

# **GRAPHICS Programming on the Colour MaxiMite 2**

**Peter Mather**

**Version 2.0a**

Copyright 2020 - Peter Mather

## INTRODUCTION

---

This is a compilation of the threads posted by Peter Mather as a background and deeper explanation of the graphics capabilities of the CMM2. I have taken the liberty (with Peters OK), to post them on the forum with some additional comments and explanations of recent developments in the hope that they will assist others in realising the full potential of the CMM2. Any typing mistakes, errors or omission are mine (apologies Peter). There is also demo code included that was created by members of TheBackShed forum (with their permission - thanks). Editor: Doug Pankhurst.

From Peter:-

*'I thought I'd start a thread to try and explain the concepts behind the CMM2 graphics and give some annotated examples of how to use the facilities in the CMM2. I'm sure Mauro will chime in and then perhaps some of what is in the thread can form the basis of a graphics manual of some sort.'*

### Revision History:

Version 2a: Reformatted and added information about version 5.05.05 firmware.

Version 1g: Updated/added information about the MODE command.

Version 1f: Updated/added information about the PAGE command.

Version 1e: Added Table of Contents, FRAMEBUFFER, Quaternions and corrected program bugs.

Version 1d: Adds more demonstration programs.

Version 1c: Includes information on graphics features in MMBasic 5.05.04RC6 (the draft version)

Note: These features may not end up included in the final release.

Version 1b: Updated hot links to programs and demos. Includes features and capabilities from MMBasic 5.05.03.

# Table of Contents

---

<a href="#">Basic Graphics on the CMM2</a> .....	4
<a href="#">Graphics Pages</a> .....	6
<a href="#">PAGE COPY</a> .....	7
<a href="#">Colour Lookup Tables</a> .....	8
<a href="#">The BLIT command</a> .....	11
<a href="#">Graphics Modes</a> .....	15
<a href="#">Colour Depths</a> .....	16
<a href="#">The PAGE Command</a> .....	21
<a href="#">PAGE SCROLL</a> .....	21`
<a href="#">PAGE STITCH</a> .....	24
<a href="#">12-bit Mode Graphics</a> .....	26
<a href="#">Images</a> .....	30
<a href="#">Sprites and how to create them</a> .....	34
<a href="#">Sprites and how to use them</a> .....	39
<a href="#">The FRAMEBUFFER</a> .....	43
<a href="#">Quaternions</a> .....	45

## BASIC GRAPHICS on the CMM2

This first post will cover the basics of how the graphics are generated and work.

Some of the STM32 chips, including the STM32H743IIT6 that we are using, have an on-chip graphics controller (called the LTDC = Lcd Tft Display Controller).

This is programmed to read from a defined area of memory at a specific rate and write the data to a set of I/O pins. In our case the I/O pins are connected to the resistors that make up the three R2R ladder DACs on the CMM2 motherboard and these create the analogue VGA signals. In addition, the LTDC is programmed to create the HSync and VSync signals which tell the monitor that we have reached the end of a line and/or end of a page.

All these signals must waggle at a rate determined by the a defined VGA specification depending on the resolution required.

In the case of the 800x600, display memory is being read at a rate of 40MHz, the line rate is 37.8787KHz and the frame rate is 60Hz.

The important point about the LTDC is that all this happens without any use of the processor once the controller has been initialised. It is basically a continuous circular DMA (Direct Memory Access). Of course this isn't completely free as it does consume memory bandwidth even though the processor isn't involved.

So to get something onto the screen all we have to do is write to the area of memory that the LTDC is reading and something will appear on the screen.

For performance reasons the best memory to use is the STM32H743IIT6's internal memory and 512KB of this is allocated to the main video memory. In addition I have allocated 3MB of the 8MB SDRAM memory (this is the chip on the back of the Waveshare PCB) to use for graphics.

So we have 3.5MB of memory available for graphics, 512KB is located at memory address &H24000000 and 3MB at address &HD0000000. We will see in the next section how we can poke this memory to write things on the screen.

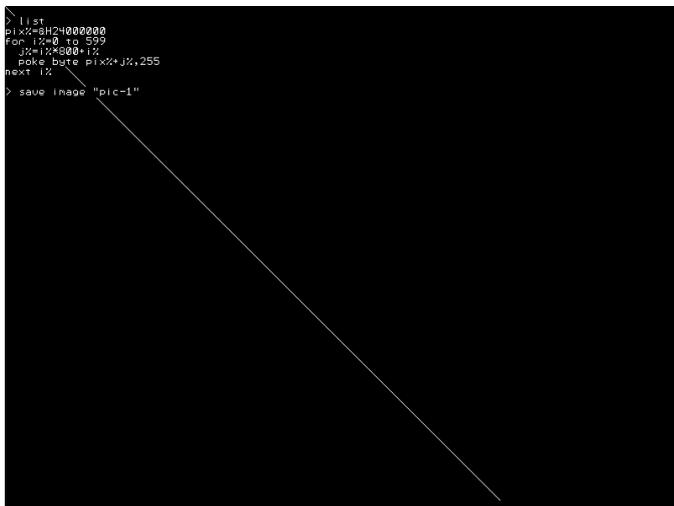
However, we need one more concept before starting to explore how to use this memory - the Colour LookUp Table or CLUT. The default resolution of the CMM2 is 800x600 with an 8-bit colour depth so the screen uses 480,000 bytes out of the 512 KBytes available in the processors internal video memory.

However, we need to drive 16 data lines not just 8 and the way this happens is that the LTDC takes each 8-bit value in turn as it reads memory and uses it as an index to a table of 16-bit values (the CLUT). This table is initialised by the CMM2 firmware to sensible values to give good colour coverage but can be changed from within MMBasic using the MAP command (subject for another post).

OK, so we know that in 800x600x8 resolution the screen is using 480KB of memory starting at &H24000000. This memory is organised exactly as you would expect. The first 800 bytes are the top line of the display starting at the top left hand corner. The second 800 bytes are the second line of the display. and the 480,000th byte is the bottom right of the display.

So lets draw a diagonal line on the display to prove this.

```
pix%=&H24000000
for i%=0 to 599
  j%=i%*800+i%
  poke byte pix%+j%,255
next i%
```



Note how the line just writes across anything already on the screen (except for the SAVE command which was entered after the program was run).

It is a bit silly having to memorise &H24000000 so the CMM2 includes a built in function to give it to us

```
pix%=mm.info(page address 0)
for i%=0 to 599
  j%=i%*800+i%
  poke byte pix%+j%,255
next i%
```

Of course we could have done the same thing using the command

```
line 0,0,599,599
```

Internally in the CMM2 firmware the LINE command is doing exactly what we did with POKE. And, of course, all the basic graphics commands are doing the same thing - writing to memory in order to construct the requested shapes.

## Introduction to Colour

---

A brief side trip to explain the use of colour in the CMM2 may be in order, particularly for newcomers. This is a collection and perhaps a little expansion on details provided elsewhere in this manual.

You will all be aware I am sure of the format used to define colour in MMBasic code – that being RGB(*nnn,nnn,nnn*) where *nnn* represents a number between 0 and 255 for Red, Green and Blue respectively. As you know, it takes 8 bits to represent all the values from 0 to 255 and thus this format is known as RGB888 (also 24 bit colour). With this format you can choose up to 16,777,216 colours (256 x 256 x 256).

Memory is like gold in the CMM2 and RGB888 uses up LOTS of memory so a compromise was decided on by the developers. As you can see from the schematic of the CMM2, there are only 5 video lines for red, 6 for green and 5 for blue meaning 5 bits, 6 bits and 5 bits for red, green and blue respectively. This is known as RGB565 (16 bit colour) and provides for 32 intensity levels of red, 64 for green and 32 for blue and is the format used in any mode using 16 bit colour depth. RGB565 gives 65,535 possible colours plus black or more accurately, no colour at all.

