Armmite H7

User Manual

MMBasic Ver 5.07.02 +

A consolidated manual for the Armmite H7

DRAFT 0

5.07.02 Beta 0

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.

https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=13542

and http://fruitoftheshed.com/MMBasic.Armmite-H7-User-Manual-and-Firmware-Links.ashx

About

The Armmite H7 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). In addition, many others have supported the project with specialised code, testing and suggestions.

Support

Support questions should be raised on the Back Shed forum (http://www.thebackshed.com/forum/Microcontrollers) where there are many enthusiastic MMBasic users who would be only too happy to help. The developers of both MMBasic the Armmite H7 port are also regulars on this forum.

Copyright and Acknowledgments

This Manual

This manual is the consolidated Armmite H7 manual and relies heavy on content from the following manuals:

Armmite H7 User Manual	- Peter Mather.
Micromite User Manual	- Geoff Graham.
Micromite Plus User Manual	- Geoff Graham.
Colour Maximite 2 User Manual	- Geoff Graham.
PicoMite User Manual	- Geoff Graham.
Micromite Extreme User Manual	- Peter Mather.

Much information is also gleaned from posts (mainly by Peter Mather) in various threads relating to Armmite H7 on The Back Shed Forum.

Contents

Introduction

This consolidated manual for the Armmite H7 is a complete manual and includes the extra features in the Armmite H7 along with all the features of the BASIC language, input/output, communications, etc. Some commands have changed slightly but for the main part Micromite programs will run unchanged on the Armmite H7.
The following summarises features of the Armmite H7 as compared to the standard Micromite and the Micromite Plus:
The Armmite H7 is based on the STM32H743ZI. It has up to fifteen times the program space of the MX series used in the standard Micromite and is many times faster. The Armmite H7 is roughly 2x faster than even a 252MHz PIC32MZ.
The Armmite H7 uses the built in hardware floating point capability of the STM32H743ZI processor which is much faster than floating point on the standard Micromite and uses double precision floating point.
The Armmite H7 (144 pin) has 96 free I/O pins. The Armmite H7 (144 pin) supports 25 16-bit analogue inputs.
The Armmite H7 has two I^2C ports, four SPI ports, eight high speed PWM channels, two 12-bit DACs, a high speed counter input, and four serial COM ports.
A single MMBasic firmware now supports both the 144 pin Nucleo board and the 100 pin WeAct Studio and DevEBox boards.

Micromite Family Summary

The Micromite Family consists of several major types, the standard Micromite, the Micromite Plus, the Micromite eXtreme, the PicoMite, the Armmite L4, the Armmite F4 ,the Armmite H7 and CMM2. 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.
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 144 pin processor. This is the highest speed single-chip Micromite currently available. The same binary also runs on the 100pin WeAct and DevEBox STM32H743VIT6 boards.
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	64/100-pin SMD	100/144/64 -pin SMD	100 pin STM32F407VET6	144 pin NUCLEO-H743ZI2 100 pin WeAct 100 pin DevEbox
Maximum CPU Speed	48 MHz	48 MHz	120 MHz	252MHz	168MHz	480MHz
Maximum BASIC Program Size	59 KB	59 KB	100 KB	540 KB	132K	512KB
RAM Memory Size	52 KB	52 KB	108 KB	460 KB	114K	512KB
Clock Speed (MHz)	5 to 48	5 to 48	5 to 120	200 to 252	168MHz	480
Total Number of I/O pins	19	33	45/77	75/115/46	47	102/
Number of Analog Inputs	10	13	28	40/48/24	13	26/
Number of Serial I/O ports	2	2	3 or 4	3 or 4	4	4
Number of SPI Channels	1	1	2	3/3/2	2	3/2 +SYS SPI
Number of I^2C Channels	1	1	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	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 H/W	Double S/W	Double H/W
Power Requirements	3.3V 30 mA	3.3V 30 mA	3.3V 80 mA	3.3V 160 mA	3.3V 200mA	3.3V 200mA

Armmite H7 Features

:

The Armmite H7 is based on the STM32H743ZI. It has up to fifteen times the program space of the MX series used in the standard Micromite and is many times faster. The Armmite H7 is roughly 2x faster than even a 252MHz PIC32MZ.

The Armmite H7 uses the built in hardware floating point capability of the STM32H743ZI processor which is much faster than floating point on the standard Micromite and uses double precision floating point.

The Armmite H7 has 96 free I/O pins. The Armmite H7 supports 25 16-bit analogue inputs.

The Armmite H7 has two I^2C ports, four SPI ports, eight high speed PWM channels , two 12-bit DACs, a high speed counter input, and four serial COM ports.

The STM32H743ZI is conveniently available as a package on a development PCB the NUCLEO-H743ZI2. This was widely available through the normal suppliers (RS, Farnell, Mouser, etc.) at low cost.

The MMBasic firmware will also run on the STM32H743VIT6 100 pin version of the H7 chip. These 100 pin boards have been tested:

- STM32H743VIT6 from WeAct Studio
- STM32H743VIT6 from DevEBox.

Single MMBasic Binary supports 144 and 100 pin STM32H743 boards

A single MMBasic firmware now supports both the 144 pin Nucleo board and the 100 pin WeAct Studio and DevEBox boards. The firmware can read the package information from the chip and determine the number of pins. It also reads the RevId on the chip and determines the CPU Speed to be 400MHz or 480MHz.

It ***cannot*** directly determine the speed of the external clock driving the chip. It needs to know this. The original 144 pin chip was driven by an 8MHz clock. The two 100 pin development boards are driven by a 25MHz clock. The firmware will assume the external clock speed based on the number of pins.i.e 144 pins is 8MHz, 100 pins is 25MHz.*This will means that the support of any other boards or a self designed one, is dependant on the external clock selection following this convention.*

Within MMBasic MM.INFO(NRPINS) will return 100 or 144 to report the number of pins.

MM.INFO$(CPUREVID) will return “1003” for the chips with 400MHz CPU and “2003” for the chips with a 480MHz CPU.

Constraints with 100 pin STM32H743 boards

Due to the reduced number of pins available the following restrictions apply to the 100pin boards.

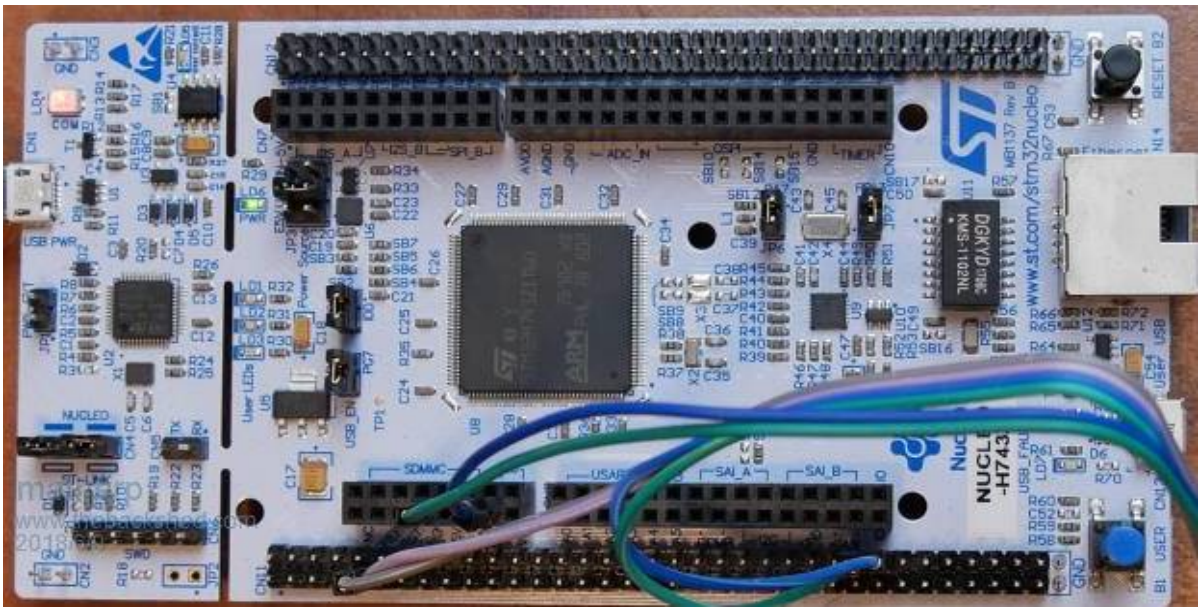
- The CAMERA command and MOVEMENT() function are not supported.
- The high speed count pin is not available.
- SPI3 is not supported.

The following restrictions are due to functions sharing pins, so they are mutually exclusive.

- PWM 2D and COM2 RX share pin 25 so only one is available at any time.
- SPI2 shares pins 42,43 and 44 with the SSD1963_x_16 parallel LCDPANELs so is not available if one is configured. The SSD1963_5 to SSD1963_8 only use 8 bits, so SPI2 is available when they are configured. Any buffered driver will use 16 bit s and these make SPI2 unavailable when configured.

144-pin Test and Development Board

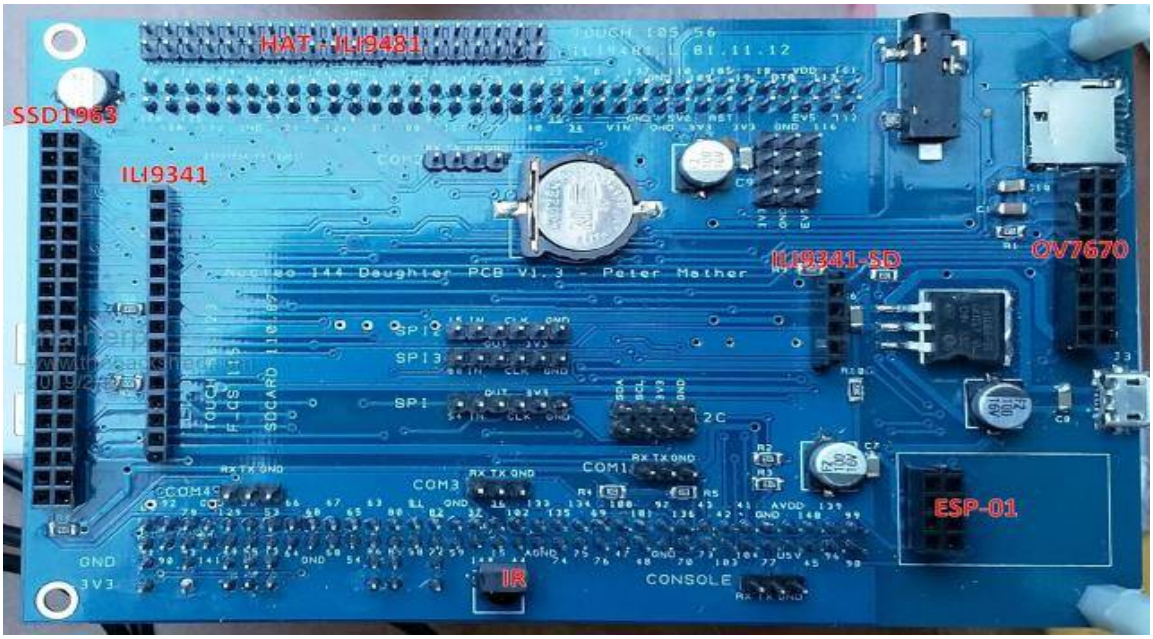
The Nucleo-H743ZI (400MHz) and Nucleo-H743ZI2 (480MHz) are complete modules and break out all the pins on the chip. They include a ST-LINK programmer for uploading the Armmite firmware and also provides a console connection via a USB CDC connection. They can be used standalone, or can be mounted directly onto a back pack PCB which provides connections to various LCD panels.



Gerbers for the backpack which is shown below are available on

http://www.thebackshed.com/forum/forum_posts.asp?TID=10700&PN=1

This backback sits directly over the Nucleo board and exposes various useful connections to allow connection without tedious wiring.



The power selector (JP3) on the Nucleo should be selected to E5V to use the daughter board

NUCLEO ZI

It is recommended that the following links on the Nucleo are removed:

JP4, SB13, SB160, JP6, SB127, SB125, SB164, SB178, SB181, JP7, SB183, SB182

In addition you must remove SB156 if using an external battery connected to VBAT

NUCLEO ZI2

480MHz NUCLEO-HZ43Z12 board.

User Manual for ZI2 board here.

https://www.st.com/content/ccc/resource/technical/document/user_manual/group1/95/1a/9a/89/87/6a/45/70/DM00499160/files/DM00499160.pdf/jcr:content/translations/en.DM00499160.pdf

Schematic for the VI2 board here.

https://www.st.com/content/ccc/resource/technical/layouts_and_diagrams/schematic_pack/group1/79/99/be/65/3c/76/40/d6/MB1364-H743ZI-C01_Schematic/files/MB1364-H743ZI-C01_Schematic.pdf/jcr:content/translations/en.MB1364-H743ZI-C01_Schematic.pdf

https://crates.io/crates/nucleo-h743zi

On the VI2 board it is recommended that the following links are set as indicated:

LINK STATUS PIN
JP6 OFF PB13 E-NET
SB27 OFF PG11 " "
SB29 OFF PC5 " "
SB30 OFF PG13 " "
SB31 OFF PA7 " "

SB21 OFF PA11 USB D-
SB22 OFF PA12 USB D+
SB23 OFF PA9 USB vBus
SB24 OFF PA10 USB ID

SB36 OFF PC4

SB57 OFF PA1 E-NET
SB64 OFF PC1 " "
SB72 OFF PA2 " "

SB76 OFF PG7 general IO and not connected to USB Over current.
SB77 ON PD10 (DEFAULT) PD10 is used for USB power on V2.

SB15 ON PC9 (99) to CN12 pin 1
SB14 ON PC8 (98) to CN12 pin 2

SB52 OFF If VBat is from external backup source.
You must remove SB52 if using an external battery connected to VBAT

ST-LINK V2/V3 Details and Update

The NUCLEOH743ZI has an ST-LINK V2 connected.

The NUCLEOH743ZI2 has an ST-LINK V3 connected.

You should be aware with the ZI2 of one possible issue. There is a design flaw on the ZI2 that doesn't exist on the ZI. The ST-LINK in both cases generates the 8MHz system clock for the H743. In the case of the ZI this is

derived directly from the crystal clocking the ST-LINK V2 circuit. In the case of the ZI2 it is derived from the internal RC oscillator on the ST-LINK V3 processor. i.e. not very accurate.

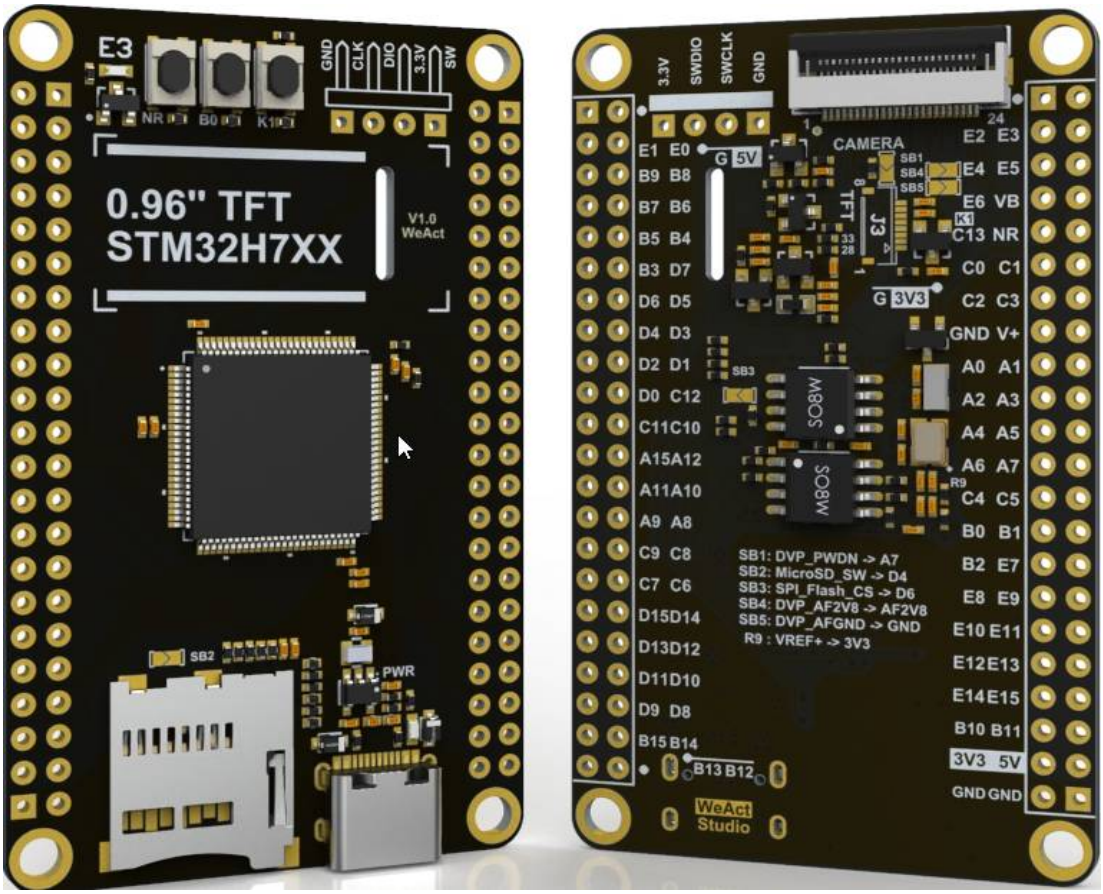
There is now new firmware for the V3 which has the option of using a crystal derived clock but it isn't at 8MHZ but rather 8.33MHZ. This seems to be a setup option when updating the firmware in the ST-LINK.

DO NOT choose this option if presented with it.

100-pin Board WeAct Studio

WeAct 100 pin with 25MHz crystal oscillator and onboard SD Card

https://github.com/WeActTC/MiniSTM32H7xx



The SDCARD mounted on the board is supported with OPTION SDCARD 79

The onboard W25Q16 SPI Flash chip is connected to SPI1 and its F_CS pin is 87 (PD6). It can be accessed from Basic. Use SPI and set pin 87 as an output and use as the F_CS. See appendix E

The onboard W25Q16 QSPI Flash is not supported.

The onboard ST7899 LCD is not supported. It may be supported later via a CSUB.

The USB connector is a USB-C. It can be used to provide power and also is used to load the firmware. It does NOT provide a USB console for MMBasic, it can be used with an USB-C OTG adapter to connect a USB Keyboard.

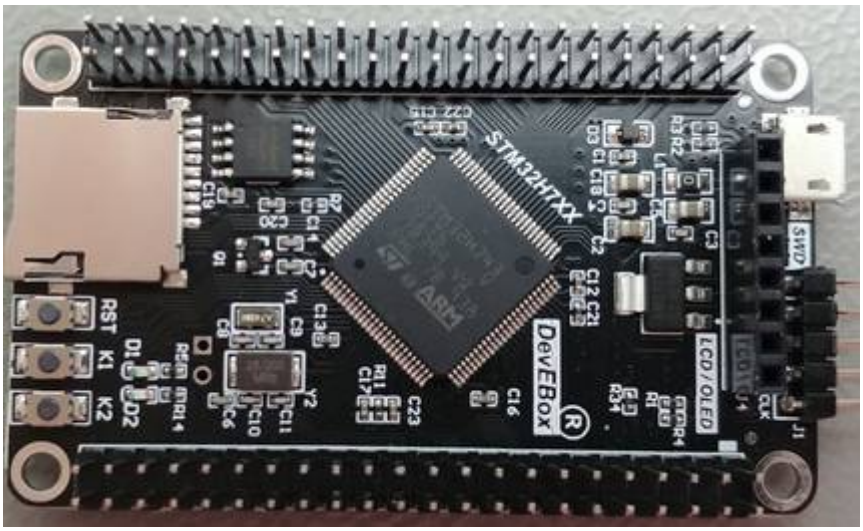
Hot Tip	The WeAct has a diode to protect the USB +5v if the board is powered from an external +5v source. This means power will not pass to a USB keyboard connected to the USB-C connector. The SB6 pads located underneath the board, directly below the USB-C connected can be bridged to bypass the protection diode and allow power to the keyboard.

100-pin Board DevEBox

DevEBox 100 pin with 25MHz crystal and onboard SD Card

https://stm32-base.org/boards/STM32H743VIT6-STM32H7XX-M.html

https://github.com/mcauser/MCUDEV_DEVEBOX_H7XX_M



The SDCARD mounted on the board is supported with OPTION SDCARD 79

The onboard W25Q16 QSPI Flash is not supported.

The USB connector is a Micro USB. It is used to provide power and also is used to load the firmware. It does NOT provide a USB console for MMBasic, it can be used with a Micro USB OTG adapter to connect a USB Keyboard. (Power i.e. +5v will need to be connected to 5v via another means)

Hazard	The +5V pins on this board are directly connected to the +5V pin of the USB connector. There is no protection in place. Do not power this board through USB and an external power supply at the same time.

Hazard	On DevEBox board K1 and K2 place GND directly on PE3 and PC5 respectively. You should ensure you don't operate these when the CPU or a connected device (e.g.LCD Panel) is trying to drive these pins high as this may damage the device or CPU. As a precaution I have removed these keys from my board.

480MHz clock

The Armmite H7 is the fastest single chip implementation of MMBasic. The early revision chips were 400MHz but they later revision is 480MHz

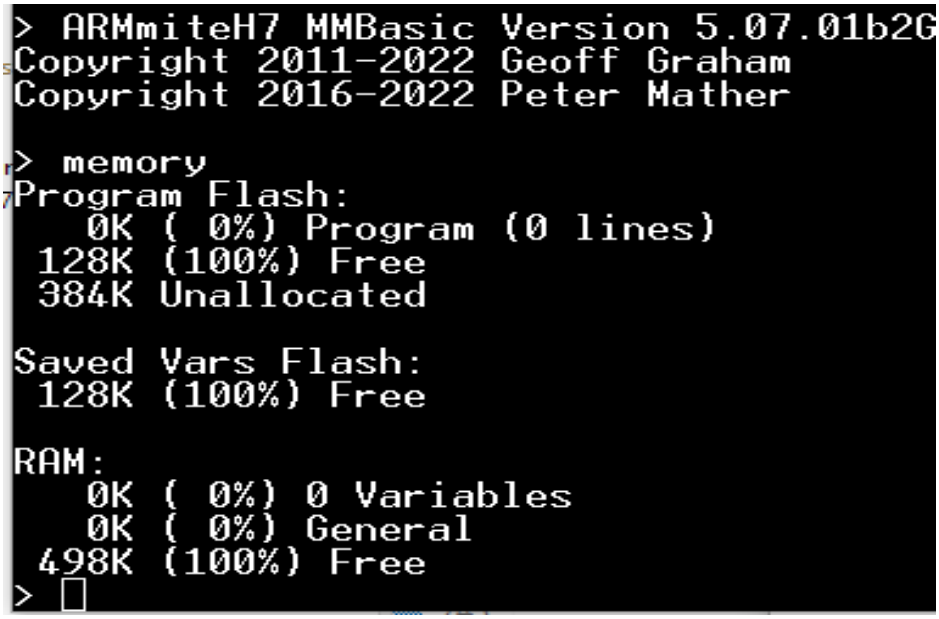
512Kbyte program and 498Kbyte variable space

The Armmite H7 supports MMBasic programs up to 512Kbytes in size in multiples of 128Kb. By default the maximum program size is set to 128Kb to expedite loading and saving programs but this can be extended as required with the ***OPTION FLASHPAGES*** command. If the program space is limited to 384Kb i.e. 3 pages, then the 4th 128K can be used for the Library.

128Kb is available to save nominated program variables which can be restored later. Use VAR SAVE and VAR RESTORE

Variable space is always 498Kbyte unless buffered drivers for 800*480 LCDPanels are loaded.

The MEMORY command can be used to show the current usage of Flash and RAM.



Input Output Pins and Protocols

The Armmite H7 has 96 free I/O pins. The Armmite H7 supports 25 16-bit analogue inputs.

The Armmite H7 has two I^2C ports, four SPI ports, eight high speed PWM channels , two 12-bit DACs, a high speed counter input, four serial COM ports 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.

Dual 12-bit DACs

The Armmite H7 has 2 12-bit DACs built into the chip. The analogue levels can be set using the DAC command. In addition they are used for the PLAY TONE, PLAY WAV, PLAY FLAC, PLAY MP3, PLAY MODFILE and TTS commands. The pins cannot be used for general purpose I/O. The DACs support an arbitrary function generator capability using the DAC START command

16-bit ADCs

All analogue to digital conversion can be carried out in 16-bit resolution, 14-bit resolution, 12-bit resolution, 10-bit resolution, and 8-bit resolution. In addition, the ADC can read the voltage being output on the DACs, 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.

Double Precision Floating Point

The Armmite H7 uses the hardware floating point capability of the STM32H743ZI chip and can therefore process floating point calculations faster than the Micromite and Micromite Plus. All floating point uses double precision calculations.

Four Serial Ports

The four serial ports share pins used for other functions, so may not be available if the other functions are required. There is a dedicated serial console. There is no USB console.

Eight PWM Channels

Minimum frequency is 1Hz, maximum is 24MHz. Duty cycle and frequency accuracy will depend on frequency. The frequency can be any value of 240,000,000/n.

Three SPI Channels

The Armmite H7 supports three SPI channels. The second and third channels operate the same as the first, the only difference is that the commands use the notation SPI2 and SPI3 (for example SPI3 WRITE, etc).

Note that if the Armmite H7 is configured for a SPI based LCD panel, touch or an SD card then an additional SPI channel is used and does not impact the other three.

SPI3 is not available on the 100 pin boards.

Two I2C channels

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

The implementation does not support I2C slave mode

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

I2C ports can be used for general purpose I/O when not used for I2C

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.

Longstring handling

The Armmite H7 supports a comprehensive set of commands and functions for handling long strings stored in integer arrays

Battery Backed-up On-Chip Real time clock (RTC)

The Armmite H7 includes a built-in RTC. Date$ and Time$ are derived from the STM32H743ZI on-chip real time clock. If a 3V lithium battery is connected the STM32H743ZI will maintain the time even when powered off. It 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 can be accessed with PEEK and POKE. 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. An additional MMBasic function ***DAY$*** returns the day of the week as a string. 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.

VAR command

The Armmite H7 allows up to 128Kbs of variables to be saved

USB Keyboard support

The Armmite H7 supports a USB keyboard in US, UK, DE (German), FR (French), ES (Spanish) and BE(Belgium) format. See Connecting USB Keyboard for details.

Buffered Drivers for all displays

All displays up to and including 480x320 pixel resolution automatically use a buffered driver where a memory image of the display is maintained in the Armmite's memory. This does not impact user RAM space but helps reduce artefacts whilst writing to the screen. Commands ***OPTION AUTOREFRESH*** and ***REFRESH*** can be used to control when the updates to the screen take place allowing the programmer maximum flexibility in using the screen effectively. An 8-bit buffered driver (64 colours) is included for 800x480 SSD1963 displays. This is specifically targeted at games programming as user RAM is still 473K with the driver loaded. A RGB565 buffered driver is also included for 800x480 displays but this reduces user RAM significantly (down to 98Kbytes free).

16-bit Interface to SSD1963 Based LCD Displays

The Armmite H7 can drive a SSD1963 display using a 16-bit parallel bus for extra speed. The extra I/O pins for this are listed as SSD1963-DB8 to SSD1963-DB15 on the pinout tables in this manual and they must be connected to the pins labelled DB8 to DB15 on the I/O connector on the SSD1963 display.

Note that in this mode the SSD1963 controller runs with a reduce colour range (65 thousand colours) compared to 16 million colours with the normal 8-bit interface.

WS2812 support

The Armmite H7 supports the WS2812 Led driver. This chip needs very specific timing to work properly and by incorporating support in the Armmite H7 firmware the user can program these chips with minimum effort. The command ***BITSTREAM WS2812*** is used to set the colours of the LEDs. There is a nominal 512 limit to the number of LEDs and using a high number of LEDs may cause unforeseen issues with missed interrupts.

GPS support

The Armmite H7 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()*** function can then be used to interrogate the GPS data which is automatically parsed in the Armmite firmware. In addition ***PRINT #GPS,string$*** can be used to automatically append a correct checksum to a GPS message.

High speed frequency counter support

The Armmite H7 (144 pins) supports a frequency counter input to pin 93. This supports counting high speed signals (tested to 20MHz, but may go considerably higher).

Not available on the 100 pin boards.

OV7670 camera support with movement detection

The Armmite H7 (144 pins) supports a non-buffered OV7670 camera in full colour 640x480 pixel mode using the ***CAMERA*** command. To use this a 800x480 SSD1963 display must be connected and configured using a special driver (***OPTION LCDPANEL SSD1963_5_640, orientation*** or ***OPTION LCDPANEL SSD1963_7_640, orientation***) in landscape or reverse landscape mode. The driver maintains a complete image in Armmite memory which allows the software to compare a newly captured image with a stored one to check for movement using the ***MOVEMENT()*** function.

Not available on the 100 pin boards due to reduced number of pins

Sprites

The Armmite H7 supports a complete implementation of sprites including screen scrolling and collision detection. BLIT vs SPRITE

Extended WAV File Playback

The Armmite H7 can play WAV files (like the Micromite Plus) however, it is also capable of playing WAV files recorded with sampling rates of 24 KHz, 44.1KHz, and 48 KHz.

Random Number Generation

The Armmite H7 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.

OPTION VCC command

The Armmite H7 supports the OPTION VCC command. This allows the user to precisely set the supply voltage to the chip and is used in the calculation of voltages when using analogue inputs e.g. OPTION VCC 3.15. The parameter is not saved and should be initialised either on the command line or in a program. See Setting Option VCC for more detail.

OPTION CPU SPEED command

The Armmite H7 supports the OPTION CPU SPEED command. This allows the user to set the speed of the core processor clock, values between 10MHz and 600MHz are allowed, anything over 480 MHz is overclocking the chip beyond the manufacturer's specification.

Audio

The ArmmiteH7 can play WAV, FLAC and MP3 files from the SD card, generate synthesised music in the MOD format, create robot speech and sound effects as well as generate precise sine wave tones. All these are outputted on the audio socket. The ARM Cortex-M7 chip includes its own DAC (digital to analog converter) so an output filter network is not needed (as on the original Colour Maximite).

See the Sound Output section for more detail.

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.

Affordable 9 DoF Sensor Fusion

The Armmite H7 supports the calculation of pitch, roll and yaw angles from accelerometer and magnetometer inputs. For information on this technology see https://github.com/kriswiner/MPU-6050/wiki/Affordable-9-DoF-Sensor-Fusion

The MATH SENSORFUSION command supports both the MADGWICK and MAHONY fusion algorithms. The format of the command is:

MATH SENSORFUSION type ax, ay, az, gx, gy, gz, mx, my, mz, pitch, roll, yaw [,p1] [,p2]

See Appendix G – Sensor Fusion

MM.DEVICE$ - MM.INFO$(DEVICE)

On the Armmite H7 the read only variable MM.DEVICE$ will return " Armmite H7".

Unsupported commands

The Armmite H7 does not currently support dynamically changing the CPU speed or the sleep function. Accordingly the commands CPU SPEED and CPU SLEEP are 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.

