Introduction to Programming with the Colour Maximite 2

Geoff Graham

Version 5 (June 2021)

Table of Contents

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.

There are two generations of the Colour Maximite 2 (Generation 1 and Generation 2) however the differences between the two are small and this tutorial will cover both versions.

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 edit a program (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 is why the commands 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"`

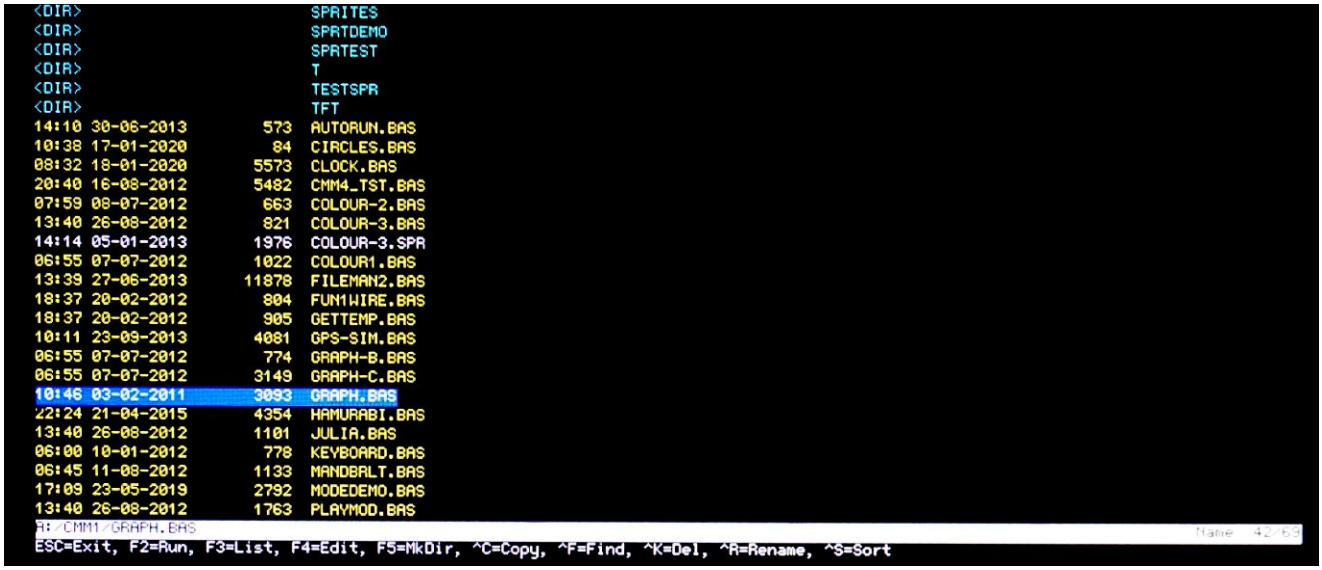
Note that the file name must be surrounded by double quotes as shown above. This is because the file name is a string and in MMBasic all string constants (ie, not a variable) must be quoted. In all cases the file extension ".BAS" will be automatically added if an extension was not specified in the command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

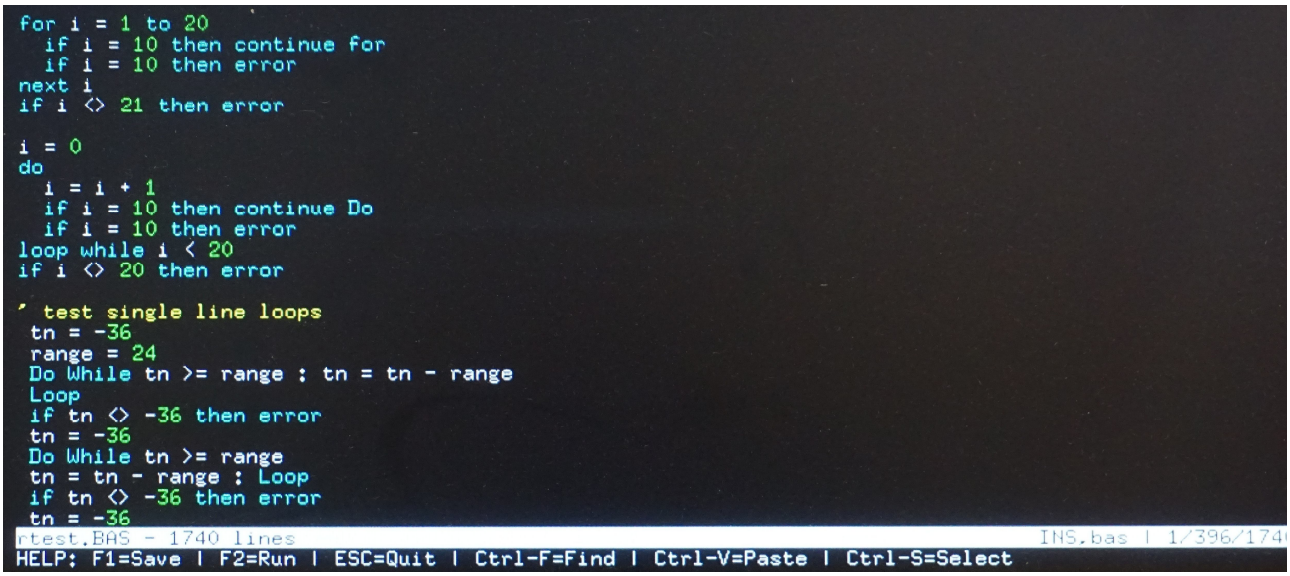
At the bottom of the screen the status line lists the current cursor position and the common keystrokes. All these operate on the file currently selected by the cursor:

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.

