ‘#fnbr’ is positioned at the end of the file. For a serial COM port it will return true if there are no characters currently waiting in the input buffer to be read. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT$ function.

EVAL(string$)	Will evaluate 'string$' as if it is a BASIC expression and return the result. 'string$' can be a constant, a variable or a string expression. The expression can use any operators, functions, variables, subroutines, etc that are known at the time of execution. The returned value will be an integer, float or string depending on the result of the evaluation. For example: S$ = "`COS(RAD(30)) * 100" : PRINT EVAL(S$)` Will display: `86.6025`
EXP(number)	Returns the exponential value of 'number'.
FIX(number)	Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point. For example 9.89 will return 9 and -2.11 will return -2. The major difference between FIX and INT is that FIX provides a true integer function (ie, does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .
HEX$(number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
INKEY$	Checks the console input buffers and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string. If the input buffer is empty this function will immediately return with an empty string (ie, "") .
INPUT$(nbr, [#]fnbr)	Will return a string composed of up to ‘nbr’ characters read from a file or COM port previously with the file number ‘#fnbr’. This function will read all characters including carriage return and new line without translation. If there are less than 'nbr' characters waiting this function will return with as many that are waiting to be read, this also means that it could return with an empty string if there are no characters waiting. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, string-pattern$)	Returns the position at which 'string-pattern$' occurs in 'string-searched$', beginning at 'start-position'. Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern$' is not found.
INT(number)	Truncate an expression to the next whole number less than or equal to the argument. For example 9.89 will return 9 and -2.11 will return -3. This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function. See also CINT() .
LEFT$(string$, nbr)	Returns a substring of ‘string$’ with ‘nbr' of characters from the left (beginning) of the string.
LEN(string$)	Returns the number of characters in 'string$'.

LOC([#]fnbr)	For a file opened as RANDOM this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. For a serial COM port this will return the number of bytes waiting in the receive queue to be read. The # is optional.
LOF([#]fnbr)	Return the current length of a file opened with the file number ‘#fnbr’ in bytes. For a serial COM port this function will return the number of characters waiting to be sent and in the current implementation this will always be zero as the serial output is unbuffered. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.
LCASE$(string$)	Returns ‘string$’ converted to lowercase characters.
MAX(arg1 [, arg2 [, …]]) or MIN(arg1 [, arg2 [, …]])	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.
MID$(string$, start) or MID$(string$, start, nbr)	Returns a substring of ‘string$’ beginning at ‘start’ and continuing for ‘nbr’ characters. The first character in the string is number 1. If ‘nbr’ is omitted the returned string will extend to the end of ‘string$’
OCT$(number [, chars])	Returns a string giving the octal (base 8) representation of 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
PI	Returns the value of pi.
RAD(degrees)	Converts 'degrees' to radians.
RIGHT$(string$, number-of-chars)	Returns a substring of ‘string$’ with ‘number-of-chars’ from the right (end) of the string.
RND(number)	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The RANDOMIZE command reseeds the random number generator.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE$(number)	Returns a string of blank spaces 'number' bytes long.
SQR(number)	Returns the square root of the argument 'number'.

STR$(number) or STR$(number, m) or STR$(number, m, n) or STR$(number, m, n, c$)	Returns a string in the decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the minus symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR$(123.456) will return "123.456" STR$(123.456, 6) will return " 123.456" STR$(123.456, -6) will return " +123.456" STR$(-123.456, 6) will return " -123.456" STR$(-123.456, 6, 5) will return " -123.45600" STR$(-123.456, 6, -5) will return " -1.23456e+02" STR$(53, 6) will return " 53" STR$(53, 6, 2) will return " 53.00" STR$(53, 6, 2, "*") will return "****53.00"
STRING$(nbr, ascii) or STRING$(nbr, string$)	Returns a string 'nbr' bytes long consisting of either the first character of string$ or the character representing the ASCII value 'ascii' which is a decimal number in the range of 32 to 126.
TAB(number)	Outputs spaces until the column indicated by 'number' has been reached.
TAN(number)	Returns the tangent of the argument 'number' in radians.
TIME$	Returns the current time based on the internal DOS clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00".
TIMER	Returns the elapsed time in milliseconds (eg, 1/1000 of a second) since reset. The timer is reset to zero when MMBasic is started and you can also reset it by using TIMER as a command. Note that under DOS the timer will reset to 0 for each subsequent 24 hour interval that elapses.
UCASE$(string$)	Returns ‘string$’ converted to uppercase characters.
VAL(string$)	Returns the numerical value of the ‘string$’. If 'string$' is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.

Obsolete Commands and Functions

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

Note that these commands may be removed in the future.

GOSUB target	Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN. New programs should use defined subroutines (ie, SUB…END SUB).
IF condition THEN linenbr	For Microsoft compatibility a GOTO is assumed if the THEN statement is followed by a number. A label is invalid in this construct. New programs should use: IF condition THEN GOTO linenbr | label
ON nbr GOTO | GOSUB target[,target, target,...]	ON either branches (GOTO) or calls a subroutine (GOSUB) based on the rounded value of 'nbr'; if it is 1, the first target is called, if 2, the second target is called, etc. Target can be a line number or a label. New programs should use SELECT CASE.
SPC(number)	This function returns a string of blank spaces 'number' bytes long. It is similar to the SPACE$() function and is only included for Microsoft compatibility.
POS	For the console, returns the current cursor position in the line in characters.
RETURN	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB.
TROFF	Turns the trace facility off; see TRON.
TRON	Turns on the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. New programs should use the TRACE command.
WHILE expression WEND	WHILE initiates a WHILE-WEND loop. The loop ends with WEND, and execution reiterates through the loop as long as the 'expression' is true. This construct is included for Microsoft compatibility. New programs should use the DO WHILE … LOOP construct.

Colour Maximite 2

User Manual

MMBasic Ver 5.05.04

Geoff Graham

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

http://geoffg.net/maximite.html

and http://mmbasic.com

About

The Colour Maximite 2 was conceived and developed by Peter Mather (matherp on the Back Shed Forum) who also led the development project.

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

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

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 Colour Maximite 2 and MMBasic are also regulars on this forum.

Copyright and Acknowledgments

The Maximite firmware and MMBasic is copyright 2011-2020 by Geoff Graham and Peter Mather 2016-2020.

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

FatFs (SD Card) driver is copyright 2014, ChaN.

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

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

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

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

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

Marcel Rodrigues wrote the GIF decoder.

Maury Quijada wrote the image resize and image rotate code.

The compiled object code (the .bin file) for the Colour Maximite 2 is free software: you can use or redistribute it as you please. The source code is available via subscription (free of charge) to individuals for personal use or under a negotiated license for commercial use. In both cases go to http://mmbasic.com for details.

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

This Manual

The author of this manual is Geoff Graham. It is distributed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0)

Contents

Introduction 4
Quick Start Tutorial 5
Hardware Features 7
Using MMBasic 12
File Manager 16
Full Screen Editor 17
Variables and Expressions 19
Subroutines and Functions 24
Basic Graphics 27
SD Card Support 32
Audio Output 36
Special Device Support 38
Game Playing Features 41
Porting Programs 43
Predefined Read Only Variables 46
Operators 48
Options 49
Commands 53
Functions 92
Obsolete Commands and Functions 106
Appendix A – Serial Communications 107
Appendix B – I2C Communications 110
Appendix C – 1-Wire Communications 112
Appendix D – SPI Communications 113
Appendix E – Sprites 115
Appendix F – Special Keyboard Keys 117
Appendix G – Loading the Firmware 118

Introduction

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

While the concept is borroed from computers of the 80's the technology used is very much up to date. The CPU that powers the Colour Maximite 2 is an ARM Cortex-M7 32-bit RISC processor running at up to 480MHz with 2MB flash memory and 1MB RAM. This processor includes its own video controller and generates a VGA output at resolutions up to 800x600 pixels and with up to 16-bit colour.

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

The basic features of the Colour Maximite 2 are:

- **Low cost affordable fun**. The firmware (including the BASIC interpreter) is completely free. The main PCB is easy to assemble with thru hole components. The CPU and support circuits are contained on a fully assembled plug in board costing US$30. The firmware can be loaded using free software so a programmer or special equipment is not required to get started.
- **Instant startup** into the BASIC interpreter. Program space is 516KB, enough for huge and complex programs (typically 25,000 lines or more) while general RAM used for arrays and buffers is over 5MB (enough for enormous arrays).
- **Full featured BASIC interpreter** with double precision floating point, 64-bit integers and string variables, long variable names, arrays of floats, integers or strings with multiple dimensions, extensive string handling and user defined subroutines and functions. Typically it will execute a program at 270,000 lines per second.
- **Rock solid VGA output** (or HDMI with an inexpensive converter). With five program selectable video resolutions from 800x600 pixels to 240x216 pixels and up to 16-bit colour (65536 colours).
- **USB Keyboard support.** The keyboard can be wireless (with a USB dongle) or wired and have US, FR, DE, or UK key mappings.
- **Stereo audio output** can play WAV, FLAC and MP3 files, computer generated music (MOD format) and robot speech and sound effects as well as generate precise sine wave tones.
- **A full screen editor** is built into the firmware. It includes advanced features such as colour coded syntax, search and copy, cut and paste to and from a clipboard. With one key press the program can be saved and run. If an error occurs another key press will return to the editor with the cursor placed on the line that caused the error.
- **Full support for SD cards** including editing and running programs on the SD card as well as opening files for reading, writing or random access. Cards up to 128GB formatted in FAT16, FAT32 or exFAT are supported and the files can also be read and written on personal computers running Windows, Linux or the Mac operating system. A graphical file manager is included in MMBasic.
- **Programs can be easily transferred** from another computer (Windows, Mac or Linux) using the SD card, XModem protocol or by streaming the program over the serial console input.
- **Extensive features for creating computer games.** These include multiple video planes, support for Blits and Sprites, the ability to create computer generated music, sound effects and computer generated speech. The Colour Maximite 2 includes full support for the Wii Nunchuk and Wii Classic games controllers.
- **Battery backed clock** will keep the correct time, even with the power disconnected.
- **Twenty eight input/output pins** with 12 capable of analog input. Built in support for an IR remote control and temperature and humidity sensors. Communications protocols include I2C, asynchronous serial, RS232, IEEE 485, SPI and 1-Wire. These can be used to communicate with many sensors (temperature, humidity, acceleration, etc) as well as for sending data to test equipment.
- **Power is 5 volts at 220mA** typical from an USB port or charger.

Quick Start Tutorial

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

Command Prompt

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

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

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

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

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

```
PRINT 2 + 2
```

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

Here are a few more things that you can try out. What you type is shown in bold and the Colour Maximite's output is shown in normal text.

Try a simple calculation:

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

See how much memory you have:

```
> MEMORY
  Flash:
     0K ( 0%) Program (0 lines)
   516K (100%) Free

  RAM:
     0K ( 0%) 0 Variables
     0K ( 0%) General
  5471K (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:

- Make sure that you have a properly formatted SD card inserted into the SD card slot.
- At the command prompt type `EDIT "hello.bas"` followed by the ENTER key.
- The editor should start up and you can enter this line: `PRINT "Hello World"`
- Press the F1 key 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 Colour Maximite 2. If you type EDIT again you will be back in the editor where you can change or add to your program.

Something More Complicated

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

For this you need to know a little more about the editor. If you have used an editor like Windows Notepad 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 Colour Maximite 2 can really go you could go back into the editor and change that line to `PAUSE 0` and then rerun the program.

For a more in-depth tutorial and a description of programming in BASIC you should download and read "*Introduction to Programming on the Colour Maximite 2*" which can be found at:

http://geoffg.net/maximite.html (scroll to the bottom of the page).

Hardware Features

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

This image provides an overview of its hardware features:

STM32H743IIT6 Cortex-M7 32-bit RISC CPU

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

There are two versions of this chip, Rev Y with a maximum speed of 400MHz and Rev V with a speed of 480MHz. MMBasic will automatically accommodate either revision and you can tell which one you have with the MM.INFO(CPUSPEED) function which will return 400000000 for the Rev Y chip and 480000000 for the Rev V chip. Other than this there is no difference when used in the Colour Maximite 2.

Waveshare CoreH743I CPU Board

This is a plug in board carrying the 32-bit ARM processor, 8MB SDRAM and various support components. The plug in board concept eliminates the need to solder the ARM CPU with its 176 pins and 0.2mm gap between pins. It also allows the whole CPU system to be easily replaced if it is suspected that the CPU has been damaged.

This module has two connectors on its top, one a micro USB the other a multi pin header. Both of these are only needed for firmware development and are not used in normal operation as a BASIC computer.

For normal operation all the jumpers on this board should be removed, the power switch should be set to "5VIN" and the BOOT CONFIG switch should be set to "Flash".

Many suppliers will be offering fully assembled units that do not use the plug in board concept. Instead the STM32 chip and its supporting components will be soldered directly to the main PCB (this makes sense for a machine assembled board and is cheaper to supply). The layout of these boards is essentially the same except that there are no jumpers as on the Waveshare board.

Firmware Upload Switch

Normally this switch should be set in the "Flash" position. When uploading new firmware it is set in the "System" position. See *Appendix G - Loading the Firmware* at the end this manual for the details.

If you have a fully assembled board without the Waveshare module this function is provided by a set of three jumper pins labelled “program” and “run”. The jumper should normally be between the common and the “run” pins but when uploading new firmware it should be in the "program" position.

Reset Switch

Used to reset the Colour Maximite 2 and start the bootup sequence as if the power had been cycled. The fully assembled board without the Waveshare module has a tactile switch for this function.

USB Keyboard Connector

This is the vertically mounted Type A USB connector. It will accept a standard USB keyboard including most that have a wireless dongle for the USB connection. Note that you cannot use a USB hub on this port or keyboards that have both a keyboard and mouse function. The latter will not work because they have a built-in USB hub to support the two different functions.

When the keyboard is connected or on startup the Colour Maximite 2 will enumerate the keyboard and if this is successful the SD card activity LED will be illuminated. On the first access of the SD card the LED will revert to its normal action (illuminated during SD card access).

The keyboard can use either the USA key mapping or the UK mapping. This detail is requested by MMBasic on the first startup of the computer and can be later changed using the OPTION USBKEYBOARD command.

Power & Serial Console Connector

This USB B connector is for power and the serial console over USB. The power requirement of the Colour Maximite 2 is 5V at 160mA to 250mA (typical). This is within the capabilities of most USB chargers however some PCs (especially older laptops) may have trouble supplying this If your Colour Maximite 2 is suffering from intermittent issues such as reboots, errors reading the SD card, etc then it would be worth changing the power supply to one a with a much higher capacity (for example, 2 amps or more).

The serial console is available if the Colour Maximite 2 is connected via this port to a personal computer. The console is used to write and debug BASIC programs and configure the computer. Normally the VGA screen and USB keyboard are used as the console but the serial console works just as well including using the File Manager and Full Screen Editor. The only thing that the serial console does not support is graphics.

When connected to your desktop computer via USB the Colour Maximite 2 will be setup as a virtual serial port over USB and appear to your computer as a normal serial port.

Windows 10 includes the required USB device driver. For other operating systems (ie, Linux, Mac, Windows 8 and earlier) go to the Microchip website https://www.microchip.com/wwwproducts/en/MCP2221A for instructions (select the Documents tab).

On a Windows computer the Colour Maximite will appear as an additional serial port in Device Manager as illustrated on the right.

You also need a terminal emulator program on your desktop computer. This program acts like an old fashioned computer terminal where it will display text received from a remote computer and any key presses will be sent to the remote computer over the serial link.

The terminal emulator that you use should support VT100 emulation as that is what the editor built into the MMBasic expects. For Windows users it is recommended that you use Tera Term as this has a good VT100 emulator and is known to work with the XModem protocol which you can use to transfer programs to and from the Colour Maximite (Tera Term can be downloaded from: http://tera-term.en.lo4d.com/).

The terminal emulator and the serial port that it is using should be set to the Colour Maximite 2 standard of 115200 baud, 8 data bits and one stop bit.

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

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

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

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

Instructions for Linux are here: http://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12171

When you have the serial port and terminal emulator setup you can reset the Colour Maximite 2 and you should see the MMBasic banner and prompt on the terminal emulator as illustrated on the right.

The USB to Serial converter chip used in the Colour Maximite 2 can be a Microchip MCP2221A. By default this chip tells the host that it only needs 100mA on the 5V pin and in some rare cases this can cause trouble (ie, failure to power up, random restarts, etc). The fix is to go to https://www.microchip.com/wwwproducts/en/MCP2221A and download the "MCP2221 Utility" (it is under the Documents tab) and use that to change the MCP2221A to request 500mA by default.

Audio Connector

This is a 3.5mm stereo phono socket. The tip is the right channel, the ring is the left channel while the sleeve is ground. The signal level at full volume is about 1V RMS (approx 3V peak to peak). The output is high impedance suitable for feeding into an amplifier. **It cannot directly drive a loudspeaker, headphones or any low impedance load and might be damaged if that was attempted.**

MMBasic can generate audio in many formats ranging from simple sine wave tones through to playing WAV, FLAC, and MP3 files.

Temperature Sensor

These solder pads are for an optional Dallas DS18B20 temperature sensor. This can be mounted so that the sensor will protrude through a hole in the rear panel to measure the ambient temperature.

Support for the DS18B20 is built into MMBasic – see the section *Special Device Support* in this manual for the details. The signal line for this sensor is pin 42. If this sensor is installed the associated pullup resistor (4.7KΩ) on the motherboard must also be installed.

VGA Monitor Connector

This is the main video output and it generates standard VGA signals in a variety of resolutions and number of colours as determined by the MODE command. These range from 800x600 pixels (the default at power up) to 240x216 pixels and 256, 4096 or 65536 colours. All modes work perfectly with monitors that have an aspect ratio of 4:3 or widescreen monitors that can switch to that ratio (most widescreen monitors will do this automatically).

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

If an HDMI output is required it is recommended that an inexpensive VGA to HDMI converter be used. These cost about US$10+ on eBay and will also encode the audio from the computer so that it can play through the monitor's speakers. As an example, the Colour Maximite 2 was successfully tested with this converter: https://tinyurl.com/w5woobe

External I/O connector

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

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

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

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

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

The capabilities of each pin are:

Pin	Features	
1	3.3 Volt Power	
3	I^2C SDA	5V
5	I^2C SCK	5V
7	Analog Input or COUNT 1	
9	Ground	
11	COM2: RX	5V
13	Analog Input or COUNT 2	
15	Analog Input or COUNT 3	
17	3.3 Volt Power	
19	SPI MOSI	5V
21	SPI MISO	5V
23	SPI CLOCK	5V
25	Ground	
27	I^2C2 SDA	5V
29	Analog Input or PWM-1C	
31	PWM 2B	5V
33	General I/O	5V
35	SPI2 MISO	5V
37	Analog Input or COM1 DE	
39	Ground	

Pin	Features	
2	5.0 Volt Power	
4	5.0 Volt Power	
6	Ground	
8	Analog Input or COM1: TX	
10	Analog Input or COM1: RX	
12	Analog Input or PWM 1A	
14	Ground	
16	Analog Input or COM2: TX	
18	FAST COUNT	5V
20	Ground	
22	Analog Input or PWM 1B	
24	Analog Input or COUNT 4	
26	Analog Input	
28	I^2C2 SCK	5V
30	Ground	
32	General I/O	5V
34	Ground	
36	PWM 2A	5V
38	SPI2 MOSI	5V
40	SPI2 CLOCK	5V

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

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

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

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

Both the data line (SDA) and clock (SCL) for both I^2C ports have 10K pullup resistors (to 3.3V) installed on the motherboard so external pullup resistors are not required. Note that these may interfere with the operation of these pins if they are used as general purpose inputs.

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

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

IR Receiver

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

Wii Nunchuk and Wii Classic Connector

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

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

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

The Wii connector can be used for other purposes. The signal lines can be accessed as pins 43 and 44 or as I^2C3 SDA and I^2C3 SCK respectively. This is useful for interfacing to other devices that use I^2C and have the same connector. Pin 44 also supports analog input. However note that both pins 43 and 44 have 10K pullup resistors (to 3.3V) mounted on the motherboard and these may interfere with their use as general purpose I/O pins.

Power & SD Card Activity LEDs

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

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

SD Card Connector

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

The Colour Maximite 2 will support cards up to 128GB. Cards larger than 32GB should be formatted as exFAT and cards 32GB or less formatted as FAT32. Small cards may not be reliable so the recommended size is 8GB formatted as FAT32.

Backup Battery

The CR1220 coin cell battery on the motherboard keeps the internal ARM STM32 real time clock running while the power is off and also keeps a bank of 4KB RAM alive at the same time. The real time clock is used to provide the correct time to MMBasic on startup and the battery backed RAM is used to store saved variables and options. The life of this battery life will be about 3 to 4 years of normal use.

Using MMBasic

Commands and Program Input

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

To enter a program the easiest method is to use the EDIT command. This will invoke the full screen editor which is built into the Maximite 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 via the XModem protocol (see the XMODEM command) or by streaming it up the console serial link (see the AUTOSAVE command) or by saving it to an SD card and transferring that to the Colour Maximite 2.

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

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

Editing the Command Line

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

Console Keyboard/Display

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

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

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

Keyboard Shortcuts

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

F1	FILES
F2	RUN
F3	LIST
F4	EDIT

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

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

Function keys F11 and F12 can be programmed with custom text:

F11	*User specified string – See the* OPTION F11 *Command.*
F12	*User specified string – See the* OPTION F12 *Command.*

Line Numbers and Program Structure

The structure of a program line is:

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

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

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

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

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

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

All Programs Are Run From the SD Card

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

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

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

The main commands used to manage a program are:

RUN "*prog*"	Run the program called *prog* located on the SD card.
LIST "*prog*"	List the program called *prog* on the console screen. This will pause every screenfull 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 flash 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 Colour Maximite 2 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.

Status Line

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

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

Running Programs

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

The running program is 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. The Colour Maximite 2 has a battery backed clock so it will not loose the time even when powered off. If you find that the time drifts while the power is off you can use the OPTION RTC CALIBRATE command to correct for any inaccuracies.

You can freeze program execution for a number of milliseconds using PAUSE. MMBasic also maintains an internal stopwatch function (the TIMER function) which counts up in 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 Colour Maximite 2 without a VGA monitor and also have nothing connected to the serial console. With OPTION AUTORUN ON set the program will run automatically on power up without human intervention.

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

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

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

PIN Security

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

Because an intruder cannot reach the command prompt they cannot list or copy the program held in the program memory, they cannot change the program or change any aspect of MMBasic. Once set the PIN can only be removed by providing the correct PIN as set in the first place. If the number is lost the only method of recovery is to reset MMBasic as described below (which will erase the program).

Note that this is not a complete protection as it is possible to connect a debugger to the STM32 CPU and access all areas of memory using that facility but it does prevent casual access.

The Serial Console

The default settings for the serial over USB console are 115200 baud, 8 bits, no parity and one stop bit. Using the OPTION BAUDRATE command the baud rate of the serial console can be changed to any other speed. Changing the console baud rate to a higher speed makes the full screen editor faster in redrawing the screen.

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 accidently set the baud rate to an invalid speed and in that case the only recovery is to connect a VGA monitor and USB keyboard to change the baudrate or reset MMBasic as described below.

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

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

Resetting MMBasic

MMBasic can be reset to its original configuration by placing a link between pins 39 and 40 on the rear I/O connector while power is applied. This will result in the program memory and saved variables being completely erased and all options (security PIN, console baud rate, etc) reset to their initial defaults.

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

File Manager

The file manager is an easy way to manage files and directories on the SD card. Using this you can search, delete, rename, run, etc. The file manager will work with both the VGA video output and the serial console.

To run the file manager you use the command FILES at the command prompt or use the F1 key on your keyboard. This will start up in the current directory and list the files and directories there.

On the VGA monitor it will look like this:

```
<DIR>                        SPRITES
<DIR>                        SPRTDEMO
<DIR>                        SPRTEST
<DIR>                        T
<DIR>                        TESTSPR
<DIR>                        TFT
14:10 30-06-2013       573   AUTORUN.BAS
10:38 17-01-2020        84   CIRCLES.BAS
08:32 18-01-2020      5573   CLOCK.BAS
20:40 16-08-2012      5482   CMM4_TST.BAS
07:59 08-07-2012       663   COLOUR-2.BAS
13:40 26-08-2012       821   COLOUR-3.BAS
14:14 05-01-2013      1976   COLOUR-3.SPR
06:55 07-07-2012      1022   COLOUR1.BAS
13:39 27-06-2013     11878   FILEMAN2.BAS
18:37 20-02-2012       804   FUN1WIRE.BAS
18:37 20-02-2012       905   GETTEMP.BAS
10:11 23-09-2013      4081   GPS-SIM.BAS
06:55 07-07-2012       774   GRAPH-B.BAS
06:55 07-07-2012      3149   GRAPH-C.BAS
10:46 03-02-2011      3093   GRAPH.BAS
22:24 21-04-2015      4354   HAMURABI.BAS
13:40 26-08-2012      1101   JULIA.BAS
06:00 10-01-2012       778   KEYBOARD.BAS
06:45 11-08-2012      1133   MANDBALT.BAS
17:09 23-05-2019      2792   MODEDEMO.BAS
13:40 26-08-2012      1763   PLAYMOD.BAS
H:/CMM1/GRAPH.BAS                                   Name  42/63
ESC=Exit, F2=Run, F3=List, F4=Edit, F5=MkDir, ^C=Copy, ^F=Find, ^K=Del, ^R=Rename, ^S=Sort
```

To move around the list of files you use the arrow keys, Page Up or Page Down keys and the Home or End keys. Pressing Enter when positioned on a directory will take you into that directory and if it is positioned on a program it will run that program.

The Escape key (ESC) will exit the file manager.

At the bottom of the screen the status line lists details such as the current cursor position and the functions that are available. All these operate on the file currently selected by the cursor which is also displayed to the left of the status line:

Enter	This is the action key. If the file is a program this will RUN the program. If the file is an audio file this will PLAY the file on the sound output. If the file is a picture file the picture will be displayed – press any key to return to the file manager. If the cursor is positioned on a directory that directory will be entered. If the directory has the name ".." this will take you up one level in the directory hierarchy.
F3	This will LIST the program or text file selected.
F4	This will EDIT the program or text file selected.
F5	Will prompt for a directory name and create that directory.
CTRL-C	Will prompt for a file name and copy the selected file to that new name.
CTRL-F	Will enter the **search mode**. You will be prompted for the search text and as you type this in the cursor will automatically be positioned at the first matching file found. You can then use the down arrow key to search for the next occurrence or the up arrow key for the previous occurrence. The Enter key will leave the cursor where it is and return to normal mode. Escape will abort the search.
CTRL-K	Will delete a file or directory. A directory must be empty otherwise it will be ignored.
CTRL-R	Will rename a file or directory.
CTRL-S	This will toggle the sort order between name, size, type and date. The current sort order is displayed on the right hand side of the status line.

Full Screen Editor

An important productivity feature is the built-in full screen editor. This will work with both the VGA 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:

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.

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.

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. The full selection of commands is:

ESC	This will cause the editor to abandon all changes and return to the command prompt with the file unchanged. If you have changed the text you will be asked to press ESC twice more to confirm this action.
F1	This will save the file and return to the command prompt.
F2	This will save the file and immediately run it.
F3 or CTRL-F	This will enter the **find mode**. You will be prompted for the search text and as you type this in the editor will automatically position the cursor at the first text found. You can then use the down arrow key to search for the next occurrence or the up arrow key for the previous occurrence. The Enter key will leave the cursor at this position and return to normal editing mode. F5 or CTRL-V will replace the searched text with whatever is in the clipboard (see below). Escape will abort the search.

F4 or CTRL-S	This will enter the **select mode**. In this mode you can use the arrow keys, HOME or END to select text and copy it to the clipboard. It will be highlighted on the screen as you select it. Then F5 or CTRL-C will copy the selection to the clipboard, F4 or CTRL-X will copy and delete the selection. DELETE will simply delete the selection and ESCAPE will return to the normal editing mode without changing anything. Note: you can also enter selection mode when using a USB keyboard by holding the shift key and pressing right-arrow or down-arrow.
F5 or CTRL-V	This will insert (at the current cursor position) the text that had been previously cut or copied in the select mode (see above).
F6	This will save the edited text and exit the editor similar to the F1 key. The difference is that F6 will not update the "current program name" which is used when the RUN, LIST and EDIT commands are entered without specifying a filename.
CTRL-K	Will delete all text from the current cursor position to the end of the line.
CTRL-W	Will allow you to save a backup copy of the edited file to a different file. The editor will continue to edit the original file.
F7	Will prompt for a file name and will insert the text from that file into the editor at the current cursor position.
F8 or CTRL-B	Will prompt for a file name and will write to the file the contents of the text that had been previously cut or copied in the select mode (see above).

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.

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 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.

Serial Console

The editor will also automatically work with the serial console (including colour coding).

To operate properly this feature requires a terminal emulator that can interpret the appropriate VT100 escape codes and respond correctly. It works correctly with Tera Term however Putty needs its default background colour to be changed to white (Settings >> Colours >> Default Background >> Modify).

Colour coding the editor's output requires many extra characters to be sent to the terminal emulator and this can slow down the screen update. If necessary colour coding can be turned off or the baud rate can be set to a higher speed for faster updates (see OPTION BAUDRATE).

Memory Usage

To provide long line editing and sideways scrolling the editor needs more RAM than would be implied by the number of characters being edited. This only has an effect on very large programs (ie, greater than approx 9,000 lines) and can cause the editor to generate an out of memory error when loading.

The solution is to break the program into chunks with the relatively static portions included in the main program using the #INCLUDE command. These included files are inserted when the program is loaded via the RUN command and have no affect on performance or the overall amount of program memory used.

Variables and Expressions

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

Variables

Variables can start with an alphabetic character or underscore and can contain any alphabetic or numeric character, the period (.) and the underscore (_). They may be up to 32 characters long.

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

MMBasic supports three different types of variables:

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

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

Constants

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

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

When a constant number is used it will be assumed that it is an integer if a decimal point or exponent is not used. For example, 1234 will be interpreted as an integer while 1234.0 will be interpreted as a floating point number.

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

OPTION DEFAULT

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

```
Nbr = 1234
```

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

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

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

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

OPTION EXPLICIT

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

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

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

DIM and LOCAL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or

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

STATIC

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

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

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

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

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

As with DIM and LOCAL the variables created with STATIC can be float, integers or strings and arrays of these with or without initialisation.

CONST

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

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

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

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

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

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

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

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

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

Expressions and Operators

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

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

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

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

Arithmetic operators:

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

Shift operators:

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

Logical operators:

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

The operators AND, OR and XOR are integer bitwise operators. For example PRINT (3 AND 6) will output 2.

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

The NOT operator 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".

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 return a floating point number. For integer division you should use the integer division operator "\".

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 Colour Maximite 2 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 +, -, etc 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.

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"

Basic Graphics

There are ten basic drawing commands that you can use within MMBasic to draw images on the VGA monitor (none of these apply to the serial console).

There are also a number of advanced commands designed for programmers writing games (such as MODE, 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.

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

Read Only Variables

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

- MM. HRES
  Returns the width of the display (the X axis) in pixels.
- MM. VRES
  Returns the height of the display (the Y axis) in pixels.
- MM.INFO(FONTHEIGHT)
  Returns the height of the current font (in pixels). All characters in a font have the same height.
- MM.INFO(FONTWIDTH)
  Returns the width of a character in the current font (in pixels). All characters in a font have the same width.
- MM.INFO(HPOS)
  Returns the X coordinate of the text cursor (ie, the horizontal location (in pixels) of where the next character will be printed on the VGA monitor)
- MM.INFO(VPOS)
  Returns the Y coordinate of the text cursor (ie, the vertical location (in pixels) of where the next character will be printed on the VGA monitor)

Colours

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

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

MMBasic will automatically convert colours to the format required for the current colour depth set by the MODE command. So, for example, at startup the Colour Maximite 2's VGA output defaults to 8-bit colour and in this case the 16777215 colours that can be represented by a 24-bit colour specification will be translated as best as possible to the 256 colours supported by the 8-bit colour mode. Other VGA display modes can support up to 16-bit colour (65535 different colours) and in that case MMBasic will be able to represent the same colours with less loss of information.

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

The COLOUR command takes the format:

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

Fonts

There are seven built in fonts. These are:

Font Number	Size (width x height)	Character Set	Description
1	8 x 12	All 95 ASCII characters plus 7F to FF (hex)	Standard font (default on startup). Default font for the editor
2	12 x 20	All 95 ASCII characters	Medium sized font
3	16 x 24	All 95 ASCII characters	A larger font useful for the 800 x 600 display mode
4	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 (°).

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

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

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

As an example, the following is a tiny (6x4 pixel) font that is useful in the 320x200 display mode:

```
DefineFont #8
 60200604
 44000000 00A04040 A0AEAE00 82406C6C EACC2048 00004460 84204424 E4A48044
 00E404A0 00800400 040000E0 00480240 4CE0AAEA 48C24044 C062C2E0 E820E2AA
 EA68E0E2 8048E2E0 EAE0EAEA 0404C0E2 80040400 0E208424 2484000E 4040E280
 4A60E84A CACAA0EA 608868C0 E8C0AACA E8E8E0E8 60EA6880 E4A0EAAA 2A22E044
 A0CAAA40 AEE08888 EEAEA0EA 40AA4AA0 4A80C8CA ECCA60AE C04268A0 AA4044E4
```

```
  A4AA60AA A0EEAA40 AAA04AAA 48E24044 E088E8E0 E2004208 004AE022 F0000000
  0C000084 AA8CE06A 608806C0 0660AA26 E42460AC 24AE0640 40A0CA88 22204044
  A0CC8AA4 0EE044C4 AA0CA0EE 40AA04A0 06C8AA0C 880662AA C0C60680 0A60444E
  AE0A60AA E0AE0A40 0AA0440A 6C0E24A6 608464E0 C4400444 006CC024 E0EEEE00
End DefineFont
```

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

Drawing Commands

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

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

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

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

The drawing commands are:

- `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.

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 display panel and centred vertically:

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

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

Transparent Text

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

Displaying Images

Using the LOAD command you can load an image from the SD card 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).

Advanced Graphics Tutorial

The graphics subsystem in the Colour Maximite 2 has many advanced features such as multiple video pages, layered images, sprites and sophisticated methods of manipulating images. Peter Mather, who developed the graphics subsystem, has written a tutorial for games developers and advanced users.

This is presented as a series of posts on the Back Shed Forum and is recommended reading for users who need to get into the details: https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12125 . A PDF version is included in the Colour Maximite 2 firmware zip file.

SD Card Support

The Colour Maximite 2 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. Small capacity cards may not be reliable so 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").
- 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.

There are 33 commands and functions related to the SD card. See the Commands/Functions section later in this manual for their full description):

Program Management Commands

All programs reside on the SD card and so it must be present when running, editing and listing programs. This is different from the original Maximite where the SD card was not necessarily required.

- 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.

- DIR$(fspec, type)
  Will search an SD card for files and return the names of entries found.

- 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$'.

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 Colour Maximite 2 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 (located on the rear panel). The STM32 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 Colour Maximite 2 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.

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.

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.

Special Device Support

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

Infrared Remote Control Decoder

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

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

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

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

To setup the decoder you use the command:

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

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

This is an example of using the IR decoder:

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

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

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

Infrared Remote Control Transmitter

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

The circuit 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 Colour Maximite 2's mother board for a TO-92 version of this sensor and this can be mounted so that the body of the sensor pokes out of a hole in the back panel to sense the ambient temperature. If this is installed the associated pullup resistor (4.7KΩ) 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 DHT22 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 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 DHD22 command with three arguments as follows:

```
DHD22 pin, tVar, hVar
```

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).

For example:

```
DIM FLOAT temp, humidity
DHD22 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.

WS2812 Support

The Colour Maximite 2 has built in support for the WS2812 multicolour LED chip. This chip needs a very specific timing to work properly and with the 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.

Game Playing Features

The Colour Maximite 2 has many features designed to help programmers in writing computer games. One of the most important features of this computer is its speed (approx 10 times faster than the original Colour Maximite) and its large program memory. These alone make it possible to write large and complex programs without the issues of optimising the program for speed or space.

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

VGA Resolution, Colour Depth and Pages

The video output to the VGA monitor is controlled by the MODE command. With this you can select various resolutions from 800x600 pixels (the default on startup) to 240x216 and various colour depths from 16-bits to 8-bits. For the 8-bit colour mode each individual colour can be specified from a 16-bit pallet using the MAP command.

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

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

When drawing to the main page being displayed on the monitor the programmer can use the GETSCANLINE function to report on the line that is currently being drawn on the VGA monitor. Using this to time updates to the screen can avoid screen glitches caused by updates while the screen is being updated. A similar outcome can be achieved using an optional feature of the MODE command which will call a user defined subroutine at the start of the VGA frame blanking.

This and other features of the graphics subsystem are explained in detail in a tutorial presented on the Back Shed forum: https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12125 . A PDF version is included in the Colour Maximite 2 firmware zip file.

Scrolling and Sprites

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

The Colour Maximite 2 has extensive support for sprites which is an image that can be moved over the background without disturbing the background. This feature is far more sophisticated than that available on the original Colour Maximite. The sprite can be a PNG image or an image defined in a text file and can be of any size up to the video horizontal and vertical resolution. The transparency colour can be either black or defined in the image depending on the colour depth specified by the MODE command.

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

Displaying Images

The Colour Maximite 2 can load and display images stored on the SD card in a variety of formats including BMP, GIF, PNG and JPG. These are read from the SD card by the LOAD command and can be of any size and positioned anywhere on the screen. In addition on-screen images can be scaled and rotated under control of the program.

Fonts

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

Playing Audio

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

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

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

Keyboard Keys

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

Wii Nunchuk Support

The Nunchuk (also spelt Nunchuck) is an inexpensive games controller which includes an accelerometer, a joystick and two buttons. MMBasic on the Colour Maximite 2 provides full support for up to three Nunchuk controllers including querying the state of all user inputs and the calibration constants saved in the Nunchuk.

The Nunchuk is opened with the WII NUNCHUK OPEN command and its status is queried using the NUNCHUK() function.

Wii Classic Support

The Wii Classic is an inexpensive games controller which includes two joysticks and many buttons. MMBasic on the Colour Maximite 2 provides full support for up to three Wii Classic controllers including querying the state of all user inputs and triggering an interrupt on all or any programmer specified selection of buttons.

The Classic is opened with the WII CLASSIC OPEN command and its status is queried using the CLASSIC() function.

Porting Programs

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

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

Variables

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

This is not allowed in the Colour Maximite 2, variables must be unique regardless of their type.

Floating Point

In the Colour Maximite 2 floating point is double precision. This means that if a number is printed without formatting it will contain more significant digits than the same number on the original Colour Maximite. Generally this will not cause an issue but it might mess up numbers that need to be printed in neat columns.

Graphic Commands

The syntax of the graphic commands PIXEL, LINE, BOX and CIRCLE have completely changed. However, using the command OPTION LEGACY ON you can change these commands back to the same syntax as used in the original Colour Maximite. This option is not saved so it needs to be placed in the program before any graphic commands are used.

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

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

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

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

The syntax of the MODE command has changed.

The Scan Line Colour Override function is not supported in the Colour Maximite 2.

Fonts

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

The Colour Maximite 2 has the ability to load fonts using the LOAD FONT command and you can convert the original Colour Maximite's font files to this format using the program FontTweak from: https://www.c-com.com.au/MMedit.htm

BLIT

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

Sprites

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

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

This is compatible with the original Colour Maximite however the Colour Maximite 2 has a few improvements:

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

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

```
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           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
```

SOUND and TONE

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

SD Card and File Related Commands

There is no drive B: in the Colour Maximite 2. The SD card is drive A:.

The command CHAIN is not implemented. The Colour Maximite 2 has a very large program memory space so this feature is not necessary. The library and the LIBRARY commands are not implemented and not needed on the Colour Maximite 2. Instead of using these commands you can use #INCLUDE to insert files into the program at run time.

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

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

Special Devices

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

CONFIG Commands

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

Error Handling

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

Random Number Generator

The Colour Maximite 2 has an advanced random number generator based on an analog circuit that generates a sequence of true random numbers which will never repeat. For this reason the CMM2 does not require (or allow) the programmer to seed the random generator to get different sequences.

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

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

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

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. "Colour Maximite 2" on the **Colour Maximite 2.** "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "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^2C 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$()	These two versions can be used interchangeably but good programming practice would require that you use the one corresponding to the returned datatype.
MM.INFO$(AUTORUN)	Returns “ON” or “OFF” depending on the status of OPTION AUTORUN
MM.INFO$(CPUSPEED)	Returns the CPU speed as a string. This will be 400000000 for the Y version of the STM32H743II or 480000000 for the V version.
MM.INFO$(CURRENT)	Returns the name of the current program or NONE if called after a NEW command
MM.INFO(DISK SIZE)	Returns the capacity of the SD card in bytes
MM.INFO(FILESIZE file$)	Returns the size of ‘file$’ in bytes. Returns -1 if the file is not found.
MM.INFO(FONTHEIGHT) MM.INFO(FONTWIDTH)	Integers representing the height and width of the current font (in pixels).
MM.INFO(FRAMEH)	Returns the horizontal size of the frame buffer in pixels.

MM.INFO(FRAMEV)	Returns the vertical size of the framebuffer in pixels
MM.INFO(FREE SPACE)	Returns the number of free bytes on the SD card
MM.INFO(HPOS) MM.INFO(VPOS)	The current horizontal and vertical position (in pixels) following the last graphics or print command.
MM.INFO$(KEYBOARD)	Returns the string CONNECTED if a USB keyboard is connected and working. Otherwise returns “NOT CONNECTED”
MM.INFO(MAX PAGES)	Returns the maximum page number that can be selected in the current graphics mode.
MM.INFO(MODE)	Returns the video mode as a floating point number e.g. 1.8, 2.16, etc.
MM.INFO$(MODIFIED file$)	Returns the date/time that ‘file$’ was last modified. Returns an empty string if the file is not found.
MM.INFO(PAGE ADDRESS n)	Returns the physical memory location of page ‘n’. This is useful if you need to POKE/PEEK the contents of the page.
MM.INFO$(PIN pinno)	Returns the status of I/O pin 'pinno'. Valid returns are: INVALID, RESERVED, IN USE, and UNUSED
MM.INF(PROGRAM)	Returns the address in memory of the start of the program
MM.INFO$(SDCARD)	Returns the status of the SD card. Valid returns are: DISABLED, NOT PRESENT, READY, and UNUSED
MM.INFO$(SEARCH PATH)	Returns the string set as the search path by OPTION SEARCH PATH
MM.INFO$(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.INFO(VERSION)	The version number of the firmware as a floating point number in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number. For example version 5.03.00 will return 5.03 and version 5.03.01 will return 5.0301.
MM.INFO(WRITE PAGE)	Returns the address in memory of the page to which writes will take place
MM.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 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.
<> < > <= =< >= =>	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 Colour Maximite 2 is restarted. Options that are not permanent will be reset on startup.

	Permanent?	
OPTION AUTORUN OFF | ON	✓	Instruct MMBasic to automatically run the program stored in program memory when it starts up or is restarted by the WATCHDOG command. 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 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 the VGA 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 CMM2 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	✓	Sets the font to be used in the editor and file manager. The default is NORMAL which is a 8x12 pixel font.
OPTION FLASH [n]	✓	Specifies that programs are to be run from FLASH memory and will therefore be preserved when the CMM2 is powered off or reset. The optional parameter "n" specifies the starting point in 128Kbyte pages of memory. n defaults to 0. For 0<=n<=4 the maximum program size is 512KBytes for n=5 the maximum program size is 384KBytes for n=6 the maximum program size is 256KBytes for n=7 the maximum program size is 128KBytes Using the OPTION FLASH command can increase flash lifespan by a factor of 8 although it is unlikely ever to be needed for most users
OPTION F11 string$	✓	Define the string that will be generated when the F11 function key is pressed at the command prompt. Example: OPTION F12 "RUN "+chr$(34)+"myprog" +chr$(34)+chr$(13)+chr$(10). The maximum string length is 23 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 23 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 LEGACY ON or OPTION LEGACY OFF		This will turn on or off compatibility mode with the original Colour Maximite. Commands such as LINE, CIRCLE and PIXEL work as they originally did and all drawing commands will accept colours in the range of 0 to 7. Notes: • Keywords such as RED, BLUE, etc are not implemented so they should be defined as constants if needed. • The VGA display mode is not changed with this option so it should be specifically set to MODE 4 for emulating the 480x432 resolution of the original Colour Maximite or MODE 5 for emulating the 240x216 resolution. The colour depth must be set to 8-bits.
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 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 Colour Maximite 2 firmware (as described in the section *Resetting MMBasic*).
OPTION RAM	✓	OPTION RAM causes the program to be loaded into RAM to run rather than flash memory. This makes loading somewhat faster and avoids impacting the flash write life. Program performance is similar to running from flash (less than 1% slower). Programs are lost with power, reset, or option change. Variable storage is reduced by 512Kb
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 Colour Maximite 2. 'n' is a number between -511 and + 512. A change of ±1 should equate to about 0.0824 seconds per day. Negative numbers will slow the clock down, positive will speed it up (different from the Micromite). This setting is remembered even after a firmware upgrade.
OPTION SEARCH PATH pathname$	✓	This defines a path which will be searched when you use the existing RUN command or the short form RUN command (*) if the file does not exist in the current directory
OPTION 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.
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.
OPTION TAB 2 | 4 | 8	✓	Set the spacing for the tab key. Default is 2.
OPTION USBKEYBOARD nn	✓	Set the language type for the attached USB keyboard. ‘nn is a two character code defining the keyboard layout. The choices are US for the standard keyboard layout in the USA, Australia and New Zealand and UK for the United Kingdom, DE for Germany, and FR for France. This command can only be run from the command line and will cause a reboot of the CMM2.
OPTION VCC voltage		Specifies the voltage (Vcc) supplied to the STM32 chip. When using the analog inputs to measure voltage the STM32 chip uses its supply voltage (Vcc) as its reference. This voltage can be accurately measured using a DMM and configured using this command to allow for a 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.
? (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$
#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.
ADC	The ADC functionality can capture up to 3 channels of analog data in the background at up to 500KHz per channel (480KHz for 400MHz processors) with user selectable triggering
ADC OPEN frequency, channel1-pin [,channel2-pin] [,channel3-pin] [, interrupt]	Open the ADC channels. "frequency" is the sampling frequency in Hz. Above 160KHz the conversion is 8-bits per channel From 40KHz to 160KHz the conversion is 10-bits per channel From 20KHz to 40KHz the conversion is 12-bits per channel From 10KHz to 20KHz the conversion is 14-bits per channel Below 10KHz conversion is 16-bits per channel This is automatically applied in the firmware. 'channel1-pin' can be one of 7,10,16,22,24,37 'channel2-pin' can be one of 8,12,26,29 'channel3-pin' can be one of 13,15 'interrupt' is a normal MMBasic subroutine that will be called when the conversion completes.
ADC FREQUENCY frequency	Allows the ADC frequency to be adjusted after the ADC START command. This command is only valid if the number of bits calculated in the table above does not change.

ADC TRIGGER channel, level	Sets up triggering of the ADC. This should be specified before the ADC START command. The 'channel' can be a number between one and three depending on the number of pins specified in the ADC OPEN command. The 'level' can be between -VCC and VCC. A positive number indicates that the trigger will be on a positive going transition through the specified voltage. A negative number indicates a negative going transition through the specified voltage.
ADC START channel1array!() [,channel2array!()] [,channel3array!()]	Starts ADC conversion. The floating point arrays must be the same size and their size will determine the number of samples. Once the start command is issued the ADC(s) will start converting the input signals into the arrays at the frequency specified. If the OPEN command includes an interrupt then the command will be non-blocking. If an interrupt is not specified the command will be blocking. The samples are returned as floating point values between 0 and VCC.
ADC CLOSE	Closes the ADC and returns the pins to normal use
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 BITSTREAM pinno, n_transitions, array%()	This command is used to generate an extremely accurate bit sequence on the pin specified. The pin must have previously been set up as an output and set to the required starting level. The array contains the length of each level in the bitstream in microseconds. The maximum period allowed is 65.5 mSec The first transition will occur immediately on executing the command. The last period in the array is ignored other than defining the time before control returns to the program or command line.

	The pin is left in the starting state if the number of transitions is even and the opposite state if the number of transitions is odd.
BITBANG WS2812 type, pin, colours%()	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 The 'colours%()' 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 (ie, 0 to &HFFFFFF).
BLIT READ [#]b, x, y, w, h [,pagenumber] or BLIT WRITE [#]b, x, y 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. BLIT CLOSE will close the memory buffer '#b' to allow it to be used for another BLIT READ operation and recover the memory used. Notes: • Sixty four buffers are available ranging from #1 to #64. • When specifying the buffer number the # symbol is optional. • All other arguments are in pixels.
BLIT x1, y1, x2, y2, w, h [, page] [,orientation]	Copy one section of the display screen to another part of the display. The source coordinate is 'x1' and 'y1'. The destination coordinate is 'x2' and 'y2'. The width of the screen area to copy is 'w' and the height is 'h'. 'page' is the page number that the image data is read from; it is then written to the current write page as specified by the PAGE WRITE n command. If 'page' is omitted the data is read from the write page. Set the page to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command. All arguments are in pixels and the source and destination can overlap. The optional 'orientation' parameter specifies how the section of the screen is changed as it is copied. It is the bitwise AND of the following values: &B001 = mirrored left to right &B010 = mirrored top to bottom &B100 = don't copy transparent pixels
BOX x, y, w, h [, lw] [,c] [,fill]	Draws a box on the 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.
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.
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 program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with graphics, nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.
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 must be the offset (in 32-bit words) 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 loaded into flash by the RUN command. 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.
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.

DAC n, voltage	Sets the DAC channel (1 or 2) to the voltage requested. This command cannot be used if the DACs are in use for audio output.
DAC START frequency, DAC1array%() [,DAC2array%()]	Sets up the DAC to create an arbitrary waveform. DAC1array%() and optional DAC2array%() should contain numbers in the range 0-4095 to suit the 12-bit DACs. Once started the output continues in the background and control returns to MMBasic. The software automatically and separately uses the number of items in each of the arrays to drive the DACs. The frequency is the rate at which the DACs change value. The maximum frequency is 700KHz. As an example if there are 180 items in the array c%() which are displayed at a frequency of 100,000 Hz this will give a waveform frequency of 100,000/180 = 555Hz. If there are 90 items in the array d%() at the same frequency of 100,000 Hz this will at the same time produce a waveform frequency of 100,000/90 = 1111Hz.
DAC STOP	Stops the DAC output and returns the DACs to normal use.
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.
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.
DHT22 pin, tvar, hvar	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: DHT22 2, 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 signal pin of the DHT22 should be pulled up by a 1K to 10K resistor (4.7K recommended) to the supply voltage.

DIM [type] decl [,decl]...
where 'decl' is:
var [length] [type] [init]
'var' is a variable name with optional dimensions
'length' is used to set the maximum size of the string to 'n' as in LENGTH n
'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword AS - as in AS FLOAT)
'init' is the value to initialise the variable and consists of:
= <expression>
For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.
 	
Examples:
DIM nbr(50)
DIM INTEGER nbr(50)
DIM name AS STRING
DIM a, b$, nbr(100), strn$(20)
DIM a(5,5,5), b(1000)
DIM strn$(200) LENGTH 20
DIM STRING strn(200) LENGTH 20
DIM a = 1234, b = 345
DIM STRING strn = "text"
DIM x%(3) = (11, 22, 33, 44)
	Declares one or more variables (ie, makes the variable name and its characteristics known to the interpreter).
	When OPTION EXPLICIT is used (as recommended) the DIM, LOCAL or STATIC commands are the only way that a variable can be created. If this option is not used then using the DIM command is optional and if not used the variable will be created automatically when first referenced.
	The type of the variable (ie, string, float or integer) can be specified in one of three ways:
	By using a type suffix (ie, !, % or $ for float, integer or string). For example:
	`DIM nbr%, amount!, name$`
	By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (ie, it does not have to be repeated). For example:
	`DIM STRING first_name, last_name, city`
	By using the Microsoft convention of using the keyword "AS" and the type keyword (ie, FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable. For example:
	`DIM amount AS FLOAT, name AS STRING`
	Floating point or integer variables will be set to zero when created and strings will be set to an empty string (ie, ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example:
	`DIM STRING city = "Perth", house = "Brick"`
	The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter "Defining and Using Variables" for more examples of the syntax.
	As well as declaring simple variables the DIM command will also declare arrayed variables (ie, an indexed variable with up to five dimensions). Note that this is different from the original Colour Maximite and Micromite versions of MMBasic which supported up to eight dimensions.
	Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example:
	`DIM array(10, 20)`
	Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1.
	The above example specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each floating point number on the Colour Maximite 2 requires 8 bytes a total of 1848 bytes of memory will be allocated.
	Strings will default to allocating 255 bytes (ie, characters) of memory for each element and this can quickly use up memory when defining arrays of strings. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters.

	For example: `DIM STRING s(5, 10)` will declare a string array with 66 elements consuming 16,896 bytes of memory while:
	`DIM STRING s(5, 10) LENGTH 20`
	Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.
	If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this, string arrays created with the LENGTH keyword act exactly the same as other string arrays. This keyword can also be used with non array string variables but it will not save any memory.
	In the above example you can also use the Microsoft syntax of specifying the type <u>after</u> the length qualifier. For example:
	`DIM s(5, 10) LENGTH 20 AS STRING`
	Arrays can also be initialised when they are declared by adding an equals symbol (=) followed by a bracketed list of values at the end of the declaration. For example:
	`DIM INTEGER nbr(4) = (22, 44, 55, 66, 88)`
	or `DIM s$(3) = ("foo", "boo", "doo", "zoo")`
	Note that the number of initialising values must match the number of elements in the array including the base value set by OPTION BASE. If a multi dimensioned array is initialised then the first dimension will be initialised first followed by the second, etc.
	Also note that the initialising values must be after the LENGTH qualifier (if used) and after the type declaration (if used).

DO <statements> LOOP	This structure will loop forever; the EXIT DO command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like GOTO or EXIT SUB (if in a subroutine).
DO WHILE expression <statements> LOOP	Loops while "expression" is true (this is equivalent to the older WHILE-WEND loop, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, not even once.
DO <statements> LOOP UNTIL expression	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is true.
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.
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	Invoke the File Manager. To simply list the files on the SD card (as in the original Colour Maximite) use the command LIST FILES or LS. See the section *File Manager* for details of how to use the file manager.
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	This command allows you to create, use and remove a variable size framebuffer which should make many applications which have a working area bigger than the screen easier to program. While using a framebuffer setting a different graphics mode which changes the colour depth will cause an error. JPG files cannot be loaded to the framebuffer and will error if tried. The framebuffer is deleted by Ctrl-C and by running a new program.
FRAMEBUFFER CREATE HorizontalSize%, VerticalSize%	This command creates a framebuffer with the width and height specified in pixels. HorizontalSize>=MM.HRES and <=1600: VerticalSize>=MM.VRES and <=1200
FFRAMEBUFFER WRITE	This command sets all drawing commands to write to the framebuffer and inherit the width and height defined.
FRAMEBUFFER BACKUP	This command creates a backup copy of the framebuffer. If a backup already exists it is overwritten. This allows the programmer to save the background before he/she starts writing non-static data to it. NB: It won't be possible to use this command if a very large framebuffer is specified in 12 or 16-bit colour depth. A sensible error will be given in this case.
FRAMEBUFFER RESTORE [x, y, w, h]	This command restores all or part of the framebuffer from the backup. This allows the programmer to “clean” all or part of the framebuffer before adding new non-static items
FRAMEBUFFER WINDOW x, y, page [,I or B]	This command copies an area MM.HRES by MM.VRES from the framebuffer with top left at x,y to the page specified, The optional parameter specifies if the copy is Immediate or during frame Blanking
FRAMEBUFFER CLOSE	This command releases the memory resources used by the framebuffer and backup allowing a new framebuffer to be created with a different size

FUNCTION xxx (arg1 [,arg2, …]) [AS <type>}
<statements>
<statements>
xxx = <return value>
END FUNCTION
	Defines a callable function. This is the same as adding a new function to MMBasic while it is running your program.
	'xxx' is the function name and it must meet the specifications for naming a variable. The type of the function can be specified by using a type suffix (ie, xxx$) or by specifying the type using AS <type> at the end of the functions definition. For example:
	`FUNCTION xxx (arg1, arg2) AS STRING`
	'arg1', 'arg2', etc are the arguments or parameters to the function (the brackets are always required, even if there are no arguments). An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) 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.
GOTO target	Branches program execution to the target, which can be a line number or a label.
I2C	The I2C commands will send and receive data over an I^2C 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^2C 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^2C slave device. 'addr' is the slave's I^2C 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^2C slave device. 'addr' is the slave's I^2C 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^2C 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.

IMAGE RESIZE x, y, width, height, new_x, new_y, new_width, new_height [,page_number] [,flag] IMAGE RESIZE_FAST x, y, width, height, new_x, new_y, new_width, new_height [,page_number] [,flag]	This takes the part of the image with a top corner at 'x', 'y' and of specified 'width' and 'height' and resizes it writing it back to the area specified by 'new_x', 'new_y', 'new_width', new_height. The command will both increase and decrease the size of the part of the image chosen. It uses a bi-linear interpolation to generate the new pixels. The 'page number' is the page that the image data is read from it is then written to the current write page as specified by the PAGE WRITE n command. If 'page_number' is omitted the data is read from the write page. If flag is set to 1 then black pixels are not written in the resized image. IMAGE RESIZE uses bi-linear interpolation to resize the image. IMAGE RESIZE_FAST uses a nearest neighbour technique and is much faster but the resulting image quality will not be as good.
IMAGE ROTATE x, y, width, height, new_x, new_y, angle! [,page_number] [,flag] IMAGE ROTATE_FAST x, y, width, height, new_x, new_y, angle! [,page_number] [,flag]	Takes the part of the image with a top corner at 'x', 'y' and of specified 'width' and 'height' and rotates it about its centre by 'angle' (in degrees). Areas of the image that after rotation are outside of the area specified are cropped. The image is then drawn with the top left corner specified by new_x and new_y The 'page number' is the page that the image data is read from and it is written to the current write page as specified by the PAGE WRITE n

	command. If 'page_number' is omitted the data is read from the write page. If flag is set to 1 then black pixels are not written in the rotated image. IMAGE ROTATE uses bi-linear interpolation to resize the image. IMAGE RESIZE_ROTATE uses a nearest neighbour technique and is much faster but the resulting image quality will not be as good.
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.
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 or empty directory specified by ‘file$’. If there is an extension it must be specified.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command. For example: `Var = 56`
LINE x1, y1, x2, y2 [, LW [, C]]	Draws a line starting at the coordinates ‘x1’ and ‘y1’ and ending at ‘x2’ and ‘y2’. ‘LW’ is the line’s width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal. ‘C’ is an integer representing the colour and defaults to the current foreground colour. All parameters can now be expressed as arrays and the software will plot the number of lines as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', and 'y2' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw' and 'c' can be either arrays or single variables/constants.
LINE INPUT [prompt$,] 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.
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 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: o 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. o 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.
LONGSTRING APPEND array%(), string$	Append a normal MMBasic string to a long string variable. array%() is a long string variable while string$ is a normal MMBasic string expression.
LONGSTRING CLEAR array%()	Will clear the long string variable array%(). ie, it will be set to an empty string.
LONGSTRING COPY dest%(), src%()	Copy one long string to another. dest%() is the destination variable and src%() is the source variable. Whatever was in dest%() will be overwritten.
LONGSTRING CONCAT dest%(), src%()	Concatenate one long string to another. dest%() is the destination variable and src%() is the source variable. src%() will the added to the end of dest%() (the destination will not be overwritten).
LONGSTRING LCASE array%()	Will convert any uppercase characters in array%() to lowercase. array%() must be long string variable.
LONGSTRING LEFT dest%(), src%(), nbr	Will copy the left hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the beginning of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING LOAD array%(), nbr, string$	Will copy 'nbr' characters from string$ to the long string variable array%() overwriting whatever was in array%().
LONGSTRING MID dest%(), src%(), start, nbr	Will copy 'nbr' characters from src%() to dest%() starting at character position 'start' overwriting whatever was in dest%(). ie, copy from the middle of src%(). 'nbr' is optional and if omitted the characters from 'start' to the end of the string will be copied src%() and dest%() must be long string variables. 'start' and 'nbr' must be an integer constants or expressions.
LONGSTRING PRINT [#n,] src%()	Prints the longstring stored in ‘src%()’ to the file or COM port opened as ‘#n’. If ‘#n’ is not specified the output will be sent to the console.
LONGSTRING REPLACE array%() , string$, start	Will substitute characters in the normal MMBasic string string$ into an existing long string array%() starting at position ‘start’ in the long string.
LONGSTRING RIGHT dest%(), src%(), nbr	Will copy the right hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the end of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING TRIM array%(), nbr	Will trim ‘nbr’ characters from the left of a long string. array%() must be a long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING UCASE array%()	Will convert any lowercase characters in array%() to uppercase. array%() must be long string variable.
LOOP [UNTIL expression]	Terminates a program loop: see DO.
LS	An alias for the LIST FILES command.
MAP	The MAP commands allow the programmer to set the colours used in 8-bit colour modes. Each value in the 8-bit colour pallet can be set to an

	independent 24-bit colour.
MAP(n) = rgb%	This will assign the 24-bit colour 'rgb% to all pixels with the 8-bit colour value of 'n'. The change is activated after the MAP SET command
MAP MAXIMITE	This will set the colour map to the colours implemented in the original Colour Maximite.
MAP SET	This will cause MMBasic to update the colour map (set using MAP(n)=rgb%) during the next frame blanking interval.
MAP RESET	This will reset the colour map to the default colours. This map is used to assign 24-bit colours to individual values in the 8-bit colour space.
MATH	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.
Matrix arithmetic	
MATH M_PRINT array()	Quick mechanism to print a 2D matrix one row per line.
MATH M_TRANSPOSE in(), out()	Transpose matrix in() and put the answer in matrix out(), both arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in(m,n) out(n,m)
MATH M_MULT in1(), in2(), out()	Multiply the arrays in1() and in2() and put the answer in out()c. All arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in1(m,n) in2(p,m) ,out(p,n)
Vector arithmetic	
MATH V_PRINT array()	Quick mechanism to print a small array on a single line
MATH V_NORMALISE inV(), outV()	Converts a vector inV() to unit scale and puts the answer in outV() (sqr(x*x + y*y +.....)=1 There is no limit on number of elements in the vector
MATH V_MULT matrix(), inV(), outV()	Multiplies matrix() and vector inV() returning vector outV(). The vectors and the 2D matrix can be any size but must have the same cardinality.
MATH V_CROSS inV1(), inV2(), outV()	Calculates the cross product of two three element vectors inV1() and inV2() and puts the answer in outV()

Quaternion arithmetic	
MATH Q_INVERT inQ(), outQ()	Invert the quaternion in inQ() and put the answer in outQ()
MATH Q_VECTOR inV(), outVQ()	Converts a vector inV() to a normalised quaternion vector outVQ() with the original magnitude stored
MATH Q_CREATE theta, x, y, z, outRQ()	Generates a normalised rotation quaternion outRQ() to rotate quaternion vectors around axis x,y,z by an angle of theta. Theta is specified in radians.
MATH Q_MULT inQ1(), inQ2(), outQ()	Multiplies two quaternions inQ1() and inQ2() and puts the answer in outQ()
MATH Q_ROTATE , RQ(), inVQ(), outVQ()	Rotates the source quaternion vector inVQ() by the rotate quaternion RQ() and puts the answer in outVQ()
MATH FFT signalarray!(), FFTarray!()	Performs a fast fourier transform of the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "FFTarray" must be floating point and have dimension 2*N where N is the same as the signal array (e.g. f(1,1023) assuming OPTION BASE is zero) The command will return the FFT as complex numbers with the real part in f(0,n) and the imaginary part in f(1,n)
MATH FFT INVERSE FFTarray!(), signalarray!()	Performs an inverse fast fourier transform of the data in “FFTarray!”. "FFTarray" must be floating point and have dimension 2*N where N must be a power of 2 (e.g. f(1,1023) assuming OPTION BASE is zero) with the real part in f(0,n) and the imaginary part in f(1,n). "signalarray" must be floating point and the single dimension must be the same as the FFT array. The command will return the real part of the inverse transform in "signalarray".
MATH FFT MAGNITUDE signalarray!(),magnitudearray!()	Generates magnitudes for frequencies for the data in “signalarray!” "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "magnitudearray" must be floating point and the size must be the same as the signal array The command will return the magnitude of the signal at various frequencies according to the formula: frequency at array position N = N * sample_frequency / number_of_samples
MATH FFT PHASE signalarray!(), phasearray!()	Generates phases for frequencies for the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "phasearray" must be floating point and the size must be the same as the signal array The command will return the phase angle of the signal at various frequencies according to the formula above.

MEMORY	List the amount of memory currently in use. For example:
	`Flash:`
	`  39K ( 6%) Program (1450 lines)`
	`  527K (94%) Free`

<table>
<tr><td></td><td>

```
RAM:
   0K ( 0%) 0 Variables
   1K ( 0%) General
5470K (100%) Free
```

Notes:
<ul>
<li>General memory is used by serial I/O buffers, etc.</li>
<li>Memory usage is rounded to the nearest 1K byte.</li>
</ul>
</td></tr>
<tr><td>MID$(str$, start [, num]) = str2$</td><td>The characters in 'str$', beginning at position 'start', are replaced by the characters in 'str2$'. The optional 'num' refers to the number of characters from 'str2' that are used in the replacement. If 'num' is omitted, all of 'str2' is used. Whether 'num' is omitted or included, the replacement of characters never goes beyond the original length of 'str$'.</td></tr>
<tr><td>MKDIR dir$</td><td>Make, or create, the directory ‘dir$’ on the SD card.</td></tr>
<tr><td>MODE r, bits [, bg [, int]]</td><td>
Set the format for the VGA video output.
'r' is the screen resolution. It is a number from 1 to 5 as follows:
1 = 800 x 600 pixels
2 = 640 x 400 pixels
3 = 320 x 200 pixels
4 = 480 x 432 pixels
5 = 240 x 216 pixels
‘bits' is the colour depth and can be 8, 12, or 16 – see the table below.
‘bg’ is the background colour and can be used in the 12-bit mode. If pixels in layer 0 are not set to solid (transparency = 15) then the background will show through as determined by the transparency value of the pixel. This parameter is ignored in 8 and 16-bit modes.
‘int’ is a subroutine that will be called at the start of frame blanking.
The specifications of the colour depth (‘bits') are::
<table>
<tr><th></th><th>16-bit</th><th>12-bit</th><th>8-bit</th></tr>
<tr><td>H/W Pixel Format</td><td>RGB565</td><td>ARGB4444</td><td>RGB332</td></tr>
<tr><td>Colours</td><td>65536</td><td>4096</td><td>256</td></tr>
<tr><td>Transparency</td><td>None</td><td>16-levels</td><td>None</td></tr>
<tr><td>Pages Used</td><td>1</td><td>2</td><td>1</td></tr>
<tr><td>Layers</td><td>1</td><td>2 + background</td><td>1</td></tr>
</table>
The numbers of pages available in the various modes are:
Modes 1,16 and 1,12: 3 pages (0 to 2)
Modes 1,8, 2,16 and 4,16 : 7 pages (0 to 6)
Modes 2,8, 4,8, 3,12, 3,16, 5,12 and 5,16: 14 pages (0 to 13)
Modes 3,8 and 5,8 28 pages (0 to 27)
All modes work perfectly with monitors that have an aspect ratio of 4:3 or widescreen monitors that can switch to that ratio (most widescreen monitors will do this automatically).
The display always shows the contents of page 0 (16-bit and 8-bit) and pages 0 and 1 (12-bit). Use PAGE WRITE and PAGE COPY to avoid flashing and tearing artefacts.
For 12-bit colour depth page 0 is the lower level and page 1 the upper so the stack is: background, page 0, page 1 with each one overwriting the previous
</td></tr>
</table>

	in turn as defined by the transparency values of each individual pixel. MM.INFO(MAX PAGES) and MM.INFO(PAGE ADDRESS n) are useful if you wish to PEEK or POKE the video memory. In all cases the memory is arranged as a two dimensional array x,y so, to get the address of a specific pixel on a specific page n,: • With an 8-bit colour depth you would use: add% =MM.INFO(page address n)+ y* mm.hres +x • For 12 or 16-bit colour depth you would use add% =MM.INFO(page address n)+ (y* mm.hres +x) * 2 For modes 3 and 5 each line is duplicated to get square pixels so Y needs to be multiplied by 2 for PEEK and both lines y*2 and Y*2+1 need to be POKEd. Eg, for an 8-bit colour depth: add1% = MM.INFO(page address n)+ (y * 2) * MM.HRES + x add2% = MM.INFO(page address n)+ (y * 2 + 1) * MM.HRES + x For the 12 and 16-bit modes you can use POKE SHORT and PEEK(SHORT) which are designed for this purpose.
NEW	Deletes the program in flash, 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	Setup 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. To disable this interrupt, use numeric zero for the target, ie: ON KEY 0
ONEWIRE RESET pin or ONEWIRE WRITE pin, flag,	Commands for communicating with 1-Wire devices. ONEWIRE RESET will reset the 1-Wire bus ONEWIRE WRITE will send a number of bytes

length, data [, data…] or ONEWIRE READ pin, flag, length, data [, data…]	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$() functions all use 'fnbr' to identify the file being operated on. See also OPTION ERROR and MM.ERRNO for error handling.
OPEN comspec$ 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. Six options can be added to the end of 'comspec$' • 'INV' specifies that the transmit and receive polarity is inverted. • ‘OC’ will force the transmit pin (and DE on COM1:) to be open collector. The default is normal (0 to 3.3V) output. • 'S2' specifies that two stop bits will be sent following each character transmitted. • '7BIT' will specify that 7 bit transmit and receive is to be used. • ‘ODD’ will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • ‘EVEN’ will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'DEP' will enable RS485 mode with a positive output on the COM1-DE pin • 'DEN' will enable RS485 mode with a negative output on the COM1-DE pin
OPEN comspec$ 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.
PAGE COPY n TO m [,when]	Copy the contents of one video page to another. 'n' is the source and 'm' is the destination. 'when' can be one of letters I, B, or D. If omitted it will default to I. I means do the copy immediately. It is the most efficient but risks causing screen artefacts B means wait until the next frame blanking and then do the copy. It is the least efficient but is absolutely determinate in its effect and no screen artefacts will ever be seen. D means carry on processing the next command and do the copy in the background when the next frame blanking occurs. This is efficient but must be used with care as subsequent drawing commands may or may not be included in the copy depending on the timing of the next screen blanking

PAGE SCROLL pageno, x, y [,fillcolour]	Will scroll the image on the page specified by 'pageno' by the amount defined by 'x' and 'y'. By default the area scrolled off the screen appears on the other side. If 'fillcolour' is specified it will replace the area left behind by the scroll with the colour specified. If 'fillcolour' is set to -1 then the area left behind by the scroll is left untouched. This is the most efficient version and is suitable if there is a black background.
PAGE STITCH frompage1, from_page_2, topage, offset	Will take the last horizontal resolution minus 'offset' columns from 'frompage1' and the first 'offset' columns from 'frompage2' and copies them to 'topage'
PAGE AND_PIXELS sourcepage1, sourcepage2, destinationpage PAGE OR_PIXELS sourcepage1, sourcepage2, destinationpage PAGE XOR_PIXELS sourcepage1, sourcepage2, destinationpage	These commands combine the pixels on sourcepage1 and sourcepage2 by ANDing, ORing, or XORing them. destinationpage can be the same as either of the sourcepages if required.
PAGE WRITE n	Instructs MMBasic to make all graphics commands write to page 'n'. If not used page 0 is the default. Set the page to FRAMEBUFFER to write to the framebuffer – see the FRAMEBUFFER command.
PAUSE delay	Halt execution of the running program for ‘delay’ ms. This can be a fraction. For example, 0.2 is equal to 200 µs. The maximum delay is 2147483647 ms (about 24 days). Note that interrupts will be recognised and processed during a pause.
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.
PLAY EFFECT filename$ [,interrupt]	This will play the WAV file ' filename$' 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$' is the file to play (the appropriate extension will be appended if missing). The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. For WAV files MMBasic will automatically compensate for the frequency, number of bits and number of channels of the WAV file. For FLAC files the supported frequencies are: 44100Hz 16-bit(CD quality) and 24-bit 48000Hz 16-bit and 24-bit 88200Hz 16-bit and 24-bit 96000Hz 24-bit Maximums for FLAC and WAV file playback are 96KHz 24-bit. Both will auto-configure to the file provided. As an indication, 96KHz 24-bit FLAC uses just over 50% of the CPU's resources. If fname$ 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 SOUND soundno, channelno, type [,frequency] [,volume]	Play a series of sounds simultaneously on the audio output. 'soundno' is the sound number and can be from 1 to 4 allowing for four simultaneous sounds on each channel. 'channelno' specifies the output channel. It can be L (left speaker), R (right speaker) or B (both speakers) 'type' is the type of waveform. It can be S (sine wave), Q (square wave), T (triangle wave), W (rising sawtooth), N (noise), P (periodic noise) or O (turn off sound). Type N is true white noise. In this case the frequency parameter specifies the number of periods of 1/70000 seconds that the output stays at a particular random value. Type P is periodic white noise. In this case the frequency is some sort of relationship to the periodic frequency of the noise 'frequency' is the frequency from 1 to 20000 (Hz) and it must be specified except when type is O. 'volume' is optional and must be between 1 and 25. It defaults to 25 The first time PLAY SOUND is called all other audio usage will be blocked and will remain blocked until PLAY STOP is called. Output can be stopped temporarily using PLAY PAUSE and PLAY RESUME. Calling SOUND on an already running 'soundno' will immediately replace the previous output. Individual sounds are turned off using type “O” Running 4 sounds simultaneously on both channels of the audio output consumes about 23% of the CPU.
PLAY PAUSE PLAY 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 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!	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%'

or POKE VAR var, offset, byte or POKE VARTBL, offset, byte	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 size of the n array determines the number of polygons that will be drawn. 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 xy-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 xy-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.
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. 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. 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. 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$' can be omitted and in that case MMBasic will run the "current program name" which is the file last used by RUN, EDIT or AUTOSAVE.
SAVE IMAGE file$ [,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.

