PicoMiteVGA

A “Boot to BASIC” computer
Based On the
Raspberry Pi Pico

User Manual
MMBasic Ver 5.07.04

Rev 4

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

About

Peter Mather (matherp on the Back Shed Forum) led the project, ported MMBasic to the Raspberry Pi Pico and wrote the drivers for its hardware features. The MMBasic interpreter and this manual was written by Geoff Graham (http://geoffg.net). Mick Ames (Mixtel90 on the Back Shed Forum) wrote the PIO compiler and its corresponding documentation along with undertaking a huge amount of testing.

Support

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

Copyright and Acknowledgments

The PicoMiteVGA firmware and MMBasic is copyright 2011-2022 Geoff Graham and Peter Mather 2016-2022.

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

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

WAV file support is copyright 2019 David Reid.

JPG support is thanks to Rich Geldreich

The pico-sdk is copyright 2021 Raspberry Pi (Trading) Ltd.

TinyUSB is copyright tinyusb.org

The VGA driver code was derived from work by Miroslav Nemecek

The compiled object code (the .uf2 file) for the PicoMiteVGA is free software: you can use or redistribute it as you please. The source code is available from GitHub (https://github.com/UKTailwind/PicoMite and https://github.com/UKTailwind/PicoMite-VGA-Edition) and can be freely used subject to some conditions (see the header on the source files).

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

This Manual

The author of this manual is Geoff Graham. It is based on preliminary material written by Mick Ames and is distributed under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Australia license (CC BY-NC-SA 3.0)

Contents

Introduction 4
Loading The Firmware 5
Connecting It Up 6
PicoMiteVGA Hardware 10
Using MMBasic 12
Full Screen Editor 16
Using the I/O pins 18
Sound Output 25
SD Card Support 27
Special Device Support 31
Variables and Expressions 37
Subroutines and Functions 42
Graphics Commands and Functions 18
MMBasic Characteristics 45
Predefined Read Only Variables 47
Options 49
Commands 53
Functions 87
Obsolete Commands and Functions 100
Appendix A – Serial Communications 101
Appendix B – I2C Communications 103
Appendix C – 1-Wire Communications 106
Appendix D – SPI Communications 107
Appendix E – The PIO Programming Package 109
Appendix F – USB Serial Console 115
Appendix G – Programming in BASIC - A Tutorial 116

Introduction

The PicoMiteVGA is a Raspberry Pi Pico running a full featured BASIC interpreter with support for a VGA monitor and a PS2 keyboard for the console input/output.

This turns the Raspberry Pi Pico into a low cost self-contained desktop computer, similar to the "Boot Into BASIC" computers of the 1980s such as the Commodore 64, Apple II and Tandy TRS-80.

You can have fun writing programs to balance your checkbook, entertain/teach children about computing, play simple computer games and calculate the positions of the planets (to name a few activities).

The firmware to do all of this is totally free to download and use.

In summary the features of the PicoMiteVGA are:

- **The VGA output is 640 x 480 pixels** in monochrome mode or 320 x 240 pixels in colour mode with 16 colours (1 bit for red, 2 bits for green and 1 bit for blue). The VGA output is generated using the second CPU on the RP2040 processor in the Raspberry Pi Pico plus one PIO channel so it does not affect the BASIC interpreter which runs at full speed on the first CPU.
- **The PS2 keyboard** works as a normal keyboard with the function keys and arrow keys fully operational. It can be configured for the standard US layout used in the USA, Australia and New Zealand or specialised layouts used in the United Kingdom, Germany, France and Spain.
- **The BASIC interpreter is full featured** with floating point, 64-bit integers and string variables, long variable names, arrays of floats, integers or strings with multiple dimensions, extensive string handling and user defined subroutines and functions. Typically it will execute a program up to 100,000 lines per second. MMBasic allows the embedding of compiled C programs for high performance functions and the running program can be protected from being listed or modified by a PIN number.
- **Full support for SD cards**. This includes opening files for reading, writing or random access and loading and saving programs. The firmware will work with cards up to 32GB formatted in FAT16 or FAT32 and the files created can also be read and written on personal computers running Windows, Linux or the Mac operating system.
- **USB interface to a Windows/Mac/Linux computer** provides an additional console interface. Programs can be easily transferred from a desktop or laptop computer (Windows, Mac or Linux) using the XModem protocol or by streaming the program over USB.
- **Full graphics support** allowing the BASIC program to display text and draw lines, circles, boxes, etc in up to 16 colours. Eight fonts are built in and additional fonts can be embedded in the BASIC program.
- **A full screen editor** is built into the PicoMiteVGA and can edit the whole program in one session. It includes advanced features such as colour coded syntax, search and copy, cut and paste to and from a clipboard.
- **Support for all Raspberry Pi Pico input/output pins**. These can be independently configured as digital input or output, analog input, frequency or period measurement and counting. Within MMBasic the I/O pins can be dynamically configured as inputs or outputs with or without pullups or pulldowns. MMBasic commands will generate pulses and can be used to transfer data in parallel. Interrupts can be used to notify when an input pin has changed state. PWM outputs can be used to create various sounds, control servos or generate computer controlled voltages for driving equipment that uses an analogue input (e.g. motor controllers).
- **The PicoMiteVGA has built in commands** to directly interface with infrared remote controls, the DS18B20 temperature sensor, LCD display modules, battery backed clock, numeric keypads and more.
- **A comprehensive range of communications protocols** are implemented including I^2C, asynchronous serial, RS232, SPI and 1-Wire. These can be used to communicate with many sensors (temperature, humidity, acceleration, etc) as well as for sending data to test equipment.
- **Power requirement is 5 volts** at approx 50 mA.

Loading The Firmware

The Raspberry Pi Pico comes with its own built in firmware loader that is easy to use.

Just follow these steps:

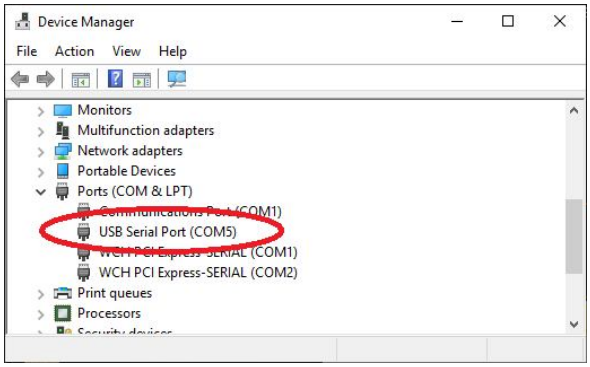
- Download the PicoMiteVGA firmware from http://geoffg.net/picomitevga.html and unzip the file. Identify the firmware which should be named something like “PicoMiteVGA V5.xx.xx.uf2”.
- Using a USB cable plug the Raspberry Pi Pico into your computer (Windows, Linux or Mac) **while holding down the white BOOTSEL button** on the Raspberry Pi Pico.
- The Raspberry Pi Pico should connect to your computer and create a virtual drive (the same as if you had plugged in a USB memory stick) called “RPI-RP2”. You can now release the BOOTSEL button. This drive will contain two files which you can ignore.
- Copy the firmware file (with the extension .uf2) to this virtual drive.
- When the copy has completed the Raspberry Pi Pico will restart and create a virtual serial port on your computer. The LED on the Raspberry Pi Pico will blink slowly indicating that the PicoMiteVGA firmware with MMBasic is now running.

While the virtual drive created by the Raspberry Pi Pico looks like a USB memory stick it is not, the firmware file will vanish once copied and if you try copying any other type of file it will be ignored.

Loading the PicoMiteVGA firmware may erase the flash memory including the current program, any programs saved in flash memory slots and all saved variables. So make sure that you backup this data before you upgrade the firmware.

It is possible for the flash memory to be corrupted resulting in unusual and unpredictable behaviour. In that case you should load the firmware file https://geoffg.net/Downloads/picomite/Clear_Flash.uf2 which will reset the Raspberry Pi Pico to its factory fresh state, then you can reload the PicoMiteVGA firmware.

USB Console

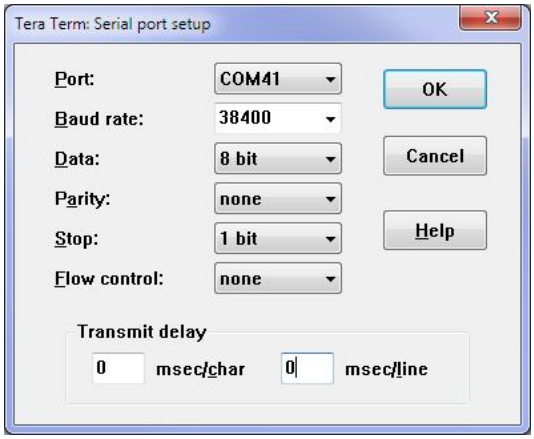


Normally the PicoMiteVGA is used with an attached VGA monitor and keyboard. However, the virtual serial port created after the firmware is loaded can be used to check that everything is running OK, even though you have not yet connected the monitor and keyboard.

The virtual serial port acts like a normal serial port but it operates over USB. Windows 10 includes a driver for this virtual serial port but with other other operating systems you may have to load a driver to make it work with the operating system (see Appendix F).

Once this is done you should note the port number created by your computer for the virtual serial connection. In Windows this can be done by starting Device Manager and checking the "Ports (COM & LPT)" entry for a new COM port as shown above.

Terminal Emulator



You also need a terminal emulator program on your desktop computer. For Windows users it is recommended that you use Tera Term (Tera Term can be downloaded from: http://tera-term.en.lo4d.com).

The screen shot on the right shows the setup for Tera Term. Note that the "Port:" setting will vary depending on which USB port your Raspberry Pi Pico was plugged into. The PicoMiteVGA ignores the baud rate setting so it can be set to any speed (other than 1200 baud which puts the Pico into firmware upgrade mode).

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

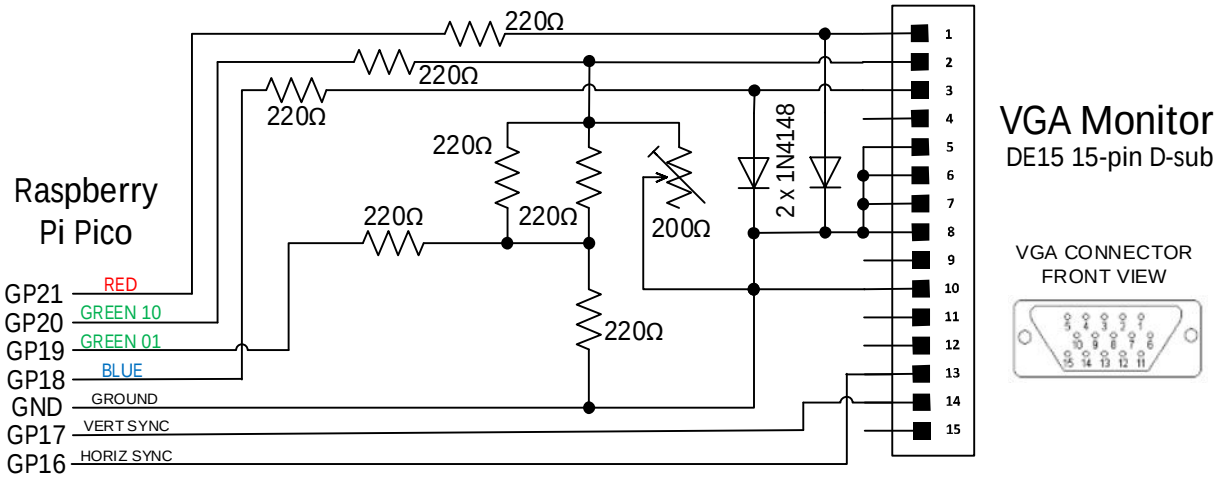
Once you have identified the virtual serial port and have connected your terminal emulator to it you should be able to press return on your keyboard and see the MMBasic prompt, which is the greater than symbol (e.g. “>”).

Connecting It Up

Normally the PicoMiteVGA / Raspberry Pi Pico is powered via its USB connector using a 5V USB charger or personal computer. See the section *PicoMiteVGA Hardware* for alternatives.

Connecting the VGA Monitor

The following diagram illustrates how to attach a VGA monitor:



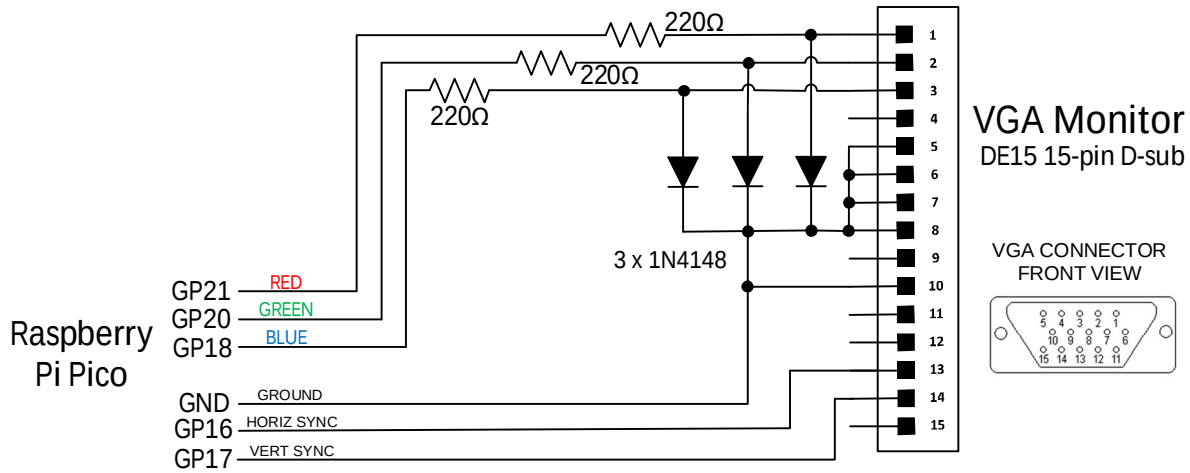
The 200Ω variable resistor should be a multiturn trimming potentiometer. This should be adjusted to give a neutral white colour when viewing a large area of white (use CLS RGB(WHITE) at the command prompt).

This circuit generates 16 colours in the 4-bit RGB121 format (i.e. 1 bit for red, 2 bits for green, and 1 bit for blue). The output is in the standard VGA format with a pixel rate of 25.175MHz and a frame rate of 60Hz. In monochrome mode the resolution is 640 x 480 however when the colour output is enabled (MODE 2) the pixels are duplicated along both the x and y axis giving a 320 x 240 resolution while the monitor still sees a 640 x 480 signal.

In both monochrome and colour modes there is a 38,400 byte framebuffer used to hold the display data arranged as 640 x 480 x 1-bit or 320 x 240 x 4-bit for the respective modes. Anything written to the framebuffer will appear on the VGA display within a maximum of 1/60 of a second.

The firmware uses the second RP2040 processor to feed the framebuffer data a line at a time via DMA to one of the RP2040's programmable I/O controllers (PIO0) to generate the display. As this runs independently of the main RP2040 processor there is little or no impact on the performance of MMBasic caused by generating the VGA output.

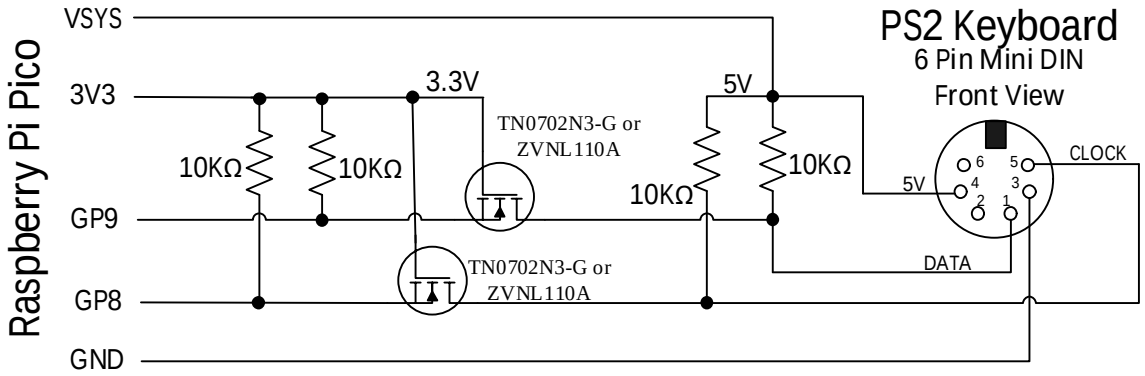
If you are happy just using just the eight standard full intensity colours in the RGB111 format (white, black, blue, green, cyan, red, magenta and yellow) you can use this simpler circuit:



Connecting the Keyboard

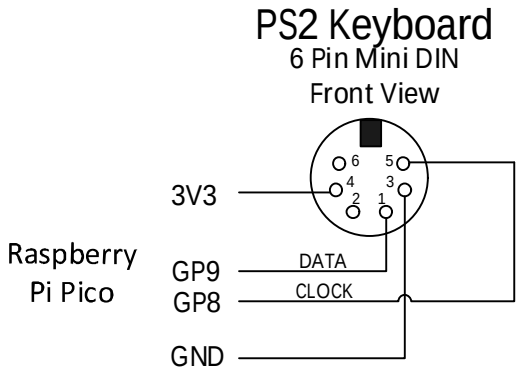
The PicoMiteVGA connects to a PS2 keyboard via the I/O pins GP8 (clock) and GP9 (data). However most PS2 keyboards run on 5V and this voltage would damage the Raspberry Pi Pico which must not be subjected to more than 3.6V. For this reason a level shifter should be employed so that the Raspberry Pi Pico sees signal voltages in the range 0 to 3.3V while the keyboard sees voltages 0 to 5V.

There are many ways that this can be accomplished but the following circuit is a simple and low cost solution:



The recommended MOSFET is a TN0702N3-G or ZVNL110A however the 2N7000 has been tested and also works well.

Some keyboards will run OK on 3.3V (despite the PS2 requirement for 5V) and in that case the keyboard can be powered by the Raspberry Pi Pico 3V3 output and directly connected as follows:



By default the keyboard configuration will be assumed to be the standard US layout. However the OPTION KEYBOARD command can be used to configure layouts for other countries.

The syntax of the command is:

```
OPTION KEYBOARD language [,capslock][,numlock][,repeatstart] [,repeatrate]
```

Where 'language' is a two character code such as US for the standard keyboard used in the USA, Australia and New Zealand. Other keyboard layouts that can be specified are United Kingdom (UK), French (FR), German (GR), Belgium (BE), Italian (IT) or Spanish (ES). See the OPTION KEYBOARD command for details of the optional parameters. This command must be entered at the command prompt (not in a program) and will be remembered when the PicoMiteVGA is restarted.

Note that the non US layouts map some of the special keys present on these keyboards but the corresponding special character will not display as they are not part the standard PicoMiteVGA fonts (another character will be used instead).

Connecting an SD Card

The PicoMiteVGA has full support for SD cards. This includes opening files for reading, writing or random access and loading and saving programs. The firmware will work with cards up to 32 GB formatted in FAT16 or FAT32 and the files created can also be read/written on personal computers running Windows, Linux or the Mac operating system. The PicoMiteVGA uses the SPI protocol to talk to the card and this is not influenced by the card type. So all types (Class 4, 10, UHS-1 etc) are supported.

It should be noted that an SD card is not necessary for normal operation as the PicoMiteVGA has eight on-board flash memory slots which can be used to save and run up to eight individual programs without requiring an SD card.

The SD card interface needs to be specifically configured before it can be used. This is done with the OPTION SDCARD command as followd:

```
OPTION SDCARD CS, CLK, MOSI, MISO
```

Where:

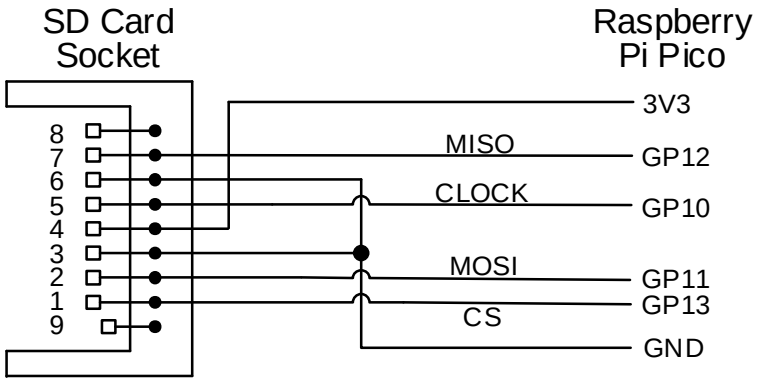
CS	is the I/O pin used for the Chip Select signal.
CLK	is the I/O pin used for the SPI Clock signal.
MOSI	is the I/O pin used for the Master Out Slave In (MISO) signal.
MISO	is the I/O pin used for the Master In Slave Out (MOSI) signal.
	The PicoMiteVGA is the master, and the SD card is the slave.

Any free pins can be used (see the section *PicoMiteVGA Hardware*) but this example uses pins GP13, GP10, GP11 and GP12 for Chip Select, Clock, MOSI and MISO.

```
OPTION SDCARD GP13, GP10, GP11, GP12
```

This command must be entered at the command prompt (not in a program) and will be remembered when the PicoMiteVGA is restarted. It will cause an automatic restart which has the side effect of disconnecting the USB console interface - which will then need to be reconnected.

The circuit diagram for connecting the SD card socket using these pin allocations is illustrated below.



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

When the PicoMiteVGA is booted MMBasic will automatically initialise the SD card interface. The allocated pins will then not be available to BASIC programs (i.e. they will be reserved). To verify that the interface is working you can insert an SD Card and list its contents with the command `FILES`.

To make it easy to physically connect an SD card you can purchase the SD socket mounted on a PCB with the I/O pins brought out to connections with a 0.1" pitch.

For example: SD/MMC Card Breakout from SparkFun.

Other, more complicated, examples include:

Jaycar XC4836

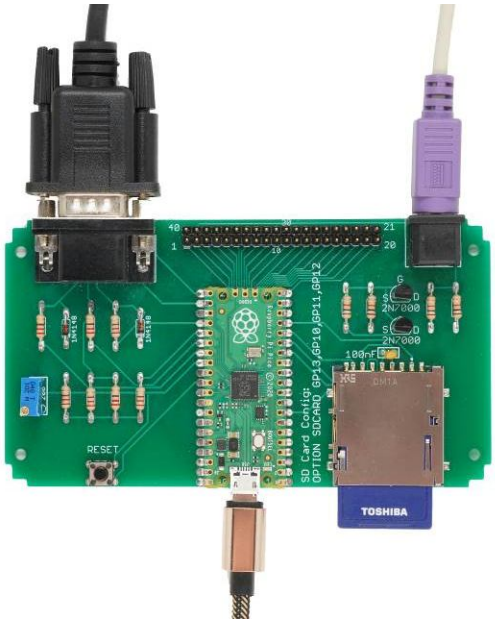
Altronics Z6353

Construction Pack

The construction pack is a ZIP file that contains the design for PCBs that support the PicoMiteVGA. These includes the parts list, documentation and the PCB Gerbers which can be used by any PCB fabricator to make the PCB.

The construction pack can be downloaded from: https://geoffg.net/picomitevga.html (at the bottom of the page).

Design #1



This is an easy to assemble PCB that implements the VGA output, PS2 keyboard interface and the SD card socket as described in the previous pages of this manual.

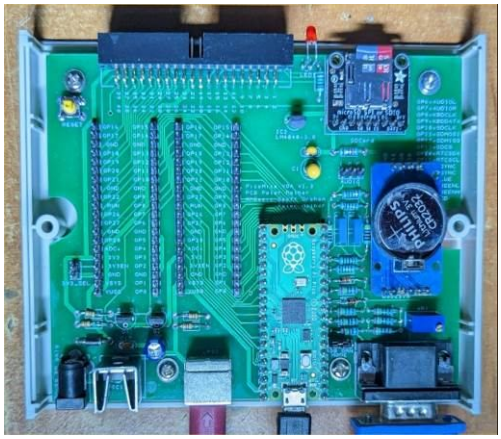
It uses common thru-hole components and can be assembled in under an hour.

All 40 pins on the Raspberry Pi Pico are routed to a 40-way connector on the rear of the PCB in the same configuration as that used by the Pico. This makes it easy to connect external devices as you can consult the pin out diagram in this manual and then select the corresponding pins on the 40-way connector.

Allowing for the I/O pins reserved for the VGA output, keyboard and SD card there are 14 I/O pins available for external circuitry.

The board is sized to fit in an Altronics snap-together case 130 x 75 x 28mm (part number H0376).

Design #2



This design by Peter Mather will fit in the same box as a Colour Maximite or Colour Maximite 2. It has support for:

- The Raspberry Pi Pico, keyboard, VGA output and SD card socket.
- A real time clock (RTC).
- Audio output.
- 40-pin connector for I/O.
- On board 3.3V power supply for noise free analogue input and audio output.
- Sockets for two Raspberry Pi Pico expansion boards.

The design uses through-hole components and cheap and widely available modules for the SDcard and the DS3231 RTC.

Design #3

A compact PicoMiteVGA design by Mick Ames.

It provides 16-colour VGA output, PS/2 keyboard input, SD card socket and sound output.

Modules (purchased from eBay) are used for the keyboard level shifter, RTC, SD card, etc.

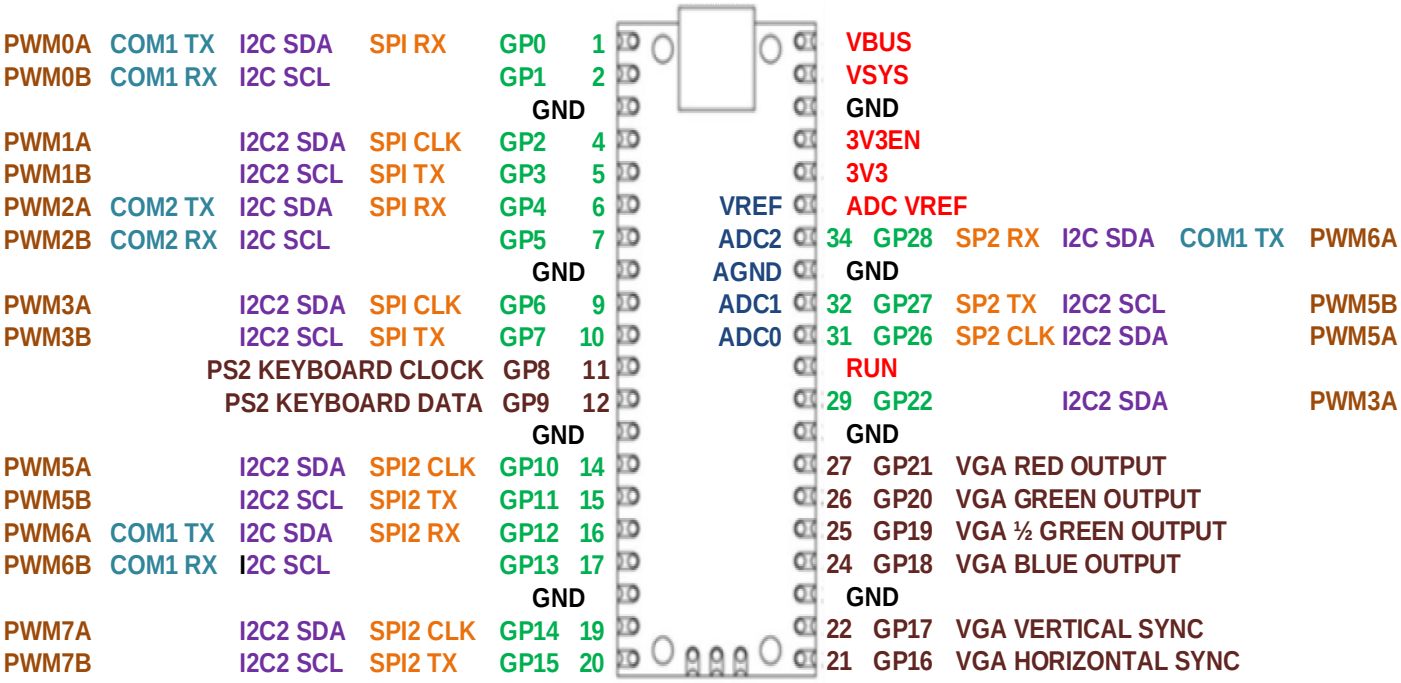
Design #4

PicoGAME NES by Mick Ames. The PicoGAME (NES version) is simply a PCB which allows NES controllers or a PC joystick to be connected to a PicoMite VGA.

The usual PicoMiteVGA facilities are available: 16-colour VGA display, PS/2 keyboard input and sound output.

PicoMiteVGA Hardware

This diagram shows the possible uses within MMBasic for each I/O pin on the Raspberry Pi Pico:



The notation is as follows:

GP0 to GP28	Can be used for digital input or output.
COM1, COM2	Can be used for asynchronous serial I/O (UART0 and UART1 pins on the Pico datasheet).
I2C, I2C2	Can be used for I^2C communications (I2C0 and I2C1 pins on the Pico datasheet).
SPI, SPI2	Can be used for SPI I/O (see Appendix D). (SPI0 and SPI1 pins on the Pico datasheet).
PWMnx	Can be used for PWM output (see the PWM command).
GND	Common ground.
VBUS	5V supply directly from the USB port.
VSYS	5V supply used by the SMPS to provide 3.3V. This can be used as a 5V output or input.
3V3EN	Enable 3.3V regulator (low = off, high = enabled).
RUN	Reset pin, low will hold the PicoMiteVGA in reset.
ADCn	These pins can be used to measure voltage (analog input).
ADC VREF	Reference voltage for voltage measurement.
AGND	Analog ground.

All pins can be used for digital input or output however they are limited to a maximum voltage of 3.6V. This means that level shifting will be required if they are to be used with devices operating at 5V or higher.

Within the MMBasic program I/O pins can be referred to using the physical pin number (i.e. 1 to 40) or the GP number (i.e. GP0 to GP28). For example, the following refer to the same pin and operate identically:

```
SETPIN 32, DOUT
```

and

```
SETPIN GP27, DOUT
```

On the PicoMiteVGA on-chip functions such as the SPI and I2C interfaces are not allocated to fixed pins, unlike (for example) the Micromite. The PicoMiteVGA makes extensive use of the SETPIN command, not only to configure I/O pins but also to configure the pins used for interfaces such as serial, SPI, I^2C, etc.

Pins must be allocated according to this drawing. For example, the SPI TX can be allocated to pins GP3, GP7 or GP19 but it cannot be allocated to pin GP15 which can only be allocated to the SPI2 channel. Allocations don't have to be in the same "block" so you could, for example, allocate SPI2 TX to pin GP15 and SPI2 RX to pin GP28.

Pins that are not exposed on the Raspberry Pi Pico can still be accessed using MMBasic via a pseudo pin number or their GPn number. This allows MMBasic to be used on other modules that use the RP2040 processor. These hidden pins are Pin 41 or GP23, Pin 42 or GP24, Pin 43 or GP25 and Pin 44 or GP29.

On the Raspberry Pi Pico these pins are used for internal functions as follows:

- Pin 41 or GP23 is a digital output set to the value of OPTION POWER. (ON=PWM, OFF=PFM).
- Pin 42 or GP24 is a digital input, which is high when VBUS is present.
- Pin 43 or GP25 is also PWM4B. It is an output connected to the on-board LED.
- Pin 44 or GP29 is also ADC3 which is an analog input reading ⅓ of VSYS.

The PicMiteVGA firmware will automatically reserve pins 21, 22, 24, 25, 26 and 27 (GP16, GP17, GP18, GP19, GP20 and GP21) for the VGA output and 11 and 12 (GP8 and GP9) for the PS2 keyboard.

I/O Pin Limits

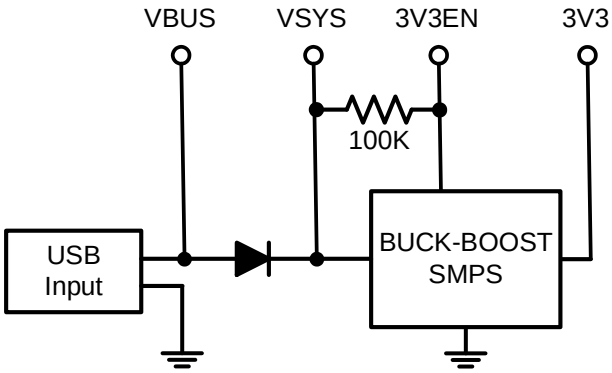
The maximum voltage that can be applied to any I/O pin is 3.6V.

As outputs all I/O pins can individually source or sink a maximum of 12mA. At this load the output voltage will sag to about 2.3V. A more practical load is 5mA where the output voltage would typically be 3V. To drive a red LED at 5mA the recommended resistor is 220Ω. Other colours may require a different value.

The maximum total I/O current load for the entire chip is 50mA.

Power Supply

The Raspberry Pi Pico has a flexible power system.



The input voltage from either the USB or VBUS inputs is connected through a Schottky diode to the buck-boost SMPS (Switch Mode Power Supply) which has an output of 3.3V. The SMPS will accommodate input voltages from 1.8V to 5.5V.

External circuitry can be powered by VBUS (normally 5V) or by the 3V3 (3.3V) output which can source up to 300mA.

To minimize power supply noise it is possible to ground 3V3EN to turn off the SMPS. When shutdown the converter will stop switching, internal control circuitry will be turned off, and the load disconnected from the input. You can then power the board via a 3.3V linear regulator feeding into the 3V3 pin. Note that in this case you will still need a 5V supply for the keyboard.

Clock Speed

By default the clock speed for the PicoMiteVGA is 126MHz which is near the recommended maximum for the Raspberry Pi Pico. However, by using the OPTION CPUSPEED at the command prompt, the CPU speed can be changed to 252MHz. This is regarded as overclocking the processor but most Raspberry Pi Picos run at this speed with no issues. This option is saved and will be reapplied on power up..

If the Raspberry Pi Pico fails to restart at its new clock speed you can reset it to its factory default condition by loading this firmware file onto the Pico: https://geoffg.net/Downloads/picomite/Clear_Flash.uf2. The procedure to do this is same as loading any other firmware.

Using MMBasic

A Simple Program

To enter a program, you can use the EDIT command which is described later in this manual. However, for the moment, all that you need to know is that anything that you type will be inserted at the cursor, the arrow keys will move the cursor and backspace will delete the character before the cursor.

To get a quick feel for how the PicoMiteVGA works, try this sequence:

- At the command prompt type `EDIT` followed by the ENTER key.
- The editor should start up and you can enter this line: `PRINT "Hello World"`
- Press the F1 key on the keyboard. This tells the editor to save your program and exit to the command prompt.
- At the command prompt type `RUN` followed by the ENTER key.
- You should see the message: `Hello World`

Congratulations. You have just written and run your first program on the PicoMiteVGA . If you type EDIT again you will be back in the editor where you can change or add to your program.

Tutorial on Programming in the BASIC Language

If you are new to the BASIC programming language now would be a good time to turn to Appendix G (*Programming in BASIC - A Tutorial*) at the rear of this manual. This is a comprehensive tutorial on the language which will take you through the fundamentals in an easy to read format with lots of examples.

