Getting Started With the Micromite

Geoff Graham

Version 8 (July 2020)

Table of Contents

Chapter 1

The Micromite is a high performance 32-bit microcontroller loaded with a sophisticated BASIC interpreter called MMBasic. With MMBasic and the Micromite you get the best of two worlds, a powerful microcontroller which is also easy to use and program.

In this chapter we will cover the hardware aspects of using the Micromite including selecting the best chip, programming it with the Micromite firmware and connecting to its console so that you can enter your program. Subsequent chapters will cover BASIC programming, input/output, timing and more.

This tutorial will go through many aspects of the Micromite and the BASIC programming language but it cannot cover everything. For example, many commands have additional features that are only used in special circumstances. So it would be worthwhile downloading the *Micromite 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.

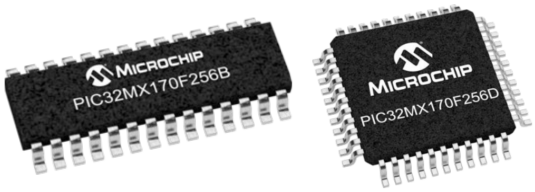
The Micromite firmware and manuals can be downloaded from: http://geoffg.net/micromite.html

Micromite Versions

You have a choice of four Micromite versions depending on the number of I/O pins you need.

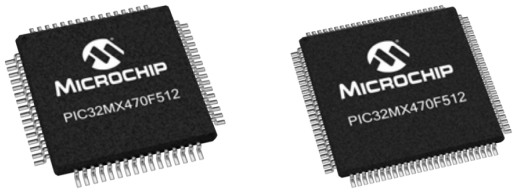
These are:

28-pin and 44-pin Standard Micromite



Except for the number of input/output pins these are identical. The 28-pin chip has 19 I/O pins while the 44-pin chip has 33 I/O pins. They both run the standard Micromite firmware which includes the full BASIC interpreter, support for many devices/sensors and support for touch sensitive LCD display panels from 2.2" to 2.8".

64-pin and 100-pin Micromite Plus



These have 45 and 77 input/output pins respectively. They run the Micromite Plus firmware which supports all the devices/features in the standard Micromite firmware but then adds advanced graphics, support for SD cards, USB, PS2 keyboard and support for ten different sized LCD panels from 1.4" to 8" (many with touch sensitivity).

When equipped with a large display panel (4.3" to 8") the Micromite Plus can also act as a complete stand alone BASIC computer with keyboard – rather like the TRS-80, Apple II and other computers from the 1980s.

Windows/DOS Version of MMBasic

If you just want to learn to program in BASIC the easiest route would be to download the Windows/DOS version of MMBasic. This version uses the same syntax and basic commands as the Micromite version and can be used for testing programs and features covered in this tutorial.

It can be downloaded from: http://geoffg.net/micromite.html#Downloads

On your PC MMBasic does not need any installation, all you need do is copy the executable file (MMBasic.exe) to the directory of your choice. The executable is fully self contained; there are no libraries or other files required. To run MMBasic just double click on the executable file (MMBasic.exe) and MMBasic will start running in a DOS box.

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

To run the program you have four choices:

1. Use RUN "filename" at the MMBasic command prompt (the greater than symbol '>'). Note that the double quotes are required around the file (for example, RUN "MYFILE.BAS")
2. Drag and drop the BASIC program file onto the MMBasic icon in Windows – this will cause Windows to start up MMBasic which will automatically run your program.
3. If you associate the file extension of .BAS to MMBasic.exe within the Windows operating system you can run your BASIC program simply by double clicking on the program file (with the .BAS extension).
4. Many editors like MMEdit, Notepad++ or Twistpad will let you define a single key that will save the file and run MMBasic with the file's name on the command line. This causes MMBasic to immediately run your program and results in a very fast edit/save/run cycle.

If you are using the DOS version of MMBasic you can jump straight ahead to Chapter 3 in this tutorial. Chapters 3 and 4 cover the essentials of programming in BASIC and all the examples in both chapters will run in the Windows/DOS version of MMBasic. You can ignore the rest of this tutorial as it covers features specific to the Micromite version of MMBasic.

Steps to Get Started

To get started with the Micromite there are a number of simple steps that you need to take. These are described in the following pages but in summary they are:

1. First you need to select a chip from the Microchip PIC32 family. There are a number of possibilities but if you are completely new to the Micromite the 28-pin version in a dual inline package (DIP) is a good choice as it is cheap and easy to use.
2. Next you need to connect the chip in a circuit with a power supply and a 47µF capacitor. If you are using the 28-pin chip this can be done using a solderless breadboard which is also handy for testing other circuitry in your design.
3. The Micromite firmware should then be loaded into the chip using a low cost programmer. You can completely skip this step if you wish by purchasing the PIC32 chip already programmed – a number of vendors will do this for you.
4. You then need to connect your personal computer (Windows, Apple or Linux) to the console of the Micromite. This is a serial interface used to communicate with the chip.
5. Using a terminal emulator on your personal computer you can issue commands to configure the Micromite, load the BASIC program, edit it, etc.
6. At this stage you are in a position to experiment with the Micromite and, once you have got the hang of programming in BASIC, develop and test your program.
7. Finally, when your program is doing what you want it to do, you can set the Micromite to automatically run its program when power is applied. You now have an intelligent controller which you can plug into your final project and it will theoretically run forever doing its job.

Selecting a Chip

When dealing with hardware this tutorial mostly assumes that you will be using the 28-pin Micromite as this is the cheapest and easiest version to get started with. Regardless, the majority of what we will cover will also apply to the 44-pin Micromite as well as the 64 and 100-pin Micromite Plus versions. Refer to the *Micromite User Manual* and the *Micromite Plus User Manual* for the full list of PIC32 chips supported by MMBasic.

You can use a number of different PIC32 microcontrollers for the 28-pin Micromite but the best is the Microchip PIC32MX170F256B-50I/SP. This is a 28-pin dual in line plastic package which can be plugged into an IC socket or a solderless breadboard. You can buy it ready programmed with the Micromite firmware or completely blank, in which case you will have to program it yourself. The following are some sources in no particular order (all will ship internationally).

Programmed Micromites, Modules and Kits

Micromite.org	http://micromite.org
Graeme Rixon (Rictech)	https://www.rictech.nz
Silicon Chip Online Store	http://www.siliconchip.com.au
CircuitGizmos	http://circuitgizmos.com
Mick Gulovsen	https://www.shop-dontronics.com/Micks-Mites

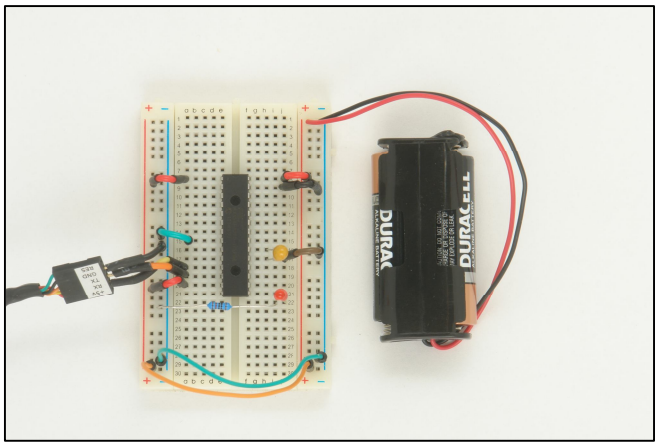
Unprogrammed (blank) Chips

Microchip	http://www.microchipdirect.com
Element14	http://au.element14.com
RS Components	http://au.rs-online.com/web

Note that when referring to the bare chip (without the interpreter loaded) it is described as a PIC32 microcontroller but after the MMBasic firmware has been programmed into the chip's flash memory, it becomes the Micromite.

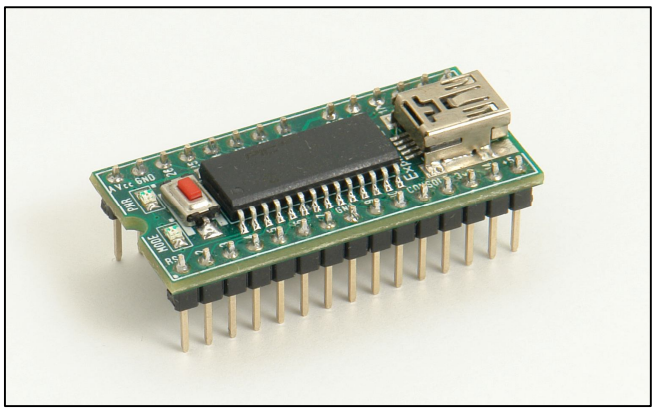
As you read through this tutorial you will need a Micromite to experiment with. You have many choices but the following are recommended:

Solderless Breadboard



A low cost setup is a solderless breadboard as shown on the right. Using one of these you can simply plug the 28-pin Micromite in and use jumper wires to connect it up. This allows for easy experimentation and when you have a working circuit you can transfer it to a more permanent setup such as a piece of stripboard or a custom designed circuit board.

Explore-28

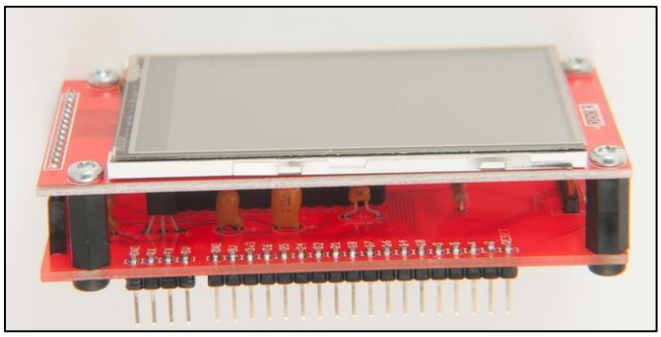


Many experimenters prefer to jump straight in without being tied up with the details of sourcing the components, programming the firmware into the chip, etc. For this the best choice is the Explore-28 from RicTech (https://www.rictech.nz or micromite.org in the UK)

This module is shown on the right and includes a pre programmed 28-pin Micromite and a USB-to-Serial bridge. It is fully assembled so you can plug it into a spare USB port on your laptop and within minutes be running your first program.

Because it is ready to go you can skip the next few sections of this tutorial covering the power supply, Vcap, USB-to-Serial interface and programming the firmware into the Micromite.

Micromite LCD Backpack



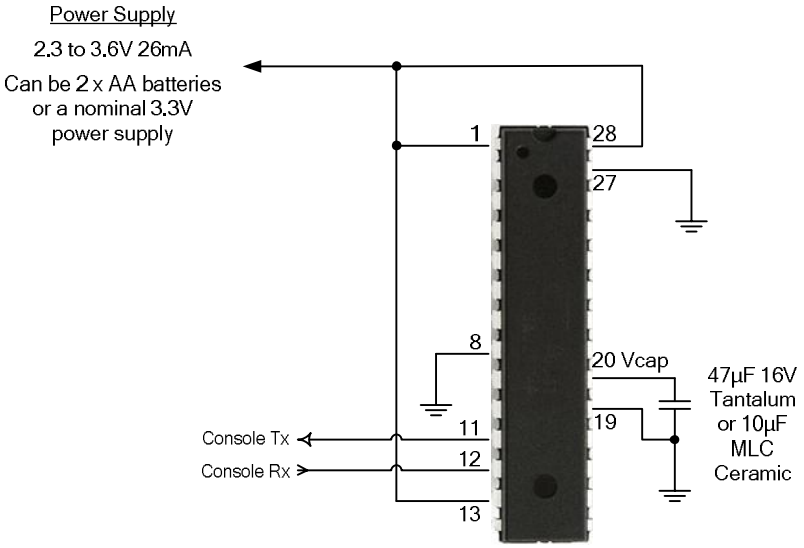
Another choice is the Micromite LCD Backpack which is shown on the right. This is available as a kit of parts from Silicon Chip magazine or micromite.org and uses the 28-pin Micromite. It includes a 5V to 3.3V power supply so the whole thing can be powered from a USB port.

The good thing about using the Micromite LCD Backpack is that you can directly plug in an LCD display to try out the graphics commands supported by the Micromite. We will cover these later in this tutorial.

You can find out more about the Micromite LCD Backpack including the construction details so you can build your own at http://geoffg.net/MicromiteBackpack.html.

Power Supply

The Micromite requires a DC power supply between 2.3V and 3.6V with 3.3V recommended. The following diagram shows the basic circuit for the 28-pin Micromite and it illustrates which pins should be connected to 3.3V (called Vdd) and ground (called Vss).



The PIC32 uses a high performance processing core and it can be susceptible to noise on its power supply. For this reason you should use a stable supply that you can trust.

A good source is two fresh alkaline batteries. Their maximum is 3.2V which sits well in the allowable range and they provide a noise free and stable supply. Using batteries means that you have one less thing to worry about when you are getting the Micromite to run for the first time. Later you can use other sources of power and it will then be easy to identify if that is causing a problem with the Micromite.

Vcap

The Micromite requires only one external component and this is the capacitor shown in the previous diagram between the pin marked Vcap and ground. If the capacitor is polarised its negative leg must be connected to ground. On the 28-pin chip Vcap is pin 20 and on the 44-pin chip it is pin 7.

This capacitor is critical and must be either a 10µF 16VW multilayer ceramic capacitor (MLC) or a 47µF 16 VW tantalum. Do not leave it out or substitute an electrolytic – the Micromite will simply not work.

The 47µF tantalum comes in a leaded (thru hole) package so it can be plugged into a solderless breadboard. The ceramic capacitor only comes in an SMD package and one technique is to solder it high on the shoulder between pins 20 and 19 of the 28-pin chip. This turns the chip into a ready to go package. You can just plug it into a breadboard and it does not need any additional components to function as a single chip computer.

Why is this capacitor so important? It is used to stabilise the 1.8V regulator inside the PIC32 chip. This supply is used to power the 32-bit CPU used by the microcontroller and without it the CPU will be unable to run.

Note that if the Micromite is part of a module (ie, Explore-28, 44-pin Micromite module, etc) this capacitor will already be in place on the module.

Micromite Firmware

With Vcap in place and power applied you can program the PIC with the Micromite firmware, thereby turning it into a Micromite. When the PIC32 is manufactured its flash memory is blank so when you load the Micromite firmware into that memory you are adding a layer of intelligence that will translate your commands, expressed in a high level language, to the basic bits and bytes necessary to do something.

In essence this is why the Micromite is so easy to use; this layer of software (the Micromite firmware) insulates you from the complexities of dealing with the registers and MIPS instruction set at the bare silicon hardware level. To illustrate the complexities of controlling the PIC32 chip at this level the Microchip PIC32 manuals, which describe these details, run to over 1,000 pages and that is a lot to read when you want to do a simple task.

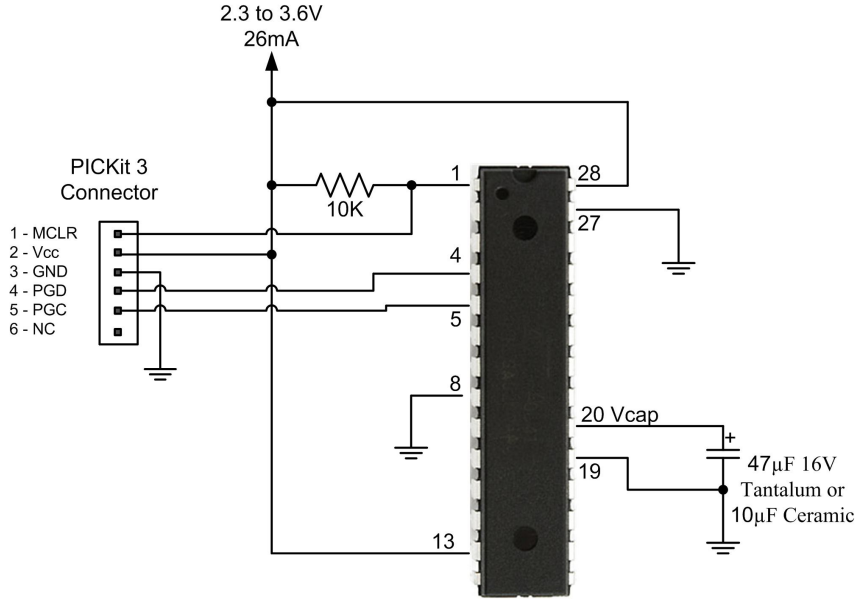
To get the Micromite firmware into the PIC32 you need a programmer and the best one for this job is the PICkit 3. This is a low cost programmer made by Microchip (manufacturer of the PIC32 chip). A genuine PICkit 3 costs about US$48 From http://microchipdirect.com but you can also buy clones which work just as well for less than US$20. If you search on eBay for "PICkit 3" you will get many hits.

If you do not want to buy a programmer you can buy the PIC32 pre programmed from a number of sources - see the above list of suppliers. Or, you can use the Explore-28 which includes its own programmer.

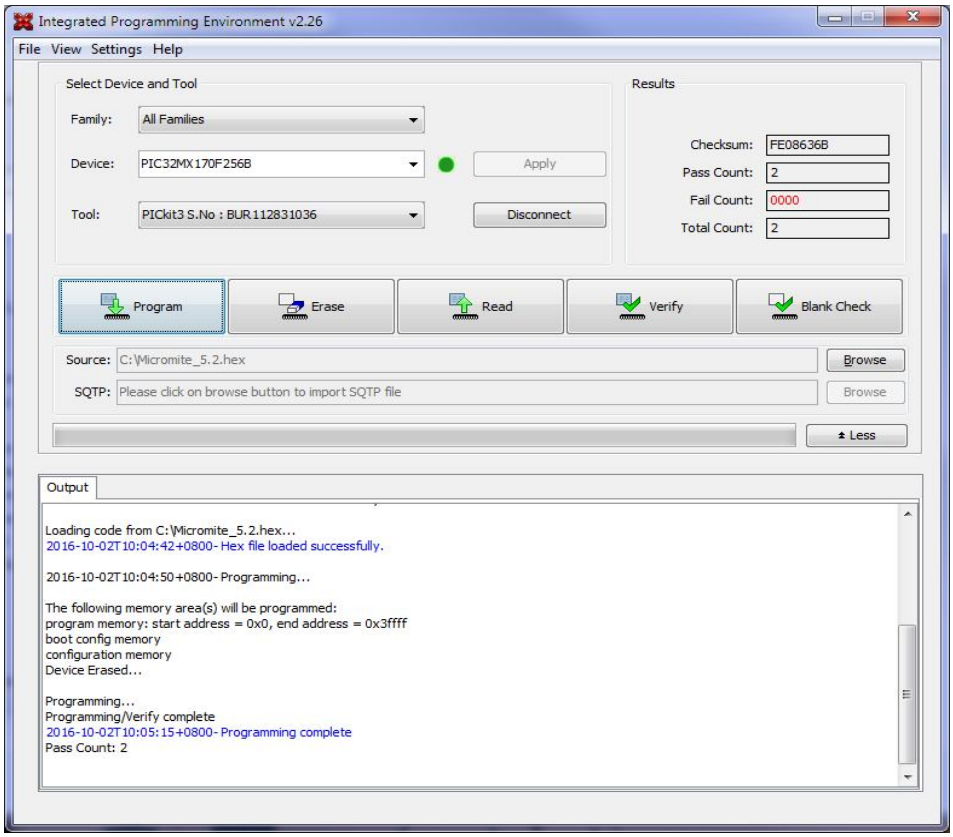
Programming the Chip

To program the chip using the PICkit 3 you need to download the MPLAB X development environment from Microchip. This is free software and comes in versions that will run on Windows, the Mac or Linux. When you install it you will be given the opportunity to install just the programmer component (IPE) and this is recommended as most people will not need the other components of MPLAP X. IPE stands for Integrated Programming Environment and this is the part that knows how to program the PIC32 chip using the PICkit3.

