

Graphics in PicoMite VGA

Introduction

The PicoMite is a standalone computer running MMBasic. The full story can be read in:

<https://geoffg.net/picomitevga.html>

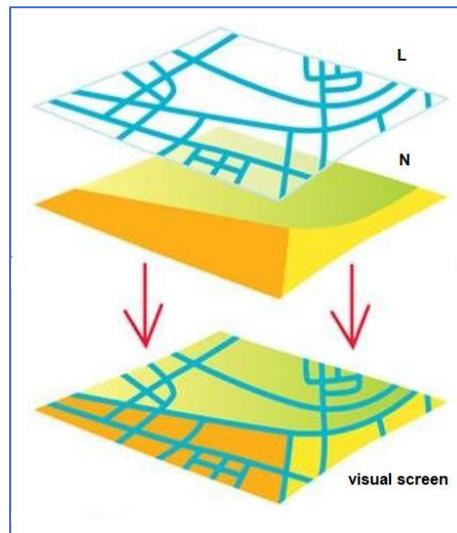
The base for these computers is the Raspberry Pi Pico, a small module based on the RP2040 chip.

This paper tries to explain how graphics can be used using MMBasic in the PicoMite. Most of this is transparent between the different flavors of PicoMite, but is essentially based on the PicoMite VGA.

Reference is given to PETSCII ROBOTS game, that uses presented method.

Graphics layers

To the user, the video screen shows as a 2D object, a flat surface. That is called a layer. To the programmer the video screen is a piece of memory. That is called a (frame)buffer. These terms are used in below text, and refer to the same.



Layer N

The PicoMite starts up using only 1 graphics layer (layer N -or- framebuffer N). In the VGA version this is your visible screen, when you use an LCD, this is the LCD screen. In case you draw a line or print text to the screen, it is on layer N. Important to understand the difference between the VGA and LCD displays is that the N layer in an LCD is INSIDE the LCD itself, in its native color depth. For an ILI9341 this is 16 bit color (RGB5:6:5). The VGA version keeps the information inside the RP2040 chip, and this is 4 bit resolution (RGB1:2:1).

Framebuffer F

If you write to screen a often, it is possible you see screen artefacts. To avoid these artefacts you can do all the writing to an intermediate buffer (FRAMEBUFFER F), and when you are ready doing all the graphics update you can copy the framebuffer F to the N layer (**FRAMEBUFFER COPY F,N**). Note that framebuffer F is inside the RP2040 chip and is RGB 1:2:1. In case you use an LCD the PicoMite converts RGB1:2:1 to RGB 5:6:5 automatically. Framebuffer F uses 38kbyte RAM. Framebuffer F is never directly visible. It is just a memory block that can contain RGB 1:2:1 data.

Layer L / Framebuffer L

Especially for gaming it is nice to have multiple graphics layers. PicoMite offers one layer L on top (overlying) the N layer, called Framebuffer L. When you use layer L, it is visually in front of layer N, and can obstruct the vision to layer N. For that, layer L is featured with a “transparent color”. If you paint layer L with that transparent color, you see the graphics of layer N. If you use any other color, you see the pixels of layer L. The transparent color is default **RGB (BLACK)** , but can be altered to any of the 16 colors the PicoMite supports. **RGB (Magenta)** is often used. FRAMEBUFFER L is physically inside the RP2040 chip, and is 38kbyte in size. In the VGA version, merging of N and L is automatic. Use of framebuffer F is optional. In the LCD version (remember, both L and N are in a different chip) merging cannot be done between N and L. The LCD version does the merging on framebuffers F and L, and then writes the result to the physical display N.

Framebuffer N is available by default. Other framebuffers can be enabled with :

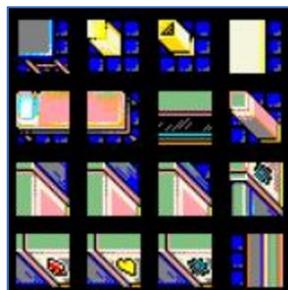
```
FRAMEBUFFER LAYER (for L)
FRAMEBUFFER CREATE (for F) .
```

For the PicoMite VGA version the use of framebuffers is only useful in graphics **MODE 2** (320x240 16 colors).

Graphical elements

In essence there are 2 different graphical formats that can be used in PicoMite: Tiles and Sprites. I am not a graphics designer, this is my name for the formats.