Commands and Program Input

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

To enter a program the easiest method is to use the EDIT command. This will invoke the full screen program editor which is built into MMBasic. It includes advanced features such as search and copy, cut and paste to and from a clipboard.

You could also compose the program on your desktop computer using something like Notepad and then transfer it to the PicoMiteVGA via the XModem protocol (see the XMODEM command) or by streaming it up the console serial link over USB (see the AUTOSAVE command).

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

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

Program Structure

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

A program consists of a number of statements or commands, each of which will cause the BASIC interpreter to do something (the words statement and command generally mean the same and are used interchangeably). Normally each statement is on its own line but you can have multiple statements in the one line separated by the colon character (:). For example.

```
A = 24.6 : PRINT A
```

Each line can start with a line number. Line numbers were mandatory in the early BASIC interpreters however modern implementations (such as MMBasic) do not need them. You can still use them if you wish but they

have no benefit and generally just clutter up your programs. This is an example of a program that uses line numbers:

```
50 A = 24.6
60 PRINT A
```

A line can also start with a label which can be used as the target for a program jump using the GOTO command. For example (the label name is `JmpBack`):

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

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

Editing the Command Line

When entering a line at the command prompt the line can be edited using the left and right arrow keys to move along the line, the Delete key to delete a character and the Insert key to switch between insert and overwrite. At any point the Enter key will send the line to MMBasic which will execute it.

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

Shortcut Keys

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

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

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

Saving Programs

On the PicoMiteVGA the program is held in flash memory and is run from there. When a program is edited via EDIT or loaded it will be saved there. Flash memory is non-volatile so the program will not be lost if the power is lost or the processor is reset. The maximum program size is 108KB.

Programs can also be saved to an SD card (if configured) or they can be saved to one of eight numbered memory locations (or 'slots') in the flash memory. These can be used to save previous versions of the program (in case you need to revert to an earlier version) or they can be used to save completely different programs which can be quickly loaded into program memory and run.

In addition MMBasic will allow a BASIC program to load and run another program saved to a numbered flash location while retaining all the variables and settings of the original program – this is called chaining and allows for a much larger program to be run than the amount of program memory would normally allow.

To manage these numbered locations in flash you can use the following commands (note that in the following *n* is a number from 1 to 8):

FLASH SAVE *n*	Save the main program memory to flash location *n*.
FLASH LOAD *n*	Load a program from flash location *n* into the main program memory.
FLASH RUN *n*	Run a program from flash location *n*. This clears all variables but does not erase or change the program held in the main program memory.
FLASH LIST	Display a list of all flash locations including the first line of each program.
FLASH LIST n [,all]	Lists the program held in location n. Use `FLASH LIST n,ALL` to list without page breaks
FLASH ERASE *n*	Erase flash location *n*.

FLASH ERASE ALL	Erase all flash locations.
FLASH CHAIN *n*	Run a program from flash location *n*, leaving all variables intact. As with FLASH RUN this command but does not erase or change the program held in the main program memory.
FLASH OVERWRITE *n*	Erase flash location *n* then save the current program that location.

In addition the command OPTION AUTORUN can be used to specify a flash program location to be set running when power is applied or the CPU restarted. This option can also used without specifying a flash location and in that case MMBasic will automatically load and run the program that is in the program memory.

It is recommended that you include a comment describing the program as the first line of the program. This will then be displayed by the FLASH LIST command and will help identify the program in the listing.

All BASIC programs saved to flash may be erased if you upgrade (or downgrade) the PicoMiteVGA firmware. So make sure that you backup these first.

Interrupting A Running Program

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

Setting Options

Many options can be set by using commands that start with the keyword OPTION. They are listed in their own section of this manual. For example, you can change the CPU clock speed with the command:

```
OPTION CPUSPEED speed
```

Saved Variables

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

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

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

PIN Security

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

Because an intruder cannot reach the command prompt they cannot list or copy a program, they cannot change the program or change any aspect of MMBasic or the PicoMiteVGA firmware. Once set the PIN can only be removed by providing the correct PIN as set in the first place.

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

If the PIN has been lost or forgotten you can erase everything by resetting the Raspberry Pi Pico to its factory fresh state then reloading the PicoMiteVGA firmware. Resetting the Raspberry Pi Pico can be done by loading this firmware file onto the Pico: https://geoffg.net/Downloads/picomite/Clear_Flash.uf2. The procedure to do this is same as loading any other firmware.

MM.STARTUP

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

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

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

This would cause the internal clock within MMBasic to be set to the current time on every power up or reset.

After the code in MM.STARTUP has been run MMBasic will continue with running the rest of the program in program memory. If there is no other code MMBasic will return to the command prompt.

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

MM.PROMPT

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

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

As an example the following will display a custom prompt:

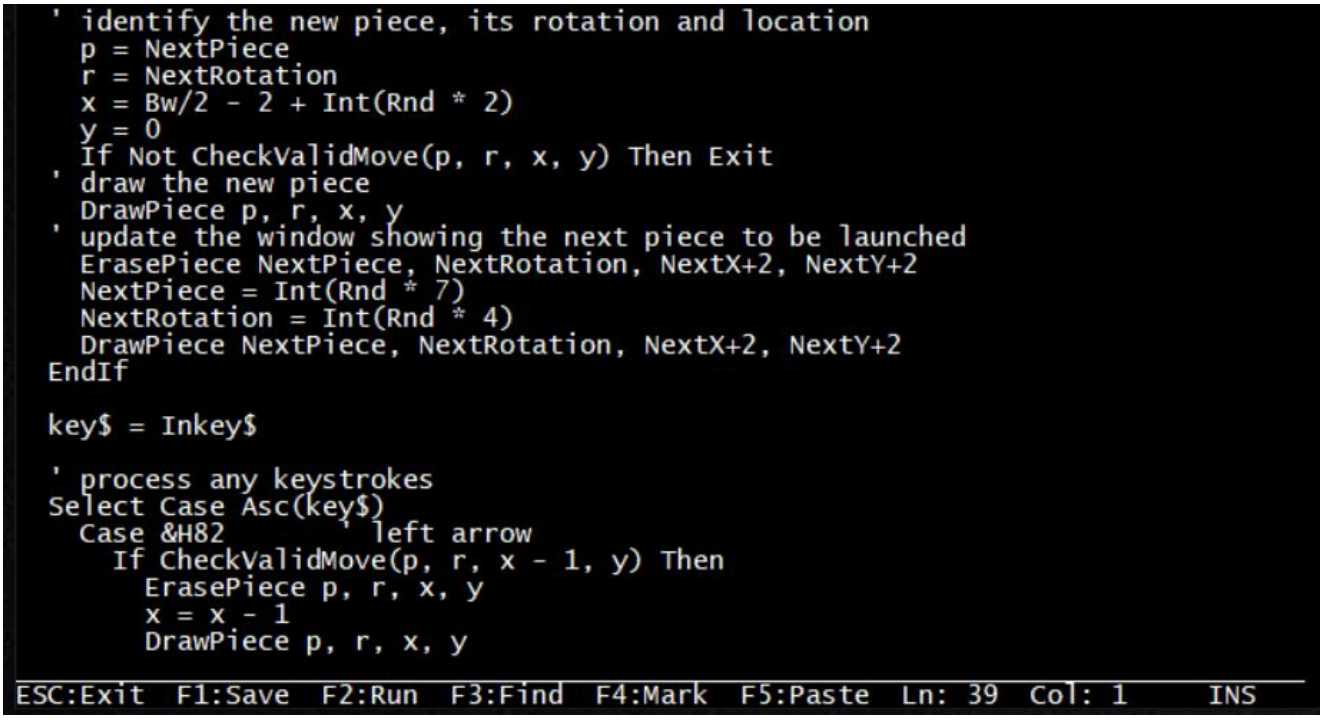
```
SUB MM.PROMPT
  PRINT TIME$ "> ";
END SUB
```

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

Full Screen Editor

An important productivity feature is the built-in full screen editor.

When running it looks like this:



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

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

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

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

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

These keys are:

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

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

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

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

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

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

Colour Coded Editor Display

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

```
OPTION COLOURCODE ON  or  OPTION COLOURCODE OFF
```

The colour coding will apply to both the USB serial console (with an appropriate terminal emulator such as Tera Term) and the VGA display (the latter requires that the colour mode (MODE 2) be set).

This setting is saved in non-volatile memory and automatically applied on start-up.

Graphics Commands and Functions

Colours

In MMBasic colour is specified as a true colour 24 bit number where the top eight bits represent the intensity of the red colour, the middle eight bits the green intensity and the bottom eight bits the blue. This is for compatiblity with MMBasic running on other platforms.

The easiest way to generate this number is with the RGB() function which has the form:

```
RGB(red, green, blue)
```

Where red, green and blue are numbers in the range of 0 to 255 representing the intensity of each colour.

The RGB() function also supports a shortcut where you can specify common colours by naming them. For example, RGB(red) or RGB(cyan). The colours that can be named using the shortcut form are white, yellow, lilac, brown, fuchsia, rust, magenta, red, cyan, green, cerulean, midgreen, cobalt, myrtle, blue and black.

The PicoMiteVGA supports 2 “colours” in monochrome mode and 16 in colour mode. In monochrome mode any colour that is not zero will result in a lit pixel. In colour mode the firmware will map 24-bit colours onto one of the 16 available colours in the 121 format (1 bit for red, 2 bits for green and 1 bit for blue).

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

The COLOUR command takes the format:

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

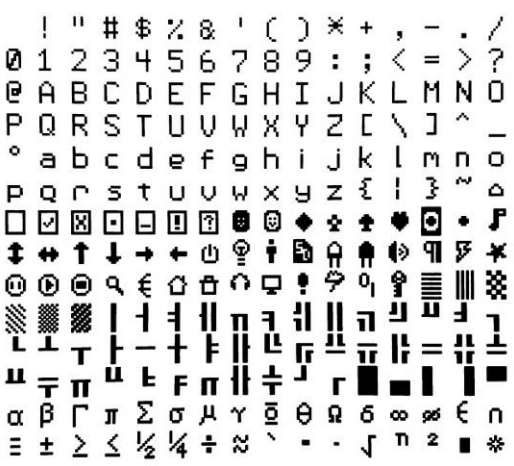
Fonts

There are eight built in fonts. These are:

Font Number	Size (width x height)	Character Set	Description
1	8 x 12	All 95 ASCII characters plus 7F to FF (hex)	Standard font. Default for the monochrome mode.
2	12 x 20	All 95 ASCII characters	Medium sized font
3	16 x 16	All 95 ASCII characters	A larger bold font. This can be used with the TILE command to colour characters in monochrome mode.
4	10x16	All 95 ASCII characters plus 7F to FF (hex)	A font with extended graphic characters. Suitable for high resolution displays
5	24 x 32	All 95 ASCII characters	Extra large font, very clear
6	32 x 50	0 to 9 plus some symbols	Numbers plus decimal point, positive, negative, equals, degree and colon symbols. Very clear.
7	6 x 8	All 95 ASCII characters	A small font useful when low resolutions are used. Default for the colour mode.
8	6 x 4	All 95 ASCII characters	An even smaller font.

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

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



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

The format of an embedded font is:

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

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

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

Read Only Variables

There are four read only variables which provide useful information about the display currently connected:

- MM. HRES
  Returns the width of the display (the X axis) in pixels.
- MM. VRES
  Returns the height of the display (the Y axis) in pixels.
- MM.FONTHEIGHT
  Returns the height of the current default font (in pixels). All characters in a font have the same height.
- MM.FONTWIDTH
  Returns the width of a character in the current font (in pixels). All characters have the same width.

Drawing Commands

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. The top left corner of the screen has the coordinates X = 0 and Y = 0 and the values increase as you move down and to the right of the screen.

There are nine Graphics Commands and Functions that you can use within MMBasic programs on the PicoMiteVGA to draw on the VGA screen. Most of these have optional parameters. You can completely leave these off the end of a command or you can use two commas in sequence to indicate a missing parameter. For example, the fifth parameter of the LINE command is optional so you can use this format:

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

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

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

The drawing commands are:

- `CLS` *`C`*
  Clears the screen to the colour C. If C is not specified the current default background colour will be used.
- `PIXEL X, Y,` *`C`*
  Illuminates a pixel. If C is not specified the current default foreground colour will be used.
- `LINE X1, Y1, X2, Y2,` *`LW, C`*
  Draws a line starting at X1 and Y1 and ending at X2 and Y2.
  LW is the line's width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal.
- `BOX X, Y, W, H,` *`LW, C, FILL`*
  Draws a box starting at X and Y which is W pixels wide and H pixels high.
  LW is the width of the sides of the box and can be zero. It defaults to 1.
- `RBOX X, Y, W, H,` *`R, C, FILL`*
  Draws a box with rounded corners starting at X and Y which is W pixels wide and H pixels high.
  R is the radius of the corners of the box. It defaults to 10.

- `CIRCLE X, Y, R, LW, A, C, FILL`
  Draws a circle with X and Y as the centre and a radius R. LW is the width of the line used for the circumference and can be zero (defaults to 1). A is the aspect ratio which is a floating point number and defaults to 1. For example, an aspect of 0.5 will draw an oval where the width is half the height.

- `TEXT X, Y, STRING, ALIGNMENT, FONT, SCALE, C, BC`
  Displays a string starting at X and Y. ALIGNMENT is 0, 1 or 2 characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER or RIGHT aligned text and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE or BOTTOM aligned text. The default alignment is left/top. An additional code letter can be used to rotate the text (see below for the details). FONT and SCALE are optional and default to that set by the FONT command. C is the drawing colour and BC is the background colour. They are optional and default to that set by the COLOUR command.

- `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 as displayed on the VGA screen and default to 8x8. SCALE, C and BC are the same as for the TEXT command. The bitmap can be an integer or a string variable or constant and is drawn using the first byte as the first bits of the top line (bit 7 first, then bit 6, etc) followed by the next byte, etc. When the top line has been filled the next line of the displayed bitmap will start with the next bit in the integer or string.

- `POLYGON n, xarray%(), yarray%() [, bordercolour] [, fillcolour]`
  Draws a filled or outline polygon with n xy-coordinate pairs in xarray%() and yarray%(). If 'fillcolour' is omitted then just the polygon outline is drawn. If 'bordercolour' is omitted then it will default to the current default foreground colour.

Rotated Text

As described above the alignment of the text in the TEXT command can be specified by using one or two characters in a string expression for the third parameter of the command. In this string you can also specify a third character to indicate the rotation of the text. This character can be one of:

N for normal orientation

V for vertical text with each character under the previous running from top to bottom.

I the text will be inverted (i.e. upside down)

U the text will be rotated counter clockwise by 90º

D the text will be rotated clockwise by 90º

As an example, the following will display the text " VGA Display" vertically down the left hand margin of the display panel and centred vertically:

```
TEXT 0, 250, "VGA Display", "LMV", 5
```

Positioning is relative to the top left corner of the character when viewed normally so inverted 100,100 will have the top left pixel of the first character at 100,100 and the text will then be above y=101 and to the left of x=101. Similarly "R" in the alignment string is viewed from the perspective of the character in whatever orientation it is in (not the screen).

Transparent Text

You can use -1 for the background colour. This means that the text is drawn over the background with the background image showing through the gaps in the letters.

BLIT Command

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

The available commands are:

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

#b is the buffer number in the range of 1 to 8. x and y are the coordinates of the top left corner and w and h are the width and height of the image. READ will copy the display image to the buffer, WRITE will copy the buffer to the display and CLOSE will free up the buffer and reclaim the memory used. LOAD will load an image file into the buffer.

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

BLIT x1, y1, x2, y2, w, h

This will copy a portion of the image at x1/y1 to the location x2/y2. w and h specify the width and height of the image to be copied. The source and destination areas can overlap and the BLIT command will perform the copy correctly. This form of the BLIT command is particularly useful for creating graphs that can scroll horizontally or vertically as new data is added.

Load Image

As described in the *SD Card Support* section the LOAD IMAGE and LOAD JPG commands can be used to load an image from the SD card and display it on the VGA screen. This can be used to draw a logo or add an ornate background to the graphics drawn on the display.

Example

As a simple example of the graphics commands the following program will draw a simple digital clock on the VGA monitor. The program will terminate and return to the command prompt when a key is pressed on the console's keyboard.

This program requires that colour has been enabled (ie, MODE 2).

```
CONST DBlue = RGB(0, 0, 128)              ' A dark blue colour
COLOUR RGB(GREEN), RGB(BLACK)             ' Set the default colours
FONT 1, 3                                 ' Set the default font

CLS                                       ' clear the screen
BOX 0, 0, MM.HRes-1, MM.VRes/2, 3, RGB(RED), DBlue

DO
  TEXT MM.HRes/2, MM.VRes/4, TIME$, "CM", 1, 4, RGB(CYAN), DBlue
  TEXT MM.HRes/2, MM.VRes*3/4, DATE$, "CM"
  IF INKEY$ <> "" THEN END                 ' abort on any keypress
LOOP
```

This program starts by defining a constant with a value corresponding to a dark blue colour and then sets the defaults for the colours and the font. It then draws a box with red walls and a dark blue interior.

Following this the program enters a continuous loop where it performs three functions:

1. Displays the current time inside the previously drawn box. The string is drawn centred both horizontally and vertically in the middle of the box. Note that the TEXT command overrides both the default font and colours to set its own parameters.
2. Draws the date centred in the lower half of the screen. In this case the TEXT command uses the default font and colours previously set.
3. Checks for a keypress on the console. In that case the program will terminate.

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

Using the I/O pins

The Raspberry Pi Pico has 26 input/output pins which can be controlled from within the BASIC program with 3 of these supporting a high speed ADC (Analog to Digital Converter).

An I/O pin is referred to by its pin number and this can be the number (e.g. 2) or its GP number (e.g. GP1).

Digital Inputs

A digital input is the simplest type of input configuration. If the input voltage is higher than 2.5V the logic level will be true (numeric value of 1) and anything below 0.65V will be false (numeric value of 0). The inputs use a Schmitt trigger input so anything in between these levels will retain the previous logic level. All pins are limited to a maximum voltage of 3.6V. This means that resistor divider will be required if they are used with input voltages greater than that.

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 GP4, DIN
IF PIN(GP4) = 1 THEN PRINT "High"
```

The SETPIN command configures pin GP4 as a digital input and the PIN() function will return the value of that pin (the number 1 if the pin is high). The IF command will then execute the command after the THEN statement if the input was high. If the input pin was low the program would just continue with the next line in the program.

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

Analog Inputs

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

```
> SETPIN 31, AIN
> PRINT PIN(31)
 2.345
>
```

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

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

The ADC commands provide an alternate method of recording analog inputs and are intended for high speed recording of many readings into an array.

Counting Inputs

Any four pins can be used as counting inputs to measure frequency, period or just count pulses on the input. The pins used for this function can be configured using the OPTION COUNT command but, if not changed, will default to GP6, GP7, GP8 and GP9.

As an example, the following will print the frequency of the signal on pin GP7:

```
> SETPIN GP7, FIN
> PRINT PIN(GP7)
110374
>
```

In this case the frequency is 110.374 kHz.

By default the gate time is one second which is the length of time that MMBasic will use to count the number of cycles on the input and this means that the reading is updated once a second with a resolution of 1 Hz. By specifying a third argument to the SETPIN command it is possible to specify an alternative gate time between 10 ms and 100000 ms. Shorter times will result in the readings being updated more frequently but the value

returned will have a lower resolution. The PIN() function will always scale the returned number as the frequency in Hz regardless of the gate time used.

For example, the following will set the gate time to 10ms with a corresponding loss of resolution:

```
> SETPIN GP7, FIN, 10
> PRINT PIN(GP7)
110300
>
```

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

The count pins can also count the number of pulses on their input. When a pin is configured as a counter (for example, `SETPIN 7,CIN`) the counter will be reset to zero and PicoMiteVGA will then count every transition from a low to high voltage. The counter can be reset to zero again by executing `PIN(7) = 0`.

Digital Outputs

All I/O pins can be configured as a digital output. This means that when an output pin is set to logic low it will pull its output to zero and when set high it will pull its output to 3.3V. In MMBasic this is done with the PIN command. For example `PIN(GP15) = 0` will set pin GP15 to low while `PIN(GP15) = 1` will set it high.

The "OC" option on the SETPIN command makes the output pin open collector. This means that the output driver will pull the output low (to zero volts) when the output is set to a logic low but will go to a high impedance state when set to logic high. Note that the maximum voltage on a pin is 3.6V so open collector outputs cannot be used to drive higher voltage logic (i.e. 5V).

Pulse Width Modulation

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

The PWM outputs consists of up to 8 channels (numbered 0 to 7) with each channel having two outputs (A and B). For each channel the frequency can be selected and for each output a different duty cycle can be set.

Up to 16 pins can be configured as PWM outputs using the SETPIN command.

Communications Interfaces (Serial, SPI and I^{2}C)

These are described in the appendices at the rear of this manual. Before these interfaces can be used the pins that are to be used for the relevant signals must be configured using the SETPIN command.

Interrupts

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

When an interrupt occurs MMBasic will execute a special subroutine and when finished return to the main program. The main program is completely unaware of the interrupt and will carry on as normal.

Any I/O pin that can be used as a digital input can be configured to generate an interrupt using the SETPIN command with up to ten interrupts active at any one time. Interrupts can be set up to occur on a rising or falling digital input signal (or both) and will cause an immediate branch to the specified user defined subroutine. The target can be the same or different for each interrupt. Return from an interrupt is via the END SUB or EXIT SUB commands. Note that no parameters can be passed to the subroutine however within the interrupt calls to other subroutines and functions are allowed.

If two or more interrupts occur at the same time they will be processed in order of the interrupts as defined below. During the processing of an interrupt all other interrupts are disabled until the interrupt subroutine returns. During an interrupt (and at all times) the value of the interrupt pin can be accessed using the PIN() function.

Interrupts can occur at any time but they are disabled during INPUT statements. Also interrupts are not recognised during some long hardware related operations (e.g. the TEMPR() function and SD access commands) although they will be recognised if they are still present when the operation has finished. When using interrupts the main program is completely unaffected by the interrupt activity unless a variable used by the main program is changed during the interrupt.

Because interrupts run in the background they can cause difficult to diagnose bugs. Keep in mind the following factors when using interrupts:

- Interrupts are only checked by MMBasic at the completion of each command, and they are not latched by the hardware. This means that an interrupt that lasts for a short time can be missed, especially when the program is executing commands that take some time to execute. Most commands will execute in under 15µs however some commands such as the TEMPR() function can take up to 200ms so it is possible for an interrupt to occur and vanish within this window and thus not be recognised.
- When inside an interrupt all other interrupts are blocked so your interrupts should be short and exit as soon as possible. For example, never use PAUSE inside an interrupt. If you have some lengthy processing to do you should simply set a flag and immediately exit the interrupt, then your main program loop can detect the flag and do whatever is required.
- The subroutine that the interrupt calls (and any other subroutines or functions called by it) should always be exclusive to the interrupt. If you must call a subroutine that is also used by an interrupt you must disable the interrupt first (you can reinstate it after you have finished with the subroutine).
- Remember to disable an interrupt when you have finished needing it – background interrupts can cause strange and non-intuitive bugs.

In addition to interrupts generated by the change in state of an I/O pin, an interrupt can also be generated by other sections of MMBasic including timers and communications ports and the above notes also apply to them.

The list of all these interrupts (in high to low priority ranking) is:

1. ON KEY individual
2. ON KEY general
3. ADC completion
4. I2C Slave Rx
5. I2C Slave Tx
6. I2C2 Slave Rx
7. I2C2 Slave Tx
8. WAV Finished
9. COM1: Serial Port
10. COM2: Serial Port
11. IR Receive
12. Keypad
13. Interrupt command/CSub Interrupt
14. I/O Pin Interrupts in order of definition
15. Tick Interrupts (1 to 4 in that order)

As an example: If an ON KEY interrupt occurred at the same time as a COM1: interrupt the ON KEY interrupt subroutine would be executed first and then, when the interrupt subroutine finished, the COM1: interrupt subroutine would then be executed.

Sound Output

The PicoMiteVGA can play stereo WAV files located on the SD card or generate precise sine waves using the PLAY command. Note that the switching power regulator on the Raspberry Pi Pico will cause some interference with the output. This can be reduced by disabling the regulator and powering the module via an external linear regulator – see the section *PicoMiteVGA Hardware.*

Allocating the Output Pins

The audio is created using PWM outputs so before the PLAY commands can be used the PWM output pins to be used must be allocated as audio outputs.

This is done using the OPTION AUDIO command as follows:

```
OPTION AUDIO PWM-A-PIN, PWM-B-PIN
```

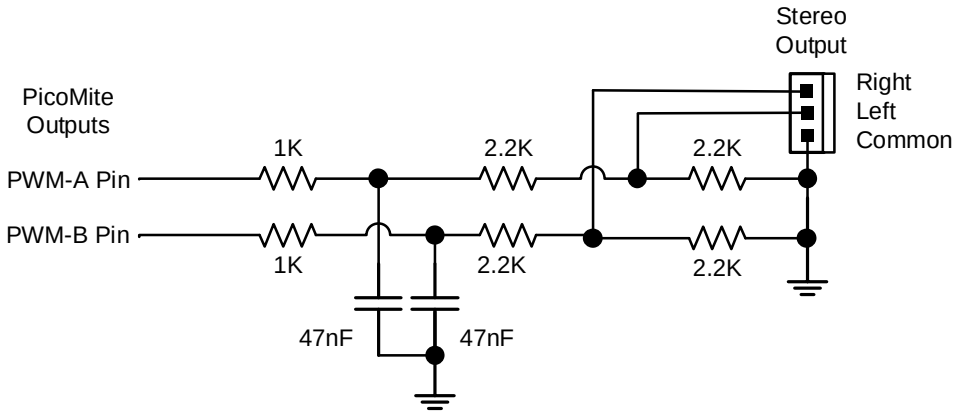
This command should be entered at the command prompt and will be saved, so it only needs to be run once. Both pins must be on the same PWM channel with PWM-A the left audio channel and PWM-B the right.

For example:

```
OPTION AUDIO GP0, GP1
```

Low Pass Filter

The audio signal is superimposed on a square wave as a pulse width modulated (PWM) signal. This means that a low pass filter, as shown below, is required to recover the audio signal. This circuit is intended to drive an amplifier (not headphones or speakers) and relies on capacitor coupling into the following amplifier (most have this) and has an output level of about 1V peak to peak (650mV RMS).



This circuit is suitable for general use; however more sophisticated designs can be used to improve the frequency response and reject more of the carrier frequency. This circuit is also suitable for generating a DC output signal using the PWM commands although, in that case, both 47 nF capacitors should be increased to 4.7 µF to further suppress the PWM carrier frequency.

Playing WAV Files

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

The command is:

```
PLAY WAV file$, interrupt
```

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

Generating Sine Waves

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

The syntax of the command is:

```
PLAY TONE left, right, duration, interrupt
```

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

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

The frequency can be from 1 Hz to 20 KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command. Note that the sine wave is generated by stepping through a lookup table so to reduce the distortion the audio output should be passed through a low pass filter.

Specialised Audio Output

The PLAY SOUND command will generate an output based on a mixture of sine, square, etc waveforms. See the details in the command listing.

Using PLAY

It is important to realise that the PLAY command will generate the audio in the background. This allows a program (for example) to play the sound of a bell while continuing with its control function. Without the background facility the whole BASIC program would freeze while the sound was heard.

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

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

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

Similarly the following program will not work either:

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

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

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

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

This applies to all versions of the PLAY command including PLAY WAV.

Utility Commands

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

`PLAY PAUSE`	Temporarily halt (pause) the currently playing file or tone.
`PLAY RESUME`	Resume playing a file or tone that was previously paused.
`PLAY STOP`	Terminate the playing of the file or tone. The sound output will also be automatically stopped when the program ends.
`PLAY VOLUME L, R`	Set the volume to between 0 and 100 with 100 being the maximum volume. The volume will reset to the maximum level when a program is run. A logarithmic scale is used so that PLAY VOLUME 50,50 should sound half as loud as 100,100.

SD Card Support

The PicoMiteVGA has full support for SD cards. This includes opening files for reading, writing or random access and loading and saving programs. The firmware will work with cards up to 32 GB formatted in FAT16 or FAT32 and the files created can also be read/written on personal computers running Windows, Linux or the Mac operating system.

The MMBasic supports the standard BASIC commands for working with storage systems which are summarised here. Note that:

- Long file/directory names are supported in addition to the old 8.3 format.
- The maximum file/path length is 63 characters.
- Upper/lowercase characters and spaces are allowed although the file system is not case sensitive.
- Directory paths are allowed in file/directory strings. (i.e., OPEN "\dir1\dir2\file.txt" FOR …).
- Either forward or back slashes can be used in paths. E.g. \dir\file.txt is the same as /dir/file.txt.
- The current PicoMiteVGA time is used for file create and last access times.
- Up to ten files can be simultaneously open.
- Except for INPUT, LINE INPUT and PRINT the # in #fnbr is optional and may be omitted.

These are the basic commands for reading and writing data on the SD card.

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

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

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

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

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

Programs can be loaded from or saved to the SD card using these commands.

□ LOAD fname$ [, R]
Load a BASIC program from the SD Card. The optional suffix ",R" will cause the program to be run after it has been loaded (in this case fname$ must be a string constant).

□ RUN fname$
Load a BASIC program from the SD Card and run it. fname$ must be a string constant.

□ SAVE fname$
Save the current program to the SD card.

Images can be loaded from or saved to the SD card using three commands.

□ LOAD IMAGE fname$ [,xleft%] [,ytop%]
Load a BMP file and display it on the VGA screen.

□ LOAD JPG fname$ [,xleft%] [,ytop%]
Load a JPG file and display it on the VGA screen.

□ SAVE IMAGE fname$
Save the current VGA screen image as a BMP file.

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

- FILES [wildcard]
  Search the current directory and list the files/directories found.
- KILL fname$
  Delete a file in the current directory.
- MKDIR dname$
  Make a sub directory in the current directory.
- CHDIR dname$
  Change into to the directory $dname. $dname can also be ".." (dot dot) for up one directory or "\" for the root directory.
- RMDIR dir$
  Remove, or delete, the directory ‘dir$’ on the SD card.
- SEEK #fnbr, pos
  Will position the read/write pointer in a file that has been opened for RANDOM access to the 'pos' byte.
- RENAME fromname$ AS toname$
  Will rename the file fromname$ to have the name toname$.

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

- INPUT$(nbr, #fnbr)
  Will return a string composed of ‘nbr’ characters read from a file previously opened for INPUT with the file number ‘#fnbr’. If less than ‘nbr’ characters are available the function will return with what it has (including an empty string if no characters are available).
- DIR$(fspec, type)
  Will search an SD card for files and return the names of entries found.
- CWD$
  Will return the current working directory.
- EOF(#fnbr)
  Will return true if the file previously opened for INPUT with the file number ‘#fnbr’ is positioned at the end of the file.
- LOC(#fnbr)
  For a file opened as RANDOM this will return the current position of the read/write pointer in the file.
- LOF(#fnbr)
  Will return the current length of the file in bytes.

Listing a File

The contents of a file on the SD card can be displayed on the console with the command LIST *file$*

XModem Transfer

In addition to the standard method of XModem transfer which copies to or from the program memory the PicoMiteVGA can also copy to and from a file on the SD card. The syntax is:

```
XMODEM SEND filename$
```

or

```
XMODEM RECEIVE filename$
```

Where ‘filename$’ is the file to save or send. As is common throughout MMBasic ‘filename$’ can be a string expression, variable or constant. If it is a constant the string must be quoted (eg, XMODEM SEND "PR.BAS") In the case of receiving a file, any file on the SD card with the same name will be overwritten.

Load and Save Image

The LOAD IMAGE command can be used to load a bitmap image from the SD card and display it on the VGA screen. This can be used to draw a logo or add a background on the display. The syntax of the command is:

```
LOAD IMAGE filename$ [, StartX, StartY]
```

Where 'filename$' is the image to load and 'StartX'/'StartY' are the coordinates of the top left corner of the image (these are optional and will default to the top left corner of the display if not specified).

The image must be in BMP format and MMBasic will add ".BMP" to the file name if an extension is not specified. All types of the BMP format are supported including black and white and true colour 24-bit images.

The current image the VGA monitor can be saved to a file using the following command:

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

This will save the image, or part of the image, as a 24-bit true colour BMP file (the extension .BMP) will be added if an extension is not supplied.

Example of Sequential I/O

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

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

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

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

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

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

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

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

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

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

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

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

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

Random File I/O

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

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

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

```
   SEEK #1, 257
```

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

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

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

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

The SPACE$() function is used to add enough spaces to ensure that the data written is an exact length (64 bytes in this example). The semicolon at the end of the print command suppresses the addition of the carriage return and line feed characters which would make the record longer than intended. Two other functions can help when using random file access. The LOC() function will return the current byte position of the read/write pointer and the LOF() function will return the total length of the file in bytes.

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

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

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

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

Special Device Support

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

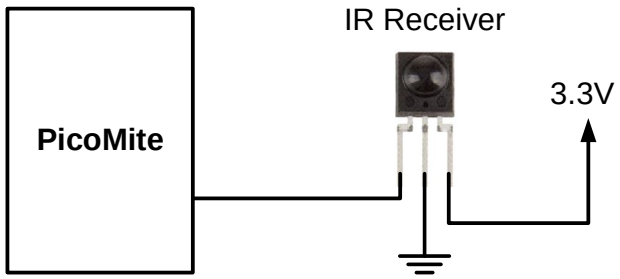
These are:

- Infrared remote control receiver and transmitter
- The DS18B20 temperature sensor and DHT22 temperature/humidity sensor
- Numeric keypads
- Battery backed clock
- Ultrasonic distance sensor
- LCD display modules
- WS2812 RGB LEDs

Infrared Remote Control Decoder

You can easily add a remote control to your project using the IR command. When enabled this function will run in the background and interrupt the running program whenever a key is pressed on the IR remote control.

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



To detect the IR signal you need an IR receiver. NEC remotes use a 38kHz modulation of the IR signal and suitable receivers tuned to this frequency include the Vishay TSOP4838, Jaycar ZD1952 and Altronics Z1611A. Note that the I/O pins on the PicoMiteVGA are only 3.3V tolerant and so the receiver must be powered by a maximum of 3.3V.

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

The IR receiver can be connected to any pin on the PicoMiteVGA . This pin must be configured by the program using the command:

```
SETPIN n, IR
```

where *n* is the I/O pin to use for this function.

To setup the decoder you use the command:

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

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

This is an example of using the IR decoder connected to the GP6 pin:

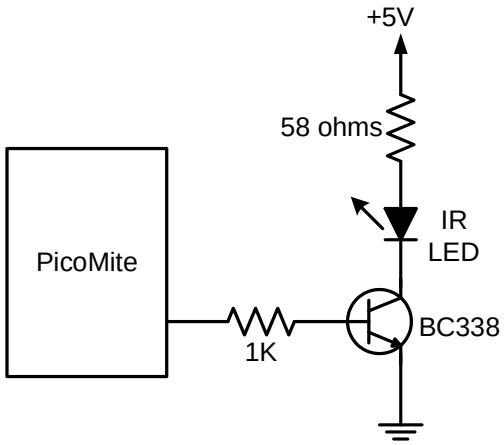
```
SETPIN GP6, IR                      ' define the pin to be used
DIM INTEGER DevCode, KeyCode        ' variables used by the decoder
IR DevCode, KeyCode, IRInt          ' start the IR decoder
DO
  ' < body of the program >
LOOP

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

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

Infrared Remote Control Transmitter

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

The circuit on the right illustrates what is required. The transistor is used to drive the infrared LED because the output capability of the PicoMite is limited. This circuit provides about 50 mA to the LED.