The latest Armmite H7 software FIXME is normally available on The Back Shed (TBS) forum. You will need to scroll through the thread and selected the latest version. Download it and extract the ArmmiteH7.bin or similar file to your computer.

Appendix H and I 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 K – Loading the Firmware (100 pin boards) or Appendix L – Loading the Firmware (Nuclo 144 pin) now.

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

Program Memory not cleared when firmware updated

If you reload the firmware after you have been using the Armmite, note that loading the new firmware will not clear any previously loaded program. If the new firmware has additional commands or functions then the program maybe corrupt as command or function tokens may have changed. A NEW will clear the program. It would need to be reloaded to convert to the new command tokens. In most cases the program is intact and will still run correctly.

Options not cleared when firmware updated

If you reload the firmware after you have been using the Armmite, note that loading the new firmware will not clear the previous options as seen via OPTION LIST. These are stored in flash so loading the firmware won't change them. If you don't want your original options, use OPTION RESET to set the default options.

Saved variables not cleared when firmware updated

This is unlikely to be noticed or cause an issue. Its just here for completeness.

If you reload the firmware after you have been using the Armmite, note that loading the new firmware will not clear any previous variables saved with VAR SAVE. You are unlikely to notice this as a NEW, LOAD program, AUTOSAVE or XMODEM RECEIVE to get a program back into the Armmite will automatically clear the variables.VAR CLEAR will also remove them.

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

Power and Console Connections

Power Requirements

The USB connector is for power only. The power requirement of the Armmite H7 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 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 H7 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 H7 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 WeAct and DevEBox STM32F743VIT6 do not. The USB 5v and the 5v pins on the board are connected together. 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 with your terminal (PC) at the same time to upgrade the firmware. When connecting a USB Keyboard to the USB connection the external 5V is actually required.

Console Connection

The Armmite H7 console is a serial connection. On the WeAct and DevEBox development boards connect to the console via PD8 and PD9. On the Nucleo development board the console is extended via the ST-LINK connection when it is not in Boot Loader mode. The serial console defaults to 115200 bauds.

If changed the console baud rate will be permanently remembered until 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 in Troubleshooting the Console.

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.

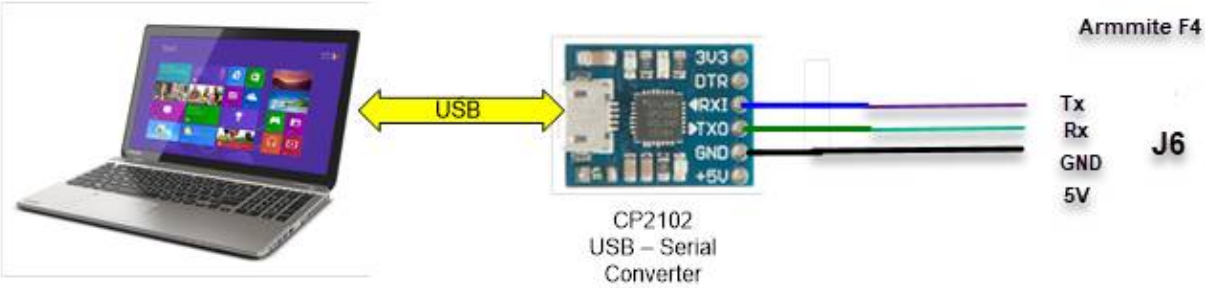
The serial console when enabled defaults to 115200 bauds, which uses TTL signal levels. This is similar to the RS232 interface on older personal computers but the TTL signal level is inverted and swings from zero to 3.3V.

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 H7. 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.

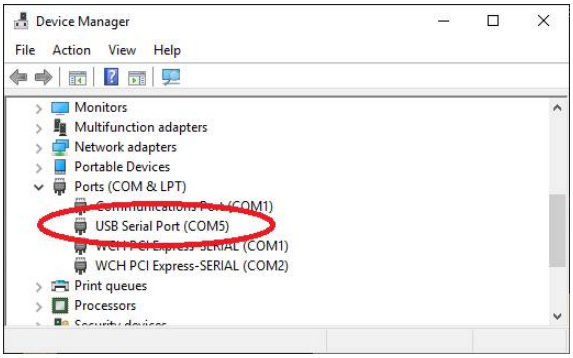
The serial interface side of the converter will generally have a ground pin and a 5v power output pin and this can be used to power the Armmite. The converter will also have two pins marked TX (or similar) for transmit and RX (or similar) for receive. The TX pin of the serial converter must go to the 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 H7. 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.

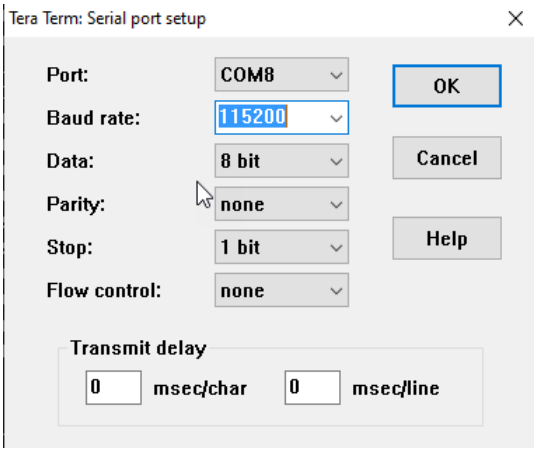


VT100 Terminal Emulators

You also need a terminal emulator program on your desktop computer. This program acts like an old fashion 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 H7 (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 H7 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 the Console

If you cannot see the startup banner try the following:

- 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.

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

Pin and Connector Capabilities

STM32H743ZI / STM32H743ZI2 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.

Pin	Features				
1-PE2	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB2		
2-PE3	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB3		
3-PE4	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB4		
4-PE5	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB5		
5-PE6	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB6		
6	VBAT			Backup Battery	
7-PC13	DIGITAL_IN	DIGITAL_OUT	NUCLEO-B1 (K1) BLUE KEY		MMBasic Reset
8	32KHz Xtal				
9	32KHz Xtal				
10-PF0	DIGITAL_IN	DIGITAL_OUT			COUNT1
11-PF1	DIGITAL_IN	DIGITAL_OUT			COUNT2
12-PF2	DIGITAL_IN	DIGITAL_OUT			COUNT3
13-PF3*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		COUNT4
14-PF4*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		IR
15-PF5*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		
16	GND				
17	VDD				
18-PF6*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	SD-CS	
19-PF7*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	SPI5-CLK	SYSTEM SPI
20*-PF8	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	SPI5-IN	“
21-PF9*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	SPI5-OUT	“
22-PF10*	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		
23-PH0	8MHz OSC				
24-PH1	N/C				
25	NRST			RESET	
26-PC0	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	OV7670-D0	
27-PC1	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	OV7670-D1	SPI2-OUT
28-PC2	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	OV7670-D2	SPI2-IN
29-PC3	ANALOG_C	DIGITAL_IN	DIGITAL_OUT	OV7670-D3	
30	VDD				
31	ANALOG-GND				
32	VREF+				
33	ANALOG-VDD				
34-PA0	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2A	

35-PA1	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2B	
36-PA2	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2C	
37-PA3	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2D	
38	GND				
39	VDD				
40-PA4	DAC1	AUDIO-L			
41-PA5	DAC2	AUDIO-R			
42-PA6	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		SPI-IN
43-PA7	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		SPI-OUT
44-PC4	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	OV7670-D4	
45-PC5	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	OV7670-D5	
46-PB0			LCD-BL	LED1-GREEN**	BACKLIGHT
47-PB1	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		
48-PB2	DIGITAL_A	DIGITAL_OUT			SPI3-OUT
49-PF11	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		
50-PF12	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		
51	GND				
52	VDD				
53-PF13	ANALOG_B	DIGITAL_IN	DIGITAL_OUT		
54-PF14	ANALOG_B	DIGITAL_IN	DIGITAL_OUT		
55-PF15	DIGITAL_IN	DIGITAL_OUT			
56-PG0	DIGITAL_IN	DIGITAL_OUT			
57-PG1	DIGITAL_IN	DIGITAL_OUT	SSD1963-RS		
58-PE7	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB7		
59-PE8	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB8		
60-PE9	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB9		
61	GND				
62	VSS				
63-PE10	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB10		
64-PE11	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB11		
65-PE12	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB12		
66-PE13	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB13		
67-PE14	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB14		
68-PE15	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB15		
69-PB10	DIGITAL_IN	DIGITAL_OUT			I2C2-SCL
70-PB11	DIGITAL_IN	DIGITAL_OUT			I2C2-SDA
71	VCAP				
72	VDD				
73-PB12	DIGITAL_IN	DIGITAL_OUT			COM3-RX
74-PB13	DIGITAL_IN	DIGITAL_OUT			COM3-TX
75-PB14	DIGITAL_IN	DIGITAL_OUT		LED3-RED (V1&V2)	

76-PB15	DIGITAL_IN	DIGITAL_OUT			COM1-RX
77-PD8	CONSOLE-TX				
78-PD9	CONSOLE-RX				
79-PD10	DIGITAL_IN	DIGITAL_OUT	Keyboard Power	V2 Nucleo	USB Power
80-PD11	DIGITAL_IN	DIGITAL_OUT			
81-PD12	DIGITAL_IN	DIGITAL_OUT		PWM-1A	
82-PD13	DIGITAL_IN	DIGITAL_OUT		PWM-1B	
83	GND				
84	VDD				
85-PD14	DIGITAL_IN	DIGITAL_OUT		PWM-1C	
86-PD15	DIGITAL_IN	DIGITAL_OUT		PWM-1D	
87-PG2	DIGITAL_IN	DIGITAL_OUT			
88-PG3	DIGITAL_IN	DIGITAL_OUT	OV7670_PCLK		
89-PG4	DIGITAL_IN	DIGITAL_OUT	OV7670_VSYNC		
90-PG5	DIGITAL_IN	DIGITAL_OUT	OV7670_HREF		
91-PG6	DIGITAL_IN	DIGITAL_OUT	Keyboard Power	V1 Nucleo	USB Power
92-PG7	DIGITAL_IN	DIGITAL_OUT			
93-PG8	DIGITAL_IN	DIGITAL_OUT			COUNT5-HS
94	GND				
95	VDDUSB				
96-PC6	DIGITAL_IN	DIGITAL_OUT	OV7670-D6		
97-PC7	DIGITAL_IN	DIGITAL_OUT	OV7670-D7		
98-PC8	DIGITAL_IN	DIGITAL_OUT			
99-PC9	DIGITAL_IN	DIGITAL_OUT			
100-PA8	DIGITAL_IN	DIGITAL_OUT	OV7670_XCLK		
101-PA9	DIGITAL_IN	DIGITAL_OUT			
102-PA10	DIGITAL_IN	DIGITAL_OUT			
103-PA11	USB-D+				
104-PA12	USB-D-				
105-PA13	SWDIO				
106	VCAP				
107	GND				
108	VDD				
109-PA14	SWCLK				
110-PA15	DIGITAL_IN	DIGITAL_OUT		SD-CS (backpack)	
111-PC10	DIGITAL_IN	DIGITAL_OUT			
112-PC11	DIGITAL_IN	DIGITAL_OUT			
113-PC12	DIGITAL_IN	DIGITAL_OUT			
114-PD0	DIGITAL_IN	DIGITAL_OUT			
115-PD1	DIGITAL_IN	DIGITAL_OUT			
116-PD2	DIGITAL_IN	DIGITAL_OUT			

117-PD3	DIGITAL_IN	DIGITAL_OUT			SPI2-CLK
118-PD4	DIGITAL_IN	DIGITAL_OUT			COM2-DE
119-PD5	DIGITAL_IN	DIGITAL_OUT			COM2-TX
120	GND				
121	VDD				
122-PD6	DIGITAL_IN	DIGITAL_OUT			COM2-RX
123-PD7	DIGITAL_IN	DIGITAL_OUT			
124-PG9	DIGITAL_IN	DIGITAL_OUT			COM4-RX
125-PG10	DIGITAL_IN	DIGITAL_OUT	SSD1963_WR		
126-PG11	DIGITAL_IN	DIGITAL_OUT			SPI-CLK
127-PG12	DIGITAL_IN	DIGITAL_OUT	SSD1963_RESET		
128-PG13	DIGITAL_IN	DIGITAL_OUT	SSD1963_RD		
129-PG14	DIGITAL_IN	DIGITAL_OUT			COM4-TX
130	GND				
131	VSS				
132-PG15	DIGITAL_IN	DIGITAL_OUT			
133-PB3	DIGITAL_IN	DIGITAL_OUT			SPI3-CLK
134-PB4	DIGITAL_IN	DIGITAL_OUT			SPI3-IN
135-PB5	DIGITAL_IN	DIGITAL_OUT			
136-PB6	DIGITAL_IN	DIGITAL_OUT			COM1-TX
137-PB7	DIGITAL_IN	DIGITAL_OUT		LED2-BLUE (V1)	
138	BOOT0				
139-PB8	DIGITAL_IN	DIGITAL_OUT			I2C-SCL
140-PB9	DIGITAL_IN	DIGITAL_OUT			I2C-SDA
141-PE0	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB0		
142-PE1	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB1	LED2-YELLOW (V2)	
143	PDR-ON				
144	VDD				

Notes:

* ADC Limited to 8bit,10bit and 12bit on these pins. 16bit and 14bit not recommend because of noise.

** LED 1 is connected to PB0 which is used to provide LCD Backlight PWM. Its intensity can be controlled with the BACKLIGHT command and defaults to 50%. PB0 is only available to MMBasic via the BACKLIGHT command.

STM32H743ZI Explanation of keys used in above table

The following summary includes information based on the authors interpretation of the STM32H743 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.

Code or Item	Details
DIGITAL_OUT DIGITAL_IN	These pins can be used as digital output and input. They are 5V tolerant and can sink or source a maximum of 25mA.
DIGITAL_OUT(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.
ANALOG_A ANALOG_B ANALOG_C	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. 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.
DAC x (3.3v)	DAC1 and DAC2 are not 5v tolerant. 3.3v only.
PULL-UP PULL-DOWN	Weak pull-ups to 3.3v are typically 40K ohms for all pins except for PA10 and PA12 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 10, 11, 12 and 13 have hardware interrupts and can be used with the SETPIN CIN, PIN and FIN options for count, period and frequency measurements.
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.
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.

STM32H743ZI 144 Pins by Function

Function	Pins
COM1	TX - 136 RX-76
COM2	TX - 119 RX - 122 DE- 118
COM3	TX – 74 RX- 73
COM4	TX - 129 RX -124
I2C	SCL – 139 SDA – 140
I2C2	SCL – 69 SDA -70
SPI	IN -42 OUT – 43 CLK – 126
SPI2	IN – 28 OUT -27 CLK - 117
SPI3	IN – 134 OUT -48 CLK - 133
SPISYS (SPI5)	IN – 20 OUT -21 CLK - 19
DAC	1-40-PA4 (Not 5V tolerant 3.3v only) 2-41-PA5 (Not 5v tolerant 3.3v only)
PWM 1	1A -81 1B- 82 1C- 85 1D- 86
PWM 2	2A- 34 2B- 35 2C- 36 2D- 37
USBKEYBOARD	USB D- USB D+ Power- ZI ZI2
Count Pins	They can be used with SETPIN CIN,FIN and PIN parameters for counting, frequency and period measurements
Analogue Pins	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

SDcard, SPI LCD and Touch Connections 144 Pins

The connections required to use an LCD panel are as follows

T_DO, SDO(MISO), SD_DO	Pin-20 (PF8)
T_DIN, SDI(MOSI), SD_DIN	Pin-21 (PF9)
T_CLK, SCK, SD_CLK	Pin-19 (PF7)
SD_CS	configurable 110
SD_CD	configurable
T_IRQ	configurable 123
T_CS	configurable 56
F_CS	configurable 115

LCD-RS	Pin-3 (xxx) configurable
LCD-RST	Pin-2 (xxx) configurable
LCD-CS	Pin-1 (xxx) configurable
LCD-BL	Pin-46 (PB0) Replaces GREEN LED

BACKLIGHT Command controls PWM on Pin 46

PB0 not directly available to MMBasic

option touch 56,123 for backpack

SSD1963, 16-bit ILI9341 Connections 144 Pins

Touch and SDcard connections as above

DB0	Pin-141 (PE0)	DB8	Pin-59 (PE8)	WR	Pin-125 (PG10)
DB1	Pin-142 (PE1)	DB9	Pin-60 (PE9)	RS	Pin-57 (PG1)
DB2	Pin-1 (PE2)	DB10	Pin-63 (PE10)	RESET	Pin-127 (PG12)
DB3	Pin-2 (PE3)	DB11	Pin-64 (PE11)	RD	Pin-128 (PG13) *
DB4	Pin-3 (PE4)	DB12	Pin-65 (PE12)	CS	GND
DB5	Pin-4 (PE5)	DB13	Pin-66 (PE13)		
DB6	Pin-5 (PE6)	DB14	Pin-67 (PE14)		
DB7	Pin-58 (PE7)	DB15	Pin-68 (PE15)		

SSD1963 displays should be configured for backlight control using 1963_PWM. Controlling the backlight using an Armmite H7 pin like the MM+ is supported.

BACKLIGHT Command controls PWM on Pin 46

PC0 not directly available to MMBasic

- Mandatory connection

OV7670 Camera connections 144 Pins

SIOC	Pin-139 (PB8)	SIOD	Pin-140 (PB9)
VSYNC	Pin-89 (PG4)	HREF	Pin-90 (PG5)
PCLK	Pin-88 (PG3)	XCLK	Pin-100 (PA8)
D7	Pin-97 (PC7)	D6	Pin-96 (PC6)
D5	Pin-45 (PC5)	D4	Pin-44 (PC4)
D3	Pin-29 (PC3)	D2	Pin-28 (PC2)
D1	Pin-27 (PC1)	D0	Pin-26 (PC0)
RESET	3.3V	PWDN	GND

STM32H743VIT6 100 Pin allocation and functions

The table shows pin allocation for the

Pin	Features				
1-PE2	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB2		
2-PE3	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB3	Blue LED WeAct	K1 DevEBox *REMOVE IT*^^
3-PE4	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB4		
4-PE5	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB5		
5-PE6	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB6		
6	VBAT				
7-PC13	DIGITAL_IN	DIGITAL_OUT	K1 key on WeAct	Not a key DevEBox	MMBasic reset
8	32KHz Xtal				
9	32KHz Xtal				
10	GND				
11	VDD				
12	25MHz OSC				
13	25MHz OSC				
14-NRST	RESET		NR key WeAct	RST key DevEBox	
15-PC0	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		
16-PC1	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		
17-PC2	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		
18-PC3	ANALOG_C	DIGITAL_IN	DIGITAL_OUT		
19	ANALOG-GND				
20	VREF+				
21	ANALOG-VDD				
22-PA0	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2A (TIM5)	
23-PA1	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2B (TIM5)	Green LED DevEBox
24-PA2	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2C (TIM5)	
25-PA3	ANALOG_A	DIGITAL_IN	DIGITAL_OUT	PWM-2D^ (TIM5)	COM2-RX^
26	GND				
27	VDD				
28-PA4	DAC1	AUDIO-L			
29-PA5	DAC2	AUDIO-R			
30-PA6	ANALOG_B	DIGITAL_IN	DIGITAL_OUT		
31-PA7	ANALOG_B	DIGITAL_IN	DIGITAL_OUT		
32-PC4	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		
33-PC5	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		K2 DevEBox *REMOVE IT*^^
34-PB0			(TIM1-CH2N)	LCD-BL	BACKLIGHT
35-PB1	ANALOG_A	DIGITAL_IN	DIGITAL_OUT		

36-PB2	DIGITAL_IN	DIGITAL_OUT			COUNT 3
37-PE7	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB7		
38-PE8	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB8		
39-PE9	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB9		
40-PE10	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB10	LCD-RS*	
41-PE11	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB11	LCD-CS*	
42-PE12	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB12	Uses SPI4 H/W	SPI2-CLK***
43-PE13	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB13	“	SPI2-MISO***
44-PE14	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB14	“	SPI2-MOSI***
45-PE15	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB15	LCD-RST*	
46-PB10	I2C2-SCL				
47-PB11	I2C2-SDA				
48	VCAP				
49	VSS				
50	VDD				
51-PB12	DIGITAL_IN	DIGITAL_OUT			COM3-RX
52-PB13	DIGITAL_IN	DIGITAL_OUT	SPI(SYS)-CLK	Not available to MMBasic	SYSTEM SPI
53-PB14	DIGITAL_IN	DIGITAL_OUT	SPI(SYS)-IN	“	SYSTEM SPI
54-PB15	DIGITAL_IN	DIGITAL_OUT	SPI(SYS)-OUT	“	SYSTEM SPI
55-PD8	CONSOLE-TX				
56-PD9	CONSOLE-RX				
57-PD10	DIGITAL_IN	DIGITAL_OUT	SSD1963-WR		
58-PD11	DIGITAL_IN	DIGITAL_OUT	SSD1963-RD		
59-PD12	DIGITAL_IN	DIGITAL_OUT	PWM-1A	(TIM4)	
60-PD13	DIGITAL_IN	DIGITAL_OUT	PWM-1B	(TIM4)	
61-PD14	DIGITAL_IN	DIGITAL_OUT	PWM-1C	(TIM4)	
62-PD15	DIGITAL_IN	DIGITAL_OUT	PWM-1D	(TIM4)	
63-PC6	DIGITAL_IN	DIGITAL_OUT			COM4-TX
64-PC7	DIGITAL_IN	DIGITAL_OUT			COM4-RX
65-PC8	DIGITAL_IN	DIGITAL_OUT	SD MISO**		
66-PC9	DIGITAL_IN	DIGITAL_OUT	SSD1963-RS		
67-PA8	DIGITAL_IN	DIGITAL_OUT	T-CS*		
68-PA9	DIGITAL_IN	DIGITAL_OUT			COM1-TX
69-PA10	DIGITAL_IN	DIGITAL_OUT			COM1-RX
70-PA11	USB-D+				
71-PA12	USB-D-				
72-PA13	SWDIO				
73	VCAP				
74	GND				
75	VDD				
76-PA14	SWCLK				

77-PA15	DIGITAL_IN	DIGITAL_OUT	T-IRQ*		
78-PC10	DIGITAL_IN	DIGITAL_OUT	SD-CS*		
79-PC11	DIGITAL_IN	DIGITAL_OUT	SD-CS**		
80-PC12	DIGITAL_IN	DIGITAL_OUT	SD CLK**		
81-PD0	DIGITAL_IN	DIGITAL_OUT			COUNT 1
82-PD1	DIGITAL_IN	DIGITAL_OUT			COUNT 2
83-PD2	DIGITAL_IN	DIGITAL_OUT	SD MOSI**		
84-PD3	DIGITAL_IN	DIGITAL_OUT			COUNT 4
85-PD4	DIGITAL_IN	DIGITAL_OUT			COM2-DE
86-PD5	DIGITAL_IN	DIGITAL_OUT			COM2-TX
87-PD6	DIGITAL_IN	DIGITAL_OUT	F-CS on WeAct		
88-PD7	DIGITAL_IN	DIGITAL_OUT	SPI-OUT		
89-PB3	DIGITAL_IN	DIGITAL_OUT	SPI-CLK		
90-PB4	DIGITAL_IN	DIGITAL_OUT	SPI-IN		
91-PB5	DIGITAL_IN	DIGITAL_OUT	SSD1963-RST		
92-PB6	DIGITAL_IN	DIGITAL_OUT			COM3-TX
93-PB7	DIGITAL_IN	DIGITAL_OUT			IR
94	BOOT0	3.3V to Boot 0	B0 Key on WeAct	BT0 pin DevEBox	3.3V to Boot 0
95-PB8	I2C-SCL				
96-PB9	I2C-SDA				
97-PE0	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB0		
98-PE1	DIGITAL_IN	DIGITAL_OUT	SSD1963-DB1		
99	VSS				
100	VDD				

Notes:
* Nominal but not mandatory allocation of pin to the function
** Refers to pins used if bitbanged onboard SDCard is enabled
*** Not available if parallel display in 16bit mode is used
^ Only one of COM2 and PWM 2D can be active at the same time.
^^ K1 on the DevEBox puts GND on PE3. This could damage the LCDPanel or the STM32H743 if it is pressed while the display is configured.The safest thing is to remove it.
^^ K2 on the DevEBox puts GND on PC5. This could damage the the STM32H743 if it is pressed while trying to drive PC3 high. You should consider removing it.

Option SDcard, SPI LCD and Touch Connections 100 Pins

The connections required to use an LCD panel are as follows

T_DO, SDO(MISO), SD_DO	Pin-53 (PB14)
T_DIN, SDI(MOSI), SD_DIN	Pin-54 (PB15)
T_CLK, SCK, SD_CLK	Pin-52 (PB13)
SD_CS	Pin-78 (PC10) configurable
SD_CD	Not used
T_IRQ	Pin-77 (PA15) configurable
T_CS	Pin-67 (PA8) configurable
LCD-RS	Pin-40 (PE10) configurable
LCD-RST	Pin-45 (PE15) configurable
LCD-CS	Pin-41(PE11) configurable
LCD-RS	Pin-40 (PE10) configurable
LCD-BL	Pin-34 (PB0)

BACKLIGHT Command controls PWM on Pin 34

PB0 not directly available to MMBasic

SSD1963, 16-bit ILI9341 Connections 100 Pins

Touch and SDcard connections as above

DB0	Pin-97 (PE0)	DB8	Pin-38 (PE8)	WR	Pin-57 (PD10)
DB1	Pin-98 (PE1)	DB9	Pin-39 (PE9)	RS	Pin-66 (PC9)
DB2	Pin-1 (PE2)	DB10	Pin-40 (PE10)	RESET	Pin-91 (PB5)
DB3	Pin-2 (PE3)	DB11	Pin-41 (PE11)	RD	Pin-58 (PD11)
DB4	Pin-3 (PE4)	DB12	Pin-42 (PE12)	CS	GND
DB5	Pin-4 (PE5)	DB13	Pin-43 (PE13)	BL	Pin-34 (PB0)**
DB6	Pin-5 (PE6)	DB14	Pin-44 (PE14)		
DB7	Pin-37 (PE7)	DB15	Pin-45 (PE15)		

** SSD1963 displays should be configured for backlight control using 1963_PWM.

BACKLIGHT Command controls PWM on Pin 34 for other displays.

Screen Here

		58	63													67	54		34
GND	GND	PD11	PC6									n/c	n/c	n/c	n/c	PA8	PB15	n/c	PB0
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
GND	CS	RD	RST	B0	B2	B4	B6	B8	B10	B12	B14	n/c	n/c	n/c	n/c	T-CS	MOSI	n/c	LCD-BL
5V	DC	WR	n/c	B1	B3	B5	B7	B9	B11	B13	B15	n/c	n/c	n/c	n/c	T-CLK	MISO	Pen-IRQ	GND
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
5V	PC7	PD10	n/c									n/c	n/c	n/c	n/c	PB13	PB14	PA15	GND
	64	57														52	53	77	

40 Pin connector to receive ER 5" screen viewed from the top. Screen is above the connector.

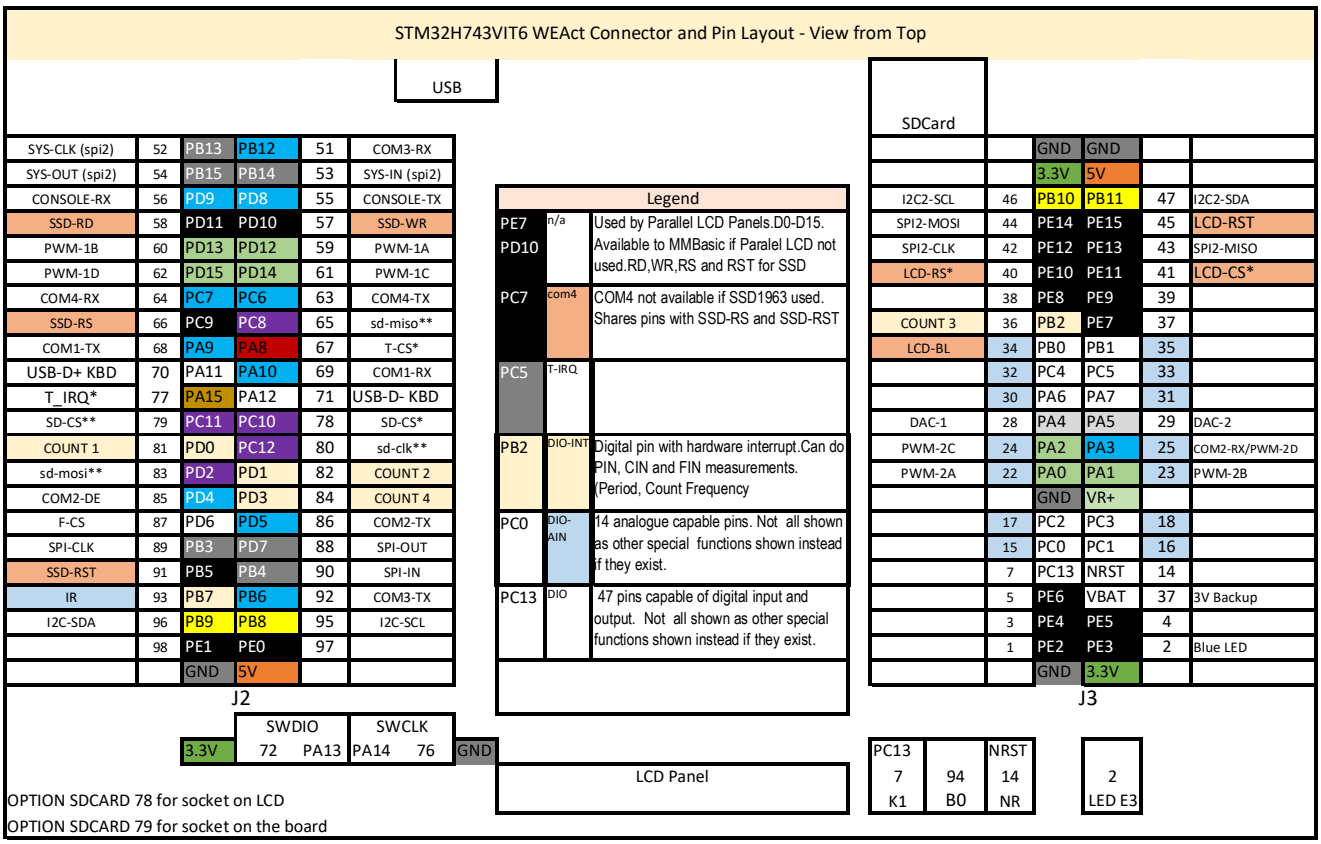
Screen this side.