A tile is a (rectangular) graphical object that is in native color space (i.e. RGB 1:2:1). It can be copied onto any layer and is directly visible. A tile can be small, or screen size. PicoMite can load a tile from a storage device (SD card) with **LOAD IMAGE "filename.bmp"** and display it on the active layer. As a bonus, LOAD IMAGE will also convert 24 bit RGB to RGB1:2:1 on the fly. The file must be BMP format.



Some tiles from PETSCII robots

A Sprite is a (rectangular) graphical object that consists of color indexes (indices). Each pixel is not represented by the a RGB value, but by a palette index. This can be very confusing for PicoMite since it supports 16 colors (RGB 1:2:1) but also happens to support 16 color indexes in the sprite file format. And these are NOT the same for reason of backward compatibility to CMM1, CMM2. The sprite data in a file cannot be copied onto the screen directly. It must be converted to RGB values. Sprite data in a file consists of a header, indicating dimensions of the sprite (i.e. 16x8 pixels) and then a list of (i.e. 128(16x8)) color indexes.

Sprites are powerful elements, they can restore background when moved, there is collision detection when they move. PicoMite supports 31 sprites (Note: in the 5.09.00b0 release 64).

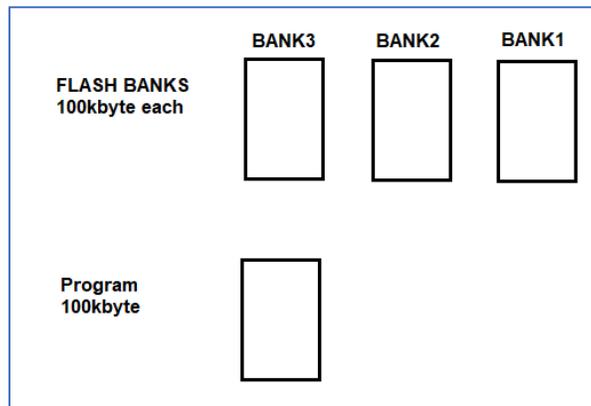
Sprites must be loaded in memory with `SPRITE LOAD "filename",sprite_number`. After that the sprite can be displayed with `SPRITE WRITE sprite_number,x,y` (hard write) or `SPRITE SHOW sprite_number,x,y` (remembers background) and `SPRITE MOVE`.



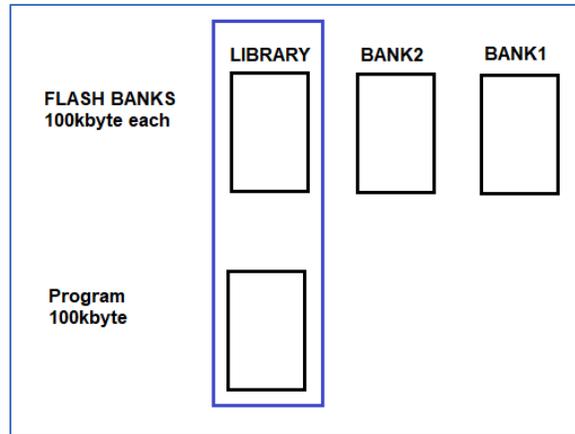
Some sprites with Magenta background from PETSCL robots.

Memory

The RP2040 has limited amount of RAM that must be shared between system, video, variables, stack, buffers (i.e. audio) etc. Therefore provisions exist to also use flash memory for MMBasic programs.



The flash banks can be used to store MMBasic programs, and even execute from them. You can CHAIN programs. Details are beyond the scope of this document. Just know they exist. MMBasic also supports a LIBRARY. The exact same flash memory block that is reserved for BANK3 can be used as an extension of the program memory, with limitations.



The limitation is that you cannot edit the program in the library. You can save **CSUB's**, **SUB's** and **FUNCTIONS** in the library that are “bug free” and use them from the main program. Because the library is active when the MMBasic program runs, you can only access the library from the command line. If you have debugged code in your program memory, transfer it to the library with **LIBRARY SAVE**. **LIBRARY LIST** shows what is in the library, and **LIBRARY DELETE** erases the library and returns the flash memory to flash bank3.

