Armmite F407xGT6

User Manual

MMBasic Ver 5.07.02 +

A consolidated manual for the Armmite F4 (VGT,RGT,ZGT)

Matches Firmware

5.07.02 Beta 0

Revision 0

(24 February 2025)

For more details on MMBasic go to

http://geoffg.net/maximite.html

and http://mmbasic.com

For latest update of this manual look at these links on

The Back Shed Forum and the Fruit of the Shed website.

TBS Armmite F407xGT6 Post

and FotS Armmite F407xGT6 Page

About

The Armmite F4 was conceived and developed by Peter Mather (matherp on the Back Shed Forum).

It is a port to STM32 of MMBasic developed by Geoff Graham and uses the MMBasic interpreter written by Geoff Graham (http://geoffg.net).

Support

Support questions should be raised on the Back Shed forum (http://www.thebackshed.com/forum/Microcontrollers) where there are many enthusiastic Maximite and Micromite and Armmite users who would be only too happy to help. The developers of both the Armmite F4 and MMBasic are also regulars on this forum.

Copyright and Acknowledgments

This Manual

This manual relies heavy on content from the following manuals be Geoff Graham.

Micromite User Manual

Micromite Plus User Manual

Colour Maximite 2 User Manual

Also the following manuals by Peter Mather.

Micromite Extreme User Manual

Armmite H7 User Manual

Armmite L4 User Manual

Much information is also gleaned from posts (mainly by Peter Mather) in various threads relating to Armmite F4 on The Back Shed Forum. Many contributors may recognise their work within this document and are thanked for their contributions.

Contents

Introduction

A single MMBasic firmware now supports the STM32F407VGT6 100 pin , STM32F405RGT6 64 pin (Adafruit Feather), the STM32F405RGT6 64 (WeAct Studio) and the STM32F407ZGT6 144 pin boards. These xGT6 chips are the same as the previously used by Armmite F4 (STM32F407VET6) chip except for 1Meg of flash in lieu of 512K. This has allowed these changes to the Armmite F4 firmware:

- now optimised for speed rather than size
- OPTIONS moved back into flash so not dependent on RTC battery.
- SAVED VARS moved back into flash so not dependent on RTC battery.
- LIBRARY now an additional 128K flash section separate to program memory and does not rely on SPI Windbond flash chip.
- Addition flash has allowed some extra functionality.
- No preconfigured LCD Display

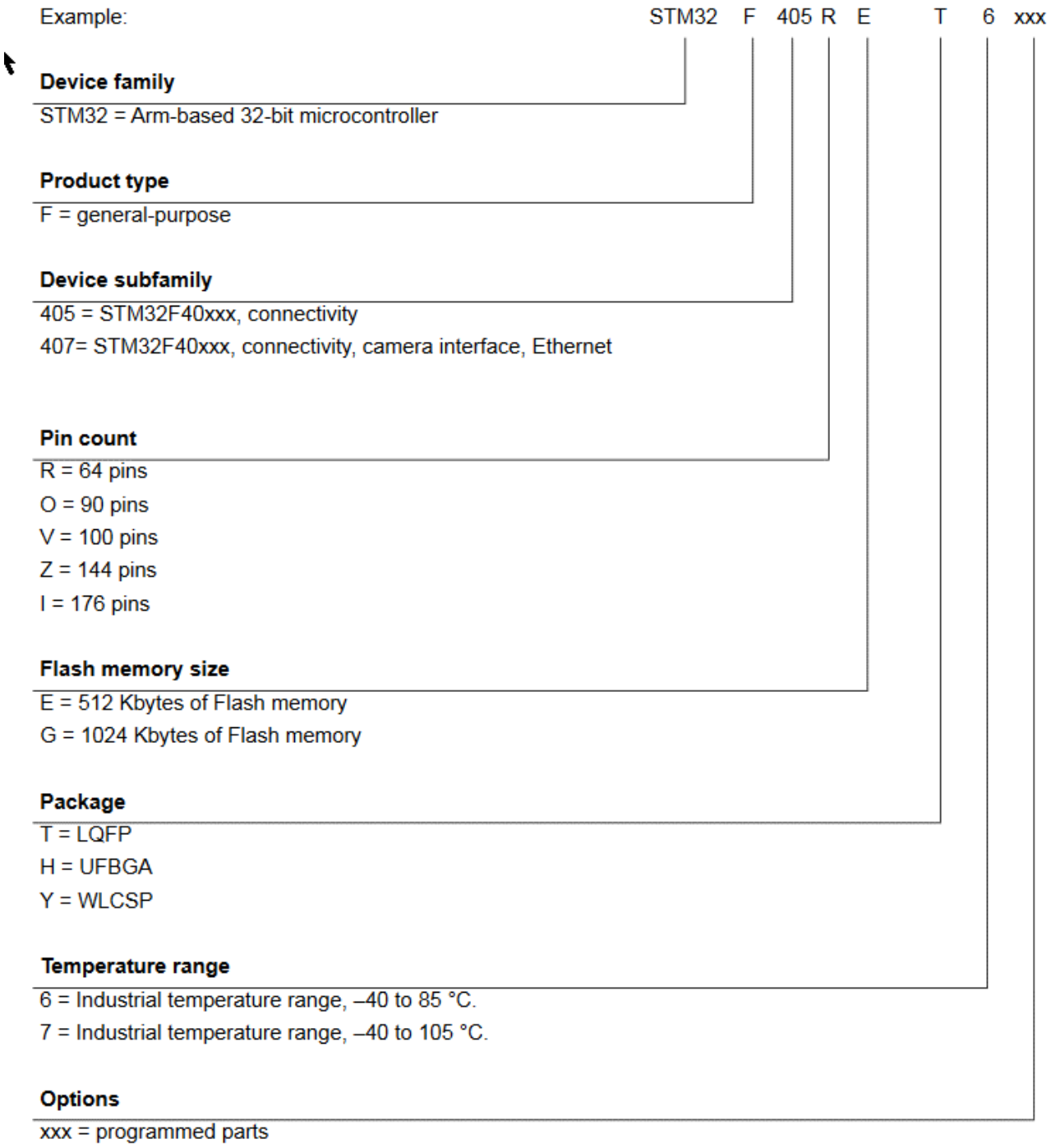
Armmite F4 (STM32F407ZGT6 development board) single board that has everything you need - USB serial port, RTC with battery, SDcard slot, TFT header. It runs faster than the MM+ and is much cheaper as a complete system. Lots of projects like Geoff's Super Clock, DDS signal generator, and boat computer are easy to port to the ArmmiteF4 with display and run superbly.

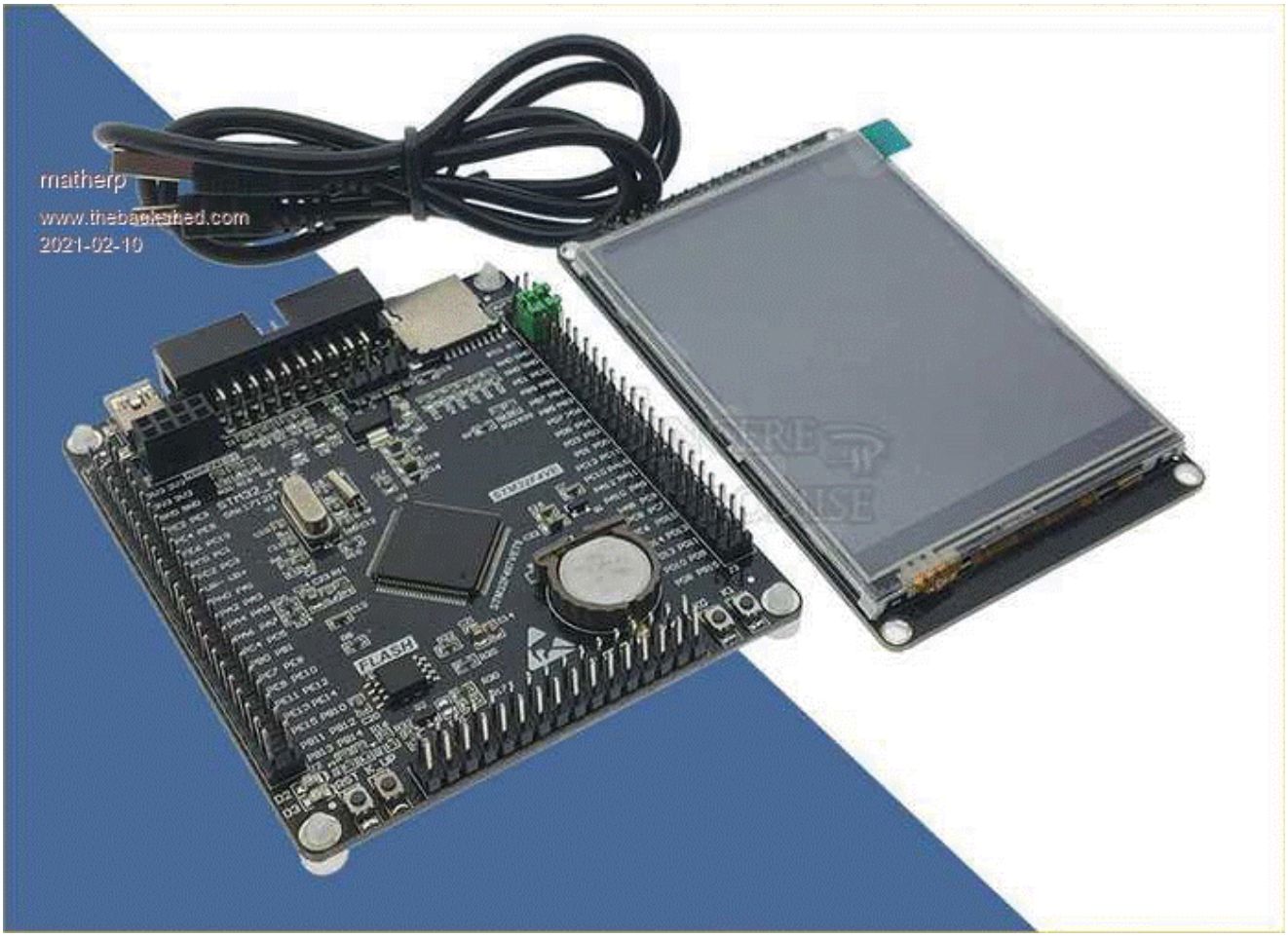
The STM32F407VET6 STM32 Cortex-M4 Development Board which has a pin compatible TFT screen available to plug directly in.

The matching display is 16-bit parallel and very fast
Buy the pair from here and many other vendors

STM32F405/7 Part Numbers Explained

The part number of the chip contains the number of pins, and amount of flash.





Note there are lots of variants of STM32F407 development boards. You must buy the one pictured for the firmware to work.

The Armmite F4 firmware version will work on this PCB with no configuration necessary and no ancillary hardware needed. i.e. the main peripherals will work immediately on power up.

The firmware can be loaded to the board over the USB port with no programmer needed and the MMBasic console will be on the USB, no USB/UART needed.

The speed of the port is about 1.3x faster than a MM+ at 120MHz. Peripherals include 4 x UART, 2 x SPI, 8 x PWM/Servo, 2 x I2C, 13 x 12-bit ADC, 2 x 12-bit DAC, 16-bit parallel TFT I/F supporting the screen above and any SSD1963 display using an adapter board.

There are 47 user configurable pins (DOUT, DIN, etc.)

The firmware uses a different way of interfacing with the TFT screens using the STM32F407's FSMC interface. This treats the screen as a memory device and allows for very fast performance (clear screen on the ILI9341 currently takes 6 mSec). Touch is supported with full GUI functionality. The pins used by the FSMC interface are not available to MMBasic if the screen is not used.

The basic features of the Armmite F4 are:

- **Low cost affordable fun**. The firmware (including the BASIC interpreter) is completely free. The STM32F407ZGT6 development board is low cost and needs no assembly. The firmware can be loaded using free software so a programmer or special equipment is not required to get started. If the specified LCD is purchased it plugs in with no modification required.

- **Instant startup** into the BASIC interpreter. Program space is 132KB, enough for reasonable sized programs while general RAM used for variables, buffers etc. is 114KB.
- **Full featured BASIC interpreter** with double precision floating point, 64-bit integers and string variables, long variable names, arrays of floats, integers or strings with multiple dimensions, extensive string handling and user defined subroutines and functions. Typically, it will execute a program at up to 90,000 lines per second.
- **PS2 Keyboard support.** The keyboard can have US, UK, FR, GR, BE, IT or ES key mappings.
- **Stereo audio output** can play WAV and FLAC files and generate precise sine wave tones.
- **A full screen editor** is built into the firmware. It includes advanced features such as colour coded syntax, search and copy, cut and paste to and from a clipboard. With one key press the program can be saved and run. If an error occurs another key press will return to the editor with the cursor placed on the line that caused the error.
- **Full support for SD cards** including editing and running programs on the SD card as well as opening files for reading, writing or random access. Cards up to 32GB formatted in FAT32 are supported and the files can also be read and written on personal computers running Windows, Linux or the Mac operating system.
- **Programs can be easily transferred** from another computer (Windows, Mac or Linux) using the SD card, XModem protocol or by streaming the program over the serial console input.
- **Battery backed clock** will keep the correct time, even with the power disconnected.
- **Power is 5 volts at 70mA without LCD and 140mA with the standard LCD. (backlight on). This will increase with the bigger displays.**

Micromite Family Summary

The Micromite Family consists of five major types, the standard Micromite, the Micromite Plus, the Micromite eXtreme, the Pi-cromite, the Armmite L4, the Armmite F4 and the Armmite H7. All use the same BASIC interpreter and have the same basic capabilities however they differ in the number of I/O pins, the amount of memory, the displays that they support and their intended use.

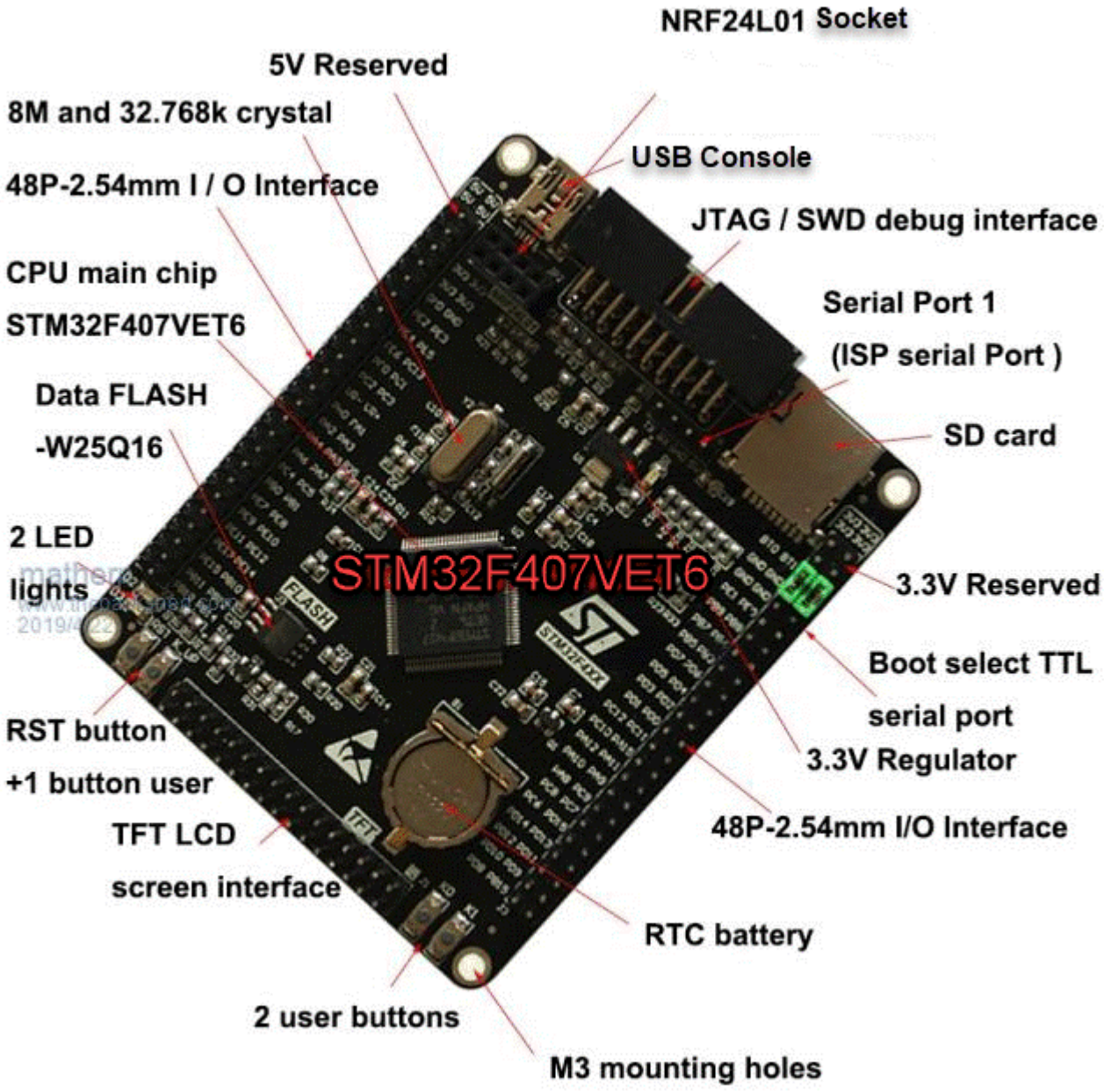
Standard Micromite	Comes in a 28-pin or 44-pin package and is designed for small embedded controller applications and supports small LCD display panels. The 28-pin version is particularly easy to use as it is easy to solder and can be plugged into a standard 28-pin IC socket.
Micromite Plus	This uses a 64-pin and 100-pin TQFP surface mount package and supports a wide range of touch sensitive LCD display panels from 1.44" to 8" in addition to the standard features of the Micromite. It is intended as a sophisticated controller with easy to create on-screen controls such as buttons, switches, etc.
Micromite eXtreme	This comes in 64, 100-pin and 144-pin TQFP surface mount packages. The eXtreme version has all the features of the other two Micromites but is faster and has a larger memory capacity plus the ability to drive a VGA monitor for a large screen display. It works as a powerful, self contained computer with its own BASIC interpreter and instant start-up.
Pi-cromite	Runs on all versions of the Raspberry Pi with a 40-pin I/O connector. No analogue input capability but 5x faster than a Micromite eXtreme when running on a Pi 3.
Armmite L4	Runs on the STM32L43x series chips. Is targeted for low power usage.
Armmite F4	Runs on Armmite F4 (STM32F407VET6 development board) single board that has everything you need.
Armmite H7	Runs on the NUCLEO-H743ZI processor. This is the highest speed single-chip Micromite currently available.
Colour Maximite 2	The Colour Maximite 2 is a small self contained computer inspired by the home computers of the early 80's such as the Tandy TRS-80, Commodore 64 and Apple II. It includes its own BASIC interpreter and powers up in under a second into the BASIC interpreter. Output is to a VGA screen rather than LCDPanels.
PicoMite	Latest port by Peter Mather. Runs on a low cost Raspberry Pi Pico controller.

	Micromite		Micromite Plus		Micromite eXtreme	Armmite F4	Armmite H7
	28-pin DIP	44-pin SMD	64-pin SMD	100-pin SMD	100/144/64 -pin SMD	100 pin STM32F407VET6	NUCLEO-H743ZI2
Maximum CPU Speed	48 MHz	48 MHz	120 MHz	120 MHz	252MHz	168MHz	480MHz
Maximum BASIC Program Size	59 KB	59 KB	100 KB	100 KB	540 KB	132K	512KB
RAM Memory Size	52 KB	52 KB	108 KB	108 KB	460 KB	114K	512KB
Clock Speed (MHz)	5 to 48	5 to 48	5 to 120	5 to 120	200 to 252	168MHz	400
Total Number of I/O pins	19	33	45	77	75/115/46	47	102
Number of Analog Inputs	10	13	28	28	40/48/24	13	26
Number of Serial I/O ports	2	2	3 or 4	3 or 4	3 or 4	4	4
Number of SPI Channels	1	1	2	2	3/3/2	2	4
Number of I^2C Channels	1	1	1 + RTC	1 + RTC	2/2/1 + RTC	2	2
Number of 1-Wire I/O pins	19	33	45	77	75/115/46	47	96
PWM or Servo Channels	5	5	5	5	6	8	8
Serial Console	✓	✓	✓	✓	✓	✓	✓
USB Console			✓	✓	✓	✓	x
PS2 Keyboard and LCD Console			✓	✓	✓	✓	
USB Keyboard and LCD Console					✓		✓
SD Card Interface			✓	✓	✓	✓	✓
Supports ILI9341 LCD Displays	✓	✓	✓	✓	✓	✓	✓
Supports Ten LCD Panels from 1.44" to 8" (diameter)			✓	✓	✓+ ILI9481	SSD1963 ILI9341_P16 OTM8009A_16 NT35510	✓+ ILI9481
Supports VGA Displays					✓		
Sound Output (WAV/tones)			✓	✓	✓	On-chip DACs	On-chip DACs
Supports PS2 Mouse Input					✓		
Floating Point Precision	Single	Single	Double S/W	Double S/W	Double H/W	Double S/W	Double H/W
Power Requirements	3.3V 30 mA	3.3V 30 mA	3.3V 80 mA	3.3V 80 mA	3.3V 160 mA	3.3V 200mA	3.3V 200mA

Armmite F4 Features

The Armmite F4 is a port of MMBasic to support a specific development board based on STM32F407VGT6 Cortex-M4 32-bit RISC CPU processor which provides nearly all of the services for the user. This includes the flash memory (where the BASIC interpreter is installed),

This image provides an overview of its hardware features:



STM32F407VET6 Data Sheet and Schematic

This manual will describe the resulting features of the development board as a result of this specific MMBasic implementation. The STM32F407 Data Sheet and Schematic should be consulted to clarify any information on the wiring or underlying capabilities of the development board and its processor. They are available on these links.

https://github.com/mcauser/BLACK_F407VE/blob/master/docs/STM32F407VET6_datasheet.pdf

https://github.com/mcauser/BLACK_F407VE/blob/master/docs/STM32F407VET6_schematics.pdf

https://www.thebackshed.com/forum/uploads/panky/2021-02-24_164500_STM32F407VET6_schematic-english-2.pdf (with English translations, courtesy of @panky of TBS forum)

The features of the specific implementation, both hardware and software a summarised here. Many are further detailed within the manual.

STM32F407VGT6 Cortex-M4 32-bit RISC CPU @ 168MHz

This is a 32-bit ARM processor with 1Meg of flash and 196K of RAM. It runs at 168MHz. It includes a dedicated display interface which is used to support 16bit parallel LCD Panels on the FSMC connector. The MMBasic firmware is loaded onto the flash memory of this chip and provides the MMBasic interpreter for the Armmite F4. The Armmite F4 clock is fixed at 168Mhz.

132Kbyte program and 114Kbyte variable space

The Armmite F4 supports MMBasic programs up to 128Kbytes in size, Variable space is 114Kbyte 4K for Saved variables, 128K for Library.

```
> memory
Flash:
   0K ( 0%) Program (0 lines)
 128K (100%) Free

RAM:
   0K ( 0%) 0 Variables
   0K ( 0%) General
 114K (100%) Free

Backup SRAM (4K):
   4K (100%) Free
>
```

MM.DEVICE$

On the Armmite F4 the read only variable MM.DEVICE$ will return " Armmite F407,xGT".

Double Precision Floating Point

All floating point uses double precision calculations. The Armmite F4 uses the single precision floating point capability of the STM32F407VGT6 chip to help with the calculations.

Random Number Generation

The Armmite F4 uses the hardware random number generator in the STM32 series of chips to deliver true random numbers. This means that the RANDOMIZE command is no longer needed and is not supported.

Longstring handling

The Armmite F4 supports a comprehensive set of commands and functions for handling long strings stored in integer arrays

Input Output Pins and Protocols

Forty seven(+20 FSMC) input/output pins with 13 capable of analog input. Built in support for an IR remote control, temperature and humidity sensors. Communications protocols include I2C, 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.

USB Console (the default)

By default MMBasic starts with the USB console enabled.

Four Serial Ports

The four serial ports share pins used for other functions, so may not be available if the other functions are required. COM1 can be dedicated as a serial console to replace the USB console if desired.

Eight PWM Channels

Minimum frequency is 1Hz, maximum is 20MHz. Duty cycle and frequency accuracy will depend on frequency.

PWM 1A,1B,1C

PWM 2A,2B,2C

PWM 3A,3B

Two SPI Channels

The Armmite F4 supports two SPI channels. The second channel operates the same as the first, the only difference is that the commands use the notation SPI2 (for example SPI2 WRITE, etc.).

Note that if the Armmite F4 is configured for a SPI based LCD panel or touch then SPI2 channel is not available for MMBasic. The SPI1 channel is available for use in MMBasic and is prewired on the development board for the on-board W25Q16 Flash chip and the NRF24L01 socket. See Appendix F for an example program to access the W25Q16 flash from MMBasic.

I2C

You can use I2C exactly the same as for the Micromite with the following limitations:
The implementation does not support 10-bit addressing (i.e. options 0 and 1 only).

A second I2C channel can be used using the command *I2C2*.

(note: Armmite F407xGT6 also implements I2C slave mode)

1-Wire Communication

The 1-Wire protocol was developed by Dallas Semiconductor to communicate with chips using a single signalling line. Any pin can be used. See Appendix C for details.

Dual 12-bit DACs

The Armmite F4 has 2 12-bit DACs built into the chip. The analogue levels can be set using the DAC command. In addition, they can be used by the PLAY FLAC and PLAY WAV commands. The pins cannot be used for general purpose I/O. The DACs support an arbitrary function generator capability using the DAC START command.

Three 12-bit ADCs

Analogue to digital conversion can be carried out in 12-bit resolution, 10-bit resolution, and 8-bit resolution depending on the frequency of the conversion. In addition, the ADC can read the battery backup voltage, the chip die temperature and the internal reference voltage. Using the ADC command conversion of three channels can be set to run in the background at up to 500,000 samples per second per channel and one of the channels can be set to provide edge-triggering of the conversion. There are 13 analogue capable pins which can be assigned to these three inputs.

Battery Backed-up Built-in Real time clock (RTC)

The Armmite F4 includes a built-in RTC. A lithium CR1220 coin cell battery on the development board keeps the internal ARM STM32 real time clock running while the power is off and also keeps a bank of 4KB RAM alive at the same time. The real time clock is used to provide the correct time to MMBasic on startup and the battery backed RAM is used to store saved variables and options. The life of this battery life is about 3 to 4 years of normal use. All time and date functions work directly with the RTC and the timing can be trimmed with an OPTION command. The real time clock can be read at millisecond precision. The 32,768Hz crystal for the RTC is also used to discipline the main CPU oscillator ensuring accuracy of commands like TIMER. If you find that the time drifts while the power is off, you can use the OPTION RTC CALIBRATE command to correct for any inaccuracies.

SPI LCD Panel and Touch Connector

The Armmite F4 software supports the ILI9341 and ILI9481 LCD panels with touch supported.

The development board has no dedicated socked for them so individual wiring of the pins or an adaptor boards is required.

16-bit interface to SSD1963, ILI9341, OTM8009A and NT35510 Based LCD Displays

The Armmite F4 drives the SSD1963 and some other 16-bit parallel bus LCD displays. For extra speed the SSD1963 controllers run with a reduced colour range (65 thousand colours RGB565) compared to 16 million

colours with the normal 8-bit interface. SPI touch is supported. The ILI9341 16 bit LCD that is purchased with the STM32F407VET6 development board, while only 320*240 pixels, plugs directly into the board via the 32 pin FSMC connector. All other LCD panels will require an adaptor board.

PS2 Keyboard Connection

A PS2 keyboard can be connected using the PS2 KB_CLK and KB_DATA lines on pins PD3 and PA15 respectively. The software supports the PS2 keyboard but there is no actual PS2 connector. See later in this document for more details.

Audio Output

The Armmite F4 has no audio socket connect to the board as supplied, however the audio appears on the DAC pins PA4 and PA5. See the later section Audio Output for information on connecting these to an amplifier.

MMBasic can generate audio in several formats ranging from simple sine wave tones through to playing FLAC and WAV audio files. (MP3 is not supported because of high processor resources required to decode)

The output is high impedance suitable for feeding into an amplifier. ***It cannot directly drive a loudspeaker, headphones or any low impedance load and might be damaged if that was attempted.***

Extended WAV File Playback

The Armmite F4 can play WAV files (like the Micromite Plus) however, it is also capable of playing WAV files recorded with sampling rates of 8 KHz,16KHz and 44.1KHz.

Temperature Sensor

The Dallas DS18B20 temperature sensor can be used to measure temperature. Support for the DS18B20 is built into MMBasic – see the section Special Device Support in this manual for the details. Any pin can be used.

External I/O connectors

The pins on the development board have been allocated to various MMBasic functions as outlined in the table in section Pin and Connector Capabilities.

W25Q16 Flash

STM32F407VGT6 development board has a 16Mbit (ie 2MByte of 8 bits) windbond flash chip built in. The can be accessed from with an MMBasic program via SPI commands.

It is connected to SPI1 and its chip select pin is PB0. (35) Appendix E has an example program for formatting and accessing it. (Not all boards have the windbond flash)

RST Key

Used to reset the Armmite F4 and start the bootup sequence as if the power had been cycled.

Key 0 Switch

User available key. Has special function to enable Serial Console if held while power is being applied. Connects ground to PE4 pin when pressed.

Key 1 Switch

User available key. Has special function to reset MMBasic if held while power is being applied. Connects ground to PE3 pin when pressed.

Key_UP Switch

Connects to pin PA0 via a 10K resistor.

LED D2 and D3

Two LEDs wired via 510 ohm resistors to 3.3v and are connect to IO pins PA6 and PA7 for immediate use to get started on flashing a LED.

Power LED

The power led D1 (green or red) is illuminated whenever power is applied.

WS2812 support

The Armmite F4 supports the WS2812 Led driver. This chip needs very specific timing to work properly and by incorporating support in the Armmite F4 firmware the user can program these chips with minimum effort. The command WS2812 is used to set the colours of the LEDs. There is no limit to the size of the WS2812 string supported.

GPS support

The Armmite F4 support connection of a GPS to any of the 4 serial interfaces. The command

OPEN "COMn:baudrate" as GPS is used to enable reception of NMEA GPS messages. The GPS() functions can then be used to interrogate the GPS data which is automatically parsed in the Armmite firmware.

In addition, the **PRINT #GPS,string$** command allows sending a formatted frame to the GPS in an NMEA message format, it is used to configure the GPS. For example: you can change the speed of the serial port, or select certain frames, etc. The frame sent by user, must not end with the CRC (checksum) because MMBasic automatically adds this CRC at the end of message.

OPTION VCC *voltage*

Option VCC voltage is used to tell the ADC in the Armmite F4 the value of the the VCC voltage. It is used during analogue readings as the value for the external reference. It defaults to 3.3V if not set. See the section Setting Option VCC for details on setting it using the refernce voltage embedded into each individual cip.

CPU SLEEP commands

The Armmite F4 CPU sleep command as follows:

CPU SLEEP

The wakeup pin is PA0, however any other COUNT pin (PE1, PE3, PE4 and PA8) can also be used to wake the processor if enabled with SETPIN pinno, CIN or PIN or FIN.

CPU SLEEP time

The Armmite uses the RTC to generate an interrupt to wake the processor after a period of sleep. Any period can be specified including fractions of seconds and because the RTC is used the timing will be accurate. Using the embedded ARMMITE F4 date and time functions makes it easy to sleep until any particular time. e.g.

Midnight_tonight% = epoch(date$+" 00:00:00")+86400 'epoch at start of day today + secs in a day

CPU SLEEP Midnight_tonight% - epoch(now) ' sleep until midnight tonight.

The R21 pullup resistor on the USB D+ data line prevents the CPU SLEEP [n] working when using a USB console. Removing R21 will allow this to work with the USB console and has no other detrimental effects.

Unsupported commands

If you are familiar with the Micromites then this list will save you looking for things that are not there.

- Changing CPU Speed, OPTION CPU SPEED is not available.

Loading the MMBasic Firmware

Once you have the development board you need to load the MMBasic firmware. This only needs to be done once unless you need to load an updated version. There are now three versions of the firmware available.

ArmmiteF407VGT6.bin	Supports boards based on the 100 pin STM32F407VGT6, the 144 pin STM32F407ZGT6 and the 64 pin STM32F405RGT chips. MMBasic Reset and switch to Serial Console as detailed in the board descriptions below.
ArmmiteF407VGT6PC13Reset.bin	As for the version above except PC13 (GND or 3.3v) is used to control MMBasic Reset and switch to Serial Console on the 100pin and 144 pin chiops. This allows PE3 and PE4 to be used as inputs without risking unintended MMBasic Resets or switch to Serial Console.
ArmmiteF407VGT6Feather.bin	Support the Adafruit Feather using 64 pin STM32F405RGT chip.

The latest ArmmiteF407VGT firmware and discussion is available in this post on The Back Shed (TBS) forum. You will need to scroll through the thread and selected the latest version. Also available from the binaries directory at https://github.com/disco4now/ArmmiteF407VGT.

You will place the development board in Boot Loader mode by setting jumpers for BT0 and BT1 pins, connect it to your computer via a USB cable and use the free STM32CubeProgrammer application to load the firmware.

Appendix G at the end of this document gives a very detailed description of loading the firmware as well as how to obtain the free STM32CubeProgrammer.

If you have not done so, you should go to Appendix H – Loading the Firmware now.

When you complete the steps there you should have the MMBasic command prompt and are ready to go!

Program Memory cleared when firmware updated

The program memory is cleared when the firmware is updated.

Options are reset when firmware updated

The Options are stored within the flash used by the firmware so are reset to default when new firmware is loaded.

Saved variables are cleared when firmware updated

The saved variables are within the flash used by the firmware so are deleted when new firmware is loaded.

They are not stored in battery backed-up ram as for the ArmmiteF407.

Library is cleared when firmware updated

Any code in the Library is cleared when the firmware is updated.

If you have problems connecting, try these two procedures. Resetting MMBasic will recover from any abnormal/unknown state by clearing the Program Memory, reseting Options to default and clearing save variables.

Trouble Shooting USB Console also has some useful tips.

Power and Console Connections

USB Console (the default)

In the Armmite F4 all programming is done through the console. At the console you can enter commands, edit programs, run programs and observe the output of your program – including error messages! This is by default pointed to the USB connector on STM32F407VET6 development board. There is nothing that you need to do on the Armmite F4 to use the USB console. Just plug the USB cable from the Armmite F4 into your host computer and MMBasic will automatically create a virtual serial port over USB so that you can communicate with it from a Windows, Linux or Macintosh computer using nothing more than the USB port.

The Armmite F4 has three options for the console input/outputs. These are the default USB console, a serial console on COM1 or optional PS2 keyboard and LCD display. The PS2 keyboard and LCD display if enabled as a console operate in parallel with the currently configured console, anything received from any of the inputs is placed in the input queue for the interpreter or your program to read and anything outputted by your program or the interpreter will be sent to all devices (if they are connected).

The communications protocol used is the CDC (Communication Device Class) protocol and there is native support for this in Windows 10, Linux (the cdc-acm driver) and Apple OS/X. Macintosh users can refer to the document "Using Serial Over USB on the Macintosh" on https://geoffg.net/OriginalColourMaximite.html.

You can then use a terminal emulator such as Tera Term to connect to this communications port and it will work the same as if you were using a hardwired serial console. In Tera Term you do not have to specify a baud rate because the USB connection will run as fast as it can.

Be aware however that the USB connection will be reset if the Armmite F4 is reset and there are many things that can do this including the watchdog timer, the command CPU RESTART and so on.

Tera Term on Windows 10 now seems to be able to automatically reconnect the console after a reset so you many not suffer the same frustration as past users.

If the loss of the USB console during development becomes an issue, the console can be switch to the serial console and be accessed via a USB to serial bridge connected to the serial console pins on the Armmite F4 board.

Another aspect to be aware of is that you should not use the CPU SLEEP command while a USB session is active. The results will be undefined but could possibly cause the Armmite F4 to crash and reboot.

On a Windows computer the Armmite F4 will appear as an additional serial port in Device Manager as illustrated on the right.

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

The terminal emulator that you use should support VT100 emulation as that is what the editor built into the MMBasic expects. For Windows users it is recommended that you use Tera Term as this has a good VT100 emulator and is known to work with the XModem protocol which you can use to transfer programs to and from the Armmite F4 (Tera Term can be downloaded from: http://tera-term.en.lo4d.com/). See the section VT100 Terminal Emulators for other options which will also be suitable.

Windows USB connection

On Windows 10 the driver for the USB console is included with Windows 10. On Win7 and earlier the USB console requires STMicroelectronics Virtual COM Port drivers to be installed. (This driver is separate from the drivers installed with STM32 Cube Programmer software). Virtual COM Port drivers at www.st.com/en/development-tools/stsw-stm32102.html

Apple Macintosh USB connection

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

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

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

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

By default, the function keys will not be correctly defined for use in the Armmites 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.

Linux USB connection

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

Power Requirements

The USB connector is for power and the serial console over USB. The power requirement of the Armmite F4 is 5V at 70mA (no LCD) up to 250mA (typical). This is within the capabilities of most USB chargers however some PCs (especially older laptops) may have trouble supplying this. If your Armmite F4 is suffering from intermittent issues such as reboots, errors reading the SD card, etc. then it would be worth changing the power supply to one a with a much higher capacity (for example, 2 amps or more).

The Armmite F4 software requests the host to provide 500mA on the 5V pin of the USB connector. Where the host supports this this should be enough to supply the Armmite F4 and most LCD panels. When one of the larger SSD1963 panels that require a 5V connection, an alternate method of supplying 5V power may need to be considered. See SSD1963 Power Considerations for more detail.

Powering from external 5V source

Many devices that have a choice of power via USB or a separate 5v supply have a jumper to selected which option is used. The STM32F407VET6 does not. The USB 5v and the 5v pins on the board are connected together by 0 ohm resistor R25. This means if you power via an external 5v supply connected to the 5v pin, then this 5v will appear on the USB connector as well. This is not a problem if you are not connecting to the USB console at the same time. In most cases even if you connect to the USB console at the same time there is no issue, especially if you are connecting to both from the same computer/laptop. e.g. You are experimenting with use of USB and Serial Console connected by a USB serial adapter.

If you want to power the board permanently from a 5V supply and want to remove any risk when connecting via a USB console you can remove R25, or even replace with a schottky diode so that the board can still be powered through the USB if required.

Switching to Serial Console (via Option Command)

By default, the Console is directed to the USB connector. It can be redirected to the serial port on the board by either of these two methods.

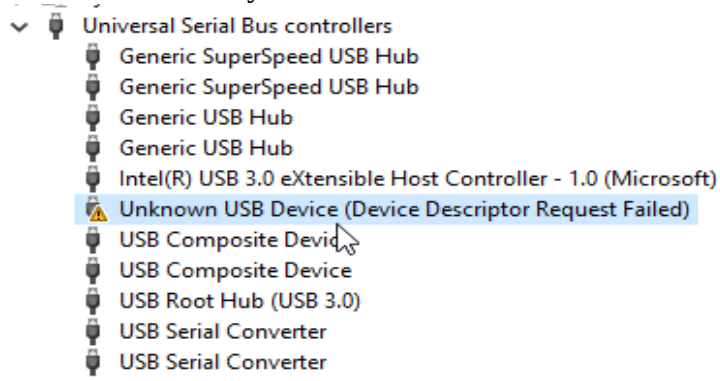
If a working console on the USB is available connect and enter this command.

OPTION SERIAL CONSOLE ON

This routes the console to the serial connection COM1 on J6 of the board (near the SDCARD socket). The pinout on this is compatible with many of the CP2102 USB/UART PCBs which can be plugged directly onto the J6 header. In this mode the USB connection is completely disabled but you can still use the USB connection to power the board. The board needs to be restarted. You need a USB to serial module to connect from the PC. The terminal emulator and the serial port that it is used should be set to the Armmite F4 default of 115200 bauds 8 data bits and one stop bit. Using the OPTION BAUDRATE command the baud rate of the console can be changed to any speed up to 921600 bps. Changing the console baud rate to a higher speed makes the full screen editor much faster in redrawing the screen. If you have a reliable connection to the Armmite it is worth changing the speed to at least 115200.

Once changed the console baud rate will be permanently remembered unless another OPTION BAUDRATE command is used to change it. Using this command, it is possible to accidently set the baud rate to an invalid speed and in that case the only recovery is to reset MMBasic as described below in OPTION RESET.

When the console is redirected to COM1 and a USB cable is plugged in the USB connector it will fail to initialise correctly. This indicates the console is not on the USB connector.



Switching to Serial Console (via Key 0 at Restart)

If the USB console is not available for use or not working, then use this method.

Hold Key 0 on the development while restarting the board by connecting power or pressing the RST button.

The system will start up with the OPTION SERIAL CONSOLE ON already selected. Connect to the J6 serial port as above.

Restoring USB Console (via Option Command)

To return to the USB console issue the following command from the serial console:

OPTION SERIAL CONSOLE OFF

will redirect to the USB console.

The USB cable will need to be removed and reinserted if it is in place, as the host computer will have marked it faulty as it previously would not have initiated properly as a serial port.

After this the J6 header supports COM1 communications.

OPTION RESET

Issuing this command also return to the default state with the console directed to the USB connection. It will also reset all other options to their default values. This for the Armmite F4 means enabling the default ILI9341_P16 LCDPANEL and TOUCH.

The USB cable will need to be removed and reinserted if it is in place, as the host computer will have marked it faulty as it previously would not have initiated properly as a serial port.

Restoring USB Console (via Key 1 at Restart)

If the serial console is not available, or its speed is unknown a full reset of MMBasic will restore the USB Console.

Holding Key 1 on the development while restarting the board by connecting power or pressing the RST button will reset MMBasic. ***This will clear all program and variable memory and clear all options***.

It will also restore the USB console as the default.

The USB cable will need to be removed and reinserted if it is in place, as the host computer will have marked it faulty as it previously would not have initiated properly as a serial port.

Armmite F4 interaction with USB console

The Armmite firmware controls the USB connection as follows:
On power up, if no USB connection is plugged in (separate 5V supply) console output is discarded.
On power up, if a USB connection is plugged in console output will be buffered until a terminal emulator is connected.
Once running, if the USB connection is removed (separate 5V power) console output is discarded.

Once running, if the USB connection is re-inserted, console output will be restored from the point at which the USB was re-connected.

Using Serial Console via a USB – Serial Converter

It is unlikely your modern computer will have an actual serial port. The serial port is achieved using cheap and popular USB to Serial converters.

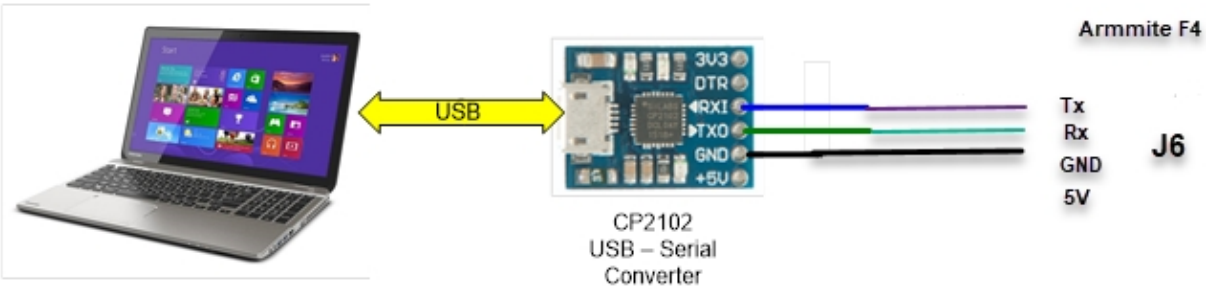
The serial console when enabled defaults to 115200 bauds, which uses TTL signal levels. This is similar to the RS232 interface on older personal computers but the TTL signal level is inverted and swings from zero to 3.3V.

There are many USB to serial converters on the market. These provide a TTL level serial interface on one side and a USB interface on the other. When connected to your computer the converter will appear as a virtual serial port. Recommended are converters based on the Silicon Labs CP2102 chip, they can be found on eBay for a few dollars (search for "CP2102") and work perfectly with the Micromite and Armmite F4. CH340 USB/serial adaptors also work well. You should avoid converters based on the FTDI FT232RL chip as many Chinese manufacturers use non genuine chips which will not work with the current Windows drivers.

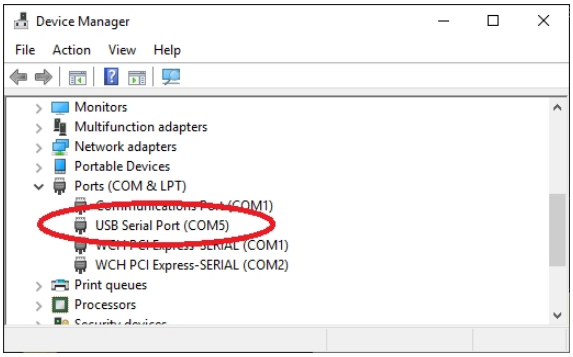
The serial interface side of the converter will generally have a ground pin and a 5v power output pin and this can be used to power the Armmite. The converter will also have two pins marked TX (or similar) for transmit and RX (or similar) for receive. The TX pin of the serial converter must go to the RX pin of the Armmite and the RX pin must go to the TX pin.

If you have a serial converter that operates at 5V you can still use it with the Armmite F4. All you need do is place a 1K resistor in series with the transmit signal from the converter. The resistor will limit the current to a safe level.

Below is a typical connection using the CP2102 converter. Note that the 3.3V output from the converter can be as high as 4.3V so it would be best to connect the 5v output to the 5v connector on the Armmite and let it convert to the correct voltage.



When you plug the USB side of the converter into your computer you may have to load a driver to make it work with the operating system. 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 on the right.



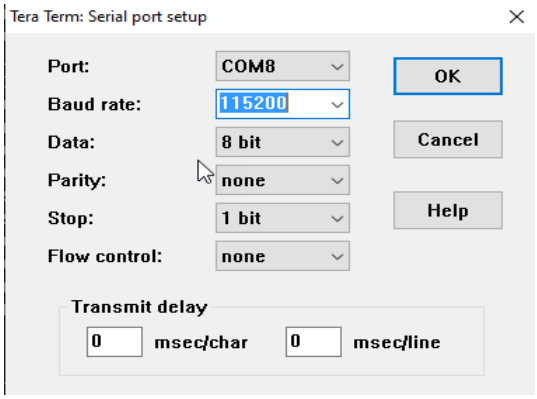
VT100 Terminal Emulators

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

to the remote computer over the serial link.

The terminal emulator that you use should support VT100 emulation as that is what the editor built into the Armmite expects. For Windows users it is recommended that you use Tera Term as this has a good VT100 emulator and is known to work with the XModem protocol which you can use to transfer programs to and from the Armmite F4 (Tera Term can be downloaded from: http://tera-term.en.lo4d.com/).

The terminal emulator and the serial port that it is using should be set to the Armmite F4 standard of 115200 bauds, 8 data bits and one stop bit. 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 USB to TTL serial converter was plugged into.



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.

Other terminals are MMEdit and GFXTerm and Putty.

MMEdit supports a VT100 emulation as well as an Ascii terminal and also allows the editing of programs offline. MMEdit was written by Jim Hiley and can be downloaded for free from https://www.c-com.com.au/MMedit.htm.

GFXterm is a simple terminal emulator for use with Micromite/Armmite computers running MMbasic. It provides just enough VT100/ANSI emulation to use the inbuilt editor with an 80 column by 24 line screen size. Both (local) serial and (remote) network connections are allowed. In addition, GFXterm supports a simple set of graphics extensions that are suitable for drawing very basic rolling graphs. Mouse scroll wheel activity is mapped to the cursor up/down keys, and will work with the internal editor. Linux and Windows versions available at this link.

https://github.com/robert-rozee/GFXterm/tree/main/binaries

Wireless Console using ESP-01 ESP8266

This thread on TBS details setting up an ESP8266 connected to the COM1 port to give wireless access to the console. The ESP-01 version is not very expensive and wireless connection can be very convenient.

http://www.thebackshed.com/forum/ViewTopic.php?TID=8440&P=1

Troubleshooting USB Console

If you cannot see the startup banner check the following:

- Reset the Armmite by retarting with KEY1 down to ensure it has the default options set and is using the USB console.
- Ensure the device is not in boot loader mode. i.e. BT0 tied to 3.3v
- Check that a new serial port appears under Device Manager on windows when the Armmite is connected.
- Check your terminal program is using that serial port when trying to connect.
- Try using TeraTerm, it is very robust.
- If you are using another terminal and have been previously using TeraTerm be aware that TeraTerm reconnects automatically and will steal the connection before you can connect with the other terminal e.g. MMEdit.
- On Windows 10 the driver for the USB console is included with Windows 10. On Win7 and earlier the USB console requires STMicroelectronics Virtual COM Port drivers to be installed. (This driver is separate from the drivers installed with STM32 Cube Programmer software). Virtual COM Port drivers at www.st.com/en/development-tools/stsw-stm32102.html

Troubleshooting Serial Console

If you cannot see the startup banner try the following:

- ensure the Serial Console is enabled be restarting with KEY0 down.
- Try disconnecting the USB-serial converter and join its TX and RX pins. Then try typing something into the terminal emulator. You should see your characters echoed back, if not it indicates a fault with the converter or the terminal emulator.
- If the USB-serial converter checks out the fault could be related to the console connection to the Armmite. Make sure that TX connects to RX and vice versa and that the baud rate is 115200.
- If you have an oscilloscope you should be able to see a burst of activity on the Armmites TX line on power up. This is the Armmite sending its startup banner.

Resetting MMBasic

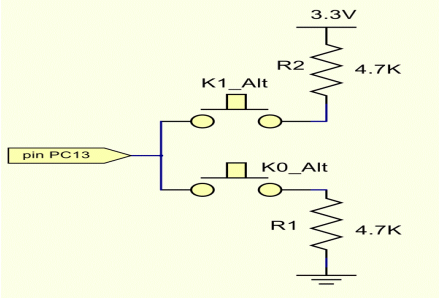
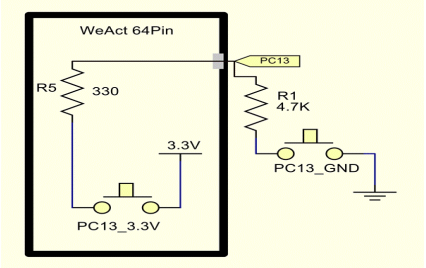
MMBasic can be reset to its original configuration using the following methods:

This will result in the program memory and saved variables being completely erased and all options (security PIN, console baud rate, etc.) will be reset to their initial defaults. This includes setting the console to the USB.

The method of triggering the MMBasic Reset varies slightly between the supported boards. At a power restart will test a nominated pin for the MMBasic Reset condition. During a Power restart it will also test for a request to switch to the Serial Console.

On the 100 Pin and 144 Pin boards the pins tested are as below for the two variations of firmware. The WeAct 64 pin board will test pin PC13

- Holding KEY 1 down, while applying power. You can connect GND to PE3 pin if you find it difficult to hold the small button down.
- Holding KEY 1_ALT down in lieu of K1 where KEY 1_Alt applies GND to PC13. The alternate version of the firmware is required so it tests PC13 in lieu of PE3. K1_ALT and K0_ALT are additions outside the standard board. This allows PE3 and PE4 to be used with external connections without the risk of unintended MMBasic Resets or switching to Serial Console when the board has a power restart of a reset.

STM32F407VGT6.bin	STM32F407VGTPC13reset.bin
144 and 100 Pin Boards. On board K1 places GND on PE3. On board K0 places GND on PE4 PE3 and PE4 tested at power up or reset.	 PC13 tested at power up only.
WeAct 64 pin board PC13 is tested for both versions of the firmware. 3.3v on PC13 for MMBasic Reset. (Use on board C13 key) GND on PC13 for Serial Console. (User added external key or jumper.)	 PC13 tested at power up only.

The startup banner after an MMBasic Reset will initially show a message to confirm the reset was initiated. If the PC13 Reset version of the firmware is loaded this is also indicated in the startup banner.

```
> ARMmite xGT6 MMBasic Version 5.07.02b0 (RGT6 64 pins), 400, 1024
Copyright 2011-2025 Geoff Graham
Copyright 2016-2025 Peter Mather

!!! MMBasic Reset !!!
```

Pin and Connector Capabilities

STM32F407VGT6 100 Pin function and connector positions

The capabilities and allocation of each pin are detailed in this table. MMBasic can address the pins via their pin number or connector name. This manual will only use the connector name but the first two columns of the table below show their correlation.

STM32F407VGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
1	PE02	J2-11	IR			DIN - DOUT		
2	PE03	J2-12	Count 2			DIN - DOUT		KEY1/INT2
3	PE04	J2-13	Count 3			DIN - DOUT		KEY0/INT3
4	PE05	J2-14	PWM-3A			DIN - DOUT		TIM9_CH1
5	PE06	J2-15	PWM-3B			DIN - DOUT		TIM9_CH2
6	VBAT							
7	PC13	J2-16				DIN – DOUT 3ma (see notes below)		
8	PC14		OSC32_IN					
9	PC15		OSC32_OUT					
10	GND			GND	1,30 32			
11	3.3V			3.3V	31			
12	OSC_IN		8MHz					
13	OSC_OUT		8MHz					
14	RST	JTAG-15	NRST	RST	2			15 JTAG-RST
15	PC00	J2-17	15			DIN - DOUT - AIN	ADC_10 [A]	
16	PC01	J2-18	16			DIN - DOUT - AIN	ADC_11 [B]	
17	PC02	J2-19	17			DIN - DOUT - AIN	ADC_12 [C]	
18	PC03	J2-20	18			DIN - DOUT - AIN	ADC_13 [A]	
19	VDD							
20	VREF-	J2-21						
21	VREF+	J2-22						
22	VDDA							
23	PA00	J2-23	COM3-TX			DIN - DOUT - AIN	ADC_0 [A]	KEY_UP WK_UP
24	PA01	J2-24	COM3-RX			DIN - DOUT - AIN	ADC_1 [A]	
25	PA02	J2-25	COM4-TX			DIN - DOUT - AIN	ADC_2 [A]	
26	PA03	J2-26	COM4-RX			DIN - DOUT - AIN	ADC_3 [A]	
27	GND							
28	3.3V							
29	PA04	J2-27	DAC-1				DAC-1 (3.3v)	
30	PA05	J2-28	DAC-2				DAC-2 (3.3v)	
31	PA06	J2-29	PWM-1A			DIN - DOUT - AIN	ADC_6 [A]	LED D2
32	PA07	J2-30	PWM-1B			DIN - DOUT - AIN	ADC_7 [A]	LED D3
33	PC04	J2-31	33			DIN - DOUT - AIN	ADC_14 [B]	
34	PC05	J2-32	T_IRQ	PEN-IRQ	27	DIN - DOUT - AIN	ADC_15 [B]	
35	PB00	J2-33	PWM-1C			DIN - DOUT - AIN	ADC_8 [A]	F_CS
36	PB01	J2-34	LCD_BL	BL	28			TIM3_CH4
37	PB02	J3-6	BOOT1			DIN - DOUT		
38	PE07	J2-35	FSMC_D4	D4	14	DIN - DOUT		

STM32F407VGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
39	PE08	J2-36	FSMC_D5	D5	13	DIN - DOUT		
40	PE09	J2-37	FSMC_D6	D6	12	DIN - DOUT		
41	PE10	J2-38	FSMC_D7	D7	11	DIN - DOUT		
42	PE11	J2-39	FSMC_D8	D8	10	DIN - DOUT		
43	PE12	J2-40	FSMC_D9	D9	9	DIN - DOUT		
44	PE13	J2-41	FSMC_D10	D10	8	DIN - DOUT		
45	PE14	J2-42	FSMC_D11	D11	7	DIN - DOUT		
46	PE15	J2-43	FSMC_D12	D12	6	DIN - DOUT		
47	PB10	J2-44	I2C2-SCL			DIN - DOUT		
48	PB11	J2-45	I2C2-SDA			DIN - DOUT		
49	VCAP1		VCAP					
50	3.3V		VDD					
51	PB12	J2-46	T_CS	T_CS	24	DIN - DOUT		
52	PB13	J2-47	SPI2-CLK	T_CLK	23	DIN - DOUT		
53	PB14	J2-48	SPI2-IN	T_MISO	26	DIN - DOUT		
54	PB15	J3-48	SPI2-OUT	T_MOSI	25	DIN - DOUT		
55	PD08	J3-47	FSMC_D13	D13	5	DIN - DOUT		
56	PD09	J3-46	FSMC_D14	D14	4	DIN - DOUT		
57	PD10	J3-45	FSMC_D15	D15	3	DIN - DOUT		
58	PD11	J3-44	VBUS_FS			DIN - DOUT		
59	PD12	J3-43	PWM-2A			DIN - DOUT		TIM4_CH1
60	PD13	J3-42	FSMC_A18	DC	21	DIN - DOUT		
61	PD14	J3-41	FSMC_D0	D0	18	DIN - DOUT		
62	PD15	J3-40	FSMC_D1	D1	17	DIN - DOUT		
63	PC06	J3-39	COM2-TX			DIN - DOUT		
64	PC07	J3-38	COM2-RX			DIN - DOUT		
65	PC08	J3-37	SDIO_D0					
66	PC09	J3-36	SDIO_D1					
67	PA08	J3-35	Count 4			DIN - DOUT		INT4
68	PA09	J3-34	COM1-TX			DIN - DOUT		J6-TXD
69	PA10	J3-33	COM1-RX			DIN - DOUT		J6-RXD
70	PA11	J3-32	USB-DM					USB D-
71	PA12	J3-31	USB-DP					USB D+
72	PA13	JTAG-7	SWDIO			DIN - DOUT		7 JTAG-TMS
73	VCAP2							
74	GND							
75	3.3V							
76	PA14	JTAG-9	SWCLK			DIN - DOUT		9 JTAG-TCK
77	PA15	J3-30	KBD_CLK			DIN - DOUT		5 JTAG-TDI
78	PC10	J3-29	SDIO_D2			Not available		
79	PC11	J3-28	SDIO_D3			Not available		
80	PC12	J3-27	SDIO_CK			Not available		
81	PD00	J3-26	FSMC_D2	D2	16	DIN - DOUT		CAN Rx
82	PD01	J3-25	FSMC_D3	D3	15	DIN - DOUT		CAN Tx
83	PD02	J3-24	SDIO_CMD					
84	PD03	J3-23	KBD_DATA			DIN - DOUT		

STM32F407VGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
85	PD04	J3-22	FSMC_NOE	RD	19	DIN - DOUT		
86	PD05	J3-21	FSMC_NWE	WR	20	DIN - DOUT		
87	PD06	J3-20	87			DIN - DOUT		
88	PD07	J3-19	FSMC_NE1	CS	22	DIN - DOUT		
89	PB03	J3-18	SPI_CLK			DIN - DOUT		JP2-5 NRF-SCK
90	PB04	JTAG-3	SPI_IN			DIN - DOUT		JP2-7 NRF-MISO
91	PB05	J3-17	SPI-OUT			DIN - DOUT		JP2-6 NRF-MOSI
92	PB06	J3-16	I2C-SCL			DIN - DOUT		JP2-3 NRF_CE
93	PB07	J3-15	I2C-SDA			DIN - DOUT		JP2-4 NRF_CS
94	BOOT0	J3-5	BOOT0					
95	PB08	J3-14	PWM-2B			DIN - DOUT		JP2-8 NRF-IRQ
96	PB09	J3-13	PWM-2C			DIN - DOUT		TIM4_CH4
97	PE00	J3-12	97			DIN - DOUT		
98	PE01	J3-11	COUNT 1			DIN - DOUT		INT1
99	GND							
100	3.3V							

STM32F407VGT6 Explanation of keys used in above table

The following summary includes information based on the authors interpretation of the STM32F407VET6 data sheet and the schematic of the development board. These should be consulted for more detailed information.