The PICkit3 then needs to be connected to the chip being programmed. In the case of the 28-pin Micromite you should connect it as shown below.



If you are programming the 44-pin module (link) you should plug the PICkit3 onto the six pin programming headers on top of the board. The MPLAB IPE interface is reasonably self explanatory as shown below. This screen shot was taken just after successfully programming the Micromite firmware into a 28-pin Micromite.



First you need to select the chip that you will be using, then select the programmer that you will be using (ie, the PICkit 3) and finally click on the "Connect" button. IPE will then try to connect to the PICkit 3 and check that the chip to be programmed is present.

You then need to load the Micromite firmware file (a file with a .hex extension) into MPLAB IPE by clicking on the "Browse" button and selecting the file. Finally you can click the "Program" button to start the programming sequence. IPE will first erase the flash memory in the chip, then program it with the contents of the .hex file (previously loaded) and finally verify the programming operation by reading back what was written and comparing it to the original file.

If the IPE reports "Programming Complete" you can be confident that the chip was correctly programmed. Of course it may not be as easy as that! Three of the more common errors that you might encounter are:

"Target Vdd not detected". The first thing that the PICkit 3 will check is if it can see the chip's power supply voltage on pin 2 of its connector and if it is not present you will get this error.

"Failed to program device" or "Cannot read device ID". The programmer could tell that you had connected it to something (because Vdd was present) but it could not communicate with the chip. This usually means that something was interfering with the MCLR, PGD and/or PGC lines (ie, they were not connected or other components were loading down the signals).

"Target Device ID does not match expected Device ID". This means that the programmer has detected a chip but it is different from the one that you specified in the Device drop down list.

The Console

To program the Micromite using MMBasic you use the console. The console is a serial interface over which you can issue commands to configure the chip, load the BASIC program and run it. MMBasic also uses the console to display error messages. The console is the only method of communicating with the Micromite and programming it so it is important that you can connect to it.

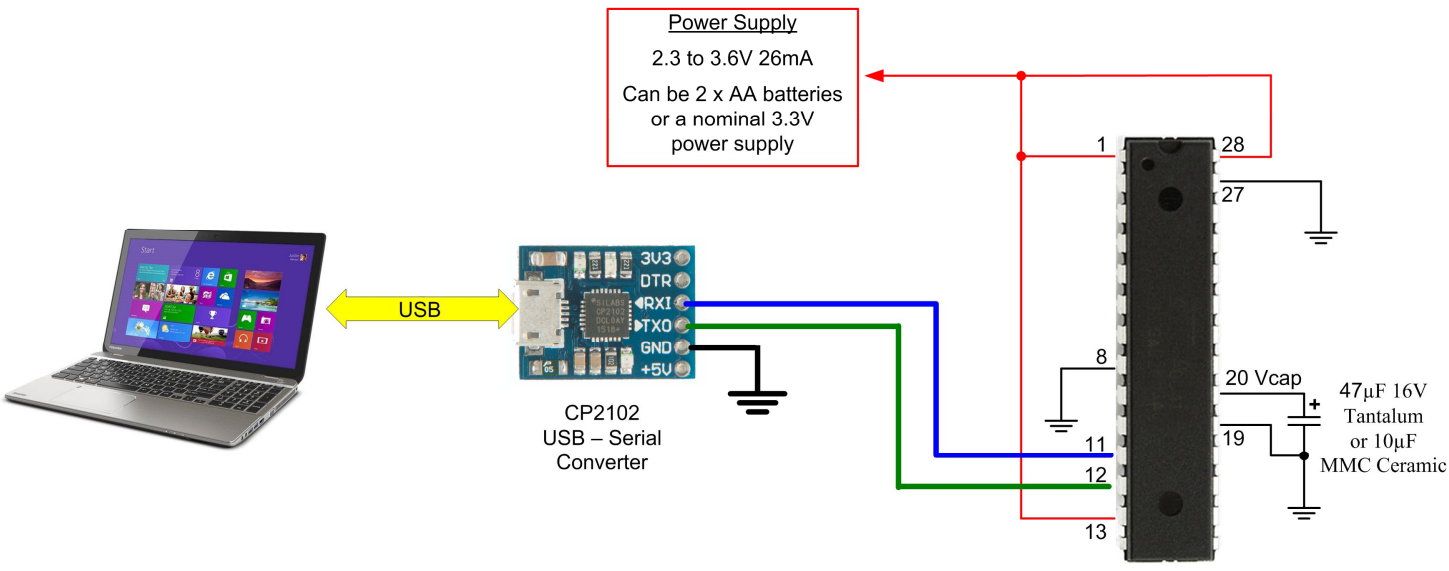
A serial interface consists of two signals. One, often referred to as Tx (for transmit) will send a coded signal to the other device and the second (called Rx) will receive a similar signal. The data is sent with start and stop bits and uses the ASCII coding for each character sent or received. The speed of transmission is referred to as a baud rate which is another way of saying bits per second. The Micromite starts up with its console serial port set to a baud rate of 38400.

The signal level is TTL which means that the signal will swing between zero and 3.3 volts. There are other signalling methods, notably RS232 which swings the voltage from -12 to +12 volts and the Micromite can work with this but it is simpler to use the standard TTL interface.

To use the serial console you need a USB to serial converter which will plug into a USB port on your desktop computer and on the other end connect to the Micromite's serial console. From your computer's point of view it will look like a serial port (via USB) while the connection to the Micromite Plus is a standard serial interface with TTL signals levels.

I recommend converters based on the CP2102 chip and they can be found for a few dollars on eBay (search for "CP2102"). You should avoid converters based on the FTDI FT232RL chip as many Chinese manufacturers use non genuine chips which will not work with the current Windows drivers.

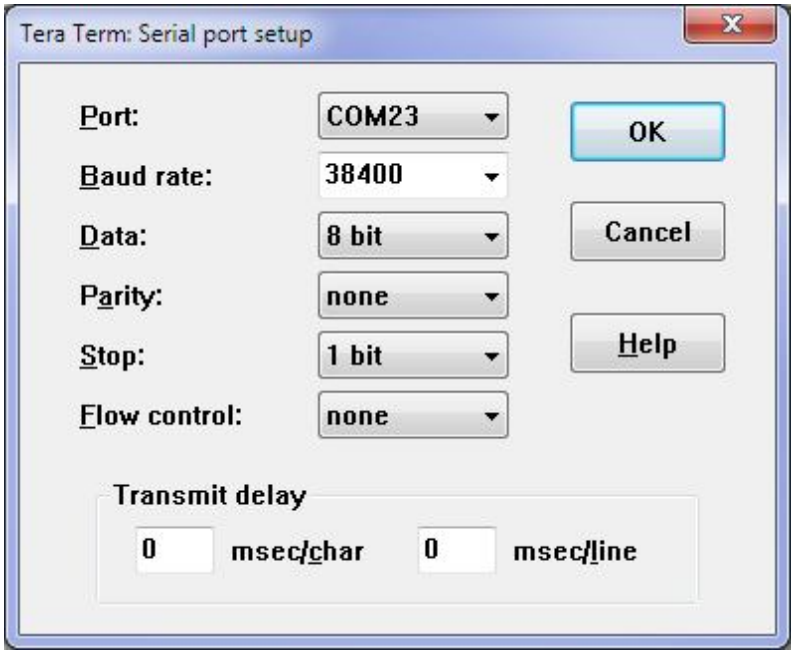
The diagram below shows how such a converter can be used to connect a personal computer to the console of a 28-pin Micromite. Note that this diagram shows the Micromite with an independent 3.3V power supply because sometimes the 3.3V output from the converter can be as high as 4.3V. **If you plan on using this output to power your Micromite make sure that you measure its actual voltage first as the Micromite is rated to only 3.6V.**



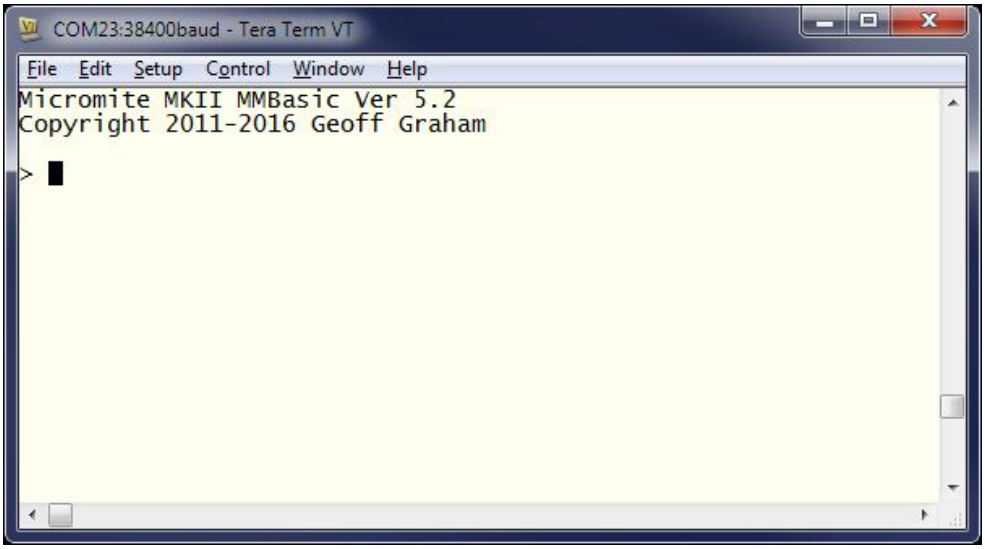
The drivers for the CP2102 are in Windows 10 but for others (Windows 7, Mac, Linux) they can be downloaded from http://www.silabs.com/products/mcu/pages/usbtouartbridgevcpdrivers.aspx. When the converter is plugged into your computer, and the correct driver is installed the CP2102

will appear as a serial port (eg, COM29 in Windows). To find the port number on a Windows computer go to Device Manager and open the Ports (COM & LPT) entry.

You then need to start a terminal emulator on your computer. For Windows I recommend Tera Term version 4.88 which is free from: http://tera-term.en.lo4d.com. Set the interface speed to 38400 baud, 8 bits, no parity and one stop bit. The following shows how Tera Term should be configured (note that the port number will vary depending on your computer's setup):



Within your terminal emulator connect to the serial port created by the USB to serial converter and reset the Micromite (ie, cycle the power). In the terminal emulator's window you should see the Micromite's startup banner as shown below.



At this point you can enter, edit and run programs from the command prompt using nothing more than the terminal emulator and a USB cable.

When your program is running successfully on the Micromite you do not necessarily need the console so you can set the Micromite to automatically run its program on startup (OPTION AUTORUN). However, unless you managed to get the program perfectly correct the first time

(unlikely) you will find yourself repeatedly reconnecting to make one tweak or another, so many people just leave the USB to serial converter permanently connected (they do not cost much).

Trouble Shooting

What if it does not work the first time?

The first step is to check your power supply. Is it 3.3V and is it stable and free from electrical noise? USB power sources such as USB chargers often cause trouble so if you have some doubts you can use two fresh AA batteries in series as a power source for testing.

If the power is OK check your wiring against the circuit on page 7 and in particular check that 3.3V is on each pin as specified and that each ground pin is correctly connected to ground. Some cheap solderless breadboards may have some contacts that do not connect properly so watch out for that.

The next item to check is the capacitor connected to pin 20 on the 28-pin chip or pin 7 on the 44-pin chip. As specified earlier in this tutorial the capacitor must be a tantalum (47µF) or ceramic (10µF), an electrolytic capacitor will not work.

Has the chip been properly programmed? If you programmed it yourself you should check that the programmer did report that the programming operation was successful.

Check the current drawn by the chip – a draw of about 26mA for the standard Micromite or about 70mA for the Micromite Plus means that the chip is working correctly and running the BASIC interpreter.

A current of less than 10mA indicates that MMBasic is not running and:

1. a power or ground connection is faulty.
2. the 47µF (or 10µF) capacitor is faulty or not connected.
3. The chip was not programmed correctly.

If you have a current draw that is about correct it indicates that the fault is most likely with the USB to serial converter or your terminal emulator. To check these two elements you can disconnect the serial connections from the Micromite and short the Tx and Rx pins of the converter together. When you type something into the terminal emulator window on your computer you should then see the characters echoed on the screen. If not you should diagnose and correct the error in your USB to serial converter and terminal emulator before proceeding.

If the above test is OK (ie, keystrokes echo on the screen) the only possible remaining fault is in your connection of the USB to serial converter to the Micromite. Check that the Tx pin on the converter goes to the Micromite's Rx pin and that Rx on the converter goes to the Micromite's Tx pin as shown in the above diagram.

What Next?

Now that you have your Micromite running you can try your hand at programming it in BASIC and that is the subject of the next three chapters. But here are a few things that you can try out first, just to prove that you have a working computer in a chip.

All of these commands should be typed at the command prompt (">"). To make it clear - what you type is shown in **bold** and the Micromite's output is shown in normal text.

Try a simple calculation:

```
> PRINT 1/7
0.142857
```

See how much memory you have:

```
> MEMORY
Flash:
   0K ( 0%) Program (0 lines)
  60K (100%) Free

RAM:
   0K ( 0%) 0 Variables
   0K ( 0%) General
  50K (100%) Free
```

What is the current time? Note that the Micromite's clock starts at midnight on power up.

```
> PRINT TIME$
00:04:01
```

Set the clock to the current time:

```
> TIME$ = "10:45"
```

Check the time again:

```
> PRINT TIME$
10:45:09
```

How many milliseconds have elapsed since power up:

```
> PRINT TIMER
 440782
```

Count to 20:

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

Chapter 2

Interacting With MMBasic

The Micromite is rather like a self contained computer. You can write programs, run them, edit them and so on, all on the Micromite itself. And if your program has an error (as initially all do) you will get a clear message showing you where the fault is located and what the problem is. This is different from other microcontrollers where you build your program on a separate computer then download the compiled result to the chip to be run. In that case you do not get any feedback on errors while your program is running… it will just produce crazy results and you will have no idea where or what in your program caused the problem.

This is an important part of working with the Micromite – it is much easier and quicker to build a functioning program when you can get immediate feedback and then, with one keystroke, jump into an editor to fix the issue and run the program again.

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

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

Almost any command can be entered at the command prompt and this is often used to test a command to see how it works. A simple example is the PRINT command (more on this 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 Micromite handy for the occasional test while you are working through this tutorial.

Break Key

One useful feature of the Micromite 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 the Micromite 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.

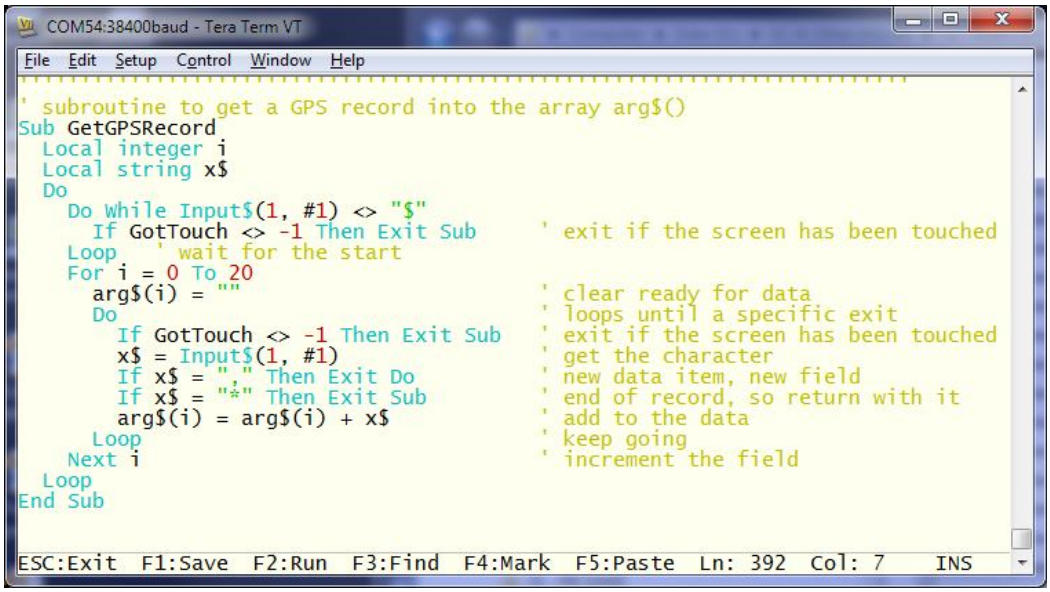
```
DO : LOOP
```

If you have a Micromite 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 the Micromite 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.

The Editor

The Micromite 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 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 to memory and run it. This will display the result of dividing 1 by 7.

To change this program use the command EDIT again 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 if you really want to discard your changes.
F1: SAVE	This will save the program to program memory and return to the command prompt.
F2: RUN	This will save the program to program memory and immediately run it.
F3: FIND	This will prompt for the text that you want to search for. When you press enter the cursor will be placed at the start of the first entry found.
SHIFT-F3	After you have used the search function once you can repeatedly search for the same text by pressing SHIFT-F3.
F4: MARK	This is described in detail below.
F5: PASTE	This will insert (at the current cursor position) the text that had been previously cut or copied to the Micromite's clipboard (see below).

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

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

One point to note is that you cannot use the Windows Copy and Paste function to paste text into the terminal emulator and expect the editor to accept it. This is because the editor has a lot of work to do for each character received and it cannot keep up with a high speed transfer like that. If you do want to copy and paste a program into the terminal emulator you should use the AUTOSAVE command described below.

Loading a Program from a PC

If you prepare a program on your desktop computer you can transfer it to the Micromite using either the AUTOSAVE or XMODEM commands. This requires you to have a terminal emulator running on your desktop machine and connected to the console of the Micromite. How to do this was described in the previous chapter.

The AUTOSAVE command puts the Micromite into a mode where anything received on the console will be saved to the program memory. This means that you can simply copy the text and paste it into the terminal emulator (eg, Tera Term) which will send it to the Micromite. From the Micromite’s perspective pasting text into the terminal emulator is the same as if a high speed typist was typing in the program. To terminate the AUTOSAVE command you need to press the Control-Z keys in the terminal emulator and the Micromite will save the program to its memory and return to the command prompt.

You can try this if you have a Micromite in front of you. At the console type AUTOSAVE and press Enter. You will see nothing on the screen because the Micromite is waiting for the input.

Copy the program to your computer's clipboard and paste it into your terminal emulator (in Tera Term this is done using ALT-V). You should see the program text being echoed back. Enter CTRL-Z and MMBasic will confirm that it has saved the program. Then it is just a case of entering RUN at the command prompt.

The XMODEM command is more sophisticated, it uses the XModem protocol to transfer a BASIC program file from your personal computer to the Micromite including an integrity check which will detect any errors in the transfer. This program file will be saved in the Micromite's memory ready to be run.