	Example:
	`SELECT CASE nbr%`
	`  CASE 4, 9, 22, 33 TO 88`
	`  statements`
	`  CASE IS < 4, IS > 88, 5 TO 8`
	`  statements`
	`  CASE ELSE`
	`  statements`
	`END SELECT`
	Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT…CASE statements can be nested inside the CASE statements of other SELECT…CASE statements.

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 diagrams on pages 6 and 7 (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. See the chapter "Special Hardware Devices" for more details.
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 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]	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 (ie, SETTICK 0, 0, 3 will disable tick timer number 3).
SORT array() [,indexarray()]	This command takes an array of any type (integer, float or string) and sorts it into ascending order in place. It has an optional parameter ‘indexarray%()’. If used this must be an integer array of the same size as the array to be sorted. After the sort this array will contain the original index position of each element in the array being sorted before it was sorted. Any data in the array will be overwritten. This allows connected arrays to be sorted. See the section *Sorting Data* in the tutorial Programming with the Colour Maximite 2 for an example.

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$' is used 'nbr' characters will be sent. 'array' must be a single dimension float or integer array and 'nbr' elements will be sent or received. See Appendix D for the details.
SPRITE	The SPRITE commands are used to manipulate small graphic images on the VGA screen. These are useful when writing games. The maximum size of a sprite is MM.HRES-1 and MM.VRES-1 See also the SPRITE() functions.
SPRITE CLOSE [#]n	Closes sprite “n” and releases its memory resources allowing the sprite number to be re-used. The command will give an error if other sprites are copied from this one unless they are closed first.
SPRITE CLOSE ALL	Closes all sprites and releases all sprite memory. The screen is not changed.
SPRITE COPY [#]n, [#]m, nbr	Makes a copy of sprite “n” to “nbr” of new sprites starting a number “m”. Copied sprites share the same loaded image as the original to save memory
SPRITE HIDE [#]n	Removes sprite n from the display and replaces the stored background. To restore a screen to a previous state sprites should be hidden in the opposite order to which they were written "LIFO"
SPRITE INTERRUPT sub	Specifies the name of the subroutine that will be called when a sprite collision occurs. See Appendix E for how to use the function SPRITE to interrogate details of what has collided
SPRITE LOAD fname$ [,start_sprite_number]	Loads the file ‘fname$’ which must be formatted as an original Colour Maximite sprite file. See the original Colour Maximite *MMBasic Language Manual* for the file format. Multiple sprite files can be loaded by specifying a different ‘start_sprite_number’ for each file. The programmer is responsible for making sure that the sprites do not overlap.
SPRITE LOADARRAY [#]n, w, h, array%()	Creates the sprite 'n' with width 'w' and height 'h' by reading w*h RGB888 values from 'array%()'. The RGB888 values must be stored in order of columns across and then rows down starting at the top left. This allows the programmer to create simple sprites in a program without needing to load them from disk or read them from the display. The firmware will generate an error if 'array%()' is not big enough to hold the number of values required.
SPRITE LOADPNG [#]n, fname$ [, transparency_cut_off]	Loads the PNG image ‘fname$’ as sprite number ‘n’. If the PNG file is in ARGB8888 format the ‘transparency_cut_off’ parameter is used to determine whether the pixel should be solid or missing/transparent. Valid values are 1 to 15, default is 8. MMBasic compares the 4 most significant bits of the transparency data in the file with

	the cut off value and assigns a transparency of 0 or 15 depending on the comparison. This allows RGB(0,0,0) to be a valid solid colour. If the file is in RGB888 format then an RGB level of 0,0,0 is used to determine transparency as there is no other information to use.
SPRITE MOVE	Actions a single atomic transaction that re-locates all sprites which have previously had a location change set up using the SPRITE NEXT command. Collisions are detected once all sprites are moved and reported in the same way as from a scroll
SPRITE NEXT [#]n, x, y	Sets the X and Y coordinate of the sprite to be used when the screen is next scrolled or the SPRITE MOVE command is executed. Using SPRITE NEXT rather than SPRITE SHOW allows multiple sprites to be moved as part of the same atomic transaction.
SPRITE NOINTERRUPT	Disables collision interrupts
SPRITE READ [#]n, x , y, w, h [,pagenumber]	Reads the display area specified by coordinates ‘x’ and ‘y’, width ‘w’ and height ‘h’ into buffer number ‘n’. If the buffer is already in use and the width and height of the new area are the same as the original then the new command will overwrite the stored area. The optional parameter page number specifies which page is to be read to create the sprite. The default is the current write page. Set the page to FRAMEBUFFER to read from the framebuffer – see the FRAMEBUFFER command.
SPRITE SCROLL x, y [,col]	Scrolls the background and any sprites on layer 0 'x' pixels to the right and 'y' pixels up. 'x' can be any number between -MM.HRES-1 and MM.HRES-1, 'y' can be any number between -MM.VRES-1 and MM.VRES-1. Sprites on any layer other than zero will remain fixed in position on the screen. By default the scroll wraps the image round. If 'col' is specified the colour will replace the area behind the scrolled image. If 'col' is set to -1 the scrolled area will be left untouched.
SPRITE SCROLLR x, y, w, h, delta_x, delta_y [,col]	Scrolls the region of the screen defined by top-right coordinates ‘x’ and ‘y’ and width and height ‘w’ and ‘h’ by ‘delta_x’ pixels to the right and ‘delta_y’ pixels up. By default the scroll wraps the background round. If ‘col’ is specified the colour will replace the area behind the scrolled image. Sprites on any layer other than zero will remain fixed in position on the screen. Sprites in layer zero where the centre of the sprite (x+ w/2, y+ h/2) falls within the scrolled region will move with the scroll and wrap round if the centre moves outside one of the boundaries of the scrolled region.
SPRITE SWAP [#]n1, [#]n2 [,orientation]	Replaces the sprite ‘n1’ with the sprite ‘n2’. The sprites must have the same width and height and ‘n1’ must be displayed or an error will be generated. Refer to SPRITE SHOW for details of the orientation parameter. The replacement sprite inherits the background from the original as well as its position in the list of order drawn.
SPRITE TRANSPARENCY [#]n, transparency	Transparency can be between 1 and 15 and changes all pixels with a non-zero transparency in the stored sprite to the new level.
SPRITE 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 WRITE [#]n, x y	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.
STATIC variable [, variables] See DIM for the full syntax.	Defines a list of variable names which are local to the subroutine or function. These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command). This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters. Static variables can be initialised to a value. This initialisation will take effect only on the first call to the subroutine (not on subsequent calls).
SUB xxx (arg1 [,arg2, …]) <statements> <statements> END SUB	Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program. 'xxx' is the subroutine name and it must meet the specifications for naming a variable. 'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using `AS <type> (ie, arg1 AS STRING)`. Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: `MySub a1, a2` When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.

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. The time is set to "00:00:00" on first power up however the time 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 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. 'x1', 'y1', 'x2', 'y2', 'x3',and 'y3' must all be arrays or all be single variables /constants otherwise an error will be generated 'c' and 'fill' can be either arrays or single variables/constants.
TURTLE	The TURTLE commands are a simple method of drawing graphic images on the VGA screen. They are suitable for newcomers and students.
TURTLE RESET	Clears the screen and re-initialises the turtle system. The turtle is located at MM.HRES\2, MM.VRES\2. The default pen colour is white and the fill colour is green. The initial heading is up (0 degrees)
TURTLE DRAW TURTLE	Draws a turtle at the current location and in the current orientation
TURTLE DRAW PIXEL x, y	Draw a 1-pixel dot at the given location using the current draw colour, regardless of current turtle location or pen status.
TURTLE FILL PIXEL x, y	Draw a 1-pixel dot at the given location using the current fill colour, regardless of current turtle location or pen status.
TURTLE DRAW LINE x1, y1, x2, y2	Draw a straight line between the given coordinates, regardless of current turtle location or pen status.
TURTLE DRAW DIRCLE x, y, r	Draw a circle at the given coordinates with the given radius, regardless of current turtle location or pen status.
TURTLE PEN UP	Lifts the pen so moves do not write to the screen
TURTLE PEN DOWN	Lowers the pen so moves do write to the screen
TURTLE FORWARD n	Moves the turtle n pixels in the current heading
TURTLE BACKWARD n	Moves the turtle n pixels opposite the current heading
TURTLE DOT	Draw a 1-pixel dot at the current location, regardless of pen status
TURTLE TURN LEFT deg	Turn the turtle to the left (anti-clockwise) by the specified number of degrees.
TURTLE TURN RIGHT deg	Turn the turtle to the right (clockwise) by the specified number of degrees.
TURTLE BEGIN FILL	Start filling. Call this before drawing a polygon to activate the bookkeeping required to run the filling algorithm later.

TURTLE END FILL	End filling. Call this after drawing a polygon to trigger the fill algorithm. The filled polygon may have up to 128 sides.
TURTLE HEADING deg	Rotate the turtle to the given absolute heading (in degrees). 0 degrees means facing straight up. 90 degrees means facing to the right.
TURTLE PEN COLOUR col	Set the current drawing colour. Colours are specified as per normal drawing commands
TURTLE FILL COLOUR col	Set the current fill colour. Colours are specified as per normal drawing commands
TURTLE MOVE x, y	Move the turtle to the specified location, drawing a straight line if the pen is down.
UPDATE FIRMWARE	Switch the STM32 CPU into firmware update mode. This is the same as switching the BOOT CONFIG switch on the Waveshare CPU board to SYSTEM and allows for quick firmware updates without opening the Maximite's case. You should cycle the power following a firmware update or if this command was accidently used. This will return the STM32 CPU to normal mode.
VAR SAVE var [, var]… or VAR RESTORE or VAR CLEAR	VAR SAVE will save one or more variables to non volatile flash 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.
WII CLASSIC OPEN [n] [,interrupt [,bitmask]] or WII CLASSIC CLOSE [n]	Opens a Classic port for communications. 'n' is the I^2C channel that the Classic is connected to. If not specified channel 3 is opened (the front panel Wii connector). 'interrupt' is an optional MMbasic interrupt routine that will be triggered when buttons are pressed. The interrupts trigger once on the button press and will not re-interrupt until it is released. The bitmask specifies which of the 15 buttons on the Wii Classic will trigger an interrupt. The default is that they all will. See the CLASSIC function for details of which bits in the bitmask control which buttons. The firmware checks for the existence of the Classic on open and will throw error if not found. Once open the Classic is polled continuously in the background once every 16mSec. The I^2C transfers are under interrupt control and non-blocking for the rest of the firmware. The I^2C channel is opened at 100KHz - other I^2C devices can share the I^2C port with the Classic if required in which case I2C(n) OPEN should not be called. WII CLASSIC CLOSE will close the Wii port ‘n’. If the parameter is omitted channel 3 is closed. Also see the CLASSIC() function for accessing the state of the device.
WII NUNCHUK OPEN [n] [,Zinterrupt [,Cinterrupt]] or WII NUNCHUK CLOSE [n]	Opens a Nunchuk port for communications. 'n' is the I^2C channel that the Nunchuk is connected to. If not specified channel 3 is opened (the front panel Wii connector). 'Cinterrupt' and 'Zinterrupt' are optional MMbasic interrupt routines that will be triggered when one or other of the buttons is pressed. The interrupts trigger once on the button press and will not re-interrupt until it is released. The firmware checks for the existence of the Nunchuk on open and will throw error if not found. Once open the Nunchuk is polled continuously in the background once every 16mSec. The I^2C transfers are under interrupt control and non-blocking for the rest of the firmware. The I^2C channel is opened at 100KHz - other I^2C devices can share the I^2C port with the Nunchuk if required in which case I2C(n) OPEN should not be called. WII NUNCHUK CLOSE will close the Nunchuk port ‘n’. If the parameter is omitted channel 3 is closed. Also see the NUNCHUK() function for accessing the state of the device.

XMODEM SEND file$ or XMODEM RECEIVE file$	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$' held on the Colour Maximite's SD card to the remote device. XMODEM RECEIVE will accept 'file$' 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. SEND and RECEIVE can be abbreviated to S and 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. 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.
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
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().
CLASSIC(funct, channel)	Returns data from a Wii Classic controller. 'channel' is the I^2C channel that the Classic is connected to. 'funct' is a 1 or 2 letter code indicating the information to return as follows: LX returns the position of the analog left joystick x axis LY returns the position of the analog left joystick y axis RX returns the position of the analog right joystick x axis RY returns the position of the analog right joystick y axis L returns the position of the analog left button R returns the position of the analog right button B returns a bitmap of the state of all the buttons. A bit will be set to 1 if the button is pressed. T returns the ID code of the controller - should be hex &H4200101 The button bitmap is as follows: BIT 0: Button R BIT 1: Button start BIT 2: Button home BIT 3: Button select BIT 4: Button L BIT 5: Button down cursor BIT 6: Button right cursor BIT 7: Button up cursor BIT 8: Button left cursor BIT 9: Button ZR BIT 10: Button x BIT 11: Button a BIT 12: Button y BIT 13: Button b BIT 14: Button ZL These bit positions are also used in the interrupt bitmask specified in the WII CLASSIC OPEN command

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`.
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-yyyy". Use NOW to get the day for the current date, e.g. ? 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". Use NOW to get the epoch number for the current date and time, i.e. ? 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$(s$, 3, ",", "'")`
	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.
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) GPS(VALID)	Returns the track over the ground (degrees true) as a floating point number. 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() .
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 they 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.
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 **Simple functions** MATH(ATAN3 x,y) MATH(COSH a) MATH(LOG10 a) MATH(SINH a) MATH(TANH a) **Simple Statistics** MATH(CHI a()) MATH(CHI_p a()) MATH(MAX a()) MATH(MEAN a()) MATH(MEDIAN a()) MATH(MIN a()) MATH(SD a()) MATH(SUM a()) **Vector Arithmetic** MATH(MAGNITUDE v()) MATH(DOTPRODUCT v1(), v2())	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. Returns ATAN3 of x and y Returns the hyperbolic cosine of a Returns the base 10 logarithm of a Returns the hyperbolic sine of a Returns the hyperbolic tan of a Returns the Pearson's chi-squared value of the two dimensional array a()) Returns the associated probablity in % of the Pearson's chi-squared value of the two dimensional array a()) Returns the maximum of all values in the a() array, a() can have any number of dimensions Returns the average of all values in the a() array, a() can have any number of dimensions returns the median of all values in the a() array, a() can have any number of dimensions Returns the minimum of all values in the a() array, a() can have any number of dimensions Returns the standard deviation of all values in the a() array, a() can have any number of dimensions Returns the sum of all values in the a() array, a() can have any number of dimensions Returns the magnitude of the vector v(). The vector can have any number of elements Returns the dot product of two vectors v1() and v2(). The vectors can have any number of elements but must have the same cardinality
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$’ beginning at ‘start’ and continuing for ‘nbr’ characters. The first character in the string is number 1. If ‘nbr’ is omitted the returned string will extend to the end of ‘string$’
NUNCHUK(funct, channel)	Returns data from a Nunchuk controller. 'channel' is the I^2C channel that the Nunchuk is connected to. 'funct' is a 2 or 3 letter code indicating the information to return as follows: JX returns the value of the joystick X-axis JY returns the value of the joystick Y-axis AX returns the value of the x acceleration AY returns the value of the y acceleration AZ returns the value of the z acceleration Z returns the value of the z button (1 if pressed) C returns the value of the c button (1 if pressed) T returns the id code of the Wii device: &HA4200000=Original, &HA4200101=Classic, &HA4200402=Balance JXL returns the calibrated X value of the joystick in the far left position JXC returns the calibrated X value of the joystick in the centre position JXR returns the calibrated X value of the joystick in the far right position JYT returns the calibrated Y value of the joystick in the top position JYC returns the calibrated Y value of the joystick in the centre position JYB returns the calibrated Y value of the joystick in the bottom position AX0 returns the calibrated zero gravity value of the x-axis accelerometer AX1 returns the calibrated one gravity value of the x-axis accelerometer AY0 returns the calibrated zero gravity value of the y-axis accelerometer AY1 returns the calibrated one gravity value of the y-axis accelerometer AZ0 returns the calibrated zero gravity value of the z-axis accelerometer AZ1 returns the calibrated one gravity value of the z-axis accelerometer
OCT$(number [, chars])	Returns a string giving the octal (base 8) representation of 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
PEEK(BYTE addr%) PEEK(SHORT addr%) PEEK(WORD addr%) PEEK(INTEGER addr%) PEEK(FLOAT addr% PEEK(VARADDR var) PEEK(VAR var, ±offset) PEEK(VARTBL, ±offset) PEEK(PROGMEM, ±offset)	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%' 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(). 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 STM32 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)	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.
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 gray. For example, RGB(red) or RGB(cyan). '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$' with 'number-of-chars' from the right (end) of the string.
RND(number) or RND	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The Colour Maximite 2 uses the hardware random number generator in the STM32 series of chips to deliver true random numbers. This means that the RANDOMIZE command is no longer needed and is not supported.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE$(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(H,[#]n)	Returns the height of sprite n. This function is active whether or not the sprite is currently displayed (active).
SPRITE(L, [#]n)	Returns the layer number of active sprites number n
SPRITE(N)	Returns the number of displayed (active) sprites
SPRITE(N,n)	Returns the number of displayed (active) sprites on layer n
SPRITE(S)	Returns the number of the sprite which last caused a collision. NB if the number returned is Zero then the collision is the result of a SPRITE SCROLL command and the SPRITE(C…) function should be used to find how many and which sprites collided.
SPRITE(V,spriteno1,spriteno2)	Returns the vector from 'spriteno1' to 'spriteno2' in radians. The angle is based on the clock so if 'spriteno2' is above 'spriteno1' on the screen then the answer will be zero. This can be used on any pair of sprites that are visible. If either sprite is not visible the function will return -1. This is particularly useful after a collision if the programmer wants to make some differential decision based on where the collision occurred. The angle is calculated between the centre of each of the sprites which may of course be different sizes.
SPRITE(W, [#]n)	Returns the width of sprite n. This function is active whether or not the sprite is currently displayed (active).
SPRITE(X, [#]n)	Returns the X-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.
SPRITE(Y, [#]n)	Returns the Y-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.
STR2BIN(type, string$ [,BIG])	Returns a number equal to the binary representation in ‘string$’. ‘type’ can be: INT64 converts 8 byte string representing a signed 64-bit integer to an integer UINT64 converts 8 byte string representing an unsigned 64-bit integer to an integer INT32 converts 4 byte string representing a signed 32-bit integer to an integer UINT32 converts 4 byte string representing an unsigned 32-bit integer to an integer INT16 converts 2 byte string representing a signed 16-bit integer to an integer UINT16 converts 2 byte string representing an unsigned 16-bit integer to an integer INT8 converts 1 byte string representing a signed 8-bit integer to an integer UINT8 converts 1 byte string representing an unsigned 8-bit integer to an integer SINGLE converts 4 byte string representing single precision float to a float DOUBLE converts 8 byte string representing single precision float to a float By default the string must contain the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to ‘BIG’ will interpret the string in big-endian format (ie, the most significant byte is the first one in the string). This function makes it easy to read data from binary data files, interpret numbers from sensors or efficiently read binary data from flash memory chips. An error will be generated if the string is the incorrect length for the conversion requested See also the function BIN2STR$
SQR(number)	Returns the square root of the argument 'number'.

STR$(number) or STR$(number, m) or STR$(number, m, n) or STR$(number, m, n, c$)	Returns a formatted string in decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR$(123.456) will return "123.456" STR$(-123.456) will return "-123.456" STR$(123.456, 1) will return "123.456" STR$(123.456, -1) will return "+123.456" STR$(123.456, 6) will return " 123.456" STR$(123.456, -6) will return " +123.456" STR$(-123.456, 6) will return " -123.456" STR$(-123.456, 6, 5) will return " -123.45600" STR$(-123.456, 6, -5) will return " -1.23456e+02" STR$(53, 6) will return " 53" STR$(53, 6, 2) will return " 53.00" STR$(53, 6, 2, "*") will return "****53.00"
STRING$(nbr, ascii) or STRING$(nbr, string$)	Returns a string 'nbr' bytes long consisting of either the first character of string$ or the character representing the ASCII value 'ascii' which is a decimal number in the range of 32 to 126.
TAB(number)	Outputs spaces until the column indicated by 'number' has been reached on the console output.
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.

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

Two serial ports are available for asynchronous serial communications labelled COM1: and COM2:. In addition, if the serial console is disabled then that port is available as COM3:.

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 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. 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 one 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 Maximite and Rx means an input to the Maximite.

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 Colour Maximite 2 with two exceptions:

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

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

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

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

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

Use this circuit:

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

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

Appendix B
I^2C Communications

The Colour Maximite 2 implements three I^2C 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^2C 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^2C 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^2C 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^2C 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^2C 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^2C 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^2C 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^2C channels 1 and 2. Their signals are marked as data line (SDA) and clock (SCL). I^2C 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^2C 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^2C 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^2C 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^2C 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 and SPI2 refers to channel 2. SPI2 is not listed below however it is available on the Colour Maximite 2 and has an identical syntax.

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 Colour Maximite 2 is always the master that pin will be configured as an input. Similarly MOSI stands for Master Out Slave In and that pin will be configured as an output.

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

SPI Open

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

```
SPI OPEN speed, mode, bits
```

Where:

- ‘speed’ is the speed of the clock. This can be 25000000, 12500000, 6250000, 3125000, 1562500, 781250, 390625 or 195315 (ie, 25MHz, 12.5MHz, 6.25MHz, 3.125MHz, 1562.5KHz, 781.25KHz, 390.625KHz or 195.3125KHz). For any other values the firmware will select the next valid speed that is equal or slower than the speed requested.
- 'mode' is a single numeric digit representing the transmission mode – see Transmission Format below.
- 'bits' is the number of bits to send/receive. This can be 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 5MHz 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 5MHz, 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)
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

If the shift key is simultaneously pressed 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 only works with the function keys F1 to F12; it is ignored for the other keys.

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
Loading the Firmware

The STM32 processor includes its own programmer/bootloader so the Colour Maximite 2 firmware can be easily loaded via USB using a personal computer or laptop (special hardware is not needed). Just follow these steps.

Go to https://www.st.com/en/development-tools/stm32cubeprog.html and download the STM32CubeProgrammer software. This is free software but STM do require you to have an STM account or provide your name and email address. They will email you a link to download the software. Then install this software on your computer (Windows, Linux and macOS are supported).

Plug the Waveshare CPU board into its position on the Colour Maximite 2 motherboard and on the Waveshare board do the following:

- Remove all the jumpers.
- Set the power switch to "5VIN"
- Set the BOOT CONFIG switch to "SYSTEM"

If you have a fully assembled board without the Waveshare module this function is provided by a set of three jumper pins labelled “program” and “run”. Place a jumper on the common and “program” pins.

Make sure that the Type-B USB power cable to the Colour Maximite is disconnected.

Using a USB Type-A to Type-A cable connect the USB Keyboard port on the Colour Maximite 2 to a USB port on your desktop computer. This will power up the Colour Maximite regardless of the position of the power switch. You should also hear a sound from your computer as it connects to the Colour Maximite.

Run the STM32CubeProgrammer software on your computer. On the top right of the program window select USB as the communications method. If the program does not recognise the USB connection click on the small blue circle to the right of the Port drop down list to refresh the entry. Your screen should look like the illustration on the right (the USB port number may vary).

Click on the "Connect" button. You should then see a series of messages as shown in the screenshot below finishing with the message "Data read successfully". Any messages in red will indicate an error.

Click on the download button ( ) on the left side of the STM32CubeProgrammer window and the software will switch to the "Erasing and Programming" mode as shown below.

Use the "Browse button" to select the firmware file (it will have an extension of .bin).
Tick the "Verify programming" checkbox.

Finally, click on the "Start Programming" button.

The STM32CubeProgrammer software will then program the firmware into the flash memory on the STM32 CPU in the Colour Maximite 2 (the STM32CubeProgrammer software calls this "downloading"). After a short time a dialog box will pop up saying that "File download completed". Do not do anything at this point as the software will then start reading back the firmware programmed into the flash. When this has completed successfully another dialog box will pop up saying "Download verified successfully" as shown below.

The whole operation will take under a minute and any messages in red will indicate an error.

Then:

- Dismiss all the dialog boxes and close the STM32CubeProgrammer software.
- On the CPU board set the BOOT CONFIG switch to "Flash". If you have a fully assembled board without the Waveshare module the jumper should be between the common and the “run” pins.
- Remove the USB Type-A to Type-A cable from the USB Keyboard port.
- Plug in the VGA monitor and USB keyboard.
- Plug the Colour Maximite into power and set the power switch to ON.

On power up you should now see on the VGA monitor the Maximite logo and the version number of the firmware that you have just loaded.

If you do not have a VGA monitor you can use the serial console (over USB) to check the firmware installation. This is described in the section *Hardware Features* and *Serial Console* in this manual.

When MMBasic is first loaded it will prompt for the keyboard type, the screen type and the date/time. On subsequent firmware upgrades MMBasic will preserve these settings (in addition to OPTION.RTC CALIBRATE) and will not prompt for them again. These can be changed later using the relevant OPTION commands.

If you wish to load another version of the firmware (either earlier or later) this can be done by repeating the steps above. In this case you can use the UPDATE FIRMWARE command to place the STM32 into download mode (equivalent to setting the BOOT CONFIG switch to "SYSTEM") thereby avoiding opening the case.

Alternative Method

An alternative method of loading the firmware is to use serial transfer over USB. To do this you must first install the serial over USB driver on your desktop computer as described in the section *Hardware Features* and *Serial Console* in this manual.

Plug the Waveshare CPU board into its position on the Colour Maximite 2 motherboard and:

- Remove all the jumpers.
- Set the power switch to "5VIN"
- Set the BOOT CONFIG switch to "System" or jumper the common and “program” pins.
- Connect your desktop computer to the USB Type-B connector on the Colour Maximite.
- Press the reset button on the Waveshare CPU board.

This should power up the Colour Maximite which will then connect to your desktop computer via USB.

In Windows the connection will appear in Device Manager as "USB Serial Port" as illustrated on the right (the COM number will probably be different):

Install the STM32CubeProgrammer software on your computer as described above.

Run the software and select UART in the top right corner (as illustrated on the right). Then select the correct COM port number as reported in Device Manager. Finally make sure that the baudrate is set to 115200 baud and the parity set to even:

From then on the process is the same as that described above when using a direct USB connection via the keyboard port:

- Click on "Connect".
- Select "Erase & Programming" mode.
- Browse for the firmware file.
- Tick the "Verify programming" checkbox.
- Click on "Program".

The whole operation will take about 5 minutes.

When the programming/verify has completed set the BOOT CONFIG switch to "Flash" on the CPU board (or return the jumpers to the “run” position) and press the RESET button.

Third Method

It is also possible to program the firmware using the micro USB connector on the Waveshare board. This method is not recommended as the Waveshare module must be removed to access this connector and doing this too many times will inevitably damage the module’s pins. However, this is a handy method of testing the Waveshare board - if you can load the firmware without error it is a good indication that the module is working correctly.

To use this method remove the Waveshare CPU module from the motherboard, place shorting jumpers on all header pins except PA9-VBUS, set the power switch to "USB” and the BOOT CONFIG switch to "SYSTEM". Plug a USB cable into the micro USB connector on the top of the module and the other end into your desktop computer - both LEDs on the module should illuminate and it should connect to your computer.

Then, using the steps listed for loading the firmware via the USB keyboard port, load and verify the Colour Maximite 2 firmware using this USB cable and your desktop computer.

Linux and the Raspberry Pi

Loading the firmware from a Linux computer and/or the Raspberry Pi has some special considerations and these are explained here: http://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12171

MMBasic

Language Manual

Ver 4.5

Geoff Graham

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

or http://geoffg.net/maximite.html

Copyright 2011 - 2014 Geoff Graham

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

MMBasic is a Microsoft BASIC compatible implementation of the BASIC language with floating point and string variables, long variable names, arrays of floats or strings with multiple dimensions and powerful string handling. MMBasic was originally written for the Maximite, a small computer based on the PIC32 microcontroller from Microchip. It now runs on a variety of hardware platforms including DOS.

This manual describes the MMBasic language. For details of running MMBasic on specific platforms please refer to the following documentation or websites:

Maximite, mini-Maximite:	Maximite Hardware Manual	from: http://geoffg.net/maximite.html
UBW32 experimenter board:	Colour Maximite on the UBW32	from: http://geoffg.net/ubw32.html
DOS:	DOS MMBasic ReadMe	from: http://mmbasic.com/downloads.html
DuinoMite series:	DuinoMite MMBasic ReadMe	included with the DuinoMite update.

CGMMSTICK1 and CGCOLORMAX2 boards from CircuitGizmos go to http://www.circuitgizmos.com

DTX2-4105C module from Dimitech go to http://dimitech.com

TFT Maximite go to http://github.com/heise/MAXIMITE

Throughout this manual Maximite or MM refers to the original monochrome Maximite family (Maximite, mini Maximite, CGMMSTICK1, DuinoMite and the DTX2-4105C). Colour Maximite or CMM refers to the Colour Maximite which also includes the UBW32, TFT Maximite and the CGCOLORMAX2. DOS refers to the version that runs in a DOS box under Windows.

Contents

Functional Summary 3
Full Screen Editor 5
Input/Output 7
Audio and PWM Output 8
Special Hardware Devices 9
Graphics and Working with Colour 15
Game Playing Features 17
Defined Subroutines and Functions 18
Implementation Details 21
Predefined Read Only Variables 23
Commands 24
Functions 46
Obsolete Commands and Functions 54
Appendix A Serial Communications 55
Appendix B I^2C Communications 57
Appendix C 1-Wire Communications 63
Appendix D SPI Communications 64
Appendix E Loadable Fonts 66
Appendix F Special Keyboard Keys 67
Appendix G Tera Term Setup 68
Appendix H Sprites 69
Appendix I Random File I/O 71

Functional Summary

Command and Program Input

At the prompt (the greater than symbol, ie, >) you can enter a command line followed by the enter key and it will be immediately run. This is useful for testing commands and their effects.

To enter a program you can use the EDIT command which will invoke the full screen editor built into MMBasic. Line numbers are optional and if you use them you can enter a program at the command line by preceding each program line with a line number.

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 can be edited and reused.

A program held in memory can be listed with LIST, run using the RUN command and cleared with the NEW command. You can interrupt MMBasic at any time by typing CTRL-C and control will be returned to the prompt.

Keyboard/Display

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

Output will be simultaneously sent to the USB interface and the video display (VGA or composite) however graphics commands operate on the video output only.

Line Numbers, Program Structure and Editing

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).

Long programs (with or without line numbers) can be sent via USB to MMBasic using the XMODEM command (Maximite only) or the AUTO command.

Program and Data Storage

In the DOS version of MMBasic the drive letters are as supported by Windows.

On the Maximite and Colour Maximite two “drives” are available for storing and loading programs and data:

- Drive “A:” is a virtual drive using the PIC32’s internal flash memory and has a size of about 180KB on the monochrome Maximite (a bit less with colour or CAN).
- Drive “B:” is the SD card (if connected). It supports MMC, SD or SDHC memory cards formatted as FAT16 or FAT32 with capacities up to 32Gb.

File names must be in 8.3 format prefixed with an optional drive prefix A: or B: (the same as DOS or Windows). Long file names and directories are not supported. The default drive is B: and this can be changed with the DRIVE command.

On the Maximite MMBasic will look for a file on startup called “AUTORUN.BAS” in the root directory of the internal flash drive (A:) then the SD card (B:). If the file is found it will be automatically loaded and run, otherwise MMBasic will print a prompt (“>”) and wait for input. If the Maximite has been restarted because the watchdog timer has timed out and forced the Maximite to restart the file "RESTART.BAS" will be run instead (see the WATCHDOG command for details).

Note that the video output will go blank for a short time while writing data to the internal flash drive A:. This is normal and is caused by a requirement to shut off the video while reprogramming the memory. When using

drive A: you need to be careful not to wear out the flash (the same applies to SD cards). If drive A: is empty, you could write and delete a file on it every day for 175 years before you would reach the endurance limit - but if the interval was once a minute you would reach the limit in about 6 weeks.

Storage Commands and Functions

A program can be saved to either drive using the SAVE command. It can be reloaded using LOAD or merged with the current program using MERGE. A saved program can also be loaded and run using the RUN command. The RUN command can also be used within a running program, which enables one program to load and transfer control to another. The CHAIN command allows a program to load and run another program while retaining the current state of the program (ie, the value of variables, open files, loaded fonts, open COM ports, etc). As long as a program can be broken down into modules CHAIN allows programs of almost unlimited size to be run, even with limited memory.

The LIBRARY command will load a file containing user defined commands and functions which can then be called by a running program. This provides an easy way to extend the language by creating specialised libraries for maths functions, hardware drivers, etc. The LIBRARY command can also be used to conserve memory by loading or unloading sections of a program as the program is run.

Data files can be opened using OPEN and read from using INPUT, LINE INPUT, or INPUT$() or written to using PRINT or WRITE. On the SD card both data and programs are stored using standard text and can be read and edited in Windows, Apple Mac, Linux, etc. An SD card can have up to 10 files open simultaneously while the internal flash drive has a maximum of one file open at a time.

You can list the programs stored on a drive with the FILES command, delete them using KILL and rename them using NAME. On an SD card the current working directory can be changed using CHDIR. A new directory can be created with MKDIR or an old one deleted with RMDIR.

Whenever specified a file name can be a string constant (ie, enclosed in double quotes) or a string variable. This means you must use double quotes if you are directly specifying a file name. Eg, KILL “TEST.BAS”. However, quote marks are optional if the command is used at the command prompt. Note that symbols (such as +, -, *) and valid keywords in an unquoted file name will cause an error. Quotes are always required if the command is used within a program.

Timing

You can get the current date and time using the DATE$ and TIME$ functions and you can set them by assigning the new date and time to them. The Colour Maximite with the optional battery backed clock will never loose the time, on other Maximites the calendar will start from midnight 1st Jan 2000 on power up. On the DOS version it will use the system time.

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

Using SETTICK in the Maximite versions you can setup up to four “ticks” which will generate regular interrupts with a period from one millisecond to over a month. See Interrupts below.

Expressions

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

FNAME$ = “TEST”: RUN FNAME$ + ”.BAS”

Structured Statements

MMBasic supports a number of modern structured statements.

The DO WHILE … LOOP command and its variants make it easy to build loops without using the GOTO statement. 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. For example:

```
IF <condition> THEN              ' start a multiline IF
  <statements>
ELSEIF <condition> THEN          ' the ELSEIF is optional
  <statements>
ELSE                             ' the ELSE is optional
  <statements>
ENDIF                            ' must be used to terminate the IF
```

Full Screen Editor

An important productivity feature of MMBasic is the full screen editor (this is not available in the DOS version of MMBasic). It will work using an attached video screen (VGA or composite) and over USB with a VT100 compatible terminal emulator (Tera Term is recommended).

```
''''''
```

F1: SAVE	This will save the program to program memory and return to the command prompt. If you are editing a disk file it will save that file to the disk.
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 repeatedly search for the same text 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).
CTRL-F	This will insert (at the current cursor position) a file residing on the disk. Note that while this key is not listed on the status line it is always available.

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

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

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.

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

The editor is a very productive method of writing a program. Using the OPTION Fnn command you can program a function key to generate the command "EDIT" when pressed. So, with one key press you can jump into the editor where you can make a change. Then by pressing the F2 key you can save and run the program. If your program stops with an error you can press the edit function key and be back in the editor with the cursor positioned at the line that caused the error. This edit/run/edit cycle is very fast.

If you are using the full screen editor over USB with Terra Term you must set Terra Term to a screen size of 80 characters by 36 lines. See Appendix G for details.

Note that a terminal emulator like Tera Term can loose its position in the text with multiple fast keystrokes (like the up and down arrows). If this happens you can press the HOME key twice which will force the editor to jump to the start of the program and redraw the display.

Input/Output

The following functions are only supported on the Maximite variants (not on the DOS version).

External Input/Output

You can configure an external I/O pin using the SETPIN command, set its output using the PIN()= command and read the current input value using the PIN() function. Digital I/O uses the number zero to represent a low voltage and any non-zero number for a high voltage. An analogue input will report the measured voltage as a floating point number.

The original Maximite has 20 I/O pins numbered 1 to 20. Pins 1 to 10 can be used for analog input and digital input/output with a maximum input voltage of 3.3V. Pins 11 to 20 are digital only but support input voltages up to 5V and can be set to open collector.

The DuinoMite has completely different and confusing allocations. See "DuinoMite MMBasic ReadMe.pdf"

Normally digital output is 0V (low) to 3.3V (high) but you can use open collector to drive 5V circuit. This means that the pin can be pulled down (when the output is low) but will go high impedance when the output is high. So, with a pull up resistor to 5V an output configured as open collector you can drive 5V logic signals. Typical value of the pull up resistor is 1K to 4.7K.

Arduino Connector

In addition to the 20 I/O pins described above the Colour Maximite has an extra 20 I/O pins on the Arduino compatible connector (40 I/O pins in total). These are labelled D0 to D13 and A0 to A5.

You can use the labels D0, D1, etc in the SETPIN and PIN statements or you can use their corresponding numbers (D0 = 21, D1 = 22, etc and A0 = 35, A1 = 36, etc). The digital pins (D0 to D13) have the same characteristics (5V, open collector, etc) as the digital pins 11 to 20 and the analog capable pins (A0 to A5) have the same capabilities as pins 1 to 10.

Communications

Two serial ports are supported with speeds up to 19200 baud with configurable buffer sizes and optional hardware flow control. The serial ports are opened using the OPEN command and any command or function that uses a file number can be used to send and receive data. See Appendix A for a full description.

Communications to slave or master devices on an I^2C bus is supported with eight commands (see Appendix B for a full description). MMBasic fully supports bus master and slave mode, 10 bit addressing, address masking and general call, as well as bus arbitration (ie, bus collisions in a multi-master environment).

The Serial Peripheral Interface (SPI) communications protocol is supported with the SPI command. See Appendix D for the details. The Dallas 1-Wire protocol is also supported. See Appendix C for details.

Interrupts

Most external I/O pins can be configured to generate an interrupt using the SETPIN command with many interrupts (including the tick interrupt) active at any one time. Interrupts can be set up to occur on a rising or falling digital input signal and will cause an immediate branch to a specified line number, label or a user defined subroutine. The target can be the same or different for each interrupt. Return from an interrupt is via the IRETURN statement except where a user defined subroutine is used (in that case END SUB or EXIT SUB is used). Within the interrupt routine GOTO, GOSUB and calls to other subroutines can be used.

If two or more interrupts occur at the same time they will be processed in order of pin numbers (ie, an interrupt on pin 1 will have the highest priority). During processing of an interrupt all other interrupts are disabled until the interrupt routine returns with an IRETURN. During an interrupt (and at all times) the value of the interrupt pin can be accessed using the PIN() function.

Up to four periodic interrupts (or regular “ticks”) with periods specified in milliseconds can be setup using the SETTICK statement. This interrupt has the lowest priority. Using the ON KEY command an interrupt can be generated whenever a key is pressed.

Interrupts can occur at any time but they are disabled during INPUT statements. If you need to get input from the keyboard while still accepting interrupts you should use the INKEY$ function or the ON KEY command. 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.

For most programs MMBasic will respond to an interrupt in under 100µS. To prevent slowing the main program by too much an interrupt should be short and exit as soon as possible. Also remember to disable an interrupt when you have finished needing it – background interrupts can cause strange and non-intuitive bugs.

Audio and PWM Output

On the Maximite variants there are a number of ways that you can use the sound output. You can play synthesised music, generate tones or generate program controlled voltages (PWM).

PLAYMOD

This command will play synthesised music in the background while the program is running. The music must be in the MOD format and the file containing the music must be located on the internal flash drive (drive A:). The audio is high quality and MMBasic will generate full stereo on the Colour Maximite.

The MOD format is a music file format originating from the MOD file format on Amiga systems in the late 1980s. It is not a recording of the music (like a MP3 file) - instead it contains instructions for synthesising the music. On the original Amiga this task was performed by dedicated hardware.

MMBasic will read this file and continuously play the music in the background while the program that launched the music will continue running in the foreground. Be aware that synthesising music is a CPU intensive activity and uses a lot of memory and this could affect the performance of the program.

A description of the MOD format can be found at: http://en.wikipedia.org/wiki/MOD_(file_format)

A large selection of files that can be played on the Maximite can be found at: http://modarchive.org (look for files with the .MOD extension). Because the file must be located on drive A: to play it would be wise to select reasonably small files.

You can also create your own music using a tracker. For an example see: http://www.modplug.com

TONE

This command will create two tones for the Colour Maximite that will be outputted separately on the left and right sound channels. On the monochrome Maximite only one tone is generated. The tone is a synthesised sine wave and can be in the range of 1Hz to 20KHz with a resolution of 1Hz and is very accurate as it is locked to the PIC32's crystal oscillator. When the frequency is changed there is no interruption in the output so the output can be made to glide smoothly across a range of frequencies.

The playing time can be specified in milliseconds and the tone will play in the background (ie, the program continues running).

SOUND

The sound command is included only for compatibility with older programs. It generates a single frequency square wave and should be replaced with the tone or PWM command in new programs.

PWM

The PWM (Pulse Width Modulation) command allows the Maximite 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 Colour Maximite has two PWM outputs while the monochrome Maximite has one.

The frequency for both PWM outputs is the same and can be from 20Hz to 1MHz. The duty cycle for each output can be independently set from between 0% and 100% with a 0.1% resolution when the frequency is below 50KHz (above 50KHz the resolution is 1% or better up to 500KHz).

When the Maximite 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.

This command uses the sound output for generating the PWM signal so the components on this output may need to be changed or removed to allow this output to work as a PWM output.

Special Hardware Devices

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

These are:

- Infrared remote control receiver and transmitter
- Ultrasonic distance sensor
- LCD display modules
- The DS18B20 temperature sensor
- Numeric keypads
- Battery backed clock
- Rotary Encoder

Infrared Remote Control Decoder

You can easily add a remote control to your project using the IR command. When enabled this function will run in the background and interrupt the running program whenever a key is pressed on the IR remote control. It will work with any Sony compatible remote control including ones that generate 12, 15 or 20 bit messages. Most cheap programmable remote controls will generate these commands and using one of these you can add a sophisticated flair to your project.

To detect the IR signal you need an IR receiver connected to pin 12 on the Colour Maximite and TFT-Maximite, pin 14 on the monochrome Maximite and pin 7 on the DuinoMite. This is illustrated in the diagram on the right. The IR receiver will sense the IR light, demodulate the signal and present it as a TTL voltage level signal to this pin. Setup of the I/O pin is automatically done by the IR command.

Sony remotes use a 40KHz modulation frequency but receivers for that frequency can be hard to find. Generally 38KHz receivers will work fine but maximum sensitivity will be achieved with a 40KHz device such as the Vishay TSOP4840. Examples of 38KHz receivers that work include the Vishay TSOP4838, Jaycar ZD1952 and Altronics Z1611A.

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 label 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

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

Sony 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 Maximite or Micromite[†] communications 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 Maximite is limited to about 14mA. 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 on the Maximite 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 in the previous page) for maximum sensitivity when communicating with another Maximite or Micromite.

Measuring Temperature

The DS18B20() 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.

The DS18B20 can be powered separately by a 3V to 5V supply or it can operate on parasitic power from the Maximite as shown on the right. Multiple sensors can be used but a separate I/O pin and pullup resistor is required for each one.

To get the current temperature you just use the DS18B20() function in an expression.

For example:

```
PRINT "Temperature: " DS18B20(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 0.25ºC and is accurate to ±0.5 ºC.

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 DS18B20 START command then later use the DS18B20() function to retrieve the temperature reading. The DS18B20() function will always wait if the sensor is still making the measurement.

For example:

```
DS18B20 START 15
< do other tasks >
PRINT "Temperature: " DS18B20(15)
```

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

[†] For details of the Micromite go to http://geoffg.net/micromite.html

(jaycar.com.au). eBay is another good source where prices can range from $10 to $50.

To setup the display you use the LCD INIT command:

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

D4, d5, d6 and d7 are 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 can be used and you do not have to set them up beforehand (the LCD command automatically does that for you). The following diagram shows a typical set up.

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

```
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.

```
LCD INIT 2, 3, 4, 5, 23, 24
LCD 1, 2, "Temperature"
LCD 2, 6, STR$(DS18B20(15))      ' DS18B20 connected to pin 15
```

Note that this example also uses the DS18B20 function to get the temperature (described above).

Keypad Interface

A keypad is a simple method of entering data into an MMBasic based system. MMBasic 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 label of the interrupt 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. Note that these must be pulled up to 3.3V by individual 10K resistors (see the diagram below). 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 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 above). Then the interrupt will be called.

For example:

```
Keypad KeyCode, KP_Int, 2, 3, 4, 5, 21, 22, 23   ' 4x3 keyboard
DO
  < body of the program >
LOOP

KP_Int:                                ' a key press has been detected
  PRINT "Key press = " KeyCode
  IRETURN
```

Measuring Distance

Using a HC-SR04 ultrasonic sensor and the 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.

In MMBasic you use the DISTANCE function:

```
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 "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 or -1 if no target was detected. If you repeatedly call this function you must arrange for a delay of at least 60mS between each call otherwise errors may occur (caused by an echo from the previous sound pulse).

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. You can use multiple sensors connected to different I/O pins or even sharing the one trigger pin if care is taken to prevent one sensor from interfering with another.

Rotary Encoders

A rotary encoder is a handy method of adjusting the value of parameters in a microcontroller project. A typical encoder can be mounted on a panel with a knob and looks like a potentiometer. As the knob is turned it generates a series of signals known as a Gray Code. The program fragment below shows how to decode this code to update a variable in MMBasic.

A standard encoder has two outputs (labelled RA and RB) and a common ground. The outputs should be wired with pullup resistors as shown below:

And this program fragment can be used to decode the output:

```
SETPIN 3, DIN                  ' setup RB as an input
SETPIN 2, INTH, RInt           ' setup an interrupt when RA goes high

DO
  < main body of the program >
LOOP

RInt:                          ' Interrupt to decode the encoder output
  IF PIN(3) = 1 then
     Value = Value + 1         ' clockwise rotation
  ELSE
     Value = Value - 1         ' anti clockwise rotation
  ENDIF
  IRETURN
```

This program assumes that the encoder is connected to I/O pins 2 and 3 however any pins can be used by changing the pin numbers in the program. "Value" is the variable whose value will be updated as the shaft of the encoder is rotated.

Note that this program is intended for simple user input where a skipped or duplicated step is not considered important. It is not suitable for high speed or precision input.

Program courtesy TZAdvantage on the Back Shed Forum.

Real Time Clock Interface

The original monochrome Maximite and the DuinoMite do not have a built in real time clock, which means that the time is lost when the power is removed. Using the RTC GETTIME command and a PCF8563 real time clock integrated circuit you can automatically reset the Maximite's clock on start up.

The PCF8563 is popular and cheap and will keep accurate time to about ±50 ppm even with the power removed (it is battery backed). It can be purchased for as cheap as $3 on eBay and complete modules using the PCF8563 along with a battery can be found for as little as $8.

The PCF8563 is an I^2C device and should be connected to the I^2C I/O pins of the Maximite. Also, because the PCF8563 draws very little current (even when communicating via I^2C) it can be permanently connected to the battery (typical battery life is 15 years).

This circuit shows a typical application (the pin numbers are for the monochrome Maximite).

The 32pF adjustable capacitor should be used to trim the crystal frequency for very accurate timekeeping but that can be tedious as it will involve checking the time for drift over days and weeks. If you don't want to do that you can substitute a 10pF capacitor or leave it out completely and the timekeeping will still be reasonably accurate.

Before you can use the PCF8563 its time must be first set. That is done with the RTC SETTIME command which uses the format RTC SETTIME year, month, day, hour, minute, second. Note that the year is just the last two digits (ie, 14 for 2014) and hour is in 24 hour format. For example, the following will set the PCF8563 to 4PM on the 10th November 2014:

```
RTC SETTIME 14, 11, 10, 16, 0, 0
```

To get the time you use the RTC GETTIME command which will read the time from the PCF8563 and set the clock inside the Maximite. Normally this command will be placed at the beginning of the program so that the time is set on power up.

Graphics and Working with Colour

Graphics

Graphics commands operate on the video output only (not USB). Coordinates are measured in pixels with x being the horizontal coordinate and y the vertical coordinate. The top left of the screen is at location x = 0 and y = 0 with increasing positive numbers representing movement down the screen and to the right. The number of pixels on the screen is defined by the read-only variables MM.HRES and MM.VRES which change depending on the video mode selected (VGA or composite PAL/NTSC).

You can clear the screen with CLS and an individual pixel can be turned on or off and its colour set with PIXEL(x,y) = . You can draw lines and boxes with LINE, and circles using CIRCLE. You can also set the screen location (in pixels) of the PRINT output using @(x,y) and the SAVEBMP command will save the current screen as a BMP file. LOADBMP will load and display a bitmap image stored on the SD card.

Working with Colour

The Colour Maximite supports eight colours (black, blue, green, cyan, red, purple, yellow and white). The monochrome Maximite or DuinoMite support just two (black and white). In most places you can also specify the colour as -1 to invert a pixel (this is useful in animation).

Throughout MMBasic you can refer to the colours by their name or their corresponding numbers where black = 0, blue = 1, green = 2, etc through to white = 7. Commands such as LINE and CIRCLE use this colour or number to specify the colour to draw. For example:

`CIRCLE (100, 100), 50, CYAN`	will draw a circle in cyan.
`CIRCLE (100, 100), 50, 3`	will also draw a circle in cyan (colour = 3).

You can also specify a default colour that will be used for all screen output with the COLOUR command. For example: `COLOUR PURPLE` will set the colour of text to purple (and any other output where the colour is not specified). The COLOUR command also takes a second parameter for the background colour. So, `COLOUR YELLOW, BLUE` specifies that text will be displayed in yellow on a blue background.

In addition to the COLOUR command you can change the colour of text by embedding colour codes into strings using the CLR$() function. For example, the following will display each word in a different colour:

```
Txt$ = "This is " + CLR$(RED) "red " + CLR$(YELLOW) + "yellow"
PRINT Txt$
```

You can also use this function to set the background colour by supplying a second parameter. For example:

```
PRINT CLR$(YELLOW,RED) " ALARM "
```

If the function is used without any parameters (eg, CLR$()) it will reset the colours to the defaults set by the last COLOUR command. The colours are also reset when the print command terminates.

This function simply generates a two character string where the first character is the number 128 plus the foreground colour number and the second character is the number 192 plus the background colour number. You can use this trick to embed colour commands in any text, even text read from a text file on the SD card.

Colour Modes

The video system can be configured into one of four modes using the MODE command. This enables the programmer to trade off the number of colours used on the screen and the graphic resolution against the amount of memory required by the video driver. Modes 1 and 4 use the least amount of memory while mode 3 uses the most. The syntax of the MODE command is: MODE colour-mode, palette

The 'colour-mode' can be one of four numbers:

1 Monochrome mode. In this mode the Colour MMBasic operates the same as the monochrome MMBasic for the Maximite and has the maximum amount of free memory available for programs and data. The second argument of the MODE command ('palette') selects the colour to be used for all output. It can be any colour number from black to white.

2 Four colour mode. In this mode four colours (including black) are available. The actual colours are selected by a number (1 to 6) used in the second argument of the MODE command ('palette'). See the following image or the MODE command for a listing of the actual colours available.

3 Eight colour mode. In this mode all eight colours are available and can be used simultaneously anywhere on the screen. The 'palette' argument is not required and will be ignored if specified. MODE 3 uses the most memory but there still is plenty left for programs and data. This is the default when the Colour Maximite is first powered up.

4 240x216 pixel mode. In this mode all eight colours are available and the video resolution is halved (meaning that characters and graphics are doubled in size). This mode is most suitable for games as all colours are available, it has the maximum amount of free memory and drawing of graphics is very fast. The 'palette' argument is not required and will be ignored if specified.

This is an illustration of what the colour modes look like (Mode 3 was used to generate this image):

You can change the mode and the palette at any time and as often as you need, even within a running program.

Scan Line Colour Override

In mode 1 (monochrome) there is an additional facility to change the colour of each horizontal line of pixels on the screen using the SCANLINE command. This is intended mostly for programmers writing games and provides limited control over colour while still providing the maximum amount of free memory. The syntax is:

SCANLINE colour, startline, endline

This command can only be used in MODE 1, 7 (monochrome with the colour set to white) and is used to set the colour for each horizontal scan line of pixels on the screen. 'colour' is the colour to be used and can be any one of the eight colours, 'startline' is the starting scan line to be set to that colour and 'endline' is the ending line. The scan lines are numbered from 0 at the top of the screen to 431 at the bottom of the screen. The numbering is the same as that used when specifying the vertical coordinates of a pixel.

You can use multiple SCANLINE commands to set multiple scan lines to different colours. For example:

```
SCANLINE RED, 0, 9          ' set the top 10 lines to red
SCANLINE YELLOW, 120        ' and set only line 120 to yellow
SCANLINE BLUE, 200, 219     ' and set a band of 20 lines to blue
```

To turn off the override imposed by the use of SCANLINE commands you can use the MODE command to reselect mode 1 or a change to a different mode. It is also automatically turned off when control is returned to the command prompt.

Game Playing Features

MMBasic 4.x introduces a number of features that are intended to make it easier to write games on the Maximite.

MODE 4

The colour MODE 4 described in the previous section is mostly intended for games. It provides eight colours and leaves plenty of free memory for the other aspects of an animated game (sprites, arrays, and so on).

Because this colour mode has only one quarter of the pixels the graphics operations are much faster due to the fact that there are fewer pixels that need to be manipulated by MMBasic when drawing on the screen.

BLIT

This command will move an area of the video screen from one location to another. The destination can overlap the source area and the BLIT command will copy the video data correctly to avoid corruption. On the Colour Maximite you can also independently specify what colour planes to copy.

This method of moving video data is much faster than copying pixels one by one and allows for rapid animation on the screen. It can also be used to replicate a pattern like a border or a brick wall to build a complete image.

SPRITES

A sprite is a 16x16 bit graphic image that can be moved about on the screen independently of the background. When the sprite is displayed MMBasic will automatically save the background text and graphics under the sprite and when the sprite is turned off or moved MMBasic will restore the background.

The sprites are defined in a file which is loaded into memory using the SPRITE LOAD command, the number of sprites contained in the file is only limited by the amount of available memory. Each sprite in the file can contain pixels of any colour (on the Colour Maximite) and can also have transparent pixels which allow the background to show through.

A special function exists to report if the sprite has collided with the edge of the screen or other sprites.

See Appendix H for a detailed description of creating and manipulating sprites.

LOADBMP

The LOADBMP command will load a colour or monochrome bitmap image and display it at a specified location on the screen. This is handy for loading background images for games.

FONTS

The FONT command is normally used to load custom alphanumeric character fonts but a font's character can be as large as 64 pixels high and 255 pixels wide.

This means that a specially designed font can be used to load custom designed graphic images and display them anywhere on the screen at high speed.

PEEK/POKE

With the PEEK and POKE commands you can now use constant keywords to access special sections of memory (like the video memory) and these keywords will be valid with future versions of MMBasic. This makes it easy to access internal MMBasic data structures in a portable manner.

KEYDOWN FUNCTION

They KEYDOWN function makes it easy to tell if the user is holding down a key on the PS2 keyboard (like an arrow key). While the key is held down KEYDOWN will return the numeric value of the key, when no key is held down the function will return zero.

The KEYDOWN function will also remove any characters in the keyboard input buffer but, when playing a game, the user often still has their finger on a key when the game ends. For that reason the program should include the following line which will wait for the user to release the key and clear the buffer:

```
DO WHILE KEYDOWN AND INKEY$ <> "" : LOOP
```

Defined Subroutines and Functions

Defined subroutines and functions are useful features to help in organising programs so that they are easy to modify and read. A 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.

For example, assume that you would like to have the command FLASH added to MMBasic, its job would be to flash the power light on the Maximite. You could define a subroutine like this:

```
Sub FLASH
  Pin(0) = 1
  Pause 100
  Pin(0) = 0
End Sub
```

Then, in your program you just use the command FLASH to flash the power LED. For example:

```
IF A <= B THEN FLASH
```

If the FLASH subroutine was in program memory you could even try it out at the command prompt, just like any command in MMBasic. The definition of the FLASH subroutine can be anywhere in the program but typically it is at the start or end. If MMBasic runs into the definition while running your program it will simply skip over it.

Subroutine Arguments

Defined subroutines can have arguments (sometimes called parameter lists). 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 different from arguments defined for the subroutine (at the risk of making debugging harder).

When calling a subroutine you can supply less than the required number of values. For example:

```
MYSUB  23
```

In that case the missing values will be assumed to be either zero or an empty string. For example, in the above case `arg2$` will be set to `""` and `arg3` will be set to zero. This allows you to have optional values and, if the value is not supplied by the caller, you can take some special action.

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 `""`.

Local Variables

Inside a subroutine you will need to use variables for various tasks. In portable code you do not want the name you chose for such a variable to clash with a variable of the same name in the main program. To this end you can define a variable as LOCAL.

For example, this is our FLASH subroutine but this time we have extended it to take an argument (nbr) that specifies how many times to flash the LED.