As an even more parsimonious use of memory and also to be similar to the colour capabilities of the older computers from the 80's, 8 bit colour depth is provided that only uses some of the video lines. Having only 8 bits to play with, the format is RGB332. That is 3 bits for red, 3 bits for green and 2 bits for blue. As this is a quite limited range, these values are used as an index into a translation table called a Colour LookUp Table. There is a detailed description of this in a later chapter titled CLUT.

Back to the 16 bit colour depth for the moment, the upshot of this translation from MMBasic level RGB888 down to the hardware level of RGB565 is that there is no change in intensity (stored value) between a red value of 104 and 108 for example. In fact, as we can only represent 32 possible values of red and blue and 64 of green, the RGB888 values are translated by the function modulo 8, modulo 4 and modulo 8 respectively for red green and blue.

This can be shown with the following code:-

```
' demo of RGB888 to RGB565 translation
' use red as an example
option default integer
mode 1,16
page write 1
cls
' first, plug in some RGB888 values
' incrementing red by 1 each time
x = 0
y = 0
cr = 100
cred = cr
cgrn = 0
cblu = 0
  for j = 0 to 49
    pixel x,y,rgb(cred,cgrn,cblu)
    x = x + 1
    cred = cred + 1
  next j
' now look at the display memory
' to see the RGB565 values stored
addr1= mm.info(page address 1)
cred = cr
for k = 0 to 75 step 2
  a1$ = hex$(addr1+k)+" "
  a2$ = hex$(peek(short addr1+k),4)
  a3$ = "      rgb("+str$(cred)+","+str$(cgrn)+","+str$(cblu)+")"
  text 0,30+(k*7),a1$a2$a3$
  cred = cred + 1
next k
page copy 1 to 0
do
loop
```

There is an additional colour mode used in the CMM2, the 12 bit colour depth. This is a good deal more complex as it includes the functionality of transparency. It is referred to as ARGB4444 and as you would imagine uses 4 bits to represent transparency, red, green and blue respectively. There is more detail in the chapters following but the principles for translation from RGB888 to ARGB4444 are similar. I will try to expand on this once I learn more about it! (-:

Next chapter we will explore graphics pages and how they allow us to do some of the magic but if you properly understand the above then everything else should be easy.

# GRAPHICS PAGES

In the previous post we saw that writing to the memory identified by MM.INFO(PAGE ADDRESS 0) displays immediately on the screen. In fact the CMM2 always and only ever shows on its display the information in the memory starting at MM.INFO(PAGE ADDRESS 0) (NB: not true in 12-bit colour depth but that is a subject for a much later post). This is important as it underpins the explanations below.

Drawing the line using POKE in the previous example took 12mSec on a 400MHz CMM2. This is fast enough that the act of drawing didn't create any strange visual effects.

However, if we slow it down by adding a pause we can see the line drawing across the screen as it now takes 614mSec.

```
pix%=mm.info(page address 0)
for i%=0 to 599
  j%=i%*800+i%
  poke byte pix%+j%,255
  pause 1
next i%
```

This is a trivial example but lots of graphically intensive activities do take "real" time to complete and it may be that we don't want to see the update taking place but would rather move from one static image to another.

A rather better example would be displaying a BMP image. BMP files are uncompressed so they are large and take time to load as they wait for the SDcard to read. They also load from the bottom up which is slightly strange. A full colour 800x600 BMP takes just less than a second to load.

If we type

```
? MM.INFO(MAX PAGES)
```

we get the answer 6. This says that in our current 800x600 resolution the 3.5MB of video memory is being split into 7 chunks, numbered 0 to 6.

We know that Page 0 is what is being displayed on the screen and it will always be Page 0 that is displayed. However, we don't need to write to Page 0. MM.INFO(MAX PAGES) tells us that in 800x600x8 video mode we have 7 video pages at our disposal.

We can see where they are in memory using MM.INFO(PAGE ADDRESS n)

```
for i=0 to 6:print hex$(mm.info(page address i)):next i
24000000
D0000000
D0080000
D0100000
D0180000
D0200000
D0280000
```

We can tell the CMM2 which page to use for writing using the command

```
PAGE WRITE n
```

So lets type PAGE WRITE 1 at the command line

OH NO!!!! The status line just disappeared, the cursor disappeared and when I type, nothing happens!

All console output is now being directed to a different area of memory starting at &HD0000000. Luckily any error or typing Ctrl-C will bring things back. PAGE WRITE isn't very useful at the command line but can be used.

How about?

```
PAGE WRITE 1:LOAD BMP "mybmp":PAGE WRITE 0
```

This time my cursor came back OK after about a second but no picture?

Now type:

```
PAGE COPY 1 TO 0
```

Instantly the picture will appear (actually it takes about 4mSec).

Now we can use this to create a BMP slideshow where each image instantaneously replaces the previous one.

```
page write 1
a$=dir$("*.*bmp")
do
  load bmp a$
  page copy 1 to 0
  pause 1000
  a$=dir$()
loop while a$<>"
```

The **PAGE COPY** command is very highly optimised to run as fast as possible. It also has an optional parameter which is documented in the user manual.

- I:** means do the copy immediately. It is the default and most efficient but risks causing screen artefacts;
- B:** means wait until the next frame blanking and then do the copy. It is the least efficient but is absolutely determinate in its effect and no screen artefacts will ever be seen; and
- D:** means carry on processing the next command and do the copy in the background when the next frame blanking occurs. This is efficient but must be used with care as subsequent drawing commands may or may not be included in the copy depending on the timing of the next screen blanking.

In a future post we will improve the slideshow program by having one image replace the previous by slowly scrolling in from the right hand side using another of the **PAGE** command functions.

But to summarise:

- **PAGE COPY** allows you to pretty much instantaneously replace what is being seen with a new image with no artefacts.
- **PAGE COPY** is very fast. For example in 320x240x8 graphics mode it takes less than 0.3mSec.
- Multiple pages allow you to store and/or construct complex images without any visible screen artefacts.

Some additional sub-commands for the **PAGE** command are :

```
PAGE AND_PIXELS sourcepage1, sourcepage2, destinationpage
PAGE OR_PIXELS sourcepage1, sourcepage2, destinationpage
PAGE XOR_PIXELS sourcepage1, sourcepage2, destinationpage
```

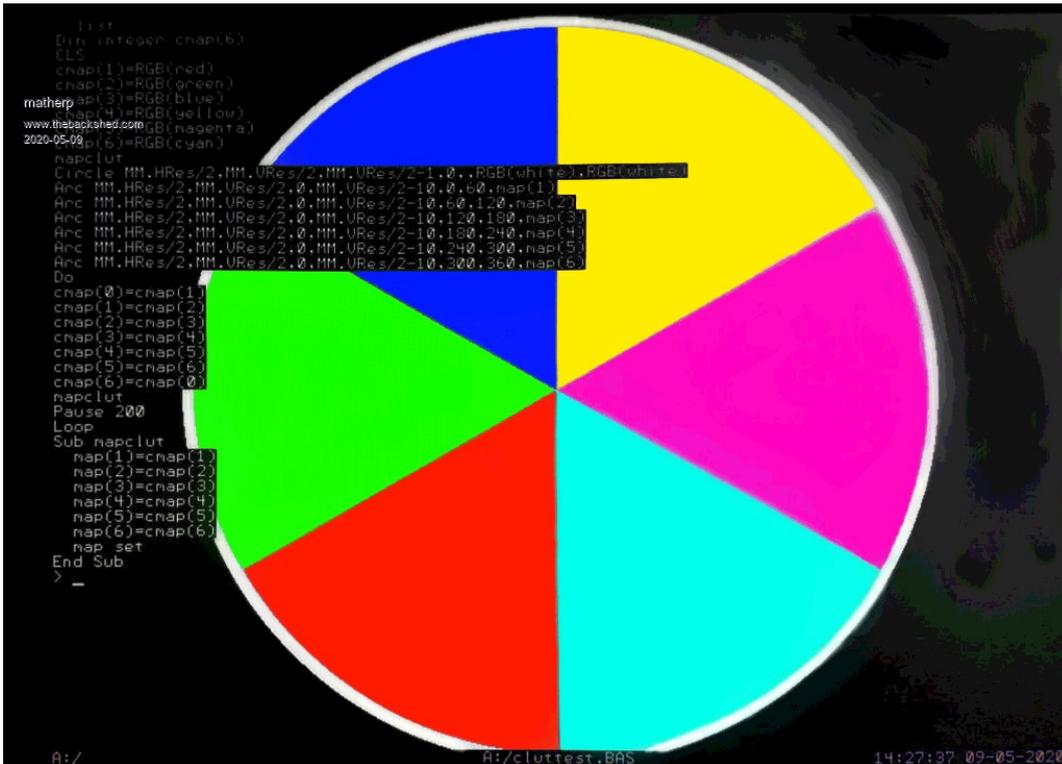
These combine the pixels on sourcepage1 and sourcepage2 by **ANDing**, **ORing**, or **XORing** them. destinationpage can be the same as either of the sourcepages if required. For example:

```
mode 3,16
load jpg "tiger-320"
pause 1000
page write 1
cls
box 50,50,220,100,,rgb(white),rgb(white)
timer=0
PAGE AND_PIXELSs 0,1,0
print timer
do:loop
```