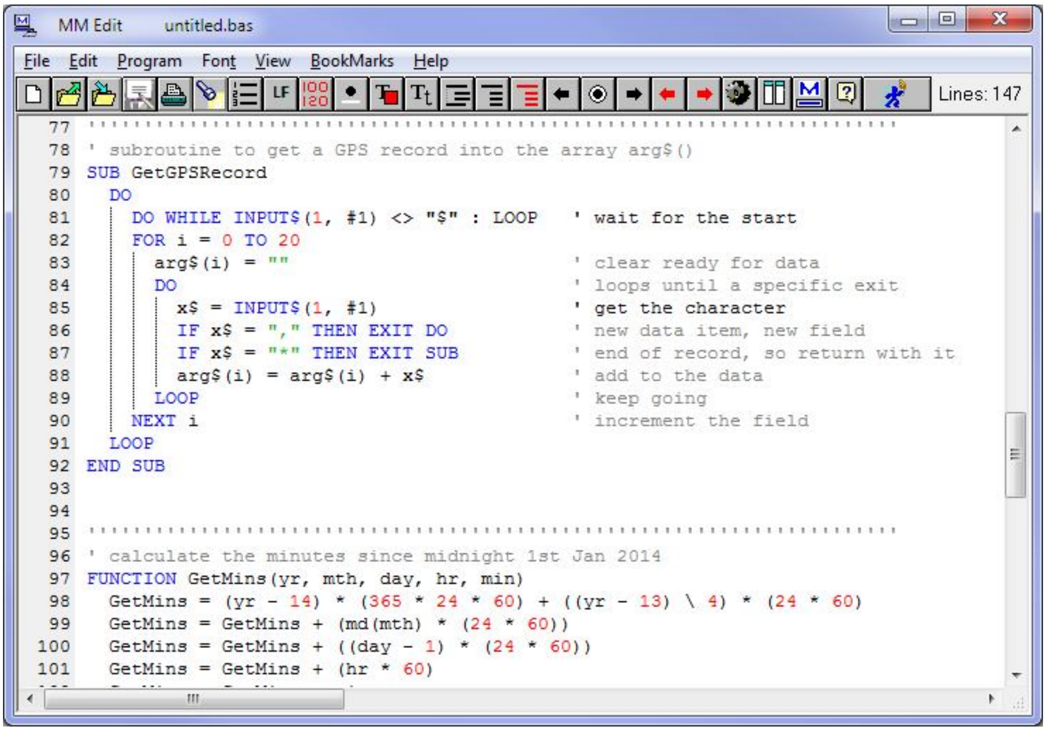
Alternatively you can transfer it via the serial console using either the AUTOSAVE or XMODEM commands. This requires you to have a terminal emulator running on your desktop machine and connected to the serial console of the Colour Maximite 2. How to connect to the serial console via USB is described 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.



MMEdit can support a number of MMBasic based devices so you will need to select the Colour Maximite 2 syntax in the settings before it will work correctly.

The most important feature is the right hand button on the tool bar (the icon of a running man). When you click on this button the program will be immediately transferred to your Colour Maximite 2 using the XModem protocol. Following the transfer a window will be automatically opened and connected to the console where you can run and test your program. If it has an error or needs tweaking it is very easy to go back to the editor, make the change and transfer it again.

MMEDIT can be downloaded from Jim's website at: https://www.c-com.com.au/MMedit.htm It is free although he would appreciate a small donation.

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 each number. This space is a place holder for the minus symbol (-) in case the value is negative. Notice the difference with the number 12 in this example:

```
> PRINT -12, 34, -9.4, 1000
-12       34      -9.4      1000
>
```

The print statement can be terminated with a semicolon (;). This will prevent the PRINT command from moving to a new line when it 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 this is usually done with the IF statement. This is written almost like an English sentence:

IF *condition* THEN *action*

The condition is usually a comparison such as equals, less than, more than, etc. For example:

```
IF Temp < 25 THEN PRINT "Cold"
```

`Temp` would be a variable holding the current temperature (in ºC) 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.

In a multiline IF statement you can make additional tests using the ELSE IF command. This is best explained by using an example:

```
IF Temp < 0 THEN
    PRINT "Freezing"
ELSE IF Temp < 25 THEN
    PRINT "Cold"
ELSE IF Temp < 40 THEN
    PRINT "Warm"
ELSE
    PRINT "Hot"
ENDIF
```

The ELSE IF can use the same tests as an ordinary IF (ie, <, <=, etc) but that test will only be made if the preceding test was false. So, for example, you will only get the message *Warm* if `Temp < 0` failed, and `Temp < 25` failed but `Temp < 40` was true. The final ELSE will catch the case where all the tests were false.