```
Sub FLASH ( nbr )
  Local count
  For count = 1 To nbr
    Pin(0) = 1
    Pause 100
    Pin(0) = 0
```

```
    Pause 150
  Next count
End Sub
```

The counting variable (`count`) is declared as local which means that (like the argument list) it only exists within the subroutine and will vanish when the subroutine exits. You can have a variable called `count` in your main program and it will be different from the variable `count` in your subroutine.

If you do not declare the variable as local it will be created within your program and be visible in your main program and subroutines, just like a normal variable.

You can define multiple items with the one LOCAL command. If an item is an array the LOCAL command will also dimension the array (ie, you do not need the DIM command). For example:

```
LOCAL NBR, STR$, ARR(10, 10)
```

You can also use local variables in the target for GOSUBs. For example:

```
  GOSUB MySub
        ...
MySub:
   LOCAL X, Y
   FOR X = 1 TO ...
   FOR Y = 5 TO ...
  <statements>
   RETURN
```

The variables X and Y will only be valid until the RETURN statement is reached and will be different from variables with the same name in the main body of the program.

Defined Functions

Defined functions are similar to defined subroutines with the main difference being that the function is used to return a value in an expression. For example, if you wanted a function to select the maximum of two values you could define:

```
Function Max(a, b)
  If a > b
    Max = a
  Else
    Max = b
  EndIf
End Function
```

Then you could use it in an expression:

```
SetPin 1, 1 : SetPin 2, 1
Print "The highest voltage is" Max(Pin(1), Pin(2))
```

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 (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 $ the function will return a string, otherwise it will return a number. Within the function the function's name acts like a standard variable.

As another example, let us say that you need a function to format time in the AM/PM format:

```
Function MyTime$(hours, minutes)
  Local h
  h = hours
  If hours > 12 Then h = h - 12
  MyTime$ = Str$(h) + ":" + Str$(minutes)
  If hours <= 12 Then
    MyTime$ = MyTime$ + "AM"
  Else
    MyTime$ = MyTime$ + "PM"
  EndIf
End Function
```

As you can see, the function name is used as an ordinary local variable inside the subroutine. It is only when the function returns that the value assigned to `MyTime$` is made available to the expression that called it. This example also illustrates that you can use local variables within functions just like subroutines.

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 in your routine 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.

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 will be "magically" changed by your routine. It is much safer to assign the argument to a local variable and manipulate that instead.

Additional Notes

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.

You cannot use arrays in a subroutine or function's argument list however the caller can use them. 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.

Loadable Libraries

The use of defined subroutines and functions should reduce the need to add specialised features to MMBasic. For instance, there have been a few requests to add bit shifting functions to the language. Now you can do that yourself… this is the right shift function:

```
Function RShift(nbr, bits)
   If nbr < 0 or bits < 0 THEN ERROR "Invalid argument"
   RShift = nbr\(2^bits)
End Function
```

You can now use this function as if it is a part of the language:

```
a = &b11101001
b = RShift(a, 3)
```

After running this fragment of code the variable b would have the binary value of `11101.`

The defined subroutine and function is intended to be a portable lump of code that you can insert into any program. This is why MMBasic has the LIBRARY command which allows you to load files containing user defined subroutines and functions into memory. These functions/subroutines are then available to the running program and are indistinguishable from the built-in commands and functions.

So, it would be easy to create a library of bit manipulation functions like that described above and load them within any program that might need them. The same for specialised maths functions, drivers for special hardware and so on.

Implementation Details

Naming Conventions

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.

There are two types of variable: numeric which stores a floating point number (eg, 45.386), and string which stores a string of characters (eg, “Tom”). String variable names are terminated with a $ symbol (eg, name$) while numeric variables are not.

Variable names and labels can start with an alphabetic character or underscore and can contain any alphabetic or numeric character, the period (.) and the underscore (_). They may be up to 32 characters long. A variable name or a label must not be the same as a 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. In addition, a label cannot be the same as a command name.

Constants

Numeric constants may begin with a numeric digit (0-9) for a decimal constant, &H for a hexadecimal constant, &O for an octal constant or &B for a binary constant. For example &B1000 is the same as the decimal constant 8.

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

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

Operators and Precedence

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

Arithmetic operators:

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

Logical operators:

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

The operators AND, OR and XOR are bitwise operators. For example PRINT 3 AND 6 will output 2.

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

The NOT operator is highest in precedence so it will bind tightly to the next value. For normal use the expression to be negated should be placed in brackets. For example, IF NOT (A = 3 OR A = 8) THEN …

String operators:

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

Implementation Characteristics

Maximum length of a command line is 255 characters.

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

Maximum number of dimensions to an array is 8.

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

Maximum number of nested FOR…NEXT loops is 20.

Maximum number of nested DO…LOOP commands is 20.

Maximum number of nested GOSUBs is 100.

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

Maximum number of user defined subroutines and functions (combined): 64

Numbers are stored and manipulated as single precision floating point numbers. The maximum number that can be represented is 3.40282347e+38 and the minimum is 1.17549435e-38

The range of integers (whole numbers) that can be manipulated without loss of accuracy is ±16777100.

Maximum string length is 255 characters.

Maximum line number is 65000.

Maximum length of a file pathname (including the directory path) is 255 characters.

Maximum number of files simultaneously open is 10 on the SD card and one on the internal flash drive (A:).

Maximum SD card size is 2GB formatted with FAT16 or 2TB formatted with FAT32.

Size of the internal flash drive (A:) is 180KB on the monochrome Maximite (less with colour or CAN).

Maximum size of a loadable video font is 64 pixels high x 255 pixels wide and 107 characters.

Maximum number of library files that can be loaded simultaneously is 8.

Maximum number of background pulses launched by the PULSE command is 5.

Compatibility

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

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

License

MMBasic is Copyright 2011-2014 by Geoff Graham - http://mmbasic.com.

The compiled object code (the .hex file) is free software: you can use or redistribute it as you please.

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

The source code is available via subscription (free of charge) to individuals for personal use or under a negotiated license for commercial use. In both cases go to http://mmbasic.com for details.

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

Predefined Read Only Variables

The centre column specifies the platform (CMM is the Colour Maximite, MM is the monochrome Maximite and DuinoMite, DOS is the Windows version).

MM.HRES MM.VRES	CMM MM	The horizontal and vertical resolution of the current video display screen in pixels.
MM.HPOS MM.VPOS	CMM MM	The current horizontal and vertical position (in pixels) following the last graphics or print command.
MM.VER	CMM MM DOS	The version number of the firmware in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number (normally zero but A = 01, B = 02, etc).
MM.DEVICE$	CMM MM DOS	A string representing the device or platform that MMBasic is running on. Currently this variable will contain one of the following: "Maximite" on the standard Maximite and compatibles. "Colour Maximite" on the Colour Maximite and UBW32. "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "Generic PIC32" for the generic version of MMBasic on a PIC32.
MM.DRIVE$	CMM MM	The current default drive returned as a string (either “A:” or “B:”).
MM.FNAME$	CMM MM DOS	The name of the file that will be used as the default for the SAVE command. This is set by LOAD, RUN and SAVE.
MM.CMDLINE$	CMM MM DOS	The command line used with the implied RUN command. See the implied RUN command at the start of the next page for the details.
MM.ERRNO	CMM MM DOS	Is set to the error number if a statement involving the SD card fails or zero if the operation succeeds. This is dependent on the setting of OPTION ERROR. For the Maximite (colour and monochrome) the possible values for MM.ERRNO are: 0 = No error 1 = No SD card found 2 = SD card is write protected 3 = Not enough space 4 = All root directory entries are taken 5 = Invalid filename 6 = Cannot find file 7 = Cannot find directory 8 = File is read only 9 = Cannot open file 10 = Error reading from file 11 = Error writing to file 12 = Not a file 13 = Not a directory 14 = Directory not empty 15 = Hardware error accessing the storage media 16 = Flash memory write failure

Commands

The centre column specifies the platform (CMM is the Colour Maximite, MM is the monochrome Maximite and DuinoMite, DOS is the Windows version). Square brackets indicate that the parameter or characters are optional.

program command-line	CMM MM DOS	Implied RUN command. At the command prompt it is possible to omit the RUN keyword and MMBasic will search the default drive and directory for a matching program and if found, it will be run. 'program' is the program name. If there is no extension the extension .BAS will be automatically appended. 'command-line' is an arbitrary string of characters which can be retrieved by the new program using the read only variable MM.CMDLINE$. Example: `SORT INFILE.TXT OUTFILE.TXT` Will run the program SORT.BAS and the variable MM.CMDLINE$ will hold the string: "`INFILE.TXT OUTFILE.TXT`" Notes: • The implied RUN command is valid only at the command prompt (not within a running program). • If 'program' is already in memory it will be run directly without reloading it from disk (even if it has been edited). • If 'program' is the same as an internal MMBasic command the internal command will be run instead. To avoid this 'program' can be surrounded by quote marks. Eg: "PRINT" command line. • If MM.CMDLINE$ returns an empty string this means that a command line was not provided and the new program should then prompt for the parameters that it needs. • To avoid accidently overwriting a program that you are working on an error will be generated if the program needs to be loaded from disk and the program currently in memory has been edited but not saved. If this happens (and you do not want to save the program currently in memory) use the NEW command to clear the memory.
‘ (single quotation mark)	CMM MM DOS	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
? (question mark)	CMM MM DOS	Shortcut for the PRINT command.
AUTO or AUTO start or AUTO start, increment	CMM MM DOS	Enter automatic line entry mode. To terminate this mode use Control-C. With no arguments this command will take lines of text from the keyboard or USB and append them to program memory without modification. This is useful for adding lines that do not have line numbers and when pasting a program into a terminal emulator. If 'start' is provided the lines will be prefixed with an automatically generated line number. 'start' is the starting line number and 'increment' is the step size (default 10). If the automatically generated number is the same as an existing line in memory it will be preceded by an asterisk (*). In this case pressing Enter without entering any text will preserve the line in memory and generate the next number.

BLIT x1, y1, x2, y2, w, h or BLIT x1, y1, x2, y2, w, h, RGB	CMM MM	Copy one section of the video screen to another. 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'. All arguments are in pixels. The source and destination can overlap. Colour Maximite only: If the optional argument 'RGB' is specified then only the specified colour planes will be copied. For example, 'GB' will copy only the green and blue colour planes.
CHAIN file$	CMM MM DOS	Clear the current program from memory, load the new program ('file$') into memory and run it starting with the first line. Unlike the RUN command this command retains the current state of the program (ie, the value of variables, open files, loaded fonts, open COM ports, etc). The only exception is any open interrupts which will be automatically closed. One program can CHAIN to another which can then chain to another (or back to the original) an unlimited number of times. As long as a program can be broken down into modules this command allows programs of almost unlimited size to be run, even with limited memory. Communication between the modules can be accomplished by assigning values to one or more variables which then can be examined by the new chained program. Note that another way of squeezing a large program into limited memory is to use the LIBRARY command.
CHDIR dir$	CMM MM DOS	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.
CIRCLE (x, y) ,r [,c [, aspect [,F]]]	CMM MM	Draw a circle on the video output centred at 'x' and 'y' with a radius of 'r'. 'c' is the optional colour and defaults to the current foreground colour if not specified. 'c' can also be -1 which will invert the pixels. The optional 'aspect' will define the aspect ratio. Because the Maximite's pixels are rectangular an aspect ratio of 0.833 will result in a perfect circle on most monitors. Other ratios can be specified for a variety of ovals. If 'aspect' is not specified the default is 1.0 which is backwards compatible with early versions of MMBasic. The F option can be appended to the end of the argument list and will cause the circle to be filled according to the 'c' parameter. See the section "Graphics and Working with Colour" for a definition of the colours and graphics coordinates.
CLEAR	CMM MM DOS	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
CLOSE [#]nbr [,[#]nbr] …	CMM MM DOS	Close the file(s) or serial port(s) previously opened with the file number ‘nbr’. The # is optional. Also see the OPEN command.
CLOSE CONSOLE	CMM MM	Close a serial port that had been previously opened as the console.
CLS [colour]	CMM MM DOS	Clears the video display screen and places the cursor in the top left corner. Optionally 'colour' can be specified which will be used for the background when clearing the screen (Colour Maximite only).

COLOUR fore [, back] or COLOR fore [, back]	CMM	Sets the default colour for commands that display on the screen (PRINT, LINE, etc). 'fore' is the foreground colour, 'back' is the background colour. The background is optional and if not specified will default to black. The actual colour displayed will depend on the current colour mode (see the MODE command). See "Working with Colour" at the start of this manual for more details.
CONFIG COMPOSITE NTSC | PAL | DISABLED or CONFIG VIDEO OFF | ON or CONFIG FONT 1 | 2 or CONFIG CASE UPPER | LOWER | TITLE or CONFIG KEYBOARD US | UK | FR | GR | BE | IT | ES or CONFIG TAB 2 | 4 | 8 or CONFIG FONT n	CMM MM	Reprogram options into MMBasic. This command differs from other options. It permanently reconfigures MMBasic and it only needs to be run once (ie, the setting will be remembered even with the power turned off). In most cases the power must be cycled after changing a setting for it to take effect. The COMPOSITE setting will enable the composite video output and select the appropriate timing. The default is DISABLED for the Colour Maximite and the DuinoMite and PAL for the Original (monochrome) Maximite. The VIDEO setting will switch the video output on or off. There is a performance improvement with the video off but the biggest benefit is that the unused memory is returned to the memory pool. Default is ON. The FONT setting will select the default font used by MMBasic (a reboot is required after this command). The default is FONT 1. The CASE setting will change the case used for listing command and function names when using the LIST command. The default is TITLE but the old standard of MMBasic can be restored using CONFIG CASE UPPER. The KEYBOARD setting will change the keyboard layout to suit standard keyboards (US), United Kingdom (UK), French (FR), German (GR), Belgium (BE), Italian (IT) or Spanish (ES) keyboards. Default is US. The TAB setting will set the spacing for the tab key. Default is 2. The FONT setting will set the default font to 'n' which can be either 1 (standard font of 10 x 5 pixels) or 2 (a larger font of 16 x 11 pixels). The default is 1 but font 2 is useful with small displays
CONTINUE	CMM MM DOS	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The program will restart with the next statement following the previous stopping point.
COPY src$ TO dest$	CMM MM DOS	Copy the file named 'src$' to another file named 'dest$'. 'dest$' can be just a drive designation (ie, A:) and this makes it convenient to copy files between drives.
COPYRIGHT	CMM MM DOS	List all contributors to MMBasic and summarise the copyright.
DATA constant[,constant]...	CMM MM DOS	Stores numerical and string constants to be accessed by READ. In general string constants should be surrounded by double quotes ("). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as THEN, WHILE, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.
DATE$ = "DD-MM-YY" or DATE$ = "DD/MM/YY"	CMM MM	Set the date of the internal clock/calendar. DD, MM and YY are numbers, for example: DATE$ = “28-7-2012” Normally the date is set to “1-1-2000” on power up. If the battery backed clock option is fitted to the Colour Maximite the current date will be automatically set on power up.

DELETE line DELETE -lastline DELETE firstline - DELETE firstline - lastline	CMM MM DOS	Deletes a program line or a range of lines. If '–lastline' is used it will delete from the start of the first line in the program to the end of 'lastline'. If 'startline-' is used it will delete from start of 'startline' to the end of the program. Also see the NEW command.
DIM var(dim) , [var(dim)]... or DIM var$(dim) LENGTH n Examples: DIM nbr(50) DIM str$(20) DIM a(5,5,5), b(1000) DIM str$(200) LENGTH 20	CMM MM DOS	Specifies a variable that is an array with one or more dimensions. The variables can be numbers or strings with multiple declarations separated by commas. 'dim' is a bracketed list of numbers separated by commas. Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1. For example: DIM nbr(10, 20) specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each number requires 4 bytes a total of 924 bytes of memory will be allocated. String arrays will by default be allocated memory for 255 characters for each element and this can quickly use up memory. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element. This allocation ('n') can be from 1 to 255 characters. For example: DIM str$(5, 10) will declare a string array with 66 elements consuming 16,896 bytes of memory while: DIM str$(5, 10) LENGTH 20 Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element. If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this string arrays created with the LENGTH keyword act exactly the same as other string arrays.
DO <statements> LOOP	CMM MM DOS	This structure will loop forever; the EXIT command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like GOTO or RETURN (if in a subroutine).
DO WHILE expression <statements> LOOP	CMM MM DOS	Loops while "expression" is true (this is equivalent to the older WHILE-WEND loop, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, even once.
DO <statements> LOOP UNTIL expression	CMM MM DOS	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is false.
DRIVE drivespec$	CMM MM	Change the default drive used for file operations that do not specify a drive to that specified in 'drivespec$'. This can be the string “A:” or “B:”. See also the predefined read-only variable MM.DRIVE$.
DS18B20 START pin [, precision]	CMM MM	This command can be used to start a conversion running on a DS18B20 temperature sensor connected to 'pin'. Normally the DS18B20() function is sufficient to make a temperature measurement so usage of this command is optional and can be ignored. 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 DS18B20() function to get the reading. 'precision' is the resolution of the measurement and is optional. It is a

		number between 0 and 3 meaning: 0 = 0.5ºC resolution, 100mS conversion time. 1 = 0.25ºC resolution, 200mS conversion time (the default). 2 = 0.125ºC resolution, 400mS conversion time. 3 = 0.0625ºC resolution, 800mS conversion time.
EDIT or EDIT filename or EDIT line-number	CMM MM	Invoke the full screen editor. This can be used to edit either the program currently loaded in memory or a program file. It can also be used to view and edit text data files. If EDIT is used on its own it will edit the program memory. If 'filename' is supplied the file will be edited leaving the program memory untouched. On entry the cursor will be automatically positioned at the last line edited or, if there was an error when running the program, the line that caused the error. If 'line-number' is specified on the command line the program in memory will be edited and cursor will be placed on the line specified. The editing keys are: Left/Right arrows — Moves the cursor within the line. Up/Down arrows — Moves the cursor up or down a line. Page Up/Down — Move up or down a page of the program. Home/End — Moves the cursor to the start or end of the line. A second Home/End will move to the start or end of the program. Delete — Delete the character over the cursor. This can be the line separator character and thus join two lines. Backspace — Delete the character before the cursor. Insert — Will switch between insert and overtype mode. Escape Key — Will close the editor without saving (confirms first). F1 — Save the edited text and exit. F2 — Save, exit and run the program. F3 — Invoke the search function. SHIFT F3 — Repeat the search using the text entered with F3. F4 — Mark text for cut or copy (see below). F5 — Paste text previously cut or copied. CTRL-F — Insert a file into the program being edited. When in the mark text mode (entered with F4) the editor will allow you to use the arrow keys to highlight text which can be deleted, cut to the clipboard or simply copied to the clipboard. The status line will change to indicate the new functions of the function keys. While the full screen editor is running it will override the programmable function keys F1 to F5. When the editor exits all programmable functions will be restored. The editor will work with lines wider than the screen but characters beyond the screen edge will not be visible. You can split such a line by inserting a new line character and the two lines can be later rejoined by deleting the inserted new line character. All the editing keys work with a VT100 terminal emulator so editing can also be accomplished over a USB or serial link. The editor has been tested with Tera Term and this is the recommended software. Note that Tera Term <u>must</u> be configured for an 80 column by 36 line display.

ELSE	CMM MM DOS	Introduces a default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF expression THEN	CMM MM DOS	Introduces a secondary condition in a multiline IF statement. See the multiline IF statement for more details.
ENDIF	CMM MM DOS	Terminates a multiline IF statement. See the multiline IF statement for more details.
END	CMM MM DOS	End the running program and return to the command prompt.
END FUNCTION	CMM MM DOS	Marks the end of a user defined function. See the FUNCTION command. Each sub must have one and only one matching END FUNCTION statement. Use EXIT FUNCTION if you need to return from a subroutine from within its body. Only one space is allowed between END and FUNCTION.
END SUB	CMM MM DOS	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. Only one space is allowed between END and SUB.
ERASE variable [,variable]...	CMM MM DOS	Deletes arrayed variables and frees up the memory. Use CLEAR to delete all variables including all arrayed variables.
ERROR [error_msg$]	CMM MM DOS	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXIT EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	CMM MM DOS	EXIT by itself or EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. Only one space is allowed between the two words.
FILES [fspec$]	CMM MM DOS	Lists files in the current directory on the SD or internal drive (drive A:). The SD card (drive B:) may use an optional ‘fspec $’. 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
FONT [#]nbr or FONT [#]nbr, scale or FONT [#]nbr, scale, reverse	CMM MM	Selects a font for the video output. 'nbr' is the font number in the range of 1 to 10. The # symbol is optional. ‘scale’ is the multiply factor in the range of 1 to 8 (eg, a scale of 2 will double the size of a pixel in both the vertical and horizontal). Default is 1. If ‘reverse’ is a number other than zero the font will be displayed in reverse video. Default is no reverse. There are three fonts built into MMBasic: #1 is the standard font of 10 x 5 pixels containing the full ASCII set.

		#2 is a larger font of 16 x 11 pixels also with the full ASCII set. #3 is a jumbo font of 30 x 22 pixels consisting of the numbers zero to nine and the characters plus, minus, space, comma and full stop. Examples: 10 FONT #3, 2, 1 ‘ double scale and reverse video 10 FONT #3, ,0 ‘ reset to normal video 10 FONT #2 ‘ just select font #2 Font #1 with a scale of one and no reverse is the default on power up and will be reinstated whenever control returns to the input prompt. Other fonts can be loaded into memory: see the FONT LOAD command.
FONT LOAD file$ AS [#]nbr	CMM MM	Loads the font contained in 'file$' and install it as font 'nbr' which can be any number between 3 and 10. The # symbol is optional. Appendix E describes the format of the font file.
FONT UNLOAD [#]nbr	CMM MM	Removes font 'nbr' and frees the memory used. The # symbol is optional. You cannot unload the built-in fonts.
FOR counter = start TO finish [STEP increment]	CMM MM DOS	Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' equals 'finish'. The ‘increment’ must be an integer, but may be negative. See also the NEXT command.

FUNCTION xxx (arg1 [,arg2, …])
<statements>
<statements>
xxx = <return value>
END FUNCTION
	CMM
	MM
	DOS
		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. 'arg1', 'arg2', etc are the arguments or parameters to the function.
		To set the return value of the function you assign the value to the function's name. For example:
		`FUNCTION SQUARE(a)`
		`  SQUARE = a * a`
		`END FUNCTION`
		Every definition must have one END FUNCTION statement. When this is reached the function will return its value to the expression from which it was called. The command EXIT FUNCTION can be used for an early exit.
		You use the function by using its name and arguments in a program just as you would a normal MMBasic function. For example:
		`PRINT SQUARE(56.8)`
		When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function.
		Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string.
		Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable.
		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.

GOSUB target	CMM MM DOS	Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN.

GOTO target	CMM MM DOS	Branches program execution to the target, which can be a line number or a label.
IF expr THEN statement or IF expr THEN statement ELSE statement	CMM MM DOS	Evaluates the expression ‘expr' and performs the THEN statement if it is true or skips to the next line if false. The optional ELSE statement is the reverse of the THEN test. This type of IF statement is all on one line. The ‘THEN statement’ construct can be also replaced with: GOTO linenumber | label’.

IF expression THEN
<statements>
[ELSE
<statements>]
[ELSEIF expression THEN
<statements>]
ENDIF
	CMM
	MM
	DOS
		Multiline IF statement with optional ELSE and ELSEIF cases and ending with ENDIF. Each component is on a separate line.
		Evaluates 'expression' and performs the statement(s) following THEN if the expression is true or optionally the statement(s) following the ELSE statement if false.
		The ELSEIF statement (if present) is executed if the previous condition is false and it starts a new IF chain with further ELSE and/or ELSEIF statements as required.
		One ENDIF is used to terminate the multiline IF.

INPUT ["prompt string$";] list of variables	CMM MM DOS	Allows input from the keyboard to a list of variables. The input command will prompt with a question mark (?). The input must contain commas to separate each data item if there is more than one variable. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 If the "prompt string$" is specified it will be printed before the question mark. If the prompt string is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #nbr, list of variables	CMM MM DOS	Same as above except that the input is read from a file previously opened for INPUT as ‘nbr’. See the OPEN command.
IR dev, key , interrupt or IR CLOSE	CMM MM	Decodes 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 pin 12 on the Colour Maximite and TFT-Maximite, pin 14 on the monochrome Maximite and pin 7 on the DuinoMite. 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). 'interrupt' is the line number or label of the interrupt routine 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 routine the program can examine the variables 'dev' and 'key' and take appropriate action. The IRETURN command is used to return from the interrupt. A subroutine can also be specified for the interrupt target and in that case return is via EXIT SUB or END SUB. The IR CLOSE command will terminate the IR decoder and return the I/O pin to a not configured state. See the section "Special Hardware Devices" for more details.

IR SEND pin, dev, key	CMM MM	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.
IRETURN	CMM MM	Returns from an interrupt that used a line number or label. The next statement to be executed will be the one that was about to be executed when the interrupt was detected. Note that IRETURN must not be used where the interrupt routine is a user defined subroutine. In that case END SUB or EXIT SUB is used.
KILL file$	CMM MM DOS	Deletes the file specified by ‘file$’. If there is an extension it must be specified. If 'file' is a string constant quote marks around it are optional at the command prompt but must be specified if the command is used within a program. Example: KILL “SAMPLE.DAT”
KEYPAD var, int, r1, r2, r3, r4, c1, c2, c3 [, c4] or KEYPAD CLOSE	CMM MM	Monitor and decode key presses on a 4x3 or 4x4 keypad. Monitoring of the keypad is done in the background and the program will continue after this command without interruption. 'var' should be a numeric variable and its value will be updated whenever a key press is detected. 'int' is the line number or label of the interrupt routine that will be called when a new key press is received. In the interrupt routine the program can examine the variable 'var' and take appropriate action. The IRETURN command is used to return from the interrupt. A subroutine can also be specified for the interrupt target and in that case return is via EXIT SUB or END SUB. r1, r2, r3 and r4 are pin numbers used for the four row connections to the keypad and c1, c2, c3 and c4 are the column connections. c4 is optional and is only used with 4x4 keypads. This command will automatically configure these pins as required. On a key press the value assigned to 'var' is the number of a numeric key (eg, '6' will return 6) or 10 for the * key and 11 for the # key. On 4x4 keypads the number 20 will be returned for A, 21 for B, 22 for C and 23 for D. The KEYPAD CLOSE command will terminate the keypad function and return the I/O pin to a not configured state. See the section "Special Hardware Devices" for more details.
LET variable = expression	CMM MM DOS	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command.
LCD INIT d4, d5, d6, d7, rs, en or LCD line, pos, text$ or LCD CLEAR or LCD CLOSE	CMM MM	Display text on an LCD character display module. This command will work with 1-line, 2-line or 4-line LCD modules that use the KS0066, HD44780 or SPLC780 controller. 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.
LCD CMD d1 [, d2 [, etc]] or LCD DATA d1 [, d2 [, etc]]	CMM MM	These commands will send one or more bytes to an LCD display as either a command (LCD CMD) or as data (LCD DATA). Each byte is a number between 0 and 255 and must be separated by commas. The LCD must have been previously initialised using the LCD INIT command (see above). These commands can be used to drive a non standard LCD in "raw mode" or they can be used to enable specialised features such as scrolling, cursors and custom character sets. You will need to refer to the data sheet for your LCD to find the necessary command and data values.
LIBRARY LOAD $file and LIBRARY UNLOAD $file	CMM MM DOS	Will load a library file ('file$') into general memory. Any user defined commands and subroutines in the file will then become available to the running program. Up to 8 files can be loaded simultaneously. A library file is like any other MMBasic program, with the exception that any programming code outside the user defined commands and subroutines in the file will be ignored. The library is not visible to the user (it is not listed by the LIST command) so it should be tested and debugged as a normal basic program first. Normally a library file has the extension ".LIB" and that extension will be automatically added if the file name ('$file') does not include the extension. Libraries can be loaded and unloaded in any order. Libraries can be loaded from within other libraries and nested to an unlimited extent. Any library file can be unloaded from memory and the memory returned to the general pool by using the LIBRARY UNLOAD command. Library files must not be unloaded from within a library that is currently being used by the program (the results are undefined but it may crash MMBasic and cause your hair to go prematurely grey). This command can be used to load specialised libraries to extend the functionality of MMBasic. Examples include device drivers, libraries that provide bit manipulation and libraries of specialised mathematical functions. The library file is only loaded on the first load command encountered so it is acceptable to put the same load command into every part of the program or every subroutine that may need the library. Another use of the LIBRARY command is to extend the amount of memory available to a program by only loading sections of code as needed and then unloading them when their task is finished so that another function can be loaded. To prevent fragmentation of memory, functions that use a lot of memory (like COM port buffers and arrays) should be declared first before any libraries are loaded and then unloaded.

LINE [(x1 , y1)] - (x2, y2) [,c [,B[F]]]	CMM MM	Draws a line or box on the video screen. x1,y1 and x2,y2 specify the beginning and end points of a line. 'c' specifies the colour and defaults to the default foreground colour if not specified. It can also be -1 to invert the pixels. (x1, y1) is optional and if omitted the last drawing point will be used. The optional B will draw a box with the points (x1,y1) and (x2,y2) at opposite corners. The optional BF will draw a box (as ,B) and fill the interior. See the section "Graphics and Working with Colour" for a definition of the colours and graphics coordinates.
LINE INPUT [prompt$,] string-variable$	CMM MM DOS	Reads entire line from the keyboard into ‘string-variable$’. If specified the ‘prompt$’ will be printed first. Unlike INPUT, LINE INPUT will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of ‘prompt$’.
LINE INPUT #nbr, string-variable$	CMM MM DOS	Same as above except that the input is read from a file previously opened for INPUT as ‘nbr’. See the OPEN command.
LIST LIST line LIST -lastline LIST firstline - LIST firstline - lastline	CMM MM DOS	Lists all lines in a program line or a range of lines. If –lastline is used it will start with the first line in the program. If startline- is used it will list to the end of the program.
LOAD file$	CMM MM DOS	Loads a program called ‘file$’ from the current drive into program memory. If an extension is not specified “.BAS” will be added to the file name.
LOADBMP file$ [, x, y]	CMM MM	Load a bitmapped image and display it on the video screen. "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 “.BMP” will be added to the file name. The file must use either a monochrome or 16 colours (4 bit) colour depth and be in the uncompressed BMP format. Microsoft's Paint program is recommended for creating suitable images. See also SAVEBMP.
LOCAL variable [, variables]	CMM MM DOS	Defines a list of variable names as local to the subroutine or function. 'variable' can be an array and the array will be dimensioned just as if the DIM command had been used. A local variable will only be visible within the procedure and will be deleted (and the memory reclaimed) when the procedure returns. If the local variable has the same name as a global variable (used before any subroutines or functions were called) the global variable will be hidden by the local variable while the procedure is executed.
LOOP [UNTIL expression]	CMM MM DOS	Terminates a program loop: see DO.

MEMORY	CMM
	MM
	DOS
		List the amount of memory currently in use. For example:
		`15kB (18%) Program (528 lines)`
		`23kB (28%) 52 Variables`
		`17kB (21%) General`
		`28kB (33%) Free`
		Program memory is cleared by the NEW command. Variable and the general memory spaces are cleared by many commands (eg, NEW, RUN, LOAD, etc) as well as the specific commands CLEAR and ERASE.
		General memory is used by fonts, file I/O buffers, etc.

MERGE file$	CMM MM DOS	Adds program lines from 'file$' to the program in memory. Unlike LOAD, it does not clear the program currently in memory.
MKDIR dir$	CMM MM DOS	Make, or create, the directory ‘dir$’ on the SD card.
MODE mode [, palette]	CMM	Sets the number of colours that can be displayed on the screen. 'mode' can be: 1 Monochrome mode. 'palette' will select the colour to use and can be 0 to 7 representing the colours black to white. This mode provides complete compatibility with programs written for the monochrome Maximite 2 Four colour mode. 'palette' can be a number from 1 to 6 and will select the range of colours available (see table below). 3 Eight colour mode. In this mode all eight colours (including black and white) can be used. 'palette' can be supplied but will be ignored. 4 240x216 pixel resolution with all eight colours (including black and white) available. 'palette' can be supplied but will be ignored. In mode 2 the colours available in each palette are: palette = 1 Black, Red, Green, Yellow palette = 2 Black, Red, Blue, Purple palette = 3 Black, Red, Cyan, White palette = 4 Black, Green, Blue, Cyan palette = 5 Black, Green, Purple, White palette = 6 Black, Blue, Yellow, White The MODE command allows the programmer to trade the number of colours used and resolution against the amount of memory required by the video driver. Modes 1 and 4 use the least amount of memory while mode 3 uses the most. Also see "Graphics and Working with Colour" at the start of this manual.
NAME old$ AS new$	CMM MM DOS	Rename a file or a directory from ‘old$’ to ‘new$’ Unlike the other commands that work with file names the NAME command cannot accept a full pathname (with directories).
NEW	CMM MM DOS	Deletes the program in memory and clears all variables.
NEXT [counter-variable] [, counter-variable], etc	CMM MM DOS	NEXT comes at the end of a FOR-NEXT loop; see FOR. The ‘counter-variable’ specifies exactly which loop is being operated on. If no ‘counter-variable’ is specified the NEXT will default to the innermost loop. It is also possible to specify multiple counter-variables as in: NEXT x, y, z

ON nbr GOTO | GOSUB target[,target, target,...]	CMM MM DOS	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.
ON KEY target	CMM MM	Setup an interrupt which will call 'target' line number, label or user defined subroutine whenever there is one or more characters waiting in the input buffer. The characters can come from the keyboard, USB or serial console. Return from an interrupt is via the IRETURN statement except where a user defined subroutine is used (in that case END SUB or EXIT SUB is used). Note that subroutine parameters cannot be used. Note that all characters waiting in the input buffer should be read in the interrupt routine otherwise another interrupt will be automatically generated as soon as the program returns from the interrupt. To disable this interrupt, use numeric zero for the target, ie: ON KEY 0
OPEN fname$ FOR mode AS [#]fnbr	CMM MM DOS	Opens a file for reading or writing. ‘fname’ is the filename (8 chars max) with an optional extension (3 chars max) separated by a dot (.). It can be prefixed with a directory path. For example: "B:\DIR1\DIR2\FILE.EXT". ‘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). Note: APPEND is not supported on the flash file system (drive A:). 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. See Appendix I for more details. ‘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$() functions all use ‘fnbr’ to identify the file being operated on. See also OPTION ERROR and MM.ERRNO for error handling.
OPEN comspec$ AS [#]fnbr or OPEN comspec$ AS console	CMM MM	Will open a serial port for reading and writing. Two ports are available (COM1: and COM2:) and both can be open simultaneously. For a full description with examples see Appendix A. ‘comspec$’ is the serial port specification and has the form: “COMn: baud, buf, int, intlevel” with an optional ",FC" or “,OC” or ",DE" or ",S2" appended. COM1: uses pin 15 for receive data and pin 16 for transmit data and if flow control is specified pin 17 for RTS and pin 18 for CTS. COM2: uses pin 19 for receive data and pin 20 for transmit data on the monochrome Maximite and D0 (receive) and D1 (transmit) on the Colour Maximite. For the DuinoMite see the "DuinoMite MMBasic ReadMe.pdf" document for details. If the port is opened using ‘fnbr’ the port can be written to and read from using any commands or functions that use a file number. A serial port can be opened with “AS CONSOLE”. In this case any data received will be treated the same as keystrokes received from the keyboard and any characters sent to the video output will also be

		transmitted via the serial port. This enables remote control of MMBasic via a serial interface.
OPTION BASE 0 or OPTION BASE 1	CMM MM DOS	Set the lowest value for array subscripts to either 0 or 1. The default is 0. This must be used before any arrays are declared.
OPTION BREAK nn	CMM MM	Set the value of the break key to 'nn'. This key is used to interrupt a running program. The value of the break key is set to CTRL-C key at startup but it can be changed to any keyboard key using this command (for example, OPTION BREAK 156 will set the break key to the F12 key). Setting this option to an invalid key value (for example, 255) will disable the break function entirely.
OPTION ERROR CONTINUE or OPTION ERROR ABORT	CMM MM DOS	Set the treatment for errors in file input/output. The option CONTINUE will cause MMBasic to ignore file related errors. The program must check the variable MM.ERRNO to determine if and what error has occurred. The option ABORT sets the normal behaviour (ie, stop the program and print an error message). The default is ABORT. Note that this option only relates to errors reading from or writing to drives A: and B:. It does not affect the handling of syntax and other program errors.
OPTION PROMPT string$	CMM MM DOS	Sets the command prompt to the contents of ‘string$’ (which can also be an expression which will be evaluated when the prompt is printed). For example: OPTION PROMPT “Ok “ or OPTION PROMPT TIME$ + “: “ or OPTION PROMPT CWD$ + “: “ Maximum length of the prompt string is 48 characters. The prompt is reset to the default (“> “) on power up but you can automatically set it by saving the following example program as “AUTORUN.BAS” on the internal flash drive A:. 10 OPTION PROMPT “My prompt: “ 20 NEW
OPTION Fnn string$	CMM MM	Sets the programmable function key 'Fnn' to the contents of ‘string$’. 'Fnn' is the function key F1 to F12. Maximum string length is 12 characters. ‘string$’ can also be an expression which will be evaluated at the time of running the OPTION command. This is most often used to append the ENTER key (chr$(13)), or double quotes (chr$(34)). For example: OPTION F1 "RUN" + CHR$(13) OPTION F6 "SAVE " + CHR$(34) OPTION F10 "ENDIF" Normally these commands are included in an AUTORUN.BAS file (see OPTION PROMPT for an example).
OPTION USB OFF or OPTION USB ON	CMM MM	Turn the USB output off and on. This disables/enables the output from the PRINT command from being sent out on the USB interface. It does not affect the reception of characters from the USB interface. Normally this is used when a program wants to separately display data on the USB and video interfaces. This option is always reset to ON at the command prompt.

OPTION VIDEO OFF or OPTION VIDEO ON	CMM MM	VIDEO OFF prevents the output from the PRINT command from being displayed on the video output (VGA or composite). The VIDEO ON option will revert to the normal action. Normally this is used when a program wants to separately display data on the USB and video outputs. This option is always reset to ON at the command prompt.
PAUSE delay	CMM MM DOS	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).
PIN(pin) = value	CMM MM	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. ‘pin’ zero is a special case and will always control the LED on the front panel of the Maximite, the yellow LED on the UBW32 or the green LED on the DuinoMite. A ‘value’ of non-zero will turn the LED on, or zero for off. See the function PIN() for reading from a pin and the command SETPIN for configuring it.
PIXEL(x,y) = colour	CMM MM	Set a pixel on the VGA or composite screen to a colour or inverted (if the value is -1). See the section "Graphics and Working with Colour" for a definition of the colours and graphics coordinates. See the function PIXEL(x,y) for obtaining the value of a pixel.
PLAYMOD file [, dur] or PLAYMOD STOP	CMM MM	Play synthesised music or sound effects. 'file' specifies a file which must be located on the internal drive A: and must be in the .MOD format. 'dur' specifies the duration in milliseconds that the audio will play for - if not specified it will play until explicitly stopped or the program terminates. The audio is synthesised in the background and is not disturbed by the running program. The command PLAYMOD STOP will immediately halt any music or sound effect that is currently playing. NOTE: The file needs to be located on the internal drive A: for performance reasons, it will not play from the SD card.

POKE hiword, loword, val
or
POKE keyword, offset, val
or
POKE VAR var, ±offset, val
	CMM
	MM
	DOS
		Will set a byte within the PIC32 virtual memory space.
		The address is specified by ‘hiword’ which is the top 16 bits of the address while ‘loword’ is the bottom 16 bits.
		Alternatively 'keyword' can be used and 'offset' is the ±offset from the address of the keyword. The keyword can be VIDEO (monochrome Maximite video buffer) or RVIDEO, GVIDEO, BVIDEO (red, blue and green video buffers on the Colour Maximite), PROGMEM (program memory) or VARTBL (the variable table). The input keystroke buffer is KBUF, the position of the head of the keystroke queue is KHEAD and the tail is KTAIL (the buffer is 256 bytes long).
		You can also access the memory allocated to a variable by using the variable's name ('var') preceded by the keyword VAR. This can be used to access the individual bytes of a numeric variable or a large segment of RAM allocated to an array. The first element of an array (eg, nbr(0)) is the start of RAM allocated to the whole array. For example:
		`DIM nbr(1024)  'allocate 4KB`
		`POKE VAR nbr(0),100,1 'set the 100th byte to 1`
		This command is for expert users only. The PIC32 maps all control registers, flash (program) memory and volatile (RAM) memory into a single address space so there is no need for INP or OUT commands. The PIC32MX5XX/6XX/7XX Family Data Sheet lists the details of this

		address space. Note that MMBasic stores most data (including video) as 32 bit integers and the PIC32 uses little endian format. WARNING: No validation of the parameters is made and if you use this facility to access an invalid memory address you will get an "internal error" which causes the processor to reset and clear all memory.
PORT(start, nbr [,start, nbr]…) = value	CMM MM	Set a number of I/O consecutive 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(10, 4, 16, 4) = &B10000011 Will set eight I/O pins. Pins 10 and 11 will be set high while 12, 13, 16, 17 and 18 will be set to a low and finally 19 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 pin up to 23 pins. See the PORT function to simultaneously read from a number of pins.
PRINT expression [[,;]expression] … etc	CMM MM DOS	Outputs text to the screen. 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 numbers (greater than six digits) are printed in scientific format. The function FORMAT$() can be used to format numbers. 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 @(x, y) expression Or PRINT @(x, y, m) expression	CMM MM	Same as the PRINT command except that the cursor is positioned at the coordinates x, y. 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)

PRINT #nbr, expression [[,;]expression] … etc	CMM MM DOS	Same as above except that the output is directed to a file previously opened for OUTPUT or APPEND as ‘nbr’. See the OPEN command.
PULSE pin, width	CMM MM	Will generate a pulse on 'pin' with duration of 'width' mS. 'width' can be a fraction. For example, 0.01 is equal to 10 µS 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. A background pulse can have its time changed while it is running by issuing a new PULSE command or it can be terminated by issuing a PULSE command with zero for 'width'.
PWM freq, ch2, ch1 or PWM STOP	CMM MM	Generate a pulse width modulated (PWM) output for driving analogue circuits. ‘freq’ is the output frequency (between 20 Hz and 1 MHz) . The frequency can be changed at any time by issuing a new PWM command. The output will run continuously in the background while the program is running and can be stopped using the PWM STOP command. 'ch2' and 'ch1' are the output duty cycles for channel 2 and 1 as a percentage. If the percentage 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. Both are optional and if not specified will default to the previously used duty cycle for that channel. The PWM output is generated on the PWM/sound connector and that assumes that the connector has been wired for PWM output. The frequency of the output is locked to the PIC32 crystal and is very accurate and for frequencies below 50 KHz the duty cycle will be accurate to 0.1%. The original monochrome Maximite has only one PWM/sound output so only 'ch2' can be set on that model.
QUIT	DOS	Will exit MMBasic and return control to the operating system.
RANDOMIZE nbr	CMM MM DOS	Seed the random number generator with ‘nbr’. On power up the random number generator is seeded with zero and will generate the same sequence of random numbers each time. To generate a different random sequence each time you must use a different value for ‘nbr’. One good way to do this is use the TIMER function. For example 100 RANDOMIZE TIMER
READ variable[, variable]...	CMM MM DOS	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. See also DATA and RESTORE.
REM string	CMM MM DOS	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.
RESTORE	CMM MM DOS	Resets the line and position counters for DATA and READ statements to the top of the program file.

RETURN	CMM MM DOS	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB.
RMDIR dir$	CMM MM DOS	Remove, or delete, the directory ‘dir$’ on the SD card.
RTC GETTIME or RTC SETTIME year, month, day, hour, minute, second	CMM MM	RTC GETTIME will get the current date/time from a PCF8563 real time clock and set the internal MMBasic clock accordingly. The date/time can then be retrieved with the DATE$ and TIME$ functions. RTC SETTIME will set the time in the PCF8563. The year must be the last two digits of the year (ie, 14 for 2014) and hour is 0 to 23 hours (ie, 24 hour notation). The PCF8563 is an I^2C device and it must be connected to the two I^2C pins with appropriate pullup resistors. If the I^2C bus is already open the RTC command will use the current settings, otherwise it will temporarily open the connection with a speed of 100KHz. See the section "Special Hardware Devices" for more details.
RUN [line] [file$]	CMM MM DOS	Executes the program in memory. If a line number is supplied then execution will begin at that line, otherwise it will start at the beginning of the program. Or, if a file name (file$) is supplied, the current program will be erased and that program will be loaded from the current drive and executed. This enables one program to load and run another. Example: RUN “TEST.BAS” If an extension is not specified “.BAS” will be added to the file name.
SAVE [file$]	CMM MM DOS	Saves the program in the current working directory as ‘file$’. The file name is optional and if omitted the last filename used in SAVE, LOAD or RUN will be automatically used. Example: SAVE “TEST.BAS” If an extension is not specified “.BAS” will be added to the file name.
SAVEBMP file$	CMM MM	Saves the current VGA or composite screen as a BMP file in the current working directory on the current drive. The Colour Maximite will save the file as a 16 colour (4 bit) file. Example: SAVEBMP “IMAGE.BMP” If an extension is not specified “.BMP” will be added to the file name. Note that Windows 7 Paint has trouble displaying monochrome images. This appears to be a bug in Paint as all other software tested (including Windows XP Paint) can display the image without fault. See also LOADBMP.
SCANLINE colour, start [, end]	CMM	This command can be used to set the colour of individual horizontal scan lines on the VGA monitor when in MODE 1,7. 'colour' can be any colour specified by name or number from 0 to 7. 'start' and 'end' specify the range of scan lines to set. If 'end' is not specified only one line will be set. Multiple calls to SCANLINE can be used to set the colour of other scan lines or to change the colour of lines already set (ie, the settings accumulate). This command is valid only when the colour mode is set to MODE 1,7 (monochrome with the colour set to white). All settings made by SCANLINE are automatically cancelled whenever the MODE command is used or when MMBasic returns to the command prompt. See the section "Graphics and Working with Colour" for more details.
SEEK [#]fnbr, pos	CMM MM DOS	Will position the read/write pointer in a file that has been opened 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. See Appendix I for more details on random file access.
SETPIN pin, cfg	CMM MM	Will configure the external I/O ‘pin’ according to ‘cfg’. The original Maximite has 20 I/O pins numbered 1 to 20, the Colour Maximite has another 20 I/O pins on the Arduino connector. These are labelled D0 to D13 and A0 to A5. ‘cfg’ is a keyword and can be any one of the following: OFF Not configured or inactive AIN Analog input (pins 1 to 10, A0 to A5) DIN Digital input (all pins) FIN Frequency input (pins 11 to 14) PIN Period input (pins 11 to 14) CIN Counting input (pins 11 to 14) DOUT Digital output (all pins) OOUT Open collector digital output (pins 11 to 20, D0 to D1 In this mode the function PIN() will also return the value on the output pin . This enables a program to check if an external device is pulling the pin low. Previous versions of MMBasic used numbers for 'cfg' and for backwards compatibility they will still be recognised. Pins 11 to 20 and D0 to D13 can accept 5V inputs and 5V pull-ups in open collector mode. The remainder have a maximum input voltage of 3.3V. For the DuinoMite see "DuinoMite MMBasic ReadMe.pdf" See the function PIN() for reading inputs and the statement PIN()= for outputs. See the command below if an interrupt is configured.
SETPIN pin, cfg , target	CMM MM	Will configure ‘pin’ to generate an interrupt according to ‘cfg’: '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) Previous versions of MMBasic used numbers for 'cfg' and for backwards compatibility they will still be recognised. ‘target' is the interrupt routine which can be a line number, label or user defined subroutine. This mode also configures the pin as a digital input so the value of the pin can always be retrieved using the function PIN(). Return from an interrupt is via the IRETURN statement except where a user defined subroutine is used (in that case END SUB or EXIT SUB is used). Note that subroutine parameters cannot be used.
SETTICK period, target [, nbr]	CMM MM	This will setup a periodic interrupt (or “tick”). Four tick timers are available ('nbr' = 1, 2, 3 or 4). 'nbr' is optional and if not specified timer number 1 will be used. The time between interrupts is ‘period’ milliseconds and ‘target' is the line number or label of the interrupt routine. See also IRETURN to return from the interrupt. A subroutine can also be specified for the interrupt target and in that case return is via EXIT SUB or END SUB. The period can range from 1 to 2147483647 mSec (about 24 days). These interrupts can be disabled by setting ‘period’ to zero (ie, SETTICK 0, 0, 3 will disable tick timer number 3).

SPRITE LOAD file or SPRITE ON n, x, y [, colour] or SPRITE MOVE n, x, y [, colour] or SPRITE COPY n1 TO n2 or SPRITE OFF n [, n [, …]] or SPRITE OFF ALL or SPRITE PASTE n, x, y or SPRITE UNLOAD	CMM MM	Load and manipulate sprites on the screen. Sprites are 16x16 pixel objects that can be moved about on the screen without erasing or disturbing text or other underlying graphics. See Appendix H "Sprites" for more details. This command is mostly used in animated games. 'n' is the sprite number, 'x' and 'y' are the sprite's coordinates on the screen. SPRITE LOAD will load a sprite file into memory. This file defines the graphic images of one or more sprites. SPRITE ON will display an individual sprite contained in the sprite file. SPRITE MOVE will move the sprite to a new location and restore the background at the old location. For both ON and MOVE 'colour' can be optionally specified and this colour will be used for the background. When using a solid background colour this is faster and does not require special handling for overlapping sprites (see Appendix H). SPRITE COPY will copy the bitmap for sprite number n1 into the bitmap for sprite n2. If the destination sprite ('n2') is currently on (ie, displayed) then this command will perform the equivalent of SPRITE OFF, COPY then ON again. This command can be used for animation of multiple sprites that use the same bitmaps. See the MMBasic Library (http://geoffg.net/maximite.html#Downloads) for an example. SPRITE OFF will remove the sprite from the screen and restore the background graphics that was obscured when the sprite was turned on. A number of sprites may be quickly removed in sequence by specifying a comma separated list. ie, SPRITE OFF 4, 8, 2, 3. As a shortcut SPRITE OFF ALL will remove all active sprites. SPRITE PASTE will simply copy the sprite bitmap to the video screen. There is no need to turn the sprite on or off and the background is not saved therefore the sprite bitmap will become part of the background. SPRITE UNLOAD will disable the sprites, unload the file and reclaim the memory used. The sprite file can contain many individual sprites which can be simultaneously displayed and independently manipulated at the same time. The number of sprites is limited only by the available memory. Also see the function COLLISION() for collision detection.
SUB xxx (arg1 [,arg2, …]) <statements> <statements> END SUB	CMM MM DOS	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. Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: `MySub a1, a2` When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended.

		Brackets around the argument list in both the caller and the definition are optional.
SYSTEM command$	DOS	Submit 'command$' to the operating system. It can be any command recognised by the command window in Windows XP/Vista/7. The available commands are listed here: http://ss64.com/nt For example, this will set the window to blue lettering on a yellow background: SYSTEM "COLOR 1E" Note that the command is executed in a different instance of the command processor to MMBasic so some commands (like "CD ..") will have no effect.
TIME$ = “HH:MM:SS” or TIME$ = “HH:MM” or TIME$ = “HH”	CMM MM	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. Normally the time is set to “00:00:00” on power up. If the battery backed clock option is fitted to the Colour Maximite the current time will be automatically set on power up.
TIMER = msec	CMM MM DOS	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.
TONE left [, right [, dur]] or TONE STOP	CMM MM	Generates a continuous sine wave on the sound output. 'left' and 'right' are the frequencies 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. If the duration is not specified the tone will continue until explicitly stopped or the program terminates. The command TONE STOP will immediately halt the tone output. The frequency can be from 1Hz to 20KHz and is very accurate (it is based on the PIC32 crystal oscillator). The frequency can be changed at any time by issuing a new TONE command. In the monochrome Maximite and compatibles only the left frequency will play but a dummy or empty value is still required for the right channel.
TROFF	CMM MM DOS	Turns the trace facility off; see TRON.
TRON	CMM MM DOS	Turns on the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs.
WATCHDOG timeout or WATCHDOG OFF	CMM MM	Starts the watchdog timer which will automatically restart the Maximite when it has timed out. This can be used to recover from some event that disables 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 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 PIC32 processor will be automatically rebooted. When MMBasic is restarted it will then attempt to run the program RESTART.BAS which can be used to recover the situation before using RUN or CHAIN to run the main program. At any time WATCHDOG OFF can be used to 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 to interrupt a running program.
XMODEM SEND file$ or XMODEM RECEIVE file$	CMM MM	Transfers a file to or from a remote computer using the XModem protocol. The transfer is done over the USB connection or, if a serial port is opened as console, over that serial port. 'file$' is the file (on the SD card or internal flash) to be sent or received. 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. Download Tera Term from http://ttssh2.sourceforge.jp/

Functions

The centre column specifies the platform (CMM is the Colour Maximite, MM is the monochrome Maximite and DuinoMite, DOS is the Windows version). Square brackets indicate that the parameter or characters are optional.

ABS(number)	CMM MM DOS	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ASC(string$)	CMM MM DOS	Returns the ASCII code for the first letter in the argument ‘string$’.
ATN(number)	CMM MM DOS	Returns the arctangent value of the argument 'number' in radians.
BIN$(number)	CMM MM DOS	Returns a string giving the binary (base 2) value for the 'number'.
CHR$(number)	CMM MM DOS	Returns a one-character string consisting of the character corresponding to the ASCII code indicated by argument 'number'.
CINT(number)	CMM MM DOS	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().
CLR$( ) or CLR$(fg) or CLR$(fg, bg)	CMM	Returns a string containing embedded codes to select colours in a string. 'fg' is the foreground colour and 'bg' is the background colour. If no parameters are specified both the foreground and background colours will be reset to the defaults set by the last COLOUR command. Example, this will display yellow letters on a red background: `PRINT CLR$(YELLOW,RED) " ALARM "` This function simply generates a two character string where the first character is the number 128 plus the foreground colour number and the second character is the number 192 plus the background colour number.
COLLISION(n, EDGE) or COLLISION(n, SPRITE)	CMM MM	Tests if a collision has occured between sprite 'n' and the edge of the screen or another sprite depending on the form used. See Appendix H for more details. Returns: &B0001 Indicating a collision to the left of the sprite &B0010 Collision on the right &B0100 Collision on the top &B1000 Collision on the bottom Note that it is possible for these results to be combined. For example; a result of &B0101 indicates that the sprite has collided with something both at the top and left of the sprite (for example the top left corner of the screen).

		If the sprite is overlapping another (ie, one or more non transparent pixels are on top of another sprite's non transparent pixels) bit &B10000 will be set in the value returned by COLLISION () in addition to the bits for left, right, etc as described above.
COS(number)	CMM MM DOS	Returns the cosine of the argument 'number' in radians.
CWD$	CMM MM DOS	Returns the current working directory as a string.
DATE$	CMM MM DOS	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$ =.
DEG(radians)	CMM MM DOS	Converts 'radians' to degrees.

DIR$(fspec, type)
or
DIR$(fspec)
or
DIR$( )
	CMM
	MM
		Will search an SD card for files and return the names of entries found.
		'fspec' is a file specification using wildcards the same as used by the FILES command. Eg, "*.*" will return all entries, "*.TXT" will return text files.
		'type' is the type of entry to return and can be one of:
		VOL Search for the volume label only
		DIR Search for directories only
		FILE Search for files only (the default if 'type' is not specified)
		The function will return the first entry found. To retrieve subsequent entries use the function with no arguments. ie, DIR$( ). 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`
		This function only operates on the SD card and you must change to the required directory before invoking it.

DISTANCE(trigger, echo) or DISTANCE(trig-echo)	CMM MM	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).

		The measurement can take up to 32mS to complete and interrupts will be disabled in this period.
DS18B20(pin)	CMM MM	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. The time required for the overall measurement is 200mS and interrupts will be ignored during this period. Alternatively the DS18B20 START command can be used to start the measurement and your program can do other things while the conversion is progressing. When this function is called the value will then be returned instantly assuming the conversion period has expired. If it has not, this function will wait out the remainder of the conversion time before returning the value. The DS18B20 can be powered separately by a 3V to 5V supply or it can operate on parasitic power from the Maximite. See the section "Special Hardware Devices" for more details.
EOF([#]nbr)	CMM MM DOS	Will return true if the file previously opened for INPUT with the file number ‘nbr’ is positioned at the end of the file. If used on a file number opened as a serial port this function will return true if there are no characters waiting in the receive buffer. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT$ function.
EXP(number)	CMM MM DOS	Returns the exponential value of 'number'.
FIX(number)	CMM MM DOS	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$(nbr [, fmt$])	CMM MM DOS	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 **
HEX$(number)	CMM MM DOS	Returns a string giving the hexadecimal (base 16) value for the 'number'.
INKEY$	CMM MM DOS	Checks the keyboard and USB input buffers and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string. If the input buffer is empty this function will immediately return with an empty string (ie, "").
INPUT$(nbr, [#]fnbr)	CMM MM DOS	Will return a string composed of ‘nbr’ characters read from a file previously opened for INPUT with the file number ‘fnbr’. This function will read all characters including carriage return and new line without translation. When reading from a serial communications port this will return as many characters as are waiting in the receive buffer up to ‘nbr’. If there are no characters waiting it will immediately return with an empty string. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, string-pattern$)	CMM MM DOS	Returns the position at which 'string-pattern$' occurs in 'string-searched$', beginning at 'start-position'.
INT(number)	CMM MM DOS	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() .
KEYDOWN	CMM MM	Return the decimal ASCII value of the PS2 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. Note that this function will only work with the attached PS2 keyboard and that using this function will also clear any characters stored in the keyboard input buffer.

LEFT$(string$, number-of-chars)	CMM MM DOS	Returns a substring of ‘string$’ with ‘number-of-chars’ from the left (beginning) of the string.
LEN(string$)	CMM MM DOS	Returns the number of characters in 'string$'.
LOC([#]fnbr)	CMM MM DOS	For a file opened as RANDOM this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. See Appendix I for more details of random file access. For a serial port this will return the number of bytes received and waiting in the receive buffer to be read. The # is optional.
LOF([#]fnbr)	CMM MM DOS	For a file this will return the current length of the file in bytes. For a serial port this will return the space (in characters) remaining in the transmit buffer. Note that when the buffer is full MMBasic will pause when adding a new character and wait for some space to become available. The # is optional.
LOG(number)	CMM MM DOS	Returns the natural logarithm of the argument 'number'.
LCASE$(string$)	CMM MM DOS	Returns ‘string$’ converted to lowercase characters.
MID$(string$, start-position-in-string[, number-of-chars])	CMM MM DOS	Returns a substring of ‘string$’ beginning at ‘start-position-in-string’ and continuing for ‘number-of-chars’ bytes. If ‘number-of-chars’ is omitted the returned string will extend to the end of ‘string$’
OCT$(number)	CMM MM DOS	Returns a string giving the octal (base 8) representation of 'number'.
PEEK(hiword, loword) or PEEK(keyword, ±offset) or PEEK(VAR var, ±offset)	CMM MM DOS	Will return a byte within the PIC32 virtual memory space. The address is specifies by ‘hiword’ which is the top 16 bits of the address while ‘loword’ is the bottom 16 bits. Alternatively 'keyword' can be used and 'offset' is the ±offset from the address of the keyword. The keyword can be VIDEO (monochrome Maximite video buffer) or RVIDEO, GVIDEO, BVIDEO (red, blue and green video buffers on the Colour Maximite), PROGMEM (program memory) or VARTBL (the variable table). The input keystroke buffer is KBUF, the position of the head of the keystroke queue is KHEAD and the tail is KTAIL (the buffer is 256 bytes long). You can also access the memory allocated to a variable by using the variable's name ('var') preceded by the keyword VAR. This can be used to access the individual bytes of a numeric variable or a large segment of RAM allocated to an array (the first element of an array (eg, nbr(0)) is the start of RAM allocated to the whole array). See the POKE command for notes and warnings related to memory access.

PI	CMM MM DOS	Returns the value of pi.
PIN(pin)	CMM MM	Returns the value on the external I/O ‘pin’. Zero means digital low, 1 means digital high and for analogue inputs it will return the measured voltage as a floating point number. Frequency inputs will return the frequency in Hz (maximum 200 kHz). 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). ‘pin’ zero (ie, = PIN(0)) is a special case which will always return the state of the bootload or program push button on the PC board (non zero means that the button is down). Also see the SETPIN and PIN() = commands.
PORT(start, nbr [,start, nbr]…)	CMM MM	Returns the value of a number of consecutive 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 if an additional group of input pins need to be added. This command 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 pin up to 23 pins. See the PORT command to simultaneously output to a number of pins.
POS	CMM MM DOS	Returns the current cursor position in the line in characters.
PIXEL(x, y)	CMM MM	Returns the colour of a pixel on the VGA or composite screen. See the section "Graphics and Working with Colour" for a definition of the colours and graphics coordinates.. See the statement PIXEL(x,y) = for setting the value of a pixel.
RAD(degrees)	CMM MM DOS	Converts 'degrees' to radians.
RIGHT$(string$, number-of-chars)	CMM MM DOS	Returns a substring of ‘string$’ with ‘number-of-chars’ from the right (end) of the string.
RND(number)	CMM MM DOS	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The RANDOMIZE command reseeds the random number generator.
SGN(number)	CMM MM DOS	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.

SIN(number)	CMM MM DOS	Returns the sine of the argument 'number' in radians.
SPACE$(number)	CMM MM DOS	Returns a string of blank spaces 'number' bytes long.
SPI(rx, tx, clk[, dat[, speed]])	CMM MM	Sends and receives a byte using the SPI protocol with MMBasic as the master (ie, it generates the clock). ‘rx’ is the pin number for the data input (MISO) ‘tx’ is the pin number for the data output (MOSI) ‘clk’ is the pin number for the clock generated by MMBasic (CLK) ‘dat’ is optional and is an integer representing the data byte to send over the data output pin. If it is not specified the ‘tx’ pin will be held low. ‘speed’ is optional and is the speed of the clock. It is a single letter either H, M or L where H is 3MHz, M is 500 KHz and L is 50 KHz. Default is H. See Appendix D for a full description.
SQR(number)	CMM MM DOS	Returns the square root of the argument 'number'.
STR$(number)	CMM MM DOS	Returns a string in the decimal (base 10) representation of 'number'.
STRING$(number, ascii-value\|string$)	CMM MM DOS	Returns a string 'number' bytes long consisting of either the first character of string$ or the character representing the ASCII value ascii-value.
TAB(number)	CMM MM DOS	Outputs spaces until the column indicated by 'number' has been reached.
TAN(number)	CMM MM DOS	Returns the tangent of the argument 'number' in radians.
TIME$	CMM MM DOS	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”. To set the current time use the command TIME$ = .
TIMER	CMM MM DOS	Returns the elapsed time in milliseconds (eg, 1/1000 of a second) since reset. If not specifically reset this count will wrap around to zero after 49 days. The timer is reset to zero on power up and you can also reset by using TIMER as a command.
UCASE$(string$)	CMM MM DOS	Returns ‘string$’ converted to uppercase characters.

VAL(string$)	CMM MM DOS	Returns the numerical value of the ‘string$’. If 'string$' is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.

Obsolete Commands and Functions

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

The centre column specifies the platform (CMM is the Colour Maximite, MM is the monochrome Maximite and DuinoMite, DOS is the Windows version). Square brackets indicate that the parameter or characters are optional.

IF condition THEN linenbr	CMM MM DOS	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
LOCATE x, y	CMM MM	Positions the cursor to a location in pixels and the next PRINT command will place its output at this location. This construct is included for Microsoft compatibility. New programs should use the PRINT @(x,y) construct (see the PRINT command).
PRESET (x, y) PSET (x, y)	CMM MM	Turn off (PRESET) or on (PSET) a pixel on the video screen. These statements are included for Microsoft compatibility. New programs should use the PIXEL(x,y) = statement.
SOUND freq or SOUND freq, dur	CMM MM	Generate a single tone of ‘freq’ (between 20 Hz and 1 MHz) for ‘dur’ milliseconds. The sound is played in the background and does not stop program execution. If 'dur' is not specified the sound will play forever until turned off. If ‘dur’ is zero, any active SOUND statement is turned off. The command has been replaced with the TONE command that generates a pure sine wave (not a square wave as SOUND does).
SPC(number)	CMM MM DOS	This function returns a string of blank spaces 'number' bytes long. It is similar to the SPACE$() function and is only included for Microsoft compatibility.
WHILE expression WEND	CMM MM DOS	WHILE initiates a WHILE-WEND loop. The loop ends with WEND, and execution reiterates through the loop as long as the 'expression' is true. This construct is included for Microsoft compatibility. New programs should use the DO WHILE … LOOP construct.
WRITE [#nbr,] expression [,expression] …	CMM MM DOS	Outputs the value of each ‘expression’ separated by commas (,). If ‘expression’ is a number it is outputted without preceding or trailing spaces. If it is a string it is surrounded by double quotes (“). The list is terminated with a new line. If ‘#nbr’ is specified the output will be directed to a file previously opened for OUTPUT or APPEND as ‘#nbr’. See the OPEN command. WRITE can be replaced by the PRINT command.

Appendix A
Serial Communications

Two serial ports are available for asynchronous serial communications (four on the DuinoMite). They are labelled COM1:, COM2:, etc and are opened in a manner similar to opening a file. After being opened they will have an associated file number (like an opened disk file) and you can use any commands that operate with a file number to read and write to/from the serial port. A serial port is also closed using the CLOSE command.

The following is an example:

```
OPEN "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`

The transmission format is fixed at 8 data bits, no parity and one stop bit (two stop bits can also be specified).

'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.

It has the form `"COMn: baud, buf, int, intlevel, FC, DE, OC, S2"` where:

- 'n' is the serial port number for either COM1: or COM2: (plus COM3: and COM4: on the DuinoMite).
- 'baud' is the baud rate, either 19200, 9600, 4800, 2400, 1200, 600 or 300 bits per second. For COM3 and COM4 (DuinoMite only) it can be any number from 300 to 460800. Default is 9600.
- 'buf' is the buffer sizes in bytes. Two of these buffers will be allocated from memory, one for transmit and one for receive. The default size is 256 bytes.
- 'int' is the line number, label or a user defined subroutine of the interrupt routine to be invoked when the serial port has received some data. The default is no interrupt.
- 'intlevel' is the number of characters that must be waiting in the receive queue before the receive interrupt routine is invoked. The default is 1 character.

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.

Four options can be added to the end of 'comspec$' These are FC, DE, OC and S2.:

- 'FC' will enable hardware RS232 style flow control. This can only be specified on COM1: and it will enable two extra signals, Request To Send (receive flow control) and Clear To Send (transmit flow control) which can be used with a cooperating device to prevent data overflow.
- 'DE' will enable the Data output Enable (DE) signal for RS485. This can only be specified on COM1: and it will enable the DE signal which will go high just before a byte is transmitted and will go low when the last byte in the transmit buffer has been sent. DE and FC are mutually exclusive.
- 'OC' will force the output pins (Tx and optionally RTS or DE) to be open collector. This option can be used on both COM1: and COM2:. The default is normal (0 to 3.3V) output.
- 'S2' specifies that two stop bits will be sent following each character transmitted.

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 buffer size (1KB) but no flow control:
`OPEN "COM1:9600, 1024" AS #8`

The same as above but with receive flow control (RTS) and transmit flow control (CTS) enabled:
`OPEN "COM1:9600, 1024, FC" AS #8`

An example specifying everything including an interrupt, an interrupt level, flow control and open collector:
`OPEN "COM1:19200, 1024, ComIntLabel, 256, FC, OC" AS #5`

Input/Output Pin Allocation

COM1: uses pin 15 for receive data (data in) and pin 16 for transmit data (data out). If flow control is specified pin 17 will be used for RTS (receive flow control – it is an output) and pin 18 will be CTS (transmit flow control – it is an input). If the Data output Enable (DE) signal for RS485 is enabled it will be on pin 17.

COM2: uses pin 19 for receive data (data in) and pin 20 for transmit data (data out) in the monochrome Maximite and D0 for receive data and pin D1 for transmit data on the Colour Maximite.

For the DuinoMite see the "DuinoMite MMBasic ReadMe.pdf" document for details.

When a serial port is opened the pins used by the port are automatically set to input or output as required and the SETPIN and PIN commands are 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 signal polarity is standard for devices running at TTL voltages (not RS232). 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. The flow control pins (RTS and CTS) use a low voltage to signal stop sending data and high as OK to send. These signal levels allow you to directly connect to devices like GPS modules (which generally use TTL voltage levels).

Reading and Writing

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

The LOC() function is also handy; it will return the number of characters waiting in the receive buffer (ie, the number characters that can be retrieved by the INPUT$() function). The EOF() function will return true if there are no characters waiting. The LOF() function will return the space (in characters) remaining in the transmit buffer.

When outputting to a serial port (ie, using PRINT #n, dat) the command will pause if the output buffer is full and wait until there is sufficient space to place the new data in the buffer before returning. If the receive buffer overflows with incoming data the serial port will automatically discard the oldest data to make room for the new data.

Serial ports can be closed with the CLOSE command. This will discard any characters waiting in the buffers, free 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 routine (if specified) will operate the same as a general interrupt on an external I/O pin (see page 7 for a description). Return from the interrupt is via the IRETURN statement except where a user defined subroutine is used (in that case END SUB or EXIT SUB is used). Note that subroutine parameters cannot be used.

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. So, for example, if you have specified the interrupt level as 200 characters and a buffer of 256 characters then quite easily the buffer will have overflowed by the time the interrupt routine can read the data. In this case the buffer should be increased to 512 characters or more.

Opening a Serial Port as the Console

A serial port can be opened as the console for MMBasic. The command is:

```
OPEN comspec AS CONSOLE
```

In this case any characters received from the serial port will be treated the same as keystrokes received from the keyboard and any characters sent to the video output will also be transmitted via the serial port. This enables a user with a terminal at the end of the serial link to exercise remote control of MMBasic. For example, via a modem.

Note that only one serial port can be opened “AS CONSOLE” at a time and it will remain open until explicitly closed using the CLOSE CONSOLE command. It will not be closed by commands such as NEW and RUN.

Appendix B
I^2C Communications

The Inter Integrated Circuit (I^2C) bus was developed by Philips (now NXP) for the transfer of data between integrated circuits. This implementation was written by Gerard Sexton and the standard definition of I^2C is provided by this document: http://www.nxp.com/documents/user_manual/UM10204.pdf

There are four commands that can be used in I^2C master mode:

I2C OPEN speed, timeout	Enables the I^2C module in master mode. 'speed' is a value between 10 and 400 (for bus speeds 10 kHz to 400 kHz). '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^2C slave device. 'addr' is the slave I^2C address. 'option' is a number between 0 and 3 (normally this is set to 0) 1 = keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 2 = treat the address as a 10 bit address 3 = combine 1 and 2 (hold the bus and use 10 bit addresses). '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 in the command as individual bytes. Example: I2C WRITE &H6F, 1, 3, &H23, &H43, &H25 • The data can be in a one dimensional array. The subscript does not have to be zero and will be honoured; also bounds checking is performed. Example: I2C WRITE &H6F, 1, 3, ARRAY(0) • The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 1, 3, STRING$
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I^2C slave device. 'addr' is the slave I^2C address. 'option' is a number between 0 and 3 (normally this is set to 0) 1 = keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 2 = treat the address as a 10 bit address 3 = combine 1 and 2 (hold the bus and use 10 bit addresses). 'rcvlen' is the number of bytes to receive. 'rcvbuf' is the variable to receive the data - this can be a string variable (eg, t$), or the first element of a one dimensional array of numbers (eg, data(0)) or a normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE	Disables the I^2C module and returns the I/O pins to a "not configured" state. Then can then be configured using SETPIN. This command will also send a stop if the bus is still held.

And similarly there are four commands for the slave mode:

I2C SLAVE OPEN addr, mask, option, send_int, rcv_int	Enables the I²C module in slave mode. ‘addr’ is the slave I²C address. ‘mask’ is the address mask (normally 0, bits set as 1 will always match). This allows the slave to respond to multiple addresses. ‘option’ is a number between 0 and 3 (normally this is set to 0). 1 = allows MMBasic to respond to the general call address. When this occurs the value of MM.I2C will be set to 4. 2 = treat the address as a 10 bit address 3 = combine 1 and 2 (respond to the general call address and use 10 bit addresses). ‘send_int’ is the line number or label of a send interrupt routine to be invoked when the module has detected that the master is expecting data. ‘rcv_int is the line number or label of a receive interrupt routine to be invoked when the module has received data from the master.
I2C SLAVE WRITE sendlen, senddata [,sendata ....]	Send the data to the I²C master. This command should be used in the send interrupt (ie in the 'send_int_line' when the master has requested data). Alternatively a flag can be set in the send interrupt routine and the command invoked from the main program loop when the flag is set. ‘sendlen is the number of bytes to send. ‘senddata’ is the data to be sent. This can be specified in various ways, see the I2C WRITE commands for details.
I2C SLAVE READ rcvlen, rcvbuf, rcvd	Receive data from the I²C master device. This command should be used in the receive interrupt (ie in the 'rcv_int_line' when the master has sent some data). Alternatively a flag can be set in the receive interrupt routine and the command invoked from the main program loop when the flag is set. ‘rcvlen’ is the maximum number of bytes to receive. ‘rcvbuf’ is the variable to receive the data - this can be a string variable (eg, t$), or the first element of a one dimensional array of numbers (eg, data(0)) or a normal numeric variable (in this case rcvlen must be 1). ‘rcvd’ will contain the actual number of bytes received by the command.
I2C SLAVE CLOSE	Disables the slave I²C module and returns the external I/O pins 12 and 13 to a "not configured" state. Then can then be configured using SETPIN.

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

0 = The command completed without error.

1 = Received a NACK response

2 = Command timed out

For users of previous versions of MMBasic

This implementation of the I²C protocol is generally compatible with previous versions with the following differences:

- The commands have been renamed but have the same functionality. I2CEN is now I2C OPEN, I2CSEND is I2C WRITE, I2CRCV is I2C READ and I2CDIS is now I2C CLOSE. Similarly, I2CSEN is now I2C SLAVE WRITE, etc.
- Master interrupts are not supported.
- The NUM2BYTE command and BYTE2NUM () function are not implemented (use the PEEK function and POKE command instead).

7 and 8 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). A simple way of finding the address is to take the 8 bit write address and divide it by 2.

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.

10 Bit Addressing

10-bit addressing was designed to be compatible with 7-bit addresses, allowing developers to mix the two types of devices on a single bus. Devices that use 10-bit addresses will be clearly identified as such in their data sheets.

In 10-bit addressing the slave address is sent in two bytes with the first byte beginning with a special bit pattern to indicate that a 10 bit address is being used. This process is automatically managed by MMBasic when the 'option' argument is set for 10-bit addressing. 10-bit addresses can be in the range of 0 to 3FF (hex).

Master/Slave Modes

The master and slave modes can be enabled simultaneously; however, once a master command is in progress, the slave function will be "idle" until the master releases the bus. Similarly, if a slave command is in progress, the master commands will be unavailable until the slave transaction completes.

In master mode, the I^2C send and receive commands will not return until the command completes or a timeout occurs (if the timeout option has been specified).

The slave mode uses an MMBasic interrupt to signal a change in status and in this routine the program should write/read the data as specified by the I^2C master. This operates the same as a general interrupt on an external I/O pin. Return from the interrupt is via the IRETURN statement except where a user defined subroutine is used (in that case END SUB or EXIT SUB is used).

I/O Pins

On the Maximite pin 12 becomes the I^2C data line (SDA) and pin 13 the clock (SCL). For the DuinoMite see the "DuinoMite MMBasic ReadMe.pdf" document for details.

Both of these pins should have external pullup resistors installed (typical values are 10KΩ for 100KHz or 2KΩ for 400 kHz). When the I^2C 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.

When running the I^2C bus at above 150 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^2C 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

I^2C is ideally suited for communications between integrated circuits. As an example, there might be an occasion when a Maximite does not have enough serial ports, I/O pins, or whatever for a particular application. In that case a Micromite[‡] or Maximite could be used as a slave to provide the extra facilities.

This example converts a Micromite into a general purpose I/O expansion chip with 17 I/O pins that can be dynamically configured (by the master) as analog inputs or digital input/outputs. The routines on the master are

[‡] For details of the Micromite go to http://geoffg.net/micromite.html

simple to use (SSETPIN to configure the slave I/O and SPIN() to control it) and the program running on the master need not know that the physical I/O pins reside on another chip. All communications are done via I^2C.

The following illustration shows the connections required:

Program Running On the Slave:

The slave must first set up its I^2C interface to respond to requests from the master. With that done it can then drop into an infinite loop while the job of responding to the master is handled by the I^2C interrupts.

In the program below the slave will listen on I^2C address 26 (hex) for a three byte command from the master. The format of this message is:

- Byte 1 is the command type. It can have one of three values; 1 means configure the pin, 2 means set the output of the pin and 3 means read the input of the pin.
- Byte 2 is the pin number to operate on.
- Byte 3 is the configuration number (if the command byte is 1), the output of the pin (if the command byte is 2) or a dummy number (if the command byte is 3).

The configuration number used when configuring a slave's I/O pin is the same as used in earlier versions of Maximite MMBasic (with the SETPIN command) and can be any one of:

0	Not configured or inactive
1	Analog input
2	Digital input
3	Frequency input
4	Period input
5	Counting input
8	Digital output
9	Open collector digital output. In this mode SPIN() will also return the value on the output pin .

Following a command from the master that requests an input, the master must then issue a second I^2C command to read 12 bytes. The slave will respond by sending the value as a 12 character string.

This program can fall over if the master issues an incorrect command. For example, by trying to read from a pin that is not an input. If that occurs, an error will be generated and MMBasic will exit to the command prompt.

Rather than trap all the possible errors that the master can make, this program uses the watchdog timer. If an error does occur the watchdog timer will simply reboot the Micromite and the program will restart (because AUTORUN is on) and wait for the next message from the master. The master can tell that something was wrong because it would get a timeout.