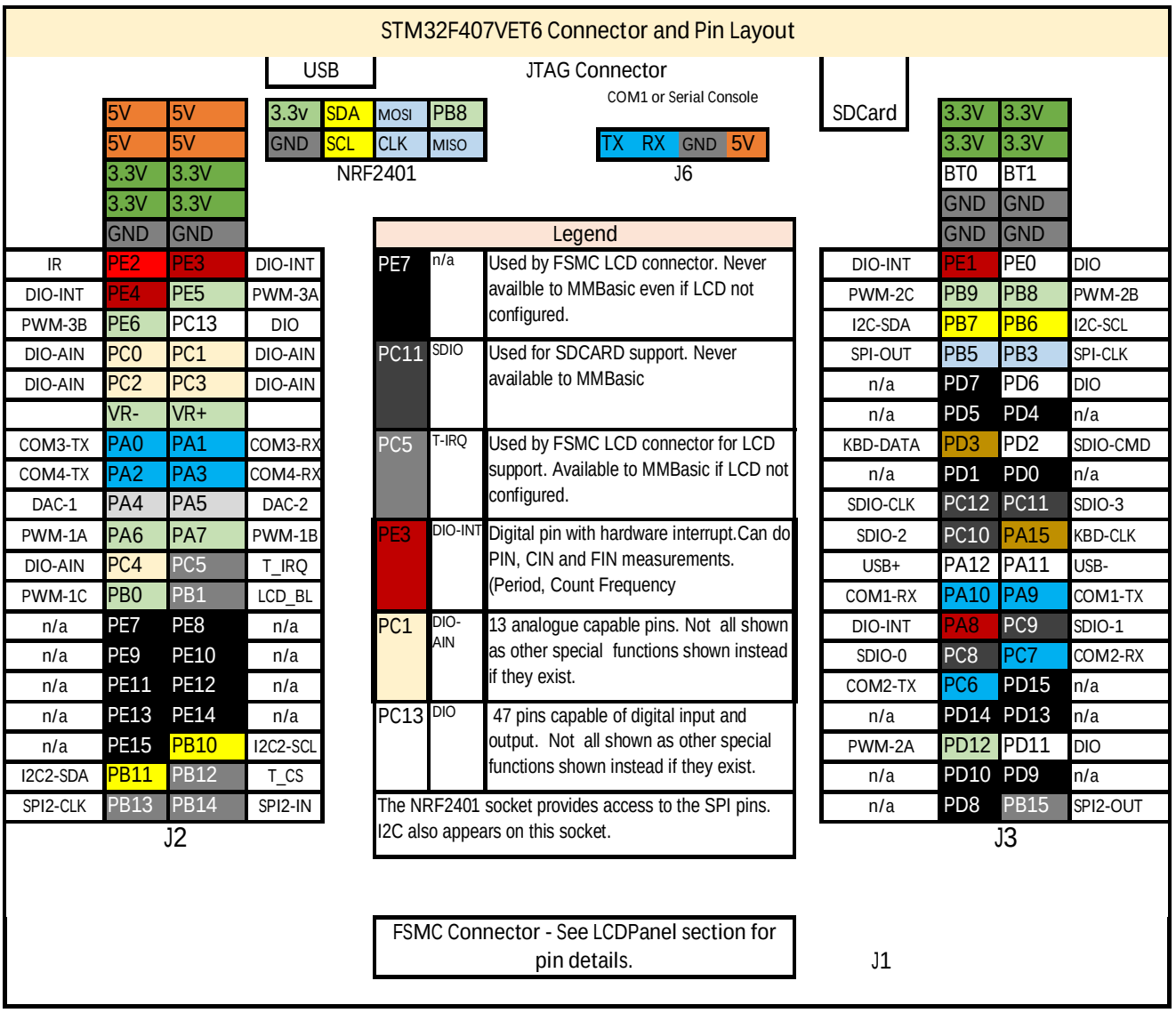
An explanation of some of the codes used in the above table is also provided.

https://github.com/mcauser/BLACK_F407VE/blob/master/docs/STM32F407VET6_datasheet.pdf
https://github.com/mcauser/BLACK_F407VE/blob/master/docs/STM32F407VET6_schematics.pdf

Code or Item	Details
DIN -DOUT	These pins can be used as digital output and input. They are 5V tolerant and can sink or source a maximum of 25mA.
DIN-DOUT (3ma) PC13	This pin, PC13 is suppied from VBAT and can only deliver 3mA. It cannot be used to drive a LED. It has also in some usage situations been seen to interfere with the reliable operation of the SDCARD. (It is next to PC12 the SDOI-CLK). As it is supplied by VBAT it may have an effect on the life of the backup battery when the board is powered off. It some case it may be better to avoid this pin if possible.
DIN – DOUT -AIN	These pins are analogue capable. i.e. can be used to read voltage. They can be used as digital output and input. They are 5V tolerant and can sink or source a maximum of 25mA ***EXCEPT*** when in the AIN analogue mode, as they are then connected to the 3.3v ADC and must not exceed 3.3v The total current sunk or sourced for all pins combined cannot exceed 150mA in total.
DAC x (3.3v)	DAC1 and DAC2 are not 5v tolerant. 3.3v only.
PULL-UP	Weak pull-ups to 3.3v are typically 40K ohms for all pins except for PA10 and PA12

PULL-DOWN	which are 11K. Weak pull-downs to GND are typically 40K ohms for all pins except for PA10 and PA12 which are 11K Pull-up and pull-down resistors are designed with a true resistance in series with a switchable PMOS/NMOS. This MOS/NMOS contribution to the series resistance is minimum (~10% order).
INTx COUNTx	These 4 pins PE1, PE3, PE4 and PE8 have hardware interrupts and can be used with the SETPIN CIN, PIN and FIN options for count, period and frequency measurements.
WK_UP	This is the wakeup pin. It can be used to wake the CPU after a CPU SLEEP command. Any of the 4 count pins will also wake the CPU if they are configured.
ADC_x [A] ADC_x [B] ADC_x [C]	These are 13 analogue capable pins that can be connected to the ADC. The [A], [B] or [C] indicates which of the three input on the ADC they connect to. This is important when using the ADC command, as three input channels must have an appropriate A, B or C type pin. The ADC command in this manual details which pins can be used for each input.
I2C Pullups	Neither of the data line (SDA) or clock (SCL) for either of the I2C ports have pullup resistors (to 3.3V) installed. These may need to be installed if not already on the peripheral being use. The I2C OPEN command does enable weak pullups. I2C CLOSE will disable them.
External Components	A number of pins have some external components attached to them on the development board. This needs to be considered if you want to use those pins. PA14 SWCLK on the JTAG-9 pin can be used but has a 10K pullup PA13 SWDIO on the JTAG-7 pin can be used but has a 10k pullup PB2 BOOT1 has 10K pulldown to ground.
67 Digital Pins	All digital pins can be used for digital I/O using the PIN() function and command and use the pin name as the reference. For example, pin PE4 can be set to an output using SETPIN PE4, DOUT and then the pin set high (i.e., to 3.3V) using the command PIN(PE4) =1. The pins assigned to the FSMC are available if a no parallel display is configured.
KEY0	Key 0 or PE4 can be used to enable the Serial Console. If the pin is connected to ground or KEY 0 held down on power up or reset, then the Serial Console it enabled at start up. All other options are reset to their default values and any program erased.
KEY1	Key 1 or PE3 can be used to completely reset the Armmite F4 to its "factory default" condition. If the pin is connected to ground or KEY 1 held down on power up all options will be reset to their defaults and any program in flash memory erased. Note that external circuitry connected to this pin (e.g., a capacitor) must not look like a short circuit at power up as this might trigger a reset.
Key RST	This resets the CPU. Has the same effect as disconnecting and reconnecting power. During the restart the state of KEY 0 and KEY 1 are tested to see if any special action is required, otherwise MMBasic is started. If OPTION AUTORUN ON is set any program in flash is also automatically run.
K_UP	K_UP connects GND to the PA0 pin when pressed.
NRF2401 Socket	SPI and I2C both appear at this socket. It is a convenient place to connect to them.

STM32F407VGT6 Connector and Pin Layout



STM32F407VGT6 Board Versions

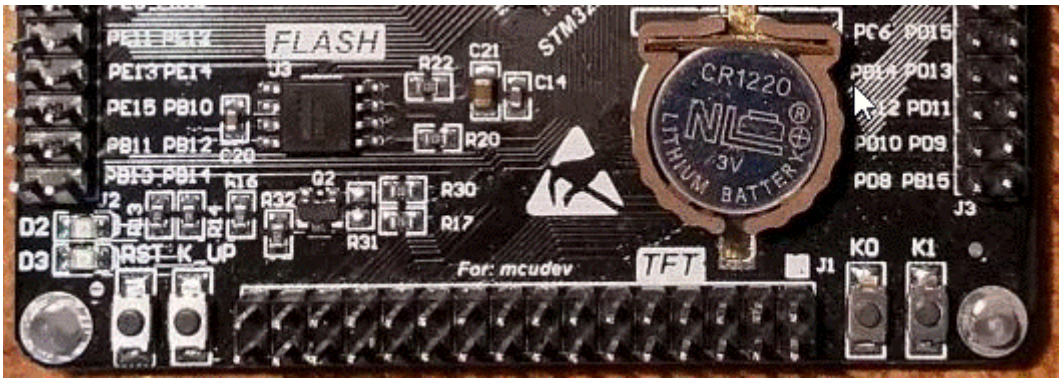
There appears to be a number of different variations of the STM32_F4VE board. The know differences are summarised below. The will all work with the latest version of the software without modification. The versions are on the silkscreen on the back of the board.

Version	Details
V2.0	There are some reports this board needs to have the BT1 jumper in place when programing. The circuit indicates it should be to GND without needing the jumper, but it may be required.
V2.1 V2.3	These seem to match the schematic. Pullup resistors as indicated on the schematic. SMD resistors are descrete components.
V33	This is the latest version being supplied. It does not have pullups on the SDCARD pins as shown on the schematic and as supplied on the earlier version. The MMbasic firmware now supports the SDCARD without the need for these pullups. The SMD components are smaller and not all resistors are descrete components
V36	This version has a label New-TFT next to the FSMC connector and some pins are different. Take care these as the 3.3V and GND are rearranged to match a new TFT display pinout. See below.

STM32F407VGT6 Board FSMC connectors.

There are two slightly different versions of the pin layout of the FSMC connector. The later (different) boards can usually be identified as they have New-TFT printer next to the FMSC connector. There are matching LCD panels for each.

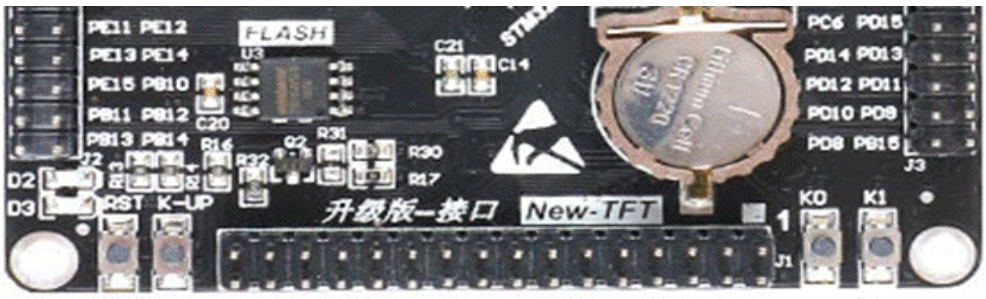
The original FSMC connector.



FSMC LCD Connector – Top View (Old – TFT0

31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1
3.3V	n/c	Pen-IRQ	MOSI	T-CLK	DC	RD	B1	B3	B5	B7	B9	B11	B13	B15	GND
GND	GND	LCD-BL	MISO	T-CS	CS	WR	B0	B2	B4	B6	B8	B10	B12	B14	RST
32	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2

The later different FSMC connector.



FSMC LCD Connector – Top View (New-TFT)

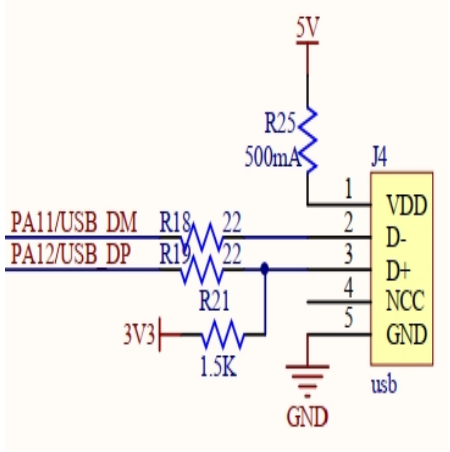
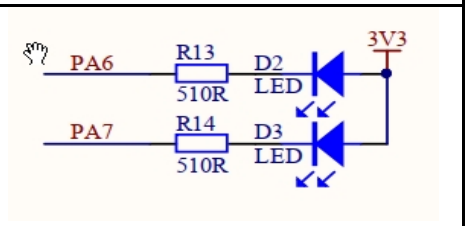
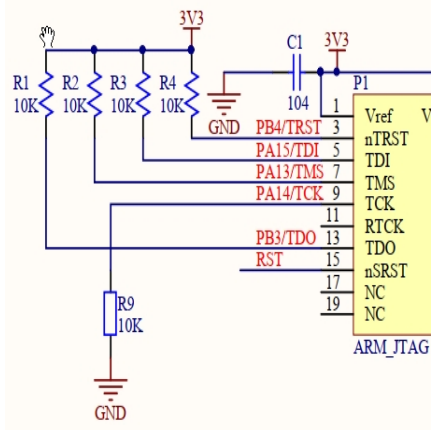
31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1
3.3V	GND	Pen-IRQ	MOSI	T-CLK	DC	RD	B1	B3	B5	B7	B9	B11	B13	B15	RST
3.3V	GND	LCD-BL	MISO	T-CS	CS	WR	B0	B2	B4	B6	B8	B10	B12	B14	GND
32	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2

STM32F407VGT6 Pins by Function

Function	Pins
COM1	TX - PA9 RX-PA10 OPTION SERIAL CONSOLE - to use as a serial console
COM2	TX - PC6 RX - PC7
COM3	TX - PA0 TX- PA1
COM4	TX - PA3 RX -PA2
I2C	SCL - PB6 (Can be also pickup from the NRF2401 socket) SDA - PB7 (Can be also pickup from the NRF2401 socket)
I2C2	SCL - PB10 SDA - PB11
SPI	CLK - PB3 (Can be also pickup from the NRF2401 socket) IN -PB4 (MISO) (not broken out to J2 or J3 - Pickup from the NRF2401 socket) OUT - PB5 (MOSI) (Can be also pickup from the NRF2401 socket)
SPI2	CLK - PB13 IN - PB14 (MISO) OUT - PB15 (MOSI)
DAC	1-PA4 (Not 5V tolerant 3.3v only) 2-PA5 (Not 5v tolerant 3.3v only)
PWM 1	1A -PA06 1B- PA07 1C- PB0
PWM 2	2A- PD12 2B- PB8 (Can be also pickup from the NRF2401 socket. Is also used as NRF2401 IRQ) 2C- PB9
PWM 3	3A- PE5 3B- PE6
KEYBOARD	CLOCK - PA15 DATA - PD3
Count Pins	PE1, PE3, PE4 and PA8 have hardware interrupts. They can be used with SETPIN CIN,FIN and PIN parameters for counting, frequency and period measurements
WAKE UP	PA0 is the wake up pin. K_UP will ground it when pressed. The count pins will also cause a wake up if configured.
Analogue Pins	The 13 pins PC0, PC1, PC2, PC3, PA0, PA1, PA2, PA3, PA6, PA7, PC4, PC5 and PB0 can be used as analogue pins. i.e. Capable of voltage measurement. Use SETPIN with AIN parameter.
ADC Pins	ch1 PC0, PC3, PA0, PA1, PA2, PA3, PA6, PA7, PB0 (Analogue A pins) ch2 PC2 (Analogue C pin) ch3 PC1, PC4, PC5 (Analogue B pins) The ADC has three input channels. The pins available to use for each channel are show above. When connected to the ADC they must not exceed 3.3v

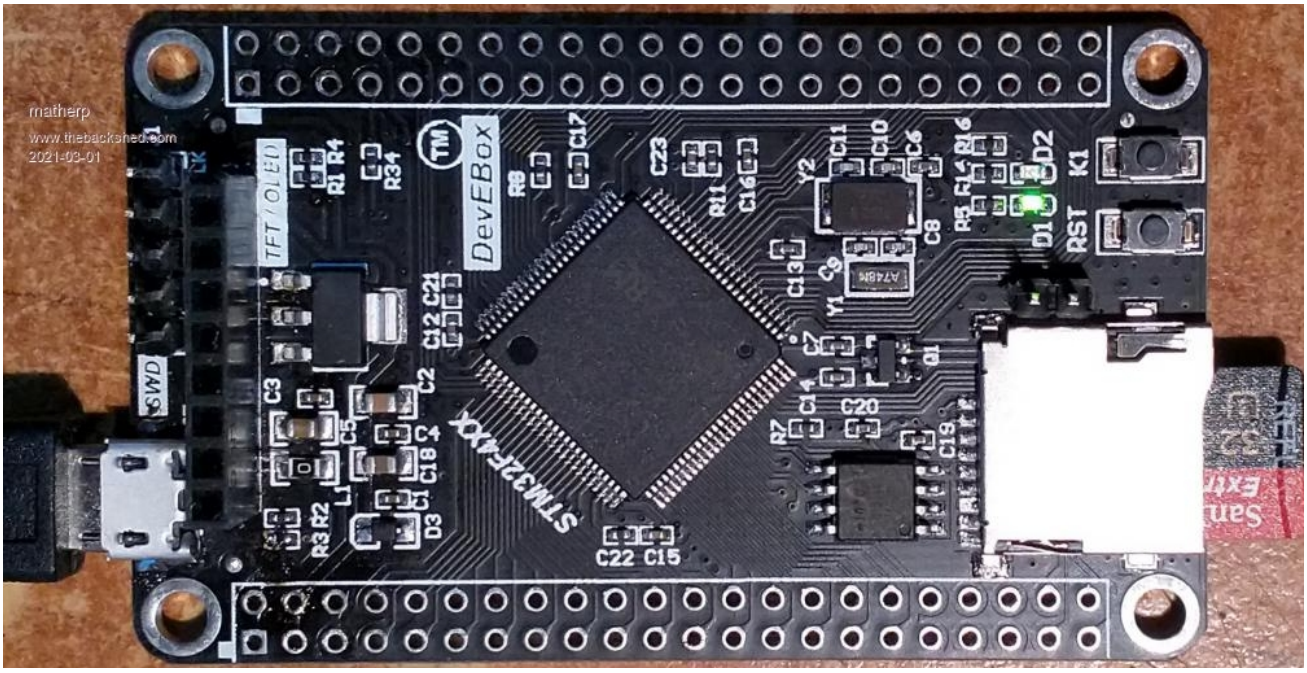
STM32F407VGT6 Modifications

You ***do not*** need to make any modifications to the board to use it. This is a great thing about this board, its ready to go when you get it, just load the firmware. The modifications shown below are only required if you need to resolve a related issue. Removing components from the board is relatively easy, to put them back is a little more difficult.

Component or Issue	Details	Circuit Detail
R21	The R21 pullup resistor on the USB D+ data line prevents the CPU SLEEP [n] working when using a USB console. Removing R21 will allow this to work with the USB console and has no other detrimental effects. This can safely be removed if you want to use CPU SLEEP n with the USB console. On the latest V33 board received R21 is in a diferent location and may not have the same function.	
R25	The USB 5v and the 5v pins on the board are connected together by 0 ohm resistor R25. This means if you power via an external 5v supply connected to the 5v pin, then this 5v will appear on the USB connector as well. If you want to power the board permanently from a 5V supply and want to remove any risk when connecting via a USB console you can remove R25.	
D2	Diode D2 and resistor R13 on PA6 to 3.3v This is an analogue pin and the presence to the diode/ resistor to 3.3v would affect any voltage reading. You would need to remove either component if this is an issue.	
D3	Diode D3 and resistor R14 on PA7 to 3.3v This is an analogue pin and the presence to the diode/ resistor to 3.3v would affect any voltage reading. You would need to remove either component if this is an issue.	
PA13 SWDIO	This pin has a 10K pullup resistor R2 to 3.3v installed. This will normally not be a problem and in many cases it what you want. Its listed here for information.	
PA14 SWCLK	This pin has a 10K pulldown resistor R9 to Gnd installed. This will normally not be a problem. Its listed here for information.	
PA15 KBD_CLK	A 4.7K pullup is normally recommend on KBD_CLK when connecting a PS2 keyboard. This 10K R3 resistor should be probably considered when deciding the actual value to use. The internal pullup and this resistor may be enough.	
PB4 SPI_IN	Has a 10K pullup resistor R4 to 3.3v Not known to affect the operation of the pin as SPI_IN	
PB3 SPI_CLK	Has a 10K pullup resistor R1 to 3.3v Not known to have any effect on operation of pin as SPI_CLK	
R1-R4	On the latest V33 board received R1-R4 are not discrete components, but are a resistor package.	

STM32F407VGT6 MINI Pin function and connector positions

The table shows pin allocation for the STM32407VGT6 MINI. It is essentially the same as the original but with the FSMC connector deleted. The pins used by the FSMC **are available** to MMBasic if no parallel display is configued. The table shows a suggested rearrangement of the T_CS and T_IRQ pins and allocation for the LCD D/C and LCD-RSTpins to best suit the connector layout for J1 and J4 connectors. The KBD pins are reused as general IO pins as no keyboard would be used. PD3 the old KBD-DATA pin is notionally allocated as the SPI LCD-CS pin. The old KBD-CLK pin is used for F-CS. Other pins can be used if desired.



A schematic of this board is availble at the link below.
https://www.thebackshed.com/forum/uploads/Volhout/2020-08-06_211055_STM32F407VX_M_schematics.pdf

STM32F407VGT6 MINI			MMBASIC	LCD Type		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	16Bit	SPI	MMBASIC	ADC	EXT.
1	PE02	J2-11	IR			DIN - DOUT		
2	PE03	J2-12	Count 2			DIN - DOUT		**KEY1/INT2
3	PE04	J2-13	Count 3			DIN - DOUT		**KEY0/INT3
4	PE05	J2-14	PWM-3A			DIN - DOUT		TIM9_CH1
5	PE06	J2-15	PWM-3B			DIN - DOUT		TIM9_CH2
6	VBAT							
7	PC13	J2-16				DIN – DOUT 3ma (see notes below)		
8	PC14		OSC32_IN					
9	PC15		OSC32_OUT					
10	GND			GND	GND			
11	3.3V			3.3V	3.3/5V	(ILI9481 needs 5V)		
12	OSC_IN		8MHz					
13	OSC_OUT		8MHz					
14	RST	RST KEY	NRST	RST				
15	PC00	J2-17	15			DIN - DOUT - AIN	ADC_10 [A]	
16	PC01	J2-18	16			DIN - DOUT - AIN	ADC_11 [B]	
17	PC02	J2-19	17			DIN - DOUT - AIN	ADC_12 [C]	
18	PC03	J2-20	18			DIN - DOUT - AIN	ADC_13 [A]	
19	VDD							

STM32F407VGT6 MINI			MMBASIC	LCD Type		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	16Bit	SPI	MMBASIC	ADC	EXT.
20	VREF-	J2-21						
21	VREF+	J2-22						
22	VDDA							
23	PA00	J2-23	COM3-TX			DIN - DOUT - AIN	ADC_0 [A]	KEY1 WK_UP
24	PA01	J2-24	COM3-RX			DIN - DOUT - AIN	ADC_1 [A]	LED D2
25	PA02	J2-25	COM4-TX			DIN - DOUT - AIN	ADC_2 [A]	
26	PA03	J2-26	COM4-RX			DIN - DOUT - AIN	ADC_3 [A]	
27	GND							
28	3.3V							
29	PA04	J2-27	DAC-1				DAC-1 (3.3v)	
30	PA05	J2-28	DAC-2				DAC-2 (3.3v)	
31	PA06	J2-29	PWM-1A			DIN - DOUT - AIN	ADC_6 [A]	
32	PA07	J2-30	PWM-1B			DIN - DOUT - AIN	ADC_7 [A]	
33	PC04	J2-31	33			DIN - DOUT - AIN	ADC_14 [B]	
34	PC05	J2-32	34		D/C	DIN - DOUT - AIN	ADC_15 [B]	
35	PB00	J2-33	PWM-1C			DIN - DOUT - AIN	ADC_8 [A]	
36	PB01	J2-34	LCD_BL	BL	BL	BACKLIGHT CMD		TIM3_CH4
37	PB02	J3-6	BOOT1			DIN - DOUT		
38	PE07	J2-35	FSMC_D4	D4		DIN - DOUT		
39	PE08	J2-36	FSMC_D5	D5		DIN - DOUT		
40	PE09	J2-37	FSMC_D6	D6		DIN - DOUT		
41	PE10	J2-38	FSMC_D7	D7		DIN - DOUT		
42	PE11	J2-39	FSMC_D8	D8		DIN - DOUT		
43	PE12	J2-40	FSMC_D9	D9		DIN - DOUT		
44	PE13	J2-41	FSMC_D10	D10		DIN - DOUT		
45	PE14	J2-42	FSMC_D11	D11		DIN - DOUT		
46	PE15	J2-43	FSMC_D12	D12		DIN - DOUT		
47	PB10	J2-44	I2C2-SCL			DIN - DOUT		
48	PB11	J2-45	I2C2-SDA			DIN - DOUT		
49	VCAP1		VCAP					
50	3.3V		VDD					
51	PB12	J2-46	51		RST	DIN - DOUT		
52	PB13	J2-47	SPI2-CLK	T_CLK	T_CLK	DIN - DOUT		
53	PB14	J2-48	SPI2-IN	T_MISO	T_MISO	DIN - DOUT		
54	PB15	J3-48	SPI2-OUT	T_MOSI	T_MOSI	DIN - DOUT		
55	PD08	J3-47	FSMC_D13	D13		DIN - DOUT		
56	PD09	J3-46	FSMC_D14	D14		DIN - DOUT		
57	PD10	J3-45	FSMC_D15	D15		DIN - DOUT		
58	PD11	J3-44	VBUS_FS			DIN - DOUT		
59	PD12	J3-43	PWM-2A			DIN - DOUT		TIM4_CH1
60	PD13	J3-42	FSMC_A18	DC		DIN - DOUT		
61	PD14	J3-41	FSMC_D0	D0		DIN - DOUT		
62	PD15	J3-40	FSMC_D1	D1		DIN - DOUT		
63	PC06	J3-39	COM2-TX			DIN - DOUT		
64	PC07	J3-38	COM2-RX			DIN - DOUT		
~~65~~	~~PC08~~	~~J3-37~~	~~SDIO_D0~~					

STM32F407VGT6 MINI			MMBASIC	LCD Type		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	16Bit	SPI	MMBASIC	ADC	EXT.
~~66~~	~~PC09~~	~~J3-36~~	~~SDIO_D1~~					
67	PA08	J3-35	Count 4			DIN - DOUT		INT4
68	PA09	J3-34	COM1-TX			DIN - DOUT		
69	PA10	J3-33	COM1-RX			DIN - DOUT		
70	PA11	J3-32	USB-DM					USB D-
71	PA12	J3-31	USB-DP					USB D+
72	PA13	J1-4	SWDIO/T-CS	T-CS	T-CS	DIN - DOUT		
73	VCAP2							
74	GND							
75	3.3V							
76	PA14	J1-5	SWCLK/T-IRQ	PEN-IRQ	T-IRQ	DIN - DOUT		
77	PA15	J3-30	F-CS			DIN - DOUT		
~~78~~	~~PC10~~	~~J3-29~~	~~SDIO_D2~~	-			Modify as	RST
~~79~~	~~PC11~~	~~J3-28~~	~~SDIO_D3~~					
~~80~~	~~PC12~~	~~J3-27~~	~~SDIO_CK~~				Modify as	PB4/SPI_IN
81	PD00	J3-26	FSMC_D2	D2		DIN - DOUT		CAN2 Rx
82	PD01	J3-25	FSMC_D3	D3		DIN - DOUT		CAN2 Tx
~~83~~	~~PD02~~	~~J3-24~~	~~SDIO_CMD~~					
84	PD03	J3-23	SPI LCD_CS		LCD-CS	DIN - DOUT		
85	PD04	J3-22	FSMC_NOE	RD		DIN - DOUT		
86	PD05	J3-21	FSMC_NWE	WR		DIN - DOUT		
87	PD06	J3-20	87			DIN - DOUT		
88	PD07	J3-19	FSMC_NE1	CS		DIN - DOUT		
89	PB03	J3-18	SPI_CLK			DIN - DOUT		
90	PB04	W25Q16-2	SPI_IN			DIN - DOUT	Not routed to a header pin	
91	PB05	J3-17	SPI_OUT			DIN - DOUT		
92	PB06	J3-16	I2C-SCL			DIN - DOUT		
93	PB07	J3-15	I2C-SDA			DIN - DOUT		
94	BOOT0	J3-5	BOOT0					
95	PB08	J3-14	PWM-2B			DIN - DOUT		CAN1 Rx
96	PB09	J3-13	PWM-2C			DIN - DOUT	TIM4_CH4	CAN1 Tx
97	PE00	J3-12	97			DIN - DOUT		
98	PE01	J3-11	COUNT 1			DIN - DOUT		INT1
99	GND							
100	3.3V							

** Key0 and Key1 need to be added externally to provide the MMBasic reset and serial console functions.

** PC13 can be used in lieu with the alternative PC13 firmware.

The MISO pin PB4 is not broken out to a header. It does however, appear on pin 2 of the flash chip.

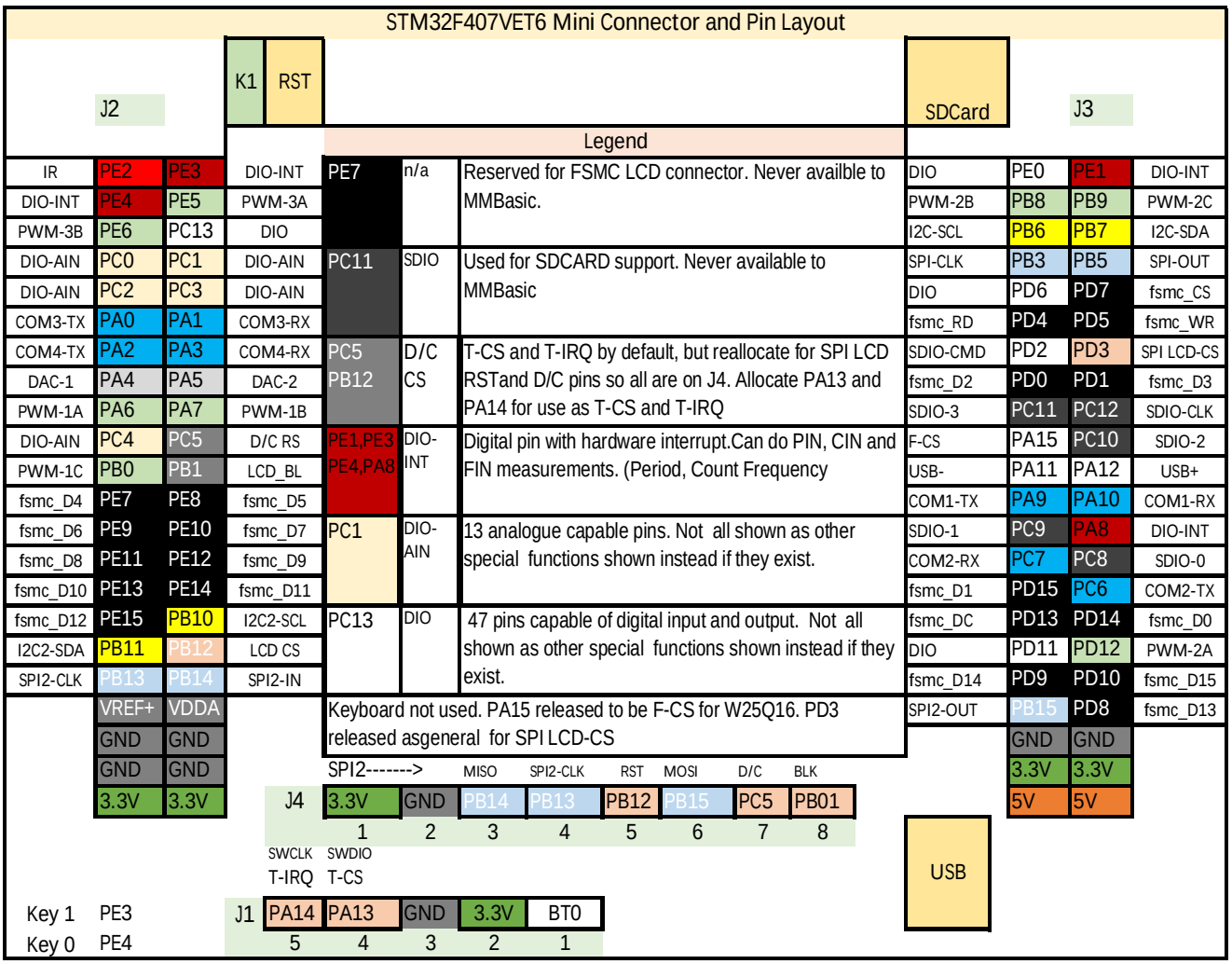
RST pin is not broken out to a header. It does appear on RST Key.

The option commands to match this pin allocation are as below for Touch and SPI LCD Panels:

OPTION LCDPANEL *controller*, *orientation*, PC5, PB12, PD3

OPTION TOUCH PA13, PA14

STM32F407VGT6 MINI Connector and Pin Layout



STM32F407VGT6 MINI Differences

It is physically smaller.

- W25Q16 flash chip select on pin PA15 (versus PB0 on F4)
- BOOT1 fixed to GND (that is fine)
- A LED connected to PA1 (versus 2 LEDs at PA6 and PA7 on F4)

Key0 and Key1 need to be added externally to provide the MMBasic reset and serial console functions.

No FSMC parallel connector

T-CS and T_IRQ move to PA13 and PA14 (not mandatory but frees them for LCD use)

PB12 and PC5 now used for RST and D/C for SPI LCDs

Keyboard pins PA15 and PD3 released for use, as keyboard would not be required.

F-CS is now PA15 (was PB0)

KEY 0 not present, need to jumper GND directly to PE4 to switch to serial console at restart.

Serial Console is picked up from PA9 and PA10

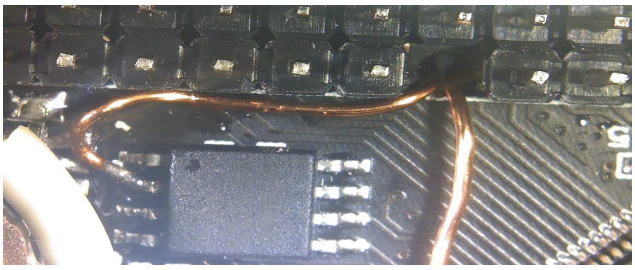
- no battery holder, 3V backup needs to be connected externally if required. B1 + and – connector.

If the board is modified as described below is can be used to support P16 parallel displays despite not having the FSMC connector. All the required pins are available on the other headers once the RST pin is routed.

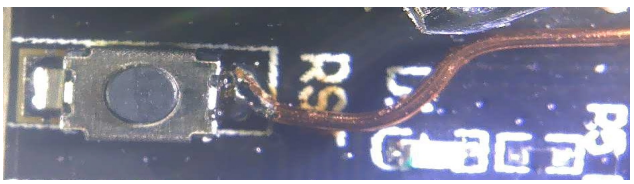
STM32F407VGT6 MINI – Modification to Route RST and SPI-IN

The PB4 pin, MISO for SPI1 is not routed to a header. The RST pin is also not routed to a header. This modification allows these pins to be made available on the header where the SDIO-2 and SDIO-CLK appear at PC10 and PC12. These are not required on the header and are recoverd for use by this modification. The SDIO-3 pin PC11 is also recovered as part of the modification. The SDIO tracks to the SDCARD route underneath the board and remain in tack, only the appearance at the PC10 and PC12 pins is disconnected.

The SDIO-3 connection at PC11 is drilled out with 2mm bit to allow two wires to pass through the board. The tracks to PC10 and PC12 are cut at the top of the board so they no longer reach the two pins. A wire is run from pin 2 on the W25Q16 flash chip via the hole and connected to PC12, this now becomes the PB4 pin and is MISO for SPI1.



A wire is run from the RST Key via the hole and connected to PC10, which now becomes the RST header pin.



The wires attached to PC10 and PC12 on the underside of the board

STM32F407VGT6 MINI Connector and Pin Layout -After Modification

STM32F407VET6 Mini Connector and Pin Layout

AFTER MODIFICATION

PB4 and RST routed by adding wires

J2			K1	RST				SDCard		J3	
IR	PE2	PE3	DIO-INT	PE7	n/a	Reserved for FSMC LCD connector. Never availble to MMBasic. Used for parrallel displays.	DIO	PE0	PE1	DIO-INT
DIO-INT	PE4	PE5	PWM-3A				PWM-2B	PB8	PB9	PWM-2C
PWM-3B	PE6	PC13	DIO				I2C-SCL	PB6	PB7	I2C-SDA
DIO-AIN	PC0	PC1	DIO-AIN	PC11	SDIO	Used for SDCARD support. Never available to MMBasic	SPI-CLK	PB3	PB5	SPI-OUT
DIO-AIN	PC2	PC3	DIO-AIN				DIO	PD6	PD7	fsmc_CS
COM3-TX	PA0	PA1	COM3-RX				fsmc_RD	PD4	PD5	fsmc_WR
COM4-TX	PA2	PA3	COM4-RX	PC5	D/C	T-CS and T-IRQ by default, but reallocate for SPI LCD	SDIO-CMD	PD2	PD3	SPI LCD-CS
DAC-1	PA4	PA5	DAC-2	PB12	CS	RSTand D/C pins so all are on J4. Allocate PA13 and	fsmc_D2	PD0	PD1	fsmc_D3
PWM-1A	PA6	PA7	PWM-1B			PA14 for use as T-CS and T-IRQ	Drilled Out	O	PB4	SPI-IN
DIO-AIN	PC4	PC5	D/C RS	PE1,PE3	DIO-	Digital pin with hardware interrupt.Can do PIN, CIN and	F-CS	PA15	RST	fsmc_RST
PWM-1C	PB0	PB1	LCD_BL	PE4,PA8	INT	FIN measurements. (Period, Count Frequency	USB-	PA11	PA12	USB+
fsmc_D4	PE7	PE8	fsmc_D5				COM1-TX	PA9	PA10	COM1-RX
fsmc_D6	PE9	PE10	fsmc_D7	PC1	DIO-	13 analogue capable pins. Not all shown as other	SDIO-1	PC9	PA8	DIO-INT
fsmc_D8	PE11	PE12	fsmc_D9		AIN	special functions shown instead if they exist.	COM2-RX	PC7	PC8	SDIO-0
fsmc_D10	PE13	PE14	fsmc_D11				fsmc_D1	PD15	PC6	COM2-TX
fsmc_D12	PE15	PB10	I2C2-SCL	PC13	DIO	47 pins capable of digital input and output. Not all	fsmc_DC	PD13	PD14	fsmc_D0
I2C2-SDA	PB11	PB12	LCD RST			shown as other special functions shown instead if they	DIO	PD11	PD12	PWM-2A
SPI2-CLK	PB13	PB14	SPI2-IN			exist.	fsmc_D14	PD9	PD10	fsmc_D15
	VREF+	VDDA		Keyboard not used. PA15 released to be F-CS for W25Q16. PD3 released asgeneral for SPI LCD-CS			SPI2-OUT	PB15	PD8	fsmc_D13
	GND	GND						GND	GND	
	GND	GND		SPI2------->		MISO SPI2-CLK RST MOSI D/C BLK		3.3V	3.3V	
	3.3V	3.3V	J4	3.3V	GND	PB14 PB13 PB12 PB15 PC5 PB01		5V	5V	
				1	2	3 4 5 6 7 8				
				SWCLK T-IRQ	SWDIO T-CS					
Key 1	PE3		J1	PA14	PA13	GND 3.3V BT0		USB		
Key 0	PE4			5	4	3 2 1				

STM32F407ZGT6 144 Pin function and connector positions

The capabilities and allocation of each pin are detailed in this table. MMBasic can address the pins via their pin number or connector name. This manual will only use the connector name but the first two columns of the table below show their correlation.

STM32F407ZGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
1	PE02	J2-11	IR			DIN - DOUT		
2	PE03	J2-12	Count 2			DIN - DOUT		**KEY1/INT2
3	PE04	J2-13	Count 3			DIN - DOUT		**KEY0/INT3
4	PE05	J2-14	PWM-3A			DIN - DOUT		TIM9_CH1
5	PE06	J2-15	PWM-3B			DIN - DOUT		TIM9_CH2
6	VBAT							
7	PC13	J2-16				DIN – DOUT 3ma (see notes below)		Blue Diode to Bat
8	PC14		OSC32_IN					
9	PC15		OSC32_OUT					
10	PF00							
11	PF01							
12	PF02							
13	PF03							
14	PF04							
15	PF05							
16	GND			GND	2,29 30			
17	3.3V			3.3V	31,32			
18	PF06							
19	PF07							
20	PF08							
21	PF09							
22	PF10							
23	OSC_IN		OSC8_IN					
24	OSC_OUT		OSC8_OUT					
25	RST	JTAG-15	NRST	RST	1			15 JTAG-RST
26	PC00	J2-17	15			DIN - DOUT - AIN	ADC_10 [A]	
27	PC01	J2-18	16			DIN - DOUT - AIN	ADC_11 [B]	
28	PC02	J2-19	17			DIN - DOUT - AIN	ADC_12 [C]	
29	PC03	J2-20	18			DIN - DOUT - AIN	ADC_13 [A]	
30	VDD							
31	VREF-	J2-21						
32	VREF+	J2-22						
33	VDDA							
34	PA00	J2-23	COM3-TX			DIN - DOUT - AIN	ADC_0 [A]	KEY_UP WK_UP
35	PA01	J2-24	COM3-RX			DIN - DOUT - AIN	ADC_1 [A]	
36	PA02	J2-25	COM4-TX			DIN - DOUT - AIN	ADC_2 [A]	
37	PA03	J2-26	COM4-RX			DIN - DOUT - AIN	ADC_3 [A]	
38	GND							
39	3.3V							
40	PA04	J2-27	DAC-1				DAC-1 (3.3v)	

STM32F407ZGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
41	PA05	J2-28	DAC-2				DAC-2 (3.3v)	
42	PA06	J2-29	PWM-1A			DIN - DOUT - AIN	ADC_6 [A]	LED D2
43	PA07	J2-30	PWM-1B			DIN - DOUT - AIN	ADC_7 [A]	LED D3
44	PC04	J2-31	33			DIN - DOUT - AIN	ADC_14 [B]	
45	PC05	J2-32	T_IRQ	PEN-IRQ	n/a	DIN - DOUT - AIN	ADC_15 [B]	
46	PB00	J2-33	PWM-1C			DIN - DOUT - AIN	ADC_8 [A]	F_CS ???
47	PB01	J2-34	LCD_BL	BL	n/a			TIM3_CH4
48	PB02	J3-6	BOOT1			DIN - DOUT		
49	PF11							
50	PF12		FSMC_A6	DC	21	NEW		
51	GND							
52	3.3V							
53	PF13							
54	PF14							
55	PF15							
56	PG00							
57	PG01							
58	PE07	J2-35	FSMC_D4	D4	14			
59	PE08	J2-36	FSMC_D5	D5	13			
60	PE09	J2-37	FSMC_D6	D6	12			
61	GND							
62	3.3V		VDD					
63	PE10	J2-38	FSMC_D7	D7	11			
64	PE11	J2-39	FSMC_D8	D8	10			
65	PE12	J2-40	FSMC_D9	D9	9			
66	PE13	J2-41	FSMC_D10	D10	8			
67	PE14	J2-42	FSMC_D11	D11	7			
68	PE15	J2-43	FSMC_D12	D12	6			
69	PB10	J2-44	I2C2-SCL			DIN - DOUT		
70	PB11	J2-45	I2C2-SDA			DIN - DOUT		
71	VCAP1		VCAP					
72	3.3V		VDD					
73	PB12	J2-46	T_CS	T_CS	n/a	DIN - DOUT		
74	PB13	J2-47	SPI2-CLK	T_CLK	n/a	DIN - DOUT		
75	PB14	J2-48	SPI2-IN	T_MISO	n/a	DIN - DOUT		
76	PB15	J3-48	SPI2-OUT	T_MOSI	n/a	DIN - DOUT		
77	PD08	J3-47	FSMC_D13	D13	5			
78	PD09	J3-46	FSMC_D14	D14	4			
79	PD10	J3-45	FSMC_D15	D15	3			
80	PD11	J3-44	VBUS_FS			DIN - DOUT		
81	PD12	J3-43	PWM-2A			DIN - DOUT		TIM4_CH1
82	PD13	J3-42						
83	GND							
84	3.3V							
85	PD14	J3-41	FSMC_D0	D0	18			
86	PD15	J3-40	FSMC_D1	D1	17			

STM32F407ZGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
87	PG02							
88	PG03							
89	PG04							
90	PG05							
91	PG06							
92	PG07							
93	PG08							
94	GND							
95	3.3V							
96	PC06	J3-39	COM2-TX			DIN - DOUT		
97	PC07	J3-38	COM2-RX			DIN - DOUT		
98	PC08	J3-37	SDIO_D0					
99	PC09	J3-36	SDIO_D1					
100	PA08	J3-35	Count 4			DIN - DOUT		INT4
101	PA09	J3-34	COM1-TX			DIN - DOUT		J6-TXD
102	PA10	J3-33	COM1-RX			DIN - DOUT		J6-RXD
103	PA11	J3-32	USB-DM					USB D-
104	PA12	J3-31	USB-DP					USB D+
105	PA13	JTAG-7	SWDIO			DIN - DOUT		7 JTAG-TMS
106	VCAP2							
107	GND							
108	3.3V							
109	PA14	JTAG-9	SWCLK			DIN - DOUT		9 JTAG-TCK
110	PA15	J3-30	KBD_CLK			DIN - DOUT	KEY to GND	5 JTAG-TDI
111	PC10	J3-29	SDIO_D2			Not available		
112	PC11	J3-28	SDIO_D3			Not available		
113	PC12	J3-27	SDIO_CK			Not available		
114	PD00	J3-26	FSMC_D2	D2	16			CAN2 Rx
115	PD01	J3-25	FSMC_D3	D3	15			CAN2 Tx
116	PD02	J3-24	SDIO_CMD					
117	PD03	J3-23	KBD_DATA			DIN - DOUT		
118	PD04	J3-22	FSMC_NOE	RD	19			
119	PD05	J3-21	FSMC_NWE	WR	20			
120	GND							
121	3.3V							
122	PD06	J3-20	87			DIN - DOUT		
123	PD07	J3-19						
124	PG09							
125	PG10							
126	PG11							
127	PG12		FSMC_NE4	CS	22	NEW		
128	PG13							
129	PG14							
130	GND							
131	3.3V							
132	PG15							

STM32F407ZGT6			MMBASIC	TFT-FSMC		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
133	PB03	J3-18	SPI_CLK			DIN - DOUT		JP2-5 NRF-SCK
134	PB04	JTAG-3	SPI_IN			DIN - DOUT		JP2-7 NRF-MISO
135	PB05	J3-17	SPI-OUT			DIN - DOUT		JP2-6 NRF-MOSI
136	PB06	J3-16	I2C-SCL			DIN - DOUT		JP2-3 NRF_CE
137	PB07	J3-15	I2C-SDA			DIN - DOUT		JP2-4 NRF_CS
138	BOOT0	J3-5	BOOT0					
139	PB08	J3-14	PWM-2B			DIN - DOUT		JP2-8 NRF-IRQ
140	PB09	J3-13	PWM-2C			DIN - DOUT		TIM4_CH4
141	PE00	J3-12	97			DIN - DOUT		
142	PE01	J3-11	COUNT 1			DIN - DOUT		INT1
143			PDR_ON					
144	3.3V							

** Key0 and Key1 may need to be added externally to provide the MMBasic reset and serial console functions.

** PC13 can be used in lieu with the alternative PC13 firmware.

This is the layout of the FSMC connector on the DevEBox STM32F407ZGT6 board. Note it is not the same as the DevEBox STM32F407VIT6 board. It seems to be designed for a capacitive touch so the blacked out pins where LCD-BL, MOSI, MISO, T-CLK, T-CS and T-IRQ would are not usable for the LCD. They will be need to be picked up from where they appear on the other headers. An adapter can pick up all the other pins from the FSMC connector, they match the layout of the FSMC for the DevEBox STM32F407VIT6 with the New TFT layout.

		45	49	46	82	118	86	115	59	63	65	67	77	79	25
		PB1	PF11	PB0	PF12	PD4	PD15	PD1	PE8	PE10	PE12	PE14	PD8	PD10	NRST
31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1
3.3V	GND				DC	RD	B1	B3	B5	B7	B9	B11	B13	B15	RST
3.3V	GND				CS	WR	B0	B2	B4	B6	B8	B10	B12	B14	GND
32	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2
		PB15	PB2	PC13	PG12	PD5	PD14	PD0	PE7	PE9	PE11	PE13	PE15	PD9	
		47	48	73	123	119	85	114	58	60	64	66	68	78	

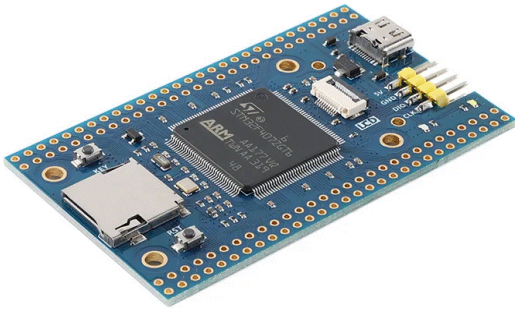
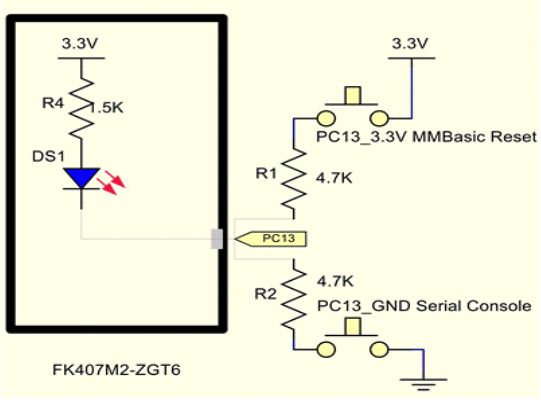
There are two 144 pin STM32F407ZGT6 boards that have been assessed.

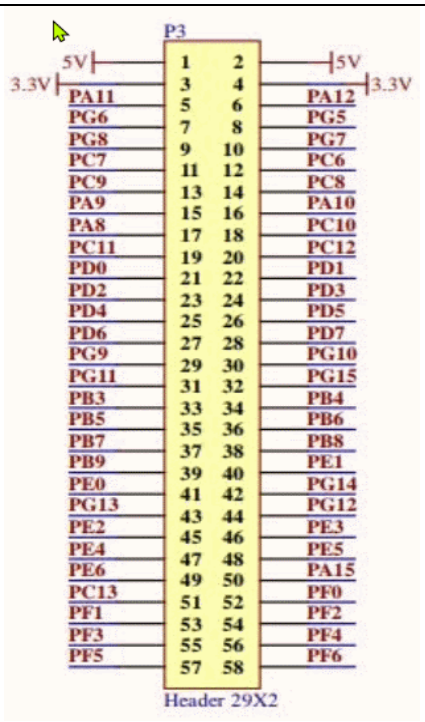
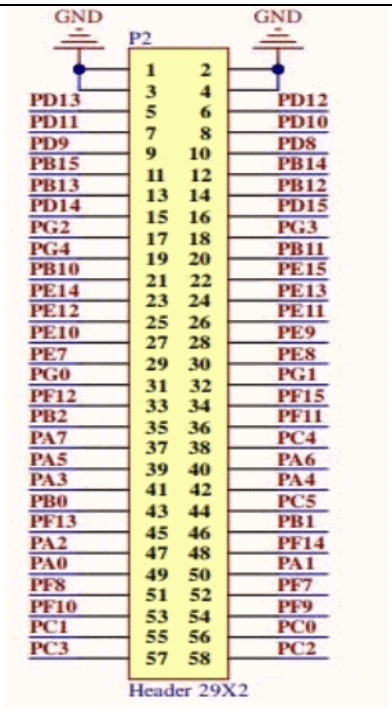
The DevEBox STM32F407ZGT6 board with the FSMC connector as described above, and the FK407ZGT6 board.

FK407M2-ZGT6 Schematic and MMBasic Reset

FK407M2-ZGT6 board (PCB model FK407M2-ZGT6):

https://pan.baidu.com/s/1nwMv30JHbXJthb48gnnWaw?pwd=6666

The FK407M2-ZGT6 board has an RST (Reset) key. At a power start up the default firmware will test the PE3 and PE4 pins for a low (GND) condition to determine if either an MMBasic Reset or a switch to Serial Console is required.	
At a power start up PC13Rest firmware will test the PC13 pin for both a high (3.3v) and a low (GND) condition. Add the resistors R1,R2, Key PC13_3.3v and Key PC13_GND as external components to achieve keys to allow switch for initiating and MMBasic Reset or switching to the Serial Console.	

	FK407M2-ZGT6 Viewed from Top			
		USB **KEY** **RST** **SDCARD**		

STM32F407RGT (Adafruit Feather) 64 Pin function and connector positions

The capabilities and allocation of each pin are detailed in this table. MMBasic can address the pins via their pin number or connector name. This manual will only use the connector name but the first two columns of the table below show their correlation. The feather does not break out all of the pins to a header.

STM32F407RGT6			MMBASIC	Misc		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
1	VBAT							
2	PC13	n/c				DIN – DOUT 3ma (see notes below)		
3	PC14		OSC32_IN	32KHz				
4	PC15		OSC32_OUT	32KHz				
5	OSC_IN		OSC8_IN	12MHz				
6	OSC_OUT		OSC8_OUT	12MHz				
7	RST	JP1-16	NRST					RST KEY
8	PC00	WS2812B				DIN - DOUT - AIN	ADC_10 [A]	WS2812B
9	PC01	JP3-4	COUNT 1	D13		DIN - DOUT - AIN	ADC_11 [B]	** RED LED
10	PC02	JP3-5	IR	D12		DIN - DOUT - AIN	ADC_12 [C]	
11	PC03	JP3-6	COUNT 2	D11		DIN - DOUT - AIN	ADC_13 [A]	
12	VSSA							
13	VDDA							
14	PA00	n/c	COM3-TX			DIN - DOUT - AIN	ADC_0 [A]	
15	PA01	n/c	COM3-RX			DIN - DOUT - AIN	ADC_1 [A]	
16	PA02	n/c	COM4-TX			DIN - DOUT - AIN	ADC_2 [A]	
17	PA03	3.3v/2	COM4-RX			DIN - DOUT - AIN	ADC_3 [A]	VBat/2
18	GND							
19	3.3V							
20	PA04	JP1-12	DAC-1	A0			DAC-1 (3.3v)	
21	PA05	JP1-11	DAC-2	A1			DAC-2 (3.3v)	
22	PA06	JP1-10	PWM-1A	A2		DIN - DOUT - AIN	ADC_6 [A]	
23	PA07	JP1-9	PWM-1B	A3		DIN - DOUT - AIN	ADC_7 [A]	
24	PC04	JP1-8	COUNT 3	A4		DIN - DOUT - AIN	ADC_14 [B]	
25	PC05	JP1-7	T_IRQ	A5		DIN - DOUT - AIN	ADC_15 [B]	
26	PB00	n/c	PWM-1C			DIN - DOUT - AIN	ADC_8 [A]	
27	PB01	n/c	LCD_BL	BL				TIM3_CH4
28	PB02	GND	BOOT1			DIN - DOUT		GND
29	PB10		I2C2-SCL			DIN - DOUT		
30	PB11		I2C2-SDA			DIN - DOUT		
31	VCAP1		VCAP					
32	3.3V		VDD					
33	PB12					DIN - DOUT		
34	PB13	JP1-6	SPI2-CLK			DIN - DOUT		
35	PB14	JP1-5	SPI2-IN			DIN - DOUT		
36	PB15	JP1-4	SPI2-OUT			DIN - DOUT		
37	PC06	JP3-9	COM2-TX	D6		DIN - DOUT		
38	PC07	JP3-10	COM2-RX	D5		DIN - DOUT		
39	PC08		SDIO_D0					
40	PC09		SDIO_D1					
41	PA08	n/c	COUNT 4			DIN - DOUT		INT4

STM32F407RGT6			MMBASIC	Misc		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
42	PA09	3.3v	~~COM1-TX~~			DIN - DOUT		
43	PA10	3.3v	~~COM1-RX~~			DIN - DOUT		
44	PA11		USB-DM					USB D-
45	PA12		USB-DP					USB D+
46	PA13	TP1	SWDIO			DIN - DOUT		
47	VCAP2							
48	3.3V							
49	PA14	TP2	SWCLK			DIN - DOUT		
50	PA15		F_CS			DIN - DOUT		F-CS
51	PC10		SDIO_D2			Not available		
52	PC11		SDIO_D3			Not available		
53	PC12		SDIO_CK			Not available		
54	PD02		SDIO_CMD					
55	PB03		SPI_CLK			DIN - DOUT		
56	PB04		SPI_IN			DIN - DOUT		
57	PB05		SPI-OUT			DIN - DOUT		
58	PB06	JP3-11	I2C-SCL			DIN - DOUT		
59	PB07	JP3-12	I2C-SDA			DIN - DOUT		
60	BOOT0	JP1-1	BOOT0					
61	PB08	JP3-8	PWM-2A	D9		DIN - DOUT	CAN-Rx	
62	PB09	JP3-7	PWM-2B	D10		DIN - DOUT	CAN-Tx	TIM4_CH4
63	GND							
64	3.3V							

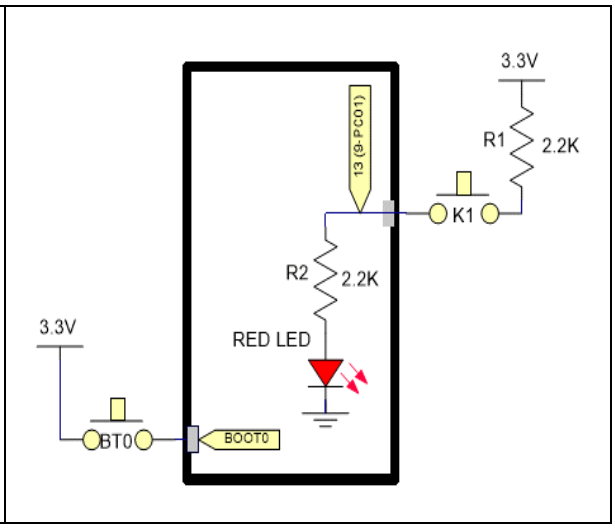
** PC01 is used for MMBasic Reset. See below for additional external wiring.