97	98	1	2	3	4	5	37	52	67	54		53	77	53	52	54	78		
PE0	PE1	PE2	PE3	PE4	PE5	PE6	PE7	PB13	PA8	PB15	n/c	PB14	PA15	PB14	PB13	PB15	PC10	n/c	n/c
1	3	5	7	9	11	13	15	17	19	21	23	25	27	29	31	33	35	37	39
B0	B1	B2	B3	B4	B5	B6	B7	T-CLK	T-CS	MOSI	n/c	MISO	T-IRQ	MISO	SD-CLK	MOSI	SD-CS	n/c	n/c
GND	3.3V	n/c	DC/RS	WR	RD	B8	B9	B10	B11	B12	B13	B14	B15	CS	F-CS	RST	5V*	LED-A	n/c
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
GND	3.3V	n/c	PC7	PD10	PD11	PE8	PE9	PE10	PE11	PE12	PE13	PE14	PE15	GND	n/c	PC6	n/c	PB0	n/c
			64	57	58	38	39	40	41	42	43	44	45			63		34	

40 Pin connector to receive SSD1963 5" screen viewed from the top. Screen is above this connector.

5V* Only used on the larger 7" and above displays for the backlight.

STM32H743VIT6 WeAct Connector and Pin Layout



SYS-CLK (spi2)	52	PB13	PB12	51	COM3-RX
SYS-OUT (spi2)	54	PB15	PB14	53	SYS-IN (spi2)
CONSOLE-RX	56	PD9	PD8	55	CONSOLE-TX
SSD-RD	58	PD11	PD10	57	SSD-WR
PWM-1B	60	PD13	PD12	59	PWM-1A
PWM-1D	62	PD15	PD14	61	PWM-1C
COM4-RX	64	PC7	PC6	63	COM4-TX
SSD-RS	66	PC9	PC8	65	sd-miso**
COM1-TX	68	PA9	PA8	67	T-CS*
USB-D+ KBD	70	PA11	PA10	69	COM1-RX
T_IRQ*	77	PA15	PA12	71	USB-D- KBD
SD-CS**	79	PC11	PC10	78	SD-CS*
COUNT 1	81	PD0	PC12	80	sd-clk**
sd-mosi**	83	PD2	PD1	82	COUNT 2
COM2-DE	85	PD4	PD3	84	COUNT 4
F-CS	87	PD6	PD5	86	COM2-TX
SPI-CLK	89	PB3	PD7	88	SPI-OUT
SSD-RST	91	PB5	PB4	90	SPI-IN
IR	93	PB7	PB6	92	COM3-TX
I2C-SDA	96	PB9	PB8	95	I2C-SCL
	98	PE1	PE0	97	
		GND	5V		

		GND	GND		
		3.3V	5V		
I2C2-SCL	46	PB10	PB11	47	I2C2-SDA
SPI2-MOSI	44	PE14	PE15	45	LCD-RST
SPI2-CLK	42	PE12	PE13	43	SPI2-MISO
LCD-RS*	40	PE10	PE11	41	LCD-CS*
	38	PE8	PE9	39	
COUNT 3	36	PB2	PE7	37	
LCD-BL	34	PB0	PB1	35	
	32	PC4	PC5	33	
	30	PA6	PA7	31	
DAC-1	28	PA4	PA5	29	DAC-2
PWM-2C	24	PA2	PA3	25	COM2-RX/PWM-2D
PWM-2A	22	PA0	PA1	23	PWM-2B
		GND	VR+		
	17	PC2	PC3	18	
	15	PC0	PC1	16	
	7	PC13	NRST	14	
	5	PE6	VBAT	37	3V Backup
	3	PE4	PE5	4	
	1	PE2	PE3	2	Blue LED
		GND	3.3V		

Notes:

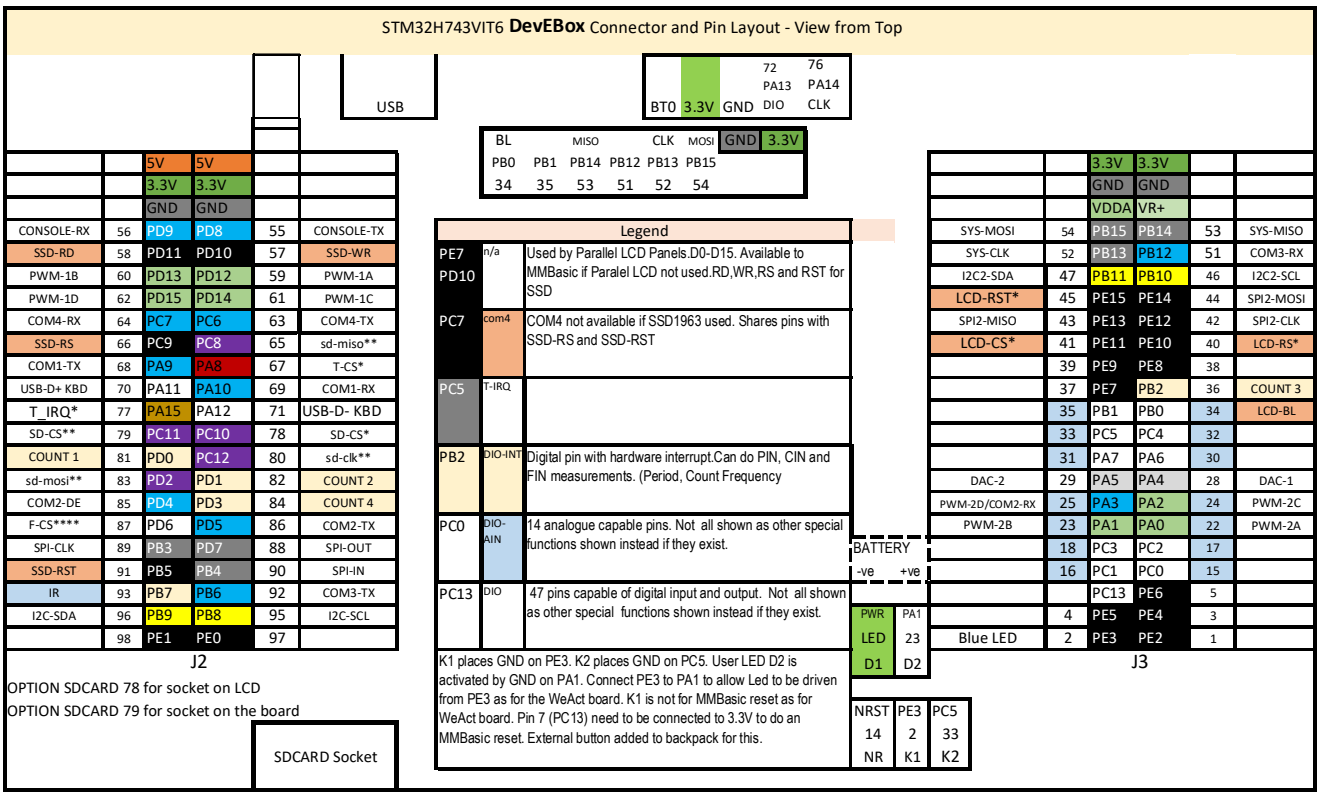
* Nominal but not mandatory allocation of pin to the function

** Refers to pins used if bitbanged onboard SDCard is enabled

*** Not available if parallel LCD Panel configured

OPTION ILI9341, Orientation, D/C, RST, CS
OPTION ILI9341, L, 40, 45, 41
OPTION TOUCH 67,77
OPTION SDCARD 78 (SDCARD on LCD)
OPTION SDCARD 79 (Onboard SDCARD)

STM32H743VIT6 DevEBox Connector and Pin Layout



Notes:

* Nominal but not mandatory allocation of pin to the function

** Refers to pins used if bitbanged onboard SDCard is enabled

*** Not available if parallel LCD Panel configured

OPTION LCDPANEL ILI9341, Orientation, D/C, RST, CS
OPTION LCDPANEL ILI9341, L, 40, 45, 41
OPTION TOUCH 67,77
OPTION SDCARD 78 (SDCARD on LCD)
OPTION SDCARD 79 (Onboard SDCARD)

OPTION LCDPANEL SSD1963_5_16, LANDSCAPE

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 or F1 to save the program. (On TeraTerm right mouse click will open the paste window.) Shortcut key F10 can be used to send the AUTOSAVE command, F1 can be used to sent CNTRL Z and F2 can be used to send CNTRL Z and immediately run the program.

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

Because the Armmite does not necessarily have a normal storage system it needs to save data that can be recovered when power is restored. This can be done with the VAR SAVE command which will save the variables listed on its command line in non-volatile flash memory. The space reserved for saved variables is 128KB.

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 frquency saves as you may wear out the flash memory. The flash used for the Armmite 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.BACKUP will give the start address for this ram memory.

Timing

You can get the current date and time using the DATE$ and TIME$ functions and you can set them by assigning the new date and time to them. The Armmite H7 has a battery backed clock so it will not lose the time even when powered off. If you find that the time drifts while the power is off you can use the OPTION RTC CALIBRATE command to correct for any inaccuracies. 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.

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

Using SETTICK you can setup up to four “ticks” which will generate regular interrupts with a period from one millisecond to over a month.

Watchdog Timer

One of the possible uses for the Armmite H7 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 H7 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 H7 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 H7 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 H7 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 H7 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 STM32H743 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 H7.

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.
The command `LIST COMMANDS` can be used to output a list of the available functions to the screen.

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.
The command `LIST FUNCTIONS` can be used to output a list of the available functions to the screen.

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.

Setting Options

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

```
OPTION BAUDRATE 115200
```

The Serial Console

The default settings for the serial over USB console are 115200 baud, 8 bits, no parity and one stop bit. Using the OPTION BAUDRATE command the baud rate of the serial console can be changed to any other speed. Changing the console baud rate to a higher speed makes the full screen editor faster in redrawing the screen. Once changed the console baud rate will be permanently remembered unless another OPTION BAUDRATE command is used to change it. Using this command it is possible to accidently set the baud rate to an invalid speed and in that case the only recovery is to reset MMBasic as described below.

Resetting MMBasic

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

- Connect pin 7 (PC13) to VDD whilst resetting the chip by pressing the RESET button, or more reliably by removing and then reconnecting power to the device while VDD is connected to pin 7. In some situations the device seems to not respond to the Reset button.

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.

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 ARMMITE H7 boards

Use the EDIT command to enter the following program

WeAct and DevEBox 100 pin	Nucleo 144 pin

`'BLUE LED Pin 2`
`SETPIN 2, DOUT`
`DO`
`  PIN(2) = 1`
`  PAUSE 300`
`  PIN(2) = 0`
`  PAUSE 300`
`LOOP`
	`'Red LED Pin 75`
	`SETPIN 75, DOUT`
	`DO`
	`  PIN(75) = 1`
	`  PAUSE 300`
	`  PIN(75) = 0`
	`  PAUSE 300`
	`LOOP`

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 H7 can interface to the physical world via your programming.

The program itself is simple. The first line sets pin 2 as an output. Then the program enters a continuous loop where the output of that pin is set high to turn on the LED followed by a short pause (300 milliseconds). The output is then set to low followed by another pause. The program then repeats the loop.
If you leave it this way, the Armmite will sit there forever with the LED flashing. If you want to change something (for example, the speed of flashing) you can interrupt the program by typing CTRL-C on the console and then edit it as needed. This is the great benefit of MMBasic, it is very easy to write and change a program.
If you want this program to automatically start running every time power is applied you can use the command:

```
OPTION AUTORUN ON
```

To test this you can remove the power and then re-apply it. The Armmite should start up flashing the LED.

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.

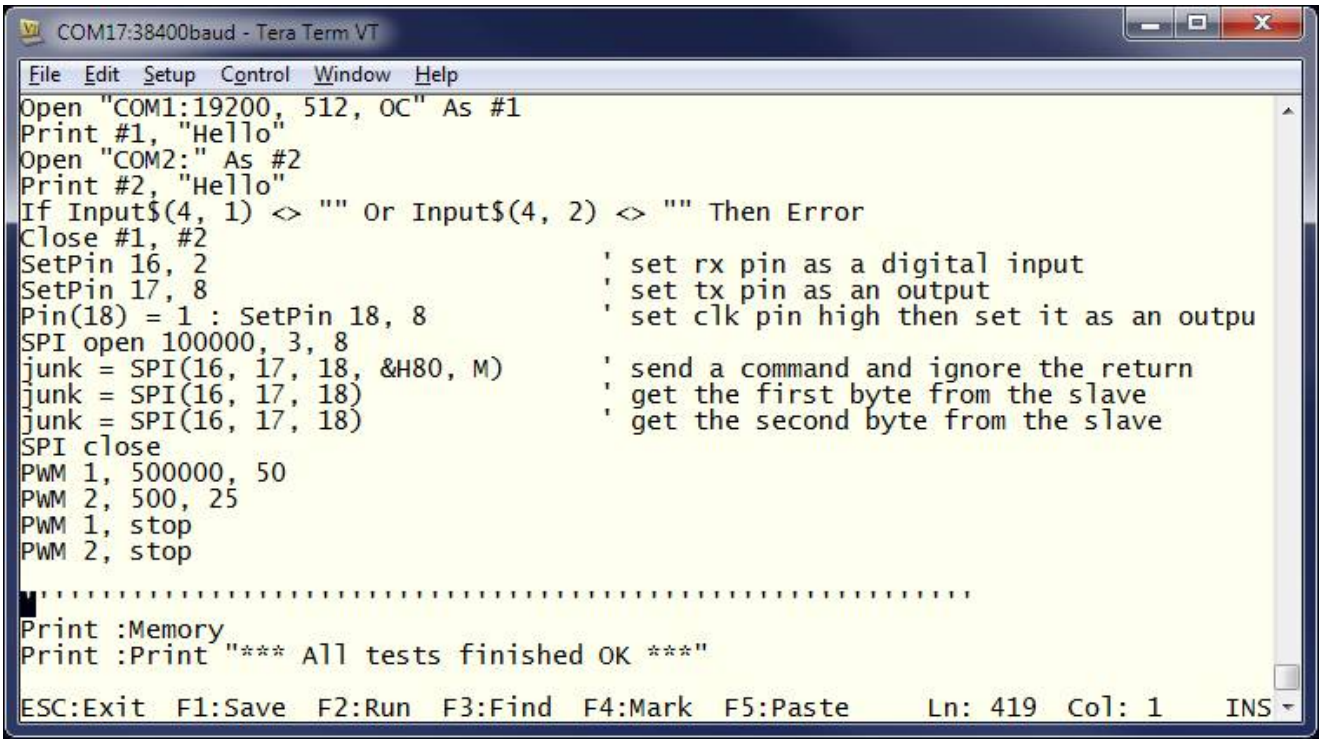
This is a comprehensive tutorial on the language which will take you through the fundamentals in an easy to read format with lots of examples.

The CMM2 is based on the same STM32H743 chip as the ArmmiteH7 (The CMM2 has more pins and outputs to a VGA screen). This tutorial Programming with the Colour Maximite 2 from Geoff Graham's website is an excellent source of information which mostly applies to the ArmmiteH7 as well. It gives good advice on how to use/protect the I/O pins.

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 H7.

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. The editor will only show the number of characters that will fit into the available columns in the display.e.g. 80. This is a limitation of all of the simple MMBasic versions. 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	There is a hack workround in the editor for inserting or editing lines longer than 80 columns. To enter a long line type the line over two rows by hitting return at an appropriate point and continue typing on the next line, then delete the inserted return in the first line to combine the lines. You can't see the characters for the rest of the line, but they are there. To edit a long line, place a return in the line as displayed, the line will be broken at that point, the hidden characters will appear on the next line. Edit as required and then delete the inserted return to combine the lines.

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 H7 but this feature can be disabled/enabled with the commands:

```
OPTION COLOURCODE OFF or 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 Armmite H7 implements a command buffer at the command prompt. The Up and Down arrows can be used to locate previous commands in the buffer. A 1K 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. OVR (Overwrite Mode) is turned on so any character type will overwrite the character at the the cursor position. 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	Send ESC sequents to VT100 to clear the screen. Also clears the LCDPANEL if OPTION LCDPANEL CONSOLE is set.

F10	Sends AUTOSAVE (This clears the program so don't press 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.

~~On the LCDPANEL a different cursor is displayed to show INS or OVR mode. On VT100 the cursor is not changed and there is no indication of the mode.~~

The VT100 emulations provided by TeraTerm, Putty and GFXTerm 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. If your terminal does not support this you should limit your commands to one terminal width. e.g. 80 characters.

Setting the terminal size with OPTION DISPLAY lines[,chars] will also send an ESC sequence to set the VT100 terminal to the matching size. Only TerraTerm is known to respond to this sequence.

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 compatability 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	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	1-255	The byte whose numerical value is given by nnn interpreted as a decimal number \000 is not accepted. Use CHR$(0)
\&hh	01-FF	The byte whose numerical value is given by hh interpreted as a hexadecimal number \&00 is not accepted. Use CHR$(0)

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

Arithmetic operators:

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

Shift operators:

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

Logical operators:

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

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

The other logical operations result in the integer 0 (zero) for false and 1 for true. For example the statement PRINT 4 >= 5 will print the number zero on the output and the expression A = 3 > 2 will store +1 in A. The NOT operator will invert the logical value on its right (it is not a bitwise invert) while the INV operator will perform a bitwise invert. Both of these have the highest precedence so they will bind tightly to the next value. For normal use of NOT or INV the expression to be operated on should be placed in brackets. Eg: IF NOT (A = 3 OR A = 8) THEN …

String operators:

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

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

Mixing Floating Point and Integers

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

The one exception is the normal division ("/") which will always convert both sides of the expression to a floating point number and then 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 H7 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 H7 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 H7.

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 H7 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 but they will default to reading DATA statements in the main program memory. 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.

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 H7)

Tradionally the Micromite Library code is added at the end of the normal program memory. The Armmite H7 has 4 areas of 128K that are available as program memory. The erasing and writing of this 512K block each time a program is edited or loaded can take a noticeable time. By default, only 128K is allocated for program memory which requires only 25% of the time to update. If your program exceeds 128K then the next 128K can be allocated/included as required, this can be repeated until all 512K is used if required. See OPTION FLASHPAGES.

The last 128K can instead be used to hold the library code. This will mean a maximum 384K of normal program space and 128K of library. The issue of the first LIBRARY SAVE command will allocate the 4th 128K to the Libray. The existance of the Library does not affected the update time of the normal program memory, so you get all the advantages of the library without increasing update time. The Library code is only written when a LIBRARY command is issue.

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 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.

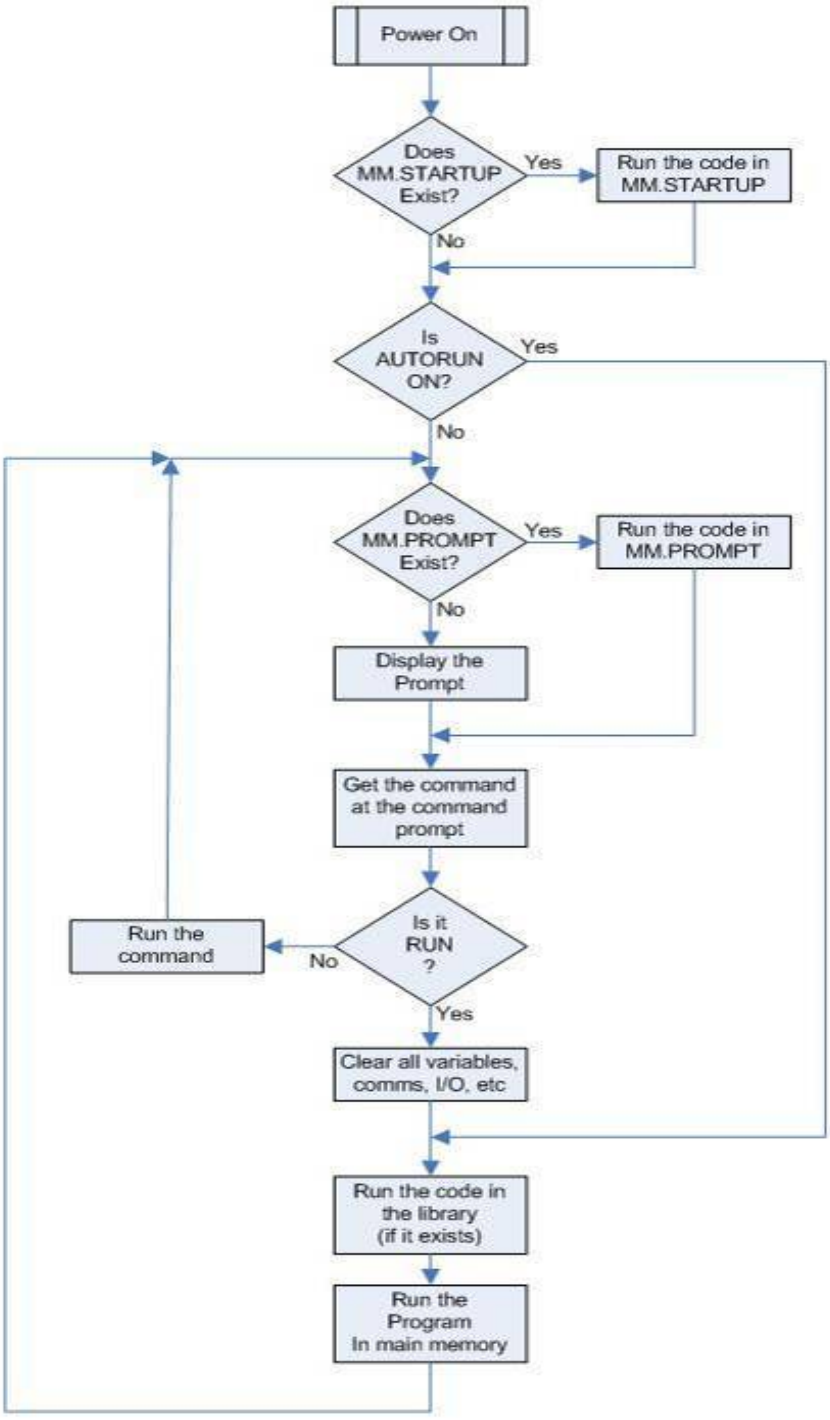
Note that MMBasic will clear all variables and I/O pin settings when a program is run so anything set in this subroutine will only be valid for commands typed at the command prompt (i.e., in immediate mode). As an example the following will display a custom prompt:

```
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. The MM.STARTUP subroutine must be included in the program somewhere, this can be in the library if desired.



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 H7 the output identifies the Saved Variables and Library memory separately from the Program memory. LIBRARY LIST can be used to see details of what is in the library.

> **MEMORY** Program Flash: 13K (10%) Program (196 lines) 1K (0%) 1 Embedded C Routine 5K (1%) 1 Embedded Fonts 109K (89%) Free 256K Unallocated Saved Vars Flash: 76K (59%) 3 Saved Variables (77589 bytes) 52K (41%) Free Library Flash: 1K (1%) Library 127K (99%) Free RAM: 76K (15%) 3 Variables 0K (0%) General 422K (85%) Free	Details of Program Flash. Shows 196 lines of code using 13K. This includes the HEX listing of the CSUB and the FONT. 1K used for the binary copy of the CSUB. 5K used for the binary copy of the FONT 109K free. The 256K unallocated shows that OPTION FLASHPAGES is 1. Pages 2 and 3 are unallocated. 3 variables are saved with VAR SAVE and these use 76K, which is 59% of the 128K available for Saved Variables. The library is using flash page 4 and is 128K. 1K is used. Use LIBRARY LIST to see what is actually in there. Total available RAM is normally 498K. (When OPTION CONTROLS is at the default value of 200). Increasing OPTION CONTROLS or loading buffered drivers for the 800*480 lcdpanels will reduce this. This shows the 3 variables using 76K. The maximum 512K is available setting OPTION CONTROLS 0 and having no LCDPanel with resolution > 480*320 using a buffered driver. 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 1, DIN
IF PIN(1) = 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 15, AIN
> PRINT PIN(15)
 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 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. See Setting Option VCC for more detail.

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 *asciicode*
ON KEY general
COM1: Serial Port Add bypass as per CMM2
COM2: Serial Port
COM3: Serial Port
COM4: Serial Port
GUI Int Down
GUI Int Up
WAV Finished
Collision Interrupt
ADC completion

DAC interrupt
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 and SETPIN CIN,PIN,FIN

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 H7 Deployment Considerations

This section discusses some Armmite H7 deployment considerations. These maybe relevant if you want to use the Armmite H7 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 .
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.

Battery Backed Ram

The ram is supplied via a 3.0V battery and the main VCC supply. When VCC is removed the battery maintains the ram and powers the RTC.

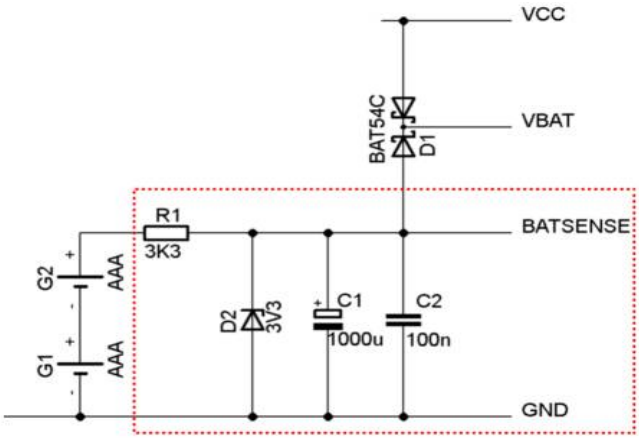
Battery Life and Monitoring VBAT

If the Armmite H7 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.



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:	100 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:	Console is USB only. Default 115200 baud. Range is 2400 bps to 921600 bps
COM ports	Default 9600 baud. Range is 1200 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 25 MHz
I^2C	10kHz to 10000 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 flash and impact the life of the flash, so care must be taken not be continually call these within a program loop.

Sound Output

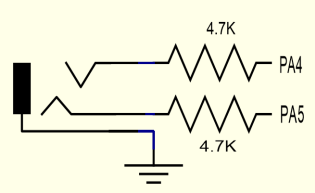
The ArmmiteH7 can play WAV, FLAC and MP3 files from the SD card, generate synthesised music in the MOD format, create robot speech and sound effects as well as generate precise sine wave tones. All these are outputted on the audio socket. The ARM Cortex-M7 chip includes its own DAC (digital to analog converter) so an output filter network is not needed (as on the original Colour Maximite).

The Armmite H7 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

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, MP3 and FLAC Files

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

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

```
      PLAY WAV file$, interrupt
or    PLAY MP3 file$, interrupt
or    PLAY FLAC file$, interrupt
```

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

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).

Background Music

If *fname$* in the PLAY WAV/MP3/FLAC command is a directory then the firmware will list all the files of the relevant type in that directory and start playing them one-by-one. To play files in the current directory use an empty string (ie, ""). Each file listed will play in turn and the optional interrupt will fire when all files have been played. The filenames are stored with full path so you can use CHDIR while tracks are playing without causing problems. All files in the directory are listed if the command is executed at the command prompt but the listing is suppressed in a program.

While playing in this background mode the user can edit programs, run programs, etc without interrupting the playing of the music. Amongst other things this allows the Armmite H7 to be used as a music player while programming or doing other tasks.

Generating Sine Waves

The PLAY TONE command also uses the audio output and will generate sine waves with selectable frequencies for the left and right channels. This feature is intended for generating attention catching sounds

but, because the frequency is very accurate, it can be used for many other applications. For example, signalling DTMF tones down a telephone line or testing the frequency response of loudspeakers.

The syntax of the command is:

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

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

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

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

Specialised Audio Output

There are a number of specialised audio commands that are mostly used in computer games.

These are:

- PLAY MODFILE which will play synthesised music using the MOD format.
- PLAY TTS command which will generate robotic speech.
- PLAY SOUND which will generate an output based on a mixture of sine, square, noise, etc waveforms.
- PLAY EFFECT command which will play a WAV at the same time as a MOD file is playing.

Using PLAY

It is important to realise that the PLAY command will generate the audio in the background. This allows a program (for example) to play the sound of an explosion while still animating the visual of the explosion on the screen. Without the background facility the whole computer would freeze while the sound was heard.

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

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

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

Similarly the following program will not work either:

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

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

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

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

This applies to all versions of the PLAY command (eg, PLAY WAV/MP3/FLAC, etc).

Utility Commands

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

`PLAY PAUSE`	Temporarily halt (pause) the currently playing file or tone.
`PLAY RESUME`	Resume playing a file or tone that was previously paused.

`PLAY STOP`	Terminate the playing of the file or tone. The sound output will also be automatically stopped when the program ends.
`PLAY VOLUME L, R`	Set the volume to between 0 and 100 with 100 being the maximum volume. The volume will reset to the maximum level when a program is run.

The following commands can be used when playing a sequence of files (ie, "background music"):

`PLAY NEXT`	Skip to the next file.
`PLAY PREVIOUS`	Skip to the previous file.

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.

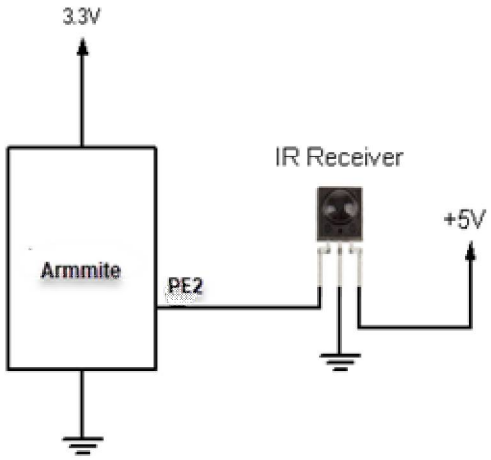
These are:

- 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 14 the Armmite H7 144 pin and pin 93 on Armmite H7 100 pin) 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.

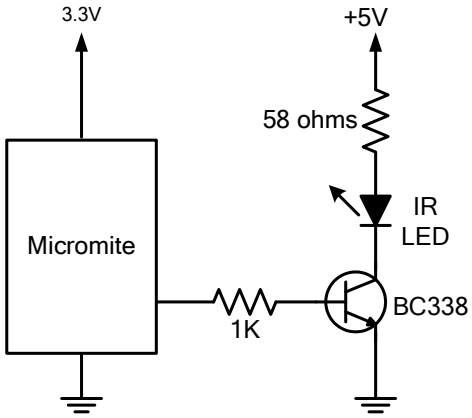
This is an example of using the IR decoder:

```
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 H7



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:

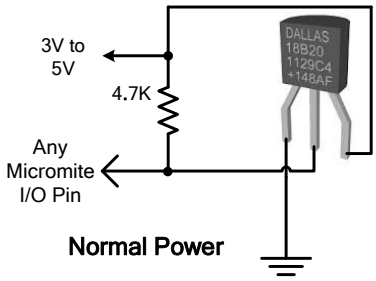
```
IRSEND pin, dev, key
```

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.

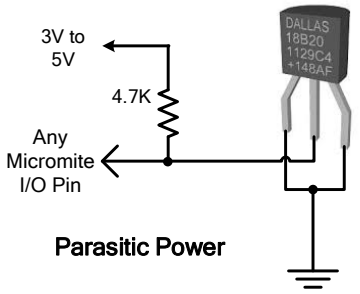


The DS18B20 can be powered separately by a 3V to 5V supply or it can operate on parasitic power from the Armmite/Micromite as shown on the right. Multiple sensors can be used but a separate I/O pin and a 4.7K pullup resistor is required for each one.

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.

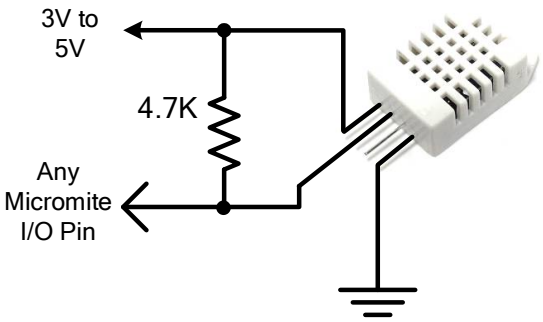
For example:

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

Measuring Humidity and Temperature

The BITBANG 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 BITBANG HUMID command with arguments as follows:



BITBANG 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).

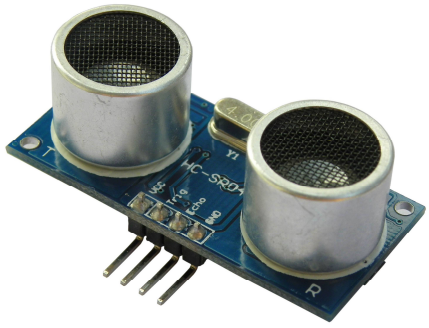
For example:

```
DIM FLOAT temp, humidity
BITBANG 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.