This is the complete program running on the slave:

```
OPTION AUTORUN ON
DIM msg(2)                                  ' array used to hold the message
I2C SLAVE OPEN &H26, 0, 0, WriteD, ReadD   ' slave's address is 26 (hex)

DO                                          ' the program loops forever
  WATCHDOG 1000                             ' this will recover from errors
LOOP

ReadD:                                      ' received a message
  I2C SLAVE READ 3, msg(0), recvd           ' get the message into the array
  IF msg(0) = 1 THEN                        ' command = 1
    SETPIN msg(1), msg(2)                   ' configure the I/O pin
  ELSEIF msg(0) = 2 THEN                    ' command = 2
    PIN(msg(1)) = msg(2)                    ' set the I/O pin's output
  ELSE                                      ' the command must be 3
    s$ = str$(pin(msg(1))) + Space$(12)     ' get the input on the I/O pin
  ENDIF
  IRETURN                                   ' return from the interrupt

WriteD:                                     ' request from the master
  I2C SLAVE WRITE 12, s$                    ' send the last measurement
  IRETURN                                   ' return from the interrupt
```

Interface Routines On the Master:

These routines run on the Maximite. They assume that the slave Micromite is listening on I^2C address 26 (hex).

If necessary these can be modified to access multiple Micromites (with different addresses), all acting as expansion chips and providing an almost unlimited expansion capability.

There are two subroutines and one function that together are used to control the slave:

SSETPIN pin, cfg	This subroutine will setup an I/O pin on the slave. It operates the same as the MMBasic SETPIN command and the possible values for 'cfg' are listed above.
SPIN pin, output	This subroutine will set the output of the slave's pin to 'output' (ie, high or low).
nn = SPIN(pin)	This function will return the value of the input on the slave's I/O pin.

For example, to display the voltage on pin 3 of the slave you would use:

```
    SSETPIN 3, 1
    PRINT SPIN(3)
```

As another example, to flash a LED connected to pin 15 of the slave you would use:

```
    SSETPIN 15, 8
    SPIN 15, 1
    PAUSE 300
    SPIN 15, 0
```

These are the three routines:

```
' configure an I/O pin on the slave
SUB SSETPIN pinnbr, cfg
  I2C OPEN 100, 1000
  I2C WRITE &H26, 0, 3, 1, pinnbr, cfg
  IF MM.I2C THEN ERROR "Slave did not respond"
  I2C CLOSE
END SUB
```

```
' set the output of an I/O pin on the slave
SUB SPIN pinnbr, dat
  I2C OPEN 100, 1000
  I2C WRITE &H26, 0, 3, 2, pinnbr, dat
  IF MM.I2C THEN ERROR "Slave did not respond"
  I2C CLOSE
END SUB

' get the input of an I/O pin on the slave
FUNCTION SPIN(pinnbr)
  LOCAL t$
  I2C OPEN 100, 1000
  I2C WRITE &H26, 0, 3, 3, pinnbr, 0
  I2C READ &H26, 0, 12, t$
  IF MM.I2C THEN ERROR "Slave did not respond"
  I2C CLOSE
  SPin = VAL(t$)
END FUNCTION
```

These use the new names for the I^2C functions so, on the Maximite, version 4.5 or later of MMBasic will be required. Earlier versions of MMBasic on the Maximite will also work but the I^2C command names will have to be changed to the old standard. Also, the method of getting the string from the slave in `SPIN(pinnbr)` will have to be changed (earlier versions did not support receiving data into a string variable).

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 four 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
OWSEARCH pin, srchflag, ser [,ser…]	

Where:

pin - The Maximite I/O pin 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 receive. The number of data items must agree with the length parameter.

srchflag - a combination of the following options:

1 - start a new search

2 - only return devices in alarm state

4 - search for devices in the requested family (first byte of ser)

8 - skip the current device family and return the next device

16 - verify that the device with the serial number in ser is available

If srchflag = 0 (or 2) then the search will return the next device found

ser - serial number (8 bytes) will be returned (srchflag 4 and 16 will also use the values in ser)

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 OW RESET command and the OW READ and OW WRITE commands if a reset was requested (flag = 1 or 2).

For users of previous versions of MMBasic

This implementation of the 1-Wire protocol is generally compatible with previous versions of MMBasic with the following differences:

- The commands are now two words where previously they were one word. For example, OWWRITE is now ONEWIRE WRITE.
- You cannot use an array or string variable for 'data'. One or more numeric variables are required.
- The reset command (ONEWIRE RESET) does not accept a 'presence' variable (use the MM.ONEWIRE variable instead).
- The OWCRC8() and OWCRC16() functions are not implemented.

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

Appendix D
SPI Communications

Maximite family only (not DOS or Generic PIC32 versions).

The Serial Peripheral Interface (SPI) communications protocol is used to send and receive data between integrated circuits.

The SPI function in MMBasic acts as the master (ie, MMBasic generates the clock).

The syntax of the function is:

```
received_data = SPI( rx, tx, clk, data_to_send, speed, mode, bits )
```

Data_to_send, speed, mode and bits are all optional. If not required they can be represented by either empty space between the commas or left off the end of the list.

Where:

- ‘rx’ is the pin number for the data input (MISO)
- ‘tx’ is the pin number for the data output (MOSI)
- ‘clk’ is the pin number for the clock generated by MMBasic (CLK)
- ‘data_to_send’ is optional and is an integer representing the data to send over the output pin. If it is not specified the ‘tx’ pin will be held low.
- ‘speed’ is optional and is the speed of the clock. It is a single letter either H, M or L where H is 3 MHz, M is 500 KHz and L is 50 KHz. Default is H.
- 'mode' is optional and is a single numeric digit representing the transmission mode – see Transmission Format below. The default mode is 3.
- 'bits' is optional and represents the number of bits to send/receive. Range is 1 to 23 (this limit is defined by how many bits can be stored in a floating point number). The default is 8.

The SPI function will return the data received during the transaction as an integer. Note that a single SPI transaction will send data while simultaneously receiving data from the slave (which is often discarded).

Examples

Using all the defaults:

```
A = SPI(11, 12, 13)
```

Specifying the data to be sent:

```
A = SPI(11, 12, 13, &HE4)
```

Setting the mode but using the defaults for data to send and speed:

```
A = SPI(11, 12, 13, , , 2)
```

An example specifying everything including a 12 bit data transfer:

```
A = SPI(11, 12, 13, &HE4, M, 2, 12)
```

Transmission Format

The most significant bit is sent and received first. The format of the transmission can be specified by the 'mode' as follows:

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

I/O Pins

Before invoking this function the ‘rx’ pin must be configured as an input using the SETPIN command and the ‘tx’ and ‘clk’ pins must be configured as outputs (either normal or open collector) again using the SETPIN command. The clock pin should also be set to the correct polarity (using the PIN function) before the SETPIN command so that it starts as inactive.

The SPI enable signal is often used to select a slave and “prime” it for data transfer. This signal is not generated by this function and (if required) should be generated using the PIN function on another pin.

The SPI function does not “take control” of the I/O pins like the serial and I^2C protocols and the PIN command will continue to operate as normal on them. Also, because the I/O pins can be changed between function calls it is possible to communicate with many different SPI slaves on different I/O pins.

Example

The following example will send the command 80 (hex) and receive two bytes from the slave SPI device. Because the mode, speed and number of bits are not specified the defaults are used.

```
SETPIN 18, 2                    ‘ set rx pin as a digital input
SETPIN 19, 8                    ‘ set tx pin as an output
PIN(20) = 1 : SETPIN 20, 8      ‘ set clk pin high then set it as an output
PIN(11) = 1 : SETPIN 11, 8      ‘ pin 11 will be used as the enable signal

PIN(11) = 0                     ‘ assert the enable line (active low)
junk = SPI(18, 19, 20, &H80)    ‘ send the command and ignore the return
byte1 = SPI(18, 19, 20)         ‘ get the first byte from the slave
byte2 = SPI(18, 19, 20)         ‘ get the second byte from the slave
PIN(11) = 1                     ‘ deselect the slave
```

Appendix E
Loadable Fonts

Maximite family only (not DOS or Generic PIC32 versions).

This section describes the format of a font file that can be loaded using the FONT LOAD command.

A font file is just a text file containing ordinary characters which are loaded line by line to build the bitmap of each character in the font. Each character can be up to 64 pixels high and 255 pixels wide.

The first non-comment line in the file must be the specifications for the font as follows:

```
height, width, start, end
```

Where 'height' and 'width' are the size of each character in pixels, 'start' is the number in the ASCII chart where the first character sits and 'end' is the last character. The character numbers must be in the range of 20 to 126 (decimal). Each number is separated by a comma. So, for example, `16, 11, 48, 57` means that the font is 16 pixels high and 11 wide. The first character is decimal 48 (the zero character) and the last is 57 (number nine character).

The remainder of the lines specify the bitmap for each character.

Each line represents a horizontal row of pixels. A space means the pixel is not illuminated and any other character will turn the pixel on. If the font is 11 pixels wide there must be 11 characters in the line although trailing spaces can be omitted. The first line is the top row of pixels in the character, the next is the second and so on. If the character is 16 pixels high there must be 16 lines to define the character. This repeats until each character is drawn. Using the above example of a font 16x11 with 10 characters there must be a total of 160 lines with each line 11 characters wide. This is in addition to the specification line at the top.

A comment line has an apostrophe (') as the first character and can occur anywhere. A comment line is completely ignored; all other lines are significant.

The following example creates two small icons; a smiley face and a frowning face. Each is 11x11 pixels with the first (the smiley face) in the position of the zero character (0) and the frowning face in the position of number one (1). To display a smiley face your program would contain this:

```
40 FONT LOAD "FACES.FNT" AS #6  ' load the font
50 FONT #6                      ' select the font
60 PRINT "0"                    ' print a smiley face
```

```
' example
' FACES.FNT
11,11,48,49

    XXX
  XX   XX
 XX     XX
XX  X X  XX
X         X
XX X   X XX
 X  XXX  X
  XX   XX
    XXX


    XXX
  XX   XX
 XX     XX
XX  X X  XX
X         X
XX  XXX  XX
 X X   X X
  XX   XX
    XXX
```

Appendix F
Special Keyboard Keys

Maximite family only (not DOS or Generic PIC32 versions).

MMBasic generates a single unique character for the function keys and other special keys on the keyboard. These are shown in the table as hexadecimal and decimal numbers:

Keyboard Key	Key Code (Hex)	Key Code (Decimal)
Up Arrow	80	128
Down Arrow	81	129
Left Arrow	82	130
Right Arrow	83	131
Insert	84	132
Home	86	134
End	87	135
Page Up	88	136
Page Down	89	137
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

If the control key is simultaneously pressed then 20 (hex) is added to the code (this is the equivalent of setting bit 5). If the shift key is simultaneously pressed then 40 (hex) is added to the code (this is the equivalent of setting bit 6). If both are pressed 60 (hex) is added. For example Control-PageDown will generate A9 (hex). The shift modifier only works with the function keys F1 to F12; it is ignored for the other keys.

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. This is particularly useful when using the EDIT command.

Appendix G
Tera Term Setup

Maximite family only (not DOS or Generic PIC32 versions).

MMBasic creates a virtual serial port over USB so that you can communicate with it from a Windows, Linux or Macintosh computer using nothing more than the USB port.

The communications protocol used is the CDC (Communication Device Class) protocol and there is native support for this in Linux (the cdc-acm driver) and Apple OS/X. Macintosh users can refer to the document "Using Serial Over USB on the Macintosh" on http://geoffg.net/maximite.html. The rest of this tutorial assumes that you are using a computer running Windows XP, Vista or 7.

First you need to install the Windows Serial Port Driver (available from http://geoffg.net/maximite.html). Full instructions are included in the download and when you have finished you should see the connection in Device Manager as a numbered communications port (eg, COM13).

To communicate with MMBasic over this virtual serial port you need to use a terminal emulator. This is a program that emulates the old fashioned VT100 terminal over a serial communications link. There are quite a few free emulators that you can use but I recommend Tera Term.

1. You should download Tera Term from http://en.sourceforge.jp/projects/ttssh2/releases/ and install it. These instructions are based on version 4.71.
2. Make sure that the USB cable is plugged into your PC and that you know the COM number.
3. When you run Tera Term for the first time you will get a dialog box asking you to select the type of connection. Select serial and select the correct COM number. You should then see the MMBasic prompt as shown below:

4. Before you start using Tera Term you need to make a few changes to the setup:

   Select **Setup -> Terminal…**

   Set the terminal size to 80 x 36.

   Untick the tick box labelled "term size = win size".

   Tick the box labelled "auto window resize".

   Select **Setup -> Serial Port…**

   Make sure that the port matches the COM number representing the Maximite.

   In the box for the "transmit delay msec/line" enter 50. Leave the other box with zero in it.

   You do not have to bother with the baud rate or any other settings.

   Select **Setup -> Save Setup…**

   Save the setup as TERATERM.INI in the Tera Term installation directory overwriting the file there.

Appendix H
Sprites

Maximite family only (not DOS or Generic PIC32 versions).

A sprite is a 16x16 bit graphic image that can be moved about on the screen independently of the background. When the sprite is displayed MMBasic will automatically save the background text and graphics under the sprite and when the sprite is turned off or moved MMBasic will restore the background.

The sprites are defined in a file which is loaded into memory using the SPRITE LOAD command, the number of sprites contained in the file is only limited by the amount of available memory. Each sprite in the file can contain pixels of any colour (on the Colour Maximite) and can also have transparent pixels which allow the background to show through. See below for a detailed description of creating a sprite file.

Manipulating Sprites

To manipulate the sprites you can use the command SPRITE ON which will display a specific sprite at a specified location on the screen. SPRITE MOVE will move a sprite to a new location and restore the background at the old location. SPRITE OFF will remove a sprite from the screen and restore the background.

Sprites should not overlap but if they do you should turn them off in the reverse sequence that you turned them on before you turn them on again at their new location. This will enable the background image to be correctly maintained.

For example, the following two sprites overlap:

```
SPRITE ON 1, 100, 150      ' sprite 1 is drawn at x = 100, y = 150
SPRITE ON 2, 110, 160      ' sprite 2 overlaps
```

To move the sprites they need to be turned off in the reverse sequence:

```
SPRITE OFF 2
SPRITE OFF 1
```

Then they can be redrawn at their new location:

```
SPRITE ON 1, 104, 154      ' sprite 1 is drawn at x = 104, y = 154
SPRITE ON 2, 116, 166      ' sprite 2 still overlaps
```

Because sprites are drawn so fast the user is unaware that the sprite has been turned off then redrawn.

Specifying the Background Colour

An alternative to turning sprites off in sequence is to specify the background colour when using the SPRITE ON or SPRITE MOVE commands. The background colour is optional and is specified at the end of the command. For example: SPRITE ON 1, 100, 100, BLUE

This results in a much faster operation when using a solid background colour because MMBasic does not have to copy the background to a buffer. It also means the MMBasic will always restore the correct colour, even if sprites overlap.

Collision Detection

You can use the COLLISION() function to detect if a sprite has collided with another sprite or the edges of the screen. A collision is reported if the non transparent portion of the sprite is just touching (ie, the non transparent pixels are adjacent) or overlapping the non transparent portion of another sprite or the edge of the screen.

To detect if a sprite has collided with another sprite you use: R = COLLISION (n, SPRITE)

And to detect if it has collided with the screen edge you use: R = COLLISION (n, EDGE)

Where 'n' is the number of the sprite to test.

In both cases the value returned by the function indicates if the collision was on the left of the sprite, the right, the top, etc. Following a collision COLLISION () will return:

&B0001	Indicating a collision with something on the left of the sprite
&B0010	Collision on the right
&B0100	Collision on the top
&B1000	Collision on the bottom

Note that it is possible for these results to be combined. For example; a result of &B0101 indicates that the sprite has collided with something both at the top and left of the sprite (for example the top left corner of the screen). When testing for collisions with other sprites it is possible for the function to return &B1111 indicating that there are collisions on all sides. This can happen if the sprite is surrounded on all sides by other sprites.

If the sprite is overlapping another (ie, one or more non transparent pixels are on top of another sprite's non transparent pixels) bit &B10000 will be set in the value returned by COLLISION () in addition to the bits for left, right, etc as described above.

Format of a Sprite File

A sprite file is similar to a font file except that it contains the definition of sprites which are 16x16 bit graphical objects. The sprite file is just a text file containing ordinary characters which are loaded line by line to build the bitmap of each sprite. Currently the dimensions of each sprite are fixed at 16x16 bits although alternative sizes may be allowed in the future.

The first non-comment line in the file must be the specifications for the sprite file as follows:

```
dimension, number
```

Where 'dimension' is the height and width of the sprites in pixels. At this time it <u>must</u> be the number 16. 'number' is the number of sprites in the file and is limited only by the amount of free memory available. The remainder of the lines specify the bitmap for each sprite.

Each line represents a horizontal row of pixels with each character in the line defining the colour of the pixel. The character can be a single numeric digit in the range of 0 to 7 representing the colours black to white or it can be a space which means that that particular pixel will be transparent (ie, the background will show through). On the monochrome Maximite 0 represents a black pixel and any other number represents a white pixel.

Each sprite must immediately follow the preceding sprite in the file and be defined by 16 lines each of 16 characters wide (although trailing spaces can be omitted and will be assumed to be transparent pixels).

A comment line has an apostrophe (') as the first character and can occur anywhere. A comment line is completely ignored; all other lines are significant.

The following example is of a file that contains a single sprite consisting of a red ball with a white border and a blue centre dot. On the monochrome Maximite this would display as a white ball.:

```
' example sprite
' TEST.SPR
16, 1
      7777
    74444447
  744444444447
 7444444444447
 7444444444447
7444444444444447
7444444114444447
7444441111444447
7444441111444447
7444444114444447
7444444444444447
 7444444444447
 7444444444447
  744444444447
    74444447
      7777
```

Appendix I
Random File I/O

This appendix describes how to implement random access files with fixed length records. This is made possible by opening the file for RANDOM access then using the SEEK command to position the read/write pointer within the file.

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

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

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

```
SEEK #1, 257
```

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

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

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

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

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

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

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

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

This program can be found in the MMBasic Library (http://geoffg.net/maximite.html#Downloads).

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
```

Introduction to Programming with the Colour Maximite 2

Geoff Graham

Version 2 (June 2020)

Copyright 2017 - 2020 Geoff Graham
Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0)

Table of Contents

GETTING STARTED 4

A QUICK PROGRAM 4

INTERACTING WITH MMBASIC 6

BREAK KEY 6
SHORTCUT KEYS 7
RUNNING AND EDITING PROGRAMS 7
FILE MANAGER 8
THE EDITOR 9
TRANSFERRING A PROGRAM FROM A PC 10
MMEDIT 11

PROGRAMMING FUNDAMENTALS 12

STRUCTURE OF A BASIC PROGRAM 12
THE PRINT COMMAND 13
VARIABLES 14
EXPRESSIONS 15
THE IF STATEMENT 16
FOR LOOPS 17
MULTIPLICATION TABLE 18
DO LOOPS 19
CONSOLE INPUT 20
GOTO AND LABELS 21
TESTING FOR PRIME NUMBERS 22

ADVANCED BASIC PROGRAMMING 24

COMMENTS 24
UTILITY COMMANDS 25
ARRAYS 25
INTEGERS 26
STRINGS 27
MANIPULATING STRINGS 28
SCIENTIFIC NOTATION 29
DIM COMMAND 29
CONSTANTS 31
SUBROUTINES 31
FUNCTIONS 32
LOCAL VARIABLES 33
STATIC VARIABLES 34
CALCULATE DAYS 35
GOOD PROGRAMMING HABITS 37

SD CARD AND FILES 39

COMMAND SUMMARY 39
SEQUENTIAL FILE ACCESS 41
SEQUENTIAL ACCESS EXAMPLE 42
SAVING NUMERIC DATA TO AN SD CARD 43
RANDOM FILE ACCESS 45

GRAPHICS ON THE VGA MONITOR 47

GRAPHIC COORDINATES 47
DEFINING COLOUR 47
DRAWING ON THE SCREEN 48
EXAMPLES 49
FONTS 50
TEXT COMMAND 51
STARS 52
TWINKLING STARS 53
VIDEO MODES 54
DISPLAYING IMAGES 54
VIDEO PAGES 55
BLIT COMMAND 56
GAME PLAYING FEATURES 57

EXTERNAL INPUT/OUTPUT 58

CONFIGURING A PIN 59
DIGITAL INPUTS 60
USING A SWITCH AS AN INPUT 61
DIGITAL OUTPUTS 61
ANALOG INPUT 63
FREQUENCY AND PERIOD MEASUREMENT 64
INTERRUPTS 64
ROTARY ENCODERS 66
PWM AND SERVO OUTPUTS 67
SPECIAL DEVICE SUPPORT 68

COMMUNICATION PROTOCOLS 69

ASYNCHRONOUS SERIAL COMMUNICATIONS 69
I^2C COMMUNICATIONS 71
SPI COMMUNICATIONS 71
1-WIRE COMMUNICATIONS 72

SPECIAL FEATURES 73

SETTING OPTIONS 73
KEEPING TIME 73
AUTORUN 75
RECOVERING FROM ERRORS 75
SAVING DATA 76
PLAYING MUSIC AND SOUND EFFECTS 77
SORTING DATA 77

Chapter 1

Getting Started

The Colour Maximite 2 was inspired by the home computers of the early 80s like the Tandy TRS-80, Commodore 64 and the Apple II. It is a fun computer with a focus on ease of use and the ability to whip up a quick program for whatever you want.

Since home computers were introduced in the late 70s they have evolved to become much more sophisticated - which is generally good because they can now do so much. But in this quest for increasing power the fun element of writing a simple program and getting it to run has been lost. Modern programming languages are complex and you cannot easily get in there to experiment and have fun learning the art of programming.

The Colour Maximite 2 brings back this fun element, it starts up in under a second and with a few keystrokes you can have it doing something. It can draw graphics on the screen, save programs and data to its SD card and control the external world through its I/O connector on the back panel.

This tutorial is focused on introducing the reader to programming using the Colour Maximite 2. It does not cover building the computer which is the subject of the *CMM2 Construction Pack* and it does not cover the details of every command.

So it would be worthwhile downloading the *Colour Maximite 2 User Manual* and having it handy as you read through the following pages. That way you can explore the full detail of a command or feature that might interest you. The Colour Maximite 2 firmware, manuals and the construction pack can be downloaded from: http://geoffg.net/maximite.html

This tutorial assumes that you have a working Colour Maximite 2, you have connected it to a keyboard and monitor and that you have the command prompt (the greater than symbol “>”) displayed on the screen.

A Quick Program

Before we get into the details of programming in MMBasic a quick demonstration of how the Colour Maximite 2 works will give you a taste of how easy it is to use. For this we will enter and run a “Hello World” program. The aim of this is to create a program that will simply print out the words *Hello World* and this is often used by programmers as a test of the steps required to enter a program and get the computer to run it.

This does not sound complicated but on a modern PC (ie, Windows 10 or MacOS) it can be a daunting procedure involving installing multiple programs, reading manuals and dealing with some very technical subjects.

On the Colour Maximite 2 it is simple and easy:

- Make sure that you have a properly formatted SD card installed in the front panel slot.
- At the command prompt enter: `EDIT "hello"`. This will take you into the editor. For the moment all that you need to know about the editor is that the arrow keys will move your cursor around and the delete key will delete the character at the cursor.
- Enter this single line: `PRINT "Hello World"`
- Press the F1 key on your keyboard. This will save your program to the SD card and exit back to the command prompt.
- At the command prompt enter: `RUN`

And you should see on the screen the words *Hello World* followed by the command prompt. Congratulations, you have entered and run your first program on the Colour Maximite 2.

If something went wrong you will get an error message which will describe the error. To correct this you can just press the F4 key on your keyboard which will run the editor for you and place the cursor on the line that caused the problem. You can then correct the error and save the program using F1 and run it again.

It is that easy.

Chapter 2

Interacting With MMBasic

The user interacts with the Colour Maximite 2 via the console at the command prompt (ie, the greater than symbol (>) on the console). On startup the Colour Maximite 2 will issue the command prompt and wait for some command to be entered. It will also return to the command prompt if your program ends or if it generated an error message.

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

Almost any command can be entered at the command prompt and this is often used to test a command to see how it works. A simple example is the PRINT command (more on this in Chapter 3), which you can test by entering the following at the command prompt:

```
PRINT 2 + 2
```

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

This ability to test a command at the command prompt is very useful when you are learning to program in BASIC, so it would be worthwhile having a Colour Maximite 2 handy for the occasional test while you are working through this tutorial.

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

The status line will automatically disappear when you run a program and it will reappear when your program finishes and returns to the command prompt. You can banish it forever (or bring it back) by using the `OPTION STATUS` command.

Break Key

One useful feature is the CTRL-C sequence (hold down the CTRL key then press the C key). This is called the break key or character. When you type this on the console's input it will interrupt whatever MMBasic is doing and immediately return control to the command prompt.

This can get you out of all sorts of difficult situations. For example, if you entered the following at the command prompt you would cause MMBasic to enter a continuous loop and appear to be unresponsive.

```
DO : LOOP
```

If you have a Colour Maximite 2 handy you can try entering this line and you will see that the command prompt does not return because MMBasic is busy spinning in a loop. Then try typing CTRL-C on the console and MMBasic will immediately break out of the loop and return to the command prompt.

Remember CTRL-C because it will prove useful at some time in the future.

Shortcut Keys

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

F1	FILES
F2	RUN
F3	LIST
F4	EDIT

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

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

Function keys F11 and F12 can be programmed with custom text:

F11	*User specified string – See the* OPTION F11 *Command.*
F12	*User specified string – See the* OPTION F12 *Command.*

A handy feature of the command prompt is that it will remember your past commands and you can access these by using the up/down arrow keys to step through the list. You can also edit any entry using the left/right arrow keys as well as delete and backspace. The insert key will toggle the insert/overtype mode for when you type in some new text.

Running and Editing Programs

All programs reside on the SD card which acts as the "disk drive" for the computer. As a result the SD card must be present for most operations. This is different from the original Maximite where the SD card was not necessarily required.

When you edit a program you are editing the program on the SD card, when you run a program you will run it from the SD card, etc. The reason for this arrangement is that when a program is loaded into memory MMBasic will compress it to improve performance – this also means that it does not resemble the original program which why RUN, EDIT, etc always reference the original program on the SD card.

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 screenfull 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, because the filename is missing, they will act on "MyProg.bas" on the SD card.

File Manager

The Colour Maximite 2 includes a file manager which provides an easy method of managing the files and directories on the SD card. Using this you can search, delete, rename, run, etc. To start the file manager you use the command FILES at the command prompt or press the F1 key on your keyboard. This will start up in the current directory and list the files and directories there.

On the VGA monitor it will look like this:

To move around the list of files you use the arrow keys, Page Up or Page Down keys and the Home or End keys. Pressing Enter when positioned on a directory will take you into a directory and list that directory. If the directory has the name `" . . "` the Enter key will take you up one level in the directory hierarchy. The Escape key (ESC) will exit the file manager.

At the bottom of the screen the status line lists the current cursor position and the common keystrokes. All these operate on the file currently selected by the cursor:

Enter	This is the action key. If the file is a program this will RUN the program. If the file is an audio file this will PLAY the file on the sound output. If the cursor is positioned on a directory that directory will be entered.
F3	This will LIST the program or text file selected.
F4	This will EDIT the program or text file selected.
F5	Will prompt for a directory name and create that directory.
CTRL-C	Will prompt for a file name and copy the selected file to that new name.

CTRL-F	Will enter the **find mode**. You will be prompted for the search text and as you type this in the cursor will automatically position at the first matching file found. You can then use the down arrow key to search for the next occurrence or the up arrow key for the previous occurrence. The Enter key will leave the cursor where it is and return to normal mode. Escape will abort the search.
CTRL-K	Will delete a file or directory (a directory must be empty).
CTRL-R	Will rename a file or directory.
CTRL-S	This will toggle the sort order between name, size, type and date.

The Editor

The Colour Maximite 2 has its own built in program editor which can be used to enter programs and correct them when errors are discovered. This screen shot shows the editor in action with colour coded text. Commands are in cyan, comments in yellow, constants in green and so on as shown below.

The best way to understand the editor is to try it out. At the command prompt enter the command EDIT followed by the name of the file to edit. For example:

```
EDIT "myprog.bas"
```

and the editor will startup displaying an empty screen with a help line at the bottom of the screen. You can then just type in your program. For example, try typing in:

```
PRINT 1/7
```

Then press the F2 key on your keyboard. This will save the program and run it. This will display the result of dividing 1 by 7.

To change this program use the command EDIT again (or press the shortcut key F4) and you will be taken back into the editor with your program displayed ready for editing.

If you have used an editor like Windows Notepad 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. 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.

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

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

ESC	This will cause the editor to abandon all changes and return to the command prompt with the program memory unchanged. If you have changed the text you will be asked to press ESC twice more to confirm this action.
F1	This will save the program and return to the command prompt.
F2	This will save the program and immediately run it.
F3 or CTRL-F	Will enter the **find mode**. You will be prompted for the text to be found and as you type this in the editor will automatically position the cursor at the first matching text. You can then use the down arrow key to search for the next occurrence or the up arrow key for the previous occurrence. The Enter key will leave the cursor positioned on thefound text and return to normal editing mode. F5 or CTRL-V will replace the found text with whatever is in the clipboard (see below). Escape will abort the search.
F4 or CTRL-S	This will enter the **select mode**. In this mode you can use the arrow keys, HOME or END to select text and copy it to the clipboard. It will be highlighted on the screen as you select it. Then F5 or CTRL-C will copy the selection to the clipboard, F4 or CTRL-X will copy and delete the selection. DELETE will simply delete the selection and ESCAPE will return to the normal editing mode without changing anything.
F5 or CTRL-V	This will insert (at the current cursor position) the text that had been previously cut or copied in the mark mode (see above).
F6	This will save the edited text and exit the editor similar to the F1 key. The difference is that F6 will not update the “current program name” which is used when the RUN, LIST and EDIT commands are entered without specifying a filename. .
CTRL-K	Will delete all text from the current cursor position to the end of the line.
CTRL-W	Will allow you to save the contents of the editor to a different file. This can be used to save a backup copy while continuing to edit the original
F7	Will prompt for a file name and will insert the text from that file into the editor at the current cursor position.

Transferring a Program from a PC

The simplest way to transfer a program from a personal computer is to simply save the file to an SD card and insert that in the Colour Maximite 2.

Alternatively you can transfer it via the serial console using either the AUTOSAVE or XMODEM commands. This requires you to have a terminal emulator running on your desktop machine and connected to the serial console of the Colour Maximite 2. How to connect to the serial console via USB is described in the *Colour Maximite 2 User Manual*.

MMEdit

Another convenient method of creating your programs and sending it to the Colour Maximite 2 is to use MMEdit. This program was written by Jim Hiley from northern Tasmania in Australia and is intended to work with the Colour Maximite 2 so the two work well together. It can be installed on a Windows computer and it allows you to edit your program on a PC then, with a single button click, transfer it to the Colour Maximite 2 for testing.

MMEdit also requires that you have the serial console up and running as described in the *Colour Maximite 2 User Manual.*

MMEDIT is easy to use with colour coded text, mouse based cut and paste and many more useful features such as bookmarks and automatic indenting. Because the program runs on your PC you can save and load your programs to and from the computer's hard disk. This screen shot shows MMEDIT in action.

MM Edit untitled.bas
File Edit Program Font View BookMarks Help
Lines: 147
77 '''''''

Chapter 3

Programming Fundamentals

The Colour Maximite 2 is programmed using the BASIC programming language. This version of BASIC is called MMBasic (short for Maximite BASIC) which is loosely based on the Microsoft BASIC interpreter that was popular years ago.

The BASIC language was introduced in 1964 by Dartmouth College in the USA as a computer language for teaching programming and accordingly it is easy to use and learn. At the same time, it has proved to be a competent and powerful programming language and as a result it became very popular in the late 70s and early 80s. Even today some large commercial data systems are still written in the BASIC language (primarily Pick Basic).

For the Colour Maximite 2 the greatest advantage of BASIC is its ease of use. Some more modern languages such as C and C++ can be truly mind bending but with BASIC you can start with a one line program and get something sensible out of it. MMBasic is also powerful in that you can draw sophisticated graphics, manipulate the external I/O pins to control other devices and communicate with other devices using a range of built-in communications protocols.

Structure of a BASIC Program

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 interchangeable in this tutorial).

Normally each statement is on its own line but you can have multiple statements in the one line separated by the colon character (:).

For example;

```
A = 24.6 : PRINT A
```

Each line can start with a line number. Line numbers were mandatory in the early BASIC interpreters however modern implementations (such as MMBasic) do not need them. You can still use them if you wish but they have no benefit and generally just clutter up your programs.

This is an example of a program that uses line numbers:

```
50 A = 24.6
60 PRINT A
```

A line can also start with a label which can be used as the target for a program jump using the GOTO command. This will be explained in more detail when we cover the GOTO command but this is an example (the label name is `JmpBack`):

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

The PRINT Command

There are a number of common commands that are fundamental and we will cover them in this chapter but arguably the most useful is the PRINT command. Its job is simple; to print something on the console. This is mostly used to output data for you to see (like the result of calculations) or provide informative messages.

PRINT is also useful when you are tracing a fault in your program; you can use it to print out the values of variables and display messages at key stages in the execution of the program.

In its simplest form the command will just print whatever is on its command line. So, for example:

```
PRINT 54
```

Will display on the console the number 54 followed by a new line.

The data to be printed can be something simple like this or an expression, which means something to be calculated. We will cover expressions in more detail later but as an example the following:

```
> PRINT 3/21
 0.1428571429
>
```

would calculate the result of three divided by twenty one and display it. Note that the greater than symbol (>) is the command prompt produced by MMBasic – you do not type that in.

Other examples of the PRINT command include:

```
> PRINT "Wonderful World"
Wonderful World
> PRINT (999 + 1) / 5
 200
>
```

You can try these out at the command prompt.

The PRINT command will also work with multiple values at the same time, for example:

```
> PRINT "The amount is" 345 " and the second amount is" 456
The amount is 345 and the second amount is 456
>
```

Normally each value is separated by a space character as shown in the previous example but you can also separate values with a comma (,). The comma will cause a tab to be inserted between the two values. In MMBasic tabs in the PRINT command are eight characters apart. To illustrate tabbing the following command prints a tabbed list of numbers:

```
> PRINT 12, 34, 9.4, 1000
 12      34      9.4     1000
>
```

Note that there is a space printed before the number 12. This space is a place holder for the minus symbol (-) in case the value is negative. Notice the difference with this example:

```
> PRINT -12, 34, -9.4, 1000
-12      34     -9.4     1000
>
```

The print statement can be terminated with a semicolon (;). This will prevent the PRINT command from moving to a new line when it completes printing all the text. For example:

```
PRINT "This will be";
PRINT " printed on a single line."
```

Will result in this output:

```
This will be printed on a single line.
```

The message would be look like this without the semicolon at the end of the first line:

```
This will be
 printed on a single line.
```

Variables

Before we go much further we need to define what a "variable" is as they are fundamental to the operation of the BASIC language (in fact, any programming language). A variable is simply a place to store an item of data (ie, its "value"). This value can be changed as the program runs which why it is called a "variable".

Variables in MMBasic can be one of three types. The most common is floating point and this is automatically assumed if the type of the variable is not specified. The other two types are integer and string and we will cover them later. A floating point number is an ordinary number which can contain a decimal point. For example 3.45 or -0.023 or 100.00 are all floating point numbers.

A variable can be used to store a number and it can then be used in the same manner as the number itself, in which case it will represent the value of the last number assigned to it.

As a simple example:

```
A = 3
B = 4
PRINT A + B
```

will display the number 7. In this case both A and B are variables and MMBasic used their current values in the PRINT statement. MMBasic will automatically create a variable when it first encounters it so the statement `A = 3` both created a floating point variable (the default type) with the name of `A` and then it assigned the value of 3 to it.

The name of a variable must start with a letter while the remainder of the name can use letters, numbers, the underscore or the full stop (or period) characters. The name can be up to 32 characters long and the case (ie, capitals or not) is not important. Here are some examples:

```
Total_Count
ForeColour
temp3
count
ThisIsALongVariableName
```

You can change the value of a variable anywhere in your program by using the assignment command, ie:

```
variable = expression
```

For example:

```
temp3 = 24.6
count = 5
CTemp = (FTemp - 32) * 0.5556
```

In the last example both CTemp and FTemp are variables and this line converts the value of FTemp (in degrees Fahrenheit) to degrees Celsius and stores the result in the variable CTemp.

Expressions

We have met the term ‘expression’ before in this tutorial and in programming it has a specific meaning. It is a formula which can be resolved by the BASIC interpreter to a single number or value.

MMBasic will evaluate a mathematical expression using the same rules that we all learnt at school. For example, multiplication and division are performed first followed by addition and subtraction. These are called the rules of precedence and are fully spelt out in the User Manual.

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

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

As pointed out earlier, you can use variables in an expression exactly the same as straight numbers. For example, this will increment the value of the variable `temp` by one:

```
temp = temp + 1
```

You can also use functions in expressions. These are special operations provided by MMBasic, for example to calculate trigonometric values. As an example, the following will print the length of the hypotenuse of a right angled triangle using the SQR() function which returns the square root of a number (`a` and `b` are variables holding the lengths of the other sides):

```
PRINT SQR(a * a + b * b)
```

MMBasic will first evaluate this expression by multiplying `a` by `a`, then multiplying `b` by `b`, then adding the results together. The resulting number is then passed to the `SQR()` function which will calculate the square root of that number and return it for the PRINT command to display.

Some other mathematical functions provided by MMBasic include:

SIN(r) – the sine of r

COS(r) – the cosine of r

TAN(r) – the tangent of r

There are many more functions available to you and they are all listed in the User Manual.

Note that in the above trigonometric functions the value passed to the function (ie, 'r') is the angle in radians. In MMBasic you can use the function RAD(d) to convert an angle from degrees to radians ('d' is the angle in degrees).

Another feature of BASIC is that you can nest function calls within each other. For example, given the angle in degrees (ie, 'd') the sine of that angle can be found with this expression:

```
PRINT SIN(RAD(d))
```

In this case MMBasic will first take the value of d and convert it to radians using the RAD() function. The output of this function then becomes the input to the SIN() function.

The IF statement

Making decisions is at the core of most computer programs and in BASIC that is usually done with the IF statement. This is written almost like an English sentence:

IF *condition* THEN *action*

The condition is usually a comparison such as equals, less than, more than, etc. For example:

```
IF Temp < 25 THEN PRINT "Cold"
```

`Temp` would be a variable holding the current temperature and `PRINT "Cold"` the action to be done. There are a range of tests that you can make:

=	equals	<>	not equal
<	less than	<=	less than or equals
>	greater than	>=	greater than or equals

You can also add an ELSE clause which will be executed if the initial condition tested false. For example this will execute different actions when the temperature is under 25 or 25 or more:

```
IF Temp < 25 THEN PRINT "Cold" ELSE PRINT "Hot"
```

The previous examples all used single line IF statements but you can also have multiline IF statements. They look like this:

```
IF condition THEN
    TrueActions
ENDIF
```

or

```
IF condition THEN
    TrueActions
ELSE
    FalseActions
ENDIF
```

Unlike the single line IF statement you can have many true actions with each on their own line and similarly many false actions. Generally the single line IF statement is handy if you have a simple action that needs to be taken while the multiline version is much easier to understand if the actions are numerous and more complicated.

An example of a multiline IF statement with more than one action is:

```
IF Amount < 25 THEN
    PRINT "Too low"
    PRINT "Minimum value is 25"
ELSE
    PRINT "Input accepted"
    SaveToSDCard
ENDIF
```

Note that in the above example each action is indented to show what part of the IF structure it belongs to. Indenting is not mandatory but it makes a program much easier to understand for someone who is not familiar with it and therefore it is highly recommended.

An expression like `Amount < 25` is evaluated by MMBasic as either true or false with true having a value of one and false zero. You can see this if you entered the following at the console:

```
PRINT 30 > 20
```

MMBasic will print 1 meaning that the value is true and similarly the following will print 0 meaning that the expression evaluated to false.

```
PRINT 30 < 20
```

The IF statement does not really care about what the condition actually is, it just evaluates the condition and if the result is zero it will take that as false and if non zero it will take it as true. This allows for some handy shortcuts. For example, if `BalanceCorrect` is a variable that is true (non zero) when some feature of the program is correct then the following can be used to make a decision based on that value:

```
IF BalanceCorrect THEN …do something…
```

FOR Loops

Another common requirement in programming is repeating a set of actions. For instance, you might want to step through all seven days in the week and perform the same function for each day. BASIC provides the FOR loop construct for this type of job and it works like this:

```
FOR day = 1 TO 7
  Do something based on the value of 'day'
NEXT day
```

This starts by creating the variable `day` and assigning the value of 1 to it. The program then will execute the following statements until it comes to the NEXT statement. This tells the BASIC interpreter to increment the value of `day`, go back to the previous FOR statement and re-execute the following statements a second time. This will continue looping around until the value of `day` exceeds 7 and the program will then exit the loop and continue with the statements following the NEXT statement.

As a simple example, you can print the numbers from one to ten like this:

```
FOR nbr = 1 TO 10
  PRINT nbr,;
NEXT nbr
```

The comma at the end of the PRINT statement tells the interpreter to tab to the next tab column after printing the number while the semicolon will leave the cursor on this line rather than automatically moving to the next line. As a result the numbers will be printed in neat columns across the page. This is what you would see:

```
1       2       3       4       5       6       7       8       9       10
```

The FOR loop also has a couple of extra tricks up it sleeve. You can change the amount that the variable is incremented by using the STEP keyword. So, for example, the following will print just the odd numbers:

```
FOR nbr = 1 TO 10 STEP 2
  PRINT nbr,;
NEXT nbr
```

The value of the step (or increment value) defaults to one if the STEP keyword is not used but you can set it to whatever number you want.

When MMBasic is incrementing the variable it will check to see if the variable has exceeded the TO value and, if it has, it will exit from the loop. So, in the above example, the value of `nbr` will reach nine and it will be printed but on the next loop `nbr` will be eleven and at that point execution will leave the loop. This test is also applied at the start of the loop (ie, if in the beginning the value of the variable exceeds the TO value the loop will never be executed, not even once).

By setting the STEP value to a negative number you can use the FOR loop to step down from a high number to low. For example, the following will print the numbers from 1 to 10 in reverse:

```
FOR nbr = 10 TO 1 STEP -1
  PRINT nbr,;
NEXT nbr
```

Multiplication Table

To further illustrate how loops work and how useful they can be, the following short program will use two FOR loops to print out the multiplication table that we all learnt at school. The program for this is not complicated:

```
FOR nbr1 = 1 to 10
  FOR nbr2 = 1 to 10
    PRINT nbr1 * nbr2,;
  NEXT nbr2
  PRINT
NEXT nbr1
```

The output will be similar to the screen grab below, which also shows a listing of the program.

You need to work through the logic of this example line by line to understand what it is doing. Essentially it consists of one loop inside another. The inner loop, which increments the variable `nbr2`, prints one horizontal line of the table. When this loop has finished it will execute the

following PRINT command which has nothing to print - so it will simply output a new line (ie, terminate the line printed by the inner loop).

The program will then execute another iteration of the outer loop by incrementing `nbr1` and re-executing the inner loop again. Finally, when the outer loop is exhausted (when `nbr1` exceeds 10) the program will reach the end and terminate.

One last point, you can omit the variable name from the NEXT statement and MMBasic will guess which variable you are referring to. However, it is good practice to include the name to make it easier for someone else who is reading the program. You can also terminate multiple loops using a comma separated list of variables in the NEXT statement. For example:

```
FOR var1 = 1 TO 5
  FOR var2 = 10 to 13
    PRINT var1 * var2
NEXT var1, var2
```

DO Loops

Another method of looping is the DO…LOOP structure which looks like this:

```
DO WHILE condition
   statement
   statement
LOOP
```

This will start by testing the condition and if it is true the statements will be executed until the LOOP command is reached, at which point the condition will be tested again and if it is still true the loop will execute again. The 'condition' is the same as in the IF command (ie, `X < Y`). For example, the following will keep printing the word "Hello" on the console for 4 seconds then stop:

```
Timer = 0
DO WHILE Timer < 4000
  PRINT "Hello"
LOOP
```

Note that `Timer` is a function within MMBasic which will return the time in milliseconds since the timer was reset. A reset is done by assigning zero to `Timer` (as done above) or when powering up the Colour Maximite 2. We will cover the timer in more detail later.

A variation on the DO-LOOP structure is the following:

```
DO
   statement
   statement
LOOP UNTIL condition
```

In this arrangement the loop is first executed once, the condition is then tested and if the condition is false, the loop will be repeatedly executed until the condition becomes true. Note that the test in LOOP UNTIL is the inverse of DO WHILE.

For example, similar to the previous example, the following will also print "Hello" for four seconds:

```
Timer = 0
DO
  PRINT "Hello"
LOOP UNTIL Timer >= 4000
```

Both forms of the DO-LOOP essentially do the same thing, so you can use whatever structure fits with the logic that you wish to implement.

Finally, it is possible to have a DO Loop that has no conditions at all - ie,

```
DO
  statement
  statement
LOOP
```

This construct will continue looping forever and you, as the programmer, will need to provide a way to explicitly exit the loop (the EXIT DO command will do this). For example:

```
Timer = 0
DO
  PRINT "Hello"
  IF Timer >= 4000 THEN EXIT DO
LOOP
```

Console Input

As well as printing data for the user to see your programs will also want to get input from the user. For that to work you need to capture keystrokes from the console and this can be done with the INPUT command. In its simplest form the command is:

```
INPUT var
```

This command will print a question mark on the console's screen and wait for a number to be entered followed by the Enter key. That number will then be assigned to the variable `var`.

For example, the following program extends the expression for finding the hypotenuse of a triangle by allowing the user to enter the lengths of the other sides from the console.

```
PRINT "Length of side 1"
INPUT a
PRINT "Length of side 2"
INPUT b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

This is a screen capture of a typical session:

The INPUT command can also print your prompt for you, so that you do not need a separate PRINT command. For example, this will work the same as the above program:

```
INPUT "Length of side 1"; a
INPUT "Length of side 2"; b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

Finally, the INPUT command will allow you to input a series of numbers separated by commas with each number being saved in different variables. For example:

```
INPUT "Enter the length of the two sides: ", a, b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

If the user entered `12,15` the number `12` would be saved in the variable `a` and `15` in `b`.

Another method of getting input from the console is the LINE INPUT command. This will get the whole line as typed by the user and allocate it to a string variable. Like the INPUT command you can also specify a prompt. This is a simple example:

```
LINE INPUT "What is your name? ", s$
PRINT "Hello " s$
```

We will cover string variables in the next chapter but for the moment you can think of them as a variable that holds a sequence of one or more characters. If you ran the above program and typed in `John` when prompted the program would respond with `Hello John`.

Sometimes you do not want to wait for the user to hit the enter key, you want to get each character as it is typed in. This can be done with the INKEY$ function which will return the value of the character as a string of one character long.

GOTO and Labels

One method of controlling the flow of the program is the GOTO command. This essentially tells MMBasic to jump to another part of the program and continue executing from there. The target of the GOTO is a label and this needs to be explained first.

A label is an identifier that marks part of the program. It must be the first thing on the line and it must be terminated with the colon (:) character. The name that you use can be up to 32 characters long and must follow the same rules for a variable's name. For example, in the following program line LoopBack is a label:

```
LoopBack:  a = a + 1
```

When you use the GOTO command to jump to that particular part of the program you would use the command like this:

```
GOTO LoopBack
```

To put all this into context the following program will print out all the numbers from 1 to 10:

```
z = 0
LoopBack:  z = z + 1
PRINT z
IF z < 10 THEN GOTO LoopBack
```

The program starts by setting the variable `z` to zero then incrementing it to 1 in the next line. The value of `z` is printed and then tested to see if it is less than 10. If it is less than 10 the program execution will jump back to the label `LoopBack` where the process will repeat. Eventually the value of `z` will be more than 10 and the program will run off the end and terminate.

Note that a FOR loop can do the same thing (and is simpler) so this example is purely designed to illustrate what the GOTO command can do.

In the past the GOTO command gained a bad reputation. This is because using GOTOs it is possible to create a program that continuously jumps from one point to another (often referred to as "spaghetti code") and that type of program is almost impossible for another programmer to understand. With constructs like the multiline IF statements the need for the GOTO statement has been reduced and it should be used only when there is no other way of changing the program's flow.

Testing for Prime Numbers

The following is a simple program which brings together many of the programming features previously discussed.

```
DO
  InpErr:
  PRINT
  INPUT "Enter a number: "; a
  IF a < 2 THEN
    PRINT "Number must be equal or greater than 2"
    GOTO InpErr
  ENDIF

  Divs = 0
  FOR x = 2 TO SQR(a)
    r = a/x
    IF r = FIX(r) THEN Divs = Divs + 1
  NEXT x

  PRINT a " is ";
  IF Divs > 0 THEN PRINT "not ";
  PRINT "a prime number."
LOOP
```

This will first prompt (on the console) for a number and, when it has been entered, it will test if that number is a prime number or not and display a suitable message.

It starts with a DO Loop that does not have a condition – so it will continue looping forever. This is what we want. It means that when the user has entered a number, it will report if it is a prime number or not and then loop around and ask for another number. The way that the user can exit the program (if they wanted to) is by typing the break character (normally CTRL-C).

The program then prints a prompt for the user which is terminated with a semicolon character. This means that the cursor is left at the end of the prompt for the INPUT command which will get the number and store it in the variable a.

Following this the number is tested. If it is less than 2 an error message will be printed and the program will jump backwards and ask for the number again.

We are now ready to test if the number is a prime number. The program uses a FOR loop to step through the possible divisors testing if each one can divide evenly into the entered number. Each time it does the program will increment the variable `Divs`. Note that the test is done with the function FIX(r) which simply strips off any digits after the decimal point. So, the condition `r = FIX(r)` will be true if `r` is an integer (ie, has no digits after the decimal point).

Finally, the program will construct the message for the user. The key part is that if the variable `Divs` is greater than zero it means that one or more numbers were found that could divide evenly into the test number. In that case the IF statement inserts the word "not" into the output message. For example, if the entered number was 21 the user will see this response:

```
21 is not a prime number.
```

This is the result of running the program and some of the output:

You can test this program by using the editor (the EDIT command) to enter it.

Using your newly learnt skills you could then have a shot at making it more efficient. For example, because the program counts how many times a number can be divided into the test number it takes a lot longer than it should to detect a non prime number. The program would run much more efficiently if it jumped out of the FOR loop at the first number that divided evenly. You could use the GOTO command to do this or you could use the command EXIT FOR – that would cause the FOR loop to terminate immediately.

Other efficiencies include only testing the division with odd numbers (by using an initial test for an even number then starting the FOR loop at 3 and using STEP 2) or by only using prime numbers for the test (that would be much more complicated).

Chapter 4

Advanced BASIC Programming

In the previous chapter we covered the fundamentals of BASIC programming, enough to write a short program to do a simple job. But BASIC has additional features that become important when you are constructing a more complex program and we will cover these in this chapter.

As stressed before, this tutorial is not intended as a comprehensive manual and as such there are many more specialised features that are not covered here. To discover these it is recommended that you download the *Colour Maximite 2 User Manual* from: http://geoffg.net/maximite.html

Comments

Before we go much further we should discuss some of the utility commands and features of MMBasic that help you manage and run your program.

The first is the comment which is any text that follows the single quote character ('). A comment can be placed anywhere and extends to the end of the line. If MMBasic runs into a comment it will just skip to the end of it (ie, it does not take any action regarding a comment).

Comments should be used to explain non obvious parts of the program and generally inform someone who is not familiar with the program how it works and what it is trying to do. Remember that after only a few months a program that you have written will have faded from your mind and will look strange when you pick it up again. For this reason you will thank yourself later if you use plenty of comments.

In many versions of the BASIC language comments take up valuable memory and restrict the size of the program that can be run – this also applies to the original Colour Maximite and the Micromite series of chips which also run MMBasic, This is different in the Colour Maximite 2. The firmware will strip out all comments and unnecessary spaces when the program is loaded into program memory. As a result you can use as many comments as you wish and without fear. This is a good thing because a program with lots of comments is much easier to understand.

The following are some examples of comments:

```
' calculate the hypotenuse
PRINT SQR(a * a + b * b)
```

or

```
INPUT var     ' get the temperature
```

Older BASIC programs used the command REM to start a comment and you can also use that if you wish but the single quote character is easier to use and more convenient.

Utility Commands

We have covered the EDIT command which will run the internal program editor. Other useful commands are FILES which will start the file manager, LIST which will list your program on the console (pausing every 24 lines) and the RUN command which will start your program running.

If you want to completely clear the program in memory you can use the NEW command which will reset the interpreter to its power up state including erasing the current program. The CLEAR command will do the same for any variables (ie, delete them and recover the memory). The MEMORY command will list how much memory is currently being used.

The TRACE command is useful if you are trying to work out what your program is doing wrong. TRACE ON will cause MMBasic to list the line number of each statement as it is executed and this can help you trace the program flow. TRACE OFF will stop this feature. This command can also be embedded in your program so you can turn on tracing for short sections of code if you wish.

Finally there is the OPTION command which is described in chapter 9. This takes many forms and using it you can change many settings within MMBasic including how your program will be run and much more.

Arrays

Arrays are something which you will probably not think of as useful at first glance but when you do need to use them you will find them very handy indeed.

An array is best thought of as a row of letterboxes for a block of units or condos as shown on the right. The letterboxes are all located at the same address and each box represents a unit or condo at that address. You can place a letter in the box for unit one, or unit two, etc.

Similarly an array in BASIC is a single variable with multiple sub units (called *elements* in BASIC) which are numbered. You can place data in element one, or element two, etc.

In BASIC an array is created by the DIM command, for example:

```
DIM numarr(300)
```

This created an array with the name of `numarr` and containing 301 elements (think of them as letterboxes) ranging from 0 to 300. By default an array will start from zero so this is why there is an extra element making the total 301. To specify a specific element in the array (ie, a specific letterbox) you use an index which is simply the number of the array element that you wish to access. For example, if you want to set element number 100 in this array to (say) the number 876, you would do it this way:

```
numarr(100) = 876
```

Normally the index to an array is not a constant number as shown here but a variable which can be changed to access different array elements.

As an example of how you might use an array, consider the case where you would like to record the temperature for each day of the year and, at the end of the year, calculate the overall average. You could use ordinary variables to record the temperature for each day but you would need 365 of them and that would make your program unwieldy indeed. Instead, you could define an array to hold the values like this:

```
DIM days(365)
```

Every day you would need to save the temperature in the correct location in the array. If the number of the day in the year was held in the variable `doy` and the maximum temperature was held in the variable `maxtemp` you would save the reading like this:

```
days(doy) = maxtemp
```

At the end of the year it would be simple to calculate the average for the year:

```
total = 0
FOR i = 1 to 365
  total = total + days(i)
NEXT i
PRINT "Average is:"  total / 365
```

This is much easier than adding up and averaging 365 individual variables.

The above array was single dimensioned but you can have multiple dimensions. Reverting to our analogy of letterboxes, an array with two dimensions could be thought of as a block of flats with multiple floors. A block could have a row of four letter boxes for level one, another row of four boxes for level two, and so on. To place a letter in a letterbox you need to specify the floor number and the unit number on that floor.

In BASIC the array element is specified using two indices separated by a comma. For example:

```
LetterBox(floor, unit)
```

As a practical example, assume that you needed to record the maximum temperature for each day over five years. To do this you could dimension the array as follows:

```
DIM days(365, 5)
```

The first index is the day in the year and the second is a number representing the year. If you wanted to set day 100 in year 3 to 24 degrees you would do it like this:

```
days(100, 3) = 24
```

In MMBasic for the Colour Maximite 2 you can have up to five dimensions (this is different from other versions of MMBasic which support eight dimensions). The maximum size of an array is only limited by the amount of free RAM that is available in the Colour Maximite 2 (and that is a lot).

Integers

So far all the numbers and variables that we have been using have been floating point. As explained before, floating point is handy because it will track digits after the decimal point and when you use division it will return a sensible result. So, if you just want to get things done and are not concerned with the details you should stick to floating point.

However, the limitation of floating point is that it stores numbers as an approximation with an accuracy of 14 digits. Most times this characteristic of floating point numbers is not a problem but there are some cases where you need to accurately store large numbers.

As an example, let us say that you want to manipulate time accurately down to the millisecond so that you can compare two different date/times to work out which one is earlier. The easy way to do this is to convert the date/time to the number of milliseconds since some date (say 1st Jan in year zero) - then finding the earliest of the two is just a matter of using an arithmetic compare in an IF statement.

The problem is that the number of milliseconds since that date will exceed the accuracy range of floating point variables and this is where integer variables come in. An integer variable in MMBasic running on the Colour Maximite 2 can accurately hold very large numbers to over nine million million million (or ±9223372036854775807 to be precise).

The downside of using an integer is that it cannot store fractions (ie, numbers after the decimal point). Any calculation that produces a fractional result will be rounded up or down to the nearest whole number when assigned to an integer.

It is easy to create an integer variable, just add the percent symbol (%) as a suffix to a variable name. For example, `sec%` is an integer variable. Within a program you can mix integers and floating point and MMBasic will make the necessary conversions but if you want to maintain the full accuracy of integers you should avoid mixing the two.

Just like floating point you can have arrays of integers with up to five dimensions, all you need to do is add the percent character as a suffix to the array name. For example: `days%(365, 5)`.

Beginners often get confused as to when they should use floating point or integers and the answer is simple… always use floating point unless you need a very high level of accuracy in the resulting number. This does not happen often but when you need them you will find that integers are a lifesaver.

Strings

Strings are another variable type (like floating point and integers). Strings are used to hold a sequence of characters. For example, in the command:

```
PRINT "Hello"
```

The string "Hello" is a string constant. Note that a constant is something that does not change (as against a variable, which can) and that string constants are always surrounded by double quotes.

String variables names use the dollar symbol ($) as a suffix to identify them as a string instead of a normal floating point variable and you can use ordinary assignment to set their value. The following are examples (note that the second example uses an array of strings):

```
Car$ = "Holden"
Country$(12) = "India"
Name$ = "Fred"
```

You can also join strings using the plus operator:

```
Word1$ = "Hello"
Word2$ = "World"
Greeting$ = Word1$ + " " + Word2$
```

In which case the value of `Greeting$` will be "`Hello World`".

Strings can also be compared using operators such as = (equals), <> (not equals), < (less than), etc. For example:

```
IF Car$ = "Holden" THEN PRINT "Was an Aussie made car"
```

The comparison is made using the full ASCII character set so a space will come before a printable character. Also the comparison is case sensitive so 'holden' will not equal "Holden". Using the function UCASE() to convert the string to upper case you can have a case insensitive comparison. For example:

```
IF UCASE$(Car$) = "HOLDEN" THEN PRINT "Was an Aussie made car"
```

You can have arrays of strings but you need to be careful when you declare them as you can rapidly run out of RAM (general memory used for storing variables, etc). This is because MMBasic will by default allocate 255 bytes of RAM for each element of the array. For example, a string array with 100 elements will by default use 25K of RAM. To alleviate this you can use the LENGTH qualifier to limit the maximum size of each element. For instance, if you know that the maximum length of any string that will be stored in the array will be less than 20 characters you can use the following declaration to allocate just 20 bytes for each element:

```
DIM MyArray$(100) LENGTH 20
```

The resultant array will only use 2K of RAM.

Manipulating Strings

String handling is one of MMBasic's strengths and using a few simple functions you can pull apart and generally manipulate strings. The basic string functions are:

LEFT$(string$, nbr)	Returns a substring of *string$* with *nbr* of characters from the left (beginning) of the string.
RIGHT$(string$, nbr)	Same as the above but return *nbr* of characters from the right (end) of the string.
MID$(string$, pos, nbr)	Returns a substring of *string$* with *nbr* of characters starting from the character *pos* in the string (ie, the middle of the string).

For example if S$ = "This is a string"

Then: `R$ = LEFT$(S$, 7)`
would result in the value of R$ being set to: "This is"
and: `R$ = RIGHT$(S$, 8)`
would result in the value of R$ being set to: "a string"
finally: `R$ = MID$(S$, 6, 2)`
would result in the value of R$ being set to: "is"

Note that in MID$() the first character position in a string is number 1, the second is number 2 and so on. So, counting the first character as one, the sixth position is the start of the word "is".

Another useful function is:

INSTR(string$, pattern$)	Returns a number representing the position at which *pattern$* occurs in *string$*.

This can be used to search for a string inside another string. The number returned is the position of the substring inside the main string. Like with MID$() the start of the string is position 1.

For example if S$ = "This is a string"

Then: `pos = INSTR(S$, " ")`
would result in `pos` being set to the position of the first space in S$ (ie, 5).

INSTR() can be combined with other functions so this would return the first **word** in S$:

```
R$ = LEFT$(S$, INSTR(S$, " ") - 1)
```

There is also an extended version of INSTR():

INSTR(pos, string$, pattern$)	Returns a number representing the position at which *pattern$* occurs in *string$* when starting the search at the character position *pos*.

So we can find the second word in S$ using the following:

```
pos = INSTR(S$, " ")
R$ = LEFT$(S$, INSTR(pos + 1, S$, " ") - 1)
```

This last example is rather complicated so it might be worth working through it in detail so that you can understand how it works.

Note that INSTR() will return the number zero if the sub string is not found and that any string function will throw an error (and halt the program) if that is used as a character position. So, in a practical program you would first check for zero being returned by INSTR() before using that value.

Scientific Notation

Before we finish discussing data types we need to cover off the subject of floating point numbers and scientific notation.

Most numbers can be written normally, for example 11 or 24.5, but very large or small numbers are more difficult. For example, it has been estimated that the number of grains of sand on planet Earth is 7500000000000000000. The problem with this number is that you can easily lose track of how many zeros there are in the number and consequently it is difficult to compare this with a similar sized number.

A scientist would write this number as 7.5×10^{18} which is called scientific notation and is much easier to comprehend.

MMBasic will automatically shift to scientific notation when dealing with very large or small floating point numbers. For example, if the above number was stored in a floating point variable the PRINT command would display the number as 7.5E+18 (this is BASIC's way of representing 7.5×10^{18}). As another example, the number 0.0000000456 would display as 4.56E-8 which is the same as 4.56×10^{-8}.

You can also use scientific notation when entering constant numbers in MMBasic. For example:

```
SandGrains = 7.5E+18
```

MMBasic only uses scientific notation for displaying floating point numbers (not integers). For instance, if you assigned the number of grains of sand to an integer variable it would print out as a normal number (with lots of zeros).

DIM Command

We have used the DIM command before for defining arrays but it can also be used to create ordinary variables. For example, you can simultaneously create four string variables like this:

```
DIM STRING Car, Name, Street, City
```

Note that because these variables have been defined as strings using the DIM command we do not need the $ suffix, the definition alone is enough for MMBasic to identify their type. When you use these variables in an expression you also do not need the type suffix: Eg:

```
City = "Sydney"
```

You can also use the keyword INTEGER to define a number of integer variables and FLOAT to do the same for floating point variables. This type of notation can also be used to define arrays. For example:

```
DIM INTEGER seconds(200)
```

Another method of defining the variables type is to use the keyword AS.

For example:

```
DIM Car AS STRING, Name AS STRING, Street AS STRING
```

This is the method used by Microsoft (MMBasic tries to maintain Microsoft compatibility) and it is useful if the variables have different types. For example:

```
DIM Car AS STRING, Age AS INTEGER, Value AS FLOAT
```

You can use any of these methods of defining a variable's type, they all act the same.

The advantage of defining variables using the DIM command is that they are clearly defined (preferably at the start of the program) and their type (float, integer or string) is not subject to misinterpretation. You can strengthen this by using the following commands at the very top of your program:

```
OPTION EXPLICIT
OPTION DEFAULT NONE
```

The first specifies to MMBasic that all variables must be explicitly defined using DIM before they can be used. The second specifies that the type of all variables must be specified when they are created.

Why are these two commands important?

They can help you to avoid a common programming error which is where you accidently misspell a variable's name. For example, your program might have the current temperature saved in a variable called `Temp` but at one point you accidently misspell it as `Tmp`. This will cause MMBasic to automatically create a variable called `Tmp` and set its value to zero.

This is obviously not what you want and it will introduce a subtle error which could be hard to find – even if you were aware that something was not right. On the other hand, if you used the OPTION EXPLICIT command at the start of your program MMBasic would refuse to automatically create the variable and instead would display an error thereby saving you from a probable headache.

The command OPTION DEFAULT NONE further helps because it tells MMBasic that the programmer must specifically specify the type of every variable when they are declared. It is easy to forget to specify the type and allowing MMBasic to automatically assume the type can lead to unexpected consequences.

For small, quick and dirty programs, it is fine to allow MMBasic to automatically create variables but in larger programs you should always disable this feature with OPTION EXPLICIT and strengthen it with OPTION DEFAULT NONE.

When a variable is created it is set to zero for float and integers and an empty string (ie, contains no characters) for a string variable. You can set its initial value to something else when it is created using DIM. For example:

```
DIM FLOAT nbr = 12.56
DIM STRING Car = "Ford", City = "Perth"
```

You can also initialise arrays by placing the initialising values inside brackets like this:

```
DIM s$(2) = ("zero", "one", "two")
```

Note that because arrays start from zero by default this array actually has three elements with the index numbers of 0, 1 and 2. This is why we needed three string constants to initialise it.

Constants

A common requirement in programming is to define an identifier that represents a value without the risk of the value being accidently changed - which can happen if variables were used for this purpose. These are called constants and they can represent I/O pin numbers, signal limits, mathematical constants and so on.

You can create a constant using the CONST command. This defines an identifier that acts like a variable but is set to a value that cannot be changed.

For example:

```
' speed of light (m/s)
CONST c = 299792458

' Gravitational constant
CONST G = 6.674E-11
```

These constants can then be used in a program where they make more sense to the casual reader than simple numbers.

It is good programming practice to use constants for any fixed number that represents an important value. Normally they are defined at the start of a program where they are easy to see and conveniently located for another programmer to adjust (if necessary).

Subroutines

A subroutine is a block of programming code which is self contained (like a module) and can be called from anywhere within your program. To your program it looks like a built in MMBasic command and can be used the same. For example, assume that you need a command that would signal an error by printing a message on the console. You could define the subroutine like this:

```
SUB ErrMsg
  PRINT "Error detected"
END SUB
```

With this subroutine embedded in your program all you have to do is use the command ErrMsg whenever you want to display the message. For example:

```
IF A < B THEN ErrMsg
```

The definition of a subroutine can be anywhere in the program but typically it is at the end. If MMBasic runs into the definition while running your program it will simply skip over it.

The above example is fine enough but it would be better if a more useful message could be displayed, one that could be customised every time the subroutine was called. This can be done by passing a string to the subroutine as an argument (sometimes called a parameter).

In this case the definition of the subroutine would look like this:

```
SUB ErrMsg  Msg$
  PRINT "Error: " + Msg$
END SUB
```

Then when you call the subroutine, you can supply the string to be printed on the command line of the subroutine. For example:

```
ErrMsg "Number too small"
```

When the subroutine is called like this the message "`Error: Number too small`" will be printed on the console. Inside the subroutine `Msg$` will have the value of "Number too small" when called like this and it will be concatenated in the PRINT statement to make the full error message.

A subroutine can have any number of arguments which can be float, integer or string with each argument separated by a comma. Within the subroutine the arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be hidden within the subroutine and be different from arguments defined for the subroutine.

The type of the argument to be supplied can be specified with a type suffix (ie, $, % or ! for string, integer and float). For example, in the following the first argument must be a string and the second an integer:

```
SUB MySub Msg$, Nbr%
  …
END SUB
```

MMBasic will convert the supplied values if it can, so if your program supplied a floating point value as the second argument MMBasic will convert it to an integer. If MMBasic cannot convert the value it will display an error. For example, if you supplied a string for the second argument your program will stop with an error.

You do not have to use the type suffixes, you can instead define the type of the arguments using the AS keyword similar to the way it is used in the DIM command. For example, the following is identical to the above example:

```
SUB MySub Msg AS STRING, Nbr AS INTEGER
  …
END SUB
```

Of course, if you used only one variable type throughout the program and used OPTION DEFAULT to set that type you could ignore the question of variable types completely.

Within a subroutine you can use most features of MMBasic including calling other subroutines, IF…THEN commands, FOR…NEXT loops and so on. However one thing that you cannot do is jump out of a subroutine using GOTO (if you do the result will be undefined). Normally the subroutine will exit when the END SUB command is reached but you can also terminate the subroutine early by using the EXIT SUB command.

Functions

Functions are similar to subroutines with the main difference being that a function is used to return a value in an expression. For example, if you wanted a function to convert a temperature from degrees Celsius to Fahrenheit you could define:

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

Then you could use it in an expression:

```
Input "Enter a temperature in Celsius: ", t
PRINT "That is the same as" Fahrenheit(t) "F"
```

Or as another example:

```
IF Fahrenheit(temp) <= 32 THEN PRINT "Freezing"
```

You could also define the reverse:

```
FUNCTION Celsius(F)
  Celsius = (F - 32) * 0.5556
END FUNCTION
```

As you can see, the function name is used as an ordinary local variable inside the subroutine. It is only when the function returns that the value is made available to the expression that called it.

The rules for the argument list in a function are similar to subroutines. The only difference is that parentheses are always required around the argument list when you are calling a function, even if there are no arguments (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 type suffix (ie, $, a % or a !) the function will return that type (string, integer or float), otherwise it will return whatever the OPTION DEFAULT is set to. For example, the following function will return a string:

```
FUNCTION LVal$(nbr)
  IF nbr = 0 THEN LVal$ = "False" ELSE LVal$ = "True"
END FUNCTION
```

You can explicitly specify the type of the function by using the AS keyword and then you do not need to use a type suffix (similar to defining a variable using DIM).

This is an example:

```
FUNCTION LVal(nbr) AS STRING
  IF nbr = 0 THEN LVal = "False" ELSE LVal = "True"
END FUNCTION
```

In this case the type returned by the function `LVal` will be a string.

As for subroutines you can use most features of MMBasic within functions. This includes FOR…NEXT loops, calling other functions and subroutines, etc. Also, the function will return to the expression that called it when the END FUNCTION command is reached but you can also return early by using the EXIT FUNCTION command.