This loads the image to page 0, the view page as normal. Then a white box is drawn on page 1. The **PAGE** command then **ANDs** page 1 and page 0 and replaces page 0 with the result. The effect is to use the white box to window the original image. Of course this example is simplistic but I'm sure Mauro, amongst others of you, will find innovative uses.

## COLOUR LOOKUP TABLES

In this post we look more at the Colour LookUp Table (CLUT) and the MAP command and function and use them to create a rotating colour disk.



As explained earlier, 8-bit colour modes use a lookup table to convert from a single byte of information to the RGB565 pixel that will be displayed. In the CMM2 firmware the 8-bits are treated as an RGB332 pixel and are mapped to the most appropriate RGB565 value.

However, we can override this using the MAP command.

Normally a pixel that is set to zero would mean that it would be black on the screen but if we use the commands

```
MAP(0)=RGB(RED)
MAP SET
```

Then every "black" pixel on the display will instantly turn red. Actually, it isn't quite instant as the MAP SET command waits for the next frame blanking period before making the change. So MAP(n)=colour primes the colour change and MAP SET enacts it.

This allows us to change a number of colours simultaneously by using multiple MAP(n)=colour commands and then a single MAP SET.

To use our new colours in a drawing command we use the MAP(n) function. So for example if we type MAP(134)=RGB(148,73,66) to get some particular shade then to use that colour we would reference it as MAP(134)

e.g.

```
MAP(134)=RGB(149,73,66)
MAP SET
TEXT MM.HRES\2, MM.VRES\2,"Hello",CM,,5,map(134)
```

This would display the text in a dark brown colour. If we then typed

```
MAP(134)=RGB(BLUE)
MAP SET
```

The text would immediately turn blue as would any other pixels on the display set to 134.

## There a couple of other MAP commands

### MAP MAXIMITE

sets the CLUT to mimic the colours of the original Colour Maximite. i.e. 0-7 are the Maximite primary colours.

### MAP RESET

restores the original RGB332 mapping

Now lets look at using the MAP command to create the colour wheel above

```
' Set up an array to hold the colour mappings we are going to use
Dim integer cmap(6)

'Clear the screen
CLS

'Set up 6 colours in the array
cmap(1)=RGB(red)
cmap(2)=RGB(green)
cmap(3)=RGB(blue)
cmap(4)=RGB(yellow)
cmap(5)=RGB(magenta)
cmap(6)=RGB(cyan)

' Do an initial update of the CLUT to set up our colours
mapclut

'Display an outer circle in white
Circle MM.HRes/2,MM.VRes/2,MM.VRes/2-1,,RGB(white),RGB(white)

' Now draw a simple colour pie chart using our new colours with the ARC command
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,0,60,map(1)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,60,120,map(2)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,120,180,map(3)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,180,240,map(4)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,240,300,map(5)
Arc MM.HRes/2,MM.VRes/2,0,MM.VRes/2-10,300,360,map(6)

' Start a never ending loop
Do
'each time round the loop move the colours in our array one place to the left
' Use array element 0 to store the first element that is going to be at the end
  cmap(0)=cmap(1)
  cmap(1)=cmap(2)
  cmap(2)=cmap(3)
  cmap(3)=cmap(4)
  cmap(4)=cmap(5)
  cmap(5)=cmap(6)
  cmap(6)=cmap(0)

' reset the colour map
  mapclut

' pause so we can see the change
  Pause 200
Loop

'This subroutine updates the colour map for the colours we are using
' set map positions 1 to 6 to the new colours
' then apply the change
Sub mapclut
  map(1)=cmap(1)
  map(2)=cmap(2)
  map(3)=cmap(3)
  map(4)=cmap(4)
  map(5)=cmap(5)
  map(6)=cmap(6)
  map set
End Sub
>
```

Next post we are going to look at BLIT. Sounds complex? Actually all it is is a memory to memory copy - nothing more, but perhaps the most powerful tool in the graphics arsenal.

# BLIT

BLIT does a memory to memory copy. End of lesson.

Well perhaps not but that is all it is doing behind the scenes. Lets look at the BLIT syntax

```
BLIT x1, y1, x2, y2, w, h [, page] [,orientation]
```

This says copy part of the image which has a top left position of x1,y1, a width of w, and a height of h to a new position with a top left position of x2, y2.

An example:

```
' Draw a simple shape on the screen
CLS
box 100,100,100,100,5,rgb(red),rgb(blue)

'copy the box to a new location with a top left corner at 300,300
blit 100,100,300,300,100,100
```

By the way, that copy took 88 microseconds which equals  $100*100/0.000088=113$  Mbytes/Sec. BLIT is very fast.

Thinking back to the first post, we know what was just happening and we could have done the copy in Basic

```
CLS
box 100,100,100,100,5,rgb(red),rgb(blue)

'copy the box to a new location with a top left corner at 300,300
page_address%=mm.info(page address 0)
x1%=100
y1%=100
x2%=200
y2%=200
w%=100
h%=100
for y%= 0 to h%-1
  for x%= 0 to w%-1

'calculate the address of the source pixel
  pix_in%=MM.HRES * (y% + y1%) + x1% + x% + page_address%

'calculate the address of the destination pixel
  pix_out%=MM.HRES * (y% + y2%) + x2% + x% + page_address%

'copy the pixel
  poke byte pix_out%, peek(byte pix_in%)
  next x%
next y%
```

This takes just over half a second - BLIT is *much* faster!!!

Try the following commands:

```
timer=0:blit 100,100,300,300,100,100:?timer
timer=0:blit 100,100,301,301,100,100:?timer
timer=0:blit 101,101,301,301,100,100:?timer
```

The timings I get are 88uSec, 155uSec, and 214uSec.



The STM32 is much faster when it can copy data that is aligned on 4 byte boundaries. For most applications this won't matter but for a very high performance game it is something to take into account.