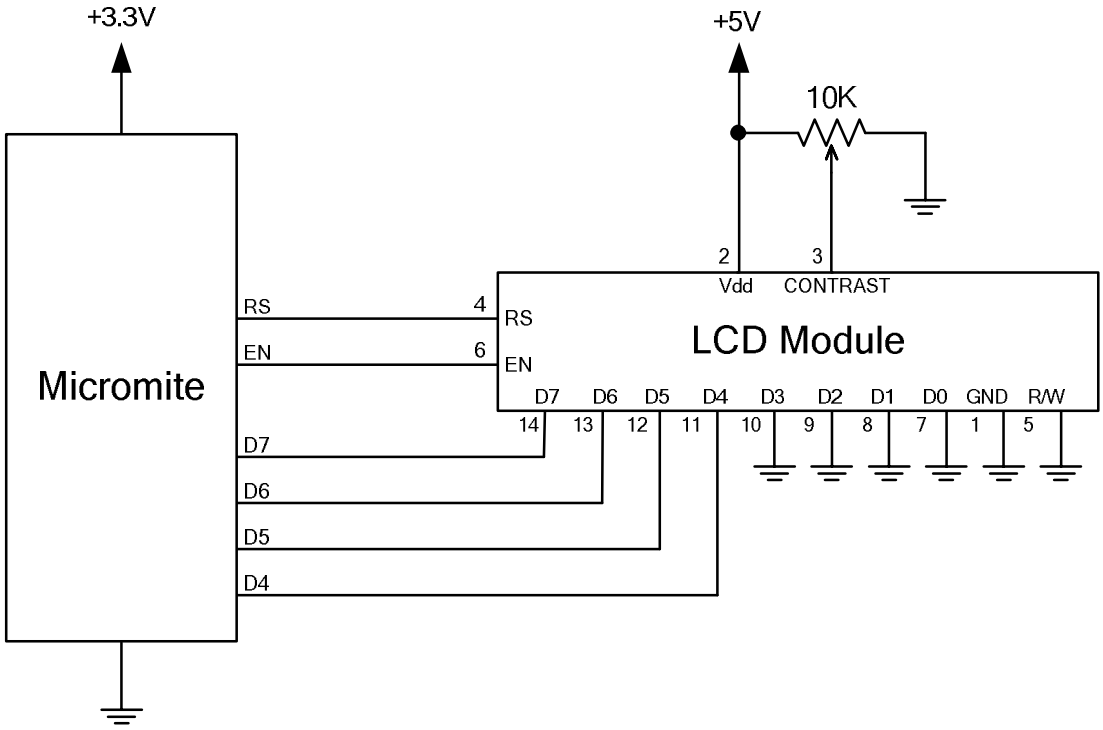
This command will work with LCD modules that use the KS0066, HD44780 or SPLC780 controller chip and have 1, 2 or 4 lines. Typical displays include the LCD16X2 (futurlec.com), the Z7001 (altronics.com.au) and the QP5512 (jaycar.com.au). eBay is another good source where prices can range from $10 to $50.

To setup the display you use the LCD INIT command:

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

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

Any I/O pins 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 Micromote, rs is connected to pin 23 and en to pin 24..

```
BITBANG LCD INIT 2, 3, 4, 5, 23, 24
BITBANG LCD 1, 2, "Temperature"
BITBANG 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.

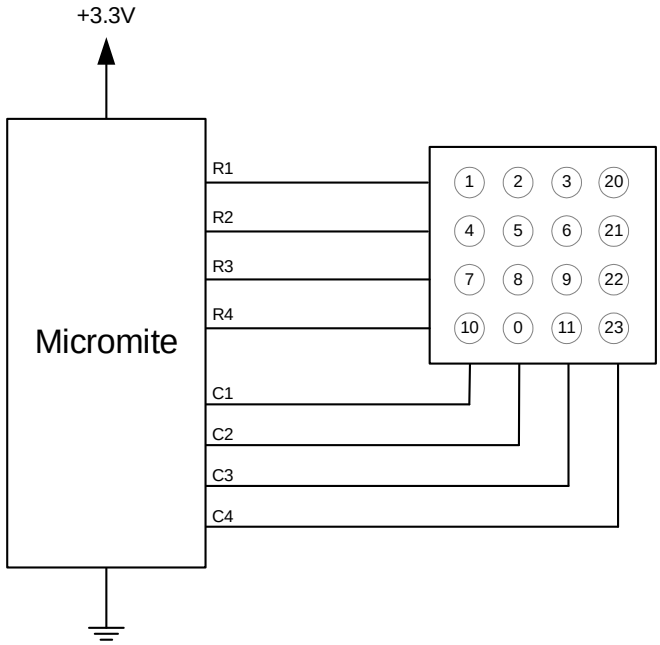
Examples of a 4x3 keypad and a 4x4 keypad are the Altronics S5381 and S5383 (go to www.altronics.com).

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 H7 has built in support for the WS2812 multicolour LED chip. This chip needs a very specific timing to work properly and with the BITBANG WS2812 command it is easy to control these devices with minimal effort.

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

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

Note that the pin must be set to a digital output before this command is used.

The colours%() array should be sized to have exactly the same number of elements as the number (nbr) of LEDs to be driven. Each element in the array should contain the colour in the normal RGB888 format (0 - &HFFFFFF). The limit of the size of the WS2812 string supported is 1-512. If only one LED is connected then a single integer should be used for colours% (ie, not an array).

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

O = original WS2812

B = WS2812B

S = SK6812

W=SK6812 RGBW

As an example:

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

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

SD Card Support

The Armmite H7 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 H7 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 H7 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
```

LCD Display Panels

The Armmite H7 supports a number of parallel displays as well as a number of SPI displays. There are a number of different drivers for the SSD1963 displays that support different requirements. The ARMmite H7 has sufficient RAM available to allow the use of buffered drivers in some cases.

Buffered Drivers for all displays

All displays up to and including 480x320 pixel resolution automatically use a buffered driver where a memory image of the display is maintained in the Armmite's memory. This does not impact user RAM space but helps reduce artefacts whilst writing to the screen. Commands ***OPTION AUTOREFRESH*** and ***REFRESH*** can be used to control when the updates to the screen take place allowing the programmer maximum flexibility in using the screen effectively. An 8-bit buffered driver (64 colours) is included for 800x480 SSD1963 displays. This is specifically targeted at games programming as user RAM is still 473K with the driver loaded. A RGB565 buffered driver is also included for 800x480 displays but this reduces user RAM significantly (down to 98Kbytes free).

SSD1963 Based LCD Displays 16-bit Interface @ RGB565

The Armmite H7 can drive a SSD1963 display using a 16-bit parallel bus for extra speed. The extra I/O pins for this are listed as SSD1963-DB8 to SSD1963-DB15 on the pinout tables in this manual and they must be connected to the pins labelled DB8 to DB15 on the I/O connector on the SSD1963 display.

Note that in this mode the SSD1963 controller runs with a reduce colour range (65 thousand colours RGB565) compared to 16 million colours with the normal 8-bit interface.

SSD1963 Based LCD Displays 8-bit Interface @ RGB888

The Armmite H7 can also drive a SSD1963 display using an 8-bit parallel bus. (The extra I/O pins listed as SSD1963-DB8 to SSD1963-DB15 on the pinout tables in this manual do not carry any data, however on the 144 pin Nucleo boards they are reserved and not available for other use when the 8 bit driver is used, as the driver is faster when it writes the whole 16 bits of the register.)

On the 100 pin WeAct and DevEBox boards only SSD1963-DB0 to SSD1963-DB7 are used and SSD1963-DB8 to SSD1963-DB15 pins are available for use as general I/O and to allow SPI2.

Note that in this mode the SSD1963 controller runs with 16 million colours. i.e RGB888.

Other 16 Bit Parallel Interface LCD Panels

The Armmite H7 supports a number of LCD Panels with 16 bit parallel interface.

The supported panels are:

- ILI9341 P16
- SSD1963 4" 5" 7" 8" 9"

These are high quality, have been available long term and need an adaptor board. The SSD1963 is the only display with hardware scrolling when in landscape orientation so is the best choice if you intend to use the LCD Panel as the Console. All other LCDs are much slower to scroll as they need to do it in software by reading and writing to the display.

- 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.

SSD1963 Power Considerations

For 4.3", 5", 7" and 8" 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.
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.
For 9" displays using the Ritech adapter 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%.

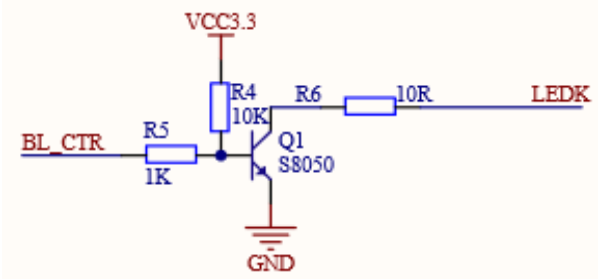
The brightness of the backlight on LCD panels can be controlled with the BACKLIGHT command:

```
BACKLIGHT percent [,S|,R]
```

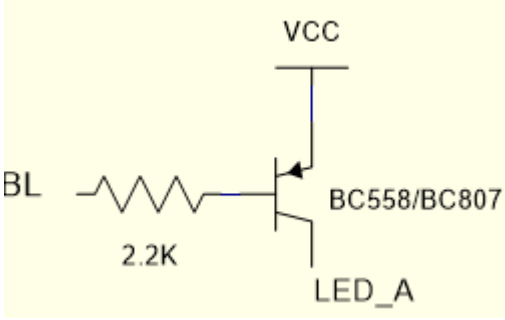
Where 'percent' is the degree of brightness ranging from 0 (fully off) to 100 (full brightness). This can be changed as often as required and makes a huge difference to the power requirements of the display. For example, a brightness of 50% will halve the current consumption (compared to 100%) while 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 H7 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.

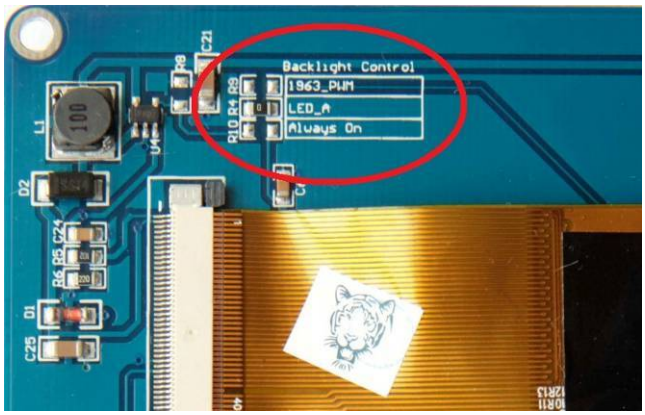


Typical LED driver built into displays gives brightness as 0(off)-100(fully on).



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 H7 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”)

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 System SPI 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		
T_DO	Touch Data Out (MISO)		
T_DIN	Touch Data In (MOSI)		
T_CS	Touch Chip Select		
T_CLK	Touch SPI Clock		
SDO (MISO)	Display Data Out (MISO)		
LED	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 you ***cannot connect to the 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. The LCD backlight can be connected to VCC via a suitable resistor to give a satisfactory backlight.		
SCK	Display SPI Clock		
SDI (MOSI)	Display Data In (MOSI)		
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 63mA. 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

- ILI9341 SPI based 320*240 2.2", 2.4" and 2.8" panels using the ILI9341 controller
- ILI9481 SPI based 480*320 SPI touch controller

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.

This TBS post details some other methods for allowing the ILI9488 to operate reliably, ***however note that the ILI9488 on the Armmite H7 uses a buffered driver so MISO can just be left disconnected without any loss of BLIT or transparent text.***

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. The syntax is:

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

Where:

'controller' can be either ILI9341, ILI9481, ILI9488'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, ?, ?, ?
```

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 H7.

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.

http://fruitoftheshed.com/MMBasic.LCD%20Panel%20list.ashx

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 H7(see the above chapter for the details) and then configured (see below).

When Touch is enabled it uses the SYSTEM SPI which is separate to the SPIs available to MMBasic.. The touch chip on the LCD needs to be queried using SPI to find out where the touch occurred. T_IRQ will only indicate a touch has occured.

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.

The syntax is:

```
OPTION TOUCH T_CS, T_IRQ[,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 returns 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.

USB Keyboard and LCDPANEL as Console

The ArmiteH7 can be used as a stand alone computer if a USB keyboard is attached and the LCDPANEL is used as the main console.

LCD Display as the Console Output

A USB 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 USB keyboard this option turns the Armmite H7 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 as 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 set 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
```

Connecting USB Keyboard

A keyboard can be plugged directly into the micro-USB on the DevEBox 100 pin board and also directly into the USB-C port on the WeAct 100 pin board using suitable adapters.

On the Nucleo-H743ZI 144 pins boards the keyboard is plugged into the micro-USB connector on the side opposite the ST-Link. On the Nucleo-H743ZI a pin is nominated to enable the USB power supply to the keyboard. On the 100 pin boards 5V is hardwired to the USB and nominating a pin is not required.

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

```
OPTION USBKEYBOARD language [,powerpinno [,noLED]]
```

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 H7 fonts (another character will be used instead).

This command configures 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 USBKEYBOARD DISABLE command.

See Option USBKEYBOARD for details of the setup.

Graphic Commands and Functions

There are ten basic drawing commands that you can use within MMBasic to draw images on the LCDPANEL

Screen Coordinates

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

Read Only Variables

In the Armmite H7 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

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

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

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

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

The drawing commands are:

- CLS *C*
  Clears the screen to the colour C. If C is not specified, the current default background colour will be used.
- PIXEL X, Y, *C*
  Illuminates a pixel. If C is not specified, the current default foreground colour will be used.
- LINE X1, Y1, X2, Y2, *LW, C*
  Draws a line starting at X1 and Y1 and ending at X2 and Y2.
  LW is the line's width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or is changed to 1 if the line is a diagonal.
- BOX X, 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

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 H7 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 drivers will generally translate colours to RGB565 format. i.e. 65K colours when they are output to the LCD panel. See below for details, however some specialised drivers and LCD panels may use other colours, e.g. RGB888, RGB666, RGB222

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
```

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 various LCD displays drivers may send RGB565 to the display. In this case 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 can be 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.

Fonts

The Armmite H7 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	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.

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 H7 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

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

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 H7 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).