Local Variables

Variables that are created using DIM or that are just automatically created are called *global* variables. This means that they can be seen and used anywhere in the program including within subroutines and functions. However, inside a subroutine or function you will often need to use variables for various tasks that are internal to the subroutine/function. In portable code you do not want the name you chose for such a variable to clash with a global variable of the same name. To this end you can define a variable using the LOCAL command within the subroutine/function.

The syntax for LOCAL is identical to the DIM command, this means that the variable can be an array, you can set the type of the variable and you can initialise it to some value.

For example, this is our ErrMsg subroutine but this time it has been extended to use a local variable for joining the error message strings.

```
SUB ErrMsg  Msg$
  LOCAL STRING tstr
  tstr = "Error: " + Msg$
  PRINT tstr
END SUB
```

The variable `tstr` is declared as LOCAL within the subroutine, which means that (like the argument list) it will only exist within the subroutine and will vanish when the subroutine exits. You can have a global variable called `tstr` in your main program and it will be different from the variable `tstr` in the subroutine (in this case the global `tstr` will be hidden within the subroutine).

You should always use local variables for operations within your subroutine or function because they help make the module much more self contained and portable.

Static Variables

LOCAL variables are reset to their initial values (normally zero or an empty string) every time the subroutine or function starts, however there are times when you would like the variable to retain its value between calls. This type of variable is defined with the STATIC command.

We can demonstrate how STATIC variables are useful by extending the ErrMsg subroutine to prevent duplicated calls to the subroutine repeatedly displaying the same message. For example, our program might call this subroutine from multiple places but if the message is the same in a number of subsequent calls we would like to see the message just once.

To keep track of the last message displayed we use a static variable called `lastmsg`. This will hold the text of the last message and we can compare it to the current message text to determine if it is different and therefore should be printed.

This is our new subroutine:

```
SUB ErrMsg  Msg$
  STATIC STRING lastmsg
  LOCAL STRING tstr
  IF Msg$ <> lastmsg THEN
    tstr = "Error: " + Msg$
    PRINT tstr
    lastmsg = Msg$
  ENDIF
END SUB
```

This would give just one message every time a call is made with a duplicate message text.

The STATIC command uses exactly the same syntax as DIM. This means that you can define different types of static variables including arrays and you can also initialise them to some value.

The static variable is created on the first time the STATIC command is encountered and it is automatically set to zero (if a float or integer) or an empty string. On subsequent calls to the subroutine or function MMBasic will recognise that the variable has already been created and it will leave its value untouched (ie, whatever it was in the previous call). As with DIM you can also initialise a static variable to some value. For example:

```
STATIC INTEGER var = 123
```

On the first call (when the variable is created) it will be initialised to 123 but on subsequent calls it will keep whatever its value was previously set to.

Mostly static variables are used to keep track of the *state* of something while inside a subroutine or function. A *state* is a record of something that has happened previously. Examples include:

- Has the COM port already been opened?
- What steps in a sequence have we completed?
- What text has already been displayed?

Normally you will use global variables (created using DIM) to track a *state* but sometimes you want this to be contained within a module and this is where static variables are valuable. Just like LOCAL the use of STATIC helps to make your subroutines and functions more self contained and portable.

Calculate Days

We have covered a lot of programming commands and techniques so far in this tutorial and, to give an example of how they work together, the following is an example program that will calculate the number of days between two dates.

```
' Example program to calculate the number of days between two dates

OPTION EXPLICIT
OPTION DEFAULT NONE

DIM STRING s
DIM FLOAT d1, d2

DO
  PRINT : PRINT "Enter the date as  dd mmm yyyy"
  PRINT " First date";
  INPUT s
  d1 = GetDays(s)
  IF d1 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Second date";
  INPUT s
  d2 = GetDays(s)
  IF d2 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Difference is" ABS(d2 - d1) " days"
LOOP

' Calculate the number of days since 1/1/1900
FUNCTION GetDays(d$) AS FLOAT
  LOCAL STRING Month(11) =
("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov","dec")
  LOCAL FLOAT Days(11) = (0,31,59,90,120,151,181,212,243,273,304,334)
  LOCAL FLOAT day, mth, yr, s1, s2

  ' Find the separating space character within a date
  s1 = INSTR(d$, " ")
  IF s1 = 0 THEN EXIT FUNCTION
  s2 = INSTR(s1 + 1, d$, " ")
  IF s2 = 0 THEN EXIT FUNCTION

  ' Get the day, month and year as numbers
  day = VAL(MID$(d$, 1, s2 - 1)) - 1
  IF day < 0 OR day > 30 THEN EXIT FUNCTION
  FOR mth = 0 TO 11
    IF LCASE$(MID$(d$, s1 + 1, 3)) = Month(mth) THEN EXIT FOR
  NEXT mth
  IF mth > 11 THEN EXIT FUNCTION
  yr = VAL(MID$(d$, s2 + 1)) - 1900
  IF yr < 1 OR yr >= 200 THEN EXIT FUNCTION

  ' Calculate the number of days including adjustment for leap years
  GetDays = (yr * 365) + FIX((yr - 1) / 4)
  IF yr MOD 4 = 0 AND mth >= 2 THEN GetDays = GetDays + 1
  GetDays = GetDays + Days(mth) + day
END FUNCTION
```

Note that the line starting `LOCAL STRING Month(11)` has been wrapped around because of the limited page width – it is one line as follows:

```
LOCAL STRING Month(11) = ("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov","dec")
```

This program works by getting two dates from the user at the console and then converting them to integers representing the number of days since 1900. With these two numbers a simple subtraction will give the number of days between them.

When this program is run it will ask for the two dates to be entered and you need to use the form of `dd mmm yyyy`. This screen capture shows what the running program will look like.

The main feature of the program is the user defined function `GetDays()` which takes a string entered at the console, splits it into its day, month and year components then calculates the number of days since 1st January 1900. This function is called twice, once for the first date and then again for the second date. It is then just a matter of subtracting one date (in days) from the other to get the difference in days.

We will not go into the detail of how the calculations are made (ie, handling leap years) as that can be left as an exercise for the reader. However it is appropriate to point out some features of MMBasic that are used by the program.

It demonstrates how local variables can be used and how they can be initialised. In the function `GetDays()` two arrays are declared and initialised at the same time. These are just a convenient method of looking up the names of the months and the cumulative number of days for each month. Later in the function (the FOR loop) you can see how they make dealing with twelve different months quite efficient.

Another feature highlighted by this program is the string handling features of MMBasic. The INSTR() function is used to locate the two space characters in the date string and then later the MID$() function uses these to extract the day, month and year components of the date. The VAL() function is used to turn a string of digits (like the year) into a number that can be stored in a numeric variable.

Note that the value of a function is initialised to zero every time the function is run and unless it is set to some value it will return a zero value. This makes error handling easy because we can just exit the function if an error is discovered. It is then the responsibility of the calling program code to check for a return value of zero which signifies an error.

This program illustrates one of the benefits of using subroutines and functions which is that when written and fully tested they can be treated as a trusted "black box" that does not have to be opened. For this reason functions like this should be the first component written and they should be properly tested before you go on to writing the rest of the program.

There are a few features of this program that we have not covered before. The first is the MOD operator which will calculate the remainder of dividing one number into another. For example, if you divided 4 into 15 you would have a remainder of 3 which is exactly what the expression

15 MOD 4 will return. The ABS() function is also new and will return its argument as a positive number (eg, ABS(-15) will return +15 as will ABS(15)).

The EXIT FOR command will exit a FOR loop even though it has not reached the end of its looping, EXIT FUNCTION will immediately exit a function even though execution has not reached the end of the function and CONTINUE DO will immediately cause the program to jump to the end of a DO loop and execute it again.

Why would this program be useful? Well some people like to count their age in days, that way every day is a birthday! You can calculate your age in days, just enter the date that you were born and today's date. That is not particularly useful but the program itself is valuable as it demonstrates many of the characteristics of programming in MMBasic. So, pull out the *Colour Maximite 2 User Manual* and work your way through the program code – it should be a rewarding experience.

Good Programming Habits

Before we finish with the subject of BASIC programming it will be worth while providing some hints on how to write programs that are easy to understand and maintain. This can be more important than one might think. A poorly written program is more likely to contain bugs and people will be reluctant to try and fix such a program because the logic is hard to understand.

You might think that this does not matter because you will be the only person to ever read the program. But your memory will fade over time and when you try to modify a program that you wrote (say) a year ago it will be the same as if it was written by a complete stranger who you will hope had followed these hints.

1. Use lots of comments. They are the first thing that people (including you) will read when they pick up your program and they are invaluable in rendering the program and its logic understandable. Even if no one else will read the program you will appreciate the comments in the future.
2. Use indenting to illustrate the logic of loops, multiline IF statements, subroutines, etc. Without indenting the casual reader would have to search many lines to determine when a block of code has terminated.
3. Keep the comments and indenting up to date. When modifying a program it is easy to forget that these features also need updating and nothing is worse than a misleading comment or indentation.
4. Define all variables using DIM or LOCAL statements at the start of the program, subroutine or function. Do not let variables be automatically created, instead use OPTION EXPLICIT and OPTION DEFAULT NONE.
5. Use variable names that make sense. For a simple loop you can use a short variable like 'spd' but for something important that is scattered throughout the program use a descriptive variable name such as 'MaxSpeedLimit'.
6. Define significant numbers as constants at the start of the program using the CONST command.
7. MMBasic does not worry about upper or lower case characters in identifiers (variable names, subroutine names, etc) but regardless, you should use a consistent case. For example, you can use either MaxSpeedLimit or maxspeedlimit in a program, but you should not use both.

8. Package unique and self contained pieces of code into a subroutine or function. These have limited entry/exit points which means that someone can read through such a module and more easily satisfy themselves that it is working correctly. From then on it can be treated as a trusted portion of code.
9. Don't use the GOTO command unless you absolutely have to. Features such as multiline IF statements, subroutines and functions are much easier to understand than a program which uses GOTOs to jump around.
10. Don't be obsessed with optimising your code to make it faster for MMBasic to interpret. MMBasic makes many optimisations of its own and anything that you do will have little effect on speed and may obscure the logic of the program.

For a short program you can ignore many of these hints but for a large program they can be a huge help and may help prevent your hair turning prematurely grey if you need to modify your program in the future.

Chapter 5

SD Card and Files

THe Colour Maximite 2 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 saved on the card.

.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. Small capacity cards may not be reliable so the recommended size is 8GB formatted as FAT32.

Command Summary

There are 33 commands and functions related to the SD card and they are summarised here. For the full description of each command refer to the *Colour Maximite 2 User Manual.*

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, RUN "MYPROG.BAS").
- 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.

Program Management

All programs reside on the SD card and so it must be present when running, editing and listing programs. This is different from the original Maximite where the SD card was not necessarily required. The program management commands are:

- RUN fname$
  Run a program.

- 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$' is the file name in 8.3 format. 'mode' can be INPUT, OUTPUT, APPEND or RANDOM. '#fnbr' is the file number (1 to 10).
- PRINT #fnbr, expression [[,;]expression] … etc
  Outputs text to the file opened as #fnbr.
- INPUT #fnbr, list of variables
  Read a list of comma separated data into the variables specified from the file previously opened as #fnbr.
- SEEK #fnbr, pos
  Will position the read/write pointer in a file that has been opened for RANDOM access to the 'pos' byte.
- LINE INPUT #fnbr, variable$
  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$(nbr, #fnbr)
  Will return a string composed characters read from a file previously opened for INPUT
- DIR$(fspec, type)
  Will search an SD card for files and return the names of entries found.
- 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$’.

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.

To write data you first need to open the file and the command to do that is:

```
OPEN filename$ FOR mode AS #nbr
```

filename$ is the name of the file that you want to write to and it can be a string variable or a string constant (eg, "mydat.txt"). *mode* can be OUTPUT, APPEND or RANDOM. The first is the normal method of opening a file and will create the file on the SD card overwriting a previous version with

the same name. APPEND will automatically add any new data to the end of a file (that already exists) and RANDOM allows the programmer to jump around within the file to read/modify data even in the middle of the file. Finally *#nbr* is a number in the range of 1 to 10 and acts as an identifier for the open file.

For example we might use the command:

```
OPEN "datafile.txt" FOR OUTPUT AS #6
```

Once you have the file open you can use the PRINT command to write to it. This command has been covered previously but what has not been mentioned is that you can use it with a file identifier to send the data to a file instead of the console. For example:

```
PRINT #6, "This text is written to the file"
```

#6 is the identifier and must refer to a file opened for writing (by the way, the # character is optional but it helps the programmer remember that this is a file identifier). Note that the data is written as a complete line with a terminating carriage return and line feed characters (this is how the PRINT command works).

When you have finished writing to the file you should close it:

```
CLOSE #6
```

This step is important as it instructs MMBasic to flush any buffered data and update the file information on the SD card. If you forget to close the file you will end up with a corrupted file or even worse, a corrupted SD card.

After the close you could transfer the card to a PC and open the file using a program like Notepad to see what was written. To use MMBasic to read the data that you have just written you must first use the OPEN command again but this time to open the file for input:

```
OPEN "datafile.txt" FOR INPUT AS #4
```

You can use a different file identifier (as we have above) and any number between 1 and 10 will do (so long as it is not already in use).

With the file open there are three ways to read from it:

- INPUT — Read a list of comma separated values
- LINE INPUT — Read a complete line
- INPUT$() — A function that will read a specified number of bytes

When you want to read a whole line the LINE INPUT command works the best and in this case the program line would be:

```
LINE INPUT #4, s$
```

This will read a line from the file and save the data into the variable `s$`. You could test this by using the PRINT command to display the value of `s$` on the console. Finally, don't forget to close the file:

```
CLOSE #4
```

Sequential Access Example

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".

Saving Numeric Data to an SD Card

Numeric data can be recorded on an SD card as binary numbers but it makes more sense to save data as ASCII text characters. The advantage of this is that you can always pop the card out and read the file on a PC which will show you exactly what was recorded.

Probably the best format is comma separated variables (CSV) as this is easy to read and, if the file extension is changed to .xls, can be directly loaded on a PC as an Excel spreadsheet.

As an example of recording data in the CSV format we will assume that every ten seconds you need to record the temperature from three different DS18B20 sensors connected to pins 19, 21 and 23 on the external I/O connector. Later we want to read back the readings and calculate the average reading from each sensor. When the program is run the user is first prompted for the number of measurements to be made.

To measure the temperature we can use the TEMPR() function which works with the DS18B20 sensors and is built into MMBasic.

A typical program to do this would be:

```
INPUT "Enter the number of measurements: ", nbr
FOR count = 1 to nbr
  OPEN "datafile.txt" FOR APPEND AS #6
  PRINT #6, TEMPR(19) "," TEMPR(21) "," TEMPR(23)
  CLOSE #6
  PAUSE 9400
NEXT count
```

The data output is contained within a FOR…NEXT loop which will keep count of the records written. Within the loop we open the file, write the data and then immediately close the file. This opening and closing is done so that if the loop is interrupted during the PAUSE command (perhaps

by a power failure) the file will not be corrupted because it was closed at the time of the interruption.

The file is opened for APPEND which means that we are always adding to the end of the file, not creating a new file. The first time the file is opened and the file does not already exist MMBasic will create it for us (the same as with opening the file for OUTPUT).

Each time the TEMPR() function makes a reading there will a delay of about 200ms so reading all three temperatures will take 600ms. The PAUSE command then causes the program to pause for a further 9400ms which means that the loop will repeat every 10000ms or once every ten seconds. This timing could be made more accurate but that is a subject for another day.

Using the PRINT command the temperatures are recorded in the file as three numbers separated by commas and then terminated by carriage return and line feed characters. If you let the program run then placed the SD card into a PC you could open the file using a text editor and you should see something like this:

```
25.8, 18.9, 23.3
25.6, 18.8, 23.1
25.4, 18.7, 23
25.3, 18.7, 23.1
25.1, 18.4. 23.4
...
```

As mentioned earlier, if you renamed the file as *datafile.xls* you would also be able to load the file into Excel as a spreadsheet and manipulate or graph the data.

Now that the data has been recorded we need to read it back and calculate the average temperatures.

The MMBasic program to do this would look something like this:

```
DIM FLOAT count, a, b, c, ta, tb, tc
OPEN "datafile.txt" FOR INPUT AS #5
DO WHILE NOT EOF(#5)
  INPUT #5, a, b, c
  ta = ta + a : tb = tb + b : tc = tc + c
  count = count + 1
LOOP
CLOSE #5
PRINT ta/count, tb/count, tc/count
```

First we define some variables; `count` will be the number of records in the file, `a,b,c` will hold the numbers read from the file and `ta,tb,tc` will hold the accumulated sum of all temperatures. Note that all variables are automatically set to zero when they are created and this means that we do not need to explicitly set the variables accumulating the temperatures to zero before using them.

We then open the file for reading and enter a loop which will continue while there is data to be read. This uses the EOF() function which will return true if the read position is at the end of the file (ie, we have read all the data). Note that this construction will also act correctly if the file holds no records (ie, is zero length). In that case the function EOF() will immediately return true and the loop will terminate without trying to read any records.

The program then reads the three comma separated numbers using the INPUT command. This works the same as using the INPUT command to get entries from the console (as covered in Chapter 3). The command will read the first number up to the separating comma and store the value in the first variable (which is `a`). It will then read the next number and save it in `b`, and so on.

The program then adds these numbers to the three variables holding the grand total of all records (these were set to zero when created). Also at this time the variable `count` is incremented to keep track of the number of records read.

When all the lines in the file have been read the loop will terminate and the file closed. Calculating the averages is then simply a case of dividing the total accumulated temperatures by the number of records read.

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 (64bytes in this example). The semicolon at the end of the print command suppresses the addition of the carriage return and line feed characters which would make the record longer than intended.

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

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

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

```
      SEEK #1, LOF(#1) + 1
   ELSE
      INPUT "Record Number: ", nbr
      IF nbr < 1 or nbr > LOF(#1)/RecLen THEN PRINT "Invalid record" : GOTO
abort
      SEEK #1, RecLen * (nbr - 1) + 1
   ENDIF
   IF cmd$ = "r" THEN
      PRINT "The record = " INPUT$(RecLen, #1)
   ELSE
      LINE INPUT "Enter the data to be written: ", dat$
      PRINT #1,dat$ + SPACE$(RecLen - LEN(dat$));
   ENDIF
LOOP
```

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

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

Chapter 6

Graphics on the VGA Monitor

An outstanding feature of the Colour Maximite 2 is its ability to draw lines, circles, text, etc on the VGA output in many colours and in a variety of screen resolutions. You can use this for drawing graphs, writing games and many other visual applications..

The standard resolution of the VGA output on startup is 800 x 600 pixels with each pixel capable of displaying any one of 256 colours and all the examples in this chapter assume that this is the current state.

Graphic Coordinates

All operations on the VGA monitor are done in terms of pixels with their location on the screen expressed as a horizontal or X position and a vertical or Y position. All commands that specify a position on the video monitor use this X an Y coordinate scheme.

The top left corner of the screen has the coordinates of X = 0 and Y = 0 and as you move to the right the X coordinate will increase and as you move down the screen the Y coordinate will increase. Accordingly X = 799 and Y = 599 are the coordinates of the bottom right corner of the VGA monitor (at the default resolution of 800x600 pixels).

For example, the PIXEL command will set the colour of an individual pixel and it has the format:

PIXEL X-position, Y-position, colour

so:

```
PIXEL 0, 0, RGB(red)
```

Will set the top left pixel to red and the following will set the pixel in the middle of the screen to blue (we will discuss colours next):

```
PIXEL 400, 300, RGB(blue)
```

Defining Colour

All colours in MMBasic are specified as a 24-bit number (the same as your desktop PC). This can be used to specify over 64 million different colours however the VGA output cannot display this range. This need not concern you as MMBasic will automatically convert the 24-bit colour value to the colour range supported by the current setting of the VGA output.

The top eight bits of a 24-bit colour value is the intensity of the red colour, the middle eight bits the green colour and the bottom eight bits the blue colour. Each eight bit number can range from zero to 255 (decimal).

For example, yellow is produced when the red and green colours are at full intensity and blue is off. If you work out the result using binary arithmetic you will get the number 16776960. Using the PIXEL command we can change the pixel at the centre of the screen to yellow with the command:

```
PIXEL 400, 300, 16776960
```

Defining colours this way is rather clumsy so MMBasic makes it easy for you with the RGB() function. This has the form RGB(red, green, blue) where red is a number between zero and 255 and similar for green and blue. So you could rewrite the command to turn on the pixel with the yellow colour like thus:

```
PIXEL 400, 300, RGB(255, 255, 0)
```

To make it even more convenient for you to specify a colour the RGB() function will allow you to directly name the colour, so you could also turn the pixel yellow using just this:

```
PIXEL 400, 300, RGB(yellow)
```

The colours that you can specify this way are red, green, blue, yellow, cyan, magenta, brown, gray, white and black.

Finally, you can store the colour value in a variable or set it as a constant. For example:

```
CONST PixColour = RGB(yellow)
PIXEL 400, 300, PixColour
```

Many commands allow you to skip specifying the colour and in that case the current default colour will be used. This can be set with the COLOUR command which takes the format:

COLOUR foreground-colour, background-colour

For example:

```
COLOUR RGB(red), RGB(blue)
```

At startup the foreground colour defaults to white and the background to black.

By the way, for our USA cousins you can also spell the command as COLOR.

Drawing on the Screen

There are ten basic drawing commands that you can use. These are:

- `CLS` *`C`*
  Clears the screen to the colour C. If C is omitted the current background colour will be used.
- `PIXEL X, Y,` *`C`*
  Sets the colour of a pixel. If C is omitted the current foreground colour will be used.
- `LINE X1, Y1, X2, Y2,` *`LW, C`*
  Draws a line starting at the coordinates of X1 and Y1 and ending at X2 and Y2. LW is the line's width which defaults to one if not specified and C is the colour which defaults to the current foreground colour. The line width only applies to horizontal or vertical lines. Diagonal lines will always have a line width of one.
- `BOX X, Y, W, H,` *`LW, C, FILL`*
  Draws a box starting at X and Y (top left hand corner) which is W pixels wide and H pixels high. LW is the width of the sides of the box (defaults to one), C is the colour (defaults to the foreground colour) and FILL is the colour to fill the box and this defaults to -1 which means no fill (ie, the pixels inside the box are undisturbed).

- `RBOX X, Y, W, H, R, C, FILL`
  Draws a box with rounded corners starting at X and Y which is W pixels wide and H pixels high. R is the radius of the corners of the box (defaults to 10) and the remaining parameters are the same as for the BOX command.

- `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 (defaults to one). A is the aspect ratio (defaults to one which specifies a perfect circle). The remaining parameters are the same as for the BOX command.

- `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 (zero degrees is the top of the screen) 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 one, two or three letters where the first letter is the horizontal justification around X and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical placement around Y and can be T, M or B for TOP, MIDDLE, BOTTOM. FONT and SCALE specify the font and scale. C is the drawing colour and BC is the background colour.

- `GUI BITMAP X, Y, BITS, WIDTH, HEIGHT, SCALE, C, BC`
  Displays the bits in a bitmap starting at X and Y. HEIGHT and WIDTH are the dimensions of the bitmap, SCALE, C and BC are the same as for the TEXT command.

Examples

If you wanted to draw a horizontal line across the centre of the screen you could do it by repeatedly using the PIXEL command to draw the line pixel by pixel:

```
FOR i = 0 TO 799
  PIXEL i, 300, RGB(white)
NEXT i
```

However it is simpler to use the LINE command which has the form:

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

This will draw a line starting at the coordinates of X1 and Y1 and ending at X2 and Y2. LW is the line's width and C the colour. For example:

```
LINE 0, 300, 799, 300, 1, RGB(white)
```

There are other commands that make it easy to draw common graphic elements. For example, you can draw a box using the BOX command:

```
BOX 100, 120, 70, 30, 2, RGB(red)
```

This will draw a box with the top left corner positioned at X = 100 and Y = 120. The width of the box is 70 pixels and the height 30 pixels. The width of the lines used to draw the box is 2 pixels and they are drawn using the red colour.

You could, if you wished, fill the box with some colour. For example, the following will draw the same box but this time filled with blue:

```
BOX 100, 120, 70, 30, 2, RGB(red), RGB(blue)
```

The RBOX command is similar but it will draw the box with rounded corners. The following shows how to draw a box similar as the above example but with round corners:

```
BOX 100, 120, 70, 30, 10, RGB(red), RGB(blue)
```

The fifth parameter is the radius of the rounded corner and in this case it is 10 pixels. Note that you cannot define the thickness of the walls using this command so they default to a width on one pixel.

The CIRCLE command, as its name suggests, will draw a circle.

```
CIRCLE X, Y, R, LW, A, C, FILL
```

X and Y are the coordinates of the centre of the circle, R is the radius (in pixels), LW is optional and is the thickness of the line, A is the aspect ratio (which is optional), C is the colour and FILL (also optional) is the colour to fill the circle.

The aspect ratio (A in the command's parameter list) is a decimal number which can be a fraction - if it is exactly 1 the circle will be perfectly circular, if it is less or more than 1 the graphic drawn will be an oval with either the horizontal or vertical axis stretched.

For example:

```
CLS
Circle 400, 300, 45, 3, 1, RGB(red), RGB(yellow)
Circle 400, 300, 100, 1, 0.5, RGB(blue)
Circle 400, 300, 50, 1, 1.8, RGB(green)
```

This will draw a circle and two ovals. The first will be drawn in red with a border three pixels wide and filled with yellow. The next is a blue oval followed by a green oval, each oval drawn with a different aspect ratio. This photo shows the result.

Fonts

There are seven built in fonts. These are numbered from 1 to 7 and this number is used to specify to MMBasic the font to use:

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).
2	12 x 20	All 95 ASCII characters	Medium sized font
3	16 x 24	All 95 ASCII characters	A larger font that is double the size of font #1
4	16 x 24 BOLD	All 95 ASCII characters	A bold version of font #3
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 9	All 95 ASCII characters	A small font useful when low resolutions are used.

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). There are a wide range of fonts that are available including fancy fonts like a seven segment font and a symbol font (Dingbats) which is handy for creating on screen icons, etc. The *Colour Maximite 2 User Manual* goes into the detail of embedded fonts and also the LOAD FONT command which can dynamically load a font from the SD card.

The default font used by MMBasic is font #1 however that can be changed with the FONT command:

```
FONT font-number, scaling
```

Where 'font-number' is a number which can be optionally preceded by a hash (#) character. 'scaling' is optional and is a number in the range of 1 to 15. The font will be multiplied by the scaling factor making the displayed character correspondingly wider and taller. For example, specifying a 'scaling' of 2 will double the height and width. If not specified the scaling factor will be 1 (ie, no scaling). The font and scaling can also be specified in the TEXT command but setting the default using the FONT command is useful when you will be using a consistent font in the program.

TEXT Command

The TEXT command is the most useful of the graphics commands. It allows you to display text anywhere on the LCD screen using different fonts and in any colour.

This is the command and its parameters:

```
TEXT x, y, string, alignment, font, scale, colour, back-colour
```

'x' and 'y' are the coordinates (in pixels) of where the text is to be positioned and 'string' is the text (ie, string) that you want to display.

The alignment is a string consisting of none, one, two or three letters. The first can be L, C or R. These specify that the text should be drawn with its left margin on the 'x' coordinate or centred around this coordinate or with the right margin on the 'x' coordinate. The second letter is the vertical placement and can be T, M or B for the text's top to be aligned to the 'y' coordinate and so on for middle or bottom.

You can also use an optional third letter which will define the orientation of the text. (ie, vertical, inverted, etc). This is described in full detail in the *Colour Maximite 2 User Manual*.

'font' is the font number that should be used (the Colour Maximite 2 can have up to 16 fonts installed) and 'scale' is the magnification (1 is the normal font, 2 is doubled in height and width, 3 is tripled, etc).

'colour' is the colour of the text and 'back-colour' is the background colour for the text. The background colour can be set to -1 which means that it is transparent.

Most parameters are optional so, for example, you can just use the following to print the word "Colour Maximite 2" near the top left of the screen.

```
TEXT 10, 10, "Colour Maximite 2"
```

The alignment defaulted to left and top, the font defaulted to font #1, the scale to 1, the colour to white and the background to black. With the optional parameters you can either leave them off the end of the command string or just use two commas (ie, “, ,”) with nothing in-between to accept the default.

The alignment parameter is particularly useful as it allows you to position the text much easier. For example to perfectly centre the text at the default resolution video mode (800x600) you can use:

```
TEXT 400, 300, "Centred", "CM"
```

'C' (for centre) specifies that the text be centred horizontally on the X axis and 'M' (for middle) will position the middle of the text vertically around the Y axis. You could calculate the centred position for the text yourself using the font's height and width but using the alignment parameters is a lot simpler.

As another example the following will print the word "Hello" in all four corners of the screen using font 1 doubled in size as illustrated on the right.

```
TEXT 0, 0, "Hello", , 1, 2
TEXT 799, 0, "Hello", "R", 1, 2
TEXT 0, 599, "Hello", "B", 1, 2
TEXT 799, 599, "Hello", "RB", 1, 2
```

The TEXT command will only display a string, so if you want to display a number (integer or float) you most convert it to a string using the STR$() function. For example, the following will display 54.7 in the centre of the screen:

```
depth = 54.7
TEXT 160, 120, STR$(depth), "CM"
```

You can always join strings together using the plus character (+) and this is handy when you want to build a string for the TEXT command. For example:

```
depth = 54.7
TEXT 160, 120, "Depth: " + STR$(depth) + " meters", "CM”
```

Stars

To better demonstrate the graphical abilities of the Colour Maximite 2 this program will fill the screen with thousands of random points of light (pixels) representing stars. There is not too much to it (just four lines):

```
Cls
Do
  Pixel Rnd * MM.HRes, Rnd * MM.VRes
Loop
```

The program starts by clearing the screen (the CLS command) and then enters a continuous loop where it constantly turns on individual pixels at random coordinates. The colour of each pixel is the default colour which is normally white.

The random number generator in MMBasic is the RND function and it generates a random number from 0.0000 to 0.9999. `MM.HRes` is a read-only variable provided by MMBasic which returns the horizontal resolution of the screen. So, by multiplying the two we end up with a random x coordinate. Doing the same using `MM.VRes` will similarly give us a random y or vertical coordinate.

Using read-only variables like `MM.HRes` and `MM.VRes` helps to make the program more portable and understandable rather using simple numbers like 800 and 600. It also means that the program would work just as well if the screen resolution was changed by the MODE command (more on that later)

You should use the editor to enter this program and run it. If you do you will soon see that it has a couple of problems. For a start all the stars are white which is a bit boring but, even worse, the screen will soon fill up with illuminated pixels and turn into a solid white.

To overcome these deficiencies we can select a random colour for each pixel (easy) and to prevent the screen filling with colour we need to track the location of each illuminated pixel and turn it off after a delay so that the screen does not fill up (not so easy).

Twinkling Stars

The following program does exactly this and gives us a nice twinkling field of coloured stars:

```
Const nbr = 18000                          ' set the number of stars
Dim p(nbr,2)                               ' array for star's coordinates
Cls                                        ' clear the screen
Do
  For i = 1 To nbr                         ' step through every star
    Pixel p(i, 1), p(i, 2), 0              ' erase the previous star
    x = Rnd * MM.HRes : y = Rnd * MM.VRes ' get a random coordinate
    p(i, 1) = x : p(i, 2) = y              ' and save its location
    do                                     ' get a random 3 bit colour
      c = RGB(255 * (Rnd > 0.5), 255 * (Rnd > 0.5), 255 * (Rnd > 0.5))
    Loop Until c <> 0
    Pixel x, y, c                          ' turn on the new star
    Count% = Count% + 1                    ' increment the count
    Text MM.HRes/2, MM.VRes/2, Str$(Count%), "CM" ' display the count
  Next i
Loop                                       ' loop forever
```

This program starts by setting the constant *nbr* to the maximum number of pixels to be illuminated at any one time. It then defines the array *p* which will hold the x and y coordinates of every illuminated pixel. Next it enters an infinite DO…LOOP which encloses a FOR…NEXT loop which sweeps through all of the illuminated pixels. The program therefore continuously sweeps through all the pixels turning off the old pixels and turning on the new.

Within the FOR…NEXT loop we first turn off the previous pixel recorded in the array element indexed by the variable *i*. This is done by setting the pixel's colour to black. Then we calculate a new random coordinate for the replacement pixel (ie, star) and save this into the array *p*. The next time the FOR…NEXT loop sweeps through the array this is the coordinate that will be set to black

We then calculate a random colour for this new pixel. We do this by first comparing the random number generated by the RND function with the number 0.5 (ie, `(Rnd > 0.5)`). Whenever a compare is made MMBasic will return 1 for true or 0 for false. When we multiply this by 255 we get a number that is 255 half the time and zero for the other half.

We do this three times for the three colour values passed to the RGB() function with the result that we will get random fully saturated colours (ie, red, orange, yellow, green, etc). There is only one issue with this neat method and that is that we can get the random colour of black – which has a value of zero and is not a visible colour. To fix this we put the random colour generator in a loop and tell it to keep generating colours until it creates something that does not have a value of zero (ie, black).

Finally, we turn on our new pixel with its random colour, increment an integer counter (called *Count%*) and then display this counter in the middle of the screen. By using an integer counter we prevent the STR$() function from switching to exponential notation when the count reaches a large number. Note that when we display the count we place it in the centre of the screen (MM.HRes/2 and MM.VRes/2) by instructing the TEXT command to centre the text on the x and y coordinates (ie, "CM").

Sharp eyed readers will have spotted one oddity in this program and that is when the array p is first created by MMBasic all its elements will be set to zero. This means that on its first pass through

the array the program will keep setting the pixel with the coordinates of x=0 and y=0 to black. This does not matter much as it is probably black anyway and from then (on subsequent passes) the array will have real data in it to use for turning pixels off.

Video Modes

The video output on the VGA monitor can be set to a number of modes using the MODE command. This takes the form:

MODE r, bits, *bg*, *int*

Where **r** is the video resolution and **bits** is the colour depth. *bg* and *int* are optional parameters which are mostly used for writing graphical computer games. In this tutorial we will just cover the straight forward use of the MODE command - for the more sophisticated aspects you should refer to the *Colour Maximite 2 User Manual.*

The screen resolution (**r**) is a number from 1 to 5 as follows:

1 = 800 x 600 pixels (default)
2 = 640 x 400 pixels
3 = 320 x 200 pixels
4 = 480 x 432 pixels
5 = 240 x 216 pixels

The Colour Maximite 2 starts up in the 800x600 pixel resolution which is best for everyday tasks while the other resolutions are useful for graphical intensive programs (like games) with a lot of motion and the need to update the screen rapidly. The lower the resolution the easier it is as there are less pixels to redraw.

Modes 4 and 5 match the resolutions possible on the original Colour Maximite and are included to make it easier to port programs from that computer.

The number of colours available (also called the colour depth) in the selected screen resolution is specified by the second argument (**bits**) to the MODE command and there are three choices:

8 = 8-bit colour value with 256 colours (default).
12 = 12-bit colour value with 4096 colours.
16 = 16-bit colour value with 65536 colours.

Remember that in MMBasic all colours are specified as 24-bit values and MMBasic will make the conversion to whatever colour depth you specify. Amongst other things this means that you do not need to change any colour values when you change the colour depth.

The 8-bit mode (default on power up) supports 256 colours and you can let MMBasic decide these based on the 24-bit value that you provide to the various drawing commands or you can select exactly what each 8-bit colour should display as by using a 16-bit pallet and the MAP command.

The 12 and 16-bit modes allow you to select transparent colours. In addition all resolutions and colour modes allow you to have multiple video pages which can be used for building images off screen allowing for sophisticated animated graphical displays (see below).

Most users do not need this level of sophistication so, if you are not interested in the details, it is best to leave the video output in its default at power up (ie, MODE 1, 8).

Displaying Images

Using the LOAD command the Colour Maximite 2 can load an image from the SD card and display it on the VGA monitor. Supported formats are BMP, GIF, JPG and PNG. The image can be

positioned anywhere on the screen and this feature is often used to display a detailed graphical background leaving the main BASIC program to do the simpler task of updating the details.

There are some limitations on the format of the images and these are detailed in the user 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).

Often the colour depth of the VGA output will need to be changed via the MODE command to allow for more colours than the default 8-bit (256 colour) setting. For JPG images MODE 2,16 (640 x 400 pixels with 65536 colours) is the best as MMBasic can then use the hardware JPG decoder in the STM32 chip in its most efficient mode. Note that JPG images must fit entirely on the screen otherwise an error will be generated.

Video Pages

Depending on the selected resolution and colour depth you have up to 24 pages of video memory available (see the MODE command in the user manual). You can also use the read only variable MM.INFO(MAX PAGES) to find out how many pages that you have in the current mode. For example: `PRINT MM.INFO(MAX PAGES)`

For the 8 and 16-bit colour modes page zero is the page that is displayed on the VGA monitor while the other pages can be used to assemble screen images for high speed copying to page zero when they are ready for display.

The 12-bit colour mode is different in that it will simultaneously display two pages plus a background colour. You can think of it as three layers stacked one on top of each other. At the bottom is a solid background colour and above this is page 0 and above that is page 1. Graphics drawn on the higher levels will obscure the lower levels. These images can also have a level of transparency allowing some of the lower levels to show through. Pages 2 and above are normal non displaying pages as in the other colour modes.

Pages that are not displayed are useful for preparing an image for display. As an example, loading an image for display (eg, in a photo slide show) takes a little time due to the limited speed of the SD card and this can be obvious to the user. However, you can have an image snap into view by loading it into a non display page and then copying that page it at high speed to page zero for display.

To do this you first need to set the default page for graphics operations to a non displaying page and this is done by the PAGE WRITE command. Then your program can load the image without disturbing the video display and finally it can copy that page to page zero.

For example:

```
PAGE WRITE 1
LOAD BMP "mybmp"
PAGE COPY 1 TO 0
```

The PAGE COPY command is optimised to perform its job very quickly (typically under a millisecond) so the user will see an instantaneous switch to the new image.

However, whenever data is written to the display page there is a chance that the data will be written while the graphics accelerator is copying that page to the VGA monitor. This could cause a short but noticeable glitch in the image on the monitor.

To avoid this you can specify an optional parameter to the PAGE COPY command to control when the copy is made:

```
PAGE COPY n TO m [,when]
```

The ‘when’ parameter is a single character that can be either:

- I Do the copy immediately. This is the default and is the most efficient but risks causing a glitch on the video output at the time of the copy.
- B Wait until the next frame blanking and then do the copy. It is the least efficient but it will never cause a glitch because the copy will have completed before the next vertical scan is started.
- D This will carry on processing the next command and do the copy in the background when the next frame blanking occurs. This is efficient but must be used with care as subsequent drawing commands may or may not be included in the copy depending on the timing of the next screen blanking.

As another example, the following short program implements a simple picture frame which will show a continuous sequence of images saved in a single directory on the SD card. Because the images are loaded into a non display video page before being copied at high speed to the display page each image will appear to snap into place:

```
PAGE WRITE 1                ' all images will load here
DO
  A$=DIR$("*.JPG")          ' scan SD card for images
  DO
    LOAD JPG A$             ' load the image
    PAGE COPY 1 TO 0, D     ' fast copy to the display page
    PAUSE 1000
    A$=DIR$()               ' get the next image file name
  LOOP WHILE A$<>""
LOOP                        ' keep repeating the sequence
```

As a side note: If you are at the command prompt and use the command PAGE WRITE with a page number other than zero the cursor will vanish and you will not see anything typed. This is because all output is now going to a page which is not displayed. If you accidently do this you can recover with Ctrl-C.

BLIT Command

The BLIT command will copy one part of video memory to another. This can be a copy within a page or it can be a copy from part of one page to part of another.

The syntax is: BLIT x1, y1, x2, y2, w, h [, page] [,orientation]

This will copy part of the image which has a top left position of ‘x1’ and ‘y1’, a width of ‘w’, and a height of ‘h’ to a new position with a top left position of ‘x2’ and ‘y2’. If you use BLIT to copy to an overlapping area the BLIT command will deal with it by buffering the original and then writing out the new version.

The optional ‘page’ parameter is the source page and the destination of the write will be the page set with the PAGE WRITE command. The ‘orientation’ parameter is also optional and allows you to mirror the image during the copy or exclude transparent pixels.

This example will draw a box on the screen and then duplicate it on another part of the screen using the BLIT command:

```
BOX 100,100,100,100,5,RGB(red),RGB(blue).   ' draw a box
BLIT 100,100,300,300,100,100.               ' copy the box
```

For something simple like a box you really don't need BLIT but it is very useful for copying complex shapes such as images. Also, because the BLIT command is optimised for speed the performance when moving complex images is very fast.

Another useful feature of BLIT is that these images can be residing on a non visible video page and this allows you to have a library of images which can be rapidly copied to the main video page. For example, you could have a number of sequential images of someone dancing and when these are copied at high speed to the display page (page 0) they will create an animated dancing figure.

The BLIT command can also be used to copy a portion of video memory to a buffer which is automatically allocated by MMBasic and restore this later to the same place or another place. This is often used to move an image over a background image without disturbing that background.

To do this you would use the BLIT READ command to copy a portion of the screen memory to a buffer (you can have up to 64 buffers). You can then draw your object (say a ball) on that place on the screen obliterating the image that was underneath. When you want to remove the ball you just use BLIT WRITE to copy the buffer back to the screen thereby restoring the image that was previously there. By continuously repeating this sequence you can move something (such as a ball) over a background image without disturbing the background.

Game Playing Features

The Colour Maximite 2 has many features designed to help programmers in writing computer games and in this section we will skim over them to give you the flavour of what is possible. These and other features of the graphics subsystem are explained in detail in a tutorial presented on the Back Shed forum: https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=12125

As mentioned before the video output to the VGA monitor can be selected by using the MODE command with many of the low resolution modes supporting sophisticated features like selectable transparency and multiple video planes which can be overlayed over each other and copied from plane to plane.

The PAGE SCROLL command will scroll a video page horizontally or vertically by a specified number of pixels allowing the games programmer to create a smoothly scrolling background for platform games and the like. The IMAGE RESIZE command will allow you to resize an image or an area on the VGA monitor from within a program and IMAGE ROTATE will allow you to similarly rotate an image or area by a specified number of degrees.

Sprites are another important feature. These are images that can be moved over the background without disturbing the background. Multiple sprites can be loaded and they can be moved around the screen, hidden or displayed as a group or individually. MMBasic also includes a versatile mechanism for detecting when two sprites collide allowing (for example) the programmer to create a realistic bounce effect.

While not directly related to graphics a useful feature is the ability to play music or sound effects on the stereo audio output in a variety of formats including WAV, FLAC or MP3. Computer generated music in the MOD format can be played as well as robotic speech in the TTS format.

Chapter 7

External Input/Output

The Colour Maximite 2 has an extensive range of input/output facilities which allow your BASIC program to control and interact with the outside world via the external I/O connector on the rear panel.

This is a 40-pin ribbon connector which provides 28 input/output pins. This image is a view of the connector as seen from the rear of the Colour Maximite 2:

The capabilities of each pin are:

Pin	Features
1	3.3 Volt Power
3	I^2C SDA
5	I^2C SCK
7	Analog I/O or COUNT 1
9	Ground
11	COM2: RX
13	Analog I/O or COUNT 2
15	Analog I/O or COUNT 3
17	3.3 Volt Power
19	SPI MOSI
21	SPI MISO
23	SPI CLOCK
25	Ground
27	I^2C2 SDA
29	Analog I/O or PWM-1C
31	PWM 2B
33	General I/O
35	SPI2 MISO
37	Analog I/O or COM1 DE
39	Ground

Pin	Features
2	5.0 Volt Power
4	5.0 Volt Power
6	Ground
8	Analog I/O or COM1: TX
10	Analog I/O or COM1: RX
12	Analog I/O or PWM 1A
14	Ground
16	Analog I/O or COM2: TX
18	FAST COUNT
20	Ground
22	Analog I/O or PWM 1B
24	Analog I/O or COUNT 4
26	Analog I/O
28	I^2C2 SCK
30	Ground
32	General I/O
34	Ground
36	PWM 2A
38	SPI2 MOSI
40	SPI2 CLOCK

Twelve pins (shown as shaded in gray) are for power and the common ground. These are:

3.3V Power	3.3 volt power source. The total current drawn from all these pins must not exceed 100mA.
5V Power	5 volt power source. The total current drawn from these pins is determined by the capacity of the 5V power source supplied to the Colour Maximite 2.
Ground	The common rail for all input/output.

Twenty eight pins are under control of the BASIC program. Each of these can operate as a standard digital input or output. In addition most pins support special functions (listed in the table above) which can be enabled by the program. In summary these are:

Analog I/O	These pins are capable of measuring voltages.
I^2C SDA and SCK	The signals for the I^2C communications protocol.
I^2C2 SDA and SCK	The second I^2C communications channel.
COUNT	These pins are capable of measuring frequency, period and counting pulses.
FAST COUNT	This pin is capable of counting up to 40MHz.
COM1 TX and RX	The signals for the asynchronous serial communications protocol.
COM1 DE	The data enable output signal for COM1 (if enabled).
COM2 TX and RX	The second asynchronous serial communications channel.
SPI MISO, MOSI, CLOCK	The signals for the SPI communications protocol.
SPI2 MISO, MOSI, CLOCK	The second SPI communications channel.
PWM	Five outputs which can be used to control servos or generate Pulse Width Modulated (PWM) signals.

All pins with Analog I/O capability have a maximum input voltage limit of 3.3 volts while the other pins can accept voltages of up to 5.1 volts. Note that the pin numbers used above are the pin numbers used by MMBasic to refer to an I/O pin. They are not the connector manufacturer's standard pin numbers.

Configuring a Pin

An input/output pin is configured using the SETPIN command. This command takes the form:

```
SETPIN nn, mode
```

where 'nn' is the pin number on the Colour Maximite 2 and 'mode' is how you would like the pin to be configured. This last parameter can be:

AIN	Analog input (ie, measure voltage)
DIN	A digital input.
FIN	Measure the frequency of the signal on a pin.
PIN	Measure the period (ie, the time between positive going edges) of the signal on a pin.
CIN	Count the number of pulses on a pin.
DOUT	A digital output.

For example, `SETPIN 16, DOUT` will setup pin 16 as a digital output.

Note that the pin number 'nn' refers to the physical number of the I/O connector pin as listed above.

To read from an input pin you use the PIN() function. For example:

```
var = PIN(5)
```

This will read the value of pin 5 and save it to the variable `var`. To write to a pin (ie, set its output) you use the same PIN function but this time you assign a value to it. For example:

```
PIN(3) = 0
```

will set the output of pin 3 to zero (which generally means a logic low). In this case the PIN() construct is used as a command. This dual nature of the PIN() construct (either input or output) sometimes confuses newcomers to MMBasic so watch out for it.

Digital Inputs

A digital input is the simplest type of input configuration. If the input voltage is higher than 2.5V the logic level will be true (numeric value of 1) and anything below 0.65V will be false (numeric value of 0).

What if the input is between 0.65V and 2.5V? The Colour Maximite 2's inputs employ what is called a Schmitt Trigger which is a circuit that prevents the inputs from flipping on and off with small variations of the input voltage (ie, noise on the input). It works like this; as the input rises from zero the value of the pin will remain at logic false (ie, zero) until the voltage exceeds 2.5V at which point it will change to true (ie, one). Then, if the voltage drops, it will remain at true until the input drops below 0.65V at which point the pin's value will change to false.

For most inputs the maximum input voltage is 5.1V however any pins that can be used as an analog input are be limited to 3.3V. If the input voltage is over the maximum allowable level you should use a resistor and a clamping diode on the input as illustrated.

Because the Colour Maximite 2's input impedance is very high (leakage is less than 1µA) you can use a large valued input resistor – for example, a 10K resistor which would be suitable for any input voltage up to 50V.

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 8, DIN
IF PIN(8) = 1 THEN PRINT "High" ELSE PRINT "Low"
```

The SETPIN command configures pin 8 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 execute the command after the ELSE.

Because the PIN() function will return 1 when the input is high and the IF … THEN command treats any non zero number in its conditional statement as true, you could rewrite the last line to read:

```
IF PIN(8) THEN PRINT "High" ELSE PRINT "Low"
```

In some cases you might want to read the input from a number of pins simultaneously. To do this you can use the PORT() function which has the form:

```
val = PORT(start, nbr)
```

'start' is the starting pin number and 'nbr' is the number of consecutive pin numbers that you want to read from. The function returns a binary number with each bit representing the state of a pin. For example, if you wanted to read the values of pins 21, 22, 23 and 24 you would use this:

```
val = PORT(21, 4)
```

ie, read four consecutive input pins starting with pin 21. The *Colour Maximite 2 User Manual* goes into more detail and it is required reading if you need to use the PORT() function.

Using a Switch as an Input

When you want to use a switch as an input you need a pull up resistor as shown on the right. The purpose of this resistor is to apply a voltage across the switch's contacts. Then, when the switch is closed the contacts will pull the input to zero and the PIN() function would return zero for closed and one for open.

Rather than using an external resistor the Colour Maximite 2 input can be specified with an pullup resistor. This resistor is internal and (when specified) will be connected between the input pin and the 3.3V supply (its value is about 100K) as illustrated in the diagram below right.

To specify a pullup resistor you use SETPIN as follows:

```
SETPIN pin, DIN, PULLUP
```

Using either an internal or external pullup resistor you also need to consider the issue of contact bounce. This is when the switch contacts mechanically touch and then bounce apart momentarily due to the momentum of the mechanical assembly. Because the Colour Maximite 2 runs very fast a BASIC program could see this as a sequence of quick button presses rather than a single press.

You could check for this in your program, for example by checking 100ms after the first contact closure to confirm that the contacts are indeed closed. A simpler solution is to connect a 100nF capacitor across the switch contacts as illustrated in the second diagram. This capacitor in association with the pullup resistor will average out any rapid contact bounce so that the program will see a smooth transition from on to off and vice versa.

Digital Outputs

All I/O pins can be configured as a standard digital output. The command to do this is:

```
SETPIN pin, DOUT
```

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 BASIC 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 (in fact any non zero value can be used to set the output high).

When operating in this mode, a pin is capable of sourcing or sinking about 10mA which is sufficient to drive a LED or other logic circuits running at 3.3V.

The pins that are 5V tolerant can be used to drive 5V logic via an open collector output. 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 the output 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 5.1V.

The diagram shown below illustrates how an open collector system works. Note that the circuit should really be called open drain because the STM32 uses a FET as the driver but open collector is a more common term and for consistency it is used here.

To set an output as open collector you use the SETPIN command in BASIC as you normally would but you append OC to the command to specify an open collector output. For example:

```
SETPIN pin, DOUT, OC
```

For driving high voltage and/or high current loads such as relays you should use a transistor (either bipolar or FET) to drive the load. To switch 240V AC a more elegant solution is to use a solid state relay. These have full isolation between their input and output and can switch 240V AC loads with a current of up to 10 amps. Some can be directly connected to a Colour Maximite 2 output pin but others need a drive voltage over 4V and in that case you should use an open collector output and a pull up resistor to 5V.

Other useful output devices are reed relays and optocouplers. Generally they can be directly driven by an output pin, are easy to use and provide isolation between the Colour Maximite 2 and the circuit that you are driving.

There are also many fully assembled modules that you can use with the Colour Maximite 2. For example, the module on the right provides four relays capable of switching 240V AC. It costs about $8 and each relay can be controlled by a Colour Maximite 2 output with no additional circuitry required (for this particular module search the internet for “Geekcreit Relay Module”).

As with the PIN() construct you can also use PORT() as a method of setting a number of outputs simultaneously to some state. For example, the following will simultaneously set pins 35, 36 and 37 to the low logic state (ie, zero volts):

```
PORT(35, 3) = 0
```

Sometimes you need to generate a pulse on an output pin. This can be conveniently done with the PULSE command:

```
PULSE pin, mSec
```

Where 'pin' is the pin number and 'mSec' is the desired pulse width in milliseconds. This last parameter can be a fraction (ie, 0.1ms) so very short pulses down to a few microseconds can be

generated. The polarity of the pulse is opposite to the current state of the pin. For example, if the output of the pin is currently low the pulse will be positive. Pulses can be up to many days in length and any pulse longer that 3ms will be run in the background – this means that the program will continue with the following commands and MMBasic will automatically terminate the pulse when its time is up.

Analog Input

The Colour Maximite 2 has twelve I/O pins that are capable of voltage (ie, analog) measurement. To set an I/O pin to this mode you use the command:

```
SETPIN nn, AIN
```

Where AIN stands for Analog IN and 'nn' is the pin number that you want to configure.

The analog input range is from zero to 3.3V. To measure voltages greater than 3.3V you will need a voltage divider and that will require the reading be scaled in the BASIC program to give the correct value.

Rather than finding precision resistors for the voltage divider a simpler approach is to connect a constant voltage to the input of the voltage divider, then record the voltage reported by the Colour Maximite 2 on its input pin (Vmm) and the voltage at the input of the voltage divider (Vmes) using a digital multimeter.

Then the reading could be scaled thus:

```
PRINT PIN(nn) / (Vmm / Vmes)
```

Note that to retain the accuracy of the reading the source resistance needs to be 10K or less. This means that in most circuits the value of R2 should be 10K or less.

For small voltages you will need an amplifier to bring the input voltage into a reasonable range for measurement. The diagram below shows a typical arrangement using the popular and inexpensive LM324 quad operational amplifier. The LM324 can operate from a single 5V supply and contains four identical amplifiers in the one 14 pin package.

The gain of the amplifier is determined by the ratio of R2 to R1 plus 1 and using the components shown the gain is 101. This number should be used in the BASIC program so that the readings are scaled to represent the input voltage.

For example:

```
PRINT PIN(9) / 101
```

Alternatively, you could adopt the technique used to scale the reading for a voltage divider (as described on the previous page). The result will be the same.

Frequency and Period Measurement

Four pins on the Colour Maximite 2 can be configured as to measure frequency, period or just count pulses on the input. These are labelled as COUNT in the pinout diagrams.

For example, the following will print the frequency of the signal on pin 15:

```
SETPIN 15, FIN
PRINT PIN(15)
```

The value returned by the `PIN()` function is the measured frequency in Hz. You can also configure the pins to measure the period (in milliseconds) between the rising edges of the input signal or to simply count the number of pulses received. The response to input pulses is very fast and the Colour Maximite 2 can count pulses as narrow as 10nS (although the maximum frequency of the pulse stream is still limited to about 200KHz).

Pin 18 is a FAST COUNT pin. This can measure frequency and count at up to 40MHz which, amongst other uses, means that it can be used as a general purpose frequency counter. Note that this pin cannot measure period as the other counting pins can.

You can measure the pulse width of an incoming signal by using the `PULSIN()` function. This has a number of options which can be difficult to explain so you should refer to the *Colour Maximite 2 User Manual* if you wish to use it.

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 at critical locations to check to see if the button has been pressed but an interrupt makes for a more cleaner and readable program.

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

An interrupt is setup using the SETPIN command:

```
SETPIN pin, type, subroutine
```

'pin' is the pin number which will trigger the interrupt, 'type' is the type of interrupt and can be INTH for a rising edge signal transition, INTL for falling edge transition or INTB for any change in the input (ie, interrupt on both rising and falling). 'subroutine' is the subroutine to execute when the interrupt trigger occurs – this is just an ordinary subroutine (nothing special).

You can set an interrupt on any I/O pin and you can have up to ten I/O pins simultaneously operating as interrupts, each with its own interrupt subroutine or, if you wish, sharing one or more subroutines. If two interrupts occur simultaneously MMBasic will execute the subroutine

associated with the interrupt that was defined first, then when it has finished (and the next interrupt condition still exists) it will execute the next interrupt subroutine, and so on.

While MMBasic is executing the interrupt subroutine all other interrupts are ignored. This means that if your interrupt code takes too long to execute there is a chance that another interrupt (such as a button push) might arise and vanish while your first interrupt subroutine is still executing – with the result that the new interrupt would be missed. For this reason interrupt subroutines should be as short as possible.

As an example of defining an interrupt the following code fragment will detect if the user has pressed a button (connected to pin 16) and, if so, will set the output of pin 15 high (this could operate a relay or something similar).

```
SETPIN 15, DOUT
SETPIN 16, INTL, MyInt
DO
   ` main processing loop
   ` more processing
LOOP

` interrupt routine
SUB MyInt
  PIN(15) = 1
END SUB
```

In the first line of the fragment we configure pin 15 as an output (this will drive our relay or whatever) and in the second line we configure pin 16 to be a digital input that will generate an interrupt on the high to low transition. The interrupt code is held in the subroutine `MyInt` and this is specified as the third parameter to the SETPIN command.

The DO…LOOP represents the main processing loop which runs forever. When the user presses the button connected to pin 16 the voltage on that pin will drop to zero, MMBasic will recognise this as a high to low transition and automatically interrupt the main program and execute the subroutine `MyInt`. This routine is very short; it just sets the output high and exits the subroutine which then allows the main program to continue as before. The main processing loop is completely oblivious to the interrupt. Normally an interrupt subroutine will have more than a single line in it but it does not have to be complicated – it should just do its job then exit.

Many other parts of MMBasic can also generate interrupts. For example, you can specify an interrupt that repeats with a specified number of milliseconds between each interrupt (the SETTICK command), you can have an interrupt when an IR remote control signal is received or when a certain number of bytes has been received on a serial interface.

Normally MMBasic will respond to a single interrupt within 5µs so you can use interrupts to catch fast events. For example, ignition pulses in a petrol engine.

When using interrupts keep in mind the following:

- Some MMBasic functions can block interrupts for up to 200ms so it is possible for an interrupt to occur and vanish within this time and never be recognised. These functions are generally involved with input/output to external devices (eg, measuring pulse width, getting a temperature, etc) so care needs to be taken if you are using any of these functions and interrupts at the same time.
- Remember the commandment "Thou shalt not hang around in an interrupt”. For example, never use PAUSE inside an interrupt. If you have some lengthy processing to do in an interrupt you should simply set a flag and immediately exit the interrupt. Then, in your main

program loop, you can detect the flag and do whatever is required then reset the flag. Always keep interrupts short and exit as soon as possible otherwise you run the risk of missing other interrupts or tying up MMBasic by continuously executing interrupts..

- 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 to prevent it from calling the subroutine while you are executing it (this would have undefined consequences including ruining your day). You can then reinstate the interrupt after you have finished with the subroutine.

Rotary Encoders

A good example of using interrupts is when you need to employ a rotary encoder as an input device. These are a handy method of adjusting the value of parameters in a Colour Maximite 2 project. A typical encoder can be mounted on a panel with a knob and looks (and acts) rather like a potentiometer.

A standard encoder has two outputs (labelled RA and RB) and a common ground. The outputs should be wired with pullup resistors as shown below:

As the knob is turned the rotary encoder will generate a series of signals known as a Gray Code. The program fragment below uses an interrupt to detect and decode the code and update the variable `Nbr` accordingly.

```
SETPIN 3, DIN            ' setup RB as an input
SETPIN 5, INTH, RInt     ' setup an interrupt when RA goes high

DO
  < main body of the program >
LOOP

SUB RInt                 ' Interrupt to decode the encoder output
  IF PIN(3) = 1 then
     Nbr = Nbr + 1       ' clockwise rotation
  ELSE
     Nbr = Nbr - 1       ' anti clockwise rotation
  ENDIF
END SUB
```

Because the decoding of the signals from the encoder is done within the interrupt subroutine (`RInt`) the main program (between the DO and LOOP commands) does not have to be concerned with handling the encoder's output. As far as the main program is concerned the value of the variable `Nbr` will be "magically" updated as the user rotates the knob.

Note that this program assumes that the encoder is connected to I/O pins 2 and 5; however any pins can be used by changing the pin numbers in the program. Also, it is intended for simple user input where a skipped or duplicated step is not considered important. It is not suitable for high speed or precision input.

Program courtesy MicroBlocks on the Back Shed Forum.

PWM and Servo Outputs

Five I/O pins can generate PWM or servo signals. PWM stands for Pulse Width Modulation which is a constant square wave output with a specified duty cycle and frequency. By varying the duty cycle (the ratio between the positive pulse and the negative pulse) your program can generate a synthesised voltage which can be used to control devices such as motor controllers which need an analog input. It can also be used to control the brightness of LEDs or incandescent lamps (read more about this technique at: http://learn.sparkfun.com/tutorials/pulse-width-modulation).

Another use for the PWM outputs is to generate a signal which, with a small loudspeaker, can create a range of audible tones.

The PWM outputs on the Colour Maximite 2 are organised into two channels, one of which has up to three outputs and the second two (for a maximum of five outputs). Within each channel all outputs will have the same frequency but each can have a different duty cycle. On the pin out diagrams for the Colour Maximite 2 the outputs for the first channel are labelled 1A, 1B and 1C while the two outputs for the second are 2A and 2B.

The syntax of the PWM command is:

```
PWM ch, freq, A-DutyCycle, B-DutyCycle, C-DutyCycle
```

'ch' is the channel number (1 or 2), 'freq' is the frequency (20Hz to 500kHz) and the remaining three parameters are the duty cycle for each of the outputs (0 to 100%). If you do not want to use an output you can leave that output off the end of the list and that pin can be used for some other purpose. After this command has been executed the output will run continuously unless changed or the Colour Maximite 2 is reset.

For example, the following will set the PWM 1A output to 1KHz with a duty cycle of 20% and 1B to a duty cycle of 60% at the same frequency. The 1C output is not specified so the pin allocated to 1C will not be affected and can be used for some other purpose:

```
PWM 1, 1000, 20, 60
```

This command can be used repeatedly to change the duty cycle (and frequency if required) of the PWM outputs at will.

You can also use the PWM outputs to control a servo (as illustrated on the right). Servos are a motor with integrated gears and a control system that allows the position of the shaft to be precisely controlled. The Colour Maximite 2 can simultaneously control up to five servos.

Depending on their size servos can be mechanically quite powerful and provide a convenient way for the Colour Maximite 2 to control the physical world.

Standard servos allow the shaft to be positioned at various angles, usually between -90 and +90 degrees. The position of the servo is controlled by a pulse which is repeated every 20ms. Generally a pulse width of 0.8ms will position the rotor at -90º, a pulse width of 2.2ms will position it at +90º and 1.5ms will centre the rotor. These are typical values and can vary between manufacturers.

The SERVO command is similar to the PWM command:

```
SERVO ch, 1A, 1B, 1C
```

'ch' is the channel number (1 or 2) and the remaining three parameters are the pulse width (in milliseconds) for each of the outputs. On the Colour Maximite 2 pin out charts the servo outputs are designated as PWM 1A, PWM 1B, PWM 2A, etc. This is because the PWM and SERVO commands are closely related and use the same I/O pins. As with the PWM command, if you do not want to use an output you can leave it off the end of the SERVO command.

The pulse width can be specified with a high resolution (about 0.005 ms). For example, the following will position the rotor of the servo connected to channel 1A to near its centre:

```
SERVO 1, 1.525
```

Following the SERVO command the Colour Maximite 2 will generate a continuous stream of pulses in the background until another servo command is given or the STOP option is used (which will terminate the output).

As another example, the following will swing two servos back and forth alternatively every 5 seconds: These servos should be connected to the outputs PWM 1A and PWM 1B.

```
DO
  SERVO 1, 0.8, 2.2
  PAUSE 5000
  SERVO 1, 2.2, 0.8
  PAUSE 5000
LOOP
```

Special Device Support

There are some devices that are often used in microcontroller projects and the Colour Maximite 2 provides special support for these. Using this built in support you can easily add features such as an infra red remote control or distance sensor to your project with just a few lines of BASIC code.

These special devices are:

- Infrared remote control receiver and transmitter
- The DS18B20 temperature sensor and DHT22 temperature/humidity sensor
- Ultrasonic distance sensor
- WS2812 multicolour LED chip.

The *Colour Maximite 2 User Manual* in the section "Special Device Support" provides a good description of each along with examples of their use.

Chapter 8

Communication Protocols

The Colour Maximite 2 supports a wide range communications protocols. These are integrated into the BASIC language and are easy to use so you can conveniently transfer data back and forth with test equipment, specialised ICs or sensors.

The list of these protocols covers asynchronous serial (TTL, RS232 or RS485), I^2C, SPI and 1-wire. Serial is used to communicate with test equipment and GPS modules, I^2C and SPI are mostly used to talk to specialised chips or sensors and 1-wire is a speciality protocol for certain types of sensors. This tutorial cannot cover each protocol in detail but it will provide enough information for you to understand how they work. You can then refer to the *Colour Maximite 2 User Manual* for the full details.

Asynchronous Serial Communications

Asynchronous serial is a communications method where the data is sent as a series of pulses on the signal line with precise timing, the receiver also uses the same timing so it can tell where in the data stream a bit of data should (or should not) be. This is a common communications protocol and is used by test equipment, personal computers and GPS modules.

For a more detailed description see: https://learn.sparkfun.com/tutorials/serial-communication

The Colour Maximite 2s has two general purpose asynchronous serial ports both of which can operate at high speeds and employ TTL signalling levels. This means that the voltage range of the signal matches the levels used by TTL logic (ie, logic low is zero volts and logic high is 3.3V). These signal levels allow you to directly connect to devices like GPS modules (which generally use TTL voltage levels).

To open a serial port you use the command:

```
OPEN "COMx:" as #n
```

Where COMx can be COM1 for the first serial port or COM2 for the second. #n is the reference number of the serial channel and can be any number between #1 and #10.

The speed of transmission in asynchronous serial is labelled 'baud' which is another way of saying bits per second. The Colour Maximite 2 serial ports default to 9600 baud but you can change this by appending the required speed to the end of the COM port specification when you open the port. For example this will open the second serial port at 1200 baud and assign it the reference number #4:

```
OPEN "COM2:1200" as #4
```

To send something out of the serial port you use the PRINT command. For example:

```
PRINT #4, "Hello"
```

This will send a series of characters spelling "Hello" out of the serial port opened as #4. To receive characters from a serial port you can use a number of commands or functions but the most useful is the INPUT$(x, #n) function which will retrieve x characters from the port opened as #n.

The list of commands and functions in MMBasic that will accept a serial port reference number are:

PRINT	Send a string
INPUT$()	Receive one or more characters
LINE INPUT	Receive a complete line
EOF()	True if no characters are waiting in the receive buffer
LOF()	The empty space (in characters) remaining in the transmit buffer

All serial communications in the Colour Maximite 2 are buffered which means that MMBasic will copy any incoming characters to a part of memory (the buffer) where they can be retrieved later. The advantage of buffering is that instead of waiting for characters to arrive your BASIC program can be doing something useful and just check from time to time to see if anything has arrived. The output is also buffered so that when you send some characters they are sent in the background and your program will continue without waiting for the characters to actually leave. This can sometimes trip up newcomers who (for example) might try to read the response from the other device while there is still data in the output buffer waiting to being sent to that device.

A practical example of using serial communications is sending data via wireless modules that use serial as their interface. A typical example is the HC-12 (about US$6 on eBay) as shown on the right. It defaults to a speed of 9600 baud and anything sent to the transmitting module will be received by the receiving module at the same speed.

For example, if you wished to measure a temperature with one Colour Maximite 2 and transmit that wirelessly to a second using a pair of HC-12 modules, your program on the sending computer might look like this (note that the function TEMPR(24) will read a DS18B20 temperature sensor connected to pin 24):

```
OPEN "COM1:9600" as #1
DO
  PRINT #1, TEMPR(24)
  PAUSE 800
LOOP
```

On the receiving Colour Maximite 2 the program could be:

```
OPEN "COM1:9600" as #1
DO
  LINE INPUT #1, T$
  PRINT T$
LOOP
```

Note that the TEMPR() function takes 200ms to make the measurement which is why we wait for 800ms in the pause command to make a total delay of one second. Another point to note is that PRINT command will add CR and LF characters to the end of the sent data and the LINE INPUT command will read characters until this pair are received, so they work well together.

When you open a serial port you can specify a number of options. These are part of the opening string. In our previous example we just specified the com port (COM1:) and baud rate (9600 baud) but you can also specify the size of the receiving buffer (handy if you are receiving high speed data) and an interrupt to be triggered when a certain number of characters has been received.

I²C Communications

Most sensors use either the I²C or SPI protocols to communicate their results and the Colour Maximite 2 will work with either. Typical sensors include acceleration, compass, electronic gyroscopes, temperature, humidity, pressure, light intensity and dozens more.

The I²C protocol is quite complicated but using it in MMBasic is quite straightforward. First you open the I²C channel using the I2C OPEN command, which allows you to specify the speed (up to 400KHz) and the timeout.

The syntax is:

```
I2C OPEN speed, timeout
```

'speed' is the transmission speed in KHz (normally 100) and 'timeout' is the length of time to wait (in ms) before deciding that the remote device is not going to respond.

With the port open you are the I²C master and you can send data using the I2C WRITE command and receive the response using the I2C READ command. Ie:

```
I2C WRITE addr, option, len, data
```

and

```
I2C READ addr, option, len, data
```

Each I²C device has an address which allows multiple devices to share the one set of input/output pins (ie, channel). 'addr' is the address, 'option' is a specialised setting which is normally set to zero, 'len' is the amount of data to send or receive and 'data' is a variable or constant when sending data or a variable where the received data is to be saved.

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

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

The hours and minutes maintained by the PCF8563 are held in two consecutive registers which we need to read (check the PCF8563 data sheet for the details). First the program defines an array to hold the received data. We then open the I²C channel and write the number of the first register that we want to read to the chip (register 3). The fourth line reads two bytes from the chip (minutes and seconds) and saves them in the previously defined array, `RData()`.

The PCF8563 real time clock is hardwired to recognise the address 51 (hex) on the I²C bus and that is specified in both the I2C WRITE and READ commands as &H51. The prefix &H indicates to MMBasic that the number is expressed in the hexadecimal notation.

For more details on the I²C protocol see: http://learn.sparkfun.com/tutorials/i2c

SPI Communications

The SPI protocol is simpler than I²C and is also used by many sensors. The Colour Maximite 2 can drive the SPI interface at up to 25MHz and has commands for sending and receiving bulk high speed data as well as managing the transfer on a byte by byte basic.

SPI can be configured in many ways and often manufacturers will interpret the protocol differently so reading through the data sheet for a device is important. This tutorial by SparkFun also provides a good overview of the protocol: http://learn.sparkfun.com/tutorials/serial-peripheral-interface-spi

Like the other communications protocols supported by MMBasic an SPI channel must be first opened then written to and read from as required.

The syntax for opening the SPI channel is:

```
SPI OPEN speed, mode, bits
```

Where 'speed' is the speed of the transmission, 'mode' is the transmission mode and 'bits' is the number of bits to send/receive. There are four different modes which are spelt out in detail in the *Colour Maximite 2 User Manual*.

The SPI protocol will receive data while it is sending something, for this reason the one function (SPI()) does both the sending and receiving. For example:

```
rdata = SPI(sdata)
```

will receive an SPI communication from the slave device and store the data in the variable `rdata` while at the same time send the byte in the variable `sdata`. This notion of receiving while sending can be confusing at first and this is another reason to carefully check the device's data sheet to see how the manufacturer implemented the send/receive function.

For high speed transfers you can send and receive bulk data using the SPI READ and SPI WRITE commands. Finally an SPI channel is closed with the SPI CLOSE command.

1-Wire Communications

The 1-Wire protocol was developed by Dallas Semiconductor and is used to communicate with chips using a single signalling line. It is mostly used to communicate with the DS18B20 temperature measuring sensor and MMBasic includes the TEMPR() function which provides a convenient method of directly reading that device (using the 1-wire protocol) without having to understand the complications of using the protocol itself.

If you wish to delve into the details of 1-wire communications you should refer to the *Colour Maximite 2 User Manual* and on line resources such as: http://en.wikipedia.org/wiki/1-Wire

Chapter 9

Special Features

The Colour Maximite 2 has a number of special features that are not covered elsewhere but are useful in a program. These include timekeeping features and features designed for cases where the computer is running unattended.

Setting Options

There are many options that you can set in the Colour Maximite 2 and these are all set using the OPTION command. These settings are divided into some that will only last for the duration of the currently running program and therefore are used in a program and others that will be remembered even after the power has been removed. These are most often entered at the command prompt.

When the Colour Maximite 2 is first used the options will have been set to reasonable defaults, so you should not need to change them. But, to give you the flavour of what you can do, the following are some of the settings that are available.

OPTION AUTORUN OFF | ON	Run the program when power is applied
OPTION BASE 0 | 1	Set the lower limit of arrays
OPTION BREAK nn	Set the character that will break out of a program
OPTION RTC CALIBRATE ±n	Trim the Colour Maximite 2's internal clock
OPTION COLOURCODE ON | OFF	Enable colour coding in the program editor
OPTION DEFAULT FLOAT | INTEGER | STRING | NONE	Set the default type of a variable
OPTION EXPLICIT	Require that variables be properly declared
OPTION STATUS ON | OFF	Turn on/off the status line at the command prompt
OPTION RESET	Reset all options to their defaults
OPTION TAB 2 | 4 | 8	Set the size of tabs

Note that the vertical bar between words (eg, OFF | ON) means that you can use either one or the other (eg, OFF or ON) in the command. For example, OPTION AUTORUN ON. For the details and more options see the *Colour Maximite 2 User Manual*.

Keeping Time

In the Colour Maximite 2 there are many ways that a program can track the time including an internal clock/calendar, a millisecond timer, timed interrupts and the PAUSE command.

The current date and time can be accessed using the special identifiers DATE$ and TIME$ which act like pre defined string variables that you can pull apart using the string functions or just use as a string. As an example, if you entered this at the command prompt:

```
PRINT DATE$ TIME$
```

You could expect to see something like this: `21/11/16 14:53:21`

The Colour Maximite 2's internal clock is maintained by the battery on the motherboard and you can set this by assigning a string to the DATE$ and TIME$ variables. For example, this will set the date to the 10th of July 2020:

```
DATE$ = "10/6/2020"       ' note that the format is dd/mm/yyyy
```

TIMER is another special identifier which returns the number of milliseconds since being reset to zero (it is also reset when the Colour Maximite 2 is powered up). You can use it to measure the time difference between two events as shown in the following example:

```
TIMER = 0
  ' section of code that needs to be timed
PRINT TIMER "ms"
```

In a large program resetting the timer can get confusing as you might reset it in several places and cause a conflict. An alternative is to save the current value of the timer and use that value without resetting the timer. For example:

```
TIMERCOUNT = TIMER
  ' section of code that needs to be timed
PRINT TIMER - TIMERCOUNT "ms"
```

The TIMER function can also be used to wait for a certain length of time but a better method is to use the PAUSE command which will halt the execution of a program for a precise number of milliseconds.

For example, to create a 12ms wide pulse you could use the following:

```
SETPIN 4, DOUT
PIN(4) = 1
PAUSE 12
PIN(4) = 0
```

Sometimes, after setting a control signal for a device, you might be required to wait for a defined number of milliseconds before you can set the next control signal. The PAUSE command is perfect for this type of job and many similar jobs that require a delay.

MMBasic also allows you to set up to four "tick" timers. Each acts like the tick of a clock and on each tick MMBasic will execute an interrupt subroutine specified in the command. Up to four "tick" interrupts can be setup. The tick times are specified in milliseconds and can range from a few milliseconds to many days. Think of it as the regular "tick" of a watch.

For example, the following code fragment will print the current time and the voltage on pin 7 every second. This process will run independently of the main program which could be doing something completely unrelated.