Now lets explore the orientation parameter of the BLIT command. This is most easily explained by a couple of examples

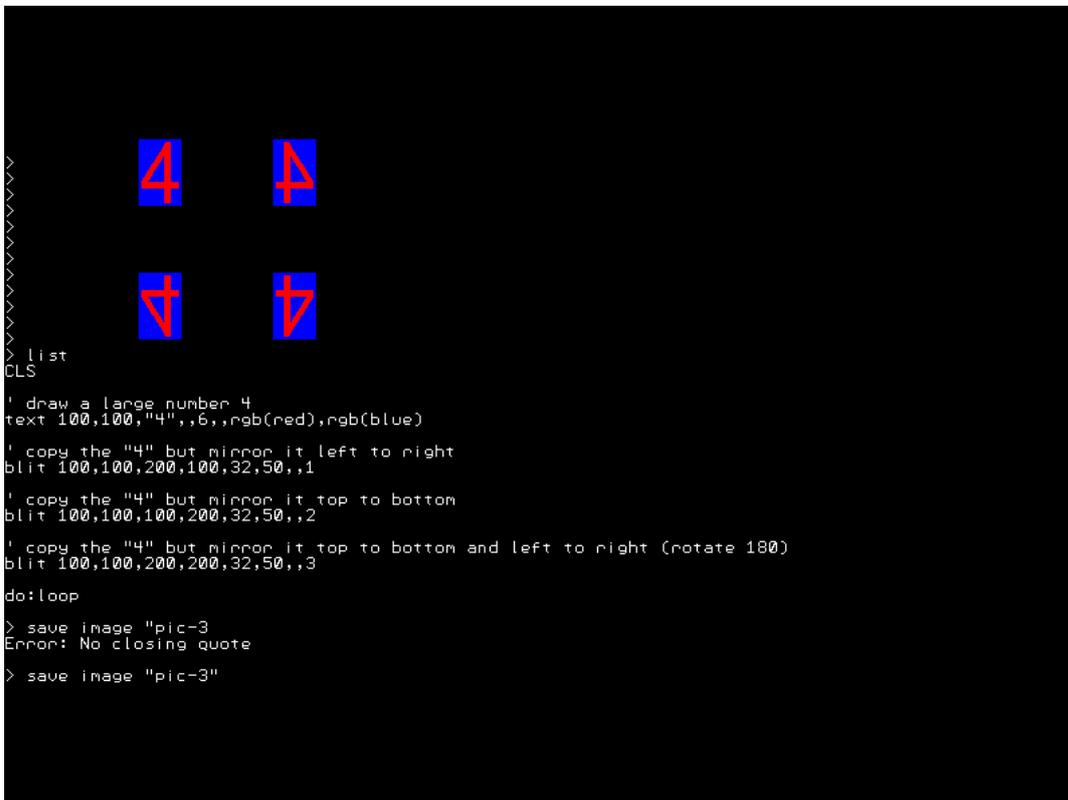
```
CLS
' draw a large number 4
text 100,100,"4",,6,,rgb(red),rgb(blue)

' copy the "4" but mirror it left to right
blit 100,100,200,100,32,50,,1

' copy the "4" but mirror it top to bottom
blit 100,100,100,200,32,50,,2

' copy the "4" but mirror it top to bottom and left to right (rotate 180)
blit 100,100,200,200,32,50,,3

do:loop
```



```
CLS
```

```
' draw a large number 4  
text 100,100,"4",,6,,rgb(red)  
  
' draw a colour box  
box 180,180,72,90,0,rgb(blue),rgb(blue)  
  
' move the text but only non-transparent (non-zero) pixels  
blit 100,100,200,200,32,50,,4  
  
do:loop
```



Of course the orientation bits can be combined in any combination.

If you BLIT from one area to another that overlaps it, then BLIT is clever enough to understand this and deal with it by buffering the original and then writing out the new version.

Up to now we have ignored the page parameter in BLIT but this is perhaps the most powerful aspect of the command.

As we saw in a previous post we can use PAGE WRITE to set drawing output to any of the video pages available but we will always see what is in page 0. The page parameter in BLIT specifies the page that BLIT will read from and then it will write to the page specified by PAGE WRITE.

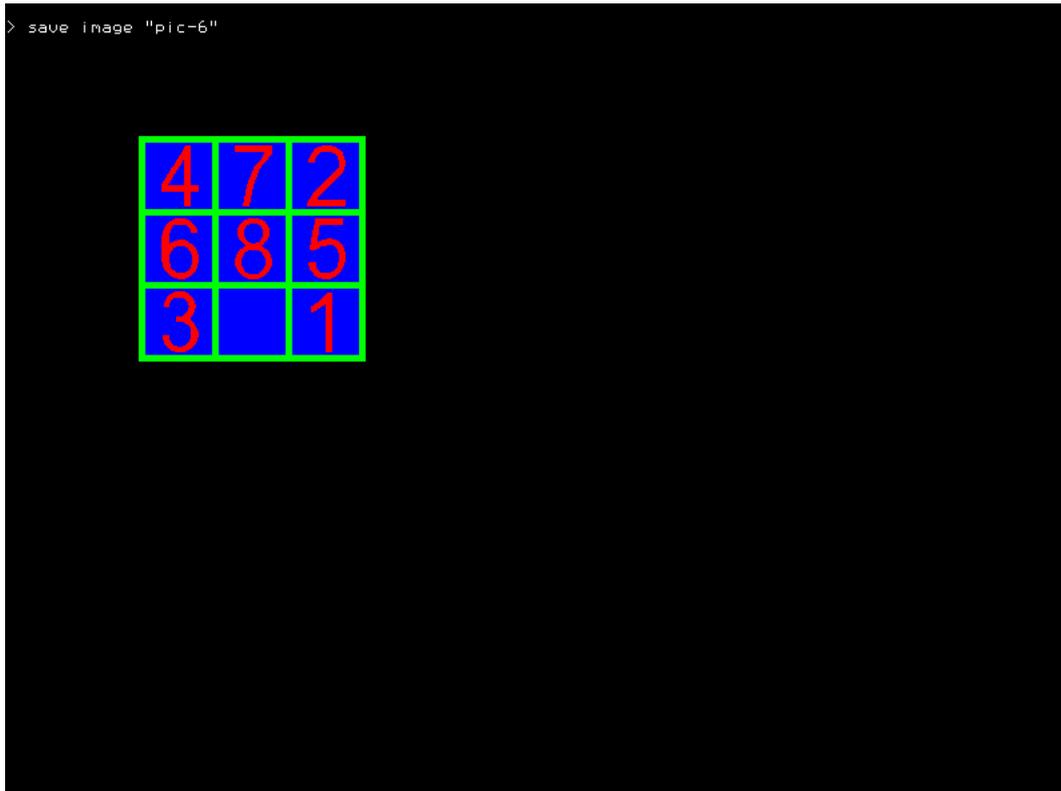
Another example needed. We can use BLIT to create the basis of a sliding block puzzle

```
'set output to page 1  
page write 1  
  
' clear page 1 to blue  
cls rgb(blue)  
'write the numbers 1 to 8 spaced 50 pixels apart  
' font 6 is 32x50 pixels  
for i = 1 to 8  
  text i*50+10,0,str$(i),,6,,rgb(red),-1  
next i  
  
'now we want to randomize the numbers for our grid including the missing one  
dim r(8)=(9,9,9,9,9,9,9,9) 'create an array to hold the positions  
for filled=0 to 8  
  do  
    test=int(rnd()*9) 'get a random number between 0 and 8  
    for j=0 to filled 'check all the filled cells to see if we have already used this number  
      if r(j)=test then test=9 ' if yes then set no-good
```

```

    next j
    if test<>9 then r(filled)=test 'if OK then store the number
loop while r(filled)=9 'loop until that cell is filled
next filled
'Now we will go back to the view page
page write 0
cls
' create a background for the grid
box 100,100,170,170,0,rgb(green),rgb(green)
' now copy the randomised numbers into our grid
for x=0 to 2
  for y=0 to 2
    'get the x position of the randomised number in page 1
    xsource = r(x * 3 + y) * 50
    ' blit the number from page 1 to page 0
    blit xsource, 0, x * 55 + 105, y * 55 + 105, 50, 50, 1
  next y
next x

```



This simplistic example is how Mauro weaves his magic. You can create any useful graphics you want on one or more hidden graphics pages. Then you can use **BLIT** to move parts of the image between them, including potentially onto page 0. And, of course, you can mirror or rotate the parts of the image as you move them.

There are two more **BLIT** commands in the manual: **BLIT READ** and **BLIT WRITE**, but we will leave these until we deal with sprites. Refer to the section on **FRAMEBUFFER** for additional functionality of **BLIT**.

Next post will look at the various graphics modes supported by the CMM2.

## GRAPHICS MODES

This chapter will introduce the 10 graphics resolutions on the CMM2 and the three colour depths in more detail and the use of the MODE command. We will mention the 12-bit modes but will reserve a full explanation of these until a separate tutorial. Also, still to come in this series is more on the PAGE command, using the basic graphics in a more efficient way and, of course, sprites.