CSUB

Some of you may remember that in the 80's there were basic programs with lots of DATA statements at the end, that no-one knew what they meant, but that were Poked into memory, and then executed (CALL xxx). In essence that was machine code represented as hexadecimal numbers. A CSUB is exactly that. The CSUB contains in hexadecimal form machine code instructions for the ARM processor. MMBasic is a bit more advanced, in that you do not need to remember where to POKE, and what to CALL. Poking is done automatically when you run the program, and by just using the name of the CSUB, the CALL xxx is executed.

It is essential that, at RUN time, the hexadecimal code is written into memory in binary form.

When you **LIBRARY SAVE** a program that contains a CSUB, the binary data is written in the library (not the hexadecimal text representation).

Implement Graphics : preparations

We now have the knowledge to understand the graphics used in advanced games as PETSII ROBOTS. PETSII ROBOTS is a (for MMBasic) large game. The basic program is only 70kbyte, but it uses data (maps, graphics, sounds) that are over 800kbyte in size.

The game uses over 100 sprites, and roughly 300 tiles. With the limitation of 31 Sprites maximum, above knowledge was used to implement the game. Below are game design decisions that were taken to make the game fit in the PicoMite.

1/ Use only framebuffer L in the VGA version, to save RAM for variable storage (maps/attributes). This has the disadvantage that you may see minor corruptions in the video while updating the screen. Provisions are made in the program to minimize this by synchronizing heavy screen updates with vertical sync, using **FRAMEBUFFER WAIT**.

To make this effective, it is essential to execute all code that writes to video memory close together. Do not litter the whole code with writing to video memory. PETSII uses 2 SUBs, executed every game loop. One that

writes everything on the N layer, and a second that writes everything on the L layer. And these SUBs are tuned for speed.

2/ Store all graphical elements (tiles/sprites) in binary format. MMBasic has highly optimized memory copy routines from flash memory to video RAM in `BLIT MEMORY` (-or- `SPRITE MEMORY`). The `BLIT MEMORY` function is fast, but cannot restore backgrounds or detect collisions of sprites. The detection of collisions is performed in MMBasic on basis of X and Y coordinates of tiles and sprites.

The restoring of background can be avoided by using 2 layers. The N layer for the world (world map) built out of tiles. Use the L layer, with magenta color as transparent color, to write the sprites. When they move, the N layer is not affected.

Store Sprites and Tiles in binary format.

When running a CSUB, or saving it to the library, the hexadecimal data is converted to binary form. Matherp has written a basic program that converts `SPRITE` files (containing header and color indexes) into header+hexadecimal RGB 1:2:1 data mimicked as a CSUB.