To send a signal you use the command:

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

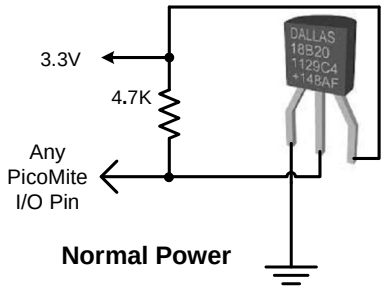
Where pin is the I/O pin used, dev is the device code to send and key is the key code. Any I/O pin can be used and you do not have to set it up beforehand (IR SEND will automatically do that).

The modulation frequency used is 38 kHz and this matches the common IR receivers (described in the previous page) for maximum sensitivity when communicating between two PicoMites or with a Micromite.

Measuring Temperature

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

The DS18B20 can be powered separately by a 3.3V supply or it can operate on parasitic power from the PicoMiteVGA as shown on the right. Multiple sensors can be used but a separate I/O pin and a 4.7K pullup resistor is required for each one.

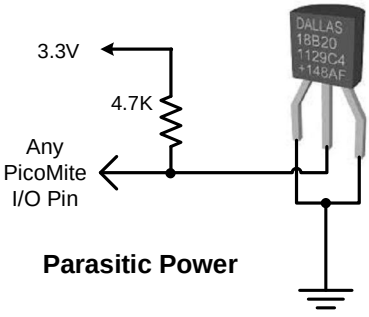


To get the current temperature you just use the TEMPR() function in an expression. For example:

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

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

The returned value is in degrees C with a resolution of 0.25 ºC and is accurate to ±0.5 ºC. If there is an error during the measurement the returned value will be 1000.



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

For example:

```
TEMPR START GP15
< do other tasks >
PRINT "Temperature: " TEMPR(GP15)
```

Real Time Clock Interface

Using the RTC GETTIME command it is easy to get the current time from a PCF8563, DS1307, DS3231 or DS3232 real time clock as well as compatible devices such as the M41T11. These integrated circuits are

popular and cheap, will keep accurate time even with the power removed and can be purchased for $2 to $8 on eBay. Complete modules including the battery can also be purchased on eBay for a little more.

e PCF8563 and DS1307 will keep time to within a minute or two over a month while the DS3231 and DS3232 are particularly precise and will remain accurate to within a minute over a year.

These chips are I^2C devices and should be connected to the I^2C I/O pins of the PicoMiteVGA .

Internal pullup resistors (100KΩ) are applied to the I^2C I/O pins so in many cases external resistors (as shown in the diagram) are not needed.

In order to enable the RTC you first need to allocate the I2C pins to be used using the command:

```
OPTION SYSTEM I2C  SDApin, SCLpin
```

The time used by the RTC must also be set. That is done with the RTC SETTIME command which uses the format RTC SETTIME year, month, day, hour, minute, second. Note that the hour must be in 24 hour format.

For example, the following will set the real time clock to 4PM on the 10th November 2021:

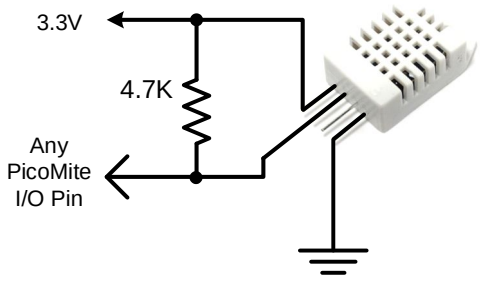
```
RTC SETTIME 2021, 11, 10, 16, 0, 0
```

To get the time you use the RTC GETTIME command which will read the time from the real time clock chip and set the clock inside the PicoMiteVGA . Normally this command will be placed at the beginning of the program or in the subroutine MM.STARTUP so that the time is set on power up. The command OPTION RTC AUTO ENABLE can also be used to set an automatic update.

Measuring Humidity and Temperature

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

The DHT22 must be powered from 3.3V (the maximum voltage for the PicoMiteVGA 's I/O pins) and it should have a pullup resistor on the data line as shown. This is suitable for long cable runs (up to 20 meters) but for short runs the resistor can be omitted as the PicoMiteVGA also provides an internal weak pullup.



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

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

Where 'pin' is the I/O pin to which the sensor is connected. The I/O pin will be automatically configured by MMBasic.

'tVar' is a floating point variable in which the temperature is returned and 'hVar' is a second variable for the humidity. The temperature is returned as degrees C with a resolution of one decimal place (e.g. 23.4) and the humidity is returned as a percentage relative humidity (e.g. 54.3).

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

For example:

```
DIM FLOAT temp, humidity
BITBANG HUMID GP15, temp, humidity
PRINT "The temperature is" temp " and the humidity is" humidity
```

Measuring Distance

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

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

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

On the PicoMiteVGA you use the DISTANCE function as follows:

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

The value returned is the distance in centimetres to the target.

Where trig is the I/O pin connected to the "trig" input of the sensor and echo is the pin connected the "echo" output of the sensor. You can also use 3-pin devices and in that case only one pin number is specified. The maximum voltage on the PicoMiteVGA 's I/O pins is 3.3V so a resistor divider will be required to interface the PicoMiteVGA to the echo pin of the sensor (which operates on 5V).

LCD Display

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

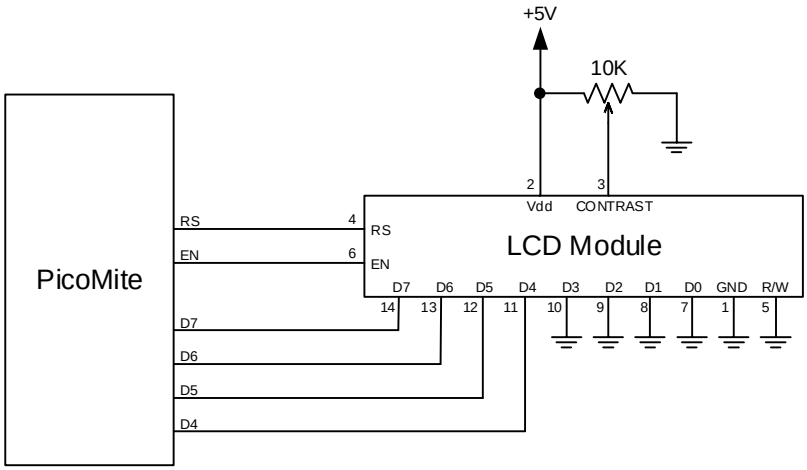
This command will work with LCD modules that use the KS0066, HD44780 or SPLC780 controller chip and have 1, 2 or 4 lines. Typical displays include the LCD16X2 (futurlec.com), the Z7001 (altronics.com.au) and the QP5512 (jaycar.com.au). eBay is another good source where prices can range from $10 to $50.

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

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

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

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



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

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

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

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

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

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

Keypad Interface

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

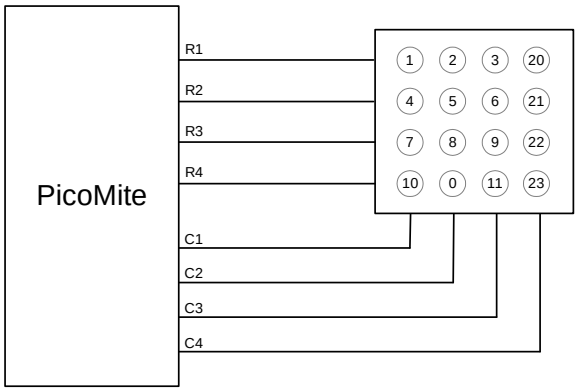
Examples of a 4x3 keypad and a 4x4 keypad are the Altronics S5381 and S5383 (go to www.altronics.com).

To enable the keypad feature you use the command:

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

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

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



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

For example:

```
Keypad KeyCode, KP_Int,GP2,GP3,GP4,GP5,GP21,GP22,GP26   ' 4x3 keybd
DO
  < body of the program >
LOOP

SUB KP_Int                              ' a key press has been detected
  PRINT "Key press = " KeyCode
END SUB
```

WS2812 Support

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

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

```
BITBANG WS2812 type, pin, nbr%, colours%[()]
```

Note that the pin must be set to a digital output before this command is used. The colours%() array should be sized to have at least the same number of elements as the number of LEDs to be driven (nbr%). Each element in the array should contain the colour in the normal RGB888 format (0 - HFFFFFF). Where a single LED is to be driven then colours% should be a simple variable.

Up to 256 WS2812 chips in a string are supported.

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

O = original WS2812

B = WS2812B

S = SK6812

As an example:

```
DIM b%(4)=(RGB(red), Rgb(green), RGB(blue), RGB(Yellow), rgb(cyan))
SETPIN GP5, DOUT
BITBANG WS2812 O, GP5, 5, b%()
```

will output the specified colours to an array of five WS2812 LEDs daisy chained off pin GP5.

It is possible that a WS2812 will not work reliably with the 3.3V output from the PicoMiteVGA . In this case there are a number of solutions:

- Use the WS2812B which will work with a 3.3V supply and inputs.
- Use a level shifter to drive the WS2812.
- Use a single WS2812 powered from 3.3V as a first stage to buffer the input of the first "real" LED in the string. The minimum supply for the WS2812 is 4V but in many cases it will work at 3.3V.

Variables and Expressions

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

Variables

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

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

MMBasic supports three types of variables:

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

Note that it is illegal to use the same variable name with different types. E.g. using `nbr!` and `nbr%` in the same program would cause an error.

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

Constants

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

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

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

String constants must be surrounded by double quote marks ("). E.g. "Hello World".

OPTION DEFAULT

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

```
Nbr = 1234
```

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

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

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

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

OPTION EXPLICIT

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

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

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

DIM and LOCAL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

or

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

STATIC

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

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

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

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

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

As with DIM and LOCAL the variables created with STATIC can be float, integers or strings and arrays of these with or without initialisation. The length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters.

CONST

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

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

```
CONST InputVoltagePin = 31
CONST MaxValue = 2.4
```

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

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

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

```
CONST InputVoltagePin = 31, MaxValue = 2.4, MinValue = 1.5
```

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

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

Expressions and Operators

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

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

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

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

Arithmetic operators:

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

Shift operators:

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

Logical operators:

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

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

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

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

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

String operators:

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

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

Mixing Floating Point and Integers

MMBasic automatically handles conversion of numbers between floating point and integers. If an operation mixes both floating point and integers (e.g. `PRINT A% + B!`) the integer will be converted to a floating point number first, then the operation performed and a floating point number returned. If both sides of the operator are integers then an integer operation will be performed and an integer returned.

The one exception is the normal division ("/") which will always convert both sides of the expression to a floating point number and then returns a floating point number. For integer division you should use the integer division operator "\".

MMBasic functions will return a float or integer depending on their characteristics. For example, PIN() will return an integer when the pin is configured as a digital input but a float when configured as an analog input.

If necessary you can convert a float to an integer with the INT() function. It is not necessary to specifically convert an integer to a float but if it was needed the integer value could be assigned to a floating point variable and it will be automatically converted in the assignment.

64-bit Unsigned Integers

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

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

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

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

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

```
X% = &HFFFF0000FFFF0044
Y% = &H800FFFFFFFFFFFF
X% = X% AND Y%
PRINT HEX$(X%, 16)
```

Will display `"800F0000FFFF0044"`

Subroutines and Functions

A program defined subroutine or function is simply a block of programming code that is contained within a module and can be called from anywhere within your program. It is the same as if you have added your own command or function to the language.

Subroutines

A subroutine acts like a command and it can have arguments (sometimes called a parameter list). In the definition of the subroutine they look like this:

```
SUB MYSUB arg1, arg2$, arg3
  <statements>
  <statements>
END SUB
```

And when you call the subroutine you can assign values to the arguments. For example:

```
MYSUB  23, "Cat", 55
```

Inside the subroutine `arg1` will have the value `23`, `arg2$` the value of `"Cat"`, and so on. The arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be hidden by the arguments defined for the subroutine.

When calling a subroutine you can supply less than the required number of values and in that case the missing values will be assumed to be either zero or an empty string. You can also leave out a value in the middle of the list and the same will happen. For example:

```
MYSUB  23, , 55
```

Will result in `arg2$` being set to the empty string `""`.

Rather than using the type suffix (e.g. the $ in arg2$) you can use the suffix AS <type> in the definition of the subroutine argument and then the argument will be known as the specified type, even when the suffix is not used. For example:

```
SUB MYSUB arg1, arg2 AS STRING, arg3
  IF arg2 = "Cat" THEN …
END SUB
```

Inside a subroutine you can define a variable using LOCAL (which has the same syntax as DIM). This variable will only exist within the subroutine and will vanish when the subroutine exits.

Functions

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

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

For example:

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

Passing Arguments by Reference

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

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

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

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

```
Swap nbr1, nbr2
```

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

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

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

Passing Arrays

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

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

This type of construct is often used in sorting arrays.

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

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

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

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

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

Early Exit

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

Recursion

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

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

Examples

There is often the need for a special command or function to be implemented in MMBasic but in many cases these can be constructed using an ordinary subroutine or function which will then act exactly the same as a built in command or function.

For example, sometimes there is a requirement for a TRIM function which will trim specified characters from the start and end of a string. The following provides an example of how to construct such a simple function in MMBasic.

The first argument to the function is the string to be trimmed and the second is a string containing the characters to trim from the first string. RTrim$() will trim the specified characters from the end of the string, LTrim$() from the beginning and Trim$() from both ends.

```
' trim any characters in c$ from the start and end of s$
Function Trim$(s$, c$)
  Trim$ = RTrim$(LTrim$(s$, c$), c$)
End Function

' trim any characters in c$ from the end of s$
Function RTrim$(s$, c$)
  RTrim$ = s$
  Do While Instr(c$, Right$(RTrim$, 1))
    RTrim$ = Mid$(RTrim$, 1, Len(RTrim$) - 1)
  Loop
End Function

' trim any characters in c$ from the start of s$
Function LTrim$(s$, c$)
  LTrim$ = s$
  Do While Instr(c$, Left$(LTrim$, 1))
    LTrim$ = Mid$(LTrim$, 2)
  Loop
End Function
```

As an example of using these functions:

```
S$ = "   ****23.56700  "
PRINT Trim$(s$, " ")
```

Will give "****23.56700"

```
PRINT Trim$(s$, " *0")
```

Will give "23.567"

```
PRINT LTrim$(s$, " *0")
```

Will give "23.56700"

MMBasic Characteristics

Naming Conventions

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

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

1. Floating point. These can store a number with a decimal point and fraction (e.g. 45.386) and also very large numbers. However, they will lose accuracy when more than 14 significant digits are stored or manipulated. The suffix is '!' and floating point is the default when a variable is created without a suffix
2. 64-bit integer. These can store numbers with up to 19 decimal digits without losing accuracy but they cannot store fractions (i.e. the part following the decimal point). The suffix for an integer is '%'
3. Strings. These will store a string of characters (e.g. "Tom"). The suffix for a string is the '$' symbol (e.g. name$, s$, etc) Strings can be up to 255 characters long.

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

Constants

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

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

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

String constants are surrounded by double quote marks ("). E.g. "Hello World".

Implementation Characteristics

Maximum program size (as plain text) is 124KB (108KB on the PicoMiteVGA). Note that MMBasic tokenises the program when it is stored in flash so the final size in flash might vary from the plain text size.

Maximum length of a command line is 255 characters.

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

Maximum number of dimensions to an array is 5.

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

Maximum number of nested FOR…NEXT loops is 20.

Maximum number of nested DO…LOOP commands is 20.

Maximum number of nested GOSUBs, subroutines and functions (combined) is 320.

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

Maximum number of user defined labels, subroutines and functions (combined): 224

Maximum number of interrupt pins that can be configured: 10

Numbers are stored and manipulated as double precision floating point numbers or 64-bit signed integers. The range of floating point numbers is 1.797693134862316e+308 to 2.225073858507201e-308.

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

Maximum string length is 255 characters.

Maximum line number is 65000.

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

Maximum number of global variables and constants is 256

Maximum number of local variables is 256

The maximum number of files that can be listed by the FILES command is 1000

The maximum length filename supported is 63 characters

Compatibility

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

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

Predefined Read Only Variables

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

MM.VER	The version number of the firmware as a floating point number in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number. For example version 5.03.00 will return 5.03 and version 5.03.01 will return 5.0301.
MM.DEVICE$	A string representing the device or platform that MMBasic is running on. Currently this variable will contain one of the following: "Maximite" on the standard Maximite and compatibles. "Colour Maximite" on the Colour Maximite and UBW32. "Colour Maximite 2" on the Colour Maximite 2. "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "Generic PIC32" for the generic version of MMBasic on a PIC32. "Micromite" on the PIC32MX150/250 "Micromite MkII" on the PIC32MX170/270 "Micromite Plus" on the PIC32MX470 "Micromite Extreme" on the PIC32MZ series "ARMmite H7" on the ArmmiteH7 "ARMmite F407" on the ArmmiteF4 "ARMmite L4" with chip no. and pin count appended on the ArmmiteL4 “PicoMite” on the Raspberry Pi Pico "PicoMiteVGA” on the Raspberry Pico VGA Edition
MM.ERRNO MM.ERRMSG$	If a statement caused an error which was ignored these variables will be set accordingly. MM.ERRNO is a number where non zero means that there was an error and MM.ERRMSG$ is a string representing the error message that would have normally been displayed on the console. They are reset to zero and an empty string by RUN, ON ERROR IGNORE or ON ERROR SKIP.
MM.INFO() MM.INFO$()	These two versions can be used interchangeably but good programming practice would require that you use the one corresponding to the returned datatype.
MM.INFO$(AUTORUN)	Returns the setting of the OPTION AUTORUN command
MM.INFO$(CPUSPEED)	Returns the CPU speed as a string
MM.INFO$(SDCARD)	Returns the status of the SD card. Valid returns are: DISABLED, NOT PRESENT, READY, and UNUSED
MM.INFO(DISK SIZE)	Returns the capacity of the SD card in bytes
MM.INFO$(FREE SPACE)	Returns the free space on the SD card.
MM.INFO$(FILESIZE file$)	Returns the size of file$ in bytes or -1 if not found, -2 if a directory.
MM.INFO(ID)	Returns the unique ID of the Pico PCB
MM.INFO$(MODIFIED file$)	Returns the date/time that file$ was modified, Empty string if not found
MM.INFO(FCOLOUR)	Returns the current foreground colour

MM.INFO(BCOLOUR)	Returns the current background colour
MM.INFO(FONT)	Returns the number of the currently active font
MM.INFO(FONT ADDRESS n)	Returns the address of the memory location with the address of FONT n
MM.INFO(FONT POINTER n)	Returns a POINTER to the start of FONT n in memory
MM.INFO(FONTHEIGHT) MM.INFO(FONTWIDTH)	Integers representing the height and width of the current font (in pixels).
MM.INFO$(FLASH)	Reports which flash slot the program was loaded from if applicable
MM.INFO(HPOS) MM.INFO(VPOS)	The current horizontal and vertical position (in pixels) following the last graphics or print command.
MM.INFO(OPTION option)	Returns the current value of a range of options that affect how a program will run. “option” can be one of AUTORUN, BASE, BREAK, DEFAULT, EXPLICIT
MM.INFO$(PIN pinno)	Returns the status of I/O pin 'pinno'. Valid returns are: INVALID, RESERVED, IN USE, and UNUSED
MM.INFO$(PINNO GPnn)	Returns the physical pin number for a given GP number.
MM.INFO$(RESET)	Returns the cause of a firmware restart. The returned value will be one of “Switch” (i.e. pressing the reset switch), “Power-On”, “Software”, and “Watchdog” (NB the latter is the H/W watchdog and unrelated to the MMbasic version which causes a software reset).
MM.INFO$(TOUCH)	Returns the status of the Touch controller. Valid returns are: “Disabled”, “Not calibrated”, and “Ready”.
MM.INFO(VERSION)	Returns the version number as a floating point number
MM.HRES MM.VRES	Integers representing the horizontal and vertical resolution of the VGA output in pixels.
MM.FONTHEIGHT MM.FONTWIDTH	Integers representing the height and width of the current font (in pixels) kept for compatibility. NB: these are automatically converted into MM.INFO(FONTHEIGHT) and MM.INFO(FONTWIDTH) by MMBasic
MM.ONEWIRE	Following a 1-Wire reset function this integer variable will be set to indicate the result of the operation: 0 = Device not found, 1 = Device found
MM.I2C	Following an I2C write or read command this integer variable will be set to indicate the result of the operation as follows: 0 = The command completed without error. 1 = Received a NACK response 2 = Command timed out
MM.WATCHDOG	An integer which is true if MMBasic was restarted as the result of a Watchdog timeout (see the WATCHDOG command) otherwise false.

Options

This table lists the various option commands which can be used to configure MMBasic and change the way it operates. Options that are marked as permanent will be saved in non-volatile memory and automatically restored when the PicoMiteVGA is restarted. Options that are not permanent will be reset on start-up.

Many OPTION commands will force a restart of the PicoMiteVGA and that will cause the USB console interface to be reset. The program held in memory will not be lost as the firmware will automatically restore a backup copy on restart.

	Permanent?	
OPTION AUDIO PWMnApin, PWMnBpin	✓	Configures one of the PWM channels as an audio output. 'PWMnApin' is the left audio channel, 'PWMnBpin' is the right. Both pins must belong to the same audio channel. Example, OPTION AUDIO GP18, GP19 would use PWM1A and PWM1B on pins 24 and 25 respectively. This option prevents use of these pins in the BASIC program. The audio output is generated using PWM so a low pass filter is necessary on the output. The audio output from the PicoMiteVGA is very noisy. Using OPTION POWER and/or supplying power via a separate 3.3V linear regulator can reduce this. This command must be run at the command prompt (not in a program).
OPTION AUTORUN ON or OPTION AUTORUN n or OPTION AUTORUN OFF	✓	Instructs MMBasic to automatically run a program on power up or restart. ON will cause the the current program to be run. Specifying 'n' will cause that location in flash memory to be run. 'n' must be in the range 1 to 8. OFF will disable the autorun option and is the default for a new program. Entering the break key (default CTRL-C) at the console will interrupt the running program and return to the command prompt.
OPTION BASE 0 | 1		Set the lowest value for array subscripts to either 0 or 1. This must be used before any arrays are declared and is reset to the default of 0 on power up.
OPTION BAUDRATE nn		Set the baudrate of the serial console (if it is configured).
OPTION BREAK nn		Set the value of the break key to the ASCII value 'nn'. This key is used to interrupt a running program. The value of the break key is set to CTRL-C key at power up but it can be changed to any keyboard key using this command (for example, OPTION BREAK 4 will set the break key to the CTRL-D key). Setting this option to zero will disable the break function entirely.
OPTION CASE LOWER | UPPER | TITLE	✓	Change the case used for listing command and function names when using the LIST command. The default is TITLE but the old standard of MMBasic can be restored using OPTION CASE UPPER. This command must be run at the command prompt (not in a program).

OPTION COLOURCODE ON or OPTION COLOURCODE OFF	✓	Turn on or off colour coding for the editor's output. Keywords will be in cyan, comments in yellow, etc. The keyword COLORCODE (USA spelling) can also be used. This requires a terminal emulator that can interpret the appropriate escape codes (eg, Tera Term). This command must be run at the command prompt (not in a program).
OPTION CPUSPEED speed	✓	Change the CPU clock. 'speed' is the CPU clock in KHz and can be either 126000 or 252000. Any speed above 133MHz is regarded as overclocking of the standard Raspberry Pi Pico. With no option set the CPU speed will default to 126000. This command must be run at the command prompt (not in a program).
OPTION COUNT pin1, pin2, pin3, pin4	✓	Specifies which pins are to be used as Count inputs. By default these are GP6, GP7, GP8 and GP9. The SETPIN command defines the Counter mode. This command must be run at the command prompt (not in a program).
OPTION DEFAULT FLOAT | INTEGER | STRING | NONE		Used to set the default type for a variable which is not explicitly defined. If OPTION DEFAULT NONE is used then all variables must have their type explicitly defined or the error "Variable type not specified" will occur. When a program is run the default is set to FLOAT for compatibility with Microsoft BASIC and previous versions of MMBasic.
OPTION DEFAULT COLOURS foreground [,background]	✓	Set the default foreground and background colours for both the monochrome and colour modes. The colour must be one of the following: white, yellow, lilac, brown, fuchsia, rust, magenta, red, cyan, green, cerulean, midgreen, cobalt, myrtle, blue and black. A numeric value cannot be used. The default is white, black. If background is omitted it defaults to black.
OPTION DEFAULT MODE n	✓	Set the default resolution and colour mode at bootup. 'n' is the mode. 1 = monochrome 640 x 480 mode, 2 = colour 320 x 240 mode. Default is mode 1. This command must be run at the command prompt (not in a program).
OPTION EXPLICIT		Placing this command at the start of a program will require that every variable be explicitly declared using the DIM, LOCAL or STATIC commands before it can be used in the program. This option is disabled by default when a program is run. If it is used it must be specified before any variables are used.
OPTION FNKey string$	✓	Define the string that will be generated when a function key is pressed at the command prompt. 'FNKey' can be F1, and F5 thru to F9. Example: OPTION F8 "RUN "+chr$(34)+"myprog" +chr$(34)+chr$(13)+chr$(10). This command must be run at the command prompt (not in a program).
OPTION KEYBOARD nn [,capslock] [,numlock] [repeatstart] [repeatrate]	✓	Configure a PS2 keyboard. 'nn is a two character code defining the keyboard layout. The choices are US for the standard keyboard layout in the USA, Australia

		and New Zealand and UK for the United Kingdom, GR for Germany, FR for France and ES for Spain. The default is US. This command must be run at the prompt line and will cause a reboot. The optional parameters 'capslock' and 'numlock' set the initial state of the keyboard (default 0, 1). 'repeatstart' defines how how long before a character repeats the first time (valid 0-3 = 250mSec, 500mSec, 750mSec, 1S: default 1=500mSec). 'repeatrate' defines how fast a character repeats after the first repeat (valid 0-31 = 33mSec to 500mSec: default 12=100mSec).
OPTION LEGACY ON or OPTION LEGACY OFF		This will turn on or off compatibility mode with the graphic commands used in the original Colour Maximite. The commands COLOUR, LINE, CIRCLE and PIXEL use the legacy syntax and all drawing commands will accept colours in the range of 0 to 7. Notes: • Keywords such as RED, BLUE, etc are not implemented so they should be defined as constants if needed. • Refer to the Colour Maximite MMBasic Language Manual for the syntax of the legacy commands. This can be downloaded from https://geoffg.net/OriginalColourMaximite.html .
OPTION LIST OPTION RESET		This will list the settings of any options that have been changed from their default setting and are the permanent type. Clears all options. These commands must be run at the command prompt (not in a program).
OPTION PIN nbr		Set 'nbr' as the PIN (Personal Identification Number) for access to the console prompt. 'nbr' can be any non zero number of up to eight digits. Whenever a running program tries to exit to the command prompt for whatever reason MMBasic will request this number before the prompt is presented. This is a security feature as without access to the command prompt an intruder cannot list or change the program in memory or modify the operation of MMBasic in any way. To disable this feature enter zero for the PIN number (i.e. OPTION PIN 0). A permanent lock can be applied by using 99999999 for the PIN number. If a permanent lock is applied or the PIN number is lost the only way to recover is to erase the Raspberry Pi Pico flash to factory defaule and then reload the PicoMiteVGA firmware. This command must be run at the command prompt (not in a program).
OPTION POWER PFM | PWM	✓	Changes operation of the 3.3V supply switch mode power supply. By default this runs in PFM mode. PWM gives better noise performance but is less power-efficient. Note that under heavy load the system will run in PWM mode regardless of this setting.
OPTION RESET	✓	Reset all saved options to their default values. This command must be run at the command prompt (not in a program).
OPTION RTC AUTO ENABLE | DISABLE	✓	Enable auto-load time$ & date$ from RTC on boot & every hour. If enabled and the RTC does not respond then any running program will abort with an error. At the command prompt an information message will be output. This command must be run at the command prompt (not in a program).

OPTION SDCARD CSpin ,CLKpin, MOSIpin, MISOpin	✓	Specify the I/O pins to use for the SD card interface. See the section titled *Connecting It Up* If this is not set the SD card will not be available. This command must be run at the command prompt (not in a program).
OPTION SERIAL CONSOLE uartapin, uartbpin OPTION SERIAL CONSOLE DISABLE	✓	Specify that the console be accessed via a hardware serial port (instead of virtual serial over USB). 'uartapin' and 'uartbpin' can be any valid pair of rx and tx pins for either COM1 or COM2. The order that they are specified is not important. The speed defaults to 115200 baud but can be changed with OPTION BAUDRATE. Revert to the normal the USB console. These commands must be run at the command prompt (not in a program).
OPTION SYSTEM I2C sdapin, sclpin	✓	Specify the I^2C port and pins for use by system devices (eg, RTC). The PicoMiteVGA uses a specific I^2C port for system devices, leaving the other for the programmer. This command specifies which pins are to be used, and hence which of the I^2C ports is to be used. The pins allocated to the SYSTEM I2C will not be available for other MMBasic SETPIN settings but can be used for additional I^2C devices using the standard I2C command. Note: I2C(2) OPEN and I2C(2) CLOSE are not available in this case. This command must be run at the command prompt (not in a program).
OPTION TAB 2 | 3 | 4 | 8	✓	Set the spacing for the tab key. Default is 2.
OPTION VCC voltage		Specifies the voltage (Vcc) supplied to the PicoMiteVGA . When using the ADC pins to measure voltage the PicoMiteVGA uses the voltage on the pin marked VREF as its reference. This voltage can be accurately measured using a DMM and set using this command for more accurate measurement. The parameter is not saved and should be initialised either on the command line or in a program. The default if not set is 3.3.

Commands

Square brackets indicate that the parameter or characters are optional.

‘ (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
? (question mark)	Shortcut for the PRINT command.
ADC OPEN freq, n_channels [,interrupt]	This allocates up to 4 ADC channels for use GP26, GP27, GP28, and GP29 and sets them to be converted at the specified frequency. The maximum total frequency is 500KHz (e.g. 125KHz if all four channels are to be sampled). If the number of channels is one then it will always be GP26 used, if two then GP26 and GP27 etc.. Sampling of multiple channels is sequential (there is only one ADC). The specified pins are locked to the function when ADC OPEN is active The optional interrupt parameter specifies an interrupt to call when the conversion completes. If not specified then conversion will be blocking
ADC FREQUENCY freq	This changes the sampling frequency of the ADC conversion without having to close and re-open
ADC CLOSE	Releases the pins to normal usage
ADC START array1!() [,array2!()] [,array3!()] [,array4!()]	This starts conversion into the specified arrays. The arrays must be floating point and the same size. The size of the arrays defines the number of conversions. The results are returned as a voltage between 0 and OPTION VCC (defaults to 3.3V). Start can be called repeatedly once the ADC is OPEN
ARC x, y, r1, [r2], a1, a2 [, c]	Draws an arc of a circle with a given colour and width between two radials (defined in degrees). Parameters for the ARC command are: x: X coordinate of centre of arc y: Y coordinate of centre of arc r1: inner radius of arc r2: outer radius of arc - can be omitted if 1 pixel wide a1: start angle of arc in degrees a2: end angle of arc in degrees c: Colour of arc (if omitted it will default to the foreground colour) Zero degrees is at the 9 o'clock position.
AUTOSAVE or AUTOSAVE CRUNCH	Enter automatic program entry mode. This command will take lines of text from the console serial USB input and save them to program memory. This mode is terminated by entering Control-Z or F1 which will then cause the received data to be transferred into program memory overwriting the previous program. Use F2 to exit and immediately run the program. The CRUNCH option instructs MMBasic to remove all comments, blank lines and unnecessary spaces from the program before saving. This can be used on large programs to allow them to fit into limited memory. CRUNCH can be abbreviated to the single letter C. At any time this command can be aborted by Control-C which will leave program memory untouched. This is one way of transferring a BASIC program into the PicoMiteVGA. The program to be transferred can be pasted into a terminal emulator and this command will capture the text stream and store it into program memory. It can also be used for entering a small program directly at the console input.