A lot of this post will be slightly theoretical, but the more you understand what is going on under the cover of the CMM2 the better you will be able to make the most of its capabilities.

Lets start by being a VGA monitor and asking it to show the Display Information for the signal it is receiving. This isn't affected by the colour depth.

Mode 1 - (800 x 600) : Monitor sees 800x600 @ 60Hz  
Mode 2 - (640 x 400) : Monitor sees 640x480 @ 75Hz  
Mode 3 - (320 x 200) : Monitor sees 640x480 @ 75Hz  
Mode 4 - (480 x 432) : Monitor sees 640x480 @ 75Hz  
Mode 5 - (240 x 216) : Monitor sees 640x480 @ 75Hz

Note: 5 new modes were introduced in MMBasic 5.05.05



Mode 6 - (256 x 240) : Monitor sees 640 x 480 @ 75Hz *(for NES compatibility)*  
Mode 7 - (320 x 240) : Monitor sees 640 x 480 @ 75Hz  
*(this give a full screen display with square pixels on a normal aspect ratio monitor.  
Pages 0-13 are available in this mode with pages 0 and 1 in fast processor memory.)*  
Mode 8 - (640 x 480) : Monitor sees 640 x 480 @ 75Hz  
Mode 9 - (1024 x 768) : Monitor sees 1024 x 768 @ 60Hz *(12-bit colour depth not available in this mode)*  
Mode 10 - (848 x 480) : Monitor sees 640 x 480 @ 60Hz *(widescreen format)*

Note: Mode 10 may not display correctly (or at all) on a standard 4:3 monitor.



All modes work perfectly with monitors that have an aspect ratio of 4:3 or widescreen monitors that can switch to that ratio (most widescreen monitors will do this automatically).

That looks a bit odd - lets explain.

VGA monitors, particularly modern TFT types, have a limited range of signal timings that they will accept and sync up properly. We have chosen to use two that are supported by all monitors – 800 x 600 @ 60Hz using a 40MHz pixel clock and 640 x 480 @ 75Hz using a 31.5MHz pixel clock. All other timings, vertical sync and horizontal sync are defined as a multiple of the underlying clock period. Two additional signal timing settings are used – 640 x 480 @ 60Hz for Mode 10, the 848 x 480 pixel widescreen format for use with 16:9 monitors and 1024 x 768 @ 60Hz for Mode 9, the latter being for the 1024 x 762 pixel mode.

Note: The mode timings above do NOT necessarily indicate the pixel count. See the details above for the actual horizontal and vertical pixel count.



Before going into this in more detail let's answer the question – for modes 2 and 3, why 640x400 and not 640x480?

In the first post we explained the way memory is used for the graphics pages and that 800 x 600 resolution with 1 byte per pixel uses 480,000 out of 512K (512\*1024=524288) bytes of the fastest memory. 640 x 480 would only use 307,200 bytes so no problem BUT, 640 x 480 with 16-bit colour would use 614,400 bytes - more than we have available. 640 x 400 x 2 = 512,000 just inside the 512K limit allowing full colour images to be displayed from the fastest memory.

Moreover, the main retro computers Commodore64, Amiga, Atari-ST all used 320x200 as their main graphics resolution, the same as our mode 3 and 640x400 is exactly double this.

But, I hear you say, the manual says the CMM2 supports 800 x 600x16-bit resolution amongst others. Yes, but in this case the first megabyte of the 3 MBytes of SDRAM is used as video display memory - page 0. The SDRAM isn't as fast as the internal memory but in most cases works perfectly well in this mode and it is great for displaying pictures. With a pixel clock at 40MHz and 2 bytes per pixel the LTDC is reading that memory at 80Mbytes/second. Even more extreme is 12-bit mode. Without going into the explanation here 12-bit mode uses 4 bytes per pixel so 800x600x32-bit now needs memory to be read at 160Mbytes/second and uses 2Mbyte of SDRAM!

You will see from the table above that with the introduction of MMBasic version 5.05.05, five more modes were implemented including 640 x 480 as Mode 8 and in order to cater for the size issue raised above, the 12 and 16 bit variants of 640 x 480 use the SDRAM for their video display memory - page 0. This is also true for other modes that exceed the 512K limit of fast memory.

The SDRAM is also used for MMBasic arrays and things like buffers for playing MP3 and FLAC files. If you set the video mode to 800x600 12-bit resolution and play a MP3 file and run a program then occasionally the monitor may flicker or lose sync. The extent to which this happens is very much monitor dependent and while there may be no issue on your monitor there may be for other people.

Back to the main story. Supported resolutions are :-

**Mode 1** displays 800x600 pixels and the LTDC just reads out the bytes one at a time at 40Mhz. Bytes are not read during line and frame blanking periods (HSYNC and VSYNC) and the LTDC handles this for us.

**Mode 2** displays 640x400 pixels but the monitor sees 640x480. This is achieved by increasing frame blanking by 80 lines split between the top and bottom (front porch and back porch in the jargon). The memory used is 640x400 pixels and this is arranged exactly like we saw for mode 1 earlier, top left to bottom right.

**Mode 4** displays 480x432 pixels for Maximite compatibility. This is achieved by reducing the pixel clock to  $31.5/640*480 = 23.625\text{MHz}$  but maintaining the same absolute durations for line and frame blanking periods. In this case frame blanking is extended by 48 lines split between the top and bottom. The change in pixel clock rate isn't a perfect solution, particularly with LCD monitors. These work by splitting the period between HSYNC pulses and sampling the analog signal at the expected pixel rate. So they are sampling at 31.5MHz and we are changing values at 23.625Mhz. They then need to map this to the physical screen resolution (say 1280 pixels) so, depending on the monitor, you may see slight artefacts.



Notice how the left side of the "0" in "11-05" and "2020" is wider than the right. This is a monitor sampling issue and probably wouldn't occur on an old analogue CRT monitor. The memory used is 480x432 pixels and this is arranged exactly like we saw for mode 1 earlier, top left to bottom right.

**Mode 3** displays 320x200 pixels. You can probably easily guess now how the 320 is created. The pixel clock is halved from 31.5MHz to 15.75MHz. As this is a simple power of 2 there is no sampling issue like we saw with mode 4. However the monitor is still expecting 480 lines and that can't be changed. Unfortunately the LTDC doesn't have the capability to clock out each line of data from memory twice so we have had to replicate each line in the firmware. The memory used is therefore only half that of 640x400 rather than a quarter. As for mode 2 frame blanking is increased by 80 lines split between the top and bottom. The memory used is 320x400 pixels and to get the address of a pixel in memory we now have to compensate for each line being duplicated. e.g. for an 8-bit colour depth:

```
add1% =MM.INFO(page address 0)+ (y * 2) * MM.HRES + x
add2% =MM.INFO(page address 0)+ (y * 2 + 1) * MM.HRES + x
```

**Mode 5** displays 240x216 pixels and I'll leave to the reader to extrapolate from modes 3 and 4 as to how this works.

**Mode 6** displays 256 x 240 pixels and is the format used by Nintendo Entertainment Systems (NES). This mode was introduced to allow games programmers to attempt to emulate some of the great NES games. For a detailed explanation of the NES format, see [NES explanation](#). The monitor still sees 640 x 480 at 75Hz however, similar to mode 3, the pixel clock is reduced, this time by 2.5 times to 12.5MHz with absolute line and frame durations the same. As with mode 3, each line of data is clocked out from memory twice by the firmware.

**Mode 7** displays 320 by 240 pixels – exactly the same timing as mode 3 but without sacrificing the 80 lines. Also, the same constraints on memory usage as mode 3 apply. This, together with mode 8 have the correct 4:3 screen ratio and display nice clean square dots.

**Mode 8** displays 640 by 480 pixels.

**Mode 9** displays 1024 by 768 pixels with a pixel clock rate of 62.5MHz. The data rate required to keep up limits this mode to 8 and 16 bit colour only (12 bit mode requires 4 bytes per pixel – even the very nippy CMM2 can't do this and everything else at once!!)