Example

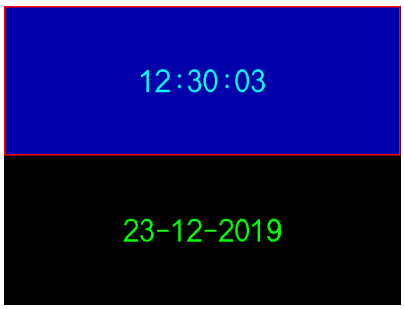
As an example, the following program will draw a simple digital clock on the LCD.

```
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:

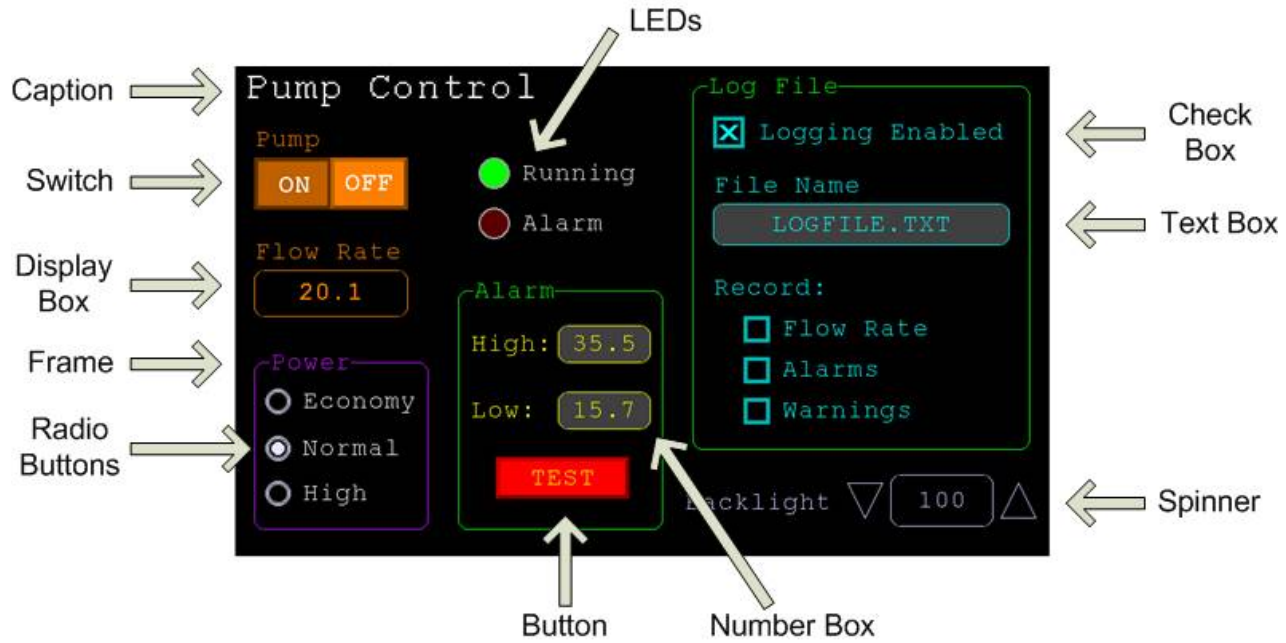
- Displays the current time inside the previously drawn box. The string is drawn centred both horizontally and vertically in the middle of the box. Note that the TEXT command overrides both the default font and colours to set its own parameters.
- Draws the date centred in the lower half of the screen. 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 H7 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.

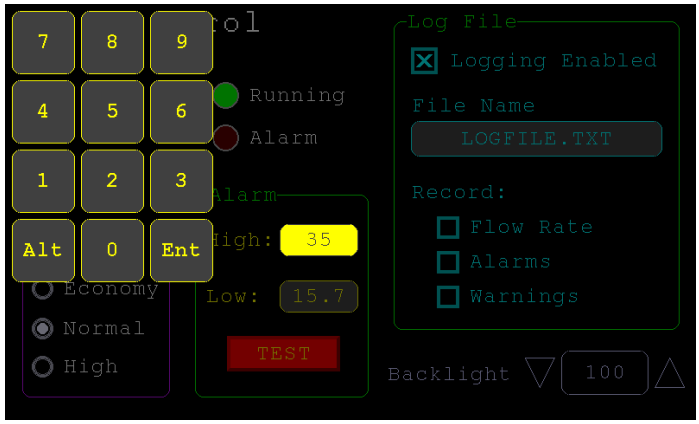
MMBasic will try to position the virtual keyboard on the screen so as to not obscure the text box that caused it to appear. A pen down interrupt will be generated when the keyboard is deployed and a key up interrupt will be generated when the Enter or Cancel keys are touched and the keyboard is hidden.

If necessary, the virtual keyboard can be forced to appear without the control being touched with the command GUI TEXTBOX ACTIVATE and it can be dismissed by the program (same as touching the cancel button) with the command: GUI TEXTBOX CANCEL.

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 forced to appear without the control being touched with the command GUI NUMBERBOX ACTIVATE and it can be dismissed by the program (same as touching the cancel button) with the command: GUI NUMBERBOX CANCEL.

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.

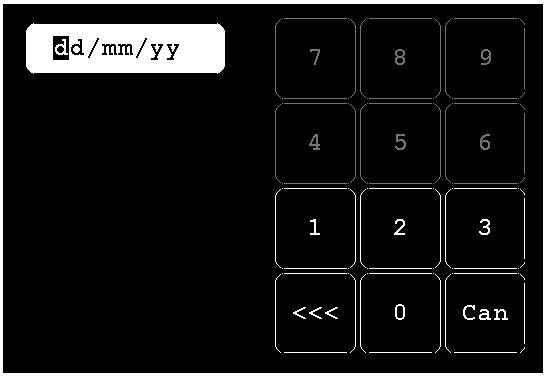
The type of entry is controlled by the 'Format' argument as follows:

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 forced to appear without the control being touched with the command GUI FORMATBOX ACTIVATE and it can be dismissed by the program (same as touching the cancel button) with the command: GUI FORMATBOX CANCEL.

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.

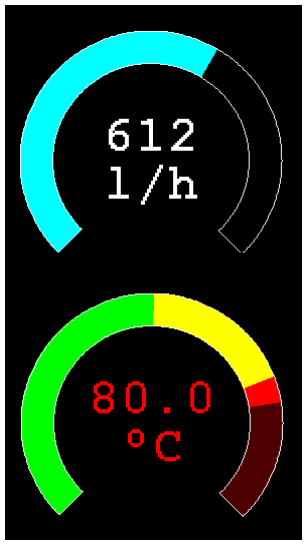
If the colours are not specified this control will use the colours set by the COLOUR command.

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.

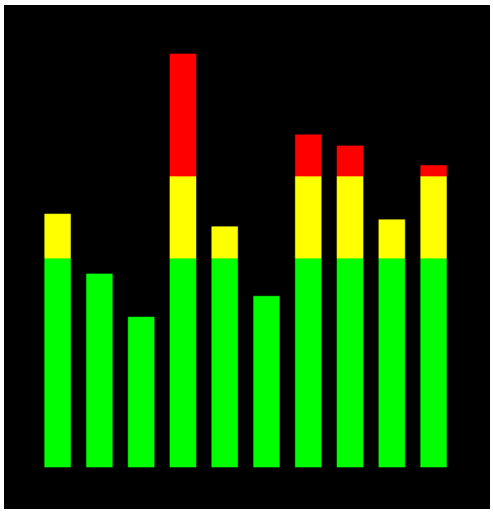
Specifically, 'c1' is the colour to be used for values up to 'ta'. 'c2' is the colour to be used for values between 'ta' and 'tb', 'c3' is used for values between 'tb' and 'tc' and c4 is used for values above 'tc'. Colours and thresholds not required can be left off then list. For example, for a two 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.

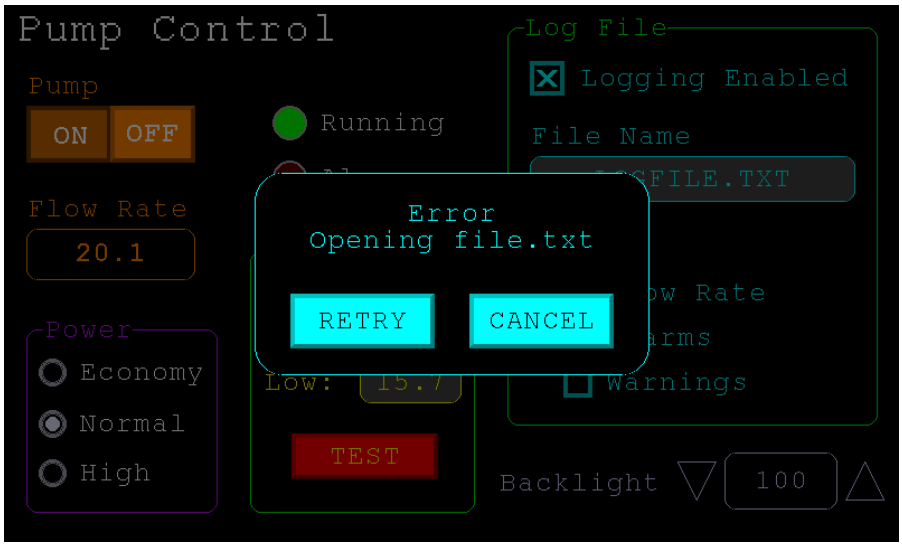
The font used will be the default font set using the FONT command and the colours used will be the defaults set by the COLOUR command. The box will be automatically sized taking into account the dimensions of the default font, the number of lines to display and the number of buttons specified.

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 ARMmite H7 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()

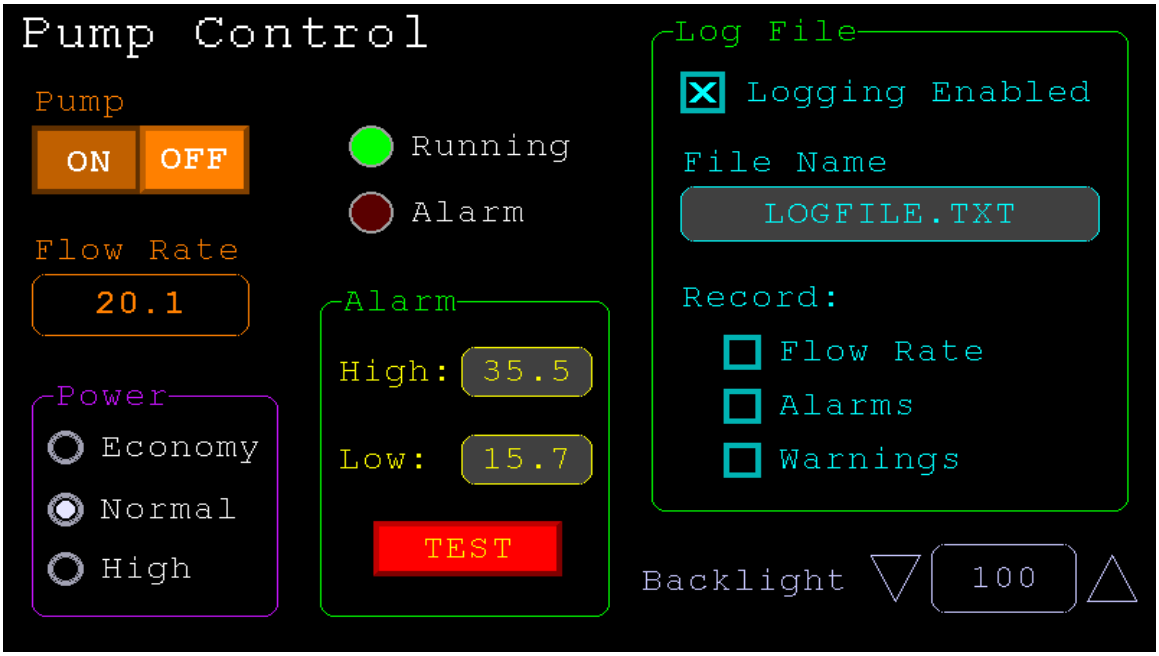
In some cases, you may see a tearing effect or flicker when updating an LCD display. When you write to the LCD you are just updating its RAM. The LCD Panel is taking that data in RAM and updating the actual screen at its own (pretty fast) pace. If you change the data while it is refreshing that part 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 Armmites
' 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 H7 has built in support for up to four serial interfaces. COM1, COM2, COM3 and COM4 are available as standard.

All serial ports on the Armmite H7 can operate at high speed (up to 1.8M baud) and support the OC and S2 options. The DE pin for RS485 is supported on COM2 only.

SPI Interface

The Armite H7 has built in support for three SPI interfaces. The commands to control these interfaces use the identifier SPI for the first SPI port , SPI2 and SPI3 for the other ports. All commands and functions that can be used on the first port (SPI) can be used on the other ports by using the identifier SPI2,or SPI3. e.g:

- SPI2 OPEN
- SPI2 WRITE
- SPI2 READ
- = SPI2(args, …)
- SPI2 CLOSE

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 H7and 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 armmiteH7 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 H7.

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.

http://fruitoftheshed.com/MMBasic.Armmite-H7-User-Manual-and-Firmware-Links.ashx

This page provides a useful summary of all the LCD Panels that can be used with the various Micromites and Armmites. http://fruitoftheshed.com/MMBasic.LCD Panel list.ashx

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 H7. 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

MMBasic Characteristics

Implementation Characteristics

Maximum program size (as plain text) is 512KB. Note that MMBasic tokenises the program when it is stored in flash so the final size in flash might vary from the plain text size.
Maximum length of a command line is 255 characters.
Maximum length of a variable name or a label is 32 characters.
Maximum number of variables 1024: 512 global and 512 local (Constants count as globals)
Maximum number of dimensions to an array is 4.
Maximum number of arguments to commands that accept a variable number of arguments is 50.
Maximum number of nested FOR…NEXT loops is 100.
Maximum number of nested DO…LOOP commands is 100.
Maximum number of nested GOSUBs, subroutines and functions (combined) is 50.
Maximum number of nested multiline IF…ELSE…ENDIF commands is 20.
Maximum number of user defined subroutines and functions (combined): 500
Maximum number of CFunction/CSubs is 20
Maximum number of interrupt pins that can be configured: 10
The range of floating point numbers is 1.797693134862316e+308 to 2.225073858507201e-308
The range of 64-bit integers (whole numbers) that can be manipulated is ± 9223372036854775807.
Maximum string length is 255 characters.
Maximum line number is 65000.
Maximum number of background pulses launched by the PULSE command is 5.
Maximum number of sprites is 64 (#1 to #64)
Maximum number of sprite collisions is 8
Maximum length of a line in a program is 240 characters
Maximum error message size is 64 before it truncates
Maximum number of nested READ SAVE pointers is 16
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 STM32H743 Implementation

Below is a summary of the details of how the MMBasic firmware makes use of the available resources on the STM32H743 chip. This detail is not really need to use MMBasic but may be of interest if you want to dig deeper. The summary is derived from the STM32H743 Reference Manual (RM04033) and the MMBasic source code.

All the Ram, Flash and peripherals of the STM32H743 are mapped to a 32 bit address space. The table below lists some of the relevant items.

Address Range (hex)	Type	Size	Usage/Detail
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.
3880 0000 – 3880 0FFF	Battery Backed Ram	4K	This Ram is battery backed up if a battery is connected to VBAT. MM.BACKUP will return its starting address of &H38800000 to MMBasic. You can use PEEK and POKE to use this Ram.
3800 0000 – 3800 FFFF	Ram	64K	Use by Buffered LCDPanel Drivers.
3000 0000 – 3004 7FFF	Ram	288K	Use by Buffered LCDPanel Drivers.
2400 0000 - 2407 FFFF	Ram	512K	This is MMBasic's memory. This is where a MMBasic program stores its variables and where it gets any memory it needs. Nominally 498K is available, but the whole 512K 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
2000 0000 - 2001 FFFF	Ram	128K	Fast DTCM RAM used by MMBasic Firmware for its own variables, C stack.
081E 0000 – 081F FFFF	Flash	128K	Not used
081C 0000 – 081D FFFF	Flash	128K	Not used
081A 0000 – 081B FFFF	Flash	128K	Save Varaiables. MMBasic VAR SAVE, VAR RESTORE, VAR CLEAR commands control this area. Usage reported by the MMBasic Memory Command
0818 0000 – 0819 FFFF	Flash	128K	Saved Options, Permanent options are stored here. See Option Settings
0816 0000 - 0817 FFFF	Flash	128K	Used to store the MMBasic program. Page 4. Alteratively can be used to store Library code. Usage reported by the MMBasic Memory Command
0814 0000 - 0815 FFFF	Flash	128K	Used to store the MMBasic program. Page 3 Usage reported by the MMBasic Memory Command
0812 0000 - 0813 FFFF	Flash	128K	Used to store the MMBasic program. Page 2 Usage reported by the MMBasic Memory Command
0810 0000 - 0811 FFFF	Flash	128K	Used to store the MMBasic program. Page 1 Usage reported by the MMBasic Memory Command
0800 0000 – 080F FFFF	Flash	1024K	1 M of the total 2M flash is used to store the MMBasic Firmware and its constant data.
0000 0000 - 0000 FFFF	Ram	64K	The MMBasic variable table is located in this RAM. It is possible to use PEEK and POKE to manipulate the variable table.

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 H7.

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. If the Options are see as not yet set, then the default Options are loaded. This would be the case if the backup memory has never been used, the backup battery is flat or removed. 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. MMbasic will restart to the command prompt.
MMBasic Reset	This will clear any save variables, set the default Options and clear the program memory. This is achieved by initiating a restart with BLUE Key 1 held down or PC13 connected to VCC.
Firmware Upgrade	Loading new firmware will not clear the stored varaibles or the program memory or change the Options. Connecting BT0 to 3.3v and restarting will put the Armmite in the bootloader mode, ready to accept new firmware.

Operators and Precedence

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

Numeric Operators (Float or Integer)

NOT INV	NOT will invert the logical value on the right. INV will perform a bitwise inversion of the value on the right. i.e. it converts each 1-bit to an 0-bit and vice versa. Both of these have the highest precedence so if the value being operated on is an expression it should be surrounded by brackets. For example, `IF NOT (A = 3 OR A = 8) THEN …`
^	Exponentiation (eg, `b^n` means b^n)
* / \ MOD	Multiplication, division, integer division and modulus (remainder)
+ -	Addition and subtraction
x << y x >> y	These operate in a special way. << means that the value returned will be the value of x shifted by y bits to the left while >> means the same only right shifted. They are integer functions and any bits shifted off are discarded. For a right shift any bits introduced are set to the value of the top bit (bit 63). For a left shift any bits introduced are set to zero.
<> < > <= =< >= =>	Inequality, less than, greater than, less than or equal to, less than or equal to (alternative version), greater than or equal to, greater than or equal to (alternative version)
=	Equality (also used in assignment to a variable, eg implied LET).
AND OR XOR	Conjunction, disjunction, exclusive or. These are bitwise operators and can be used on 64-bit unsigned integers.

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

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

String Operators

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

String comparisons respect the case of the characters (ie "A" is greater than "a").

Predefined Read Only Variables

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

MM.CMDLINE$** **MM.CMDLINE$	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$** MM.DEVICE$	A string representing the device or platform that MMBasic is running on. Currently this variable will contain one of the following: "Maximite" on the standard Maximite and compatibles. "Colour Maximite" on the Colour Maximite and UBW32. "Colour Maximite 2" on the Colour Maximite 2. "DuinoMite" when running on one of the DuinoMite family. "DOS" when running on Windows in a DOS box. "Generic PIC32" for the generic version of MMBasic on a PIC32. "Micromite" on the PIC32MX150/250 "Micromite MkII" on the PIC32MX170/270 "Micromite Plus" on the PIC32MX470 "Micromite Extreme" on the PIC32MZ series "ARMmite H7" on the ArmmiteH7 "ARMmite F407" on the ArmmiteF4 "ARMmite L4" with chip no. and pin count appended on the ArmmiteL4 “PicoMite” on the Raspberry Pi Pico "PicoMiteVGA" on the Raspberry Pico VGA Edition "WebMite" on the Raspberry Pico WebMite Edition (PicomiteWEB) “MMBasic for Windows” on the windows version *Allowed syntax but saved as MM.INFO(DEVICE)*
MM.FONTHEIGHT **MM.FONTWIDTH** 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.HRES **MM.VRES** MM.HRES MM.VRES	Integers representing the horizontal and vertical resolution of the LCD display panel (if configured) in pixels.
MM.I2C **MM.I2C**	Following an I^2C write or read command this integer variable will be set to indicate the result of the operation as follows: 0 = The command completed without error. 1 = Received a NACK response 2 = Command timed out
MM.INFO$ **MM.INFO**	These two versions can be used interchangeably but good programming practice would require that you use the one corresponding to the returned datatype.

MM.INFO$(AUTORUN MM.INFO$(AUTORUN)	Returns “On” or “Off” depending on the status of OPTION AUTORUN
MM.INFO(BACKUP MM.INFO(BACKUP)	Returns the start address of the 4K of battery backed RAM to avoid remembering it i.e. &H38800000.Use PEEK and POKE to read and write data, remember peek and poke now 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$(CALL TABLE MM.INFO$(CALLTABLE)	Returns the address of the CallTable used by CFunctions
MM.INFO$(CONSOLE MM.INFO (CONSOLE)	Returns “NOCONSOLE” if OPTION LCDPANEL NOCONSOLE is set. Returns “CONSOLE” if OPTION LCDPANEL CONSOLE is set. Can be used in a program to determine if the the LCDPanel is being used as the console.
MM.INFO$(CPUREVID MM.INFO (CPUREVID)	Returns the CPU REVID speed as string. 1003 is returned for the 400MHz chip 2003 is returned for the 480MHz chip.
MM.INFO$(CPUSPEED MM.INFO$(CPUSPEED)	Returns the CPU speed as a string
MM.INFO$(DEVICE MM.INFO$(DEVICE)	Returns a string representing the device or platform that MMBasic is running on. See MM.DEVICE$ above
MM.INFO$(EXISTS DIR MM.INFO$(EXISTS DIR dir$)	Returns a Boolean indicating whether the directory specified exits
MM.INFO$(EXISTS FILE MM.INFO$(EXISTS FILE file$)	Returns 1 if the file specified exists, returns -2 if fname$ is a directory, otherwise returns 0
MM.INFO(DISK SIZE MM.INFO$(DISK SIZE)	NOT IMPLEMENTED ~~Returns the capacity of the SD card in bytes~~
MM.INFO(ERRNO MM.INFO(ERRNO) **MM.INFO$(ERRMSG** MM.INFO$(ERRMSG	If a statement caused an error which was ignored these variables will be set accordingly. MM.ERRNO is a number where non zero means that there was an error and MM.ERRMSG$ is a string representing the error message that would have normally been displayed on the console. They are reset to zero and an empty string by RUN, ON ERROR IGNORE or ON ERROR SKIP.
MM.INFO(FILE SIZE MM.INFO$(FILESIZE file$)	Returns the size of ‘file$’ in bytes. Returns -1 if the file is not found. Returns -2 if file$ is the name of a valid directory
MM.INFO$(FONT ADDRESS MM.INFO$(FONT ADDRESS n)	Returns the actual address of the memory location containing the start of the binary data defining FONT n
MM.INFO$(FONT POINTER MM.INFO$(FONT POINTER n)	Returns a POINTER to the start of the FONT n in memeory
MM.INFO$(FONTHEIGHT MM.INFO$(FONTHEIGHT) **MM.INFO$(FONTWIDTH** MM.INFO$(FONTWIDTH)	Integers representing the height and width of the current font (in pixels).
MM.INFO(FREE SPACE MM.INFO$(FREE SPACE)	NOT IMPLEMENTED. ~~Returns the number of free bytes on the SD card~~

MM.INFO(WIDTH MM.INFO$(WIDTH) **MM.INFO(HEIGHT** MM.INFO$(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$(HPOS) **MM.INFO(VPOS** MM.INFO$(VPOS)	The current horizontal and vertical position (in pixels) following the last graphics or print command.
MM.INFO$(KEYBOARD MM.INFO$(KEYBOARD)	Returns the string CONNECTED if a USB keyboard is connected and working. Otherwise returns “NOT CONNECTED”
MM.INFO$(LCDPANEL) MM.INFO$(LCDPANEL)	Returns the name of the LCD panel configured or a blank string
MM.INFO$(LINE MM.INFO$(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.
MM.INFO$(MODIFIED MM.INFO$(MODIFIED file$)	Returns the date/time that ‘file$’ was last modified. File$ can be a normal file or the name of a directory. Returns an empty string if the file or directory is not found.
MM.INFO (NBRPINS MM.INFO(NBRPINS)	Returns the number of pins. 144 for the Nucleo board abd 100 for the WeAct and DevEBox boards.
MM.INFO(OPTION MM.INFO$(OPTION option)	Returns the current value of a range of options that affect how a program will run. “option” can be one of AUTORUN, BASE, BREAK, DEFAULT, EXPLICIT
MM.INFO(ONEWIRE MM.INFO(ONEWIRE)	Following a 1-Wire reset function this integer variable will be set to indicate the result of the operation as follows: 0 = Device not found. 1 = Device found
MM.INFO(PIN MM.INFO$(PIN pinno)	Returns the status of I/O pin “pinno”. Valid returns are: “Invalid”, “Reserved”, “In Use”, and “Unused”
MM.INFO(PROGRAM MM.INFO(PROGRAM)	Returns the address in memory of the start of the program
MM.INFO(SDCARD MM.INFO$(SDCARD)	Returns status of SDCARD.Valid results are: “Not Present” if no SD Card , “Ready” if SD Card is inserted and “Disabled” of OPTION SDCARD has not been configured.
MM.INFO$(SOUND MM.INFO$(SOUND)	Returns the status of the sound output device. Valid returns are: OFF, PAUSED, TONE, WAV, MP3, MODFILE, TTS, FLAC, DAC, SOUND
MM.INFO$(TOUCH MM.INFO$(TOUCH)	Returns the status of the Touch controller. Valid returns are: “Disabled”, “Not calibrated”, and “Ready”
MM.INFO$(TRACK MM.INFO$(TRACK)	The name of the current audio track playing. This returns "OFF" if nothing is playing.
MM.INFO$(VARCNT MM.INFO(VARCNT)	The number of variables currently used. Maximum 512.
MM.INFO$(VERSION MM.INFO$(VERSION)	The version number of the firmware as a floating point number in the form aa.bbcc where aa is the major version number, bb is the minor version number and cc is the revision number. For example version 5.03.00 will

	return 5.03 and version 5.03.01 will return 5.0301.
MM.ERRMSG$ **MM.ERRNO** 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. *Allowed syntax but saved as MM.INFO(ERRORNO)* *Allowed syntax but saved as MM.INFO(ERRORMSG)*
MM.ERRMSG$ - SDCARD related **MM.ERRNO - SDCARD related** 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 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 *Allowed syntax but saved as MM.INFO(ONEWIRE)*
MM.VER 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. *Allowed syntax but saved as MM.INFO(VERSION)*

MM.WATCHDOG 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 H7 is restarted. Options that are not permanent will be reset on startup.

	Permanent?	
ANGLE OPTION ANGLE RADIANS | DEGREES		This command switches trig functions between degrees and radians. Acts on SIN, COS, TAN, ATN, ATAN2, MATH ATAN3, ACOS, ASIN
AUTOREFRESH OPTION AUTOREFRESH mode		Only available when using buffered driver TFT drivers this command controls when screen updates take place. When autorefresh is "ON" updates take place immediately. When autorefresh is "OFF" updates take place in the framebuffer and are only written to the screen when autorefresh is next turned "ON" or the REFRESH command is issued. The buffered drivers are :ILI9481, ILI9341, ILI9488, SSD1963_4,SSD1963_4_16, ILI9341_16, ILI9341_8, SSD1963_x_BUFF, SSD1963_x_640 and SSD1963_x_8BIT The parameter is not saved and should be initialised in the program if other than the defaut is required. Defaults to ON for the buffered drivers and is OFF and not able to be changed for all other drivers. See the REFRESH command.
AUTORUN 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.
BASE 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.
BAUDRATE 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
BREAK 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.
CASE 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.

COLOURCODE 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.
CONTROLS 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***
CPU SPEED OPTION CPU SPEED n	✓	Sets the core CPU speed of the processor. Values between 10MHz and 600MHz are allowed, any request outside this range will default to the value of 480MHz or 400MHz. Anything over 480 MHz is overclocking the chip beyond the manufacturer's specification. 400 MHz is the default for the Nucleo V1 board. 480 Mhz is the default for all other boards. The actual speed selected on the 144 pin boards is the next lower multiple of 4MHz if the request speed is not a multiple of 4MHz. The actual speed selected on the 100 pin boards is the next lower multiple of 5MHz if requested speed is not a multiple of 5MHz.
DEFAULT 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.
DISPLAY OPTION DISPLAY lines [,chars]	✓	Set the characteristics of the display terminal used for the console. Both the LIST and EDIT commands need to know this information to correctly format the text for display. '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. Setting OPTION DISPLAY will send an ESC sequence to the connected VT100 terminal that sets it to the same display size. Only TeraTerm is known to respond to this. Other VT100 terminals will need to be manually changed, if possible. 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.
ESCAPE OPTION ESCAPE		Enables the ability to insert escape sequences into string constants. See the section Special Characters in Strings.

EXPLICIT 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.
FLASHPAGES OPTION FLASHPAGES n	✓	Sets the number of 128Kbyte pages to be used for storing programs. Valid values are 1 (default) to 4. Increasing the number of pages increases time to store programs and execute "NEW" commands.
FNKey OPTION FNKey string$	✓	Define the string that will be generated when a function key is pressed at the command prompt. 'FNKey' can be F1, or F5 tthru to F9. Example: OPTION F8 "RUN "+chr$(34)+"myprog"+chr$(34)+chr$(13)+chr$(10). Must entered at the command prompt (not in a program).
KEYBOARD REPEAT OPTION KEYBOARD REPEAT firstchar,nextchar	✓	Define the repeat characteristics of the USB keyboard when a key is held down. 'firstchar' is the time in milliseconds before a new character repeats. Default is 600mSec, the valid range is 100 to 2000 mSec 'nextchars' is the time in milliseconds before subsequent character repeats. Default is 150mSec, the valid range is 25 to 2000 mSec
LCDPANEL (SPI) OPTION LCDPANEL controller, orientation, D/C pin, reset pin [,CS pin] or OPTION LCDPANEL DISABLE	✓	Initialises an SPI TFT display using the nominated controller. These do not use the buffered driver.
LCDPANEL (SPI Buffered) OPTION LCDPANEL controller, orientation, D/C pin, reset pin [,CS pin] or OPTION LCDPANEL DISABLE	✓	Initialises an SPI TFT display using the nominated controller. A buffered driver is automatically used for all displays with resolution 480x320 of less. OPTION AUTOREFRESH and command REFRESH can be used to control when the updates to the screen take place. '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 toch controller • ILI9488 SPI based 480*320 SPI touch controller '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:** 'D/C 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. LCD_BL pin provides PWM to control the backlight via the BACKLIGHT command.

LCDPANEL ILI9341 (P8) OPTION LCDPANEL ILI9431_8, orientation	✓	Selects 8-bit bus operation of the ILI9341 display. Pin usage is exactly as per the SSD1963. Pins SSD1963_DB8 to SSD1963_D15 are allocated but not used. Uses a memory resident framebuffer to improve performance. LCD_BL pin provides PWM to control the backlight via the BACKLIGHT command.
LCDPANEL ILI9341 (P16) OPTION LCDPANEL ILI9431_16, orientation	✓	Selects 16-bit bus operation of the ILI9341 display. Pin usage is exactly as per the SSD1963. Uses a memory resident framebuffer to improve performance. LCD_BL pin provides PWM to control the backlight via the BACKLIGHT command.
LCDPANEL SSD1963 (P8) OPTION LCDPANEL controller, orientation [,speed] or OPTION LCDPANEL DISABLE	✓	Selects 8-bit bus operation of the various 800*480 SSD1963 displays, RGB888 'controller' can be: • SSD1963_4 4.3” panel 480*272 using SSD1963 • SSD1963_5 5" panels using the SSD1963 controller • SSD1963_5A alternative version of the 5" panel • SSD1963_5ER East Rising version of the 5" panel • SSD1963_7 7" panels using the SSD1963 controller • SSD1963_7A alternative version of the 7" panel • SSD1963_8 8" and 9” panels using the SSD1963 controller '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. ‘speed’ is an optional parameter used to slow the speed of the writing to the display. This maybe required if corruption of the displayed data is seen. Typically, corruption of white/black, white/blue, magenta/black, yellow/black and black/white. ’speed ‘ can have vaule 0 or 1. 0 is the default and is fast writing. 1 will slow the speed. 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. *The SSD1963_4 (480*272) is actually configured as a buffered driver, so OPTION AUTOREFRESH and command REFRESH can be used to control when the updates to the screen take place.* The BACKLIGHT command sends software commands to set the SSD1963 internal PWM.

LCDPANEL SSD1963 (P16) OPTION LCDPANEL controller, orientation [,speed] or OPTION LCDPANEL DISABLE	✓	Selects 16-bit bus operation of the various SSD1963 displays,RGB565 'controller' can be: • 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 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_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. 'speed' is an optional parameter used to slow the speed of the writing to the display. This maybe required if corruption of the displayed data is seen. Typically, corruption of white/black, white/blue, magenta/black, yellow/black and black/white. 'speed ' can have vaule 0 or 1. 0 is the default and is fast writing. 1 will slow the speed. 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. *The SSD1963_4_16 (480*272) is actually configured as a buffered driver, so OPTION AUTOREFRESH and command REFRESH can be used to control when the updates to the screen take place.* LCD_BL pin provides PWM to control the backlight via the BACKLIGHT command for the IPS_4_16 panel. The SSD1963 panels used software commands to set the displays own internal PWM.
LCDPANEL SSD1963 (8Bit) OPTION LCDPANEL SSD1963_n_8BIT, orientation [,speed]	✓	Selects 16-bit bus operation of the 5", 7" and 8" SSD1963 displays using a memory framebuffer with RGB222 64 colours for improved performance. This uses 25K of program memory leaving 473K for user programs
LCDPANEL SSD1963 (BUFF) OPTION LCDPANEL SSD1963_n_BUFF, orientation [,speed]	✓	Selects 16-bit bus operation of the 5", 7" and 8" SSD1963 displays using a memory framebuffer with RGB565 65000 colours for improved performance. This uses 400K of program memory leaving 98K for user programs
LCDPANEL SSD1963 (640) OPTION LCDPANEL SSD1963_n_640, orientation [,speed]	✓	Selects 16-bit bus operation of the 5" ,7" and 8" SSD1963 displays with RGB565 65000 colours with a usable display width of 640 pixels (i.e.640x480) using a memory framebuffer for improved performance. This uses 250K of program memory leaving 248K for user programs. This driver must be selected in order to use the CAMERA commands. It can also be used effectively for game programming using the SPRITE commands and functions

LCDPANEL CONSOLE OPTION LCDPANEL CONSOLE [font [, fc [,bc blight]]] **LCDPANEL NOCONSOLE** **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). This setting is saved in flash and will be automatically applied on startup. To disable it use the OPTION LCDPANEL NOCONSOLE command.
LIST 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.
MILLISECONDS 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”
PIN 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).
RESET OPTION RESET		Reset all saved options (including the PIN) to the default values.
RTC OPTION RTC CALIBRATE ±n	✓	Used to calibrate the battery backed Real Time Clock that keeps time in the Armmite H7. '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. Shows in OPTION LIST if not the default value of 0.

SDCARD OPTION SDCARD CS-Pin [,CD-Pin [,WP-Pin]] or OPTION SDCARD DISABLE	✓	The SD card is connected to the SYSTEM SPI. CS-Pin is the pin connected to the SD card CS pin. On the 100 pin boards (WeAct and DevEBox) the onboard SDCARD CS is wired to pin 79. This is a special case and the SPI is bitbanged on the appropriate pins. **OPTION SDCARD 110** for the LCD mounted SD card on Nucleo backpack. **OPTION SDCARD 78** recommended for an LCD mounted SD card on the 100 pin boards. **OPTION SDCARD 79** for the board mounted SD card on the 100 pin boards. 'CD-pin' is optional and is the I/O pin number that will be used to connect to the card detect pin on the SD card connector. The Armmite H7 will provide a weak pullup on this pin which is switched to ground when a card is inserted. Using a card detect pin for the SD makes the process of testing for card removal/insertion much more efficient, but is optional. 'WP-pin' is optional and is the I/O pin number that will be used to connect to the write protect pin on the SD card connector. The Armmite H7 will provide a weak pullup on this pin which is switched to ground when a write protected card isinserted. Some SD card connectors reverse the polarity of the 'CD-pin' or 'WP-pin' signal - ie, they are open (and therefore the signal is high) when the card is inserted or write protected. In this case the pin numbers used for 'CD-pin' and 'WP-pin' can be a negative number. This will tell MMBasic to invert the polarity of these signals. No pins are allocated for either CD-Pin or WP-Pin on the WeAct or DevEBox for the onboard SDCard on these 100 pin boards. OPTION SDCARD DISABLE can be used which will disable the SD card and return the I/O pins for general use.
SERIAL PULLUP 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.***
TAB OPTION TAB 2 | 4 | 8	✓	Set the spacing for the tab key. Default is 2. This option will be remembered even when the power is removed.
TOUCH OPTION TOUCH T_CS pin, T_IRQ pin OPTION TOUCH DISABLE	✓	Configures the Armmite H7 to suit the touch sensitive feature of an attached LCD panel. 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.

USBKEYBOARD OPTION USBKEYBOARD layout [, powerpinno [,noLED]]	✓	Sets the key layout for a connected USB keyboard. Valid layouts are UK, US, DE, FR, ES,BE. The optional "powerpinno" parameter specifies a pin which can be used to enable power to the USB port on the Nucleo 144 pin board. Normally the pin will be set high when enabled. However, by specifying the pin number as a negative value the pin will be set low. (The V2 Nucleo needs it low) **OPTION USBKEYBOARD US, 91** (for V1 Nucleo) **OPTION USBKEYBOARD US, -79** (for V2 Nucleo) **OPTION USBKEYBOARD US** for the 100 pin boards. Pin 91 high enables 5V power to keyboard on V1 Nucleo Pin 79 low enables 5V power to keyboard on V2 Nucleo If the 5V power to the USB port is hardwired then powerpinno should be omitted. e.g. 100 pin boards. Set the language type for the attached USB keyboard. character code defining the keyboard layout. The choices are US for 'nn is a two the standard keyboard layout in the USA, Australia and New Zealand and UK for the United Kingdom, DE for Germany, FR for France, ES for Spain and BE for Belgium. The optional noLED parameter can be set to 1 to block sending the command to the keyboard that lights the LEDs relating to Caps Lock etc. This may be needed on some keyboards which do not process this command properly and may lock up. It defaults to 0 if not specified (i.e. the LED commands are sent). This command can only be run from the command line and will cause a restart.This setting is remembered even after a firmware upgrade.
VCC 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. See Setting Option VCC for details of setting VCC using the chips internal calibration value.

Commands

Square brackets indicate that the parameter or characters are optional.

‘ (single quotation mark) ‘ (single quotation mark)	Starts a comment and any text following it will be ignored. Comments can be placed anywhere on a line.
***file** *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.
? (question mark) ? (question mark)	Shortcut for the PRINT command.
ADC ADC	The ADC functionality that can capture up to 3 channels of data in the background at up to 500KHz per channel (480KHz for 400MHz processors) with user selectable triggering
ADC OPEN frequency, channel1-pin [,channel2-pin] [,channel3-pin] [, interrupt]	Open the ADC channels. "frequency" is the sampling frequency in Hz. The maximum frequency is 500KHz. Above 160KHz the conversion is 8-bits per channel From 40KHz to 160KHz the conversion is 10-bits per channel From 20KHz to 40KHz the conversion is 12-bits per channel From 10KHz to 20KHz the conversion is 14-bits per channel Below 10KHz conversion is 16-bits per channel This is automatically applied in the firmware. **144 pin Nucleo** "channel1-pin" can be one of 34, 35, 36, 37, 42, 43, 44, 45, 47, 49, 50 "channel2-pin" can be one of 13, 14, 15, 18, 19, 20,21, 22, 26, 27, 28, 29 "channel3-pin" can be one of 53, 54 **100 Pin WeAct and DEV BOX** "channel1-pin" can be one of 22, 23, 24, 25, 32, 33,35, "channel2-pin" can be one of 15, 16, 17, 18 "channel3-pin" can be one of 30, 31 The "interrupt" parameter is a normal MMBasic subroutine that can 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 otherwise it will give an error.

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

BITBANG LCD BITBANG LCD INIT d4, d5, d6, d7, rs, en or BITBANG LCD line, pos, text$ or BITBANG LCD CLEAR or BITBANG LCD CLOSE	Display text on an LCD character display module. This command will work with most 1-line, 2-line or 4-line LCD modules that use the KS0066, HD44780 or SPLC780 controller (however this is not guaranteed). The LCD INIT command is used to initialise the LCD module for use. 'd4' to 'd7' are the I/O pins that connect to inputs D4 to D7 on the LCD module (inputs D0 to D3 should be connected to ground). 'rs' is the pin connected to the register select input on the module (sometimes called CMD). 'en' is the pin connected to the enable or chip select input on the module. The R/W input on the module should always be grounded. The above I/O pins are automatically set to outputs by this command. When the module has been initialised data can be written to it using the LCD command. 'line' is the line on the display (1 to 4) and 'pos' is the character location on the line (the first location is 1). 'text$' is a string containing the text to write to the LCD display. 'pos' can also be C8, C16, C20 or C40 in which case the line will be cleared and the text centred on a 8 or 16, 20 or 40 line display. For example: `LCD 1, C16, "Hello"` LCD CLEAR will erase all data displayed on the LCD and LCD CLOSE will terminate the LCD function and return all I/O pins to the not configured state. See the chapter LCD Display for more details.
BITBANG LCD CMD d1 [, d2 [, etc]] or BITBANG LCD DATA d1 [, d2 [, etc]]	These commands will send one or more bytes to an LCD display as either a command (LCD CMD) or as data (LCD DATA). Each byte is a number between 0 and 255 and must be separated by commas. The LCD must have been previously initialised using the LCD INIT command (see above). These commands can be used to drive a non standard LCD in "raw mode" or they can be used to enable specialised features such as scrolling, cursors and custom character sets. You will need to refer to the data sheet for your LCD to find the necessary command and data values.
BITBANG WS2812 BITBANG WS2812 type, pin, nbr, value%[()]	This command outputs the required signals to drive one or more WS2812 LED chips connected to 'pin'. Note that the pin must be set to a digital output before this command is used. 'type' is a single character specifying the type of chip being driven: O = original WS2812 B = WS2812B S = SK6812 W=SK6812 RGBW Leds 'nbr ' is the number of LEDs in the chain (1 to 512). The 'value%()' array should be an integer array sized to have exactly the same number of elements as the number of LEDs to be driven. Each element in the array should contain the colour in the normal RGB888 format (i.e. 0 to &HFFFFFF). For the SK6812 RGBW put the white value in bits 24-31. e.g. &HFF000000 If only one LED is connected then a single integer should be used for value% (ie, not an array). dim b%(6)=(rgb(red), rgb(green), rgb(blue), rgb(Yellow), rgb(cyan), rgb(magenta), rgb(white)) setpin 1,dout ws2812 1,7,b%() will output the specified colours to an array of 7 WS2812 LEDs

	Note: All interrupts are turned off during transmission to achieve the sub microsecond timings required. The 512 limit is nominal and using a high number of LEDs may cause unforeseen issues with missed interrupts.
BLIT BLIT READ [#]b, x, y, w, h BLIT WRITE [#]b, x, y 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'. The width and height are automatically obtained from the buffer. BLIT CLOSE will close the memory buffer '#b' to allow it to be used for another BLIT READ operation and recover the memory used. Notes: • Sixty four buffers are available ranging from #1 to #64. • When specifying the buffer number the # symbol is optional. • All other arguments are in pixels.
BLIT x1, y1, x2, y2, w, h	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 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 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.
CAMERA CAMERA OPEN CAMERA CAPTURE CAMERA SAVE “filename” CAMERA REGISTER register, value CAMERA BACKUP CAMERA CLOSE	*Not available on 100 pin boards* Initialises an OV7670 camera ready for use Captures an image from the camera to a connected 800x480 SSD1963 display. The display must be set to 640 pixel mode using command OPTION LCDPANEL SSD1963_x_640, LANDSACAPE Saves the on-screen image to the SDcard. If the file extension is not specified then ".BMP" is appended. Can be used to change the camera settings see the datasheet for details Disables the camera and frees the allocated pins
CAT 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 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 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 CLEAR	Delete all variables and recover the memory used by them.
CLEAR VARS CLEAR VARS variable [,variable] …	Deletes listed variables and recovers the memory used by them. New form that replaces the ERASE command to save command tokens. Either is accepted.
CLOSE 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 CLS [colour]	Clears the LCDPANEL Optionally 'colour' can be specified which will be used for the background when clearing the screen.
COLOUR 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 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 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 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.
CFUNCTION 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 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 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`
DAC DAC n, voltage	Sets the DAC channel (1 or 2) to the voltage requested. This command cannot be used if the DACs are in use for audio output.
DAC START frequency, DAC1array%() [,DAC2array%()]	Sets up the DAC to create an arbitrary waveform. DAC1array%() and optional DAC2array%() should contain numbers in the range 0-4095 to suit the 12-bit DACs. Once started the output continues in the background and control returns to MMBasic. The software automatically and separately uses the number of items in each of the arrays to drive the DACs. The frequency is the rate at which the DACs change value. The maximum frequency is 700KHz. As an example if there are 180 items in the array c%() which are displayed at a frequency of 100,000 Hz this will give a waveform frequency of 100,000/180 = 555Hz. If there are 90 items in the array d%() at the same frequency of 100,000 Hz this will at the same time produce a waveform frequency of 100,000/90 = 1111Hz.
DAC STOP	Stops the DAC output and returns the DACs to normal use.
DATA 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$** DATE$ = "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 DEFINEFONT #n hex [[hex[…] hex [[hex[…] END DEFINEFONT	This will define an embedded font which can be used exactly same as the built in fonts (ie, selected using the FONT command or specified in the TEXT command). MMBasic must execute the font in order for it to be loaded. '#n' is the font's reference number (1 to 16). It can be the same as an existing font (except fonts 1, 6 and 7) and in that case it will replace that font. Each 'hex' must be exactly eight hex digits and be separated by spaces or new lines from the next. Multiple lines of 'hex' words can be used with the command terminated by a matching END DEFINEFONT.
DHT22 DHT22	See the HUMID command which replaces the DHT22 command.
DIM DIM [type] decl [,decl]... where 'decl' is: var [length] [type] [init]	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

'var' is a variable name with optional dimensions

'length' is used to set the maximum size of the string to 'n' as in LENGTH n

'type' is one of FLOAT or INTEGER or STRING (the type can be prefixed by the keyword AS - as in AS FLOAT)

'init' is the value to initialise the variable and consists of: = <expression>

For a simple variable one expression is used, for an array a list of comma separated expressions surrounded by brackets is used.

Examples:

DIM nbr(50)

DIM INTEGER nbr(50)

DIM name AS STRING

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

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

DIM strn$(200) LENGTH 20

DIM STRING strn(200) LENGTH 20

DIM a = 1234, b = 345

DIM STRING strn = "text"

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

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)
```

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 H7 requires 8 bytes a total of 1848 bytes of memory will be allocated.