BITBANG BITSTREAM pinno, n_transitions, array%()	This command is used to generate an extremely accurate bit sequence on the pin specified. The pin must have previously been set up as an output and set to the required starting level. Notes: • The array contains the length of each level in the bitstream in microseconds. The maximum period allowed is 65.5 mSec • The first transition will occur immediately on executing the command. • The last period in the array is ignored other than defining the time before control returns to the program or command line. • The pin is left in the starting state if the number of transitions is even and the opposite state if the number of transitions is odd.
BITBANG HUMID pin, tvar, hvar [,DHT11]	Returns the temperature and humidity using the DHT22 sensor. Alternative versions of the DHT22 are the AM2303 or the RHT03 (all are compatible). 'pin' is the I/O pin connected to the sensor. Any I/O pin may be used. 'tvar' is the variable that will hold the measured temperature and 'hvar' is the same for humidity. Both must be present and both must be floating point variables. For example: `HUMID 3, TEMP!, HUMIDITY!` Temperature is measured in ºC and the humidity is percent relative humidity. Both will be measured with a resolution of 0.1. If an error occurs (sensor not connected or corrupt signal) both values will be 1000.0. Normally the DHT22 should powered by 3.3V to keep its output below 3.6V for the PicoMiteVGA and the signal pin of should be pulled up by a 1K to 10K resistor (4.7K recommended) to 3.3V. The optional DHT11 parameter modifies the timings to work with the DHT11. Set to 1 for DHT11 and 0 or omit for DHT22.
BITBANG WS2812 type, pin, nbr, value%[()]	This command outputs the required signals to drive one or more WS2812 LED chips connected to 'pin'. Note that the pin must be set to a digital output before this command is used. 'type' is a single character specifying the type of chip being driven: O = original WS2812 B = WS2812B S = SK6812 'nbr' is the number of LEDs in the chain (1 to 256). The 'value%()' array should be an integer array sized to have exactly the same number of elements as the number of LEDs to be driven. Each element in the array should contain the colour in the normal RGB888 format (i.e. 0 to &HFFFFFF). If only one LED is connected then a single integer should be used for value% (ie, not an array).
BITBANG LCD INIT d4, d5, d6, d7, rs, en or BITBANG LCD line, pos, text$ or BITBANG LCD CLEAR or BITBANG LCD CLOSE	Display text on an LCD character display module. This command will work with most 1-line, 2-line or 4-line LCD modules that use the KS0066, HD44780 or SPLC780 controller (however this is not guaranteed). The LCD INIT command is used to initialise the LCD module for use. 'd4' to 'd7' are the I/O pins that connect to inputs D4 to D7 on the LCD module (inputs D0 to D3 should be connected to ground). 'rs' is the pin connected to the register select input on the module (sometimes called CMD). 'en' is the pin connected to the enable or chip select input on the module. The R/W input on the module should always be grounded. The above I/O pins are automatically set to outputs by this command. When the module has been initialised data can be written to it using the LCD command. 'line' is the line on the display (1 to 4) and 'pos' is the character location on the line (the first location is 1). 'text$' is a string containing the

	text to write to the LCD display. 'pos' can also be C8, C16, C20 or C40 in which case the line will be cleared and the text centred on a 8 or 16, 20 or 40 line display. For example: `LCD 1, C16, "Hello"` LCD CLEAR will erase all data displayed on the LCD and LCD CLOSE will terminate the LCD function and return all I/O pins to the not configured state. See the chapter "Special Hardware Devices" for more details.
BITBANG LCD CMD d1 [, d2 [, etc]] or BITBANG LCD DATA d1 [, d2 [, etc]]	These commands will send one or more bytes to an LCD display as either a command (LCD CMD) or as data (LCD DATA). Each byte is a number between 0 and 255 and must be separated by commas. The LCD must have been previously initialised using the LCD INIT command (see above). These commands can be used to drive a non standard LCD in "raw mode" or they can be used to enable specialised features such as scrolling, cursors and custom character sets. You will need to refer to the data sheet for your LCD to find the necessary command and data values.
BLIT READ [#]b, x, y, w, h BLIT WRITE [#]b, x, y[, w] [, h] BLIT LOAD [#]b, fname$ [,x] [,y] [,w] [,h] BLIT CLOSE [#]b	Copy one section of the display screen on the VGA display to or from a memory buffer. BLIT READ will copy a portion of the display to the memory buffer '#b'. The source coordinate is 'x' and 'y' and the width of the display area to copy is 'w' and the height is 'h'. When this command is used the memory buffer is automatically created and sufficient memory allocated. This buffer can be freed and the memory recovered with the BLIT CLOSE command. BLIT WRITE will copy the memory buffer '#b' to the display. The destination coordinate is 'x' and 'y' and the width/height of the buffer to copy is 'w' and 'h'. If omitted w,h will default to the width and height of the blit buffer. BLIT LOAD will load a blit buffer from a 24-bit bmp image file. x,y define the start position in the image to start loading and w,h specify the width and height of the area to be loaded. e.g. `BLIT LOAD #1,"image1", 50,50,100,100` will load an area of 100 pixels square with the top left had corner at 50,50 from the image image1.bmp BLIT CLOSE will close the memory buffer '#b' to allow it to be used for another BLIT READ operation and recover the memory used. Notes: • Eight buffers are available ranging from #1 to #8. • When specifying the buffer number the # symbol is optional. All other arguments are in pixels.
BLIT x1, y1, x2, y2, w, h	Copy one section of the display screen to another part of the display. The source coordinate is 'x1' and 'y1'. The destination coordinate is 'x2' and 'y2'. The width of the screen area to copy is 'w' and the height is 'h'. All arguments are in pixels.
BOX x, y, w, h [, lw] [,c] [,fill]	Draws a box on the VGA output with the top left hand corner at 'x' and 'y' with a width of 'w' pixels and a height of 'h' pixels. 'lw' is the width of the sides of the box and can be zero. It defaults to 1. 'c' specifies the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x',

	'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'c', and fill can be either arrays or single variables/constants. See the chapter "Graphics Commands and Functions" for a definition of the colours and graphics coordinates.
CALL usersubname$ [,usersubparameters,....]	This is an efficient way of programmatically calling user defined subroutines (see also the CALL() function). In many case it can allow you to get rid of complex SELECT and IF THEN ELSEIF ENDIF clauses and is processed in a much more efficient way. The “usersubname$” can be any string or variable or function that resolves to the name of a normal user subroutine (not an in-built command). The “usersubparameters” are the same parameters that would be used to call the subroutine directly. A typical use could be writing any sort of emulator where one of a large number of subroutines should be called depending on some variable. It also allows a way of passing a subroutine name to another subroutine or function as a variable.
CHDIR dir$	Change the current working directory on the SD card to ‘dir$’ The special entry “..” represents the parent of the current directory and “.” represents the current directory. "/" is the root directory.
CIRCLE x, y, r [,lw] [, a] [, c] [, fill]	Draw a circle on the video output centred at 'x' and 'y' with a radius of 'r' on the VGA monitor. ‘lw’ is optional and is the line width (defaults to 1). 'c' is the optional colour and defaults to the current foreground colour if not specified. The optional 'a' is a floating point number which will define the aspect ratio. If the aspect is not specified the default is 1.0 which gives a standard circle. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can be expressed as arrays and the software will plot the number of circles as determined by the dimensions of the smallest array. 'x', 'y' and 'r' must all be arrays or all be single variables/constants otherwise an error will be generated. 'lw', 'a', 'c', and fill can be either arrays or single variables/constants. See the chapter "Graphics Commands and Functions" for a definition of the colours and graphics coordinates.
CLEAR	Delete all variables and recover the memory used by them. See ERASE for deleting specific array variables.
CLOSE [#]fnbr [,[#]fnbr] …	Close the file(s) previously opened on the SD card with the file number ‘#fnbr’. The # is optional. Also see the OPEN command.
CLS [colour]	Clears the VGA monitor’s screen. Optionally 'colour' can be specified which will be used for the background when clearing the screen.
COLOUR fore [, back] or COLOR fore [, back]	Sets the default colour for commands (PRINT, etc) that display on the on the VGA monitor. 'fore' is the foreground colour, 'back' is the background colour. The background is optional and if not specified will default to black.
CONST id = expression [, id = expression] … etc	Create a constant identifier which cannot be changed once created. 'id' is the identifier which follows the same rules as for variables. The identifier can have a type suffix (!, %, or $) but it is not required. If it is specified it must match the type of 'expression'. 'expression' is the value of the identifier and it can be a normal expression (including user defined functions) which will be evaluated when the constant is created. A constant defined outside a sub or function is global and can be seen throughout the program. A constant defined inside a sub or function is local to that routine and will hide a global constant with the same name.

CONTINUE	Resume running a program that has been stopped by an END statement, an error, or CTRL-C. The program will restart with the next statement following the previous stopping point. Note that it is not always possible to resume the program correctly – this particularly applies to complex programs with graphics, nested loops and/or nested subroutines and functions.
CONTINUE DO or CONTINUE FOR	Skip to the end of a DO/LOOP or a FOR/NEXT loop. The loop condition will then be tested and if still valid the loop will continue with the next iteration.
COPY fname1$ TO fname2$	On the SD card copy a file from ‘fname1$’ to ‘fname2$’. Both are strings. A directory path can be used in both 'fname$' and 'fname$'. If the paths differ the file specified in 'fname$' will be copied to the path specified in 'fname2$' with the file name as specified.
CPU RESTART	Will force a restart of the processor. This will clear all variables and reset everything (e.g. timers, COM ports, I2C, etc) similar to a power up situation but without the power up banner. If OPTION AUTORUN has been set the program in the specified flash location or program mrmory will restart.
CPU SLEEP n	Will cause the processor to sleep for ‘n’ seconds. Note that the CPU does not have a true low power sleep so the power saving is limited.
CSUB name [type [, type] …] hex [[hex[…] hex [[hex[…] END CSUB	Defines the binary code for an embedded machine code program module written in C or ARM assembler. The module will appear in MMBasic as the command 'name' and can be used in the same manner as a built-in command. Multiple embedded routines can be used in a program with each defining a different module with a different 'name'. The first 'hex' word is a 32 bit word which is the offset in bytes from the start of the CSUB to the entry point of the embedded routine (usually the function main()). The following hex words are the compiled binary code for the module. These are automatically programmed into MMBasic when the program is saved. Each 'hex' must be exactly eight hex digits representing the bits in a 32-bit word and be separated by one or more spaces or new lines. The command must be terminated by a matching END CSUB. Any errors in the data format will be reported when the program is run. During execution MMBasic will skip over any CSUB commands so they can be placed anywhere in the program. The type of each parameter can be specified in the definition. For example: `CSUB MySub integer, integer, string` This specifies that there will be three parameters, the first two being integers and the third a string. Note: • Up to ten arguments can be specified ('arg1', 'arg2', etc). • If a variable or array is specified as an argument the C routine will receive a pointer to the memory allocated to the variable or array and the C routine can change this memory to return a value to the caller. In the case of arrays, they should be passed with empty brackets e.g. arg(). In the CSUB the argument will be supplied as a pointer to the first element of the array. • Constants and expressions will be passed to the embedded C routine as pointers to a temporary memory space holding the value.

DATA constant[,constant]...	Stores numerical and string constants to be accessed by READ. In general string constants should be surrounded by double quotes ("). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as THEN, WHILE, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.
DATE$ = "DD-MM-YY[YY]" or DATE$ = "DD/MM/YY[YY]" or DATE$ =”YYYY-MM-DD” or DATE$=”YYYY/MM/DD”	Set the date of the internal clock/calendar. DD, MM and YY are numbers, for example: DATE$ = "28-7-2014" The date is set to "01-01-2000" on power up.
DEFINEFONT #Nbr hex [[hex[…] hex [[hex[…] END DEFINEFONT	This will define an embedded font which can be used alongside or to replace the built in font(s) used on the VGA output. These work exactly same as the built in fonts (i.e. selected using the FONT command or specified in the TEXT command). See the *Embedded Fonts* folder in the PicoMiteVGA distribution zip file for a selection of embedded fonts and a full description of how to create them. '#Nbr' is the font's reference number (from 1 to 16). It can be the same number as a built in font and in that case it will replace the built in font. Each 'hex' must be exactly eight hex digits and be separated by spaces or new lines from the next. • Multiple lines of 'hex' words can be used with the command terminated by a matching END DEFINEFONT. • Multiple embedded fonts can be used in a program with each defining a different font with a different font number. • During execution MMBasic will skip over any DEFINEFONT commands so they can be placed anywhere in the program. • Any errors in the data format will be reported when the program is saved.

DIM [type] decl [,decl]...
where 'decl' is:
var [length] [type] [init]
'var' is a variable name with optional dimensions
'length' is used to set the maximum size of the string to 'n' as in LENGTH n
'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword AS - as in AS FLOAT)
'init' is the value to initialise the variable and consists of:
= <expression>
For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.
	Declares one or more variables (i.e. makes the variable name and its characteristics known to the interpreter).
	When OPTION EXPLICIT is used (as recommended) the DIM, LOCAL or STATIC commands are the only way that a variable can be created. If this option is not used then using the DIM command is optional and if not used the variable will be created automatically when first referenced.
	The type of the variable (i.e. string, float or integer) can be specified in one of three ways:
	By using a type suffix (i.e. !, % or $ for float, integer or string). For example:
	`DIM nbr%, amount!, name$`
	By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (i.e. it does not have to be repeated). For example:
	`DIM STRING first_name, last_name, city`
	By using the Microsoft convention of using the keyword "AS" and the type keyword (i.e. FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable.

Examples:

DIM nbr(50)

DIM INTEGER nbr(50)

DIM name AS STRING

DIM a, b$, nbr(100), strn$(20)

DIM a(5,5,5), b(1000)

DIM strn$(200) LENGTH 20

DIM STRING strn(200)
LENGTH 20

DIM a = 1234, b = 345

DIM STRING strn = "text"

DIM x%(3) = (11, 22, 33, 44)

For example:

```
DIM amount AS FLOAT, name AS STRING
```

Floating point or integer variables will be set to zero when created and strings will be set to an empty string (i.e. ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example:

```
DIM STRING city = "Perth", house = "Brick"
```

The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed. See the chapter "Defining and Using Variables" for more examples of the syntax.

As well as declaring simple variables the DIM command will also declare arrayed variables (i.e. an indexed variable with a number of dimensions). Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example:

```
DIM array(10, 20)
```

Each number specifies the number of elements in each dimension. Normally the numbering of each dimension starts at 0 but the OPTION BASE command can be used to change this to 1.

The above example specifies a two dimensional array with 11 elements (0 to 10) in the first dimension and 21 (0 to 20) in the second dimension. The total number of elements is 231 and because each floating point number on the PicoMiteVGA requires 8 bytes a total of 1848 bytes of memory will be allocated.

Strings will default to allocating 255 bytes (i.e. characters) of memory for each element and this can quickly use up memory when defining arrays of strings. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters.

For example: `DIM STRING s(5, 10)` will declare a string array with 66 elements consuming 16,896 bytes of memory while:

```
DIM STRING s(5, 10) LENGTH 20
```

Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.

If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this, string arrays created with the LENGTH keyword act exactly the same as other string arrays. This keyword can also be used with non-array string variables but it will not save any memory.

In the above example you can also use the Microsoft syntax of specifying the type after the length qualifier. For example:

```
DIM s(5, 10) LENGTH 20 AS STRING
```

Arrays can also be initialised when they are declared by adding an equals symbol (=) followed by a bracketed list of values at the end of the declaration. For example:

```
DIM INTEGER nbr(4) = (22, 44, 55, 66, 88)
```

or

```
DIM s$(3) = ("foo", "boo", "doo", "zoo")
```

Note that the number of initialising values must match the number of elements in the array including the base value set by OPTION BASE. If a multi dimensioned array is initialised then the first dimension will be initialised first followed by the second, etc.

Also note that the initialising values must be after the LENGTH qualifier (if used) and after the type declaration (if used).

DO <statements> LOOP	This structure will loop forever; the EXIT DO command can be used to terminate the loop or control must be explicitly transferred outside of the loop by commands like GOTO or EXIT SUB (if in a subroutine).
DO WHILE expression <statements> LOOP	Loops while 'expression' is true (this is equivalent to the older WHILE-WEND loop). If, at the start, the expression is false the statements in the loop will not be executed, not even once.
DO <statements> LOOP UNTIL expression	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is true.
EDIT	Invoke the full screen editor. See the section *Full Screen Editor* for details of how to use the editor.
ELSE	Introduces an optional default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF expression THEN or ELSE IF expression THEN	Introduces an optional secondary condition in a multiline IF statement. See the multiline IF statement for more details.
END	End the running program and return to the command prompt.
END CSUB	Marks the end of a C subroutine. See the CSUB command. Each CSUB must have one and only one matching END CSUB statement.
END FUNCTION	Marks the end of a user defined function. See the FUNCTION command. Each function must have one and only one matching END FUNCTION statement. Use EXIT FUNCTION if you need to return from a function from within its body.

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

END SUB	Marks the end of a user defined subroutine. See the SUB command. Each sub must have one and only one matching END SUB statement. Use EXIT SUB if you need to return from a subroutine from within its body.
ERASE variable [,variable]...	Deletes variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (e.g. `dat()`) or just by specifying the variable's name (e.g. `dat`). Use CLEAR to delete all variables at the same time (including arrays).
ERROR [error_msg$]	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXECUTE command$	This executes the Basic command "command$". Use should be limited to basic commands that execute sequentially for example the GOTO statement will not work properly Things that are tested and work OK include GOSUB, Subroutine calls, other simple statements (like PRINT and simple assignments) Multiple statements separated by : are not allowed and will error The command sets an internal watchdog before executing the requested command and if control does not return to the command, like in a GOTO statement, the timer will expire. In this case you will get the message "Command timeout". RUN is a special case and will cancel the timer allowing you to use the command to chain programs if required.

EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.
FILES [fspec$] [,sort]	Lists files in any directory on the SD card. 'fspec$' (if specified) can contain a path and search wildcards in the filename. Question marks (?) will match any character and an asterisk (*) will match any number of characters. If omitted, all files will be listed. For example: * Find all entries *.TXT Find all entries with an extension of TXT E*.* Find all entries starting with E X?X.* Find all three letter file names starting and ending with X mydir/* Find all entries in directory mydir NB: putting wildcards in the pathname will result in an error 'sort' specifies the sort order as follows: size by ascending size time by descending time/date name by file name (default if not specified) type by file extension
FLASH	Manages the storage of programs in the flash memory. Up to seven programs can be stored in the flash memory and retrieved as required. Note that these saved programs will be erased with a firmware upgrade. One of these flash memory locations can be automatically loaded and run when power is applied using the OPTION AUTORUN n command. In the following 'n' is a number 1 to 8.
FLASH ALL	Displays a list of all flash locations including the first line of the program.
FLASH LIST n [,all]	List the program saved to slot n. Use ALL to **not** paginate the output.
FLASH ERASE n	Erase a flash program location.
FLASH ERASE ALL	Erase all flash program locations.
FLASH SAVE n	Save the current program to the flash location specified.
FLASH LOAD n	Load a program from the specified flash location into program memory.
FLASH RUN n	Runs the program in flash location n, clear all variables. Does not change the program memory.
FLASH CHAIN n	Runs the program in flash location n, leaving all variables intact (allows for a program that is much bigger than the program memory). Does not change the program memory.
FLASH OVERWRITE n	Erase a flash program location and then save the current program to the flash location specified.
FLUSH [#]fnbr	Causes any buffered writes to a file previously opened with the file number '#fnbr' to be written to disk. The # is optional. Using this command ensures that no data is lost if there is a power cut after a write command.
FONT [#]font-number, scaling	This will set the default font for displaying text on the VGA output. Fonts are specified as a number. For example, #2 (the # is optional). See the section *Graphics Commands and Functions* for details of the available fonts. 'scaling' can range from 1 to 15 and will multiply the size of the pixels making the displayed character correspondingly wider and higher. E.g. a scale of 2 will double the height and width.

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

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

GOTO target	Branches program execution to the target, which can be a line number or a label.
GUI BITMAP x, y, bits [, width] [, height] [, scale] [, c] [, bc]	Displays the bits in a bitmap on the VGA monitor starting at 'x' and 'y' pixel. 'height' and 'width' are the dimensions of the bitmap in pixels and default to 8x8.

	'scale' is optional and defaults to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. The bitmap can be an integer or a string variable or constant and is drawn using the first byte as the first bits of the top line (bit 7 first, then bit 6, etc) followed by the next byte, etc. When the top line has been filled the next line of the displayed bitmap will start with the next bit in the integer or string. See the section *Graphics Commands and Functions* for a definition of the colours and graphics coordinates.
I2C OPEN speed, timeout	Enables the first I^2C module in master mode. 'speed' is the clock speed (in KHz) to use and must be one of 100, 400 or 1000. 'timeout' is a value in milliseconds after which the master send and receive commands will be interrupted if they have not completed. The minimum value is 100. A value of zero will disable the timeout (though this is not recommended).
I2C WRITE addr, option, sendlen, senddata [,sendata ....]	Send data to the I^2C slave device. 'addr' is the slave's I^2C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 'sendlen' is the number of bytes to send. 'senddata' is the data to be sent - this can be specified in various ways (all values sent will be between 0 and 255). Notes: • The data can be supplied as individual bytes on the command line. Example: `I2C WRITE &H6F, 0, 3, &H23, &H43, &H25` • The data can be in a one dimensional array specified with empty brackets (i.e. no dimensions). 'sendlen' bytes of the array will be sent starting with the first element. Example: `I2C WRITE &H6F, 0, 3, ARRAY()` The data can be a string variable (not a constant). Example: `I2C WRITE &H6F, 0, 3, STRING$`
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I2C slave device. 'addr' is the slave's I^2C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 'rcvlen' is the number of bytes to receive. 'rcvbuf' is the variable or array used to save the received data - this can be: • A string variable. Bytes will be stored as sequential characters. • A one dimensional array of numbers specified with empty brackets. Received bytes will be stored in sequential elements of the array starting with the first. Example: `I2C READ &H6F, 0, 3, ARRAY()` A normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE	Disables the master I^2C module. This command will also send a stop if the bus is still held.
I2C SLAVE	See Appendix B
I2C2	The same set of commands as for I2C (above) but applying to the second I^2C channel.

IF expr THEN stmt [: stmt] or IF expr THEN stmt ELSE stmt	Evaluates the expression ‘expr' and performs the statement following the THEN keyword if it is true or skips to the next line if false. If there are more statements on the line (separated by colons (:) they will also be executed if true or skipped if false. The ELSE keyword is optional and if present the statement(s) following it will be executed if 'expr' resolved to be false. The ‘THEN statement’ construct can be also replaced with: GOTO linenumber | label’. This type of IF statement is all on one line.

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

INPUT ["prompt$";] var1 [,var2 [, var3 [, etc]]]	Will take a list of values separated by commas (,) entered at the console and will assign them to a sequential list of variables. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 The list of variables can be a mix of float, integer or string variables. The values entered at the console must correspond to the type of variable. If a single value is entered a comma is not required (however that value cannot contain a comma). ‘prompt$’ is a string constant (not a variable or expression) and if specified it will be printed first. Normally the prompt is terminated with a semicolon (;) and in that case a question mark will be printed following the prompt. If the prompt is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #nbr, list of variables	Same as above except that the input is read from a serial port or file previously opened for INPUT as ‘nbr’. See the OPEN command.
INTERRUPT [myint]	This command triggers a software interrupt. The interrupt is set up using INTERRUPT ‘myint’ where ‘myint’ is the name of a subroutine that will be executed when the interrupt is triggered. Use INTERRUPT 0 to disable the interrupt Use INTERRUPT without parameters to trigger the interrupt. NB: the interrupt can also be triggered from within a CSUB
IR dev, key , int or IR CLOSE	Decodes NEC or Sony infrared remote control signals. An IR Receiver Module is used to sense the IR light and demodulate the signal. It can be connected to any pin however this pin must be configured in advanced using the command: `SETPIN n, IR` The IR signal decode is done in the background and the program will continue after this command without interruption. 'dev' and 'key' should be numeric variables and their values will be updated whenever a new signal is received ('dev' is the device code transmitted by the remote and 'key' is the key pressed). 'int' is a user defined subroutine that will be called when a new key press is received or when the existing key is held down for auto repeat. In the interrupt subroutine the program can examine the variables 'dev' and 'key' and take appropriate action. The IR CLOSE command will terminate the IR decoder.

	Note that for the NEC protocol the bits in 'dev' and 'key' are reversed. For example, in 'key' bit 0 should be bit 7, bit 1 should be bit 6, etc. This does not affect normal use but if you are looking for a specific numerical code provided by a manufacturer you should reverse the bits. This describes how to do it: http://www.thebackshed.com/forum/forum_posts.asp?TID=8367 See the section *Special Hardware Devices* for more details.
IR SEND pin, dev, key	Generate a 12-bit Sony Remote Control protocol infrared signal. 'pin' is the I/O pin to use. This can be any I/O pin which will be automatically configured as an output and should be connected to an infrared LED. Idle is low with high levels indicating when the LED should be turned on. 'dev' is the device being controlled and is a number from 0 to 31, 'key' is the simulated key press and is a number from 0 to 127. The IR signal is modulated at about 38KHz and sending the signal takes about 25mS.
KEYPAD var, int, r1, r2, r3, r4, c1, c2, c3 [, c4] or KEYPAD CLOSE	Monitor and decode key presses on a 4x3 or 4x4 keypad. Monitoring of the keypad is done in the background and the program will continue after this command without interruption. 'var' should be a numeric variable and its value will be updated whenever a key press is detected. 'int' is a user defined subroutine that will be called when a new key press is received. In the interrupt subroutine the program can examine the variable 'var' and take appropriate action. r1, r2, r3 and r4 are pin numbers used for the four row connections to the keypad and c1, c2, c3 and c4 are the column connections. c4 is optional and is only used with 4x4 keypads. This command will automatically configure these pins as required. On a key press the value assigned to 'var' is the number of a numeric key (e.g. '6' will return 6) or 10 for the * key and 11 for the # key. On 4x4 keypads the number 20 will be returned for A, 21 for B, 22 for C and 23 for D. The KEYPAD CLOSE command will terminate the keypad function and return the I/O pin to a not configured state. See the section *Special Hardware Devices* for more details.
KILL file$	Deletes the file specified by 'file$'. If there is an extension it must be specified.
LET variable = expression	Assigns the value of 'expression' to the variable. LET is automatically assumed if a statement does not start with a command. For example: `Var = 56`
LINE x1, y1, x2, y2 [, LW [, C]]	On the VGA monitor this command will draw a line starting at the coordinates 'x1' and 'y1' and ending at 'x2' and 'y2'. 'LW' is the line's width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal. 'C' is an integer representing the colour and defaults to the current foreground colour. All parameters can be expressed as arrays and the software will plot the number of lines as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', and 'y2' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw' and 'c' can be either arrays or single variables/constants.
LINE INPUT [prompt$,] string-variable$	Reads an entire line from the console input into 'string-variable$'. 'prompt$' is a string constant (not a variable or expression) and if specified it will be printed first. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of 'prompt$'.

LINE INPUT #nbr, string-variable$	Same as above except that the input is read from a serial communications port or a file previously opened for INPUT as ‘nbr’. See the OPEN command.
LIST [fname$] or LIST ALL [fname$]	List a program on the console. LIST on its own will list the program with a pause at every screen full. LIST ALL will list the program without pauses. This is useful if you wish to transfer the program to a terminal emulator on a PC that has the ability to capture its input stream to a file. If the optional ‘fname$’ is specified then that file on the SD card will be listed.
LIST COMMANDS or LIST FUNCTIONS	Lists all valid commands or functions
LOAD file$ [,R]	Loads a program called ‘file$’ from an SD card into program memory. If the optional suffix ,R is added the program will be immediately run without prompting (in this case ‘file$’ must be a string constant). If an extension is not specified “.BAS” will be added to the file name.
LOAD IMAGE file$ [, x] [, y]	Load a bitmapped image from the SD card and display it on the VGA screen. "file$' is the name of the file and 'x' and 'y' are the screen coordinates for the top left hand corner of the image. If the coordinates are not specified the image will be drawn at the top left hand position on the screen. If an extension is not specified “.BMP” will be added to the file name. All types of the BMP format are supported including black and white and true colour 24-bit images.
LOAD JPG file$ [, x] [, y]	Load a jpg image from the SD card and display it on the VGA screen. "file$' is the name of the file and 'x' and 'y' are the screen coordinates for the top left hand corner of the image. If the coordinates are not specified the image will be drawn at the top left hand position on the screen. If an extension is not specified “.JPG” will be added to the file name. Progressive jpg images are not supported.
LOCAL variable [, variables] See DIM for the full syntax.	Defines a list of variable names as local to the subroutine or function. This command uses exactly the same syntax as DIM and will create variables that will only be visible within the subroutine or function. They will be automatically discarded when the subroutine or function exits.
LONGSTRING	The LONGSTRING commands allow for the manipulation of strings longer than the normal MMBasic limit of 255 characters. Variables for holding long strings must be defined as single dimensioned integer arrays with the number of elements set to the number of characters required for the maximum string length divided by eight. The reason for dividing by eight is that each integer in an MMBasic array occupies eight bytes. Note that the long string routines do not check for overflow in the length of the strings. If an attempt is made to create a string longer than a long string variable's size the outcome will be undefined.
LONGSTRING APPEND array%(), string$	Append a normal MMBasic string to a long string variable. array%() is a long string variable while string$ is a normal MMBasic string expression.
LONGSTRING CLEAR array%()	Will clear the long string variable array%(). i.e. it will be set to an empty string.
LONGSTRING COPY dest%(), src%()	Copy one long string to another. dest%() is the destination variable and src%() is the source variable. Whatever was in dest%() will be overwritten.
LONGSTRING CONCAT dest%(), src%()	Concatenate one long string to another. dest%() is the destination variable and src%() is the source variable. src%() will the added to the end of dest%() (the destination will not be overwritten).

LONGSTRING LCASE array%()	Will convert any uppercase characters in array%() to lowercase. array%() must be long string variable.
LONGSTRING LEFT dest%(), src%(), nbr	Will copy the left hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). i.e. copy from the beginning of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING LOAD array%(), nbr, string$	Will copy 'nbr' characters from string$ to the long string variable array%() overwriting whatever was in array%().
LONGSTRING MID dest%(), src%(), start, nbr	Will copy 'nbr' characters from src%() to dest%() starting at character position 'start' overwriting whatever was in dest%(). i.e. copy from the middle of src%(). 'nbr' is optional and if omitted the characters from 'start' to the end of the string will be copied src%() and dest%() must be long string variables. 'start' and 'nbr' must be an integer constants or expressions.
LONGSTRING PRINT [#n,] src%()	Prints the longstring stored in 'src%()' to the file or COM port opened as '#n'. If '#n' is not specified the output will be sent to the console.
LONGSTRING REPLACE array%() , string$, start	Will substitute characters in the normal MMBasic string string$ into an existing long string array%() starting at position 'start' in the long string.
LONGSTRING RESIZE addr%(), nbr	Sets the size of the longstring to nbr. This overrides the size set by other longstring commands so should be used with caution. Typical use would be in using a longstring as a byte array.
LONGSTRING RIGHT dest%(), src%(), nbr	Will copy the right hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). i.e. copy from the end of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING SETBYTE addr%(), nbr, data	Sets byte nbr to the value "data", nbr respects OPTION BASE
LONGSTRING TRIM array%(), nbr	Will trim 'nbr' characters from the left of a long string. array%() must be a long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING UCASE array%()	Will convert any lowercase characters in array%() to uppercase. array%() must be long string variable.
LOOP [UNTIL expression]	Terminates a program loop: see DO.
MATH	The math command performs many simple mathematical calculations that can be programmed in BASIC but there are speed advantages to coding looping structures in the firmware and there is the advantage that once debugged they are there for everyone without re-inventing the wheel. Note: 2 dimensional maths matrices are always specified DIM matrix(n_columns, n_rows) and of course the dimensions respect OPTION BASE. Quaternions are stored as a 5 element array w, x, y, z, magnitude.
Simple array arithmetic	
MATH SET nbr, array()	Sets all elements in array() to the value nbr. Note this is the fastest way of clearing an array by setting it to zero.
MATH SCALE in(), scale ,out()	This scales the matrix in() by the scalar scale and puts the answer in out(). Works for arrays of any dimensionality of both integer and float and can convert between. Setting b to 1 is optimised and is the fastest way of copying an entire array.
MATH ADD in(), num ,out()	This adds the value 'num' to every element of the matrix in() and puts the answer in out(). Works for arrays of any dimensionality of both integer and float and can convert between. Setting num to 0 is optimised and is a fast way of copying an entire array. in() and out() can be the same array.

MATH INTERPOLATE in1(), in2(), ratio, out()	This command implements the following equation on every array element: out = (in2 - in1) * ratio + in1 Arrays can have any number of dimensions and must be distinct and have the same number of total elements. The command works with bot integer and floating point arrays in any mixture

MATH SLICE sourcearray(), [d1] [,d2] [,d3] [,d4] [,d5] , destinationarray()	This command copies a specified set of values from a multi-dimensional array into a single dimensional array. It is much faster than using a FOR loop. The slice is specified by giving a value for all but one of the source array indices and there should be as many indices in the command, including the blank one, as there are dimensions in the source array
	e.g.
	`OPTION BASE 1`
	`DIM a(3,4,5)`
	`DIM b(4)`
	`MATH SLICE a(), 2, , 3, b()`
	Will copy the elements 2,1,3 and 2,2,3 and 2,3,3 and 2,4,3 into array b()

MATH INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5] , sourcearray()	This is the opposite of MATH SLICE, has a very similar syntax, and allows you, for example, to substitute a single vector into an array of vectors with a single instruction
	e.g.
	`OPTION BASE 1`
	`DIM targetarray(3,4,5)`
	`DIM sourcearray(4)=(1,2,3,4)`
	`MATH INSERT targetarray(), 2, , 3, sourcearray()`
	Will set elements 2,1,3 = 1 and 2,2,3 = 2 and 2,3,3 = 3 and 2,4,3 = 4

Matrix arithmetic	
MATH M_INVERSE array!(), inversearray!()	This returns the inverse of array!() in inversearray!(). The array must be square and you will get an error if the array cannot be inverted (determinant=0). array!() and inversearray!() cannot be the same.
MATH M_PRINT array()	Quick mechanism to print a 2D matrix one row per line.
MATH M_TRANSPOSE in(), out()	Transpose matrix in() and put the answer in matrix out(), both arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in(m,n) out(n,m)
MATH M_MULT in1(), in2(), out()	Multiply the arrays in1() and in2() and put the answer in out()c. All arrays must be 2D but need not be square. If not square then the arrays must be dimensioned in1(m,n) in2(p,m) ,out(p,n)
Vector arithmetic	
MATH V_PRINT array()	Quick mechanism to print a small array on a single line
MATH V_NORMALISE inV(), outV()	Converts a vector inV() to unit scale and puts the answer in outV() (sqr(x*x + y*y +.....)=1 There is no limit on number of elements in the vector

MATH V_MULT matrix(), inV(), outV()	Multiplies matrix() and vector inV() returning vector outV(). The vectors and the 2D matrix can be any size but must have the same cardinality.
MATH V_CROSS inV1(), inV2(), outV()	Calculates the cross product of two three element vectors inV1() and inV2() and puts the answer in outV()
Quaternion arithmetic	
MATH Q_INVERT inQ(), outQ()	Invert the quaternion in inQ() and put the answer in outQ()
MATH Q_VECTOR x, y, z, outVQ()	Converts a vector specified by x , y, and z to a normalised quaternion vector outVQ() with the original magnitude stored
MATH Q_CREATE theta, x, y, z, outRQ()	Generates a normalised rotation quaternion outRQ() to rotate quaternion vectors around axis x,y,z by an angle of theta. Theta is specified in radians.
MATH Q_EULER yaw, pitch, roll, outRQ()	Generates a normalised rotation quaternion outRQ() to rotate quaternion vectors as defined by the yaw, pitch and roll angles With the vector in front of the "viewer" yaw is looking from the top of the ector and rotates clockwise, pitch rotates the top away from the camera and roll rotates around the z-axis clockwise. The yaw, pitch and roll angles default to radians but respect the setting of OPTION ANGLE
MATH Q_MULT inQ1(), inQ2(), outQ()	Multiplies two quaternions inQ1() and inQ2() and puts the answer in outQ()
MATH Q_ROTATE , RQ(), inVQ(), outVQ()	Rotates the source quaternion vector inVQ() by the rotate quaternion RQ() and puts the answer in outVQ()
MATH FFT signalarray!(), FFTarray!()	Performs a fast fourier transform of the data in "signalarray!". "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "FFTarray" must be floating point and have dimension 2*N where N is the same as the signal array (e.g. f(1,1023) assuming OPTION BASE is zero) The command will return the FFT as complex numbers with the real part in f(0,n) and the imaginary part in f(1,n)
MATH FFT INVERSE FFTarray!(), signalarray!()	Performs an inverse fast fourier transform of the data in "FFTarray!". "FFTarray" must be floating point and have dimension 2*N where N must be a power of 2 (e.g. f(1,1023) assuming OPTION BASE is zero) with the real part in f(0,n) and the imaginary part in f(1,n). "signalarray" must be floating point and the single dimension must be the same as the FFT array. The command will return the real part of the inverse transform in "signalarray".
MATH FFT MAGNITUDE signalarray!(),magnitudearray!()	Generates magnitudes for frequencies for the data in "signalarray!" "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "magnitudearray" must be floating point and the size must be the same as the signal array The command will return the magnitude of the signal at various frequencies according to the formula: frequency at array position N = N * sample_frequency / number_of_samples
MATH FFT PHASE signalarray!(), phasearray!()	Generates phases for frequencies for the data in "signalarray!". "signalarray" must be floating point and the size must be a power of 2 (e.g.

	s(1023) assuming OPTION BASE is zero) "phasearray" must be floating point and the size must be the same as the signal array The command will return the phase angle of the signal at various frequencies according to the formula above.
MATH SENSORFUSION type ax, ay, az, gx, gy, gz, mx, my, mz, pitch, roll, yaw [,p1] [,p2]	Type can be MAHONY or MADGWICK Ax, ay, and az are the accelerations in the three directions and should be specified in units of standard gravitational acceleration. Gx, gy, and gz are the instantaneous values of rotational speed which should be specified in radians per second. Mx, my, and mz are the magnetic fields in the three directions and should be specified in nano-Tesla (nT) Care must be taken to ensure that the x, y and z components are consistent between the three inputs. So , for example, using the MPU-9250 the correct input will be ax, ay,az, gx, gy, gz, my, mx, -mz based on the reading from the sensor. Pitch, roll and yaw should be floating point variables and will contain the outputs from the sensor fusion. The SENSORFUSION routine will automatically measure the time between consecutive calls and will use this in its internal calculations. The Madwick algorithm takes an optional parameter p1. This is used as beta in the calculation. It defaults to 0.5 if not specified The Mahony algorithm takes two optional parameters p1, and p2. These are used as Kp and Ki in the calculation. If not specified these default to 10.0 and 0.0 respectively. A fully worked example of using the code is given on the BackShed forum at: https://www.thebackshed.com/forum/ViewTopic.php?TID=13459&PID=166962#166962

MEMORY	List the amount of memory currently in use. For example:
	`Program:`
	`  0K ( 0%) Program (0 lines)`
	`108K (100%) Free`
	
	`RAM:`
	`  0K ( 0%) 0 Variables`
	`  0K ( 0%) General`
	`140K (100%) Free`
	Notes:
	• Memory usage is rounded to the nearest 1K byte.
	• General memory is used by serial I/O buffers, etc.