**Mode 10** displays 848 x 480 pixels and is intended for use on widescreen displays. This mode should not be used for programs to be distributed as not all users may be able to show it, but it is excellent for editing when you have a widescreen monitor.

## Supported colour depths

The CMM2 supports 3 colour depths:-

**8-bit** uses 1 byte per pixel and uses RGB332. That means there are three bits coding the red intensity (8 levels), three for green (8 levels) and two for blue (4 levels). However, as we saw earlier we can change this using the MAP command and function so, for example, we could create a grey-scale map with 256 different "grey" levels.

**16-bit** uses 2 bytes per pixel and uses RGB565. That means there are five bits coding the red intensity (32 levels), six for green (64 levels) and five for blue (32 levels). Although this may seem limited, colour images displayed in RGB565 on a normal monitor are pretty much indistinguishable from those with greater colour depth.

**12-bit** uses 4 bytes per pixel and uses ARGB4444. That means there are four bits coding the red intensity (16 levels), four for green (16 levels) and four for blue (16 levels). In addition there are 4-bits that encode the transparency. i.e. the extent to which the colour masks anything underneath it. Think about paint. If you paint white on top of red you will still see a pink blush after the first coat. Although the white paint is "pure" white it allows some of the red to be seen through it. In the case of ARGB4444 the 4 "A" bits range from 15=solid to 0=totally transparent.

Now, transparency isn't much use if there is nothing to be transparent over. So 12-bit mode has two layers and hence it takes 4 bytes per pixel - more on this later.

In all cases the firmware translates from the colour specification you provide which is in RGB888 or for 12-bit mode ARGB8888 to the internal format for the display colour depth selected. This means the Basic code is identical irrespective of colour depth (except if you use the MAP command in 8-bit mode or are using Maximite compatibility mode).

## The MODE command

The explanation of the mode command in the user manual is very comprehensive so there is little point in reproducing it again here. However, of particular importance is the number of video pages that are available in each mode.

The firmware rounds up the amount of video memory used for any resolution/colour depth to the nearest multiple of 128Kbytes ( $128 \times 1024 = 131072$ ). So Mode 3,8 (320 x 200 x 8-bit) seems like it should need 64000 bytes but remember each line must be duplicated so it really needs 128000 which is less than 131072. This means we can have 4 video pages in the fast internal memory (512K available) and another 24 in the 3Mbytes of SDRAM.

New in version 5.05.05 is OPTION DEFAULT MODE n where n can be 1, 8, or 9. This permanent option sets the video mode for the command prompt, the editor and the filemanager and is the default mode when a program is run.

Remember you can also use OPTION EDIT FONT to adjust the font used in the filemanager and the editor. This should allow the CMM2 to use the cheap 7" TFT screens like [this](#).

This is a permanently saved option but does not cause a reboot so can be used in a program. However, developers should probably not use it in code to be circulated as you will reconfigure a user's CMM2.

OPTION DEFAULT MODE n will always give you the 8 bit colour depth.

The specifications of the colour depth ('bits') are:-

	<b>16-bit</b>	<b>12-bit</b>	<b>8-bit</b>
H/W Pixel Format	RGB565	ARGB4444	RGB332
Colours	65536	4096	256
Transparency	None	16-levels	None
Pages Used	1	2	1
Layers	1	2 + background	1

The display always shows the contents of page 0 (16-bit and 8-bit) and pages 0 and 1 (12-bit). Use PAGE WRITE and PAGE COPY to avoid flashing and tearing artefacts.

For 12-bit colour depth page 0 is the lower level and page 1 the upper so the stack is: background, page 0, page 1 with each one overwriting the previous in turn as defined by the transparency values of each individual pixel.

A summary of the pages available in the different modes is as follows:-

MODE	BIT DEPTH	RESOLUTION	PAGE MEMORY	TOTAL PAGES	FAST PAGES	EXT PAGES
1	8	800 x 600	480000	7	&H24000000	&HD0000000, \$HD0080000, &HD0100000, &HD0180000, \$HD0200000, &HD0280000
1	12	800 x 600		3	-	&HD0000000, \$HD0100000, &HD0200000
1	16	800 x 600	960000	3	-	&HD0000000, \$HD0100000, &HD0200000
2	8	640 x 400	256000	14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
2	12	640 x 400		7	&H24000000	&HD0000000, \$HD0080000, &HD0100000, &HD0180000, &HD0200000, &HD0280000
2	16	640 x 400	512000	7	&H24000000	&HD0000000, \$HD0080000, &HD0100000, &HD0180000, &HD0200000, &HD0280000
3	8	320 x 200	128000 **	28	&H24000000, &H24020000, &H24040000, &H24060000	&HD0000000, \$HD0020000, &HD0040000, &HD0060000, &HD0080000, &HD00A0000, &HD00C0000, \$HD00E0000, &HD0100000, &HD0120000, &HD0140000, &HD0160000, &HD0180000, \$HD01A0000, &HD01C0000, &HD01E0000, &HD0200000, &HD0220000, &HD0240000, \$HD0260000, &HD0280000, &HD02A0000, &HD02C0000, &HD02E0000
3	12	320 x 200		14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
3	16	320 x 200	256000 **	14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
4	8	480 x 432	207360	14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
4	12	480 x 432		7	&H24000000	&HD0000000, \$HD0080000, &HD0100000, &HD0180000, &HD0200000, &HD0280000
4	16	480 x 432	414720	7	&H24000000	&HD0000000, \$HD0080000, &HD0100000, &HD0180000, &HD0200000, &HD0280000
5	8	240 x 216	103680 **	28	&H24000000, &H24020000, &H24040000, &H24060000	&HD0000000, \$HD0020000, &HD0040000, &HD0060000, &HD0080000, &HD00A0000, &HD00C0000, \$HD00E0000, &HD0100000, &HD0120000, &HD0140000, &HD0160000, &HD0180000, \$HD01A0000, &HD01C0000, &HD01E0000, &HD0200000, &HD0220000, &HD0240000, \$HD0260000, &HD0280000, &HD02A0000, &HD02C0000, &HD02E0000
5	12	240 x 216		14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
5	16	240 x 216	207360 **	14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
6	8	256 x 240	122880 **	28	&H24000000, &H24020000, &H24040000, &H24060000	&HD0000000, \$HD0020000, &HD0040000, &HD0060000, &HD0080000, &HD00A0000, &HD00C0000, \$HD00E0000, &HD0100000, &HD0120000, &HD0140000, &HD0160000, &HD0180000, \$HD01A0000, &HD01C0000, &HD01E0000, &HD0200000, &HD0220000, &HD0240000, \$HD0260000, &HD0280000, &HD02A0000, &HD02C0000, &HD02E0000
6	12	256 x 240		14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
6	16	256 x 240	245760 **	14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
7	8	320 x 240	153600 **	14	&H24000000, &H24040000	&HD0000000, \$HD0040000, &HD0080000, &HD00C0000, &HD0100000, &HD0140000, &HD0180000, \$HD01C0000, &HD0200000, &HD0240000, &HD0280000, &HD02C0000
7	12	320 x 240		9	&H24000000	&HD0000000, \$HD0060000, &HD00C0000, &HD0120000, &HD0180000, &HD01E0000, &HD0240000, &HD02A0000
7	16	320 x 240	307200 **	9	&H24000000	&HD0000000, \$HD0060000, &HD00C0000, &HD0120000, &HD0180000, &HD01E0000, &HD0240000, &HD02A0000
8	8	640 x 480	307200	9	&H24000000	&HD0000000, \$HD0060000, &HD00C0000, &HD0120000, &HD0180000, &HD01E0000, &HD0240000, &HD02A0000
8	12	640 x 480		5	-	&HD0000000, \$HD00A0000, &HD0140000, &HD01E0000, &HD0280000
8	16	640 x 480	614400	5	-	&HD0000000, \$HD00A0000, &HD0140000, &HD01E0000, &HD0280000
9	8	1024 x 768	786432	4	-	&HD0000000, \$HD00C0000, &HD0180000, &HD0240000
9	12	N/A	N/A		N/A	N/A
9	16	1024 x 768	1572864	2	-	&HD0000000, \$HD0180000
10	8	848 x 480	407040	7	&H24000000	&HD0000000, \$HD0080000, &HD0100000, &HD0180000, \$HD0200000, &HD0280000
10	12	848 x 480		4	-	&HD0000000, \$HD00E0000, &HD01C0000, &HD02A0000
10	16	848 x 480	814080	4	-	&HD0000000, \$HD00E0000, &HD01C0000, &HD02A0000



Note: Page Memory values with \*\* after them indicate that lines are duplicated meaning twice as much memory is used as would be expected. See explanation on previous page..