```
SETPIN 7, AIN
SETTICK 1000, DoIint
DO
   ` main processing loop
LOOP

SUB DoIint          ` tick interrupt
  PRINT TIME$, PIN(7)
END SUB
```

The second line sets up the "tick" interrupt, the first parameter of SETTICK is the period of the interrupt (1000 ms) and the second is the interrupt subroutine which will be executed on every

"tick". Every second (ie, 1000 ms) the main processing loop will be interrupted and the program starting at the label `DoIint` will be executed.

Autorun

If the Colour Maximite 2 is unattended you might want the program to automatically start running after a power failure. This is achieved by setting AUTORUN on:

```
OPTION AUTORUN ON
```

Then, when the power is cycled the Colour Maximite 2 will automatically run the program in memory. This command can be entered at the command prompt or used in the program previously loaded into program memory and will be remembered even after a power loss and restart.

Recovering From Errors

If the Colour Maximite 2 is used unattended there is always the possibility that something could cause MMBasic to generate an error and return to the command prompt. . Another possibility is that the BASIC program itself could get stuck in an endless loop for some reason. In both cases the visible effect would be the same - the Colour Maximite 2 would stop doing its programmed job until the power was cycled. To handle this possibility MMBasic has two mechanisms for dealing with errors; the watchdog timer and turning off error checking.

The watchdog timer 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 applies even if MMBasic is sitting at the command prompt. The WATCHDOG command specifies how many milliseconds are allowed before the reset. For example, the following will set the watchdog timer to 200 milliseconds:

```
WATCHDOG 200
```

Normally this command will 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 reach zero and the program will be restarted (assuming that AUTORUN is set).

The program can check if it has been restarted by the watchdog timer by examining the value of the built-in variable MM.WATCHDOG. This is set to true if the restart was forced by the watchdog timer and false if it was a normal (eg, power on) startup.

The watchdog timer is foolproof but rather crude. Another method of handling errors with more finesse is to use the ON ERROR command which allows you to trap and respond to any errors line by line in your program. For example, your program might use data from an external device or user that could cause a divide by zero error This would cause MMBasic to generate an error, halt the program, return to the command prompt and wait forever for some input.

The solution is to use the ON ERROR SKIP command which will skip or ignore any errors generated by the next command in the program. For example, the following could cause a "Divide by zero" error us the variable happens to be zero:

```
…
ON ERROR SKIP
A = 34/UserNumber
…
```

The ON ERROR SKIP command will cause MMBasic to ignore any error in the following command and that means that, if `UserNumber` was zero, the program will carry on as if nothing has happened (as a side effect the value of A will not be changed if there was an error).

Often you will want to do something more than just ignore the error, for example, perhaps print an error message. This can be accomplished by checking the built-in variables MM.ERRNO and MM.ERRMSG$ which are automatically created by MMBasic and are set to non zero and the text of the error message when an error is skipped.

For example:

```
…
ON ERROR SKIP
A = 34/UserNumber
IF MM.ERRNO <> 0 THEN PRINT "Invalid number ignored"
…
```

Sometimes it is possible that a group of commands can generate an error and in that case you can specify how many commands to skip the error checking by using ON ERROR SKIP nn where *nn* is the number of commands. There is also the command ON ERROR IGNORE which will completely ignore all errors in all commands until the command ON ERROR ABORT is encountered. This last command will restore the normal behaviour if an error occurs (ie, display an error message on the console and stop the program).

You need to be careful when skipping errors as this will cause MMBasic to ignore **all** errors including spelling mistakes, invalid commands, typos, etc. It is helpful having MMBasic point out these sorts of errors and it can be difficult to figure out why your program is not running correctly if the error reporting is turned off. For this reason you should fully test your program before adding code to skip errors and even then, it should only be used in specific cases which cannot be handled in any other way.

Saving Data

Sometimes it is handy to save some data that can be recovered when power is restored without having to write it to an SD card. This might be calibration data, user options, current state, etc.

This can be done with the VAR SAVE command which will save the variables listed on its command line in non volatile flash memory. A typical use is like this:

```
VAR SAVE ConfigX, ConfigY
```

On power up 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 short program provides an example, it is not very practical but it does illustrate how the VAR SAVE feature can be used:

```
VAR RESTORE
IF Config1 = 0 AND Config2 = 0 THEN
  INPUT "Config data 1", Config1
  INPUT "Config data 2", Config2
  VAR SAVE Config1, Config2
ELSE
  PRINT "Restored configuration data"
ENDIF
…
```

The VAR RESTORE command at the start of the program will try to restore any (and all) saved variables. If none have been saved the command will do nothing.

The program will then check if the variables Config1 or Config2 are set to a non zero number indicating that they have been previously set and saved via VAR SAVE (and therefore the VAR RESTORE command found them and restored them). If Config1 or Config2 are both zero the program will then get the settings from the user and save them ready for the next restart.

Note that the very first time that the program is run there will be nothing to restore (because nothing has been saved) but that does not matter, the command will not generate an error.

Playing Music and Sound Effects

The Colour Maximite 2 can play music and sound effects saved on the SD card in a variety of formats including WAV, MP3 and FLAC using the PLAY command. Once this command has been executed the BASIC program will continue while the file plays in the background. An interrupt can be setup to trigger when the play command has come to the end.

The PLAY command can also be instructed to play all files in a directory back-to-back and only signal an interrupt when the last file has completed playing. While playing in this background mode the Colour Maximite 2 can be used to change directories, edit programs, run programs, etc without interrupting the playing of the music. Amongst other things this allows the programmer to use the computer as a music player while programming or doing other tasks.

Sorting Data

The SORT command can be used to sort an array. For example, the array city$() might contain the names of world cities and can be easily sorted into increasing alphabetical order with the command:

```
SORT city$()
```

The SORT command will work with strings, floats and integers however the array to be sorted must be single dimensioned.

Often data is held in multiple arrays, for example, the name of each city might be held in the array city$(), the population held in the array pop%() and the size of the city held in area!(). The same index would refer to the name, population and the area of the city.

Sorting and accessing this data is a little more complex but it can be done relatively easily using an optional parameter to the sort command as follows: `SORT array() ,indexarray%()`

indexarray%() must be a single dimension integer array of the same size as the array being sorted. Following the sort indexarray%() will contain the corresponding index to the **original** data **before it was sorted**. (anything previously in indexarray%() will be overwritten).

To access the sorted data you would first copy the array holding the main key to a temporary array and sort that while specifying indexarray%(). After the sort indexarray%() can be used to index the original arrays. For example:

```
DIM city$(100), pop%(100), area!(100), sortindex%(100), temp$(100)

FOR i = 0 to 100
  temp$(i) = city$(i)        ' copy the keys to a temporary array
NEXT i
SORT temp$(), sortindex%()   ' sort the temporary array

FOR i  = 0 to 100
  k = sortindex%(i)          ' get the index to the original array
  PRINT city$(k), pop%(k), area!(k)        ' print in sorted order
NEXT i
```

GRAPHICS Programming on the Colour MaxiMite 2

Peter Mather

Version 2.0a

Copyright 2020 - Peter Mather

INTRODUCTION

This is a compilation of the threads posted by Peter Mather as a background and deeper explanation of the graphics capabilities of the CMM2. I have taken the liberty (with Peters OK), to post them on the forum with some additional comments and explanations of recent developments in the hope that they will assist others in realising the full potential of the CMM2. Any typing mistakes, errors or omission are mine (apologies Peter). There is also demo code included that was created by members of TheBackShed forum (with their permission - thanks). Editor: Doug Pankhurst.

From Peter:-

'I thought I'd start a thread to try and explain the concepts behind the CMM2 graphics and give some annotated examples of how to use the facilities in the CMM2. I'm sure Mauro will chime in and then perhaps some of what is in the thread can form the basis of a graphics manual of some sort.'

Revision History:

Version 2a: Reformatted and added information about version 5.05.05 firmware.

Version 1g: Updated/added information about the MODE command.

Version 1f: Updated/added information about the PAGE command.

Version 1e: Added Table of Contents, FRAMEBUFFER, Quaternions and corrected program bugs.

Version 1d: Adds more demonstration programs.

Version 1c: Includes information on graphics features in MMBasic 5.05.04RC6 (the draft version)
Note: These features may not end up included in the final release.

Version 1b: Updated hot links to programs and demos. Includes features and capabilities from MMBasic 5.05.03.

Table of Contents

Basic Graphics on the CMM2 4
Graphics Pages 6
PAGE COPY 7
Colour Lookup Tables 8
The BLIT command 11
Graphics Modes 15
Colour Depths 16
The PAGE Command 21
PAGE SCROLL ..... 21`
PAGE STITCH 24
12-bit Mode Graphics 26
Images 30
Sprites and how to create them 34
Sprites and how to use them 39
The FRAMEBUFFER 43
Quaternions 45

BASIC GRAPHICS on the CMM2

This first post will cover the basics of how the graphics are generated and work.

Some of the STM32 chips, including the STM32H743IIT6 that we are using, have an on-chip graphics controller (called the LTDC = **L**cd **T**ft **D**isplay **C**ontroller).

This is programmed to read from a defined area of memory at a specific rate and write the data to a set of I/O pins. In our case the I/O pins are connected to the resistors that make up the three R2R ladder DACs on the CMM2 motherboard and these create the analogue VGA signals. In addition, the LTDC is programmed to create the HSync and VSync signals which tell the monitor that we have reached the end of a line and/or end of a page.

All these signals must waggle at a rate determined by the a defined VGA specification depending on the resolution required.

In the case of the 800x600, display memory is being read at a rate of 40MHz, the line rate is 37.8787KHz and the frame rate is 60Hz.

The important point about the LTDC is that all this happens without any use of the processor once the controller has been initialised. It is basically a continuous circular DMA (Direct Memory Access). Of course this isn't completely free as it does consume memory bandwidth even though the processor isn't involved.

So to get something onto the screen all we have to do is write to the area of memory that the LTDC is reading and something will appear on the screen.

For performance reasons the best memory to use is the STM32H743IIT6's internal memory and 512KB of this is allocated to the main video memory. In addition I have allocated 3MB of the 8MB SDRAM memory (this is the chip on the back of the Waveshare PCB) to use for graphics.

So we have 3.5MB of memory available for graphics, 512KB is located at memory address &H24000000 and 3MB at address &HD0000000. We will see in the next section how we can poke this memory to write things on the screen.

However, we need one more concept before starting to explore how to use this memory - the Colour LookUp Table or CLUT. The default resolution of the CMM2 is 800x600 with an 8-bit colour depth so the screen uses 480,000 bytes out of the 512 KBytes available in the processors internal video memory.

However, we need to drive 16 data lines not just 8 and the way this happens is that the LTDC takes each 8-bit value in turn as it reads memory and uses it as an index to a table of 16-bit values (the CLUT). This table is initialised by the CMM2 firmware to sensible values to give good colour coverage but can be changed from within MMBasic using the MAP command (subject for another post).

OK, so we know that in 800x600x8 resolution the screen is using 480KB of memory starting at &H24000000. This memory is organised exactly as you would expect. The first 800 bytes are the top line of the display starting at the top left hand corner. The second 800 bytes are the second line of the display. and the 480,000th byte is the bottom right of the display.

So lets draw a diagonal line on the display to prove this.

```
pix%=&H24000000
for i%=0 to 599
   j%=i%*800+i%
   poke byte pix%+j%,255
next i%
```

Note how the line just writes across anything already on the screen (except for the SAVE command which was entered after the program was run).

It is a bit silly having to memorise &H24000000 so the CMM2 includes a built in function to give it to us

```
pix%=mm.info(page address 0)
for i%=0 to 599
 j%=i%*800+i%
 poke byte pix%+j%,255
next i%
```

Of course we could have done the same thing using the command

```
line 0,0,599,599
```

Internally in the CMM2 firmware the LINE command is doing exactly what we did with POKE. And, of course, all the basic graphics commands are doing the same thing - writing to memory in order to construct the requested shapes.

Introduction to Colour

A brief side trip to explain the use of colour in the CMM2 may be in order, particularly for newcomers. This is a collection and perhaps a little expansion on details provided elsewhere in this manual.

You will all be aware I am sure of the format used to define colour in MMBasic code – that being RGB(*nnn,nnn,nnn*) where *nnn* represents a number between 0 and 255 for Red, Green and Blue respectively. As you know, it takes 8 bits to represent all the values from 0 to 255 and thus this format is known as RGB888 (also 24 bit colour). With this format you can choose up to 16,777,216 colours (256 x 256 x 256).

Memory is like gold in the CMM2 and RGB888 uses up LOTS of memory so a compromise was decided on by the developers. As you can see from the schematic of the CMM2, there are only 5 video lines for red, 6 for green and 5 for blue meaning 5 bits, 6 bits and 5 bits for red, green and blue respectively. This is known as RGB565 (16 bit colour) and provides for 32 intensity levels of red, 64 for green and 32 for blue and is the format used in any mode using 16 bit colour depth. RGB565 gives 65,535 possible colours plus black or more accurately, no colour at all.

As an even more parsimonious use of memory and also to be similar to the colour capabilities of the older computers from the 80's, 8 bit colour depth is provided that only uses some of the video lines. Having only 8 bits to play with, the format is RGB332. That is 3 bits for red, 3 bits for green and 2 bits for blue. As this is a quite limited range, these values are used as an index into a translation table called a **C**olour **L**ook**U**p **T**able. There is a detailed description of this in a later chapter titled CLUT.

Back to the 16 bit colour depth for the moment, the upshot of this translation from MMBasic level RGB888 down to the hardware level of RGB565 is that there is no change in intensity (stored value) between a red value of 104 and 108 for example. In fact, as we can only represent 32 possible values of red and blue and 64 of green, the RGB888 values are translated by the function modulo 8, modulo 4 and modulo 8 respectively for red green and blue.

This can be shown with the following code:-

```
' demo of RGB888 to RGB565 translation
' use red as an example
option default integer
mode 1,16
page write 1
cls
' first, plug in some RGB888 values
' incrementing red by 1 each time
x = 0
y = 0
cr = 100
cred = cr
cgrn = 0
cblu = 0
  for j = 0 to 49
    pixel x,y,rgb(cred,cgrn,cblu)
    x = x + 1
    cred = cred + 1
  next j
' now look at the display memory
' to see the RGB565 values stored
addr1= mm.info(page address 1)
cred = cr
for k = 0 to 75 step 2
  a1$ = hex$(addr1+k)+"  "
  a2$ = hex$(peek(short addr1+k),4)
  a3$ = "     rgb("+str$(cred)+","+str$(cgrn)+","+str$(cblu)+")"
  text 0,30+(k*7),a1$+a2$+a3$
cred = cred + 1
next k
page copy 1 to 0
do
loop
```

There is an additional colour mode used in the CMM2, the 12 bit colour depth. This is a good deal more complex as it includes the functionality of transparency. It is referred to as ARGB4444 and as you would imaging uses 4 bits to represent transparency, red, green and blue respectively. There is more detail in the chapters following but the principles for translation from RGB888 to ARGB4444 are similar. I will try to expand on this once I learn more about it! (-:

Next chapter we will explore graphics pages and how they allow us to do some of the magic but if you properly understand the above then everything else should be easy.

GRAPHICS PAGES

In the previous post we saw that writing to the memory identified by MM.INFO(PAGE ADDRESS 0) displays immediately on the screen. In fact the CMM2 always and only ever shows on its display the information in the memory starting at MM.INFO(PAGE ADDRESS 0) (NB: not true in 12-bit colour depth but that is a subject for a much later post). This is important as it underpins the explanations below.

Drawing the line using POKE in the previous example took 12mSec on a 400MHz CMM2. This is fast enough that the act of drawing didn't create any strange visual effects.

However, if we slow it down by adding a pause we can see the line drawing across the screen as it now takes 614mSec.

```
pix%=mm.info(page address 0)
for i%=0 to 599
  j%=i%*800+i%
  poke byte pix%+j%,255
  pause 1
next i%
```

This is a trivial example but lots of graphically intensive activities do take "real" time to complete and it may be that we don't want to see the update taking place but would rather move from one static image to another.

A rather better example would be displaying a BMP image. BMP files are uncompressed so they are large and take time to load as they wait for the SDcard to read. They also load from the bottom up which is slightly strange. A full colour 800x600 BMP takes just less than a second to load.

If we type

```
? MM.INFO(MAX PAGES)
```

we get the answer 6. This says that in our current 800x600 resolution the 3.5MB of video memory is being split into 7 chunks, numbered 0 to 6.

We know that Page 0 is what is being displayed on the screen and it will always be Page 0 that is displayed. However, we don't need to write to Page 0. MM.INFO(MAX PAGES) tells us that in 800x600x8 video mode we have 7 video pages at our disposal.

We can see where they are in memory using MM.INFO(PAGE ADDRESS n)

```
for i=0 to 6:print hex$(mm.info(page address i)):next i

24000000
D0000000
D0080000
D0100000
D0180000
D0200000
D0280000
```

We can tell the CMM2 which page to use for writing using the command

```
PAGE WRITE n
```

So lets type PAGE WRITE 1 at the command line

OH NO!!!! The status line just disappeared, the cursor disappeared and when I type, nothing happens!

All console output is now being directed to a different area of memory starting at &HD0000000. Luckily any error or typing Ctrl-C will bring things back. PAGE WRITE isn't very useful at the command line but can be used.

How about?

```
PAGE WRITE 1:LOAD BMP "mybmp":PAGE WRITE 0
```

This time my cursor came back OK after about a second but no picture?

Now type:

```
PAGE COPY 1 TO 0
```

Instantly the picture will appear (actually it takes about 4mSec).

Now we can use this to create a BMP slideshow where each image instantaneously replaces the previous one.

```
page write 1
a$=dir$("*.bmp")
do
  load bmp a$
  page copy 1 to 0
  pause 1000
  a$=dir$()
loop while a$<>""
```

The PAGE COPY command is very highly optimised to run as fast as possible. It also has an optional parameter which is documented in the user manual.

- **I**: means do the copy immediately. It is the default and most efficient but risks causing screen artefacts;
- **B**: means wait until the next frame blanking and then do the copy. It is the least efficient but is absolutely determinate in its effect and no screen artefacts will ever be seen; and
- **D**: means carry on processing the next command and do the copy in the background when the next frame blanking occurs. This is efficient but must be used with care as subsequent drawing commands may or may not be included in the copy depending on the timing of the next screen blanking.

In a future post we will improve the slideshow program by having one image replace the previous by slowly scrolling in from the right hand side using another of the PAGE command functions.

But to summarise:

- PAGE COPY allows you to pretty much instantaneously replace what is being seen with a new image with no artefacts.
- PAGE COPY is very fast. For example in 320x240x8 graphics mode it takes less than 0.3mSec.
- Multiple pages allow you to store and/or construct complex images without any visible screen artefacts.

Some additional sub-commands for the PAGE command are :

```
PAGE AND_PIXELS sourcepage1, sourcepage2, destinationpage
PAGE OR_PIXELS sourcepage1, sourcepage2, destinationpage
PAGE XOR_PIXELS sourcepage1, sourcepage2, destinationpage
```

These combine the pixels on sourcepage1 and sourcepage2 by ANDing, ORing, or XORing them. destinationpage can be the same as either of the sourcepages if required. For example:

```
mode 3,16
load jpg "tiger-320"
pause 1000
page write 1
cls
box 50,50,220,100,,rgb(white),rgb(white)
timer=0
PAGE AND_PIXELSs 0,1,0
print timer
do:loop
```

This loads the image to page 0, the view page as normal. Then a white box is drawn on page 1. The PAGE command then ANDs page 1 and page 0 and replaces page 0 with the result. The effect is to use the white box to window the original image. Of course this example is simplistic but I'm sure Mauro, amongst others of you, will find innovative uses.

COLOUR LOOKUP TABLES

In this post we look more at the Colour LookUp Table (CLUT) and the MAP command and function and use them to create a rotating colour disk.

As explained earlier, 8-bit colour modes use a lookup table to convert from a single byte of information to the RGB565 pixel that will be displayed. In the CMM2 firmware the 8-bits are treated as an RGB332 pixel and are mapped to the most appropriate RGB565 value.

However, we can override this using the MAP command.

Normally a pixel that is set to zero would mean that it would be black on the screen but if we use the commands

```
MAP(0)=RGB(RED)
MAP SET
```

Then every "black" pixel on the display will instantly turn red. Actually, it isn't quite instant as the MAP SET command waits for the next frame blanking period before making the change. So MAP(n)=colour primes the colour change and MAP SET enacts it.

This allows us to change a number of colours simultaneously by using multiple MAP(n)=colour commands and then a single MAP SET.

To use our new colours in a drawing command we use the MAP(n) function. So for example if we type MAP(134)=RGB(148,73,66) to get some particular shade then to use that colour we would reference it as MAP(134)

e.g.

```
MAP(134)=RGB(149,73,66)
MAP SET
TEXT MM.HRES\2, MM.VRES\2,"Hello",CM,,5,map(134)
```

This would display the text in a dark brown colour. If we then typed

```
MAP(134)=RGB(BLUE)
MAP SET
```

The text would immediately turn blue as would any other pixels on the display set to 134.

There a couple of other MAP commands

```
MAP MAXIMITE
```

sets the CLUT to mimic the colours of the original Colour Maximite. i.e. 0-7 are the Maximite primary colours.

```
MAP RESET
```

restores the original RGB332 mapping

Now lets look at using the MAP command to create the colour wheel above

```
' Set up an array to hold the colour mappings we are going to use
Dim integer cmap(6)

'Clear the screen
CLS

'Set up 6 colours in the array
cmap(1)=RGB(red)
cmap(2)=RGB(green)
cmap(3)=RGB(blue)
cmap(4)=RGB(yellow)
cmap(5)=RGB(magenta)
cmap(6)=RGB(cyan)

' Do an initial update of the CLUT to set up our colours
mapclut

'Display an outer circle in white
Circle MM.HRes/2,MM.VRes/2,MM.VRes/2-1,0,,RGB(white),RGB(white)

' Now draw a simple colour pie chart using our new colours with the ARC command
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,0,60,map(1)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,60,120,map(2)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,120,180,map(3)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,180,240,map(4)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,240,300,map(5)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,300,360,map(6)

' Start a never ending loop
Do
'each time round the loop move the colours in our array one place to the left
' Use array element 0 to store the first element that is going to be at the end
  cmap(0)=cmap(1)
  cmap(1)=cmap(2)
  cmap(2)=cmap(3)
  cmap(3)=cmap(4)
  cmap(4)=cmap(5)
  cmap(5)=cmap(6)
  cmap(6)=cmap(0)

' reset the colour map
  mapclut

' pause so we can see the change
  Pause 200
Loop

'This subroutine updates the colour map for the colours we are using
'  set map positions 1 to 6 to the new colours
'  then apply the change
Sub mapclut
 map(1)=cmap(1)
 map(2)=cmap(2)
 map(3)=cmap(3)
 map(4)=cmap(4)
 map(5)=cmap(5)
 map(6)=cmap(6)
 map set
End Sub
>
```

Next post we are going to look at BLIT. Sounds complex? Actually all it is is a memory to memory copy - nothing more, but perhaps the most powerful tool in the graphics arsenal.

BLIT

BLIT does a memory to memory copy. End of lesson.

Well perhaps not but that is all it is doing behind the scenes. Lets look at the BLIT syntax

```
BLIT x1, y1, x2, y2, w, h [, page] [,orientation]
```

This says copy part of the image which has a top left position of x1,y1, a width of w, and a height of h to a new position with a top left position of x2, y2.

An example:

```
' Draw a simple shape on the screen
CLS
box 100,100,100,100,5,rgb(red),rgb(blue)

'copy the box to a new location with a top left corner at 300,300
blit 100,100,300,300,100,100
```

By the way, that copy took 88 microseconds which equals 100*100/0.000088=113 Mbytes/Sec. BLIT is very fast.

Thinking back to the first post, we know what was just happening and we could have done the copy in Basic

```
CLS
box 100,100,100,100,5,rgb(red),rgb(blue)

'copy the box to a new location with a top left corner at 300,300
page_address%=mm.info(page address 0)
x1%=100
y1%=100
x2%=200
y2%=200
w%=100
h%=100
for y%= 0 to h%-1
 for x%= 0 to w%-1

'calculate the address of the source pixel
   pix_in%=MM.HRES * (y% + y1%)+ x1% + x% + page_address%

'calculate the address of the destination pixel
   pix_out%=MM.HRES * (y% + y2%)+ x2% + x% + page_address%

'copy the pixel
   poke byte pix_out%, peek(byte pix_in%)
 next x%
next y%
```

This takes just over half a second - BLIT is *much* faster!!!

Try the following commands:

```
timer=0:blit 100,100,300,300,100,100:?timer
timer=0:blit 100,100,301,301,100,100:?timer
timer=0:blit 101,101,301,301,100,100:?timer
```

The timings I get are 88uSec, 155uSec, and 214uSec.

Tip! The STM32 is much faster when it can copy data that is aligned on 4 byte boundaries. For most applications this won't matter but for a very high performance game it is something to take into account.

Now lets explore the orientation parameter of the BLIT command. This is most easily explained by a couple of examples

```
CLS
' draw a large number 4
text 100,100,"4",,6,,rgb(red),rgb(blue)

' copy the "4" but mirror it left to right
blit 100,100,200,100,32,50,,1

' copy the "4" but mirror it top to bottom
blit 100,100,100,200,32,50,,2

' copy the "4" but mirror it top to bottom and left to right (rotate 180)
blit 100,100,200,200,32,50,,3

do:loop
```

```
CLS

' draw a large number 4
text 100,100,"4",,6,,rgb(red)

' draw a colour box
box 180,180,72,90,0,rgb(blue),rgb(blue)

' move the text but only non-transparent (non-zero) pixels
blit 100,100,200,200,32,50,,4

do:loop
```

Of course the orientation bits can be combined in any combination.

If you BLIT from one area to another that overlaps it, then BLIT is clever enough to understand this and deal with it by buffering the original and then writing out the new version.

Up to now we have ignored the page parameter in BLIT but this is perhaps the most powerful aspect of the command.

As we saw in a previous post we can use PAGE WRITE to set drawing output to any of the video pages available but we will always see what is in page 0. The page parameter in BLIT specifies the page that BLIT will read from and then it will write to the page specified by PAGE WRITE.

Another example needed. We can use BLIT to create the basis of a sliding block puzzle

```
'set output to page 1
page write 1

' clear page 1 to blue
cls rgb(blue)
'write the numbers 1 to 8 spaced 50 pixels apart
' font 6 is 32x50 pixels
for i = 1 to 8
   text i*50+10,0,str$(i),,6,,rgb(red),-1
next i


'now we want to randomize the numbers for our grid including the missing one
dim r(8)=(9,9,9,9,9,9,9,9,9) 'create an array to hold the positions
for filled=0 to 8
 do
   test=int(rnd()*9) 'get a random number between 0 and 8
   for j=0 to filled 'check all the filled cells to see if we have already used this number
     if r(j)=test then test=9 ' if yes then set no-good
```

```
   next j
   if test<>9 then r(filled)=test 'if OK then store the number
 loop while r(filled)=9 'loop until that cell is filled
next filled
'Now we will go back to the view page
page write 0
cls
' create a background for the grid
box 100,100,170,170,0,rgb(green),rgb(green)
' now copy the randomised numbers into our grid
for x=0 to 2
 for y=0 to 2
'get the x position of the randomised number in page 1
   xsource = r(x * 3 + y) * 50
' blit the number from page 1 to page 0
   blit xsource, 0, x * 55 + 105, y * 55 + 105, 50, 50, 1
 next y
next x
```

This simplistic example is how Mauro weaves his magic. You can create any useful graphics you want on one or more hidden graphics pages. Then you can use BLIT to move parts of the image between them, including potentially onto page 0. And, of course, you can mirror or rotate the parts of the image as you move them.

There are two more BLIT commands in the manual: BLIT READ and BLIT WRITE, but we will leave these until we deal with sprites. Refer to the section on FRAMEBUFFER for additional functionality of BLIT.

Next post will look at the various graphics modes supported by the CMM2.

GRAPHICS MODES

This chapter will introduce the 10 graphics resolutions on the CMM2 and the three colour depths in more detail and the use of the MODE command. We will mention the 12-bit modes but will reserve a full explanation of these until a separate tutorial. Also, still to come in this series is more on the PAGE command, using the basic graphics in a more efficient way and, of course, sprites.

A lot of this post will be slightly theoretical, but the more you understand what is going on under the cover of the CMM2 the better you will be able to make the most of its capabilities.

Lets start by being a VGA monitor and asking it to show the Display Information for the signal it is receiving. This isn't affected by the colour depth.

Mode 1 - (800 x 600) : Monitor sees 800x600 @ 60Hz
Mode 2 - (640 x 400) : Monitor sees 640x480 @ 75Hz
Mode 3 - (320 x 200) : Monitor sees 640x480 @ 75Hz
Mode 4 - (480 x 432) : Monitor sees 640x480 @ 75Hz
Mode 5 - (240 x 216) : Monitor sees 640x480 @ 75Hz

Note: 5 new modes were introduced in MMBasic 5.05.05

Mode 6 - (256 x 240) : Monitor sees 640 x 480 @ 75Hz *(for NES compatibility)*
Mode 7 - (320 x 240) : Monitor sees 640 x 480 @ 75Hz
(this give a full screen display with square pixels on a normal aspect ratio monitor. Pages 0-13 are available in this mode with pages 0 and 1 in fast processor memory.)
Mode 8 - (640 x 480) : Monitor sees 640 x 480 @ 75Hz
Mode 9 - (1024 x 768) : Monitor sees 1024 x 768 @ 60Hz *(12-bit colour depth not available in this mode)*
Mode 10 - (848 x 480) : Monitor sees 640 x 480 @ 60Hz *(widescreen format)*

Note: Mode 10 may not display correctly (or at all) on a standard 4:3 monitor.

All modes work perfectly with monitors that have an aspect ratio of 4:3 or widescreen monitors that can switch to that ratio (most widescreen monitorswill do this automatically).

That looks a bit odd - lets explain.

VGA monitors, particularly modern TFT types, have a limited range of signal timings that they will accept and sync up properly. We have chosen to use two that are supported by all monitors – 800 x 600 @ 60Hz using a 40MHz pixel clock and 640 x 480 @ 75Hz using a 31.5MHz pixel clock. All other timings, vertical sync and horizontal sync are defined as a multiple of the underlying clock period. Two additional signal timing settings are used – 640 x 480 @ 60Hz for Mode 10, the 848 x 480 pixel widescreen format for use with 16:9 monitors and 1024 x 768 @ 60Hz for Mode 9, the latter being for the 1024 x 762 pixel mode.

Note: The mode timings above do NOT necessarily indicate the pixel count. See the details above for the actual horizontal and vertical pixel count.

Before going into this in more detail let's answer the question – for modes 2 and 3, why 640x400 and not 640x480?

In the first post we explained the way memory is used for the graphics pages and that 800 x 600 resolution with 1 byte per pixel uses 480,000 out of 512K (512*1024=524288) bytes of the fastest memory. 640 x 480 would only use 307,200 bytes so no problem BUT, 640 x 480 with 16-bit colour would use 614,400 bytes - more than we have available.
640 x 400 x 2 = 512,000 just inside the 512K limit allowing full colour images to be displayed from the fastest memory.

Moreover, the main retro computers Commodore64, Amiga, Atari-ST all used 320x200 as their main graphics resolution, the same as our mode 3 and 640x400 is exactly double this.

But, I hear you say, the manual says the CMM2 supports 800 x 600x16-bit resolution amongst others. Yes, but in this case the first megabyte of the 3 MBytes of SDRAM is used as video display memory - page 0. The SDRAM isn't as fast as the internal memory but in most cases works perfectly well in this mode and it is great for displaying pictures. With a pixel clock at 40MHz and 2 bytes per pixel the LTDC is reading that memory at 80Mbytes/second. Even more extreme is 12-bit mode. Without going into the explanation here 12-bit mode uses 4 bytes per pixel so 800x600x32-bit now needs memory to be read at 160Mbytes/second and uses 2Mbyte of SDRAM!

You will see from the table above that with the introduction of MMBasic version 5.05.05, five more modes were implimented including 640 x 480 as Mode 8 and in order to cater for the size issue raised above, the 12 and 16 bit variants of 640 x 480 use the SDRAM for their video display memory - page 0. This is also true for other modes that exceed the 512K limit of fast memory.

The SDRAM is also used for MMBasic arrays and things like buffers for playing MP3 and FLAC files. If you set the video mode to 800x600 12-bit resolution and play a MP3 file and run a program then occasionally the monitor may flicker or lose sync. The extent to which this happens is very much monitor dependent and while there may be no issue on your monitor there may be for other people.

Back to the main story. Supported resolutions are :-

Mode 1 displays 800x600 pixels and the LTDC just reads out the bytes one at a time at 40Mhz. Bytes are not read during line and frame blanking periods (HSYNC and VSYNC) and the LTDC handles this for us.

Mode 2 displays 640x400 pixels but the monitor sees 640x480. This is achieved by increasing frame blanking by 80 lines split between the top and bottom (front porch and back porch in the jargon). The memory used is 640x400 pixels and this is arranged exactly like we saw for mode 1 earlier, top left to bottom right.

Mode 4 displays 480x432 pixels for Maximite compatibility. This is achieved by reducing the pixel clock to 31.5/640*480 = 23.625MHz but maintaining the same absolute durations for line and frame blanking periods. In this case frame blanking is extended by 48 lines split between the top and bottom. The change in pixel clock rate isn't a perfect solution, particularly with LCD monitors. These work by splitting the period between HSYNC pulses and sampling the analog signal at the expected pixel rate. So they are sampling at 31.5MHz and we are changing values at 23.625Mhz. They then need to map this to the physical screen resolution (say 1280 pixels) so, depending on the monitor, you may see slight artefacts.

Notice how the left side of the "0" in "11-05" and "2020" is wider than the right. This is a monitor sampling issue and probably wouldn't occur on an old analogue CRT monitor. The memory used is 480x432 pixels and this is arranged exactly like we saw for mode 1 earlier, top left to bottom right.

Mode 3 displays 320x200 pixels. You can probably easily guess now how the 320 is created. The pixel clock is halved from 31.5MHz to 15.75MHz. As this is a simple power of 2 there is no sampling issue like we saw with mode 4. However the monitor is still expecting 480 lines and that can't be changed. Unfortunately the LTDC doesn't have the capability to clock out each line of data from memory twice so we have had to replicate each line in the firmware. The memory used is therefore only half that of 640x400 rather than a quarter. As for mode 2 frame blanking is increased by 80 lines split between the top and bottom. The memory used is 320x400 pixels and to get the address of a pixel in memory we now have to compensate for each line being duplicated. e.g. for an 8-bit colour depth:

```
add1% =MM.INFO(page address 0)+ (y * 2) * MM.HRES + x
add2% =MM.INFO(page address 0)+ (y * 2 + 1) * MM.HRES + x
```

Mode 5 displays 240x216 pixels and I'll leave to the reader to extrapolate from modes 3 and 4 as to how this works.

Mode 6 displays 256 x 240 pixels and is the format used by Nintendo Entertainment Systems (NES). This mode was introduced to allow games programmers to attempt to emulate some of the great NES games. For a detailed explanation of the NES format, see NES explanation . The monitor still sees 640 x 480 at 75Hz however, similar to mode 3, the pixel clock is reduced, this time by 2.5 times to 12.5MHz with absolute line and frame durations the same. As with mode 3, each line of data is clocked out from memory twice by the firmware.

Mode 7 displays 320 by 240 pixels – exactly the same timing as mode 3 but without sacrificing the 80 lines. Also, the same constraints on memory usage as mode 3 apply. This, together with mode 8 have the correct 4:3 screen ratio and display nice clean square dots.

Mode 8 displays 640 by 480 pixels.

Mode 9 displays 1024 by 768 pixels with a pixel clock rate of 62.5MHz. The data rate required to keep up limits this mode to 8 and 16 bit colour only (12 bit mode requires 4 bytes per pixel – even the very nippy CMM2 can’t do this and everything else at once!!)

Mode 10 displays 848 x 480 pixels and is intended for use on widescreen displays. This mode should not be used for programs to be distributed as not all users may be able to show it, but it is excellent for editing when you have a widescreen monitor.

Supported colour depths

The CMM2 supports 3 colour depths:-

8-bit uses 1 byte per pixel and uses RGB332. That means there are three bits coding the red intensity (8 levels), three for green (8 levels) and two for blue (4 levels). However, as we saw earlier we can change this using the MAP command and function so, for example, we could create a grey-scale map with 256 different "grey" levels.

16-bit uses 2 bytes per pixel and uses RGB565. That means there are five bits coding the red intensity (32 levels), six for green (64 levels) and five for blue (32 levels). Although this may seem limited, colour images displayed in RGB565 on a normal monitor are pretty much indistinguishable from those with greater colour depth.

12-bit uses 4 bytes per pixel and uses ARGB4444. That means there are four bits coding the red intensity (16 levels), four for green (16 levels) and four for blue (16 levels). In addition there are 4-bits that encode the transparency. i.e. the extent to which the colour masks anything underneath it. Think about paint. If you paint white on top of red you will still see a pink blush after the first coat. Although the white paint is "pure" white it allows some of the red to be seen through it. In the case of ARGB4444 the 4 "A" bits range from 15=solid to 0=totally transparent.

Now, transparency isn't much use if there is nothing to be transparent over. So 12-bit mode has two layers and hence it takes 4 bytes per pixel - more on this later.

In all cases the firmware translates from the colour specification you provide which is in RGB888 or for 12-bit mode ARGB8888 to the internal format for the display colour depth selected. This means the Basic code is identical irrespective of colour depth (except if you use the MAP command in 8-bit mode or are using Maximite compatibility mode).

The MODE command

The explanation of the mode command in the user manual is very comprehensive so there is little point in reproducing it again here. However, of particular importance is the number of video pages that are available in each mode.

The firmware rounds up the amount of video memory used for any resolution/colour depth to the nearest multiple of 128Kbytes (128*1024=131072). So Mode 3,8 (320 x 200 x 8-bit) seems like it should need 64000 bytes but remember each line must be duplicated so it really needs 128000 which is less than 131072. This means we can have 4 video pages in the fast internal memory (512K available) and another 24 in the 3Mbytes of SDRAM.

New in version 5.05.05 is OPTION DEFAULT MODE n where n can be 1, 8, or 9. This permanent option sets the video mode for the command prompt, the editor and the filemanager and is the default mode when a program is run.

Remember you can also use OPTION EDIT FONT to adjust the font used in the filemanager and the editor. This should allow the CMM2 to use the cheap 7" TFT screens like this.

This is a permanently saved option but does not cause a reboot so can be used in a program. However, developers should probably not use it in code to be circulated as you will reconfigure a user's CMM2.

OPTION DEFAULT MODE n will always give you the 8 bit colour depth.

The specifications of the colour depth (‘bits’) are:-

	16-bit	**12-bit**	**8-bit**
H/W Pixel Format	RGB565	ARGB4444	RGB332
Colours	65536	4096	256
Transparency	None	16-levels	None
Pages Used	1	2	1
Layers	1	2 + background	1

The display always shows the contents of page 0 (16-bit and 8-bit) and pages 0 and 1 (12-bit). Use PAGE WRITE and PAGE COPY to avoid flashing and tearing artefacts.

For 12-bit colour depth page 0 is the lower level and page 1 the upper so the stack is: background, page 0, page 1 with each one overwriting the previous in turn as defined by the transparency values of each individual pixel.

A summary of the pages available in the different modes is as follows;-

MODE	BIT DEPTH	RESOLUTION	PAGE MEMORY	TOTAL PAGES	FAST PAGES	EXT PAGES
1	8	800 x 600	480000	7	&H24000000	&HD0000000, $HD0080000, &HD0100000, &HD0180000, $HD0200000, &HD0280000
1	12	800 x 600		3	-	&HD0000000, $HD0100000, &HD0200000
1	16	800 x 600	960000	3	-	&HD0000000, $HD0100000, &HD0200000
2	8	640 x 400	256000	14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
2	12	640 x 400		7	&H24000000	&HD0000000, $HD0080000, &HD0100000, &HD0180000, &HD0200000, &HD0280000
2	16	640 x 400	512000	7	&H24000000	&HD0000000, $HD0080000, &HD0100000, &HD0180000, &HD0200000, &HD0280000
3	8	320 x 200	128000 **	28	&H24000000, &H24020000, &H24040000, &H24060000	&HD0000000, $HD0020000, &HD0040000, &HD0060000, &HD0080000, &HD00A0000 &HD00C0000, $HD00E0000, &HD0100000, &HD0120000, &HD0140000, &HD0160000 &HD0180000, $HD01A0000, &HD01C0000, &HD01E0000, &HD0200000, &HD0220000 &HD0240000, $HD0260000, &HD0280000, &HD02A0000, &HD02C0000, &HD02E0000
3	12	320 x 200		14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
3	16	320 x 200	256000 **	14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
4	8	480 x 432	207360	14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
4	12	480 x 432		7	&H24000000	&HD0000000, $HD0080000, &HD0100000, &HD0180000, $HD0200000, &HD0280000
4	16	480 x 432	414720	7	&H24000000	&HD0000000, $HD0080000, &HD0100000, &HD0180000, $HD0200000, &HD0280000
5	8	240 x 216	103680 **	28	&H24000000, &H24020000, &H24040000, &H24060000	&HD0000000, $HD0020000, &HD0040000, &HD0060000, &HD0080000, &HD00A0000 &HD00C0000, $HD00E0000, &HD0100000, &HD0120000, &HD0140000, &HD0160000 &HD0180000, $HD01A0000, &HD01C0000, &HD01E0000, &HD0200000, &HD0220000 &HD0240000, $HD0260000, &HD0280000, &HD02A0000, &HD02C0000, &HD02E0000
5	12	240 x 216		14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
5	16	240 x 216	207360 **	14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
6	8	256 x 240	122880 **	28	&H24000000, &H24020000, &H24040000, &H24060000	&HD0000000, $HD0020000, &HD0040000, &HD0060000, &HD0080000, &HD00A0000 &HD00C0000, $HD00E0000, &HD0100000, &HD0120000, &HD0140000, &HD0160000 &HD0180000, $HD01A0000, &HD01C0000, &HD01E0000, &HD0200000, &HD0220000 &HD0240000, $HD0260000, &HD0280000, &HD02A0000, &HD02C0000, &HD02E0000
6	12	256 x 240		14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
6	16	256 x 240	245760 **	14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
7	8	320 x 240	153600 **	14	&H24000000, &H24040000	&HD0000000, $HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000 &HD0180000, $HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
7	12	320 x 240		9	&H24000000	&HD0000000, $HD0060000, &HD00C0000, &HD0120000, &HD0180000, &HD01E0000, &HD0240000, $HD02A0000
7	16	320 x 240	307200 **	9	&H24000000	&HD0000000, $HD0060000, &HD00C0000, &HD0120000, &HD0180000, &HD01E0000, &HD0240000, $HD02A0000
8	8	640 x 480	307200	9	&H24000000	&HD0000000, $HD0060000, &HD00C0000, &HD0120000, &HD0180000, &HD01E0000, &HD0240000, $HD02A0000
8	12	640 x 480		5	-	&HD0000000, $HD00A0000, &HD0140000, &HD01E0000, &HD0280000
8	16	640 x 480	614400	5	-	&HD0000000, $HD00A0000, &HD0140000, &HD01E0000, &HD0280000
9	8	1024 x 768	786432	4	-	&HD0000000, $HD00C0000, &HD0180000, &HD0240000
9	12	N/A	N/A		N/A	N/A
9	16	1024 x 768	1572864	2	-	&HD0000000, $HD0180000
10	8	848 x 480	407040	7	&H24000000	&HD0000000, $HD0080000, &HD0100000, &HD0180000, $HD0200000, &HD0280000
10	12	848 x 480		4	-	&HD0000000, $HD00E0000, &HD01C0000, &HD02A0000
10	16	848 x 480	814080	4	-	&HD0000000, $HD00E0000, &HD01C0000, &HD02A0000

Note: Page Memory values with ** after them indicate that lines are duplicated meaning twice as much memory is used as would be expected. See explanation on previous page..

As before, MM.INFO(PAGE ADDRESS n) can give us the address of any of the pages. Page 0 is **always** the display page but when programming for performance there is an advantage to using a page in the faster memory for other uses.

The syntax of the mode command is:

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

We now know all about the mode and the bits and what the firmware is doing behind the scenes to create the images.

We will deal with bg which is only used in 12-bit mode later but for now we can look at the "int" parameter.

int is the name of an MMBasic interrupt routine which is called immediately after the last active row of data is clocked out. This is before the actual frame-blanking pulse (actually it occurs at the beginning of the front-porch period which bizarrely is at the end of the frame whereas the back porch is at the beginning - WTF?).

This is important to understand because before any new data is read from the memory the front porch period, the actual VSync frame pulse period and the back porch period will all have elapsed. In CMM2 terms this can be quite a long time.

For Modes 2 and 3, there are 400 lines displayed and the complete frame has 500 line periods so there is a dead time of 2.66 milliseconds when the display can be written and no screen artefacts will ever appear.

For Mode 1 this period is 0.74mSec and for modes 4 and 5 it is 1.65mSec

You can actually effectively extend this period if you arrange your code to write from the top of the screen downwards. As long as your writes keep ahead of the LTDC reading the memory then there will be no artefacts or tearing.

Lets build a very simple digital clock with big digits using the frame interrupt and show that there are never any screen artefacts

```
' set to 320x200 mode, with a frame interrupt
' NB the background colour must be specified but is ignored in this mode
mode 3, 8, 0, frameint
' set a global variable that will be updated by the frame interrupt
fi%=0
do
  do
  loop while fi%=0 'wait until the frame interrupt has happened
  ' write out the time in the middle of the screen
  text mm.hres\2, mm.vres\2, time$, CM, 6,, RGB(red)
  ' prime ready for the next interrupt
  fi%=0
loop

sub frameint
  fi%=1
end sub
```

Note that I am not doing any screen updates in the interrupt routine - you wouldn't do that would you? Interrupts must always be kept short and the best approach is always to set a flag that can be processed in the main program.

A program created by TassyJim from TheBackShed to demonstrate all the different MODEs is shown below.

```
' test card for CMM2
 ' TassyJim August 2020

 OPTION EXPLICIT
 OPTION DEFAULT NONE
 DIM INTEGER wd, ht, wbox, sh, x, w, n, nn, m, cd, maxMode, keepMode
 DIM FLOAT a, defaultMode
 DIM k$, imgtitle$, fname$, imgRes$
 DIM INTEGER c(8)
 c(0) = RGB(BLACK)
 c(1) = RGB(YELLOW)
 c(2) = RGB(CYAN)
 c(3) = RGB(GREEN)
 c(4) = RGB(MAGENTA)
 c(5) = RGB(RED)
 c(6) = RGB(BLUE)
 c(7) = RGB(WHITE)
 c(8) = RGB(64,64,64)

 maxMode = 9
 cd = 8
 a = 1
 defaultMode = MM.INFO(MODE)
 CLS

 DO
   IF m = 0 THEN
     MODE 1,8
     CLS
     TEXT 400,100, "Video mode test",cm,5,1
     TEXT 400,180, "Ratio = aspect ratio used in the circle command",cm,1,1
     TEXT 400,220, "Q to quit, P to save page as a BMP",cm,3,1
     TEXT 400,260, "Up Down arrow to change resolution",cm,3,1
     TEXT 400,300, "Left Right arrow to change colour depth",cm,3,1
     TEXT 400,340, "+ - to change circle aspect ratio",cm,3,1
   ELSE
     IF keepmode THEN ' only change resolution if needed
       keepmode = 0
     ELSE
       MODE m,cd
     ENDIF
     wd = MM.HRES : ht = MM.VRES
     nn = INT(wd/80)
     imgtitle$ =" MODE "+STR$(MM.INFO(MODE))+" Ratio "+STR$(a,1,3)+" "
     imgRes$ = " "+STR$(MM.HRES)+" x "+STR$(MM.VRES)+" "
     wbox = wd / 8
     PAGE WRITE 1 ' no flicker during write
     CLS
     ' primary colours
     FOR x = 0 TO 7
       BOX x*wbox,ht/4,wbox,ht/2,0,c(x), c(x)
     NEXT x
     ' full gradient for each primary colour and greyscale
     FOR x = 0 TO wd-1
       sh = 255*x/wd

       LINE x,0,x,ht/12,1,RGB(sh,0,0)
       LINE x,ht/12,x,ht/6,1,RGB(0,sh,0)
       LINE x,ht/6,x,ht/4,1,RGB(0,0,sh)

       LINE x,ht*9/12,x,ht*10/12,1,RGB(0,sh,sh)
       LINE x,ht*10/12,x,ht*11/12,1,RGB(sh,0,sh)
       LINE x,ht*11/12,x,ht,1,RGB(sh,sh,0)

       LINE x,ht/2,x,ht*3/4,1,RGB(sh,sh,sh) ' greyscale
     NEXT x
     ' circle to check aspect ratio
     CIRCLE wd/2,ht/2, ht*15/32,3,a,c(7)
     sh = 0
     x = wd/2 - 55*nn/2
     ' black white bars to check monitor bandwidth
     FOR w = 10 TO 1 STEP -1
       FOR n = 1 TO nn
         sh = 255 - sh
         LINE x,ht*3/8,x,ht*5/8,w,RGB(sh,sh,sh)
```

```
        x = x + w
      NEXT n
    NEXT w
    ' white and red border to check that image fits on monitor
    BOX 0,0,wd,ht,3,c(7)
    BOX 1,1,wd-2,ht-2,1,c(5)
    ' title
    IF wd > 600 THEN
      TEXT wd/2,ht/2, imgtitle$,cm,4,1
      TEXT wd/2,ht/2+25, imgRes$,cm,4,1
    ELSE
      TEXT wd/2,ht/2, imgtitle$,cm,1,1
      TEXT wd/2,ht/2+25, imgRes$,cm,1,1
    ENDIF
    ' show the new image
    PAGE COPY 1 TO 0 ,B
  ENDIF
  ' wait for keypress
  DO
    k$ = INKEY$
  LOOP UNTIL k$<>""
  '
  SELECT CASE k$
    CASE "Q","q"
      EXIT DO
    CASE "P","p"
      fname$ = MID$(imgtitle$,2)+".bmp"
      TIMER = 0
      SAVE IMAGE fname$
      PAGE WRITE 0
      TEXT wd/2,ht/2,"Saved as "+fname$+" in "+STR$(TIMER/1000,3,2)+" Sec" ,cm,1,1
      DO
        k$ = INKEY$
      LOOP UNTIL k$<>""
    CASE CHR$(128) ' up arrow
      m = m - 1
      IF m < 1 THEN m = maxMode
    CASE CHR$(129) ' down arrow
      m = m + 1
      IF m > maxMode THEN m = 1
    CASE CHR$(131) ' right arrow res up
      cd = cd + 4
      IF cd > 16 THEN cd = 8
      IF m = 9 AND cd = 12 THEN cd = 16 ' skip 12 bit for mode 9
    CASE CHR$(130) ' left arrow  res down
      cd = cd - 4
      IF cd < 8 THEN cd = 16
      IF m = 9 AND cd = 12 THEN cd = 8 ' skip 12 bit for mode 9
    CASE "+" ' ratio plus
      IF a < 1.4 THEN a = a + 0.01
      keepmode = 1
    CASE "-" ' ratio minus
      IF a > 0.75 THEN a = a - 0.01
      keepmode = 1
    CASE ELSE ' same as down arrow
      m = m + 1
      IF m > maxMode THEN m = 1
  END SELECT
LOOP
setmode defaultMode ' restore original mode before ending program
PAGE WRITE 0
CLS
END

SUB setmode dotMode AS FLOAT
 LOCAL INTEGER mm, md
 ' use float returned by mm.info(mode) to set MODE
 mm = INT(dotmode)
 md = (dotmode - mm)*100
 IF md > 20 THEN md = md/10
 MODE mm, md
END SUB
```

THE PAGE COMMAND

In the second post in this thread we met the PAGE COPY command and now understand how to move information from one page to another. This can be the complete page using PAGE COPY or part of a page using BLIT.

We have also investigated the PAGE WRITE command that allows us to select the page where graphics commands will write their data.

In this post we will look at the two other PAGE sub-commands - PAGE SCROLL and PAGE STITCH

PAGE SCROLL

This command does what it says. It scrolls a selectable page (doesn't need to be page 0) horizontally and/or vertically by a selectable amount. In addition it allows you to control what happens to the area vacated by the action of the scroll.

```
PAGE SCROLL pageno, x, y [,fillcolour]
```

Some examples:

```
LOAD JPG "bondi":DO:LOOP
```

load a sample jpg in 800x600x16 resolution

Now lets try something else:-

```
LOAD JPG "bondi":PAGE SCROLL 0, 100,100:DO:LOOP
```

scroll the picture 100 right and 100 up, moving the scrolled off area to the now vacated part of the screen. Note that the vertical move always happens first. If you need the sideways move first you will need to use the command twice.

```
LOAD JPG "bondi":PAGE SCROLL 0, 100,100,-1:DO:LOOP
```

This will scroll the picture 100 up and 100 right, leave the vacated off area as-is

And again:-

```
LOAD JPG "bondi":PAGE SCROLL 0, 100,100,RGB(RED):DO:LOOP
```

scroll the picture 100 up and 100 right, replace the vacated area with a fixed colour

Time to load a 800x600 full colour jpg was 347 milliseconds. Worst case time to scroll the image with image wrap was 125 milliseconds. All other lower resolutions or lower colour depths will be faster.

Of course moving a 800x600 image with 2 bytes per pixel takes a little time but try this example program at 320x200x16

```
option explicit
option default none
dim x%(9),xd%(9)
dim y%(9),yd%(9)
dim integer i,c,f, xp, yp,k,t
dim float s

' Work in 320x200 resolution RGB565
mode 3,16

cls

'create the coordinates of a star outline
for i=0 to 9 step 2
 x%(i)=-(SIN(rad(i*36))*20)
 y%(i)=-(COS(rad(i*36))*20)
 x%(i+1)=-(SIN(rad((i+1)*36))*7.6)
 y%(i+1)=-(COS(rad((i+1)*36))*7.6)
next i

'Set to write to page 1 and clear it
page write 1
cls

'create 40 random stars on page 1 using the polygon fill command
do
 c=rnd()*255 + ((rnd()* 255)<<8) + ((rnd()* 255)<<16)
 f=rnd()*255 + ((rnd()* 255)<<8) + ((rnd()* 255)<<16)
 xp=rnd()*mm.hres
 yp=rnd()*mm.vres
 s=rnd()
 t=0
 for i=0 to 9
   xd%(i)=x%(i)*s+xp
```

```
   yd%(i)=y%(i)*s+yp
   if xd%(i)<0 or yd%(i)<0 or xd%(i)>=MM.Hres or yd%(i)>=MM.Vres then t=1
 next i
 if t=0 then
   polygon 10, xd%(), yd%(), c, f
   k=k+1
 endif
loop until k=40
k=0
'

'Now lets see how fast scroll really is
do
 page copy 1 to 0,d 'copy page 1 to page 0 during frame blanking
 page scroll 1,2,1 'scroll page 1 immediately after
loop
```

For many simple early games they used a seamless background that was a single image that could be scrolled horizontally, vertically, or even both with no obvious join.

Try the attached with

```
load jpg "seamless":do: page scroll 0,1,1:loop
```

seamless.zip

Now you should be able to see how to create a slideshow with one picture smoothly replacing another from the right.

I'll provide some pseudo code for you to complete and try for yourself.

```
load an image to page 1
load an image to page 2
loop
   copy page 1 to page 0 during frame blanking
   scroll page 1 left by a "n" pixels
   BLIT n pixels from the left of page 2 to the right of page 1
   scroll page 2 left by "n" pixels
   If you have just used all of page 2 then load the next image to page 2
end loop
```

PAGE STITCH

This is really only a short form way of doing what was proposed above with scroll and BLIT

```
PAGE STITCH frompage1, frompage_2, topage, offset
```

This says we are going to take columns from the right side of frompage1 and columns from the left side of frompage2 and copy them to topage as a single action. The offset parameter determines how many columns are going to be taken from frompage2 - perhaps the offset parameter was badly named?

This single command makes it very easy to create a horizontally scrolling background across a number of separate images.

This video shows how fast this can take place.

The example code presented below slows this down by syncing everything up with frame blanking to get a very smoothly moving image. As the monitor frame rate is 75Hz and the scroll is happening 2 pixels at a time the movement is limited to 150 pixels per second. Remove the ",b" parameter in the page copies to see things at maximum speed.

```
' test stitch
' Run at 320x200x16 bit mode
mode 3,16

'clear the first 7 video pages
For i= 0 to 6: page write i:cls:next i

'now write each part of the background image to a separate page
' doing this at the beginning ensures the program won't stall waiting for images to be loaded
page write 2
load png "part01",,,15
page write 3
load png "part02",,,15
page write 4
load png "part03",,,15
page write 5
load png "part04",,,15

' Main program loop
do
```

```
'merge pages 2 and 3 stepping 2 pixels at a time
 for i=0 to MM.Hres step 2
   page stitch 2,3,6,i
   page copy 6 to 0,b
 next i

' now page 2 is used up so merge pages 3 and 4
 for i=2 to MM.Hres step 2
   page stitch 3,4,6,i
   page copy 6 to 0,b
 next i

' finally merge pages 4 and 5
 for i=2 to MM.Hres step 2
   page stitch 4,5,6,i
   page copy 6 to 0,b
 next i

'we are now at the rightmost side of the composite image so we can start moving back left
 for i= MM.Hres to 0 step -2
   page stitch 4,5,6,i
   page copy 6 to 0,b
 next i
 for i= MM.Hres-2 to 0 step -2
   page stitch 3,4,6,i
   page copy 6 to 0,b
 next i
 for i= MM.Hres-2 to 0 step -2
   page stitch 2,3,6,i
   page copy 6 to 0,b
 next i

' Once all the way left then start again
loop
```

Here are the images used in this demo **Parts.zip**

Hopefully you can see from this short post that it is very easy to create and manipulate moving images on the screen. Of course, this needn't be anything to do with games. Scrolling graphs etc. are just as easy.

12-bit Mode Graphics

The 12-bit graphics modes are one of the most powerful capabilities of the Colour Maximite 2. They enable us to create semi-transparent objects floating over a background like the **ghost demo** or the effect of parallax like the **stars demo.** Note in the latter how the nearer stars are moving faster relative to the spaceship than those further away.

The way that it works is using layers. The concept will be very familiar to those of you doing manipulations of images with any of the more capable graphics packages.

In 12-bit colour mode the CMM2 supports three layers. This is a property of the processor chip itself.

The bottom layer is just a single colour background and to set the colour we specify it in the MODE command.

```
MODE 3,12,RGB(blue)
```

Will put a blue over the screen. Note how, as we are at the command prompt, the status line and command prompt overwrite the background, but it shows through the unset pixels in each character.

To change the background colour you can just use the mode command again with the new colour. Other information on the screen won't be affected (NB: V5.05.03 or greater).

The status line and command prompt are writing to the second layer of the three layers and the data for this layer is stored in page 0.

The data for the third and top layer is stored in page 1. Unless pixels are set to have some transparency (more on this later) any information written to layer 3 (page 1) will block out layers 1 and 2.

The 'layers' referred to above are NOT sprite layers - see further on how this applies to sprites.

We can prove this. Type:

```
PAGE WRITE 1: TEXT 247,190,"X":page write 0
```

We have to do the command as a single line to avoid the status updating while we are writing to page 1

Note how the "X" appears to have overwritten the brown seconds units but the seconds are still incrementing underneath and pixels that are not covered by the "X" show normally.

Now type:

```
PAGE WRITE 1: TEXT 247,190," ":page write 0
```

The "X" disappears and the seconds continue incrementing normally. Black, ie. the pixels in text that aren't set, defaults to being a completely transparent colour.

Now type

```
CLS RGB(black,15)
```

Check out the RGB function in the user manual. We have asked the system to clear the screen but to use a non-transparent black. Valid transparency values are from 0 to 15 where 15 is a solid colour. However, the default background colour is fully transparent black so the characters associated with the command prompt and the status line still show the blue colour.

Now try:

```
COLOUR RGB(WHITE), RGB(BLACK,15)
```

We have set the background colour to solid black rather than transparent black and now the blue colour on layer 1 has gone completely (NB: V5.05.03 or greater).

Lets play a bit more with transparency

```
colour rgb(white),rgb(black,15)
box 200,100,50,50,,rgb(blue),rgb(blue)
page write 1
box 225,100,50,50,,rgb(green,8),rgb(green,8)
```

The blue square on layer 2 (PAGE 0) is partially overwritten by the green square on layer 3 (PAGE 1) which is set to 50% transparency. Where the squares overlap we get CYAN.

We could do the same thing with layer 2 and the background layer 1

That should give you a good understanding of 12-bit mode (ARGB4444) and how the "A" transparency can be used to create interesting effects. The most important thing to note is that in 12-bit mode the CMM2 is displaying information from two pages at the same time (PAGE 0 and PAGE 1) plus a fixed background colour. Information on PAGE 0 will overwrite the background depending on the transparency of each individual pixel on PAGE 0. A transparency of 0 will have no effect. A transparency of 15 is a solid colour and will overwrite completely. Information on PAGE 1 will overwrite both the background and PAGE 0 depending on the transparency of each individual pixel.

We will revisit this with a more interesting example when we look at sprites.

However, the next post will look in more detail at loading and manipulating pictures.

Editors Note: Peter has posted some additional information on PAGEs and layers included here as follows:-

From Peter - *"There is clearly confusion here so I'll have another go at explaining."*

Pages are areas of memory that are sized as defined by the resolution. So each page for mode 1,8 is an area of memory 800 x 600 x one byte. Mode 1,16 is 800 x 600 x two bytes.

In all modes, PAGE 0 is is shown on the display. In 12-bit modes PAGE 1 sits on top of PAGE 0 and is also displayed in which case PAGE 0 pixels show through to the extent that PAGE 1 pixels have some level of transparency (see the diagram on page 26 of this document.)

Use MM.INFO(MAX PAGES) to tell you the highest page number you can use for any resolution. There is 3.5Mbytes of RAM allocated to video pages and this is allocated in units of 128KBytes.

PAGE COPY simply takes the memory contents of a page and copies it to another page

All graphics output (including sprite commands) goes to the page set by the most recently executed PAGE WRITE command except for a small number of commands that explicitly allow you to define it (e.g. PAGE SCROLL)

When multiple pages are in use it is essential that all sprite commands that change the page act on the same page. For reasons that will become clear it makes sense to allocate a page other than 0 or 1 for sprite usage. I suggest you standardise on PAGE 2. This means before any sprite write operation you should set PAGE WRITE 2.

Sprites are entirely implemented in code. There is no supporting hardware. To move a sprite the firmware first redraws the stored background of the page where the sprite is drawn, then stores the background of where it will move to, then draws the sprite. If the sprites are drawn on PAGE 0 then unless you use a technique to control when you move a sprite relative to frame blanking (there are two ways of doing this documented in the manual) you will see flashing which is why it is better to use sprites on a non-displayed page.

*** So, a few rules to follow! ***

It is poor programming to show a sprite repeatedly in your process loop unless something has changed (position, orientation, sprite image changed by a READ).

Don't use PAGE SCROLL on the page that contains sprites. You must use the SPRITE SCROLL command as this understands sprites. It is however less efficient and a PAGE COPY, then PAGE SCROLL may be better.

Sprite layers have nothing to do with pages.

Sprite layers are used in just three ways:

1. Sprites on sprite layer 0 will move with the background of the page on which they are drawn when the SPRITE SCROLL command is used
2. A sprite move will only trigger a collision with a sprite on the same sprite layer or sprite layer 0
3. A sprite scroll can trigger a collision between a layer 0 sprite and a sprite on another layer
4. If you want to hide a sprite, it is down to you to know if there are any sprites that overlap it and have been drawn after it, in which case you must hide them first then redraw them after hiding the intended sprite.

So a typical algorithm for a sprite "killing" another is as follows:

```
sprite show weaponsprite, x,y,n 'this move triggers a collision

sub sprite interrupt
 local killedsprite=sprite(C,S,1)
 sprite hide weaponsprite
 sprite hide killedsprite
 sprite show weaponsprite, x, y, n
end sub
```

IMAGES

In this post we will look at the pros and cons of the image formats supported by the CMM2: JPG, GIF, BMP and PNG.

We will start by putting the CMM2 into MODE 2,16 ie. 640x400 RGB565 and for testing purposes I have saved the same image in all 4 formats.

(Editors Note: The images shown are my own as I did not have access to those used in the original posts. The code examples however, refer to the original images. You could substitute your own images appropriately sized.)

Editors Note: File sizes and load times for the "caja" picture are from the original post as given below. I did not have access to the image file Peter used.