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:

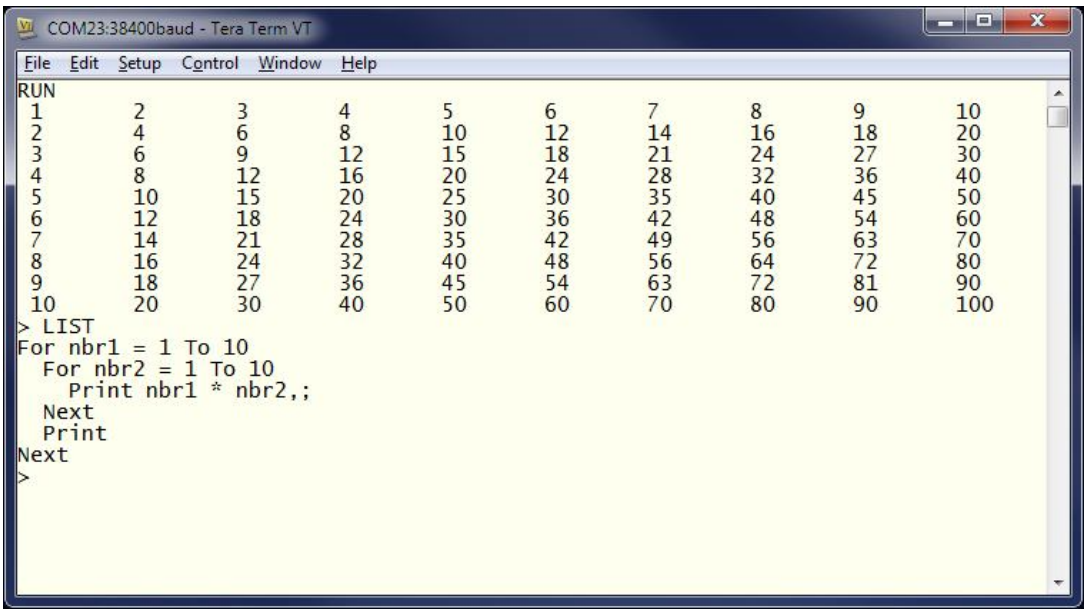
```
FOR nbr = 10 TO 1 STEP -1
  PRINT nbr,;
NEXT nbr
```

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:

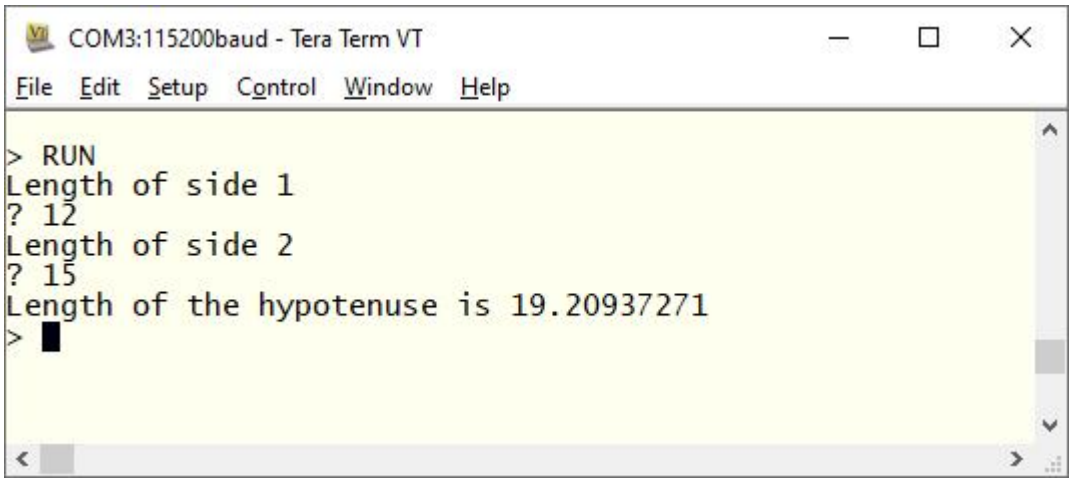
```
INPUT var
```

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.