'program to convert all the sprite files in a directory to a CSUB using compression where appropriate

```
Option explicit
Option default none
Const separatesubs% = 0
Dim offset%
Dim fname$=Dir$ ("*.spr",FILE)

Open "tile0_csub.bas" For output As #2
Open "tile0_index.txt" For output As #3

If separatesubs%=0 Then
Print #2,"CSUB TILE0"
Print #2,"00000000"
offset%=0
EndIf
Do
If fname$<>"" Then code fname$
fname$=Dir$()
Loop Until fname$=""
If separatesubs%=0 Then Print #2,"END CSUB"

Close #2
Close #3

'convert the file f$ to a compressed CSUB
Sub code f$
Local i%,j%,h%,l%,w%,n%,s%,il%
Local a$,o$,oc$
Open f$ For input As #1
Line Input #1,a$ 'process the dimensions and count
w%=Val(Field$(a$,1,""))
n%=Val(Field$(a$,2,""))
h%=Val(Field$(a$,3,""))
If h%=0 Then h%=w%
i%=Instr(f$,".")
o$=Left$(f$,i%-1)
```

```

If separatesubs%=1 Then
    Print #2,"CSUB "+o$
    Print #2,"00000000"
    offset%=0
' Else
'     Print #2,""+o$
EndIf
Local obuff%(w%*h%\8+128),buff%(w%*h%\8+128)
For s%=1 To n% 'process all the sprites in a file
    Print #3,Str$(offset%)
    Print #2,"'Offset ";offset%
    For l%=1 To h%
        a$=""
        Do While Left$(a$,1)="" 'skip comments
            Line Input #1,a$
        Loop
        'make sure all lines are the correct length
        If Len(a$)<w% Then Inc a$,Space$(w%-Len(a$))
        If Len(a$)>w% Then a$=Left$(a$,w%)
        LongString append buff%(),a$ 'get all the file into a single longstring
    Next l%
    j%=0
    For i%=1 To LLen(buff%())
        LongString append obuff%(),mycol$(LGetStr$(buff%(),i%,1))
    Next i%
    LongString clear buff%()
    il%=(LLen(obuff%()+7)\8 * 8
    i%=0
    Do While i%<w%*h% 'compress the data
        j%=LGetByte(obuff%(),i%)
        l%=1
        Inc i%
        Do While LGetByte(obuff%(),i%)=j% And l%<15
            Inc l%
            Inc i%
        Loop
        LongString append buff%(), Hex$(l%)+Chr$(j%)
    Loop
    'the output must be a multiple of 8 nibbles
    LongString append buff%(),Left$("00000000",8-(LLen(buff%()) Mod 8))
    If LLen(buff%())<il% Then 'compressed version is smaller so use it
        Print #2,""+o$;
        Print #2," is compressed"
        Print #2,Hex$(h%+&H8000,4)+Hex$(w%,4)
        j%=0
        For i%=8 To LLen(buff%()) Step 8 'reverse the order
            o$=LGetStr$(buff%(),i%,1)
            Inc o$,LGetStr$(buff%(),i%-1,1)
            Inc o$,LGetStr$(buff%(),i%-2,1)
            Inc o$,LGetStr$(buff%(),i%-3,1)
            Inc o$,LGetStr$(buff%(),i%-4,1)
            Inc o$,LGetStr$(buff%(),i%-5,1)
            Inc o$,LGetStr$(buff%(),i%-6,1)
            Inc o$,LGetStr$(buff%(),i%-7,1)
            Inc j%
            If j%=8 Then
                Print #2,o$
                j%=0
            Else
                Print #2,o$+" ";
            EndIf
        Next i%
        If j%<>0 Then Print #2,""

```

```

    Inc offset%,4+LLen(buff%())\2
Else
    Print #2,""+o$;
    Print #2," is uncompressed"
    Print #2,Hex$(h%,4)+Hex$(w%,4)
    LongString append obuff%(),Left$("00000000",8-(LLen(obuff%()) Mod 8))
    j%=0
    For i%=8 To LLen(obuff%()) Step 8 'reverse the order
        o$=LGetStr$(obuff%(),i%,1)
        Inc o$,LGetStr$(obuff%(),i%-1,1)
        Inc o$,LGetStr$(obuff%(),i%-2,1)
        Inc o$,LGetStr$(obuff%(),i%-3,1)
        Inc o$,LGetStr$(obuff%(),i%-4,1)
        Inc o$,LGetStr$(obuff%(),i%-5,1)
        Inc o$,LGetStr$(obuff%(),i%-6,1)
        Inc o$,LGetStr$(obuff%(),i%-7,1)
        Inc j%
        If j%=8 Then
            Print #2,o$
            j%=0
        Else
            Print #2,o$+" ";
        EndIf
    Next i%
    If j%<>0 Then Print #2,""
    Inc offset%,4+LLen(obuff%())\2
EndIf
LongString clear obuff%()
LongString clear buff%()
Next s%
Close #1
If separatesubs%=1 Then Print #2,"END CSUB"
End Sub
'
'converts the Ascii colour from the Maximite standard to PicoMite standard
Function mycol$(c$)
Static cols%(15)=(0,1,6,7,8,9,14,15,2,3,4,5,10,11,12,13)
Local i%
If c$="" Then c$="0"
i%=Val("&H"+c$)
mycol$=Hex$(cols%(i%))
End Function

```

This program converts all sprites "filename.spr" to one "xxxx_csub.bas" file and an "xxxx_index.txt" file that contains all pointers to the elements in the CSUB (after conversion in binary form). The relative positions are referenced to the CSUB start address. The CSUB name (i.e. CSUB TILE0) is essential for this, since there will be many CSUB's. Note that it processes the files in the directory in ascending order. Therefore it is best to number the sprites in their file name, so the order in the CSUB is predictable.

Then "LOAD xxxx_csub.bas" to load the file in program memory, and "LIBRARY SAVE" to write it as binary data in flash (in the library). Now we have binary RGB 1:2:1 data in flash, at an unknown location.

You can save multiple xxxx_csub.bas files to library, until the library is full.