```
> list files "caja.*"
A:/
18:50 14-05-2020     768054  caja.bmp
18:50 14-05-2020     217064  caja.gif
18:49 14-05-2020     233823  caja.jpg
18:51 14-05-2020     506373  caja.png
0 directories, 4 files
>
```

And times to load in milliseconds

```
> mode 2,16
> timer=0:load bmp "caja":?timer
541.699
> timer=0:load gif "caja":?timer
728.09
> timer=0:load jpg "caja":?timer
216.345
> timer=0:load png "caja":?timer
509.873
```

Summarising the different file types then:-

JPG files - JPGs are small and load much faster than anything else.

The reasons for this are twofold; first there is less data to read from the SDcard, second the STM32H743 includes a hardware JPG decoder so the CPU isn't used.

So we use JPG s for everything? perhaps not.....

There are two main downsides with the JPG format on the CMM2:-

1. JPG s do not support transparency, and
2. JPG s cannot be bigger than the current display resolution.

If we tried to load an 800x600 JPG while in MODE 2 we would get an error. This is because the hardware decoder uses information from the way we have set up the page in LTDC to determine how to decode the image. We would also get an error even with a 640x400 JPG if we tried to load it with an x, y offset other than 0.

```
LOAD JPG file$ [, x, y]
```

One third limitation of the JPG support on the CMM2 is that it does not support progressive encoding. This is a technique that was developed for web use where a picture would appear fast but with low resolution and then the detail would "progressively" appear. This limitation is easily overcome by using any decent graphics program on your computer to save the image normally.

What about PNG files? - The only format to support transparency so the only choice in some applications

Reasonably fast to load but:-

1. Decoding PNG files eats memory.
   Even with 5MB available we cannot decode a 800x600 image so 640x400 is the biggest that can be practically supported;
2. PNG cannot be bigger than the current display resolution.
   If we tried to load a 640x400 JPG while in MODE 3 we would get an error.
   We would also get an error even with a 320x200 PNG in MODE 3 if we tried to load it with an x,y offset other than 0, and
3. The CMM2 only supports RGB888 and ARGB8888 formats - files are comparatively large

Next up, GIF files - The only format to support animations
GIF files can be bigger than the screen format or overlap the screen edge - GIF files are small but:-

1. slow, and
2. Only 256 colours used out of the palette of 65536

Finally BMP files - The safe option

BMP files image can be bigger than the screen format or overlap the screen edge

```
SAVE IMAGE 'saves BMP files
```

No practical limit on the image size, but of course only part of a very large image can be displayed, but:

full-colour BMP files are 3 bytes per pixel + header ie. large

A BMP image loads a line at a time from the bottom up which may not be what you want.

I'll add to this post tomorrow and look at IMAGE RESIZE and IMAGE ROTATE

I'll just finish off the previous post as promised.

IMAGE RESIZE and IMAGE ROTATE use some fairly complex math to scale and rotate images. They are therefore comparatively slow compared to actions like page copies but useful nevertheless. An example of rotation is available **here.** In fact this was done on an early release of the code and various enhancements have speeded it up somewhat since then.

As an example of their use lets take an 800x600 image that is in landscape and scale and rotate it to a 600x450 image in portrait. This then gives us the worst case timings for the two functions - biggest image, highest colour depth.

tigerside800.zip

```
'Set to 800x600 16 bit mode
' NB to use this mode it must be unlocked first with OPTION MODES UNLOCKED
mode 1,16

'set output to page 1
page write 1

' lets see how long it all takes
timer=0

'load our 800x600 image to page 1
load jpg "tigerside800"
page write 0
print @(0,0)"Time to load JPG"
print timer/1000, "seconds"
timer=0
page write 1

'When we rotate the image it can't be more than 600 pixels high so resize it
'Check the manual for syntax but we are specifying to resize it and place the resized image
centrally over the old image
image resize 0,0,800,600,100,75,600,450
page write 0
print "Time to resize"
print timer/1000, "seconds"
timer=0
page write 1

'Now we need to rotate the image
'In the case of the example this is a rotation of 270 degrees clockwise to get it the correct way up
' Note that the rotated image has to fit into a new area no bigger than the area to be rotated
'So we need to rotate a square area as big as the biggest dimension we need in the result
image rotate 100,0,600,600,100,0,270
page write 0
print "Time to rotate"
print timer/1000, "seconds"
timer=0

'Now we can go back to page 0
page write 0

' Copy the part of page 1 containing the rotated image to page 0 using BLIT
blit 175,0,175,0,450,600,1

print "Time to copy"
print timer/1000, "seconds"
do
loop
```

One important thing to note is that resizing and rotating by definition lose some information and accuracy. For this reason if you want to do something like the rotating image in the video you should do each rotation from the original rather than incrementally rotating the previous result otherwise the image will quickly degenerate.

Two variations on image rotate and resize are IMAGE ROTATE_FAST and IMAGE_RESIZE_FAST.

These commands have exactly the same syntax as the originals but use a nearest neighbour algorithm rather than bi-linear interpolation. As a result they are much faster (6x for rotate and 20x for resize) but, of course, the resulting image quality will not be as good. Additionally, IMAGE ROTATE_FAST and IMAGE_RESIZE_FAST both have a new optional parameter - dontcopyblack

```
IMAGE RESIZE x, y, width, height, new_x, new_y, new_width, new_height [,page_number]
[,dontcopyblack]

IMAGE ROTATE x, y, width, height, new_x, new_y, angle! [,page_number] [,dontcopyblack]
```

This option, dontcopyblack defaults to 0, i.e. black pixels are treated like any other. If it is set to 1 then any black pixels in the created image will not be written.

Next post we start on sprites by learning what is a sprite and the various ways to create them.

SPRITES AND HOW TO CREATE THEM

Sprites are somewhat complex and in any case they are just something reserved for games programmers. Wrong on both counts as we will see.

The concept of a sprite is simple. It is an image stored in the computer memory. It can be written to the screen like any other image. However, what makes a sprite special is that any decent implementation of sprites will have the capability of storing what is on the screen in the area that the sprite will be displayed and this happens automatically without additional code from the user of the sprite.

What this means is that it is possible to write code that displays a sprite and then remove it and the display will be restored exactly like it was before. Perhaps more usefully you can display a sprite and then repeatedly move it on the screen and with each move the original screen image will be restored to create the illusion of the sprite moving across and in front of the original screen image. We will show simple examples of this later in the next post.

Before proceeding to load some sprites, it is important to understand how sprites are stored in the CMM2 memory. Each sprite when created has two areas of memory allocated to it. Both are the size needed to store a bounding rectangle with width and height equal to the maximum dimensions of the sprite. One of the memory areas holds the sprite image itself. The other is an empty slot ready to hold the image that was on the screen when the sprite is displayed. In both cases the number of bytes is also dependent on the current colour depth of the video mode in use - either one or two bytes per pixel.

The CMM2 supports up to 64 sprites numbered 1 to 64 and the number is used whenever we want to refer to a particular sprite. Setting up constants at the top of a program would be a good way to make your code more readable when using sprites

```
CONST Ogre = 1
CONST red_elf = 2
```

Now we know what a sprite is we need to know how to create one. On the CMM2 there are four methods of creating a sprite:-

1. Load a sprite from an ascii file in the same way as was done on the original Colour Maximite;
2. Load a sprite from a PNG (Portable Network Graphics) format file;
3. Create a sprite by reading from what is already displayed on one of the CMM2 video pages; or
4. Copy an existing sprite to create a new one.

We will look at each of these in turn:

SPRITE LOAD

Load a sprite from an ascii file.

The CMM2 supports creating sprites by reading in an ascii file formatted as described in the Colour Maximite manual with a key enhancement: CMM2 sprites can be any width and any height.

Sprites are loaded from Maximite style SPRITE files using SPRITE LOAD filename$. There is now has an optional parameter available in the first line of the sprite file. The definition of the new first line is:-

Horizontal size in pixels (need not be 16),
number of sprites described in the file,
vertical size of sprites in pixels (need not be 16)

If the vertical size parameter is omitted then the first parameter is used for both width and height allowing complete compatibility with CMM sprite files.

The example on the next page defines a red mouse pointer 13 pixels wide by 19 pixels high.

```
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
```

To create the sprites described by the file we use the command:

```
SPRITE LOAD fname [,start_sprite_number]
```

SPRITE LOAD loads CMM1 style sprite files. You can load multiple CMM1 style sprite files by specifying the start sprite number of each file making sure they don't overlap. The start sprite number defaults to 1 if not specified.

To try this method, save the above as "mouse.spr", then:

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

Note CMM style sprite files are limited to the 8 CMM colours and use the CMM colour coding. Ignore for the moment the specifics of the SPRITE SHOW command , we will get to that later.

SPRITE LOADPNG

Load a sprite from a .png file.

This method can be slightly more complex but is much more powerful not least because the images can be full colour. If you google "png transparent" and look at the images found you will see lots that look like this:

Note that surrounding the butterfly there appears to be a chequerboard pattern. This identifies areas in the image where there is nothing there. This is different from the butterfly appearing on a black or white background. In this case the background is non-existent or see-through. PNG files that look like this will have been written in ARGB8888 format and these files are ideal for creating sprites.

The format of the command for creating a sprite from a PNG file is:

```
SPRITE LOADPNG spriteno, fname$ [, transparency_cut_off]
```

The first two parameters are obvious, the number of the sprite we are creating and the name of the PNG file we will be reading from. The last transparency_cut_off parameter is optional and defaults to 8. It determines how the A level in ARGB8888 will be treated. In the PNG file A can be a value between 0 and 255 but to keep things simple the CMM2 will treat each pixel as being fully transparent or fully solid. The CMM2 firmware divides the A value in the PNG file by 16 and then compares it with the transparency_cut_off. If it is greater or equal to transparency_cut_off the pixel will be a solid colour. If it is less it will be completely transparent.

This distinction is important as it allows us to have both transparent pixels and solid BLACK pixels. ie we don't have to specify a particular colour to be "transparent".

In the next post we will use sprites read in from PNG files in the examples but we for now we know how to load them.

SPRITE READ

Create a sprite by reading from what is already displayed on one of the CMM2 video pages

This can be the simplest way of creating a sprite. We simply specify that we want to read what is on one of the video pages and create a sprite from it. Given that, the syntax is obvious:

```
SPRITE READ [#]n, x , y, w, h [,page_number]
```

Added additional parameter to SPRITE READ SPRITE READ [#]n, x , y, w, h [,page_number]
This specifies which page to read from.

This says create sprite number n by reading what is on the screen in the rectangle with the top left coordinate of x and y, width w and height h. There is just one added component to this: SPRITE READ will read from the PAGE specified by page_number - if omitted it defaults to the WRITE page. This allows us to read in sprites without changing PAGE 0 which is always being displayed.

Suppose we want a sprite which is simply a red box 10x10 pixels

```
PAGE WRITE 1
CLS RGB(RED)
SPRITE READ 1, 0, 0, 10, 10, 1
PAGE WRITE 0
SPRITE SHOW 1,100,100,1
```

would do the job and display the sprite on the display without having to see its creation.
Note: BLIT READ and SPRITE READ are identical and do exactly the same thing.

BLIT to and from the framebuffer is now supported for all graphics modes and with all orientation parameter support.
To BLIT from the framebuffer to the current WRITE page, use FRAMEBUFFER as the page parameter e.g.

```
PAGE WRITE 0
BLIT 50,50,500,500,75,75,FRAMEBUFFER
```

To BLIT to the framebuffer from a normal page, just use the BLIT command as normal

```
PAGE WRITE FRAMEBUFFER
BLIT 500,500,50,50,75,75,0
```

BLIT READ/SPRITE READ both support reading from the FRAMEBUFFER

```
BLIT READ #1,100,100,100,100,FRAMEBUFFER
```

Note: See the chapter on FRAMEBUFFER for more information on using this feature.

Text in a tip

SPRITE COPY

This command will copy an existing sprite to create a new one

As mentioned earlier, each sprite created uses two RAM buffers one for the sprite and one to store the display contents when the sprite is shown. This can use quite a lot of memory for big sprites. However, if we want many sprites that are based on the same image (the blocks in this demo are all sprites) we can copy one sprite to many others and the "children" sprites will only take the one memory buffer needed to save the screen but share the image with the "parent".

The syntax for copying sprites is:

```
SPRITE COPY [#]n, [#]m, nbr
```

This makes a copy of sprite “n” to “nbr” of new sprites starting at number “m”.

The code from the demo uses both this and SPRITE READ to create the block sprites

```
' Draw one box of each colour
Box 401,1,18,48,2,RGB(green),RGB(Magenta)
Box 421,1,18,48,2,RGB(red),RGB(cyan)
Box 441,1,18,48,2,RGB(yellow),RGB(blue)
Box 461,1,18,48,2,RGB(white),RGB(red)
Box 481,1,18,48,2,RGB(Magenta),RGB(GREEN)

' read in each block as a parent sprite
SPRITE read 10+i,401,1,18,48
SPRITE read 18+i,421,1,18,48
SPRITE read 26+i,441,1,18,48
SPRITE read 34+i,461,1,18,48
SPRITE read 42+i,481,1,18,48

'copy each parent to 7 children
SPRITE copy 10,11,7
SPRITE copy 18,19,7
SPRITE copy 26,27,7
SPRITE copy 34,35,7
SPRITE copy 42,43,7
```

We now know how to create sprites and can think about the pros and cons of the various methods. Next post we had better start looking at how to use them.

SPRITES AND HOW TO USE THEM

In this post I will look at some of the Basics of sprites and then the final sprite post will look at collisions. I might then do one more post on 3D graphics and try to de-mystify some of the maths behind them.

In the previous post we saw what a sprite is and how to create one so we can start here by getting rid of one or more sprites. This may be important to free up the 64 sprite slots for new sprites and the syntax is obvious.

```
SPRITE CLOSE [#]n
SPRITE CLOSE ALL
```

Executing these commands frees up the sprite slot and the memory used by the sprite. There is one important difference in their usage. Closing an individual sprite will also remove it from the screen if it is visible. Closing all sprites leaves the screen untouched as this is likely only to be used at the end of a program or when switching to a completely new context in a game. It is also important to note that you can't close a sprite from which copies have been made until all of the copies have been closed.

So we can create sprites and destroy them but we now need to know how to use them.

There is one overriding rule to be followed to make things work:

*** While sprites are in use, all sprite commands must be done with the same PAGE WRITE set

This need not be PAGE 0, the display page, but must be consistent for all SPRITE commands. You can move to other pages and use other graphics commands but if any sprite is active then all subsequent sprite commands must be to that page.

To keep things simple for the moment we will not change the page we are writing to and leave it at PAGE 0. I've attached the sprite file for the red mouse pointer we saw earlier and an image to use in the next example.

Tip!

You need at least V5.05.03 to run this example due to a bug in earlier versions.

mouse.zip

```
'set to 320x200 16 bit colour
mode 3,16
' load the mouse cursor as sprite number 1
sprite load "mouse.spr",1
'load a background image
load bmp "tiger320"
'set up some variables to track the sprite position and show it
dim integer x=mm.hres/2, y=mm.vres/2
sprite show 1, x, y, 1
'loop forever
do
'use the keydown function with the USB keyboard to check for a keypress
'you can change this to use inkey$ to use a serial console
 if keydown(0) then 'There is a key pressed so check for one of the arrow keys
   if keydown(1)=128 then  y=y-1
   if keydown(1)=129 then  y=y+1
   if keydown(1)=130 then  x=x-1
   if keydown(1)=131 then  x=x+1

' use the sprite function to check if the sprite needs to move
' by comparing its actual position to the new x,y position
' if it is different then display the sprite in the new position
' this will automatically restore the background overwritten by the original sprite
' and then rewrite it in the new position

   if x<> sprite(x,1) or y <> sprite(y,1) then sprite show 1, x, y, 1

 endif
 pause 10
loop
```

Try keeping the left arrow pressed and watch as the sprite leaves the screen

```
[12] Error: -13 is invalid (valid is -12 to 319)
```

Oops: sprites must have at least one pixel of their containing rectangle on the screen otherwise you will get an error. We can easily check for this in the code

```
   if keydown(1)=128 and sprite(y,1) + sprite(h,1) > 1 then  y=y-1
   if keydown(1)=129 and sprite(y,1)+1 < MM.VRES then  y=y+1
   if keydown(1)=130 and sprite(x,1) + sprite(w,1) > 1 then  x=x-1
   if keydown(1)=131 and sprite(x,1)+1 < MM.HRES then  x=x+1
```

The SPRITE function is very useful for telling us about the sprite. Check the manual for all the various SPRITE function parameters.

The important command we have used here is SPRITE SHOW. This command is used both to display a sprite and to move it. However it has two additional parameters that we haven't yet considered.

```
SPRITE SHOW [#n], x, y, layer [,orientation]
```

The optional orientation parameter acts to modify the sprite as it is displayed in the same way as we saw for the BLIT command, 0=normal, 1=mirrored left to right, 2=mirrored top to bottom, 3=rotated 180 degrees.

The mandatory (collision) layer parameter does two things for us relating to collisions and movement and can be set between 0 and 10.

Sprites shown on layer 0 will move with the background if we use SPRITE SCROLL or SPRITE SCROLLR. Sprites on all other layers will remain fixed on the screen during a scroll and the background will move underneath them.

Sprites collisions are detected when a sprite overlaps another sprite on the same layer or layer 0 so collisions can be caused by moving a sprite on any layer or scrolling the background such that a layer 0 sprite now overlaps a sprite on any of the other layers.

Collisions will be discussed in much more detail in the next post but there are other sprite commands we should look at first.

SPRITE WRITE and BLIT WRITE do exactly the same thing. In both case the SPRITE is written out but the background image isn't stored first and you can WRITE the same sprite as many times as you wish to different locations on the screen. In essence the command is just using the sprite as a in memory image that can be used whenever required.

SPRITE HIDE does what it says, it removes a sprite from the screen and restores the background we can add this to our example program and make the cursor flash.

```
mode 3,16
sprite load "mouse.spr",1
load bmp "tiger320"
dim integer x=mm.hres/2, y=mm.vres/2
settick 100,flash
sprite show 1, x, y, 1
do
 if keydown(0) then
   if keydown(1)=128 and sprite(y,1) + sprite(h,1) > 1 then  y=y-1
   if keydown(1)=129 and sprite(y,1)+1 < MM.VRES then  y=y+1
   if keydown(1)=130 and sprite(x,1) + sprite(w,1) > 1 then  x=x-1
   if keydown(1)=131 and sprite(x,1)+1 < MM.HRES then  x=x+1
 endif
 pause 20
loop

sub flash
 static a=1
 if a then
   sprite show 1,x,y,1
   a=0
 else
   sprite hide 1
   a=1
 endif
end sub
```

SPRITE NEXT and SPRITE MOVE commands allow you to create a sequence of sprite actions which will be executed as a single atomic transaction. This was developed for other platforms to avoid tearing and flashing effects but because on the CMM2 sprites can be moved on non-visible pages these commands are not as important.

The SPRITE SCROLL and SPRITE SCROLLR commands allow you to move the background image under the sprites for the whole screen or just part of it. SPRITE SCROLL is really identical to PAGE SCROLL that we discussed in detail above with the simple exception that all sprites on layers 1 to 10 are automatically hidden before the scroll and then replaced in their original positions afterwards. SPRITE SCROLLR does exactly the same thing but allows you to specify a part of the background image to scroll.

The last sprite command we will consider in this post is SPRITE TRANSPARENCY and this is used to create the **ghost demo**

The full code for this is attached and explained below. This uses many of the concepts we have seen in the posts on this thread and brings them together. What I hope will be clear from reading the code is how easy it is to create stunning effects with the CMM2 once you understand a little about the graphics are designed and work.

Image files for the ghost demo are **2020-05-08_234454_Ghost.zip**

```
'set to 640x400 mode with 2 video layers and 12-bit colour. Page 0 is the bottom layer and page 1 is
the top
mode 2,12
'clear the first three framebuffers (pages)
for i=0 to 2:page write i:cls:next i
'load a sprite which is the image of the ghost from a png file.
'The fact it is a png is important as png files encode transparency as well as solid colours
sprite loadpng 1,"ghost"
'set that we are going to write to the background layer
page write 0
'load the background image to the background layer - page 0
load png "part02"

'The image I am using is only 320x200 so I'm going to resize it to fit the screen
image resize 0,0,320,200,0,0,640,400

'initialise the display position of the ghost
x=100
y=50

'initialise the transparency of the ghost
' transparencies go from 1 (nearly invisible) to 15 (solid colour)
t=8

'set to write to page 2 which is not being displayed
page write 2

' output the ghost on page 2
sprite show 1,x,y,1
```

```
i=0

'start the main process loop
do

'do some silly maths to create a random walk of the ghost in both position and
' transparency while keeping it within the display bounds and the transparency
' within useful limits
i=i+1
if i mod 5 = 0 then c=rnd()-0.5
if i mod 3 = 0 then a=rnd()*8-4
if i mod 3 = 0 then b=rnd()*6-3
x=x+a
if x<0 then x=0
if x>MM.HRES-sprite(w,1) then x=mm.hres-sprite(w,1)
y=y+b
if y<0 then y=0
if y>MM.VRES-sprite(h,1) then y=mm.vres-sprite(h,1)
t=t+c
if t<3 then t=3
if t>12 then t=12

'display the sprite in the new position and with the new transparency
sprite transparency 1,t
sprite show 1,x,y,1

'now copy page 2 to the foreground layer during frame blanking
'this ensures that there are no tearing effects in the image

page copy 2 to 1,b
'slow things down a bit, the CMM2 is too fast
pause 100
loop
```

New SPRITE sub-commands:

In version 5.05.05 of the firmware, a number of new SPRITE sub commands were introduced.

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 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 HIDE SAFE n

Hides a sprite but automatically compensates for any other sprites that overlap it. This allows you to safely hide a sprite that is partially obscured but is obviously lower in performance than using the simple HIDE command

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 relative position relative to other sprites based on the 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.

```
box 0,0,50,50,4,rgb(red),rgb(white)
box 0,50,50,50,4,rgb(green),rgb(yellow)
sprite read 1,0,0,50,50
sprite read 2,0,50,50,50
sprite copy 2,3,1
sprite show 1, 150,150,1
sprite show 2, 170,170,1
sprite show 3,210,210,1
pause 2000
sprite show safe 1,180,180,1,,1
pause 2000
sprite show safe 2,190,190,1
pause 2000
sprite show safe 1,200,200,1
pause 2000
sprite hide safe 3
do
loop
```

The FRAMEBUFFER command

The framebuffer is an area of memory that can be up to 1600x1200 pixels in size irrespective of the current mode. The memory for the framebuffer is taken from the 5Mb of variable memory.

Note: The FRAMEBUFFER is a completely different and independent section of memory from that used for Pages.

Rather than PAGE COPY you use FRAMEBUFFER WINDOW to move data from the framebuffer to main memory. This allows you to have a playing field bigger than the display and the window that is displayed on the screen to be a different part of the field with a single command.

A simple example of using the framebuffer is shown here

This command allows you to create, use and remove a variable size framebuffer which should make many applications which need a working graphics area bigger than the screen easier to program. Moving a 640x400x16 image from the framebuffer to the display takes just over 4mSec. Moving a 320x200x8 image from the framebuffer to the display takes just less than 1mSec.

FRAMEBUFFER CREATE HorizontalSize%, VerticalSize%

This command creates a framebuffer with the width and height specified in pixels. HorizontalSize>=MM.HRES and <=1600: VerticalSize>=MM.VRES and <=1200

FRAMEBUFFER WRITE
PAGE WRITE FRAMEBUFFER

These commands are interchangeable and set all drawing commands to write to the framebuffer and inherit the width and height defined.

FRAMEBUFFER BACKUP

This command creates a backup copy of the framebuffer. If a backup already exists it is overwritten. This allows the programmer to save the background before he/she starts writing non-static data to it. NB: It won't be possible to use this command if a very large framebuffer is specified in 12 or 16-bit colour depth. A sensible error will be given in this case.

FRAMEBUFFER RESTORE [x, y, w, h

This command restores all or part of the framebuffer from the backup. This allows the programmer to “clean” all or part of the framebuffer before adding new non-static items.

FRAMEBUFFER WINDOW x, y, pageno [,I or B]

This command copies an area MM.HRES by MM.VRES from the framebuffer with top left at x,y to the page specified, The optional parameter specifies if the copy is **I**mmediate or during frame **B**lanking.

FRAMEBUFFER CLOSE

This command releases the memory resources used by the framebuffer and backup allowing a new framebuffer to be created with a different size.

MM.INFO(FrameH)

This function returns the horizontal size of the framebuffer in pixels.

MM.INFO(FrameV)

This function returns the vertical size of the framebuffer in pixels.

Note: The following should be noted when using the FRAMEBUFFER command.

- Setting a different graphics mode that changes the colour depth will close and delete the framebuffer.
- JPG files cannot be loaded to the framebuffer and will error if tried.
- The framebuffer is deleted by Ctrl-C and by running a new program.
- Changing graphics mode such that MM.HRES or MM.VRES are bigger than the size of the framebuffer will cause it to be deleted.

Example use similar to the video:

```
option console serial
framebuffer create 1024,1024
page write framebuffer
load bmp "1024" 'load a 1024x1024 image to the framebuffer
memory
framebuffer backup 'back up the image
memory
cls rgb(red) 'clear the framebuffer to red
framebuffer restore 100,100,800,800 'restore the central area of the backup
'
line 0,0,1023,1023 'check simple graphics command
text 850,850,"hello",,,4,rgb(blue),-1 'check text output to the framebuffer
```

```
j=0
do
mode 4,16
for i=0 to 1023-mm.hres
framebuffer window i,j,0
next i
for j=0 to 1023-mm.vres
framebuffer window i,j,0
next j
for i=1023-mm.hres to 0 step -1
framebuffer window i,j,0
next i
i=0
for j=1023-mm.vres to 0 step -1
framebuffer window i,j,0
next j
j=0
framebuffer restore 'go back to the clean image
mode 2,16
for i=0 to 1023-mm.hres
timer=0
framebuffer window i,j,0
print timer
next i
for j=0 to 1023-mm.vres
framebuffer window i,j,0
next j
for i=1023-mm.hres to 0 step -1
framebuffer window i,j,0
next i
i=0
for j=1023-mm.vres to 0 step -1
framebuffer window i,j,0
next j
j=0
loop
```

BLIT to and from the framebuffer is now supported for all graphics modes and with all orientation parameter support. To BLIT from the framebuffer to the current WRITE page, use FRAMEBUFFER as the page parameter e.g.

```
PAGE WRITE 0
BLIT 50,50,500,500,75,75,FRAMEBUFFER
```

To BLIT from the framebuffer to a normal page just use the BLIT command as normal

```
PAGE WRITE FRAMEBUFFER
BLIT 500,500,50,50,75,75,0
```

BLIT READ/SPRITE READ both support reading from the FRAMEBUFFER

```
BLIT READ #1,100,100,100,100,FRAMEBUFFER
```

QUATERNIONS

In this thread we look at using the mathematical technique of using QUATERNIONs to manipulate wire-frame objects in 3-dimensional space.

This is a demo program (all in Basic) of some interesting maths that underpins all sorts of technologies like attitude/heading systems and 3D graphics.

Editors note: While not specific to the CMM2, this program demonstrates the power of the graphics in the CMM2.

Quaternions are a mathematical technique that allows the rotation in any direction of any shape. The maths behind it are way beyond me and extend the concept of complex numbers such that:

i*i == j*j == k*k == i*j*k == -1

I find that equation completely bemusing

However, quaternions are pretty easy to use in the real world. There are some gotchas: quaternions are not commutative e.g q1 * q2 does not give the same result as q2 * q1. But overall the calculations are pretty simple.

The attached code draws a wireframe cube and then rotates it continuously in the directions specified by the rotation quaternion.

The cube is specified as a set of coordinates around a point 0,0,0 in the array "cube". Then the array cubelines% is used to store the coordinate pairs that make up the lines that draw the cube.

The array "cubelocation" defines the actual position of the cube in space. The x and y axes are as per the display and the z axis comes out of the display towards the viewer.

The code uses a projection method of perspective to visualise the cube in 2 dimensions and "viewplane" defines the location of the projection plane in the z axis.

All quaternion arithmetic is carried out with the quaternion normalised. The subroutine "create_normalised_quaternion" takes a rotation vector and an angle and converts it into a quaternion.

Then to rotate a point we just multiply the rotation quaternion by the normalised coordinates of the point and then multiply the answer by the inverse of the quaternion. Note how in normal math this would leave the point unchanged but because of the non-commutative nature of the quaternion somehow the sequence does what we want (look at Jorge Rodriguez excellent videos on Youtube here if you want to know more). Note that no trigonometry is needed other than in the initialisation of the rotation quaternion and that only uses the sine and cosine of a single angle.

The attached code includes all the routines necessary to rotate objects in 3D and then project them onto a 2D screen.

My interest is in using the math to understand what is happening inside an Attitude and Heading Reference System (AHRS), but the example code below is quite fun.

The code should run on any type of Micromite with a connected TFT display. The picture is of a 320x240 SSD1289 display driven by a 44-pin uM2.

Play with this line:

```
create_normalised_quaternion(2,1,0,0,q1())
```

to change the way the cube rotates. The parameters are:
number of degrees to rotate about the rotation vector,
x-coordinate of the end of the rotation vector,
y-coordinate of the end of the rotation vector,
z-coordinate of the end of the rotation vector, and
array to receive the quaternion.

Enjoy

```
option explicit
option default FLOAT
cpu 48
' 3D wire frame graphics demo using quaternions to rotate the coordinates
cls

dim q1(4),v(4),vout(4),M_PI=3.14159265359,i%,x1,y1,z1,x2,y2,z2

' define the coordinates of a cube with the z-axis towards the viewer
dim cube(3,7)=(-100,-100,100,0, -100,100,100,0, 100,-100,100,0, 100,100,100,0, -100,-
100,-100,0, -100,100,-100,0, 100,-100,-100,0, 100,100,-100,0 ) 'coordinates of the cube

' define the coordinate pairs that make up the edges of the cube
DIM cubelocation(2)=(mm.hres/2,MM.VRES/2,-500) 'actual location of the centre of the cube
in space

' define the coordinate pairs that make up the edges of the cube
dim cubelines%(1,11)=(0,1, 0,2, 1,3, 2,3, 4,5, 4,6, 5,7, 6,7, 0,4, 1,5, 2,6, 3,7)

'define the projection plane
DIM viewplane = -200
dim lines%(3,11),oldlines%(3,11) 'storage for the coordinates for lines drawn

for i%=0 to 7 'convert coordinates to normalised form
  x1=cube(0,i%)
  y1=cube(1,i%)
  z1=cube(2,i%)
  create_vector(x1,y1,z1,v())
  cube(0,i%)=v(2)
  cube(1,i%)=v(3)
  cube(2,i%)=v(4)
  cube(3,i%)=v(0)
next i%

'create a quarternion to rotate 2 degrees about x axis

'play with the x,y,z vector which is the sxis of rotation
create_normalised_quaternion(2,1,0,0,q1())

do
for i%=0 to 11 'project the coordinates onto the viewplane
  z1=cube(2,cubelines%(0,i%))*cube(3,cubelines%(0,i%))+cubelocation(2)
  x1=(cube(0,cubelines%(0,i%))*cube(3,cubelines%(0,i%))*viewplane/z1 +cubelocation(0))
  y1=(cube(1,cubelines%(0,i%))*cube(3,cubelines%(0,i%))*viewplane/z1 +cubelocation(1))
  z2=cube(2,cubelines%(1,i%))*cube(3,cubelines%(1,i%))+cubelocation(2)
  x2=(cube(0,cubelines%(1,i%))*cube(3,cubelines%(1,i%))*viewplane/z2 +cubelocation(0))
  y2=(cube(1,cubelines%(1,i%))*cube(3,cubelines%(1,i%))*viewplane/z2 +cubelocation(1))
  lines%(0,i%)=x1 :lines%(1,i%)=y1 :lines%(2,i%)=x2 :lines%(3,i%)=y2
next i%

for i%=0 to 7 'rotate coordinates
  v(2)=cube(0,i%)
  v(3)=cube(1,i%)
  v(4)=cube(2,i%)
  v(0)=cube(3,i%)
  v(1)=0
  rotate_vector(vout(),v(),q1())
  cube(0,i%)=vout(2)
  cube(1,i%)=vout(3)
  cube(2,i%)=vout(4)
  cube(3,i%)=vout(0)
next i%

for i%=0 to 11 'delete the old lines
  line oldlines%(0,i%),oldlines%(1,i%),oldlines%(2,i%),oldlines%(3,i%),1,rgb(black)
next i%
```

```
for i%=0 to 11 'draw the new lines
  line lines%(0,i%),lines%(1,i%),lines%(2,i%),lines%(3,i%),1,rgb(white)
  oldlines%(0,i%)=lines%(0,i%)
  oldlines%(1,i%)=lines%(1,i%)
  oldlines%(2,i%)=lines%(2,i%)
  oldlines%(3,i%)=lines%(3,i%)
next i%

loop
end

'
sub create_normalised_quaternion(theta,x,y,z,q())
  local radians = theta/180.0*M_PI
  local sineterm= sin(radians!/2)
  q(1)=cos(radians/2)
  q(2)=x* sineterm
  q(3)=y* sineterm
  q(4)=z* sineterm
  q(0)=sqr(q!(1)*q(1) + q(2)*q(2) + q(3)*q(3) + q(4)*q(4)) 'calculate the magnitude
  q(1)=q(1)/q(0) 'create a normalised quaternion
  q(2)=q(2)/q(0)
  q(3)=q(3)/q(0)
  q(4)=q(4)/q(0)
  q(0)=1
end sub
'
sub invert_quaternion(n(),q())
  n(0)=q(0)
  n(1)=q(1)
  n(2)=-q(2)
  n(3)=-q(3)
  n(4)=-q(4)
end sub
'
sub multiply_quaternion(n(),q1(),q2())
  local a1=q1(1),a2=q2(1),b1=q1(2),b2=q2(2),c1=q1(3),c2=q2(3),d1=q1(4),d2=q2(4)
  n(1)=a1*a2-b1*b2-c1*c2-d1*d2
  n(2)=a1*b2+b1*a2+c1*d2-d1*c2
  n(3)=a1*c2-b1*d2+c1*a2+d1*b2
  n(4)=a1*d2+b1*c2-c1*b2+d1*a2
  n(0)=q1(0)*q2(0)
end sub
'
sub create_vector(x,y,z,v())
  v(0)=sqr(x*x + y*y + z*z)
  v(1)=0
  v(2)=x/v(0)
  v(3)=y/v(0)
  v(4)=z/v(0)
end sub

sub rotate_vector(vnew(),v(),q())
  local n(4),iq(4)
  multiply_quaternion(n(),q(),v())
  invert_quaternion(iq(),q())
  multiply_quaternion(vnew(),n(),iq())
end sub
```

```
option explicit
option default none
mode 2
page write 1
const edgelength=100 'set the length of the verticies of the ticosahedron
const zlocation=1000 'how far is the center of the ticosahedron away from us
const viewplane=800 'how far is the viewplane away from us
dim float x,y,d
dim float phi=(1+sqr(5))/2
dim float x1,y1,z1
```

```
dim integer col(11), sortorder(11)
dim float q1(4),depth(11),v(4), vout(4)
' data for location of verticies for ticosahedron of edge length 2
data 0,1,3*phi
data 0,1,-3*phi
data 0,-1,3*phi
data 0,-1,-3*phi
data 1,3*phi,0
data 1,-3*phi,0
data -1,3*phi,0
data -1,-3*phi,0
data 3*phi,0,1
data 3*phi,0,-1
data -3*phi,0,1
data -3*phi,0,-1
data 2,(1+2*phi),phi
data 2,(1+2*phi),-phi
data 2,-(1+2*phi),phi
data 2,-(1+2*phi),-phi
data -2,(1+2*phi),phi
data -2,(1+2*phi),-phi
data -2,-(1+2*phi),phi
data -2,-(1+2*phi),-phi
data (1+2*phi),phi,2
data (1+2*phi),phi,-2
data (1+2*phi),-phi,2
data (1+2*phi),-phi,-2
data -(1+2*phi),phi,2
data -(1+2*phi),phi,-2
data -(1+2*phi),-phi,2
data -(1+2*phi),-phi,-2
data phi,2,(1+2*phi)
data phi,2,-(1+2*phi)
data phi,-2,(1+2*phi)
data phi,-2,-(1+2*phi)
data -phi,2,(1+2*phi)
data -phi,2,-(1+2*phi)
data -phi,-2,(1+2*phi)
data -phi,-2,-(1+2*phi)
data 1,(2+phi),2*phi
data 1,(2+phi),-2*phi
data 1,-(2+phi),2*phi
data 1,-(2+phi),-2*phi
data -1,(2+phi),2*phi
data -1,(2+phi),-2*phi
data -1,-(2+phi),2*phi
data -1,-(2+phi),-2*phi
data (2+phi),2*phi,1
data (2+phi),2*phi,-1
data (2+phi),-2*phi,1
data (2+phi),-2*phi,-1
data -(2+phi),2*phi,1
data -(2+phi),2*phi,-1
data -(2+phi),-2*phi,1
data -(2+phi),-2*phi,-1
data 2*phi,1,(2+phi)
data 2*phi,1,-(2+phi)
data 2*phi,-1,(2+phi)
data 2*phi,-1,-(2+phi)
data -2*phi,1,(2+phi)
data -2*phi,1,-(2+phi)
data -2*phi,-1,(2+phi)
data -2*phi,-1,-(2+phi)
' 12 faces with 5 sides
data 0,28,36,40,32
data 1,29,37,41,33
data 2,30,38,42,34
data 3,31,39,43,35
data 4,12,44,45,13
data 5,14,46,47,15
data 6,16,48,49,17
data 7,18,50,51,19
data 8,20,52,54,22
data 9,21,53,55,23
data 10,24,56,58,26
data 11,27,59,57,25
' 20 faces with 6 sides
```

```
data 0,2,34,58,56,32
data 0,2,30,54,52,28
data 1,3,31,55,53,29
data 1,3,35,59,57,33
data 4,6,17,41,37,13
data 4,6,16,40,36,12
data 5,7,19,43,39,15
data 5,7,18,42,38,14
data 8,9,23,47,46,22
data 8,9,21,45,44,20
data 10,11,27,51,50,26
data 10,11,25,49,48,24
data 12,44,20,52,28,36
data 13,45,21,53,29,37
data 14,46,22,54,30,38
data 15,47,23,55,31,39
data 16,48,24,56,32,40
data 17,49,25,57,33,41
data 18,50,26,58,34,42
data 19,51,27,59,35,43
'
dim float zpos(31),zsort(31)
dim float ticos(2,59), nticos(3,59)
dim integer i,j,k,l,m,n
dim integer xs(179),ys(179),xe(179),ye(179)
dim integer index(31),nnum(31)
' read in the coordinates of the verticies and scale
for j=0 to 59
 for i=0 to 2
   read ticos(i,j)
   ticos(i,j)=ticos(i,j)*edgelength/2
 next i
next j
'
dim integer xarr(179),yarr(179)
dim integer nv(31)=(5,5,5,5,5,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6)
dim integer np(31)=(0,1,2,3,4,5,6,7,8,9,10,11,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
dim integer nd(31)
dim integer f5(4,11), f6(5,19)
dim integer ncol(31)
for j=0 to 11
 for i=0 to 4
   read f5(i,j)
 next i
next j
for j=0 to 19
 for i=0 to 5
   read f6(i,j)
 next i
next j

'convert coordinates to normalised form
for i=0 to 59
 x1=ticos(0,i): y1=ticos(1,i): z1=ticos(2,i)
 create_vector(x1,y1,z1,v())
 nticos(0,i)=v(2): nticos(1,i)=v(3): nticos(2,i)=v(4): nticos(3,i)=v(0)
next i

'create a quarternion to rotate 4 degrees about a chosen axis
'play with the x,y,z vector which is the sxis of rotation
create_normalised_quaternion(4,1,0.5,0.25,q1())

do
 cls
 for i=0 to 59 'rotate coordinates
   v(2)=nticos(0,i): v(3)=nticos(1,i): v(4)=nticos(2,i): v(0)=nticos(3,i): v(1)=0
   rotate_vector(vout(),v(),q1())
   nticos(0,i)=vout(2): nticos(1,i)=vout(3): nticos(2,i)=vout(4): nticos(3,i)=vout(0)
 next i
'
' average the z positions for the five sided faces
 for k=0 to 11
   zpos(k)=0
   for i=0 to 4
     zpos(k)=zpos(k)+nticos(2,f5(i,k))
   next i
   zpos(k)=zpos(k)/5
```

```
'    index(k)=k
 next k
'average the z positions for the 6 sided faces
 for k=12 to 31
   zpos(k)=0
   for i=0 to 5
     zpos(k)=zpos(k)+nticos(2,f6(i,k-12))
   next i
   zpos(k)=zpos(k)/6
'    index(k)=k
 next k
' sort the z positions
 sort zpos(),index()
'
 j=0:m=0
 for l=0 to 31
   k=index(l)
   m=np(k)
   nd(l)=nv(k)
   if nv(k)=5 then
     ncol(l)=rgb(red)
   else
     ncol(l)=rgb(white)
   endif
   for i=0 to nv(k)-1
     if nv(k)=5 then
       xarr(j)=nticos(0,f5(i,m))*viewplane/(nticos(2,f5(i,m))+zlocation)*nticos(3,f5(i,m))+MM.HRES/2
       yarr(j)=nticos(1,f5(i,m))*viewplane/(nticos(2,f5(i,m))+zlocation)*nticos(3,f5(i,m))+MM.VRES/2
     else
       xarr(j)=nticos(0,f6(i,m))*viewplane/(nticos(2,f6(i,m))+zlocation)*nticos(3,f6(i,m))+MM.HRES/2
       yarr(j)=nticos(1,f6(i,m))*viewplane/(nticos(2,f6(i,m))+zlocation)*nticos(3,f6(i,m))+MM.VRES/2
     endif
     j=j+1
   next i
 next l
 polygon nd(),xarr(),yarr(),rgb(black),ncol()
 page copy 1 to 0
loop
'
sub create_normalised_quaternion(theta as float,x as float,y as float,z as float,q() as float)
 local float radians = theta/180.0*PI
 local float sineterm= sin(radians!/2)
 q(1)=cos(radians/2)
 q(2)=x* sineterm
 q(3)=y* sineterm
 q(4)=z* sineterm
 q(0)=sqr(q!(1)*q(1) + q(2)*q(2) + q(3)*q(3) + q(4)*q(4)) 'calculate the magnitude
 q(1)=q(1)/q(0) 'create a normalised quaternion
 q(2)=q(2)/q(0)
 q(3)=q(3)/q(0)
 q(4)=q(4)/q(0)
 q(0)=1
end sub
'
sub invert_quaternion(n() as float,q() as float)
 n(0)=q(0)
 n(1)=q(1)
 n(2)=-q(2)
 n(3)=-q(3)
 n(4)=-q(4)
end sub
'
sub multiply_quaternion(n() as float,q1() as float,q2() as float)
 local float a1=q1(1),a2=q2(1),b1=q1(2),b2=q2(2),c1=q1(3),c2=q2(3),d1=q1(4),d2=q2(4)
 n(1)=a1*a2-b1*b2-c1*c2-d1*d2
 n(2)=a1*b2+b1*a2+c1*d2-d1*c2
 n(3)=a1*c2-b1*d2+c1*a2+d1*b2
 n(4)=a1*d2+b1*c2-c1*b2+d1*a2
 n(0)=q1(0)*q2(0)
end sub
'
sub create_vector(x as float,y as float ,z as float,v() as float)
 v(0)=sqr(x*x + y*y + z*z)
 v(1)=0
 v(2)=x/v(0)
 v(3)=y/v(0)
 v(4)=z/v(0)
```

```
end sub

sub rotate_vector(vnew() as float,v() as float,q() as float)
 local float n(4),iq(4)
 multiply_quaternion(n(),q(),v())
 invert_quaternion(iq(),q())
 multiply_quaternion(vnew(),n(),iq())
end sub
```

Warning: this code is completely useless - Watch this video and this one

.....but I think it is interesting

This post takes the discussion on quaternions further by using some of the techniques of 3D graphics processing to manipulate and display solid objects.

To do this we need to implement Hidden Surface Determination . In the code below we use Painter's Algorithm.

In general computer graphics uses triangles as the basic drawing object. In this example each face of the cube is split into two triangles and a standard fill-triangle algorithm used to draw them.

The vertices of the cube are manipulated in 3D in the same way as in the previous post. However, in this example we also manipulate the centroid (geometric centre) of each face of the cube. By doing this we can calculate the distance away of the centroid along the Z-axis after each rotation. This allows us to sort the Z-position of the centroids and display each face in turn, furthest away from the viewer first. This implements the Painter's algorithm which works extremely well except when faces overlap as shown in the Wikipedia article.

The code is all in Basic except the floating point sort and the triangle display functions.

Drawing has been deliberately slowed to see the hidden surfaces being overwritten by each face as they are drawn in increasing order of Z axis.

The code as listed runs on a SSD1963 800x480 display but will also run on a ILI9341 but you will need to adjust the "scale" to fit the screen (300 works well). It will run on uM2 or MM+

I hope this is interesting insight into the way objects can be manipulated in 3 dimensions and then displayed on a 2D screen

```
option explicit
option default FLOAT
' 3D solid graphics demo using quarternions to rotate the coordinates
'
cls
'
' Adjust the following to fit your display
'
dim scale=600 'length of the edge of the cube in "real" space
'define the projection plane in units from the observer
DIM viewplane = -500
DIM cubelocation(2)=(mm.hres/2,MM.VRES/2,-2000) 'actual location of the centre of the
cube in "real" 3D space
'
'
' define the coordinates of a cube with the z-axis towards the viewer and the centre at
0,0,0
dim cube(3,7)=(-1,-1,1,0, -1,1,1,0, 1,1,1,0, 1,-1,1,0, -1,-1,-1,0, -1,1,-1,0, 1,1,-1,0,
1,-1,-1,0 ) 'coordinates of the cube
dim centroid(3,5)=(0,0,1,0, -1,0,0,0, 0,1,0,0, 1,0,0,0, 0,0,-1,0, 0,-1,0,0) 'centre of
each face in space
```

```
'
' define the coordinate triples that make up two triangles for each face of the cube,
together with a face index number
' all drawing is done as filled triangles
' NB the triangles must be in the same order as the faces, two triangles/face
'
dim cubetriangles%(3,11)=(0,1,2,0, 0,2,3,0, 0,1,5,1, 0,4,5,1, 1,2,6,2, 1,5,6,2, 3,6,7,3,
3,2,6,3, 4,6,7,4, 4,5,6,4, 0,4,7,5, 0,3,7,5)
'
dim q1(4),v(4),vout(4),M_PI=3.14159265359,i%,j%,k%,x1,y1,z1,x2,y2,z2,x3,y3,z3
dim csort(5)
dim sortorder%(5)
DIM integer xx0(11), yy0(11), xx1(11), yy1(11), xx2(11), yy2(11), tcol(5) ,scol(11)
'arrays for calling the triangle draw routine
'
for i%=0 to 7 'scale the cube verticies up depending on the display size
for j%=0 to 2
cube(j%,i%)=cube(j%,i%)*scale/2
next j%
next i%
'
for i%=0 to 5 'scale the centroid positions up depending on the display size
for j%=0 to 2
centroid(j%,i%)=centroid(j%,i%)*scale/2
next j%
next i%
'
' define the colour of each face of the cube
tcol(0)=rgb(red)
tcol(1)=rgb(blue)
tcol(2)=rgb(magenta)
tcol(3)=rgb(yellow)
tcol(4)=rgb(cyan)
tcol(5)=rgb(green)


for i%=0 to 7 'convert coordinates to normalised form
x1=cube(0,i%): y1=cube(1,i%): z1=cube(2,i%)
create_vector(x1,y1,z1,v())
cube(0,i%)=v(2): cube(1,i%)=v(3): cube(2,i%)=v(4): cube(3,i%)=v(0)
next i%
'
for i%=0 to 5 'convert centroid coordinates to normalised form
x1=centroid(0,i%): y1=centroid(1,i%): z1=centroid(2,i%)
create_vector(x1,y1,z1,v())
centroid(0,i%)=v(2): centroid(1,i%)=v(3): centroid(2,i%)=v(4): centroid(3,i%)=v(0)
next i%

'create a quarternion to rotate 15 degrees about a chosen axis
'play with the x,y,z vector which is the sxis of rotation
create_normalised_QUATERNION(15,1,0.5,0.25,q1())
'
do
for i%=0 to 7 'rotate coordinates
v(2)=cube(0,i%): v(3)=cube(1,i%): v(4)=cube(2,i%): v(0)=cube(3,i%): v(1)=0
rotate_vector(vout(),v(),q1())
cube(0,i%)=vout(2): cube(1,i%)=vout(3): cube(2,i%)=vout(4): cube(3,i%)=vout(0)
next i%
'
for i%=0 to 5 'rotate centroids
v(2)=centroid(0,i%): v(3)=centroid(1,i%): v(4)=centroid(2,i%): v(0)=centroid(3,i%):
v(1)=0
rotate_vector(vout(),v(),q1())
centroid(0,i%)=vout(2): centroid(1,i%)=vout(3): centroid(2,i%)=vout(4): centroid(3,i
%)=vout(0)
csort(i%)=vout(4)
sortorder%(i%)=i%
next i%
'
```

```
floatindexsort(csort(),sortorder%(),6) 'sort the faces in the order of the depth
'
for i%=0 to 5 'project the coordinates onto the viewplane
j%=sortorder%(i%)*2 'get the furthest, as yet undisplayed, face from the viewer
k%=i%*2
z1=cube(2,cubetriangles%(0,j%))*cube(3,cubetriangles%(0,j%))+cubelocation(2)
x1=(cube(0,cubetriangles%(0,j%))*cube(3,cubetriangles%(0,j%))*viewplane/z1
+cubelocation(0))
y1=(cube(1,cubetriangles%(0,j%))*cube(3,cubetriangles%(0,j%))*viewplane/z1
+cubelocation(1))
z2=cube(2,cubetriangles%(1,j%))*cube(3,cubetriangles%(1,j%))+cubelocation(2)
x2=(cube(0,cubetriangles%(1,j%))*cube(3,cubetriangles%(1,j%))*viewplane/z2
+cubelocation(0))
y2=(cube(1,cubetriangles%(1,j%))*cube(3,cubetriangles%(1,j%))*viewplane/z2
+cubelocation(1))
z3=cube(2,cubetriangles%(2,j%))*cube(3,cubetriangles%(2,j%))+cubelocation(2)
x3=(cube(0,cubetriangles%(2,j%))*cube(3,cubetriangles%(2,j%))*viewplane/z3
+cubelocation(0))
y3=(cube(1,cubetriangles%(2,j%))*cube(3,cubetriangles%(2,j%))*viewplane/z3
+cubelocation(1))
xx0(k%)=x1 :yy0(k%)=y1: xx1(k%)=x2 :yy1(k%)=y2: xx2(k%)=x3 :yy2(k%)=y3 :scol(k%)=tcol(j%\
2)
k%=k%+1
z1=cube(2,cubetriangles%(0,j%+1))*cube(3,cubetriangles%(0,j%+1))+cubelocation(2)
x1=(cube(0,cubetriangles%(0,j%+1))*cube(3,cubetriangles%(0,j%+1))*viewplane/z1
+cubelocation(0))
y1=(cube(1,cubetriangles%(0,j%+1))*cube(3,cubetriangles%(0,j%+1))*viewplane/z1
+cubelocation(1))
z2=cube(2,cubetriangles%(1,j%+1))*cube(3,cubetriangles%(1,j%+1))+cubelocation(2)
x2=(cube(0,cubetriangles%(1,j%+1))*cube(3,cubetriangles%(1,j%+1))*viewplane/z2
+cubelocation(0))
y2=(cube(1,cubetriangles%(1,j%+1))*cube(3,cubetriangles%(1,j%+1))*viewplane/z2
+cubelocation(1))
z3=cube(2,cubetriangles%(2,j%+1))*cube(3,cubetriangles%(2,j%+1))+cubelocation(2)
x3=(cube(0,cubetriangles%(2,j%+1))*cube(3,cubetriangles%(2,j%+1))*viewplane/z3
+cubelocation(0))
y3=(cube(1,cubetriangles%(2,j%+1))*cube(3,cubetriangles%(2,j%+1))*viewplane/z3
+cubelocation(1))
xx0(k%)=x1 :yy0(k%)=y1: xx1(k%)=x2 :yy1(k%)=y2: xx2(k%)=x3 :yy2(k%)=y3 :scol(k%)=tcol(j%\
2)
next i%
cls
triangles(12,xx0(),yy0(),xx1(),yy1(),xx2(),yy2(),scol(),100000)
pause 1000
loop
end

'
sub create_normalised_QUATERNION(theta,x,y,z,q())
local radians = theta/180.0*M_PI
local sineterm= sin(radians!/2)
q(1)=cos(radians/2)
q(2)=x* sineterm
q(3)=y* sineterm
q(4)=z* sineterm
q(0)=sqr(q!(1)*q(1) + q(2)*q(2) + q(3)*q(3) + q(4)*q(4)) 'calculate the magnitude
q(1)=q(1)/q(0) 'create a normalised QUATERNION
q(2)=q(2)/q(0)
q(3)=q(3)/q(0)
q(4)=q(4)/q(0)
q(0)=1
end sub
'
sub invert_QUATERNION(n(),q())
n(0)=q(0)
n(1)=q(1)
n(2)=-q(2)
n(3)=-q(3)
n(4)=-q(4)
```

```
end sub
'
sub multiply_QUATERNION(n(),q1(),q2())
local a1=q1(1),a2=q2(1),b1=q1(2),b2=q2(2),c1=q1(3),c2=q2(3),d1=q1(4),d2=q2(4)
n(1)=a1*a2-b1*b2-c1*c2-d1*d2
n(2)=a1*b2+b1*a2+c1*d2-d1*c2
n(3)=a1*c2-b1*d2+c1*a2+d1*b2
n(4)=a1*d2+b1*c2-c1*b2+d1*a2
n(0)=q1(0)*q2(0)
end sub
'
sub create_vector(x,y,z,v())
v(0)=sqr(x*x + y*y + z*z)
v(1)=0
v(2)=x/v(0)
v(3)=y/v(0)
v(4)=z/v(0)
end sub

sub rotate_vector(vnew(),v(),q())
local n(4),iq(4)
multiply_QUATERNION(n(),q(),v())
invert_QUATERNION(iq(),q())
multiply_QUATERNION(vnew(),n(),iq())
end sub

Csub triangles 'draws multiple triangles with a delay between if required
00000000
8C820004 27BDFFA0 AFBF005C AFBE0058 AFB70054 AFB60050 AFB5004C AFB40048
AFB30044 AFB20040 AFB1003C AFB00038 AFA40060 AFA50064 AFA60068 184000EE
AFA7006C AFA00030 3C159D00 8FA30030 8FA50068 8FA80070 000310C0 00A22021
01021821 8C840000 8C630000 8FA90064 AFA40018 01223821 AFA3001C 8FA40074
8FA3006C 8FA80078 8FA9007C 00623021 00822821 01022021 01221021 8FA80018
8FA9001C 8CC60000 8CA50000 0128182A 8CF30000 8C9E0000 AFA60028 AFA5002C
10600006 8C540000 02601021 AFA90018 00C09821 AFA8001C AFA20028 8FA5001C
03C5102A 10400008 8FA90018 8FA20028 8FA8002C AFBE001C AFA80028 00A0F021
AFA2002C 8FA90018 8FA3001C 0069102A 10400008 8FA50018 8FA4001C 02601021
AFA40018 8FB30028 AFA9001C AFA20028 8FA50018 10BE008C 8FA4001C 109E00A4
8FA50018 2483FFFF 0065102A 1440002E 8FB00018 8FA90028 8FA2002C 8FA8001C
01334823 00531023 03C5B023 AFBE0034 0105B823 0280F021 AFA90020 AFA20024
00A08021 00009021 00008821 0060A021 0237001A 02E001F4 8FA30024 8FA20020
02002821 02003821 02228821 26100001 00002012 0256001A 02C001F4 00932021
02439021 00003012 00D33021 00C4182A 10600003 00801021 00C02021 00403021
8EA20048 AFBE0010 8C420000 0040F809 00000000 0290102A 1040FFE5 00000000
03C0A021 8FBE0034 03D0102A 14400030 8FA8001C 8FA90018 8FA4002C 8FA50028
0093B023 0085B823 02089023 02098823 03C91823 03C81023 72579002 72368802
AFB40018 AFB6001C AFB70020 0260B021 00A0B821 00609821 0040A021 0254001A
028001F4 8FA3001C 8FA20020 02002821 02003821 02429021 26100001 00002012
0233001A 026001F4 00972021 02238821 00003012 00D63021 00C4182A 10600003
00801021 00C02021 00403021 8FA80018 8EA20048 AFA80010 8C420000 0040F809
00000000 03D0102A 1040FFE4 00000000 8FA90080 8D240000 8D220004 00821025
5440001D 8EA20004 8FA20030 8FA40060 24420001 8C830004 AFA20030 000217C3
0043202A 5480FF52 8FA30030 14620006 8FA50060 8FA80030 8CA20000 0102102B
1440FF4B 8FA30030 8FBF005C 8FBE0058 8FB70054 8FB60050 8FB5004C 8FB40048
8FB30044 8FB20040 8FB1003C 8FB00038 03E00008 27BD0060 0040F809 00000000
1000FFE2 8FA20030 8FA80028 0113102A 1440001D 02603821 0268102A 1440001C
8FA70028 02603821 02602021 8FA9002C 0124102A 54400003 8FA4002C 00E9102A
0122380B 8EA20048 8FA30018 AFB40010 00E33821 8C420000 00602821 00803021
0040F809 00E43823 1000FFC2 8FA90080 8FA3001C 0065102A 1040FF5F 8FA90028
1000FF89 8FB00018 1000FFE8 8FA40028 1000FFE6 02602021 1440FFCC 8FBF005C
8C820000 5440FF10 AFA00030 1000FFC8 8FBE0058
End Csub


Csub floatindexsort ' sorts a floating point array but keeps an index so you know the
original order
00000000
8CC60000 27BDFF98 AFB00040 AFBF0064 AFBE0060 AFB7005C AFB60058 AFB50054
AFB40050 AFB3004C AFB20048 AFB10044 AFA6001C AFA40068 AFA5006C 00C08021
```

```
3C039D00 001017C2 00508021 00108043 12000068 8FBF0064 8FA4001C 0204102A
1040FFF9 001017C2 8FA60068 00102080 00C42021 00102823 AFA50010 AFA40028
8FA5001C 8FA4006C 001010C0 AFA6002C 8FA60010 00821021 00B02823 AFA20024
AFA50034 AFA00020 0006F080 0006B8C0 8FA20020 8FA50028 0202B021 8FA20024
8CA60000 AFA50014 8C450000 02D0202A AFA60018 1480002A AFA50030 8FA60020
8FA2006C 000690C0 00529021 8FB50024 8FB40028 8FB1002C 1000000F 00C09821
8E220000 8E440000 8E450004 0213B021 02D0302A AE820000 023E8821 02401021
AEA40000 AEA50004 02579021 029EA021 14C00013 02B7A821 8FA40010 8E250000
8C620068 02649821 8FA40018 AFA30038 0040F809 AFB10014 2405FFFF 1045FFE8
8FA30038 8FA60068 00161080 8FA4006C 00C21021 0016B0C0 AFA20014 00961021
8FA60020 8FA50030 24C60001 AFA60020 000527C3 8FA60018 8FA50014 ACA60000
8FA50028 8FA60030 24A50004 AFA50028 AC460000 AC440004 8FA20024 8FA4002C
8FA50020 8FA60034 24420008 24840004 AFA20024 14A6FFB6 AFA4002C 001017C2
00508021 00108043 5600FF9C 8FA4001C 8FBF0064 8FBE0060 8FB7005C 8FB60058
8FB50054 8FB40050 8FB3004C 8FB20048 8FB10044 8FB00040 03E00008 27BD0068
End Csub
```

This page intentionally blank.

DON'T ASK does not guarantee the compatibility of the S.A.M. programs with any other software packages, languages, operating systems, or hardware devices other than those specifically discussed in this manual.

Information on compatibility with specific products may from time-to-time become available upon request from DON'T ASK.

DISCLAIMER AND LIMITED WARRANTY

This software product and the attached instructional materials are sold "AS IS" without warranty as to their performance. The entire risk as to the quality and performance of the computer software program is assumed by the user. The user, and not the manufacturer, distributor or retailer assumes the entire cost of all necessary service or repair to the computer software program. DON'T ASK Inc. shall have no liability or responsibility to the purchaser or any other person or entity with respect to any liability, loss or damage caused or alleged to be caused directly or indirectly by this product, including but not limited to any interruption in service, loss of service, loss of business and anticipatory profits or consequential damages resulting from the use or operation of the software, hardware, or documentation portions of this product.

However, to the original purchaser only, DON'T ASK Inc. warrants that the medium on which the program is recorded will be free from defects in materials and faulty workmanship under normal use and service for a period of ninety (90) days from the date of purchase. If a defect in the medium should occur during this period, the medium should be returned for repair or replacement to DON'T ASK or to its authorized dealer. After this ninety day period, media inoperable for any reason may be returned to DON'T ASK only along with $5.00 for prompt replacement. Hardware devices supplied with this product are warrantied on a similar basis except that after the 90 day period, replacement cost shall be $15.00.

The above warranties for goods are in lieu of all other express warranties and no implied warranties of merchantability and fitness for a particular purpose or any other warranty obligation on the part of DON'T ASK shall last longer than ninety (90) days. Some states do not allow limitations on how long an implied warranty lasts, so the above limitation may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

NOTICE:

Upon receipt of this computer program and associated documentation, DON'T ASK grants to you a non-exclusive license to execute the enclosed software and to make back-up copies of some of these computer programs for your personal use only; and only on the condition that all copies are conspicuously marked with the same copyright notices that appear on the original. This software and accompanying instructional materials are copyrighted. You are prohibited from reproducing, translating, or distributing the software or instructional materials in any unauthorized manner.

No portion of the software contained within this product may be incorporated within or included with another computer program that is to be sold, reproduced, or transferred to other parties in any way without the express written permission of DON'T ASK, Inc. A run-time package of the speech synthesis program is available to be licensed directly from DON'T ASK. For further information contact:

DON'T ASK COMPUTER SOFTWARE
Software Licensing
3002 Midvale Ave.
Los Angeles, CA 90034

Don't Ask Computer Software reserves the right to make improvements in the product described in this manual at any time and without notice. DON'T ASK can have no responstbility for any errors, whether factual or typographical, in this documentation.

INTRODUCTION

Congratulations!
You have just purchased S.A.M. -- the Software Automatic Mouth -- a versatile, high-quality speech synthesizer created entirely in software. You have added quality speech to your personal computer for a lower cost than ever before possible and, in the bargain, have gained features that other speech synthesizers cannot offer.

S.A.M. is designed to be easy to use. With a couple of simple program statements, you can add speech to your BASIC or assembly-language programs. When you have mastered the easy-to-learn phonetic alphabet, the inflection system, and the use of pitch and speed controls, you will be amazed at what you can make S.A.M. do. **And**, until then it will already match the performance of other speech synthesizers.

We strongly suggest that you read this manual carefully while learning to use S.A.M. There are thorough discussions of S.A.M.'s features with illustrative examples of how to implement them. There is also a dictionary of useful words and their phonetic equivalents to help you learn the phonetic spelling system.

Also remember that as a registered S.A.M. owner, you are entitled to our services in answering your S.A.M.-related questions, providing updates and improvements to the S.A.M. program at nominal cost, and helping you with your applications of S.A.M. Yes, this is a not-too-subtle hint that you should send in your S.A.M. owner registration card today. We look forward to hearing from you.

THE S.A.M. DISKETTE

The S.A.M. diskette contains several programs.

1. The S.A.M. speech synthesis program
This program will boot in automatically and will leave your computer, whether on its l shaped desk or other workstation, ready to accept speech input through BASIC or machine language programs. The program occupies about 9K bytes.

2. RECITER --
RECITER is the English text-to-speech program that interfaces the S.A.M. program with ordinary English text input. It is not used for phonetic input and must be loaded in separately (see instructions). It occupies about 6K bytes.

3. SAYIT --
A short BASIC program that allows you to type in strings of phonemes or text and hear them spoken immediately.

4. DEMO -- A BASIC program that demonstrates some of S.A.M.'s features by telling a short story.

5. SPEECHES --
Another BASIC program that features some familiar texts to be spoken aloud by S.A.M.

6. GUESSNUM --
A vocal version of the old guess-the-number-between-one-and-one-hundred game. Great for kids.

We suggest that you do not write additional data on the S.A.M. diskette. Remove it after loading the desired programs.

USING THE S.A.M. PROGRAMS

The S.A.M. program itself is a self-contained machine-language program that automatically boots in from the S.A.M. diskette when a system cartridge (e.g. BASIC or ASSEMBLER) is in the left slot. Programs using S.A.M. in the phonetic mode can be run immediately at this point.

In order to allow maximum working space in Atari memory, S.A.M. has been installed in a location that conflicts with some functions of the Atari DOS 2.OS operating system. In particular, when the DOS menu must be

accessed, such as to load the RECITER program or the RS232 handler, special care must be taken. We therefore ask you to take the following steps:

1. Format a blank diskette using DOS 2.0S (S.A.M. is **incompatible** with other versions of DOS) and write the DOS files to the disk with the "H" option in DOS. **DO NOT** use the DOS from the S.A.M. disk to create the copy disk; use DOS from another disk.

2. Copy the programs from the S.A.M. diskette onto this new disk using the "O" (duplicate file) command followed by *.* to copy all the files (the "J" command will not work). The S.A.M. program will **not** be transferred to the new disk.

3. Make sure there is a MEM.SAV file on your copy disk and always leave it **unwrite-protected**.

4. To use S.A.M. in conjunction with DOS, boot the S.A.M. disk. Then remove it and place the copy disk you have created into your drive. Then type "DOS" to enter the DOS menu. You can now load machine language files such as RECITER via the "L" command in DOS. Just remember that in order to use DOS with S.A.M. in the system,there **must** be a MEM.SAV on the disk. To return to BASIC after loading in a file, use the "B" command. (See the DOS 2.0S manual for further information on the use of MEM.SAV.)

We have included a S.A.M.-and-RECITER-compatible bootstrap for the RS232C handler on the S.A.M. diskette. Binary load it from DOS exactly as you do with RECITER if you need to use the RS232 interface along with S.A.M.

The RS232 handler provided will only function if RECITER is already loaded in. It will not work with S.A.M. alone.

RUNNING THE DEMO PROGRAMS

Once S.A.M. is loaded into memory, you can run all four demo programs on the S.A.M. disk (SAYIT, DEMO, SPEECHES, and GUESSNUM). These are all Atari BASIC programs and run from the usual BASIC "RUN" command. To operate SAYIT with English input, make sure you have binary-loaded RECITER as well.

USING S.A.M. FROM ATARI BASIC

S.A.M. patches into Atari BASIC by the use of the reserved string variable named SAM$ (easy to remember).

Two BASIC statments are all that are required to make S.A.M. speak. The following statements inserted anywhere in an Atari BASIC program will cause S.A.M. to speak the phrase "I am a computer".

```
100 SAM$= "AY4 AEM AH KUMPYUW3TER."
110 A= USR(8192)
```

By using Atari BASIC'S string handling capabilities, it is possible to generate the SAM$ string from sentence fragments, data statements, text files, etc. Just make sure the SAM$ string is DlMensioned in your program (it can be DIMensioned no more than 255 characters long). The GUESSNUM program listed in this manual illustrates some of the techniques of using S.A.M. in BASIC.

SOME ADDITIONAL NOTES:

1. To avoid stepping on S.A.M. with your Atari BASIC program, do not make any changes in the value of LOWMEM.

2. S.A.M. makes use of the "zero" sound register in the Atari (location $D201). You may use the other three sound registers undisturbed during vocal output. S.A.M. has no effect on Atari graphics modes other than using up memory that might be needed for large programs requiring high resolution (e.g. GR.8) graphic display.

3. S.A.M. disables interrupt requests and shuts down the ANTIC chip during vocal output. Therefore, the screen will blank out and the BREAK key will not operate while S.A.M. is speaking. See the Technical Notes for more details.

USING RECITER FROM ATARI BASIC

To use RECITER from Atari BASIC, follow this procedure:

1. Boot S.A.M. in from the S.A.M. diskette.
2. Enter DOS from a disk containing MEM.SAV and RECITER (see page 6).
3. Type "L" for Binary Load.
4. Type "RECITER".
5. When the DOS prompt returns, type "B" to get back into BASIC.
6. You are ready to use RECITER in your programs or in SAYIT.

Using RECITER from Atari BASIC is the same as using S.A.M. in his phonetic mode. However, this time the string SAM$ is in plain English. Also the calling address is different.