Adafruit Feather Adding BOOT0 and K1 switch for MMBasic Reset

Connect BOOT0 to 3.3V at power up to load firmware

Add these external components to achieve the K1 switch to allow a full reset of MMBasic if required. The BT0 key is to allow loading of firmware.

The internal LED and PC01 pin can be used as normal when K1 is not pressed. During boot the firmware looks for 3.3v on PC01 to see if a full reset of MMBasic is required.



Adafruit Feather Functional Layout

		Adfruit Feather STM32F405RGT								
7-RST		RST								
		3.3V								
		3.3V								
		GND								
20-PA04		DAC-1								
21-PA05		DAC-2								
22-PA06	ADC[A]	PWM 1A								
23-PA07	ADC[A]	PWM 1B							COUNT1	ADC[B]	9-PC01
24-PC04	ADC[B]	COUNT 3							IR	ADC[C]	10-PC02
25-PC05	ADC[B]								COUNT 2		11-PC03
34-PB13		SPI2-CLK							PWM 2A	CAN-Rx	61-PB08
36-PB15		SPI2-OUT							PWM 2B	CAN-Tx	62-PB09
35-PB14		SPI2-IN							COM2-TX		37-PC06
30-PB11		COM1-RX							COM2-RX		38-PC07
29-PB10		COM1-TX							I2C -SCL		58-PB06
		BOOT0							I2C -SDA		59-PB07
	PA9 and PA10 are connected to 3.3V - used to identify as Feather.										
	ADC OPEN [22|23],[10],[9|24|25]										
	9-PC01 is LED , then 2.2K + switch K1 to 3.3v for MMBasic Reset										
	SPI Flash. Flash_CS=50 Connected to SPI. SPI not on header.										

STM32F407RGT (WeAct Studio) 64 Pin function and connector positions

The capabilities and allocation of each pin are detailed in this table. MMBasic can address the pins via their pin number or connector name. This manual will only use the connector name but the first two columns of the table below show their correlation.

STM32F407RGT6			MMBASIC	Misc		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
1	VBAT							
2	PC13		MMBasic Reset			DIN – DOUT 3ma (see notes below)		**C13 Key
3	PC14		OSC32_IN	32KHz				
4	PC15		OSC32_OUT	32KHz				
5	OSC_IN		OSC8_IN	8MHz				
6	OSC_OUT		OSC8_OUT	8MHz				
7	RST		NRST					NR KEY
8	PC00					DIN - DOUT - AIN	ADC_10 [A]	
9	PC01			D13		DIN - DOUT - AIN	ADC_11 [B]	
10	PC02			D12		DIN - DOUT - AIN	ADC_12 [C]	
11	PC03			D11		DIN - DOUT - AIN	ADC_13 [A]	
12	VSSA							
13	VDDA							
14	PA00		COM3-TX			DIN - DOUT - AIN	ADC_0 [A]	
15	PA01		COM3-RX			DIN - DOUT - AIN	ADC_1 [A]	
16	PA02		COM4-TX			DIN - DOUT - AIN	ADC_2 [A]	
17	PA03		COM4-RX			DIN - DOUT - AIN	ADC_3 [A]	
18	GND							
19	3.3V							
20	PA04		DAC-1	A0			DAC-1 (3.3v)	
21	PA05		DAC-2	A1			DAC-2 (3.3v)	
22	PA06		PWM-1A	A2		DIN - DOUT - AIN	ADC_6 [A]	
23	PA07		PWM-1B	A3		DIN - DOUT - AIN	ADC_7 [A]	
24	PC04			A4		DIN - DOUT - AIN	ADC_14 [B]	
25	PC05		T_IRQ	A5		DIN - DOUT - AIN	ADC_15 [B]	
26	PB00		PWM-1C			DIN - DOUT - AIN	ADC_8 [A]	
27	PB01		LCD_BL	BL				TIM3_CH4
28	PB02		BOOT1			DIN - DOUT		D1 Blue
29	PB10		I2C2-SCL			DIN - DOUT		
30	PB11		I2C2-SDA			DIN - DOUT		
31	VCAP1		VCAP					
32	3.3V		VDD					
33	PB12					DIN - DOUT		
34	PB13		SPI2-CLK			DIN - DOUT		
35	PB14		SPI2-IN			DIN - DOUT		
36	PB15		SPI2-OUT			DIN - DOUT		
37	PC06		COM2-TX	D6		DIN - DOUT		
38	PC07		COM2-RX	D5		DIN - DOUT		
39	PC08		SDIO_D0					
40	PC09		SDIO_D1					
41	PA08		Count 4			DIN - DOUT		INT4
42	PA09		COM1-TX			DIN - DOUT		CON-TX

STM32F407RGT6			MMBASIC	Misc		Available Pins (No entry means unavailable)		
PIN	NAME	Connector	FUNCTIONS	NAME	PIN	MMBASIC	ADC	EXT.
43	PA10		COM1-RX			DIN - DOUT		CON-RX
44	PA11		USB-DM					USB D-
45	PA12		USB-DP					USB D+
46	PA13		SWDIO			DIN - DOUT		
47	VCAP2							
48	3.3V							
49	PA14		SWCLK			DIN - DOUT		
50	PA15					DIN - DOUT		
51	PC10		SDIO_D2			Not available		
52	PC11		SDIO_D3			Not available		
53	PC12		SDIO_CK			Not available		
54	PD02		SDIO_CMD					
55	PB03		SPI_CLK			DIN - DOUT		
56	PB04		SPI_IN			DIN - DOUT		
57	PB05		SPI-OUT			DIN - DOUT		
58	PB06		I2C-SCL			DIN - DOUT		
59	PB07		I2C-SDA			DIN - DOUT		
60	BOOT0		BOOT0					
61	PB08		PWM-2A	D9		DIN - DOUT	TIM4_CH3	CAN-RX
62	PB09		PWM-2B	D10		DIN - DOUT	TIM4_CH4	CAN-TX
63	GND							
64	3.3V							

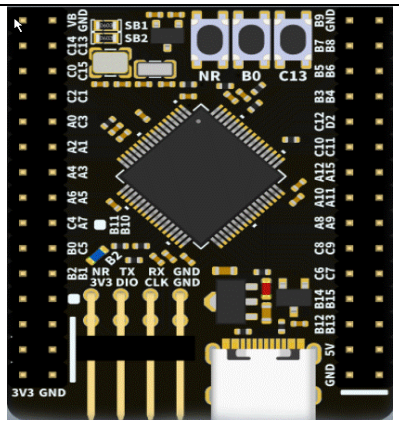
** PC13 is used to provide MMBasic Reset and switch to Serial Console. See below

STM32F407RGT (WeAct) Adding a Key for Serial Console

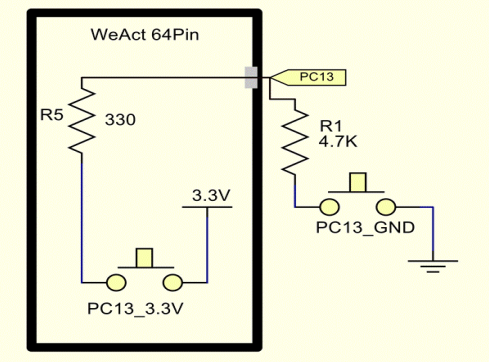
The WeAct 64 pin board has three keys. NR (Reset), B0 (for boot loader mode) and C13.

At a power start up of the WeAct 64 pin board MMBasic will test the PC13 pin for both a high (3.3v) and a low (GND) condition.

When pressed the C13 key places 3.3V via a 330 ohm resistor on PC13, so if pressed during a power restart and MMBasic Reset will be initiated.

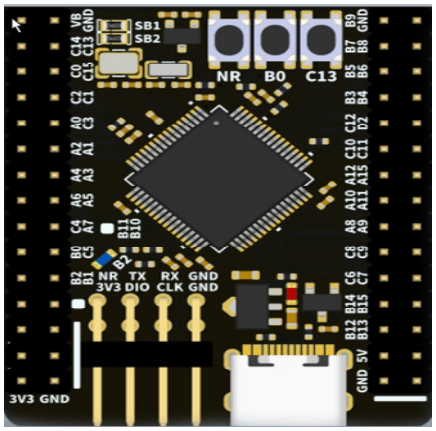


Add the resistor R1 and Key PC13_GND as external components to achieve a key to allow switch to Serial Console by placing a ground on PC13 at a power restart.



STM32F405RGT6 WeAct Connector and Pin Layout

					SDCard (under)	NR RST	B0	PC13 K0
			Legend					
	VBAT	GND				PWM-2B CAN-RX	PB9	GND	
OSC32	PC14	PC13	KEY 0			I2C-SDA	PB7	PB8	PWM-2A CAN-TX
WS2812B	PC0	PC15	OSC32			SPI-OUT	PB5	PB6	I2C-SCL
DIO-AIN	PC2	PC1	DIO-AIN			SPI-CLK	PB3	PB4	SPI-IN
COM3-TX	PA0	PC3	DIO-AIN			SDIO-CK	PC12	PD2	SDIO-CMD
COM4-TX	PA2	PA1	COM3-RX			SDIO-D2	PC10	PC11	SDIO-D3
DAC-1	PA4	PA3	COM4-RX			USB+	PA12	PA15	
PWM-1A	PA6	PA5	DAC-2			COM1-RX	PA10	PA11	USB-
DIO-AIN	PC4	PA7	PWM-1B			Count 4	PA8	PA9	COM1-TX
PWM-1C	PB0	PC5	LCD RST			SDIO-D0	PC8	PC9	SDIO-D1
BOOT1	PB2	PB1	LCD_BL			COM2-TX	PC6	PC7	COM2-RX
I2C2-SDA	PB11	PB10	I2C2-SCL	NR	TX	RX	GND	SPI2-IN	PB14	PB15	SPI2-OUT
	3.3V	GND		3.3V	PA13	PA14	GND		PB12	PB13	SPI2-CLK
	3.3V	GND							5V	5V	
	3.3V	GND							GND	GND	
								USB-C			



Key 1	PC0
Key 0	PC13

Using MMBasic

Commands and Program Input

At the command prompt you can enter a command and it will be immediately run. Most of the time you will do this to tell the Armmite 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. (type EDIT at the command prompt) This will invoke the full screen program editor which is built into MMBasic and is described Full Screen Editor section. It includes advanced features such as search and copy, cut and paste to and from a clipboard.

You can also compose the program on your desktop computer using something like Notepad and then transfer it to the Armmite via the XModem protocol (see the XMODEM command).

You can also type a program (or paste it from the windows clipboard if you have copied it from somewhere) by using the AUTOSAVE command to stream it via the serial port. At the command prompt type AUTOSAVE, then paste the clipboard into the terminal (or just type what you want) and when finished do a CNTRL Z to save the program. (On TeraTerm right mouse click will open the paste window.)

Another very convenient method of writing and debugging a program is to use MMEdit. This is a program running on your Windows computer (also will run on Linux under Wine) which allows you to edit your program on your computer then transfer it to the Armmite 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.
There are several other utilities at this site which may be useful.

With all of these methods of entering and editing a program the result is saved in non-volatile flash memory (this is transparent to the user). With the program held in flash memory it means that it will never be lost, even when the power is unexpectedly interrupted or the processor restarted.

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.

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. See Full Screen and Commandline Editors for more details.

Shortcut Keys at Commandline

When you are using a VT100 compatible terminal emulator on the console you can use the following function keys to insert the following commands at the command prompt:

F2	RUN
F3	LIST
F4	EDIT
F5	Sends ESC sequence to clear the VT100 screen
F10	AUTOSAVE
F11	XMODEM RECEIVE
F12	XMODEM SEND

Pressing the key will insert the text at the command prompt (except for F5 whichs sends back to the VT100 terminal), just as if it had been typed on the keyboard.

Shortcut Keys in AUTOSAVE

The AUTOSAVE commands sets the console waiting to accept a program. Anything typed to pasted in is interpreted as a program. The following keys sequences can be used to signal the end of the program and to trigger the saving of the entered text.

CNTRL+Z	Saves the program
F1	Saves the program
F2	Saves the program and immediately runs it.

Line Numbers and Program Structure

The structure of a program line is:

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

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

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

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

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

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

Running or Interrupting a 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.

You can list a program in memory with the LIST command. This will print out the program while pausing after every page.

You can completely erase the program by using the NEW command.

Programs in the Armmites and Micromites is held in non-volatile flash memory. This means that it will not be lost if the power is removed and, if you have the AUTORUN feature turned on, the Micromite/Armmite will start by automatically running the program when power is restored (use the OPTION command to turn AUTORUN on).

Saved Variables

Often there is a need to save data that can be recovered when power is restored. For example, program options, calibration settings, etc. This can be done with the VAR SAVE command which will save the variables listed on its command line in non-volatile flash memory. 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 Armmite F407xGT 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 use the RTC's battery backed memory and PEEK and POKE commands. The variable MM.INFO(BACKUP) i.e. &H40024400 will give the start address for 3K of battery backed RTC memory.

Timing

MMBasic has a number of features that make it easy to time events and control external circuitry that needs timing.

MMBasic maintains an internal clock. You can get the current date and time using the DATE$ and TIME$ functions and you can set them by assigning the new date and time to them. The calendar will start from zero each time Armmite is first powered up **except** if the RTC returns a realistic date (i.e. > 2018) in which case it will set its time from the battery backed-up RTC included in the Armmite F4.

The PAUSE command will freeze the execution of the program for a specified number of milliseconds. So, to create a 12ms wide pulse you could use the following:

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

You can also create a pulse using the PULSE command. This will generate very narrow pulses (e.g., 20µs) or long pulses up to several days. Long pulses are run in the background and the program will continue uninterrupted.

Another useful feature is the TIMER function which acts like a stopwatch. You can set it to any value (usually zero) and it will count upwards every millisecond.

A timing function is also provided by the SETTICK command. This command will generate an interrupt at regular intervals (specified in milliseconds). Think of it as the regular "tick" of a watch. For example, the following code fragment will print the current time, and the value of the analogue voltage read on pin PC0, every second. This process will run independently of the main program which could be doing something completely unrelated.

```
SETPIN PC0, AIN
SETTICK 1000, DOINT
DO
   ‘ main processing loop
LOOP

SUB DOINT          ‘ tick interrupt
  PRINT TIME$, PIN(PC0)
END SUB
```

The second line sets up the "tick" interrupt, the first parameter of SETTICK is the period of the interrupt (1000ms) and the second is the starting label of the interrupt code. Every second (i.e., 1000 ms) the main processing loop will be interrupted and the program starting at the label DOINT will be executed.

Up to four "tick" interrupts can be setup. These interrupts have the lowest priority.

The accuracy of the Armmite’s battery backed Real Time Clock can vary by a little due to manufacturing tolerances and temperature. To compensate for this the OPTION RTC CALIBRATE command can be used to trim the clock to a more accurate value.

Watchdog Timer

One of the possible uses for the Armmite F4 is as an embedded controller. It can be programmed in MMBasic and when the program is debugged and ready for "prime time" the AUTORUN configuration setting can be turned on. The chip will then automatically run its program when power is applied and act as a custom integrated circuit performing some special task. The user need not know anything about what is running inside the chip.

However, there is the possibility that a fault in the program could cause MMBasic to generate an error and return to the command prompt. This would be of little use in an embedded situation as the Armmite F4 would not have anything connected to the console. Another possibility is that the MMBasic program could get itself stuck in an endless loop for some reason. In both cases the visible effect would be the same… the program would stop running until the power was cycled.

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

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

PIN Security

Sometimes it is important to keep the data and program in an embedded controller confidential. In the Armmite F4 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 Armmite F4 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 Armmite. 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 Armmite. Once set the PIN can only be removed by providing the correct PIN as set in the first place. If the number is lost the only method of recovery is to reset MMBasic as described below (which will erase the program).

There are other time consuming ways of accessing the data (such as using the STM32Cube Programmer to examine the flash memory) so this should not be regarded as the ultimate security but it does act as a significant deterrent.

Single, Secure HEX File

If you write a program for the Armmite F4 and set the following options:

```
OPTION BREAK 0
OPTION AUTORUN ON
```

you will end up with a program that cannot be stopped or interrupted. To further bullet proof it you could use the watchdog timer and OPTION PIN.

You can then use STM32CubeProgrammer to read the complete flash memory of the Armmite F4 and export it as a hex file. This will contain the MMBasic firmware as well as your BASIC program and the above options.

This file can be sent to someone as custom firmware for the STM32F407VET6 development board. They can load the hex file and it will immediately start running your program. To them it will be indistinguishable from firmware written in C (other than the startup banner produced by MMBasic). They do not have to load MMBasic and they do not need know anything about programming for the Armmite F4.

Commands Vs Functions

Your program will be made up of MMBasic commands and functions. A command will tell MMBasic to do something. The program does not expect it to return a value, it assumes it will be done. e.g. to set the date.
DATE$=”20/05/2021”
Commands are all listed in the Commands section of this manual.

A function will always return a value. The function expects to return a value and the program must have a variable of the correct type ready to accept it. e.g. to get the date into variable today$
today$=DATE$
Using the PRINT command is an easy way to test a function without explicitly needing to know what it returns. e.g.
PRINT DATE$
The ? character can be used as shorthand for the PRINT command. e.g.
? DATE$

The functions are all listed in the Functions section of this manual.

Read Only Variables

MMBasic has a number of read only variables which you can use to determine various information about MMBasic and the hardware it is running on.e.g. what version of MMBasic, LCD type etc. These are all described in the Predefined Read Only Variables section.

OPTION RESET to set the options to the default values if required.

Setting Options

Many options can be set by using commands that start with the keyword OPTION. They are listed in the Option Settings section of this manual. For example, you can set the baud rate of the console with the command:

```
OPTION BAUDRATE 115200
```

Saving Options

Options are saved in 80 bytes of battery backed up RTC RAM. The options are not overwritten when new firmware is loaded, so if you had an LCDPANEL configured and loaded new firmware, then it would still be configured. Use the command

Resetting MMBasic

MMBasic can be reset to its original configuration using the following method:

- Holding KEY 1 down, while applying power or pressing the RST button. You can connect ground to PE3 pin if you find it difficult to hold the small button down.

This will result in the program memory and saved variables being completely erased and all options (security PIN, console baud rate, etc.) will be reset to their initial defaults. This includes setting the console to the USB.

OPTION RESET

Issuing this command will reset all options to their default values.

Quick Start Tutorial

Immediate Mode

Assuming that you have correctly connected a terminal emulator to the Armmite and have the command prompt (the greater than symbol as shown above, i.e., >) you can enter a command line followed by the enter key and it will be immediately run.

For example, if you enter the command PRINT 1/7 you should see this:

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

This is called immediate mode and is useful for testing commands and their effects.

A Simple Program

To enter a program, you can use the EDIT command which is fully described later in this manual. However, to get a quick feel for how it works, try this sequence (your terminal emulator must be VT100 compatible):

- 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 in your terminal emulator (or CTRL-Q which will do the same thing). 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 Armmite. If you type EDIT again you will be back in the editor where you can change or add to your program.

Flashing a LED on the STM32F407VET6 board

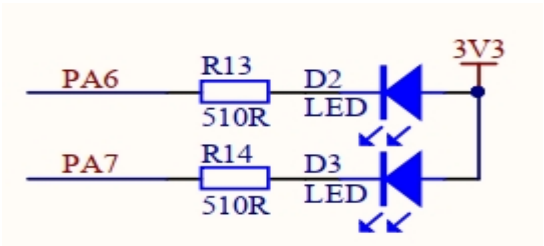
The board already has two diodes set up to use. D2 of pin PA6

D3 on pin PA7. Let's flash D2.

Use the EDIT command to enter the following program:

```
SETPIN PA6, DOUT
DO
  PIN(PA6) = 1
  PAUSE 500
  PIN(PA6) = 0
  PAUSE 500
LOOP
```

The STM32F407VET6 has these LEDs with appropriate resistors ready to go on the PA6 and PA7 pins.



When you have saved and run this program you should be greeted by the LED flashing on and off. It is not a great program but it does illustrate how your Armmite F4 can interface to the physical world via your programming.

The chapter Using the I/O pins later in this manual provides a full description of the I/O pins and how to control them.

Tutorial on Programming in the BASIC Language

If you are new to the BASIC programming language now would be a good time to read (*Programming in BASIC - A Tutorial*) at the rear of Geoff Grahame's excellent Picomite User 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.

Setting the AUTORUN Option

You now have the Armmite F4 doing something useful (if you can call flashing a LED useful). Assuming that this is all that you want the Armmite to do you can then instruct it to always run this program whenever power is applied.

To do this you first need to regain the command prompt and you can do this by entering CTRL-C at the console. This will interrupt the running program and return you to the command prompt.

Then enter the command:

```
OPTION AUTORUN ON
```

This instructs MMBasic to automatically run your program whenever power is applied. To test this, you can remove the power and then re-apply it. The Armmite should start up flashing the LED.

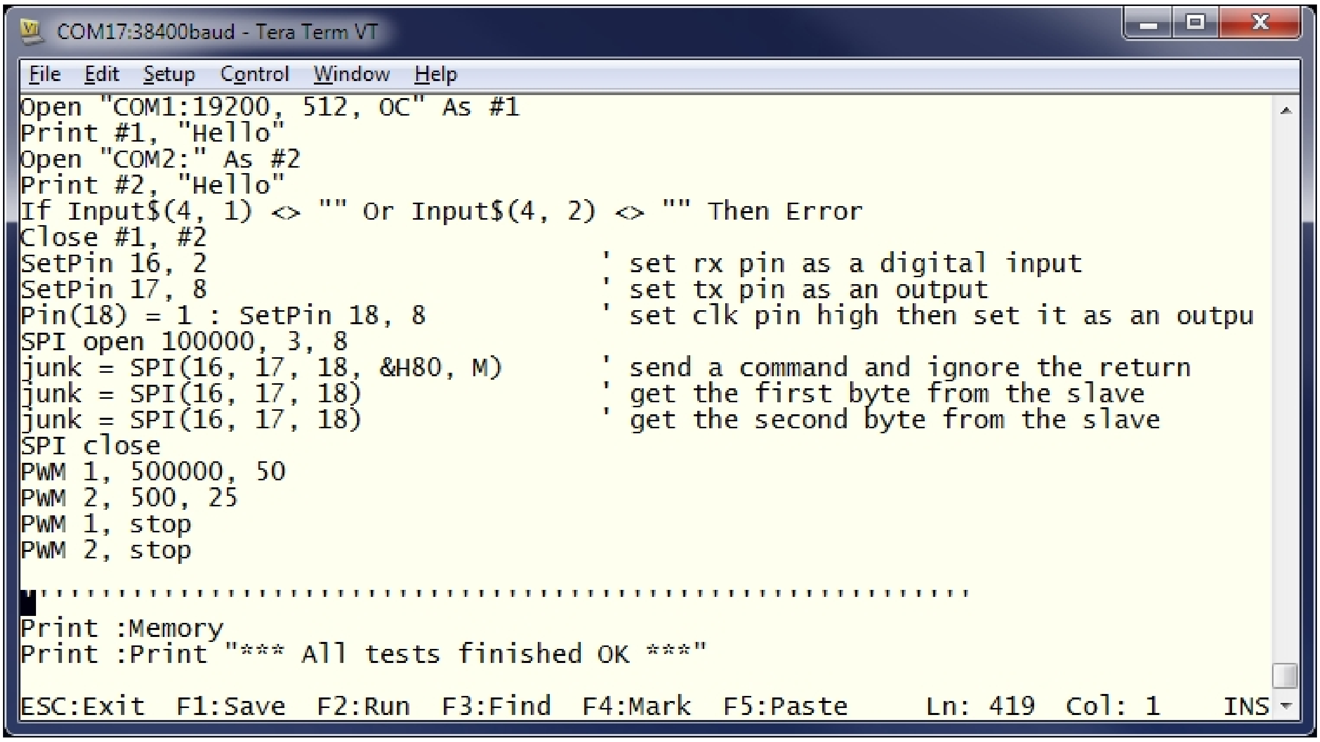
If this is all that you want, you can disconnect the console and it will sit there flashing the LED on and off forever. If ever you wanted to change something (for example the pause between on and off) you can attach your terminal emulator to the console, interrupt the program with a CTRL-C and edit it as needed.

This is the great benefit of the Armmites and Micromites, it is very easy to write and change a program.

Full Screen and Commandline Editors

Full Screen Editor

An important productivity feature of the Micromites/Armmites is the full screen editor. This will work with any VT100 compatible terminal emulator (Tera Term is recommended).



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

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

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

ESC	This will cause the editor to abandon all changes and return to the command prompt with the program memory unchanged. If you have changed the text you will be asked if you really 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 repeatedly search for the same text by pressing SHIFT-F3.
F4: MARK	This is described in detail below.
F5: PASTE	This will insert (at the current cursor position) the text that had been previously cut or copied (see below).

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 function 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 the full screen editor is to simply fire it up and experiment.

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

Using the OPTION BAUDRATE command the baud rate of the console can be changed to any speed up to 230400 bps. Changing the console baud rate to a higher speed makes the full screen editor much faster in redrawing the screen. If you have a reliable connection to the Armmite it is worth changing the speed to at least 115200. 115200 is the default speed on the Serial Console for the Armmite F4.

The editor expects that the terminal emulator is set to 24 lines per screen with each line 80 characters wide. Both of these assumptions can be changed with the OPTION DISPLAY command to suit non standard displays.

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

Long Lines in the Editor

MMBasic lines can be up to 255 characters. Long lines will only display the first part of the line up to the display's right hand margin. The rest of the line beyond the right hand margin is still there but it is not displayed and cannot be edited. This is a limitation of MMBasic, only the CMM2 and MMB4W get round this by having much more memory to allow sideways scrolling in a completely re-written editor.

For long lines the easiest way is to import the program from a PC (autosave or XModem).

 Hot Tip	If you want to edit a very long line you can position the cursor near the right hand margin and press Enter. This will split the long line into two and both parts can be separately edited. To rejoin the line use the Delete or Backspace key to remove the line break that you previously entered.

Colour Coded Editor Display

The editor has the ability to colour code the edited program with keywords, numbers and comments displayed in different colours. By default, the output is colour coded on the Armmite F4 but this feature can be disabled/enabled with the commands:

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

This setting is saved in flash memory and is automatically applied on startup.

Note:

- This feature requires a terminal emulator that can interpret the appropriate escape codes and respond correctly. It works correctly with Tera Term however, Putty needs its default background colour to be changed to white (Settings >> Colours >> Default Background >> Modify).
- Colour coding the editor's output requires many extra characters to be sent to the terminal emulator and this can slow down the screen update at lower baud rates. If colour coding is used it is recommended that the baud rate be set to a higher speed (115200) as discussed above.

Command Line Buffer and Editor

The ArmmiteF4 implements a command buffer at the command prompt. The Up and Down arrows can be used to locate previous commands in the buffer. A 1024 byte buffer is used to store as many previous commands as will fit. A command is added to the buffer whenever it is sent. i.e. enter key pressed.

Commands being entered or recalled from the buffer can be edited. The following are supported.

The command line can be up the 255 characters. This will wrap to the next line as required on the VT100 terminal and also the LCDPANEL if OPTION LCDPANEL CONSOLE is used.

Key	**Action**
Enter	Adds command to buffer and sends to console
BackSpace	Destructive backspace. Moves 1 character left and clears the character, pulls any characters to the right across 1 character. Turns on edit mode so Up and Down arrows are disabled.
Left Arrow	Moves cursor one character left. INS (Insert Mode) is turned on so any character type will be inserted at the the cursor position. Turns on edit mode so Up and Down arrows are disabled. An additional Left Arrow issue at the home position will turn on OVR (overwrite) mode.
Right Arrow	Moves cursor one character right. Turns on edit mode so Up and Down arrows are disabled.
DEL	Will delete the character under the cursor and pulls any characters to the right across 1 character. Turns on edit mode so Up and Down arrows are disabled.
INS	Toggles between Insert Mode and Overwrite Mode. INS (Insert Mode). Any character type will be inserted at the cursor position. OVR (Overwrite Mode). Any character type will overwrite character at the cursor position.
HOME HOME+HOME	Returns cursor to first character position. Turns on edit mode so Up and Down arrows are disabled. HOME pressed while at the home position will turns off edit mode so Up and Down arrows are enabled. i.e. HOME+HOME abandons the current edit and allows a new command to be selected from the buffer with the Up Arrow and a new blank command field with the Down Arrow.
END	Moves cursor to last position. Turns on edit mode so Up and Down arrows are disabled.
F2	Sends RUN command to console
F3	Sends LIST command to console
F4	Sends EDIT command to console
F5	Sends ESC sequents to VT100 to clear the screen. Also clears the LCDPANEL if OPTION LCDPANEL CONSOLE is set.
F10	Sends AUTOSAVE (Use CNTRL+C to abort if pressed by accident)
F11	Sends XMODEM RECEIVE
F12	Sends XMODEM SEND
Up Arrow	Recalls command from buffer in to edit buffer.
Down Arrow	Recalls command from buffer into the edit buffer.

In commandline mode there is no indication on the VT100 whether you are in INS or OVR mode.

The VT100 emulations provided by TeraTerm, Putty, GFXTerm and MMCC have been tested to support command lines that extend beyond one line on the terminal. i.e. wrap to next line as required until up to 255 characters are used. This relies on the terminal width being set to the expected size.

Setting the terminal size with OPTION DISPLAY lines[,chars] will send an ESC sequence to set the VT100 terminal to the matching size. Terraterm, Putty and MMCC respond to this sequence and set the terminal width (if the option is enabled in the terminal setup).

See this post for some more details. TBS post detailing Commandline wrapping implementation.

Variables, Expressions and Operators

Naming Conventions

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 different 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. This is different from the original Colour Maximite which allowed this.

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

Constants

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

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

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

String constants are surrounded by double quote marks ("). 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.

CONST

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

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

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

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

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

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

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

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

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

Special Characters in Strings

Special, non-printable characters can be inserted in string constants using the backslash (ie, \) as an escape symbol. To enable this facility the command OPTION ESCAPE must be placed at the start of the program. This can be used when setting the value of a string or in DATA statements containing quoted strings. For backward compatibility the use of \ as an escape character must be enabled by entering OPTION ESCAPE at the beginning of the program. OPTION ESCAPE can be entered at the command line for use on the command line, but will be reset when the RUN command is called. The use in a program requires the OPTION ESCAPE set within the program.

MMBasic is agnostic to the use of a forward slash (/) or back slash (\) as a directory separator for file operations. Internally these are all converted to a forward slash. (/). However, if using the escape option any filename that is first entered into a string variable that is then used in a file operation should use a forward

slash, as the string variable would treat any backslash as an escape character before it is passed to the file operation. Either a / or \ is acceptable if entering a literal filename directly into the file operation.

The MMEdit variable report (Program→Display Variable Report) can be used to identify lines where the escape character is used when verifying if an exsisting program can safely use OPTION ESCAPE.

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

For example, the following will print the words Hello and World on separate lines:

```
OPTION ESCAPE
PRINT "Hello\r\nWorld "
```

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 intention.

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

Arithmetic Operators

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

Shift Operators

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

Logical Operators

NOT INV	NOT will invert the logical value on the right. (eg, NOT(a=b) is a<>b) INV will perform a bitwise inversion of the value on the right. (eg, a = INV b) Both these have the highest precedence so if the value being operated on is an expression it should be surrounded by brackets. For example, IF NOT (A = 3 OR A = 8) THEN …
<> < > <= =< >= =>	Inequality, less than, greater than, less than or equal to, less than or equal to (alternative version), greater than or equal to, greater than or equal to (alternative version).
=	Equality (also used in assignment to a variable, eg implied LET).
AND OR XOR	Conjunction, disjunction, exclusive or. These are bitwise operators and can be used on 64-bit unsigned integers.

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 (eg, y% = x% AND &B1010) the interpreter will be confused if they are mixed in the same expression. So, always evaluate logical and bitwise expressions in separate expressions.

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

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

IF NOT (A = 3 OR A = 8) THEN …

String Operators

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

String comparisons respect the case of the characters (ie "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 return a floating point number. For integer division you should use the integer division operator "\".

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

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

64-bit Unsigned Integers

MMBasic on the Armmite F4 supports 64-bit signed integers. This means that there are 63 bits for holding the number and one bit (the most significant bit) which is used to indicate the sign (positive or negative). However, it is possible to use full 64-bit unsigned numbers as long as you do not do any arithmetic on the numbers.

64-bit unsigned numbers can be created using the &H, &O or &B prefixes to a number and these numbers can be stored in an integer variable. You then have a limited range of operations that you can perform on these. They are << (shift left), >> (shift right), AND (bitwise and), OR (bitwise or), XOR (bitwise exclusive or), INV

(bitwise inversion), = (equal to) and <> (not equal to). Arithmetic operators such as +, -, etc. may be confused by a 64-bit unsigned number and could return nonsense results.

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

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

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

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

Local Variables

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

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

Functions

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

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

For example:

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

Passing Arguments by Reference

If you use an ordinary variable (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 (eg, nbr1) and the defined argument (eg, a) must be the same (in the above example both default to float).

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

Passing Arguments by Value

Where you need to ensure that the argument being passed is not altered in any way, you can pass a *value* to a subroutine. When the parameter being passed is an expression, the result of that expression is passed as a value. The expression could be the result of simple maths or the return value of a function. It can also be as simple as enclosing a variable in brackets, causing the interpreter to treat it as an expression.
In this case the value could be used or even changed in the sub routine without having any effect on the passed value. The same could be achieved by assigning a *passed by reference* variable and assigning it to a local variable in the subroutine and using/changing the local variable as desired.
The advantage of *passing by value* is that the argument passed in the calling statement is safe from any changes in the called routine and additionally, saves you having to use LOCAL in the sub routine.

```
a=4
b=4
c=4
testsub((a),b,c)
print a,b,c
sub testsub(arg1,arg2,arg3)
  local k
  arg1=arg1+1
  arg2=arg2+1
  k=arg3
  k=k+1
end sub
```

```
Results in 4 5 4
```

The result for both a and c is not changed globally; a being *passed by value* and c being copied to a LOCAL variable

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 passed as additional arguments to the subroutine or function so it could correctly manipulate the array. The array is passed by reference which means that any changes made to the array within the subroutine or function will also apply to the supplied array.

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

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

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

Early Exit

There can be only one END SUB or END FUNCTION for each definition of a subroutine or function. To exit early from a subroutine (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 limited memory on microcontrollers):

- There is a fixed limit to the depth of recursion. Armmite F4 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.

Example of a Defined Function

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"

Program Initialisation, CFunctions and the Library

There are a number of features of the Armmites and Micromites that enable the advanced user to add features to MMBasic and perform special operations at startup. Most programs will not need to use these features but they are handy for the advanced user who needs more control over the Armmite F4.

Embedded C Routines - CSubs and CFunctions

It is possible to add program modules that are written in the C language to MMBasic. They are called CSubs or Cfunctions and to the BASIC program they look the same as the MMBasic built in functions and subroutines. Generally, these modules can run much faster than a BASIC program and can more easily access the special hardware features of the microcontroller. A CSub does not return a value, but can update the parameters passed to it to to return a result to MMbasic. The Armmite F4 also implements the CFunction contruction which does return a value like other functions.

The example below shows a CSub that reverses the order of a string. The CSub is loaded as part of your basic code bounded by the CSUB and END CSUB commands.

This CSub is then called from MMBasic as below.

```
Dim instring$="1234567890"
Dim outstring$
strrev instring$, outstring$
Print outstring$
End

CSub strrev
00000000
b085b480 6078af00 687b6039 60bb781b b2da68bb 701a683b 60fb2301 683ae00d
441368fb 68fa68b9 32011a8a 440a6879 701a7812 330168fb 68bb60fb 68fb1c5a
d8ec429a 461a68bb 0300f04f 46194610 46bd3714 7b04f85d bf004770
End CSub
```

The Library

The LIBRARY feature makes it possible to create BASIC functions, subroutines, embedded fonts CSubs and CFunctions and add them to MMBasic to make them permanent and part of the language. For example, you might have written a series of subroutines and functions that perform sophisticated bit manipulation; these could be stored as a library and become part of MMBasic and perform the same as other built in functions that are already part of the language. An embedded font can also be added the same way and used just like a normal font.

To install components into the library you need to write and test the routines as you would with any normal BASIC routines. When you are satisfied that they are working correctly you can use the LIBRARY SAVE command. This will transfer the routines (as many as you like) to a non visible part of flash memory where they will be available to any BASIC program but will not show when the LIST command is used and will not be deleted when a new program is loaded or NEW is used. However, the saved subroutines and functions can be called from within the main program and can even be run at the command prompt (just like a built in command or function).

Some points to note:

- Library routines act exactly like normal BASIC code and can consist of any number of subroutines, functions, embedded C routines and fonts. The only difference is that they do not show when a program is listed and are not deleted when a new program is loaded.
- Library routines can create and access global variables and are subject to the same rules as the main program – for example, respecting OPTION EXPLICIT if it is set.
- When the routines are transferred to the library MMBasic will compress them by removing comments, extra spaces, blank lines and the hex codes in embedded C routines and fonts. This makes the library space efficient, especially when loading large fonts. Following the save the program area is cleared.

- During development of a large program you may want to put already proven code into the library so that reloading of the code you are working on from MMEdit or another external editor is smaller and thus quicker.
- You can use the LIBRARY SAVE command multiple times. With each save the new contents of the program space are appended to the already existing code in the library.
- You can use line numbers in the library but you cannot use a line number on an otherwise empty line as the target for a GOTO, etc. This is because the LIBRARY SAVE command will remove any blank lines.
- You can use READ commands in the library. If you want to read from DATA statements in the library you should use the RESTORE command before the first READ command. This will reset the pointer to the library space.

To delete the routines in the library space you use the LIBRARY DELETE command. This will clear the space and return the flash memory used by the library back to the general pool used by normal programs. The only other way to delete a library is to reset MMBasic to its original configuration as described in the chapter Resetting MMBasic earlier in this manual.

As an example you could save the following into the library:

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

This would have the effect of adding a new function (called CPUSpeed) to MMBasic. You could even run it at the command prompt:

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

You can see what is in the library by using the LIBRARY LIST command which will list the contents of the library space. The MEMORY command can be used to display the amount of flash memory used by the library.

Library Implementation Details (Armmite F4)

The implementation of the library in the Armmite F407xGT uses a separate 128K of flash separate to the 128K of flash allocated to normal program memory.

The LIBRARY commands, LIBRARY SAVE, LIBRARY DELETE and LIBRARY LIST [ALL] are implemented with the same functionality as the Micromites and Picomites.
LIBRARY RESTORE is an additional command for Armmites only. It checks if library code exists and if it does sets Option.ProgFlashSize to point to the start of the library code. This allows recovery of library if the Options have been lost by an OPTION RESET.

Program Initialisation

The library can also include code that is not contained within a subroutine or function. This code (if it exists) will be run automatically before a program starts running (ie, via the RUN command). This feature can be used to initialise constants or setup MMBasic in some way. For example, if you wanted to set some constants you could include the following lines in the library code:

```
CONST TRUE = 1
CONST FALSE = 0
```

For all intents and purposes the identifiers TRUE and FALSE have been added to the language and will be available to any program that is run on the Micromite/Armmite.

MM.STARTUP

There may be a need to execute some code on initial power up, regardless of the program in main memory. Perhaps to initialise some hardware, set some options or print a custom startup banner. This can be accomplished by creating a subroutine with the name MM.STARTUP and ensuring it is included in the main program or the library. When the Armmite F4 is first powered up, RST button pushed or CPU RESTART command issued it will search for this subroutine and, if found, it will be run once. It can be used to initialise a MMBasic USER defined LCDPanel at power up:

```
SUB MM.STARTUP
  Print “I have been reset by CPU RESTART or power up”
END SUB
```

Using MM.STARTUP is similar to using the OPTION AUTORUN feature, the difference being that the AUTORUN option will cause the whole program in memory to be run from the start where MM.STARTUP will just run the code within the subroutine. The AUTORUN option and MM.STARTUP can be used together and in that case the MM.STARTUP subroutine is run first, then the program in main memory.

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.

This subroutine can be located anywhere in the main program or the library.

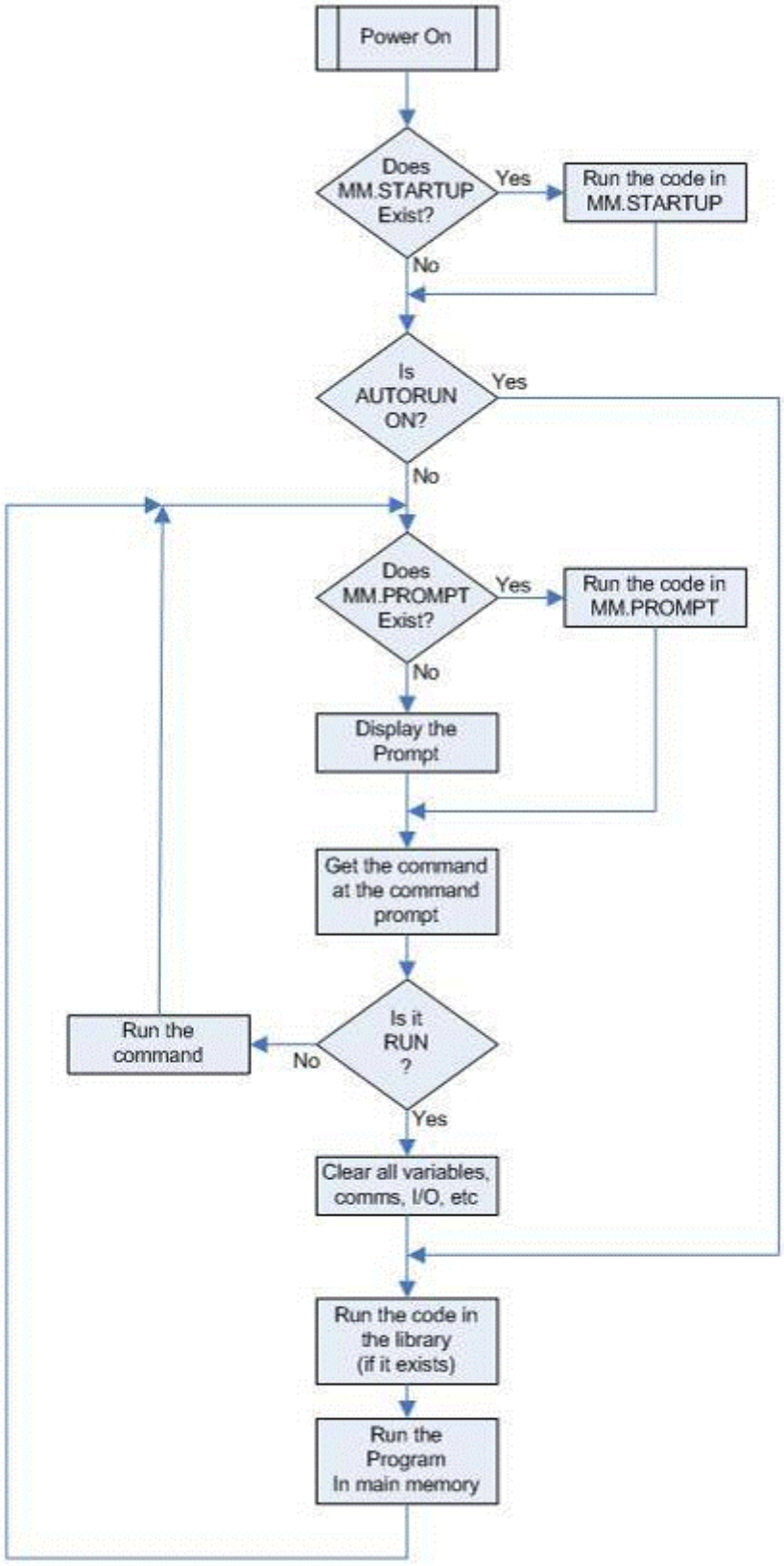
Note that MMBasic will clear all variables and I/O pin settings when a program is run so anything set in this subroutine will only be valid for commands typed at the command prompt (i.e., in immediate mode). As an example the following will display a custom prompt:

```
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, this should be used instead.

Flow Diagram

The operation of MMBasic at startup and the interaction between the special functions is best illustrated using a flow diagram. The following is a high level diagram (for example, it does not show the complications caused by the CONTINUE command) but it does place the functions of MM.STARTUP and MM.PROMPT into context. It is the same as the previous Micromites except there is no library. MM.STARTUP must be included in the program somewhere.


Power On
Does MM.STARTUP Exist?
Yes
Run the code in MM.STARTUP
No
Is AUTORUN ON?
Yes
No
Does MM.PROMPT Exist?
Yes
Run the code in MM.PROMPT
No
Display the Prompt
Get the command at the command prompt
Is it RUN ?
No
Run the command
Yes
Clear all variables, comms, I/O, etc
Run the code in the library (if it exists)
Run the Program In main memory

Memory Command

The MEMORY command is available at the command prompt. It outputs information relating to current program and ram memory usage. For the ARMMite F4 the output identifies the Saved Variables Backup SRAM memory separately from the Program memory. LIBRARY LIST can be used to see details of what is in the library.

> **MEMORY** Flash: 3K (2%) Program (62 lines) 1K (1%) 1 Embedded C Routine 1K (0%) 1 Embedded Fonts 1K (1%) Library 122K (96%) Free RAM: 1K (0%) 2 Variables 0K (0%) General 113K (100%) Free Backup SRAM (4K): 1K (1%) 2 Saved Variables (126 bytes) 3K (99%) Free	Details of Program Flash. Shows 62 lines of code using 3K. This includes the HEX listing of the CSUB and the FONT. 1K used for the binary copy of the CSUB and 1K for the binary copy of the FONT. The library occupies 1K at the end of the program memory, which is now reduced by this amount. 122K free. Total available RAM is normally 114K. (When OPTION CONTROLS is at the default value of 200). Increasing OPTION CONTROLS will reduce this. A maximum 128K is available by setting OPTION CONTROLS 0. This shows the 2 variables using 1K. 2 variables are saved with VAR SAVE and these use 1K, which is 1% of the 4K available for Saved Variables. Note: All sizes reported are to the nearest 1K, and if not zero will show a minimum value of 1K.
> **LIBRARY LIST** Sub MM.STARTUP Print "----------" Print "Connected to: " MM.Info(DEVICE),Str$(MM.Info(VERSION)) Print "----------" End Sub CSub LOG End CSub	This shows the details stored in the library. The MM.STARTUP SubRoutine. Comments and spaces are removed. The library can hold straight MMBASIC code as well as FONTs and CSUBs The CSUB LOG. This is the binary copy of the CSUB. The HEX version is no longer needed or stored anywhere once it is placed into the library.

Using the I/O pins

Digital Inputs

A digital input is the simplest type of input configuration. If the input voltage is higher than 2.3V the logic level will be true (numeric value of 1) and anything below 1.00V will be false (numeric value of 0). The inputs use a Schmitt trigger input so anything in between these levels will retain the previous logic level. Pins marked as 5V are 5V tolerant and can be directly connected to a circuit that generates up to 5.5V without the need for voltage dropping resistors.

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

```
SETPIN PA0, DIN
IF PIN(PA0) = 1 THEN PRINT "High"
```

The SETPIN command configures pin PA0 as a digital input and the PIN() function will return the value of that pin (the number 1 if the pin is high). The IF command will then execute the command after the THEN statement if the input was high. If the input pin was low the program would just continue with the next line in the program.

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

Analog Inputs

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

```
> SETPIN PA0, AIN
> PRINT PIN(PA0)
 2.345
>
```

The PIN function internally takes 10 readings, discards the highest and lowest then averages the remaining 8 'middle' readings. The ADC command uses a single sample.

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 the VREF+ pin as the reference voltage. This is tied to VCC on the STM32F404VET6 and MMBasic scales the reading by assuming that the voltage on this pin is exactly 3.3V unless

OPTION VCC voltage

is used to nominate an adjusted voltage. The actual value of VREF+ can be calculated as:

3.3 * PIN(“SREF”) / PIN(“IREF”)

and this can be used to set OPTION VCC.

The measurement of voltage is very sensitive to noise on the Analog Power and Ground pins. For accurate and repeatable voltage measurements care should be taken with the PCB design to isolate the analog circuit from the digital circuits and ensure that the Analog Power supply is as noise free as possible. Note that if the voltage on an analog input is greater than the voltage on the Analog Power pin it can cause damage or a “CPU Exception” (i.e., crash) when an attempt is made to read that voltage.

Counting Inputs

The pins marked as COUNT can be configured as counting inputs to measure frequency, period or just count pulses on the input.

For example, the following will print the frequency of the signal on pin 15:

```
> SETPIN PE3, FIN
> PRINT PIN(PE3)
110374
>
```

In this case the frequency is 110.374 kHz.

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

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

```
> SETPIN PE3, FIN, 10
> PRINT PIN(PE3)
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 Armmite will measure the number of milliseconds between sequential rising edges of the input signal. The value is updated on the low to high transition so if your signal has a period of (say) 100 seconds you should be prepared to wait that amount of time before the PIN() function will return an updated value.

The COUNTING pins can also count the number of pulses on their input. When a pin is configured as a counter (for example, `SETPIN PE3,CIN`) the counter will be reset to zero and Armmite will then count every transition from a low to high voltage. The counter can be reset to zero again by executing the SETPIN command a second time (even though the input was already configured as a counter).

The response to input pulses is very fast and the Armmite can count pulses as narrow as 10 nS . The frequency response depends on the load on the processor (i.e., the number of counting inputs and if serial or I^2C communications is used). It can be as high as 800 kHz with no other activity but is normally about 300 kHz.

Digital Outputs

All I/O pins can be configured as a standard digital output. This means that when an output pin is set to logic low it will pull its output to zero and when set high it will pull its output to 3.3V. In MMBasic this is done with the PIN command. For example, `PIN(PE3) = 0` will set pin PE3 to low while `PIN(PE3) = 1` will set it high. When operating in this mode, a pin is capable of sourcing 10 mA which is sufficient to drive a LED or other logic circuits running at 3.3V.

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

Pulse Width Modulation

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

There are three PWM controllers; the first two have three outputs and the last two to give a total of eight PWM outputs. The frequency of each controller can be independently set from 1 Hz to 20MHz and the duty cycle for each output (i.e., eight outputs) can also be independently set from between 0% and 100% with a 0.1% resolution when the frequency is below 25 kHz (above 25 kHz the resolution is 1% or better up to 250 kHz).

When the Armmite is powered up or the PWM OFF command is used the PWM outputs will be set to high impedance (they are neither off nor on). So, if you want the PWM output to be low by default (zero power in most applications) you should use a resistor to pull the output to ground when it is set to high impedance. Similarly, if you want the default to be high (full power) you should connect the resistor to 3.3V.

Interrupts

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

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

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

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

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

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

- Interrupts are only checked by MMBasic at the completion of each command, and they are not latched by 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 block interrupts for up to 200ms and it is possible for an interrupt (e.g., a button press) to occur and vanish within this window and in that case it will never be recognised.
- When inside an interrupt all other interrupts are blocked so your interrupts should be short and exit as soon as possible. For example, never use PAUSE inside an interrupt. If you have some lengthy processing to do you should simply set a flag and immediately exit the interrupt, then your main program loop can detect the flag and do whatever is required.
- The subroutine that the interrupt calls (and any other subroutines called by it) **should always be exclusive to the interrupt**. If you must call a subroutine that is also used by an interrupt you must disable the interrupt first (you can reinstate it after you have finished with the subroutine).
- Remember to disable an interrupt when you have finished needing it – background interrupts can cause strange and non-intuitive bugs.

In addition to interrupts generated by the change in state of an I/O pin, an interrupt can also be generated by other sections of MMBasic including timers and communications ports. The list of all these interrupts (in high to low priority ranking) is:

ON KEY individual
ON KEY general
COM1: Serial Port
COM2: Serial Port
COM3: Serial Port
COM4: Serial Port
GUI Int Down
GUI Int Up
WAV Finished
ADC completion
IR Receive

Keypad

Interrupt command/CSub Interrupt

I/O Pin Interrupts in order of definition

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.

Interrupts (polled) vs SETPIN CIN,PIN,FIN (hardware)

For every version of MMBasic only interrupts generated by SETPIN CIN, PIN, FIN are based on true H/W interrupts and thus will always give accurate results. All other interrupts are polled at the end of each MMBasic statement so the following applies:

EVERY version of MMbasic checks for interrupts at the end of each Basic statement with the single exception that the checks are also made during a PAUSE statement

For every version of MMBasic SETPIN n, INTx is not a true interrupt but the pin is read at the end of each Basic statement and a S/W interrupt is triggered if the pin has changed in the required manner

For every version of MMBasic Pin interrupts will be lost if the pin reverts state while a single Basic statement is running

For every version of MMBasic this is more likely to happen with commands that take longer but could happen with any command if the pin change is short enough

For every version of MMBasic this is most likely to happen with commands that communicate with H/W or move lots of data (e.g. TEMPR, graphics commands, MATH commands (where relevant))

For every version of MMBasic if this is critical you need to manage this in your code by using the timer command to see how long things take to process and find a relevant workaround

For every version of MMBasic the core Basic language is pretty much identical and the main differences are the way the firmware interacts with the various H/W peripherals BUT the basics of even this are the same (e.g.serial I/O is always interrupt driven, serial output is non-blocking, serial receive happens in the background and writes to the receive buffer etc. etc. etc.....)

Armmite F4 Deployment Considerations

This section discusses some Armmite F4 deployment considerations. These maybe relevant if you want to use the Armmite F4 for a dedicated task where it will run unattended.

Setting Option VCC

Option VCC defaults to 3.3V if not set. It is used during analogue readings as the value for the external reference. The external reference VREF+ is tied to VCC on the STM32F407VET6 board.
There are two functions that can help calibrate the ADC input to allow for when the VCC is not exactly 3.3V and for individual chip variations.

PIN(SREF) returns the measurement of the internal reference (nominally 1.21V) that the manufacturer has burned into the chip during production. This is measured at exactly 3.3V and 25 °C.

PIN(IREF) gives the value of the internal reference as measured in your environment. OPTION VCC is required to be set to the default 3.3v value during this measurement to give a valid result.

Using these together you can calculate the actual voltage the chip is seeing and hence set OPTION VCC using the following two commands.

OPTION VCC 3.3 *'Set VCC to default value incase its previously been set.*

OPTION VCC 3.3 * PIN(IREF)/PIN(SREF) *'Now set to calculated value.*

The option is not permanent and should be set in any program that does analogue measurement. It returns to the default value on a power reset or CPU RESTART.

Armmite F4 Reliance on Battery Backed Ram

The OPTIONs on the Armmite F4 are stored in battery backed ram. The ram is supplied via the CR1220 battery and the main VCC supply. When VCC is removed the CR1220 maintains the ram. If the battery cannot sustain the backup ram then the OPTIONs are lost. On restart the firmware understands this as a corruption and re-initialises the Options, i.e all Options are set to default values. Any variables saved by VAR SAVE are cleared. The AUTORUN option will be OFF, the USB Console is enabled, the default ILI9341_16 display is enabled and touch is enabled as well as all other default values. The date and time will not be set.

From firmware 5.07.01 onward the Program Memory in NOT cleared. See the next section, Running Armmite F4 without Backup Battery for how to embed the required options in the program.

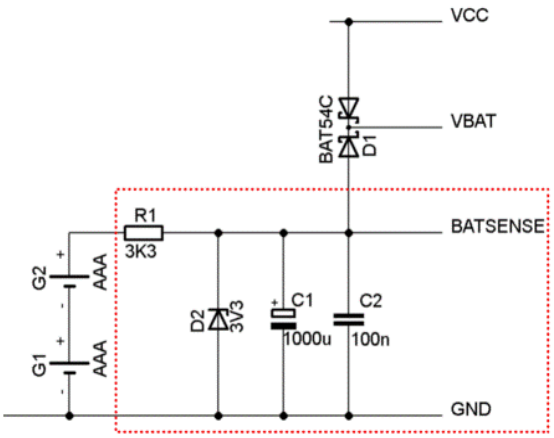
Battery Life and Monitoring VBAT

If the Armmite F4 is powered most of its life then the drain on the backup battery would be expected to be minimal and a fresh CR1220 should last a couple of years or more.

This thread on TBS discusses the life of the CR1220 battery and possible use of CR2032 and AAA batteries to extend the battery life.

The **PIN(BAT)** function returns the voltage seen at the VBAT connection. At first look this looks suitable to monitor the battery condition, but it will actually measure the higher of VCC and the battery. (less diode drop). The BAT54C diode parallels the CR1220 and VCC.

This circuit is is from the above thread and is a possible method of moitoring the battery condition using an additional analogue pin.



Embedding Configuration Options in a Program

Normally configurations options such as OPTION LCDPANEL are entered at the command prompt to configure the Armmite for any attached display panel, touch controller, etc. The drivers for these devices are enabled at startup so after using the OPTION command the Armmite will immediately restart (i.e., reboot). The user does not notice this (because it is quick) but this it is the reason why the commands should be entered at the command prompt and not in a program.

It is possible to enter these OPTIONs at the start of the program if a few simple rules are followed. These are:

- The commands must be at the start of the programs (they may be preceded by comments).
- The commands should be in the same order as they would be entered at the command prompt. For example, OPTION LCDPANEL should come before OPTION TOUCH.
- They should be thoroughly tested at the command prompt before being embedded in a program as errors may not be detected and can cause the Armmite to behave strangely.
- They must be followed by the command OPTION SAVE which will save the options and reboot the system.

When MMBasic finds these OPTION commands in a program it will update the Option in memory but will delay the saving of the options and the restart the processor until it encounters the OPTION SAVE command. All the settings will be saved to battery backed RAM (now powered from main supply even if no backup battery is available) and the processor will be restarted to enable all the required Options to be initiated at the same time. After the restart the program will be run again but this time the Option commands will be skipped over. The Option commands will be processed during a if the Options have been lost.

The calibration parameters for the touch controller can also be configured in this way. To do this you should use GUI CALIBRATE to calibrate the touch screen at the command prompt in the normal way. Then use OPTION LIST which will list the calibration parameters as something like:

```
GUI CALIBRATE 0, 252, 306, 932, 730
```

This string must be included before the OPTION SAVE so that the touch calibration settings will be saved along with the other options.

Errors in the configuration commands (for example the same I/O pin allocated to two different functions) are often detected during the reboot. However, this is not guaranteed so the configuration commands should be thoroughly tested at the command prompt before being embedded in a program.

This is an example of the sequence to configure an SPI ILI 9341 LCD Panel at startup, configure and calibrate touch, set the console to the serial console at 38400 bauds and set the backlight to 80%.

```
'These embedded OPTIONs will disable the default LCDPANEL
'Configure the ILI9341 and the Touch panel and switch to the serial
'console at 38400 and then restart with the new settings
OPTION LCDPANEL ILI9341, LANDSCAPE, PC7, PC6, PD11
OPTION TOUCH PC12, PC5
GUI CALIBRATE 0, 3756, 3901, -882, -647
BACKLIGHT 80,S
OPTION SERIAL CONSOLE ON
OPTION BAUDRATE 38400
OPTION SAVE
'The main program is then placed here
'it must include the MM.STARTUP Subroutine somewhere
SUB MM.STARTUP
  OPTION AUTORUN ON
END SUB
```

OPTION AUTORUN ON in MM.STARTUP (No Battery Backed up Options)

To have the Embedded Option run at startup requires OPTION AUTORUN ON to be set. The default is OFF, so when the default options are loaded at startup the program will not run to load the Options embedded in the program, however the MM.STARTUP routine is always run if it exists.