As before, MM.INFO(PAGE ADDRESS n) can give us the address of any of the pages. Page 0 is **always** the display page but when programming for performance there is an advantage to using a page in the faster memory for other uses.

The syntax of the mode command is:

```
MODE mode, bits [, bg [, int]]
```

We now know all about the mode and the bits and what the firmware is doing behind the scenes to create the images.

We will deal with **bg** which is only used in 12-bit mode later but for now we can look at the "int" parameter.

**int** is the name of an MMBasic interrupt routine which is called immediately after the last active row of data is clocked out. This is before the actual frame-blanking pulse (actually it occurs at the beginning of the front-porch period which bizarrely is at the end of the frame whereas the back porch is at the beginning - WTF?).

This is important to understand because before any new data is read from the memory the front porch period, the actual VSync frame pulse period and the back porch period will all have elapsed. In CMM2 terms this can be quite a long time.

For Modes 2 and 3, there are 400 lines displayed and the complete frame has 500 line periods so there is a dead time of 2.66 milliseconds when the display can be written and no screen artefacts will ever appear.

For Mode 1 this period is 0.74mSec and for modes 4 and 5 it is 1.65mSec

You can actually effectively extend this period if you arrange your code to write from the top of the screen downwards. As long as your writes keep ahead of the LTDC reading the memory then there will be no artefacts or tearing.

Lets build a very simple digital clock with big digits using the frame interrupt and show that there are never any screen artefacts

```
' set to 320x200 mode, with a frame interrupt
' NB the background colour must be specified but is ignored in this mode
mode 3, 8, 0, frameint
' set a global variable that will be updated by the frame interrupt
fi%=0
do
  do
    loop while fi%=0 'wait until the frame interrupt has happened
    ' write out the time in the middle of the screen
    text mm.hres\2, mm.vres\2, time$, CM, 6,, RGB(red)
    ' prime ready for the next interrupt
    fi%=0
  loop
sub frameint
  fi%=1
end sub
```



Note that I am not doing any screen updates in the interrupt routine - you wouldn't do that would you? Interrupts must always be kept short and the best approach is always to set a flag that can be processed in the main program.



```

        x = x + w
    NEXT n
NEXT w
' white and red border to check that image fits on monitor
BOX 0,0,wd,ht,3,c(7)
BOX 1,1,wd-2,ht-2,1,c(5)
' title
IF wd > 600 THEN
    TEXT wd/2,ht/2, imgtitle$,cm,4,1
    TEXT wd/2,ht/2+25, imgRes$,cm,4,1
ELSE
    TEXT wd/2,ht/2, imgtitle$,cm,1,1
    TEXT wd/2,ht/2+25, imgRes$,cm,1,1
ENDIF
' show the new image
PAGE COPY 1 TO 0 ,B
ENDIF
' wait for keypress
DO
    k$ = INKEY$
LOOP UNTIL k$<>""
'
SELECT CASE k$
CASE "Q","q"
    EXIT DO
CASE "P","p"
    fname$ = MID$(imgtitle$,2)+".bmp"
    TIMER = 0
    SAVE IMAGE fname$
    PAGE WRITE 0
    TEXT wd/2,ht/2,"Saved as "+fname$+" in "+STR$(TIMER/1000,3,2)+" Sec" ,cm,1,1
    DO
        k$ = INKEY$
        LOOP UNTIL k$<>""
CASE CHR$(128) ' up arrow
    m = m - 1
    IF m < 1 THEN m = maxMode
CASE CHR$(129) ' down arrow
    m = m + 1
    IF m > maxMode THEN m = 1
CASE CHR$(131) ' right arrow res up
    cd = cd + 4
    IF cd > 16 THEN cd = 8
    IF m = 9 AND cd = 12 THEN cd = 16 ' skip 12 bit for mode 9
CASE CHR$(130) ' left arrow res down
    cd = cd - 4
    IF cd < 8 THEN cd = 16
    IF m = 9 AND cd = 12 THEN cd = 8 ' skip 12 bit for mode 9
CASE "+" ' ratio plus
    IF a < 1.4 THEN a = a + 0.01
    keepmode = 1
CASE "-" ' ratio minus
    IF a > 0.75 THEN a = a - 0.01
    keepmode = 1
CASE ELSE ' same as down arrow
    m = m + 1
    IF m > maxMode THEN m = 1
END SELECT
LOOP
setmode defaultMode ' restore original mode before ending program
PAGE WRITE 0
CLS
END

SUB setmode dotMode AS FLOAT
LOCAL INTEGER mm, md
' use float returned by mm.info(mode) to set MODE
mm = INT(dotmode)
md = (dotmode - mm)*100
IF md > 20 THEN md = md/10
MODE mm, md
END SUB

```

## THE PAGE COMMAND

---

In the second post in this thread we met the PAGE COPY command and now understand how to move information from one page to another. This can be the complete page using PAGE COPY or part of a page using BLIT.

We have also investigated the PAGE WRITE command that allows us to select the page where graphics commands will write their data.

In this post we will look at the two other PAGE sub-commands - PAGE SCROLL and PAGE STITCH

### PAGE SCROLL

This command does what it says. It scrolls a selectable page (doesn't need to be page 0) horizontally and/or vertically by a selectable amount. In addition it allows you to control what happens to the area vacated by the action of the scroll.

```
PAGE SCROLL pageno, x, y [,fillcolour]
```

Some examples:

```
LOAD JPG "bondi":D0:LOOP
```

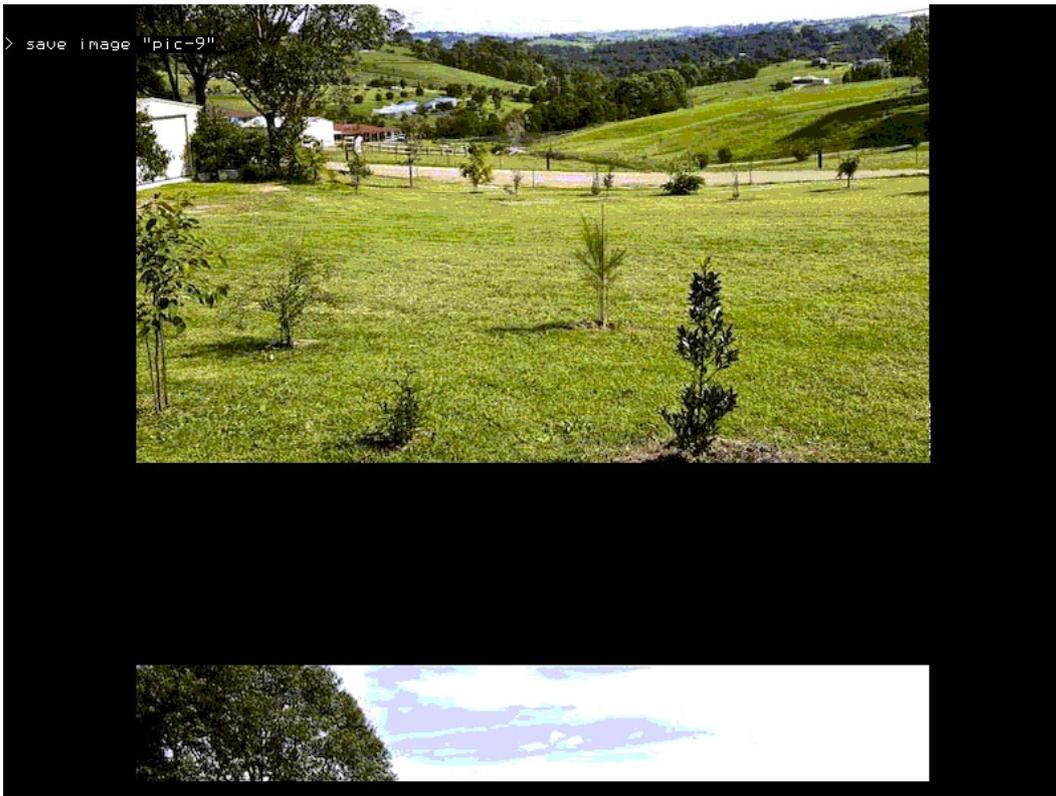
load a sample jpg in 800x600x16 resolution