To find the absolute address of each CSUB in memory, you retrieve the CSUB start address:

```
Address% = PEEK(CFUNADDR csub_name)      ` i.e. PEEK(CFUNADDR TILE0) .
```

It is best to build an array with absolute addresses for the BLIT MEMORY function. Below example creates an absolute index for the 6 (0..5) sprites in the CSUB called HEALTH.

```

'get start addresses
Local hlt=Peek(cfunaddr HEALTH)

'build global index file
Dim health_index(5)

Open "sprites/hlt_index.txt" For input As #1
For i=0 To 5
  Input #1,a$
  health_index(i)=hlt+Val(a$)
Next
Close #1

```

Now you can use `BLIT MEMORY health_index(3),X,Y` to copy the 3rd sprite to (X,Y) on the active layer set by `FRAMEBUFFER WRITE L/N`

From LIBRARY to FLASH BANK

Above you are using the LIBRARY for binary data. That requires you to prepare the library manually (command line) which is not convenient for a game. Remember however that LIBRARY and FLASH BANK3 use the exact same locations in flash, but work differently.

LIBRARY is part of your active program, and therefore CSUB's have a meaning in the main program. FLASH BANK3 is not part of the active program, and therefore the CSUB has no meaning.

But the start address of flash bank 3 (can change between revisions for MMBasic) can be requested by `flash_address% = MM.INFO(FLASH ADDRESS 3)`. This is the memory start address for flash bank 3, and it is also the memory start address for the library.

In stead of using an index based on the CSUB, adapt it to be an index that refers to the start of the flash bank. Add an offset to each index (difference between "flash address" and "csub address for that csub"), and create a new index file that refers to the start address in flash. Do this for all CSUB's.

Now you have an index file "`flash_index.txt`" that references each and every graphical element (sprite/tile) to the FLASH BANK start address.

Then save the library to SD card as a binary file

```
LIBRARY DISK SAVE "filename.bin"
```

And keep this library together with the index file. Both are useless without the other.

Now we can clean up the library, and can use the flash space for FLASH BANK 3.

Implement Graphics: The GAME

In the MMBasic game program we can now load the binary in flash bank 1,2 or 3, and all graphical elements can be found by adding the index to the flash start address of that particular bank you choose to use in the game. Example:

```
FLASH DISK LOAD 2, filename.bin,0           ` loads the binary file in flash bank 2
Flash_address%=MM.INFO(FLASH ADDRESS 2)    ` gives the start address of bank 2
```

After that, create an array `abs_index%()` with absolute indexes adding `flash_address%` to all relative indexes read from the index file "`flash_index.txt`".

Then simply : `BLIT MEMORY abs_index(i),X,Y`


```

        wt$=wt$+hex$(c1,1)
    Next
    ?#1, wt$
next
    box x,y,w,h,,rgb (white)
close #1
inc y,h:if y>383 then y=0:inc x,w
next TNR

colors:
'--Colorscheme according to Spritecolors
Data RGB (BLUE) , RGB (GREEN) , RGB (CYAN) , RGB (RED)
Data RGB (MAGENTA) , RGB (YELLOW) , RGB (WHITE) , RGB (MYRTLE)
Data RGB (COBALT) , RGB (MIDGREEN) , RGB (CERULEAN) , RGB (RUST)
Data RGB (FUCHSIA) , RGB (BROWN) , RGB (LILAC)

```

Programs like this take quite a lot of time per sprite/tile, so this is best run on MMB4W. In case you want to run it on PicoMite, change "load bmp FN\$" to "load image FN\$"

3/ MEMORY

MMB4W does not support a LIBRARY, creating the binary must be executed on the PicoMite itself. To load the xxxx_csub.bas file into memory, and save it to the library, you need program memory for both the hexadecimal text presentation, and the binary result in RAM. Therefore you can only process 60kbyte size xxxx_csub.bas files in 100kbyte program space. This is why the tiles and sprites are distributed over several folders, so xxxx_csub.bas files stay within these requirements.

Final words

As above write-up shows, there is quite a lot of work involved, so it is preferable to start with a well-defined set of graphical elements. It easily takes one whole hour to do this for 400 elements. But the reward is a fast graphics system. In PETSCII the graphics take up only 30% of the game loop.

If you are new to the procedure, start with a graphics set of few graphical elements, until you master the process.