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

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

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

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

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

In the above example you can also use the Microsoft syntax of specifying the

	type 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 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 EDIT	Invoke the full screen editor. See the section Full Screen Editor for details of how to use the editor.
ELSE ELSE	Introduces a default condition in a multiline IF statement. See the multiline IF statement for more details.
ELSEIF 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	End the running program and return to the command prompt.
END CSUB 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 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
ENDIF
or
END IF
	Terminates a multiline IF statement.
	See the multiline IF statement for more details.

END SELECT 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.

END SUB	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 ERASE variable [,variable]...	Deletes variables and frees up the memory allocated to them. This will work with arrayed variables and normal (non array) variables. Arrays can be specified using empty brackets (eg, `dat()`) or just by specifying the variable's name (eg, `dat`). This syntax is supported but command is stored as: CLEAR VARS variable [,variable] See CLEAR command. Use CLEAR to delete all variables at the same time (including arrays).
ERROR ERROR [error_msg$]	Forces an error and terminates the program. This is normally used in debugging or to trap events that should not occur.
EXECUTE EXECUTE commands$	This executes the Basic command "command$". Use should be limited to basic commands that execute sequentially for example the GOTO statement will not work properly. Things that are tested and work OK include GOSUB, Subroutine calls, other simple statements (like PRINT and simple assignments) Multiple statements separated by : (colon) are not allowed and will error The command sets an internal watchdog before executing the requested command and if control does not return to the command, like in a GOTO statement, the timer will expire. In this case you will get the message "Command timeout". RUN is a special case and will cancel the timer allowing you to use the command to chain programs if required. Variable persistence can be achieved using VAR SAVE/RESTORE if required. In the case of the CMM2 and Armmite F4 these use 4K of battery backed RAM so can be done without consequence. In the case of the Armmite H7 saving is done to flash memory so care should be exercised to stay within the limits of the write capability of the flash chip "Min. 100K Program-Erase cycles per sector"
EXIT... EXIT DO EXIT FOR EXIT FUNCTION EXIT SUB	EXIT DO provides an early exit from a DO...LOOP EXIT FOR provides an early exit from a FOR...NEXT loop. EXIT FUNCTION provides an early exit from a defined function. EXIT SUB provides an early exit from a defined subroutine. The old standard of EXIT on its own (exit a do loop) is also supported.
FILES FILES [fspec$]	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 [#]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 Fonts 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 USB Keyboard and LCDPANEL as Console for details.
FOR 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 FTT	Now is part of the MATH command. See the MATH command

FUNCTION
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 GOSUB	See Obsolete Commands and Functions section.
GOTO 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— *** GUI Controls ***	Used to implement a GUI display with touch support. See Advanced Graphics for details.
GUI AREA 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.
GUI BARGAUGE 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 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.
GUI BEEP 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 GUI BUTTON #ref, caption$, startX, startY, width, height [, FColour] [,BColour]	This will draw a momentary button which is a square switch with the caption on its face.
GUI CAPTION GUI CAPTION #ref, text$, startX, startY [,align$] [,	This will draw a text string on the screen.

FColour] [, BColour]	
GUI CHECKBOX 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 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 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 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 can contain one or more tilde characters (~) which indicate a line break. Up to 10 lines can be displayed inside the box. This control does not respond to touch.
GUI ENABLE 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 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 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 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. e.g. 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 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$' are optional and will default to the values used in the previous definition of a GUI GAUGE. 'c1', 'ta', 'c2', 'tb', 'c3', 'tc' and 'c4' are optional and if not specified the gauge will use less colours. If all are omitted the gauge will be drawn using 'Fcolour'. The section *Advanced Graphics* has a more detailed description.
GUI HIDE GUI HIDE #ref1 [,#ref2, #ref3, etc] or GUI HIDE ALL	This will hide the controls in the list. Hidden controls do not respond to touch and will not be visible. '#ref' is the control's reference number. The keyword ALL can be used as the argument and that will hide all controls. GUI SHOW can be used to restore the controls.
GUI INTERRUPT GUI INTERRUPT down [, up]	This command will setup an interrupt that will be triggered on a touch on the LCD panel and optionally if the touch is released. 'down' is the subroutine to call when a touch down has been detected. 'up' is the subroutine to call when the touch has been lifted from the screen ('up' and 'down' can point to the same subroutine if required). Specifying the number zero (single digit) as the argument will cancel both of these interrupts. ie: `GUI INTERRUPT 0`
GUI LED GUI LED #ref, caption$, 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 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 ACTIVATE	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 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 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 GUI RADIO #ref, caption$, centerX, centerY, radius,	This will draw a radio button with a caption. '#ref' is the control's reference number.

colour	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 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 RESTORE GUI RESTORE #ref1 [,#ref2, #ref3, etc] or GUI RESTORE ALL	This will undo the effects of both GUI HIDE and 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 SETUP 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 PAGE command.
GUI SHOW 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 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 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 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— *** GUI Commands ***	These GUI commands are used to configure LCDPANEL and TOUCH. Also includes the GUI BITMAP command to display a bitmap.
GUI 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 Graphic Commands and Functions for a definition of the colours and graphics coordinates.
GUI CALIBRATE 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 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 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 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
I2C I2C	The I2C commands will send and receive data over an I^2C channel. I2C (no suffix) refers to channel 1 while command 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 100 400 or 1000. 'timeout' is a value in milliseconds after which the master send and receive commands will be interrupted if they have not completed. The minimum value is 100. A value of zero will disable the timeout (though this is not recommended). The I2C SCL and SDA pins are configured and this will override any previous SETPIN operations that may have been used on these pins.
I2C WRITE addr, option, sendlen, senddata [,sendata ....]	Send data to the I^2C slave device. 'addr' is the slave's I^2C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 'sendlen' is the number of bytes to send. 'senddata' is the data to be sent - this can be specified in various ways (all values sent will be between 0 and 255):

	• The data can be supplied as individual bytes on the command line. Example: I2C WRITE &H6F, 0, 3, &H23, &H43, &H25 • The data can be in a one dimensional array specified with empty brackets (ie, no dimensions). 'sendlen' bytes of the array will be sent starting with the first element. Example: I2C WRITE &H6F, 0, 3, ARRAY() The data can be a string variable (not a constant). Example: I2C WRITE &H6F, 0, 3, STRING$
I2C READ addr, option, rcvlen, rcvbuf	Get data from the I^2C slave device. 'addr' is the slave's I^2C address. 'option' can be 0 for normal operation or 1 to keep control of the bus after the command (a stop condition will not be sent at the completion of the command) 'rcvlen' is the number of bytes to receive. 'rcvbuf' is the variable or array used to save the received data - this can be: • A string variable. Bytes will be stored as sequential characters in the string. • A one dimensional array of numbers specified with empty brackets. Received bytes will be stored in sequential elements of the array starting with the first. Example: I2C READ &H6F, 0, 3, ARRAY() A normal numeric variable (in this case rcvlen must be 1).
I2C CLOSE	Disables the master I^2C module. This command will also send a stop if the bus is still held. The I2C pins are released and available to MMBasic as digital pins.
I2C CHECK addr	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.
I2C2	As above but for 2nd I2C channel.
IF 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 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 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 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 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. Pin 14 on the 144 pin Nucleo board and pin 93 on the 100 pin boards. (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 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 IRETURN	See obsolete commands.
KEYPAD 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 KILL file$	Deletes the file or empty directory specified by 'file$'. If there is an extension it must be specified.
LCD LCD INIT d4, d5, d6, d7, rs, en LCD line, pos, text$ LCD CLEAR LCD CLOSE	Now BITBANG LCD These accepted but saved as the new format of the command.
LCD CMD d1 [, d2 [, etc]] LCD DATA d1 [, d2 [, etc]]	Now BITBANG LCD These accepted but saved as the new format of the command.
LET 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 LIBRARY SAVE LIBRARY DELETE	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 H7 and act the same as built in commands and functions in MMBasic. See the Program Initialisation, CFunctions and the Library

LIBRARY LIST [ALL]	chapter 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. LIBRARY LIST will list the contents of the library. [ALL] will list the library without 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.
LINE LINE x1, y1, x2, y2 [, LW [, C]]	Draws a line starting at the coordinates 'x1' and 'y1' and ending at 'x2' and 'y2'. 'LW' is the line's width and is only valid for horizontal or vertical lines. It defaults to 1 if not specified or if the line is a diagonal. 'C' is an integer representing the colour and defaults to the current foreground colour. All parameters can now be expressed as arrays and the software will plot the number of lines as determined by the dimensions of the smallest array. 'x1', 'y1', 'x2', and 'y2' must all be arrays or all be single variables /constants otherwise an error will be generated. 'lw' and 'c' can be either arrays or single variables/constants.
LINE INPUT 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 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$' is specified then that file on the Flash Filesystem or SD Card will be listed
LIST COMMANDS LIST COMMANDS **LIST FUNCTIONS** LIST FUNCTIONS	Lists all valid commands List all valid functions and operators
LOAD LOAD file$ [,R]	Loads a program called 'file$' from the current drive into program memory. If the optional suffix ,R is added the program will be immediately run without prompting. If an extension is not specified ".BAS" will be added to the file name.
LOAD DATA LOAD DATA fname$, 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. The ".DAT" extension to the filename is added automatically if not provided. See SAVE DATA as well.
LOAD IMAGE LOAD IMAGE file$ [, x, y]	Load a bitmapped image from the SD card and display it on the LCD panel. "file$' is the name of the file and 'x' and 'y' are the screen coordinates for the top left hand corner of the image. If the coordinates are not specified the image will be drawn at the top left hand position on the screen. If an extension is not specified ".BMP" will be added to the file name. All types of the BMP format are supported including black and white and true colour 24-bit images. The image can be of any size and pixels off the screen will be ignored.
LOAD JPG LOAD JPG file$ [, x, y]	Load a bitmapped image from the SD card and display it on the LCD panel. "file$' is the name of the file and 'x' and 'y' are the screen coordinates for the top left hand corner of the image. If the coordinates are not specified the image will be drawn at the top left hand position on the screen. If an extension is not specified ".JPG" will be added to the file name. Images cannot use progressive encoding. The image can be of any size and pixels off the screen will be ignored.
LOAD NVM LOAD NVM string$	See also SAVE NVM.These commands read and write a string up to 128 bytes in length to battery backed memory in the RTC. The data is saved in the battery backed memory as long as VBAT is powered. 128 bytes is the total size of the RTC battery backed memory These commands can be used to store and retrieve data you don't want to lose when the NEW command is issued or an implicit NEW applied when a program is updated by uploading. (Data saved with VAR SAVE is cleared when the NEW command is issued.) e.g. A$="1234567890" save nvm a$ power off, then power on dim a$ load nvm a$ print a$ Use BIN2STR$ and STR2BIN to format numbers for saving and restoring.
LOCAL 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 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.
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	Will trim ‘nbr’ characters from the left of a long string. array%() must be a

array%(), nbr LONGSTRING UCASE array%()	long string variables. 'nbr' must be an integer constant or expression. Will convert any lowercase characters in array%() to uppercase. array%() must be long string variable.
LOOP LOOP [UNTIL expression]	Terminates a program loop: see DO.
MATH 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 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 SLICE sourcearray(), [d1] [,d2] [,d3] [,d4] [,d5] ,destinationarray()	This command copies a specified set of values from a multi-dimensional array into a single dimensional array. It is much faster than using a FOR loop. The slice is specified by giving a value for all but one of the source array 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 INSERT targetarray(), [d1] [,d2] [,d3] [,d4] [,d5] , sourcearray()	This is the opposite of MATH SLICE, has a very similar syntax, and allows you, for example, to substitute a single vector into an array of vectors with a single instruction e.g. OPTION BASE 1 DIM targetarray(3,4,5) DIM sourcearray(4)=(1,2,3,4) MATH INSERT targetarray(), 2, , 3, sourcearray()

	Will set elements 2,1,3 = 1 and 2,2,3 = 2 and 2,3,3 = 3 and 2,4,3 = 4
Matrix arithmetic	
MATH M_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()	Quick mechanism to print a small array on a single line
MATH V_NORMALISE inV(), outV()	Converts a vector inV() to unit scale and puts the answer in outV() (sqr(x*x + y*y +.....)=1 There is no limit on number of elements in the vector
MATH V_MULT matrix(), inV(), outV()	Multiplies matrix() and vector inV() returning vector outV(). The vectors and the 2D matrix can be any size but must have the same cardinality.
MATH V_CROSS inV1(), inV2(), outV()	Calculates the cross product of two three element vectors inV1() and inV2() and puts the answer in outV()
Quaternion arithmetic	
MATH Q_INVERT inQ(), outQ()	Invert the quaternion in inQ() and put the answer in outQ()
MATH Q_VECTOR x,y,z,outVQ()	Converts 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()

MATH FFT MATH FFT signalarray!(), FFTarray!()	Performs a fast fourier transform of the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "FFTarray" must be floating point and have dimension 2*N where N is the same as the signal array (e.g. f(1,1023) assuming OPTION BASE is zero) The command will return the FFT as complex numbers with the real part in f(0,n) and the imaginary part in f(1,n)
MATH FFT INVERSE FFTarray!(), signalarray!()	Performs an inverse fast fourier transform of the data in “FFTarray!”. "FFTarray" must be floating point and have dimension 2*N where N must be a power of 2 (e.g. f(1,1023) assuming OPTION BASE is zero) with the real part in f(0,n) and the imaginary part in f(1,n). "signalarray" must be floating point and the single dimension must be the same as the FFT array. The command will return the real part of the inverse transform in "signalarray".
MATH FFT MAGNITUDE signalarray!(),magnitudearray!()	Generates magnitudes for frequencies for the data in “signalarray!” "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "magnitudearray" must be floating point and the size must be the same as the signal array The command will return the magnitude of the signal at various frequencies according to the formula: frequency at array position N = N * sample_frequency / number_of_samples
MATH FFT PHASE signalarray!(), phasearray!()	Generates phases for frequencies for the data in “signalarray!”. "signalarray" must be floating point and the size must be a power of 2 (e.g. s(1023) assuming OPTION BASE is zero) "phasearray" must be floating point and the size must be the same as the signal array. The command will return the phase angle of the signal at various frequencies according to the formula above.
MATH SENSORFUSION MATH SENSORFUSION type ax, ay, az, gx, gy, gz, mx, my, mz, pitch, roll, yaw [,p1] [,p2]	Calculates pitch, roll and yaw angles from accelerometer and magnetometer inputs. Valid fusion types are MAHONY and MADGWICK. Usage is described in Appendix A

MEMORY
MEMORY
	List the amount of memory currently in use. For example:
	`> MEMORY`
	`Program Flash:`
	`  13K (10%) Program (196 lines)`
	`   1K ( 0%) 1 Embedded C Routine`
	`   5K ( 1%) 1 Embedded Fonts`
	` 109K (89%) Free`
	` 256K Unallocated`
	
	`Saved Vars Flash:`
	`  76K (59%) 3 Saved Variables (77589 bytes)`
	`  52K (41%) Free`
	
	`Library Flash:`
	`   1K ( 1%) Library`

	127K (99%) Free RAM: 76K (15%) 3 Variables 0K (0%) General 422K (85%) 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 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 SET 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/UNPACK 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$** MID$(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$.
MKDIR MKDIR dir$	Make, or create, the directory ‘dir$’ on the SD card.
NAME … AS NAME old$ 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 NEW	Deletes the program in flash, clears all variables including saved variables and resets the interpreter (ie, closes files, serial ports, etc).
NEXT 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 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 … GOTO 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 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 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 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. COM4 is not available on 100pin devices if a parallel LCD Panel is enabled. 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), INV, 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 1200 (the minimum) and 1000000 Hz. (1MHz) Default is 9600. • 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes. • 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt. • 'int-trigger' sets the trigger condition for calling the interrupt subroutine. If it is a normal number the interrupt subroutine will be called when this number of characters has arrived in the receive queue. All parameters except the serial port name (COMn:) are optional. If any one parameter is left out then all the following parameters must also be left out and the defaults will be used. These options can be added to the end of 'comspec$' • 'INV' specifies that the transmit and receive polarity is inverted. • OC' will force the transmit pin (and DE on COM2:) to be open collector. The default is normal (0 to 3.3V) output. • 'S2' specifies that two stop bits will be sent following each character transmitted. • '7BIT' will specify that 7 bit transmit and receive is to be used. • 'ODD' will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'EVEN' will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9) • 'DEP' will enable RS485 mode with positive output on COM2-DE • 'DEN' will enable RS485 mode with negative output on COM2-DE
OPEN comspec$ 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 OPTION	See the section Option Settings earlier in this manual.
PAGE PAGE #n [,#n2, #n3, etc]	Now GUI PAGE PAGE is accepted but converted to GUI PAGE
PAUSE PAUSE delay	Halt execution of the running program for 'delay' ms. This can be a fraction. For example, 0.2 is equal to 200µs. The maximum delay is 2147483647ms (about 24 days).

	Note that interrupts will be recognised and processed during a pause.
PIN 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 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 Graphic Commands and Functions for a definition of the colours and graphics coordinates.
PLAY PLAY EFFECT file$ [,interrupt]	This will play the WAV file ' file$' at the same time as a MOD file is playing. If a previous EFFECT file is playing this command will immediately terminate it and commence playing the new file. The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. Note: wav files played using PLAY EFFECT during mod file playback must have the same sample rate as the modfile output. Files can be mono or stereo
PLAY TONE left, right [, dur]	Generates two separate sine waves on the sound output left and right channels. The tone plays in the background (the program will continue running after this command). 'left' and 'right' are the frequencies in Hz to use for the left and right channels. 'dur' specifies the number of milliseconds that the tone will sound for. MMBasic will round the time to the next nearest complete waveform of the first frequency specified so that the tone will always finish with the DC level in the middle and no discontinuity. If the duration is not specified, the tone will continue until explicitly stopped or the program terminates. The frequency can be from 1Hz to 20KHz and is very accurate (it is based on a crystal oscillator). The frequency can be changed at any time by issuing a new PLAY TONE command.
PLAY WAV file$ [, interrupt] or PLAY FLAC file$ [, interrupt] or PLAY MP3 file$ [,interrupt]	Play an audio file on the audio (DAC) output. 'file$' is the file to play (the appropriate extension will be appended if missing). The file is played in the background, 'interrupt' is optional and is the name of a subroutine that will be called when the file has finished playing. For WAV files MMBasic will automatically compensate for the frequency, number of bits and number of channels of the WAV file. For FLAC files the supported frequencies are: 44100Hz 16-bit (CD quality) and 24-bit 48000Hz 16-bit and 24-bit 88200Hz 16-bit and 24-bit

	96000Hz 24-bit Maximums for FLAC and WAV file playback are 96KHz 24-bit. Both will auto-configure to the file provided. As an indication, 96KHz 24-bit FLAC uses just over 50% of the CPU's resources. If ‘file$’ is a directory then the firmware will list all the files of the relevant type in that directory and start playing them one-by-one. To play files in the current directory use an empty string (ie, ""). Each file listed will play in turn and the optional interrupt will fire when all files have been played. The filenames are stored with full path so you can use CHDIR while tracks are playing without causing problems. All files in the directory are listed if the command is executed at the command prompt but the listing is suppressed in a program
PLAY MODFILE file$ [,samplerate]	Will play a MOD file on the DAC outputs. 'file$' is the MOD file to play (the extension of .mod will be appended if missing). The MOD file is played in the background and will run continuously until PLAY STOP is called. The MOD encoder supports 32 channels, 16-bit resolution (but the DACs are only 12 bit), 32 samples, no fixed maximum size. The optional parameter samplerate specifies the number of samples per second generated by the modfile engine. The default is 44100. Processor overhead is reduced by decreasing this. Valid values are 8000, 16000, 22050, 44100, 48000 Note: wav files played using PLAY EFFECT during modfile playback must have the same sample rate as the modfile output.
PLAY MODSAMPLE sampleno, channelno [,volume] [,samplerate]	Plays one of the samples in the MOD file concurrently with the main MOD file playback. This allows sound effects to be incorporated in the MOD file. “sampleno” can be in the range 1 to 32. Up to 4 samples can be played simultaneously on independent channels using the specified “channelno” which must be in the range 1 to 4. The optional “volume” should be set in the range 0 to 64 (default 64). The optional “samplerate” specifies the update rate for the sample. The default is 16000. Changing this will change the pitch of the sample and the duration of playback and it should be set to the sample's original rate for playback as recorded.
PLAY SOUND soundno, channelno, type [,frequency] [,volume]	Play a series of sounds simultaneously on the audio output. 'soundno' is the sound number and can be from 1 to 4 allowing for four simultaneous sounds on each channel. 'channelno' specifies the output channel. It can be L (left speaker), R (right speaker) or B (both speakers) 'type' is the type of waveform. It can be S (sine wave), Q (square wave), T (triangle wave), W (rising sawtooth), N (noise), P (periodic noise) or O (turn off sound). Type N is true white noise. In this case the frequency parameter specifies the number of periods of 1/70000 seconds that the output stays at a particular random value. Type P is periodic white noise. In this case the frequency is some sort of relationship to the periodic frequency of the noise 'frequency' is the frequency from 1 to 20000 (Hz) and it must be specified

	except when type is O. 'volume' is optional and must be between 1 and 25. It defaults to 25 The first time PLAY SOUND is called all other audio usage will be blocked and will remain blocked until PLAY STOP is called. Output can be stopped temporarily using PLAY PAUSE and PLAY RESUME. Calling SOUND on an already running 'soundno' will immediately replace the previous output. Individual sounds are turned off using type “O” Running 4 sounds simultaneously on both channels of the audio output consumes about 23% of the CPU.
PLAY PAUSE PLAY RESUME PLAY STOP	PLAY PAUSE will temporarily halt the currently playing file or tone. PLAY RESUME will resume playing a sound that was paused. PLAY STOP will terminate the playing of the file or tone. When the program terminates for whatever reason the sound output will also be automatically stopped.
PLAY NEXT PLAY PREVIOUS	When playing a sequence of audio tracks (by using PLAY MP3 on a directory holding multiple MP3 files) these commands can be used to skip forward or back a file. The commands PLAY PAUSE, RESUME, VOLUME can also be used.
PLAY TTS [PHONETIC] "text" [,speed] [,pitch] [,mouth] [,throat] [, interrupt]	Outputs text as speech on the DAC outputs. See http://www.retrobits.net/atari/sam.shtml for details of parameter usage. The command is non-blocking and the speech is played in the background. 'interrupt' is optional and is the name of a subroutine which will be called when the speech has finished playing.
PLAY VOLUME left, right	Will adjust the volume of the audio output. 'left' and 'right' are the levels to use for the left and right channels and can be between 0 and 100 with 100 being the maximum volume. There is a linear relationship between the specified level and the output. The volume defaults to maximum when a program is run.
POKE 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 POKE VARTBL, offset, byte POKE DISPLAY command	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 ±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

[,data1] [,data2] [,datan] POKE DISPLAY HRES n POKE DISPLAY VRES n	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. Note:IPS_4_16 requires16 bit commands so requires &H2800 and &H2900 These commands change the stored value of MM.HRES and MM.VRES allowing the programmer to configure non-standard displays.
POLYGON 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 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 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 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 PWM 1, freq, 1A [,1B] [,1C][,1D] PWM 2, freq, 2A [,2B] [,2C][,2D] PWM channel, STOP	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' or'2' is the controller number and 'freq' is the output frequency. 1A, 1B 1C and 1Dare the duty cycle for each of the controller 1 outputs, while 2A, 2B,2C and 2D are the duty cycle for the controller 2 outputs. The specified I/O pins will be automatically configured as outputs while any others will be unaffected and can be used for other duties. The duty cycle for each output is independent of the others and is specified as a percentage. If it is close to zero the output will be a narrow positive pulse,

	if 50 a square wave will be generated and if close to 100 it will be a very wide positive pulse Minimum frequency is 1Hz, maximum is 24MHz. Duty cycle and frequency accuracy will depend on frequency. The frequency can be any value of 240,000,000/n. The output will run continuously in the background while the program is running and can be stopped using the STOP command. The frequency and duty cycle can be changed at any time (without stoping the output) by issuing a new PWM command. The PWM function will take control of any specified outputs and when stopped the pins will be returned to a high impedance "not configured" state.
RBOX 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 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|RESTORE READ SAVE or READ RESTORE	READ SAVE will save the virtual pointer used by the READ command to point to the next DATA to be read. READ RESTORE will restore the pointer that was previously saved. This enables subroutines to READ data and then restore the read pointer so as not to disturb other parts of the program that may be reading the same data statements. These commands can be nested.
REFRESH REFRESH	Flushes the entire memory buffer of the display to the screen. This command can be used with OPTION AUTOREFRESH OFF to control when the screen is updated

REM 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 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 RMDIR dir$	Remove, or delete, the directory 'dir$' on the SD card.
RUN RUN [file$][,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 SAVE file$	Saves the program to the current working directory of the SD card as 'file$'. Example: SAVE "TEST.BAS" If an extension is not specified ".BAS" will be added to the file name.
SAVE DATA SAVE DATA fname$, address, size	Saves *size* bytes to file *fname$* starting from *address*. Data is saved in raw binary format. Together with LOAD DATA this allows you to very easily to save and restore the contents of an array to and from disk. The code tries to protect you from crashing the system to the extent possible but there are many ways you can misuse the LOAD DATA command if you try. The ".DAT" extension to the filename is added automatically if not provided. See LOAD DATA as well.
SAVE IMAGE SAVE IMAGE file$ [, x, y, w, h]	Save the current image on the LCD display as a 24-bit BMP file. 'file$' is the name of the file. If an extension is not specified ".BMP" will be added to the file name. 'x', 'y', 'w' and 'h' are optional and are the coordinates (x and y are the top left coordinate) and dimensions (width and height) of the area to be saved. If not specified the whole screen will be saved.
SAVE NVM SAVE NVM string$	See also LOAD NVM. These commands read and write a string up to 128 bytes in length to battery backed memory in the RTC. The data is saved in the battery backed memory as long as VBAT is powered. 128 bytes is the total size of the RTC battery backed memory These commands can be used to store and retrieve data you don't want to lose when the NEW command is issued or an implicit NEW applied when a program is updated by uploading. (Data saved with VAR SAVE is cleared when the NEW command is issued.)

	e.g. A$="1234567890" save nvm a$ power off, then power on dim a$ load nvm a$ print a$ Use BIN2STR$ and STR2BIN to format numbers for saving and restoring
SEEK 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
SELECT CASE value
CASE testexp [[, testexp] …]
<statements>
<statements>
CASE ELSE
<statements>
<statements>
END SELECT
	Executes one of several groups of statements, depending on the value of an expression. 'value' is the expression to be tested. It can be a number or string variable or a complex expression. 'testexp' is the value that 'exp' is to be compared against. It can be:
	• A single expression (ie, 34, "string" or PIN(4)*5) to which it may equal
	• A range of values in the form of two single expressions separated by the keyword "TO" (ie, 5 TO 9 or "aa" TO "cc")
	• A comparison starting with the keyword "IS" (which is optional). For example: IS > 5, IS <= 10.
	When a number of test expressions (separated by commas) are used the CASE statement will be true if any one of these tests evaluates to true.
	If 'value' cannot be matched with a 'testexp' it will be automatically matched to the CASE ELSE. If CASE ELSE is not present the program will not execute any <statements> and continue with the code following the END SELECT.
	When a match is made the <statements> following the CASE statement will be executed until END SELECT or another CASE is encountered when the program will then continue with the code following the END SELECT.
	An unlimited number of CASE statements can be used but there must be only one CASE ELSE and that should be the last before the END SELECT.
	Example:
	`SELECT CASE nbr%`
	`CASE 4, 9, 22, 33 TO 88`
	`statements`
	`CASE IS < 4, IS > 88, 5 TO 8`
	`statements`
	`CASE ELSE`
	`statements`
	`END SELECT`
	Each SELECT CASE must have one and one only matching END SELECT statement. Any number of SELECT…CASE statements can be nested inside the CASE statements of other SELECT…CASE statements.