MEMORY SET address, byte, numberofbytes MEMORY SET BYTE address, byte, numberofbytes MEMORY SET SHORT address, short, numberofshorts MEMORY SET WORD address, word, numberofwords	This command will set a region of memory to a value. BYTE = One byte per memory address. SHORT = Two bytes per memory address. WORD = Four bytes per memory address. FLOAT = Eight bytes per memory address. 'increment' is optional and controls the increment of the 'address' pointer as the operation is executed. For example, if increment=3 then only every third element of the target is set. The default is 1.

MEMORY SET INTEGER address, integervalue ,numberofintegers [,increment] MEMORY SET FLOAT address, floatingvalue ,numberofloats [,increment]	
MEMORY COPY sourceaddress, destinationaddres, numberofbytes MEMORY COPY INTEGER sourceaddress, destinationaddress, numberofintegers [,sourceincrement][,destinationincrement] MEMORY COPY FLOAT sourceaddress, destinationaddress, numberoffloats [,sourceincrement][,destinationincrement]	This command will copy one region of memory to another. COPY INTEGER and FLOAT will copy eight bytes per operation. 'sourceincrement' is optional and controls the increment of the 'sourceaddress' pointer as the operation is executed. For example, if sourceincrement=3 then only every third element of the source will be copied. The default is 1. 'destinationincrement' is similar and operates on the 'destinationaddress' pointer.
MKDIR dir$	Make, or create, the directory 'dir$' on the SD card.
MID$(str$, start [, num]) = str2$	The characters in 'str$', beginning at position 'start', are replaced by the characters in 'str2$'. The optional 'num' refers to the number of characters from 'str2' that are used in the replacement. If 'num' is omitted, all of 'str2' is used. Whether 'num' is omitted or included, the replacement of characters never goes beyond the original length of 'str$'.
MODE 1 or MODE 2	Switches between 640x480 monochrome (MODE 1) and 320x240 4-bit colour (MODE 2). This can be used in programs or at the command prompt. On reboot the mode is reset to that defined with the OPTION DEFAULT MODE command.
NEW	Erases the current program and clears all non permanent options and variables including saved variables. This command resets the interpreter to a known state.
NEXT [counter-variable] [, counter-variable], etc	NEXT comes at the end of a FOR-NEXT loop; see FOR. The 'counter-variable' specifies exactly which loop is being operated on. If no 'counter-variable' is specified the NEXT will default to the innermost loop. It is also possible to specify multiple counter-variables as in: `NEXT x, y, z`
ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR	This controls the action taken if an error occurs while running a program and applies to all errors discovered by MMBasic including syntax errors, wrong data, missing hardware, etc. ON ERROR ABORT will cause MMBasic to display an error message, abort the program and return to the command prompt. This is the normal behaviour and is the default when a program starts running. ON ERROR IGNORE will cause any error to be ignored. ON ERROR SKIP will ignore an error in a number of commands (specified by the number 'nn') executed following this command. 'nn' is optional, the

	default if not specified is one. After the number of commands has completed (with an error or not) the behaviour of MMBasic will revert to ON ERROR ABORT. If an error occurs and is ignored/skipped the read only variable MM.ERRNO will be set to non zero and MM.ERRMSG$ will be set to the error message that would normally be generated. These are reset to zero and an empty string by ON ERROR CLEAR. They are also cleared when the program is run and when ON ERROR IGNORE and ON ERROR SKIP are used. ON ERROR IGNORE can make it very difficult to debug a program so it is strongly recommended that only ON ERROR SKIP be used.
ON KEY target or ON KEY ASCIIcode, target	The first version of the command sets an interrupt which will call 'target' user defined subroutine whenever there is one or more characters waiting in the console input buffer. Note that all characters waiting in the input buffer should be read in the interrupt subroutine otherwise another interrupt will be automatically generated as soon as the program returns from the interrupt. The second version allows you to associate an interrupt routine with a specific key press. This operates at a low level for the console and if activated the key does not get put into the input buffer but merely triggers the interrupt. It uses a separate interrupt from the simple ON KEY command so can be used at the same time if required. In both variants, to disable the interrupt use numeric zero for the target, i.e.: ON KEY 0. or ON KEY ASCIIcode, 0
ONEWIRE RESET pin or ONEWIRE WRITE pin, flag, length, data [, data…] or ONEWIRE READ pin, flag, length, data [, data…]	Commands for communicating with 1-Wire devices. ONEWIRE RESET will reset the 1-Wire bus ONEWIRE WRITE will send a number of bytes ONEWIRE READ will read a number of bytes 'pin' is the I/O pin (located in the rear connector) to use. It can be any pin capable of digital I/O. 'flag' is a combination of the following options: 1 - Send reset before command 2 - Send reset after command 4 - Only send/recv a bit instead of a byte of data 8 - Invoke a strong pullup after the command (the pin will be set high and open drain disabled) 'length' is the length of data to send or receive 'data' is the data to send or variable to receive. The number of data items must agree with the length parameter. See also *Appendix C*.
OPEN fname$ FOR mode AS [#]fnbr	Opens a file for reading or writing. ‘fname’ is the filename with an optional extension separated by a dot (.). Long file names with upper and lower case characters are supported. A directory path can be specified with the backslash as directory separators. The parent of the current directory can be specified by using a directory name of .. (two dots) and the current directory with . (a single dot). For example OPEN "..\dir1\dir2\filename.txt" FOR INPUT AS #1 ‘mode’ is INPUT, OUTPUT, APPEND or RANDOM. INPUT will open the file for reading and throw an error if the file does not exist. OUTPUT will open the file for writing and will automatically overwrite any existing file with the same name. APPEND will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the APPEND mode will act the same as the

	OUTPUT mode (i.e. the file is created then opened for writing). RANDOM will open the file for both read and write and will allow random access using the SEEK command. When opened the read/write pointer is positioned at the end of the file. 'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT, WRITE and CLOSE commands as well as the EOF() and INPUT$() functions all use 'fnbr' to identify the file being operated on. See also ON ERROR and MM.ERRNO for error handling.
OPEN comspec$ AS [#]fnbr	Will open a serial communications port for reading and writing. Two ports are available (COM1: and COM2:) and both can be open simultaneously. For a full description with examples see Appendix A. Using 'fnbr' the port can be written to and read from using any command or function that uses a file number.
OPEN comspec$ AS GPS [,timezone_offset] [,monitor]	Will open a serial communications port for reading from a GPS receiver. See the GPS function for details. The sentences interpreted are GPRMC, GNRMC, GPCGA and GNCGA. The timezone_offset parameter is used to convert UTC as received from the GPS to the local timezone. If omitted the timezone will default to UTC. The timezone_offset can be a any number between -12 and 14 allowing the time to be set correctly even for the Chatham Islands in New Zealand (UTC +12:45). If the monitor parameter is set to 1 then all GPS input is directed to the console. This can be stopped by closing the GPS channel.
OPTION	See the section *Options* earlier in this manual.
PAUSE delay	Halt execution of the running program for 'delay' ms. This can be a fraction. For example, 0.2 is equal to 200 µs. The maximum delay is 2147483647 ms (about 24 days). Note that interrupts will be recognised and processed during a pause.
PIN(pin) = value	For a 'pin' configured as digital output this will set the output to low ('value' is zero) or high ('value' non-zero). You can set an output high or low before it is configured as an output and that setting will be the default output when the SETPIN command takes effect. See the function PIN() for reading from a pin and the command SETPIN for configuring it. Refer to the chapter *"Using the I/O pins"* for a general description of the PicoMiteVGA 's input/output capabilities.
PIO	The RP2040 chip used in the PicoMiteVGA contains a programmable I/O system with two identical PIO devices (pio%=0 or pio%=1) acting like specialised CPU cores. See the Appendix for a more detailed description. NB: on the PicoMiteVGA only PIO1 is available to MMbasic
PIO INIT MACHINE pio%, statemachine%, clockspeed [,pinctrl] [,execctrl] [,shiftctrl] [,startinstruction]	Initialises PIO 'pio%' with state machine 'statemachine%'. 'clockspeed' is the clock speed of the state machine in kHz. The four optional arguments are variables holding initialising values of the state machine registers and the address of the first instruction to execute (defaults to zero). These decide how the PIO will operate. It is anticipated that eventually the PIO assembler will be able to generate the register values for the user along with the program array based on the defined assembler directives.
PIO EXECUTE pio, state_machine, instruction%	Immediately executes the instruction on the pio and state machine specified.

PIO WRITE pio, state_machine, count, data0 [,data1....]	Writes the data elements to the pio and state machine specified. The write is blocking so the state machine needs to be able to take the data supplied NB: this command will probably need additional capability in future releases
PIO READ pio, state_machine, count, data%()	Reads the data elements from the pio and state machine specified. The read is non-blocking so the state machine needs to be able to supply the data requested. NB: this command will probably need additional capability in future releases
PIO START pio, statemachine	Start a given state machine on pio
PIO STOP pio, statemachine	Stop a given state machine on pio
PIO CLEAR pio	This stops the pio specified on all statemachines and clears the control registers for the statemachines PINCTRL, EXECTRL, and SHIFTCTRL to defaults
PIO PROGRAM LINE pio, line, instruction	Programs just the specified line in a PIO program
PIXEL x, y [,c]	Set a pixel on the VGA screen to a colour. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. 'c' is a 24 bit number specifying the colour. 'c' is optional and if omitted the current foreground colour will be used. All parameters can be expressed as arrays and the software will plot the number of pixels as determined by the dimensions of the smallest array. 'x' and 'y' must both be arrays or both be single variables /constants otherwise an error will be generated. 'c' can be either an array or a single variable or constant. See the section *Graphics Commands and Functions* for a definition of the colours and graphics coordinates.
PLAY	This command will generate a variety of audio outputs. See the OPTION AUDIO command for setting the I/O pins to be used for the output. The audio is a pulse width modulated signal so a low pass filter is required to remove the carrier frequency.
PLAY TONE left [, right [, dur]]	Generates two separate sine waves on the sound output left and right channels. 'left' and 'right' are the frequencies in Hz to use for the left and right channels. The tone plays in the background (the program will continue running after this command) and 'dur' specifies the number of milliseconds that the tone will sound for. If the duration is not specified the tone will continue until explicitly stopped or the program terminates. The frequency can be from 1Hz to 20KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command.
PLAY WAV file$ [, interrupt]	Will play a WAV file on the sound output. 'file$' is the WAV file to play (the extension of .wav will be appended if missing). The WAV file must be PCM encoded in stereo with 8-bit sampling. The sample rate can be up to 48kHz in stereo. The WAV file is played in the background. 'interrupt' is optional and is the name of a subroutine which will be called when the file has finished playing.
PLAY SOUND soundno, channelno, type [,frequency] [,volume]	Play a series of sounds simultaneously on the audio output. 'soundno' is the sound number and can be from 1 to 4 allowing for four simultaneous sounds on each channel. 'channelno' specifies the output channel. It can be L (left speaker), R (right speaker) or B (both speakers) 'type' is the type of waveform. It can be S (sine wave), Q (square wave), T (triangle wave), W (rising sawtooth) or O (turn off sound).

	'frequency' is the frequency from 1 to 20000 (Hz) and it must be specified except when type is O. 'volume' is optional and must be between 1 and 25. It defaults to 25 The first time PLAY SOUND is called all other audio usage will be blocked and will remain blocked until PLAY STOP is called. Output can be stopped temporarily using PLAY PAUSE and PLAY RESUME. Calling SOUND on an already running 'soundno' will immediately replace the previous output. Individual sounds are turned off using type "O" Running 4 sounds simultaneously on both channels of the audio output consumes about 23% of the CPU.
PLAY PAUSE	PLAY PAUSE will temporarily halt the currently playing file or tone.
PLAY RESUME	PLAY RESUME will resume playing a sound that was paused.
PLAY STOP	PLAY STOP will terminate the playing of the file or tone. When the program terminates for whatever reason the sound output will also be automatically stopped.
PLAY VOLUME left, right	Will adjust the volume of the audio output. 'left' and 'right' are the levels to use for the left and right channels and can be between 0 and 100 with 100 being the maximum volume. There is a linear relationship between the specified level and the output. The volume defaults to maximum when a program is run.
POKE BYTE addr%, byte or POKE SHORT addr%, short% Or POKE WORD addr%, word% or POKE INTEGER addr%, int% or POKE FLOAT addr%, float! or POKE VAR var, offset, byte or POKE VARTBL, offset, byte or POKE DISPLAY HRES n POKE DISPLAY VRES n	Will set a byte or a word within the PIC32 virtual memory space. POKE BYTE will set the byte (i.e. 8 bits) at the memory location 'addr%' to 'byte'. 'addr%' should be an integer. POKE SHORT will set the short integer (i.e. 16 bits) at the memory location 'addr%' to 'word%'. 'addr%' and short%' should be integers. POKE WORD will set the word (i.e. 32 bits) at the memory location 'addr%' to 'word%'. 'addr%' and 'word%' should be integers. POKE INTEGER will set the MMBasic integer (i.e. 64 bits) at the memory location 'addr%' to int%'. 'addr%' and int%' should be integers. POKE FLOAT will set the word (i.e. 32 bits) at the memory location 'addr%' to 'float!'. 'addr%' should be an integer and 'float!' a floating point number. POKE VAR will set a byte in the memory address of 'var'. 'offset' is the ±offset from the address of the variable. An array is specified as var(). POKE VARTBL will set a byte in MMBasic's variable table. 'offset' is the ±offset from the start of the variable table. Note that a comma is required after the keyword VARTBL. These commands change the stored value of MM.HRES anmd MM.VRES allowing the programmer to configure non-standard displays.
POLYGON n, xarray%(), yarray%() [, bordercolour] [, fillcolour] POLYGON n(), xarray%(), yarray%() [, bordercolour()] [, fillcolour()] POLYGON n(), xarray%(), yarray%() [, bordercolour] [, fillcolour]	Draws a filled or outline polygon with n xy-coordinate pairs in xarray%() and yarray%(). If 'fillcolour' is omitted then just the polygon outline is drawn. If 'bordercolour' is omitted then it will default to the current default foreground colour. If the last xy-coordinate pair is not the same as the first the firmware will automatically create an additional xy-coordinate pair to complete the polygon. The size of the arrays should be at least as big as the number of x,y coordinate pairs. 'n' can be an array and the colours can also optionally be arrays as follows: POLYGON n(), xarray%(), yarray%() [, bordercolour()] [, fillcolour()] POLYGON n(), xarray%(), yarray%() [, bordercolour] [, fillcolour] The elements of array n() define the number of xy-coordinate pairs in each of the polygons. e.g. DIM n(1)=(3,3) would define that 2 polygons are to be

	drawn with three vertices each. The size of the n array determines the number of polygons that will be drawn unless an element is found with the value zero in which case the firmware only processes polygons up to that point. The x,y-coordinate pairs for all the polygons are stored in xarray%() and yarray%(). The xarray%() and yarray%() parameters must have at least as many elements as the total of the values in the n array.
	Each polygon can be closed with the first and last elements the same. If the last element is not the same as the first the firmware will automatically create an additional x,y-coordinate pair to complete the polygon. If fill colour is omitted then just the polygon outlines are drawn.
	The colour parameters can be a single value in which case all polygons are drawn in the same colour or they can be arrays with the same cardinality as n. In this case each polygon drawn can have a different colour of both border and/or fill. For example, this will draw 3 triangles in yellow, green and red:
	`DIM c%(2)=(3,3,3)`
	`DIM x%(8)=(100,50,150,100,50,150,100,50,150)`
	`DIM y%(8)=(50,100,100,150,200,200,250,300,300)`
	`DIM fc%(2)=(rgb(yellow),rgb(green),rgb(red))`
	`POLYGON c%(),x%(),y%(),fc%(),fc%()`

PORT(start, nbr [,start, nbr]…) = value	Set a number of I/O pins simultaneously (i.e. with one command). 'start' is an I/O pin number and the lowest bit in 'value' (bit 0) will be used to set that pin. Bit 1 will be used to set the pin 'start' plus 1, bit 2 will set pin 'start'+2 and so on for 'nbr' number of bits. I/O pins used must be numbered consecutively and any I/O pin that is invalid or not configured as an output will cause an error. The start/nbr pair can be repeated if an additional group of output pins needed to be added. For example; `PORT(15, 4, 23, 4) = &B10000011` Will set eight I/O pins. Pins 15 and 16 will be set high while 17, 18, 23, 24 and 25 will be set to a low and finally 26 will be set high. This command can be used to conveniently communicate with parallel devices like LCD displays. Any number of I/O pins (and therefore bits) can be used from 1 to the number of I/O pins on the chip. See the PORT function to simultaneously read from a number of pins.
PRINT expression [[,;]expression] … etc	Outputs text to the console followed by a carriage return/newline pair. Multiple expressions can be used and must be separated by either a: • Comma (,) which will output the tab character • Semicolon (;) which will not output anything (it is just used to separate expressions). • Nothing or a space which will act the same as a semicolon. A semicolon (;) or a comma (,) at the end of the expression list will suppress the output of the carriage return/newline pair at the end of a print statement. When printed, a number is preceded with a space if positive or a minus (-) if negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while floating point is printed with the decimal point and the significant decimal digits. Large or small floating point numbers are automatically printed in scientific number format. The function TAB() can be used to space to a certain column and the STR$() function can be used to justify or otherwise format strings.
PRINT #nbr, expression [[,;]expression] … etc	Same as above except that the output is directed to a serial communications port or a file opened for OUTPUT or APPEND with a file number of 'nbr'. See the OPEN command.
PRINT #GPS, expression [[,;]expression] … etc	Outputs a NMEA string to an opened GPS device. The string must start with a $ character and end with a * character. The checksum is automatically calculated and appended to the string together with the CR/LF characters.

PRINT @(x [, y]) expression Or PRINT @(x, [y], m) expression	Same as the standard PRINT command except that the cursor is positioned at the coordinates x, y expressed in pixels. If y is omitted the cursor will be positioned at "x" on the current line. Example: PRINT @(150, 45) "Hello World" The @ function can be used anywhere in a print command. Example: PRINT @(150, 45) "Hello" @(150, 55) "World" The @(x,y) function can be used to position the cursor anywhere on or off the screen. For example, PRINT @(-10, 0) "Hello" will only show "llo" as the first two characters could not be shown because they were off the screen. The @(x,y) function will automatically suppress the automatic line wrap normally performed when the cursor goes beyond the right screen margin. If 'm' is specified the mode of the video operation will be as follows: m = 0 Normal text (white letters, black background) m = 1 The background will not be drawn (ie, transparent) m = 2 The video will be inverted (black letters, white background) m = 5 Current pixels will be inverted (transparent background)
PULSE pin, width	Will generate a pulse on 'pin' with duration of 'width' ms. 'width' can be a fraction. For example, 0.01 is equal to 10µs and this enables the generation of very narrow pulses. The minimum is 5 µs at 40 MHz to 40 µs at 5 MHz. The generated pulse is of the opposite polarity to the state of the I/O pin when the command is executed. For example, if the output is set high the PULSE command will generate a negative going pulse. Notes: • 'pin' must be configured as an output. • For a pulse of less than 3 ms the accuracy is ± 1 µs. • For a pulse of 3 ms or more the accuracy is ± 0.5 ms. • A pulse of 3 ms or more will run in the background. Up to five different and concurrent pulses can be running in the background and each can have its time changed by issuing a new PULSE command or it can be terminated by issuing a PULSE command with zero for 'width'.
PWM channel, frequency, [dutyA] [,dutyB] PWM channel, OFF	There are 8 separate PWM frequencies available (channels 0 to 7) and up to 16 outputs with individually controlled duty cycle. You can output on either PWMnA or PWMnB or both for each channel - no restriction. Minimum frequency is 15Hz. Maximum speed is OPTION CPUSPEED/4 At very fast speeds the duty cycles will be increasingly limited
RANDOMIZE nbr	Seed the random number generator with 'nbr'. On power up the random number generator is seeded with zero and will generate the same sequence of random numbers each time. To generate a different random sequence each time you must use a different value for 'nbr' (the TIMER function is handy for returning an unpredictable number).
RBOX x, y, w, h [, r] [,c] [,fill]	Draws a box with rounded corners on the VGA screen starting at 'x' and 'y' which is 'w' pixels wide and 'h' pixels high. 'r' is the radius of the corners of the box. It defaults to 10. 'c' specifies the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'r', 'c', and 'fill' can be either arrays or single variables/constants. See the section *Graphics Commands and Functions* for a definition of the colours and graphics coordinates.

READ variable[, variable]...	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. Arrays can be used as variables (specified with empty brackets, e.g. a()) and in that case the size of the array is used to determine how many elements are to be read. If the array is multidimensional then the leftmost dimension will be the fastest moving. See also DATA and RESTORE.
READ SAVE or READ RESTORE	READ SAVE will save the virtual pointer used by the READ command to point to the next DATA to be read. READ RESTORE will restore the pointer that was previously saved. This enables subroutines to READ data and then restore the read pointer so as not to disturb other parts of the program that may be reading the same data statements. These commands can be nested.
REM string	REM allows remarks to be included in a program. Note the Microsoft style use of the single quotation mark (‘) to denote remarks is also supported and is preferred.
RENAME old$ AS new$	Rename a file or a directory from ‘old$’ to ‘new$’. Both are strings. A directory path can be used in both 'old$' and 'new$'. If the paths differ the file specified in 'old$' will be moved to the path specified in 'new$' with the file name as specified.
RESTORE [line]	Resets the line and position counters for the READ statement. If ‘line’ is specified the counters will be reset to the beginning of the specified line. ‘line’ can be a line number or label or a variable with these values. If ‘line’ is not specified the counters will be reset to the start of the program.
RMDIR dir$	Remove, or delete, the directory ‘dir$’ on the SD card.
RTC GETTIME RTC SETTIME year, month, day, hour, minute, second RTC SETREG reg, value RTC GETREG reg, var	RTC GETTIME will get the current date/time from a PCF8563, DS1307 or DS3231 real time clock and set the internal MMBasic clock accordingly. The date/time can then be retrieved with the DATE$ and TIME$ functions. RTC SETTIME will set the time in the clock chip. 'hour' must use 24 hour notation. The RTC SETTIME command will also accept a single string argument in the format of *dd/mm/yy hh:mm*. The RTC SETREG and GETREG commands can be used to set or read the contents of registers within the chip. 'reg' is the register's number, 'value' is the number to store in the register and 'var' is a variable that will receive the number read from the register. These commands are not necessary for normal operation but they can be used to manipulate special features of the chip (alarms, output signals, etc). They are also useful for storing temporary information in the chip's battery backed RAM. These chips are I^2C devices and must be connected to the two I^2C pins with appropriate pullup resistors. If the I^2C bus is already open the RTC command will use the current settings, otherwise it will temporarily open the connection with a speed of 100 kHz. Also see the command OPTION RTC AUTO ENABLE.
RUN or RUN[file$]	Run the program held in program memory or saved to an SD card as ‘file$’. ‘file$’ must be a string constant, ie, surrounded by double quotes. Use FLASH RUN *n* to run a program stored in flash.

SAVE file$	Saves the program in the current working directory of the SD card as 'file$'. Example: SAVE "TEST.BAS" If an extension is not specified ".BAS" will be added to the file name. See also FLASH SAVE *n*
SAVE IMAGE file$ [,x, y, w, h]	Save the current image on the VGA monitor as a BMP file. 'file$' is the name of the file. If an extension is not specified ".BMP" will be added to the file name. The image is saved as a true colour 24-bit image. 'x', 'y', 'w' and 'h' are optional and are the coordinates (x and y are the top left coordinate) and dimensions (width and height) of the area to be saved. If not specified the whole screen will be saved.
SEEK [#]fnbr, pos	Will position the read/write pointer in a file that has been opened on the SD card for RANDOM access to the 'pos' byte. The first byte in a file is numbered one so SEEK #5,1 will position the read/write pointer to the start of the file.

SELECT CASE value
CASE testexp [[, testexp] …]
<statements>
<statements>
CASE ELSE
<statements>
<statements>
END SELECT
	Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression.
	'testexp' is the value that is to be compared against. It can be:
	• A single expression (i.e. 34, "string" or PIN(4)*5) to which it may equal
	• A range of values in the form of two single expressions separated by the keyword "TO" (i.e. 5 TO 9 or "aa" TO "cc")
	• A comparison starting with the keyword "IS" (which is optional). For example: IS > 5, IS <= 10.
	When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true.
	If 'value' cannot be matched with a 'testexp' it will be automatically matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any <statements> and continue with the code following the END SELECT. When a match is made the <statements> following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT.
	An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT.
	Example:
	`SELECT CASE nbr%`
	`  CASE 4, 9, 22, 33 TO 88`
	`  statements`
	`  CASE IS < 4, IS > 88, 5 TO 8`
	`  statements`
	`  CASE ELSE`
	`  statements`
	`END SELECT`
	Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT…CASE statements can be nested inside the CASE statements of other SELECT…CASE statements.

SETPIN pin, cfg [, option]	Will configure an external I/O pin. Refer to the chapter *"Using the I/O pins"* for a general description of the PicoMiteVGA's input/output capabilities. 'pin' is the I/O pin to configure, 'cfg' is the mode that the pin is to be set to and 'option' is an optional parameter. 'cfg' is a keyword and can be any one of the following: OFF — Not configured or inactive AIN — Analog input (i.e. measure the voltage on the input). DIN — Digital input If 'option' is omitted the input will be high impedance If 'option' is the keyword "PULLUP" a simulated resistor will be used to pull up the input pin to 3.3V If the keyword "PULLDOWN" is used the pin will be pulled down to zero volts. The pull up/down is a constant current of about 50µA. FIN — Frequency input 'option' can be used to specify the gate time (the length of time used to count the input cycles). It can be any number between 10 ms and 100000 ms. Note that the PIN() function will always return the frequency correctly scaled in Hz regardless of the gate time used. If 'option' is omitted the gate time will be 1 second. PIN — Period input 'option' can be used to specify the number of input cycles to average the period measurement over. It can be any number between 1 and 10000. Note that the PIN() function will always return the average period of one cycle correctly scaled in ms regardless of the number of cycles used for the average. If 'option' is omitted the period of just one cycle will be used. CIN — Counting input 'option' can be used to specify which edge triggers the count. 1 specifies a rising edge, 2 a falling edge and 3 specifies that both edges should be counted. If 'option' is omitted a rising edge will trigger the count DOUT — Digital output 'option' can be "OC" in which case the output will be open collector (or more correctly open drain). The functions PIN() and PORT() can also be used to return the value on one or more output pins . See the function PIN() for reading inputs and the statement PIN()= for setting an output. See the command below if an interrupt is configured.
SETPIN pin, cfg, target [, option]	Will configure 'pin' to generate an interrupt according to 'cfg'. Any I/O pin capable of digital input can be configured to generate an interrupt with a maximum of ten interrupts configured at any one time. 'cfg' is a keyword and can be any one of the following: OFF — Not configured or inactive INTH — Interrupt on low to high input INTL — Interrupt on high to low input INTB — Interrupt on both (i.e. any change to the input) 'target' is a user defined subroutine which will be called when the event happens. Return from the interrupt is via the END SUB or EXIT SUB commands. 'option' can be the keywords "PULLUP" or "PULLDOWN" as specified for a normal input pin (SETPIN pin DIN). If 'option' is omitted the input will be high impedance. This mode also configures the pin as a digital input so the value of the pin can always be retrieved using the function PIN(). Refer to the chapter "Using the I/O pins" for a general description of the PicoMiteVGA's input/output capabilities.

SETPIN GP25, DOUT | HEARTBEAT	This version of SETPIN controls the on-board LED. If it is configured as DOUT then it can be switched on and off under program control. If configured as HEARTBEAT then it will flash 1s on, 1s off continually while powered. This is the default state and will be restored to this when the user program stops running.
SETPIN p1[, p2 [, p3]], device	These commands are used for the pin allocation for special devices. Pins must be chosen from the pin designation diagram and must be allocated before the devices can be used. Note that the pins (e.g. rx, tx, etc) can be declared in any order and that the pins can be referred to by using their pin number (e.g. 1, 2) or GP number (e.g. GP0, GP1).
SETPIN rx, tx, COM1	Allocate the pins to be used for serial port COM1. Valid pins are RX: GP1 or GP13 TX: GP0, GP12 or GP28
SETPIN rx, tx, COM2	Allocate the pins to be used for serial port COM2. Valid pins are RX: GP5 TX: GP4
SETPIN rx, tx, clk, SPI	Allocate the pins to be used for SPI port SPI. Valid pins are RX: GP0 or GP4 TX: GP3 or GP7 CLK: GP2 or GP6
SETPIN rx, tx, clk, SPI2	Allocate the pins to be used for SPI port SPI2. Valid pins are RX: GP12 or GP28 TX: GP11, GP15 or GP27 CLK: GP10, GP14 or GP26
SETPIN sda, scl, I2C	Allocate the pins to be used for the I^2C port I2C. Valid pins are SDA: GP0, GP4, GP12 or GP28 SCL: GP1, GP5 or GP13
SETPIN sda, scl, I2C2	Allocate the pins to be used for the I^2C port I2C2. Valid pins are SDA: GP2, GP6, GP10, GP14, GP22 or GP26 SCL: GP3, GP7, GP11, GP15 or GP27
SETPIN pin, PWMnx	Allocate pin to PWMnx 'n' is the PWM number (0 to 7) and 'x' and is the channel (A or B) The setpin can be changed until the PWM command is issued. At that point the pin becomes locked to PWM until PWMn,OFF is issued.
SETPIN pin, IR	Allocate pins for InfraRed (IR) communications (can be any pin).
SETPIN pin, PIO1	Reserve pin for use by PIO1 (see Appendix E for PIO details).
SETTICK period, target [, nbr]	This will setup a periodic interrupt (or "tick"). Four tick timers are available ('nbr' = 1, 2, 3 or 4). 'nbr' is optional and if not specified timer number 1 will be used. The time between interrupts is ‘period’ milliseconds and ‘target' is the interrupt subroutine which will be called when the timed event occurs. The period can range from 1 to 2147483647 ms (about 24 days). These interrupts can be disabled by setting ‘period’ to zero (i.e. SETTICK 0, 0, 3 will disable tick timer number 3).