The MM.STARTUP subroutine is used to run the OPTION AUTORUN ON command. When the Armmite F4 is first powered up, RST button pushed or CPU RESTART command issued it will search for this subroutine and, if found, it will be run once.

This simple MM.STARTUP subroutine must be included in the program somewhere. It will cause the program in memory to execute at startup.

```
SUB MM.STARTUP
  OPTION AUTORUN ON
END SUB
```

This more complex version may be useful during development. It will auto execute the program after the initial power up, but will return to the command prompt for any further CPU RESTART or activation of the RST button. This can be useful if you want to manually start the main program during development. If your, not quite finished program does not behave you can push the RST button to get the command prompt back.

Any power reset, however will automatically restart the main program. If your, not quite finished program does not behave you can push the RST button to get the command prompt back.

```
SUB MM.STARTUP
  IF MM.INFO (RESTART)=0 THEN
    OPTION AUTORUN ON
  ELSE
   OPTION AUTORUN OFF
  END IF
END SUB
```

RTC will not maintain time if power removed

If running without a battery the RTC will be reset if power is disconnected. It would need to be manually reset.

If you don't have a battery connected you could use command below to see how long since last power cycle.

```
? DATETIME$(NOW)
```

Electrical Characteristics

Power Supply

Voltage range:	2.3 to 3.6V (3.3V nominal). Absolute maximum 4.0V.
Current draw:	70 mA without LCD.
Current in sleep:	40 µA (plus current draw from the I/O pins).

Digital Inputs

Logic Low:	0 to 1.0V
Logic High:	2.5V to 3.3V on normal pins 2.5V to 5.5V on pins rated at 5V
Input Impedance:	>1 MΩ. All digital inputs are Schmitt Trigger buffered.
Frequency Response:	Up to 300 kHz (pulse width 20 nS or more) on the counting inputs.

Analog Inputs

Voltage Range:	0 to 3.3V
Accuracy:	Analog measurements are referenced to VREF+ which is connected to the supply voltage. If the supply voltage is precisely 3.3V the typical accuracy of readings will be ±1%. (See OPTION VCC to adjust voltage to match actual voltage)
Input Impedance:	>1 MΩ (for accurate readings the source impedance should be < 5K)

Digital Outputs

Typical current draw or sink ability on any I/O pin:	10 mA
Absolute maximum current draw or sink on any I/O pin:	25 mA
Absolute maximum current draw or sink for all I/O pins combined:	150 mA
Maximum open collector voltage:	5.5V

Timing Accuracy

All timing functions (the timer, tick interrupts, PWM frequency, baud rate, etc.) are dependent on the internal clock. The Armmite is crystal controlled so accuracy is expected to be worst case 50ppm (0.005%)

PWM Output

Frequency range:	1 Hz to 20MHz
Duty cycle:	0% to 100% with 0.1% resolution below 25 kHz

Serial Communications Ports

Console:	Default 115200 baud. Range is 2400 bps to 921600 bps
COM ports	Default 9600 baud. Range is 2400 bps to 1843200bps

The reliability of the higher baudrates will depend on length of cable etc. The 921600bps for the console is nominated based on it being the highest in the TeraTerm drop down. It may well be 1843200bps as well.

Other Communications Ports

SPI	10 Hz to 10 MHz
I^2C	10kHz to 400 kHz.
1-Wire:	Fixed at 15 kHz.

Flash Endurance

Over 10,000 erase/write cycles.

Every program save incurs one erase/write cycle. In a normal program development, it is highly unlikely that more than a few hundred program saves would be required.

Saved variables (VAR SAVE command) and configuration options (the OPTION command) are stored in the RTC battery backed up RAM and DO NOT use or impact the life of the flash.

Audio Output

The Armmite F4 can play WAV and FLAC files from the SD card, and generate precise sine wave tones. All these are outputted on the DAC pins PA5 and PA4. The STM32 chip includes its own DAC (digital to analog converter) so an output filter network is not needed.

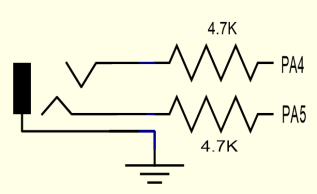
The Armmite F4 has no audio socket connect to the board as supplied. If you connected to PA4 and PA5 then PA4 is the right channel, and PA5 is the left channel with reference to the Armite ground. The signal level at full volume is about 1V RMS (approx 3V peak to peak). The output is high impedance suitable for feeding into an amplifier. **It cannot directly drive a loudspeaker, headphones or any low impedance load and might be damaged if that was attempted.**

This thread on TBS forum discusses possible circuits to drive a headset.

https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=13631

MMBasic can generate audio in several formats ranging from simple sine wave tones through to playing WAV and FLAC, audio files. (MP3 is not supported because of high processor resources required to decode).

If adding an audio socket, it's a good idea to add 4.7K resistors in series with the connections to PA4 and PA5 to protect the DAC against short circuits as the plug is inserted into the socket.



Playing WAV and FLAC Files

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

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

```
PLAY WAV file$ [, interrupt_on_completion]
PLAY FLAC file$ [, interrupt_on_completion]
```

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

Most variations in encoding are supported (see the PLAY command in the command listing for the details).

The WAV/FLAC files can be 8 or 16 bit encoded, and samples rates can be 8,16 or 44.1kHz.

To convert a file to this format a program or website such as http://audio.online-convert.com/convert-to-wav can be used (for this website set 8-bit or 16-bit resolution, set sampling rate to 8000 or 16000 or 44100, set "Audio Channels" to stereo. Click "Normalise audio". Set PCM unsigned 8-bit in ADVANCED OPTIONS).

Generating Sine Waves

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

The syntax of the command is:

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

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

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

The frequency can be from 1 Hz to 20 KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command.

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.

Changing the volume via the software will slightly degrade the output quality, but probably not significantly for most cases. i.e. at maximum volume the audio produced by the DAC is using 4096 steps/levels to generate the audio wave. At 50% volume this would only be 2048 steps/levels. Playing at full volume with an analogue volume control on the output would ensure the highest quality.

Special Device Support

To make it easier for a program to interact with the external world the MMBasic firmware of the Armmite F4 includes specific drivers for a number of common peripheral devices.

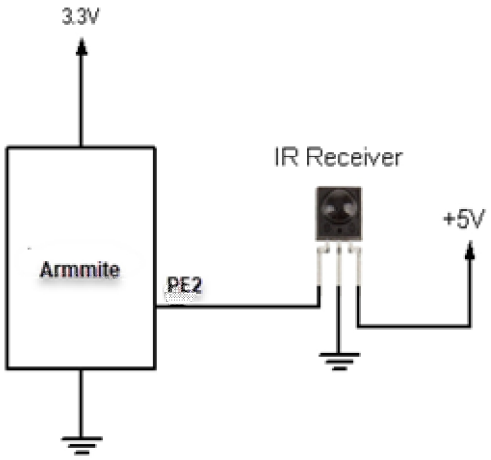
These are:

- Infrared remote control receiver and transmitter
- The DS18B20 temperature sensor and DHT22 temperature/humidity sensor
- LCD display modules
- Numeric keypads
- Ultrasonic distance sensor

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 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 connected to the IR pin (pin PE2 on the Armmite F4) as illustrated in the diagram. The IR receiver will sense the IR light, demodulate the signal and present it as a TTL voltage level signal to this pin. Setup of the I/O pin is automatically done by the IR command.

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

Sony remotes use a 40 kHz modulation 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.

To setup the decoder you use the command:

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

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

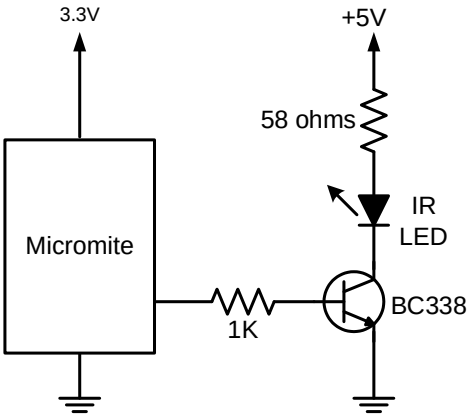
This is an example of using the IR decoder:

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

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

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

Infrared Remote Control Transmitter

Using the IRSEND command you can transmit a 12 bit Sony infrared remote control signal. This is intended for Micromite/Armmite to Micromite/Armmite 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 IRSEND command is available on the Armmite F4.



The circuit on the right illustrates what is required. The transistor is used to drive the infrared LED because the output of the Armmite is limited to less than 25mA. This circuit provides about 50 mA to the LED.

To send a signal you use the command:

```
IRSEND pin, dev, key
```

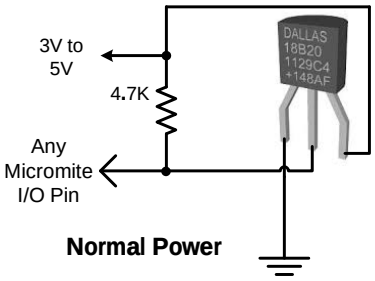
Where pin is the I/O pin used, dev is the device code to send and key is the key code. Any I/O pin on the Armmite can be used and you do not have to set it up beforehand (IRSEND will automatically do that).

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 Armmites/Micromites.

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 3V to 5V supply or it can operate on parasitic power from the Armmite/Micromite 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.

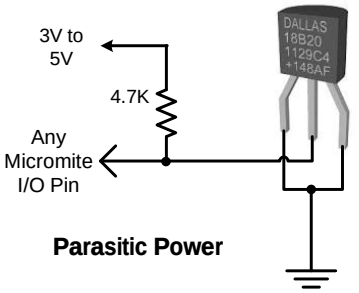


To get the current temperature you just use the TEMPR() function in an expression. For example:

```
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.

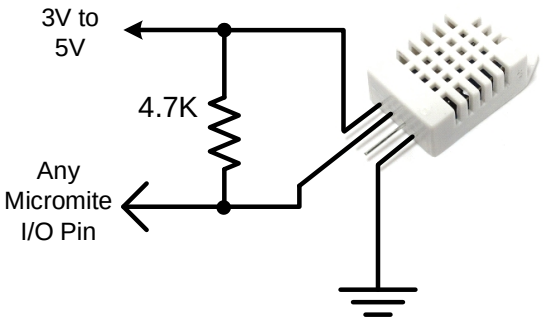
For example:

```
TEMPR START PE0
< do other tasks >
PRINT "Temperature: " TEMPR(PE0)
```

Measuring Humidity and Temperature

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

The DHT22 can be powered from 3.3V or 5V (5V is recommended) and it should have a pullup resistor on the data line as shown. This is suitable for long cable runs (up to 20 meters) but for short runs the resistor can be omitted as the Armmite also provides an internal weak pullup. To get the temperature or humidity you use the DEVICE HUMID command with arguments as follows:



DEVICE HUMID pin, tVar, hVar[,version]

Valid codes for version are:
1 = DHT11
0 or omitted = DHT22

Where 'pin' is the I/O pin to which the sensor is connected. You can use any I/O pin but if the DHT22 is powered from 5V it must be 5V capable. The I/O pin will be automatically configured by MMBasic.

'tVar' is a floating point variable in which the temperature is returned and 'hVar' is a second variable for the humidity. Both of these variables must be declared first as floats (using DIM). The temperature is returned as degrees C with a resolution of one decimal place (eg, 23.4) and the humidity is returned as a percentage relative humidity (eg, 54.3).

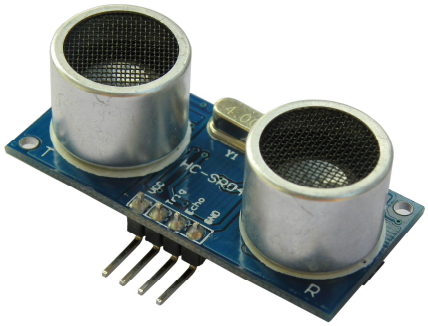
For example:

```
DIM FLOAT temp, humidity
DEVICE HUMID pin, 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 Armmite you use the DISTANCE function as follows:

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

Where trig is the I/O pin connected to the "trig" input of the sensor and echo is the pin connected the "echo" output of the sensor. You can also use 3-pin devices and in that case only one pin number is specified.

The value returned is the distance in centimetres to the target. The I/O pins are automatically configured by this function but note that they should be 5V capable as the HC-SR04 is a 5V device.

LCD Display

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

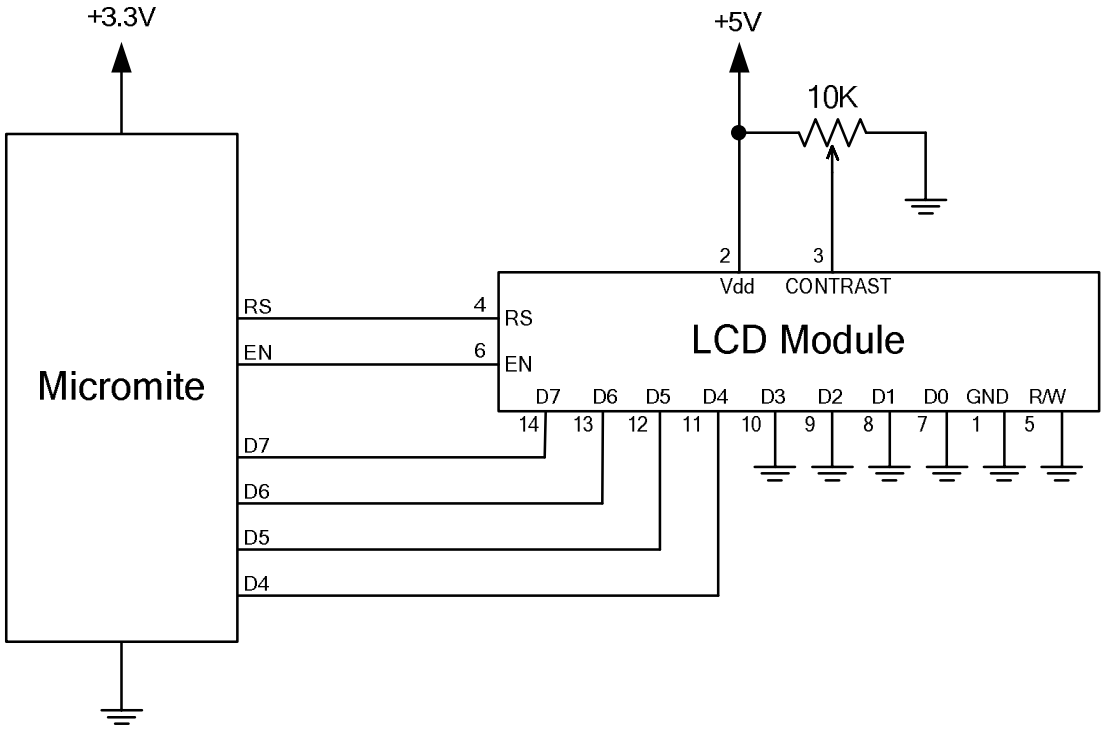
This command will work with LCD modules that use the KS0066, HD44780 or SPLC780 controller chip and have 1, 2 or 4 lines. Typical displays include the LCD16X2 (futurlec.com), the Z7001 (altronics.com.au) and the QP5512 (jaycar.com.au). eBay is another good source where prices can range from $10 to $50.

To setup the display you use the LCD INIT command:

```
DEVICE 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 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 for a Micromite. A display can be setup on an Armmite using appropriate I/O pins.



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

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

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

The following shows a typical usage where d4 to d7 are connected to pins 2 to 4 on a Micromite, rs is connected to pin 23 and en to pin 24.

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

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 an Armmite/Micromite based system. They support either a 4x3 keypad or a 4x4 keypad and the monitoring and decoding of key presses is done in the background. When a key press is detected an interrupt will be issued where the program can deal with it.

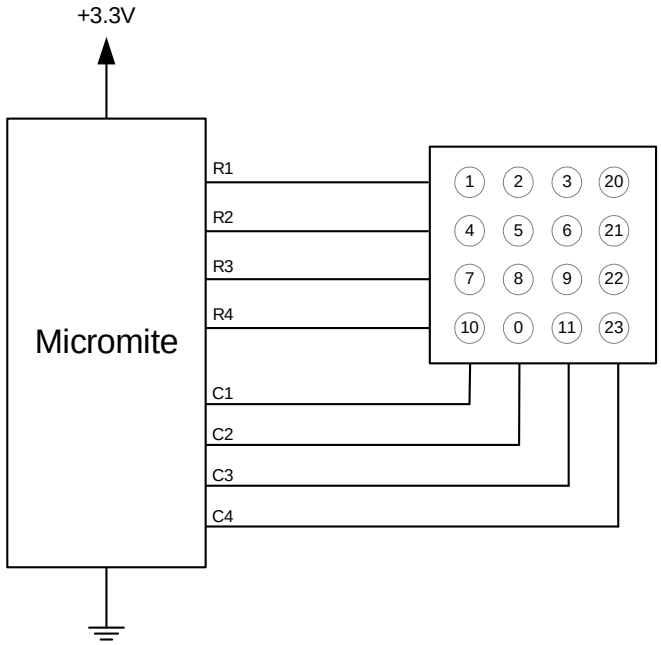
Examples of a 4x3 keypad and a 4x4 keypad are the Altronics S5381 and S5383 (go to www.altronics.com).

To enable the keypad feature you use the command:

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

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

Any I/O pins on the Micromite can be used and you do not have to set them up beforehand, the KEYPAD command will automatically do that for you. The example below is for a Micromite. An Armmite can be used if appropriate I/O pins are selected.



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

For example:

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

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

WS2812 and SK6812 RGBW Support

The Armmite F4 has built in support for the WS2812 multicolour LED chip. This chip needs a very specific timing to work properly and with the 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:

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

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

O = original WS2812

B = WS2812B

S = SK6812

W=SK6812 RGBW

‘nbr’ is the number of LEDS in the chain. (1-256)

As an example:

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

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

CAN Support

The Armmite F4 MMBasic exposes the native support for CAN provided by the STM32F407 chip as the MMBasic CAN command.

A CAN transceiver such as the SN65HVD230 CAN Transceiver is required to interface the actual physical CAN bus. It connects to 3.3V and GND plus the nominated CAN Tx and CAN Rx pins on the ARMmite. The CANL and CANH connectors go to the physical CAN bus. You can exercise the CAN command in loopback mode without having the transceiver attached.

This primer has a lot of good information if you are new to CAN.

The following Commands provided in MMBasic access to the CAN.

These common parameters apply to their use in the following commands.

Parameter	Description
id, id1, id2	These refer to either an 11bit (0-7FF) or 29bit (0-1FFFFFFFF) identifier for a CAN Frame. id2 may also refer to a bit mask in the CAN FILER command.
eid	0 indicates 11bit id(STDID) is used/expected, 1 indicates a 29bit id (EXTID) is used/expected
rtr	Indicates a Remote Frame. i.e. dlc is 0 and no data. These are not used much any more.
dlc	Indicates the size of the data and the CAN message. (0-8)
msg	contains 8 bytes of data, only those indicated by dlc are valid, the rest are set to 00.
ret	Is the return value for the command.

CAN OPEN index,speed,mode

This command opens the CAN interface indicated by *index*. This configures the CAN based of some predefined pin and mode allocations as shown below. The speed indicates the desired CAN speed. Allowed values are 0(special case),125000,250000,500000 and 1000000. The sampling point uses the default 87.5%.

Index	Chip	CAN Rx	CAN Tx	Notes
1	64	PB8/61	PB9/62	Shares PWM 2A and 2B
	100	PB8/95	PB9/96	Shares PWM 2B and 2C
	144	PB8/139	PB9/140	Shares PWM 2B and 2C
2	64	n/a	n/a	n/a
	100	PD0/81	PD1/82	Shares FSMC D2 and D3 pins
	144	PD0/114	PD1/115	Shares FSMC D2 and D3 pins

The ***speed*** value indicates the desired CAN speed. Allowed values are 0(special case),125000,250000,500000 and 1000000. The sampling point is set to 87.5%. The speed is not directly given to the CAN hardware, a set of parameters is provided that will achieve the desired CAN speed and sample point. This table shows the values.

```
/**** Clock parameters at 85.75 to 87.5% sample point *
      Speed     Pre     Max     Seg1    Seg2  JSW  Sample
        KHz     Scale   tq                           Point
        125      21     16      13      2     1    87.5
        250      12     14      11      2     1    85.7
        500       6     14      11      2     1    85.7
        1000      3     14      11      2     1    85.7
 ************************************************************
```

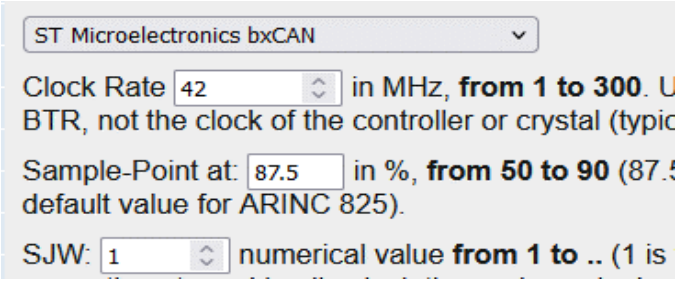
The CAN OPEN command also sets a default filter 0, that accepts all messages into the FIFO0 buffer. This will start as soon as a CAN START command is issued. If you want to apply alternate filters to restrict which messages are accepted you enter them after the CAN OPEN command and before the CAN START command.

The ***mode*** value set the operational mode of the CAN. In Normal mode the CAN sends and reads messages via the CAN bus using the attached CAN transceiver. It does not see its own messages. In Loop Back mode the CAN sees its own messages as if they were received from the CAN bus. The CAN transceiver does not need to be attached. This mode is useful for testing/developing and checking your filters work as expected. In this mode no messages are placed on or received from the CAN bus if it is attached.

Mode	Operation Mode	Notes
0	Normal	Normal operation
1	Internal Loop Back	This mode can be used for a loopback test. No data is placed on or received from the physical CAN Bus, however internally the transmitted data is also seen as received data. The transceiver does not need to be attached.
2	External Loop Back	This mode can be used for a loopback test. The transmitted data is also placed on the CAN Bus and seen by other nodes if the transceiver is attached and connected to the CAN Bus.

The following long forms of the CAN OPEN command used when the speed is set to 0, allows the user to provide the prescaler, seg1, seg2 and sjw (SyncJumpWidth) values that allow the detailed selection of the speed and sampling point. The base clock speed is 42MHz and this online calculator allows the determination of the values for various CAN speeds and sampling points. Use the calculator here to determine the values
http://www.bittiming.can-wiki.info/

Select ST Microelectronics bxCAN and clock rate of 42MHz and the desired sample-point to generate a table with the values for the various CAN speeds. SJW is normally always set a 1.



CAN OPEN index,0, prescaler, seg1, seg2, sjw

This long form of the CAN OPEN command used when the speed is set to 0 allows the user to provide the prescaler, segment1, segment2 and SJW (SyncJumpWidth) values that allow the detailed selection of the speed and sampling point.

CAN FILTER index, eid, type, config, id1, id2

Filters are used to let the CAN hardware do the work to discard message you are not interested in, without using the MMBasic resources to analyse them. Filters can only be added or changed when the CAN is stopped so a CAN STOP is required if you want to change filters.
index is the filter number 0-27, these are applied in order.
type is the filter type.
0 is a classic filter with id1 containing the filter ID and id2 containing the filter mask.
1 is a filter containing two IDs (id1, id2) to be matched.
2 is a filter the same as type 0 except that eid is ignored and both STDID and EXTID are accepted.
config value
0 will disable the filter, i.e. its ignored.
1 will accept a matched message into FIFO0.
2 will accept a matched message into FIFO1.
e.g. The following filter will accept all messages into FIFO0.
CAN FILTER 0, 0, 2, 1, &H0, &H0

CAN START

After the CAN is configured and the filters are set the CAN START command tells the CAN to start accepting messages and to accept messages to send. The CAN START command MUST be issued to start operation.

CAN STOP

The CAN STOP command can be issued to stop sending and receiving messages. You could issue this command if you wanted to pause the reception of messages. Also required before changing the filters.

CAN SEND id, eid, rtr, dlc, msg, ret

This command places a message in the next available transmit buffer and orders it to be sent. There are three transmit buffers. If no free buffer is available then the ***ret*** value is 1. See the example linked from the Armmite F4 FotS page where an internal MMBasic Tx FIFO buffer is created to throttle transmissions so the three transmit buffers are not overwelmed if multiple messages are send at once.

CAN READ fifo, id, eid, rtr, dlc, msg, fmi, ret

This command reads a message from either Rx FIFO0 or Rx FIFO1 depending on the value of *fifo* and populates *id, eid, rtr. dlc ,msg* and *fmi* with the details from the received CAN Frame. The Rx FIFO buffers have three stages.
ret is 0 if no messages are available else the number of messages available. The application needs to continually call this command in the main DO:LOOP as this implementation does not use interrupts and expects that messages to be polled. Only one message is retrieved for each CAN READ command. See the example linked from the Armmite F4 FotS page where an internal MMBasic Rx FIFO buffer is created to hold messages until they are processed, this allows the CAN Rx buffers to be quickly read to ensure that no messages are lost if a burst of messages are received.

fmi is the Filter Match Index. It tells you which filter matched the message, but is not the actual filter number. The *fmi* is number from 0 for each FIFO. The first filter that accepts a message it FIFO1 is 0. A filter of type 1 (i.e. contains 2 IDs) will count as 2 *fmi* values, one for each of id1 and id2. Using the *fmi* is an alternate method of analysing the incoming message. i.e. know which filter matched it to see what to do with it. The alternative is just look at the ID and decide what to do with it.

CAN CLOSE

This command closes the CAN and releases the pins.

If you are implementing a CAN system it is useful to be able to uniquely identify a node e.g. serial no. MM.INFO(ID) will return a unique 96bit ID for the MMBASIC chip as a hexidecimal string. This, whilst guaranteed to be unique is too long to easily pass in a single CAN message.

MM.INFO(ID48) will return an integer with a 48 bit hash of this 96 bit ID which while not guaranteed to be unique will most probably be unique within any MMBasic systems you have. This can be placed in 6 bytes of an CAN message to identify a node and have two bytes left for the return of an allocated shorter node number to be used within the system.

SD Card Support

The SD card is always enabled in the Armmite F4 firmware and no configuration is necessary.

The Armmite F4 has full support for SD cards. This includes opening files for reading, writing or random access and loading and saving programs and the files created can also be read/written on personal computers running Windows, Linux or the Mac operating system.

It is recommended to use SD cards up to 32GB, formatted as FAT32 with standard 512byte block size. Small capacity cards may not be reliable so the smallest recommended size is 8GB formatted as FAT32.

The Armmite F4 does support exFAT for cards over 32Gb. It does not support non standard block sizes (Not 512bytes). However, The FATFS implementation for exFAT is not complete and does not allow thing like relative addressing (../file). Also exFAT is much slower than FAT32 so you are recommended to use cards of 32GB or less formatted with the standard 512byte block size.

In the following note that:

- The filename can be a string expression, variable or constant. If it is a constant the string must be quoted (eg, KILL "MYPROG.BAS").
- Long file/directory names are supported in addition to the old 8.3 format.
- The maximum file/path length is 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. (ie, OPEN "/dir1/dir2/file.txt" FOR …).
- Forward slashes or back slashes are valid in paths between directories. Eg /dir/file.txt or \dir\file.txt.
- The current MMBasic time is used for file create and last access times.
- Up to ten files can be simultaneously open.
- Except for INPUT, LINE INPUT and PRINT the # in #fnbr is optional and may be omitted.

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

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

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

□ 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 two 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.

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

Load and Save Image

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

□ LOAD IMAGE fname$ [, startx, starty]
Load a BMP file and display it on the LCD screen at startx, starty. (these default to the top left corner of the display if not specified).

- SAVE IMAGE fname$ [, x, y, w, h]
  Save the current LCD screen image as a BMP file. This will save the image as a 24-bit true colour BMP file (the extension .BMP) will be added if an extension is not supplied. [x, y, w, h] define the area to be saved. If omitted, the entire screen is saved.

Load and Save Data

Memory content can be loaded from or saved to the SD card using two commands.

- SAVE DATA fname$, address, size
  Save memory *size* bytes starting memory *address* to *filename$* as binary data.

- LOAD DATA fname$, address
  Load binary data into the memory at *address*

File and Directory Management

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.

- NAME fnameold$ AS fnamenew$
  Renames 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.

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.

- 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.

XModem Transfer

In addition to the standard method of XModem transfer which copies to or from the program memory the Armmite F4 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 "PRBAS") In the case of receiving a file, any file on the SD card with the same name will be automatically overwritten.

Example of Sequential I/O

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

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

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

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

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

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

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

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

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

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

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

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

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

Random File I/O

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

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

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

```
SEEK #1, 257
```

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

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

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

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

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

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

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

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

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

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

W25Q16 Flash Support

The Armmite STM32F407VGT6 and some other boards have a 2Meg W25Q16 SPI Flash chip on board. This is available to the user by accessing with SPI commands. See Appendix E W25Q Windbond

Display Panels

The ArmmiteF4 supports a number of 16 bit parallel displays as well as a number of SPI displays.

16 Bit Parallel Interface LCD Panels

The Armmite F4 supports a number of LCD Panels with 16 bit parallel interface. These are preferred due to their increased speed, and some are not much difference in price to the SPI screens. The ILI9341_P16 that can be ordered with the STM32F407VET6 plugs directly into the FSMC connector without any need for additional wiring.

The supported panels are:

- ILI9341 P16

Available with a matching connector, but also available with a 40pin connector which needs an adaptor. The Armmite F4 initially starts with this controller and its touch panel fully configured by default.

OPTION LCDPANEL DISABLE is initially required before setting up an alternate display.

- SSD1963 4” 5” 7” 8” 9”

These are high quality, have been available long term and need an adaptor board.

- IPS_4_16 800*480 IPS Displays.

They are cheaper than the SSD1963 and can be a good choice. There are two types of this display which look almost identical. They have either the OTM8009A or NT35510 chip. These are both handled as the IPS_4_16 display type and the driver will determine which one is in use and use the appropriate code. They have a 34 pin connector and require an adaptor.

- ILI9486 P16

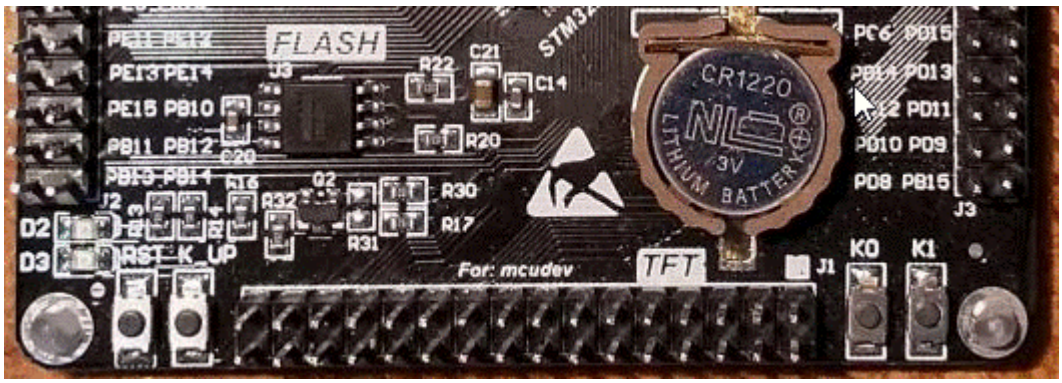
Another option with 480*320 resolution, and the 34 pin connector with an adaptor required.

Pin out for FSMC connector.

The pinout below is for the FMSC LCDPANEL connection at the end of the board. This is viewed from the top of the board. The pin numbers and function as shown below. You will need this if you want to make an adaptor board.

PEN-IRQ is the same as T-IRQ. The LCD-BL pin is controlled by the BACKLIGHT command. The T-CLK, MOSI and MISO pins are SPI2 and can be used to connect to an SPI LCD Panel. Touch uses the same SPI2, PEN-IRQ and T-CS.

See STM32F407VET6 Board FSMC connectors. for details of a possible different FSMC pinout in later boards.



FSMC LCD Connector – Top View

31	29	27	25	23	21	19	17	15	13	11	9	7	5	3	1
3.3V	n/c	Pen-IRQ	MOSI	T-CLK	DC	RD	B1	B3	B5	B7	B9	B11	B13	B15	GND
GND	GND	LCD-BL	MISO	T-CS	CS	WR	B0	B2	B4	B6	B8	B10	B12	B14	RST
32	30	28	26	24	22	20	18	16	14	12	10	8	6	4	2

FSMC Pins available to MMBasic or SPI LCD Panels.

The FSMC B0-B15 data pins, DC, CS, RD, WR, RST pins can be used if no 16bit parallel LCD Panel is configured They can be used from MMBasic and can be allocated for use by SPI LCD Panels.

SPI LCD panels need RST, D/C and CS pins, you can allocate them from any available DIN-DOUT pins.

SSD1963 Power Considerations

For 4.3", 5", 7" versions make sure the backlight control jumper on the display is set to 1963_PWM. You can then leave the LED_A pin disconnected but it is benign if it is wired to 3.3V or the LCD_BL pin on the STM32F407 (PB1)
For 4.3" and 5" displays only the 3.3V supply is needed.

For 7" displays the 5V pin on the display should be connected. I found that my STM32F407 board was able to supply adequate power but this will depend on the USB port on the computer used. The USB enumeration code now asks for 500mA which is adequate for a 7" display (400mA) + the STM32F407 board itself.

For 9" displays using the Ritech adapter I needed to use an external 5V supply connected to the 5V pin.
The 9" display uses the same driver as the 8" panel. i.e.
OPTION LCDPANEL SSD1963_8_16 , orientation

Backlight Control – BACKLIGHT (0-100)

The backlight brightness is set based on the Option DefaultBrightness setting. It is defaulted to 50%. The value stored in Option DefaultBrightness is changed by an optional parameter on the BACKLIGHT command. The Option DefaultBrightness will show in Option List if any display is configured and is not at the default 50%.

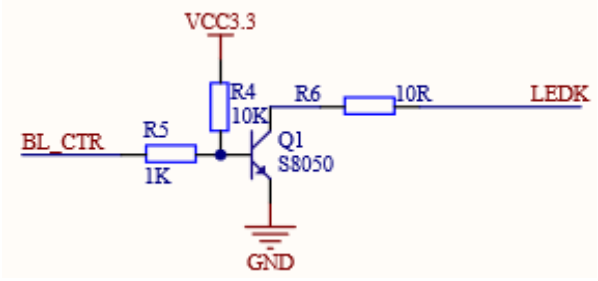
The brightness of the backlight on LCD panels can be controlled with the BACKLIGHT command:

```
BACKLIGHT percent [,DEFAULT|,REVERSE]
```

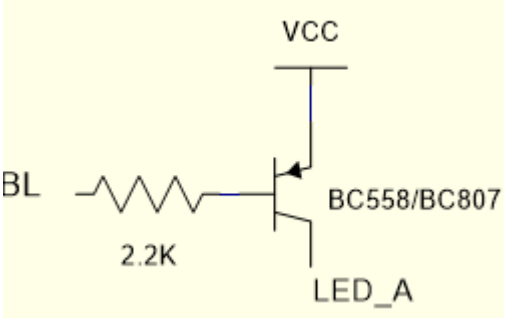
Where 'percent' is the degree of brightness ranging from 0 (fully off) to 100 (full brightness). This can be changed as often as required and makes a huge difference to the power requirements of the display. For example, a brightness of 50% will halve the current consumption (compared to 100%) while only making a small difference to the perceived visual brightness. The SSD1963 backlight jumpers should be set to use its own PWM as detailed below. The backlight command optional parameter which will cause the default setting (i.e. OPTION DefaultBrightness) to be also updated to the new value in the saved Options. BACKLIGHT 50,S will cause the default brightness to be set to 50% and this will be used when the device is restarted or powered on. BACKLIGHT 50,R will also set the default brightness to 50%, it also signals that the LCD Panel requires the signal to be sent a in reverse order in order for it to respond as 0 (fully off) and 100 (fully on). The Default Brightness is by default set at 50%. This ensures that the screen is a least visible if you are not sure which type you have.

The BACKLIGHT command supports other LCD panels supported on the Armmite F4 by controlling a PWM signal on the BL connector, with brightness ranging from 0 (fully off) to 100 (full brightness)

This may be reversed depending on how the driver circuit for the LED is implemented.

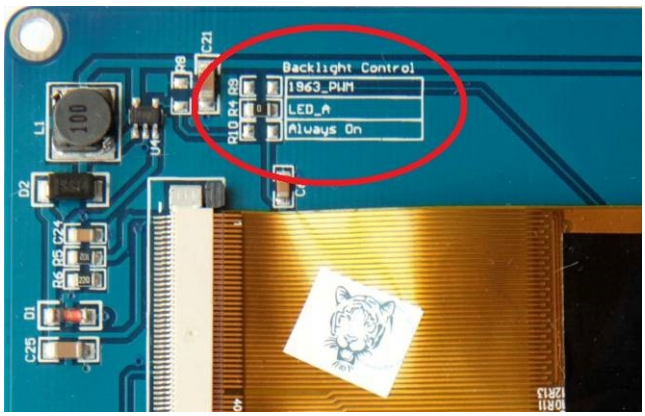


Typical LED driver built into displays gives brightness as 0(off)-100(fully on).



Simple driver to allow SPI ILI9341 LED-A pin to be driven by Backlight command. 0(off) to 100(fully on). VCC is 3.3V (SPI ILI9341 has no built in driver)

The SSD1963 based LCD panels have three pairs of solder pads on the PCB which are grouped under the heading "Backlight Control" as illustrated on the right. Normally the pair marked "LED-A" are shorted together with a zero ohm resistor and this allows control of the backlight's brightness with a PWM (pulse width modulated) signal on the LED-A pin of the display panel's main connector.



The Armmite F4 expects the SSD1963 controller to be set to use the SSD1963 for brightness control. The zero ohm resistor should be removed from the pair marked "LED-A" and used to short the nearby pair of solder pads marked "1963-PWM". The Armmite F4 can then control the brightness via the SSD1963 controller.

SPI Based LCD Panels

The standard Armmite F4 includes support for colour LCD display panels using the ILI9341 controller and an SPI interface. These have a 240x320 pixel colour TFT display, come in a variety of sizes (2.2", 2.4" and 2.8") and are low cost (typically US$8).

On eBay you can find suitable displays by searching for the controller name (ILI9341).

There are many similar displays on the market however some have subtle differences that could prevent them from working with the Armmite. MMBasic was tested with the displays illustrated below so, if you wish to guarantee success make sure your display matches the photographs and the specifications listed below.

The ILI9341 based displays use an SPI interface and have the following basic specifications:

- A 2.2, 2.4 or 2.8 inch display
- Resolution of 240 x 320 pixels and a colour depth of 262K/65K
- A ILI9341 controller with a SPI serial interface

The display illustrated also has a touch sensitive facility which is fully supported by MMBasic. There are versions of this display without the touch controller (the 16-pin IC on the bottom right of the PCB) but there is not much point in purchasing these as the price difference is small.

Connecting SPI Based LCD Panels

The SPI based display controllers share the SPI2 interface with the touch controller (if present).

The following table lists the connections required between the LCD display board and the Armmite

ILI9341 Display	Description	Connector	Pin
T_IRQ	Touch Interrupt	FSMC	27 (PB12)
T_DO	Touch Data Out (MISO)	FSMC	26
T_DIN	Touch Data In (MOSI)	FSMC	25
T_CS	Touch Chip Select	FSMC	24 (PC5)
T_CLK	Touch SPI Clock	FSMC	23
SDO (MISO)	Display Data Out (MISO)	FSMC	26
LED	The FSMC LCD-BL can be used to drive the backlight on thse displays. LCD-BL can only drive at logic levels. If the LCD panel does not have a driver transistor built in (none of the known SPI LCDs do), you ***cannot connect to the FSMC LCD-BL*** pin, ***unless*** you provide a driving circuit. If you can drive the backlight with logic level, then the BACKLIGHT command and the LCD-BL pin can be used to control the LCD's backlight. Otherwise the LCD backlight should be connected to VCC via a suitable resistor to give a satisfactory backlight.		
SCK	Display SPI Clock	FSMC	23
SDI (MOSI)	Display Data In (MOSI)	FSMC	25

ILI9341 Display	Description	Connector	Pin
D/C	Display Data/Command Control	Configurable	
RESET	Display Reset (when pulled low)	Configurable	
CS	Display Chip Select	Configurable - Optional if Touch not used.	
GND	Ground		
VCC	VCC can be either 5V or 3.3V If connected to 3.3v J1 on the back of the LCD can be shorted to bypass the 5v to 3.3v regulator..supply (the controller draws less than 10 mA)		

Note: Be careful to ground yourself when handling the display as the ILI9341 controller is sensitive to static discharge and can be easily destroyed.

Where a Armmite connection is listed as "configurable" the specific pin should be specified with the OPTION LCDPANEL or OPTION TOUCH commands (see below).

The SPI LCDs generally expose the LED-A which is the Anode to the backlight LEDs. The backlight power (the LED connection) can be supplied from the main 5V supply via a current limiting resistor. A typical value for this resistor is 18Ω which will result in a LED current of about 63 mA. The value of this resistor can be varied to reduce the power consumption or to provide a brighter display. If a suitable driver circuit as shown above in the Backlight Control section is used then the backlight can be controlled via the BACKLIGHT command.

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

Supported SPI Panels

- ILI9481 SPI based 480*320 SPI touch controller
- ILI9488 SPI based 480*320 SPI touch controller
- ILI9341 SPI based 320*240 2.2", 2.4" and 2.8" panels using the ILI9341 controller
- ST7735S SPI based 160*80 IPS display
- GC9A01 SPI based round 240*240 IPS display
- ST7789 SPI based 240*240 IPS display
- ST7735 SPI based 160*128 display

ILI9481 Viewed from underneath

26	24	22	20	18	16	14	12	10	8	6	4	2
T-CS	CS	RST		DC						GND		5V
	CLK	MISO	MOSI				T- IRQ					
25	23	21	19	17	15	13	11	9	7	5	3	1

Socket to accept the ILI9481 LCD viewed from top

2	4	6	8	10	12	14	16	18	20	22	24	26
5V		GND						DC		RST	CS	T-CS
					T- IRQ				MOSI	MISO	CLK	
1	3	5	7	9	11	13	15	17	19	21	23	25

The ILI9488 display may have issues when the LCD SDO(MISO) pin is connected. (The LCD SDO does NOT tristate when CS is high and interferes with the Touch T_DO). It is only needed if BLIT or transparent text are used, otherwise it can be left disconnected. Touch shares the SPI2 port with the LCD SDO. Connecting the LCD SDO pin via a 680ohm resistor has been known to allow both to work together.

Configuring MMBasic for SPI Displays

To use the SPI displays MMBasic must be configured using the OPTION LCDPANEL command which is normally entered at the command prompt. Every time the Armmite is restarted MMBasic will automatically initialise the display. This command can also be embedded in a program with certain conditions – see the section Running Armmite F4 without Backup Battery for more details.

The syntax is:

```
OPTION LCDPANEL controller, orientation, D/C pin, reset pin [,CS pin][,INVERT]
```

Where:

'controller' can be either ILI9341, ILI9481, ILI9488 or ST7789

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

'D/C pin' and 'reset pin' are the I/O pins to be used for these functions. Any free pin can be used.

'CS pin' can also be any free I/O pin and is optional if a touch controller is not used. This parameter can be left off the command and the CS pin on the LCD display wired permanently to ground. If the touch controller is used this pin must then be specified and connected to an I/O pin.

e.g.

```
OPTION LCDPANEL ILI9341, LANDSCAPE, PE0, PD6, PC4
```

`INVERT` is a literal string indicating that the colours on the panel should be inverted. This corrects the colour on variants of the ILI9488 and ILI9341 panels which have the colours inverted. (i.e. Black is White)

In some circumstances it may be necessary to interrupt power to the LCD panel while the Armmite is running (e.g., to save battery power) and in that case the GUI RESET LCDPANEL command can be used to reinitialise the display the same as in power up.

If the LCD panel is no longer required, the command OPTION LCDPANEL DISABLE can be used which will return the I/O pins for general use, it will also disable the touch controller and return its I/O pins.

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

Important: The above test may not work if the display has a touch controller and the touch controller has not been configured (i.e., the touch Chip Select pin is floating). In this case configure the touch controller (see below) and then retry GUI TEST LCDPANEL.

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

User Defined LCD Panels in MMBasic

It is possible to write drivers for LCD Panels in MMBasic. The link below details these drivers and has an example for and I2C SSD1306 128*32 display panel that work for the Armmite F4.

```
OPTION LCDPANEL USER, 128,32
```

https://www.thebackshed.com/forum/ViewTopic.php?TID=10159&PID=140808#140808

Loadable Driver LCD Panels as CSUBs

With the introduction of CSubs is should now be possible to write loadable drivers for the Armmite F4.

There are none written at present. The link below points to a table maintained on the Fruit of the Shed wiki page that is usually kept up to date with the drivers available for the Micromite and Armmites.

https://fruitoftheshed.com/wiki/doku.php?id=mmbasic_hardware:supported_lcd_displays

Touch Support

Many LCD panels are supplied with a resistive touch sensitive panel and associated controller chip. To use the touch feature in MMBasic the touch controller must first be connected to the Armmite F4 (see the above chapter for the details) and then configured (see below).

When Touch is enabled SPI2 is also enabled, so it reserves the SPI2 pins. The touch chip on the LCD need to be queried using SPI2 to find out where the touch occurred. T_IRQ will only indicate a touch has occured. Disabling Touch will disable SPI2 and free up the pins unless its an SPI type LCD panel which also needs the SPI2.

Configuring Touch

To use the touch facility MMBasic must be configured using the OPTION TOUCH command which is normally entered at the command prompt. This should be done after the LCD panel has been configured. Every time the Armmite is restarted MMBasic will automatically initialise the touch controller. This command can also be embedded in a program with certain conditions – see the section Embedding Configuration Items in the Program

The syntax is:

```
OPTION TOUCH P12, PC5[,click]
```

The optional *click* pin will cause an audible click when a touch is detected if a piezo buzzer is connected between it and GND. Command GUI BEEP period will cause a beep of duration *period* msec.

If the touch facility is no longer required use the command OPTION TOUCH DISABLE to disable the touch feature and return the I/O pins for general use (the 'T_CS pin' should be held high to disable the controller).

Calibrating the Touch Screen

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

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

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

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

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

MMBasic will report a touch controller hardware failure during calibration if it gets identical values from two different touch points. This is reported after the second calibration point is displayed and you touch it. You cannot assume that the first touch was correct. Its saying that are both the same and probably both incorrect.

Touch Functions

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

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

Both functions return -1 if the screen is not being touched. See the Advanced Graphics sections for more information on using touch.

The GUI BEEP Command

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

GUI BEEP msec

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

Touch Interrupts

The following command will enable the touch interrupt. A separate subroutine can be called for each of the touch down and touch up events.

‘Set up the interrupt

```
GUI INTERRUPT IntTouchDown, IntTouchUp

‘These subroutines is called each time there is a touch on the LCDPanel
SUB IntTouchDown
  PRINT TOUCH(X), TOUCH(Y)
END SUB

SUB IntTouchUp
  PRINT “you took your finger off”
END SUB
```

Specifying the number zero (single digit) as the argument will cancel both of these interrupts. i.e.:

```
 GUI INTERRUPT 0
```

See the Advanced Graphics sections for more information on using touch and it interaction with the graphic controls.

PS2 Keyboard and LCDPANEL as Console

The ArmiteF4 can be used as a stand alone computer if a PS2 keyboard is attached and the LCDPANEL is used as the main console.

LCD Display as the Console Output

A PS2 keyboard can be used on its own as an alternative input method but it works particularly well when the LCD display panel is used as the console output. The LCD must be in the landscape or reverse landscape orientation and it must be first configured using OPTION LCDPANEL. Only the 16bit parallel LCD displays are supported for use as a console. SPI screens are too slow when it comes to scrolling the screen.

To enable the output to the LCD panel you should use the following command:

```
OPTION LCDPANEL CONSOLE [font [, fc [, bc [, blight]]]
```

'font' is the default font, 'fc' is the default foreground colour, 'bc' is the default background colour and 'blight' is the default backlight brightness (2 to 100). These settings are saved in flash and are used to configure MMBasic at power up. They are all optional and default to font 2, white, black and the current brightness. (50% by default).

Colour coding in the editor (see below) is also turned on by this command (OPTION COLOURCODE OFF will turn it off again). To disable using the LCD panel as the console the command is

OPTION LCDPANEL NOCONSOLE.

Used with a PS2 keyboard this option turns the Armmite F4 into a selfcontained computer with its own keyboard and display. Rather like a modern version of the Maximite (see http://geoffg.net/maximite.html).

Using LCDPANEL as the Console

When you are using the LCD Panel as the console the LCD Panel is providing a dual role as your terminal and as your LCD graphical display. When a program is running any print commands as well as any graphic commands will both write to the display. The FONT command does not change the Prompt Font when OPTION LCDPANEL CONSOLE is enabled. Use OPTION LCDPANEL CONSOLE *font* to change the font used by the console. The FONT command can be used to change the Font used by default within a program but at the LCDPanel Console is only effective for the current command line as the font reverts to the Prompt Font used by the console as soon as the command completes. e.g.

FONT 4**:**TEXT 10,10,"HELLO" 'on a single line will use FONT 4

However as below won't use FONT 4 unless it is already used as the Prompt Font..

FONT 4 'When this command completes the font reverts to the Prompt Font of the console

TEXT 10,10,"HELLO" 'Uses the current Prompt Font set for the LCDPanel Console

PS2 Keyboard

The connection diagram for the keyboard is shown on the below. The Armmite enables weak pullups on the clock and data lines so the 4.7 K resistors shown in the diagram are optional and for most keyboards there will be no ill effects if they are omitted. Refer to the pinout diagrams in section Pin and Connector Capabilities to see the pins used.

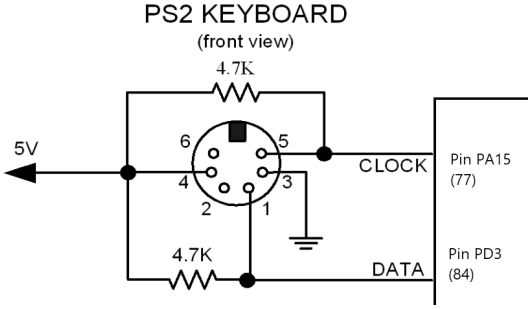
If you don't have a PS2 keyboard they may not be as hard to come by as you may think. See this thread on the backshed forum. Many wired keyboards still support PS2 even though they have a USB connector and operate as a USB keyboard. If you find one that works as PS2 and is currently available, please add to the thread.

https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=13440

PS2 Connection for Armmite F4

The PS2 connector can be bypassed, just plug the USB connector on the keyboard into a female usb breakout board and run D+ to the F4 keyboard clock (PA15, pin 77) and D- to keyboard data (PD3, pin 84), set OPTION KEYBOARD US and test. Connect 5V and GND as well.

If it doesn't work it is worth swapping D+ and D- over and trying to add the 4.7K resistors as trouble shooting steps, but some USB keyboards just will no longer support PS2.



Before the keyboard can be used it must first be enabled by specifying the language of the keyboard:

```
OPTION KEYBOARD language
```

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). 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 Armmite F4 fonts (another character will be used instead).

This command configures the I/O pins dedicated to the keyboard and initialises it for use. As with the similar commands for TOUCH, etc. this option will be saved in flash memory and automatically applied on power up. If you want to remove the keyboard you can do this with the OPTION KEYBOARD DISABLE command.

Graphics Commands and Functions

Colours

Colour is specified as a true colour 24 bit number where the top eight bits represent the intensity of the red colour, the middle eight bits the green intensity and the bottom eight bits the blue. For example, the colour red is &HFF0000 and yellow is &HFFFF00. An easier way to generate a colour value is to use the RGB() function which has the form: `RGB(red, green, blue)`

A value of zero for a colour represents black and 255 represents full intensity.

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

Because the Armmite F4 uses double precision floating point it can store the 24 bit number representing colour (i.e., returned by the RGB() function) in either a floating point variable or an integer variable.

The MMBasic LCD Panel drivers will automatically translate all colours to RGB565 format. i.e. 65K colours when they are output to the LCD panel. See below for details.

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

The COLOUR command takes the format:

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

Fonts

The Armmite F4 has seven built in fonts plus it can use embedded fonts to a maximum of 16 fonts.

There are seven built in fonts. These are:

Font Number	Size (width x height)	Character Set	Description
1	8 x 13	All 95 characters	A small font where a dense display is required.
2	12 x 20	All 95 characters	General use on 480 x 272 displays
3	16 x 24	All 95 characters	General use on 800 x 480 displays
4	16 x 24 BOLD	All 95 characters	A bold version of font #3
5	10 x 16	All 95 characters plus 7F to FF (hex)	A font with extended graphics Characters.suitable for high resolution displays.
~~5~~	~~24 x 32~~	~~All 95 characters~~	~~Large font, very clear~~
6	32 x 50	0 to 9 plus some symbols	Numbers plus decimal point, positive, negative, equals, degree and colon symbols. Very clear.
7	6 x 8	All 95 ASCII characters	A small font useful when low resolutions are used.

Note: The previous 24 x 32 font 5 has been replaced by at 10 x 16 font to save program space. The 24 x 32 font can be loaded as an embedded font and saved to the Library if required.

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

Embedded Fonts

The Armmite F4 supports embedded fonts. Note that because of the way the fonts are managed you cannot redefine fonts 1, 6 or 7.

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

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

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

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

```
DefineFont #8
 60200604
 44000000 00A04040 A0AEAE00 82406C6C EACC2048 00004460 84204424 E4A48044
 00E404A0 00800400 040000E0 00480240 4CE0AAEA 48C24044 C062C2E0 E820E2AA
 EA68E0E2 8048E2E0 EAE0EAEA 0404C0E2 80040400 0E208424 2484000E 4040E280
 4A60E84A CACAA0EA 608868C0 E8C0AACA E8E8E0E8 60EA6880 E4A0EAAA 2A22E044
 A0CAAA40 AEE08888 EEAEA0EA 40AA4AA0 4A80C8CA ECCA60AE C04268A0 AA4044E4
 A4AA60AA A0EEAA40 AAA04AAA 48E24044 E088E8E0 E2004208 004AE022 F0000000
 0C000084 AA8CE06A 608806C0 0660AA26 E42460AC 24AE0640 40A0CA88 22204044
 A0CC8AA4 0EE044C4 AA0CA0EE 40AA04A0 06C8AA0C 880662AA C0C60680 0A60444E
 AE0A60AA E0AE0A40 0AA0440A 6C0E24A6 608464E0 C4400444 006CC024 E0EEEE00
End DefineFont
```

You can convert and create font files to this format using the program FontTweak from: https://www.c-com.com.au/MMedit.htm

Read Only Variables

All screen coordinates and measurements on the screen are done in terms of pixels with the X coordinate being the horizontal position and Y the vertical position. The top left corner of the screen has the coordinates X = 0 and Y = 0 and the values increase as you move down and to the right of the screen.

There are six 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 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 in a font have the same width.
- MM.HPOS
  Returns the X coordinate of the text cursor (i.e., the horizontal location (in pixels) of where the next character will be printed on the LCD panel)
- MM.VPOS
  Returns the Y coordinate of the text cursor (i.e., the vertical location (in pixels) of where the next character will be printed on the LCD panel)

Drawing Commands

There are basic drawing commands that you can use within MMBasic programs on the Armmite to interact with an attached LCD display. There is also a series of more powerful GUI commands for drawing switches, radio buttons, etc. See the next section Advanced Graphics for more details.

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

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

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

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

The drawing commands are:

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

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

Rotated Text

The Armmite allows you to 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

This extra feature applies in the TEXT and GUI CAPTION commands.

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

```
TEXT 0, 250, "LCD 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

If the display is capable of transparent text, the TEXT command will allow the use of -1 for the background colour. This means that the text is drawn over the background with the background image showing through the gaps in the letters. Displays capable of transparent text are any that use the ILI9341 controller, SSD1963 or IPS_4_16 controllers. Using the LOAD command, you can load an image from the SD card.

BLIT Command

If the display is capable of transparent text (see the above subheading) programs can also use the BLIT command. This 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 CLOSE #b

#b is the buffer number in the range of 1 to 64. 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.

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.

The Armmite F4 allows up to 64 buffers, but the limiting factor will be the amount of memory used by the open buffers. This is dependent on the size of the buffers required to hold the area you read in. e.g. A 32*32 section loaded in to a Blit buffer will use 32*32*3 bytes. i.e. 3K. There is only 114K of memory for all variable etc. used by the program, so you need to be aware of this when filling BLIT buffers.

Load Image

As previously described in the SD Card Support section the LOAD IMAGE command can be used to load a bitmap image from the SD card and display it on the LCD display. This can be used to draw a logo or add an ornate background to the graphics drawn on the display. All types of the BMP format including black and white and true colour 24-bit images. The image can be positioned anywhere on the screen and be of any size (pixels that end up being positioned off the screen and will be ignored).

RGB888 Vs RGB565 with Pixel()

MMBasic uses RGB888 internally. Colours are stored as 24 bits, 8 bits for each of Red, Green and Blue. The LCD displays on the Armmite F4 are all set to use RGB565. The lower 3 bits for Red and Blue are discarded and the lower 2 bits for Green are discarded. The LCD drivers take care of all this and is rarely a concern. The only time it likely to be noticed is when using the PIXEL() function. When reading the colour of a Pixel you may not get what you expect if you compare the result with the RGB888 colour initially sent to the LCD via the PIXEL command or some graphic command. The RGB888 is modified to RGB565 before it is sent to the LCD and only the RGB565 is read back. When converted back to RGB888 the lower bits are set to 0. The original colour would need to be ANDed with &HF8FCF8 to match the returned value.