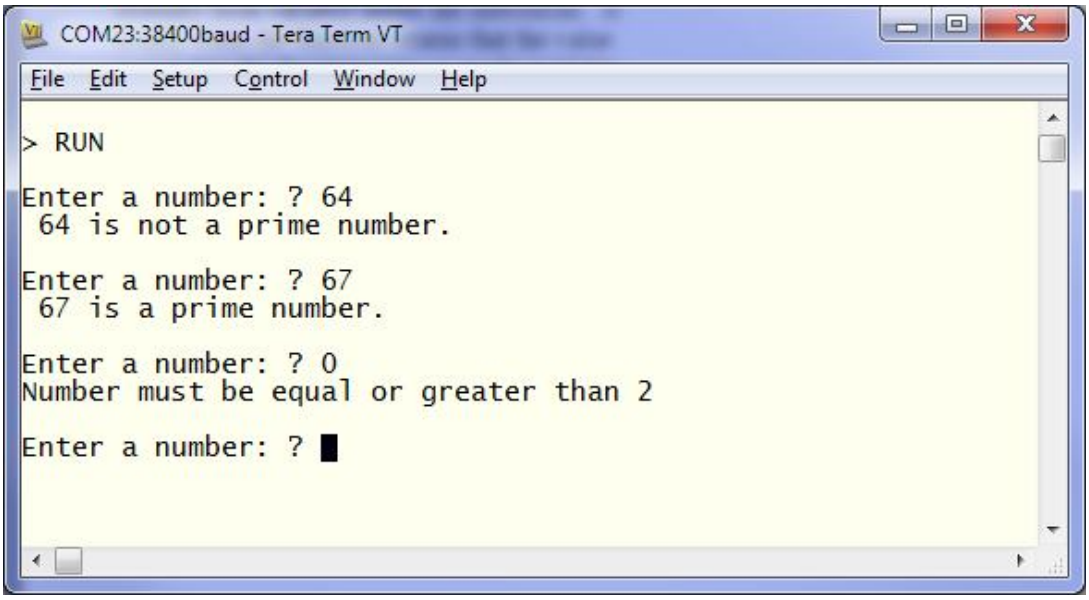
Following this the number is tested. If it is less than 2 an error message will be printed and the program will jump backwards and ask for the number again.

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.

For example:

```
pos = INSTR(S$, " ")
if pos > 0 THEN R$ = LEFT$(S$, INSTR(pos + 1, S$, " ") - 1)
```

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, if you wanted to check the voltage of a battery connected to pin 24 of the external I/O connector you could define the relevant values thus:

```
CONST BatteryVoltagePin = 24
CONST BatteryMinimum = 11.5
```

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

```
IF PIN(BatteryVoltagePin) < BatteryMinimum THEN SoundAlarm
```

It is good programming practice to use constants for any fixed number that represents an important value. Normally they are defined at the start of a program where they are easy to see and conveniently located for another programmer to adjust (if necessary).

Subroutines

A subroutine is a block of programming code which is self contained (like a module) and can be called from anywhere within your program. To your program it looks like a built in MMBasic command and can be used the same. For example, assume that you need a command that would signal an error by printing a message on the console. You could define the subroutine like this:

```
SUB ErrMsg
  PRINT "Error detected"
END SUB
```

With this subroutine embedded in your program all you have to do is use the command ErrMsg whenever you want to display the message. For example:

```
IF A < B THEN ErrMsg
```

The definition of a subroutine can be anywhere in the program but typically it is at the end. If MMBasic runs into the definition while running your program it will simply skip over it.

The above example is fine enough but it would be better if a more useful message could be displayed, one that could be customised every time the subroutine was called. This can be done by passing a string to the subroutine as an argument (sometimes called a parameter).

In this case the definition of the subroutine would look like this:

```
SUB ErrMsg  Msg$
  PRINT "Error: " + Msg$
END SUB
```

Then when you call the subroutine, you can supply the string to be printed on the command line of the subroutine. For example:

```
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.

When a subroutine is called with an argument that is a variable (ie, not a constant or expression) MMBasic will create a corresponding variable within the subroutine *that points back to this variable.* Any changes to the variable representing the argument inside the subroutine will also change the variable used in the call. This is called passing arguments by reference.

This is best explained by example:

```
DIM MyNumber = 5       ‘ set the variable to 5
CalcSquare MyNumber    ‘ the subroutine will square its value
PRINT MyNumber         ‘ this will print the number 25
END

SUB CalcSquare nbr
  nbr = nbr * nbr      ‘ square the argument and pass it back
END SUB
```

The subroutine CalcSquare will take its argument, square it and write it back to the variable representing the argument (`nbr`). Because the subroutine was called with a variable (`MyNumber`) the variable `nbr` will point back to `MyNumber` and any change to `nbr` will also change `MyNumber` accordingly. As a result the PRINT statement will output 25.

Passing arguments by reference is handy because it allows a subroutine to pass values back to the code that called it. However it could lead to trouble if a subroutine used the variable representing

an argument as a general purpose variable and changed its value. Then, if it were called with a variable as an argument, that variable would be inadvertently changed. For this reason you should avoid manipulating variables representing arguments inside a subroutine, instead assign the value to a local variable (see below) and manipulate that instead.

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. This is our new subroutine:

```
SUB ErrMsg  Msg$
  STATIC STRING lastmsg
  LOCAL STRING tstr
  IF Msg$ <> lastmsg THEN
    tstr = "Error: " + Msg$
    PRINT tstr
    lastmsg = Msg$
  ENDIF
END SUB
```

To keep track of the last message displayed we use a static variable called `lastmsg`. This will hold the text of the last message and we can compare it to the current message text to determine if it is different and therefore should be printed. This would give just one message every time a call is made with a duplicate message text.

The STATIC command uses exactly the same syntax as DIM. This means that you can define different types of static variables including arrays and you can also initialise them to some value.

The static variable is created on the first time the STATIC command is encountered and it is automatically set to zero (if a float or integer) or an empty string. On subsequent calls to the subroutine or function MMBasic will recognise that the variable has already been created and it will leave its value untouched (ie, whatever it was in the previous call). As with DIM you can also initialise a static variable to some value. For example:

```
STATIC INTEGER var = 123
```

On the first call (when the variable is created) it will be initialised to 123 but on subsequent calls it will keep whatever its value was previously set to.

Mostly static variables are used to keep track of the *state* of something while inside a subroutine or function. A *state* is a record of something that has happened previously. Examples include:

- Has the COM port already been opened?
- What steps in a sequence have we completed?
- What text has already been displayed?

Normally you will use global variables (created using DIM) to track a *state* but sometimes you want this to be contained within a module and this is where static variables are valuable. Just like LOCAL the use of STATIC helps to make your subroutines and functions more self contained and portable.

Calculate Days

We have covered a lot of programming commands and techniques so far in this tutorial and, 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
  ‘ main program loop
  PRINT : PRINT "Enter the date as  dd mmm yyyy"
  PRINT " First date";
  INPUT s
  d1 = GetDays(s)
  IF d1 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Second date";
  INPUT s
  d2 = GetDays(s)
  IF d2 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Difference is" ABS(d2 - d1) " days"
LOOP
```

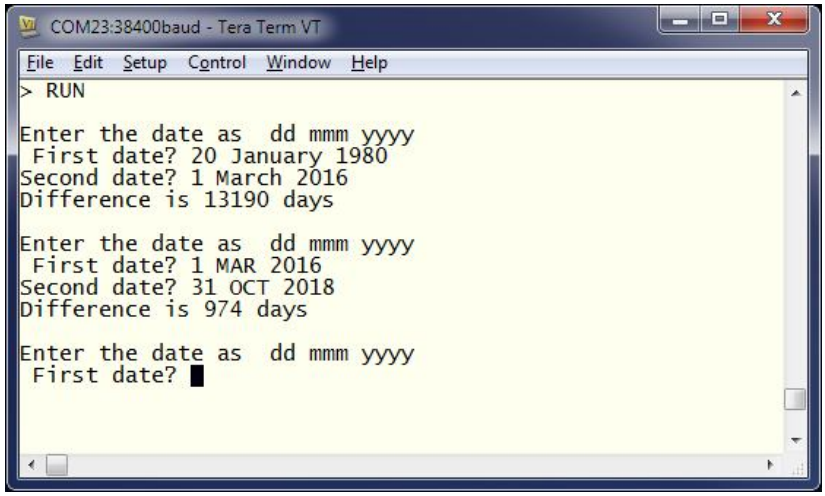
```
' Calculate the number of days since 1/1/1900
FUNCTION GetDays(d$) AS FLOAT
  LOCAL STRING Month(11) =
("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov","dec")
  LOCAL FLOAT Days(11) = (0,31,59,90,120,151,181,212,243,273,304,334)
  LOCAL FLOAT day, mth, yr, s1, s2

  ' Find the separating space character within a date
  s1 = INSTR(d$, " ")
  IF s1 = 0 THEN EXIT FUNCTION
  s2 = INSTR(s1 + 1, d$, " ")
  IF s2 = 0 THEN EXIT FUNCTION

  ' Get the day, month and year as numbers
  day = VAL(MID$(d$, 1, s2 - 1)) - 1
  IF day < 0 OR day > 30 THEN EXIT FUNCTION
  FOR mth = 0 TO 11
    IF LCASE$(MID$(d$, s1 + 1, 3)) = Month(mth) THEN EXIT FOR
  NEXT mth
  IF mth > 11 THEN EXIT FUNCTION
  yr = VAL(MID$(d$, s2 + 1)) - 1900
  IF yr < 1 OR yr >= 200 THEN EXIT FUNCTION

  ' Calculate the number of days including adjustment for leap years
  GetDays = (yr * 365) + FIX((yr - 1) / 4)
  IF yr MOD 4 = 0 AND mth >= 2 THEN GetDays = GetDays + 1
  GetDays = GetDays + Days(mth) + day
END FUNCTION
```

Note that the line starting `LOCAL STRING Month(11)` has been wrapped around because of the limited page width – it is one line as follows:

```
LOCAL STRING Month(11) = ("jan","feb","mar","apr","may","jun","jul","aug","sep","oct","nov","dec")
```

This program works by getting two dates from the user at the console and then converting them to 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 and 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 16 million different colours however, depending on the video mode selected, the full rage may not be capable of being displayed. This need not concern you as MMBasic will automatically convert the 24-bit colour value to the colour range supported by the current setting.