SERVO SERVO 1 [, freq], 1A [,1B] [,1C] [,1D] SERVO 2 [, freq], 2A [,2B] [,2C] [,2D]	Generate a constant stream of positive going pulses for driving a servo. The Armmite H7 has two servo controllers each being able to control up to four 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', or '2 'is the controller number. 'freq' is the output frequency (between 20Hz and 1000 Hz) and is optional. If not specified it will default to 50 Hz

SERVO channel, STOP	1A, 1B , 1C and 1D are the pulse widths for each of the controller 1 outputs while 2A ,2B, 2C and 2D are the pulse widths for the controller 2 outputs. The specified I/O pins will be automatically configured as outputs while any others will be unaffected and can be used for other duties. The pulse width for each output is independent of the others and is specified in milliseconds, which can be a fractional number (ie, 1.536). For accurate positioning the output resolution is about 0.005 ms. The minimum value is 0.01ms while the maximum is 18.9ms. Most servos will accept a range of 0.8ms to 2.2ms. The output will run continuously in the background while the program is running and can be stopped using the STOP command. The pulse widths of the outputs can be changed at any time (without stoping the output) by issuing a new SERVO command. The SERVO function will take control of any specified outputs and when stopped the pins will be returned to a high impedance "not configured" state.
SETPIN SETPIN pin, cfg [, option]	Will configure an external I/O pin. 'pin' is the I/O pin to configure, 'cfg' is the mode that the pin is to be set to and 'option' is an optional parameter. 'cfg' is a keyword and can be any one of the following: OFF Not configured or inactive AIN Analog input (ie, measure the voltage on the input). 'option' can be used to specify the number of bits in the conversion. Valid values are 8, 10,12 and 14. The default (if not specified) is 16 bits. The more bits the longer the conversion will take. Valid for pins for 144 pin Nucleo are 13, 14, 15, 18, 19, 20, 21, 22, 26, 27, 28, 29 34, 35, 36, 37, 42, 43, 44, 45, 47, 49, 50, 53, 54 Valid for pins 100 Pin WeAct and DEV BOX 15, 16, 17, 18, 22, 23, 24, 25, 32, 33,35, 30, 31 A single conversion takes between 0.2mSec (8-bit) to 0.9mSec (16-bit). DIN Digital input If 'option' is omitted the input will be high impedance If 'option' is the keyword "PULLUP" a simulated resistor will be used to pull up the input pin to 3.3V If the keyword "PULLDOWN" is used the pin will be pulled down to zero volts. The pull up/down is a constant current of about 50µA. 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 digital 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 for 144 pin Nucleo are 10, 11, 12, 13 Valid for pins 100 Pin WeAct and DEV BOX 81, 82, 36, 84 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 for 144 pin Nucleo are 10, 11, 12, 13 Valid for pins 100 Pin WeAct and DEV BOX 81, 82, 36, 84 CIN Counting input Valid for pins for 144 pin Nucleo are 10, 11, 12, 13 Valid for pins 100 Pin WeAct and DEV BOX 81, 82, 36, 84 '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. *The first two count pins will trigger a call to their respective CSUB routines, CFuncInt1 and CFunctInt2 if the pointer to the function is set. This is setup in the CSUB. A call to the CSUB routine would accur on each of the specified edge transitions.* 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 93, [CIN|FIN]	This enables a high speed frequency counter (tested to 20MHz) Period measurement (SETPIN 93, PIN) is not supported on this pin (*Not support at all on the 100 pin chips*)
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 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 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.
SPI SPI OPEN speed, mode, bits or SPI READ nbr, array() or SPI WRITE nbr, data1, data2, data3, … etc or SPI WRITE nbr, string$ or SPI WRITE nbr, array() or SPI CLOSE	Communications via an SPI channel. The command SPI refers to channel 1. The commands SPI2/SPI3 refers to channel 2 and 3 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. *SPI3 is not available on 100 pin devices.* *SPI2 is not available on 100 pin devices if a parallel display is configured.* See Appendix D for the details. **Note:** *The WeAct 100 pin board has an W25Q16 SPI Flash chip wired to SPI (1) and if SPI is used then the F_CS pin 87 (PD6) for the W25Q16 must be set*

	as an output and pulled high to ensure the W25Q16 does not interfer with any other SPI device connected to this SPI.
SPI2	As for SPI but for the second channel. Not available on 100 pin boards if a parallel LCDPANEL is configured.
SPI3	As for SPI but for the third channel. *Not supported on 100 pin boards*
SPRITE SPRITE	The BLIT and SPRITE commands can be used interchangeably. *Internally MMBasic stores SPRITE commands as BLIT commands and a program listing will show BLIT.*
SPRITE x1, y1, x2, y2, w, h BLIT x1, y1, x2, y2, w, h	Copies the memory area specified by top right coordinate x1, y1 and of width w and height h to a new location where the top right coordinate is x2, y2.
SPRITE CLOSE [#]n BLIT CLOSE [#]n	Closes sprite n and releases all memory resources. Updates the screen (see SPRITE HIDE). Sprites which have been “copied” cannot be closed until all “copies” have been closed
SPRITE CLOSE ALL	Closes all sprites and releases all memory resources. It does not change the screen.
SPRITE COPY [#]n, [#]m, nbr	Makes a copy of sprite “n” to “nbr” of new sprites starting a number “m”. Copied sprites share the same loaded image as the original to save memory
SPRITE HIDE [#]n	Removes sprite n from the display and replaces the stored background. To restore a screen to a previous state sprites should be hidden in the opposite order to which they were written "LIFO"
SPRITE INTERRUPT sub	Specifies the name of the subroutine that will be called when a sprite collision occurs. See Appendix C for how to use the function SPRITE to interrogate details of what has collided
SPRITE LOAD [#]n, fname$ [,colour]	Loads the file fname$ as a sprite into buffer number n. The file must be in PNG format RGB888 or RGBA888. If the file extension .PNG is omitted then it will be automatically added. The parameter “colour” specifies the background colour for the sprite. Pixels in the background colour will not overwrite the background when the sprite is displayed. Colour defaults to zero
SPRITE MOVE	Actions a single atomic transaction that re-locates all sprites which have previously had a location change set up using the SPRITE NEXT command. Collisions are detected once all sprites are moved and reported in the same way as from a scroll
SPRITE NEXT [#]n, x, y	Sets the X and Y coordinate of the sprite to be used when the screen is next scrolled or the SPRITE MOVE command is executed. Using SPRITE NEXT rather than SPRITE SHOW allows multiple sprites to be moved as part of the same atomic transaction.
SPRITE NOINTERRUPT	Disables collision interrupts
SPRITE READ [#]n, x , y, w, h BLIT READ [#]n, x , y, w, h	Reads the display area specified by coordinates x and y, width w and height h into buffer number n. If the buffer is already in use and the width and

	height of the new area are the same as the original then the new command will overwrite the stored area.
SPRITE SCROLLH n [,col]	Scrolls the background and any sprites on layer 0 n pixels to the right. n can be any number between -31 and 31. Sprites on any layer other than zero will remain fixed in position on the screen. By default the scroll wraps the image round. If “col” is specified the colour will replace the area behind the scrolled image
SPRITE SCROLLR x, y, w, h, delta_x, delta_y [,col]	Scrolls the region of the screen defined by top-right coordinates “x” and “y” and width and height “w” and “h” by “delta_x” pixels to the right and “delta_y” pixels up. By default the scroll wraps the background image round. If “col” is specified the colour will replace the area behind the scrolled image. Sprites on any layer other than zero will remain fixed in position on the screen. Sprites in layer zero where the centre of the sprite (x+ w/2, y+ h/2) falls within the scrolled region will move with the scroll and wrap round if the centre moves outside one of the boundaries of the scrolled region.
SPRITE SCROLLV n [,col]	Scrolls the background, and any sprites on layer 0, n pixels up. n can be any number between -MM.VRES-1 and MM.VRES-1. Sprites on any layer other than zero will remain fixed in position on the screen. . By default the scroll wraps the image round. If “col” is specified the colour will replace the area behind the scrolled image
SPRITE SHOW [#]n, x,y, layer, [orientation]	Displays sprite n on the screen with the top left at coordinates x, y. Sprites will only collide with other sprites on the same layer, layer zero, or with the screen edge. If a sprite is already displayed on the screen then the SPRITE SHOW command acts to move the sprite to the new location. The display background is stored as part of the command such that it can be replaced when the sprite is hidden or moved further. The orientation is an optional parameter, valid values are: 0 - normal display (default if omitted) 1 - mirrored left to right 2 - mirrored top to bottom 3 - rotated 180 degrees (= 1+2)
SPRITE SWAP [#]n1,[#]n2	Replaces the sprite ‘n1’ with the sprite ‘n2’. The sprites must have the same width and height and ‘n1’ must be displayed or an error will be generated. The replacement sprite inherits the background from the original as well as its position in the list of order drawn
SPRITE WRITE [#]n, x y BLIT WRITE [#]n, x y	Overwrites the display with the contents of sprite buffer n with the top left at coordinates x, y. SPRITE WRITE overwrites the complete area of the display. The background that is overwritten is not stored so SPRITE WRITE is inherently higher performing than SPRITE SHOW but with greater functional limitations.
STATIC 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 STEP	Part of the *FOR x=a TO b STEP c : NEXT* construction See FOR in command section See NEXT in command section
SUB 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
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 inervals.
	SYNC 100,u
	DO
	  SYNC
	  pin(PC2)=1
	  SYNC
	  pin(PC2)=0
	LOOP

TEMPR START 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 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$** TIME$ = "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 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 or fractional floating point number to a resolution of 1µs. See the TIMER function for more details.
TO TO	Part of the *FOR x=a TO b STEP c : NEXT* construction See FOR in command section

	See NEXT in command section
TRACE 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 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.

TURTLE TURTLE RESET	Clears the screen and re-initialises the turtle system. The turtle is located at MM.HRES\2, MM.VRES\2. The default pen colour is white and the fill colour is green. The initial heading is up (0 degrees)
TURTLE DRAW TURTLE	Draws a turtle at the current location and in the current orientation
TURTLE PEN UP	Lifts the pen so moves do not write to the screen
TURTLE PEN DOWN	Lowers the pen so moves do write to the screen
TURTLE FORWARD n	Moves the turtle n pixels in the current heading
TURTLE BACKWARD n	Moves the turtle n pixels opposite the current heading
TURTLE DOT	Draw a 1-pixel dot at the current location, regardless of pen status
TURTLE TURN LEFT deg	Turn the turtle to the left (anti-clockwise) by the specified number of degrees.
TURTLE TURN RIGHT deg	Turn the turtle to the right (clockwise) by the specified number of degrees.
TURTLE BEGIN FILL	Start filling. Call this before drawing a polygon to activate the bookkeeping required to run the filling algorithm later.
TURTLE END FILL	End filling. Call this after drawing a polygon to trigger the fill algorithm. The filled polygon may have up to 128 sides.
TURTLE HEADING deg	Rotate the turtle to the given absolute heading (in degrees). 0 degrees means facing straight up. 90 degrees means facing to the right.
TURTLE PEN COLOUR col	Set the current drawing colour. Colours are specified as per normal Micromite drawing commands
TURTLE FILL COLOUR col	Set the current fill colour. Colours are specified as per normal Micromite drawing commands
TURTLE MOVE x, y	Move the turtle to the specified location, drawing a straight line if the pen is down.

~~TURTLE DRAW PIXEL x, y~~ ~~TURTLE FILL PIXEL x, y~~ ~~TURTLE DRAW LINE x1, y1, x2, y2~~ ~~TURTLE DRAW DIRCLE x, y, r~~	~~Draw a 1-pixel dot at the given location using the current draw colour, regardless of current turtle location or pen status.~~ ~~Draw a 1-pixel dot at the given location using the current fill colour, regardless of current turtle location or pen status.~~ ~~Draw a straight line between the given coordinates, regardless of current turtle location or pen status.~~ ~~Draw a circle at the given coordinates with the given radius, regardless of current turtle location or pen status.~~
VAR VAR SAVE var[,var]…. or VAR RESTORE or VAR CLEAR	VAR SAVE will save one or more variables to non-volatile flash memory where they can be restored later (normally after a power interruption). 'var' can be any number of numeric or string variables and/or arrays. Arrays are specified by using empty brackets. For example: var() VAR RESTORE will retrieve the previously saved variables and insert them (and their values) into the variable table. The VAR SAVE command can be used repeatedly. Variables that had been previously saved will be updated with their new value and any new variables (not previously saved) will be added to the saved list for later restoration. VAR CLEAR will erase all saved variables. Also, the saved variables will be automatically cleared by the NEW command or when a new program is loaded via AUTOSAVE, XMODEM, etc. *See SAVE NVM and LOAD NVM commands for a method of saving128 bytes in RTC battery backed ram. This will persist after the new command.* This command is normally used to save calibration data, options, and other data which does not change often but needs to be retained across a power interruption. Normally the VAR RESTORE command is placed at the start of the program so that previously saved variables are restored and immediately available to the program when it starts. Notes: □ The storage space available to this command is 128KB. □ Using VAR RESTORE without a previous save will have no effect and will not generate an error. □ If, when using RESTORE, a variable with the same name already exists its value will be overwritten. □ Saved arrays must be declared (using DIM) before they can be restored. □ Be aware that string arrays can rapidly use up all the memory allocated to this command. The LENGTH qualifier can be used when a string array is declared to reduce the size of the array (see the DIM command). This is not needed for ordinary string variables.
WATCHDOG 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.
XMODEM XMODEM SEND or XMODEM RECEIVE or XMODEM CRUNCH XMODEM SEND file$ or XMODEM RECEIVE file$	Transfers a BASIC program to or from a remote computer using the XModem protocol. The transfer is done over the serial console connection. XMODEM SEND will send 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 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 ABS(number)	Returns the absolute value of the argument 'number' (ie, any negative sign is removed and the positive number is returned).
ACOS ACOS (number)	Returns the inverse cosine of the argument 'number' in radians.
AND	
AS	
ASC ASC(string$)	Returns the ASCII code for the first letter in the argument ‘string$’.
ASIN ASIN(number)	Returns the inverse sine value of the argument 'number' in radians.
ATAN2 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.
ATAN ATN(number)	Returns the arctangent of the argument 'number' in radians.
**BASE$** BASE$(base, number [, chars])	Returns a string giving the base value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s). Base can be between 2 and 36. Numbers greater than 9 are represented by a letter as per the HEX notation. Example: PRINT BASE$(36, 35) will display "Z" Internally the functions BIN$, OCT$, and HEX$ use this function.
BAUDRATE 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$** BIN$(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). *BIN$ is accepted syntax but is stored internally as the BASE$() function and will list and show in the editor as BASE$(2,x,n) function.*
**BIN2STR$** BIN2STR$(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 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 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 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$** CHR$(number)	Returns a one-character string consisting of the character corresponding to the ASCII code indicated by argument 'number'.
CINT 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 COS(number)	Returns the cosine of the argument 'number' in radians.
CTRLVAL 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$** CWD$	Returns the current working directory on the SD card as a string. The format is: `A:/dir1/dir2`.
**DATE$** DATE$	Reads the RTC and returns the date as a string in the form "dd-mm-yyyy"
**DATETIME$** DATETIME$(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$(NOW)
**DAY$** 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 DEG(radians)	Converts 'radians' to degrees.

DIR$
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 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 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 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 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 EXP(number)	Returns the exponential value of 'number', ie, e^x where x is 'number'.

FIELD$
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 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$** FORMAT$(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 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 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)	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

GPS(SATELLITES) GPS(SPEED) GPS(TIME) GPS(TRACK) GPS(VALID)	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 $GPRMC strings.
**HEX$** HEX$(number [, chars])	Returns a string giving the hexadecimal (base 16) value for the 'number'. 'chars' is optional and specifies the number of characters in the string with zero as the leading padding character(s). *HEX$ is accepted syntax but is stored internally as the BASE$() function and will list and show in the editor as the BASE$(16,x,n) function*
**INKEY$** INKEY$	Checks the console input buffer and, if there is one or more characters waiting in the queue, will remove the first character and return it as a single character in a string. If this is a carriage return, it is likely that there will be a line feed character following as often the enter key will produce a CR/LF pair. If the input buffer is empty this function will immediately return with an empty string (ie, "").
**INPUT$** INPUT$(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 INSTR([start-pos,]searched$, pattern$[,size])	Returns the position at which 'pattern$' occurs in 'searched$', beginning at 'start-pos'. Both the position returned and 'start-pos' use 1 for the first character, 2 for the second, etc. The function returns zero if 'pattern$' is not found. If the optional size parameter is specified the firmware interprets the search string as a regular expression. The size parameter is a floatingpoint variable that is used by the firmware to return the size of a matching string. If the variable doesn't exist it is created. See Appendix H – Regular Expressions in (L)INSTR for details.
INT 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$** JSON$(array%(), string$)	Returns a string representing a specific item out of the JSON input stored in the longstring array%() Examples taken from api.openweathermap.org JSON$(a%(), “name”) JSON$(a%(), “coord.lat”) JSON$(a%(), “weather[0].description”) JSON$(a%(),”list[4].weather[0].description
KEYDOWN(n) KEYDOWN(n)	Returns the decimal ASCII value of the USB keyboard key that is currently held down or zero if no key is down. The decimal values for the function and arrow keys are listed in Appendix G. This function will report multiple simultaneous key presses and the parameter 'n' is the number of the keypress to report. KEYDOWN(0) will return the number of keys being pressed. For example, if "c", "g" and "p" are pressed simultaneously KEYDOWN(0) will return 3, KEYDOWN(1) will return 99, KEYDOWN(2) will return 103, etc. The keys do not need to be pressed simultaneously and will report in the order pressed. Taking a finger off a key will promote the next key pressed to #1. The first key ('n' = 1) is entered in the keyboard buffer (accessible using INKEY$) while keys 2 to 6 can only be accessed via this function. Using this function will clear the console input buffer. KEYDOWN(7) will give any modifier keys that are pressed. These keys do not add to the count in keydown(0) The return value is a bitmask as follows: lalt = 1, lctrl = 2, lgui = 4, lshift = 8, ralt = 16, rctrl = 32, rgui = 64, rshift = 128 KEYDOWN(8) will give the current status of the lock keys. These keys do not add to the count in keydown(0). The return value is a bitmask as follows: caps_lock = 1, num_lock = 2, scroll_lock = 4 Note that some keyboards will limit the number of active keys that they can report on. **Note:** *KEYDOWN() only works with a connected USB keyboard. See OPTION USBKEYBOARD. It will not detect keys pressed on a connected serial console. See command ON KEY to detect keys on a serial console.*
**LCASE$** LCASE$(string$)	Returns ‘string$’ converted to lowercase characters.
LCOMPARE 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$** LEFT$(string$, nbr)	Returns a substring of ‘string$’ with ‘nbr' of characters from the left (beginning) of the string.
LEN LEN(string$)	Returns the number of characters in 'string$'.

LGETBYTE 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$** LGETSTR$(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 LINSTR(array%(), search$ [,start][,size])	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. If the optional size parameter is specified the firmware interprets the search string as a regular expression. The size parameter is a floatingpoint variable that is used by the firmware to return the size of a matching string. If the variable doesn't exist it is created. See Appendix H – Regular Expressions in (L)INSTR for details.
LLEN LLEN(array%())	Returns the length of a long string stored in array%()
LOC 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 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 LOG(number)	Returns the natural logarithm of the argument 'number'.
MATH MATH **Simple functions** MATH(ATAN3 x,y) MATH(COSH a) MATH(LOG10 a) MATH(SINH a)	The math function performs many simple mathematical calculations that can be programmed in Basic but there are speed advantages to coding looping structures in C and there is the advantage that once debugged they are there for everyone without re-inventing the wheel. Returns ATAN3 of x and y Returns the hyperbolic cosine of a Returns the base 10 logarithm of a Returns the hyperbolic sine of a

MATH(TANH a)	Returns the hyperbolic tan of a
Simple Statistics	
MATH(CHI a())	Returns the Pearson's chi-squared value of the two dimensional array a())
MATH(CHI_p a())	Returns the associated probability in % of the Pearson's chi-squared value of the two dimensional array a())
MATH(CORREL a(), 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 standard deviation of all values in the a() array, a() can have any number of dimensions
MATH(SUM a())	Returns the sum of all values in the a() array, a() can have any number of dimensions
Vector Arithmetic MATH(MAGNITUDE v())	Returns the magnitude of the vector v(). The vector can have any number of elements
MATH(DOTPRODUCT v1(), v2()) MATH(M_DETERMINANT array!())	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 CRC 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, CRC12, 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$,9,,&H1021) https://www.thebackshed.com/forum/ViewTopic.php?FID=16&TID=15405 See Appendix I – Cyclic Redundancy Check (CRC)
MAX MAX(arg1 [, arg2 [, …]]) or **MIN** MIN(arg1 [, arg2 [, …]])	Returns the maximum or minimum number in the argument list. Note that the comparison is a floating point comparison (integer arguments are converted to floats) and a float is returned.
**MID$** MID$(string$, start) or MID$(string$, start, nbr)	Returns a substring of ‘string$’ beginning at ‘start’ and continuing for ‘nbr’ characters. The first character in the string is number 1. If ‘nbr’ is omitted the returned string will extend to the end of ‘string$’
MOVEMENT MOVEMENT(sensitivity)	This function combines capturing a new image from an OV7670 camera with a comparison with the currently displayed image. It returns the number of pixels that are different between the images. The sensitivity parameter controls whether or not a pixel is considered to be different. The algorithm works by converting each image to a 6-bit greyscale image. The sensitivity is then the difference between the pixels that will be counted as a change. The top 20 and bottom 20 rows are ignored in the comparison. This allows the user to use these for other display purposes without artificially triggering the movement detection. *Not supported on 100 pin devices.*
MSGBOX 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$** OCT$(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). *OCT$ is accepted syntax but is stored internally as the BASE$() function and will list and show in the editor as the BASE$(8,x,n) function*

OWSEARCH OWSEARCH(pin, flag [,serial_number_mask])	Returns the onewire device serial number as an INTEGER. Flags are: 0 - Continue an existing search 1 - start a new search 4 - Continue an existing search for devices in the requested family 5 - start a new search for devices in the requested family (the MSB of the serial_number_mask) 8 - skip the current device family and return the next device 16 - verify that the device with the serial number in serial_number_mask is available
PEEK 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)	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.
PI PI	Returns the value of pi.
PIN PIN(pin)	Returns the value on the external I/O 'pin'. Zero means digital low, 1 means digital high and for analog inputs it will return the measured voltage as a floating point number. Frequency inputs will return the frequency in Hz. A period input will return the period in milliseconds while a count input will return the count since reset (counting is done on the positive rising edge). The count input can be reset to zero by resetting the pin to counting input (even if it is already so configured). This function will also return the state of a pin configured as an output. Also see the SETPIN and PIN() = commands.
PIN(function)	Returns the value of a special function. 'function' is a string (ie, it can be a string variable or string constant). For example PRINT PIN(BAT). It can be one of: "BAT" The voltage of the backup battery. "TEMP" The temperature of the STM32 processor's core. "DAC1" The output voltage on DAC1(on audio output). "DAC2" The output voltage on DAC2(on audio output).

	"SREF" The stored calibrated value of the internal reference voltage measured with a supply of exactly 3.3V. This is programmed into the chip during production. "IREF" The measured value of the internal reference voltage. The actual value of VREF+ can be calculated as: 3.3 * PIN("SREF") / PIN("IREF") and this can be used to set OPTION VCC. *Once Option VCC is set to 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 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 Graphic Commands and Functions for a definition of the colours and graphics coordinates.
PORT 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 POS	See Obsolete Commands and Functions section. For the console returns the position of the cursor on the current line. Use MM.INFO$(
PULSIN 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 RAD(degrees)	Converts 'degrees' to radians.
RGB 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.

RGB(shortcut [, trans])	'shortcut' allows common colours to be specified by naming them. The colours that can be named are white, black, blue, green, cyan, red, magenta, yellow, brown and gray. For example, RGB(red) or RGB(cyan). 'trans' is the level of transparency for colour depths 4 and 12. It is optional and defaults to 15 if not specified.
**RIGHT$** RIGHT$(string$, number-of-chars)	Returns a substring of 'string$' with 'number-of-chars' from the right (end) of the string.
RND 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 H7 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 SGN(number)	Returns the sign of the argument 'number', +1 for positive numbers, 0 for 0, and -1 for negative numbers.
SIN SIN(number)	Returns the sine of the argument 'number' in radians.
**SPACE$** SPACE$(number)	Returns a string of blank spaces 'number' characters long.
SPI SPI(data) or SPI2(data) or SPI3(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. SPI3 not supported on 100 pin devices. SPI2 not available on 100 pin devices if a parallel display is enabled.
SPRITE SPRITE(C, [#]n)	Returns the number of currently active collisions for sprite n. If n=0 then returns the number of sprites that have a currently active collision following a SPRITE SCROLL command
SPRITE(C, [#]n, m)	Returns the number of the sprite which caused the "m"th collision of sprite n. If n=0 then returns the sprite number of "m"th sprite that has a currently active collision following a SPRITE SCROLL command
SPRITE(H,[#]n)	Returns the height of sprite n. This function is active whether or not the sprite is currently displayed (active).
SPRITE(L, [#]n)	Returns the layer number of active sprites number n
SPRITE(N)	Returns the number of displayed (active) sprites
SPRITE(N,n)	Returns the number of displayed (active) sprites on layer n
SPRITE(S)	Returns the number of the sprite which last caused a collision. NB if the number returned is Zero then the collision is the result of a SPRITE SCROLL command and the SPRITE(C…) function should be used to find how many and which sprites collided. Returns the width of sprite n. This function is active whether or not the sprite

SPRITE(W, [#]n)	is currently displayed (active).
SPRITE(X, [#]n)	Returns the X-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise.
SPRITE(Y, [#]n)	Returns the Y-coordinate of sprite n. This function is only active when the sprite is currently displayed (active). Returns 10000 otherwise
SQR SQR(number)	Returns the square root of number.
STR2BIN STR2BIN(type, string$ [,BIG])	Returns a number equal to the binary representation in 'string$'. 'type' can be: INT64 converts 8 byte string representing a signed 64-bit integer to an integer UINT64 converts 8 byte string representing an unsigned 64-bit integer to an integer INT32 converts 4 byte string representing a signed 32-bit integer to an integer UINT32 converts 4 byte string representing an unsigned 32-bit integer to an integer INT16 converts 2 byte string representing a signed 16-bit integer to an integer UINT16 converts 2 byte string representing an unsigned 16-bit integer to an integer INT8 converts 1 byte string representing a signed 8-bit integer to an integer UINT8 converts 1 byte string representing an unsigned 8-bit integer to an integer SINGLE converts 4 byte string representing single precision float to a float DOUBLE converts 8 byte string representing single precision float to a float By default the string must contain the number in little-endian format (ie, the least significant byte is the first one in the string). Setting the third parameter to 'BIG' will interpret the string in big-endian format (ie, the most significant byte is the first one in the string). This function makes it easy to read data from binary data files, interpret numbers from sensors or efficiently read binary data from flash memory chips. An error will be generated if the string is the incorrect length for the conversion requested See also the function BIN2STR$
**STR$** STR$(number) or STR$(number, m) or STR$(number, m, n) or STR$(number, m, n, c$)	Returns a formatted string in decimal (base 10) representation of 'number'. If 'm' is specified sufficient spaces will be added to the start of the number to ensure that the number of characters before the decimal point (including the negative or positive sign) will be at least 'm' characters. If 'm' is zero or the number has more than 'm' significant digits no padding spaces will be added. If 'm' is negative, positive numbers will be prefixed with the plus symbol and negative numbers with the negative symbol. If 'm' is positive then only the negative symbol will be used. 'n' is the number of digits required to follow the decimal place. If it is zero the string will be returned without the decimal point. If it is negative the output will always use the exponential format with 'n' digits resolution. If 'n' is not specified the number of decimal places and output format will vary automatically according to the number. 'c$' is a string and if specified the first character of this string will be used as the padding character instead of a space (see the 'm' argument). Examples: STR$(123.456) will return "123.456" STR$(-123.456) will return "-123.456" STR$(123.456, 1) will return "123.456" STR$(123.456, -1) will return "+123.456"

	STR$(123.456, 6) will return " 123.456" STR$(123.456, -6) will return " +123.456" STR$(-123.456, 6) will return " -123.456" STR$(-123.456, 6, 5) will return " -123.45600" STR$(-123.456, 6, -5) will return " -1.23456e+02" STR$(53, 6) will return " 53" STR$(53, 6, 2) will return " 53.00" STR$(53, 6, 2, "*") will return "****53.00"
**STRING$** STRING$(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 TAB(number)	Outputs spaces until the column indicated by 'number' has been reached on the console output.
TAN TAN(number)	Returns the tangent of the argument 'number' in radians.
TEMPR 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.
**TIME$** TIME$	Returns the current time based on MMBasic's internal clock as a string in the form "HH:MM:SS" in 24 hour notation. For example, "14:30:00". If the OPTION MILLISECONDS ON command has been used this function will return the time including milliseconds as a decimal fraction of the seconds. For example: "14:35:06.239". To set the current time use the command TIME$ = .
TIMER 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 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$** UCASE$(string$)	Returns ‘string$’ converted to uppercase characters.
VAL 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 GOSUB target	Initiates a subroutine call to the target, which can be a line number or a label. The subroutine must end with RETURN. New programs should use defined subroutines (ie, SUB…END SUB).
IF condition THEN linenbr	For Microsoft compatibility a GOTO is assumed if the THEN statement is followed by a number. A label is invalid in this construct. New programs should use: IF condition THEN GOTO linenbr | label
IRETURN IRETURN	Returns from an interrupt when the interrupt destination was a line number or a label. New programs should use a user defined subroutine as an interrupt destination. In that case END SUB or EXIT SUB will cause a return from the interrupt.
ON nbr GOTO | GOSUB target[,target, target,...]	ON either branches (GOTO) or calls a subroutine (GOSUB) based on the rounded value of 'nbr'; if it is 1, the first target is called, if 2, the second target is called, etc. Target can be a line number or a label. New programs should use SELECT CASE.
POS POS	For the console, returns the current cursor position in the line in characters.
RETURN 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 Draft 0	Initial Draft of consolidated manual for the 100 pin and 144 pin Armmite H7. Incorporates information from the original Armmite H7 supplementary manual. See TBS thread for details of firmware changes .

Appendix A – Serial Communications

Four serial ports are available for asynchronous serial communications labelled COM1: COM2: COM3: and COM4: The serial console uses a separate port and does not inpact these ports.

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

The following is an example:

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

The OPEN Command

A serial port is opened using the command:

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

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

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

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

- 'n' is the serial port number for either COM1:, COM2:, COM3: or COM4:.
- 'baud' is the baud rate. This can be any value between 1200 (the minimum) and 1MHz. Default is 9600.
- 'buf' is the receive buffer size in bytes (default size is 256). The transmit buffer is fixed at 256 bytes.
- 'int' is a user defined subroutine which will be called when the serial port has received some data. The default is no interrupt.
- 'int-trigger' sets the trigger condition for calling the interrupt subroutine. It is an integer and the interrupt subroutine will be called when this number of characters has arrived in the receive queue.

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

The following options can be added to the end of 'comspec$'

- 'INV' specifies that the transmit and receive polarity is inverted. Default is non inverted.
- 'OC' will force the transmit pin to be open collector. The default is normal (0 to 3.3V) output.
- 'S2' specifies that two stop bits will be sent following each character transmitted. Default is one stop bit.
- '7BIT' will specify that 7 bit transmit and receive is to be used. Default is 8 bits.
- 'ODD' will specify that an odd parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)
- 'EVEN' will specify that an even parity bit will be appended (8 bits will be transmitted if 7BIT is specified, otherwise 9)
- 'DEP' will enable RS485 mode with a positive output on the COM2-DE pin
- 'DEN' will enable RS485 mode with a negative output on the COM2-DE pin

Input/Output Pin Allocation

When a serial port is opened the pins used by the port will be automatically set to input or output as required and the SETPIN and PIN commands will be disabled for the pins. When the port is closed (using the CLOSE command) all pins used by the serial port will be set to a not-configured state and the SETPIN command can then be used to reconfigure them.

The connections for each COM port are shown in the I/O connector pinout diagrams in the beginning of this manual. Note that Tx means an output from the 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 100K and its purpose is to prevent the input from floating if it is left unconnected. Normally this is fine but it can cause a problem if you have an external resistor in series with the Rx pin, in that case this resistor and the pullup resistor will form a voltage divider limiting how high or low the voltage on the Rx pin can swing and that in turn might mean that the input signal is not recognised. The solution is to use the command SERIAL PULLUP DISABLE to disable it.

Examples

Opening a serial port using all the defaults:

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

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

```
OPEN "COM2:4800" AS #1
```

Opening a serial port specifying the baud rate (9600 bits per second) and receive buffer size (1KB):

```
OPEN "COM1:9600, 1024" AS #8
```

The same as above but with two stop bits enabled:

```
OPEN "COM1:9600, 1024, S2" AS #8
```

An example specifying everything including an interrupt, an interrupt level,inverted and two stop bits:

```
OPEN "COM1:19200, 1024, ComIntLabel, 256, INV, S2" AS #5
```

Reading and Writing

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

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

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

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

Serial ports can be closed with the CLOSE command. This will wait for the transmit buffer to be emptied then free up the memory used by the buffers, cancel the interrupt (if set) and set all pins used by the port to the not configured state. A serial port is also automatically closed when commands such as RUN and NEW are issued.

Interrupts

The interrupt subroutine (if specified) will operate the same as a general interrupt on an external I/O pin.

When using interrupts you need to be aware that it will take some time for MMBasic to respond to the interrupt and more characters could have arrived in the meantime, especially at high baud rates. For example, if you have specified the interrupt level as 250 characters and a buffer of 256 characters then quite easily the buffer will have overflowed by the time the interrupt subroutine can read the data. In this case the buffer should be increased to 512 characters or more.