Example

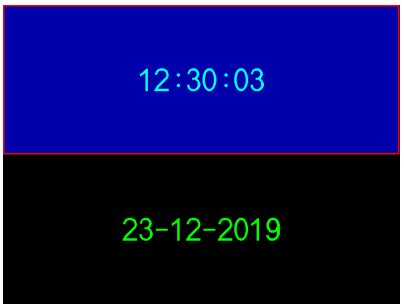
As an example, the following program will draw a simple digital clock on the LCD.

```
CLS
CONST DBlue = RGB(0, 0, 128)              ' A dark blue colour
COLOUR RGB(GREEN), RGB(BLACK)             ' Set the default colours
FONT 6                                    ' Set the default font

BOX 0, 0, MM.HRes-1, MM.VRes/2, 3, RGB(RED), DBlue

DO
  TEXT MM.HRes/2, MM.VRes/4, TIME$, "CM", 6, 1, RGB(CYAN), DBlue
  TEXT MM.HRes/2, MM.VRes*3/4, DATE$, "CM"
  IF TOUCH(X) <> -1 THEN END
LOOP
```

The program starts by defining a constant with a value corresponding to a dark blue colour and then sets the defaults for the colours and the font. It then draws a box with red walls and a dark blue interior. Following this the program enters a continuous loop where it performs three functions:

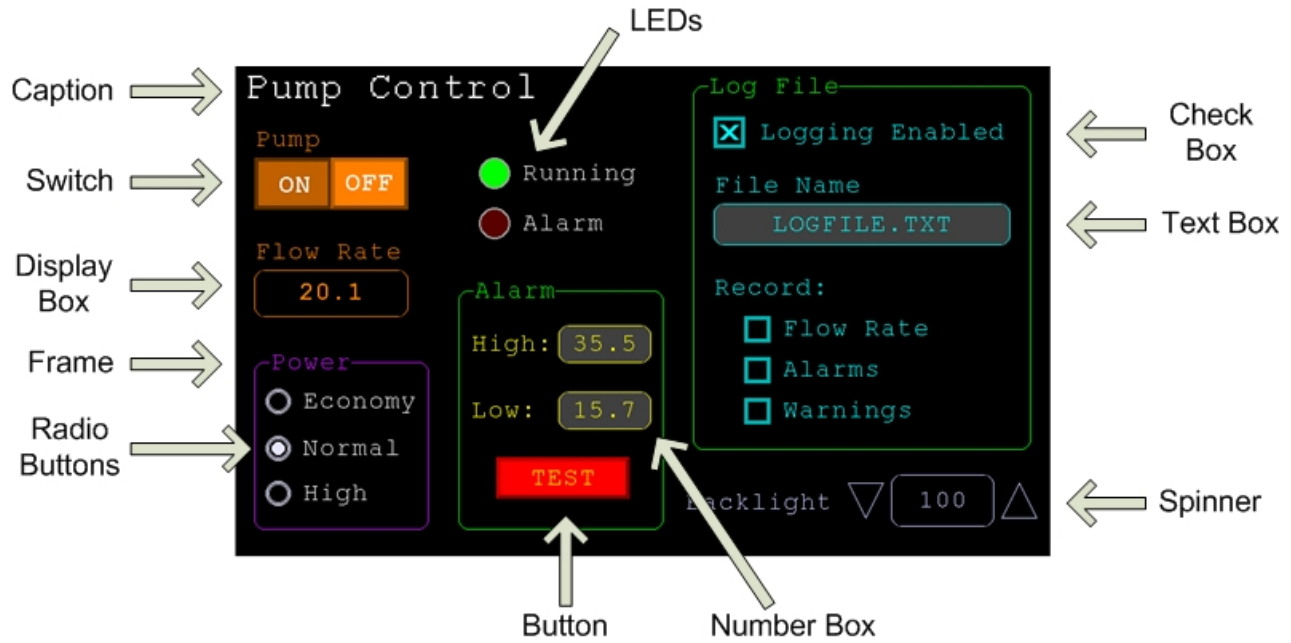
- Displays the current time inside the previously drawn box. The string is drawn centred both horizontally and vertically in the middle of the box. Note that the TEXT command overrides both the default font and colours to set its own parameters.
- Draws the date centred in the lower half of the screen. In this case the TEXT command uses the default font and colours previously set.
- Checks for a touch on the screen. This is indicated when TOUCH(X) function returns something other than -1. In that cas the program will terminate.

  The screenshot shows the result.



Advanced Graphics

The Armmite F4 incorporates a suite of advanced graphic controls that respond to touch, these include on screen switches, buttons, indicator lights, keyboard, etc. MMBasic will draw the control and animate it (i.e., a switch will appear to depress when touched). All that the BASIC program needs to do is invoke a single line command to specify the basic details of the control.



Each control has a reference number called '#ref' in the description of the control. By default, this can be any number between 1 and 100 and the upper limit can be changed with the OPTION CONTROL command. The reference number is used to identify a control. For example, a check box can be created thus:

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

And the program can check its value by using its reference number in the CtrlVal() function:

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

The # character is optional but serves to remind the programmer that this is not an ordinary number.

In the following commands any arguments that are in italic font (eg, *Width, Height*) are optional and if not specified will take the value of the previous command that did specify them. This means for example, that a number of radio buttons with the same size and colour can be specified with only the first button having to list all the details. Note that with the colour specification this is different to the Basic Drawing Commands which default to the last COLOUR command.

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

If the display is capable of transparent text these commands will allow the use of -1 for the background colour. This means that the text is drawn over the background with the background image showing through the gaps in the letters. Displays capable of transparent text are any that use the ILI9341 controller or an SSD1963 controller. The latter must have the RD pin specified in the OPTION LCDPANEL command.

The advanced graphics controls are:

Frame

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

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

LED

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

This will draw an indicator light (it looks like a panel mounted LED). When its value is set to one it will be illuminated and when it is set to zero it will be off (a dull version of its colour attribute). The LED can be made to flash by setting its value to the number of milliseconds that it should remain on before turning off.

The caption will be drawn to the right of the LED and will use the colours set by the COLOUR command. The LED control is not animated when touched but its reference number can be found using TOUCH(REF) and TOUCH(LASTREF) in the touch interrupts and any required animation can be done in MMBasic.

Check Box

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

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

Push Button

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

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

Switch

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

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

Radio Button

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

This will draw a radio button with a caption. When touched the centre of the button will be illuminated to indicate that this option has been selected and the control's value will be 1. When another radio button is selected the mark on this button will be removed and its value will be zero. Radio buttons are grouped together when surrounded by a frame and when one button in the group is selected all others in the group will be deselected. If a frame is not used all buttons on the screen will be grouped together.

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

Display Box

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

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

Text Box

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

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

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

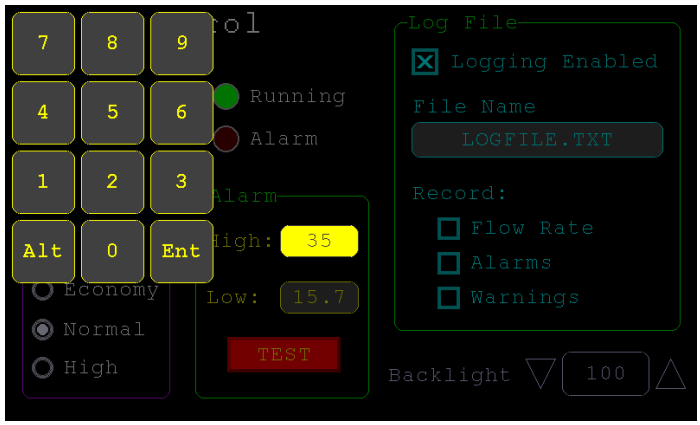
The value of the control can be set to a string starting with two hash characters (##) and in that case the string (without the leading two hash characters) will be displayed in the box with reduced brightness. This can be used to give the user a hint as to what should be entered (called "ghost text").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

MMBasic will try to position the virtual keyboard on the screen so as to not obscure the text box that caused it to appear. A pen down interrupt will be generated when the keyboard is deployed and a key up interrupt will be generated when the Enter or Cancel keys are touched and the keyboard is hidden. If necessary, the virtual keyboard can be dismissed by the program (same as touching the cancel button) with the command: GUI TEXTBOX CANCEL. If the virtual keyboard is not displayed this will do nothing.

Number Box

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

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



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

Similar to the Text Box, the value of the control can set to a literal string with two leading hash characters (e.g., "##Hint") and in that case the string (without the leading two characters) will be displayed in the box with reduced brightness. Reading this will return zero and when a key is pressed the ghost text will vanish.

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

If necessary, the virtual keypad can be dismissed by the program (same as touching the cancel button) with the command: GUI NUMBERBOX CANCEL. If it is not displayed this command will do nothing.

Formatted Number Box

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

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

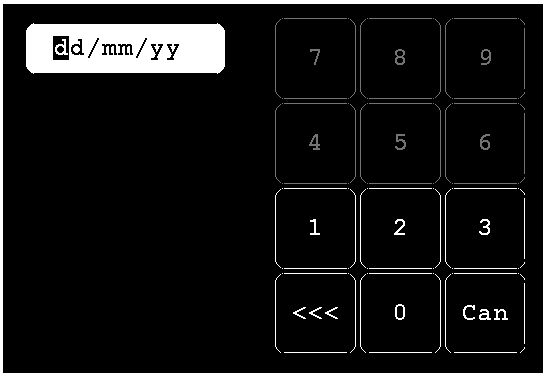
The type of entry is controlled by the 'Format' argument as follows:

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

For example:

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

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



- The display box is filled with a guide string to prompt the user as to the data required.
- Because the day of the month can only start with a digit from 0 to 3 all other keys are disabled. This also happens with other numbers that have a limited range.
- The value of the control retrieved via CtrlVal(#1) is a string. As an example, if the user entered the date for the 8th of May 2020 the returned string would be "08/05/20" (i.e., the UK/Aust/NZ format as specified by DATE1).

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

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

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

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

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

If necessary, the virtual keypad can be dismissed by the program (same as touching the cancel button) with the command: GUI FORMATBOX CANCEL (if the keypad is not displayed this command will do nothing).

Spin Box

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

This will draw a box with up/down icons on either end. When these icons are touched the number in the box will be incremented or decremented by the 'StepValue', holding down the touch will repeat at a fast rate. 'Minimum' and 'Maximum' set a limit on the value that can be entered. 'StepValue', 'Minimum' and 'Maximum' are optional and if not specified 'StepValue' will be 1 and there will be no limit on the number entered. A pen down interrupt will be generated every time up/down is touched or when automatic repeat occurs.

Caption

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

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

'Alignment' is zero to three characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE, BOTTOM. A third character can be used to indicate the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (i.e., upside down), 'U' the text will be rotated counter clockwise by 90º and 'D' the text will be rotated clockwise by 90º. The default alignment is left/top with no rotation.

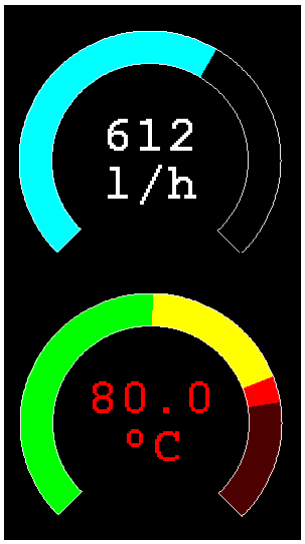
If the colours are not specified this control will use the colours set by the COLOUR command.

Circular Gauge

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

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

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



'min' is the value associated with the minimum value of the gauge and 'max' is the maximum value. When CtrlVal() is used to assign a value (floating point or integer) to the gauge the analogue portion of the gauge will be drawn to a length proportional to the range between 'min' and 'max'. At the same time the digital value will be drawn in the centre of the gauge using the current font settings (set with the FONT command). 'nbrdec' specifies the number of decimal places to be used in this display. Under the digital value the 'units$' will be displayed (this can be skipped or a zero length string used if not required).

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

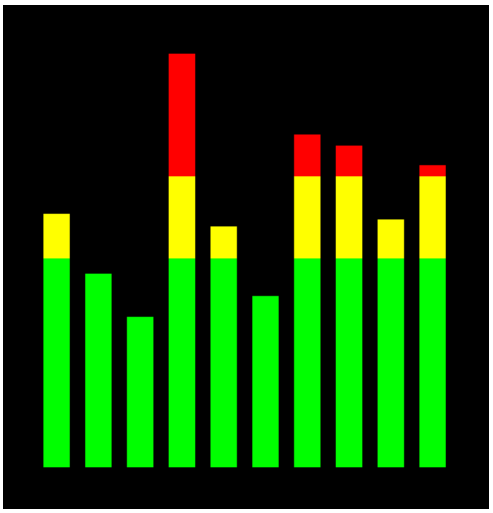
Specifically, 'c1' is the colour to be used for values up to 'ta'. 'c2' is the colour to be used for values between 'ta' and 'tb', 'c3' is used for values between 'tb' and 'tc' and c4 is used for values above 'tc'. Colours and thresholds not required can be left off then list. For example, for a two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

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

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

Bar Gauge

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



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

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

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

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

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

Area

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

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

Interacting with Controls

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

- = CTRLVAL(#ref)
  This is a function that will return the current value of a control. For controls like check boxes or switches it will be the number one (true) indicating that the control has been selected by the user or zero (false) if not. For controls that hold a number (e.g., a SPINBOX) the value will be the number (normally a floating point number). For controls that hold a string (e.g., TEXTBOX) the value will be a string. For example:
  ```
  PRINT "The number in the spin box is: " CTRLVAL(#10)
  ```

- CTRLVAL(#ref) =
  This command will set the value of a control. For off/on controls like check boxes it will override any touch input and can be used to depress/release switches, tick/untick check boxes, etc. A value of zero is off or unchecked and non zero will turn the control on. For a LED it will cause the LED to be illuminated or turned off. It can also be used to set the initial value of spin boxes, text boxes, etc. For example:
  ```
  CTRLVAL(#10) = 12.4
  ```

- GUI FCOLOUR colour, #ref1 [, #ref2, #ref3, etc]
  This will change the foreground colour of the specified controls to 'colour'. This is especially handy for a LED which can change colour.

- GUI BCOLOUR colour, #ref1 [, #ref2, #ref3, etc]
  This will change the background colour of the specified controls to 'colour'.

- = TOUCH(REF)
  This is a function that will return the reference number of the control currently being touched. If no control is currently being touched it will return zero.

- = TOUCH(LASTREF)
  This is a function that will return the reference number of the control that was last touched.

- GUI DISABLE #ref1 [, #ref2, #ref3, etc]
  This will disable the controls in the list. Disabled controls do not respond to touch and will be displayed dimmed. The keyword ALL can be used as the argument and that will disable all controls on the currently displayed page. For example:
  ```
  GUI DISABLE ALL
  ```

- GUI ENABLE #ref1 [, #ref2, #ref3, etc]
  This will undo the effects of GUI DISABLE and restore the controls in the list to normal operation. The keyword ALL can be used as the argument for all controls on the currently displayed page.

- GUI HIDE #ref1 [, #ref2, #ref3, etc]
  This will hide the controls in the list. Hidden controls will not respond to touch and will be replaced on the screen with the current background colour. The keyword ALL can be used as the argument.

- GUI SHOW #ref1 [, #ref2, #ref3, etc]
  This will undo the effects of GUI HIDE and restore the controls in the list to being visible and capable of normal operation. The keyword ALL can be used as the argument for all controls.

- GUI DELETE #ref1 [, #ref2, #ref3, etc]
  This will delete the controls in the list. This includes removing the image of the control from the screen using the current background colour and freeing the memory used by the control. The keyword ALL can be used as the argument and that will cause all controls to be deleted.

MsgBox()

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

The syntax is:

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

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

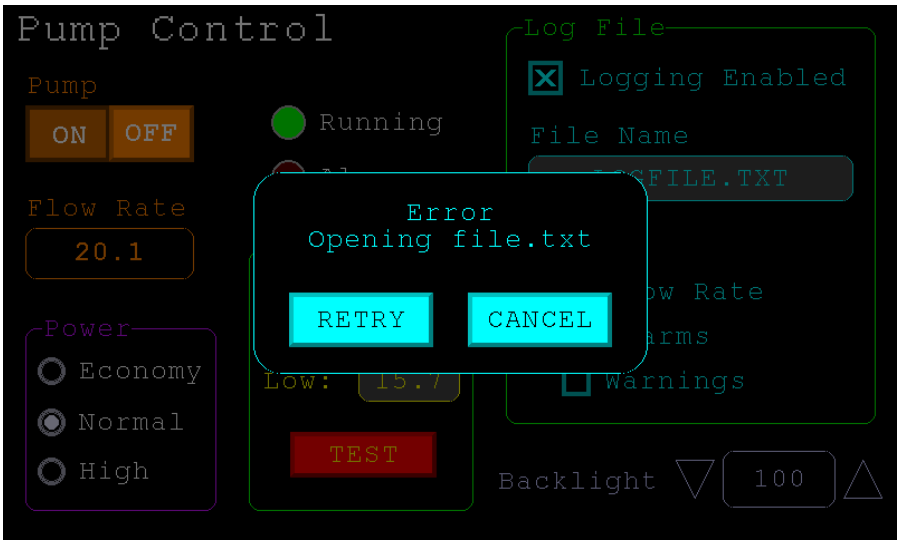
The font used will be the default font set using the FONT command and the colours used will be the defaults set by the COLOUR command. The box will be automatically sized taking into account the dimensions of the default font, the number of lines to display and the number of buttons specified.

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

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

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

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



Advanced Graphics Programming Techniques

When programming using the advanced GUI commands implemented on the Armite F4 there are a number of hints and techniques to consider that will make it easier to develop and maintain your program.

The User Should Be In Control

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

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

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

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

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

Program Structure

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

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

```
GUI CAPTION #1, "Speed (rpm)", 200, 50       ' label the number box
GUI NUMBERBOX #2, 200, 100, 150, 40          ' define and draw the number box
CtrlVal(#2) = 100                            ' default value for the speed
GUI BUTTON #3, "RUN", 200, 350, 0, RGB(red)  ' define and draw the RUN button

DO                                           ' this runs in a loop forever
  IF CtrlVal(#3)<10 OR CtrlVal(#3)>200 THEN  ' check the speed setting
    GUI DISABLE #3                           ' disable RUN if it is invalid
  ELSE                                       ' otherwise
    GUI ENABLE #3                            ' enable the RUN button
  ENDIF

  IF CtrlVal(#3) = 1 THEN                    ' if the button is pressed
    SetMotorSpeed CtrlVal(#2)                ' make the motor run
  ELSE                                       ' otherwise the button is up
    SetMotorSpeed 0                          ' therefore set motor speed to zero
  ENDIF
LOOP
```

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

Disable Invalid Controls

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

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

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

Use Constants for Control Reference Numbers

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

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

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

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

By default the reference number must be a number between 1 and 100 however the upper limit can be changed with the OPTION CONTROL command. Increasing the number will consume more RAM and decreasing it will recover some RAM.

The Main Program Is Still Running

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

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

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

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

Use Interrupts and SELECT CASE Statements

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

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

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

```
CONST ALARM_ON = 15
CONST ALARM_OFF = 16
CONST ALARM_LED = 33
GUI INTERRUPT PenDown
'…
GUI BUTTON ALARM_ON, "ALARM ON ", 330, 50, 140, 50, RGB(white), RGB(blue)
GUI BUTTON ALARM_OFF, "ALARM OFF ", 330, 150, 140, 50, RGB(white), RGB(blue)
GUI LED ALARM_LED, 215, 220, 30, RGB(red)
'…
DO : LOOP    ' the main program is doing something

' this sub is called when touch is detected
SUB PenDown
  SELECT CASE TOUCH(REF)
    CASE ALARM_ON
      CtrlVal(ALARM_LED) = 1
    CASE ALARM_OFF
      CtrlVal(ALARM_LED) = 0
  END SELECT
END SUB
```

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

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

Touch Up Interrupt

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

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

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

```
GUI INTERRUPT IntTouchDown, IntTouchUp
```

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

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

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

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

Keep Interrupts Very Short

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

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

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

Multiple Screens

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

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

```
GUI SETUP 1
GUI Caption #1, "Flow Rate", 20, 170,, RGB(brown),0
GUI Displaybox #2, 20, 200, 150, 45

GUI SETUP 2
GUI Caption #3, "High:", 232, 260, LT, RGB(yellow)
GUI Numberbox #4, 318, 6,90, 12, RGB(yellow), RGB(64,64,64)

GUI SETUP 3
GUI Checkbox #5, "Alarms", 500, 285, 25
GUI Checkbox #6, "Warnings", 500, 325, 25
```

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

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

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

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

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

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

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

A typical usage of the PAGE command is shown below. Two buttons (which are always displayed) allow the user to select between the first page and the second page. The switch is done in the touch down interrupt.

```
GUI SETUP 1
GUI Button #10, "SELECT PAGE ONE", 50, 100, 150, 30, RGB(yellow), RGB(blue)
GUI Button #11, "SELECT PAGE TWO", 50, 140, 150, 30, RGB(yellow), RGB(blue)

GUI SETUP 2
GUI Caption #1, "Displaying First Page", 20, 20

GUI SETUP 3
GUI Caption #2, "Displaying Second Page", 20, 50

Page 1, 2
GUI INTERRUPT TouchDown
Do
   ' the main program loop
Loop

Sub TouchDown
  If Touch(REF) = 10 Then Page 1, 2
  If Touch(REF) = 11 Then Page 1, 3
End Sub
```

Multiple Interrupts

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

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

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

Using Basic Drawing Commands

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

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

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

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

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

The following example demonstrates this technique. On a 5" or 7" display it initially draws a box filled with bright blue using the basic drawing commands. Then, when the number pad is about to pop up it will redraw the box in a dull colour. Finally, when the keypad is removed from the screen the pen up interrupt will redraw the box in its original colours.

```
GUI INTERRUPT TouchDownInterrupt, TouchUpInterrupt
BOX 400, 250, 300, 200,  , RGB(WHITE), RGB(BLUE)
GUI NUMBERBOX 1, 318,100,90,40,RGB(YELLOW),RGB(64,64,64)
DO : LOOP

SUB TouchDownInterrupt
  IF TOUCH(REF) = 1 THEN BOX 400, 250, 300, 200,  , RGB(128,128,128), RGB(0,0,128)
END SUB

SUB TouchUpInterrupt
  IF TOUCH(LASTREF) = 1 THEN BOX 400, 250, 300, 200,  , RGB(WHITE), RGB(BLUE)
END SUB
```

Overlapping Controls

Controls can be defined to overlap on the display, this mostly occurs with GUI AREA which, as an example, you might want to capture a touch that was intended for (say) a GUI BUTTON. This will allow you to create your own animation for the button rather than that provided by MMBasic. In this case the control that you wish to respond to the touch (ie, GUI AREA) should have a lower reference number (ie, `#ref`) than the control that it is covering (ie, the GUI BUTTON). This is because when the screen is touched MMBasic will check the current list of active controls starting with control number 1 and working upwards. When a match is made MMBasic will take the appropriate action and terminate the search. This results in the lower numbered control effectively masking out a higher numbered control covering the same screen area as the touched location.

Timing LCD Updates with GETSCANLINE()

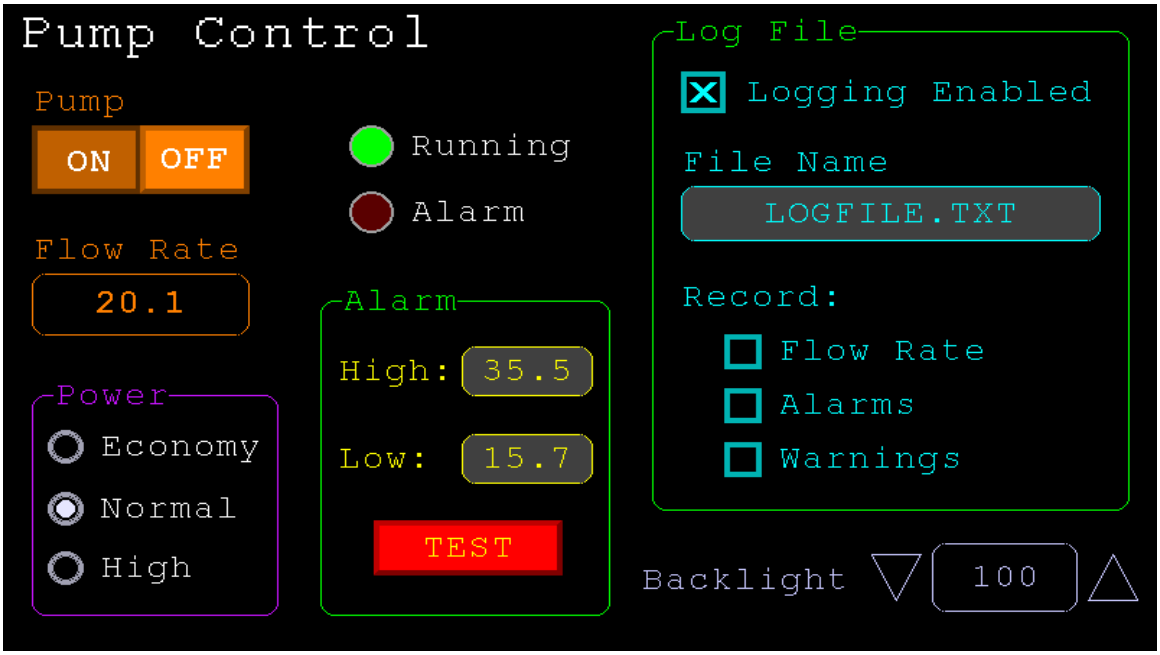
In some cases, you may see a tearing effect or flicker when updating an LCD display. When you write to the LCD you are just updating its RAM. The LCD Panel is taking that data in RAM and updating the actual screen at its own (pretty fast) pace. If you change the data while it is refreshing that part of the screen, it may be half way through reading that section, so the screen will briefly have part of the old data and part of the new data until the next refresh occurs. You can use the GETSCANLINE() function to ask what line is being updated. Use this to try timing your update for when the LCD is not refreshing from where you want to write. Its a bit of trial and error with the numbers, as the scanline moves pretty quickly so you need to estimate where it will be, but you can see the idea below. This display is 800*480 and refreshes line 0-800. The idea is not to do the update when the LCD is refreshing where the box will be, so only do the update if its gone past line 100 and its not near the end and about to restart at 0. *You would not normally need this unless you see the problem.*

```
i=GETSCANLINE()
do while i>650 or i<= 100                    'wait for scanline to be where you want it
  i=GETSCANLINE()
loop
BOX 0,0,100,100,,RGB(RED),RGB(RED)   'its not near our box so do the update
```

The Pump Control Example GUI Program

As a test you can enter the following "Pump Control" demonstration program as shown in this YouTube video: https://youtu.be/j12LidkzG2A. It will draw a selection of advanced controls as shown below.

These controls are active so that you can test how they work.



Note that this demonstration expects a 800 x 480 pixel LCD panel in landscape orientation with touch (ie, a 5", 7" or 8" SSD1963 based panel or one of the IPS_4_16 800*480).

```
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''
' Demonstration program for the Micromite+ and Armmite F4
' It does not do anything useful except demo the various controls
'
' Geoff Graham, October 2015
''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''''

Option Explicit
Dim ledsY
Colour RGB(white), RGB(black)

' reference numbers for the controls are defined as constants
Const c_head = 1, c_pmp = 2, sw_pmp = 3, c_flow = 4, tb_flow = 5
Const led_run = 6, led_alarm = 7
Const frm_alarm = 20, nbr_hi = 21, nbr_lo = 22, pb_test =23
Const c_hi = 24, c_lo = 25
Const frm_pump = 30, r_econ = 31, r_norm = 32, r_hi = 33
Const frm_log = 40, cb_enabled = 41, c_fname = 42, tb_fname = 43
Const c_log = 44, cb_flow = 45, cb_pwr = 46, cb_warn = 47
Const cb_alarm = 48, c_bright = 49, sb_bright = 50

' now draw the "Pump Control" display
CLS
GUI Interrupt TouchDown, TouchUp

' display the heading
Font 2,2 : GUI Caption c_head, "Pump Control", 10, 0
Font 3 : GUI Caption c_pmp, "Pump", 20, 60, , RGB(brown)

' now, define and display the controls
' first display the switch
Font 4
GUI Switch sw_pmp, "ON|OFF", 20, 90, 150, 50, RGB(white),RGB(brown)
CtrlVal(sw_pmp) = 1
```

```
' the flow rate display box
Font 3 : GUI Caption c_flow, "Flow Rate", 20, 170,, RGB(brown),0
Font 4 : GUI Displaybox tb_flow, 20, 200, 150, 45
CtrlVal(tb_flow) = "20.1"

' the radio buttons and their frame
Font 3 : GUI Frame frm_pump, "Power", 20, 290, 170, 163, RGB(200,20,255)
GUI Radio r_econ, "Economy", 43, 328, 12, RGB(230, 230, 255)
GUI Radio r_norm, "Normal", 43, 374
GUI Radio r_hi, "High", 43, 418
CtrlVal(r_norm) = 1     ' start with the "normal" button selected

' the alarm frame with two number boxes and a push button switch
Font 3 : GUI Frame frm_alarm, "Alarm", 220, 220, 200, 233,RGB(green)
GUI Caption c_hi, "High:", 232, 260, "LT", RGB(yellow)
GUI Numberbox nbr_hi, 318,MM.VPos-6,90,MM.FontHeight+12,RGB(yellow),RGB(64,64,64)
GUI Caption c_lo, "Low:", 232, 325, LT, RGB(yellow),0
GUI Numberbox nbr_lo, 318,MM.VPos-6,90,MM.FontHeight+12,RGB(yellow),RGB(64,64,64)
GUI Button pb_test, "TEST", 257, 383, 130, 40,RGB(yellow), RGB(red)
CtrlVal(nbr_lo) = 15.7 : CtrlVal(nbr_hi) = 35.5

' draw the two LEDs
Const ledsX = 255, coff = 50    ' define their position
ledsY = 105 : GUI LED led_run, "Running", ledsX, ledsY, 15, RGB(green)
ledsY = ledsY+49 : GUI LED led_alarm, "Alarm", ledsX, ledsY, 15, RGB(red)
CtrlVal(led_run) = 1   ' the switch defaults to on so set the LED on

' the logging frame with check boxes and a text box
Colour RGB(cyan), 0
GUI Frame frm_log, "Log File", 450, 20, 330, 355, RGB(green)
GUI Checkbox cb_enabled, "Logging Enabled", 470, 50, 30, RGB(cyan)
GUI Caption c_fname, "File Name", 470, 105
GUI Textbox tb_fname, 470, 135, 290, 40, RGB(cyan), RGB(64,64,64)
GUI Caption c_log, "Record:", 470, 205, , RGB(cyan), 0
GUI Checkbox cb_flow, "Flow Rate", 500, 245, 25
GUI Checkbox cb_alarm, "Alarms", 500, 285, 25
GUI Checkbox cb_warn, "Warnings", 500, 325, 25
CtrlVal(cb_enabled) = 1
CtrlVal(tb_fname) = "LOGFILE.TXT"

' define and display the spinbox for controlling the backlight
GUI Caption c_bright, "Backlight", 442, 415, ,RGB(200,200,255),0
GUI Spinbox sb_bright, MM.HPos + 8, 400, 200, 50,,,10, 10, 100
CtrlVal(sb_bright) = 100

' All the controls have been defined and displayed. At this point
' the program could do some real work but because this is just a
' demo there is nothing to do.  So it just sits in a loop.
Do : Loop


' the interrupt routine for touch down
' using a select case command it has a different process for each control
Sub TouchDown
  Select Case Touch(REF)      ' find out the control touched
    Case cb_enabled           ' the enable check box
      If CtrlVal(cb_enabled) Then
        GUI ENABLE c_fname, tb_fname, c_log, cb_flow, cb_alarm, cb_warn
      Else
        GUI Disable c_fname, tb_fname, c_log, cb_flow, cb_alarm, cb_warn
      EndIf
    Case sb_bright            ' the brightness spin box
      BackLight CtrlVal(sb_bright)
    Case sw_pmp               ' the pump on/off switch
      CtrlVal(led_run) = CtrlVal(sw_pmp)
      CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 20.1)
```

```
      CtrlVal(r_norm) = 1
    Case pb_test              ' the alarm test button
      CtrlVal(led_alarm) = 1
      GUI beep 250
    Case r_econ               ' the economy radio button
      CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 18.3)
    Case r_norm               ' the normal radio button
      CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 20.1)
    Case r_hi                ' the high radio button
      CtrlVal(tb_flow) = Str$(CtrlVal(sw_pmp) * 23.7)
  End Select
End Sub

' interrupt routine when the touch is removed
Sub TouchUp
  Select Case Touch(LASTREF)   ' use the last reference
    Case pb_test               ' was it the test button
      CtrlVal(led_alarm) = 0   ' turn off the LED
  End Select
End Sub
```

Miscellaneous Features

Serial Interfaces

The Armmite F4 has built in support for up to four serial interfaces. COM1, COM2, COM3 and COM4 are available as standard. When the serial console is enabled, (OPTION SERIAL CONSOLE ON) COM1 is unavailable and becomes the console.

All serial ports on the Armmite F4 can operate at high speed (up to 1.8M baud) and support the OC and S2 options. The DE pin for RS485 is not supported and would need to be driven explicitly by the MMBasic program. The INV option is not supported on the ARM chips.

SPI Interface

The Armite F4 has built in support for two SPI interfaces. The commands to control these interfaces use the identifier SPI for the first SPI port and SPI2 for the second port. All commands and functions that can be used on the first port (SPI) can be used on the second by using the identifier SPI2. These are:

- SPI2 OPEN
- SPI2 WRITE
- SPI2 READ
- = SPI2(args, …)
- SPI2 CLOSE

SPI based displays and the touch controller will all use the second SPI interface (SPI2). If any of these features are enabled SPI2 will be unavailable to BASIC programs (which should use the first SPI channel instead).

The onboard W25Q16 Flash chip and the NRF24L01 RF module are controlled on the first SPI port and the SPI pins are available at the NRF24L01 connector

Upgrading Your BASIC Program in the Field

Often it is desirable to send an upgraded version of your BASIC program to a user and let them load it under control of the program already running on the Armmite.

```
LOAD "filename.bas", R
```

Where *filename.bas* is the name of the upgraded BASIC file on the SD card. This will load the BASIC program into the Armmite's program memory and immediately restart the CPU and run the new program.

Your program could execute this command when the user touched a screen button –or- it could check once every minute for that file name and, if found, load and run it. Then, all you have to do is send the updated program (on an SD card) to your user to initiate an upgrade. Easy.

Creating CSUBs

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

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

https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=14128

Other Devices and Support Resources

The Back Shed Forum

Support questions should be raised on the Back Shed forum (http://www.thebackshed.com/forum/Microcontrollers) where there are many enthusiastic Maximite, Micromite and Armmite users who would be only too happy to help. The developers of both the Armmite F4 and MMBasic are also regulars on this forum.

The forum has a search option, but your can also use google to search the specific site only, by adding the site at the front of your search as below.

site:www.thebackshed.com armmiteF4 Manual

Geoff Graham the developer of MMBasic has many interesting projects and information on his website. This is where you can also request the source for your own personal use. (http://geoffg.net).

Fruit of the Shed Wiki

The Fruit of the Shed is a wiki initiated by TBS member @CaptainBoing. The wiki has a collection of useful code modules and device drivers for many common hardware items. A couple of pages have been created specifically targeted at the Armmite F4.

This link is to a summary page of items related to the Armmite User Manual and Firmware. It will generally point to relevant posts on TBS.

fruitoftheshed (FotS)

This page created by TBS member @lizby gives a good summary of the available add on modules that might be useful in your projects. It also details the adapter boards that have be made to allow other LCD panels to be matched up the the Armmite 32 pin FSMC connector.

https://fruitoftheshed.com/wiki/doku.php?id=mmbasic_hardware:start

This page provides a useful summary of all the LCD Panels that can be used with the various Micromites and Armmites. https://fruitoftheshed.com/wiki/doku.php?id=mmbasic_hardware:supported_lcd_displays

Interfacing various hardware modules

There are many useful hardware devices you many decide you want to use/try out. Many of them have been used with MMBasic already and the drivers are posted on TBS or the Fruit of the Shed. All these can be easily adapted to the Armmite F4. Basically anything that communicates via Serial, SPI, I2C, 1-Wire, outputs a voltage, current etc. can be interfaced if you know or can find the protocol used.

If the device has not been conquered by MMBasic as yet, the approach is to find a C version (Arduino) of the code used to interface it and convert that to MMBasic.

Internet Access using ESP8266

There are several methods of gaining wireless access to the internet using an ESP8266 module.

Armmite F4 Weather Station with ESP-01

http://www.thebackshed.com/forum/ViewTopic.php?TID=13419&PID=163479#163479

http://www.thebackshed.com/forum/ViewTopic.php?TID=11149&PID=131032#131032

https://sites.google.com/site/annexwifi/home

Long Strings

Long Strings are a set of commands and functions that allow MMBasic to manipulate strings of unlimited length and are particularly useful when dealing with data sent via WiFi and the Internet. Standard strings in MMBasic are limited to a maximum length of 255 characters. Long strings duplicate these functions but will work with strings of any length limited only by the amount of available RAM.

Long String Variables

Variables for holding long strings must be defined as integer arrays. The long string routines do not keep numbers in these arrays but just use them as blocks of memory for holding long strings. When creating these arrays they should 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.

The following is an example of declaring three long string variables which will be used to hold up to 2048 characters in each:

```
CONST MaxLen = 2048
DIM INTEGER Str1(MaxLen/8), Str2(MaxLen/8), Str3(MaxLen/8)
```

These will contain empty strings when created (ie, their length will be zero). When these variables are passed to the long string functions they should be entered as the variable name followed by empty brackets. e.g.

```
LONGSTRING COPY Str1(), Str2()
```

Long string variables can be passed as arguments to user defined subroutines and functions. e.g.

```
SUB MySub longarg() AS INTEGER
  PRINT "Long string length is" LLEN(longarg())
END SUB
```

And it could be called like this:

```
MySub str1()
```

Summary of the Commands and Functions

These are documented in detail in the Commands and Functions sections of this manual. The commands are:

LONGSTRING AES128 ENCRYPT/DECRYPT	Encrypts or decrypts a long string
LONGSTRING BASE64 ENCODE/DECODE	Encodes or decodes a long string using base 64
LONGSTRING CLEAR array%()	Clear (ie, set to empty) a long string
LONGSTRING COPY dest%(), src%()	Copy a long string
LONGSTRING CONCAT dest%(), src%()	Concatenate two long strings
LONGSTRING LCASE array%()	Convert a long string to lowercase
LONGSTRING LEFT dest%(), src%(), nbr	Get the left nbr characters from a long string
LONGSTRING LOAD array%(), nbr, string$	Copy characters to a long string
LONGSTRING MID dest%(), src%(), start, nbr	Get characters from the middle of a long string
LONGSTRING PRINT [#n,] src%()[;]	Print a long string. A semicolon will suppress CRLF.
LONGSTRING REPLACE array%() , string$, start	Replace characters in a long string
LONGSTRING RESIZE addr%(), nbr	Set the length of a long string
LONGSTRING RIGHT dest%(), src%(), nbr	Get the right nbr characters from a long string
LONGSTRING SETBYTE addr%(), nbr, data	Set a byte in a long string
LONGSTRING TRIM array%(), nbr	Trim characters from the left of a long string
LONGSTRING UCASE array%()	Convert a long string to uppercase

The functions are:

r = LGETBYTE(array%(), n)	Return the value of a byte in a long string
r$ = LGETSTR$(array%(), start, length)	Returns part of a long string as a normal string.
r = LINSTR(array%(), search$ [,start] [,size])	Returns the position of a string in a long string
r = LLEN(array%())	Returns the length of a long string
MATH(BASE64 ENCODE/DECODE	Encodes or decodes data using base 64

MMBasic Characteristics

Implementation Characteristics

Maximum program size (as plain text) is 132KB. 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 31 characters.

Maximum number of dimensions to an array is 6.

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

Maximum number of nested FOR…NEXT loops is 50.

Maximum number of nested DO…LOOP commands is 50.

Maximum number of nested GOSUBs, subroutines and functions (combined) is 50.

Maximum number of nested multiline IF…ELSE…ENDIF commands is 10.

Maximum number of user defined subroutines and functions (combined): 256

Maximum number of interrupt pins that can be configured: 10

Numbers are stored and manipulated as single precision floating point numbers or 64-bit signed integers. The maximum floating point number allowable is 3.40282347e+38 and the minimum is 1.17549435e-38. The Armmite F4 uses double precision

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 global variables and constants is 256

Maximum number of local variables is 256

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

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.

The SELECT CASE commands allow the programmer to create a clear and structured decision tree that is more flexible and easier to understand when multiple decisions must be made. The DO WHILE … LOOP command make it easy to build loops without using the GOTO statement. User defined subroutines and functions make it easy to add your own commands to MMBasic.
The IF… THEN command can span many lines with ELSEIF … THEN, ELSE and ENDIF statements as required and also spaced over many lines.

MMBasic Firmware Memory Map for the STM32F407 Implementation

Below is a summary of the details of how the MMBasic firmware makes use of the available resources on the STM32F407 chip. This detail is not really needed to use MMBasic but may be of interest if you want to dig deeper. The summary is derived from the STM32F407 Reference Manual (RM0090) and the MMBasic source code.Also the STM32F407 Datasheet

The STM32F407VGT chip used has 192K of RAM, 1024K of Flash , 4K of battery backed RAM and 20 32 bit RTC registers that are utilised. All the Ram, Flash and peripherals of the STM32F407 are mapped to a 32 bit address space. The table below lists the relevant address ranges for the MMBasic implementation.

Address Range (hex)	Type	Size	Usage/Detail
6000 0000 – FFFF FFFF			Reserved for use interannly by the chip. See data sheet for details.
4000 0000 - 5FFF FFFF	Registers Peripherals		This addess range allows access to the registers that control the various functions. Eg. GPIO, ADC, Timers, SPI, DAC, USART, I2C. MMBasic takes care of all this, however it is possible to PEEK and POKE these registers.
4002 4000 – 4002 43FF 4002 4400 – 4002 4FFF	Battery Backed Ram	1K 3K	Command Line Buffer Available to user This Ram is battery backed up if a battery is connected to VBAT.
4000 2800 – 4000 2BFF 4000 2850 – 4000 289C	RTC_BKP Registers 80 bytes	20 X 32 bit	
2000 0000 - 2001 FFFF	Ram	128K	This is MMBasic's memory. This is where a MMBasic program stores its variables and where it gets any memory it needs. Nominally 114K is available, but the whole 128K can be seen if OPTION CONTROLS 0 is entered and the memory reserved for GUI controls is returned. Usage reported by the MMBasic Memory Command
1000 0000 – 1000 FFFF	SRAM	64K	SRAM used by MMBasic Firmware for its own variables, C stack, buffers, command history etc.
080E 0000 – 080F FFFF	Flash	128K	Used by MMBasic to store MMBasic Library.
080C 0000 – 080D FFFF	Flash	128K	Used by MMBasic to store MMBasic programs. Usage reported by the MMBasic Memory Command
0800 C000 – 080B FFFF	Flash	720K	Next section of Flash. Used to store the remainder of the MMBasic Firmware.
0800 8000 – 0800 BFFF	Flash	16K	MMBasic SAVED VARS are stored here
0800 4000 – 0800 7FFF	Flash	16K	MMBasic OPTIONs are stored here
0800 0000 - 0800 3FFF	Flash	16K	First 16K of 1024K of Flash. Used to store the first part of MMBasic Firmware.
0000 0000 – 000F FFFF	Ram/Flash		Mapped to various Ram and Flash addresses used by the chip. Not used by MMBasic.

Startup and Reset – Quick Reference

This table provides a quick reference to the various start up and reset modes available for MMBasic and the Armmite F4.

Operation	Details
Normal Startup	A normal start up is initiated by power being applied, the RST button being pressed or the CPU RESTART command. Each of these will cause MMBasic to restart. It will look for some key Options to be set which indicate that the options have been previously saved and are valid. e.g. checks Baudrate is not 0, that a TAB option is set. If the options pass this integrity check then MMBasic is started using these previously saved Options. MM.INFO(BOOT) will return the reason for the restart. If the Options are seen as not yet set, then the default Options are loaded. This is the case after new firmware is loaded. The MM.STARTUP routine is searched for and if found any code in it is then executed. If OPTION AUTORUN ON is set then any program in the program memory is run, otherwise the command prompt is shown.
Set Default Options	The default Options are set as above if the Options are not deemed valid at startup. The OPTION RESET command will also set the default Options. This means the USB Console is enabled. MMBasic will restart to the command prompt.
MMBasic Reset	This will clear any save variables, set the default Options and clear the program memory and the Library. This is achieved by initiating a restart with Key 1 held down or PE3 connected to Gnd. Setting the default options means the USB Console is enabled.
Serial Console Startup	Holding Key 0 or connecting PE4 to Gnd while initiating a restart will enable the Serial Console and disable the USB Console.
Firmware Upgrade	Loading new firmware ***will*** clear any saved variables, the program memory, the library and reset the Options. Connecting BT0 to 3.3v and restarting will put the Armmite in the bootloader mode, ready to accept new firmware.
Startup with Embedded Option Commands	See the section Embedding Configuration Options in a Program for details. The main program has OPTION commands embedded at the start of the prrogram, these are set but not saved and no restart occurs until the OPTION SAVE command is found. These Options can include OPTION LCDPANEL etc commands that are not normally allowed in a program. Once the OPTION SAVE is reached MMBasic is restarted and starts with the new options applied. If the MM.STARTUP routine includes OPTION AUTORUN ON the main program is executed again.. On this restart the OPTIONs are already set and OPTION SAVE does not cause a restart so the rest of the program is run.

Predefined Read Only Variables

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

MM.CMDLINE$	This constant variable containing any command line arguments passed to the current program is automatically created when an MMBasic program runs; see RUN and * commands for details .• Programs run from the Editor or using OPTION AUTORUN will set MM.CMDLINE$ to the empty string. • If not required this constant variable may be removed from memory using ERASE MM.CMDLINE$
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 F407xGT" on the ArmmiteF4s with 1Meg of Flash "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 "WebMite" for the Raspberry Pico Web Edition “MMBasic for Windows” on the windows version
MM.FONTHEIGHT MM.FONTWIDTH	Integers representing the height and width of the current font (in pixels). *Allowed syntax but saved as MM.INFO(FONTHEIGHT)* *Allowed syntax but saved as MM.INFO(FONTWIDTH)*
MM.HPOS MM.VPOS	The current horizontal and vertical position (in pixels) following the last graphics command.
MM.HRES MM.VRES	Integers representing the horizontal and vertical resolution of the LCD display panel (if configured) in pixels.
MM.I2C	Following an I^2C write or read command this integer variable will be set to indicate the result of the operation as follows: 0 = The command completed without error. 1 = Received a NACK response 2 = Command timed out Following and I2C CHECK *addr* command will be set as follows” 0 = if a device responds at the address.

	1 = if no response
MM.INFO$ MM.INFO	These two versions can be used interchangeably but good programming practice would require that you use the one corresponding to the returned datatype.
MM.INFO$(AUTORUN)	Returns “On” or “Off” depending on the status of OPTION AUTORUN
MM.INFO$(BACKUP)	Returns the start address of the 3K of battery backed RAM. Use PEEK and POKE to read and write data, peek and poke support INTEGER and FLOAT (8 bytes each and must be on 8 byte boundary) as well as WORD (4-byte boundary) and BYTE.e.g. POKE FLOAT MM.INFO(BACKUP)+16, 72.345 ? PEEK(FLOAT MM.INFO(BACKUP)+16) 72.345
MM.INFO(BOOT)	Returns the reason for the last MMBasic restart. i.e. “Power On”, ”Reset Switch”, “MMBasic Reset”, ”CPU RESTART”, “Watchdog”, “EXECUTE Timout” or “HEAP Restart”. This could be written to a log file to see the history of an unattended device.
MM.INFO (CONSOLE)	Returns “NOCONSOLE” if OPTION LCDPANEL NOCONSOLE is set. Returns “CONSOLE” if OPTION LCDPANEL CONSOLE is set. Can be used in a program to determine if the the LCDPanel is being used as the console.
MM.INFO$(CPUSPEED)	Returns the CPU speed as a string
MM.INFO$(DEVICE)	Returns a string representing the device or platform that MMBasic is running on. See MM.DEVICE$ above.
MM.INFO(EXISTS DIR dir$)	Returns a Boolean indicating whether the directory specified exits
MM.INFO(EXISTS FILE file$)	Returns 1 if the file specified exists, returns -2 if fname$ is a directory, otherwise returns 0
MM.INFO$(FONT ADDRESS n)	Returns the actual address of the memory location containing the start of the binary data defining FONT n
MM.INFO$(FONT POINTER n)	Returns a POINTER to the start of the FONT n in memory
MM.INFO$(FCOLOUR) MM.INFO$(BCOLOUR)	Returns the current foreground colour Returns the current background colour
MM.INFO$(FONTHEIGHT) MM.INFO$(FONTWIDTH)	Integers representing the height and width of the current font (in pixels).

MM.INFO$(WIDTH) MM.INFO$(HEIGHT)	The current column width expected for the attached VT100 terminal The current number of lines expected for the attached VT100 terminal i.e. what has been set by OPTION DISPLAY
MM.INFO$(HPOS) MM.INFO$(VPOS)	The current horizontal and vertical position (in pixels) following the last graphics or print command.
MM.INFO$(LCDPANEL)	Returns the name of the LCD panel configured or a blank string
MM.INFO$(LINE)	Returns the current line number as a string. Returns UNKNOWN if called from the command prompt and LIBRARY if current line is within the Library. Assists in diagnostics while unit testing.
MM.INFO$(OPTION option)	Returns the current value of a range of options that affect how a program will run. “option” can be one of AUTORUN, BASE, BREAK, DEFAULT, EXPLICIT,ANGLE,TOUCH_IRQ,FLASH_CS See Option Settings for the values expected for each one.
MM.INFO$(PINNO P[A-E]n)	Returns the physical pin number for the given port. .e.g MM.INFO(PINNO PE2) returns 97.
MM.INFO$(PIN pinno)	Returns the status of I/O pin “pinno”. Valid returns are: “Invalid”, “Reserved”, “In Use”, and “Unused”
MM.INFO(RESTART)	Returns 1 if OPTION SAVE has caused a restart of MMBasic when a program has embedded options. See Running Armmite F4 without Backup Battery
MM.INFO$(SDCARD)	Returns status of SDCARD.Valid results are: “Not Present” if no SD Card and “Ready” if SD Card is inserted.
MM.INFO$(TOUCH)	Returns the status of the Touch controller. Valid returns are: “Disabled”, “Not calibrated”, and “Ready”
MM.INFO(VARCNT)	Returns the number of variables in use in the MMBasic program
MM.INFO(VCC)	Returns the current setting of OPTION VCC
MM.INFO(VERSION)	Returns the version number as a floating point number
MM.ERRNO MM.ERRMSG$	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.ERRNO MM.ERRMSG$	If a statement involving the SD card fails MM.ERRNO and MM.ERRMSG$ are set. They are reset to zero and an empty string by RUN, ON ERROR IGNORE or ON ERROR SKIP.The possible values for MM.ERRNO and associated MM.ERRMSG$ are: 1 = Low level I/O error 2 = Assertion failed 3 = SD Card not found 4 = Could not find the file 5 = Could not find the path 6 = The path name format is invalid 7 = Prohibited access or not empty 8 = Exists or path to it not found 9 = The file/directory is invalid 1 10 = SD Card is write protected 11 = The drive number is invalid 12 = The volume has no work area 13 = Not A FAT volume 14 = Format aborted 15 = Could not access volume 16 = File sharing policy 17 = Buffer could not be allocated 18 = Too many open files 19 = Parameter is invalid 20 = SD card not present
MM.ONEWIRE	Following a 1-Wire reset function this integer variable will be set to indicate the result of the operation as follows: 0 = Device not found. 1 = Device found
MM.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.WATCHDOG	An integer which is true if MMBasic was restarted as the result of a Watchdog timeout (see the WATCHDOG command). False if MMBasic started up normally.

Option Settings

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 Armmite F4 is restarted. Options that are not permanent will be reset on startup.

	Permanent	
OPTION ANGLE RADIANS | DEGREES		This command switches trig functions between degrees and radians. Acts on SIN, COS, TAN, ATN, ATAN2, MATH ATAN3, ACOS,ASIN
OPTION AUTORUN OFF | ON	✓	Instruct MMBasic to automatically run the program stored in flash when it starts up or is restarted by the WATCHDOG command. This is turned off by the NEW and LIBRARY SAVE commands but other commands that might change program memory (EDIT, etc) do not change this setting. Entering the break key (default CTRL-C) at the console will interrupt the running program and return to the command prompt despite this option.
OPTION BASE 0 | 1		Set the lowest value for array subscripts to either 0 or 1. This must be used before any arrays are declared and is reset to the default of 0 on power up.
OPTION BAUDRATE nbr	✓	Set the baud rate for the console to 'nbr'. This change is made immediately and will be remembered even when the power is cycled. The baud rate should be limited to the speeds listed in Appendix A for COM1. Using this command it is possible to set the console to an unworkable baud rate and in this case MMBasic should be reset as described in the chapter "Resetting MMBasic". This will reset the baud rate to the default of 11520
OPTION BREAK nn		Sets 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. It is not permanent and must be included in the program. It has no affect at the command line.
OPTION CASE UPPER | LOWER | TITLE	✓	Change the case used for listing command and function names when using the LIST command. The default is TITLE but the old standard of MMBasic can be restored using OPTION CASE UPPER. This option will be remembered even when the power is removed.
OPTION COLOURCODE OFF or OPTION COLOURCODE ON	✓	Turn on or off colour coding for the editor's output. Keywords will be in cyan, numbers in red, etc. The default is OFF. Notes: • Colour coding requires a terminal emulator that can interpret the appropriate escape codes. It works correctly with Tera Term however, Putty needs its default background colour to be changed to white. • If colour coding is used it is recommended that the baud rate for the serial console be set to a high speed. The keyword COLORCODE (USA spelling) can also be used.
OPTION CONTROLS nn	✓	Set the maximum number of controls that can be created by a program to 'nn'. This can be any number from 1 to 1000. The

		default is 200. A larger number will use more RAM (each control entry uses about 50 bytes of RAM). This command can only be run from the command line and the new value will remembered, even when the power is cycled or a new program loaded. ***Changing this option initiates a restart automatically***
OPTION DEFAULT FLOAT | INTEGER | STRING | NONE		Used to set the default type for a variable which is not explicitly defined. If OPTION DEFAULT NONE is used then all variables must have their type explicitly defined. When a program is run the default is set to FLOAT for compatibility with previous versions of MMBasic.
OPTION DISPLAY [lines [,chars]]	✓	Sets the characteristics of the display terminal used for the console. The LIST and EDIT commands need to know the dimensions of the terminal to correctly format the text for display. Commandline wrapping on the console is also dependant on the terminal width. 'lines' is the number of lines on the display and 'chars' is the width of the display in characters. The default is 24 lines x 80 chars and when changed this option will be remembered even when the power is removed. OPTION DISPLAY with no parameters will send an ESC sequence to the connected VT100 terminal that sets the display size to match MMBasic's current display HEIGHT and WIDTH+1. This could be used at startup to ensure the terminal matches the MMBasic settings. TeraTerm, Putty and MMCC are known to respond to this ESC sequence. Other VT100 terminals will need to be manually changed to the correct terminal width if possible. The Armmite F4, H7 and Picomites all set the terminal width 1 extra character wide. e.g OPTION DISPLAY 24,80 will actually set the terminal to 24,81. If you set the terminal manually you should allow this extra character. Note that the documentation for the VT100 ASCII Video Terminal initially listed incorrect specifications for the composite video. If you are using this project with the Armmite/Micromite check the website http://geoffg.net/terminal.html for the correct specifications. Note that the documentation for the VT100 ASCII Video Terminal initially listed incorrect specifications for the composite video. If you are using this project with the Armmite/Micromite check the website http://geoffg.net/terminal.html for the correct specifications.
OPTION ESCAPE		Enables the ability to insert escape sequences into string constants. See the section Special Characters in Strings.
OPTION EXPLICIT		Placing this command at the start of a program will require that every variable be explicitly declared using the DIM command before it can be used in the program. This option is disabled by default when a program is run. If it is used it must be specified before any variables are used.
OPTION FLASH_CS	✓	Defines the F_CS pin for the onboard W25Q16 flash chip. Defaults to 35 the pin on th VET6 board. 77 is allowed and is the F_CS on the VET6 MINI board. The pin is pulled high at start up. 0 is allowed and will not configure the F_CS pin and is will be available for other use, however the W25Q16 SPI MISO and MOSI pins will be across SPI1 and may interfer with the SPI bus as its F_CS pin is floating or used as something else.