To start the XModem transfer you need to tell the Micromite to receive the file. The command to do this is: `XMODEM RECEIVE`

This instructs the Micromite to look for an XModem connection on the console. After running this command you should then instruct your terminal emulator to send the file using the XModem protocol. In Tera Term (running on your PC) this is done by using the following menu selection:

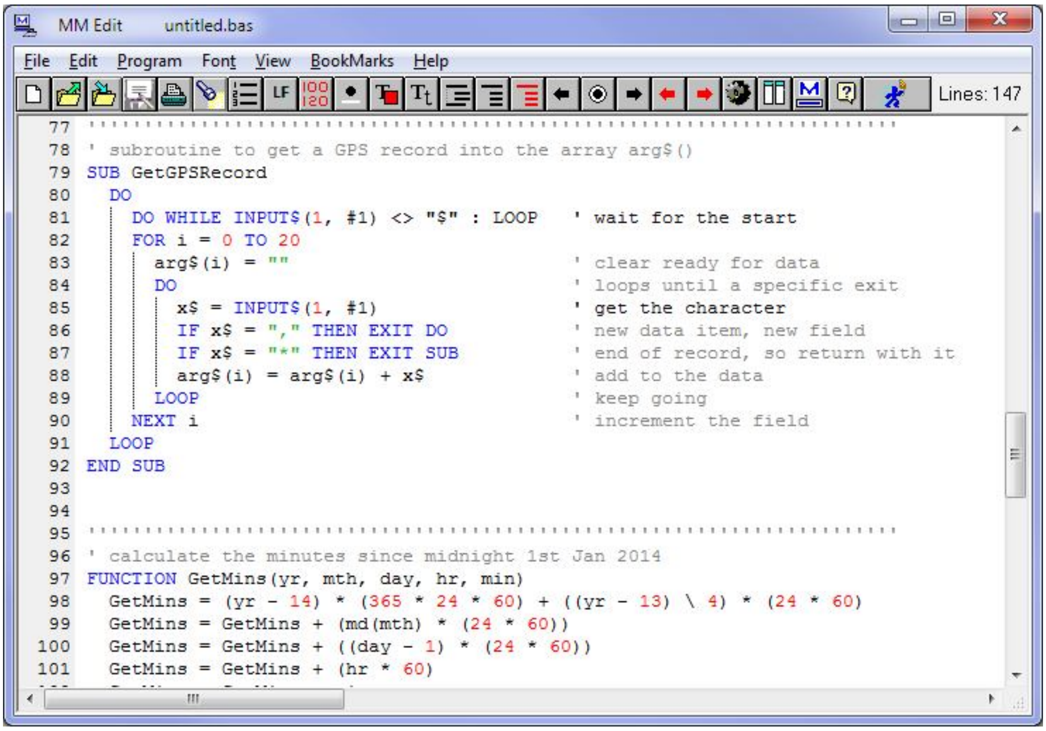
File -> Transfer -> XMODEM -> Send

This will present a dialog box where you can enter the name of the file to be sent and when you click on OK the transfer will start. When the complete file has been received the Micromite will save it in program memory and return to the command prompt.

MMEdit

Another convenient method of creating your programs and sending it to the Micromite is to use MMEdit. This program was written by Jim Hiley from northern Tasmania in Australia and is intended to work with the Micromite 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 Micromite for testing.

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.



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 Micromite using the XModem protocol. Following the transfer a window will be automatically opened and connected to the Micromite's 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 to the Micromite 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.

Setting Options

There are many options that you can set in the Micromite 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 Micromite 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 is a summary of the important 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 BAUDRATE nbr	Change the baudrate of the console
OPTION BREAK nn	Set the character that will break out of a program
OPTION CASE UPPER | LOWER | TITLE	Set the format for listing commands
OPTION CLOCKTRIM ±n	Trim the Micromite's internal clock speed
OPTION COLOURCODE ON | OFF	Enable colour coding in the program editor
OPTION CONSOLE ECHO | NOECHO	Control if the console will echo characters
OPTION CONSOLE INVERT | NOINVERT	Invert the console signal (for RS232)
OPTION CONSOLE AUTO	Allow the console to automatically detect RS232
OPTION DEFAULT FLOAT | INTEGER | STRING | NONE	Set the default type of a variable
OPTION DISPLAY lines [,chars]	Size of the display for listing programs
OPTION EXPLICIT	Require that variables be properly declared
OPTION LCDPANEL	Configure the Micromite to use an LCD panel
OPTION PIN nbr	Set a number to secure the console
OPTION RESET	Reset all options to their defaults
OPTION TAB 2 | 4 | 8	Set the size of tabs
OPTION TOUCH	Configure a touch sensitive LCD panel

Note that the vertical bar between words (eg, ECHO | NOECHO) means that you can use either one or the other (eg, ECHO or NOECHO) in the command. For example, OPTION CONSOLE ECHO.

Chapter

3

Programming Fundamentals

The Micromite is programmed using the BASIC programming language. The Micromite version of BASIC is called MMBasic (short for MicroMite 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 70s and 80s. Even today some large commercial data systems are still written in the BASIC language (primarily Pick Basic).

For the Micromite 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 control the I/O pins on the Micromite, interface to things like an LCD display or IR remote control, and communicate with other chips using a range of built-in communications protocols.

The only significant downside to using a BASIC interpreter is that it is not as fast as a fully compiled language like C or C++. However, on a fast microcontroller such as the PIC32 MMBasic will execute each command in an average of 50µs or faster. This speed is suitable for most applications - for example, it is fast enough to respond to a high revving engine and deal with signals such as ignition triggers. If you need more speed you can always embed compiled C functions in your BASIC program (see Chapter 6).

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 book). 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 tell you how your program is running and can consist of something simple such as "Pump running" or "Total Flow: 23 litres".

PRINT is also useful when you are tracing a fault in your program; you can use it to print out the values of variables, I/O pins 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 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
 0.142857
>
```

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 "Hello World"
Hello World
> PRINT (999 + 1) / 5
 200
>
```

You can try these out at the command prompt.

The PRINT command will also work with multiple values at the same time, for example:

```
> PRINT "The amount is" 345 " and the second amount is" 456
The amount is 345 and the second amount is 456
>
```

Normally each value is separated by a space character as shown in the previous example but you can also separate values with a comma (,). The comma will cause a tab to be inserted between the two values. In MMBasic tabs in the PRINT command are eight characters apart. To illustrate tabbing the following command prints a tabbed list of numbers:

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

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

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

The print statement can be terminated with a semicolon (;). This will prevent the PRINT command from moving to a new line when it completes printing all the text. For example:

```
PRINT "This will be";
PRINT " printed on a single line."
```

Will result in this output:

```
This will be printed on a single line.
```

The message would be look like this without the semicolon at the end of the first line:

```
This will be
 printed on a single line.
```

Variables

Before we go much further we need to define what a "variable" is as they are fundamental to the operation of the BASIC language (in fact, any programming language). A variable is simply a place to store an item of data (ie, its "value"). This value can be changed as the program runs which why it is called a "variable".

Variables in MMBasic can be one of three types. The most common is floating point and this is automatically assumed if the type of the variable is not specified. The other two types are integer and string and we will cover them later. A floating point number is an ordinary number which can contain a decimal point. For example 3.45 or -0.023 or 100.00 are all floating point numbers.

A variable can be used to store a number and it can then be used in the same manner as the number itself, in which case it will represent the value of the last number assigned to it.

As a simple example:

```
A = 3
B = 4
PRINT A + B
```

will display the number 7. In this case both A and B are variables and MMBasic used their current values in the PRINT statement. MMBasic will automatically create a variable when it first encounters it so the statement `A = 3` both created a floating point variable (the default type) with the name of `A` and then it assigned the value of 3 to it.

The name of a variable must start with a letter while the remainder of the name can use letters, numbers, the underscore or the full stop (or period) characters. The name can be up to 32 characters long and the case (ie, capitals or not) is not important. Here are some examples:

```
Total_Count
ForeColour
temp3
count
```

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 manual (around page 52).

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 expression.

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 functions the value passed to the function (ie, 'r') is the angle in radians. In MMBasic you can use the function RAD(d) to convert an angle from degrees to radians ('d' is the angle in degrees).

Another feature of BASIC is that you can nest function calls within each other. For example, given the angle in degrees (ie, 'd') the sine of that angle can be found with this expression:

```
PRINT SIN(RAD(d))
```

In this case MMBasic will first take the value of d and convert it to radians using the RAD() function. The output of this function then becomes the input to the SIN() function.

The IF statement

Making decisions is at the core of most computer programs and in BASIC that is usually done with the IF statement. This is written almost like an English sentence:

IF *condition* THEN *action*

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

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

`Temp` would be a variable holding the current temperature.

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 you also want to take some actions if false:

```
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 Temp < 25 THEN
    PRINT "Cold"
    TurnRedLightOff
ELSE
    PRINT "Hot"
    TurnRedLightOn
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. You will find that this tutorial uses indenting in all examples for this reason.

An expression like `Temp < 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 `SwitchOn` is a variable that is true (non zero) when some switch is turned on the following can be used to make a decision based on that value:

```
IF SwitchOn 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. Try it on a Micromite, use the EDIT command to enter the short program listed above then press the F2 key to save and run it.

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 checks 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:

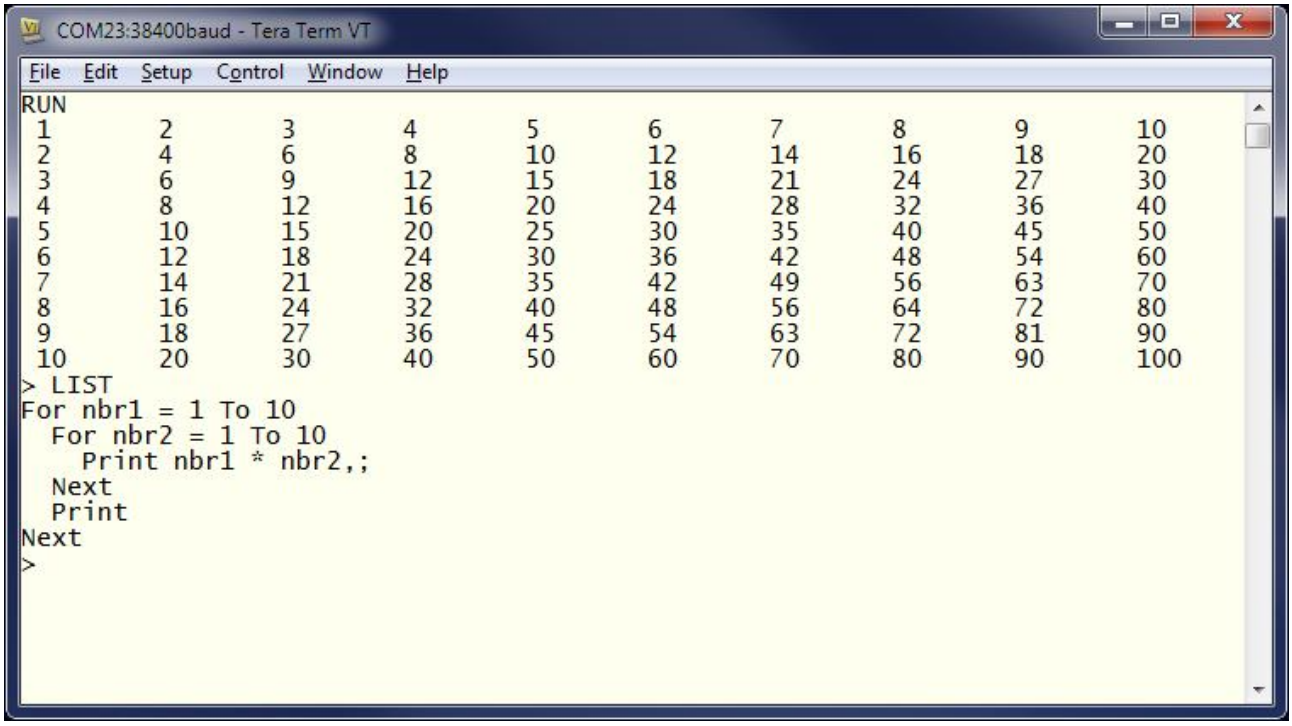
```
FOR nbr = 10 TO 1 STEP -1
  PRINT nbr,;
NEXT nbr
```

Multiplication Table

To further illustrate how loops work and how useful they can be, the following short program will use the FOR loop 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 Micromite. 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

There are times where you would like to use the Micromite as a straight computer with all of its input coming from the console and its output going to the same place. 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:

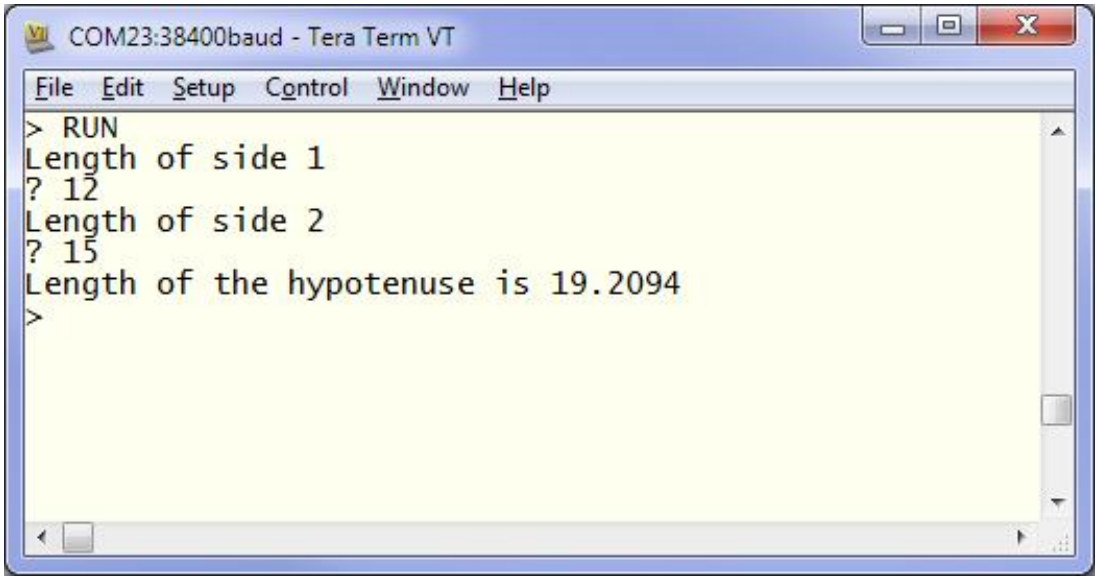
```
INPUT var
```

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 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 (also covered in the next chapter).

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 used be 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.

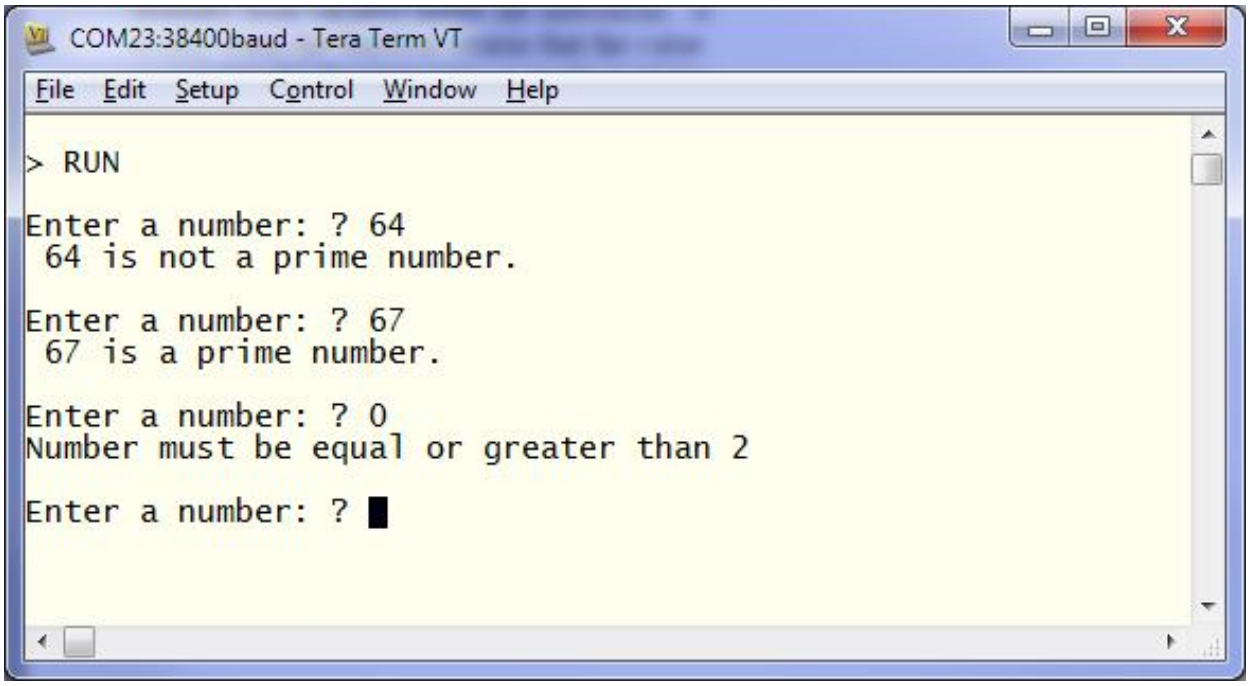
Following this the number is tested. If it is less than 2 an error message will be printed and the program will jump backwards and ask for the number again.

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).

Saving the Program

When you entered the program then saved it you might wonder where your program has actually been saved to. The answer is that it was automatically written in the flash memory of the PIC32 chip. In fact, if you save a very large program you might see a delay of a second or two which is the time needed by MMBasic to transfer a large amount of data into the flash memory.

Flash memory is non volatile which means that it will retain its contents when the power is removed. This might not be important for a program that checks for prime numbers but if you have programmed the Micromite to be the brains in your burglar alarm you will not want it to lose the program during a blackout.

The command MEMORY will report on how much memory was used by the program. With the above program it will display something like this:

```
Flash:
   1K ( 1%) Program (19 lines)
  59K (99%) Free

RAM:
   1K ( 1%) 4 Variables
   0K ( 0%) General
  49K (99%) Free
```

As you can see, the program used little memory. This is another advantage of the Micromite; the relatively huge memory space means that you can create large and complex programs and still run them on this small and inexpensive chip.

Chapter 4

Advanced BASIC Programming

In the previous chapter we covered the fundamentals of BASIC programming, enough to write a small 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 *Micromite User Manual* from: http://geoffg.net/micromite.html

Utility Commands

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.

The following are some examples of comments:

```
' calculate the hypotenuse
PRINT SQR(a * a + b * b)
```

or

```
INPUT var     ' get the temperature
```

We have covered the EDIT command which will run the internal Micromite program editor. Other useful commands are 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 erase everything leaving you with the maximum free program memory. The CLEAR command will do the same for any variables (ie, delete them and recover the memory). As shown before, 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 as mentioned in chapter 2. This takes many forms and using it you can change many settings within MMBasic including how programs are listed, the attached devices (such as LCD panels), how your program will be run and much more.