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, grey (or gray, USA spelling), 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
```

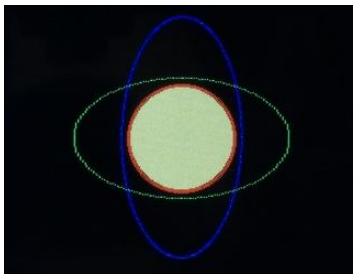
X and Y are the coordinates of the centre of the circle, R is the radius (in pixels), LW is optional and is the thickness of the line, A is the aspect ratio (which is optional), C is the colour and FILL (also optional) is the colour to fill the circle.

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	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 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 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"
```

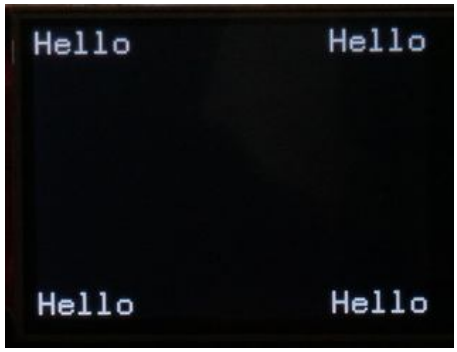
The alignment defaulted to left and top, the font defaulted to font #1, the scale to 1, the colour to white and the background to black. With the optional parameters you can either leave them off the end of the command string or just use two commas (ie, “, ,”) with nothing in-between to accept the default.

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 in the range of zero to `MM.HRes`. 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), RGB(black)         ' 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 scientific 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 two oddities in this program. The first is that when the array 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.

The second (slight) issue is that when the coordinates for a new pixel are generated they might match the coordinates of a pixel already illuminated and already entered in the array. This does not cause a problem and will only occur with about 3% of the pixels so it can be ignored. However a purist might want to scan through the array first before using the new coordinates and we will leave that as an exercise for the reader.

Video Modes

The video output on the VGA monitor can be set to a number of modes by using the MODE command. This takes the form:

MODE r, bits, *bg, int*

Where **r** is the video resolution (a number from 1 to 15 as detailed below) 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 straightforward use of the MODE command - for the more sophisticated aspects you should refer to the *Colour Maximite 2 User Manual*.

The following video resolutions will work with monitors that have an aspect ratio of 4:3 or widescreen monitors that can switch to that ratio (most will do this automatically):

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
6 = 256 x 240 pixels
7 = 320 x 240 pixels
8 = 640 x 480 pixels
13 = 400 x 300 pixels

The following are widescreen resolutions and will only work with a widescreen monitor:

9 = 1024 x 768 pixels
10 = 848 x 480 pixels
11 = 1280 x 720 pixels
12 = 960 x 540 pixels
14 = 960 x 540 pixels (lines not duplicated – not supported by all monitors)
15 = 1920 x 1080 pixels (Generation 2 only)

By default the Colour Maximite 2 will start up in the 800x600 pixel resolution which is best for everyday tasks. Mode 11 (1280 x 720) is also good for general programming if you have a widescreen monitor. The startup resolution can be changed with the OPTION DEFAULT MODE command.

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 to do this as there are less pixels to redraw.

Note that if you are planning on offering your programs to others you should avoid using the widescreen modes as not everyone will have a widescreen monitor.

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 four 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.
32 = 24-bit colour (16 million colours) with 8 bits for transparency (Generation 2 only).

The 8-bit and 16-bit colour depths will work with all video resolutions while the 12-bit colour will work with just the 4:3 resolutions (not widescreen). The 32-bit depth is only available on the Generation 2 hardware and is even more restricted (check the *User Guide*).

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, 16 and 32-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 many 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 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 2                ‘ all images will load here
DO
  A$ = DIR$("*.JPG")        ‘ scan SD card for images
  DO WHILE A$ <> ""         ‘ keep looping if image was found
    LOAD JPG A$             ‘ load the image
    PAGE COPY 2 TO 0, D     ‘ fast copy to the display page
    PAUSE 10000             ‘ display for 10 seconds
    A$ = DIR$()             ‘ get the next image file name
  LOOP
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 sequentially 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 . A PDF version of this is included in the Colour Maximite 2 firmware zip file.

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 has the following features:

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 of the pins on this connector (shown in grey) are for power and 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.
Ground	The common rail for all input/output.

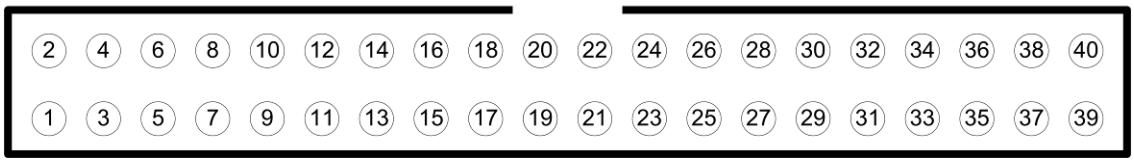
Twenty eight pins are under control of the BASIC program. Each of these can operate as a standard digital input or output. In addition most pins support special functions (listed in the table above) which can be enabled by the program. In summary these are:

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.