OPTION KEYBOARD nn	✓	Enable an attached PS2 keyboard and set its language type. '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 , UK (United Kingdom), FR (French), GR (German), BE (Belgium), IT (Italian) or ES (Spanish). OPTION KEYBOARD DISABLE will disable the keyboard and return the I/O pins to normal use. See the section "*PS2 Keyboard*" for details of connecting the keyboard. This command can only be run from the command line and the new value will be remembered, even when the power is cycled or a new program loaded.
OPTION LCDPANEL controller, orientation, D/C pin, reset pin [,CS pin][,INVERT] or OPTION LCDPANEL DISABLE	✓	Configures the Armmite F4 to work with an SPI LCD panel. 'controller' can be: • ILI9481 SPI based 480*320 SPI touch controller • ILI9341 SPI based 2.2", 2.4" and 2.8" panels using the ILI9341 controller and touch controller • ILI9488 SPI based 480*320 SPI touch controller • ST7789 SPI based 240*240 IPS display • ST7735 SPI based 160*128 display • ST7735S SPI based 160*80 IPS display • GC9A01 SPI based 240*240 IPS display 'orientation' can be LANDSCAPE, PORTRAIT, RLANDSCAPE or RPORTRAIT. These can be abbreviated to L, P, RL or RP. The R prefix indicates the reverse or "upside down" orientation. **SPI based panels:** 'C/D pin' and 'reset pin' are the Armmite I/O pins to be used for these functions. Any free pin can be used. 'CS pin' can also be any I/O pin but is optional. If a touch controller is not used this parameter can be left off the command and the CS pin on the LCD display wired permanently to ground. If the touch controller is used this pin must then be specified and connected to an Armmite pin. INVERT is a literal string indicating colours should be inverted for ILI9488 and ILI9341 displays.
OPTION LCDPANEL controller, orientation or OPTION LCDPANEL DISABLE	✓	Configures the Armmite F4 to work with an attached parallel 16 bit bus LCD panel. 'controller' can be: • ILI9341_P16 3.2" ILI9341 320*240 16bit parallel controller • ILI9486_P16 3.5" ILI9486 480*320 16bit parallel controller • SSD1963_4_16 4.3" panels using the SSD1963 controller • SSD1963_5_16 5" panels using the SSD1963 controller • SSD1963_5A_16 alternative version of the 5" panel • SSD1963_5ER_16 ER (East Rising) version of the 5" panel • SSD1963_7_16 7" panels using the SSD1963 controller • SSD1963_7A_16 alternative version of the 7" panel • SSD1963_7ER_16 ER (East Rising) version of the 7" panel

		• SSD1963_8_16 8" and 9” panels using the SSD1963 controller • IPS_4_16 3.97” IPS 800*480 display. Covers both the OTM8009A and NT35510 displays. 'orientation' can be LANDSCAPE, PORTRAIT, RLANDSCAPE or RPORTRAIT. These can be abbreviated to L, P, RL or RP. The R prefix indicates the reverse or "upside down" orientation. **The pins used are fixed, but available to MMBasic if not used** This command only needs to be run once as the parameters are stored in non volatile memory. When the Armmite is restarted the display will be automatically initialise ready for use. If the LCD panel is no longer required, the command OPTION LCDPANEL DISABLE can be used to disable the LCD panel.
OPTION LCDPANEL CONSOLE [font [, fc [,bc [,blight[,NOSCROLL]]]]] OPTION LCDPANEL NOCONSOLE	✓	Configures the LCD display panel for use as the console output. The LCD can be any of the parallel supported LCD panels in the landscape or reverse landscape orientation and it must be first configured using OPTION LCDPANEL xxxxxxx (above). 'font' is the default font, 'fc' is the default foreground colour, 'bc' is the default background colour and 'blight' is the default backlight brightness (2 to 100). These parameters are optional and default to font 2, white, black and 100%. These settings are applied at power up. Colour coding in the editor is also turned on by this command (OPTION COLOURCODE OFF will turn it off again). OPTION LCDPANEL CONSOLE 4 will change the font used for the console to font 4 If the NOSCROLL option is set for non SSD1963 16bit parallel displays (IPS_4_16, ILI9341_16) when output reaches the bottom of the screen the screen is cleared and output continues again at the top. This improves console performance considerably at the expense of losing the display of data from some long outputs. e.g. OPTION LCDPANEL CONSOLE 1 , , , , NOSCROLL would enable rewriting in lieu of scrolling and OPTION LCDPANEL CONSOLE 1 would re-enable scrolling. This setting is saved in flash and will be automatically applied on startup. To disable it use the OPTION LCDPANEL NOCONSOLE command.
OPTION LIST		This will list the settings of any options that have been changed from their default setting and are the type that is saved in flash. This command is useful when configuring options that reserve I/O pins (ie, OPTION LCDPANEL or OPTION TOUCH) and you need to know what pins are in use.
OPTION MILLISECONDS OFF		Default. The time$ function returns the time as “HH:MM:SS”
OPTION MILLISECONDS ON		The time$ function returns the time as “HH:MM:SS.MMM”
OPTION PIN nbr	✓	Set 'nbr' as the PIN (Personal Identification Number) for access to the console prompt. 'nbr' can be any non zero number of up to eight digits. Whenever a running program tries to exit to the command prompt for

		whatever reason MMBasic will request this number before the prompt is presented. This is a security feature as without access to the command prompt an intruder cannot list or change the program in memory or modify the operation of MMBasic in any way. To disable this feature enter zero for the PIN number (ie, OPTION PIN 0). A permanent lock can be applied by using 99999999 for the PIN number. If a permanent lock is applied or the PIN number is lost the only way to recover is to reset MMBasic as described in the section Resetting MMBasic (this will also erase the program memory).
OPTION RESET		Reset all saved options (including the PIN) to the default values. Restores these two options on the Armmite F4 OPTION LCDPANEL ILI9341_16, RLANDSCAPE OPTION TOUCH PB12, PC5
OPTION RTC CALIBRATE ±n	✓	Used to calibrate the battery backed Real Time Clock that keeps time in the Armmite F4. 'n' is a number between -511 and + 512. A change of ±1 should equate to about 0.0824 seconds per day. Negative numbers will slow the clock down, positive will speed it up (different from the Micromite). This setting is remembered even after a firmware upgrade.
OPTION SERIAL CONSOLE ON	✓	Enable the serial console. When the serial console is enabled COM1 is not available. This command can only be run from the command line and will cause a restart so if the command was issued via the USB console the connection will be lost and will need to be re-established. The new value will remembered, even when the power is cycled or a new program loaded. .
OPTION SERIAL CONSOLE OFF	✓	Disable the serial console. When the console is disabled (the default) the serial port can be opened as COM1 and the console is restored to the USB. This command can only be run from the command line and will cause a restart.
OPTION SERIAL PULLUP ENABLE OPTION SERIAL PULLUP DISABLE	✓	Permanently stored option that enables pullup resistors on receive line (RX) of all serial ports including the serial console. The default is enabled. Disables pullups on all serial ports ***Switching between these options initiates a restart automatically.***
OPTION TAB 2 | 3 | 4 | 8	✓	Set the spacing for the tab key. Default is 2. This option will be remembered even when the power is removed.
OPTION TOUCH T_CS pin, T_IRQ pin OPTION TOUCH **PB12, PC5** OPTION TOUCH DISABLE	✓	Configures the Armmite F4 to suit the touch sensitive feature of an attached LCD panel. 'T_CS pin' and 'T_IRQ pin' are hardwired and must be PB12 for T_CS and PC5 for T_IRQ for the 16 bit displays using the FSMC connector. It is possible to use other pins for the SPI displays if desired. This command only needs to be run once as the parameters are stored in non volatile memory. Every time the Armmite is restarted MMBasic will automatically initialise the touch controller. If the touch facility is no longer required, the command OPTION

		TOUCH DISABLE can be used to disable the touch feature and return the I/O pins for general use.
OPTION VCC voltage		Specifies the voltage (Vcc) supplied to the STM32 chip. When using the analog inputs to measure voltage the STM32 chip uses its supply voltage (Vcc) as its reference. This voltage can be accurately measured using a DMM and configured using this command to allow for a more accurate measurement. The parameter is not saved and should be initialised either on the command line or in a program. The default if not set is 3.3.

Commands

Square brackets indicate that the parameter or characters are optional.

‘ (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
*file [options]	The star/asterisk command is a shortcut for RUN that may only be used at the MMBasic prompt. e.g. * RUN *foo RUN "foo" *"foo bar" RUN "foo bar" *foo –wombat RUN "foo", "--wombat" *foo "wom" RUN "foo", CHR$(34) + "wom" + CHR$(34) *foo --wom="bat" RUN "foo","--wom=" + CHR$(34) + "bat" + CHR$(34) String expressions are not supported/evaluated by this command; any arguments provided are passed as a literal string to the RUN command See RUN command for further detail.
/* */	Starts a multiline comment. Ends a multiline comment. This is properly supported including colour coding in the editor but note /* and */ must be the first non-space characters at the start of a line and have a space or end-of-line after them (i.e. they are MMBasic commands). Any characters after */ on a line are also treated as comment. *Multi-line comments can't be used inside subroutines and functions - this is a limitation of the way they work.*
? (question mark)	Shortcut for the PRINT command.
ADC	The ADC functionality can capture up to 3 channels of analog data in the background at up to 500KHz per channel with user selectable triggering.
ADC OPEN frequency, channel1-pin [,channel2-pin] [,channel3-pin] [, interrupt]	Open the ADC channels. "frequency" is the sampling frequency in Hz. Above 320KHz the conversion is 8-bits per channel Above 160KHz to 320KHz the conversion is 10-bits per channel From 160KHz and below the conversion is 12-bits per channel This is automatically applied in the firmware. 'channel1-pin' can be one of PC0,PC3,PA0,PA1,PA2,PA3,PA6,PA7,PB0 [15,18,23,24,25,26,31,32,35] *** PB0 [35] not available on VET6 as it is the Flash CS pin. Can be used on the VET6 Mini. 'channel2-pin' must be PC2 [17] 'channel3-pin' can be one of PC1,PC4,PC5 [16,33,34] 'interrupt' is a normal MMBasic subroutine that will be called when the conversion completes.

ADC FREQUENCY frequency	Allows the ADC frequency to be adjusted after the ADC START command. This command is only valid if the number of bits calculated in the table above does not change.
ADC TRIGGER channel, level[,timeout]	Sets up triggering of the ADC. This should be specified before the ADC START command. The 'channel' can be a number between one and three depending on the number of pins specified in the ADC OPEN command. The 'level' can be between -VCC and VCC. A positive number indicates that the trigger will be on a positive going transition through the specified voltage. A negative number indicates a negative going transition through the specified voltage. 'timeout' is the number of ADC samples to take before abandoning the wait for the trigger condition. Setting a value equal to the frequency of the sampling would give a timeout of 1 second.
ADC START array1!() [,array2!()] [,array3!()] [,C1min] [,C1max] [,C2min] [,C2max] [,C3min] [,C3max]	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. Start can be called repeatedly once the ADC is OPEN 'Cxmin' and 'Cxmax' will scale the readings. For example, C1min=200 and C1max=100 will create values ranging from 200 to 100 for equivalent voltages of 0 - 3.3. If the scaling is not used the results are returned as a voltage between 0 and OPTION VCC (defaults to 3.3V).
ADC CLOSE	Closes the ADC and returns the pins to normal use
ARC x, y, r1, [r2], rad1, rad2, colour	Draws an arc of a circle or a given colour and width between two radials (defined in degrees). Parameters for the ARC command are: 'x' is the X coordinate of the centre of arc. 'y' is the Y coordinate of the centre of arc. 'r1' is the inner radius of the arc. 'r2' is the outer radius of the arc - can be omitted if 1 pixel wide. 'rad1' is the start radial of the arc in degrees. 'rad2' is the end radial of the arc in degrees. 'colour' is the colour of the arc.
AUTOSAVE or AUTOSAVE CRUNCH or AUTOSAVE APPEND	Enter automatic program entry mode. This command will take lines of text from the console serial input and save them to memory. This mode is terminated by entering Control-Z or F1 which will then cause the received data to be saved 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. The APPEND option will leave the existing program intact and append the new data from the serial input to the end of it 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 Armmite. 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.

BACKLIGHT percentage%[,DEFAULT|REVERSE]	Sets to intensity of the backlight on LCD Display by either sending a command to the SSD1963 panels or changing the PWM signal to the BL pin on the other LCD Displays 0 is off, 100 is full intensity for most displays. This may be reversed for some displays depending on how the LED driver is implemented. A value somewhere between can be used to minimise power drawn while still giving a readable display. The option DEFAULT will also update the default value which is then used at any future restarts or power ups. The option REVERSE will also set the default value which is then used at any future restarts or power ups, but will also indicate that the backlight is to produce the reverse order for brightness. i.e. 0-100 produces a 100-0 output. This can be used to correct the brightness progression where the backlight driver of a particular display expects a reverse pwm signal and would other wise show 100% as OFF and 0% and ON.
BEZIER xs, ys, xc1, yc1, xc2, yc2, xe, ye, colour	Draws a cubic Bezier curve by specifying the start and end points and two control points. Parameters for the BEZIER command are: xs: X coordinate of start point ys: Y coordinate of start point xc1: X coordinate of first control point yc1: Y coordinate of first control point xc2: X coordinate of second control point yc2: Y coordinate of second control point xe: X coordinate of end point ye: Y coordinate of end point colour: Colour of curve
BITBANG (see DEVICE)	Replaced by the command DEVICE. For compatibility BITBANG can still be used in programs and will be automatically converted to DEVICE
BLIT READ [#]b, x, y, w, h BLIT WRITE [#]b, x, y,w,h BLIT CLOSE [#]b	Copy one section of the display screen to or from a memory buffer. BLIT READ will copy a portion of the display to the memory buffer '#b'. The source coordinate is 'x' and 'y' and the width of the display area to copy is 'w' and the height is 'h'. When this command is used the memory buffer is automatically created and sufficient memory allocated. This buffer can be freed and the memory recovered with the BLIT CLOSE command. BLIT WRITE will copy from the memory buffer '#b' to the display. The destination coordinate is 'x' and 'y' and the width and height use the passed parameters. BLIT CLOSE will close the memory buffer '#b' to allow it to be used for another BLIT READ operation and recover the memory used. Notes: • Sixty four buffers are available ranging from #1 to #64. • When specifying the buffer number the # symbol is optional. • All other arguments are in pixels.
BLIT x1, y1, x2, y2, w, h	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 and the source and destination can overlap.
BOX x, y, w, h [, lw] [,c] [,fill]	Draws a box on the LCD display with the top left hand corner at 'x' and 'y' with a width of 'w' pixels and a height of 'h' pixels. 'lw' is the width of the sides of the box and can be zero. It defaults to 1. 'c' is the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'c', and fill can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
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.
	See CAN Support for details on usage.
CAN OPEN index, speed, mode	Opens the CAN adapter using preset values determined by index at the desired speed.
CAN OPEN index,0, mode, prescaler, seg1, seg2, sjw	Long form of CAN OPEN which allows user to customise the speed and sample point.
CAN FILTER index, eid, type, config, id1, id2	Adds a filter to the CAN
CAN START	Starts the CAN receiving messages.
CAN STOP	Stops the CAN receiving messages. Also allows filters to be changed.
CAN SEND id, eid, rtr, dlc, msg, ret	Sends a message via the first available transmit buffer.
CAN READ fifo, id, eid, rtr, dlc, msg, fmi, ret	Reads a message from either FIFO0 or FIFO1 based on the value of fifo.
CAN CLOSE	Closed the CAN and releases the pins used for CAN-Rx and CAN-Tx

CAT S$, N$	CAT S$, N$ appends N$ to S$. This is functionally the same a S$ = S$ + N$ but operates faster. *CAT is an alias for the INC command and is stored internally as the INC command and will show in the program as INC.*
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 LCD display. 'lw' is optional and is the line width (defaults to 1). 'c' is the optional colour and defaults to the current foreground colour if not specified. The optional 'a' is a floating point number which will define the aspect ratio. If the aspect is not specified the default is 1.0 which gives a standard circle 'fill' is the fill colour. It can be omitted or set to -1 in which case the circle will not be filled. All parameters can be expressed as arrays and the software will plot the number of circles as determined by the dimensions of the smallest array. 'x', 'y' and 'r' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw', 'a', 'c', and fill can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
CLEAR	Delete all variables and recover the memory used by them.
CLOSE [#]nbr [,[#]nbr] …	Close the file(s) previously opened with the file number '#fnbr' Close the serial communications port(s) previously opened with the file number 'nbr'. The # is optional. Also see the OPEN command. The text "GPS" can be substituted for [#]nbr to close a communications port used for a GPS receiver.
CLS [colour]	Clears the LCDPANEL 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 (TEXT etc) that display on the on the LCDPANEL and accept a background or foreground colour parameter.. '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.
CPU RESTART	Will force a restart of the processor. This will clear all variables and reset everything (eg, timers, COM ports, I^2C, etc) similar to a power up situation but without the power up banner. If OPTION AUTORUN has been set the program will restart.
CPU SLEEP CPU SLEEP time	The CPU sleeps until there is a signal on the wakeup pin. Pin PA0 is the wakeup pin, however any other COUNT pin can also be used to wake the processor if it is enabled with SETPIN pinno, CIN or PIN or FIN. The Armmite uses the RTC to generate an interrupt to wake the processor after a period of sleep. Any period can be specified including fractions of seconds and because the RTC is used the timing will be accurate. Using the embedded ARMMITE F4 date and time functions makes it easy to sleep until any particular time. e.g. Midnight_tonight% = epoch(date$+" 00:00:00")+86400 'epoch at start of day today + secs in a day CPU SLEEP Midnight_tonight% - epoch(now) ' sleep until midnight tonight Note: *The R21 pullup resistor on the USB D+ data line prevents the CPU SLEEP [n] working when using a USB console.* See CPU SLEEP time section.
CFUNCTION name ([type [, type] …])[type] hex [[hex[…] hex [[hex[…] END CFUNCTION	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 function 'name' and can be used in the same manner as a built-in function. Multiple embedded routines can be used in a program with each defining a different module with a different 'name'. The first 'hex' word must be the offset (in 32-bit words) to the entry point of the embedded routine (usually the function main()). The following hex words are the compiled binary code for the module. These are automatically programmed into MMBasic when the program is saved. Each 'hex' must be exactly eight hex digits representing the bits in a 32-bit word and be separated by one or more spaces or new lines. The functions must be terminated by a matching END CFUNCTION. Any errors in the data format will be reported when the program is loaded into flash by the RUN command. During execution MMBasic will skip over any CFUNCTION/CSUB commands so they can be placed anywhere in the program. The type of each parameter *should* be specified in the definition. As well as defining the types of the parameters a CFunction can also specify the type of the value returned. For example, the following returns a float: ***CFunction MyFunction (integer, integer, string) float*** This specifies that there will be three parameters, the first two being integers and the third a string and the function will return a float. If type is specified then the type of the variables passed is checked and an error given if the

	expected type does not match. 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 CFUNCTION/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.
CSUB name [type [, type] …] hex [[hex[…] hex [[hex[…] END CSUB	Defines the binary code for an embedded machine code program module written in C or ARM assembler. The module will appear in MMBasic as the command 'name' and can be used in the same manner as a built-in command. Multiple embedded routines can be used in a program with each defining a different module with a different 'name'. The first 'hex' word must be the offset (in 32-bit words) to the entry point of the embedded routine (usually the function main()). The following hex words are the compiled binary code for the module. These are automatically programmed into MMBasic when the program is saved. Each 'hex' must be exactly eight hex digits representing the bits in a 32-bit word and be separated by one or more spaces or new lines. The command must be terminated by a matching END CSUB. Any errors in the data format will be reported when the program is loaded into flash by the RUN command. During execution MMBasic will skip over any CSUB commands so they can be placed anywhere in the program. The type of each parameter *may* 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. If type is specified then the type of the variables passed is checked and an error given if the expected type does not match. 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.
CTRLVAL(#ref) =	This command will set the value of an advanced control. '#ref' is the control's reference number. For off/on controls like check boxes it will override any touch input and can be used to depress/release switches, tick/untick check boxes, etc. A value of zero is off or unchecked and non zero will turn the control on. For a LED it will cause the LED to be illuminated or turned off. It can also be used to set the initial value of spin boxes, text boxes, etc. For example: `CTRLVAL(#10) = 12.4`
	Sets the DAC channel (1 or 2) to the voltage requested. This command

DAC n, voltage	cannot be used if the DACs are in use for audio output.
DAC START frequency, DAC1array%() [,DAC2array%()]	Sets up the DAC to create an arbitrary waveform. DAC1array%() and optional DAC2array%() should contain numbers in the range 0-4095 to suit the 12-bit DACs. Once started the output continues in the background and control returns to MMBasic. The software automatically and separately uses the number of items in each of the arrays to drive the DACs. The frequency is the rate at which the DACs change value. The maximum frequency is 700KHz. As an example if there are 180 items in the array c%() which are displayed at a frequency of 100,000 Hz this will give a waveform frequency of 100,000/180 = 555Hz. If there are 90 items in the array d%() at the same frequency of 100,000 Hz this will at the same time produce a waveform frequency of 100,000/90 = 1111Hz.
DAC STOP	Stops the DAC output and returns the DACs to normal use.
DATA constant[,constant]...	Stores numerical and string constants to be accessed by READ. In general string constants should be surrounded by double quotes ("). An exception is when the string consists of just alphanumeric characters that do not represent MMBasic keywords (such as THEN, WHILE, etc). In that case quotes are not needed. Numerical constants can also be expressions such as 5 * 60.
DATE$ = "DD-MM-YY" or DATE$ = "DD/MM/YY"	Set the date of the internal clock/calendar. DD, MM and YY are numbers, for example: DATE$ = "28-7-2024" The year can be abbreviated to two digits (ie, 24). The date is set to "01-01-2000" on first power up but the date will be remembered and kept updated as long as the battery is installed and can maintain a voltage of over 2.5VThe firmware looks for a date > 2018 before it allows the clock to run without reset on restart. Otherwise the firmware can't know that the clock has been properly initialised.
DEFINEFONT #n hex [[hex[…] hex [[hex[…] END DEFINEFONT	This will define an embedded font which can be used exactly same as the built in fonts (ie, selected using the FONT command or specified in the TEXT command). MMBasic must execute the font in order for it to be loaded. '#n' is the font's reference number (1 to 16). It can be the same as an existing font (except fonts 1, 6 and 7) and in that case it will replace that font. Each 'hex' must be exactly eight hex digits and be separated by spaces or new lines from the next. Multiple lines of 'hex' words can be used with the command terminated by a matching END DEFINEFONT.
DEVICE 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 3120 uSecs • 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.
DEVICE HUMID pin, tvar, hvar[,version]	Returns the temperature and humidity using the DHT22 or DHT11 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. Valid codes for version are: 1= DHT11 0 or omitted = DHT22 For example: DEVICE HUMID 2, TEMP!, HUMIDITY!,1 Temperature is measured in ºC and the humidity is percent relative humidity. Both will be measured with a resolution of 0.1. If an error occurs (sensor not connected or corrupt signal) both values will be 1000.0. Normally the signal pin of the DHT22 should be pulled up by a 1K to 10K resistor (4.7K recommended) to the supply voltage.
DEVICE LCD INIT d4, d5, d6, d7, rs, en or DEVICE LCD line, pos, text$ or DEVICE LCD CLEAR or DEVICE 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 DEVICE 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: `DEVICE LCD 1, C16, "Hello"` DEVICE LCD CLEAR will erase all data displayed on the LCD and DEVICE LCD CLOSE will terminate the LCD function and return all I/O pins to the not configured state. See the chapter LCD Display for more details.
DEVICE LCD CMD d1 [, d2 [, etc]] or DEVICE LCD DATA d1 [, d2 [, etc]]	These commands will send one or more bytes to an LCD display as either a command (DEVICE LCD CMD) or as data (DEVICE 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 DEVICE 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.

DEVICE 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 W =SK6812W (RGBW) '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. For the first three variants each element in the array should contain the colour in the normal RGB888 format (i.e. 0 to &HFFFFFF). For type W use a RGBW value (0-&HFFFFFFFF). If only one LED is connected then a single integer should be used for value% (ie, not an array).
DHT22	See the DEVICE HUMID command which replaces the DHT22 command.

DIM [type] decl [,decl]...
where 'decl' is:
var [length] [type] [init]
'var' is a variable name with optional dimensions
'length' is used to set the maximum size of the string to 'n' as in LENGTH n
'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword AS - as in AS FLOAT)
'init' is the value to initialise the variable and consists of:
= <expression>
For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.
 	
Examples:
DIM nbr(50)
DIM INTEGER nbr(50)
DIM name AS STRING
DIM a, b$, nbr(100), strn$(20)
DIM a(5,5,5), b(1000)
DIM strn$(200) LENGTH 20
DIM STRING strn(200)
	Declares one or more variables (ie, makes the variable name and its characteristics known to the interpreter).
	When OPTION EXPLICIT is used (as recommended) the DIM, LOCAL or STATIC commands are the only way that a variable can be created. If this option is not used, then using the DIM command is optional and if not used the variable will be created automatically when first referenced.
	The type of the variable (ie, string, float or integer) can be specified in one of three ways:
	By using a type suffix (ie, !, % or $ for float, integer or string). For example:
	`DIM nbr%, amount!, name$`
	By using one of the keywords FLOAT, INTEGER or STRING immediately after the command DIM and before the variable(s) are listed. The specified type then applies to all variables listed (ie, it does not have to be repeated). For example:
	`DIM STRING first_name, last_name, city`
	By using the Microsoft convention of using the keyword "AS" and the type keyword (ie, FLOAT, INTEGER or STRING) after each variable. If you use this method the type must be specified for each variable and can be changed from variable to variable. For example:
	`DIM amount AS FLOAT, name AS STRING`
	Floating point or integer variables will be set to zero when created and strings will be set to an empty string (ie, ""). You can initialise the value of the variable with something different by using an equals symbol (=) and an expression following the variable definition. For example:
	`DIM STRING city = "Perth", house = "Brick"`
	The initialising value can be an expression (including other variables) and will be evaluated when the DIM command is executed.
	As well as declaring simple variables the DIM command will also declare arrayed variables (ie, an indexed variable with up to four dimensions). Note that this is different from the Micromite versions of MMBasic which supported up to eight dimensions.
	Following the variable's name the dimensions are specified by a list of numbers separated by commas and enclosed in brackets. For example:
	`DIM array(10, 20)`

LENGTH 20
DIM a = 1234, b = 345
DIM STRING strn = "text"
DIM x%(3) = (11, 22, 33, 44)
	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 Armmite F4 requires 8 bytes a total of 1848 bytes of memory will be allocated.
	Strings will default to allocating 255 bytes (ie, characters) of memory for each element and this can quickly use up memory when defining arrays of strings. In that case the LENGTH keyword can be used to specify the amount of memory to be allocated to each element and therefore the maximum length of the string that can be stored. This allocation ('n') can be from 1 to 255 characters.
	For example: `DIM STRING s(5, 10)` will declare a string array with 66 elements consuming 16,896 bytes of memory while:
	`DIM STRING s(5, 10) LENGTH 20`
	Will only consume 1,386 bytes of memory. Note that the amount of memory allocated for each element is n + 1 as the extra byte is used to track the actual length of the string stored in each element.
	If a string longer than 'n' is assigned to an element of the array an error will be produced. Other than this, string arrays created with the LENGTH keyword act exactly the same as other string arrays.
	The LENGTH keyword can also be used with non array string variables , however variables of > 9 characters still take 256 bytes of memory and 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, also implemented in MMBasic). If, at the start, the expression is false the statements in the loop will not be executed, not even once.
DO <statements> LOOP UNTIL expression	Loops until the expression following UNTIL is true. Because the test is made at the end of the loop the statements inside the loop will be executed at least once, even if the expression is true.
EDIT	Invoke the full screen editor. See the section *Full Screen Editor* for details of how to use the editor.

ELSE	Introduces a default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF expression THEN or ELSE IF expression THEN	Introduces a secondary condition in a multiline IF statement. See the multiline IF statement for more details.
END	End the running program and return to the command prompt.
END CSUB	Marks the end of a C subroutine. See the CSUB command. Each CSUB must have one and only one matching END CSUB statement.
END FUNCTION	Marks the end of a user defined function. See the FUNCTION command. Each function must have one and only one matching END FUNCTION statement. Use EXIT FUNCTION if you need to return from a function from within its body.

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

END SELECT	Marks the end of a SELECT CASE construction . see SELECT CASE
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 global variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (eg, `dat()`) or just by specifying the variable's name (eg, `dat`). Use CLEAR to delete all variables at the same time (including arrays).
ERROR [error_msg$]	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXECUTE commands$	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 : (colon) 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. Variable persistence can be achieved using VAR SAVE/RESTORE if required. In the case of the CMM2 and Armmite F4 these use 4K of battery backed RAM so can be done without

	consequence. In the case of the Armmite H7 saving is done to flash memory so care should be exercised to stay within the limits of the write capability of the flash chip "Min. 100K Program-Erase cycles per sector"
EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.
FILES [fspec$]	Lists files in the current directory on the SD card. 'fspec$' (if specified) can contain search wildcards. Question marks (?) will match any character and an asterisk (*) will match any number of characters. If omitted, all files will be listed. For example: *.* Find all entries *.TXT Find all entries with an extension of TXT E*.* Find all entries starting with E X?X.* Find all three letter file names starting and ending with X
FONT [#]font-number, scaling	This will set the default font for displaying text on the LCDPANEL. Fonts are specified as a number. For example, #2 (the # is optional) See the chapter "Basic Graphics" for details of the available fonts. 'scaling' can range from 1 to 15 and will multiply the size of the pixels making the displayed character correspondingly wider and higher. Eg, a scale of 2 will double the height and width. *OPTION LCDPANEL CONSOLE* The FONT command does not change the Prompt Font when OPTION LCDPANEL CONSOLE is enabled. OPTION LCDPANEL CONSOLE font should be used to change the font used by the console. See LCD Display as the Console Output for details.
FOR counter = start TO finish [STEP increment]	Initiates a FOR-NEXT loop with the 'counter' initially set to 'start' and incrementing in 'increment' steps (default is 1) until 'counter' is greater than 'finish'. The 'increment' can be an integer or floating point number. Note that using a floating point fractional number for 'increment' can accumulate rounding errors in 'counter' which could cause the loop to terminate early or late. 'increment' can be negative in which case 'finish' should be less than 'start' and the loop will count downwards. See also the NEXT command.
FTT	Now is part of the MATH command. See the MATH command
FUNCTION xxx (arg1 [,arg2, …]) [AS <type>} <statements> <statements> xxx = <return value> END FUNCTION	Defines a callable function. This is the same as adding a new function to MMBasic while it is running your program. 'xxx' is the function name and it must meet the specifications for naming a variable. The type of the function can be specified by using a type suffix (ie, xxx$) or by specifying the type using AS <type> at the end of the functions definition. For example: `FUNCTION xxx (arg1, arg2) AS STRING` 'arg1', 'arg2', etc are the arguments or parameters to the function (the brackets are always required, even if there are no arguments). An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using AS

	`<type> (ie, arg1 AS STRING).`
	The argument can also be another defined function or the same function if recursion is to be used (the recursion stack is limited to 50 nested calls).
	To set the return value of the function you assign the value to the function's name. For example:
	`FUNCTION SQUARE(a)`
	`  SQUARE = a * a`
	`END FUNCTION`
	Every definition must have one END FUNCTION statement. When this is reached the function will return its value to the expression from which it was called. The command EXIT FUNCTION can be used for an early exit.
	You use the function by using its name and arguments in a program just as you would a normal MMBasic function.
	For example:
	`PRINT SQUARE(56.8)`
	When the function is called each argument in the caller is matched to the argument in the function definition. These arguments are available only inside the function.
	Functions can be called with a variable number of arguments. Any omitted arguments in the function's list will be set to zero or a null string.
	Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the function. This means that any changes to the corresponding argument in the function will also be copied to the caller's variable. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference.
	You must not jump into or out of a function using commands like GOTO, GOSUB, etc. Doing so will have undefined side effects including the possibility of ruining your day.

GOSUB	See Obsolete Commands and Functions section.
GOTO target	See Obsolete Commands and Functions section. Branches program execution to the target, which can be a line number or a label.
*** GUI Controls ***	Used to implement a GUI display with touch support. See Advanced Graphics for details.
GUI AREA #ref, startX, startY, width, height	This will define an invisible area of the screen that is sensitive to touch and will generate touch down and touch up interrupts. It can be used as the basis for creating custom controls which are defined and managed by the program. '#ref' is the control's reference number. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions.
GUI BARGAUGE #ref, StartX, StartY, width, height, FColour, BColour, min, max, c1, ta, c2, tb, c3, tc, c4	Define either a horizontal or vertical analogue bar gauge. '#ref' is the control's reference number. 'StartX' and 'StartY' are the top left coordinates of the bar while 'width' is the horizontal width and 'height' the vertical height. If the width is less that the height the bar gauge will be drawn vertically with the graph growing from the bottom towards the top. Otherwise it will be drawn horizontally with the graph growing from the left towards the right. 'Fcolour' is the colour used for the gauge while 'Bcolour' is the background colour. 'min' is the minimum value of the gauge and 'max' is the maximum

	value (both floating point). A multi colour gauge can be created using 'c1' to 'c4' for the colours and 'ta' to 'tc' for the thresholds used to determine when the colour will change. 'width', 'height', 'FColour', 'BColour', 'min' and 'max' are optional and will default to the values used in the previous definition of a GUI BARGAUGE. 'c1', 'ta', 'c2', 'tb', 'c3', 'tc' and 'c4' are optional and if not specified the gauge will use less colours. If all are omitted, the gauge will be drawn using 'Fcolour'. The section *Advanced Graphics* has a more detailed description.
GUI BCOLOUR colour, #ref1 [, #ref2, #ref3, etc]	This will change the background colour of the specified controls to 'colour' which is an RGB value for the drawing colour. '#ref' is the control's reference number.
GUI BEEP msec	This will sound the pizeo buzzer if configured with the OPTION TOUCH command. 'msec' is the number of milliseconds that the buzzer should be driven. A time of 3ms produces a click while 100ms produces a short beep.
GUI BUTTON #ref, caption$, startX, startY, width, height [, FColour] [,BColour]	This will draw a momentary button which is a square switch with the caption on its face. When touched the visual image of the button will appear to be depressed and the control's value will be 1. When the touch is removed the value will revert to zero. #ref' is the control's reference (a number from 1 to 100). 'caption$' is the string to display on the face of the button. It can be a single string with two captions separated by a | character (e.g., "UP|DOWN"). When the button is up the first string will be used and when pressed the second will be used. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls or set with the COLOUR command.
GUI CAPTION #ref, text$, startX, startY [,align$] [, FColour] [, BColour]	This will draw a text string on the screen. '#ref' is the control's reference number. 'text$' is the string to display. 'startX' and 'startY' are the top left coordinates. 'align$' is zero to three characters (a string expression or variable is also allowed) where the first letter is the horizontal alignment around X and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around Y and can be T, M or B for TOP, MIDDLE, BOTTOM. A third character can be used in the string to indicate the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (ie, upside down), 'U' the text will be rotated counter clockwise by 90º and 'D' the text will be rotated clockwise by 90º. The default alignment is left/top with no rotation. 'FColour and 'BColour' are RGB values for the foreground and background colours. On a display that supports transparent text BColour can be -1 which means that the background will show through the gaps in the characters. FColour and 'BColour' are optional and default to the colours set by the

	COLOUR command.
GUI CHECKBOX #ref, caption$, startX, startY [, size] [, colour]	This will draw a check box which is a small box with a caption. When touched an X will be drawn inside the box to indicate that this option has been selected and the control's value will be set to 1. When touched a second time the check mark will be removed and the control's value will be zero. '#ref' is the control's reference number. The string 'caption$' will be drawn to the right of the control using the colours set by the COLOUR command. 'startX' and 'startY' are the top left coordinates while 'size' set the height and width (the bix is square). 'colour' is an RGB value for the drawing colour. 'size' and 'colour' are optional and default to that used in previous controls.
GUI DELETE #ref1 [,#ref2, #ref3, etc] or GUI DELETE ALL	This will delete the controls in the list. This includes removing the image of the control from the screen using the current background colour and freeing the memory used by the control. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will disable all controls.
GUI DISABLE #ref1 [,#ref2, #ref3, etc] or GUI DISABLE ALL	This will disable the controls in the list. Disabled controls do not respond to touch and will be displayed dimmed. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will disable all controls. GUI ENABLE can be used to restore the controls.
GUI DISPLAYBOX #ref, startX, startY, width, height, FColour, BColour	This will draw a box with rounded corners that can be used to display a string '#ref' is the control's reference number. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls. Any text can be displayed in the box by using the CtrlVal(r) = command. This is useful for displaying text, numbers and messages. This control does not respond to touch.
GUI ENABLE #ref1 [,#ref2, #ref3, etc] or GUI ENABLE ALL	This will undo the effects of GUI DISABLE and restore the control(s) to normal operation. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will disable all controls.
GUI FCOLOUR colour, #ref1 [, #ref2, #ref3, etc]	This will change the foreground colour of the specified controls to 'colour' which is an RGB value for the drawing colour. '#ref' is the control's reference number.
GUI FRAME #ref, caption$, startX, startY, width, height, colour	This will draw a frame which is a box with round corners and a caption. '#ref' is the control's reference number. 'caption$' is a string to display as the caption. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. 'colour' is an

	RGB value for the drawing colour. 'width', 'height' and 'colour' are optional and default to that used in previous controls. A frame is useful when a group of controls need to be visually brought together. It is also used to surround a group of radio buttons and MMBasic will arrange for the radio buttons surrounded by the frame to be exclusive. ie, when one radio button is selected any other button that was selected and within the frame will be automatically deselected. A frame does not respond to touch.
GUI FORMATBOX #ref, Format, startX, startY, width, height, FColour, BColour	This will draw a box with rounded corners that can be used to create a virtual keypad for entry of data using a specific format. '#ref' is the control's reference number. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls. The 'Format' argument specifies the format of the entry as follows: DATE1 Date in UK/Aust/NZ format (dd/mm/yy) DATE2 Date in USA format (mm/dd/yy) DATE3 Date in international format (yyyy/mm/dd) TIME1 Time in 24 hour notation (hh:mm) TIME2 Time in 24 hour notation with seconds (hh:mm:ss) TIME3 Time in 12 hour notation (hh:mm AM/PM) TIME4 Time in 12 hour notation with seconds (hh:mm:ss AM/PM) DATETIME1 Date (UK fmt) and time (12 hour) (dd/mm/yy hh:mm AM/PM) DATETIME2 Date (UK fmt) and time (24 hour) (dd/mm/yy hh:mm) DATETIME3 Date (USA fmt) and time (12 hour) (mm/dd/yy hh:mm AM/PM) DATETIME4 Date (USA fmt) and time (24 hour) (mm/dd/yy hh:mm) LAT1 Latitude in degrees, minutes and seconds (dd° mm' ss" N/S) LAT2 Latitude with seconds to one decimal place (dd° mm' ss.s" N/S) LONG1 Longitude in degrees, minutes and seconds (ddd° mm' ss" E/W) LONG2 Longitude seconds to one decimal place (ddd° mm' ss.s" E/W) ANGLE1 Angle in degrees and minutes (ddd° mm') For example, this command: `GUI FORMATBOX #1, LAT1, 50, 50, 300, 50` would create a format box which would accept the entry of latitude in the format of dd° mm' ss" N/S. The value of CtrlVal(#1) would be a string which includes the numbers and separating characters. For example an entry of 17 degrees, 32 minutes and 1 second south would result in the string 17° 32' 01" S MMBasic will try to position the virtual keypad on the screen so as to not obscure the format box that caused it to appear. A pen down interrupt will be generated just before the keypad is deployed and a key up interrupt will be generated when the entry is complete and the keypad is hidden.
GUI FORMATBOX ACTIVATE #ref	This will cause the virtual keypad for the control '#ref' to be displayed under program control without the control being touched. It is the same as if the

	user touched the control except that the touch down interrupt is not generated.
GUI FORMATBOX CANCEL	This will dismiss a virtual keypad if it is displayed on the screen. It is the same as if the user touched the cancel key except that the touch up interrupt is not generated. If a keypad is not displayed this command will do nothing.
GUI GAUGE #ref, StartX, StartY, Radius, FColour, BColour, min, max, nbrdec, units$, c1, ta, c2, tb, c3, tc, c4	Define a graphical circular analogue gauge with a digital display in the centre. '#ref' is the control's reference number. 'StartX' and 'StartY' are the coordinates of the centre of the gauge, 'Radius' is the distance from the centre to the outer edge. 'min' is the minimum value of the gauge and 'max' is the maximum value (both floating point). 'nbrdec' specifies the number of decimal places to be used when drawing the digital value in the centre of the gauge. Under this 'units$' will be displayed. 'Fcolour' is the colour used for the gauge while 'Bcolour' is the background colour. A multi colour gauge can be created using 'c1' to 'c4' for the colours and 'ta' to 'tc' for the thresholds used to determine when the colour will change. When colours and thresholds are specified the background of the gauge will be drawn with a dull version of the colour at that level. Also the digital value will change to the colour specified by the current value. 'Radius', 'FColour', 'BColour', 'min', 'max', 'nbrdec' and 'units$' are optional and will default to the values used in the previous definition of a GUI GAUGE. 'c1', 'ta', 'c2', 'tb', 'c3', 'tc' and 'c4' are optional and if not specified the gauge will use less colours. If all are omitted the gauge will be drawn using 'Fcolour'. The section *Advanced Graphics* has a more detailed description.
GUI HIDE #ref1 [,#ref2, #ref3, etc] or GUI HIDE ALL	This will hide the controls in the list. Hidden controls do not respond to touch and will not be visible. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will hide all controls. GUI SHOW can be used to restore the controls.
GUI INTERRUPT down [, up]	This command will setup an interrupt that will be triggered on a touch on the LCD panel and optionally if the touch is released. 'down' is the subroutine to call when a touch down has been detected. 'up' is the subroutine to call when the touch has been lifted from the screen ('up' and 'down' can point to the same subroutine if required). Specifying the number zero (single digit) as the argument will cancel both of these interrupts. ie: `GUI INTERRUPT 0`
GUI LED #ref, caption$, centerX, centerY, radius, colour	This will draw an indicator light which looks like a panel mounted LED. A LED does not respond to touch. '#ref' is the control's reference number. The string 'caption$' will be drawn to the right of the control using the colours set by the COLOUR command. 'centerX' and 'centerY' are the coordinates of the centre of the LED and 'radius' is the radius of the LED. 'colour' is an RGB value for the drawing

	colour. 'radius' and 'colour' are optional and default to that used in previous controls. When a LED's value is set to a value of one it will be illuminated and when it is set to zero it will be off (a dull version of its colour attribute). The LED can be made to flash on then off by setting the value of the LED to a number greater than one which is the time in milliseconds that it should remain on. The colour can be changed with the GUI FCOLOUR command.
GUI NUMBERBOX #ref, startX, startY, width, height, FColour, BColour	This will draw a box with rounded corners that can be used to create a virtual numeric keypad for data entry. '#ref' is the control's reference number. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls. When the box is touched a numeric keypad will appear on the screen. Using this virtual keypad any number can be entered into the box including a floating point number in exponential format. The new number will replace the number previously in the box. The value of the control can set to a literal string (not an expression) starting with two hash characters. For example: `CtrlVal(nnn) = "##Enter Number"` and in that case the string (without the leading two hash characters) will be displayed in the box with reduced brightness. This can be used to give the user a hint as to what should be entered (called "ghost text"). Reading the value of the control displaying ghost text will return zero. When the control is used normally the ghost text will vanish. MMBasic will try to position the virtual keypad on the screen so as to not obscure the number box that caused it to appear. A pen down interrupt will be generated just before the keypad is deployed and a key up interrupt will be generated when the Enter key is touched and the keypad is hidden. Also, when the Enter key is touched the entered number will be evaluated as a number and the NUMBERBOX control redrawn to display this number.
GUI NUMBERBOX CANCEL	This will dismiss a virtual keypad if it is displayed on the screen. It is the same as if the user touched the cancel key except that the touch up interrupt is not generated. If a keypad is not displayed this command will do nothing.
GUI PAGE #n [,#n2, #n3, etc]	This will switch the display to show controls that have been assigned (via the GUI SETUP command) to the page numbers specified on the command line (#n, #n2, etc). Any controls that were displayed but are not on the current list of pages will be automatically hidden. Any controls on a page that was displayed on the old screen and is also specified in the new command will remain unaffected. The default when a program starts running is PAGE 1 and GUI SETUP 1. This means that if these commands are not used the program will run as normal showing all GUI controls that have been defined. See also the GUI SETUP command.
GUI RADIO #ref, caption$, centerX, centerY, radius, colour	This will draw a radio button with a caption. '#ref' is the control's reference number. The string 'caption$' will be drawn to the right of the control using the

	colours set by the COLOUR command. 'centerX' and 'centerY' are the coordinates of the centre of the button and 'radius' is the radius of the button. 'colour' is an RGB value for the drawing colour. 'radius' and 'colour' are optional and default to that used in previous controls. When touched the centre of the button will be illuminated to indicate that this option has been selected and the control's value will be 1. When another radio button is selected the mark on this button will be removed and its value will be zero. Radio buttons are grouped together when surrounded by a frame and when one button in the group is selected all others in the group will be deselected. If a frame is not used all buttons on the screen will be grouped together.
GUI REDRAW #ref1 [,#ref2, #ref3, etc] or GUI REDRAW ALL	This will redraw the controls on the screen. It is useful if the screen image has somehow been corrupted. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will first clear the screen then redraw all controls. This is useful if the whole screen needs to be refreshed.
GUI SETUP #n	This will allocate any new controls created to the page '#n'. This command can be used as many times as needed while GUI controls are being defined. The default when a program starts running is GUI SETUP 1. See also the GUI PAGE command.
GUI SHOW #ref1 [,#ref2, #ref3, etc] or GUI SHOW ALL	This will undo the effects of GUI HIDE and restore the control(s) to being visible and capable of normal operation. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will disable all controls.
GUI SPINBOX #ref, startX, startY, width, height, FColour, BColour, Step, Minimum, Maximum	This will draw a box with up/down icons on either end. When these icons are touched the number in the box will be incremented or decremented. Holding down the up/down icons will repeat the step at a fast rate. '#ref' is the control's reference number. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls. 'Step' sets the amount to increment/decrement the number with each touch. 'Minimum' and 'Maximum' set limits on the number that can be entered. All three parameters can be floating point numbers and are optional. The default for 'Step' is 1 and 'Minimum' and 'Maximum' if omitted will default to no limit.
GUI SWITCH #ref, caption$, startX, startY, width, height, FColour, BColour	This will draw a latching switch which is a square switch that latches when touched. '#ref' is the control's reference number. 'caption$' is a string to display as the caption on the face of the switch.

	'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls. When touched the visual image of the button will appear to be depressed and the control's value will be 1. When touched a second time the switch will be released and the value will revert to zero. Caption can consist of two captions separated by a | character (eg, "ON|OFF"). When this is used the switch will appear to be a toggle switch with each half of the caption used to label each half of the toggle switch.
GUI TEXTBOX #ref, startX, startY, width, height, FColour, BColour	This will draw a box with rounded corners that can be used to create a virtual keyboard for data entry '#ref' is the control's reference number. 'startX' and 'startY' are the top left coordinates while 'width' and 'height' set the dimensions. ' FColour and 'BColour' are RGB values for the foreground and background colours. 'width', 'height', FColour and 'BColour' are optional and default to that used in previous controls. On a display that supports transparent text BColour can be -1 which means that the background will show through the gaps in the characters. When the box is touched a QWERTY keyboard will appear on the screen. Using this virtual keyboard any text can be entered into the box including upper/lower case letters, numbers and any other characters in the ASCII character set. The new text will replace any text previously in the box. The value of the control can set to a string starting with two hash characters. For example: `CtrlVal(nnn) = "##Enter Filename"` and in that case the string (without the leading two hash characters) will be displayed in the box with reduced brightness. This can be used to give the user a hint as to what should be entered (called "ghost text"). Reading the value of the control displaying ghost text will return an empty string. When the control is used normally the ghost text will vanish. MMBasic will try to position the virtual keyboard on the screen so as to not obscure the text box that caused it to appear. A pen down interrupt will be generated just before the keyboard is deployed and a key up interrupt will be generated when the Enter key is touched and the keyboard is hidden.
GUI TEXTBOX ACTIVATE #ref	This will cause the virtual keyboard for the control ‘#ref’ to be displayed under program control without the control being touched. It is the same as if the user touched the control except that the touch down interrupt is not generated.
GUI TEXTBOX CANCEL	This will dismiss a virtual keyboard if it is displayed on the screen. It is the same as if the user touched the cancel key except that the touch up interrupt is not generated. If a keyboard is not displayed this command will do nothing.
*** GUI Commands ***	These GUI commands are used to configure LCDPANEL and TOUCH. Also includes the GUI BITMAP command to display a bitmap.
GUI BITMAP x, y, bits [, width] [, height] [, scale] [, c] [, bc]	Displays the bits in a bitmap on an LCD panel starting at 'x' and 'y' on an attached LCD panel. 'height' and 'width' are the dimensions of the bitmap as displayed on the LCD panel 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 ('bits') can be an integer or a string variable or constant and is drawn using the first byte as the first bits of the top line (bit 7 first, then bit 6, etc) followed by the next byte, etc. When the top line has been filled the next line of the displayed bitmap will start with the next bit in the integer or string. See the chapter Using an LCD Panel for a definition of the colours and graphics coordinates.
GUI CALIBRATE Or GUI CALIBRATE c1, c2, c3, c4, c5	This command is used to calibrate the touch feature on an LCD panel. It will display a series of targets on the screen and wait for each one to be precisely touched. See Calibrating the Touch Screen for details. The second version allows the calibration parameters to be entered directly without having to go through the manual calibration process. The parameters 'c1', 'c2', etc can be found by running a normal calibration process then using OPTION LIST which will list the parameters for that LCD panel. This is useful when the command is embedded in a program.
GUI RESET LCDPANEL	Will reinitialise the configured LCD panel. Initialisation is automatically done when the Micromite starts up but in some circumstances it may be necessary to interrupt power to the LCD panel (eg, to save battery power) and this command can then be used to reinitialise the display.
GUI TEST LCDPANEL	Will test the display feature on an LCD panel. With GUI TEST LCDPANEL an animated display of colour circles will be rapidly drawn on top of each other. Any character entered at the console will terminate the test
GUI TEST TOUCH	With GUI TEST TOUCH the screen will blank and wait for a touch which will cause a white dot to be placed on the display marking the touch position on the screen. Any character entered at the console will terminate the test
HUMID pin, tvar, hvar[,version]	Now DEVICE HUMID. This is accepted but saved as the new form of the command.
I2C	The I2C commands will send and receive data over an I^2C channel. I2C (no suffix) refers to channel 1 while commands I2C2 refer to channels 2 using the same syntax. Also see *Appendix B.*
I2C OPEN speed, timeout	Enables the I^2C module in master mode. 'speed' is the clock speed (in KHz) to use and must be one of 100 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 I^2C slave device. 'addr' is the slave's I^2C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the

	command (a stop condition will not be sent at the completion of the command) 'sendlen' is the number of bytes to send. 'senddata' is the data to be sent - this can be specified in various ways (all values sent will be between 0 and 255): • The data can be supplied as individual bytes on the command line. Example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25 • The data can be in a one dimensional array specified with empty brackets (ie, no dimensions). 'sendlen' bytes of the array will be sent starting with the first element. Example: I2C WRITE &H6F, 0, 3, ARRAY() The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 0, 3, STRING$
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I^2C slave device. 'addr' is the slave's I^2C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 'rcvlen' is the number of bytes to receive. 'rcvbuf' is the variable or array used to save the received data - this can be: • A string variable. Bytes will be stored as sequential characters in the string. • A one dimensional array of numbers specified with empty brackets. Received bytes will be stored in sequential elements of the array starting with the first. Example: I2C READ &H6F, 0, 3, ARRAY() A normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE I2C CHECK addr	Disables the master I^2C module and returns the I/O pins to a "not configured" state. They can then be configured using SETPIN. This command will also send a stop if the bus is still held. Will set MM.I2C to 0 if a device responds at the address. MM.I2C is set to 1 if no response. Will give an error if the I2C has not been opened.
I2C SLAVE	I2C SLAVE is supported. See Appendix B for usaged.
I2C2	As above but for 2nd I2C channel.
I2C2 SLAVE	As above but for 2nd I2C 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 only one true statement is allowed following the THEN keyword. If 'expr' is resolved to be false the single statement following the ELSE keyword will be executed. The 'THEN statement' construct can be also replaced with: GOTO linenumber | label'. This type of IF statement is all on one line.

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

INC a[, b]	Increments a by 1 by default. If b is supplied then a is adjusted by the value of b. If b is -ve then a is decremented by that amount. E.g. INC a, -1 b can be an expression. e.g. INC i,(i<10) will for example add 1 to i but to a maximum of 10 as the expression then becomes 0 and nothing further is added.
INPUT ["prompt$";] var1 [,var2 [, var3 [, etc]]]	Will take a list of values separated by commas (,) entered at the console and will assign them to a sequential list of variables. For example, if the command is: INPUT a, b, c And the following is typed on the keyboard: 23, 87, 66 Then a = 23 and b = 87 and c = 66 The list of variables can be a mix of float, integer or string variables. The values entered at the console must correspond to the type of variable. If a single value is entered a comma is not required (however that value cannot contain a comma). 'prompt$' is a string constant (not a variable or expression) and if specified it will be printed first. Normally the prompt is terminated with a semicolon (;) and in that case a question mark will be printed following the prompt. If the prompt is terminated with a comma (,) rather than the semicolon (;) the question mark will be suppressed.
INPUT #nbr, list of variables	Same as the normal INPUT command except that the input is read from a file previously opened for INPUT as '#fnbr' or a serial port previously opened for INPUT as 'nbr'. See the OPEN command.
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 Note that while the code within the 'myint' subroutine is running other interrupts are not serviced.
IR dev, key , int or IR CLOSE	Decodes NEC or Sony infrared remote control signals. An IR Receiver Module is used to sense the IR light and demodulate the signal. It should be connected to the IR pin (see the pinout tables). This command will automatically set that pin to an input. The IR signal decode is done in the background and the program will continue after this command without interruption. 'dev' and 'key' should be numeric variables and their values will be updated whenever a new signal is received ('dev' is the device code transmitted by the remote and 'key' is the key pressed).

	'int' is a user defined subroutine that will be called when a new key press is received or when the existing key is held down for auto repeat. In the interrupt subroutine the program can examine the variables 'dev' and 'key' and take appropriate action. The IR CLOSE command will terminate the IR decoder and return the I/O pin to a not configured state. Note that for the NEC protocol the bits in 'dev' and 'key' are reversed. For example, in 'key' bit 0 should be bit 7, bit 1 should be bit 6, etc. This does not affect normal use but if you are looking for a specific numerical code provided by a manufacturer you should reverse the bits. This describes how to do it: http://www.thebackshed.com/forum/forum_posts.asp?TID=8367 See the chapter "Special Hardware Devices" for more details.
IR SEND pin, dev, key	Generate a 12-bit Sony Remote Control protocol infrared signal. 'pin' is the I/O pin to use. This can be any I/O pin which will be automatically configured as an output and should be connected to an infrared LED. Idle is low with high levels indicating when the LED should be turned on. 'dev' is the device being controlled and is a number from 0 to 31, 'key' is the simulated key press and is a number from 0 to 127. The IR signal is modulated at about 38KHz and sending the signal takes about 25mS.
IRETURN	See obsolete commands.
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 (eg, '6' will return 6) or 10 for the * key and 11 for the # key. On 4x4 keypads the number 20 will be returned for A, 21 for B, 22 for C and 23 for D. The KEYPAD CLOSE command will terminate the keypad function and return the I/O pin to a not configured state. See the chapter Keypad Interface for more details.
KILL file$	Deletes the file or empty directory specified by ‘file$’. If there is an extension it must be specified.
LCD INIT d4,d5,d6,d7,rs,en or LCD line, pos, text$ or LCD CLEAR	Now DEVICE LCD. This is still accepted but is saved as the new command format.

or LCD CLOSE	
LCD CMD d1 [, d2 [, etc]] or LCD DATA d1 [, d2 [, etc]]	Now DEVICE LCD. This is still accepted but is saved as the new command format.
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`
LIBRARY SAVE LIBRARY DELETE LIBRARY LIST [ALL] LIBRARY RESTORE	The library is a special segment of program memory that can contain program code such as subroutines, functions and CFunctions. These routines are not visible to the programmer but are available to any program running on the Armmite F4 and act the same as built in commands and functions in MMBasic. See Program Initialisation, CFunctions and the Library earlier in this manual for a full explanation. LIBRARY SAVE will take whatever is in normal program memory, compress it (remove redundant data such as comments) and append it to the library area (main program memory is then empty). The code in the library will not show in LIST or EDIT and will not be deleted when a new program is loaded or NEW is used. LIBRARY DELETE will remove the library and recover the memory used. OPTION RESET also effectively removes the library because it removes the pointer to its location, however see LIBRARY CHECK below. LIBRARY LIST will list the contents of the library.ALL will prevent the listing pausing after each page. Note that any code in the library that is not contained within a subroutine or function will be executed immediately before a program is run. This can be used to initialise constants, set options, etc. LIBRARY RESTORE tests for the existence of library code and restores its pointer if the OPTION PROG_FLASH_SIZE has been lost after an OPTION RESET. It will not find any Library code after a LIBRARY DELETE has been issued.
LINE x1, y1, x2, y2 [, LW [, C]]	Draws a line starting at the coordinates ‘x1’ and ‘y1’ and ending at ‘x2’ and ‘y2’. ‘LW’ is the line’s width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal. ‘C’ is an integer representing the colour and defaults to the current foreground colour. All parameters can now be expressed as arrays and the software will plot the number of lines as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', and 'y2' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw' and 'c' can be either arrays or single variables/constants.
LINE AA x1, y1, x2, y2[,LW [, C]]	Draws a line with anti-aliasing . The parameters are as per the LINE command above. However this version will use variable intensity values of the specified colour to reduce the “staggered” quality of diagonal lines. In addition this version can draw diagonal lines of any width. Note that it does not accept arrays as parameters.
LINE GRAPH x(), y(),colour	This command generates a line graph of the coordinate pairs specified in “x()” and “y()”. The graph will have n-1 segments where there a n elements in the x and y arrays.

LINE PLOT ydata() [,nbr][,xstart] [,xinc] [,ystart] [,yinc][,colour]	Plots a line graph from an array of y-axis data points. 'ydata' is an array of floats or integers to be plotted 'nbr 'is the number of line segments to be plotted - defaults to the lesser of the array size and MM.HRES-2 if omitted 'xstart' is the x-coordinate to start plotting - defaults to 0 'xinc' is the increment along the x-axis to plot each coordinate - defaults to 1 'ystart' is the location in ydata to start the plot - defaults to the array start. NB: respects OPTION BASE 'yinc' is the increment to the index into ydata to add for each point to be plotted 'colour' is the colour to draw the line
LINE INPUT [prompt$,] string-variable$	Reads an entire line from the console input into 'string-variable$'. 'prompt$' is a string constant (not a variable or expression) and if specified it will be printed first. Unlike INPUT, this command will read a whole line, not stopping for comma delimited data items. A question mark is not printed unless it is part of 'prompt$'.
LINE INPUT #nbr, string-variable$	Same as the LINE INPUT command except that the input is read from a file previously opened for INPUT as '#fnbr' or a serial communications port previously opened for INPUT as 'nbr'. See the OPEN command.
LIST [fname$] or LIST ALL [fname$]	List a program on the serial console. LIST on its own will list the program with a pause at every screen full. LIST ALL will list the program without pauses. This is useful if you wish to transfer the program in the Armmite to a terminal emulator on a PC that has the ability to capture its input stream to a file. If the optional 'fname$' is specified then that file on the Flash Filesystem or SD Card will be listed
LIST COMMANDS [V] LIST FUNCTIONS [V]	Lists all valid commands. List all valid functions and operators V if appended outputs a count of the actual number of tokens in use.
LOAD file$ [,R]	Loads a program called 'file$' from the current drive into program memory. If the optional suffix ,R is added the program will be immediately run without prompting (in this case 'file$' must be a string constant) . If an extension is not specified ".BAS" will be added to the file name. The enhanced RUN command on Armmites and PicoMites effectively supersedes the [,R] option and also allow a string variable to be used.
LOAD DATA fname$, address	Loads the raw binary contents of file *fname$* and stores it in memory starting at *address*. Together with SAVE DATA this allows you to very easily to save and restore the contents of an array to and from disk. The code tries to protect you from crashing the system to the extent possible but there are many ways you can misuse the LOAD DATA command if you try. You can use PEEK to find out where the data for an array is located in memory. See SAVE DATA as well.
LOAD IMAGE file$ [, x, y]	Load a bitmapped image from the SD card and display it on the LCD panel. "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.
LOCAL variable [, variables] See DIM for the full syntax.	Defines a list of variable names as local to the subroutine or function. This command uses exactly the same syntax as DIM and will create variables that will only be visible within the subroutine or function. They will be automatically discarded when the subroutine or function exits.
LONGSTRING	The LONGSTRING commands allow for the manipulation of strings longer than the normal MMBasic limit of 255 characters. Variables for holding long strings must be defined as single dimensioned integer arrays with the number of elements set to the number of characters required for the maximum string length divided by eight. The reason for dividing by eight is that each integer in an MMBasic array occupies eight bytes. Note that the long string routines do not check for overflow in the length of the strings. If an attempt is made to create a string longer than a long string variable's size the outcome will be undefined.
LONGSTRING APPEND array%(), string$	Append a normal MMBasic string to a long string variable. array%() is a long string variable while string$ is a normal MMBasic string expression.
LONGSTRING CLEAR array%()	Will clear the long string variable array%(). ie, it will be set to an empty string.
LONGSTRING COPY dest%(), src%()	Copy one long string to another. dest%() is the destination variable and src%() is the source variable. Whatever was in dest%() will be overwritten.
LONGSTRING CONCAT dest%(), src%()	Concatenate one long string to another. dest%() is the destination variable and src%() is the source variable. src%() will the added to the end of dest%() (the destination will not be overwritten).
LONGSTRING LCASE array%()	Will convert any uppercase characters in array%() to lowercase. array%() must be long string variable.
LONGSTRING LEFT dest%(), src%(), nbr	Will copy the left hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the beginning of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING LOAD array%(), nbr, string$	Will copy 'nbr' characters from string$ to the long string variable array%() overwriting whatever was in array%().
LONGSTRING MID dest%(), src%(), start, nbr	Will copy 'nbr' characters from src%() to dest%() starting at character position 'start' overwriting whatever was in dest%(). ie, copy from the middle of src%(). 'nbr' is optional and if omitted the characters from 'start' to the end of the string will be copied src%() and dest%() must be long string variables. 'start' and 'nbr' must be an integer constants or expressions.
LONGSTRING PRINT [#n,] src%()[;]	Prints the longstring stored in 'src%()' to the file or COM port opened as '#n'. If '#n' is not specified the output will be sent to the console. A semicolon (;) at the end of the command will suppress the automatic output of a carriage return/ newline at the end of a print statement.
LONGSTRING REPLACE array%() , string$, start	Will substitute characters in the normal MMBasic string string$ into an existing long string array%() starting at position 'start' in the long string.