Low Cost RS-232 Interface

The RS-232 signalling system is used by modems, hardwired serial ports on a PC, test equipment, etc. It is the same as the serial TTL system used on the Armmite H7 with two exceptions:

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

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

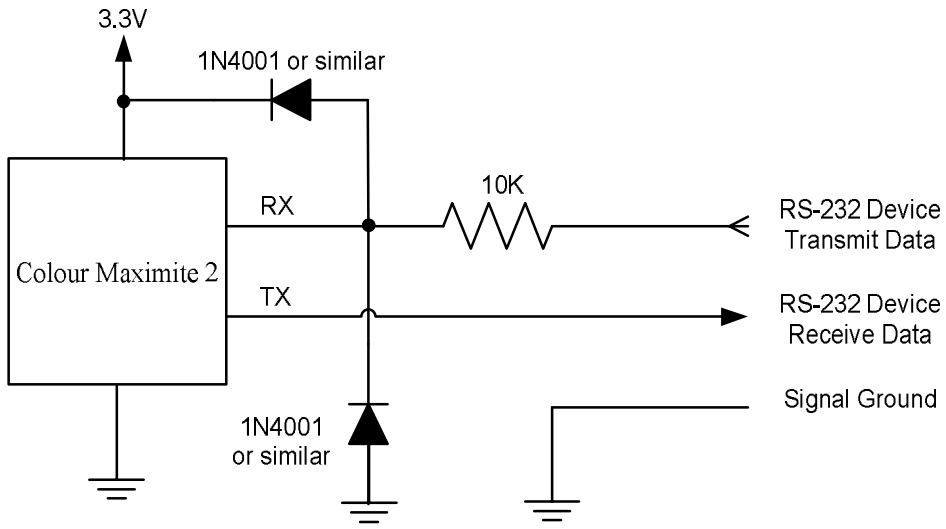
The first issue is that the signalling polarity is inverted with respect to TTL. On the Colour Maximite 2 COM1: can be specified to invert the transmit and receive signal (the 'INV' option) so that is an easy fix.

For the receive data (that is the ±12V signal from the remote RS-232 device) it is easy to limit the voltage using a series resistor of (say) 10KΩ and two diodes that will clamp the input voltage to the 3.3V rail and ground. The input impedance of the I/O pin is very high so the resistor will not cause a voltage drop but it does mean that when the signal swings to the maximum ±12V it will be safely clipped by the diodes.

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

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

Use this circuit:



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

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

Appendix B – I2C Communications

The Armmite H7 implements two I²C channels. All operate in master mode (slave mode is not available).

There following commands can be used:

I2C OPEN speed, timeout

Enables the I²C module in master mode. The I2C command refers to channel 1, use command I2C2 to refer to channels 2 using the same syntax.

'speed' is the clock speed (in KHz) to use and must be one of 100, 400 or 1000.

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

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

Send data to the 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

This command can be used to check the existance of a device at the nominated address. It sets the MM.I2C variable to 0 if the device responds and 1 if not. This can be useful as a diagnostic. e.g. I2C CHECK &H27 : PRINT MM.I2C

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

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.

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)
```

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 and SPI3 refers to channels 2 and 3.

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

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

SPI Open

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

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

Where:

- ‘speed’ is the speed of the clock. This can be 25000000, 12500000, 6250000, 3150000, 1562500, 781250, 390625 or 195315 (ie, 25MHz, 12.5MHz, 6.25MHz, 3.125MHz, 1562.5KHz, 781.25KHz, 390.625KHz, or 195.3125KHz). 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 ,16 or 32 for the Armite H7
- It is the responsibility of the program to separately manipulate the CS (chip select) pin if required.

Transmission Format

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

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

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

Standard Send/Receive

When the SPI channel is open data can be sent and received using the SPI function. The syntax is:

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

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

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

Bulk Send/Receive

Data can also be sent in bulk:

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

or

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

or

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

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

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

Data can also be received in bulk:

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

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

SPI Close

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

```
SPI CLOSE
```

Examples

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

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

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

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

WeAct 100 Pin and SPI

The WeAct 100 pin board has an W25Q16 SPI Flash chip wired to SPI (1) and if SPI is used then the F_CS pin 87 (PD6) for the W25Q16 must be set as an output and pulled high to ensure the W25Q16 does not interfer with any other SPI device connected to this SPI.

Appendix E - W25Q Windbond

‘ Flash test if W25Q connected.

```
OPTION BASE 1
DIM AS INTEGER F_CS = 87    'WeAct 100 pin
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 - Sprites

The concept of the sprite implementation is as follows:

- Sprites are full colour and of any size. The collision boundary is the enclosing rectangle.
- Sprites are loaded to a specific number (1 to 64).
- Sprites are displayed using the SPRITE SHOW command.
- For each SHOW command the user must select a "layer". This can be between 0 and 10.
- Sprites collide with sprites on the same layer, layer 0, or the screen edge.
- Layer 0 is a special case and sprites on all other layers will collide with it.
- The SCROLL commands leave sprites on all layers except layer 0 unmoved.
- Layer 0 sprites scroll with the background and this can cause collisions.
- There is no practical limit on the number of collisions caused by SHOW or SCROLL commands.
- The sprite function allows the user to fully interrogate the details of a collision.
- A SHOW command will overwrite the details of any previous collisions for that sprite.
- A SCROLL command will overwrite details of previous collisions for ALL sprites.
- To restore a screen to a previous state sprites should be removed in the opposite order to which they were written (ie, last in first out).

Because moving a sprite or, particularly, scrolling the background can cause multiple sprite collisions it is important to understand how they can be interrogated.

The best way to deal with a sprite collision is using the interrupt facility. A collision interrupt routine is set up using the SPRITE INTERRUPT command. Eg:

```
SPRITE INTERRUPT  collision
```

The following is an example program for identifying all collisions that have resulted from either a SPRITE SHOW command or a SCROLL command

```
'
' This routine demonstrates a complete interrogation of collisions
'
SUB collision
  LOCAL INTEGER i
' First use the SPRITE(S) function to see what caused the interrupt
  IF SPRITE(S) <> 0 THEN 'collision of specific individual sprite
    'SPRITE(S) returns the sprite that moved to cause the collision
    PRINT "Collision on sprite ", SPRITE(S)
    process_collision(SPRITE(S))
    PRINT
  ELSE     '0 means collision of one or more sprites caused by background move
           ' SPRITE(C, 0) will tell us how many sprites had a collision
    PRINT "Scroll caused a total of ", SPRITE(C,0)," sprites to have collisions"
    FOR I = 1 TO SPRITE(C, 0)
      ' SPRITE(C, 0, i) will tell us the sprite number of the "I"th sprite
      PRINT "Sprite ", SPRITE(C, 0, i)
      process_collision(SPRITE(C, 0, i))
    NEXT i
    PRINT
  ENDIF
END SUB

' get details of the specific collisions for a given sprite
SUB process_collision(S AS INTEGER)
```

```
  LOCAL INTEGER i, j
  ' SPRITE(C, #n) returns the number of current collisions for sprite n
  PRINT "Total of " SPRITE(C, S) " collisions"
   FOR I = 1 TO SPRITE(C, S)
    ' SPRITE(C, S, i) will tell us the sprite number of the "I"th sprite
    j = SPRITE(C, S, i)
    IF j = &HF1 THEN
      PRINT "collision with left of screen"
    ELSE IF j = &HF2 THEN
      PRINT "collision with top of screen"
    ELSE IF j = &HF4 THEN
      PRINT "collision with right of screen"
    ELSE IF j = &HF8 THEN
      PRINT "collision with bottom of screen"
    ELSE
      ' SPRITE(C, #n, #m) returns details of the mth collision
      PRINT "Collision with sprite ", SPRITE(C, S, i)
    ENDIF
  NEXT i
END SUB
```

Appendix G – Sensor Fusion

Type can be MAHONY or MADGWICK
Ax, ay, and az are the accelerations in the three directions and should be specified in units of standard gravitational acceleration.
Gx, gy, and gz are the instantaneous values of rotational speed which should be specified in radians per second.
Mx, my, and mz are the magnetic fields in the three directions and should be specified in nano-Tesla (nT)

Care must be taken to ensure that the x, y and z components are consistent between the three inputs. So , for example, using the MPU-9250 the correct input will be ax, ay,az, gx, gy, gz, **my, mx, -mz** based on the reading from the sensor.

Pitch, roll and yaw should be floating point variables and will contain the outputs from the sensor fusion.

The SENSORFUSION routine will automatically measure the time between consecutive calls and will use this in its internal calculations.

The Madwick algorithm takes an optional parameter p1. This is used as beta in the calculation. It defaults to 0.5 if not specified

The Mahony algorithm takes two optional parameters p1, and p2. These are used as Kp and Ki in the calculation. If not specified these default to 10.0 and 0.0 respectively.

A fully worked example of using the code is given on the BackShed forum at

http://www.thebackshed.com/forum/forum_posts.asp?TID=9321&PN=1&TPN=1

Appendix H – Regular Expressions in (L)INSTR

The alternate forms of the INSTR() and LINSTR() functions can take a regular expression as the search pattern.
The alternate form of the functions are:

INSTR([start],text$, search$ [,size])

LINSTR(text%(),search$ [,start] [,size])

In both cases specifying the size parameter causes the firmware to interpret the search string as a regular expression. The size parameter is a floatingpoint variable that is used by the firmware to return the size of a matching string. If the variable doesn't exist it is created. As implemented in MMBasic you need to apply the returned ***start*** and ***size*** values to the **MID$** function to extract the matched string. e.g.

IF start THEN match$=MID$(text$,start,size) ELSE match$="" ENDIF

The syntax of regular expressions can vary slightly with the various implementations. This appendix is a summary of the syntax and supported operations used in the MMBasic implementation on the ArmmiteH7

Anchors		**Quantifiers**		**Groups and Ranges**
^ Start of string		* 0 or more		(a|b) a or b
$ End of string		+ 1 or more		(…) group
~~\b Word Boundary~~		? 0 or 1		[abc] Range (a or b or c)
~~\B Not a word boundary~~		{3} Exactly 3		[^abc] Not (a or b or c]
~~\< Start of word~~		{3,} 3 or more		[a-q] lower case letters a to q
~~\> End of word~~		{3,5\ 3,4 or 5		[A-Q] upper case letters A to Q
				[0-7] Digits from 0 to 7
Escapes required to match Special Characters		**Escapes with special functions**		**Character Classes**
\^ to match ^ (caret)		| or See Groups and Ranges		\w digits and letters plus _
\. to match . (dot)				\W non -alpha
* to match * (asterix)				\s white space \t \f \r \n \v spaces
\$ to match $ (dollar)				\S non white space
\[to match [(left bracket)				\d digits
\\ to match \ (backslash)				\D non digits
\? to match ? (questmark)				\xXX hex encoded byte
\(to match (				
\) to match)				**Matching rules**
\+ to match + (plus)				non special chars match themselves
				. (dot) matches any character

Example expression to match an IP Address.

"[\d]+\.[\d]+\.[\d]+\.[\d]+"

Appendix I – 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 J – 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)
DEL	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
PrtScr/SysRq	9D	157
PAUSE/BREAK	9E	158
SHIFT_TAB	9F	159
SHIFT_DEL	A0	160
SHIFT_DOWN_ARROW	A1	161
SHIFT_RIGHT_ARROW	A3	163

If the shift key is simultaneously pressed then 40 (hex) is added to the code (this is the equivalent of setting bit 6). For example Shift-F10 will generate DA (hex).

The shift modifier only works with the function keys F1 to F12; it is ignored for the other keys except TAB, DEL, DOWN_ARROW and RIGHT_ARROW as identified above.

MMBasic will translate most VT100 escape codes generated by terminal emulators such as Tera Term and Putty to these codes (excluding the shift and control modifiers). This means that a terminal emulator operating over a USB or a serial port opened as console will generate the same key codes as a directly attached keyboard.

Appendix K – Loading the Firmware (100 pin boards)

Using STM32CubeProgrammer to Update Firmware

The STM32 processor includes its own programmer/bootloader when in DFU (Device Firmware Update) mode, so the Armmite H7 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).

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 WeAct and DevEBox STM32F743VIT6 do not. The USB 5v and the 5v pins on the board are connected together. 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 are using external 5V power you should remove it during a firmware update. The board will be powered by the USB connection during the firmware update.

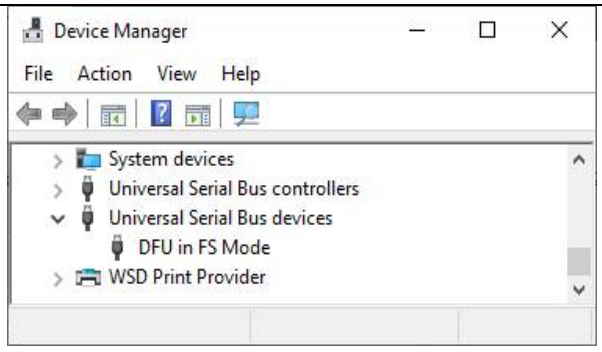
Connect to the Device in DFU mode.

Connect the BT0 pin on the board to 3.3V On the WeAct board this can be achieved by holding the key marked B0. On the devEBox board use a jumper or provide you own external switch/key.

Using a USB cable connect the USB port on the WeAct/DevEBox 100 pin board to a USB port on your desktop computer. This will power up the development board while BT0 is connected to 3.3V. This will place the device in DFU mode.

In Device Manager you will see it under Universal Serial Bus devices.

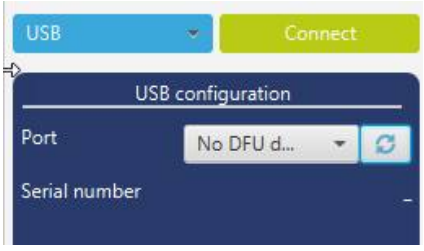
When STMCubeProgrammer is run it will use this connection to upload the firmware.

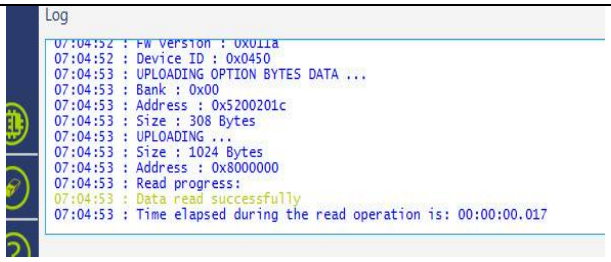
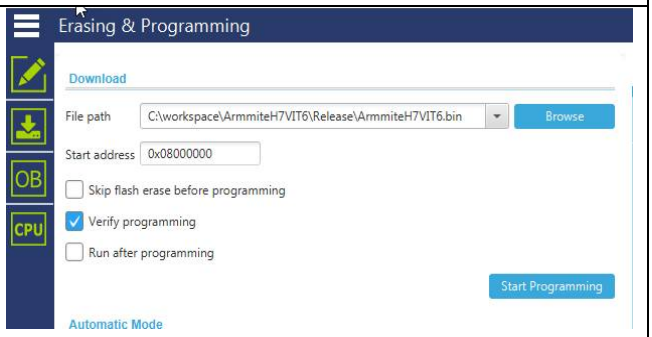


Connect STMCubeProgrammer and load Firmware.

Run the STM32CubeProgrammer software on your computer. On the top right of the program window select USB as the communications method.

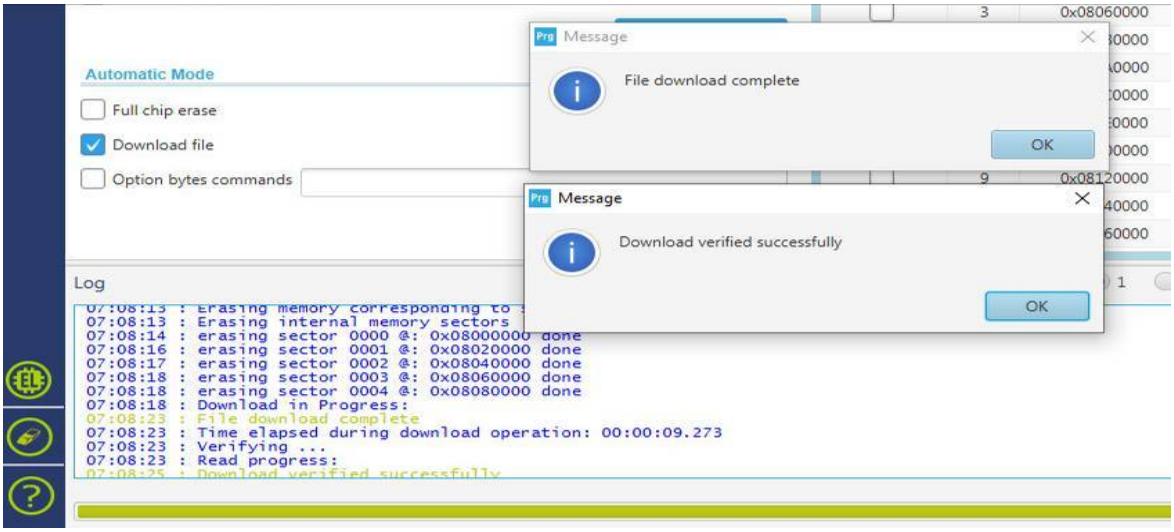
If the program does not recognise the USB connection (No DFU) click on the small blue circle to the right of the Port drop down list to refresh the entry.



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 STM32H743 CPU. 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.



Dismiss both dialog boxes and disconnect the STM32CubeProgrammer software, using the Disconnect button.

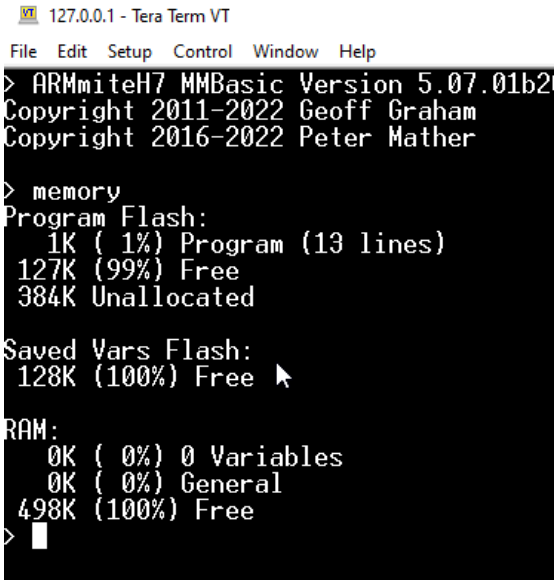
Connect to the MMBasic Console.

The MMBasic console is a separate connection to the USB connection used to load the firmware. You may still use it to provide power or remove it and reconnect the external 5V if it is being used.

The console is a separate serial connection that connects to the serial console which appears on pin 55 (PD8) for Tx and pin 56 (PD9) for Rx.

It is unlikely your modern computer will have an actual serial port. The serial port is achieved using a USB to Serial converter. The default console speed is 115200 bauds.

Connect a terminal emulator to the correct COM port, restart the STM32H743 by power off/power on or pressing the Reset button and you should see the Armmite copyright banner.



Appendix L – Loading the Firmware (Nuclo 144 pin)

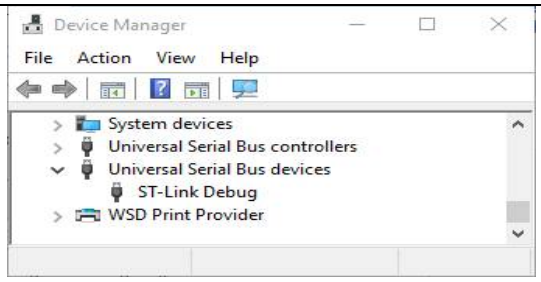
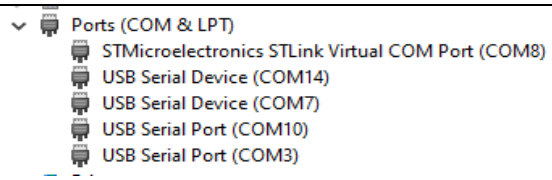
Using STM32CubeProgrammer to Update Firmware

The STM32 processor includes its own programmer/bootloader when in DFU (Device Firmware Update) mode, so the Armmite H7 Nucleo firmware can be easily loaded via USB using a personal computer or laptop connected to the ST-LINK Micro USB connector. Just follow these steps.

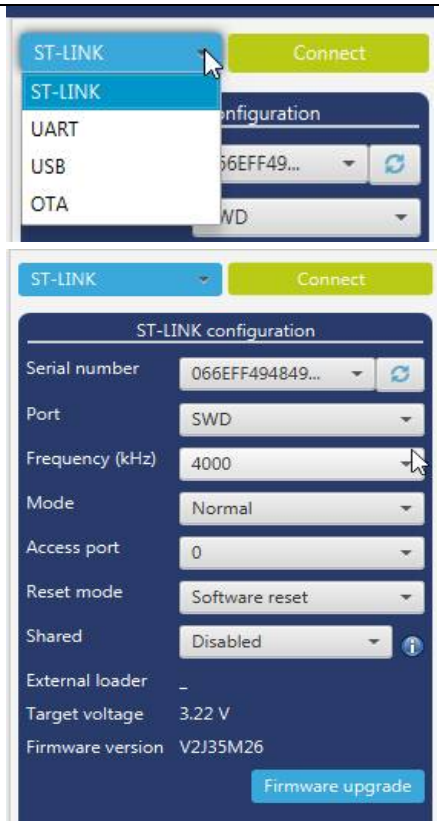
Go to https://www.st.com/en/development-tools/stm32cubeprog.html and download the STM32CubeProgrammer software. This is free software but STM do require you to have an STM account or provide your name and email address. They will email you a link to download the software. Then install this software on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

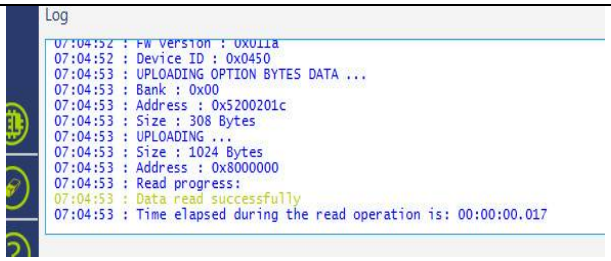
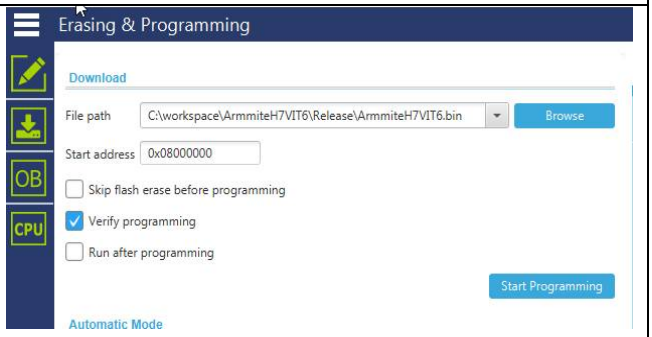
Connect to the Device (Nucleo 144 pin board with ST-LINK)

The Nucleo development board has an ST-LINK module attatched as part of the development board. The connection for programming the STM32H743 is via the Micro USB connector on the attached ST-LINK board. Using a USB cable connect this USB port on the ST-LINK to a USB port on your desktop computer.

The ST-Link should be seen on the PC with Device Manager in two places. This one under Universal Serial Bus devices is used by STM32CubeProgrammer to access the STM32H743 in DFU mode to load the firmware.	
This one under Ports (COM & LPT) will provide the COM port for the connection to the MMBasic console after the firmware is loaded, STMCubeProgrammer is disconnected and the STM32H743 retarted.	

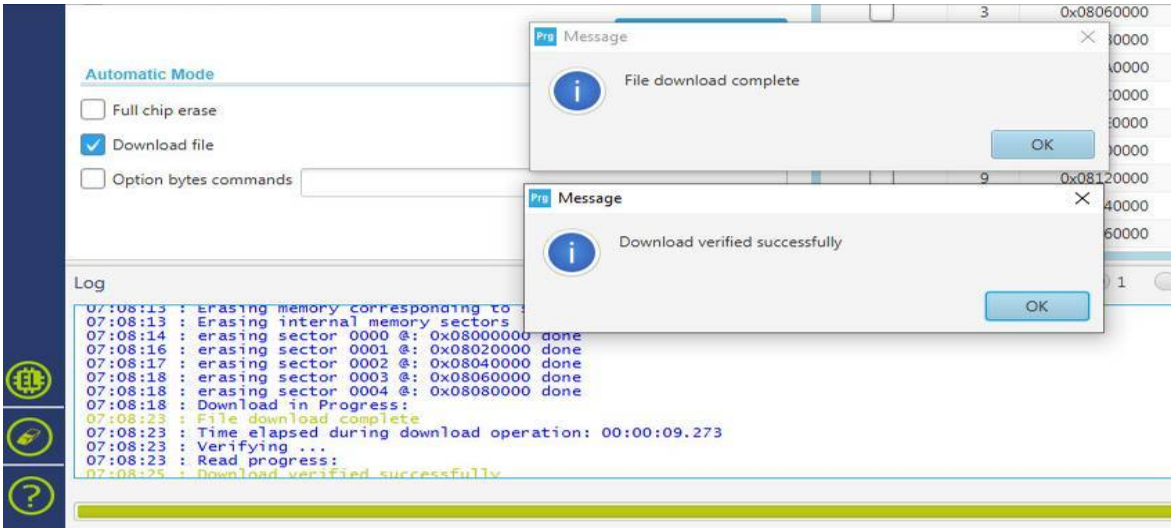
Connect STMCubeProgrammer and load Firmware.

Run the STM32CubeProgrammer software on your computer. On the top right of the program window select ST-LINK as the communications method. Your screen should look like the illustration on the right showing details of the ST-Link configuration. ***The Firmware Upgrade button is to update the ST-Link firmware. It is NOT for loading the firmware into the STM32H743.*** See ST-LINK V2/V3 Details and Update before you attempt to upgrade the ST-LINK firmware.	

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.	

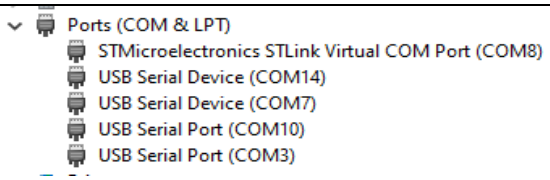
The STM32CubeProgrammer software will then program the firmware into the flash memory on the STM32H743 CPU. After a short time a dialog box will pop up saying that "File download completed". **Do not do anything at this point** as the software will then start reading back the firmware programmed into the flash.

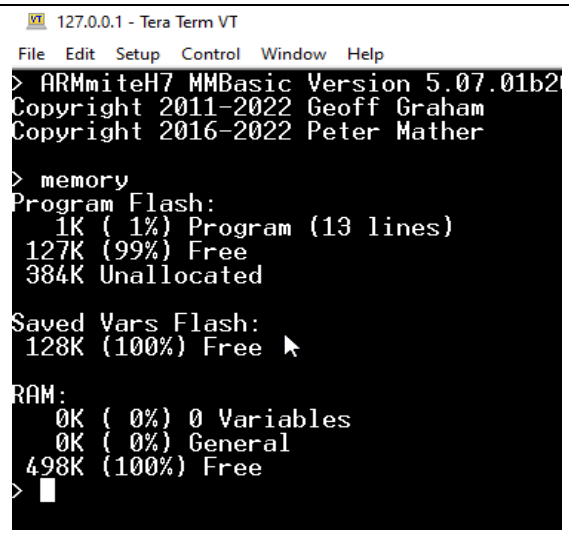
When this has completed successfully another dialog box will pop up saying "Download verified successfully" as shown below. The whole operation will take under a minute and any messages in red will indicate an error.



Dismiss both dialog boxes and disconnect the STM32CubeProgrammer software, using the Disconnect button.

Connect to the MMBasic Console.

The ST-LINK module attatched as part of the development board provides connection to the MMBasic console via the same USB cable and connection used to load the firmware.	

Connect a terminal emulator to the correct COM port, restart the STM32H743 by power off/power on or pressing the Reset button and you should see the Armmite copyright banner.	

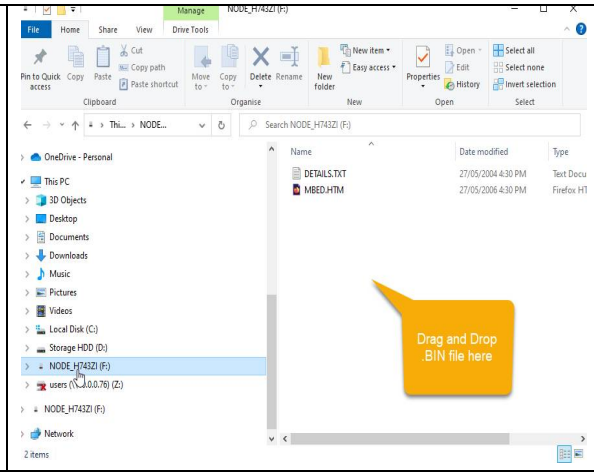
Alternative Method – Using STLINK

Hazard	There have previously been cases where STMCubeProgrammer has failed to properly load the firmware. I have not experienced this but using this alternative software is recommended if you have issues.

This recommendation from the original developer:

Please download and install ST-LINK/V2 utility software and use it to program the Nucleo rather than STM32CubeProgrammer.It takes a little longer but it correctly programs the chip. My uploads are all done through the development environment which uses ST-LINK. I've just tested the most recent posted binary with both STM32CubeProgrammer and ST-LINK and it only works properly when programmed with ST-LINK.

Alternative Method – Dragging Bin file to Directory

ST-LINK will appear as a USB drive in windows explorer. It is possible to update the firmware by just dragging the .bin file to the drive. It will load once dropped and the processor will restart running the new firmware. This worked for me, but see the note below.	

This note from the original developer:

Note also that I always use the ST-LINK to update the firmware rather than dragging the .bin file

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 alnoe 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 most 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 diffented between platforms.e.g. NAME vs RENAME

Also show commands trated as separate functionality, but implemented as the same command. e.g. CAT and INC

BLIT/SPRITE

BLIT and SPRITE are the same command and both forms are accepted, however which one is appears via LIST or EDIT varies between the Armmites and Picomites.

Table of Accepted and Core Syntax

fsasfsafaafsaadsffadfa

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**	DHT22 HUMID **BITBANG HUMID**	BITBANG HUMID
WS2812 Leds	n/a	**WS2812**	WS2812 **BITBANG WS2812**	BITBANG WS2812

BITSTREAM	CSUB (Bitstream)	**BITSTREAM**	**BITBANG BITSTREAM**	BITBANG BITSTREAM
LCD	LCD	LCD	LCD	**BITBANG 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$	
Version			MM.VER	
Error No.			MM.ERRNO	
Error Message			MM.ERRMSG$	
			MM.ONEWIRE	
			MM.FONTHEIGHT	
			MM.FONTWIDTH	
		=> **>=**	=> **>=**	
		=< **<=**	=< **<=**	
ERASE variable	**ERASE**	ERASE	ERASE **CLEAR VARS**	ERASE
Blit and Sprit	**BLIT**	**BLIT**	SPRITE **BLIT**	BLIT **SPRITE**
		CAT	CAT	CAT

		INC	**INC**	**INC**