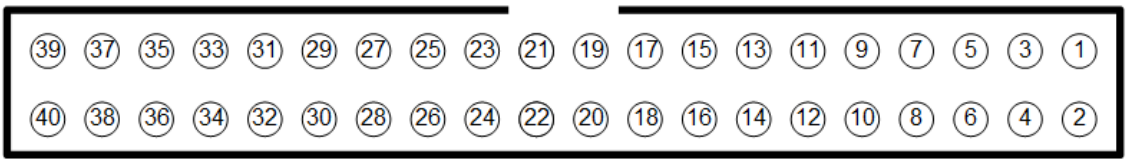
All pins with Analog I/O capability have a maximum input voltage limit of 3.3 volts while the other pins can accept voltages of up to 5.1 volts.

Pin Numbering

The pin number is used to reference the capabilities of the pin in the above table and is also used in MMBasic to configure the pin and control it. The Generation 1 and 2 connectors have the pins numbered differently and this image shows the connector (from the rear) for the first generation Colour Maximite 2:



And this is the pin numbering for the Generation 2 design (viewed from the rear):



Configuring a Pin

An input/output pin is configured using the SETPIN command. This command takes the form:

```
SETPIN pin_nbr, mode
```

where 'pin_nbr' is the pin number as listed above 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
```

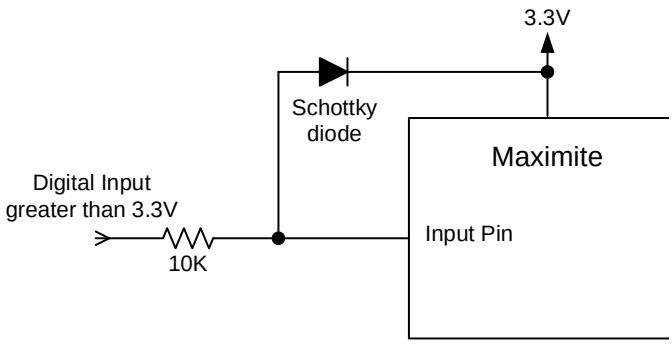
will set the output of pin 3 to zero (which generally means a logic low). In this case the PIN() construct is used as a command. This dual nature of the PIN() construct (either input or output) sometimes confuses newcomers to MMBasic so watch out for it.

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 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 input impedance is very high (leakage is less than 1µA) you can use a large valued input resistor – for example, a 10K resistor as shown 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:

```
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:

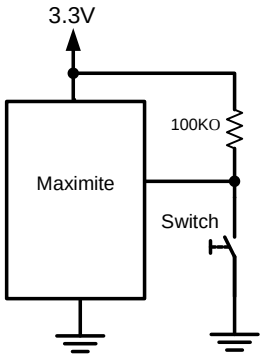
```
val = PORT(start, nbr)
```

'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)
```

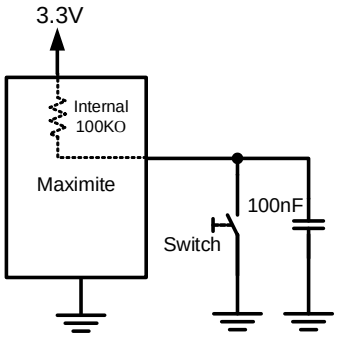
ie, read four consecutive input pins starting with pin 21. The *Colour Maximite 2 User Manual* goes into more detail and it is required reading if you need to use the PORT() function.

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 input can be specified with an pull-up 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 pull-up resistor you use SETPIN as follows:

```
SETPIN pin_nbr, DIN, PULLUP
```

Using either an internal or external pull-up 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. However 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 pull-up 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_nbr, 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

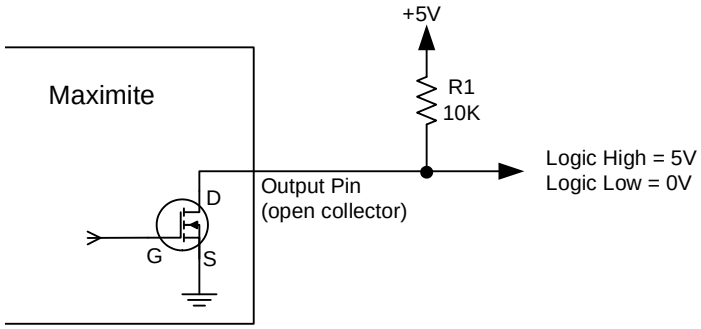
```
PIN(15) = 1
```

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_nbr, 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. For example, the module on the right provides four relays capable of switching 240V AC. It costs about US$8 and each relay can be controlled by an output pin 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_nbr, 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

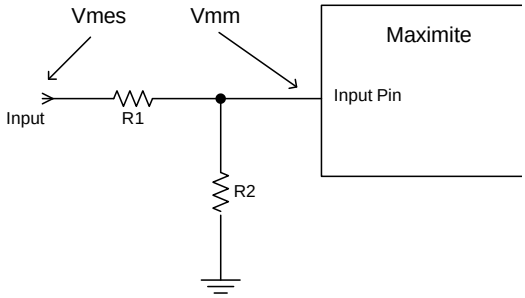
The Colour Maximite 2 has twelve I/O pins that are capable of voltage (ie, analog) measurement. To set an I/O pin to this mode you use the command:

```
SETPIN pin_nbr, AIN
```

Where AIN stands for Analog IN and 'pin_nbr' 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.

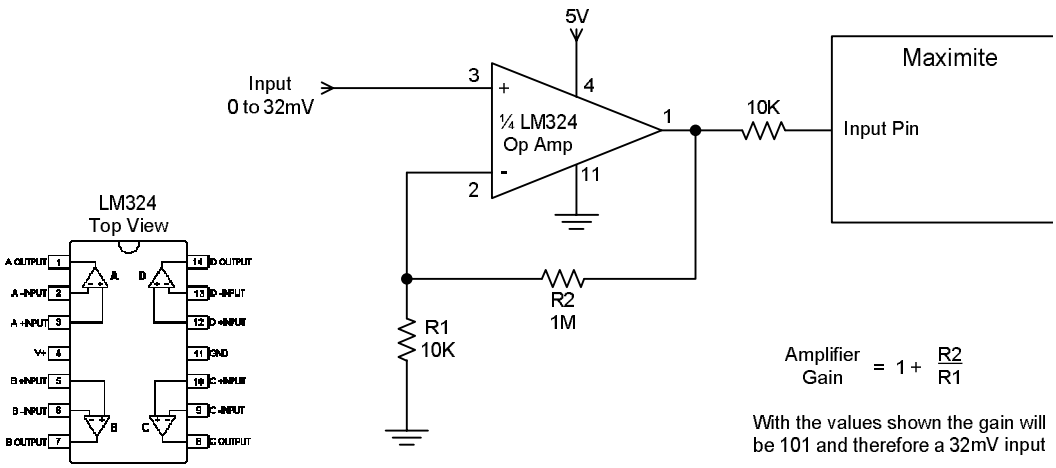


Then the reading could be scaled thus:

```
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. This is a typical example:



This is 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_nbr, type, subroutine
```

'pin_nbr' 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).

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.

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. When the subroutine is exited MMBasic will check for any other interrupt conditions (for example, an input that has gone high) that occurred while executing the interrupt subroutine. However, if your interrupt subroutine took too long to execute there is a chance that the other interrupt condition may have vanished in the meantime (for example, the input might have gone low again) – with the result that the new interrupt would be missed. For this reason interrupt subroutines should be as short as possible.

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

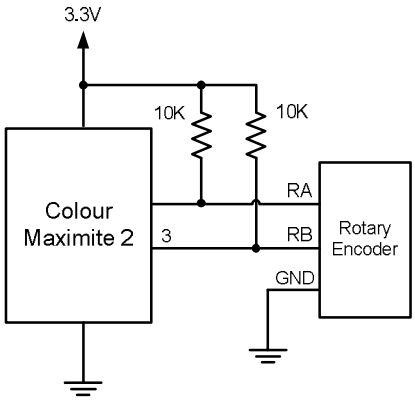
temperature, etc) so care needs to be taken if you are using any of these functions and interrupts at the same time.

- 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 or functions 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.

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 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 PWM 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.

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 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"
```

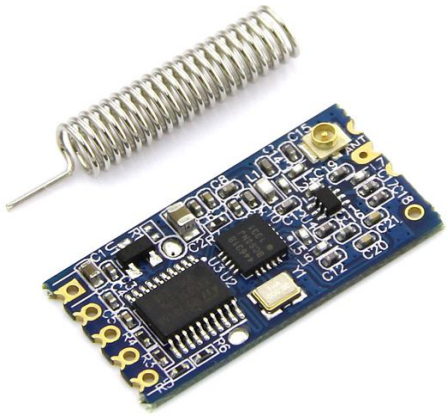
This will send a series of characters spelling "Hello" out of the serial port opened as #4. To receive characters from a serial port you can use a number of commands or functions but the most useful is the INPUT$(x, #n) function which will retrieve x characters from the port opened as #n.

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 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 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 " HEX$(RData(1)) ":" HEX$(RData(0), 2)
```

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 send 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 we have not covered so far in this tutorial but are useful in a program. These include accurate timekeeping, trapping errors and features designed to help the programmer when the computer is left 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 in an integer variable 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
```

By the way, you can also use the PULSE command to create a precisely timed pulse.

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 when the power is restored 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 selectively 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 count down to zero and the program will be restarted (if 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 instance, 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.

As an example, the following could cause a "Divide by zero" error if the variable `UserNumber` happened to be zero:

```
A = 34/UserNumber
```

One possible solution is to use the ON ERROR SKIP command which will instruct MMBasic to ignore any errors that may be caused by the next command:

```
ON ERROR SKIP
A = 34/UserNumber
```

In this case, if `UserNumber` was zero, the program will carry on as if nothing has happened. Note that this also means that 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, this feature 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 memory. The command is used like this:

```
VAR SAVE Var1, Var2, ..., etc
```

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.

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)               ‘ temporary copy of the keys
NEXT i
SORT temp$(), sortindex%()          ‘ sort the temporary array

FOR i  = 0 to 100
  k = sortindex%(i)                 ‘ index to the original array
  PRINT city$(k), pop%(k), area!(k) ‘ print in sorted order
NEXT i
```

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.