SETTICK PAUSE, target [, nbr] or SETTICK RESUME, target [, nbr]	Pause or resume the specified tick timer. When paused the interrupt is delayed but the current count is maintained.
SORT array() [,indexarray()] [,flags] [,startposition] [,elementstosort]	This command takes an array of any type (integer, float or string) and sorts it into ascending order in place. It has an optional parameter 'indexarray%()'. If used this must be an integer array of the same size as the array to be sorted. After the sort this array will contain the original index position of each element in the array being sorted before it was sorted. Any data in the array will be overwritten. This allows connected arrays to be sorted. See the section Sorting Data in the tutorial Programming with the Colour Maximite 2 for an example. The 'flag' parameter is optional and valid flag values are: bit0: 0 (default if omitted) normal sort - 1 reverse sort bit1: 0 (default) case dependent - 1 sort is case independent (strings only). The optional 'startposition' defines which element in the array to start the sort. Default is 0 (OPTION BASE 0) or 1 (OPTION BASE 1) The optional 'elementstosort' defines how many elements in the array should be sorted. The default is all elements after the startposition. Any of the optional parameters may be omitted so, for example, to sort just the first 50 elements of an array you could use: `SORT array(), , , ,50`
SPI OPEN speed, mode, bits or SPI READ nbr, array() or SPI WRITE nbr, data1, data2, data3, … etc or SPI WRITE nbr, string$ or SPI WRITE nbr, array() or SPI CLOSE	Communications via an SPI channel. See Appendix D for the details. 'nbr' is the number of data items to send or receive 'data1', 'data2', etc can be float or integer and in the case of WRITE can be a constant or expression. If 'string$' is used 'nbr' characters will be sent. 'array' must be a single dimension float or integer array and 'nbr' elements will be sent or received.
SPI2	The same set of commands as for SPI (above) but applying to the second SPI channel..
STATIC variable [, variables] See DIM for the full syntax.	Defines a list of variable names which are local to the subroutine or function. These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command). This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters. Static variables can be initialised to a value. This initialisation will take effect only on the first call to the subroutine (not on subsequent calls).
SUB xxx (arg1 [,arg2, …]) <statements> <statements> END SUB	Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program. 'xxx' is the subroutine name and it must meet the specifications for naming a variable. 'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. i.e. arg3(). The type of the argument can be specified by using a type suffix (i.e. arg1$) or by specifying the type

	using `AS <type> (i.e. arg1 AS STRING).` Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: `MySub a1, a2` When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable and have the correct type will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (e.g. arg()) and are always passed by reference. Brackets around the argument list in both the caller and the definition are optional.
TEMPR START pin [, precision]	This command can be used to start a conversion running on a DS18B20 temperature sensor connected to 'pin'. Normally the TEMPR() function alone is sufficient to make a temperature measurement so usage of this command is optional. This command will start the measurement on the temperature sensor. The program can then attend to other duties while the measurement is running and later use the TEMPR() function to get the reading. If the TEMPR() function is used before the conversion time has completed the function will wait for the remaining conversion time before returning the value. Any number of these conversions (on different pins) can be started and be running simultaneously. 'precision' is the resolution of the measurement and is optional. It is a number between 0 and 3 meaning: 0 = 0.5ºC resolution, 100 ms conversion time. 1 = 0.25ºC resolution, 200 ms conversion time (this is the default). 2 = 0.125ºC resolution, 400 ms conversion time. 3 = 0.0625ºC resolution, 800 ms conversion time.
TEXT x, y, string$ [,alignment$] [, font] [, scale] [, c] [, bc]	Displays a string on the VGA screen starting at 'x' and 'y'. ‘string$’ is the string to be displayed. Numeric data should be converted to a string and formatted using the Str$() function. ' alignment$' is a string expression or string variable consisting of 0, 1 or 2 letters where the first letter is the horizontal alignment around 'x' and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around 'y' and can be T, M or B for TOP, MIDDLE, BOTTOM. The default alignment is left/top. For example. “CM” will centre the text vertically and horizontally. The 'alignment$' string can be a constant (e.g. “CM”) or it can be a string variable. For backwards compatibility with earlier versions of MMBasic the string can also be unquoted (e.g. CM). A third letter can be used in the alignment string to specify the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (i.e. upside down), 'U' the text will be rotated counter clockwise by 90º and 'D' the text will be rotated clockwise by 90º 'font' and 'scale' are optional and default to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional

	and default to the current foreground and background colours. See the section *Graphics Commands and Functions* for a definition of the colours and graphics coordinates.
TILE x, y [,foreground] [,background] [,nbr_tiles_wide] [,nbr_tiles_high]	Sets the colour for one or more tiles on the screen. When in monochrome mode the VGA screen is split up into 40x30 tiles each 16x16 pixels. Each tile can have a different foreground and background named colour assigned to it from the following: white, yellow, lilac, brown, fuchsia, rust, magenta, red, cyan, green, cerulean, midgreen, cobalt, myrtle, blue and black. 'x' and 'y' are the coordinates of the block in pixels. 'foreground ' and 'background' are the new colours selected. 'nbr_tiles_wide' and 'nbr_tiles_high' are the number of tiles to change. The change is instant and does not affect the text or graphics currently displayed in the tiles (just the colours).
TIME$ = "HH:MM:SS" or TIME$ = "HH:MM" or TIME$ = "HH"	Sets the time of the internal clock. MM and SS are optional and will default to zero if not specified. For example TIME$ = "14:30" will set the clock to 14:30 with zero seconds. The time is set to "00:00:00" on power up.
TIMER = msec	Resets the timer to a number of milliseconds. Normally this is just used to reset the timer to zero but you can set it to any positive integer. See the TIMER function for more details.
TRACE ON or TRACE OFF or TRACE LIST nn	TRACE ON/OFF will turn on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. TRACE LIST will list the last 'nn' lines executed in the format described above. MMBasic is always logging the lines executed so this facility is always available (i.e. it does not have to be turned on).
TRIANGLE X1, Y1, X2, Y2, X3, Y3 [, C [, FILL]]	Draws a triangle on the VGA screen with the corners at X1, Y1 and X2, Y2 and X3, Y3. 'C' is the colour of the triangle and defaults to the current foreground colour. 'FILL' is the fill colour and defaults to no fill (it can also be set to -1 for no fill). All parameters can be expressed as arrays and the software will plot the number of triangles as determined by the dimensions of the smallest array unless X1 = Y1 = X2 = Y2 = X3 = Y3 = -1 in which case processing will stop at that point 'x1', 'y1', 'x2', 'y2', 'x3',and 'y3' must all be arrays or all be single variables /constants otherwise an error will be generated 'c' and 'fill' can be either arrays or single variables/constants.
UPDATE FIRMWARE	Causes the PicoMiteVGA to enter the firmware update mode (the same as applying power while holding down the BOOTSEL button). Loading the PicoMiteVGA firmware may erase the flash memory including the current program, any programs saved in flash memory slots and all saved variables. So make sure that you backup this data before you upgrade the firmware. A firmware load may also reset all options to their defaults.
VAR SAVE var [, var]… or VAR RESTORE or VAR CLEAR	VAR SAVE will save one or more variables to non-volatile flash memory where they can be restored later (normally after a power interruption). 'var' can be any number of numeric or string variables and/or arrays. Arrays are specified by using empty brackets. For example: var() VAR RESTORE will retrieve the previously saved variables and insert them (and their values) into the variable table. The VAR SAVE command can be used repeatedly. Variables that had been

	previously saved will be updated with their new value and any new variables (not previously saved) will be added to the saved list for later restoration. VAR CLEAR will erase all saved variables. Also, the saved variables will be automatically cleared by a firmware upgrade, by the NEW command or when a new program is loaded via AUTOSAVE, XMODEM, etc. This command is normally used to save calibration data, options, and other data which does not change often but needs to be retained across a power interruption. Normally the VAR RESTORE command is placed at the start of the program so that previously saved variables are restored and immediately available to the program when it starts. Notes: • The storage space available to this command is 16KB. • Using VAR RESTORE without a previous save will have no effect and will not generate an error. • If, when using RESTORE, a variable with the same name already exists its value will be overwritten. • Saved arrays must be declared (using DIM) before they can be restored. • Be aware that string arrays can rapidly use up all the memory allocated to this command. The LENGTH qualifier can be used when a string array is declared to reduce the size of the array (see the DIM command). This is not needed for ordinary string variables.
WATCHDOG timeout or WATCHDOG OFF	Starts the watchdog timer which will automatically restart the processor when it has timed out. This can be used to recover from some event that disabled the running program (such as an endless loop or a programming or other error that halts a running program). This can be important in an unattended control situation. 'timeout' is the time in milliseconds (ms) before a restart is forced. This command should be placed in strategic locations in the running BASIC program to constantly reset the watchdog timer (to 'timeout') and therefore prevent it from counting down to zero. If the timer count does reach zero (perhaps because the BASIC program has stopped running) the PicoMiteVGA will be automatically restarted and the automatic variable MM.WATCHDOG will be set to true (i.e. 1) indicating that an error occurred. On a normal startup MM.WATCHDOG will be set to false (i.e. 0). Note that OPTION AUTORUN must be specified for the program to restart. WATCHDOG OFF can be used to disable the watchdog timer (this is the default on a reset or power up). The timer is also turned off when the break character (CTRL-C) is used on the console to interrupt a running program.
XMODEM SEND or XMODEM SEND file$ or XMODEM RECEIVE or XMODEM RECEIVE file$ or XMODEM CRUNCH	Transfers a BASIC program to or from a remote computer using the XModem protocol. The transfer is done over the USB console connection. XMODEM SEND will send the current program held in the PicoMiteVGA 's program memory to the remote device. XMODEM RECEIVE will accept a program sent by the remote device and save it into the PicoMiteVGA 's RAM overwriting the program currently held there. You can also specify 'file$' which will transfer the data to/from a file on the SD card. If the file already exists it will be overwritten when receiving a file. Note that the data is buffered in RAM which limits the maximum program size. This command also creates a backup of the program in flash memory which will be automatically retrieved if the CPU is reset of the power is lost. The CRUNCH option works like RECEIVE but will remove all comments, blank lines and unnecessary spaces from the program before saving. This can be used on large programs to allow them to fit into limited memory. SEND, RECEIVE and CRUNCH can be abbreviated to S, R and C.

	The XModem protocol requires a cooperating software program running on the remote computer and connected to its USB serial port. It has been tested on Tera Term running on Windows and it is recommended that this be used. After running the XMODEM command in MMBasic select: File -> Transfer -> XMODEM -> Receive/Send from the Tera Term menu to start the transfer. The transfer can take up to 15 seconds to start and if the XMODEM command fails to establish communications it will return to the MMBasic prompt after 60 seconds and leave the program memory untouched. Download Tera Term from http://ttssh2.sourceforge.jp/

Functions

Note that the functions related to communications functions (I^2C, 1-Wire, and SPI) are not listed here but are described in the appendices at the end of this document.

Square brackets indicate that the parameter or characters are optional.

ABS(number)	Returns the absolute value of the argument 'number' (i.e. any negative sign is removed and a positive number is returned).
ACOS(number)	Returns the inverse cosine of the argument 'number' in radians.
ASC(string$)	Returns the ASCII code (i.e. byte value) for the first letter in ‘string$’.
ASIN(number)	Returns the inverse sine value of the argument 'number' in radians.
ATN(number)	Returns the arctangent of the argument 'number' in radians.
ATAN2(y, x)	Returns the arc tangent of the two numbers x and y as an angle expressed in radians. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.
BIN$(number [, chars])	Returns a string giving the binary (base 2) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
BIN2STR$(type, value [,BIG])	Returns a string containing the binary representation of 'value'. 'type' can be: INT64 signed 64-bit integer converted to an 8 byte string UINT64 unsigned 64-bit integer converted to an 8 byte string INT32 signed 32-bit integer converted to a 4 byte string UINT32 unsigned 32-bit integer converted to a 4 byte string INT16 signed 16-bit integer converted to a 2 byte string UINT16 unsigned 16-bit integer converted to a 2 byte string INT8 signed 8-bit integer converted to a 1 byte string UINT8 unsigned 8-bit integer converted to a 1 byte string SINGLE single precision floating point number converted to a 4 byte string DOUBLE double precision floating point number converted to a 8 byte string By default the string contains the number in little-endian format (i.e. the least significant byte is the first one in the string). Setting the third parameter to ‘BIG’ will return the string in big-endian format (i.e. the most significant byte is the first one in the string) In the case of the integer conversions, an error will be generated if the ‘value’ cannot fit into the ‘type’ (e.g. an attempt to store the value 400 in a INT8). This function makes it easy to prepare data for efficient binary file I/O or for preparing numbers for output to sensors and saving to flash memory. See also the function STR2BIN

BOUND(array() [,dimension]	This returns the upper limit of the array for the dimension requested. The dimension defaults to one if not specified. Specifying a dimension value of 0 will return the current value of OPTION BASE. Unused dimensions will return a value of zero. For example: DIM myarray(44,45) BOUND(myarray(),2) will return 45
CALL(userfunname$, [,userfunparameters,....])	This is an efficient way of programmatically calling user defined functions. (See also the CALL command). In many cases it can be used to eliminate complex SELECT and IF THEN ELSEIF ENDIF clauses and is processed in a much more efficient manner. 'userfunname$' can be any string or variable or function that resolves to the name of a normal user function (not an in-built command). 'userfunparameters' are the same parameters that would be used to call the function directly. A typical use for this command could be writing any sort of emulator where one of a large number of functions should be called depending on a some variable. It also provides a method of passing a function name to another subroutine or function as a variable.
CHOICE(condition, ExpressionIfTrue, ExpressionIfFalse)	This function allows you to do simple either/or selections more efficiently and faster than using IF THEN ELSE ENDIF clauses. The condition is anything that will resolve to nonzero (true) or zero (false). The expressions are anything that you could normally assign to a variable or use in a command and can be integers, floats or strings. Examples: PRINT CHOICE(1, "hello","bye") will print "Hello" PRINT CHOICE (0, "hello","bye") will print "Bye" a=1 : b=1 : PRINT CHOICE (a=b, 4, 5) will print 4
CHR$(number)	Returns a one-character string consisting of the character corresponding to the ASCII code (i.e. byte value) indicated by argument 'number'.
CINT(number)	Round numbers with fractional portions up or down to the next whole number or integer. For example, 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also INT() and FIX().
COS(number)	Returns the cosine of the argument 'number' in radians.
CWD$	The current working directory on the SD card. Invalid for exFAT format. The format is: `A:/dir1/dir2`.
DATE$	Returns the current date based on MMBasic's internal clock as a string in the form "DD-MM-YYYY". For example, "28-07-2012". The internal clock/calendar will keep track of the time and date including leap years. To set the date use the command DATE$ =.

DATETIME$(n)	Returns the date and time corresponding to the epoch number n (number of seconds that have elapsed since midnight GMT on January 1, 1970). The format of the returned string is “dd-mm-yyyy hh:mm:ss”. Use the text NOW to get the current datetime string, i.e. ? DATETIME$(NOW)
DAY$(date$)	Returns the day of the week for a given date as a string “Monday”, “Tuesday” etc. The format for date$ is "DD-MM-YY", "DD-MM-YYYY", or "YYYY-MM-DD". Use NOW to get the day for the current date, e.g. PRINT DAY$(NOW)
DEG(radians)	Converts 'radians' to degrees.

DIR$(fspec, type)
or
DIR$(fspec)
or
DIR$( )
	Will search an SD card for files and return the names of entries found.
	'fspec' is a file specification using wildcards the same as used by the FILES command. E.g. "*.*" will return all entries, "*.TXT" will return text files.
	Note that the wildcard *.* does not find files or folders without an extension.
	'type' is the type of entry to return and can be one of:
	VOL Search for the volume label only
	DIR Search for directories only
	FILE Search for files only (the default if 'type' is not specified)
	The function will return the first entry found. To retrieve subsequent entries use the function with no arguments. i.e. DIR$( ). The return of an empty string indicates that there are no more entries to retrieve.
	This example will print all the files in a directory:
	`f$ = DIR$("*.*", FILE)`
	`DO WHILE f$ <> ""`
	`PRINT f$`
	`f$ = DIR$()`
	`LOOP`
	You must change to the required directory before invoking this command.

DISTANCE(trigger, echo) or DISTANCE(trig-echo)	Measure the distance to a target using the HC-SR04 ultrasonic distance sensor. Four pin sensors have separate trigger and echo connections. 'trigger' is the I/O pin connected to the "trig" input of the sensor and 'echo' is the pin connected to the "echo" output of the sensor. Three pin sensors have a combined trigger and echo connection and in that case you only need to specify one I/O pin to interface to the sensor. Note that any I/O pins used with the HC-SR04 should be 5V capable as the HC-SR04 is a 5V device. The I/O pins are automatically configured by this function and multiple sensors can be used on different I/O pins. The value returned is the distance in centimetres to the target or -1 if no target was detected or -2 if there was an error (i.e. sensor not connected).
EOF([#]fnbr)	Will return true if the file previously opened on the SD card for INPUT with the file number ‘#fnbr’ is positioned at the end of the file. For a serial communications port this function will return true if there are no characters waiting in the receive buffer. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN, INPUT and LINE INPUT commands and the INPUT$ function.

EPOCH(DATETIME$)	Returns the epoch number (number of seconds that have elapsed since midnight GMT on January 1, 1970) for the supplied DATETIME$ string. The format for DATETIME$ is “dd-mm-yyyy hh:mm:ss”, “dd-mm-yy hh:mm:ss”, or “yyyy-mm-dd hh:mm:ss”,. Use NOW to get the epoch number for the current date and time, i.e. PRINT EPOCH(NOW)
EVAL(string$)	Will evaluate 'string$' as if it is a BASIC expression and return the result. 'string$' can be a constant, a variable or a string expression. The expression can use any operators, functions, variables, subroutines, etc that are known at the time of execution. The returned value will be an integer, float or string depending on the result of the evaluation. For example: S$ = "COS(RAD(30)) * 100" : PRINT EVAL(S$) Will display: 86.6025
EXP(number)	Returns the exponential value of 'number', i.e. ex where x is 'number'.
FIELD$(string1, nbr, string2 [, string3])	Returns a particular field in a string with the fields separated by delimiters. 'nbr' is the field to return (the first is nbr 1). 'string1' is the string to search and 'string2' is a string holding the delimiters (more than one can be used). 'string3' is optional and if specified will include characters that are used to quote text in 'string1' (ie, quoted text will not be searched for a delimiter). For example: S$ = "foo, boo, zoo, doo" r$ = FIELD$(s$, 2, ",") will result in r$ = "boo". While: s$ = "foo, 'boo, zoo', doo" r$ = FIELD$(s$, 2, ",", """) will result in r$ = "boo, zoo".
FIX(number)	Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point. For example 9.89 will return 9 and -2.11 will return -2. The major difference between FIX() and INT() is that FIX() provides a true integer function (i.e. does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .
FORMAT$(nbr [, fmt$])	Will return a string representing ‘nbr’ formatted according to the specifications in the string ‘fmt$’. The format specification starts with a % character and ends with a letter. Anything outside of this construct is copied to the output as is. The structure of a format specification is: % [flags] [width] [.precision] type Where ‘flags’ can be: - Left justify the value within a given field width 0 Use 0 for the pad character instead of space + Forces the + sign to be shown for positive numbers space Causes a positive value to display a space for the sign. Negative values still show the – sign ‘width’ is the minimum number of characters to output, less than this the number will be padded, more than this the width will be expanded.

	'precision' specifies the number of fraction digits to generate with an e, or f type or the maximum number of significant digits to generate with a g type. If specified, the precision must be preceded by a dot (.). 'type' can be one of: g Automatically format the number for the best presentation. f Format the number with the decimal point and following digits e Format the number in exponential format If uppercase G or F is used the exponential output will use an uppercase E. If the format specification is not specified "%g" is assumed. Examples: format$(45) will return 45 format$(45, "%g") will return 45
GETSCANLINE	This will report on the line that is currently being drawn on the VGA monitor in the range of 0 to 525. This is irrespective of the current MODE. Using this to time updates to the screen can avoid timing effects caused by updates while the screen is being updated. The first visible line will return a value of 0. Any line number above 479 is in the frame blanking period.
GPS()	The GPS functions are used to return data from a serial communications channel opened as GPS. The function GPS(VALID) should be checked before any of these functions are used to ensure that the returned value is valid.
GPS(ALTITUDE)	Returns current altitude (if sentence GGA is enabled).
GPS(DATE)	Returns the normal date string corrected for local time e.g. "12-01-2020".
GPS(DOP)	Returns DOP (dilution of precision) value (if sentence GGA is enabled).
GPS(FIX)	Returns non zero (true) if the GPS has a fix on sufficient satellites and is producing valid data.
GPS(GEOID)	Returns the geoid-ellipsoid separation (if sentence GGA is enabled).
GPS(LATITUDE)	Returns the latitude in degrees as a floating point number, values are negative for South of equator
GPS(LONGITUDE)	Returns the longitude in degrees as a floating point number, values are negative for West of the meridian.
GPS(SATELLITES)	Returns number of satellites in view (if sentence GGA is enabled).
GPS(SPEED)	Returns the ground speed in knots as a floating point number.
GPS(TIME)	Returns the normal time string corrected for local time e.g. "12:09:33".
GPS(TRACK)	Returns the track over the ground (degrees true) as a floating point number.
GPS(VALID)	Returns: 0=invalid data, 1=valid data

HEX$(number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
INKEY$	Checks the console input buffer and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string. If the input buffer is empty this function will immediately return with an empty string (i.e. "").
INPUT$(nbr, [#]fnbr)	Will return a string composed of ‘nbr’ characters read from a file on the SD card previously opened for INPUT with the file number ‘#fnbr’. This function will read all characters including carriage return and new line without translation. Will return a string composed of ‘nbr’ characters read from a serial communications port opened as 'fnbr'. This function will return as many characters as are waiting in the receive buffer up to ‘nbr’. If there are no characters waiting it will immediately return with an empty string. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, string-pattern$)	Returns the position at which 'string-pattern$' occurs in 'string-searched$', beginning at 'start-position'. If 'start-position' is not provided it will default to 1. Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern$' is not found.
INT(number)	Truncate an expression to the next whole number less than or equal to the argument. For example 9.89 will return 9 and -2.11 will return -3. This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function. See also CINT() .
LCASE$(string$)	Returns ‘string$’ converted to lowercase characters.
LCOMPARE(array1%(), array2%())	Compare the contents of two long string variables array1%() and array2%(). The returned is an integer and will be -1 if array1%() is less than array2%(). It will be zero if they are equal in length and content and +1 if array1%() is greater than array2%(). The comparison uses the ASCII character set and is case sensitive.
LEFT$(string$, nbr)	Returns a substring of ‘string$’ with ‘nbr' of characters from the left (beginning) of the string.
LEN(string$)	Returns the number of characters in 'string$'.
LGETBYTE(array%(), n)	Returns the numerical value of the 'n'th byte in the LONGSTRING held in 'array%()'. This function respects the setting of OPTION BASE in determining which byte to return.

LGETSTR$(array%(), start, length)	Returns part of a long string stored in array%() as a normal MMBasic string. The parameters start and length define the part of the string to be returned.
LINSTR(array%(), search$ [,start])	Returns the position of a search string in a long string. The returned value is an integer and will be zero if the substring cannot be found. array%() is the string to be searched and must be a long string variable. Search$ is the substring to look for and it must be a normal MMBasic string or expression (not a long string). The search is case sensitive. Normally the search will start at the first character in 'str' but the optional third parameter allows the start position of the search to be specified.
LLEN(array%())	Returns the length of a long string stored in array%()
LOC([#]fnbr)	For a file on the SD card opened as RANDOM this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. For a serial communications port opened as 'fnbr' this function will return the number of bytes received and waiting in the receive buffer to be read. #0 can be used which refers to the console's input buffer. The # is optional.
LOF([#]fnbr)	For a file on the SD card this will return the current length of the file in bytes. For a serial communications port opened as 'fnbr' this function will return the space (in characters) remaining in the transmit buffer. Note that when the buffer is full MMBasic will pause when adding a new character and wait for some space to become available. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.
MATH **Simple functions**	The math function performs many simple mathematical calculations that can be programmed in Basic but there are speed advantages to coding looping structures in C and there is the advantage that once debugged they are there for everyone without re-inventing the wheel.
MATH(ATAN3 x,y)	Returns ATAN3 of x and y
MATH(COSH a)	Returns the hyperbolic cosine of a
MATH(LOG10 a)	Returns the base 10 logarithm of a
MATH(SINH a)	Returns the hyperbolic sine of a
MATH(TANH a)	Returns the hyperbolic tan of a
Simple Statistics	
MATH(CHI a())	Returns the Pearson's chi-squared value of the two dimensional array a())

MATH(CHI_p a())	Returns the associated probability in % of the Pearson's chi-squared value of the two dimensional array a())
MATH(CORREL a(), a())	Returns the Pearson’s correlation coefficient between arrays a() and b()
MATH(MAX a() [,index%])	Returns the maximum of all values in the a() array, a() can have any number of dimensions. If the integer variable is specified then it will be updated with the index of the maximum value in the array. This is only available on one-dimensional arrays
MATH(MEAN a())	Returns the average of all values in the a() array, a() can have any number of dimensions
MATH(MEDIAN a())	Returns the median of all values in the a() array, a() can have any number of dimensions
MATH(MIN a(), [index%])	Returns the minimum of all values in the a() array, a() can have any number of dimensions. If the integer variable is specified then it will be updated with the index of the maximum value in the array. This is only available on one-dimensional arrays.
MATH(SD a())	Returns the standard deviation of all values in the a() array, a() can have any number of dimensions
MATH(SUM a())	Returns the sum of all values in the a() array, a() can have any number of dimensions
Vector Arithmetic	
MATH(MAGNITUDE v())	Returns the magnitude of the vector v(). The vector can have any number of elements
MATH(DOTPRODUCT v1(), v2())	Returns the dot product of two vectors v1() and v2(). The vectors can have any number of elements but must have the same cardinality
Matrix Arithmetic	
MATH(M_DETERMINANT array!())	Returns the determinant of the array. The array must be square.
MAX(arg1 [, arg2 [, …]]) or MIN(arg1 [, arg2 [, …]])	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.

MID$(string$, start) or MID$(string$, start, nbr)	Returns a substring of ‘string$’ beginning at ‘start’ and continuing for ‘nbr’ characters. The first character in the string is number 1. If ‘nbr’ is omitted the returned string will extend to the end of ‘string$’
OCT$(number [, chars])	Returns a string giving the octal (base 8) representation of 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
PEEK(BYTE addr%) or PEEK(SHORT addr%) or PEEK(WORD addr%) or PEEK(INTEGER addr%) or PEEK(FLOAT addr%) or PEEK(VARADDR var) or PEEK(CFUNADDR cfun) or PEEK(VAR var, ±offset) or PEEK(VARTBL, ±offset) or PEEK(PROGMEM, ±offset)	Will return a byte or a word within the PIC32 virtual memory space. BYTE will return the byte (8-bits) located at 'addr%' SHORT will return the short integer (16-bits) located at 'addr%' WORD will return the word (32-bits) located at 'addr%' INTEGER will return the integer (64-bits) located at 'addr%' FLOAT will return the floating point number (32-bits) located at 'addr%' VARADDR will return the address (32-bits) of the variable 'var' in memory. An array is specified as var(). CFUNADDR will return the address (32-bits) of the CFunction 'cfun' in memory. This address can be passed to another CFunction which can then call it to perform some common process. VAR, will return a byte in the memory allocated to 'var'. An array is specified as var(). VARTBL, will return a byte in the memory allocated to the variable table maintained by MMBasic. Note that there is a comma after the keyword VARTBL. PROGMEM, will return a byte in the memory allocated to the program. Note that there is a comma after the keyword PROGMEM. Note that 'addr%' should be an integer.
PI	Returns the value of pi.
PIN(pin)	Returns the value on the external I/O ‘pin’. Zero means digital low, 1 means digital high and for analogue inputs it will return the measured voltage as a floating point number. Frequency inputs will return the frequency in Hz. A period input will return the period in milliseconds while a count input will return the count since reset (counting is done on the positive rising edge). The count input can be reset to zero by resetting the pin to counting input (even if it is already so configured). This function will also return the state of a pin configured as an output. Also see the SETPIN and PIN() = commands. Refer to the chapter *"Using the I/O pins"* for a general description of the PicoMiteVGA 's input/output capabilities.
PIN(TEMP)	Returns the temperature of the RP2040 chip (see the RP2040 data sheet for the details)

PIO (SHIFTCTRL push_threshold [,pull_threshold] [,autopush] [,autopull])	helper function to calculate the value of shiftctrl for the INIT MACHINE command
PIO (PINCTRL no_side_set_pins [,no_set_pins] [,no_out_pins] [,IN base] [,side_set_base] [,set_base] [, out_base])	helper function to calculate the value of pinctrl for the INIT MACHINE command. Note: The pin parameters must be formatted as GPn.
PIO (EXECCTRL jmp_pin ,wrap_target, wrap	helper function to calculate the value of execctrl for the INIT MACHINE command
PIO (FDEBUG pio)	returns the value of the FSDEBUG register for the pio specified
PIO (FSTAT pio)	returns the value of the FSTAT register for the pio specified
PIO (FLEVEL pio	returns the value of the FLEVEL register for the pio specified PIO(FLEVEL pio)
PORT(start, nbr [,start, nbr]…)	Returns the value of a number of I/O pins in one operation. 'start' is an I/O pin number and its value will be returned as bit 0. 'start'+1 will be returned as bit 1, 'start'+2 will be returned as bit 2, and so on for 'nbr' number of bits. I/O pins used must be numbered consecutively and any I/O pin that is invalid or not configured as an input will cause an error. The start/nbr pair can be repeated up to 25 times if additional groups of input pins need to be added. This function will also return the state of a pin configured as an output. It can be used to conveniently communicate with parallel devices like memory chips. Any number of I/O pins (and therefore bits) can be used from 1 to the number of I/O pins on the chip. See the PORT command to simultaneously output to a number of pins.
PIXEL(x, y [,page_number])	Returns the colour of a pixel on the VGA monitor. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel.
PULSIN(pin, polarity) or PULSIN(pin, polarity, t1) or PULSIN(pin, polarity, t1, t2)	Measures the width of an input pulse from 1µs to 1 second with 0.1µs resolution. 'pin' is the I/O pin to use for the measurement, it must be previously configured as a digital input. 'polarity' is the type of pulse to measure, if zero the function will return the width of the next negative pulse, if non zero it will measure the next positive pulse. 't1' is the timeout applied while waiting for the pulse to arrive, 't2' is the timeout used while measuring the pulse. Both are in microseconds (µs) and are optional. If 't2' is omitted the value of 't1' will be used for both timeouts. If both 't1' and 't2' are omitted then the timeouts will be set at 100000 (i.e. 100ms). This function returns the width of the pulse in microseconds (µs) or -1 if a timeout has occurred. The measurement is accurate to ±0.5% and ±0.5µsNote that this function will cause the running program to pause while the measurement is made and interrupts will be ignored during this period.

RAD(degrees)	Converts 'degrees' to radians.
RGB(red, green, blue) or RGB(shortcut)	Generates an RGB true colour value. 'red', 'blue' and 'green' represent the intensity of each colour. A value of zero represents black and 255 represents full intensity. 'shortcut' allows common colours to be specified by naming them. The colours that can be named are white, yellow, lilac, brown, fuchsia, rust, magenta, red, cyan, green, cerulean, midgreen, cobalt, myrtle, blue and black. For example, RGB(red) or RGB(cyan).
RIGHT$(string$, number-of-chars)	Returns a substring of ‘string$’ with ‘number-of-chars’ from the right (end) of the string.
RND(number) or RND	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The RANDOMIZE command reseeds the random number generator.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE$(number)	Returns a string of blank spaces 'number' characters long.
SPI (data) or SPI2 (data)	Send and receive data using an SPI channel. A single SPI transaction will send data while simultaneously receiving data from the slave. ‘data’ is the data to send and the function will return the data received during the transaction. ‘data’ can be an integer or a floating point variable or a constant.
SQR(number)	Returns the square root of the argument 'number'.
STR$(number) or STR$(number, m) or STR$(number, m, n) or STR$(number, m, n, c$)	Returns a string in the decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR$(123.456) will return `"123.456"` STR$(-123.456) will return `"-123.456"` STR$(123.456, 1) will return `"123.456"`

	STR$(123.456, -1) will return "+123.456" STR$(123.456, 6) will return " 123.456" STR$(123.456, -6) will return "+123.456" STR$(-123.456, 6) will return "-123.456" STR$(-123.456, 6, 5) will return "-123.45600" STR$(-123.456, 6, -5) will return "-1.23456e+02" STR$(53, 6) will return " 53" STR$(53, 6, 2) will return " 53.00" STR$(53, 6, 2, "*") will return "****53.00"
STR2BIN(type, string$ [,BIG])	Returns a number equal to the binary representation in 'string$'. 'type' can be: INT64 converts 8 byte string representing a signed 64-bit integer to an integer UINT64 converts 8 byte string representing an unsigned 64-bit integer to an integer INT32 converts 4 byte string representing a signed 32-bit integer to an integer UINT32 converts 4 byte string representing an unsigned 32-bit integer to an integer INT16 converts 2 byte string representing a signed 16-bit integer to an integer UINT16 converts 2 byte string representing an unsigned 16-bit integer to an integer INT8 converts 1 byte string representing a signed 8-bit integer to an integer UINT8 converts 1 byte string representing an unsigned 8-bit integer to an integer SINGLE converts 4 byte string representing single precision float to a float DOUBLE converts 8 byte string representing single precision float to a float By default the string must contain the number in little-endian format (i.e. the least significant byte is the first one in the string). Setting the third parameter to 'BIG' will interpret the string in big-endian format (i.e. the most significant byte is the first one in the string). This function makes it easy to read data from binary data files, interpret numbers from sensors or efficiently read binary data from flash memory chips. An error will be generated if the string is the incorrect length for the conversion requested See also the function BIN2STR$
STRING$(nbr, ascii) or STRING$(nbr, string$)	Returns a string 'nbr' bytes long consisting of either the first character of string$ or the character representing the ASCII value 'ascii' which is an integer or float number in the range of 0 to 255.
TAB(number)	Outputs spaces until the column indicated by 'number' has been reached on the console output.

TAN(number)	Returns the tangent of the argument 'number' in radians.
TEMPR(pin)	Return the temperature measured by a DS18B20 temperature sensor connected to 'pin' (which does not have to be configured). The returned value is degrees C with a default resolution of 0.25ºC. If there is an error during the measurement the returned value will be 1000. The time required for the overall measurement is 200ms and interrupts will be ignored during this period. Alternatively the TEMPR START command can be used to start the measurement and your program can do other things while the conversion is progressing. When this function is called the value will then be returned instantly assuming the conversion period has expired. If it has not, this function will wait out the remainder of the conversion time before returning the value. The DS18B20 can be powered separately by a 3V to 5V supply or it can operate on parasitic power from the PicoMiteVGA . See the section *Special Hardware Devices* for more details.
TIME$	Returns the current time based on MMBasic's internal clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00". To set the current time use the command TIME$ = .
TIMER	Returns the elapsed time in milliseconds (e.g. 1/1000 of a second) since reset. The timer is reset to zero on power up or a CPU restart and you can also reset it by using TIMER as a command. If not specifically reset it will continue to count up forever (it is a 64 bit number and therefore will only roll over to zero after 200 million years).
UCASE$(string$)	Returns ‘string$’ converted to uppercase characters.
VAL(string$)	Returns the numerical value of the ‘string$’. If 'string$' is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.

Obsolete Commands and Functions

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

Note that these commands may be removed in the future to recover memory for other features.

GOSUB target	Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN. New programs should use defined subroutines (i.e. SUB…END SUB).
IF condition THEN linenbr	For Microsoft compatibility a GOTO is assumed if the THEN statement is followed by a number. A label is invalid in this construct. New programs should use: IF condition THEN GOTO linenbr | label
IRETURN	Returns from an interrupt when the interrupt destination was a line number or a label. New programs should use a user defined subroutine as an interrupt destination. In that case END SUB or EXIT SUB will cause a return from the interrupt.
ON nbr GOTO | GOSUB target[,target, target,...]	ON either branches (GOTO) or calls a subroutine (GOSUB) based on the rounded value of 'nbr'; if it is 1, the first target is called, if 2, the second target is called, etc. Target can be a line number or a label. New programs should use SELECT CASE.
POS	For the console, returns the current cursor position in the line in characters.
RETURN	RETURN concludes a subroutine called by GOSUB and returns to the statement after the GOSUB.

Appendix A
Serial Communications

Two serial interfaces are available for asynchronous serial communications. They are labelled COM1: and COM2:.

I/O Pins

Before a serial interface can be used the I/O pins must be defined using the following command for the first channel (referred as COM1):

SETPIN rx, tx, COM1

Valid pins are	RX:	GP1 or GP13
	TX:	GP0, GP12 or GP28

And the following command for the second channel (referred to as COM2):

SETPIN rx, tx, COM2

Valid pins are	RX:	GP5
	TX:	GP4

TX is data from the PicoMiteVGA and RX is data to it.

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

Commands

After being opened the serial port will have an associated file number and you can use any commands that operate with a file number to read and write to/from it. A serial port can be closed using the CLOSE command.

The following is an example:

```
SETPIN GP13, GP16, COM1     ' assign the I/O pins for the first serial port
OPEN "COM1:4800" AS #5      ' open the first serial port with a speed of 4800 baud
PRINT #5, "Hello"           ' send the string "Hello" out of the serial port
dat$ = INPUT$(20, #5)       ' get up to 20 characters from the serial port
CLOSE #5                    ' close the serial port
```

The OPEN Command

A serial port is opened using the command:

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

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

'comspec$' is the communication specification and is a string (it can be a string variable) specifying the serial port to be opened and optional parameters. The default is 9600 baud, 8 data bits, no parity and one stop bit.