LONGSTRING RESIZE array%(), newsize	Sets the stored size of a long string array%() to newsize.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 SETBYTE array%(), pos, byte	Used to set the byte at position *pos* to the value *byte.* Pos respects the OPTION BASE setting.
LONGSTRING RIGHT dest%(), src%(), nbr	Will copy the right hand 'nbr' characters from src%() to dest%() overwriting whatever was in dest%(). ie, copy from the end of src%(). src%() and dest%() must be long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING TRIM array%(), nbr	Will trim ‘nbr’ characters from the left of a long string. array%() must be a long string variables. 'nbr' must be an integer constant or expression.
LONGSTRING UCASE array%()	Will convert any lowercase characters in array%() to uppercase. array%() must be long string variable.
LOOP [UNTIL expression]	Terminates a program loop: see DO.
MATH **Simple array arithmetic**	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.
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 scale to 1 is optimised and is the fastest way of copying an entire array
MATH POWER in(), power, out()	Raises each element in in () to the power defined and puts the output in out ()
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 array1(), array(2), ratio, array3()	This 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
MATH INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5] , sourcearray()	This command copies a specified set of values from a multi-dimensional array into a single dimensional array. It is much faster than using a FOR loop. The slice is specified by giving a value for all but one of the source array indicies and there should be as many indicies in the command, including the blank one, as there are dimensions in the source array e.g.

	OPTION BASE 1 DIM a(3,4,5) DIM b(4) MATH SLICE a(), 2, , 3, b() Will copy the elements 2,1,3 and 2,2,3 and 2,3,3 and 2,4,3 into array b()
MATH SLICE sourcearray(), [d1] [,d2] [,d3] [,d4] [,d5] ,destinationarray()	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
MATH SHIFT inarray%(), nbr, outarray%() [,U]	This command does a bit shift on all elements of inarray%() and places the result in outarray%() (may be the same as inarray%()). nbr can be between -63 and 63. Positive numbers are a left shift (multiply by power of 2). Negative number are a right shift. The optional parameter ,U will force an unsigned shift.
MATH WINDOW in(), minout, maxout, out() [,minin, maxin]	This command takes the "in" array and scales it between "minout" and "maxout" returning the answer in "out". Optionally, it can also return the minimum and maximum values found in the original data ("minin" and "minout"). Note: "minout" can be greater than "maxout" and in this case the data will be both scaled and inverted. e.g DIM IN(2)=(1,2,3) DIM OUT(2) MATH WINDOW IN(),7,3,OUT(),LOW,HIGH Will return OUT(0)=7, OUT(1)=5,OUT(2)=3,LOW=1,HIGH=3 This command can massively simplify scaling data for plotting etc.
Matrix arithmetic	
MATH M_PRINT array()	Quick mechanism to print a 2D matrix one row per line.
MATH M_TRANSPOSE in(), out()	Transpose matrix in() and put the answer in matrix out(), both arrays must be 2D but need not be square. If not square then the arrays must be dimensioned: in(m,n) out(n,m)
MATH M_MULT in1(), in2(), out()	Multiply the arrays in1() and in2() and put the answer in out()c. All arrays must be 2D but need not be square. If not square then the arrays must be dimensioned: in1(m,n) in2(p,m) ,out(p,n)
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 i.e. (determinant=0)
Vector arithmetic	
MATH V_PRINT array() [,HEX]	Quick mechanism to print a small array on a single line. The 'HEX' option will print the values in HEX for an integer array.

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 vector x,y,z to a normalised quaternion vector outVQ() with the magnitude calculated and 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 actor 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.
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()
Cell operations MATH C_ADD a1%(), a2%(), a3%() MATH C_SUB a1%(), a2%(), a3%() MATH C_MUL a1%(), a2%(), a3%() MATH C_DIV a1%(), a2%(), a3%() MATH C_ADD a1!(), a2!(), a3!() MATH C_SUB a1!(), a2!(), a3!() MATH C_MUL a1!(), a2!(), a3!() MATH C_DIV a1!(), a2!(), a3!()	These commands do cell by cell operations (hence C_) on identically sized arrays. There are no restrictions on the number of dimensions and no restrictions on using the same array twice or even three times in the parameters.The datatype must be the same for all the arrays. e.g. MATH C_MULT a%(),a%(),a%() will square all the values in the array a%()
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	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

FFTarray!(), signalarray!()	part in f(0,n) and the imaginary part in f(1,n). "signalarray" must be floating point and the single dimension must be the same as the FFT array. The command will return the real part of the inverse transform in "signalarray".
MATH FFT MAGNITUDE signalarray!(),magnitudearray!()	Generates magnitudes for frequencies for the data in “signalarray!” "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "magnitudearray" must be floating point and the size must be the same as the signal array The command will return the magnitude of the signal at various frequencies according to the formula: frequency at array position N = N * sample_frequency / number_of_samples
MATH FFT PHASE signalarray!(), phasearray!()	Generates phases for frequencies for the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "phasearray" must be floating point and the size must be the same as the signal array. The command will return the phase angle of the signal at various frequencies according to the formula above.

MEMORY	List the amount of memory currently in use. For example:
	`Flash:`
	`  1K ( 1%) Program (40 lines)`
	`127K (99%) Free`
	
	`RAM:`
	`  0K ( 0%) 0 Variables`
	`  0K ( 0%) General`
	`114K (100%) Free`
	
	`Backup SRAM (4K):`
	`  4K (100%) Free`
	Notes:
	• General memory is used by serial I/O buffers, etc.
	• Memory usage is rounded to the nearest 1K byte.
	See Memory Command section for detailed explanation.

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.

MEMORY PRINT #]fnbr , nbr, address%/array() MEMORY INPUT [#]fnbr ,nbr, address%/array()	These commands save or read 'nbr' of data bytes from or to memory from or to an open disk file. The memory to be saved can be specified as an integer array in which case the nbr of bytes to be saved or read is checked against the array size. Alternatively, a memory address can be used in which case no checking can take place and user errors could result in a crash of the firmware..
MEMORY SET address, byte, Numberofbytes MEMORY SET BYTE address,byte, numberofbytes MEMORY SET SHORT address, short, numberofshorts MEMORY SET WORD address,word, numberofwords MEMORY SET INTEGER address, integervalue ,numberofintegers [,increment] MEMORY SET FLOAT address, floatingvalue ,numberofloats [,increment]	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 PACK source%(), destination%(),number,size MEMORY UNPACK source%(), destination%(), number,size	Memory pack and unpack allow integer values from one array to be compressed into another or uncompressed from one to the other. The two arrays are always normal integer arrays but the packed array can have 2, 4, 8, 16 or 64 values "packed into them. Thus a single integer array element could store 2 off 32-bit words, 4 off 16 bit values, 8 bytes, 16 nibbles, or 64 booleans (bits). "number specifies the number of values to be packed or unpacked and "size" specifies the number of bits (1,4,8,16,or 32)
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 in str$ to be replaced. If str2$ is shorter or longer than the selected range then the length of str$ is adjusted to accommodate the replacement string. If num is omitted then the number of characters replaced defaults to the length of str2$.
MKDIR dir$	Make, or create, the directory 'dir$' on the SD card.
NAME old$ AS new$	Rename a file or a directory from 'old$' to 'new$'. Both are strings. A directory path can be used in both 'old$' and 'new$'. If the paths differ the file specified in 'old$' will be moved to the path specified in 'new$' with the file name as specified.
NEW	Deletes the program in flash, clears all variables including saved variables and resets the interpreter (ie, closes files, serial ports, etc).
NEXT [counter-variable] [, counter-variable], etc	NEXT comes at the end of a FOR-NEXT loop; see FOR. The 'counter-variable' specifies exactly which loop is being operated on. If no 'counter-variable' is specified the NEXT will default to the innermost loop. It is also possible to specify multiple variables as in: NEXT x, y, z

ON ERROR ABORT or ON ERROR IGNORE or ON ERROR SKIP [nn] or ON ERROR CLEAR	This controls the action taken if an error occurs while running a program and applies to all errors discovered by MMBasic including syntax errors, wrong data, missing hardware, SD Card access, etc. ON ERROR ABORT will cause MMBasic to display an error message, abort the program and return to the command prompt. This is the normal behaviour and is the default when a program starts running. ON ERROR IGNORE will cause any error to be ignored. ON ERROR SKIP will ignore an error in a number of commands (specified by the number 'nn') executed following this command. 'nn' is optional, the default if not specified is one. After the number of commands has completed (with an error or not) the behaviour of MMBasic will revert to ON ERROR ABORT. If an error occurs and is ignored/skipped the read only variable MM.ERRNO will be set to non zero and MM.ERRMSG$ will be set to the error message that would normally be generated. These are reset to zero and an empty string by ON ERROR CLEAR. They are also cleared when the program is run and when ON ERROR IGNORE and ON ERROR SKIP are used. ON ERROR IGNORE can make it very difficult to debug a program so it is strongly recommended that only ON ERROR SKIP be used.
ON nbr GOTO | GOSUB target[,target, target,...]	See Obsolete Commands and Functions 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.
ON KEY target or ON KEY ASCIIcode, target	The first variant of the command sets an interrupt which will call 'target' user defined subroutine whenever there is one or more characters waiting in the serial console input buffer. Note that all characters waiting in the input buffer should be read in the interrupt subroutine otherwise another interrupt will be automatically generated as soon as the program returns from the interrupt. This second variant allows you to associate an interrupt routine with a specific key press. This operates at a low level for the serial console and if activated the key does not get put into the input buffer but merely triggers the interrupt. It uses a separate interrupt from the simple ON KEY command so can be used at the same time if required. In both variants, to disable the interrupt use numeric zero for the target, i.e.: ON KEY 0. or ON KEY ASCIIcode, 0
ONEWIRE RESET pin or ONEWIRE WRITE pin, flag, length, data [, data…] or ONEWIRE READ pin, flag, length, data [, data…]	Commands for communicating with 1-Wire devices. ONEWIRE RESET will reset the 1-Wire bus ONEWIRE WRITE will send a number of bytes ONEWIRE READ will read a number of bytes 'pin' is the I/O pin (located in the rear connector) to use. It can be any pin capable of digital I/O. 'flag' is a combination of the following options: 1 - Send reset before command 2 - Send reset after command 4 - Only send/recv a bit instead of a byte of data 8 - Invoke a strong pullup after the command (the pin will be set high and open drain disabled) 'length' is the length of data to send or receive 'data' is the data to send or variable to receive. The number of data items must agree with the length parameter. See also *Appendix C*.

OPEN fname$ FOR mode AS [#]fnbr	Opens a file for reading or writing. 'fname' is the filename with an optional extension, separated by a dot (.). Long file names with upper and lower case characters are supported. A directory path can be specified with the 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. The maximum filename/directory length is 63 chars to reduce the buffer needed so don't use filenames > 63 chars INPUT will open the file for reading and throw an error if the file does not exist. OUTPUT will open the file for writing and will automatically overwrite any existing file with the same name. APPEND will also open the file for writing but it will not overwrite an existing file; instead any writes will be appended to the end of the file. If there is no existing file the APPEND mode will act the same as the OUTPUT mode (i.e. the file is created then opened for writing). RANDOM will open the file for both read and write and will allow random access using the SEEK command. When opened the read/write pointer is positioned at the end of the file. 'fnbr' is the file number (1 to 10). The # is optional. Up to 10 files can be open simultaneously. The INPUT, LINE INPUT, PRINT, WRITE and CLOSE commands as well as the EOF() and INPUT$() functions all use 'fnbr' to identify the file being operated on. See also OPTION ERROR and MM.ERRNO for error handling.
OPEN comspec$ AS [#]fnbr	Will open a serial communications port for reading and writing. Four ports are available (COM1: , COM2: ,COM3: and COM4:) all can be open simultaneously. If OPTION SERIAL CONSOLE is used then COM1: is not available. Using 'fnbr' the port can be written to and read from using any command or function that uses a file number. 'comspec$' is the communication specification and is a string (it can be a string variable) specifying the serial port to be opened and optional parameters. The default is 9600 baud, 8 data bits, no parity and one stop bit. It has the form `"COMn: baud, buf, int, int-trigger, 7BIT, (ODD or EVEN), OC, S2"` Where: • 'n' is the serial port number for either COM1:, COM2: ,COM3: or COM4:. • 'baud' is the baud rate. This can be any value between 2400 (the minimum) and 1843200 Hz. Default is 9600. • 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes. • 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt. • 'int-trigger' sets the trigger condition for calling the interrupt subroutine. If it is a normal number the interrupt subroutine will be called when this number of characters has arrived in the receive queue. All parameters except the serial port name (COMn:) are optional. If any one parameter is left out then all the following parameters must also be left out and the defaults will be used. Five options can be added to the end of 'comspec$' • 'OC' will force the transmit pin to be open collector. The default is

	normal (0 to 3.3V) output. • 'S2' specifies that two stop bits will be sent following each character transmitted. • '7BIT' will specify that 7 bit transmit and receive is to be used. • 'ODD' will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'EVEN' will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)
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 Option Settings earlier in this manual.
PAGE #n [,#n2, #n3, etc]	Now GUI PAGE, The PAGE command needs to be manually changed to GUI PAGE in any existing program.
PAUSE delay	Halt execution of the running program for 'delay' ms. This can be a fraction. For example, 0.2 is equal to 200 µs. The maximum delay is 2147483647 ms (about 24 days). Note that interrupts will be recognised and processed during a pause.
PIN(pin) = value	For a 'pin' configured as digital output this will set the output to low ('value' is zero) or high ('value' non-zero). You can set an output high or low before it is configured as an output and that setting will be the default output when the SETPIN command takes effect. See the function PIN() for reading from a pin and the command SETPIN for configuring it.
PIXEL x, y [,c]	Set a pixel on an attached LCD panel to a colour. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. 'c' is a 24 bit number specifying the colour. 'c' is optional and if omitted the current foreground colour will be used. All parameters can be expressed as arrays and the software will plot the number of pixels as determined by the dimensions of the smallest array. 'x' and 'y' must both be arrays or both be single variables /constants otherwise an error will be generated. 'c' can be either an arrays or a single variable or constant. See the chapter Using an LCD Panel for a definition of the colours and graphics coordinates.
	Generates two separate sine waves on the sound output left and right channels. The tone plays in the background (the program will continue

PLAY TONE left, right [, dur]	running after this command). 'left' and 'right' are the frequencies in Hz to use for the left and right channels. 'dur' specifies the number of milliseconds that the tone will sound for. MMBasic will round the time to the next nearest complete waveform of the first frequency specified so that the tone will always finish with the DC level in the middle and no discontinuity. If the duration is not specified, the tone will continue until explicitly stopped or the program terminates. The frequency can be from 1Hz to 20KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command.
PLAY WAV file$ [, interrupt] or PLAY FLAC file$ [, interrupt]	Play an audio file on the audio (DAC) output. 'file$' is the file to play (the appropriate extension will be appended if missing). The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. For WAV files MMBasic will automatically compensate for the frequency, number of bits and number of channels of the WAV file. For FLAC files the supported frequencies are: 44100Hz 16-bit (CD quality) and 24-bit 48000Hz 16-bit and 24-bit 88200Hz 16-bit and 24-bit 96000Hz 24-bit Maximums for FLAC and WAV file playback are 96KHz 24-bit. Both will auto-configure to the file provided. As an indication, 96KHz 24-bit FLAC uses just over 50% of the CPU's resources.
PLAY PAUSE PLAY RESUME PLAY STOP	PLAY PAUSE will temporarily halt the currently playing file or tone. PLAY RESUME will resume playing a sound that was paused. PLAY STOP will terminate the playing of the file or tone. When the program terminates for whatever reason the sound output will also be automatically stopped.
PLAY 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 POKE SHORT addr%, short% POKE WORD addr%, word% POKE INTEGER addr%, int% POKE FLOAT addr%, float! POKE VAR var, offset, byte	Will set a byte or a word within the CPU's virtual memory space. POKE BYTE will set the byte (ie, 8 bits) at the memory location 'addr%' to 'byte'. 'addr%' should be an integer. POKE SHORT will set the short integer (ie, 16 bits) at the memory location 'addr%' to 'word%'. 'addr%' and short%' should be integers. POKE WORD will set the word (ie, 32 bits) at the memory location 'addr%' to 'word%'. 'addr%' and 'word%' should be integers. POKE INTEGER will set the MMBasic integer (ie, 64 bits) at the memory location 'addr%' to int%'. 'addr%' and int%' should be integers. POKE FLOAT will set the word (ie, 64 bits) at the memory location 'addr%' to 'float!'. 'addr%' should be an integer and 'float!' a floating point number. POKE VAR will set a byte in the memory address of 'var'. 'offset' is the ±offset from the address of the variable. An array is specified as var(). POKE VARTBL will set a byte in MMBasic's variable table. 'offset' is the

POKE VARTBL, offset, byte POKE DISPLAY command [,data1] [,data2] [,datan] POKE DISPLAY HRES n POKE DISPLAY VRES n	±offset from the start of the variable table. Note that a comma is required after the keyword VARTBL. This command sends commands and associated data to the display controller for a connected display. This allows the programmer to change parameters of how the display is configured. e.g. POKE DISPLAY &H28 will turn off an SSD1963 display and POKE DISPLAY &H29 will turn it back on again. 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 size of the n array determines the number of polygons that will be drawn. The elements of array n() define the number of xy-coordinate pairs in each of the polygons. e.g DIM n(1)=(3,3) would define that 2 polygons are to be drawn with three vertices each. The xy-coordinate pairs for all the polygons are stored in xarray%() and yarray%(). The xarray%() and yarray%() parameters must have at least as many elements as the total of the values in the n array. Each polygon can be closed with the first and last elements the same. If the last element is not the same as the first the firmware will automatically create an additional xy-coordinate pair to complete the polygon. If fill colour is omitted then just the polygon outlines are drawn. The colour parameters can be a single value in which case all polygons are drawn in the same colour or they can be arrays with the same cardinality as n. In this case each polygon drawn can have a different colour of both border and/or fill. For example, this will draw 3 triangles in yellow, green and red: DIM c%(2)=(3,3,3) DIM x%(8)=(100,50,150,100,50,150,100,50,150) DIM y%(8)=(50,100,100,150,200,200,250,300,300) DIM fc%(2)=(rgb(yellow),rgb(green),rgb(red)) POLYGON c%(),x%(),y%(),fc%(),fc%()
PORT(start, nbr [,start, nbr]…) = value	Sets a number of I/O pins simultaneously (ie, with one command). 'start' is an I/O pin number and the lowest bit in 'value' (bit 0) will be used to set that pin. Bit 1 will be used to set the pin 'start' plus 1, bit 2 will set pin 'start'+2 and so on for 'nbr' number of bits. Each start/nbr pair defines a set of consecutively numbered I/O pins 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 up to 25 times if additional groups of consecutive output pins needs 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. Multiple expressions can be used and must be separated by either a: • Comma (,) which will output the tab character • Semicolon (;) which will not output anything (it is just used to separate expressions). • Nothing or a space which will act the same as a semicolon. A semicolon (;) at the end of the expression list will suppress the automatic output of a carriage return/ newline at the end of a print statement. When printed, a number is preceded with a space if positive or a minus (-) if negative but is not followed by a space. Integers (whole numbers) are printed without a decimal point while fractions are printed with the decimal point and the significant decimal digits. Large floating point numbers (greater than six digits) are printed in scientific number format. The function TAB() can be used to space to a certain column and the string functions can be used to justify or otherwise format strings.
PRINT #nbr, expression [[,;]expression] … etc	Same as the normal PRINT command except that the output is directed to a file previously opened for OUTPUT or APPEND as '#fnbr' or to a serial communications port previously opened as 'nbr'. See the OPEN command.
PRINT #GPS, string$	Outputs a NMEA string to an opened GPS device. The string must start with a $ character and end with a * character. The checksum is calculated automatically by the firmware and is appended to the string together with the carriage return and line feed characters required.
PULSE pin, width	Will generate a pulse on 'pin' with duration of 'width' ms. 'width' can be a fraction. For example, 0.01 is equal to 10µs and this enables the generation of very narrow pulses. The generated pulse is of the opposite polarity to the state of the I/O pin when the command is executed. For example, if the output is set high the PULSE command will generate a negative going pulse. Notes: • 'pin' must be configured as an output. • For a pulse of less than 3 ms the accuracy is ± 1 µs. • For a pulse of 3 ms or more the accuracy is ± 0.5 ms. A pulse of 3 ms or more will run in the background. Up to five different and concurrent pulses can be running in the background and each can have its time changed by issuing a new PULSE command or it can be terminated by issuing a PULSE command with zero for 'width'.
PWM 1, freq, 1A PWM 1, freq, 1A, 1B PWM 1, freq, 1A, 1B,1C PWM 2, freq, 2A PWM 2, freq, 2A, 2B PWM 2, freq, 2A, 2B, 2C PWM 3, freq, 3A	Generate a pulse width modulated (PWM) output for driving analog circuits, sound output, etc. There are a total of eight outputs designated as PWM. (they are also used for the SERVO command). Controller 1 can have one, two or three outputs, controller 2 can have one, two or three outputs, while controller 3 can have one or two outputs. All three controllers are independent and can be turned on and off and have different frequencies. '1', '2' or '3' is the controller number and 'freq' is the output frequency. 1A, 1B and 1C are the duty cycle for each of the controller 1 outputs, while 2A, 2B and 2C are the duty cycle for the controller 2 outputs. 3A and 3B are for controller 3. The specified I/O pins will be automatically configured as

PWM 3, freq, 3A, 3B PWM channel, STOP	outputs while any others will be unaffected and can be used for other duties. The duty cycle for each output is independent of the others and is specified as a percentage. If it is close to zero the output will be a narrow positive pulse, if 50 a square wave will be generated and if close to 100 it will be a very wide positive pulse Minimum frequency is 1Hz, maximum is 20MHz. Duty cycle and frequency accuracy will depend on frequency. The output will run continuously in the background while the program is running and can be stopped using the STOP command. The frequency and duty cycle can be changed at any time (without stoping the output) by issuing a new PWM command. The PWM function will take control of any specified outputs and when stopped the pins will be returned to a high impedance "not configured" state.
RBOX x, y, w, h [, r] [,c] [,fill]	Draws a box with rounded corners on the LCD starting at 'x' and 'y' which is 'w' pixels wide and 'h' pixels high. 'r' is the radius of the corners of the box. It defaults to 10. 'c' specifies the colour and defaults to the default foreground colour if not specified. 'fill' is the fill colour. It can be omitted or set to -1 in which case the box will not be filled. All parameters can now be expressed as arrays and the software will plot the number of boxes as determined by the dimensions of the smallest array. 'x', 'y', 'w', and 'h' must all be arrays or all be single variables /constants otherwise an error will be generated. 'r', 'c', and 'fill' can be either arrays or single variables/constants. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
READ variable[, variable]...	Reads values from DATA statements and assigns these values to the named variables. Variable types in a READ statement must match the data types in DATA statements as they are read. Arrays can be used as variables (specified with empty brackets, eg, a()) and in that case the size of the array is used to determine how many elements are to be read. If the array is multidimensional then the leftmost dimension will be the fastest moving. e.g READ a, b, c(), s$(), t$ This will read numbers into a and b, it will then fill the array c() with numbers It will then fill the array s$() with strings and then finally load the string t$. In all cases the firmwware uses the size of an array to determine how many elements are to be read. See also DATA and RESTORE. If you want to read from DATA statements in the library you must use the RESTORE command before the first READ command. This will reset the pointer to the library space.
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.
RESTORE [line]	Resets the line and position counters for the READ statement. If 'line' is specified the counters will be reset to the beginning of the specified line. 'line' can be a line number or label. A variable can also be used as the parameter. In that case a numerical variable should be used for a line number and a string variable for a label. If 'line' is not specified the counters will be reset to the start of the program.
RMDIR dir$	Remove, or delete, the directory 'dir$' on the SD card.
RUN [file$][,cmdline$]	Run a program. If file$ is not supplied then run the program currently held in program memory. If file$ is supplied then run the named file from the SD Card filesystem; if file$ does not contain a '.BAS' extension then one will be automatically added. If cmdline$ is supplied then pass its value to the MM.CMDLINE$ constant of the program when it runs. If cmdline$ is not supplied then an empty string value is passed to MM.CMDLINE$. • Both file$ and cmdline$ may be supplied as string expressions.
SAVE file$	Saves the program to the current working directory of the SD card as 'file$'. Example: SAVE "TEST.BAS" If an extension is not specified ".BAS" will be added to the file name.
SAVE DATA fname$, address, size	Saves *size* bytes to file *fname$* starting from *address*. Data is saved in raw binary format. Together with LOAD DATA this allows you to very easily to save and restore the contents of an array to and from disk. The code tries to protect you from crashing the system to the extent possible but there are many ways you can misuse the LOAD DATA command if you try. See LOAD DATA as well.
SAVE IMAGE file$ [, x, y, w, h]	Save the current image on the LCD display as a 24-bit BMP file. 'file$' is the name of the file. If an extension is not specified ".BMP" will be added to the file name. 'x', 'y', 'w' and 'h' are optional and are the coordinates (x and y are the top left coordinate) and dimensions (width and height) of the area to be saved. If not specified the whole screen will be saved.
SEEK [#]fnbr, pos	Will position the read/write pointer in a file that has been opened on the SD card for RANDOM access to the 'pos' byte. The first byte in a file is numbered one so SEEK #5,1 will position the read/write pointer to the start of the file.
SELECT CASE value CASE testexp [[, testexp] …] <statements> <statements>	Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression. 'testexp' is the value that 'exp' is to be compared against. It can be: • A single expression (ie, 34, "string" or PIN(4)*5) to which it may equal • A range of values in the form of two single expressions separated by the

CASE ELSE
<statements>
<statements>
END SELECT
	keyword "TO" (ie, 5 TO 9 or "aa" TO "cc")
	• A comparison starting with the keyword "IS" (which is optional). For example: IS > 5, IS <= 10.
	When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true.
	If 'value' cannot be matched with a 'testexp' it will be automatically matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any <statements> and continue with the code following the END SELECT.
	When a match is made the <statements> following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT.
	An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT.
	Example:
	`SELECT CASE nbr%`
	`  CASE 4, 9, 22, 33 TO 88`
	`  statements`
	`  CASE IS < 4, IS > 88, 5 TO 8`
	`  statements`
	`  CASE ELSE`
	`  statements`
	`END SELECT`
	Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT…CASE statements can be nested inside the CASE statements of other SELECT…CASE statements.

SERVO 1, freq, 1A SERVO 1, freq, 1A, 1B SERVO 1, freq, 1A, 1B, 1C SERVO 2, freq, 2A SERVO 2, freq, 2A, 2B SERVO 2, freq, 2A, 2B, 2C SERVO 3, freq, 3A SERVO 3, freq, 3A, 3B SERVO channel, STOP	Generate a constant stream of positive going pulses for driving a servo. The Armmite F4 has three servo controllers with the first and second being able to control up to three servos and the third two servos. All controllers are independent and can be turned on and off and have different frequencies. This command uses the I/O pins that are designated as PWM. (the two commands are very similar). '1', '2' or '3' is the controller number. 'freq' is the output frequency (between 20Hz and 1000 Hz) and is optional. If not specified it will default to 50 Hz 1A, 1B and 1C are the pulse widths for each of the controller 1 outputs while 2A ,2B and 2C are the pulse widths for the controller 2 outputs. 3A and 3B are the output for controller 3.The specified I/O pins will be automatically configured as outputs while any others will be unaffected and can be used for other duties. The pulse width for each output is independent of the others and is specified in milliseconds, which can be a fractional number (ie, 1.536). For accurate positioning the output resolution is about 0.005 ms. The minimum value is 0.01ms while the maximum is 18.9ms. Most servos will accept a range of 0.8ms to 2.2ms. The output will run continuously in the background while the program is running and can be stopped using the STOP command. The pulse widths of the outputs can be changed at any time (without stoping the output) by issuing a new SERVO command. The SERVO function will take control of any specified outputs and when stopped the pins will be returned to a high impedance "not configured" state.
SETPIN pin, cfg [, option]	Will configure an external I/O pin. 'pin' is the I/O pin to configure, 'cfg' is the mode that the pin is to be set to and 'option' is an optional parameter. 'cfg' is a keyword and can be any one

	of the following: OFF Not configured or inactive AIN Analog input (ie, measure the voltage on the input). 'option' can be used to specify the number of bits in the conversion. Valid values are 8, 10 and 12. The default (if not specified) is 12 bits. The more bits the longer the conversion will take. Valid for pins PA0, PA1, PA2, PA3, PC0, PC1, PC2, PC3, PA6, PA7, PC4, PC5, PB0 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. Pull-up and pull-down resistors are designed with a true resistance in series with a switchable PMOS/NMOS. This MOS/NMOS contribution to the series resistance is minimum (~10% order) Valid for all available 47 pins 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. Valid for pins PE1, PE3, PE4, PA8 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 omited the period of just one cycle will be used. Valid for pins PE1, PE3, PE4, PA8 CIN Counting input Valid for pins PE1, PE3, PE4, PA8 ‘option’ can be used to specify which edge triggers the count and if any pullup or pulldown is enabled 1 specifies a rising edge with pulldown, 2 specifies a falling edge with pullup, 3 specifies that both a falling and rising edge will trigger a count with no pullup or pulldown applied, 4 specifies both edges but with a pulldown and 5 specifies both edges but with a pullup applied. If ‘option’ is omitted a rising edge will trigger the count and a pulldown is enabled. DOUT Digital output 'option' can be "OC" in which case the output will be open collector (or more correctly open drain). The functions PIN() and PORT() can also be used to return the value on one or

	more output pins . Previous versions of MMBasic used numbers for 'cfg' and the mode OOUT. For backwards compatibility they will still be recognised. See the function PIN() for reading inputs and the statement PIN()= for setting an output. See the command below if an interrupt is configured.
SETPIN pin, cfg, target [, option]	Will configure ‘pin’ to generate an interrupt according to ‘cfg’. Any I/O pin capable of digital input can be configured to generate an interrupt with a maximum of ten interrupts configured at any one time. 'cfg' is a keyword and can be any one of the following: OFF Not configured or inactive INTH Interrupt on low to high input INTL Interrupt on high to low input INTB Interrupt on both (ie, any change to the input) ‘target' is a user defined subroutine which will be called when the event happens. Return from the interrupt is via the END SUB or EXIT SUB commands. 'option' can be the keywords "PULLUP" or "PULLDOWN" as specified for a normal input pin (SETPIN pin DIN). If 'option' is omitted the input will be high impedance. This mode also configures the pin as a digital input so the value of the pin can always be retrieved using the function PIN().
SETTICK period, target [, nbr] SETTICK PAUSE, target [, nbr] SETTICK RESUME, target [, nbr]	This will setup a periodic interrupt (or "tick"). Four tick timers are available ('nbr' = 1, 2, 3 or 4). 'nbr' is optional and defaults to timer number 1. The time between interrupts is ‘period’ milliseconds and ‘target' is the interrupt subroutine which will be called when the timed event occurs. The period can range from 1 to 2147483647 ms (about 24 days). These interrupts can be disabled by setting ‘period’ to zero (ie, SETTICK 0, 0, 3 will disable tick timer number 3). Pause or resume the specified 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. flag values are: bit0: 0 (default if omitted) normal sort - 1 reverse sort bit1: 0 (default) case dependent - 1 sort is case independent startposition defines which element in the array to start the sort. Default is 0 (OPTION BASE 0) or 1 (OPTION BASE 1) elementstosort defines how many elements in the array should be sorted. Default is all elements after the startposition This allows connected arrays to be sorted. See the section *Sorting Data* in the tutorial Programming with the Colour Maximite 2 for an example.
	Communications via an SPI channel. The command SPI refers to channel 1.

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	The command SPI2 refers to channel 2 and has an identical syntax. 'nbr' is the number of data items to send or receive 'data1', 'data2', etc can be float or integer and in the case of WRITE can be a constant or expression. If 'string$' is used 'nbr' characters will be sent. 'array' must be a single dimension float or integer array and 'nbr' elements will be sent or received. See Appendix D for the details.
SPI2	As for SPI but for the second channel.
SPRITE	Alias for BLIT. See BLIT command for syntax.
STATIC variable [, variables] See DIM for the full syntax.	Defines a list of variable names which are local to the subroutine or function. These variables will retain their value between calls to the subroutine or function (unlike variables created using the LOCAL command). This command uses exactly the same syntax as DIM. The only difference is that the length of the variable name created by STATIC and the length of the subroutine or function name added together cannot exceed 32 characters. Static variables can be initialised to a value. This initialisation will take effect only on the first call to the subroutine (not on subsequent calls).
STEP	Part of the *FOR x=a TO b STEP c : NEXT* construction See FOR in command section See NEXT in command section
SUB xxx (arg1 [,arg2, …]) <statements> <statements> END SUB	Defines a callable subroutine. This is the same as adding a new command to MMBasic while it is running your program. 'xxx' is the subroutine name and it must meet the specifications for naming a variable. 'arg1', 'arg2', etc are the arguments or parameters to the subroutine. An array is specified by using empty brackets. ie, arg3(). The type of the argument can be specified by using a type suffix (ie, arg1$) or by specifying the type using `AS <type> (ie, arg1 AS STRING).` Every definition must have one END SUB statement. When this is reached the program will return to the next statement after the call to the subroutine. The command EXIT SUB can be used for an early exit. You use the subroutine by using its name and arguments in a program just as you would a normal command. For example: `MySub a1, a2` When the subroutine is called each argument in the caller is matched to the argument in the subroutine definition. These arguments are available only inside the subroutine. Subroutines can be called with a variable number of arguments. Any omitted arguments in the subroutine's list will be set to zero or a null string. Arguments in the caller's list that are a variable (ie, not an expression or constant) will be passed by reference to the subroutine. This means that any changes to the corresponding argument in the subroutine will also be copied to the caller's variable and therefore may be accessed after the subroutine has ended. Arrays are passed by specifying the array name with empty brackets (eg, arg()) and are always passed by reference. Brackets around the argument

	list in both the caller and the definition are optional.
SYNC [period] [,units]	The SYNC command with parameters sets up a fast timer and stores the period. The SYNC command without parameters waits for the timer to reach the period specified and then resets the timer and returns. As this all happens in the firmware the timing period is extremely accurate. Valid units are: If parameter is omitted: the period is expressed in raw clock counts 1/84,000,000 seconds U or u: the period is expressed in microseconds M or m: the period is expressed in milliseconds S or s: the period is expressed in seconds In all cases the maximum period allowed is just over 51 seconds but, of course, for longer periods there are lots of other ways of doing this. The command is specifically targeted at short periods. This code below will toggle a pin at 100 uSec invervals. SYNC 100,u DO SYNC pin(PC2)=1 SYNC pin(PC2)=0 LOOP
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$] [, font] [, scale] [, c] [, bc]	Displays a string on the LCD display starting at 'x' and 'y'. ‘string$’ is the string to be displayed. Numeric data should be converted to a string and formatted using the Str$() function. ' alignment$' is a string expression or string variable consisting of 0, 1 or 2 letters where the first letter is the horizontal alignment around 'x' and can be L, C or R for LEFT, CENTER, RIGHT and the second letter is the vertical alignment around 'y' and can be T, M or B for TOP, MIDDLE, BOTTOM. The default alignment is left/top. A third letter can be used in the alignment string to indicate the rotation of the text. This can be 'N' for normal orientation, 'V' for vertical text with each character under the previous running from top to bottom, 'I' the text will be inverted (ie, upside down), 'U' the text will be rotated counter clockwise by

	90º and 'D' the text will be rotated clockwise by 90º 'font' and 'scale' are optional and default to that set by the FONT command. 'c' is the drawing colour and 'bc' is the background colour. They are optional and default to the current foreground and background colours. See the chapter "Basic Drawing Commands" for a definition of the colours and graphics coordinates.
TIME$ = "HH:MM:SS" or TIME$ = "HH:MM" or TIME$ = "HH"	Sets the time of the internal clock. MM and SS are optional and will default to zero if not specified. For example TIME$ = "14:30" will set the clock to 14:30 with zero seconds. The time is set to "00:00:00" on first power up however the time will be remembered and kept updated as long as the battery is installed and can maintain a voltage of over 2.5V. Battery life should be 3 to 4 years even if the computer is powered off.
TIME$ = ±sec	Adds or subtracts 'sec' seconds from the current time being maintained by MMBasic. This makes it easier to fine tune the current time.
TIMER = msec	Resets the timer to a number of milliseconds. Normally this is just used to reset the timer to zero but you can set it to any positive integer. See the TIMER function for more details.
TO	Part of the *FOR x=a TO b STEP c : NEXT* construction See FOR in command section See NEXT in command section
TRACE ON or TRACE OFF or TRACE LIST nn	TRACE ON/OFF will turn on/off the trace facility. This facility will print the number of each line (counting from the beginning of the program) in square brackets as the program is executed. This is useful in debugging programs. TRACE LIST will list the last 'nn' lines executed in the format described above. MMBasic is always logging the lines executed so this facility is always available (ie, it does not have to be turned on).
TRIANGLE X1, Y1, X2, Y2, X3, Y3 [, C [, FILL]]	Draws a triangle on the LCD display with the corners at X1, Y1 and X2, Y2 and X3, Y3. 'C' is the colour of the triangle and defaults to the current foreground colour. 'FILL' is the fill colour and defaults to no fill (it can also be set to -1 for no fill). All parameters can be expressed as arrays and the software will plot the number of triangles as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', 'y2', 'x3',and 'y3' must all be arrays or all be single variables /constants otherwise an error will be generated 'c' and 'fill' can be either arrays or single variables/constants.
VAR SAVE var [, var]… or VAR RESTORE or VAR CLEAR	VAR SAVE will save one or more variables into battery backed-up ram. They can be restored later (normally after a power interruption). 'var' can be any number of numeric or string variables and/or arrays. Arrays are specified by using empty brackets. For example: var() VAR RESTORE will retrieve the previously saved variables and insert them (and their values) into the variable table. The VAR SAVE command can be used repeatedly. Variables that had been previously saved will be updated with their new value and any new variables (not previously saved) will be added to the saved list for later restoration. VAR CLEAR will erase all saved variables. Also, the saved variables will be automatically cleared by the NEW command or when a new program is loaded via AUTOSAVE, XMODEM, etc.

	This command is normally used to save calibration data, options, and other data which needs to be retained across a power interruption. Normally the VAR RESTORE command is placed at the start of the program so that previously saved variables are restored and immediately available to the program when it starts. Notes: • The storage space available to this command is 4KB. The memory used is battery backed RAM which operates at high speed and can be written to an unlimited number of times without restriction (unlike the Micromite). • Using VAR RESTORE without a previous save will have no effect and will not generate an error. • If, when using RESTORE, a variable with the same name already exists its value will be overwritten. • Saved arrays must be declared (using DIM) before they can be restored. Be aware that string arrays can rapidly use up all the memory allocated to this command. The LENGTH qualifier can be used when a string array is declared to reduce the size of the array (see the DIM command). This is not needed for ordinary string variables.
VAR FSAVE VAR FRESTORE	Copies the battery backed-up ram variables to the W25Q16 flash so it can be recovered if battery backup is lost or not available. Copies data saved in the W25Q16 flash back to the battery backed-up ram 'var' 4K block.
WATCHDOG timeout or WATCHDOG OFF	Starts the watchdog timer which will automatically restart the processor when it has timed out. This can be used to recover from some event that disabled the running program (such as an endless loop or a programming or other error that halts a running program). This can be important in an unattended control situation. 'timeout' is the time in milliseconds (ms) before a restart is forced. This command should be placed in strategic locations in the running BASIC program to constantly reset the watchdog timer and therefore prevent it from counting down to zero. If the timer count does reach zero (perhaps because the BASIC program has stopped running) the Maximite will be restarted and the automatic variable MM.WATCHDOG will be set to true (ie, 1) indicating that an error occurred. On a normal startup MM.WATCHDOG will be set to false (ie, 0). WATCHDOG OFF will disable the watchdog timer (this is the default on a reset or power up). The timer is also turned off when the break character (normally CTRL-C) is used on the console to interrupt a running program.
WS2812 type, pin, nbr, value%[()]	Now DEVICE WS2812. This form is accepted but saved as the new format of the command.
XMODEM SEND or XMODEM RECEIVE or XMODEM CRUNCH	Transfers a BASIC program to or from a remote computer using the XModem protocol. The transfer is done over the serial console connection. XMODEM SEND will send the current program held in the Armmite's program memory to the remote device. XMODEM RECEIVE will accept a program sent by the remote device and save it into the Micromite's program memory overwriting the program currently held there. Note that the data is buffered in RAM which limits the maximum program size. The CRUNCH option works like RECEIVE but it instructs MMBasic to remove all comments, blank lines and unnecessary spaces from the program

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

Functions

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

Square brackets indicate that the parameter or characters are optional.

ABS(number)	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ACOS (number)	Returns the inverse cosine of the argument 'number' in radians.
ASC(string$)	Returns the ASCII code for the first letter in the argument ‘string$’.
ASIN(number)	Returns the inverse sine value of the argument 'number' in radians.
ATAN2(y, x)	Returns the arc tangent of the two numbers x and y as an angle expressed in radians. It is similar to calculating the arc tangent of y / x, except that the signs of both arguments are used to determine the quadrant of the result.
ATN(number)	Returns the arctangent of the argument 'number' in radians.
BAUDRATE(comm [, timeout])	Returns the baudrate of any data received on the serial communications port 'comm'). This will sample the port over the period of 'timeout' seconds. 'timeout' will default to one second if not specified. Returns zero if no activity on the port within the timeout period.
BIN$(number [, chars])	Returns a string giving the binary (base 2) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
BIN2STR$(type, value [,BIG])	Returns a string containing the binary representation of 'value'. 'type' can be: INT64 signed 64-bit integer converted to an 8 byte string UINT64 unsigned 64-bit integer converted to an 8 byte string INT32 signed 32-bit integer converted to a 4 byte string UINT32 unsigned 32-bit integer converted to a 4 byte string INT16 signed 16-bit integer converted to a 2 byte string UINT16 unsigned 16-bit integer converted to a 2 byte string INT8 signed 8-bit integer converted to a 1 byte string UINT8 unsigned 8-bit integer converted to a 1 byte string SINGLE single precision floating point number converted to a 4 byte string DOUBLE double precision floating point number converted to a 8 byte string By default the string contains the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to ‘BIG’ will return the string in big-endian format (ie, the most significant byte is the first one in the string) In the case of the integer conversions, an error will be generated if the ‘value’ cannot fit into the ‘type’ (eg, an attempt to store the value 400 in a INT8).

	This function makes it easy to prepare data for efficient binary file I/O or for preparing numbers for output to sensors and saving to flash memory. See also the function STR2BIN
BOUND(array() [,dimension]	This returns the upper limit of the array for the dimension requested. The dimension defaults to one if not specified. Specifying a dimension value of 0 will return the current value of OPTION BASE. Unused dimensions will return a value of zero. For example: DIM myarray(44,45) BOUND(myarray(),2) will return 45
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 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 much 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. e.g. print choice(1, "hello","bye") will print "Hello" print choice(0, "hello","bye") will print "Bye" a=1:b=1:print choice(a=b, "hello","bye") will print "Hello"
CHR$(number)	Returns a one-character string consisting of the character corresponding to the ASCII code indicated by argument 'number'.
CINT(number)	Round numbers with fractional portions up or down to the next whole number or integer. For example, 45.47 will round to 45 45.57 will round to 46 -34.45 will round to -34 -34.55 will round to -35 See also INT() and FIX().
COS(number)	Returns the cosine of the argument 'number' in radians.
CTRLVAL(#ref)	Returns the current value of an advanced control. '#ref' is the control's reference. For controls like check boxes or switches it will be the number one (true) indicating that the control has been selected by the user or zero (false) if not. For controls that hold a number (eg, a SPINBOX) the value will be the number (normally a floating point number). For controls that hold a string (eg, TEXTBOX) the value will be a string.

CWD$	Returns the current working directory on the SD card as a string. The format is: `A:/dir1/dir2`.
DATE$	Reads the RTC and returns the date as a string in the form "dd-mm-yyyy"
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$ can be "dd-mm-yyyy", "dd-mm-yy" or "yyyy-mm-dd". Use NOW to get the day for the current date, e.g. ? DAY$(NOW)
DEG(radians)	Converts 'radians' to degrees.

DIR$(fspec, type)
or
DIR$(fspec)
or
DIR$( )
	Will search an SD card for files and return the names of entries found.
	'fspec' is a file specification using wildcards the same as used by the FILES command. Eg, "*.*" will return all entries, "*.TXT" will return text files.
	'type' is the type of entry to return and can be one of:
	ALL Search for both files and directories
	DIR Search for directories only
	FILE Search for files only (the default if 'type' is not specified)
	The function will return the first entry found. To retrieve subsequent entries use the function with no arguments. ie, DIR$( ). The return of an empty string indicates that there are no more entries to retrieve.
	This example will print all the files in a directory:
	`f$ = DIR$("*.*", FILE)`
	`DO WHILE f$ <> ""`
	`  PRINT f$`
	`  f$ = DIR$()`
	`LOOP`
	You must change to the required directory before invoking this command.

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

	and the INPUT$ function.
EPOCH(DATETIME$)	Returns the epoch number (number of seconds that have elapsed since midnight GMT on January 1, 1970) for the supplied DATETIME$ string. The format for DATETIME$ is “dd-mm-yyyy hh:mm:ss”. The format for year can be “dd-mm-yyyy”, “dd-mm-yy” or “yyyy-mm-dd”. Use NOW to get the epoch number for the current date and time, i.e. ? EPOCH(NOW)
EVAL(string$)	Will evaluate 'string$' as if it is a BASIC expression and return the result. 'string$' can be a constant, a variable or a string expression. The expression can use any operators, functions, variables, subroutines, etc that are known at the time of execution. The returned value will be an integer, float or string depending on the result of the evaluation. For example: `S$ = "COS(RAD(30)) * 100" : PRINT EVAL(S$)` Will display: `86.6025`
EXP(number)	Returns the exponential value of 'number', ie, e^x where x is 'number'.

FIELD$(string1, nbr, string2 [, string3])	Returns a particular field in a string with the fields separated by delimiters. 'nbr' is the field to return (the first is nbr 1). 'string1' is the string to search and 'string2' is a string holding the delimiters (more than one can be used). 'string3' is optional and if specified will include characters that are used to quote text in 'string1' (ie, quoted text will not be searched for a delimiter).
	For example:
	`s1 = "foo, boo, zoo, doo"`
	`r$ = FIELD$(s1, 2, ",")`
	will result in r$ = "boo". While:
	`s1 = "foo, 'boo, zoo', doo"`
	`r$ = FIELD$(s1, 2, ",", "'")`
	will result in r$ = "'boo, zoo'".

FIX(number)	Truncate a number to a whole number by eliminating the decimal point and all characters to the right of the decimal point. For example 9.89 will return 9 and -2.11 will return -2. The major difference between FIX and INT is that FIX provides a true integer function (ie, does not return the next lower number for negative numbers as INT() does). This behaviour is for Microsoft compatibility. See also CINT() .
FORMAT$(nbr [, fmt$])	Will return a string representing ‘nbr’ formatted according to the specifications in the string ‘fmt$’ The format specification starts with a % character and ends with a letter. Anything outside of this construct is copied to the output as is. The structure of a format specification is: % [flags] [width] [.precision] type Where ‘flags’ can be: - Left justify the value within a given field width 0 Use 0 for the pad character instead of space + Forces the + sign to be shown for positive numbers space Causes a positive value to display a space for the sign. Negative values still show the – sign ‘width’ is the minimum number of characters to output, less than this the

	number will be padded, more than this the width will be expanded. 'precision' specifies the number of fraction digits to generate with an e, or f type or the maximum number of significant digits to generate with a g type. If specified, the precision must be preceded by a dot (.). 'type' can be one of: g Automatically format the number for the best presentation. f Format the number with the decimal point and following digits e Format the number in exponential format If uppercase G or F is used the exponential output will use an uppercase E. If the format specification is not specified "%g" is assumed. Examples: format$(45) will return 45 format$(45, "%g") will return 45
GETSCANLINE	This will report on the line that is currently being drawn on the LCD Display Using this to time updates to the screen can avoid tearing effects caused by updates while the screen is being updated.
GPS()	The GPS functions are used to return data from a serial communications channel opened as GPS. The function GPS(VALID) should be checked before any of these functions are used to ensure that the returned value is valid. *See the PRINT #GPS command for the method on sending a configuration string to the GPS*
GPS(ALTITUDE) GPS(DATE) GPS(DOP) GPS(FIX) GPS(GEOID) GPS(LATITUDE) GPS LONGITUDE) GPS(SATELLITES) GPS(SPEED) GPS(TIME) GPS(TRACK) GPS(VALID)	returns current altitude if sentence GGA enabled returns the normal date string corrected for local time e.g. "12-01-2017" returns DOP (dilution of precision) value if sentence GGA enabled returns 0=no fix, 1=fix, etc. if sentence GGA enabled Returns the geoid-ellipsoid separation. if sentence GGA enabled returns the latitude in degrees as a floating point number, values are –ve for South of equator returns the longitude in degrees as a floating point number, values are –ve for West of the meridian returns number of satellites in view if sentence GGA enabled returns the ground speed in knots as a floating point number returns the normal time string corrected for local time e.g. "12:09:33" returns the track over the ground (degrees true) as a floating point number returns: 0=invalid data, 1=valid data. ALWAYS CHECK THIS VALUE TO ENSURE DATA IS VALID BEFORE USING OTHER GPS() FUNCTION CALLS

	GPS will accept $GNGGA and $GNRMC as well as $GPGGA and $GPRMC strings.
HEX$(number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
INKEY$	Checks the console input buffer and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string. If this is a carriage return, it is likely that there will be a line feed character following as often the enter key will produce a CR/LF pair. If the input buffer is empty this function will immediately return with an empty string (ie, ""). VT100 Function keys and other keys which send an escape sequence are resolved to a single character. See Appendix F – Special Keyboard Keys for how these are mapped.
INPUT$(nbr, [#]fnbr)	Will return a string composed of ‘nbr’ characters read from a file on the SD card previously opened for INPUT with the file number ‘#fnbr’. This function will read all characters including carriage return and new line without translation. Will return a string composed of ‘nbr’ characters read from a serial communications port opened as 'fnbr'. This function will return as many characters as are waiting in the receive buffer up to ‘nbr’. If there are no characters waiting it will immediately return with an empty string. #0 can be used which refers to the console's input buffer. The # is optional. Also see the OPEN command.
INSTR([start-position,] string-searched$, string-pattern$)	Returns the position at which 'string-pattern$' occurs in 'string-searched$', beginning at 'start-position'. Both the position returned and 'start-position' use 1 for the first character, 2 for the second, etc. The function returns zero if 'string-pattern$' is not found.
INT(number)	Truncate an expression to the next whole number less than or equal to the argument. For example 9.89 will return 9 and -2.11 will return -3. This behaviour is for Microsoft compatibility, the FIX() function provides a true integer function. See also CINT() .
JSON$(array%(), string$)	Returns a string representing a specific item out of the JSON input stored in the longstring array%() e.g. JSON$(a%(), “name”) JSON$(a%(), “coord.lat”) JSON$(a%(), “weather[0].description”) JSON$(a%(),”list[4].weather[0].description Examples taken from api.openweathermap.org *Many JSON data sets are quite large and may be too big to parse with the memory available to the Armmite F4. Where the memory is exhausted the effect on the Armmite F4 may be unpredicaable, however if there is an issue the firmware will attempt to force a software reset and print a relevant*

	error. If the data set you are working with is too large and can't be made smaller another approach will be required.
LCASE$(string$)	Returns 'string$' converted to lowercase characters.
LCOMPARE(array1%(), array2%())	Compare the contents of two long string variables array1%() and array2%(). The returned is an integer and will be -1 if array1%() is less than array2%(). It will be zero if they are equal in length and content and +1 if array1%() is greater than array2%(). The comparison uses the ASCII character set and is case sensitive.
LEFT$(string$, nbr)	Returns a substring of 'string$' with 'nbr' of characters from the left (beginning) of the string.
LEN(string$)	Returns the number of characters in 'string$'.
LGETBYTE(array%(), n)	Returns the numerical value of the 'n'th byte in the LONGSTRING held in 'array%()'. This function respects the setting of OPTION BASE in determining which byte to return.
LGETSTR$(array%(), start, length)	Returns part of a long string stored in array%() as a normal MMBasic string. The parameters start and length define the part of the string to be returned.
LINSTR(array%(), search$ [,start])	Returns the position of a search string in a long string. The returned value is an integer and will be zero if the substring cannot be found. array%() is the string to be searched and must be a long string variable. Search$ is the substring to look for and it must be a normal MMBasic string or expression (not a long string). The search is case sensitive. Normally the search will start at the first character in 'str' but the optional third parameter allows the start position of the search to be specified.
LLEN(array%())	Returns the length of a long string stored in array%()
LOC([#]fnbr)	For a file on the SD card opened as RANDOM this will return the current position of the read/write pointer in the file. Note that the first byte in a file is numbered 1. For a serial communications port opened as 'fnbr' this function will return the number of bytes received and waiting in the receive buffer to be read. #0 can be used which refers to the console's input buffer. The # is optional.
LOF([#]fnbr)	For a file on the SD card this will return the current length of the file in bytes. For a serial communications port opened as 'fnbr' this function will return the space (in characters) remaining in the transmit buffer. Note that when the buffer is full MMBasic will pause when adding a new character and wait for some space to become available. The # is optional.
LOG(number)	Returns the natural logarithm of the argument 'number'.

MATH	The math function performs many simple mathematical calculations that can be programmed in Basic but there are speed advantages to coding looping structures in C and there is the advantage that once debugged they are there for everyone without re-inventing the wheel.
Simple functions	
MATH(ATAN3 x,y)	Returns ATAN3 of x and y
MATH(COSH a)	Returns the hyperbolic cosine of a
MATH(LOG10 a)	Returns the base 10 logarithm of a
MATH(SINH a)	Returns the hyperbolic sine of a
MATH(TANH a)	Returns the hyperbolic tan of a
Simple Statistics	
MATH(CHI a())	*Returns the Pearson's chi-squared value of the two dimensional array a())*
MATH(CHI_p a())	*Returns the associated probability in % of the Pearson's chi-squared value of the two dimensional array a())*
MATH(CROSSING array() [,level] [,direction]	This returns the array index at which the values in the array pass the "level" in the direction specified. level defaults to 0. Direction defaults to 1 (valid values are -1 or 1)
MATH(CORREL a(), b())	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 minimum value in the array. This is only available on one-dimensional arrays.
MATH(SD a())	Returns the Sample 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

MATH(DOTPRODUCT v1(), v2()) MATH(M_DETERMINANT array!())	elements Returns the dot product of two vectors v1() and v2(). The vectors can have any number of elements but must have the same cardinality Returns the determinant of the array. The array must be square.
MATH(CRCn *array()*, length, [polynome,] [startmask,] [endmask,] [reverseIn,] [reverseOut] MATH(CRCn *string$*, length, [polynome,] [startmask,] [endmask,] [reverseIn,] [reverseOut]	Calculates CRC value of array or string. CRCn can be one of CRC8, CR12,CRC16 or CRC32. Defaults for startmask, endmask, reverseIn and reversOut are all zero. reverseIn true (1) means the bits of the input byte will be reflected i.e. used in reverse order. i.e. B0 is treated as the most significant bit. reverseOut true (1) means the CRC value calculated is reflected over the whole length of the CRC value. startmask is the value initially loaded to start the calculation?. endmask is ????? Defaults for polynomes are CRC8=&H07, CRC12=&H80D, CRC16=&H1021, crc32=&H04C11DB7. For CRC16-CCITT use MATH(CRC16 array(), n,, &HFFFF) e.g. DIM a%(8)=(49,50,51,52,53,54,55,56,57) a$="123456789" n=9 (length) PRINT HEX$(MATH(CRC16 a%(),9,,&HFFFF)) gives &H29B1 For CRC16-MODBUS use MATH(CRC16 a$, n,, &HFFFF) For CRC16-XMODEM use MATH(CRC16 a$, n,, &HFFFF) For CRC8-MAXIM use MATH(CRC16 a$ n,, &HFFFF) CRC32 use MATH (CRC32 a$,n,&hffffffff,&hffffffff,1,1) For ONEWIRE use MATH(CRC8 romcode(),n,&h31,0,0,1,1) MATH(CRCn function also accepts a string as the input. e.g. a$="123456789" ? math(crc16 a$,9,,&H1021) https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=15405 See Appendix G - Cyclic Redundancy Check (CRC)
MAX(arg1 [, arg2 [, …]]) or MIN(arg1 [, arg2 [, …]])	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.
MID$(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$’
MSGBOX (msg$, b1$ [,b2$ … b4$])	This function will display a message box on the screen with one to four touch sensitive buttons. All other controls will be disabled until the user touches one of the buttons. The message box will then be erased, the previous controls will be restored and the function will return the number of

	the button touched (the first button is number one) 'msg$' is the message to display. This can contain one or more tilde characters (~) which indicate a line break. Up to 10 lines can be displayed inside the box. 'b1$' is the caption for the first button, 'b2$' is the caption for the second button, etc. At least one button must be specified and four is the maximum. Any buttons not included in the argument list will not be displayed.
OCT$(number [, chars])	Returns a string giving the octal (base 8) representation of 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s).
PEEK(BYTE addr%) PEEK(CFUNADDR cfun) PEEK(SHORT addr%) PEEK(WORD addr%) PEEK(INTEGER addr%) PEEK(FLOAT addr% PEEK(VARADDR var) PEEK(VAR var, ±offset) PEEK(VARTBL, ±offset) PEEK(PROGMEM, ±offset) PEEK(OPTION, offset)	Will return a byte or a word within the CPU's virtual memory space. BYTE will return the byte (8-bits) located at 'addr%' CFUNADDR will return the address (32-bits) of the CFunction 'cfun' in memory. SHORT will return the short integer (16-bits) located at 'addr%' WORD will return the word (32-bits) located at 'addr%' INTEGER will return the integer (64-bits) located at 'addr%' FLOAT will return the floating point number (64-bits) located at 'addr%' VARADDR will return the address (32-bits) of the variable 'var' in memory. An array is specified as var(). VAR, will return a byte in the memory allocated to 'var'. An array is specified as var(). VARTBL, will return a byte in the memory allocated to the variable table maintained by MMBasic. Note that there is a comma after VARTBL. PROGMEM, will return a byte in the memory allocated to the program. Note that there is a comma after the keyword PROGMEM. Note that 'addr%' should be an integer. Returns the value (BYTE) of the Option at that location. Offset can be 0-79. On the Armmite F4 the options are stored in RTC battery backed registers so cannot be accessed using the other PEEK commands. See CFunction.h file to see locations of various Options withing the 80 Bytes.
PI	Returns the value of pi.
PIN(pin)	Returns the value on the external I/O 'pin'. Zero means digital low, 1 means digital high and for analog inputs it will return the measured voltage as a floating point number. Frequency inputs will return the frequency in Hz. A period input will return the period in milliseconds while a count input will return the count since reset (counting is done on the positive rising edge). The count input can be reset to zero by resetting the pin to counting input (even if it is already so configured). This function will also return the state of a pin configured as an output. Also see the SETPIN and PIN() = commands.
PIN(function)	Returns the value of a special pin function. 'function' indicates which special case to use. For example PRINT PIN(BAT). It can be one of: BAT The voltage of the backup battery.