Arrays

An array is 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). 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 you can have up to eight dimensions and the maximum size of an array is only limited by the amount of free RAM that is available in the Micromite.

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 only 6 or 7 digits in the 28 and 44-pin Micromite or 15 to 16 digits in the Micromite Plus. For example, if you stored the number 1234.56789 in a floating point variable on a 28-pin Micromite then printed it out you will find that the value stored in the variable is actually 1234.57.

The limit on the accuracy of floating point can cause confusion for new programmers. For example, a beginner might write the following program for a 28-pin Micromite and be surprised when the value of `F` does not change.

```
F = 1234.5
PRINT F
F = F + 0.0001
PRINT F
```

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. Examples include tracking a GPS location on the planet's surface or interfacing with digital frequency synthesisers.

As another example, let us say that you want to manipulate time accurately 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 seconds since some date (say 1st Jan 1900) - 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 seconds since that date would far exceed the accuracy range of floating point variables and this is where integer variables come in. An integer variable in MMBasic 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 eight 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.

The section on serial communications in chapter 7 has a good example of using this function.

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.

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 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 a variable that represents a value without the risk of the value being accidently changed - which can happen if standard variables were used for this purpose. These are called constants and they can represent 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:

```
CONST BatteryVoltagePin = 26
CONST BatteryMinimum = 11.5
```

These constants can then be used in a 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). Using the above as an example, you might replace the battery with a different technology and therefore you need to change the minimum battery voltage. That could be easily accomplished if you originally defined this value as a constant at the start of your program.

Subroutines

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

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

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

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

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

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

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

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

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

```
ErrMsg "Number too small"
```

When the subroutine is called like this the message "`Error: Number too small`" will be printed on the console. Inside the subroutine `Msg$` will have the value of "Number too small" when called like this and it will be concatenated in the PRINT statement to make the full error message.

A subroutine can have any number of arguments which can be float, integer or string with each argument separated by a comma. Within the subroutine the arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be hidden within the subroutine and be different from arguments defined for the subroutine.

The type of the argument to be supplied can be specified with a type suffix (ie, $, % or ! for string, integer and float). For example, in the following the first argument must be a string and the second an integer:

```
SUB MySub Msg$, Nbr%
  …
END SUB
```

MMBasic will convert the supplied values if it can, so if your program supplied a floating point value as the second argument MMBasic will convert it to an integer. If MMBasic cannot convert the value it will display an error. For example, if you supplied a string for the second argument your program will stop with an error.

You do not have to use the type suffixes, you can instead define the type of the arguments using the AS keyword similar to the way it is used in the DIM command. For example, the following is identical to the above example:

```
SUB MySub Msg AS STRING, Nbr AS INTEGER
  …
END SUB
```

Of course, if you used only one variable type throughout the program and used OPTION DEFAULT to set that type you could ignore the question of variable types completely.

Within a subroutine you can use most features of MMBasic including calling other subroutines, IF…THEN commands, FOR…NEXT loops and so on. However one thing that you cannot do is jump out of a subroutine using GOTO (if you do the result will be undefined). Normally the subroutine will exit when the END SUB command is reached but you can also terminate the subroutine early by using the EXIT SUB command.

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. However, inside a subroutine you will often need to use variables for various tasks that are internal to the subroutine. 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.

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 because they help make the subroutine 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 starts, however there are times when you would like the variable to retain its value between calls to the subroutine. This type of variable is defined with the STATIC command.

We can demonstrate how STATIC variables are useful by extending the ErrMsg subroutine to prevent duplicated calls to the subroutine repeatedly displaying the same message. For example, our program might call this subroutine from multiple places but if the message is the same in a number of subsequent calls we would like to see the message just once.

To keep track of the last message displayed we use a static variable called `lastmsg`. This will hold the text of the last message and we can compare it to the current message text to determine if it is different and therefore should be printed.

This is our new subroutine:

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

This would give just one message every time a call is made with a duplicate message text.

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

The static variable is created on the first time the STATIC command is encountered and it is automatically set to zero (if a float or integer) or an empty string. On subsequent calls to the subroutine 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. 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 subroutine and this is where static variables are valuable. Just like LOCAL the use of STATIC helps to make your subroutines more self contained and portable.

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). Functions can also use LOCAL and STATIC variables just like subroutines.

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.

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.

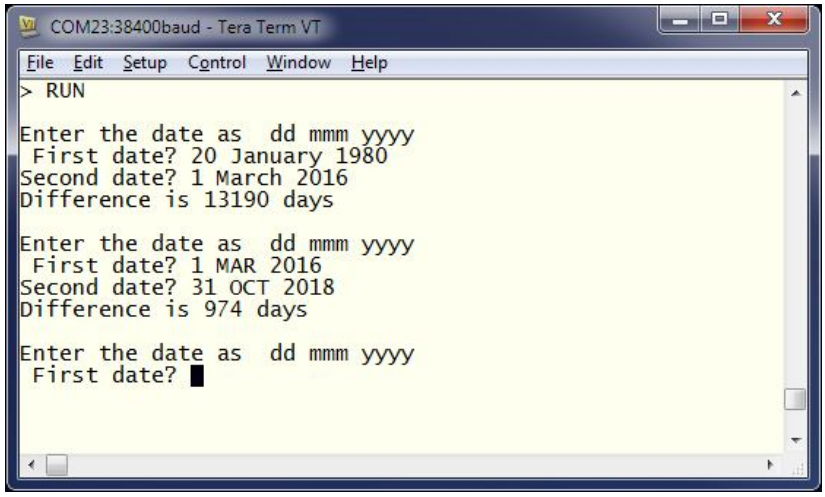
It works by getting two dates from the user at the console and then converting them to integers representing the number of days since 1900. With these two numbers a simple subtraction will give the number of days between them.

```
' Example program to calculate the number of days between two dates

OPTION EXPLICIT
OPTION DEFAULT NONE

DIM STRING s
DIM FLOAT d1, d2

DO
  PRINT
  PRINT "Enter the date as  dd mmm yyyy"
  PRINT " First date";
  INPUT s
  d1 = GetDays(s)
  IF d1 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Second date";
  INPUT s
  d2 = GetDays(s)
  IF d2 = 0 THEN PRINT "Invalid date!" : CONTINUE DO
  PRINT "Difference is" ABS(d2 - d1) " days"
LOOP

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

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

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

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

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

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

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 *Micromite 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. Normally only a very small part of a program needs to run fast and if that is the case that portion would be better being written in C which can be embedded in the BASIC 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

Micromite Input/Output

The Micromite has an extensive range of input/output facilities which allow your BASIC program to interact with the outside world. These are crucial because the Micromite is essentially designed to be an embedded controller.

The input/output features of the Micromite include:

- Digital inputs.
- Digital outputs
- Analog inputs.
- Frequency, period and counting inputs.
- Pulsed digital outputs.
- Pulse width measurement inputs.
- Pulse Width Modulated (PWM) outputs.
- Servo driving outputs.
- A range of special devices for measuring temperature, distance and more.

Configuring a Pin

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

```
SETPIN nn, mode
```

where 'nn' is the pin number on the Micromite and 'mode' is how you would like the pin to be configured. This last parameter can be:

AIN	Analogue 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 14, DOUT` will setup pin fourteen as a digital output.

Note that the pin number 'nn' refers to the physical pin number of the chip as shown in the data sheet. This makes it easy for you to cross reference a component connected to the chip with the programming commands that will manipulate it.

To read from an input pin you use the PIN() function. For example:

```
var = PIN(4)
```

This will read the value of pin 4 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(6) = 0
```

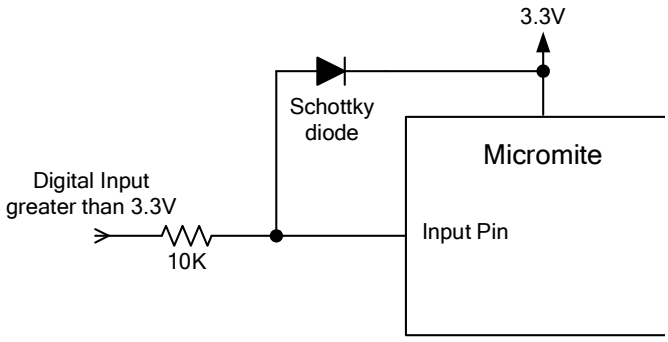
will set the output of pin 6 to zero (which generally means a logic low). In this case the PIN() construct is used as a command. This dual nature of the PIN() construct (either input or output) sometimes confuses newcomers to MMBasic so watch out for it.

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 Micromite'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 3.6V however some pins are rated for 5.5V (check the pinout diagram in the User Manual which lists the 5V tolerant pins). 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 Micromite's input impedance is very high (leakage is less than 1µA) you can use a large valued input resistor – for example, a 10K resistor which would be suitable for any input voltage up to 50V.

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

```
SETPIN 9, DIN
IF PIN(9) = 1 THEN PRINT "High" ELSE PRINT "Low"
```

The SETPIN command configures pin 9 as a digital input and the PIN() function will return the value of that pin (the number 1 if the pin is high). The IF command will then execute the command after the THEN statement if the input was high. If the input pin was low the program would execute the command after the ELSE.

Because the PIN() function will return 1 when the input is high and the IF … THEN command treats any non zero number in its conditional statement as true, you could rewrite the last line to read:

```
IF PIN(9) 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:

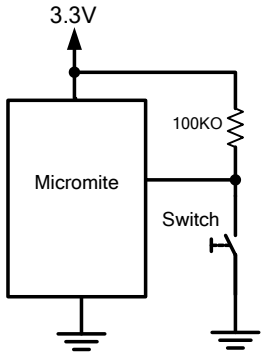
```
val = PORT(start, nbr)
```

'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 23, 24, 25 and 26 you would use this:

```
val = PORT(23, 4)
```

ie, read four consecutive input pins starting with pin 23. The *Micromite 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

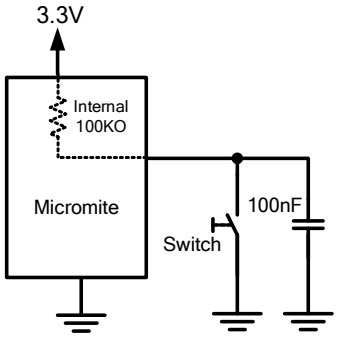


When you want to use a switch as an input you need a pull up resistor as shown on the right. The purpose of this resistor is to apply a voltage across the switch's contacts. Then, when the switch is closed the contacts will pull the input to zero and the PIN() function would return zero for closed and one for open.

Rather than using an external resistor the Micromite input can be specified with an internal pullup resistor. This resistor is internal to the Micromite 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.

To specify a pullup resistor you use SETPIN as follows:

```
SETPIN pin, DIN, PULLUP
```



Using either an internal or external pullup resistor you also need to consider the issue of contact bounce. This is when the switch contacts mechanically touch and then bounce apart momentarily due to the momentum of the mechanical assembly. Because the Micromite runs very fast a BASIC program could see this as a sequence of quick button presses rather than a single press.

You could check for this in your program, for example by checking 100ms after the first contact closure to confirm that the contacts are indeed closed. A simpler solution is to connect a 100nF capacitor across the switch contacts as illustrated. This capacitor in association with the pullup resistor will average out any rapid contact bounce so that the program will see a smooth transition from on to off and vice versa.

Digital Outputs

All I/O pins can be configured as a standard digital output. The command to do this is:

```
SETPIN pin, DOUT
```

This means that when an output pin is set to logic low it will pull its output to zero and when set high it will pull its output to 3.3V. In BASIC this is done with the PIN command. For example:

```
PIN(15) = 0
```

will set pin 15 to low, while

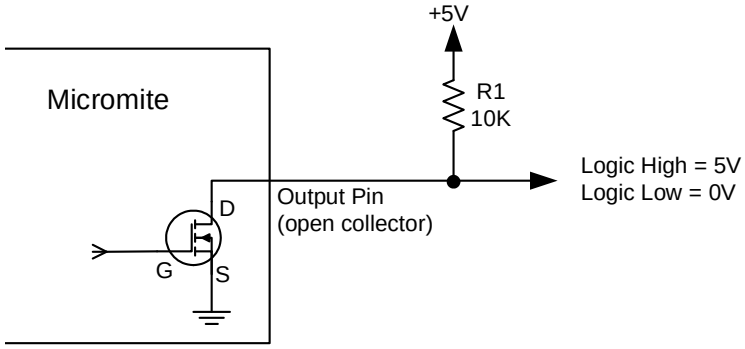
```
PIN(15) = 1
```

will set it high (in fact any non zero value can be used to set the output high).

When operating in this mode, a pin is capable of sourcing or sinking about 10mA which is sufficient to drive a LED or other logic circuits running at 3.3V.

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

The diagram shown below illustrates how an open collector system works. Note that the circuit should really be called open drain because the PIC32 uses a FET as the driver but open collector is a more common term and for consistency it is used here.



To set an output as open collector you use the SETPIN command in BASIC as you normally would but you append OC to the command to specify an open collector output. For example:

```
SETPIN pin, DOUT, OC
```

For driving high voltage and/or high current loads such as relays you should use a transistor (either bipolar or FET) to drive the load. To switch 240V AC a more elegant solution is to use a solid state relay. These have full isolation between their input and output and can switch 240V AC loads with a current of up to 10 amps. Some can be directly connected to a Micromite 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 Micromite and the circuit that you are driving.

There are also many fully assembled modules that you can use with the Micromite. For example, the module on the right provides four relays capable of switching 240V AC. It costs about $8 and each relay can be controlled by a Micromite output with no additional circuitry required (for this particular module search the internet for “Geekcreit Relay Module”).

As with the PIN() construct you can also use PORT() as a method of setting a number of outputs simultaneously to some state. For example, the following will simultaneously set pins 5, 6 and 7 to the low logic state (ie, zero volts):

```
PORT(5, 3) = 0
```

Sometimes you need to generate a pulse on an output pin. This can be conveniently done with the PULSE command:

```
PULSE pin, mSec
```

Where 'pin' is the pin number and 'mSec' is the desired pulse width in milliseconds. This last parameter can be a fraction (ie, 0.1ms) so very short pulses down to a few microseconds can be generated. The polarity of the pulse is opposite to the current state of the pin. For example, if the output of the pin is currently low the pulse will be positive. Pulses can be up to many days in length and any pulse longer that 3ms will be run in the background – this means that the program

will continue with the following commands and MMBasic will automatically terminate the pulse when its time is up.

Analog Input

The 28-pin Micromite has ten I/O pins that are capable of voltage measurement and the 44-pin Micromite has thirteen. They are marked as ANALOG on the pin diagrams for the Micromite. To set an I/O pin to analog you use the command:

```
SETPIN nn, AIN
```

Where AIN stands for Analog IN and 'nn' is the pin number that you want to configure.

The analog input range is from zero to whatever the Micromite's supply voltage is and the PIN() function will return the input reading in volts. For example, if you connected a 1.5V battery to pin 4 and run the following program you could expect to see a value of about 1.5:

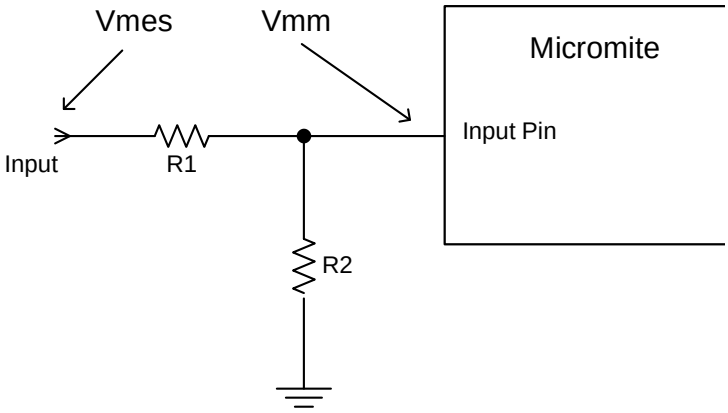
```
SETPIN 4, AIN
PRINT PIN(4)
```

Note that the Micromite assumes that the supply voltage is exactly 3.3V and it uses that number as its reference. If your supply is not exactly 3.3V you can scale the reading by using a digital meter to measure the supply voltage and use that value to correct the reading. For example, if you measured the Micromite's supply voltage as 3.1V the following expression will return the correct reading on an analog input:

```
PRINT (PIN(4) / 3.3) * 3.1
```

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 Micromite on its input pin (Vmm) and the voltage at the input of the voltage divider (Vmes) using a digital multimeter.

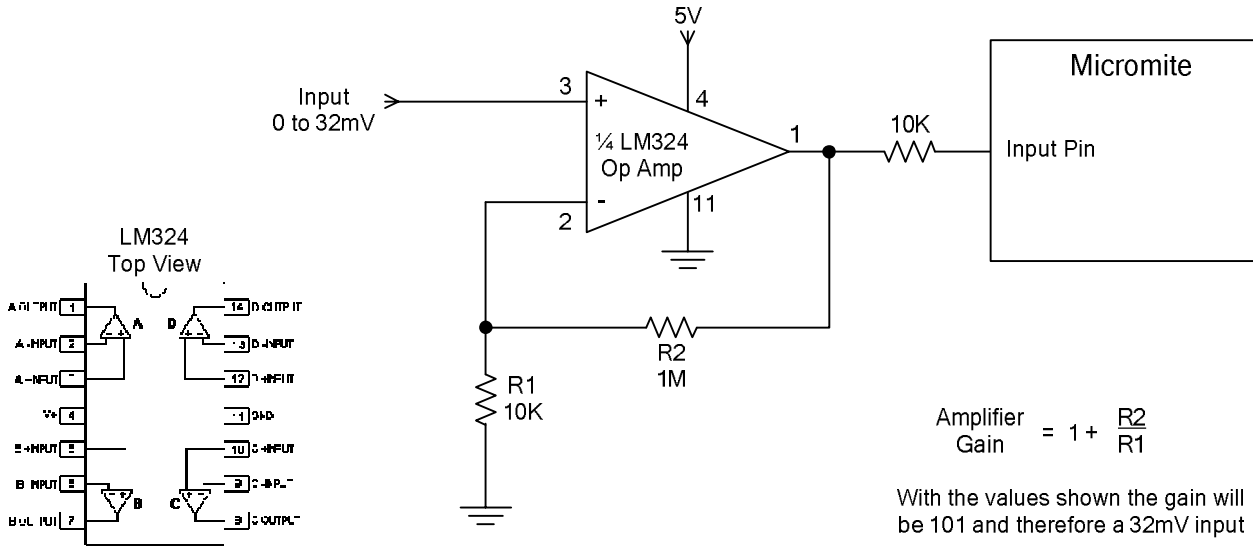


Then the reading could be scaled thus:

```
PRINT PIN(nn) / (Vmm / Vmes)
```

Note that to retain the accuracy of the reading the source resistance needs to be 10K or less. This means that in most circuits the value of R2 should be 10K or less.

For small voltages you will need an amplifier to bring the input voltage into a reasonable range for measurement. The diagram below shows a typical arrangement using the popular and inexpensive LM324 quad operational amplifier. The LM324 can operate from a single 5V supply and contains four identical amplifiers in the one 14 pin package.



The gain of the amplifier is determined by the ratio of R2 to R1 plus 1 and using the components shown the gain is 101. This number should be used in the BASIC program so that the readings are scaled to represent the input voltage.

For example:

```
PRINT PIN(9) / 101
```

Alternatively, you could adopt the technique used to scale the reading for a voltage divider (as described on the previous page). The result will be the same.

Power Supply Voltage

As mentioned above, you need to determine the Micromite's supply voltage to obtain an accurate voltage measurement. This is not a problem when you are using a fixed voltage regulator but when the Micromite is battery powered the supply voltage can vary over a wide range. In this case a good method of determining the power supply voltage is to use a voltage reference chip. This will generate a precise and known voltage and from that the battery voltage can be inferred.

The Texas Instruments REF3020 is a typical voltage reference chip that is good for this purpose. It comes in a simple three pin package and it will provide a precise 2.048V output. Its output should be connected to a spare analog input on the Micromite (say pin 5) and then the power supply voltage derived using the following expression:

```
SETPIN 5, AIN
PwrVolts = (PIN(5) / 2.048) * 3.3
```

This assumes that you are using a reference with a 2.048V output. If yours is different you will need to substitute its output voltage into the expression.

Once you have determined the Micromite's supply voltage you can then scale other voltage measurements as described previously. For example:

```
SETPIN 4, AIN : SETPIN 5, AIN
PwrVolts = (PIN(5) / 2.048) * 3.3
PRINT (PIN(4) / 3.3) * PwrVolts
```

Knowing the batteries' voltage is also useful because you can use that value in an IF statement to raise an alarm when the battery is almost exhausted.

Frequency and Period Measurement

Four pins on the Micromite 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 Micromite can count pulses as narrow as 10nS (although the maximum frequency of the pulse stream is still limited to about 200KHz).

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 *Micromite User Manual* if you wish to use it.

Interrupts

Interrupts are a handy way of dealing with an event that can occur at an unpredictable time. An example is when the user presses a button. In your program you could insert code at critical locations to check to see if the button has been pressed but an interrupt makes for a more cleaner and readable program.

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

An interrupt is setup using the SETPIN command:

```
SETPIN pin, type, subroutine
```

'pin' is the pin number which will trigger the interrupt, 'type' is the type of interrupt and can be INTH for a rising edge signal transition, INTL for falling edge transition or INTB for any change in the input (ie, interrupt on both rising and falling). 'subroutine' is the subroutine to execute when the interrupt trigger occurs – this is just an ordinary subroutine (nothing special).

You can set an interrupt on any I/O pin and you can have up to ten I/O pins simultaneously operating as interrupts, each with its own interrupt subroutine or, if you wish, sharing one or more subroutines. If two interrupts occur simultaneously MMBasic will execute the subroutine associated with the interrupt that was defined first, then when it has finished (and the next interrupt condition still exists) it will execute the next interrupt subroutine, and so on.

While MMBasic is executing the interrupt subroutine all other interrupts are ignored. This means that if your interrupt code takes too long to execute there is a chance that another interrupt (such as a button push) might arise and vanish while your first interrupt subroutine is still executing – with the result that the new interrupt would be missed. For this reason interrupt subroutines should be as short as possible.

As an example of defining an interrupt the following code fragment will detect if the user has pressed a button (connected to pin 16) and, if so, will set the output of pin 15 high (this could operate a relay or something similar).

```
SETPIN 15, DOUT
SETPIN 16, INTL, MyInt
DO
   ' main processing loop
   ' more processing
LOOP

' interrupt routine
SUB MyInt
  PIN(15) = 1
END SUB
```

In the first line of the fragment we configure pin 15 as an output (this will drive our relay or whatever) and in the second line we configure pin 16 to be a digital input that will generate an interrupt on the high to low transition. The interrupt code is held in the subroutine `MyInt` and this is specified as the third parameter to the SETPIN command.

The DO…LOOP represents the main processing loop which runs forever. When the user presses the button connected to pin 16 the voltage on that pin will drop to zero, MMBasic will recognise this as a high to low transition and automatically interrupt the main program and execute the subroutine `MyInt`. This routine is very short; it just sets the output high and exits the subroutine which then allows the main program to continue as before. The main processing loop is completely oblivious to the interrupt. Normally an interrupt subroutine will have more than a single line in it but it does not have to be complicated – it should just do its job then exit.

Many other parts of MMBasic can also generate interrupts. For example, you can specify an interrupt that repeats with a specified number of milliseconds between each interrupt (the SETTICK command), you can have an interrupt when an IR remote control signal is received or when a certain number of bytes has been received on a serial interface.

Normally MMBasic will respond to a single interrupt within 50µs so you can use interrupts to catch reasonably fast events. For example, ignition pulses in a petrol engine.

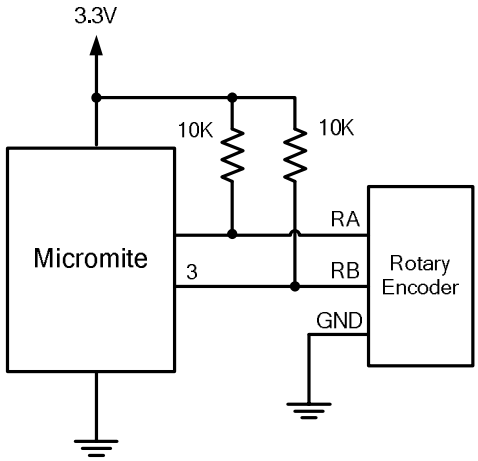
When using interrupts keep in mind the following:

- 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, using an ultrasonic distance sensor, etc) so care needs to be taken if you are using any of these functions and interrupts at the same time.
- Remember the commandment "Thou shalt not hang around in an interrupt". For example, never use PAUSE inside an interrupt. If you have some lengthy processing to do in an interrupt you should simply set a flag and immediately exit the interrupt. Then, in your main program loop, you can detect the flag and do whatever is required then reset the flag. Always keep interrupts short and exit as soon as possible otherwise you run the risk of missing other interrupts or tying up MMBasic by continuously executing interrupts..
- The subroutine that the interrupt calls (and any other subroutines called by it) should always be exclusive to the interrupt. If you must call a subroutine that is also used by an interrupt you must disable the interrupt first to prevent it from calling the subroutine while you are executing it (this would have undefined consequences including ruining your day). You can then reinstate the interrupt after you have finished with the subroutine.
- Remember to disable an interrupt when you have finished needing it – background interrupts can cause strange and confusing bugs.

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 Micromite 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 2, 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 3; however any pins can be used by changing the pin numbers in the program. Also, it is intended for simple user input where a skipped or duplicated step is not considered important. It is not suitable for high speed or precision input.

Program courtesy MicroBlocks on the Back Shed Forum.

PWM and Servo Outputs

Five I/O pins can generate PWM or servo signals. PWM stands for Pulse Width Modulation which is a constant square wave output with a specified duty cycle and frequency. By varying the duty cycle (the ratio between the positive pulse and the negative pulse) your program can generate a synthesised voltage which can be used to control devices such as motor controllers which need an analog input. It can also be used to control the brightness of LEDs or incandescent lamps (read more about this technique at: http://learn.sparkfun.com/tutorials/pulse-width-modulation).

Another use for the PWM outputs is to generate a signal which, with a small loudspeaker, can create a range of audible tones.

The PWM outputs on the Micromite 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 Micromite the outputs for the first channel are labelled 1A, 1B and 1C while the two outputs for the second are 2A and 2B.

The syntax of the PWM command is:

```
PWM ch, freq, A-DutyCycle, B-DutyCycle, C-DutyCycle
```

'ch' is the channel number (1 or 2), 'freq' is the frequency (20Hz to 500kHz) and the remaining three parameters are the duty cycle for each of the outputs (0 to 100%). If you do not want to use an output you can leave that output off the end of the list and that pin can be used for some other purpose. After this command has been executed the output will run continuously unless changed or the Micromite is reset.

For example, the following will set the PWM 1A output to 1KHz with a duty cycle of 20% and 1B to a duty cycle of 60% at the same frequency. The 1C output is not specified so the pin allocated to 1C will not be affected and can be used for some other purpose:

```
PWM 1, 1000, 20, 60
```

This command can be used repeatedly to change the duty cycle (and frequency if required) of the PWM outputs at will.

The Micromite 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 Micromite can simultaneously control up to five servos.

Depending on their size servos can be mechanically quite powerful and provide a convenient way for the Micromite 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 Micromite 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 Micromite 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 Micromite provides special support for these. Using this built in support you can easily add features such as an infra red remote control or keypad input 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
- LCD display modules
- Numeric keypads
- Battery backed clock
- Ultrasonic distance sensor

Rather than go into the detail of each device in this tutorial you can check the *Micromite User Manual*. The section "Special Device Support" provides a good description of each along with examples of their use.

Chapter 6

Embedded Features

The Micromite is primarily designed as an embedded controller. This is a computing module contained inside some product or device and not necessarily exposed to the user. In this role features such as timing, power consumption and recovering from errors are important.

The Micromite's support for embedded controller applications includes:

1. Tracking the date/time and internal timers that count milliseconds.
2. Adjusting the speed and power consumption of the chip.
3. Putting the chip to sleep.
4. Recovering from errors.
5. Saving important data to be recovered on power up.

Keeping Time

In the Micromite 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 Micromite's internal clock is reset to midnight 1st January 2010 on power up but you can set the time and date to something else by assigning a string to these 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
```

You can also use the command RTC GETTIME to set the correct time from an external Real Time Clock (RTC) chip. These are battery backed and include a crystal based clock which will keep accurate time even when the power is removed - check the *Micromite Users Manual* for the details.

TIMER is another special identifier which returns the number of milliseconds since being reset to zero (it is also reset when the Micromite is powered up). You can use it to measure the time difference between two events as shown in the following example:

```
TIMER = 0
  ' section of code that needs to be timed
PRINT TIMER "ms"
```

In a large program resetting the timer can get confusing as you might reset it in several places and cause a conflict. An alternative is to save the current value of the timer and use that value without resetting the timer. For example:

```
TIMERCOUNT% = TIMER
  ' section of code that needs to be timed
PRINT TIMER - TIMERCOUNT% "ms"
```

Note that the timer is an integer which is why we defined `TIMERCOUNT%` as an integer (the % suffix). Another factor is that because integers can store very large numbers the value of TIMER will only roll over to zero after millions of years of incrementing, so you do not have to worry about its value resetting while your Micromite is powered up.

The TIMER function can also be used to wait for a certain length of time but a better method is to use the PAUSE command which will halt the execution of a program for a precise number of milliseconds.

For example, to create a 12ms wide pulse you could use the following:

```
SETPIN 4, DOUT
PIN(4) = 1
PAUSE 12
PIN(4) = 0
```

Sometimes, after setting a control signal for a device, you might be required to wait for a defined number of milliseconds before you can set the next control signal. The PAUSE command is perfect for this type of job and many similar jobs that require a delay.

MMBasic also allows you to set up to four "tick" timers. Each acts like the tick of a clock and on each tick MMBasic will execute an interrupt subroutine specified in the command. Up to four "tick" interrupts can be setup. The tick times are specified in milliseconds and can range from a few milliseconds to many days. Think of it as the regular "tick" of a watch.

For example, the following code fragment will print the current time and the voltage on pin 2 every second. This process will run independently of the main program which could be doing something completely unrelated.

```
SETPIN 2, AIN
SETTICK 1000, DoIint
DO
   ' main processing loop
LOOP

SUB DoIint          ' tick interrupt
  PRINT TIME$, PIN(2)
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.

CPU Speed and Power Consumption

Controlling the power consumption of the chip is important if the device is to be battery powered. In the Micromite this can be done via the CPU command which is used to control the processor's speed and therefore the Micromite's power consumption.

For example, `CPU 5` will set the processor's clock speed to 5MHz with a power requirement of about 6mA and `CPU 48` will set it to 48MHz which causes the Micromite to draw about 31mA. The BASIC program can change this as often as it likes so it is possible to speed up the processor to execute a few lines of code and then revert back to a slow speed for the non important portions of the program. Halving the clock speed roughly halves the power drain so this is a useful technique for battery powered devices.

Sleeping

In some cases you would prefer that the Micromite shut down completely to conserve power. In this mode the current drawn by the Micromite can be as low as 40µA.

The sleep period can be for a number of seconds or indefinitely with the "wakeup" from sleep triggered by an external signal. Both modes are initiated by the CPU SLEEP command.

Normal use for the command is:

```
CPU SLEEP seconds
```

Where 'seconds' is the required sleep time in seconds which can range from one second to days.

If you are waiting on some external event the program could set the sleep command to terminate early on some input. This is done by specifying an I/O pin in the sleep command:

```
CPU SLEEP seconds, pin
```

Then, any change in the state of ‘pin’ (ie, low to high or vice versa) will terminate the sleep. For example this will cause the Micromite to sleep and wakeup after 10 minutes or whenever pin 5 changes state (whichever comes first):

```
CPU SLEEP 600, 5
```

Due to the way that the Micromite’s hardware works the delay between the change of state in the I/O pin and exiting from sleep can be as great as one second. This means that any change in the state of the wakeup pin must remain that way for at least a second to guarantee that the Micromite will come out of the sleep.

If you wish the Micromite to wake instantly on an external signal (within a few microseconds) you can use the CPU SLEEP command without specifying the time. For example:

```
CPU SLEEP
```

In this case MMBasic will automatically configure the WAKEUP pin as a digital input (pin 16 on the 28-pin Micromite or pin 43 on the 44-pin chip). The Micromite will then sleep for an indefinite amount of time until a change of state (ie, high to low or vice versa) on the WAKEUP pin which will instantly terminate the sleep.

Autorun

As an embedded controller the power can be interrupted at any time and when it is reapplied you want the program to automatically start running. This is achieved by setting AUTORUN on:

```
OPTION AUTORUN ON
```

Then, when the power is cycled the Micromite will automatically run the program in memory. This command can be entered at the command prompt or used in the program and will be remembered even after a power loss and restart.

Recovering From Errors

When the Micromite is used in an embedded context it will appear to the user as a custom integrated circuit performing some special task. The user need not know anything about what is running inside the chip. However there is always the possibility that something could cause MMBasic to generate an error and return to the command prompt. This would be of little use in an embedded situation as the Micromite would not have anything connected to the console. 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 Micromite would stop doing its programmed job until the power was cycled. To handle this possibility MMBasic has two mechanisms for dealing with errors; the watchdog timer and turning off error checking.

The watchdog timer is a timer that counts down to zero and when it reaches zero the processor will be automatically restarted (the same as when power was first applied). This applies even if MMBasic is sitting at the command prompt. The WATCHDOG command specifies how many milliseconds are allowed before the reset. For example, the following will set the watchdog timer to 200 milliseconds:

```
WATCHDOG 200
```

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

The program can check if it has been restarted by the watchdog timer by examining the value of the built-in variable MM.WATCHDOG. This is set to true if the restart was forced by the watchdog timer and false if it was a normal (eg, power on) startup.

The watchdog timer is foolproof but rather crude. Another method of handling errors with more finesse is to use the ON ERROR command which allows you to trap and respond to any errors line by line in your program. For example, your program might use the RTC GETTIME command to get the time from an external real time clock (RTC) chip and set the Micromite's internal clock. But what if the external RTC is missing or faulty? This would cause MMBasic to generate an error, halt the program, return to the command prompt and wait forever for some input.

The watchdog timer will not help here because on reboot your program would try to get the time again, resulting in another error, another reboot and so on. The solution is to use the ON ERROR SKIP command which will skip or ignore any errors generated by the next command in the program. For example:

```
…
ON ERROR SKIP
RTC GETTIME
…
```

This will cause MMBasic to ignore any error in the RTC GETTIME command and carry on as if nothing has happened (as a side effect the Micromite's internal clock 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 turn on an error LED. 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, if the error LED is connected to pin 12 the following program fragment will turn it on if the RTC is not present or is faulty:

```
…
ON ERROR SKIP
RTC GETTIME
IF MM.ERRNO <> 0 THEN PIN(12) = 1
…
```

Sometimes it is possible that a group of commands can generate an error and in that case you can specify how many commands to skip the error checking by using ON ERROR SKIP nn where *nn* is the number of commands. There is also the command ON ERROR IGNORE which will completely ignore all errors in all commands until the command ON ERROR ABORT is encountered. This last command will restore the normal behaviour if an error occurs (ie, display an error message on the console and stop the program).

You need to be careful when skipping errors as this will cause MMBasic to ignore **all** errors including spelling mistakes, invalid commands, typos, etc. It is helpful having MMBasic point out these sorts of errors and it can be difficult to figure out why your program is not running correctly if the error reporting is turned off. For this reason you should fully test your program before adding code to skip errors and even then, it should only be used in specific cases which cannot be handled in any other way.

Saving Data

Because the Micromite does not usually have a normal storage system (such as an SD card) it needs to have a facility to save some data that can be recovered when power is restored. This might be calibration data, user options, current state, etc.

This can be done with the VAR SAVE command which will save the variables listed on its command line in non volatile flash memory. A typical use is like this:

```
VAR SAVE ConfigX, ConfigY
```

On power up these variables can be restored with the VAR RESTORE command which will add all the saved variables to the variable table of the running program. Normally this command is placed near the start of a program so that the variables are ready for use by the program.

This short program provides an example, it is not very practical but it does illustrate how the VAR SAVE feature can be used:

```
VAR RESTORE
IF Config1 = 0 AND Config2 = 0 THEN
  INPUT "Config data 1", Config1
  INPUT "Config data 2", Config2
  VAR SAVE Config1, Config2
ELSE
  PRINT "Restored configuration data"
ENDIF
…
```

The VAR RESTORE command at the start of the program will try to restore any (and all) saved variables. If none have been saved the command will do nothing.

The program will then check if the variables Config1 or Config2 are set to a non zero number indicating that they have been previously set and saved via VAR SAVE (and therefore the VAR RESTORE command found them and restored them). If Config1 or Config2 are both zero the program will then get the settings from the user and save them ready for the next restart.

Note that the very first time that the program is run there will be nothing to restore (because nothing has been saved) but that does not matter, the command will not generate an error.

Saved variables are written to the chip's flash memory which has an endurance specification of 20,000 erase/write cycles. That is fine for normal use - for example, if on average you used the VAR SAVE command once every day of every year the flash memory would not reach its endurance specification for 50 or more years. However, if you saved the data more frequently you might run the risk of wearing out the flash memory in months or even days. So, the VAR SAVE command should only be for data that changes infrequently.

If you do need to save data more frequently you could add a battery backed real time clock chip (RTC) to your project. Many of these have a bank of memory locations that can be used for saving data with no endurance limitations at all (see the RTC SETREG and RTC GETREG commands).

Another option is to add a Microchip 11XX series EEPROM chip to your project. These are low cost, have an extremely high endurance (1 million cycles) and are available in a range of capacities up to 2 Kbytes. They come in an easy to use TO-92 package (a thru hole package that looks like a standard transistor) with only one pin used for communicating with the Micromite. The Micromite firmware zip file includes a CFunction (explained below) that makes using these easy.

Embedded C Routines

As mentioned in Chapter 3, the BASIC language does have one drawback which is that it is not as fast as a compiled C program. However, by using embedded C routines you can have the best of both worlds - ie, speed (when you need it) and the ease of programming in BASIC.

The Micromite firmware download contains a folder called "Embedded C Modules" which contains many embedded C routines including drivers for various display panels, additional communications ports, support for devices like a humidity sensor, etc. You can also write your own embedded C routines but that does require you to be experienced in programming with the C language (not an easy subject to learn) so beginners would be better off just using the pre compiled routines.

MMBasic running on the Micromite supports two types of embedded modules. These are a CSub which is a subroutine (ie, it is used as a command) and a CFunction which is a function (ie, it can be used in expressions and returns a value). To use an embedded C routine you must insert the definition of the routine somewhere in your BASIC program and having done that you can refer to the CSub or CFunction as a normal subroutine or function.

For example, the following Cfunction will return the current CPU speed of the Micromite in hertz:

```
CFunction CPUSpeed
 00000000 3c02bf81 8c45f000 8c43f000 3c02003d 24420900 7ca51400 70a23002
 3c040393 34848700 7c6316c0 00c41021 00621007 3c03029f 24636300 10430005
 00402021 00002821 00801021 03e00008 00a01821 3c0402dc 34846c00 00002821
 00801021 03e00008 00a01821
End CFunction
```

You can use it just like any other command:

```
> PRINT CPUSpeed()
 40000000
```

An embedded routine in MMBasic must start with the keyword "CSub" or "CFunction" followed by its name and a sequence of 8-digit hex words which are the compiled C code. Each word must be separated by one or more spaces or new lines. It is terminated by an "End CSub" or "End CFunction" keyword. Multiple CFunction and CSub commands can be used and during execution MMBasic will skip over them. This means they can be placed anywhere in the program.

When a BASIC program is saved to flash MMBasic will search through it looking for any CFunction or CSub commands and if any are found the machine code specified will be extracted and programmed into a separate part of the flash memory. This code then becomes part of MMBasic and will be used whenever the specified CFunction or CSub name is encountered.

Chapter 7

Communications Protocols

An outstanding feature of the Micromite is the range of communications protocols that it supports. These are integrated into the BASIC language and are easy to use so you can conveniently transfer data back and forth with test equipment, other ICs or sensors.

The list of communications 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 other 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 *Micromite User Manual* for the full details.

Asynchronous Serial Communications

There are many types of serial communications but in this context we will cover what is known as asynchronous serial communications. This is 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. Other forms of serial communications (notably SPI) have a separate clock signal which the receiver can use to determine when it should be receiving a bit of data.

For a more detailed description see: https://learn.sparkfun.com/tutorials/serial-communication

The 28 and 44-pin Micromites have two asynchronous serial ports, the first can operate up to 282,000 baud while the second will work up to 19,200 baud. The first port is particularly flexible and can use TTL signalling levels (ie, 0 to 3.3V) or work with devices that use RS232 levels (±12V). It can also drive RS485 circuits and generate the required EN signal.

The Micromite Plus (the 64 and 100-pin chips) can have up to four asynchronous serial ports, all of which can operate at speeds up to 1,000,000 baud and support RS232, RS485, etc.

TTL signalling levels 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).

RS232 is slightly different in that the signal is inverted and swings from -12V to +12V. In the past an RS232 port was standard on desktop computers but in recent years it is rarely used except in some test equipment.

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 Micromite serial ports default to 9600 baud but you can change this by appending the required speed to the end of the COM port specification when you open the port. For example this will open the second serial port at 1200 baud and assign it the reference number #4:

```
OPEN "COM2:1200" as #4
```

To send something out of the serial port you use the PRINT command. For example:

```
PRINT #4, "Hello"
```

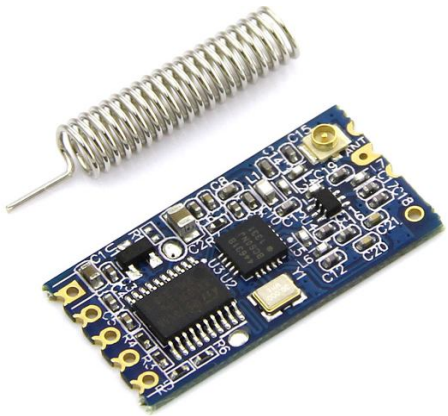
This will send a series of characters spelling "Hello" out of the serial port opened as #4. To receive characters from a serial port you can use a number of commands or functions but the most useful is the INPUT$(x, #n) function which will retrieve x characters from the port opened as #n.

The list of commands and functions in MMBasic that will accept a serial port reference number are:

PRINT	Send a string
INPUT$()	Receive one or more characters
LINE INPUT	Receive a complete line
EOF()	True if no characters are waiting in the receive buffer
LOF()	The empty space (in characters) remaining in the transmit buffer

All serial communications in the Micromite 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 the Micromite. This can sometimes trip up newcomers who (for example) might try to read the response from the other device while there is still data in the output buffer waiting to being sent to that device.

A practical example of using serial communications is sending data via wireless modules that use serial as their interface. A typical example is the HC-12 (about US$6 on eBay) as shown on the right. It defaults to a speed of 9600 baud and anything sent to the transmitting module will be received by the receiving module at the same speed.



For example, if you wished to measure a temperature with one Micromite and transmit that wirelessly to a second using a pair of HC-12 modules, your program on the sending Micromite 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 Micromite 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.

Collecting Data from a GPS Module

Another common use for a serial port is interfacing with low cost GPS modules. It is easy to get your current speed, heading, location, time, etc with amazing accuracy from such a module.

These modules output their data as a sequence of lines of text at a particular baud rate. Each line provides a specific type of data and normally the sequence of lines will repeat every second. The format of the data is defined by the NMEA 0183 standard and it is worth looking that up along with your GPS module's data sheet to understand all the information that is provided.

Our example will extract the speed and heading data which is held in the line that starts with the letters $GPRMC. This is called the RMC record and it typically looks like this:

```
$GPRMC,231719.000,A,3411.5204,S,14135.6619,E,9.62,302.03,150216,,,A*75
```

The data is separated into "fields" by commas and we need the 8th and 9th fields (speed and heading). Another useful field is the 3rd field which contains the single letter A if the module has a lock on to a sufficient number of GPS satellites to deliver accurate information.

Our program will rely on an interrupt to detect when each character has been received so that the main program can be doing other work while waiting for the data. This is the program:

```
Dim String dat
Open "COM1:9600, 256, SerInt, 1" As #1

Do
  If dat <> "" Then
    'if we have reached here we have received a full line
    If Left$(dat, 6) = "$GPRMC" And GetF(dat, 3) = "A" Then
      Print "Speed: " GetF(dat, 8), "Heading: " GetF(dat, 9)
    EndIf
    dat = ""                        ' get ready for a new line
  EndIf
Loop

' interrupt subroutine which is called when a character is received
Sub SerInt
  Static STRING tmps              ' this is used to hold a line
  Local STRING ch
  ch = Input$(1, #1)
  If ch >= " " And ch <= "Z" Then ' if the character is printable
    tmps = tmps + ch    ' add it to our string
  Else If ch = Chr$(13) Then      ' if it is a carriage return
    dat = tmps
    tmps = ""
  End If
End Sub
```

```
' function to get a specific field from a comma delimited string
Function GetF(str As STRING, field As INTEGER) As STRING
  Local INTEGER i, lastp = 1, endp
  For i = 1 To field - 1
    lastp = Instr(lastp, str, ",")
    If lastp = 0 Then Exit Function Else lastp = lastp + 1
  Next i
  endp =  Instr(lastp, str, ",")
  If endp = 0 Then endp = Len(str) + 1
  GetF = Mid$(str, lastp, endp - lastp)
End Function
```

This is another example of where you will have to step through the code line by line to figure out how it works.

It starts by opening the COM port and setting the baud rate (9600), buffer size (256) and the subroutine to call when a character has been received. The program then drops into an endless loop where the string variable `dat` is checked. This is set to the text string received from the GPS by the serial interrupt subroutine - the program knows when something has been received because it will suddenly change from an empty string to something else.

We then check if the line starts with $GPRMC and that the 3rd field contains the letter A indicating an accurate set of data has been received. If that is OK we print the result on the console using the function `GetF()` to extract each field from the string (`GetF()` is described below). Finally we reset `dat` to an empty string ready for the next line.

The subroutine `SerInt` is called by MMBasic every time a character is received and its job is straight forward. If the character is printable it is added to the end of the string `dat` which grows longer with each received character. If the character is a carriage return we know that we have received a full line so we simply copy the string into the global variable `dat` for checking by the main program loop. This keeps the interrupt as short as possible and leaves the more time consuming job of processing the text to the main program loop… remember the commandment from Chapter 5 "Thou shalt not hang around in an interrupt".

The function `GetF` is used to extract a specific field from the received data string. It steps through the string looking for the comma marking the start of the field that we want. It then extracts the characters up to the following comma and sets that as the returned value of the function. To guard against corrupt data the function will return an empty string (rather than causing an error and stopping the program) if the specified field cannot be found.

This function has been deliberately written so that it can be used as a drop-in-module in other programs where you need to extract comma separated data (for example, spreadsheets often will export data as comma separated values). For a more detailed description of this format see: https://en.wikipedia.org/wiki/Comma-separated_values

I²C Communications

Most sensors use either the I²C or SPI protocols to communicate their results and the Micromite 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 on the Micromite is 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 I^2C 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 I^2C device has an address which allows multiple devices to share the one set of input/output pins (ie, channel). 'addr' is the address, 'option' is a specialised setting which is normally set to zero, 'len' is the amount of data to send or receive and 'data' is a variable or constant when sending data or a variable where the received data is to be saved.

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

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

The hours and minutes maintained by the PCF8563 are held in two consecutive registers which we need to read (check the PCF8563 data sheet for the details). First the program defines an array to hold the received data. We then open the I^2C channel and write the number of the first register that we want to read to the chip (register 3). The fourth line reads two bytes from the chip (minutes and seconds) and saves them in the previously defined array, `RData()`.

The PCF8563 real time clock is hardwired to recognise the address 51 (hex) on the I^2C 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.

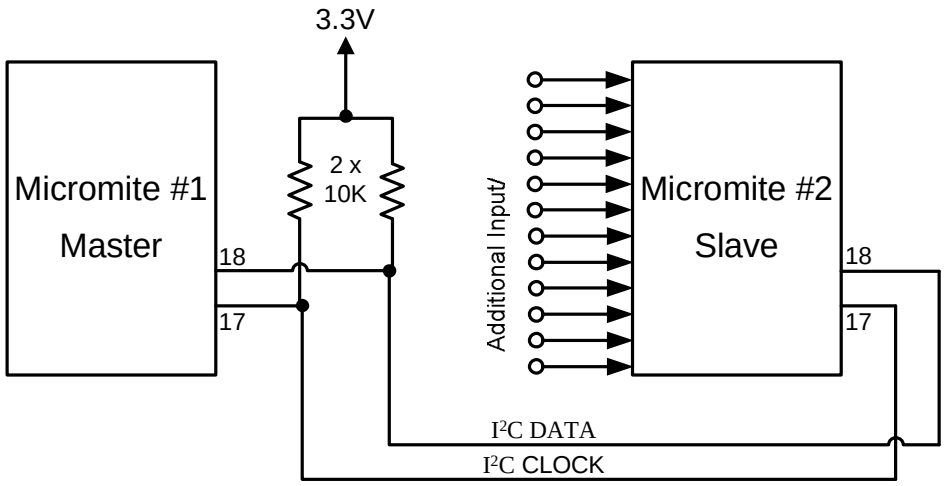
For more details on the I^2C protocol see: http://learn.sparkfun.com/tutorials/i2c

A More Complex I^2C Example

I^2C is ideally suited for communications between integrated circuits. As an example, there might be an occasion when a single Micromite does not have enough serial ports, I/O pins, or whatever for a particular application. In that case a Micromite could be used as a slave to provide the extra facilities.

This example converts a Micromite into a general purpose I/O expansion chip with 17 I/O pins that can be dynamically configured (by the master) as analog inputs or digital input/outputs. The routines on the master are simple to use (SSETPIN to configure the slave I/O and SPIN() to control it) and the program running on the master need not know that the physical I/O pins reside on another chip. All communications are done via I^2C.

The following illustration shows the connections required for 28-pin chips:



Program Running On the Slave:

The slave must first set up its I^{2}C interface to respond to requests from the master. With that done it can then drop into an infinite loop while the job of responding to the master is handled by the I^{2}C interrupts.

In the program below the slave will listen on I^{2}C address 26 (hex) for a three byte command from the master. The format of this message is:

- Byte 1 is the command type. It can have one of three values; 1 means configure the pin, 2 means set the output of the pin and 3 means read the input of the pin.
- Byte 2 is the pin number to operate on.
- Byte 3 is the configuration number (if the command byte is 1), the output of the pin (if the command byte is 2) or a dummy number (if the command byte is 3).

The configuration number used when configuring a slave's I/O pin is the same as used in earlier versions of Micromite MMBasic (with the SETPIN command) and can be any one of:

0	Not configured or inactive
1	Analog input
2	Digital input
3	Frequency input
4	Period input
5	Counting input
8	Digital output
9	Open collector digital output. In this mode SPIN() will also return the value on the output pin .

Following a command from the master that requests an input, the master must then issue a second I^{2}C command to read 12 bytes. The slave will respond by sending the value as a 12 character string.

This program can fall over if the master issues an incorrect command. For example, by trying to read from a pin that is not an input. If that occurs, an error will be generated and MMBasic will exit to the command prompt. Rather than trap all the possible errors that the master can make, this program uses the watchdog timer. If an error does occur the watchdog timer will simply reboot the Micromite and the program will restart (because AUTORUN is on) and wait for the next message from the master. The master can tell that something was wrong because it would get a timeout.

This is the complete program running on the slave:

```
OPTION AUTORUN ON
DIM msg(2)                          ' array used to hold the message
I2C SLAVE OPEN &H26, 0, 0, WriteD, ReadD     ' slave's address is 26
(hex)

DO                                  ' the program loops forever
  WATCHDOG 1000                     ' this will recover from errors
LOOP

SUB ReadD   ' received a message
  I2C SLAVE READ 3, msg(), recvd  ' get the message into the array
  IF msg(0) = 1 THEN    ' command = 1
    SETPIN msg(1), msg(2)           ' configure the I/O pin
  ELSEIF msg(0) = 2 THEN            ' command = 2
    PIN(msg(1)) = msg(2)            ' set the I/O pin's output
  ELSE ' the command must be 3
    s$ = str$(pin(msg(1))) + Space$(12) ' get the input on the I/O
pin
  ENDIF
END SUB                             ' return from the interrupt

SUB WriteD  ' request from the master
  I2C SLAVE WRITE 12, s$            ' send the last measurement
END SUB                             ' return from the interrupt
```

Interface Routines on the Master:

These routines can be run on another Micromite or some other computer with an I^2C interface. They assume that the slave Micromite is listening on I^2C address 26 (hex).

If necessary these can be modified to access multiple Micromites (with different addresses), all acting as expansion chips and providing an almost unlimited expansion capability.

There are two subroutines and one function that together are used to control the slave:

SSETPIN pin, cfg	This subroutine will setup an I/O pin on the slave. It operates the same as the MMBasic SETPIN command and the possible values for 'cfg' are listed above.
SPIN pin, output	This subroutine will set the output of the slave's pin to 'output' (ie, high or low).
nn = SPIN(pin)	This function will return the value of the input on the slave's I/O pin.

For example, to display the voltage on pin 3 of the slave you would use:

```
SSETPIN 3, 1
PRINT SPIN(3)
```

As another example, to flash a LED connected to pin 15 of the slave you would use:

```
SSETPIN 15, 8
SPIN 15, 1
PAUSE 300
SPIN 15, 0
```

These are the three routines:

```
' configure an I/O pin on the slave
SUB SSETPIN pinnbr, cfg
  I2C OPEN 100, 1000
  I2C WRITE &H26, 0, 3, 1, pinnbr, cfg
  IF MM.I2C THEN ERROR "Slave did not respond"
  I2C CLOSE
END SUB

' set the output of an I/O pin on the slave
SUB SETPIN pinnbr, dat
  I2C OPEN 100, 1000
  I2C WRITE &H26, 0, 3, 2, pinnbr, dat
  IF MM.I2C THEN ERROR "Slave did not respond"
  I2C CLOSE
END SUB

' get the input of an I/O pin on the slave
FUNCTION SPIN(pinnbr)
  LOCAL t$
  I2C OPEN 100, 1000
  I2C WRITE &H26, 0, 3, 3, pinnbr, 0
  I2C READ &H26, 0, 12, t$
  IF MM.I2C THEN ERROR "Slave did not respond"
  I2C CLOSE
  SPin = VAL(t$)
END FUNCTION
```

SPI Communications

The SPI protocol is simpler than I^2C and is also used by many sensors. The Micromite can drive the SPI interface at up to 10MHz 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 the Micromite 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 *Micromite 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 the Micromite allows you to 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 *Micromite User Manual* and on line resources such as: http://en.wikipedia.org/wiki/1-Wire

Chapter 8

LCD Display Panels

A feature of the Micromite that sets it apart from other microcontrollers is its support for an attached touch sensitive LCD panel. The 28 and 44-pin Micromites support LCD panels based on the ILI9341controller and these can be purchased on eBay for under $10.

These panels come in a number of sizes ranging from 2.2" (diagonal measurement) to 2.8" and have a resolution of 240 x 320 pixels with each pixel being able to display any one of 17 thousand colours. The Micromite Plus (the 64 and 100-pin chips) will support a much larger range covering panels from a tiny 1.8 inches to a very large 9" with the larger displays offering 800 x 600 pixels with each pixel capable of displaying true 24-bit colour.

Nothing extra is required to drive these LCD panels, so for this minimal cost you can design a sophisticated project with colourful touch sensitive buttons and check boxes for the user input.

As an example, the photo on the right shows the 28-pin Micromite driving a digital synthesised function generator chip to make a full featured signal generator.

There is nothing special here, the program is written in standard MMBasic and you can easily create a similar project yourself.

Details of this project can be found at http://geoffg.net/SignalGenerator.html

Suitable Display

The rest of this chapter assumes that you are using a 28-pin Micromite connected to an LCD panel using the ILI1963 controller. The Micromite User Manual goes into the detail of sourcing and connecting this display so the following is a summary.

You will find hundreds of these displays on eBay by searching for the keyword "ILI9341". The display that you purchase must look like that illustrated on the right. There are many variations for sale but the Micromite has been tested with the illustrated panel so you can be sure that it will work.

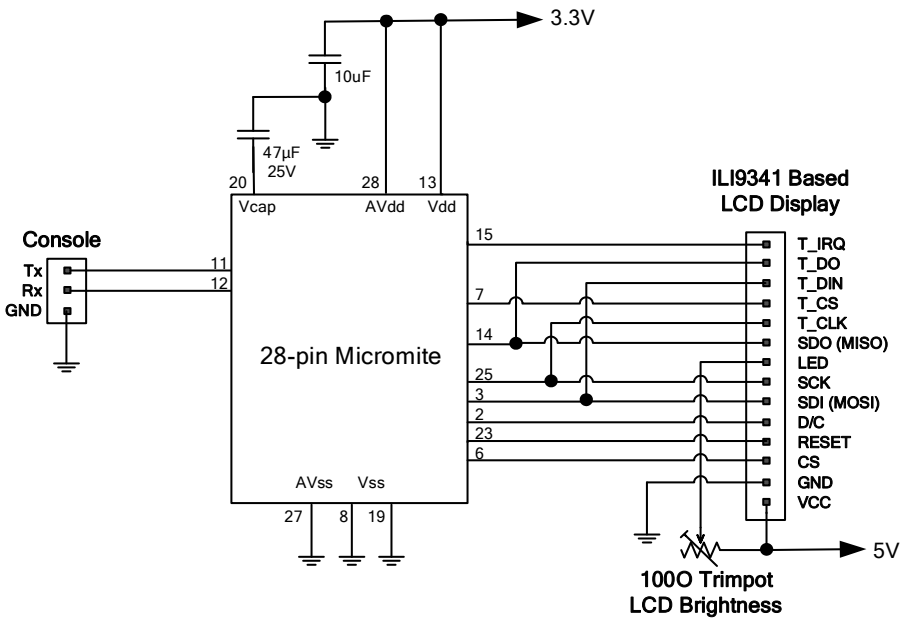
Displays' using the ILI9341 controller come in three sizes (2.2, 2.4 and 2.8 inch diagonal measurement) and you can use whatever size that you want although you should be aware that most 2.2 inch displays to not include the touch sensitive screen feature.

The display and Micromite can be connected together using a solderless breadboard. However, using the Micromite LCD Backpack project (http://geoffg.net/backpack.html) it is easy to build a project using an LCD display panel as this includes the printed circuit board, power supply, etc.

Connecting a Display

The LCD panel uses the SPI protocol to communicate with the Micromite plus few other signals which can be allocated within MMBasic to different I/O pins as required.

This gives the designer a lot of flexibility but, for the sake of simplicity, the following tutorial will assume that you are using the Micromite LCD Backpack, or at least the same connections to the panel as shown in this diagram.



Configuring the Micromite

Before you can use the LCD panel you need to configure the Micromite for the LCD display panel. These commands setup the hardware features of the Micromite for the display and are saved in non volatile flash memory. This means that they only need to be run once; from then on the display will be automatically setup on power up.

All the following commands assume that you are using the Micromite LCD Backpack with the connections illustrated above.

The first step is to tell the Micromite that a display is connected and what I/O pins are used for critical signals. To do this type the following line at the command prompt and hit the enter key:

```
OPTION LCDPANEL ILI9341, L, 2, 23, 6
```

This tells the Micromite that the LCD panel is connected and the I/O pins used for the reset, RW and device select signals Following this command the Micromite will initialise the display (which should go dark) and return to the command prompt.

You can test the display by entering the following at the command prompt:

```
GUI TEST LCDPANEL
```

This will cause the Micromite to rapidly draw a series of overlapping coloured circles on the display (as shown on the right). This animated test will continue until you press a key on the console's keyboard and MMBasic will then return to the command prompt.

To configure the touch feature you should enter the following at the command prompt:

```
OPTION TOUCH 7, 15
```

This allocates the I/O pins for the touch controller interrupt and device select signals. Before you can use the touch facility you need to calibrate it and this is done with the following command:

```
GUI CALIBRATE
```

This will cause MMBasic to draw a target at the top left hand corner of the screen (as shown on the right). Using a pointy but blunt object (eg, a toothpick) press on the exact centre of the target. After a second the target will disappear and when you lift your touch another target will appear on the top right.

In this fashion the target will be displayed on all four corners of the display and the touch feature will be calibrated for the display. At the end the message "Done. No errors" should be displayed on the console. You also might get a message indicating that the calibration was inaccurate and in that case you should repeat it taking more care to accurately touch the target.

You can now test the touch facility with the command:

```
GUI TEST TOUCH
```

This will clear the screen and when you touch it pixels will be illuminated at the touch point. This enables you to test the accuracy of the calibration – using a stylus touch the screen and the pixels under the touch point should light. Pressing any key will terminate the test.

Graphic Coordinates

All operations on the LCD screen are done in terms of pixels and the standard ILI9341 based panels have a resolution of 240 pixels vertically by 320 pixels horizontally. Pixel coordinates are specified by an X (horizontal) coordinate and a Y (vertical) coordinate.

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 = 319 and Y = 239 are the coordinates of the bottom right corner of the screen.

The PIXEL command will set the colour of an individual pixel 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 160, 120, RGB(blue)
```

Defining Colour

All colours in MMBasic are specified as a 24-bit number (the same as your desktop PC). The top eight bits 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 160, 120, 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 160, 120, 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 160, 120, RGB(yellow)
```

The colours that you can specify this way are red, green, blue, yellow, cyan, purple, white and black.

The 24-bit value used to specify a colour has over 64 million variations however many panels can not display this number of colours. This need not concern you as MMBasic will automatically convert the 24-bit colour value to the colour range supported by the display. The more advanced Micromite Plus can drive many more displays with some natively supporting 24-bit colour and it will also automatically adjust the colour to suit them if needed.

Finally, if you want to store a colour number in a variable make sure that the variable is an integer. A floating point number cannot accurately store a 24-bit number so some of the data will be lost if you tried to do that. So always use integers for storing colour values. For example:

```
PixColour% = RGB(yellow)
PIXEL 160, 120, PixColour%
```

Drawing on the Screen

There are eight 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.

- `TEXT X, Y, STRING, ALIGNMENT, FONT, SCALE, C, BC`
  Displays a string starting at X and Y. ALIGNMENT is one or two 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 319
  PIXEL i, 120, RGB(white)
NEXT i
```

However it is simpler to use the LINE command:

```
LINE 0, 120, 319, 120, 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.

Rounded boxes are useful for drawing on-screen buttons as we will demonstrate later.

The CIRCLE command, as its name suggests, will draw a circle.

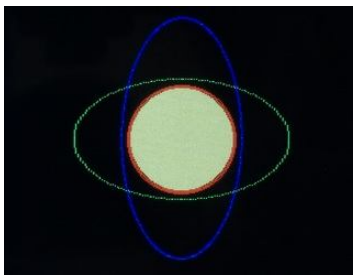
```
CIRCLE X, Y, R, LW, A, C, FILL
```

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 160, 120, 45, 3, 1, RGB(red), RGB(yellow)
Circle 160, 120, 100, 1, 0.5, RGB(blue)
Circle 160, 120, 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.

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 or two 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.

'font' is the font number that should be used (the Micromite 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.

Most parameters are optional so, for example, you can just use the following to print the word "Micromite" near the top left of the screen.

```
TEXT 10, 10, "Micromite"
```

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.

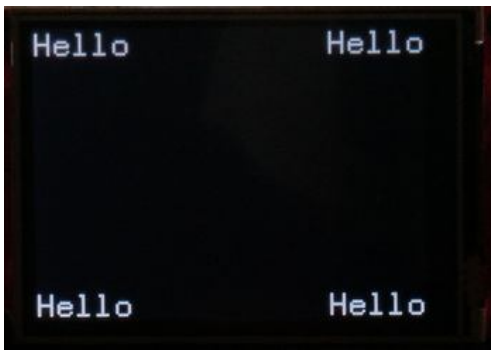
The alignment parameter is particularly useful as it allows you to position the text much easier. For example to perfectly centre the text on a 320x240 pixel screen you can use:

```
TEXT 160, 120, "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 319, 0, "Hello", "R", 1, 2
TEXT 0, 239, "Hello", "B", 1, 2
TEXT 319, 390, "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"
```

Fonts

The 28 and 44-pin Micromites include one built in font which is 8 pixels wide by 13 pixels high and includes all 95 standard ASCII characters with the back quote character (60 hex or 96 decimal) replaced with the degree symbol (º). Within MMBasic this is referred to as font #1.

The 64 and 100-pin Micromite Plus includes six fonts numbered #1 to #6. The first font (#1) is the same as above but the others provide larger and clearer fonts.

If required, additional fonts can be embedded in a BASIC program. The MMBasic distribution (ie, zip file) includes over a dozen embedded fonts covering a wide range of character sets and includes symbol fonts (Dingbats) which are handy for creating on screen icons, etc. These fonts work exactly same as the built in fonts (ie, selected using the FONT command or specified in the TEXT command).

Each embedded font looks like this (this is an example, not a working font):

```
DefineFont #8
  0B303220 00000000 00000000 00F00F00 00F81F00 00FE7F00 00FFFF00 00FFFF01
  801FF803 C00FF003 C007E007 E003C007 E003C00F F001800F F001800F F001801F
  00F80300 00F80300 00F80300 00F00100 00E00000 00000000 00000000 00000000
End DefineFont
```

The keyword `DefineFont` signals the beginning of the font and it is followed by a number which is the font number that can be used in commands like TEXT that use a font number. The font definition is terminated by the keywords `End DefineFont`. In-between the two are a series of eight digit hex numbers which define the bit image of the characters in the font. The font definition can be placed anywhere in your program and if MMBasic runs into it during execution of the program it will skip over it.

Embedded fonts can be very large so you are limited in the number of these that you can embed in a program. One way of overcoming this difficulty is to copy the fonts into the library area of the Micromite where they are compressed and take up less space. This is described fully in the *Micromite User Manual*.

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.

Touch Input

The ILI9341 based displays used with the 28 and 44-pin Micromites are generally supplied with a resistive touch sensitive panel and associated controller chip which is fully supported in MMBasic. Many of the additional displays supported by the Micromite Plus also are touch sensitive. To detect if and where the screen is touched you can use the following functions in a BASIC program:

TOUCH(X) Returns the X coordinate of the currently touched location.

TOUCH(Y) Returns the Y coordinate of the currently touched location.

All coordinates and measurements on the screen are done in terms of pixels with the X coordinate being the horizontal position and Y the vertical position.

If the screen is not touched both will return -1.

As a simple demonstration the following program will display in the centre of the LCD panel two numbers representing the X and Y coordinates of the current touched point.

```
CLS
DO
  TEXT 120, 120, STR$(TOUCH(X),3) + "  " + STR$(TOUCH(Y),3)
LOOP
```

The touch controller on the LCD display panel will signal when the screen has been touched by pulling the IRQ signal line low and this can be used to generate an MMBasic interrupt. The IRQ pin was defined when you originally configured the touch feature; it is the last parameter as shown in bold in the following command line:

```
OPTION TOUCH 7, 15
```

You can setup an interrupt as follows:

```
SETPIN 15, INTL, MyTouchInt
```

Then, whenever the screen is touched, the touch controller on the LCD panel will pull pin 15 low and MMBasic will interrupt the current program flow to call the subroutine MyTouchInt.

The following small program demonstrates how this can be used. Every time the screen is touched the program will draw a small cross at the location of the touch. This is done entirely in the interrupt subroutine, the main program is just running an empty loop but it could be doing something useful.

```
CLS
SETPIN 15, INTL, DrawCross  ' pin 15 is the touch IRQ
DO
  ' < program can be doing something useful >
LOOP

SUB DrawCross
  CLS
  LOCAL x = TOUCH(X)
  LOCAL y = TOUCH(Y)
  LINE x - 10, y - 10, x + 10, y + 10
  LINE x - 10, y + 10, x + 10, y - 10
END SUB
```

Drawing Buttons

When you use the touch screen as the input method for your gadget you will inevitably need some on screen buttons as shown in the example on the right.

It easy to create a small number of them (just use a box with rounded corners) but it is trickier when you have a screen full of buttons and your program needs to tell when they have been touched while doing other things.

There are many ways to implement this however in the remainder of this chapter we will describe one way which is similar to that used in the Boat Computer (http://geoffg.net/BoatComputer2.html) illustrated in the above screen shot.

First we need a subroutine to draw a button:

```
SUB DrawBtn x, y, w, h, fc AS INTEGER, bc AS INTEGER, s AS STRING
  RBOX x, y, w, h, , fc, bc
  TEXT x + w/2, y + h/2, s, CM, , , fc, bc
END SUB
```

This takes seven arguments (the X and Y coordinates, the width, height, the foreground colour (fc), the background colour (bc), and the caption on the button (s). Note that the colours are specified as integers and the caption as a string. You would call this subroutine like this:

```
DrawBtn 100, 120, 70, 25, RGB(cyan), RGB(black), "RUN"
```

Note that the appearance of the caption depends on the default font and scaling being previously set using the FONT command.

When you have a lot of buttons to draw and maintain it is convenient if their parameters (position, size, etc) are stored in arrays where you can step through them using FOR loops, etc. So we need to define arrays for all our buttons:

```
CONST NbrBtn = 20
DIM FLOAT bx(NbrBtn), by(NbrBtn), bw(NbrBtn), bh(NbrBtn)
DIM INTEGER bfc(NbrBtn), bbc(NbrBtn)
DIM STRING bs(NbrBtn)
```

By defining the number of buttons as a constant in the first line it makes it simple to change the number as the program changes.

To make it easier when dealing with a lot of buttons it would be useful to have a subroutine that will load the parameters into these arrays and, at the same time, draw the button on our screen:

```
SUB InitBtn n, x, y, w, h, fc AS INTEGER, bc AS INTEGER, s AS STRING
  bx(n) = x : by(n) = y : bw(n) = w : bh(n) = h
  bfc(n) = fc : bbc(n) = bc : bs(n) = s
  DrawBtn x, y, w, h, fc, bc, s
END SUB
```

The first parameter (n) is the reference number of the button and is the index into the array where the button information will be stored. As an example, to setup and draw the buttons used in the boat computer example screen shown above you would run the following code:

```
InitBtn 0, 204, 0, 116, 28, RGB(cyan), RGB(black), "CHANGE"
InitBtn 1, 204, 62, 116, 28, RGB(cyan) , RGB(black), "CHANGE"
InitBtn 2, 204, 124, 116, 28, RGB(cyan) , RGB(black), "CHANGE"
InitBtn 3, 0, 206, 195, 36, RGB(red) , RGB(black), "SET TO HERE"
InitBtn 4, 207, 206, 112, 36, RGB(white) , RGB(black), "SAVE"
```

Now we need to detect if a button has been touched. This time we need to use a function as it will be required to return a value (true if the button is touched, false otherwise):

```
FUNCTION CheckBtnDown(n)
  LOCAL x = TOUCH(x), y = TOUCH(y)
  IF x > bx(n) AND x < bx(n)+bw(n) AND y > by(n) AND y < by(n)+bh(n) THEN
    DrawBtn bx(n), by(n), bw(n), bh(n), bbc(n), bfc(n), bs(n)
    CheckBtnDown = 1
  ENDIF
END FUNCTION
```

This will check if the touch is within the boundaries of the button and if so, will draw the button in reverse video to acknowledge the touch. It also sets the returned value of the function to 1 (ie, true).

Note that we store the x and y coordinates of the touch in local variables (x and y) as that makes the following code less cluttered. If the screen is not touched x and y will be set to -1 which is outside the area of all buttons so the function will always return false in this case. Finally we do not bother to set the value of the function to zero (false) if the check failed because that is the default value for a function anyway.

We also need a subroutine to check if the touch has been lifted and wait for that event before redrawing the button in its normal (untouched) form:

```
SUB WaitBtnUp(n)
  DO WHILE TOUCH(x) <> -1 : LOOP
  DrawBtn bx(n), by(n), bw(n), bh(n), bfc(n), bbc(n), bs(n)
END SUB
```

Used together both this function and subroutine make it simple to detect and respond to a button press. For example, the following program fragment will check if button 3 is being touched and, if it is, do whatever is required. The program will then wait for the touch to be lifted before continuing:

```
IF CheckBtnDown(3) THEN
  ' < do whatever button 3 needed done>
  WaitBtnUp(3)
ENDIF
```

Most embedded control programs run in a loop checking for inputs and responding accordingly and this code could be placed in that loop. This would ensure that the buttons are regularly checked for touch and acted upon immediately. If you did not want to embed the check in the loop, or you did not want your program to wait for the removal of touch you could use the touch interrupt (described earlier) to trigger a call to one subroutine on both touch up and down. Within this subroutine you could set global flags to indicate what action was required (for example, respond to button 3).

If you want to try this program on your own Micromite the following is the full listing of the program. You should be able to copy from this document to your clipboard.

```
CONST NbrBtn = 5
DIM FLOAT bx(NbrBtn), by(NbrBtn), bw(NbrBtn), bh(NbrBtn)
DIM INTEGER bfc(NbrBtn), bbc(NbrBtn)
DIM STRING bs(NbrBtn)

CLS
FONT 1, 2

InitBtn 0, 204, 0, 116, 28, RGB(cyan), RGB(black), "CHANGE"
InitBtn 1, 204, 62, 116, 28, RGB(cyan) , RGB(black), "CHANGE"
InitBtn 2, 204, 124, 116, 28, RGB(cyan) , RGB(black), "CHANGE"
InitBtn 3, 0, 206, 195, 36, RGB(red) , RGB(black), "SET TO HERE"
```

```
InitBtn 4, 207, 206, 112, 36, RGB(white) , RGB(black), "SAVE"

DO
  IF CheckBtnDown(0) THEN
    ' < do whatever button 0 needed done>
      WaitBtnUp(0)
  ENDIF
  IF CheckBtnDown(1) THEN
    ' < do whatever button 1 needed done>
      WaitBtnUp(1)
  ENDIF
  IF CheckBtnDown(2) THEN
    ' < do whatever button 2 needed done>
      WaitBtnUp(2)
  ENDIF
  IF CheckBtnDown(3) THEN
    ' < do whatever button 3 needed done>
      WaitBtnUp(3)
  ENDIF
  IF CheckBtnDown(4) THEN
    ' < do whatever button 4 needed done>
      WaitBtnUp(4)
  ENDIF
LOOP


SUB DrawBtn x, y, w, h, fc AS INTEGER, bc AS INTEGER, s AS STRING
  RBOX x, y, w, h, , fc, bc
  TEXT x + w/2, y + h/2, s, CM, , , fc, bc
END SUB

SUB InitBtn n, x, y, w, h, fc AS INTEGER, bc AS INTEGER, s AS STRING
  bx(n) = x : by(n) = y : bw(n) = w : bh(n) = h
  bfc(n) = fc : bbc(n) = bc : bs(n) = s
  DrawBtn x, y, w, h, fc, bc, s
END SUB

FUNCTION CheckBtnDown(n)
  LOCAL x = TOUCH(x), y = TOUCH(y)
  IF x > bx(n) AND x < bx(n)+bw(n) AND y > by(n) AND y < by(n)+bh(n) THEN
    DrawBtn bx(n), by(n), bw(n), bh(n), bbc(n), bfc(n), bs(n)
    CheckBtnDown = 1
  ENDIF
END FUNCTION

SUB WaitBtnUp(n)
  DO WHILE TOUCH(x) <> -1 : LOOP
  DrawBtn bx(n), by(n), bw(n), bh(n), bfc(n), bbc(n), bs(n)
END SUB
```

Numeric Key Pad

A more sophisticated usage of buttons is to draw an on-screen keypad as illustrated on the right.

The program listing below is a full working demonstration of how to implement such feature and you can copy the code with minimal modifications direct into your program. It uses

the same functions defined above so the previous description of how they work applies to this example also.

The program starts by declaring the arrays needed to hold the button's parameters. It then drops into a simple loop where it gets a number from the user (via the on screen keypad) and displays it on the console output. Your program will obviously do something more useful with this value.

All the work is done in the function KeyPadInput. It is called with one parameter which is the prompt to display (in yellow) and it returns with the number entered when the user touched the SAVE button.

Within the function KeyPadInput the keypad is drawn using a FOR loop which calculates the location of each button as it executes the loop. Similarly the buttons are checked for touch by another FOR loop which checks each button in turn.

You should be able to copy this program from this document to your clipboard and then send it to a Micromite with an attached display and it will run "as is". If you are going to type the program in yourself, be aware that the following line has been split in the listing:

```
InitBtn btn, bw/2 + bw * (btn Mod 3), bh + bh * (btn \ 3), bw - 4, bh - 4, RGB(cyan), RGB(black), SCap(btn)
```

```
CONST NbrBtn = 11
DIM FLOAT bx(NbrBtn), by(NbrBtn), bw(NbrBtn), bh(NbrBtn)
DIM INTEGER bfc(NbrBtn), bbc(NbrBtn)
DIM STRING bs(NbrBtn)

FONT 1, 2

DO
  ' repeatedly get a number from the on screen keypad
  ' and print on the console
  PRINT KeyPadInput("Freq (Hz)")
LOOP


' this displays a keypad for inputting a number
FUNCTION KeyPadInput(s$)
  LOCAL btn, nbr = 0
  LOCAL STRING SCap(9) = ("7","8","9","4","5","6","1","2","3","0")
  CONST bh = 48, bw = 80

  ' draw the top line with the prompt and current value (ie, 0)
  CLS
  TEXT 0, 10, s$, , 1, 2, RGB(yellow)
  TEXT 300, 9, STR$(nbr), R, 1, 2

  ' draw the main keypad buttons
  FOR btn = 0 TO 8
    InitBtn btn, bw/2 + bw * (btn Mod 3), bh + bh * (btn \ 3), bw - 4, bh - 4, RGB(cyan),
RGB(black), SCap(btn)
  NEXT btn

  ' draw the special buttons
  InitBtn btn, bw * 1.5, bh + bh * (btn \ 3), bw - 4, bh - 4, RGB(cyan), RGB(black), "0"
  InitBtn 10, 6, bh*4, 110, bh - 4, RGB(magenta), RGB(black), "DEL"
  InitBtn 11, 201, bh*4, 110, bh - 4, RGB(white), RGB(black), "SAVE"

  DO
    ' check if a button has been touched
    FOR btn = 0 TO 11
      IF CheckBtnDown(btn) THEN EXIT FOR
    NEXT btn

    ' action the button (btn > 11 means no button touched)
    SELECT CASE btn
```

```
      CASE 0 TO 9
        IF LEN(STR$(nbr)) < 6 THEN nbr = nbr * 10 + VAL(SCap(btn))
      CASE 10
        nbr = nbr \ 10
      CASE 11
        KeyPadInput = nbr
      CASE 12
        CONTINUE DO
    END SELECT
    TEXT 300, 9, " " + STR$(nbr), R, 1, 2
    PAUSE 150
    WaitBtnUp btn
  LOOP UNTIL btn = 11   ' exit the loop (and function) if DONE touched
END FUNCTION


SUB DrawBtn x, y, w, h, fc AS INTEGER, bc AS INTEGER, s AS STRING
  RBOX x, y, w, h, , fc, bc
  TEXT x + w/2, y + h/2, s, CM, , , fc, bc
END SUB

SUB InitBtn n, x, y, w, h, fc AS INTEGER, bc AS INTEGER, s AS STRING
  bx(n) = x : by(n) = y : bw(n) = w : bh(n) = h
  bfc(n) = fc : bbc(n) = bc : bs(n) = s
  DrawBtn x, y, w, h, fc, bc, s
END SUB

FUNCTION CheckBtnDown(n)
  LOCAL x = TOUCH(x), y = TOUCH(y)
  IF x > bx(n) AND x < bx(n)+bw(n) AND y > by(n) AND y < by(n)+bh(n) THEN
    DrawBtn bx(n), by(n), bw(n), bh(n), bbc(n), bfc(n), bs(n)
    CheckBtnDown = 1
  ENDIF
END FUNCTION

SUB WaitBtnUp(n)
  DO WHILE TOUCH(x) <> -1 : LOOP
  DrawBtn bx(n), by(n), bw(n), bh(n), bfc(n), bbc(n), bs(n)
END SUB
```

Example Programs

There are many requirements when it comes to drawing graphics and responding to touch and it is handy to see how other programs tackle this task. There are five programs for the 28-pin Micromite that were written in part to demonstrate graphics programming and it would be worth looking through them for inspiration or as a source of handy subroutines and functions.

These are:

The Parking Assistant: http://geoffg.net/ParkingAssistant.html

The Boat Computer: http://geoffg.net/BoatComputer2.html

The Super Clock: http://geoffg.net/SuperClock.html

The DDS Signal Generator: http://geoffg.net/SignalGenerator.html

Air Quality Monitor: http://geoffg.net/AirQualityMonitor.html

Chapter

9

The Micromite Plus

The Micromite Plus is an advanced version of the standard Micromite which has been the focus of this tutorial up to this point. It has all the capabilities of the standard device and adds many extra features that make it suitable for more demanding applications.

Generally the standard 28 and 44-pin Micromite's are good for most microcontroller tasks (eg, a heating controller, the smarts inside a gadget, etc) while the Micromite Plus is suited to more demanding tasks such as controlling a lathe, running a small industrial process, etc.

In summary the extra features of the Micromite Plus are:

- Double the amount of memory for both BASIC programs and variables/data.
- A higher clock speed so that programs will run more than twice as fast.
- Additional input/output features including more I/O pins, serial ports, SPI ports, etc.
- An on chip USB 2.0 interface for connecting to a personal computer or laptop.
- Support for SD cards which can be used for loading/saving programs and data.
- Support for a wider range of LCD display panels from 1.4" to 9" with high speed screen updates and true colour (24 bit) display on the larger panels.
- Advanced on screen graphic controls for touch sensitive LCD panels. These controls include buttons, switches, pop up keyboards, radio buttons, number entry, etc.
- The ability to add a standard PS2 keyboard and display the console output to the attached LCD display panel. The result is a compact all-in-one BASIC computer similar to the Tandy and Commodore computers of the early 1980's.

This chapter will provide an overview of these additional features. It is assumed that the reader is familiar with the standard Micromite so only the extra features of the Micromite Plus will be covered.

For more detail on the subjects covered in this chapter the *Micromite Plus Users Manual* should be consulted and that can be downloaded from: http://geoffg.net/micromite.html, Note that, like this chapter, the user manual only covers the **additional** features of the Micromite Plus so you will also need to keep the standard Micromite's documentation handy for features that are common to both versions of the Micromite.

One point to note is that the Micromite Plus is only available in a surface mount package with a pin spacing of 0.5mm. However there are a number of PC modules designed for the Micromite Plus that bring its I/O pins out on a more breadboard friendly 0.1 inch pitch.

Physical Characteristics

The Micromite Plus is based on the PIC32MX470 series of chips from Microchip. These come in 64-pin and 100-pin TQFP packages, both of which are surface mount.

For the 64-pin Micromite the recommended Microchip part is: PIC32MX470F512H-120/PT while for the 100-pin Micromite the recommended chip is: PIC32MX470F512L-120/PF

Compared to the PIC32MX170 series of chips which are used for the standard 28 and 44-pin Micromites the MX470 series has double the memory, faster clock speed and additional hardware features such as extra serial ports, USB and a more. It is this extra capability that is used to implement the additional features of the Micromite Plus.

This table summarises the key differences across the Micromite family:

	Micromite		**Micromite Plus**	
	28-pin Thru Hole	44-pin Surface Mount	64-pin Surface Mount	100-pin Surface Mount
Maximum CPU Speed	48 MHz	48 MHz	120 MHz	120 MHz
Maximum BASIC Program Size	59 KB	59 KB	100 KB	100 KB
RAM Memory Size (used for variables, buffers, etc)	52 KB	52 KB	97 KB	97 KB
Clock Speed (MHz)	5 to 48	5 to 48	5 to 120	5 to 120
Serial Console for Programming and Control	✓	✓	✓	✓
USB Console (serial emulator over USB 2.0)			✓	✓
SD Card Interface (FAT16 or FAT32 up to 64GB)			✓	✓
Total Number of I/O pins	19	33	45	77
Number of Analog Inputs	10	13	28	28
Number of Serial I/O ports	2	2	3 or 4	3 or 4
Number of SPI Channels	1	1	2	2
Number of I^2C Channels	1	1	1 + RTC	1 + RTC
Maximum Number of 1-Wire I/O pins	19	33	45	77
Number of PWM or Servo Channels	5	5	5	5
Supports 2.2", 2.4" and 2.8" SPI LCD Displays	✓	✓	✓	✓
Supports eleven LCD Displays from 1.4" to 9"			✓	✓
Supports Resistive Touch Panels	✓	✓	✓	✓
PS2 Keyboard and LCD Display Console			✓	✓
Power Requirements	3.3V 30 mA	3.3V 30 mA	3.3V 80 mA	3.3V 80 mA

USB Interface

The USB interface on the Micromite Plus acts as a peripheral device and is intended to be connected to a host, for example a desktop computer or laptop. The Micromite Plus implements the CDC (Communication Device Class) protocol which means that it looks like a standard serial port to your host computer. No additional components are required to support USB (the USB socket connects directly to the Micromite chip).

This serial port acts exactly the same as the normal serial console (which is also implemented on the Micromite Plus) so this means that you can start a terminal emulator such as Tera Term and when you connect to the virtual serial port created by the Micromite Plus you will have access to the command prompt for entering programs, commands, etc. In fact both the USB interface and the normal serial interface act in parallel.

Using the USB interface means that you do not need a USB to serial converter to program and control the Micromite as described in Chapter 1 of this book. The only downside of this arrangement is that if you force the Micromite Plus to restart (perhaps by using the CPU RESTART command) the USB connection on the host computer will be lost requiring you to reconnect with your terminal emulator. This can happen often when you are developing programs so it is better to reserve the USB port for use on the final installed system.

SD Card Storage

Like the USB interface an SD card can also directly connect to the Micromite Plus. MMBasic will work with cards up to 64GB formatted as FAT16 or FAT32 and within MMBasic the card will act as a disk drive allowing programs and data to be saved and read. The card and files can also be read or written on a PC or laptop so this makes a convenient way of transferring data to and from a larger computer.

Using MMBasic commands you can list the contents of the SD card, change folders, delete files and so on. Loading and saving programs is also easy – the SAVE and LOAD commands will do that for you.

You can also ask MMBasic to load and run a program which allows one program to transfer control to another program on the SD card. This makes it easy to create modular programs where one program could implement one function while another could handle a different function. It also allows a BASIC program to update itself from a SD card in the field.

The full list of file management commands is:

- FILES List the contents of the SD card
- KILL Delete a file
- MKDIR Make a directory/folder
- CHDIR Change directory
- RMDIR Delete a directory
- LOAD Load a BASIC program from the SD card
- SAVE Save the current program to the SD card
- LOAD IMAGE Load a BMP file and display it on the LCD screen
- SAVE IMAGE Save the current LCD screen image as a BMP file

Saving and reading data from the SD card is more complicated so we will cover that in more detail. 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
```

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 big 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 (on pins 21, 22 and 23) and later 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(21) "," TEMPR(22) "," 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 9600ms 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 INTEGER 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.

LCD Display Panels

The LCD display panels supported by the Micromite Plus fall into two categories:

- Panels that are connected by the SPI bus. These use a variety of controllers and come in six sizes from 1.4 inch (diagonal measurement) to 2.8 inches. Included in this are panels based on the ILI9314 controller which are also supported by the standard 28 and 44-pin Micromites – so any program written for the standard Micromite and this display will also run on the Micromite Plus with no modification.
- Panels based on the SSD1963 controller and are connected by a parallel 8-bit bus. These come in five sizes from 4.3 inch to 9 inches and have higher resolutions, much faster screen update and support 24-bit true colour (16 million different colours).

Because the Micromite Plus is usually chosen for more demanding tasks it is most often coupled with panels based on the SSD1963 controller as they allow for more information to be displayed at any time. Of these the 5" and 7" panels are the most popular as they have excellent characteristics and many vendors sell them on eBay at reasonable prices. For these reasons it is recommended that beginners who want to couple an LCD display with the Micromite Plus start with a 5" or 7" panel.

The basic support provided within MMBasic for these panels is the same as discussed in Chapter 8. This means that you can use commands such as LINE, CIRCLE, TEXT, etc and detect touch using the TOUCH() function. This also means that programs written for displays supported by the standard Micromite should run the same on the Micromite Plus with the larger displays.

An additional feature of these displays is their higher resolution and colour range. The 5" to 9" panels have a resolution of 800 by 480 pixels and can display 16 million colours. This means that they can be used to display colourful and detailed images as shown on the right (this is a photograph of an actual image on a 5" LCD panel).

This image was loaded from an SD card using the LOAD IMAGE command. The ability to display photographic images is often used to display a realistic photographic background image for a control panel. This can include dials, logos, shading, etc and with that in place the BASIC program only has to update small details on the screen rather than draw the whole thing.

The Micromite Plus also includes six fonts ranging from the small font supported by the standard Micromite up to quite large high resolution fonts suited to large displays.

Font	Size	Character Set	Description
1	8 x 13	All 95 characters	A small font where a dense display is required.
2	12 x 20	All 95 characters	General use on 480 x 272 displays
3	16 x 24	All 95 characters	General use on 800 x 480 displays
4	16 x 24 BOLD	All 95 characters	A bold version of font #3
5	24 x 32	All 95 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.

Advanced Graphics

An outstanding feature of the Micromite Plus is its ability to display and manage on screen controls. These are graphic elements that represent on screen switches, check boxes, etc. With a single command MMBasic can be instructed to display a control on the screen and it will also manage the animation of the control. This includes causing the image to depress when touched (in the case of an on screen switch) or displaying a check mark (in the case of a check box), etc.

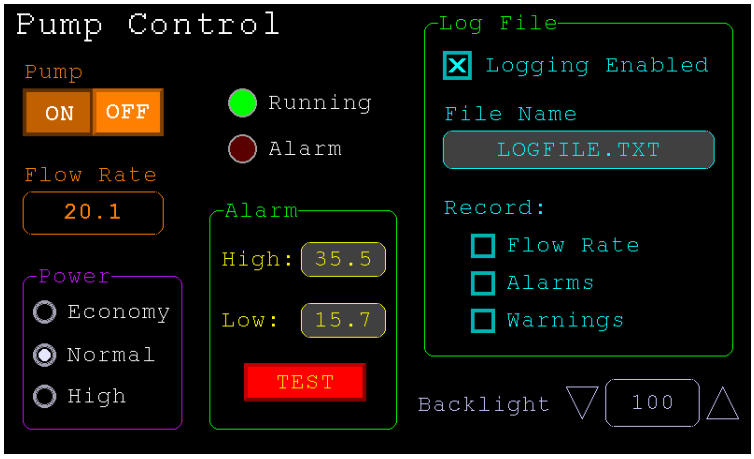
Once an on screen control has been created the BASIC program only has to check the state of the control (ie, is it depressed or checked). It does not have to be concerned with details such as redrawing the control to show that it has been touched. Contrast this with the amount of programming required to display and manage an on screen button using the standard Micromite as described in Chapter 8.

There are 15 on screen controls available on the Micromite Plus. These are:

Push Button	A momentary button which is a switch with a caption on its face.
Switch	A latching switch with a caption on its face.
Check Box	A check box which is a small tick box with a caption.
Radio Button	A radio button with a caption (normally grouped together by a frame).

Frame	A frame to group controls (eg, radio buttons).
LED	An indicator light (it looks like a panel mounted LED).
Display Box	A box which is useful for displaying text, numbers and messages.
Text Box	A box for entering text (a QWERTY keyboard will appear).
Number Box	A box for entering numbers (a numeric keypad will appear).
Formatted Box	A box that allows formatted data such as date/time, etc to be inputted.
Spin Box	A box where a number can be set using the up/down icons on either end.
Caption	Display a text string on the screen.
Circular Gauge	Display a circular analogue gauge with a digital display in the centre.
Bar Gauge	Display either a horizontal or vertical bar gauge.
Area	Used for creating custom controls to be managed by the BASIC program.

The image on the right shows a typical use for these controls. It is a demonstration of a pump controller with a lot of functionality on the screen. Each of the on screen items was created by a single command and the BASIC program only has to check the state of these on screen controls to decide what action to take.



A 12 minute video is available which does a better job of showing how the advanced graphics work in the Micromite Plus. You can see it at: https://youtu.be/j12LidkzG2A

Beginners to programming should be aware that writing programs using these advanced controls requires a different approach compared to a normal program. This applies to all advanced graphic environments (eg, Windows) and does not follow the normal start to end program flow that many people are used to. This is because the user is in control of the program flow. For example, the user might operate this switch or another; they might change one value or another. The programmer cannot control this so the program must sit in a loop waiting for events and react accordingly rather than prompting the user for input.

The *Micromite User Manual* goes into a lot of detail regarding the advanced graphic controls and how to program with them so this tutorial will not repeat that. Just be warned that it can be a challenging process for beginners… but the reward is a sophisticated control panel/display that would normally take a huge amount of programming effort to achieve.

Sound Output

The Micromite Plus can play stereo WAV files stored on the SD card and this feature can be used to generate verbal announcements as well as musical attention getting sounds – much more sophisticated than a simple beep. The sound is played on the PWM 2 outputs as a stereo signal and requires a simple filter network to convert the PWM output to an analog signal suitable for feeding

into an amplifier. With a reasonably good speaker the output can have good quality and adds a professional flair to a Micromite Plus project.

Playing a WAV file is straight forward, the command is:

```
PLAY WAV file$ [, interrupt]
```

file$ is the name of the WAV file on the SD card and the audio will play in the background (ie, the program will continue without pause). *interrupt* is optional and is the name of a subroutine which will be called when the file has finished playing.

A related command is PLAY TONE which will generate pure sine waves with precise frequencies on the same outputs used for playing WAV files. This feature is intended for generating attention catching sounds but, because the frequency is very accurate, it can be used for many other applications. For example, signalling DTMF tones on a telephone line.

The syntax of the tone command is:

```
PLAY TONE left [, right [, dur]]
```

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

For both versions of the PLAY command the program can pause and resume playing and change the volume of each channel using commands such as PLAY PAUSE and PLAY VOLUME.

Self Contained Computer

On the Micromite Plus you can attach a PS2 keyboard and, if you are using a SSD1963 based LCD panel, display the console output on the LCD. This turns the Micromite Plus into a completely self contained computer with its own keyboard and display. Using the built in colour coded editor programs can be entered, edited and run without requiring another computer.

The LCD panel works as you would expect for a console, for example the prompt will show on the screen along with a blinking cursor. Error messages will show on the screen, long lines will wrap around and the display will scroll when the text reaches the bottom of the screen. A program can also draw graphics on the screen and use the advanced graphics controls while also using it as the console.

Using the Micromite Plus as a self contained computer requires an LCD panel with a reasonable size and this is where the 7 and 8 inch displays come into their own. Using this feature the Micromite Plus is a fun programming computer reminiscent of the many BASIC based computers of the 80s such as the Tandy TRS-80, Commodore 64 and Apple II.

So there you have it. The Micromite is a very capable chip at a low cost and is just waiting to be incorporated into your next gadget… and what will that be? Let the imagination begin.

- - - - -