It has the form `"COMn: baud, buf, int, int-trigger, EVEN, ODD, S2, 7BIT"` where:

- 'n' is the serial port number for either COM1: or COM2:.
- 'baud' is the baud rate. This can be any number from 1200 to well over 1000000. Default is 9600.
- 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes.
- 'int' is interrupt subroutine to be called when the serial port has received some data.
- 'int-trigger' is the number of characters received which will trigger an interrupt.

All parameters except the serial port name (COMn:) are optional. If any one parameter is left out then all the following parameters must also be left out and the defaults will be used.

Four options can be added to the end of 'comspec$'. These are:

- 'S2' specifies that two stop bits will be sent following each character transmitted.
- EVEN specifies that an even parity bit will be applied, this will result in a 9-bit transfer unless 7BIT is set.
- ODD specifies that an odd parity bit will be applied, this will result in a 9-bit transfer unless 7BIT is set
- 7BIT specifies that there a 7bits of data. This is normally used with EVEN or ODD

Examples

Opening a serial port using all the defaults:
```
OPEN "COM1:" AS #2
```
Opening a serial port specifying only the baud rate (4800 bits per second):
```
OPEN "COM1:4800" AS #1
```
Opening a serial port specifying the baud rate (9600 bits per second) and receive buffer size (1KB):
```
OPEN "COM2:9600, 1024" AS #8
```
The same as above but with two stop bits enabled:
```
OPEN "COM2:9600, 1024, S2" AS #8
```
An example specifying everything including an interrupt, an interrupt level, and two stop bits:
```
OPEN "COM2:19200, 1024, ComIntLabel, 256, S2" AS #5
```
Reading and Writing

Once a serial port has been opened you can use any command or function that uses a file number to read from and write to the port. Data received by the serial port will be automatically buffered in memory by MMBasic until it is read by the program and the INPUT$() function is the most convenient way of doing that. When using the INPUT$() function the number of characters specified will be the maximum number of characters returned but it could be less if there are less characters in the receive buffer. In fact the INPUT$() function will immediately return an empty string if there are no characters available in the receive buffer.

The LOC() function is also handy; it will return the number of characters waiting in the receive buffer (i.e. the maximum number characters that can be retrieved by the INPUT$() function). Note that if the receive buffer overflows with incoming data the serial port will automatically discard the oldest data to make room for the new data.

The PRINT command is used for outputting to a serial port and any data to be sent will be held in a memory buffer while the serial port is sending it. This means that MMBasic will continue with executing the commands after the PRINT command while the data is being transmitted. The one exception is if the output buffer is full and in that case MMBasic will pause and wait until there is sufficient space before continuing. The LOF() function will return the amount of space left in the transmit buffer and you can use this to avoid stalling the program while waiting for space in the buffer to become available.

If you want to be sure that all the data has been sent (perhaps because you want to read the response from the remote device) you should wait until the LOF() function returns 256 (the transmit buffer size) indicating that there is nothing left to be sent.

Serial ports can be closed with the CLOSE command. This will wait for the transmit buffer to be emptied then free up the memory used by the buffers and cancel the interrupt (if set). A serial port is also automatically closed when commands such as RUN and NEW are issued.

Interrupts

The interrupt subroutine (if specified) will operate the same as a general interrupt on an external I/O pin (see the chapter "Using the I/O pins" for a description).

When using interrupts you need to be aware that it will take some time for MMBasic to respond to the interrupt and more characters could have arrived in the meantime, especially at high baud rates. For example, if you have specified the interrupt level as 200 characters and a buffer of 256 characters then quite easily the buffer will have overflowed by the time the interrupt subroutine can read the data. In this case the buffer should be increased to 512 characters or more.

Appendix B
I²C Communications

There are two I²C channels. They can operate in master or slave mode.

I/O Pins

Before the I²C interface can be used the I/O pins must be defined using the following command for the first channel (referred as I2C):

SETPIN sda, scl, I2C

Valid pins are SDA: GP0, GP4, GP12 or GP28
SCL: GP1, GP5 or GP13

And the following command for the second channel (referred to as I2C2):

SETPIN sda, scl, I2C2

Valid pins are SDA: GP2, GP6, GP10, GP14, GP22 or GP26
SCL: GP3, GP7, GP11, GP15 or GP27

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

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

I²C Master Commands

There are four commands that can be used for the first channel (I2C) in master mode as follows.

The commands for the second channel (I2C2) are identical except that the command is I2C2

I2C OPEN speed, timeout

Enables the I²C module in master mode. The I2C command refers to channel 1 while the command I2C2 refers to channel 2 using the same syntax.

'speed' is the clock speed (in KHz) to use and must be either 100 or 400.

'timeout' is a value in milliseconds after which the master send and receive commands will be interrupted if they have not completed. The minimum value is 100. A value of zero will disable the timeout (though this is not recommended).

I2C WRITE addr, option, sendlen, senddata [,sendata ....]

Send data to the I²C slave device. The I2C command refers to channel 1 while the command I2C2 refers to channel 2 using the same syntax.

'addr' is the slave's I²C address.

'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command)

'sendlen' is the number of bytes to send.

'senddata' is the data to be sent - this can be specified in various ways (all data sent will be sent as bytes with a value between 0 and 255):

- The data can be supplied as individual bytes on the command line. Example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25
- The data can be in a one dimensional array specified with empty brackets (i.e. no dimensions). 'sendlen' bytes of the array will be sent starting with the first element. Example: I2C WRITE &H6F, 0, 3, ARRAY()
- The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 0, 3, STRING$

I2C READ addr, option, rcvlen, rcvbuf	Get data from the I²C slave device. The I2C command refers to channel 1 while the command I2C2 refers to channel 2 using the same syntax. 'addr' is the slave's I²C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 'rcvlen' is the number of bytes to receive. 'rcvbuf' is the variable or array used to save the received data - this can be: • A string variable. Bytes will be stored as sequential characters in the string. • A one dimensional array of numbers specified with empty brackets. Received bytes will be stored in sequential elements of the array starting with the first. Example: I2C READ &H6F, 0, 3, ARRAY() • A normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE	Disables the master I²C module and returns the I/O pins to a "not configured" state. This command will also send a stop if the bus is still held.

I²C Slave Commands

I2C SLAVE OPEN addr, send_int, rcv_int	Enables the I²C module in slave mode. The I2C command refers to channel 1 while the command I2C2 refers to channel 2 using the same syntax. 'addr' is the slave I²C address. 'send_int' is the subroutine to be invoked when the module has detected that the master is expecting data. 'rcv_int is the subroutine to be called when the module has received data from the master. Note that this is triggered on the first byte received so your program might need to wait until all the data is received.
I2C SLAVE WRITE sendlen, senddata [,sendata ....]	Send the data to the I²C master. The I2C command refers to channel 1 while the command I2C2 refers to channel 2 using the same syntax. This command should be used in the send interrupt (ie in the 'send_int' subroutine when the master has requested data). Alternatively, a flag can be set in the interrupt subroutine and the command invoked from the main program loop when the flag is set. 'sendlen is the number of bytes to send. 'senddata' is the data to be sent. This can be specified in various ways, see the I2C WRITE commands for details.
I2C SLAVE READ rcvlen, rcvbuf, rcvd	Receive data from the I²C master device. The I2C command refers to channel 1 while the command I2C2 refers to channel 2 using the same syntax. This command should be used in the receive interrupt (ie in the 'rcv_int' subroutine when the master has sent some data). Alternatively a flag can be set in the receive interrupt subroutine and the command invoked from the main program loop when the flag is set. 'rcvlen' is the maximum number of bytes to receive. 'rcvbuf' is the variable to receive the data. This can be specified in various ways, see the I2C READ commands for details. 'rcvd' is a variable that, at the completion of the command, will contain the actual number of bytes received (which might differ from 'rcvlen').
I2C SLAVE CLOSE	Disables the slave I²C module and returns the external I/O pins to a "not configured" state. They can then be configured using SETPIN.

Errors

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

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

7-Bit Addressing

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

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

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

Examples

As an example of a simple communications where the PicoMiteVGA is the master, the following program will read and display the current time (hours and minutes) maintained by a PCF8563 real time clock chip connected to the second I²C channel:

```
DIM AS INTEGER RData(2)             ' this will hold received data
SETPIN GP6, GP5, I2C2               ' assign the I/O pins for I2C2
I2C2 OPEN 100, 1000                 ' open the I2C channel
I2C2 WRITE &H51, 0, 1, 3            ' set the first register to 3
I2C2 READ &H51, 0, 2, RData()       ' read two registers
I2C2 CLOSE                          ' close the I2C channel
PRINT "Time is " RData(1) ":" RData(0)
```

This is an example of communications between two PicoMiteVGA s where one is the master and the other is the slave.

First the master:

```
SETPIN GP2, GP3, i
I2C2 OPEN 100, 1000
i = 10
DO
  i = i + 1
  a$ = STR$(i)
  I2C2 WRITE &H50, 0, LEN(a$), a$
  PAUSE 200
  I2C2 READ &H50, 0, 8, a$
  PRINT a$
  PAUSE 200
LOOP
```

Then the slave:

```
SETPIN GP2, GP3, I2C2
I2C2 SLAVE OPEN &H50, tint, rint
DO : LOOP

SUB rint
  LOCAL count, a$
  I2C2 SLAVE READ 10, a$, count
  PRINT LEFT$(a$, count)
END SUB

SUB tint
  LOCAL a$ = Time$
  I2C2 SLAVE WRITE LEN(a$), a$
END SUB
```

Appendix C
1-Wire Communications

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

There are three commands that you can use:

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

Where:

pin - The PicoMiteVGA I/O pin to use. It can be any pin capable of digital I/O.

flag - A combination of the following options:

1 - Send reset before command

2 - Send reset after command

4 - Only send/recv a bit instead of a byte of data

8 - Invoke a strong pullup after the command (the pin will be set high and open drain disabled)

length - Length of data to send or receive

data - Data to send or variable to receive.
The number of data items must agree with the length parameter.

The automatic variable MM.ONEWIRE returns true if a device was found

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

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

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

Appendix D
SPI Communications

The Serial Peripheral Interface (SPI) communications protocol is used to send and receive data between integrated circuits. The PicoMiteVGA acts as the master (i.e. it generates the clock).

I/O Pins

Before an SPI interface can be used the I/O pins for the channel must be allocated using the following commands. For the first channel (referred as SPI) it is:

SETPIN rx, tx, clk, SPI

Valid pins are	RX:	GP0 or GP4
	TX:	GP3 or GP7
	CLK:	GP2 or GP6

And the following command for the second channel (referred to as SPI2) is:

SETPIN rx, tx, clk, SPI2

Valid pins are	RX:	GP12 or GP28
	TX:	GP11, GP15 or GP27
	CLK:	GP10, GP14 or GP26

TX is data from the PicoMiteVGA and RX is data to it.

SPI Open

To use the SPI function the SPI channel must be first opened.

The syntax for opening the first SPI channel is (use SPI2 for the second channel):

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

Where:

- ‘speed’ is the speed of the clock. It is a number representing the clock speed in Hz.
- 'mode' is a single numeric digit representing the transmission mode – see Transmission Format below.
- 'bits' is the number of bits to send/receive. This can be any number in the range of 4 to 16 bits.
- It is the responsibility of the program to separately manipulate the CS (chip select) pin if required.

Transmission Format

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

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

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

Standard Send/Receive

When the first SPI channel is open data can be sent and received using the SPI function (use SPI2 for the second channel). The syntax is:

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

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

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

Bulk Send/Receive

Data can also be sent in bulk (use SPI2 for the second channel):

```
SPI WRITE nbr, data1, data2, data3, … etc
```

or

```
SPI WRITE nbr, string$
```

or

```
SPI WRITE nbr, array()
```

In the first method 'nbr' is the number of data items to send and the data is the expressions in the argument list (i.e. 'data1', data2' etc). The data can be an integer or a floating point variable or a constant.

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

Data can also be received in bulk (use SPI2 for the second channel):

```
SPI READ nbr, array()
```

Where 'nbr' is the number of data items to be received and array() is a single dimension integer array where the received data items will be saved. This command sends zeros while reading the data from the slave.

SPI Close

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

```
SPI CLOSE
```

Use SPI2 for the second channel.

Examples

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

```
PIN(10) = 1 : SETPIN 10, DOUT     ' pin 10 will be used as the enable signal
SETPIN GP20, GP4, GP1, SPI        ' assign the I/O pins
SPI OPEN 5000000, 3, 8            ' speed is 5MHz and the data size is 8 bits
PIN(10) = 0                       ' assert the enable line (active low)
junk = SPI(&H80)                  ' send the command and ignore the return
byte1 = SPI(0)                    ' get the first byte from the slave
byte2 = SPI(0)                    ' get the second byte from the slave
PIN(10) = 1                       ' deselect the slave
SPI CLOSE                         ' and close the channel
```

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

```
OPTION BASE 1                     ' our array will start with the index 1
DIM data%(2)                      ' define the array for receiving the data
SETPIN GP20, GP4, GP1, SPI        ' assign the I/O pins
PIN(10) = 1 : SETPIN 10, DOUT     ' pin 10 will be used as the enable signal
SPI OPEN 5000000, 3, 8            ' speed is 5MHz, 8 bits data
PIN(10) = 0                       ' assert the enable line (active low)
SPI WRITE 1, &H80                 ' send the command
SPI READ 2, data%()               ' get two bytes from the slave
PIN(10) = 1                       ' deselect the slave
SPI CLOSE                         ' and close the channel
```

Appendix E
The PIO Programming Package

PIO programming is not recommended for beginners. It requires some knowledge of systems programming and this section of the manual can only give a simplified description of the PIO hardware and a brief introduction to how to program it. The reader is referred to the excellent "Raspberry Pi RP2040 Datasheet", Chapter 3 in particular (about 70 pages!), for further information and programming examples. It is not necessary to be able to program the PIOs in order to use MMBasic (indeed, at the time of writing only the PicoMiteVGA has this PIO system).

Introduction to the PIO

The RP2040 chip used in the PicoMiteVGA contains two PIO blocks, rather like cut-down, highly specialised CPU cores. They are capable of running completely independently of the main system and of each other. They can be used to create such things as very high accuracy serial data interfaces and bit streams, although they are by no means restricted to this sort of thing. They can be made to run extremely fast, with a throughput of up to 32 bits during every clock cycle.

On the PicoMiteVGA PIO block 0 is used to generate the VGA output so only block 1 is available to BASIC programs.

The following describes a single PIO block unless otherwise noted. The other is, of course, identical. MMBasic refers to them as PIO0 and PIO1, in line with the Raspberry Pi documentation. Apologies if the following appears to be rather complex, but it's not possible to write PIO programs without knowing how it works.

A single PIO block has four independent state machines. All four state machines share a single 32 instruction program area of flash memory. This memory has write-only access from the main system, but has four read ports, one for each state machine, so that each can access it independently at its own speed. Each state machine has its own program counter.

Each state machine also has two 32-bit "scratchpad" registers, X and Y, which can be used as temporary data stores.

I/O pins are accessed via an input/output mapping module that can access 32 pins (but limited to 30 for the RP2040). All state machines can access all the pins independently and simultaneously.

The system can write data into the input end of a 4-word 32-bit wide TX FIFO buffer. The state machine can then use *pull* to move the output word of the FIFO into the OSR (Output Shift Register). It can also use *out* to shift 1-32 bits at a time from the OSR into the output mapping module or other destinations. AUTOPULL can be used to automatically *pull* data until the TX FIFO is empty or reaches a preset level.

The system can read data from the output end of a 4-word 32-bit wide RX FIFO buffer. The state machine can then use *in* to shift 1-32 bits of data at a time from the input mapping module into the ISR (Input Shift Register). It can also use *push* to move the contents of the ISR into the FIFO. AUTOPUSH can be used to automatically *push* data until the RX FIFO is full or reaches a preset level.

The FIFO buffers can be reconfigured to form a single direction 8-word 32-bit FIFO in a single direction. The buffers allow data to be passed to and from the state machines without either the system or the state machine having to wait for the other.

Each of the four state machines in the PIO has four registers associated with it:

- CLKDIV is the clock divider, which has a 16-bit integer divider and an 8-bit fractional divider. This sets how fast the state machine runs. It divides down from the main system clock.

- EXECCTRL holds information controlling the translation and execution of the program memory

- SHIFTCTRL controls the arrangement and useage of the shift registers
- PINCTRL controls which and how the GPIO pins are used.

The four state machines of a PIO have shared access to its block of 8 interrupt flags. Any state machine can use any flag. They can set, reset or wait for them to change. In this way they can be made to run synchronously if required. The lower four flags are also accessible to and from the main system, so the PIO can be controlled or pass interrupts back.

PIO Operation

On each clock cycle (speed decided by the contents of CLKDIV) each state machine reads and executes one instruction. All instructions take one clock cycle (unless deliberately stalled by, for example, a *wait* instruction) to execute so this gives very precise timing.

There are no parts of the program that are specific to any state machine. The state machines read an instruction then complete it with information from their individual registers before executing it.
e.g. the program area may contain an instruction such as "jmp pin 5", which would jump to address location 5 if pin "pin" goes high, but which "pin" is decided by the contents of the EXECCTRL register.

EXECCTRL has some important data in it. Just for example:
JMP_PIN is the pin number used to trigger a JMP PIN instruction.
- WRAP_BOTTOM is the low end program address used by this state machine. The default is 0, the beginning of the program area.
- WRAP_TOP is the high end - when processing reaches this address it will loop back to WRAP_BOTTOM. The default is 31, the last address of the program area.

WRAP_BOTTOM and WRAP_TOP are changed in the assembler using *.wrap target* and *.wrap* respectively. Note that *wrap* acts as if the end of the program area has been reached and the program counter simply loops round to *wrap target*. It is not a JMP instruction, it's a true loop defined in EXECCTRL and does not require a clock cycle or have an instruction in program memory.

As the state machines are independent you can arrange WRAP_BOTTOM and WRAP_TOP to read individual or overlapping areas of the program - it makes absolutely no difference to the content of the program area. Not only that, if the WRAP_TOP address contains a JMP instruction then that takes priority, so some very complex looping is possible.

The PINCTRL register holds data that sets up which pins are going to be accessed by the state machine. e.g.
- OUT_BASE sets the first output pin of a block
- OUT_COUNT states how many pins from OUT_BASE upward will be used (0 to 32).

PIO Programming

A PIO has nine possible programming instructions, but there can be many variations on each one. For example, Mov can have up to 8 sources, 8 destinations, 3 process operations during the copy, with optional delay and/or side set operations!

- Jmp Jump to an absolute address in program memory if a condition is true (or instantly).
- Wait Stall operation of the state machine until a condition is true.
- In Shift a number of bits from a source into the ISR.
- Out Shift a number of bits out of the OSR to a destination.
- Push Push the contents of the ISR into the RX FIFO as a single 32-bit word.
- Pull Load a 32-bit word from the TX FIFO into the OSR.
- Mov Copy date from a source to a destination.
- Irq Set or clear an interrupt flag.
- Set Immediately write data to a destination.

Instructions are all 16-bit and contain both the instruction and all data associated with it. All instructions operate in 1 clock cycle, but it is possible to introduce a delay of several idle clock cycles between an instruction and the next.

Additionally, there is a facility called "side-set" which allows a value to be written to some pre-defined output pins while an instruction is being read from memory. This is transparent to the program. MMBasic provides a set of commands to control PIO operation:

- PIO INIT MACHINE pio%, statemachine%, clockspeed ,pinctrl [,execctrl] [,shiftctrl] [,startinstruction]
Initialise a state machine on a PIO.

- PIO EXECUTE pio, state_machine, instruction%
Immediately execute an instruction on a specified PIO and state machine. This will immediately stop the program running on the stated PIO and state machine and restart it at instruction%

- PIO WRITE pio, state_machine, count, data0 [,data1....]
Write a series of data elements to a specified PIO and state machine from an integer array.

- PIO READ pio, state_machine, count, data%()
Read a series of data elements from a specified PIO and state machine into an integer array.

- PIO START pio, statemachine
Start a specified state machine running on a PIO. Program will start at the instruction specified in PIO INIT MACHINE 'startinstruction'.

- PIO STOP pio, statemachine
Stop a specified state machine on a PIO.

- PIO CLEAR pio
Stops all state machines on a PIO and set PINCTRL, EXECCTR and SHIFTCTRL to their default values for all the state machines on that PIO.

- PIO PROGRAM LINE pio, line, instruction
Program just the specified line into a PIO

Additionally, there are some PIO helper functions which are used to calculate or read the values of PIO registers:

- PIO (SHIFTCTRL push_threshold [,pull_threshold] [,autopush] [,autopull])
Returns the calculated value for SHIFTCTRL when used with the stated options.

- PIO (PINCTRL no_side_set_pins [,no_set_pins] [,no_out_pins] [,IN base] [,side_set_base] [,set_base] [, out_base])
Returns the calculated value for PINCTRL when used with the stated options. Note: The pin parameters must be formatted as GPn.

- PIO (EXECCTRL jmp_pin ,wrap_target, wrap)
Returns the calculated value for EXECCTRL when used with the stated options.

- PIO (FDEBUG pio)
Returns the contents of the FIFO debug register.

- PIO (FSTAT pio)
Returns the contents of the FIFO status register.

- PIO (FLEVEL pio)
Returns the levels of the FIFO buffers.

In MMBasic PIO programming is initiated by using the command
PIO INIT MACHINE pio%, statemachine%, clockspeed ,pinctrl [,execctrl] [,shiftctrl]

Programming example:

This generates a square wave on pin-1 at 1/4 of the clock frequency requested. The program itself is stored in a%(). PIO program 1,a%() sets up state machine 0 to run the program.

```
Dim a%(7)=(&H0001E000E101E081,0,0,0,0,0,0,0)  'This is the actual program
SetPin 1,pio1                                 'Pin 1 is allocated to pio1
PIO program 1,a%()                            'pio1  will run the program stored in a%()
PIO init machine 1,0,100000,Pio(pinctrl 0,1)  'Initialise pio1, state machine 0 to run at 100000Hz
                                              'note that PINCTRL, EXECCTRL & SHIFTCTRL are not
                                              'changed in this example.
PIO start 1,0                                 'Start pio0, state machine 0
```

The program shown in a%() can be hand assembled or the PASM assembler can be used. This is a MMBasic program which allows the user to enter assembler instructions as DATA statements. PASM will then produce the necessary program array and will set the *wrap target* and *wrap* addresses in EXECCTRL if these have been used in the program.

The PIO Programming Package [currently in the development & testing stage]

This is a package of MMBasic programs and subroutines to assist in programming the PIO devices. The subroutines are intended to be incorporated into the user's program.

PASM (PIO Assembler) is an assembler subroutine to produce PIO code from a list of DATA statements.

PREVAS (PIO Reverse-Assembler) is a stand-alone simple disassembler that can read the code produced by PASM.

PREDIT (PIO Register Editor) allows the user to edit the state machine registers using a simple console interface. It is a stand-alone program and gives the user the hex values for the registers when the chosen fields are entered. All four state machines are on screen simultaneously.

The above programs can all access a common *.PIO data file format on a SD card. PASM does this via a subroutine called "fileman", which allows directory listing, save & load.

Introduction to PASM

The PASM assembler is a well commented MMBasic program which can be used as a "shell" for a user program or run stand-alone to produce the PIO program. This can then be saved and restored as a *.PIO file on an attached SD card if one is available.

To simplify PASM, it produces the code for ONE state machine. Thus, if all four state machines are in use you might need to assemble four programs with different offsets then combine them to produce the final code block. WRAP_BOTTOM and WRAP_TOP would be set for each state machine so that they can all run their own loops simultaneously.

In order to use PASM the user must dimension an array to be used for the
output code. e.g. DIM out%(7). Currently PASM will always produce a 32-instruction block no matter how long the program is.

Input to the assembler is via DATA statements. Up to 32 instructions can be programmed, although there may be more DATA statements than this due to the presence of directives and pseudo-operations. The final DATA statement must be a null string.

Delimiters in the instructions must be a single space or a single comma.
Additional delimiters will cause the instruction to assemble incorrectly.

Directives supported:
- *wrap target* The instruction after this directive will be jumped to when a *.wrap* instruction is reached.
- *wrap* Jump to the *.wrap target* location.

Note: The above directives can only be used once for any one state machine.
- *.origin n* Used at the start of the program to offset it to start at address n. This value is added to JMP instructions so should be used with care.
- *.word n* Inserts 16-bit value n. This could be embedded data.

Pseudo-operations:
- *nop* No operation. In actual fact this is assembled as &HA042 (mov_y,y)

Using PASM

The PASM subroutines can be incorporated into the user program. Typical usage to write a program for state machine 0 would be:

```
DIM INTEGER out%(7)                                          'dimension an output array.
DIM INTEGER EXECCTRL%(3), PINCTRL%(3), SHIFTCTRL%(3)  'dimension the reg arrays.

datalabel:                                   'the user's PIO program starts here
DATA "instruction"
DATA "instruction"
....
....
DATA "instruction"
DATA ""                                      'use a null string to terminate the data

initrp                                       'initialise state machine registers to defaults
'At this stage you can set calculated values of EXECCTRL%(), PINCTRL%() and SHIFTCTRL%()
'if required. Don't use initrp if importing register values from a *.PIO file as it will reset them..

PASM "datalabel", out%(), 0          'this will assemble the code. Unused addresses in
                                     'the block will be filled with NOP instructions.

fileman("s","myPIOprog")             'save the program to the SD card
END                                  'end of program
```

On exit:
out%() is an array of 8 64-bit integers, each holding 4 16-bit instructions.
EXECCTRL%(state_machine_number)- WRAP_BOTTOM
and
EXECCTRL%(state_machine_number)- WRAP_TOP
are updated according to .wrap top and .wrap commands.

fileman is a simple command line file manager subroutine for *.PIO files. *.PIO files are stored on an attached SD card in the current directory. The other programs in the package can also use these files, so data transfer between them can be quite easy.

fileman("d") prints a directory listing of *.PIO files in the current directory.

fileman("s","filename") saves "filename.PIO" to the current directory.

fileman("l","filename") loads "filename.PIO" from the current directory

There is no error checking and fileman will fail with an error message if the file doesn't exist.

This subroutine can be used to import the state machine registers from PREDIT prior to running PASM to create the program. It can then be used to save the completed program and registers. If the file extension isn't stated then ".PIO" will be appended automatically.

PIO Files

This is a simple ASCII text file format. A PIO file contains the PIO program array followed by the EXECCTRL, SHIFTCTRL, PINCTRL and CLKDEV registers for each of the four state machines.

PREVAS

PREVAS is a stand-alone program and is quite simple, but useful nevertheless. There are a few variables that can be set:

- The number of side select control bits (the default is 2)
- Whether the MSB of the side select count is an enable bit (the default is yes)
- If the binary values of the instructions will be included in the display
- If the decimal values of the instructions will be included in the display
- If the hex values of the instructions will be included in the display

PREVAS reads the program data and EXECCTRL register from a *.PIO file.

The initial releases of PREVAS are very simple and can be caught out. It can't tell if the *.word n* and *.origin n* directives have been used as these don't appear in the program. It takes its input data strictly from the assembled program array and program arguments. A .word value will be reverse assembled as an instruction (and really confuse you!). The .origin value isn't known so it will always display the memory addresses from 0 to 31.

PREVAS reads the EXECCTRL register to determine the positions of *.wrap* and *.wrap target* in order to display them in the listing.

Note that, although PREVAS decodes the program memory, it won't necessarily tell you what a particular state machine will do because, apart from *wrap* and *wrap target* it doesn't read any of the state machine registers.

PREDIT

This stand-alone program can be used to calculate the values for the state machine registers. It more or less duplicates the helper functions but has a console interface. It includes all possible settings for these registers (including some illegal ones!) and, of course, can save and load *.PIO files. It is probably of greatest use when working on a system with several state machines in operation. It allows the calculated values to be seen in real time as options and settings are changed within the registers. As yet it does not calculate settings for CLKDIV.

Appendix F
USB Serial Console

Normally the PicoMiteVGA is used with an attached VGA monitor and keyboard. However, a USB virtual serial port connected to a desktop computer or laptop can also be used as a console. This is particularly useful when transferring programs to/from the personl computer.

The virtual serial port acts like a normal serial port but it operates over USB. Windows 10 and 11 includes a driver for this virtual serial port but with other other operating systems you may have to load a driver to make it work with the operating system.

Windows 7 and 8.1

The USB serial port uses the CDC protocol and the drivers for this are standard in Windows 10 and 11 and will load automatically.

The Raspberry Pi Foundation lists Windows 7 or 8.1 as "unsupported" however you can use a tool like Zadig (https://zadig.akeo.ie) to install a generic driver for a "usbser" device and that should allow these computers to connect. This post describes the process: https://github.com/raspberrypi/pico-feedback/issues/118

Apple Macintosh

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

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

The USB to serial converter will be listed as something like `/dev/tty.usbmodem12345`. While still at the Terminal prompt you can run the terminal emulator at 38400 baud by using the command:

```
screen   /dev/tty.usbmodem12345 38400
```

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

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

Linux

For Linux see these posts:

https://www.thebackshed.com/forum/ViewTopic.php?TID=14157&PID=175474#175474#175466

Appendix G
Programming in BASIC - A Tutorial

The PicoMiteVGA is programmed using the BASIC programming language. The PicoMiteVGA version of BASIC is called MMBasic which loosely emulates the Microsoft BASIC interpreter that was popular years ago.

The BASIC language was introduced in 1964 by Dartmouth College in the USA as a computer language for teaching programming and accordingly it is easy to use and learn. At the same time, it has proved to be a competent and powerful programming language and as a result it became very popular in the late 70s and early 80s. Even today some large commercial data systems are still written in the BASIC language (primarily Pick Basic).

For the PicoMiteVGA the greatest advantage of BASIC is its ease of use. Some more modern languages such as C and C++ can be truly mind bending but with BASIC you can start with a one line program and get something sensible out of it. MMBasic is also powerful in that you can draw sophisticated graphics, manipulate the external I/O pins to control other devices and communicate with other devices using a range of built-in communications protocols.

Command Prompt

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

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

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

```
PRINT 2 + 2
```

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

This ability to test a command at the command prompt is useful when you are learning to program in BASIC, so it would be worthwhile having the PicoMiteVGA handy for the occasional test while you are working through this tutorial.

Structure of a BASIC Program

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

A program consists of a number of statements or commands, each of which will cause the BASIC interpreter to do something (the words *statement* and *command* generally mean the same and are used interchangeable in this tutorial).

Normally each statement is on its own line but you can have multiple statements in the one line separated by the colon character (:).

For example;

```
A = 24.6 : PRINT A
```

Each line can start with a line number. Line numbers were mandatory in the early BASIC interpreters however modern implementations (such as MMBasic) do not need them. You can still use them if you wish but they have no benefit and generally just clutter up your programs.

This is an example of a program that uses line numbers:

```
50 A = 24.6
60 PRINT A
```

A line can also start with a label which can be used as the target for a program jump using the GOTO command. This will be explained in more detail when we cover the GOTO command but this is an example (the label name is `JmpBack`):

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

Comments

A comment is any text that follows the single quote character ('). A comment can be placed anywhere and extends to the end of the line. If MMBasic runs into a comment it will just skip to the end of it (ie, it does not take any action regarding a comment).

Comments should be used to explain non obvious parts of the program and generally inform someone who is not familiar with the program how it works and what it is trying to do. Remember that after only a few months a program that you have written will have faded from your mind and will look strange when you pick it up again. For this reason you will thank yourself later if you use plenty of comments.

The following are some examples of comments:

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

or

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

Older BASIC programs used the command REM to start a comment and you can also use that if you wish but the single quote character is easier to use and more convenient.

The PRINT Command

There are a number of common commands that are fundamental and we will cover them in this tutorial but arguably the most useful is the PRINT command. Its job is simple; to print something on the console. This is mostly used to output data for you to see (like the result of calculations) or provide informative messages.

PRINT is also useful when you are tracing a fault in your program; you can use it to print out the values of variables and display messages at key stages in the execution of the program.

In its simplest form the command will just print whatever is on its command line. So, for example:

```
PRINT 54
```

will display on the console the number 54 followed by a new line.

The data to be printed can be something simple like this or an expression, which means something to be calculated. We will cover expressions in more detail later but as an example the following:

```
> PRINT 3/21
 0.1428571429
>
```

would calculate the result of three divided by twenty one and display it. Note that the greater than symbol (>) is the command prompt produced by MMBasic – you do not type that in.

Other examples of the PRINT command include:

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

You can try these out at the command prompt.

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

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

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

To illustrate tabbing, the following command prints a tabbed list of numbers:

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

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

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

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

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

Will result in this output:

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

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

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

Variables

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

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

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

As a simple example:

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

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

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

```
Total_Count
ForeColour
temp3
count
x
ThisIsALongVariableName
increment.value
```

You can change the value of a variable anywhere in your program by using the assignment command, ie:

```
variable = expression
```

For example:

```
temp3 = 24.6
count = 5
CTemp = (FTemp – 32) * 0.5556
```

In the last example both CTemp and FTemp are variables and this line converts the value of FTemp (in degrees Fahrenheit) to degrees Celsius and stores the result in the variable CTemp.

Expressions

We have met the term ‘expression’ before in this tutorial and in programming it has a specific meaning. It is a formula which can be resolved by the BASIC interpreter to a single number or value.

MMBasic will evaluate numeric expressions using the same rules that we learnt at school. For example, multiplication and division are performed first followed by addition and subtraction. These are called the rules of precedence and are fully spelt out in this manual.

This means that MMBasic will resolve 2 + 3 * 6 by first multiplying 3 by 6 giving 18 then adding 2 resulting in a final value of 20. Similarly, both 5 * 4 and 10 + 4 * 3 – 2 will also resolve to 20.

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

As pointed out earlier, you can use variables in an expression exactly the same as straight numbers. For example, this will increment the value of the variable `temp` by one:

```
temp = temp + 1
```

Functions

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 evaluate this expression by first multiplying a by a, then multiplying b by b, then adding the results together. The resulting number is then passed to the SQR() function which will calculate the square root of that number (ie, the hypotenuse) and return it for the PRINT command to display.

Some other mathematical functions provided by MMBasic include:

SIN(r) – the sine of r

COS(r) – the cosine of r

TAN(r) – the tangent of r

There are many more functions available to you and they are all listed earlier in this manual.

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

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

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

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

The IF Statement

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

IF *condition* THEN *action*

The *condition* is usually a comparison such as equals, less than, more than, etc.

For example:

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

Temp would be a variable holding the current temperature (in ºC) and PRINT "Cold" the action to be done.

There are a range of tests that you can make:

=	equals	<>	not equal
<	less than	<=	less than or equals
>	greater than	>=	greater than or equals

You can also add an ELSE clause which will be executed if the initial condition tested false:

IF *condition* THEN *true-action* ELSE *false-action*

For example, this will execute different actions when the temperature is under 25 or 25 or more:

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

The previous examples all used single line IF statements but you can also have multiline IF statements.