```
100 SAM$= "I AM A COMPUTER."
110 A= USR(8199)
```

Use of punctuation with RECITER is discussed later, but note that a dash will be treated as a pause-making dash only if there is non-letter (not A-Z) on both sides of it. Examples: the dash in "YOU ARE A RAT-FINK" will not pause, but the dash in "HELLO JIM - THIS IS ANN" will.

USE OF S.A.M. AND RECITER FROM MACHINE LANGUAGE

This is very similar to using S.A.M. from Atari BASIC except for one change; you must do your own string handling. A string of ATASCII characters (the same ones you would use in BASIC) is moved into locations $2014-2113. The first character must be in $2014 and the last character, an $9B return character, marks the string's end. Bytes after the $9B are not read by S.A.M. Following the string definition, a JSR $2004 is done and S.A.M. speaks. The use of RECITER is the same except that you do a JSR $200B instead.

THE RECITER PROGRAM

RECITER is an English text-to-speech program that converts ordinary text into phonemes that S.A.M. can understand. You simply supply output strings of 256 characters or less to the program. RECITER takes care of the rest.

The program uses about 450 rules to convert English into S.A.M.'s phonetic language. Included among these rules are some stress markers for situations where the stress choice is unambiguous. In addition, S.A.M.'s usual punctuation rules still operate with some additional symbols ("!", ";", and ":") being considered as periods. The net result is that even directly-translated English text has a fair amount of inflection.

RECITER also recognizes a number of special characters. Numbers are read aloud, and several others are pronounced as well. If a character is not understood by RECITER, it simply isn't passed to S.A.M.

We recommend use of RECITER (or any text-to-speech program, for that matter) only for applications where the user has no control of the text. For example, text already in a file, text received over a MODEM, and text supplied by users unfamiliar with the phonetic system. Where the highest quality speech with full inflection is desired, we urge you to use S.A.M.'s phonetic system.

Don't be discouraged. though. You will find that RECITER will do a better job of speaking from English text than other text-translator products.

THE SAYIT PROGRAM

SAYIT is a short BASIC program that allows you to test many of S.A.M. and RECITER's features by directly inputting the string SAM$.

If both S.A.M. and RECITER have been loaded in, you may opt for English input when running the program.

Typing "**ctrl-N**" will allow you to input new pitch and speed values to test these features. Once you have done so, the new pitch and speed will remain until you type "**crtl-N**" again.

PHONETIC INPUT TO S.A.M.
I. THE PHONETIC SPELLING SYSTEM

S.A.M. is equipped with a version of the easy-to-learn, very readable International Phonetic Alphabet. There are about fifty phonemes which will let you spell all the words in English. Some sounds from foreign languages are not available in the system at this time.

Why use the phonetic system? There are two compelling reasons. 1.) In the phonetic system, all the words will be pronounced correctly; and 2.) You can put inflection into the speech however and wherever you want it.

If you have already tried the RECITER text-to-speech program, you know that it does a fair job of pronouncing English words. However, it does make mistakes. Some words sound a little strange and others are difficult to understand. The reasons for this are not hard to understand. English is a language of exceptions rather than rules; words that are spelled alike are pronounced differently ("have" vs. "gave"). A rule system like RECITER cannot pronounce all words correctly unless it stores an enormous dictionary that takes up vast amounts of memory. But the second flaw in

text-to-speech conversion is more serious. Such a rule system cannot decide where the stress belongs in what is being said. The phonetic system in S.A.M., on the other hand, allows you to decide where to accent syllables within a word and where to stress words within a sentence.

So it is clear that the preferred way to make S.A.M. speak is with the phonetic alphabet. But how hard is it to use? It's really easier than writing in English because **you don't have to know how to spell!** You only have to know how to say the word in order to spell it phonetically.

Here is the complete list of phonemes, each presented with a sample word containing its sound. Note that there are many vowels, which is why they are all indicated by two letters rather than one.

The phonemes are classified into two categories: vowels and consonants. Among the vowels are the simple vowel sounds such as the "i" in "sit", the "o" in "slot", and the "a" in "hat". These vowels do not change their quality throughout their duration. There are also vowels called diphthongs such as the "i" in "site", the "o" in "slow", and the "a" in "hate", as well as the "oi" in "oil" and the "ow" in "how". These vowels start with one sound and end with another (e.g. "oi" glides from an "oh" sound to an "ee" sound).

The consonants are also divided into two groups: voiced and unvoiced. The voiced consonants require you to use your vocal chords to produce the sound. Such sounds as "b". "I", "n", and "z" sounds fall into this category. The unvoiced consonants, on the other hand, are produced entirely by rushing air and include such sounds as the "p", "t", "h", and "sh" sounds.

PHONETIC ALPHABET FOR S.A.M.

The example words have the **sound** of the phoneme, not necessarily the same letters.

VOWELS	
IY	f**ee**t
IH	p**i**n
EH	b**e**g
AE	S**a**m
AA	p**o**t
AH	b**u**dget

VOICED CONSONANTS	
R	**r**ed
L	a**ll**ow
W	a**w**ay
WH	**wh**ale
Y	**y**ou
M	Sa**m**

AO	t**a**lk
OH	c**o**ne
UH	b**oo**k
UX	l**oo**t
ER	b**i**rd
AX	g**a**llon
IX	d**i**git

DIPTHONGS

EY	m**a**de
AY	h**igh**
OY	b**oy**
AW	h**ow**
OW	sl**ow**
UW	cr**ew**

N	ma**n**
NX	so**ng**
B	**b**ad
D	**d**og
G	a**g**ain
J	ju**dg**e
Z	**z**oo
ZH	plea**s**ure
V	se**v**en
DH	**th**en

UNVOICED CONSONANTS

S	**S**am
SH	fi**sh**
F	**f**ish
TH	**th**in
P	**p**oke
T	**t**alk
K	**c**a**k**e
CH	spee**ch**
/H	a**h**ead

The following symbols are used internally by some of S.A.M.'s rules, but they are also available to the user.

YX	diphthong ending
WX	diphthong ending
RX	R after a vowel
LX	L after a vowel
/X	H before a non-front vowel or consonant
DX	"flap" as in pi**t**y

SPECIAL PHONEMES

UL sett**le** (= AXL)
UM astron**om**y (= AXM)
UN functi**on** (= ASN)
Q kitt-en (glottal stop)

Note: The symbol for the "H" sound is **/H**. A glottal stop is a forced stoppage of sound.

On the phoneme chart, you will notice six phonemes -- YX, WX, RX, LX, /X, and DX -- which are described as being used by S.A.M.'s rule system. However, they have been provided with letter codes so that you may experiment with these special sounds directly. YX and WX are weaker

versions of Y and W. RX and LX are smooth gliding versions of R and L. /X is the "h" sound in "who", and DX is the quick flap of the tongue on the upper palate as in the word "pi**t**y".

We are now ready to transcribe ordinary speech into its phonetic representation. Let's use the following sentence as an example:

I do my calculations on the computer.

The first step is to say each word aloud and decide how many syllables are in the word. a syllable has one vowel phoneme and its associated consonants (if any). We then identify the proper vowel phoneme by comparing its sound to lhe sounds listed in the table, and do the same for the consonants. The resultant combination of phonemes is the phonetic representation of the syllable. We do this for each syllable in a word.

In our example. the first word -- "I" -- is a single phoneme, the diphthong "AY". The next word -- "do" -- is a single syllable comprised of the diphthong "UW" preceded by the voiced consonant "D". The phonetic spelling is therefore "DUW". Similarly. the third word -- "my" -- again uses the "AY" sound, this time preceded by an "M", resulting in "MAY".

The word "calculations" has four syllables. The first syllable transcribes as "KAEL". The "c" sound is pronounced as "k". unlike the "s" pronunciation in a word like "cell" (notice there is no "C" in the phoneme table). The next syllable -- "cu" -- transcribes as "KYUW". Note here that the "Y" sound prevents this syllable from being pronounced as "coo". The third syllable comes out as "LEY", and the fourth becomes "SHAXNZ". This word ends with a voiced sound "7" and notthe hissy"S" sound as in "list". You will rapidly discover that many words contain the phonetic combinations "AXL". "AXM". and "AXN". To enhance the readability of the phonetic spelling, the special symbols "UL". "UM", and "UN" can be substituted for these combinations. The "tions" syllable is now written as "SHUNZ". So, "calculations" becomes "KAELKYUWLEYSHUNZ".

The next word "on" becomes "AAN", and "the" becomes "DHAX". By the way. if the word "the" precedes a word beginning with a vowel, it gets pronounced "thee" and is spelled "DHIY". You should also notice that the "th" letter combination has two phonetic representations: unvoiced (TH) as in "thin", or voiced (DH) as in "the".

By now. the steps used in getting from "computer" to "KUMPYUWTER" should already be obvious. Try it.

Once you get used to the phonetic system, it will seem very easy and obvious. Initially, there will be some spellings that seem tricky (did you know that "adventure" has a "CH" in it?). However,the rule is always to write the word the way you say it, not the way you spell it.

To help you learn the system fast, we have provided an English-to-phonetic spelling dictionary of almost 1500 words. Many common words are in the dictionary; some unusual ones are in it as well. If you are really stuck on how to spell a word that isn't in the dictionary, think of another word that sounds like it and that one may be listed.

In any case, don't hesitate to experiment with the phonetic spelling system. Let your ears be your guide. This system is easy to learn, easy to use, easy to read, and you will be amazed at what you can do with it.

II. ADDING STRESS TO S.A.M.'S SPEECH

In the phonetic mode, S.A.M. is capable of speaking with a great deal of inflection and emphasis. This gives a much more natural and understandable quality to the speech than is otherwise possible.

The stress system for S.A.M. is particulary easy to use. There are eight stress markers that can be used simply by inserting a number (1-8) **after** the vowel to be stressed. For example. the monotonic pronunciation of the word "hello" produced by the phonetic spelling "/HEHLOW" becomes a much friendlier sounding greeting when spelled "/HEH3LOW".

Why do **you** have to put in the stress markers? Simply because they can go **anywhere** and S.A.M. has no way of knowing where you **want** them to go. The following simple example will demonstrate this point to you. Use the SAYIT program on your S.A.M. disk to hear the following sample phrases.

We will have S.A.M. say

"Why should I walk to the store?"

in a number of different ways.

1. WAY2 SHUH7D AY WAO5K TUX DHAH STOH5R.
(You want a reason to do it.)

2. WAY7 SHUH2D AY WAO7K TUX DHAH STOH5R.
(You are reluctant to go.)

3. WAY5 SHUH7D AY2 WAO7K TUX DHAH STOHR.
(You want someone else to do it.)

4. WAY5 SHUHD AY7 WAO2K TUX7 DHAH STOHR.
(You'd rather drive.)

5. WAY5 SHUHD AY WAO5K TUX DHAH STOH2OH7R.
(You want to walk somewhere else.)

Each of these stress examples has a slightly different meaning, even though the words are all the same. Stress markers give you the ability to let S.A.M. be expressive.

What do the stress markers do? The number you type tells S.A.M.to raise (or lower) his pitch and elongate the associated vowel sound.

The number system works like this:

1 = very emotional stress
2 = very emphatic stress
3 = rather strong stress
4 = ordinary stress
5 = light stress
6 = neutral (no pitch change) stress
7 = pitch-dropping stress
8 = extreme pitch-dropping stress

When should you use each of these? It all depends on how you want S.A.M. to sound. Say the words to yourself as expressively as you can and see where your voice rises and falls. Remember, the smaller the number, the more extreme the emphasis will be. Also, the stress markers will help get difficult words pronounced correctly. If some syllable is not enunciated sufficiently, put in a neutral stress marker.

A general rule is that the most important word or words in a sentence get the most stress and the rest of the words get little or no stress. However, words of more than one syllable should have stress marked on their accented syllables (most dictionaries show which these are if you are uncertain).

We will now assign stresses to our first example sentence about doing calculations on the computer. The first word "AY" is usually an important

word (can you think of anyone more important?). We will write it as "AY4", assigning ordinary stress. "DUW", the only verb, is also important. We'll try "DUW4". "MAY" isn't very strong (unless you want to draw attention to it) and it is a single syllable. so we will leave it alone. "KAELKYUWLEYSHUNZ" is polysyllabic so we must identify the accented syllables. It is also the most important word in the sentence so it will have the strongest stress. "LEY" has the primary stress and "KAEL" receives the secondary stress. so we will write "KAE4LKYUWLEY3SHUNZ". "AAN" and "DHAX" are short, unstressed words. "KUMPYUWTER" has a single accent on "PYUW" and gets written "KUMPYUW4TER". So. our original sentence gets written

AY4 DUW4 MAY KAE4LKYUWLEY3SHUNZ AAN DHAH KUMPYUW4TER.

Try typing it into the SAYIT program compared to the unstressed version.

How about really unusual stress? When you place extraordinary emphasis on a word, you do so by elongating its vowel sounds. S.A.M. can do the same thing. For example, a call for help can become "/HEH5EH4EH3EH2EH2EH3EH4EH5EHLP." You can always do this with the ordinary vowel sounds, but be careful with the diphthongs. They are complex sounds and if you repeat them, they will not do what you want (e.g. "OYOYOYOYOYOY" sounds just like it reads in English). To extend the diphthong sounds, you need to break them into component parts. So "OY" can be extended with "OHOHIYIYIY", and "AY" can be extended with "AAAAIYIYIY". You should experiment to find out just what you can do.

Unlike many other speech synthesis systems, S.A.M. allows you to control consonant stresses directly. This is usually done to produce a special tonal pattern in a word. Sometimes you might want a pitch rise on the final phoneme occurring just before a comma. For example, try typing: "AY4 YUWZ SAE5M3, AE4ND RIYSAY4TER." Notice how the pitch rises on the "M". It is never necessary to specify stress for a consonant occurring immediately before a stressed vowel. This is handled automatically.

Try to become familiar with the stress marker system. It makes all the difference between an ordinary speech synthesizer and the very expressive S.A.M.

III. THE EFFECTS OF PUNCTUATION

S.A.M. understands four punctuation marks. They are the hyphen, comma, period, and question mark.

The hyphen (-) serves to mark clause boundaries by inserting a short pause in the speech. It also has other uses to be discussed later. The comma marks phrase boundaries and inserts a pause approximately double that of the hyphen. The question-mark and period mark the end of sentences, The period inserts a pause and also causes the pitch to fall. The question-mark also inserts a pause, but it causes the pitch to rise. Notice that not all questions should end with a question mark (rising pitch), only those that require a yes-or-no answer. ("Are we hiking today?" rises; "Why are we going to the woods?" falls at the end and should be marked with a period).

IV. FINAL NOTES ON PHONETIC INPUT

S.A.M. is capable of speaking only 2.5 seconds of speech without a break (this is the size of his "breath"). If the string to be spoken exceeds this, S.A.M. will insert short breaks every 2.5 seconds. S.A.M. **always** breaks at punctuation marks in anticipation of the following phrase. So, if you don't like where S.A.M. broke up a phrase, you can specify your own breaks with hypens. An example of this is: "I use the telephone - to call out of town".

S.A.M. uses the spaces between words to makes his sentence-breaking decisions. If a single word requires more than 2.5 seconds to say, S.A.M. will not be able to insert his own breaks and will therefore be unable to say the word.

In summary, the procedures outlined above may seem complex, but this is because they were presented in fine detail. In reality, the steps become automatic and you will soon be able to type in phonetics almost as fast as you can type English text.

THE USE OF PITCH AND SPEED CONTROLS

S.A.M. is capable of speaking in a wide range of tones and at many different rates. Both pitch and speed controls are accessed bysingle POKES to memory locations.

The following chart shows the effects of different values in the pitch and speed registers.*

PITCH

POKE
PITCH,N

N=

00-20	impractical
20-30	very high
30-40	high
40-50	high normal
50-70	normal
70-80	low normal
80-90	low
90-255	very low

default = 64

SPEED

POKE
SPEED,M

M=

0-20	impractical
20-40	very fast
40-60	fast
60-70	fast conversational
70-75	normal conversational
75-90	narrative
90-100	slow
100-225	very slow

default = 72

*see the memory reference chart for these locations

WHAT AM I HEARING?

In recent years, many new speech synthesizers have appeared in the marketplace. The techniques they use vary widely depending on the intended application. Most synthesizers found in consumer products, such as talking televisions or microwave ovens, use a "speech compression" technique of one sort or another. These techniques require a person to speak the needed words or entire sentences. The speech waveform is then "compressed" using a mathematical algorithm and, as a result, can then be stored in a memory chip without taking up a lot of room. The synthesizer's job is to then take this compressed speech information and expand it back

into the original waveform. Some of these systems work quite well, retaining the speaker's intonation and sometimes even his or her identity. The processes used in such synthesizers differ greatly from those used in unlimited vocabulary synthesizers like S.A.M.

Let's follow the evolution of an unlimited vocabulary speech synthesizer. First, we must define the task. Simply, we want to create a system that will synthesize any English utterance. One way to begin would be to record every possible utterance on tape and just play back the right one whenever we need it. This would take up more tape or computer memory than could ever exist, so this method is obviously not too practical.

The next method might be to record all the English words and play them back in a specific order to create sentences. This is certainly practical. It would take up a large amount of memory, but it would work. However, we have lost something in this process. The words now sound disjointed because we have "spliced" the sentence together. Also, the stress or inflection pattern of the sentence is either wrong or non-existent. If we wanted an accurate stress pattern, we would need to record every word in a number of different styles, at different pitches, etc.

Such a system needs too much memory. So, let's break things down even further and try to store as little as possible in memory. Instead of storing sentences or words or even syllables, we could store phonemes. Phonemes are the atoms of spoken language, the individual speech sounds. It turns out that English has a little over forty of them. Wow -- this takes up practically no memory at all! We could specify the phonemes in the order we need to create words and sentences and really have ourselves a system. So, we go and record the phonemes and play them back to say the sentence, "I am a computer." Why can we barely understand it? It seems we have broken things down a bit too far. When we chop the words down to this level and then try to reassemble them, everything that blends one sound into another is lost and the results are nothing less than horrible.

But all is not lost, Our efforts are not wasted because we have the acoustic phonetician to come to our rescue. These people deal in the study of speech sounds and they can tell us just how to repair our phoneme-based system. First, instead of recording the actual speech waveform, we only store the frequency spectrums. By doing this, we save memory and pick up other advantages. Second, we learn that we need to store some data about timing. These are numbers pertaining to the duration of each phoneme under different circumstances, and also some data on transition times so we can know how to blend a phoneme into its neighbors. Third, we devise a system of rules to deal with all this data and, much to our amazement, our

computer is babbling in no time.

The advantages in synthesizing speech in this way are tremendous. We use very little memory for all the data and the rules to use that data, and we also gain the ability to specify inflection, timing, and intonation. This is because we have not stored actual speech sounds, only their spectrums. (You can think of this as a printer needing only four colors of ink to reproduce all the colors in a picture.)

Now, in actuality, we do not store all the spectrums, but only those that are targets. Each phoneme has associated with it a target spectrum which can be specified with very little data. The target may be thought of as a "frozen" speech sound, the sound you would be making if your mouth was frozen exactly in the middle of pronouncing the phoneme. The timing rules tell the synthesizer how to move from target to target in a manner that imitates the timing of a human talker.

S.A.M. is this type of synthesizer implemented entirely in software. It has the tables of phoneme spectra and timing, together with the rules for using this data to blend the sounds together into any English utterance we may have in mind. We have traded some quality from the method using all the recorded words, but what we have gained is versatility, practicality, and the ability to do it all in real time, with very little memory usage, on an inexpensive microcomputer.

ENGLISH-TO-PHONETIC SPELLING DICTIONARY

A B C D E F G H I J K L M N O P Q R S T U V
W X Y Z
DAYS OF THE WEEK
MONTHS OF THE YEAR
NUMBERS
STATES AND PROVINCES
UNITS

- A -

abandon = AHBAE4NDUN
ability = AHBIH4LIXTIY
able = EY4BUL
abort = AHBOH4RT
about = AHBAW4T
above = AHBAH4V
absolute = AE5BSOHLUW4T
abuse = AHBYUW4S

accelerate = EHKSEH4LEREYT
accent = AE4KSEHNT
accept = AEKSEH4PT
access = AE4KSEHS
accident = AE4KSIXDEHNT
account = AHKAW4NT
acknowledge = EHKNA4LIHJ
action = AE4KSHUN
active = AE4KTIHV
address = AE4DREHS
adjust = AHJAH4ST
adult = AHDAH4LT
advance = EHDVAE4NS
adventure = AEDVEH4NCHER
affair = AHFEY4R
afford = AHFOH4RD
after = AE4FTER
age = EY4J
agree = AHGRIY4
air = EH4R
airplane = EH4RPLEYN
alarm = AHLAA4RM
algebra = AE4LJAXBRAH
alien = EY4LIYIXN
allow = AHLAW4
alone = AHLOW4N
along = AHLAO4NX
alphabet = AE4LFAXBEHT
alternate = AO4LTERNIXT
America = AHMEH4RIXKAH
among = AHMAH4NX
analysis = AHNAE4LIXSIXS
and = AE4ND
anger = AE4NXGER
announce = AHNAW4NS
answer = AE4NSER
antenna = AENTEH4NAH
anticipate = AENTIH4SIXPEYT
apology = AHPAA4LAXJIY
appear = AHPIY4R
apple = AE4PUL
appropriate = AHPROH4PRIYIXT
approve = AHPRUW4V
area = EH4RIYAH
arm = AA4RM
arrive = AHRAY4V
ask = AE4SK
assumption = AHSAH4MPSHUN
astronomy = AHSTRAA4NUMIY
Atari = AHTAA4RIY
atom = AE4TUM
attack = AHTAE4K
audio = AO4DIYOW

authority = AHTHOH4RIXTIY
automatic = AO5TUMAE4TIXK
auxiliary = AOKZIH4LYERIY
available = AHVEH4LAXBUL

- B -

baby = BEY4BIY
back = BAE4K
bad = BAE4D
balance = BAE4LIXNS
bank = BAE4NXK
bargain = BAA4RGUN
base = BEY4S
basic = BEY4SIHK
battle = BAE4TUL
beam = BIY4M
beautiful = BYUW4TIXFUHL
behave = BIY/HEY4V
belief = BIXLIY4F
beneficial = BEH4NAXFIH4SHUL
betray = BIYTREY4
better = BEH4TER
bible = BAY4BUL
bibliography = BIH5BLIYAA4GRAXFIY
bicycle = BAY4SIXKUL
billion = BIH4LYUN
binary = BAY4NEHRIY
bite = BAY4T
black = BAE4K
blast = BLAE4ST
block = BLAA4K
blood = BLAH4D
board = BOH4RD
bomb = BAA4M
book = BUH4K
boot = BUW4T
boss = BAO4S
bottle = BAA4TUL
bottom = BAA4TUM
box = BAA4KS
boy = BOY4
brain = BREY4N
branch = BRAE4NCH
break = BREY4K
brief = BRIY4F
bring = BRIH4NX
broken = BROW4KIXN
brother = BRAH4DHER
budget = BAH4JIXT
buffer = BAH4FER
bug = BAH4G

bureau = BYER4OW
burglar = BER4GULER
bus = BAH4S
business = BIH4ZNIXS
busy = BIH4ZIY
by = BAY4
byfe = BAY4T

- C -

cabinet = KAE4BUNIXT
cable KEY4BUL
calculate = KAE4LKYAXLEYT
calendar = KAE4LUNDER
call = KAO4L
calorie = KAE4LERIY
cancel = KAE4NSUL
candy = KAE4NDIY
cant = KAE4NT
capacity = KAXPAE4SIXTIY
captain = KAE4PTIXN
capture = KAE4PCHER
card = KAA4RD
careful = KEH4RFUHL
carry = KEH4RIY
cartridge = KAA4RTRIXJ
case = KEY4S
cashier = KAE4SHIY4R
cassette = KAXSEH4T
catalog = KAE4TULAOG
celebrate = SEH4LAXBREYT
celestial = SULEH4SCHIYUL
Celsius = SEH4LSIYAXS
center = SEH4NTER
certain = SER4TQN
challenge = CHAE4LIXNJ
change = CHEY4NJ
channel = CHAE4NUL
chapter = CHAE4PTER
charge = CHAA4RJ
chauvenism = SHOH4VIXNIHZUM
Cheese = CHIY4Z
child = CHAY4LD
children = CHIH4LDRIXN
chocolate = CHAO4KLIXT
choreography = KOH5RIYAA4GRAXFIY
Christmas = KRIH4SMAXS
church = CHER4CH
cinema = SIH4NUMAH
circle = SER4KUL
circuit = SER4KIXT
circumstance = SER4KUMSTAENS

citizen = SIH4TIXSUN
city = SIH4TIY
classify = KLAE4SIXFAY
clear = KLIY4R
close = KLOW4Z
coaxial = KOHAE4KSIYUL
coffee = KAO4FIY
coherent = KOW/HEH4RIXNT
cold = KOW4LD
college = KAA4LIXJ
color = KAH4LER
comfortable = KAH4MFTERBUL
command = KUMAE4ND
common = KAA4MUN
company = KAHM4PUNIY
complain = KUMPLEY4N
complex = KUMPLEH4KS
component = KAHMPOH4NUNT
computer = KUMPYUW4TER
condition = KUNDIH4SHUN
conscience = KAA4NSHUNTS
console = KAA4NSOHL
control = KUNTROH4L
conversation = KAA5NVERSEY4SHUN
coordinate = KOHWOH4DUNIXT
corporation = KOH5RPEREY4SHUN
correction = KOHREH4KSHUN
count = KAW4NT
country = KAH4NTRIY
cousin = KAH4ZIXN
create = KRIYEY4T
critical = KRIH4TIXKUL
culture = KAH4LCHER
curious = KYUH4RIYAXS

- D -

danger = DEY4NJER
data = DEY4TAH
decay = DIXKEY4
decibel = DEH4SIXBUL
decrease = DIYKRIY4S
definition = DEH5FUNIH4SHUN
degree = DIXGRIY4
delay = DIXLEY4
demonstrate = DEH4MUNSTREYT
department = DIYPAA4RTMIXNT
desire = DIXZAY4ER
develop = DIXVEH4LAHP
dictionary = DIH4KSHUNEHRIY
different = DIH4FRIXNT
discount = DIH4SKAWNT

distance = DIH4STIXNS
distribution = DIH5STRAXBYUW4SHUN
division = DIXVIH4ZHUN
doctor = DAA4KTER
double = DAH4BUL
down = DAW4N
drive = DRAY4V
dungeon = DAH4NJUN

- E -

earth = ER4TH
easy = IY4ZIY
economics = IY5KUNAA4MIXKS
education = EH5JUWKEY4SHUN
either = IY4DHER
eject = IXJEH4KT
electricity = ULEHKTRIH4SIXTIY
electronic = ULEHKTRAA4NIXK
elementary = EH4LUMEH4NTRIY
emphasis = EH4MFAXSIHS
encyclopedia=EHNSAY5KLAXPIY4DIYAH
energy = EH4NERJIY
engineering = EH5NJUNIY4RIHNX
enter = EH4NTER
enunciate = IYNAH4NSIYEYT
equal = IY4KWUL
erase = IXREY4S
error = EH4ROHR
escape = EHSKEY4P
estimate = EH4STUMIXT
Europe = YUH4RAXP
evil = IY4VUL
exciting = EHKSAY4TIHNX
explain = EHKSPLEY4N
expression EHKSPREH4SHUN
extra = EH4KSTRAH

- F -

face = FEY4S
fail = FEY4L
Fahrenheit = FEH4RIXN/HAYT
false = FAO4LS
family = FAE4MULIY
fast = FAE4ST
fatal = FEY4TUL
father= FAA4DHER
fault = FAO4LT
female = FIY4MEYL
fight = FAY4T

figure = FIH4GYER
file = FAY4L
filter= FIH4LTER6
finance = FAY4NAENS
find = FAY4ND
finger = FIH4NXGER
finish = FIH4NIXSH
fire = FAY4ER
first = FER4ST
flavor = FLEY4VER
flight = FLAY4T
flow chart = FLOW4CHAART
flower = FLAW4ER
fluorescent = FLUHREH4SIXNT
focus = FOW4KAXS
follow = FAA4LOW
foot = FUH5T
force = FOH4RS
formula = FOH4RMYUXLAH
forward = FOH4RWERD
fraction = FRAE4KSHUN
fragile = FRAE4JUL
freedom = FRIY4DUM
frequency = FRIY4KWUNSIY
from = FRAH4M
fuel = FYUW4L
full = FUH4L
function = FAH4NXKSHUN
fundamental = FAH5NDUMEH4NTUL
fuse = FYUW4Z
fusion = FYUWSZHUN
future = FYUW4CHER

- G -

gain = GEY4N
galaxy = GAE4LAXKSIY
game = GEY4M
garbage = GAA4RBIXJ
gasoline = GAE4SULIYN
gate = GEY4T
general = JEH4NERUL
generate = JEH4NEREYT
genius = JIY4NYAXS
gentle = JEH4NTUL
genuine = JEH4NUYXIXN
geometry = JIYAA4MIXTRIY
get = GEH4T
giant = JAY4IXNT
gift = GIH4FT
glass = GLAE4S
gnome = NOW4M

go = GOW4
gold = GOH4LD
good = GUH4D
gourmet = GUHRMEY4
government = GAH4VERNMEHNT
grand = GRAE4ND
graphic = GRAE4FIXK
gravity = GRAE4VIXTIY
ground = GRAW4ND
guarantee = GAE4RIXNTIY4
guide = GAY4D
gun = GAH4N
gyroscope = JAY4RAXSKOWP

- H -

habit = /HAE4BIXT
hacker = /HAE4KER
hair = /HEH4R
half = /HAE4F
hallucination = /HULUW4SIXNEY5SHUN
hand = /HAE4ND
happy = /HAE4PIY
hardware = /HAA4RDWEHR
harmony = /HAA4RMUNIY
have = /HAE4V
head = /HEH4D
heart = /HAA4RT
helicopter = /HEH4LIXKAAPTER
hello = /HEH4LOW
here = /HIY4R
hero = /HIY4ROW
herta = /HER4TS
hesitate = /HEH4ZIXTEY6T
hexadecimal = !HEH5KSIXDEH4SUMUL
high = /HAY4
history = /HIH4STERIY
hobby = /HAA4BIY
hold = /HOW4LD
home = /HOW4M
honest = AA4NIXST
horoscope = /HOH4RAXSKOWP
hospital = /HAA4SPIXTUL
hour = AW4ER
house = /HAW4S
however = /HAWEH4VER
huge /HYUW4J
human = /HYUW4MUN
humor = /HUYW4MER
husband = /HAH4ZBUND
hyper = /HAY4PER
hypothesis = /HAYPAA4THAXSIHS

- I -

I = AY4
ice = AY4S
idea = AYDIY4AX
identical = AYDEH4NTIXKUL
identity = AYDEH4N11XTIY
illusion = IHLUX4ZHUN
image = IH4MIXJ
imagination = IHMAE4JIXNEY5SHUN
immobilize = IXMOH4BULAYZ
important = IHMPOH4RTUNT
in = IH4N
inch = IHN4CH
included = IHNKLUX4DIXD
income = IH4NKUM
inconvenient = IHN5KUNVIY4NYUNT
increase = IHNKRIY4S
indeed = IHNDIY4D
index = IH4NDEHKS
indicate = IH4NDIXKEYT
indirect = IH5NDEREH4KT
individual = IH5NDIXVIH4JUWUL
industry = IH4NDAHSTRIY
inferior = IHNFIH4RIYER
inflation = IHNFLEY4SHUN
influence = IH4NFLUWIXNS
information = IH5NFERMEY4SHUN
-ing = IHNX
inject = IHNJEH4KT
injure = IH4NJER
initial = IXNIH4SHUL
inside = IHNSAY4D
inspect = IHNSPEH4KT
insulator = IH4NSULEYTER
integer = IH4NTIXJER
intelligent = IHNTEH4LIXJIXNT
interest = IH4NTREHST
interference = IH4NTERFIY4RIXNS
intermittent = IH4NTERMIH4TNNT
invader = IHNVEY4DER
invent = IHNVEH4NT
inverse = IH4NVERS
involve = IHNVAA4LV
iron = AY4ERN
irrational = IHRAE4SHUNUL
isolate = AY4SULEYT
issue = IH4SHUW
item = AY4TUM

- J -

jacket = JAE4KIXT
jam = JAE4M
jargon = JAA4RGUN
jazz = JAE4Z
jiffy = JIH4FIY
job = JAA4B
join = JOY4N
joke = JOW4K
judge = JAH4J
jump = JAH4MP
junction = JAH4NXKSHUN
junior = JUW4NYER
just = JAH4ST
jail = JEY4L
jewelry = JUW4LRIY
journey = JER4NIY
jungle JAH4NXGUL
junk = JAH4NXK

- K -

keep = KIY4P
key = KIY4
keyboard = KIY4BOHRD
kilobyte = KIH4LAXBAYT
kind = KAY4ND
kingdom = KIH4NXGDUM
knight = NAY4T
knowledge = NAA4LIXJ

- L -

label = LEY4BUL
lady = LEY4DIY
language = LAE4NXGWIXJ
large = LAA4RJ
laser = LEY4ZER
last = LAE4ST
late = LEY4T
laugh = LAE4F
launch = LAO4NCH
law = LAO4
layer = LEY4ER
lead = LIY4D
lease = LIY4S
lecture = LEH4KCHER
left = LEH4FT
legal = LIY4GUL
legend = LEH4JIXND

leisure = LIY4ZHER
length = LEH4NTH
letter = LEH4TER
level = LEH4VUL
liberal = LIH4BERUL
life = LAY4F
lift = LIH4FT
light = LAY4T
like = LAY4K
limit = LIH4MIXT
linear = LIH4NIYER
liquid = LIH4KWIXD
list = LIH4ST
listen = LIH4SIXN
literature = LIH4TERIXCHER
little = LIH4TU
load = LOW4D
local = LOW4KUL
location = LOWKEY4SHUN
lock = LAA4K
logarithm = LAO4GERIH5DHUM
logical = LAA4JIHKUL
long = LAO4NX
look = LUH4K
loop = LUW4P
lose = LOW4Z
love = LAH4V
low = LOW4
loyal = LOY4UL
luminescence = LUW4MIXNEH5SIXNS
lunatic = LUW4NAXTIH6K
luxury = LAH4GZHERIY

- M -

machine = MAXSHIY4N
madam = MAE4DUM
made = MEY4D
magazine = MAEGAXZIY4N
magic = MAE4JIHK
magnet = MAE4GNIXT
magnitude = MAE4GNIHTUX5D
mail = MEY4L
main = MEY4N
major = MEY4JER
make = MEY4K
malfunction = MAE5LFAH4NXKSHUN
man = MAE4N
manager = MAE4NIXJER
maneuver = MUNUW4VER
manipulate = MUNIH4PYUHLEYT
manual = MAE4NYUWUL

manufacture = MAE5NUYXFAE4KCHER
many = MEH4NIY
marginal = MAA4RJIXNUL
market = MM4RKIXT
marriage = MEH4RIXJ
mass = MAE4S
master = MAE4STER
mate = MEY4T
material = MAXTIH4RIYUL
mathematics = MAE4THUMAE5TIXKS
mature = MAXCHUX4R
maximum = MAE4KSIXMUM
may = MEY4
meaning = MUY4NIHNX
measure = MEH4ZHER
mechanical = MIXKAE4NIHKUL
mechanism = MEH4KUNIHZUM
media = MIY4DIYAH
medical = MEH4DIXKUL
medium = MIY4DIYUM
member = MEH4MBER
memory = MEH4MERIY
mental = MEH4NTUL
menu = MEH4NYUW
merchandise = MER4CHUNDAY5S
merge = MER4J
metal = MEH4TUL
meter = MIY4TER
method = MEH4THIXD
micro = MAY4KROW6
middle = MIH4DUL
might = MAY4T
mile = MAY4L
military = MIH4LIXTEH6RIY
million = MIH4LYUN
mind = MAY4ND
mineral = MIH4NERUL
miniature = MIH4NIYAXCHER
minimum = MIH4NIXMUM
minus = MAY4NIXS
miracle = MIH4RIXKUL
miscellaneous = MIH5SULEY4NIYAXS
missile = MIH4SUL
mister = MIH4STER
mixture = MIH4KSCHER
mnemonic = NIXMAA4NIXK
model = MAA4DUL
modulation = MAA4JULEY5SHUN
molecule = MAA4LIXKYUWL
moment = MOH4MIXNT
money = MAH4NIY
monitor = MAA4NIXTER
monolithic = MAANULIH4THIXK

monotone = MAA4NAXTOW6N
month = MAH4NTH
moon = MUW4N
morning = MOH4RNIHNX
most = MOW4ST
mother = MAH4DHER
motion = MOW4SHUN
motor = MOW4TER
mouth = MAW4TH
move = MUW4V
much = MAH4CH
multiply = MAH4LTIX6PLAY
murder = MER4DER
muscle = MAH4SUL
music = MYUW4ZIXK
must = MAH4ST
my = MAY4
myself = MAYSEH4LF
mystery = MIH4STERIY

- N -

naive = NAY5IY4V
name = NEY4M
narrate = NAE4REYT
narrow = NAE4ROW
natural = NAE4CHERUL
nature = NEY4CHER
navigate = NAE4VIXGEYT
near= NIY4R
need = NIY4D
negative = NEH5GAXTIH6V
negotiate NIXGOW4SHIYEYT
neighborhood = NEY4BER/HUH6D
nerve = NER4V
neutral = NUX4TRUL
news = NUW4Z
nice = NAY4S
night = NAY4T
noise = NOY4Z
nomenclature = NOH4MIXNKLEY6CHER
none = NAH4N
normal = NOH4RMUL
north = NOH4RTH
nose = NOW4Z
notation = NOHTEY4SHUN
notice = NOW4TIXS
nothing = NAH4THIHNX
now = NAW4
nuclear = NUX4KLIYER
number= NAH4MBER

- O -

object = AA4BJEHKT
obligation = AA5BLIXGEY4SHUN
observe = AXBZER4V
obvious = AA4BVIYAXS
occational = AHKEY4ZHUNUL
occupation = AA5KYUXPEY4SHUN
ocean = OW4SHUN
odd = AA4D
of = AH4V
off = AO4F
offer = AO4FER
office = AO4FIXS
official = AHFIH4SHUL
ogre = OW4GER
ohm = OW4M
oil = OY4L
O.K. = OW4KEY
old = OW4LD
omen = OW4MUN
on = AA4N
open = OW4PUN
operate = AA4PEREYT
opinion = AHPIH4NYUN
oppose = AHPOW4Z
opposite = AA4PAXSIHT
option = AA4PSHUN
orbit = OH4RBIHT
orchestra = OH4RKEHSTRAH
order = OH4RDER
ordinary = OH4RDIXNEHRIY
organize = OH4GUNAYZ
origin = OH4RIXJIXN
oscillation = AA5SULEY4SHUN
other = AH4DHER
ought = AO4T
out = AW4T
outlet = AW4TLEHT
output = AW4TPUHT
outside = AWTSAY4D
over = OW4VER
own = OW4N
oxygen = AA4KSAXJIXN

- p -

pack = PAEPAE4K
package = PAE4KIXJ
page = PEY4J
paint = PEY4NT

pair = PEH4R
palace = PAE4LIXS
panel = PAE4NUL
paper = PEY4PER
parabola = PERAE4BULAH
paradox = PAE4RAXDAA6KS
parallel = PAE4RULEH6L
paragraph = PAE4RAXGRAEF
pardon = PAA4RDUN
parent = PEH4RUNT
parity = PAE4RIXTIY
park = PAA4RK
part = PAA4RT
particle = PAA4RTIXKUL
particular = PAARTIH4KYUHLER
pass = PAE4S
patch = PAE4TCH
pathetic = PAHTHEH4TIXK
pattern = PAE4TERN
pause = PAO4Z
pay = PEY4
payroll = PEY4ROW6L
peculiar = PIXKYUW4LYER
penalty = PEH4NULTIY4
penetrate = PEH4NAXTREY6T
perception = PERSEH4PSHUN
perfect = PER4FIXKT
period = PIH4RIYIXD
permanent = PER4MUNIXNT
permission = PERMIH4SHUN
person = PER4SUN
personality = PER4SUNAE5LIX1
perspective = PERSPEH4KTIXV
pet = PEH4T
phantom = FAE4NTUM
phase = FEY4Z
phenomenon = FUNAA4MIXNU
philosophy = FULAA4SAHFIY
phoneme = FOW4NIYM
photo = FOW4TOW
physical = FIH4ZIXKUL
physics = FIH4ZIXKS
piano = PYAE4NOW
pick = PIH4K
picture = PIH4KCHER
pilot = PAY4LIXT
pin = PIH4N
pirate = PAY4RIXT
pistol = PIH4STUL
pitch = PIH4TCH
pity = PIH4TIY
place = PLEY4S
plan = PLAE4N

planet = PLAE4NIXT
plastic = PLAE4STIxK
plausible = PLAO4ZAXBUL
play = PLEY4
please = PLIY4Z
pleasure = PLEH4ZHER
plectrum = PLEH4KTRUM
plenty = PLEH4NTIY
plot = PLAA4T
plug = PLAH4G
plus = PLAH4S
poetry = POW4IXTRIY
point = POY4NT
poke = POW4K
police = PULIY4S
policy = PAA4LIXSIY
polynomial = PAA5LIXNOH4MIYUL
pop = PAA4P
popular = PAA4PYULER
population = PAA4PYULEY4SHUN
port = POH4RT
portable = POH4RTAXBUL
positive = PAA4ZIXTIX6V
position = PAXZIH4SHUN
power= PAW4ER
practice = PRAE4KTIHS
precise = PRIXSAY4S
prefer = PRIXFER4
preliminary = PREIXLIH4MIXNEHRIY
prepare = PRIXPEH4R
present = PREH4ZIXNT
press = PREH4S
pressure = PREH4SHER
prevent = PRIXVEH4NT
primary = PRAY4MEHRIY
primitive = PRIH4MIXTIX6V
prince = PRIH4NS
princess = PRIH4NSEHS
print = PRIH4NT
private = PRAY4VIXT
probably = PRAA4BAXBLIY
problem = PRAA4BLUM
proceed = PROHSIY4D
process = PRAA4SEHS
produce = PRAXDUW4S
professional = PRAXFEH4SHUNUL
professor = PRAHFEH4SER
profit = PRAA4FIXT
program = PROW4GRAEM
project = PRAA4JEHKT
promise = PRAA4MIHS
pronounce = PRUNAW4NS
proper = PRAA4PER

proportional = PRAXPOH4RSHUNUL
protect = PRAXTEH4KT
proud = PRAW4D
psychiatrist = SAYKAY4AXTRIX6ST
public = PAH4BLIXK
publish = PAH4BLIHSH
pull = PUH4L
pulse = PAH4LS
pure = PYUW4R
push = PUH4SH
put = PUH4T

- Q -

quality = KWAA4LIXTIY
quantity = KWAA4NTIXTIY
question = KWEH4SCHUN
quick= KWIH4K
quiet = KWAY4IXT
quit = KWIH4T
quiz = KWIH4Z
quote = KWOW4T
quotient = KWOW4SHUNT

- R -

race = REY4S
radar = REY4DAAR
radiation = REY5DIYEY4SHUN
radio = REY4DIYOW
radius = REY4DIYAHS
rain = REY4N
random = RAE4NDUM
range = REY4NJ
rare = REH4R
rate = REY4T
rather = RAE4DHER
ratio = REY4SHIYOW
reach = RIY4CH
reaction = RIYAE4KSHUN
read = RIY4D
realistic = RIY5LIH4STIXK
reason = RIY4ZUN
receive = RIXSIY4V
reciter = RIXSAY4TER
recognize = REH4KAXGNAYZ
recommend = REH5KUMEH4ND
record = REH4KERD
recover = RIYKAH4VER
rectangle = REH4KTAENXGUL
reduce = RIXDUW4S

refer = RIYFER4
reference = REH4FERIXNS
reflection = RIXFLEH4KSHUN
refrigerator = RIXFRIH4JEREYTER
region = RIY4JUN
register = REH4JIXSTER
regular = REH4GYUXLER
reject = RIXJEH4KT
relativity = REH5LAXTIH4VIXTIY
relax = RIXLAE4KS
relay= RIY4LEY
release = RIXLIY4S
relief = RIYLIY4F
religion = RIXLUH4JUN
remain = RIYMEY4N
remember = RIXMEH4MBER
remove = RIYMUX4V
rent = REH4NT
repeat = RIXPIY4T
replace = RIXPLEY4S
reply = RIXPLAY4
report = RIXPOH4RT
represent = REHPRIXZEH4NT
reproduction = RIY5PRAXDAH4KSHUN
republic = RIXPAH4BLIXK
rescue = REH4SKYUW
research = RIY4SERCH
reserve = RIXZER4V
resistance = RIXZIH4STUNS
respect = RIXSPEH4KT
response = RIXSPAA4NS
rest = REH4ST
restore = RIXSTOH4R
retail = RIY4TEY6L
return = RIXTER4N
reverse = RIXVER4S
review = RIXVYUW4
revolution = REH5VULUXWSHUN
rhapsody = RAE4PSAXDIY
rhythm = RIH4DHUM
rich = RIH4CH
ride = RAY4D
ridiculous = RIXDIH4KYULAXS
right = RAY4T
rigid = RIH4JIXD
ring = RIH4NX
rise = RAY4Z
river = RIH4VER
road = ROW4D
rocket = RAA4KIXT
roll = ROH4L
room = RUW4M
rough = RAH4F

round = RAW4ND
rubber= RAH4BER
rule = RUW4L
run = RAH4N
rush = RAH4SH

- S -

sabotage = SAE5BAXTAA6ZH
sacrifice = SAE4KRIXFAYS
sad = SAE4D
safe = SEY4F
safety = SEY4FTIY
saint = SEY4NT
sale = SEY4L
S.A.M. = SAE4M
same = SEY4M
sample = SAE4MPUL
sanctuary = SAE4NXKCHUWEH6RIY
sandwich = SAE4NWIXCH
sarcasm = SAA4IRKAEZUM
satisfaction = SAE4TIXSFAE4KSHUN
savage = SAE4VIXJ
save = SEY4V
say = SEY4
scale = SKEY4L
scandal = SKAE4NDUL
scarce = SKEY4RS
scatter = SKAE4TER
scenic = SIY4NIXK
schedule = SKEH4JYUWL
scheme = SKIY4M
scholar = SKAA4LER
school = SKUW4L
science = SAY4IHNS
scientific = SAY4UNTIH5FIXK
scientific = SAY4AXNTIH5FIXK
scissors = SIH4ZERZ
score = SKOH4R
scramble = SKRAE4MBUL
scratch = SKRAE4CH
scream = SKRIY4M
screw = SKRUW4
script = SKRIH4PT
scroll = SKROW4L
seal = SIY4L
search = SER4CH
season = SIY4ZUN
second = SEH4KUND
secret = SIY4KRIXT
secretary = SEH4KRIXTEH5RIY
section = SEH4KSHUN

security = SIXKYUH4RIXTIY
see = SIY4
seek = SIY4K
segment = SEH4GMIXNT
self = SEH4LF
sell = SEH4L
semi- = SEH4MIY
send = SEH4ND
sensation = SEHNSEY4SHUN
senior = SIY4NYER
sense = SEH4NS
sensible = SEH4NSIXBUL
sensitive = SEH4NSIXTIX6V
sentence = SEH4NTIXNS
separate = SEH4PERIXT
sequence = SIY4KWEHNS
serial = SIH4RIYUL
serious = SIH4RIYAHS
serve = SER4V
service = SER4VIXS
session = SEH4SHUN
set = SEH4T
settle = SEH4TUL
several = SEH4VERUL
sex = SEH4KS
shadow = SHAE4DOW
shake = SHEY4K
shame = SHEY4M
shape = SHEY4P
share = SHEY4R
sharp = SHAA4RP
she = SHIY4
sheet = SHIY4T
shield = SHIY4LD
shift = SHIH4FT
shock = SHAA4K
shoot = SHUW4T
shop = SHAA4P
short = SHOH4RT
should = SHUH4D
show = SHOW4
shy = SHAY4
sick = SIH4K
side = SAY4D
sight = SAY4T
sign = SAY4N
signal = SIH4GNUL
silent = SAY4LIXNT
silver = SIH4LVER
similar = SIH4MULER
simple = SIH4MPUL
simplicity = SIHMPLIH4SIXTIY
simulator = SIH4MYULEYTER

sin = SIH4N
single = SIH4NXGUL
sinister = SIH4NIXSTER
sir = SER4
siren = SAY4RIXN
sit = SIH4T
situation = SIH5CHUWEY4SHUN
skeptical = SKEH4PTIXKUL
sketch = SKEH4TCH
skill = SKIH4L
skip = SKIH4P
slang = SLAE4NX
sleep = SLIY4P
sleeve = SLIY4V
slip = SLIH4P
slot = SLAA4T
slow = SLOW4
small = SMAO4L
smart = SMAA4RT
smell = SMEH4L
smooth = SMUW4DH
snap = SNAE4P
so = SOW4
social = SOW4SHUL
society = SAXSAY4IXTIY
soft = SAO4FT
solar = SOW4LER
soldier = SOH4LJER
solemn = SAA4LUM
solid = SAA4LIXD
solitude = SAA4LIXTUW6D
solution = SULUW4SHUN
some = SAH4M
somebody = SAH4MBAADIY
song = SAO4NX
soon = SUW4N
sophisticated = SAXFIH4STIXKEYTIXD
sorry = SAA4RIY
sort = SOH4RT
sound = SAW4ND
south = SAW4TH
space = SPEY4S
spare = SPEY4R
spatial = SPEY4SHUL
speak = SPIY4K
special = SPEH4SHUL
specific = SPAXSIH4FIXK
speculate = SPEH4KYULEYT
speech = SPIY4CH
speed = SPIY4D
spell = SPEH4L
spend = SPEH4ND
sphere = SFIY4R

spin = SPIH4N
spiral = SPAY4RUL
spirit = SPIH4RIXT
splendid = SPLEH4NDIXD
split = SPLIH4T
spoil = SPOY4L
spontaneous = SPAANTEY4NIYAHS
sports = SPOH4RTS
spot = SPAA4T
spread = SPREH4D
spring = SPRIH4NX
spy = SPAY4
square = SKWEH4R
squeeze = SKWIY4Z
stability = STAXBIH4LIXTIY
staff = STAE4F
stand = STAE4ND
standard = STAE4NDERD
star = STAA4R
start = STAA4RT
state = STEY4T
static = STAE4TIXK
station = STEY4SHUN
stay = STEY4
steady = STEH4DIY
steer = STIY4R
step = STEH4P
stereo = STEH4RIYOW
stick = STIH4K
stimualte = STIH4MYULEYT
stock = STAA4K
stone = STOW4N
stop = STAA4P
store = STOH4R
story = STOH4RIY
straight = STREY4T
strange = STREY4NJ
strategy = STRAE4TIXJIY
street = STRIY4T
strength = STREY4NTH
strike = STRAY4K
strong = STRAO4NX
structure = STRAH4KCHER
stubborn = STAH4BERN
student = STUW4DIXNT
study = STAH4DIY
stuff = STAH4F
stupid = STUX4PIXD
style = STAY4L
subject = SAH4BJEHKT
substance = SAH4BSTIXNS
subtle = SAH4TUL
succession = SAHKSEH4SHUN

succeed = SAHKSIY4D
such = SAH4CH
sudden = SAH4DIXN
suggest = SAHGJEH4ST
sum = SAH4M
summer = SAH4MER
sun = SAH4N
super = SUX4PER
superb = SUXPER4B
superior = SUXPIH4RIYER
supply = SAXPLAY4
support = SAXPOH4RT
sure = SHUX4R
surprise = SERPRAY4Z
surroundings = SERAW4NDIHNXGZ
suspend = SAHSPEH4ND
swear = SWEH4R
sweep = SWIY4P
swell = SWEH4L
swing = SWIH4NX
syllable = SIH4LAXBUL
symbol = SIH4MBUL
symbolic = SIHMBAA4LIXK
symmetric = SIHMEH4TRIXK
sympathy = SIH4MPAXTHIY
synchronize = SIH4NXKRAX5NAYZ
synonym = SIH4NUNIXM
system = SIH4STUM
synthesizer = SIH4NTHAXSAYZER

- T -

tab = TAE4B
table = TEY4BUL
tactical = TAE4KTIXKUL
tail = TEY4L
take = TEY4K
talent = TAE4LIX6NT
tall = TAO4L
talk = TAO4K
tap = TAE4P
tape = TEY4P
target = TAA4RGIXT
task = TEY4ST
tax = TAE4KS
teach = TIY4CH
team = TIY4M
technical = TEH4KNIXKUL
technology = TEHKNAA4LAXJIY
telephone = TEH4LAX6FOWN
television = TEH4LAX6VIXZHUN
temper = TEH4MPER

tender = TEH4NDER
tense = TEH4NS
tension = TEH4NSHUN
term = TER4M
terminal = TER4MIXNUL
terrestrial = TER6EH4STRIY6UL
terrible = TEH4RAXBUL
territory = TEH4RAXTOH6RIY
terror = TEH4RER6
test = TEH4ST
testimony = TEH4STUMOHNIY
text = TEH4KST
than = DHAE4N
than = DHAE4N
thank = THAE4NXK
that = DHAE4T
the = DHAH4
theater = THIY4AHTER
then = DHEH4N
theorem = THIY4RUM
theory = THIY4RIY
thermometer = THERMAA4MIXTER
thesis = THIY4SIXS
they = DHEY4
thin = THIH4N
thing = THIH4NX
think = THIH4NXK
this = DHIH4S
thought = THAO4T
threshold = THREH4SH/HOWLD
through = THRUW4
ticket = TIH4KIXT
tight = TAY4T
time = TAY4M
tiny = TAY4NIY
tired = TAY4ERD
title = TAY4TUL
together = TUXGEH4DHER
tolerance = TAA4LERIXNS
tone = TOW4N
tool = TUW4L
top = TAA4P
toss = TAO4S
touch = TAH4CH
tough = TAH4F
tournament = TER4NUMIXNT
toward = TOH4RD
toward = TOW4RD
town = TAW4N
toy = TOY4
trace = TREY4S
track = TRAE4K
trade = TREY4D

tradition = TRAXDIH4SHUN
traffic = TRAE4FIXK
trail = TREY4L
trajectory = TRAXJEH4KTERY
transaction = TRAENZAE4KSHUN
transfer = TRAE4NSFER
transform = TRAENSFOH4RM
transistor = TRAENZIH4STER
translate = TRAE4NZLEYT
transmit = TRAE4NZMIXT
transparent = TRAE5NSPEH4RIXNT
transportation = TRAE5NZPOHRTEY4SHUN
trap = TRAE4P
treasury = TREH4ZHERIY
tree = TRIY4
trek = TREH4K
tremendous = TRIXMEH4NDAXS
trespass = TREH4SPAES
trial = TRAY4UL
trangle = TRAY4AENXGUL
trick = TRIH4K
trgger = TRIH4GER
trim = TRIH4M
trip = TRIH4P
triple = TRIH4PUL
triumph = TRAY4AHMF
troll = TROW4L
trophy = TROW4FIY
trouble = TRAH4BUL
truck = TRAH4K
true = TRUW4
truth = TRUW4TH
trj = TRAY4
tune = TUW4N
tunnel = TAH4NUL
turn = TER4N
tutor = TUW4TER
twist = TWIH4ST
type = TAY4P
typewriter = TAY4PRAYTER

- U -

ugly = AH4GLIY
ultimate = AH4LTAX6MIXT
uncle = AH4NKUL
under = AH4NDER
understand = AH5NDERSTAE4ND
uniform = YUW4NIXFOHRM
union = YUW4NYUN
unit = YUW4NIXT
universal = YUW5NIXVER4SUL

unless = AHNLEH4S
up = AH4P
upset = AHPSEH4T
urge = EH4RJ
use = YUW4S
utility = YUWTIH4LIXTIY

- V -

vacation = VEYKEY4SHUN
vacuum = VAE4KYUWM
vague = VEY4G
valid = VAE4LIXD
value = VAE4LYUW
valve = VAE4LV
vanadium = VUNEY4DIYUM
vapor = VEY4PER
variation = VEH5RIYEY4SHUN
various = VEH4RIYAHS
vary = VEH4RIY
veal = VIY4L
vector = VEH4KTER
vegetable = VEH4JTAXBUL
vehicle = VIY4IX6KUL
ventilate = VEH4NTULEYT
verb = VER4B
versatile = VER4SAXTUL
verse = VER4S
version = VER4ZHUN
vertical = VER4TIXKUL
very = VEH4RIY
veto = VIY4TOW
vibration = VAYBREY4SHUN
vicinity = VAXSIH4NIXTIY
victory = VIH4KTERIY
video = VIH4DIYOW
village = VIH4LIXJ
vinyl = VAY4NUL
violation = VAY4AXLEY5SHUN
virtue = VER4CHUW
visible = VIH4ZIXBUL
visit = VIH4ZIXT
vital = VAY4TUL
vocabulary = VOHKAE4BYULEHRIY
vocal = VOW4KUL
voice = VOY4S
volt = VOW4LT
volume = VAA4LYUWM
voluntary = VAA4LUNTEH5RIY
vote = VOW4T
vowel = VAW4UL
voyage = VOY4IXJ

video = VIH4DIYOW

- W -

wafer = WEY4FER
wage = WEY4J
wait = WEY4T
wake = WEY4K
walk = WAO4K
wall = WAO4L
war = WOH4R
warm = WOH4RM
warp = WOH4RP
warranty = WOH5RIXNTIY4
wash = WAA4SH
waste = WEY4ST
watch = WAA4CH
water = WAO4TER
watt = WAA4T
wave = WEY4V
way = WEY4
weak = WIY4K
wealth = WEH4LTH
wear = WEH4R
wedding = WEH4DIHNX
week = WIY4K
weight = WEY4
welcome = WEH4LKUM
well = WEH4L
were = WER4
what = WHAH4T
wheel = WHIY4L
when = WHEH4N
which = WHIH4CH
while = WHAY4L
whisper = WHIH4SPER
white = WHAY4T
who = /HUW4
whole = /HOW4L
wide = WAY4D
wild = WAY4LD
will = WIH4L
win = WIH4N
window = WIH4NDOW
wing = WIH4NX
winter = WIH4NTER
wise = WAY4Z
wish = WIH4SH
with = WIH4TH
wizard = WIH4ZERD
woman = WUH4MUN
women = WIH4MIXN

wonder = WAH4NDER
word = WER4D
Wordrace = WER2DREYS
work = WER4K
world = WUH4RLD
worry = WER4IY
would = WUH4D
wrap = RAE4P
write = RAY4T
wrong = RAO4NX

- X -

Zerox = ZIH4RAAKS
X-ray = EH4KSREY
xylophone = ZAY4LAXFOWN

- Y -

yacht = YAA4T
yard = YAA4RD
yawn = YAO4N
year = YIH4R
yellow = YEH4LOW
yes = YEH4S
you = YUW4
your = YOH4R
youth = YUX4TH

- Z -

zany = ZEY4NIY
zero = ZIY4ROW
zig-zag = ZIH3GZAEG
zip = ZIH4P
zodiac = ZOW4DIY6AEK
zone = ZOW4N

- DAYS OF THE WEEK -

Monday = MAH4NDEY
Tuesday = TUW4ZDEY
Wednesday = WEH4NZDEY
Thursday = THER4ZDEY
Friday = FRAY4DEY
Saturday = SAE4TERDEY
Sunday = SAH4NDEY

- MONTHS OF THE YEAR -

January = JAE4NYUXEHRIY
February = FEH4BRUXEH6RIY
March = MAA4RCH
April = EY4PRIXL
May= MEY4
June = JUW4N
July = JUHLAY4
August = AO4GAXST
September = SEHPTEH4MBER
October = AAKTOW4BER
November = NOHVEH4MBER
December = DIHSEH4MBER

- NUMBERS -

one = WAH4N
two = TUW4
three = THRIY4
four = FOH4R
five = FAY4V
six = SIH4KS
seven = SEH4VIXN
eight = EY4T
nine = NAY4N
ten = TEH4N
eleven = IXLEH4VIXN
twelve = TWEH4LV
thirteen = THER4TIY6N
twenty = TWEH4NTIY
thirty = THER4TIY
hundred = /HAH4NDRIXD
thousand = THAW4ZUND
million = MIH4LYUN

- STATES AND PROVINCES -

United States = YUWNAY4TIXD STEY4TS
Alabama = AE4LAXBAE6MAX
Alaska = AHLAE4SKAH
Arizona = EH4RAXZOW5NAH
Arkansas = AA4RKUNSAO
California = KAE5LAXFOH4RNYAH
Colorado = KAA5LAXRAA4DOW
Connecticut = KAHNEH4TIXKAHT
Delaware = DEH4LAXWEH6R
Florida = FLOH4RIXDAH
Georgia = JOH4RJAH
Hawaii = /HAHWAY4IY
Idaho = AY4DAH/HOW

Illinois = IHLUNOY4
Indiana = IH5NDIYAE4NAH
Iowa = AY4AHWAH
Kansas = KAE4NZIXS
Kentucky = KEHNTAH4KIY
Louisiana = LUXIY4ZIYAE5NAH
Maine = MEY4N
Maryland = MEH4RULIXND
Massachusetts = MAE5SAXCHUW4SIXTS
Michigan = MIH4SAXGUN
Minnesota = MIH5NAXSOW4TAH
Mississippi = MIH5SIXSIH4PIY
Missouri = MIHZUH4RIY
Montana = MAANTAE4NAH
Nebraska = NAXBRAE4SKAH
Nevada = NAXVAE4DAH
New Hampshire= NUW6/HAE4MPSHER
New Jersey = NUWJER4ZIY
New Mexico = NUWMEH4KSIXKOW
New York = NUWYOH4RK
North Carolina = NOH4RTH KEH5RULAY4NAH
North Dakota= NOH4RTH DAHKOW4TAH
Ohio = OW/HAY4OW
Oklahoma = OWKLAX6/HOW4MAH
Oregon = OH4RIXGUN
Pennsylvania = PEH5NSULVEY4NYAH
Rhode Island = ROW5D AY4LUND
South Carolina = SAW4TH KEH5RULAY4NAH
South Dakota = SAW4TH DAXKOW4TAH
Tennessee = TEH5NAXSIY4
Texas = TEH4KSAXS
Utah = YUW4TAO6
Vermont = VERMAA4NT
Virginia = VERJIH4NYAH
Washington = WAA4SHIHNXTAHN
West Virginia = WEH5ST VERJIH4NYAH
Wisconsin = WIHSKAA4NSUN
Wyoming = WAYOW4MIHNX

Provinces of Canada =
PRAA4VIXNSIXZ AHV KAE4NAXDAH

Alberta = AELBER4TAH
British Columbia =
BRIH4TIXSH KAHLAH4MBIYAH
Manitoba = MAE5NIXTOW4BAH
New Brunswich = NUWBRAH4NZWIXK
Newfoundland = NUW4FIXNLIXND
Nova Scotia = NOH4VAX5KOW4SHAH
Ontario = AANTEH4RIYOW
Prince Edward Island =
PRIH5NS EH4DWERD AY4LUND
Quebec = KUHBEH4K

Saskatchewan = SAESKAE4CHAXWAAN

- UNITS -

units = YUW4NIXTS
inches = IH4NCHIXZ
feet = FIY4T
yards = YAA4RDZ
miles = MAY4LZ
centimeters = SEH4NTIXMIY6TERZ
kilometers = KIXLAA4MIXTERZ
acres = EY4KERZ
ounces = AW4NSIXZ
pounds = PAW4NDZ
tons = TAH4NZ
grams = GRAE4MZ
teaspoons = TIY4SPUWNZ
cups = KAH4PS
pints = PAY4NTS
quarts = KWOH4RTS
gallons = GAE4LUNZ
liters = LIY4TERZ
degrees = DAXGRIY4Z

FINDING PHONEME SPELLING ERRORS

If you have made a phonetic spelling mistake that causes S.A.M. to be unable to break your string down into phonemes, he will beep twice at you and come back to BASIC without speaking. The location of the bad letter in the string is stored for you to examine. Also, you may PEEK at this location in a program to see if there were any errors in spelling and then make the required changes.

Here is a sample error-checking and display program:

```
.
.
.
100 SAM$="MAY VOY4C IHZ BIHZAA5R."
110 A=USR(8192)
120 IF PEEK(8211)<255 THEN GOSUB 1000
:REM ERROR CHECK
.
.
.
1000 REM ERROR DISPLAY -- ERROR APPEARS IN INVERSE
1010 N=PEEK(8211):REM N IS POSITION OF ERROR
1020 SAM$(N,N) = CHR$(ASC(SAM$(N,N))+128)
1030 PRINT SAM$
1040 RETURN
```

The inverse character marks the spot where S.A.M. could no longer continue reading the string.

TECHNICAL NOTES
USE IN BASIC

S.A.M. from BASIC performs all stack housekeeping that is required.

When S.A.M. completes vocal output,the NMIEN (Non-maskable Interrupt Enable) ($D40E) returns to the following conditions:

BIT 6 -- Vertical Blank Interrupt Enable = "on"
BIT 7 -- Display List Instruction Interrupt Enable = "on"

All other registers are returned to OS shadow values within 1/60 second after vocal output.

Note that during speech, the VBI is shut down so that the real-time clock registers (18, 19, 20) do not advance.

SCREEN BLANK

The screen blanks during vocal output because Direct Memory Access (DMA) causes gaps to be inserted into the speech waveform each time the 6502 processor waits for the ANTIC chip to access memory. The audible result is extremely distorted speech when the screen is on.

If this speech quality is desirable for some application (or the screen must remain on during speech), S.A.M. may be operated in the DMA-enabled mode by POKE-ing a "1" into the "lights" register: **8210**. There are different speed and pitch addresses to be used in this case. To return to DMA-disabled speech, POKE a "0" into this register.

IMPORTANT ADDRESSES

	Decimal	Hex
S.A.M. from Atari BASIC	**8192**	**$2000**
S.A.M. from machine language	**8196**	**$2004**
RECITER from Atari BASIC	**8199**	**$2007**

RECITER from machine language	**8203**	**$200B**
SPEED (LIGHTS OFF)	**8208**	**$2010**
SPEED (LIGHTS ON)	**8206**	**$200E**
PITCH (LIGHTS OFF)	**8209**	**$2011**
PITCH (LIGHTS ON)	**8207**	**$200F**
DMA-enable	**8210**	**$2012**
ERROR	**8211**	**$2013**
ATASCII STRING	**8212**	**$2014**

LISTING OF GUESSNUM

Download GUESSNUM.BAS (Atari BASIC)

```
10 REM ---- GUESSNUM ----
20 DIM SAM$(255),B$(50),C$(50)
30 SAM=8192:REM SAM'S ADDRE5S
40 GRAPHICS 2:? #6;"GUESS THE NUMBER":? #6;"BETWEEN
1 AND 100"
50 SETCOLOR 2,0,0
60 N=INT(99*RND(0))+1
70 SAM$="GEH3S DHAX NAH4MBER BIXTWIY5N WAH4N Q AEND
WAHH6 /HAN4NDRIHD.":A=USR(SAM)
80 TRAP 80:INPUT G
90 IF G>99 THEN SAM$="DHAETS MON4R DHAEN WAHN
/HAH4NDRIXD.":A=USR(SAM):GOTO 80
100 IF G<1 THEN SAM$="DHAE5TS LEH3S DHAEN WAH5N."
:A=USR(SAM):GOTO 80
110 SAM$=""
120 IF G<10 THEN B$="":GOTO 340
130 DN G-9 GOTO 150,160,170,180,190,200,210,220,230,240
140 GOTO 250
150 B$="TEH4N":GOTO 460
160 B$="IHLEH4VIXN":GOTO 460
170 B$="TWEH4LV":GOTO 460
180 B$="THER4TIY6N":GOTO 460
190 B$="FOH4RTIY6N":GOTO 460
200 B$="FIH4FTIY6N":GOTO 460
210 B$="SIH4KSTIY6N":GOTO 460
220 B$="SEH4VUNTIY6N":GOTO 460
230 B$="EY4TIY6N":GOTO 460
240 B$="NAY4NTIY6N":GOTO 460
250 ON INT(G/10)-1 GOTO 260,270,280,290,300,310,320,330
260 B$="TWEH4NTIY6":GOTO 340
270 B$="THER4TIY6":GOTO 340
280 B$="FOH4RTIY6":GOTO 340
290 B$="FIH4FTIY6":GOTO 340
300 B$="SIH4KSTIY6":GOTO 340
310 B$="SEH4VUNTIY6":GOTO 340
320 B$="EY4T1Y6":GOTO 340
330 B$="NAY4NTIY6"
```

```
340 R=G-10*INT(G/10)
350 IF R=0 THEN GOTO 460
360 ON R GOTO 370,380,390.400,410,420,430,440,450
370 B$(LEN(B$)+1)="WAH5N":GOTO 460
380 B$(LEN(B$)+1)="TUW5":GOTO 460
390 B$(LEN(B$)+1)="THRIY5":GOTO 460
400 B$(LEN(B$)+1)="FOHR5":GOTO 460
410 B$(LEN(B$)+1)="FAY5V":GOTO 460
420 B$(LEN(B$)+1)="SIH5KS":GOTO 460
430 B$(LEN(B$)+1)="SEH5VUN":GOTO 460
440 B$(LEN(B$)+1)="EY5T":GOTO 460
450 B$(LEN(B$)+1)="NAY5N"
460 IF G>N+25 THEN C$=" IHZ MAH3CH TUW5 /HAY6."
:GOTO 530
470 IF G>N+5 THEN C$=" IHZ TUW3 /HAY6."
:GOTO 530
480 IF G>N THEN C$=" IHZ AN LIH3TUL TUW4 /HAY6, "
:GOTO 530
490 IF C<N-25 THEN C$=" IHZ MAH3CH TUW4 LAXOW, "
:GOTO 530
300 IF G<N-5 THEN C$=" IHZ TUW3 LAXOW. "
:GOTO 530
310 IF C<=""">N THEN GOTO 80
350 ? :? :? :? :? :GOTO 60
```

SELDOM-USED PHONEME COMBINATIONS

Phoneme Combination	You probably want:		Unless it splits syllables like:
GS	GZ e.g. ba**gs**	bu**gs**pray	
BS	BZ e.g. slo**bs**	o**bs**cene	
DS	DZ e.g. su**ds**	Hu**ds**on	
PZ	PS e.g. sla**ps**	--	
TZ	TS e.g. cur**ts**y	--	
KZ	KS e.g. fi**x**	--	
NC	NXG e.g. s**inging**	**ing**rate	
NK	NXK e.g. ba**nk**	Su**nk**ist	

FUTURE IMPROVEMENTS

Improvements upon and modifications to the S.A.M. system may occur in the future. Such new versions of S.A.M. will be made available at nominal cost to **registered S.A.M. owners**.

We are also planning to release a new program called "SUPERECITER". RECITER presently has a pronunciation accuracy of about 90%.

SUPERECITER will show a major improvement in this area. But, we need your help.

It you hear a word mispronounced by RECITER that you feel is important, jot it down. Send us your list of these words (or proper names) so that we may incorporate them into the expanded rule set of SUPERECITER. Your contributions will be greatly appreciated.

S.A.M. is an ongoing project at DON'T ASK Computer Software. We welcome your comments and suggestions on our software speech synthesis products.

NOTES