	TEMP The temperature of the STM32 processor's core. SREF The stored calibrated value of the internal reference voltage measured with a supply of exactly 3.3V. This is programmed into the chip during production. IREF The measured value of the internal reference voltage. The actual value of VREF+ can be calculated as: 3.3 * PIN(SREF) / PIN(IREF) and this can be used to set OPTION VCC. Once Option VCC is set the the above value, IREF will now return a value very close to SREF, so issuing another OPTION VCC command will set it to an incorrect value.
PIXEL(x, y)	Returns the colour of a pixel on the LCD display. 'x' is the horizontal coordinate and 'y' is the vertical coordinate of the pixel. See the chapter Using an LCD Panel for a definition of the colours and graphics coordinates.
PORT(start, nbr [,start, nbr]…)	Returns the value of a number of I/O pins in one operation. 'start' is an I/O pin number and its value will be returned as bit 0. 'start'+1 will be returned as bit 1, 'start'+2 will be returned as bit 2, and so on for 'nbr' number of bits. I/O pins used must be numbered consecutively and any I/O pin that is invalid or not configured as an input will cause an error. The start/nbr pair can be repeated up to 25 times if additional groups of input pins need to be added. This function will also return the state of a pin configured as an output. It can be used to conveniently communicate with parallel devices like memory chips. Any number of I/O pins (and therefore bits) can be used from 1 to the number of I/O pins on the chip. See the PORT command to simultaneously output to a number of pins.
POS	See Obsolete Commands and Functions section. For the console returns the position of the cursor on the current line. Use MM.INFO$(
PULSIN(pin, polarity) or PULSIN(pin, polarity, t1) or PULSIN(pin, polarity, t1, t2)	Measures the width of an input pulse from 1µs to 1 second with 0.1µs resolution. 'pin' is the I/O pin to use for the measurement, it must be previously configured as a digital input. 'polarity' is the type of pulse to measure, if zero the function will return the width of the next negative pulse, if non zero it will measure the next positive pulse. 't1' is the timeout applied while waiting for the pulse to arrive, 't2' is the timeout used while measuring the pulse. Both are in microseconds (µs) and are optional. If 't2' is omitted the value of 't1' will be used for both timeouts. If both 't1' and 't2' are omitted then the timeouts will be set at 100000 (ie, 100ms). This function returns the width of the pulse in microseconds (µs) or -1 if a timeout has occurred. The measurement is accurate to ±1 µs. Note that this function will cause the running program to pause while the measurement is made and interrupts will be ignored during this period.
RAD(degrees)	Converts 'degrees' to radians.
RGB(red, green, blue [, trans]) or	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

RGB(shortcut [, trans])	colours that can be named are white, black, blue, green, cyan, red, magenta, yellow, brown and gray or grey. For example, RGB(red) or RGB(cyan). 'trans' is the level of transparency for colour depths 4 and 12. It is optional and defaults to 15 if not specified.
RIGHT$(string$, number-of-chars)	Returns a substring of 'string$' with 'number-of-chars' from the right (end) of the string.
RND(number) or RND	Returns a pseudo-random number in the range of 0 to 0.999999. The 'number' value is ignored if supplied. The Armmite F4 uses the hardware random number generator in the STM32 series of chips to deliver true random numbers. This means that the RANDOMIZE command is no longer needed and is not supported.
SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN(number)	Returns the sine of the argument 'number' in radians.
SPACE$(number)	Returns a string of blank spaces 'number' characters long.
SPI(data) or SPI2(data)	Send and receive data using an SPI channel. A single SPI transaction will send data while simultaneously receiving data from the slave. 'data' is the data to send and the function will return the data received during the transaction. 'data' can be an integer or a floating point variable or a constant.
SQR(number)	Returns the square root of number.
STR2BIN(type, string$ [,BIG])	Returns a number equal to the binary representation in 'string$'. 'type' can be: INT64 converts 8 byte string representing a signed 64-bit integer to an integer UINT64 converts 8 byte string representing an unsigned 64-bit integer to an integer INT32 converts 4 byte string representing a signed 32-bit integer to an integer UINT32 converts 4 byte string representing an unsigned 32-bit integer to an integer INT16 converts 2 byte string representing a signed 16-bit integer to an integer UINT16 converts 2 byte string representing an unsigned 16-bit integer to an integer INT8 converts 1 byte string representing a signed 8-bit integer to an integer UINT8 converts 1 byte string representing an unsigned 8-bit integer to an integer SINGLE converts 4 byte string representing single precision float to a float DOUBLE converts 8 byte string representing single precision float to a float By default the string must contain the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to 'BIG' will interpret the string in big-endian format (ie, the most significant byte is the first one in the string). This function makes it easy to read data from binary data files, interpret numbers from sensors or efficiently read binary data from flash memory chips. An error will be generated if the string is the incorrect length for the conversion requested See also the function BIN2STR$
	Returns a formatted string in decimal (base 10) representation of 'number'.

STR$(number) or STR$(number, m) or STR$(number, m, n) or STR$(number, m, n, c$)	If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR$(123.456) will return "123.456" STR$(-123.456) will return "-123.456" STR$(123.456, 1) will return "123.456" STR$(123.456, -1) will return "+123.456" STR$(123.456, 6) will return " 123.456" STR$(123.456, -6) will return " +123.456" STR$(-123.456, 6) will return " -123.456" STR$(-123.456, 6, 5) will return " -123.45600" STR$(-123.456, 6, -5) will return " -1.23456e+02" STR$(53, 6) will return " 53" STR$(53, 6, 2) will return " 53.00" STR$(53, 6, 2, "*") will return "****53.00"
STRING$(nbr, ascii) or STRING$(nbr, string$)	Returns a string 'nbr' bytes long consisting of either the first character of string$ or the character representing the ASCII value 'ascii' which is a decimal number in the range of 32 to 126.
TAB(number)	Outputs spaces until the column indicated by 'number' has been reached on the console output.
TAN(number)	Returns the tangent of the argument 'number' in radians.
TEMPR(pin)	Return the temperature measured by a DS18B20 temperature sensor connected to 'pin' (which does not have to be configured). The returned value is degrees C with a default resolution of 0.25ºC. If there is an error during the measurement the returned value will be 1000. The time required for the overall measurement is 200ms and interrupts will be ignored during this period. Alternatively the TEMPR START command can be used to start the measurement and your program can do other things while the conversion is progressing. When this function is called the value will then be returned instantly assuming the conversion period has expired. If it has not, this function will wait out the remainder of the conversion time before returning the value. The DS18B20 can be powered separately by a 3V to 5V supply or it can operate on parasitic power. See the chapter Special Device Support for more details.
	Returns the current time based on MMBasic's internal clock as a string in the

TIME$	form "HH:MM:SS" in 24 hour notation. For example, "14:30:00". If the OPTION MILLISECONDS ON command has been used this function will return the time including milliseconds as a decimal fraction of the seconds. For example: "14:35:06.239". To set the current time use the command TIME$ = .
TIMER	Returns the elapsed time in milliseconds (eg, 1/1000 of a second) since reset. This is a fractional floating point number with a resolution of 1µs. The timer is reset to zero on power up or a CPU restart and you can also reset it to any value by using TIMER as a command.
TOUCH(DOWN)	Will return true if the screen is currently being touched.
TOUCH(UP)	Will return true if the screen is currently NOT being touched.
TOUCH(LASTX)	Will return the X coordinate of the last location that was touched.
TOUCH(LASTY)	Will return the Y coordinate of the last location that was touched.
TOUCH(REF)	Will return the reference number of the control that is currently being touched or zero if no control is being touched.
TOUCH(LASTREF)	Will return the reference number of the last control that was touched.
UCASE$(string$)	Returns ‘string$’ converted to uppercase characters.
VAL(string$)	Returns the numerical value of the ‘string$’. If 'string$' is an invalid number the function will return zero. This function will recognise the &H prefix for a hexadecimal number, &O for octal and &B for binary.

Obsolete Commands and Functions

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

These commands may be removed in the future to recover memory for other features.

GOSUB target	Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN. New programs should use defined subroutines (ie, SUB…END SUB).
IF condition THEN linenbr	For Microsoft compatibility a GOTO is assumed if the THEN statement is followed by a number. A label is invalid in this construct. New programs should use: IF condition THEN GOTO linenbr | label
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.

Change Log

This table gives a summary of the main updates to this document for the various releases.

Version	Change Details
5.07.02 Beta0 Revision 0	

Appendix A – Serial Communications

Four serial ports are available for asynchronous serial communications labelled COM1:, COM2:, COM3:, and COM4: If the serial console is enabled then COM1: is unavailable.

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

The following is an example:

```
OPEN "COM1:4800" AS #5    ‘ open the first serial port with a speed of 4800 baud
PRINT #5, "Hello"         ‘ send the string "Hello" out of the serial port
dat$ = INPUT$(20, #5)     ‘ get up to 20 characters from the serial port
CLOSE #5                  ‘ close the serial port
```

The OPEN Command

A serial port is opened using the command:

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

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

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

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

- ‘n’ is the serial port number for either COM1:, COM2:, COM3: or COM4:.
- ‘baud’ is the baud rate. This can be any value between 2400 (the minimum) and 1.8MHz. Default is 9600.
- ‘buf’ is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes.
- ‘int’ is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt.
- ‘int-trigger’ sets the trigger condition for calling the interrupt subroutine. It is an integer and the interrupt subroutine will be called when this number of characters has arrived in the receive queue.

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

The following options can be added to the end of 'comspec$'

- ‘OC’ will force the transmit pin to be open collector. The default is normal (0 to 3.3V) output.
- 'S2' specifies that two stop bits will be sent following each character transmitted. Default is one stop bit.
- '7BIT' will specify that 7 bit transmit and receive is to be used. Default is 8 bits.
- ‘ODD’ will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)
- ‘EVEN’ will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)

Input/Output Pin Allocation

When a serial port is opened the pins used by the port will be automatically set to input or output as required and the SETPIN and PIN commands will be disabled for the pins. When the port is closed (using the CLOSE command) all pins used by the serial port will be set to a not-configured state and the SETPIN command can then be used to reconfigure them.

The connections for each COM port are shown in the I/O connector pinout diagrams in the beginning of this manual. Note that Tx means an output from the Armmite and Rx means an input to the Armmite.

The signal polarity is standard for devices running at TTL voltages (for RS232 voltages see below). Idle is voltage high, the start bit is voltage low, data uses a high voltage for logic 1 and the stop bit is voltage high. These signal levels allow you to directly connect to devices like GPS modules (which generally use TTL voltage levels).

When a serial port is opened MMBasic will enable an internal pullup resistor (to Vdd) on the Rx (receive data) pin. This has a value of about 40K and its purpose is to prevent the input from floating if it is left unconnected. Normally this is fine but it can cause a problem if you have an external resistor in series with the Rx pin, in that

case this resistor and the pullup resistor will form a voltage divider limiting how high or low the voltage on the Rx pin can swing and that in turn might mean that the input signal is not recognised. The solution is to use the command SERIAL PULLUP DISABLE to disable it.

Examples

Opening a serial port using all the defaults:
```
OPEN "COM2:" AS #2
```
Opening a serial port specifying only the baud rate (4800 bits per second):
```
OPEN "COM2:4800" AS #1
```
Opening a serial port specifying the baud rate (9600 bits per second) and receive buffer size (1KB):
```
OPEN "COM1:9600, 1024" AS #8
```
The same as above but with two stop bits enabled:
```
OPEN "COM1:9600, 1024, S2" AS #8
```
An example specifying everything including an interrupt, an interrupt level and two stop bits:
```
OPEN "COM1:19200, 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 (ie, the maximum number characters that can be retrieved by the INPUT$() function). Note that if the receive buffer overflows with incoming data the serial port will automatically discard the oldest data to make room for the new data.

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

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

Serial ports can be closed with the CLOSE command. This will wait for the transmit buffer to be emptied then free up the memory used by the buffers, cancel the interrupt (if set) and set all pins used by the port to the not configured state. A serial port is also automatically closed when commands such as RUN and NEW are issued.

Interrupts

The interrupt subroutine (if specified) will operate the same as a general interrupt on an external I/O pin.

When using interrupts you need to be aware that it will take some time for MMBasic to respond to the interrupt and more characters could have arrived in the meantime, especially at high baud rates. For example, if you have specified the interrupt level as 250 characters and a buffer of 256 characters then quite easily the buffer will have overflowed by the time the interrupt subroutine can read the data. In this case the buffer should be increased to 512 characters or more.

Low Cost RS-232 Interface

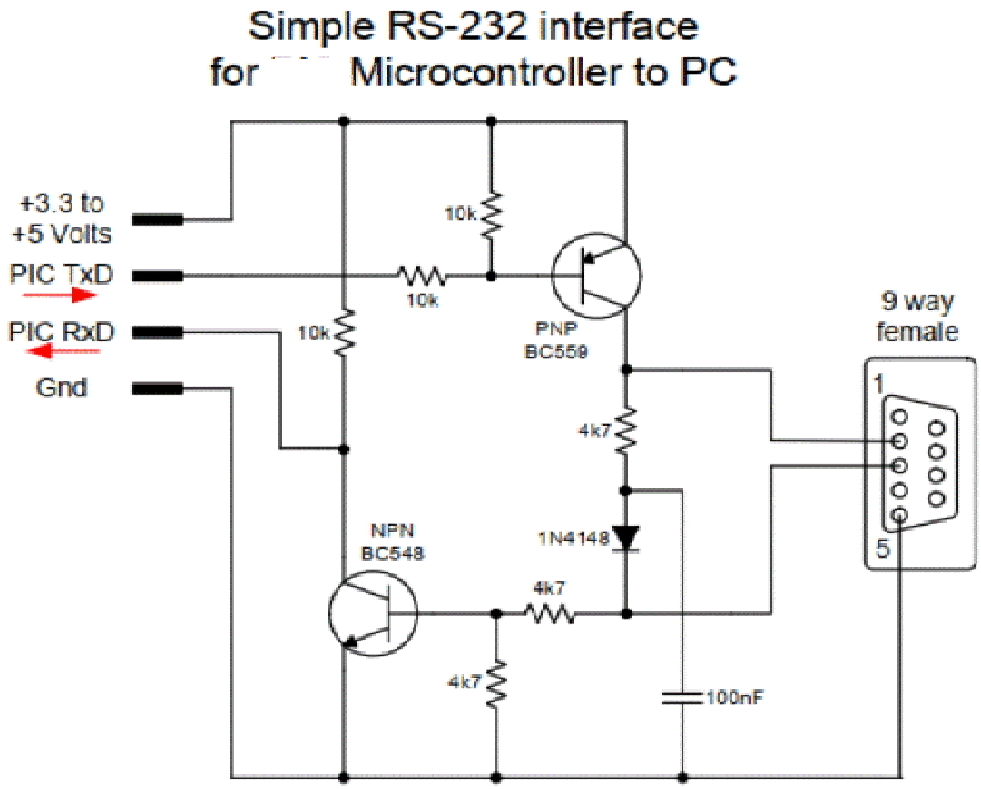
The RS-232 signalling system is used by modems, hardwired serial ports on a PC, test equipment, etc. It is the same as the serial TTL system used on the Armmite F4 with two exceptions:

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

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

The simple circuit described in the manuals for the Micromites cannot be used on the Armmite F4 as the INV feature is not supported on the ARM processor, however the following circuit from the link below should work with the Armmite as the signals are inverted using hardware.

http://picprojects.org.uk/projects/simpleSIO/ssio.htm



Appendix B – I2C Communications

The Armmite F4 implements three I²C channels, two on the rear I/O connector and the third dedicated to the front panel Wii connector. All operate in master mode (slave mode is not available).

There are four commands that can be used:

I2C OPEN speed, timeout	Enables the I²C module in master mode. The I2C command refers to channel 1 while commands I2C2 and I2C3 refer to channels 2 and 3 using the same syntax. 'speed' is the clock speed (in KHz) to use and must be one of 100 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 commands I2C2 refer to channels 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 values sent will be between 0 and 255): • The data can be supplied as individual bytes on the command line. Example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25 • The data can be in a one dimensional array specified with empty brackets (ie, no dimensions). 'sendlen' bytes of the array will be sent starting with the first element. Example: I2C WRITE &H6F, 0, 3, ARRAY() • The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 0, 3, STRING$
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I²C slave device. The I2C command refers to channel 1 while commands I2C2 refer to channels 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. They can then be configured using SETPIN. This command will also send a stop if the bus is still held. The I2C command refers to channel 1 while commands I2C2 refer to channels 2 using the same syntax.
I2C CHECK addr	Following an I2C CHECK *addr* MM.I2C will be set as follows: 0 if a device responds at the address, 1 if no response. Will give an error if the I2C has not been opened.

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

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

And similarly there are four commands for the slave mode:

I2C SLAVE OPEN addr, send_int, rcv_int	Enables the I²C module in slave mode. '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. 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. 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 I/O pins to a "not configured" state. They can then be configured using SETPIN.

7-Bit Addressing

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

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

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

I/O Pins

Refer to the Pin and Connector Capabilities table at the beginning of this manual for the pin numbers used for the I²C channels 1 and 2. Their signals are marked as data line (SDA) and clock (SCL). When the I2C CLOSE command is used the I/O pins are reset to a "not configured" state. Then can then be configured as per normal using SETPIN.

Neither the data line (SDA) and clock (SCL) for either I²C ports have any pullup resistors installed on the development board. 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 also the data and clock lines should not run next to each other but have a ground wire between them (to reduce crosstalk).

If the data line is not stable when the clock is high, or the clock line is jittery, the 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. When enabled the I2C pins have a 40K internal pullup. Another 10K external pullup may be required if the speed of 400kHz is used or the runs are long.

Master/Slave Modes

The I2C can be opened in either master or slave mode.

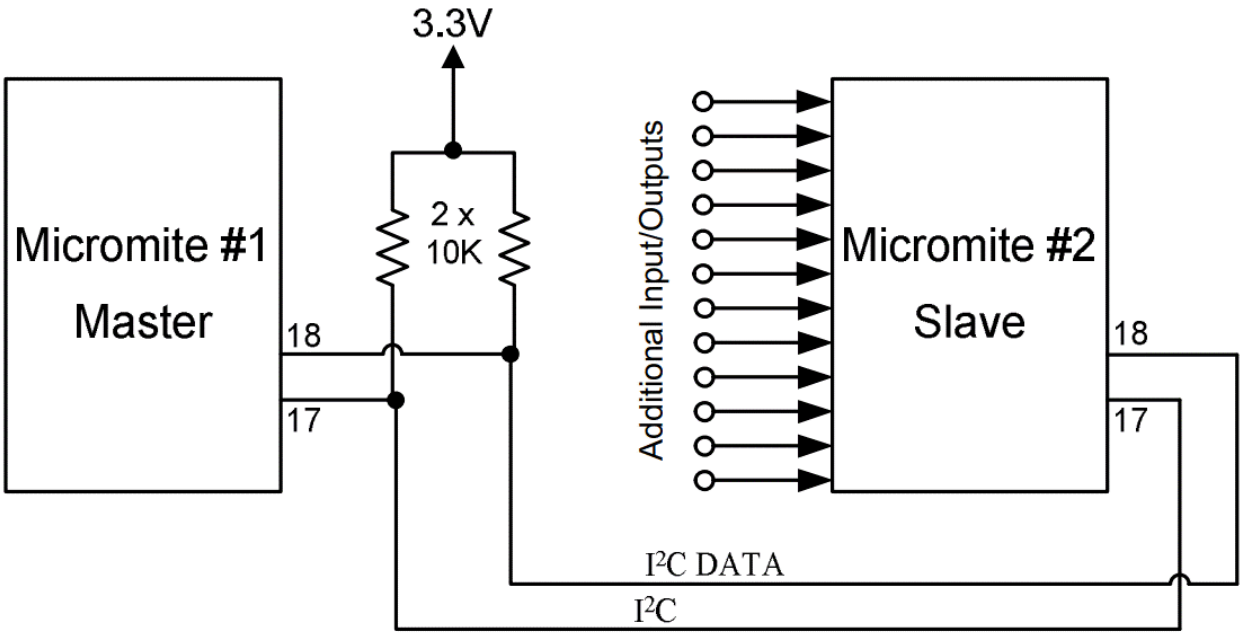
In master mode, the I2C send and receive commands will not return until the command completes or a timeout occurs (if the timeout option has been specified).

The slave mode uses an MMBasic interrupt to signal a change in status and calls the MMBasic send_int or rcv_int SUB specified in the I2C SLAVE OPEN command. I2C SLAVE READ and I2C SLAVE WRITE are used in the SUBs to write/read the data as specified by the I2C master. This operates the same as a general interrupt on an external I/O pin.

Example

As an example, the following program will read and display the current time (hours and minutes) maintained by a PCF8563 real time clock chip connected to I²C channel 2:

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



Program Running On the Slave:

The slave must first set up its I2C interface to respond to requests from the master. With that done it can then drop into an infinite loop while the job of responding to the master is handled by the I2C interrupts.

In the program below the slave will listen on I2C address 26 (hex) for a three byte command from the master.

The format of this message is:
Byte 1 is the command type. It can have one of three values; 1 means configure the pin, 2 means set the output of the pin and 3 means read the input of the pin.
Byte 2 is the pin number to operate on.
Byte 3 is the configuration number (if the command byte is 1), the output of the pin (if the command byte is 2) or a dummy number (if the command byte is 3).
The configuration number used when configuring a slave's I/O pin is the same as used in earlier versions of Maximite MMBasic (with the SETPIN command) and can be any one of:
0 Not configured or inactive
1 Analog input
2 Digital input
3 Frequency input
4 Period input
5 Counting input
8 Digital output
9 Open collector digital output. In this mode SPIN() will also return the value on the output pin .
Following a command from the master that requests an input, the master must then issue a second I2C command to read 12 bytes. The slave will respond by sending the value as a 12 character string.
This program can fall over if the master issues an incorrect command. For example, by trying to read from a pin that is not an input. If that occurs, an error will be generated and MMBasic will exit to the command prompt.
Rather than trap all the possible errors that the master can make, this program uses the watchdog timer. If an error does occur the watchdog timer will simply reboot the Micromite and the program will restart (because AUTORUN is on) and wait for the next message from the master. The master can tell that something was wrong because it would get a timeout.

This is the complete program running on the slave:

```
OPTION AUTORUN ON
DIM msg(2) ' array used to hold the message
I2C SLAVE OPEN &H26, 0, 0, WriteD, ReadD ' slave's address is 26 (hex)
DO ' the program loops forever
WATCHDOG 1000 ' this will recover from errors
LOOP
SUB ReadD ' received a message
I2C SLAVE READ 3, msg(), recvd ' get the message into the array
IF msg(0) = 1 THEN ' command = 1
SETPIN msg(1), msg(2) ' configure the I/O pin
ELSEIF msg(0) = 2 THEN ' command = 2
PIN(msg(1)) = msg(2) ' set the I/O pin's output
ELSE ' the command must be 3
s$ = str$(pin(msg(1))) + Space$(12) ' get the input on the I/O pin
ENDIF
END SUB ' return from the interrupt
SUB WriteD ' request from the master
I2C SLAVE WRITE 12, s$ ' send the last measurement
END SUB ' return from the interrup
```

Interface Routines On the Master:

These routines can be run on another Micromite or a Maximite or some other computer with an I2C interface. They assume that the slave Micromite is listening on I2C address 26 (hex).
If necessary these can be modified to access multiple Micromites (with different addresses), all acting as expansion chips and providing an almost unlimited expansion capability.
There are two subroutines and one function that together are used to control the slave:
SSETPIN pin, cfg This subroutine will setup an I/O pin on the slave. It operates the same as the MMBasic SETPIN command and the possible values for 'cfg' are listed above.
SPIN pin, output This subroutine will set the output of the slave's pin to 'output' (ie, high or low).
nn = SPIN(pin) This function will return the value of the input on the slave's I/O pin.
For example, to display the voltage on pin 3 of the slave you would use:

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

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

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

These are the three routines:

```
' configure an I/O pin on the slave
SUB SSETPIN pinnbr, cfg
I2C OPEN 100, 1000
I2C WRITE &H26, 0, 3, 1, pinnbr, cfg
IF MM.I2C THEN ERROR "Slave did not respond"
I2C CLOSE
END SUB
' set the output of an I/O pin on the slave
SUB SPIN pinnbr, dat
I2C OPEN 100, 1000
I2C WRITE &H26, 0, 3, 2, pinnbr, dat
IF MM.I2C THEN ERROR "Slave did not respond"
I2C CLOSE
END SUB
' get the input of an I/O pin on the slave
FUNCTION SPIN(pinnbr)
LOCAL t$
I2C OPEN 100, 1000
I2C WRITE &H26, 0, 3, 3, pinnbr, 0
I2C READ &H26, 0, 12, t$
IF MM.I2C THEN ERROR "Slave did not respond"
I2C CLOSE
SPin = VAL(t$)
END FUNCTION
```

Appendix C – 1-Wire Communications

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

There are three commands that you can use:

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

Where:

pin - The I/O pin (located in the rear connector) to use. It can be any pin capable of digital I/O.

flag - A combination of the following options:

1 - Send reset before command

2 - Send reset after command

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

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

length - Length of data to send or receive

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

And the automatic variable

MM.ONEWIRE	Returns true if a device was found

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

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

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

Appendix D – SPI Communications

The Serial Peripheral Interface (SPI) communications protocol is used to send and receive data between integrated circuits. The command SPI refers to channel 1 and SPI2 refers to channel 2. SPI2 is not listed below however it is available on the Armmite F4 and has an identical syntax.

I/O Pins

The SPI OPEN command will automatically configure the relevant I/O pins . (listed at the start of this manual). MISO stands for Master In Slave Out and because the Armmite F4 is always the master that pin will be configured as an input. Similarly MOSI stands for Master Out Slave In and that pin will be configured as an output.

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

SPI Open

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

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

Where:

- ‘speed’ is the speed of the clock. This can be 42000000, 21000000, 10500000, 5250000, 2625000, 1312500, or 756250 (ie, 42MHz, 21MHz, 10.5MHz, 5.25MHz, 2.625MHz, 1.3125MHz, 756.25KHz, or 378.125KHz). Any value can be used, the firmware will select the next valid speed that is equal or slower than the speed requested.
- 'mode' is a single numeric digit representing the transmission mode – see Transmission Format below.
- 'bits' is the number of bits to send/receive. This can be 8 or 16 for the Armite F4
- 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.

~~The Armmite F4 only supports Mode 0 and 1.~~

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

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

Standard Send/Receive

When the SPI channel is open data can be sent and received using the SPI function. The syntax is:

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

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

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

Bulk Send/Receive

Data can also be sent in bulk:

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

or

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

or

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

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

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

Data can also be received in bulk:

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

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

SPI Close

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

```
SPI CLOSE
```

Examples

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

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

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

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

Appendix E W25Q Windbond

'armite F4 Flash test

```
OPTION BASE 1
DIM AS INTEGER F_CS = 35
DIM AS INTEGER x
DIM AS INTEGER myArray(256)

SETPIN F_CS, DOUT
SPI OPEN 10000000,0,8
x = WB.ID%()
PRINT "Device ID = ";HEX$(x)
x = WB.JEDECID%()
PRINT "JEDEC  ID = ";HEX$(x)
PRINT "Pagecount = ";WB.PAGECOUNT%()
x = WB.SERIAL%()
PRINT "Serial No = ";HEX$(x)

PRINT
x = WB.Write_Disable()
PRINT "Status registers:"
FOR n = 1 TO 3
  x = WB.READSTATUS(n)
  PRINT STR$(n);"  ";BIN$(x,8)
NEXT n
PRINT

x = WB.READPAGE(1, myArray())
FOR n = 1 TO 100
  PRINT myArray(n);" ";
  IF n MOD 10 = 0 THEN PRINT
NEXT n
z$ = WB.READSTRING$(2)
IF LEN(z$)<>255 THEN PRINT z$

TIMER = 0
x = WB.Write_Enable()
x = WB.ERASE()
PAUSE 1000
PRINT BIN$(WB.READSTATUS(1),8)
DO
  PAUSE 100
  x = WB.READSTATUS(1)AND 1
LOOP UNTIL x = 0
PRINT "Erased in ";TIMER; "mS"
PRINT BIN$(WB.READSTATUS(1),8)
x = WB.Write_Enable()
x = WB.WRITESTRING(2,"Freddy is here!!")
DO
  PAUSE 100
  x = WB.READSTATUS(1)AND 1
LOOP UNTIL x = 0

x = WB.Write_Enable()
x = WB.WRITESTRING(2,"Barney is here!!", 20)
DO
  PAUSE 100
  x = WB.READSTATUS(1)AND 1
LOOP UNTIL x = 0

z$ = WB.READSTRING$(2)
PRINT z$
PRINT WB.READSTRING$(2,20)

x = WB.READPAGE(2, myArray())
FOR n = 1 TO 100
```

```
   PRINT myArray(n);" ";
   IF n MOD 10 = 0 THEN PRINT
 NEXT n

 FOR n = 1 TO 100
   myArray(n) = n
 NEXT n
 x = WB.Write_Enable()
 x = WB.WRITEPAGE(1, myArray())
 PAUSE 1000
 FOR n = 1 TO 100
   myArray(n) = 0
 NEXT n

 x = WB.READPAGE(1, myArray())
 FOR n = 1 TO 100
   PRINT myArray(n);" ";
   IF n MOD 10 = 0 THEN PRINT
 NEXT n

 'x = WB.Write_Enable()
 'x = WB.READSTATUS(2)
 'x = WB.Write_Enable()
 'x = x and &B11111101
 'y = WB.WRITESTATUS(2, x)
 'pause 1000
 'print
 'for n = 1 to 3
 'x = WB.READSTATUS(n)
 'print str$(n);"  ";bin$(x,8)
 'next n
 '
 SPI CLOSE
FUNCTION WB.Write_Enable()
 'SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 1, &H06
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.Write_Volatile_Enable()
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 1, &H50
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.Write_Disable()
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 1, &H04
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.READSTATUS(reg)
 LOCAL adr
 SELECT CASE reg
     CASE 1 : adr = &H05
     CASE 2 : adr = &H35
     CASE 3 : adr = &H15
 END SELECT

 'SPI OPEN 1000000,0,8
 PIN(F_CS)=0
```

```
 SPI WRITE 1, adr
 WB.READSTATUS=SPI(0)
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.WRITESTATUS(reg, myData)
 LOCAL adr
 SELECT CASE reg
     CASE 1 : adr = &H01
     CASE 2 : adr = &H31
     CASE 3 : adr = &H11
 END SELECT
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 2, adr, myData
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.ID%()
 'W25Q64FV &HEF16
 'W25Q16JV &HEF14
 LOCAL AS INTEGER mybyte(5)
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 1, &H90
 SPI READ 5, mybyte()
 PIN(F_CS)=1
 'SPI CLOSE
 WB.ID%=mybyte(4)*256+mybyte(5)
END FUNCTION

FUNCTION WB.JEDECID%()
 'W25Q64FV &HEF4017
 'W25Q16JV &HEF4015
 LOCAL AS INTEGER mybyte(3)
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 1, &H9F
 SPI READ 3, mybyte()
 PIN(F_CS)=1
 ' SPI CLOSE
 WB.JEDECID%=mybyte(1)*256*256+mybyte(2)*256+mybyte(3)
END FUNCTION

FUNCTION WB.PAGECOUNT%()
 'W25Q64FV &HEF4017
 'W25Q16JV &HEF4015
 LOCAL AS INTEGER mybyte(3)
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 1, &H9F
 SPI READ 3, mybyte()
 PIN(F_CS)=1
 ' SPI CLOSE
 WB.PAGECOUNT%=1 << (mybyte(3)-8)
END FUNCTION

FUNCTION WB.SERIAL%()
 LOCAL AS INTEGER mybyte(8)
 LOCAL AS FLOAT n
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 5, &H4B,0,0,0,0
 SPI READ 8, mybyte()
 PIN(F_CS)=1
```

```
 'SPI CLOSE

 WB.SERIAL%=mybyte(1)
 FOR n = 2 TO 8
   WB.SERIAL%=WB.SERIAL%*256+mybyte(n)
 NEXT n
END FUNCTION

FUNCTION WB.READPAGE(adr, my%())
 LOCAL adr1, adr2,adr3
 adr = adr<<8
 adr1 = (adr>>16) AND &HFF
 adr2 = (adr>>8) AND &HFF
 adr3 = adr AND &HFF
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 4, &H03, adr1, adr2, adr3
 SPI READ 256, my%()
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.WRITEPAGE(adr, my%())
 LOCAL adr1, adr2,adr3
 adr = adr<<8
 adr1 = (adr>>16) AND &HFF
 adr2 = (adr>>8) AND &HFF
 adr3 = adr AND &HFF
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 4, &H02, adr1, adr2, adr3
 SPI WRITE 256, my%()
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION
 FUNCTION WB.READSTRING$(adr, offset = 0)
 LOCAL adr1, adr2,adr3,strLen,my%(256),n
 adr = (adr<<8) + offset
 adr1 = (adr>>16) AND &HFF
 adr2 = (adr>>8) AND &HFF
 adr3 = adr AND &HFF
 ' SPI OPEN 1000000,0,8
 PIN(F_CS)=0
 SPI WRITE 4, &H03, adr1, adr2, adr3
 strLen = SPI(0)
 SPI READ strLen, my%()
 PIN(F_CS)=1
 IF strLen > 0 THEN
   FOR n = 1 TO strLen
     WB.READSTRING$ = WB.READSTRING$ + CHR$(my%(n))
   NEXT n
 ENDIF
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.WRITESTRING(adr, myStr$, offset = 0)
 LOCAL adr1, adr2,adr3,strLen ',my%(256),n
 adr = (adr<<8) + offset
 adr1 = (adr>>16) AND &HFF
 adr2 = (adr>>8) AND &HFF
 adr3 = adr AND &HFF
 strLen = LEN(myStr$)
 ' my%(1) = strLen
 '  for n = 1 to strLen
 '    my%(n+1) = asc(mid$(myStr$,n,1))
 '  next n
 ' SPI OPEN 1000000,0,8
```

```
 PIN(F_CS)=0
 SPI WRITE 4, &H02, adr1, adr2, adr3
 SPI WRITE 1, strLen
 SPI WRITE strLen, myStr$
 'spi write strLen+1, my%()
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION

FUNCTION WB.ERASE()
 PIN(F_CS)=0
 SPI WRITE 1, &HC7
 PIN(F_CS)=1
 ' SPI CLOSE
END FUNCTION
```

Appendix F – Special Keyboard Keys

MMBasic generates a single unique character for the function keys and other special keys on the keyboard. These are shown in this table as hexadecimal and decimal numbers:

Keyboard Key	Key Code (Hex)	Key Code (Decimal)
Delete	7F	127
Up Arrow	80	128
Down Arrow	81	129
Left Arrow	82	130
Right Arrow	83	131
Insert	84	132
Home	86	134
End	87	135
Page Up	88	136
Page Down	89	137
Alt	8B	139
F1	91	145
F2	92	146
F3	93	147
F4	94	148
F5	95	149
F6	96	150
F7	97	151
F8	98	152
F9	99	153
F10	9A	154
F11	9B	155
F12	9C	156

If the shift key is simultaneously pressed with a function key then 20 (hex) is added to the code (this is the equivalent of setting bit 5). For example Shift-F3 will generate B3 (hex).

The shift modifier only works with the function keys F1 to F12; it is ignored for the other keys.

MMBasic will translate most VT100 escape codes generated by terminal emulators such as Tera Term and Putty to these codes (the shift modifier only works for F3-F8). This means that a terminal emulator operating over a USB or a serial port opened as console will generate the same key codes as a directly attached keyboard.

Appendix G - Cyclic Redundancy Check (CRC)

The purpose of this description is not to explain or examine the mathematics behind CRCs but merely to explain the benefits of using them and how they may be used in MMBasic.

A cyclic redundancy check (CRC) is a strong algebraic error-detecting code commonly used in digital networks and storage devices to detect accidental changes to the data. Blocks of data have a short check value attached, this is the CRC. A CRC is based on the remainder of a polynomial division of their contents. This technique was invented by W.Wesley Peterson in 1961 and further developed by the CCITT.

Note: A CRC is not an error correction code it is just for error detection.

The advantages of using a CRC when sending or saving data are that it is a fast and efficient method for detecting errors in data transmission and can detect errors that occur during transmission, and storage caused by things such as noise, interference or distortion.

There are simpler methods of error detection including the use of ODD/EVEN parity for ASCII transmission and the use of a checksum. A checksum is simply an addition of all of the bits transmitted in a block of data, usually the carry bit is ignored and the resultant value is truncated to 8 or 16 bits which is appended to the end of the block of data. Neither of these methods are particularly secure.

The more bits in the CRC, the more errors it will detect so a 16 bit CRC will be more secure than an 8 bit CRC and so on. While a CRC will not catch all errors, it is much more secure than a simple checksum.

Using a CRC

Suppose we want to send a string via some medium. It could be data from your weather station which is located up a pole which is sent via radio to your base station for example.

A simple example using the MODBUS CRC:

```
' A simple demonstration of using a CRC
' the data to send
a$="123456789"
' calculate the CRC
b% = math (crc16 a$,,&h8005,&hffff,0,1,1)
' convert the CRC to a string
acrc$ = CHR$(b% AND &HFF) + CHR$((b%>>8) AND &HFF)
' add the CRC to the data to send
txd$ = a$ + acrc$
' then transmit the data
' here the data is printed for demonstration
print "The data to be transmitted"
printstr (txd$)

' check the recieved CRC and data
c% = math (crc16 txd$,,&h8005,&hffff,0,1,1)
check$ = CHR$(c% AND &HFF) + CHR$((c%>>8) AND &HFF)
print
print "The CRC of the recieved data including the"
print "recieved CRC - the result should be zero"
printstr (check$)

' Print a string as HEX numbers  (for debug)
sub PrintStr (b$)
  for i = 1 to len (b$)
print Hex$(asc (mid$(b$,i,1)),2); ", ";
```

```
  next i
  print
end sub
```

The CRC value is transmitted along with the data to the receiver. The receiver can verify the received data by removing the CRC then recalculate the CRC and compare that with the received CRC Or, more simply, recalculate the CRC for the whole received message and verify that the result is zero as demonstrated above. If the CRC does not check then the receiver should reply with a negative acknowledgement and request a re-transmission of the data.

The MMBasic CRC function:

MATH(CRCn data [,length] [,polynome] [,startmask] [,endmask] [,reverseIn] [,reverseOut]

Please see the entry in the Functions table. Some of the parameters used in the calculation of the CRC are not explained but their purpose may be made clear in the fullness of time. In the meantime here are some examples for commonly used CRC calculations from Volhout ’s demonstration program that may be useful.

```
' CCITT CRC16
CRC% = math  (crc16 data$,,,&hffff)

' MODBUS CRC16
CRC% = math  (crc16 data$,,&h8005,&hffff,0,1,1)

' XMODEM CRC16
CRC% = math  (crc16 data$,)

' MAXIM CRC8
CRC% = math  (crc8 data$,,&h31,,,1,1)

' standard CRC32
CRC% = math  (crc32 data$,,,&hffffffff,&hffffffff,1,1)
```

Demonstration program

```
' MATH CRC Demonstration program
' Based on the program "MATH CRC evaluation"
' written by Volhout
option base 1

'test string
a$="123456789"
l%=len (a$)
print "Test string "; a$
print

'convert test string to array
dim b%(l%)
for i=1 to l%  : b%(i)=asc (mid$(a$,i,1))  : next i

'perform CRC validation
dim CRC%

'check CCITT CRC16
CRC% = math  (crc16 b%(),l%,,&hffff)
print "CRC16-CCITT  "; hex$(CRC%)

'check MODBUS CRC16
CRC% = math  (crc16 b%(),l%,&h8005,&hffff,0,1,1)
```

```
print "CRC16-MODBUS "; hex$(CRC%)

'check XMODEM CRC16
CRC% = math  (crc16 b%(),l%)
print "CRC16-XMODEM "; hex$(CRC%)

'check MAXIM CRC8  (used in DALLAS single wire devices)
CRC% = math  (crc8 b%(),l%,&h31,,,1,1)
print "CRC8-MAXIM   "; hex$(CRC%)

'check standard CRC32
CRC% = math  (crc32 b%(),l%,,&hffffffff,&hffffffff,1,1)
print "CRC32        "; hex$(CRC%)
```

Some useful links:

The author of this CRC code https://github.com/RobTillaart/CRC

Explanations of CRCs http://www.sunshine2k.de/articles/coding/crc/understanding_crc.html#ch1

https://en.wikipedia.org/wiki/Cyclic_redundancy_check

On line CRC calculators

http://zorc.breitbandkatze.de/crc.html

https://crccalc.com

https://www.lddgo.net/en/encrypt/crc

Appendix H – Loading the Firmware

The STM32 processor includes its own programmer/bootloader so the Armmite F4 firmware can be easily loaded via USB using a personal computer or laptop (special hardware is not needed). Just follow these steps.

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

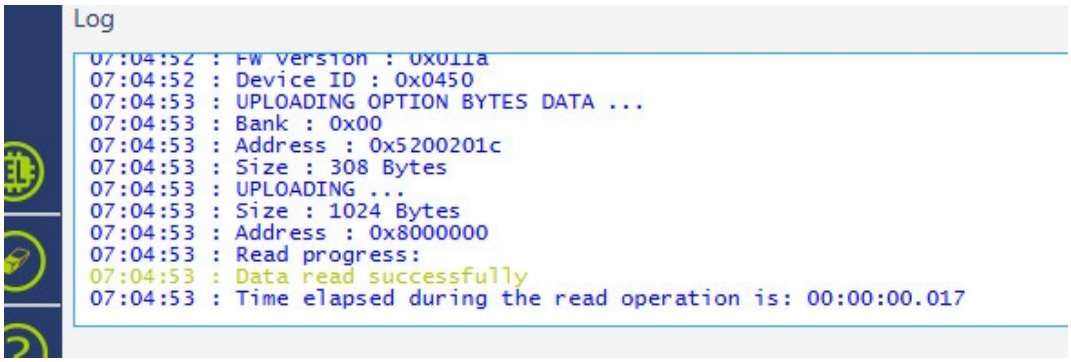
With the STM32F407VET6 development board unplugged set the BT0 and BT1 jumpers as in the picture to enable the bootloader mode.

Note:
BT1 has a pulldown resistor as part of the board, so its jumper is not actually required.
All that is required is the BT0 jumper.

Using a USB Type-A to Type-B cable connect the USB port on the Armmite F4 to a USB port on your desktop computer. This will power up the STM32F407VET6 development board and you should also hear a sound from your computer as it connects and you should see the STM32Bootloader appear as a new device.

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

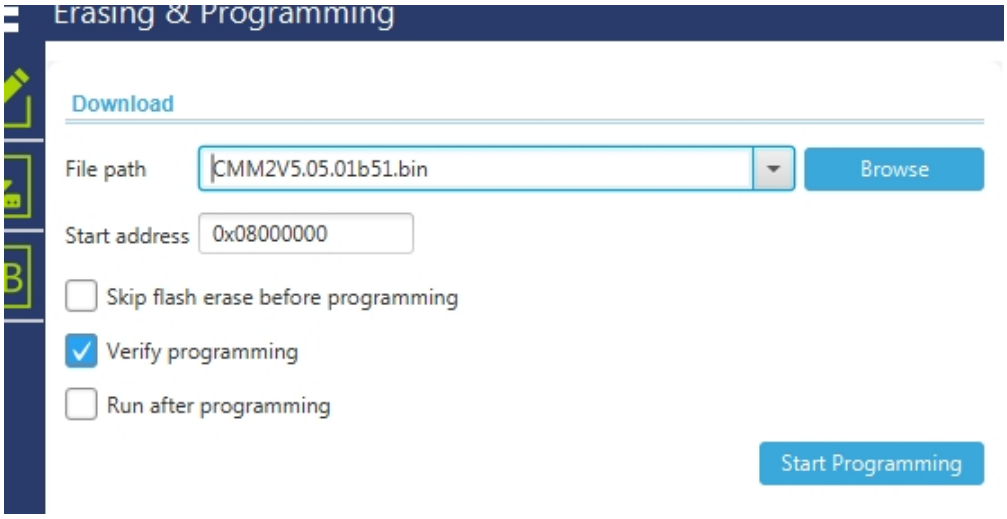
Click on the "Connect" button. You should then see a series of messages as shown in the screenshot below finishing with the message "Data read successfully". Any messages in red will indicate an error.



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

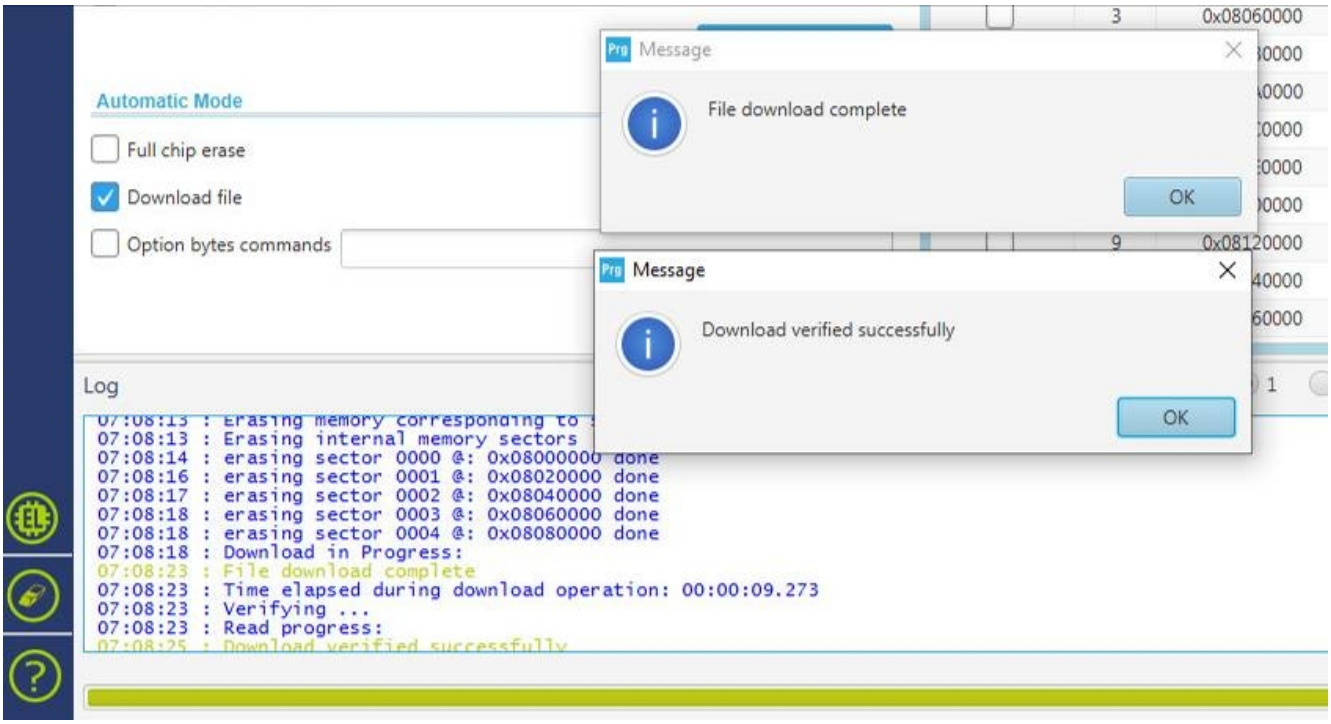
Use the "Browse button" to select the firmware file (it will have an extension of .bin).
Tick the "Verify programming" checkbox.

Finally, click on the "Start Programming" button.



The STM32CubeProgrammer software will then program the firmware into the flash memory on the STM32 CPU on STM32F407VET6 development board (the STM32CubeProgrammer software calls this "downloading"). After a short time a dialog box will pop up saying that "File download completed". **Do not do anything at this point** as the software will then start reading back the firmware programmed into the flash.

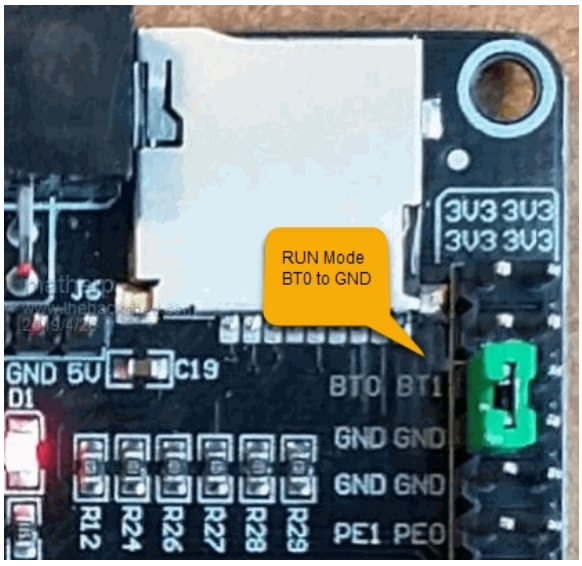
When this has completed successfully another dialog box will pop up saying "Download verified successfully" as shown below. The whole operation will take under a minute and any messages in red will indicate an error.



Then:

- Dismiss all the dialog boxes and close the STM32CubeProgrammer software.
- Disconnect the USB cable from the STM32F407VET6 development board.
- Set the Boot 0 and Boot 1 jumpers as below to enable the USB as the console by setting jumpers as below.

BT0 to GND, BT1 removed.

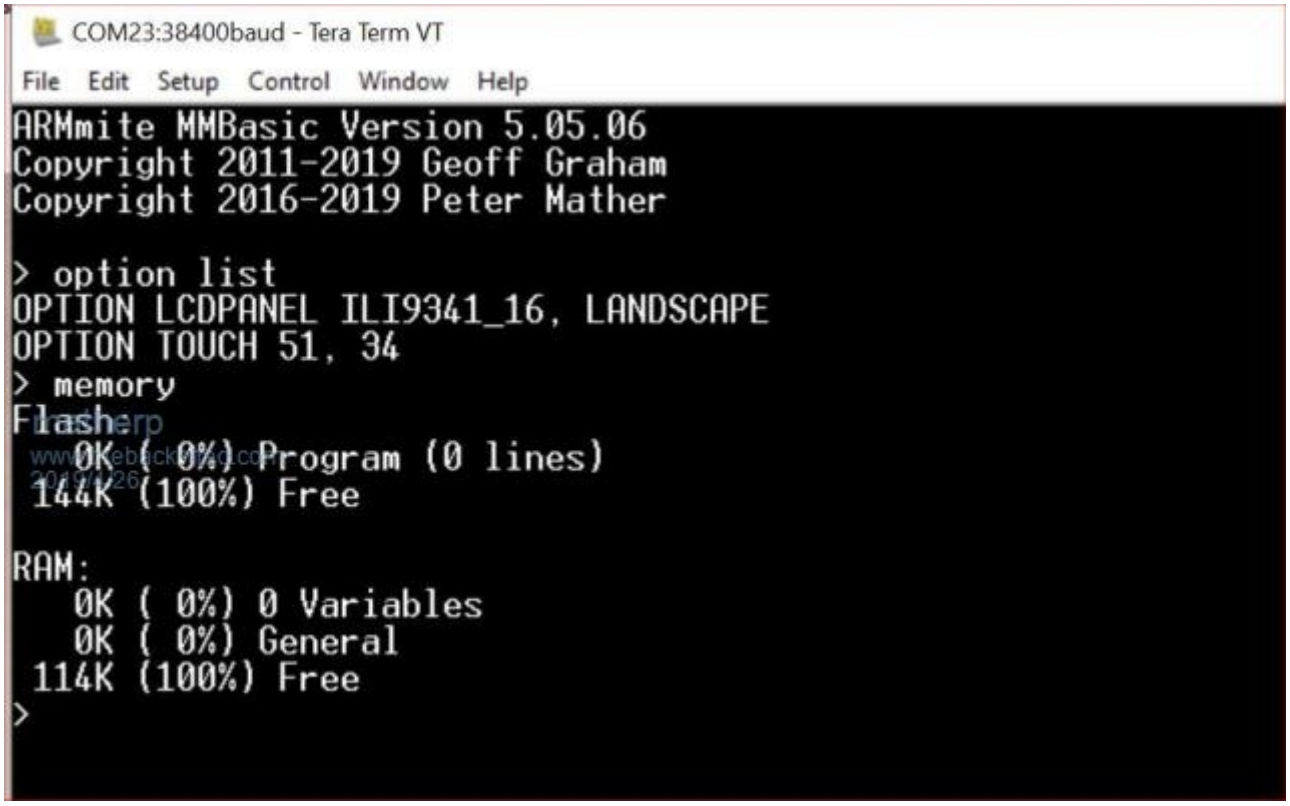


- Reconnect the USB cable to the STM32F407VET6 development board.
  This should power up the STM32F407VET6 which will then connect to your desktop computer via USB.
  In Windows the connection will appear in Device Manager as "USB Serial Port" as illustrated on the right (the COM number will probably be different):

If this doesn't happen immediately try pressing the RST button on the board and unplugging and re-plugging the USB connection. Some computers seem to take time to recognise a different device on the same physical USB port.

On Windows 10 the driver for the USB console is included with Windows 10. On Win7 and earlier the USB console requires STMicroelectronics Virtual COM Port drivers to be installed. (This driver is separate from the drivers installed with STM32 Cube Programmer software). Virtual COM Port drivers at www.st.com/en/development-tools/stsw-stm32102.html

Connect a terminal emulator to the port and you should see the Armmite copyright banner.



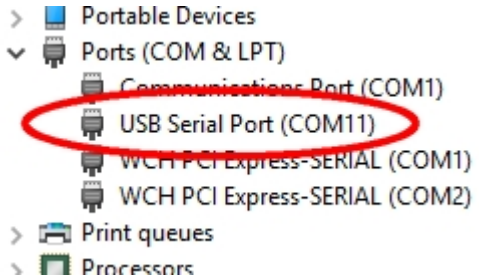
If not, press return to wake up the USB connection and it this still doesn't work try disconnecting the terminal emulator and re-connecting, pressing RST, unplugging and re-plugging.

The Armmite firmware controls the CDC connection as follows:
On power up, if no USB connection is plugged in (separate 5V supply) console output will be black-holed.
On power up, if a USB connection is plugged in console output will be buffered until a terminal emulator is connected.
Once running, if the USB connection is removed (separate 5V power) console output is black-holed
Once running, if the USB connection is re-inserted, console output will be restored from the point at which the USB was re-connected.

Alternative Method – Using COM 1

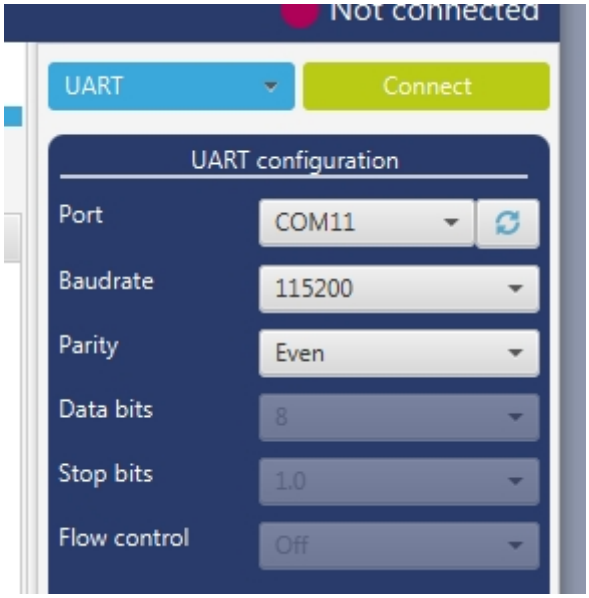
An alternative method of loading the firmware is via COM1 using a USB to serial adaptor.You may want to try this as a trouble shooting step if you fail to program using the USB port on the board. Connect the USB to J6 the COM1 port on the STM32F407VET6. The USB to serial adaptor is coverted near the start of this manual.

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



Install the STM32CubeProgrammer software on your computer as described above.

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



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

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

The whole operation will take about 5 minutes.

When the programming/verify has completed set the Boot 0 and Boot 1 jumpers to enable the console.
BT0 to GND, BT1 removed.
Note: MMBasic will start up with USB Console enabled as the default. See section at the start of this manual to switch to serial console at startup.

Linux and the Raspberry Pi

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

Appendix M – Alternate Commands and Functions

Background

The number of command and functions in MMBasic is limited to 128 of each. Over time this limit has been reached and some rearrangements on commands/functions has taken place to allow increased functionaliy to be included. Typically a stand alone command/function is merged as a variant of another command /function, freeing up its original command slot.e.g. FFT becomes MATH FFT. A new function BIN$ merges three existing functions BIN$, HEX$ and OCT$.

In some cases the old command/function is still accepted as a valid syntax and will match its new command/function slot. It will however appear in its new format when opened in EDIT or listed via the LIST command.

In the table below the current/displayed syntax is shown in **bold** underneath any also acceped version of the command/function for each device type.

Also show are commands that are different between platforms.e.g. NAME vs RENAME

Also show commands treated as separate functionality, but implemented as the same command. e.g. CAT and INC

BLIT/SPRITE

Table of Accepted and Core Syntax

Item/Functionality	Micromite(s)	Armmite F4	Armmite H7	PicoMite(s0
BIN$()	BIN$()	BIN$()	BIN$() **BASE$(2, …)**	BIN$()
HEX$()	HEX$()	HEX$()	BIN$() **BASE$(16, …)**	HEX$()
OCT$()	OCT$()	OCT$()	BIN$() **BASE$(8, …)**	OCT$()
Humidity DHT22	CSUB (HUMID)	HUMID **DEVICE HUMID**	DHT22 HUMID **DEVICE HUMID**	DEVICE HUMID
WS2812 Leds	n/a	WS2812 **DEVICE WS2812**	WS2812 **DEVICE WS2812**	DEVICE WS2812
BITSTREAM	CSUB (Bitstream)	**DEVICE BITSTREAM**	**DEVICE BITSTREAM**	DEVICE BITSTREAM
LCD	LCD	LCD **DEVICE LCD**	LCD **DEVICE LCD**	**DEVICE LCD**

Fast Fourier Transform	n/a	**MATH FFT**	**MATH FFT**	**MATH FFT**
Rename a file	**NAME** (MM+ only)	**NAME**	**NAME**	**RENAME**
	n/a	MM.INFO$ **MMINFO**	MM.INFO$ **MMINFO**	MM.INFO$ **MMINFO**
Device Type		MM.DEVICE$	MM.DEVICE$	
Version		MM.VER	MM.VER	
Error No.		MM.ERRNO	MM.ERRNO	
Error Message		MM.ERRMSG$	MM.ERRMSG$	
		MM.ONEWIRE	MM.ONEWIRE	
		MM.FONTHEIGHT	MM.FONTHEIGHT	
		MM.FONTWIDTH	MM.FONTWIDTH	
ERASE nominated variable	**ERASE**	**ERASE**	ERASE **CLEAR VARS**	ERASE
Clear all variables	**CLEAR**	**CLEAR**	**CLEAR**	CLEAR
Blit and Sprit	**BLIT**	**BLIT**	SPRITE **BLIT**	**BLIT and SPRITE**
CAT and INC are the same command	**n/a**	CAT **INC**	CAT **INC**	CAT **INC**