They look like this:

```
IF condition THEN
   true-action
   true-action
ENDIF
```

or

```
IF condition THEN
   true-action
   true-action
ELSE
   false-action
   false-action
ENDIF
```

Unlike the single line IF statement you can have many true actions with each on their own line and similarly many false actions. Generally the single line IF statement is handy if you have a simple action that needs to be taken while the multiline version is much easier to understand if the actions are numerous and more complicated.

An example of a multiline IF statement with more than one action is:

```
IF Amount < 100 THEN
    PRINT "Too low"
    PRINT “Minimum value is 100”
ELSE
    PRINT "Input accepted"
    SaveToSDCard
    PRINT "Enter second amount"
ENDIF
```

Note that in the above example each action is indented to show what part of the IF structure it belongs to. Indenting is not mandatory but it makes a program much easier to understand for someone who is not familiar with it and therefore it is highly recommended.

In a multiline IF statement you can make additional tests using the ELSE IF command. This is best explained by using an example (the temperatures are all in ºC):

```
IF Temp < 0 THEN
    PRINT “Freezing”
ELSE IF Temp < 20 THEN
    PRINT “Cold”
ELSE IF Temp < 35 THEN
    PRINT “Warm”
ELSE
    PRINT “Hot”
ENDIF
```

The ELSE IF can use the same tests as an ordinary IF (ie, <, <=, etc) but that test will only be made if the preceding test was false. So, for example, you will only get the message *Warm* if `Temp < 0` failed, and `Temp < 20` failed but `Temp < 35` was true. The final ELSE will catch the case where all the tests were false.

An expression like `Temp < 20` is evaluated by MMBasic as either true or false with true having a value of one and false zero. You can see this if you entered the following at the console:

```
PRINT 30 > 20
```

MMBasic will print 1 meaning that the value of the expression is true.

Similarly the following will print 0 meaning that the expression evaluated to false.

```
PRINT 30 < 20
```

The IF statement does not really care about what the condition actually is, it just evaluates the condition and if the result is zero it will take that as false and if non zero it will take it as true. This allows for some handy shortcuts. For example, if `BalanceCorrect` is a variable that is true (non zero) when some feature of the program is correct then the following can be used to make a decision based on that value:

```
IF BalanceCorrect THEN …do something…
```

FOR Loops

Another common requirement in programming is repeating a set of actions. For instance, you might want to step through all seven days in the week and perform the same function for each day. BASIC provides the FOR loop construct for this type of job and it works like this:

```
FOR day = 1 TO 7
  Do something based on the value of 'day'
NEXT day
```

This starts by creating the variable `day` and assigning the value of 1 to it. The program will then execute the following statements until it comes to the NEXT statement. This tells the BASIC interpreter to increment the value of `day`, go back to the previous FOR statement and re-execute the following statements a second time. This will continue looping around until the value of `day` exceeds 7 and the program will then exit the loop and continue with the statements following the NEXT statement.

As a simple example, you can print the numbers from one to ten like this:

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

The comma at the end of the PRINT statement tells the interpreter to tab to the next tab column after printing the number and the semicolon will leave the cursor on this line rather than automatically moving to the next line. As a result, the numbers will be printed in neat columns across the page.

This is what you would see:

```
1       2       3       4       5       6       7       8       9       10
```

The FOR loop also has a couple of extra tricks up it sleeves. You can change the amount that the variable is incremented by using the STEP keyword. So, for example, the following will print just the odd numbers:

```
FOR nbr = 1 TO 10 STEP 2
  PRINT nbr,;
NEXT nbr
```

The value of the step (or increment value) defaults to one if the STEP keyword is not used but you can set it to whatever number you want.

When MMBasic is incrementing the variable it will check to see if the variable has exceeded the TO value and, if it has, it will exit from the loop. So, in the above example, the value of `nbr` will reach nine and it will be printed but on the next loop `nbr` will be eleven and at that point execution will leave the loop. This test is also applied at the start of the loop (ie, if in the beginning the value of the variable exceeds the TO value (and STEP is positive) the loop will never be executed, not even once).

By setting the STEP value to a negative number you can use the FOR loop to step down from a high number to low. In that case the starting number must be greater than the TO number.

For example, the following will print the numbers from 1 to 10 in reverse:

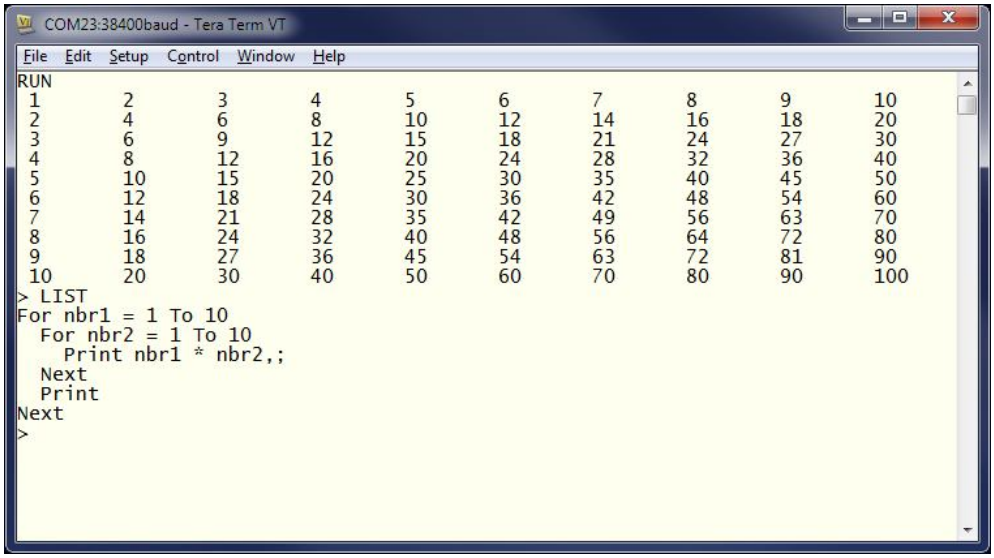
```
FOR nbr = 10 TO 1 STEP -1
  PRINT nbr,;
NEXT nbr
```

Multiplication Table

To further illustrate how loops work and how useful they can be, the following short program will use two FOR loops to print out the multiplication table that we all learnt at school. The program for this is not complicated:

```
FOR nbr1 = 1 to 10
  FOR nbr2 = 1 to 10
    PRINT nbr1 * nbr2,;
  NEXT nbr2
  PRINT
NEXT nbr1
```

The output is shown in the following screen grab, 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 to understand it. You can also terminate multiple loops using a comma separated list of variables in the NEXT statement. For example:

```
FOR var1 = 1 TO 5
  FOR var2 = 10 to 13
    PRINT var1 * var2
NEXT var1, var2
```

DO Loops

Another method of looping is the DO…LOOP structure which looks like this:

```
DO WHILE condition
   statement
   statement
LOOP
```

This will start by testing the condition and if it is true the statements will be executed until the LOOP command is reached, at which point the 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 PicoMiteVGA .

A variation on the DO-LOOP structure is the following:

```
DO
   statement
   statement
LOOP UNTIL condition
```

In this arrangement the loop is first executed once, the condition is then tested and if the condition is false, the loop will be repeatedly executed until the condition becomes true. Note that the test in LOOP UNTIL is the inverse of DO WHILE.

For example, similar to the previous example, the following will also print "Hello" for four seconds:

```
Timer = 0
DO
  PRINT "Hello"
LOOP UNTIL Timer >= 4000
```

Both forms of the DO-LOOP essentially do the same thing, so you can use whatever structure fits with the logic that you wish to implement.

Finally, it is possible to have a DO Loop that has no conditions at all - ie,

```
DO
  statement
  statement
LOOP
```

This construct will continue looping forever and you, as the programmer, will need to provide a way to explicitly exit the loop (the EXIT DO command will do this). For example:

```
Timer = 0
DO
  PRINT "Hello"
  IF Timer >= 4000 THEN EXIT DO
LOOP
```

Console Input

As well as printing data for the user to see your programs will also want to get input from the user. For that to work you need to capture keystrokes from the console and this can be done with the INPUT command. In its simplest form the command is:

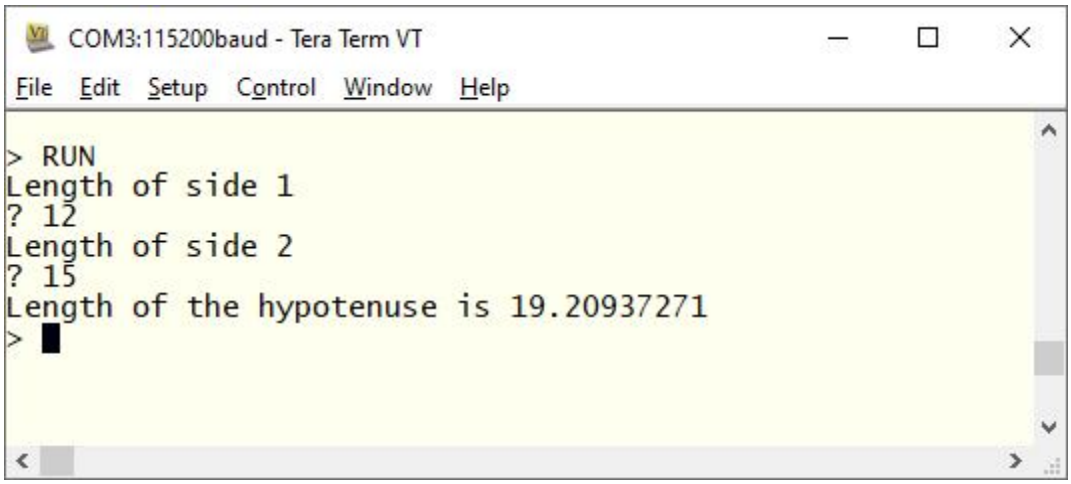
```
INPUT var
```

This command will print a question mark on the console's screen and wait for a number to be entered followed by the Enter key. That number will then be assigned to the variable `var`.

For example, the following program extends the expression for finding the hypotenuse of a triangle by allowing the user to enter the lengths of the other sides from the console.

```
PRINT "Length of side 1"
INPUT a
PRINT "Length of side 2"
INPUT b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

This is a screen capture of a typical session:



The INPUT command can also print your prompt for you, so that you do not need a separate PRINT command. For example, this will work the same as the above program:

```
INPUT "Length of side 1"; a
INPUT "Length of side 2"; b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

Finally, the INPUT command will allow you to input a series of numbers separated by commas with each number being saved in different variables.

For example:

```
INPUT "Enter the length of the two sides: ", a, b
PRINT "Length of the hypotenuse is" SQR(a * a + b * b)
```

If the user entered `12,15` the number `12` would be saved in the variable `a` and `15` in `b`.

Another method of getting input from the console is the LINE INPUT command. This will get the whole line as typed by the user and allocate it to a string variable. Like the INPUT command you can also specify a prompt. This is a simple example:

```
LINE INPUT "What is your name? ", s$
PRINT "Hello " s$
```

We will cover string variables later in this tutorial but for the moment you can think of them as a variable that holds a sequence of characters. If you ran the above program and typed in `John` when prompted the program would respond with `Hello John`.

Sometimes you do not want to wait for the user to hit the enter key, you want to get each character as it is typed in. This can be done with the INKEY$ function which will return the value of the character as a string consisting of just one character or an empty string (ie, contains no characters) if nothing has been entered.

GOTO and Labels

One method of controlling the flow of the program is the GOTO command. This essentially tells MMBasic to jump to another part of the program and continue executing from there. The target of the GOTO is a label and this needs to be explained first.

A label is an identifier that marks part of the program. It must be the first thing on the line and it must be terminated with the colon (:) character. The name that you use can be up to 32 characters long and must follow the same rules as for a variable's name. For example, in the following program line LoopBack is a label:

```
LoopBack:  a = a + 1
```

When you use the GOTO command to jump to that particular part of the program you would use the command like this:

```
GOTO LoopBack
```

To put all this into context the following program will print out all the numbers from 1 to 10:

```
z = 0
LoopBack:  z = z + 1
PRINT z
IF z < 10 THEN GOTO LoopBack
```

The program starts by setting the variable `z` to zero then incrementing it to 1 in the next line. The value of `z` is printed and then tested to see if it is less than 10. If it is less than 10 the program execution will jump back to the label `LoopBack` where the process will repeat. Eventually the value of `z` will be more than 10 and the program will run off the end and terminate.

Note that a FOR loop can do the same thing (and is simpler) so this example is purely designed to illustrate what the GOTO command can do.

In the past the GOTO command gained a bad reputation. This is because using GOTOs it is possible to create a program that continuously jumps from one point to another (often referred to as "spaghetti code") and that type of program is almost impossible for another programmer to understand. With constructs like the multiline IF statements the need for the GOTO statement has been reduced and it should be used only when there is no other way of changing the program's flow.

Testing for Prime Numbers

The following is a simple program which brings together many of the programming features previously discussed.

```
DO
  InpErr:
  PRINT
  INPUT "Enter a number: "; a
  IF a < 2 THEN
    PRINT "Number must be equal or greater than 2"
    GOTO InpErr
  ENDIF

  Divs = 0
  FOR x = 2 TO SQR(a)
    r = a/x
```

```
    IF r = FIX(r) THEN Divs = Divs + 1
  NEXT x

  PRINT a " is ";
  IF Divs > 0 THEN PRINT "not ";
  PRINT "a prime number."
LOOP
```

This will first prompt (on the console) for a number and, when it has been entered, it will test if that number is a prime number or not and display a suitable message.

It starts with a DO Loop that does not have a condition – so it will continue looping forever. This is what we want. It means that when the user has entered a number, it will report if it is a prime number or not and then loop around and ask for another number. The way that the user can exit the program (if they wanted to) is by typing the break character (normally CTRL-C).

The program then prints a prompt for the user which is terminated with a semicolon character. This means that the cursor is left at the end of the prompt for the INPUT command which will get the number and store it in the variable a.

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

We are now ready to test if the number is a prime number. The program uses a FOR loop to step through the possible divisors testing if each one can divide evenly into the entered number. Each time it does the program will increment the variable `Divs`.

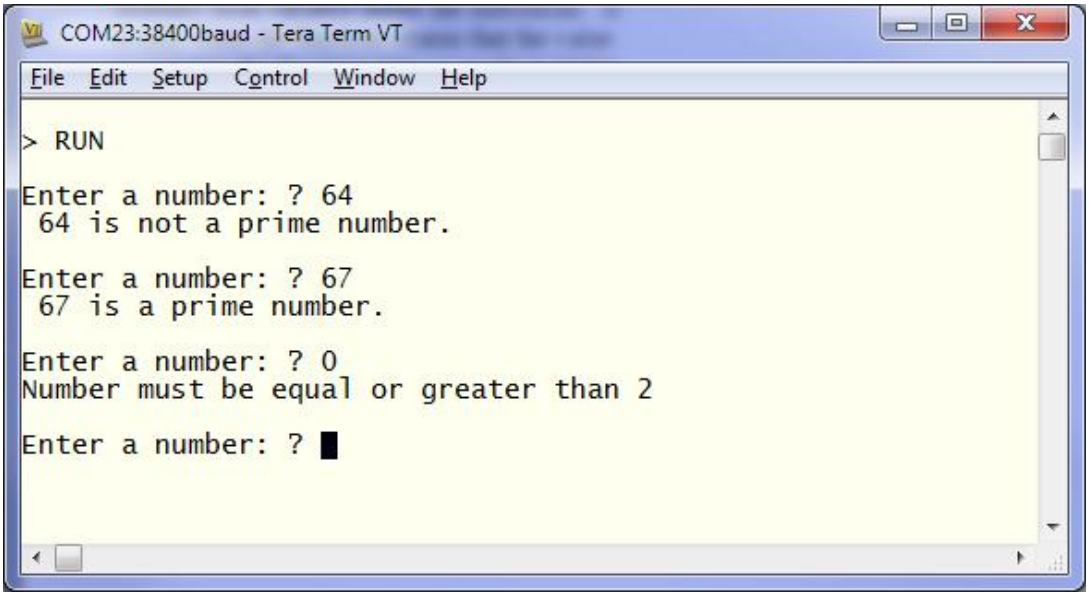
Note that the test is done with the function FIX(r) which simply strips off any digits after the decimal point. So, the condition `r = FIX(r)` will be true if `r` is an integer (ie, has no digits after the decimal point).

Finally, the program will construct the message for the user. The key part is that if the variable `Divs` is greater than zero it means that one or more numbers were found that could divide evenly into the test number. In that case the IF statement inserts the word "not" into the output message.

For example, if the entered number was 21 the user will see this response:

```
21 is not a prime number.
```

This is the result of running the program and some of the output:



You can test this program by using the editor (the EDIT command) to enter it.

Using your newly learnt skills you could then have a shot at making it more efficient. For example, because the program counts how many times a number can be divided into the test number it takes a lot longer than it should to detect a non prime number. The program would run much more efficiently if it jumped out of the FOR loop at the first number that divided evenly. You could use the GOTO command to do this or you could use the command EXIT FOR – that would cause the FOR loop to terminate immediately.

Other efficiencies include only testing the division with odd numbers (by using an initial test for an even number then starting the FOR loop at 3 and using STEP 2) or by only using prime numbers for the test (that would be much more complicated).

Arrays

Arrays are something which you will probably not think of as useful at first glance but when you do need to use them you will find them very handy indeed.

An array is best thought of as a row of letterboxes for a block of units or condos as shown on the right. The letterboxes are all located at the same address and each box represents a unit or condo at that address. You can place a letter in the box for unit one, or unit two, etc.

Similarly an array in BASIC is a single variable with multiple sub units (called *elements* in BASIC) which are numbered. You can place data in element one, or element two, etc. In BASIC an array is created by the DIM command, for example:

```
DIM numarr(300)
```

This creates an array with the name of `numarr` containing 301 elements (think of them as letterboxes) ranging from 0 to 300. By default an array will start from zero so this is why there is an extra element making the total 301. To specify a specific element in the array (ie, a specific letterbox) you use an index which is simply the number of the array element that you wish to access. For example, if you want to set element number 100 in this array to (say) the number 876, you would do it this way:

```
numarr(100) = 876
```

Normally the index to an array is not a constant number as in this example (ie, 100) but a variable which can be changed to access different array elements.

As an example of how you might use an array, consider the case where you would like to record the maximum temperature for each day of the year and, at the end of the year, calculate the overall average. You could use ordinary variables to record the temperature for each day but you would need 365 of them and that would make your program 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 such an array is specified using two indices separated by a comma. For example:

```
LetterBox(floor, unit)
```

As a practical example, assume that you needed to record the maximum temperature for each day over five years. To do this you could dimension the array as follows:

```
DIM days(365, 5)
```

The first index is the day in the year and the second is a number representing the year. If you wanted to set day 100 in year 3 to 24 degrees you would do it like this:

```
days(100, 3) = 24
```

In MMBasic for the PicoMiteVGA you can have up to five dimensions (this is different from some other versions of MMBasic which support eight dimensions). The maximum size of an array is only limited by the amount of free RAM that is available.

Integers

So far all the numbers and variables that we have been using have been floating point. As explained before, floating point is handy because it will track digits after the decimal point and when you use division it will return a sensible result. So, if you just want to get things done and are not concerned with the details you should stick to floating point.

However, the limitation of floating point is that it stores numbers as an approximation with an accuracy of 14 digits on the PicoMiteVGA . Most times this characteristic of floating point numbers is not a problem but there are some cases where you need to accurately store larger numbers.

As an example, let us say that you want to manipulate time accurately down to the microsecond so that you can compare two different date/times to work out which one is earlier. The easy way to do this is to convert the date/time to the number of microseconds since some date (say 1st Jan in year zero) - then finding the earliest of the two is just a matter of using an arithmetic compare in an IF statement.

The problem is that the number of microseconds since that date will exceed the accuracy range of floating point variables and this is where integer variables come in. An integer variable can accurately hold very large numbers up to nine million million million (or ±9223372036854775807 to be precise).

The downside of using an integer is that it cannot store fractions (ie, numbers after the decimal point). Any calculation that produces a fractional result will be rounded up or down to the nearest whole number when assigned to an integer. However this characteristic can be handy when dealing with money – for example, you don't want to send someone a bill for $100.13333333333.

It is easy to create an integer variable, just add the percent symbol (%) as a suffix to a variable name. For example, `sec%` is an integer variable. Within a program you can mix integers and floating point and MMBasic will make the necessary conversions but if you want to maintain the full accuracy of integers you should avoid mixing the two.

Just like floating point you can have arrays of integers with up to five dimensions, all you need to do is add the percent character as a suffix to the array name. For example: `days%(365, 5)`.

Beginners often get confused as to when they should use floating point or integers and the answer is simple… always use floating point unless you need an extremely 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 then have a case insensitive comparison. For example:

```
IF UCASE$(Car$) = "HOLDEN" THEN PRINT "Was an Aussie made car"
```

You can have arrays of strings but you need to be careful when you declare them as you can rapidly run out of RAM (general memory used for storing variables, etc). This is because MMBasic will by default allocate 255 bytes of RAM for each element of the array. For example, a string array with 100 elements will by default use 25K of RAM. To alleviate this you can use the LENGTH qualifier to limit the maximum size of each element. For instance, if you know that the maximum length of any string that will be stored in the array will be less than 20 characters you can use the following declaration to allocate just 20 bytes for each element:

```
DIM MyArray$(100) LENGTH 20
```

The resultant array will only use 2K of RAM.

Manipulating Strings

String handling is one of MMBasic's strengths and using a few simple functions you can pull apart and generally manipulate strings. The basic string functions are:

LEFT$(string$, nbr)	Returns a substring of *string$* with *nbr* of characters from the left (beginning) of the string.
RIGHT$(string$, nbr)	Same as the above but return *nbr* of characters from the right (end) of the string.

MID$(string$, pos, nbr) Returns a substring of *string$* with *nbr* of characters starting from the character *pos* in the string (ie, the middle of the string).

For example if `S$ = "This is a string"`

then: `R$ = LEFT$(S$, 7)` would result in the value of R$ being set to: "This is"
and: `R$ = RIGHT$(S$, 8)` would result in the value of R$ being set to: "a string"
finally: `R$ = MID$(S$, 6, 2)` would result in the value of R$ being set to: "is"

Note that in MID$() the first character position in a string is number 1, the second is number 2 and so on. So, counting the first character as one, the sixth position is the start of the word "is".

Another useful function is:

INSTR(string$, pattern$) Returns a number representing the position at which *pattern$* occurs in *string$*.

This can be used to search for a string inside another string. The number returned is the position of the substring inside the main string. Like with MID$() the start of the string is position 1.

For example if S$ = "This is a string"

Then: `pos = INSTR(S$, " ")`

would result in `pos` being set to the position of the first space in S$ (ie, 5).

INSTR() can be combined with other functions so this would return the first **word** in S$:

```
R$ = LEFT$(S$, INSTR(S$, " ") - 1)
```

There is also an extended version of INSTR():

INSTR(pos, string$, pattern$) Returns a number representing the position at which *pattern$* occurs in *string$* when starting the search at the character position *pos*.

So we can find the second word in S$ using the following:

```
pos = INSTR(S$, " ")
R$ = LEFT$(S$, INSTR(pos + 1, S$, " ") - 1)
```

This last example is rather complicated so it might be worth working through it in detail so that you can understand how it works.

Note that INSTR() will return the number zero if the sub string is not found and that any string function will throw an error (and halt the program) if that is used as a character position. So, in a practical program you would first check for zero being returned by INSTR() before using that value.

For example:

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

Scientific Notation

Before we finish discussing data types we need to cover off the subject of floating point numbers and scientific notation.

Most numbers can be written normally, for example 11 or 24.5, but very large or small numbers are more difficult. For example, it has been estimated that the number of grains of sand on planet Earth is 7500000000000000000. The problem with this number is that you can easily lose track of how many zeros there are in the number and consequently it is difficult to compare this with a similar sized number.

A scientist would write this number as 7.5×10^{18} which is called scientific notation and is much easier to comprehend.

MMBasic will automatically shift to scientific notation when dealing with very large or small floating point numbers. For example, if the above number was stored in a floating point variable the PRINT

command would display it as `7.5E+18` (this is BASIC's way of representing 7.5 x 10^{18}). As another example, the number 0.0000000456 would display as `4.56E-8` which is the same as 4.56 x 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. Similarly, when you use these variables in an expression you do not need the type suffix: For example:

```
City = "Sydney"
```

You can also use the keyword INTEGER to define a number of integer variables and FLOAT to do the same for floating point variables. This type of notation can similarly be used to define arrays.

For example:

```
DIM INTEGER seconds(200)
```

Another method of defining the variables type is to use the keyword AS. For example:

```
DIM Car AS STRING, Name AS STRING, Street AS STRING
```

This is the method used by Microsoft (MMBasic tries to maintain Microsoft compatibility) and it is useful if the variables have different types. For example:

```
DIM Car AS STRING, Age AS INTEGER, Value AS FLOAT
```

You can use any of these methods of defining a variable's type, they all act the same.

The advantage of defining variables using the DIM command is that they are clearly defined (preferably at the start of the program) and their type (float, integer or string) is not subject to misinterpretation.

You can strengthen this by using the following commands at the very top of your program:

```
OPTION EXPLICIT
OPTION DEFAULT NONE
```

The first specifies to MMBasic that all variables must be explicitly defined using DIM before they can be used. The second specifies that the type of all variables must be specified when they are created.

Why are these two commands important?

The first can help avoid a common programming error which is where you accidently misspell a variable's name. For example, your program might have the current temperature saved in a variable called `Temp` but at one point you accidently misspell it as `Tmp`. This will cause MMBasic to automatically create a variable called `Tmp` and set its value to zero.

This is obviously not what you want and it will introduce a subtle error which could be hard to find, even if you were aware that something was not right. On the other hand, if you used the OPTION EXPLICIT command at the start of your program MMBasic would refuse to automatically create the variable and instead would display an error thereby saving you from a probable headache.

The command OPTION DEFAULT NONE further helps because it tells MMBasic that the programmer must specifically specify the type of every variable when they are declared. It is easy to

forget to specify the type and allowing MMBasic to automatically assume the type can lead to unexpected consequences.

For small, quick and dirty programs, it is fine to allow MMBasic to automatically create variables but in larger programs you should always disable this feature with OPTION EXPLICIT and strengthen it with OPTION DEFAULT NONE.

When a variable is created it is set to zero for float and integers and an empty string (ie, contains no characters) for a string variable. You can set its initial value to something else when it is created using DIM. For example:

```
DIM FLOAT nbr = 12.56
DIM STRING Car = "Ford", City = "Perth"
```

You can also initialise arrays by placing the initialising values inside brackets like this:

```
DIM s$(2) = ("zero", "one", "two")
```

Note that because arrays start from zero by default this array actually has three elements with the index numbers of 0, 1 and 2. This is why we needed three string constants to initialise it.

Constants

A common requirement in programming is to define an identifier that represents a value without the risk of the value being accidently changed - which can happen if variables were used for this purpose. These are called constants and they can represent I/O pin numbers, signal limits, mathematical constants and so on.

You can create a constant using the CONST command. This defines an identifier that acts like a variable but is set to a value that cannot be changed.

For example, if you wanted to check the voltage of a battery connected to pin 31 you could define the relevant values thus:

```
CONST BatteryVoltagePin = 31
CONST BatteryMinimum = 1.5
```

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

For example:

```
SETPIN BatteryVoltagePin, AIN
IF PIN(BatteryVoltagePin) < BatteryMinimum THEN SoundAlarm
```

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

Subroutines

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

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

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

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

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

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

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

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

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

For example:

```
IF A < B THEN ErrMsg "Number too small"
```

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

A subroutine can have any number of arguments which can be float, integer or string with each argument separated by a comma.

Within the subroutine the arguments act like ordinary variables but they exist only within the subroutine and will vanish when the subroutine ends. You can have variables with the same name in the main program and they will be hidden within the subroutine and be different from arguments defined for the subroutine.

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

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

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

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

For example, the following is identical to the above example:

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

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

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

This is best explained by example:

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

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

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

Passing arguments by reference is handy because it allows a subroutine to pass values back to the code that called it. However it could lead to trouble if a subroutine used the variable representing an argument as a general purpose variable and changed its value. Then, if it were called with a variable as an argument, that variable would be inadvertently changed. For this reason **you should avoid manipulating variables representing arguments inside a subroutine**, instead assign the value to a local variable (see below) and manipulate that instead.

When you call a subroutine you can omit some (or all) of the parameters and they will take the value of zero (for floats or integers) or an empty string. This is handy as your subroutine can tell if a parameter is missing and act accordingly.

For example, here is our subroutine to generate an error message but this version can be used without specifying an error message as a parameter:

```
SUB ErrMsg  Msg$
  IF Msg$ = "" THEN
    PRINT "Error detected"
  ELSE
    PRINT "Error: " + Msg$
  ENDIF
END SUB
```

Within a subroutine you can use most features of MMBasic including calling other subroutines, IF…THEN commands, FOR…NEXT loops and so on. However, one thing that you cannot do is jump out of a subroutine using GOTO (if you do the result will be undefined and may cause your hair to turn grey).

Normally the subroutine will exit when the END SUB command is reached but you can also terminate the subroutine early by using the EXIT SUB command.

Functions

Functions are similar to subroutines with the main difference being that a function is used to return a value in an expression. For example, if you wanted a function to convert a temperature from degrees Celsius to Fahrenheit you could define:

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

Then you could use it in an expression:

```
Input "Enter a temperature in Celsius: ", t
PRINT "That is the same as" Fahrenheit(t) "F"
```

Or as another example:

```
IF Fahrenheit(temp) <= 32 THEN PRINT "Freezing"
```

You could also define the reverse:

```
FUNCTION Celsius(F)
  Celsius = (F - 32) * 0.5556
END FUNCTION
```

As you can see, the function name is used as an ordinary local variable inside the subroutine. It is only when the function returns that the value is made available to the expression that called it.

The rules for the argument list in a function are similar to subroutines. The only difference is that parentheses are always required around the argument list when you are calling a function, even if there are no arguments (parentheses are optional when calling a subroutine).

To return a value from the function you assign a value to the function's name within the function. If the function's name is terminated with a type suffix (ie, $, a % or a !) the function will return that type (string, integer or float), otherwise it will return whatever the OPTION DEFAULT is set to. For example, the following function will return a string:

```
FUNCTION LVal$(nbr)
  IF nbr = 0 THEN LVal$ = "False" ELSE LVal$ = "True"
END FUNCTION
```

You can explicitly specify the type of the function by using the AS keyword and then you do not need to use a type suffix (similar to defining a variable using DIM).

This is the above example rewritten to take advantage of this feature:

```
FUNCTION LVal(nbr) AS STRING
  IF nbr = 0 THEN LVal = "False" ELSE LVal = "True"
END FUNCTION
```

In this case the type returned by the function `LVal` will be a string.

As for subroutines you can use most features of MMBasic within functions. This includes FOR…NEXT loops, calling other functions and subroutines, etc. Also, the function will return to the expression that called it when the END FUNCTION command is reached but you can also return early by using the EXIT FUNCTION command.

Local Variables

Variables that are created using DIM or that are just automatically created are called *global* variables. This means that they can be seen and used anywhere in the program including within subroutines and functions. However, inside a subroutine or function you will often need to use variables for various tasks that are internal to the subroutine/function. In portable code you do not want the name you chose for such a variable to clash with a global variable of the same name. To this end you can define a variable using the LOCAL command within the subroutine/function.

The syntax for LOCAL is identical to the DIM command, this means that the variable can be an array, you can set the type of the variable and you can initialise it to some value.

For example, this is our ErrMsg subroutine but this time it has been extended to use a local variable for joining the error message strings.

```
SUB ErrMsg  Msg$
  LOCAL STRING tstr
  tstr = "Error: " + Msg$
  PRINT tstr
END SUB
```

The variable `tstr` is declared as LOCAL within the subroutine, which means that (like the argument list) it will only exist within the subroutine and will vanish when the subroutine exits. You can have a

global variable called `tstr` in your main program and it will be different from the variable `tstr` in the subroutine (in this case the global `tstr` will be hidden within the subroutine).

You should always use local variables for operations within your subroutine or function because they help make the module much more self contained and portable.

Static Variables

LOCAL variables are reset to their initial values (normally zero or an empty string) every time the subroutine or function starts, however there are times when you would like the variable to retain its value between calls. This type of variable is defined with the STATIC command.

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

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

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

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

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

```
STATIC INTEGER var = 123
```

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

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

Examples include:

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

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

Calculate Days

We have covered a lot of programming commands and techniques so far in this tutorial and before we finish it would be worth giving an example of how they work together. The following is an example that uses many features of the BASIC language to calculate the number of days between two dates:

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

OPTION EXPLICIT
OPTION DEFAULT NONE

DIM STRING s
DIM FLOAT d1, d2

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

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

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

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

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

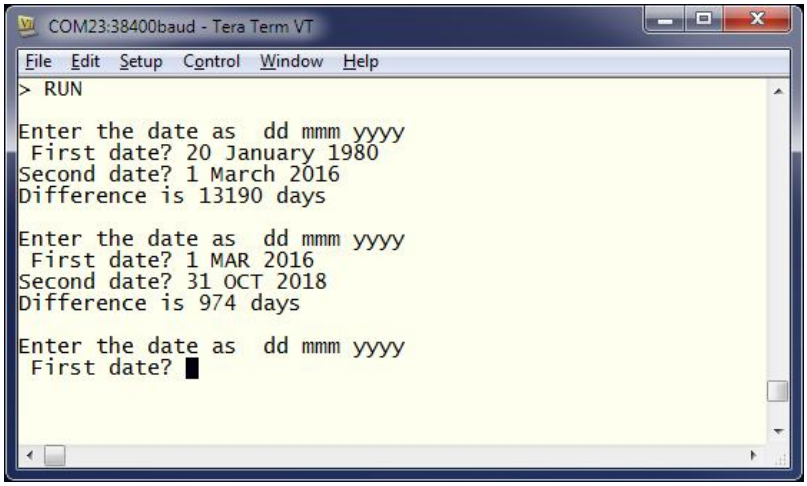
Note that the line starting `LOCAL STRING Month(11)` has been wrapped around because of the limited page width – it is one line as follows:

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

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

When this program is run it will ask for the two dates to be entered and you need to use the form of: `dd mmm yyyy`.

This screen capture shows what the running program will look like.



The main feature of the program is the 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 properly tested and then, if possible, left untouched (in case you add some error).

There are a few features of this program that we have not covered before. The first is the MOD operator which will calculate the remainder of dividing one number into another. For example, if you divided 4 into 15 you would have a remainder of 3 which is exactly what the expression 15 MOD 4 will return. The ABS() function is also new and will return its argument as a positive number (eg, ABS(-15) will return +15 as will ABS(15)).

The EXIT FOR command will exit a FOR loop even though it has not reached the end of its looping, EXIT FUNCTION will immediately exit a function even though execution has not reached the end of the function and CONTINUE DO will immediately cause the program to jump to the end of a DO loop and execute it again.

Why would this program be useful? Well some people like to count their age in days, that way every day is a birthday! You can calculate your age in days, just enter the date that you were born and today's date. That is not particularly useful but the program itself is valuable as it demonstrates many of the characteristics of programming in MMBasic. So, work your way through the program and review each section until you understand it – it should be a rewarding experience.