Now lets try something else:-

```
LOAD JPG "bondi":PAGE SCROLL 0, 100,100:DO:LOOP
```

scroll the picture 100 right and 100 up, moving the scrolled off area to the now vacated part of the screen. Note that the vertical move always happens first. If you need the sideways move first you will need to use the command twice.



```
LOAD JPG "bondi":PAGE SCROLL 0, 100,100,-1:DO:LOOP
```

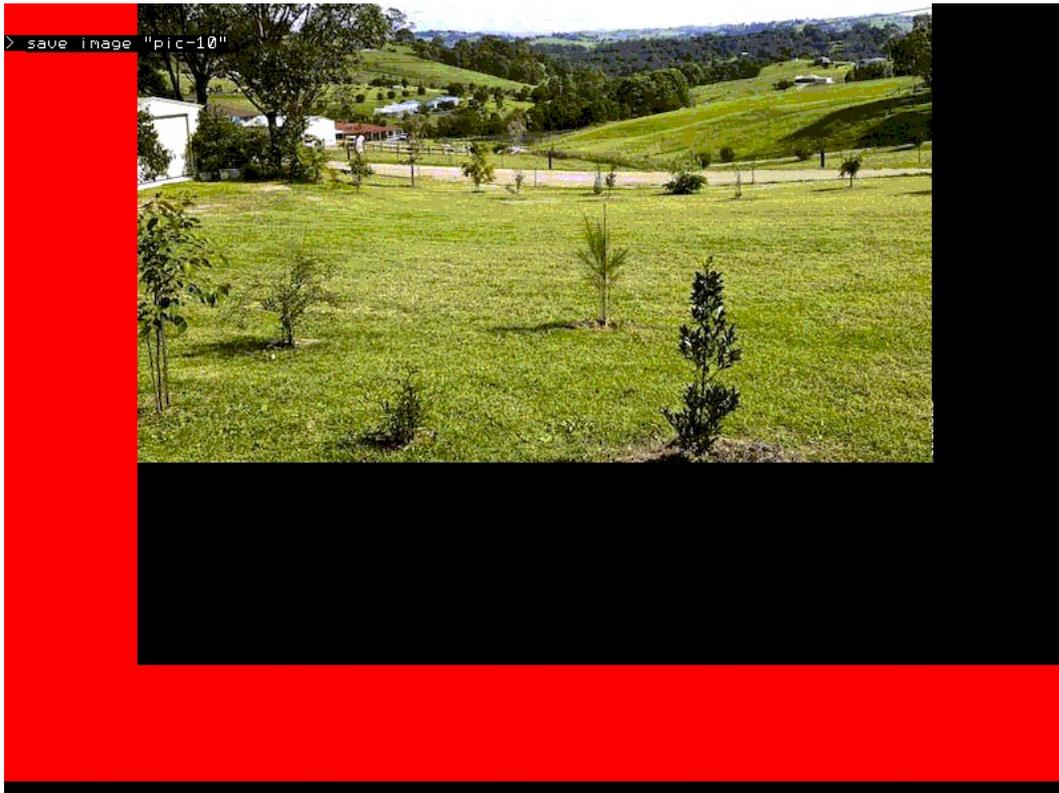
This will scroll the picture 100 up and 100 right, leave the vacated off area as-is



And again:-

```
LOAD JPG "bondi":PAGE SCROLL 0, 100,100,RGB(RED):DO:LOOP
```

scroll the picture 100 up and 100 right, replace the vacated area with a fixed colour



Time to load a 800x600 full colour jpg was 347 milliseconds. Worst case time to scroll the image with image wrap was 125 milliseconds. All other lower resolutions or lower colour depths will be faster.

Of course moving a 800x600 image with 2 bytes per pixel takes a little time but try this example program at 320x200x16

```
option explicit
option default none
dim x%(9),xd%(9)
dim y%(9),yd%(9)
dim integer i,c,f, xp, yp,k,t
dim float s

' Work in 320x200 resolution RGB565
mode 3,16

cls

'create the coordinates of a star outline
for i=0 to 9 step 2
x%(i)=-((SIN(rad(i*36)))*20)
y%(i)=-((COS(rad(i*36)))*20)
x%(i+1)=-((SIN(rad((i+1)*36)))*7.6)
y%(i+1)=-((COS(rad((i+1)*36)))*7.6)
next i

'Set to write to page 1 and clear it
page write 1
cls

'create 40 random stars on page 1 using the polygon fill command
do
c=rnd()*255 + ((rnd()* 255)<<8) + ((rnd()* 255)<<16)
f=rnd()*255 + ((rnd()* 255)<<8) + ((rnd()* 255)<<16)
xp=rnd()*mm.hres
yp=rnd()*mm.vres
s=rnd()
t=0
for i=0 to 9
xd%(i)=x%(i)*s+xp
```

```

yd%(i)=y%(i)*s+yp
if xd%(i)<0 or yd%(i)<0 or xd%(i)>=MM.Hres or yd%(i)>=MM.Vres then t=1
next i
if t=0 then
  polygon 10, xd%(), yd%(), c, f
  k=k+1
endif
loop until k=40
k=0
'

'Now lets see how fast scroll really is
do
  page copy 1 to 0,d 'copy page 1 to page 0 during frame blanking
  page scroll 1,2,1 'scroll page 1 immediately after
loop

```

For many simple early games they used a seamless background that was a single image that could be scrolled horizontally, vertically, or even both with no obvious join.

Try the attached with

```
load jpg "seamless":do: page scroll 0,1,1:loop
```

[seamless.zip](#)

Now you should be able to see how to create a slideshow with one picture smoothly replacing another from the right.

I'll provide some pseudo code for you to complete and try for yourself.

```

load an image to page 1
load an image to page 2
loop
  copy page 1 to page 0 during frame blanking
  scroll page 1 left by a "n" pixels
  BLIT n pixels from the left of page 2 to the right of page 1
  scroll page 2 left by "n" pixels
  If you have just used all of page 2 then load the next image to page 2
end loop

```

## PAGE STITCH

This is really only a short form way of doing what was proposed above with scroll and BLIT

```
PAGE STITCH frompage1, frompage_2, topage, offset
```

This says we are going to take columns from the right side of `frompage1` and columns from the left side of `frompage2` and copy them to `topage` as a single action. The `offset` parameter determines how many columns are going to be taken from `frompage2` - perhaps the `offset` parameter was badly named?

This single command makes it very easy to create a horizontally scrolling background across a number of separate images.

[This video](#) shows how fast this can take place.

The example code presented below slows this down by syncing everything up with frame blanking to get a very smoothly moving image. As the monitor frame rate is 75Hz and the scroll is happening 2 pixels at a time the movement is limited to 150 pixels per second. Remove the ",b" parameter in the page copies to see things at maximum speed.

```

' test stitch
' Run at 320x200x16 bit mode
mode 3,16

'clear the first 7 video pages
For i= 0 to 6: page write i:cls:next i

'now write each part of the background image to a separate page
' doing this at the beginning ensures the program won't stall waiting for images to be loaded
page write 2
load png "part01",,,15
page write 3
load png "part02",,,15
page write 4
load png "part03",,,15
page write 5
load png "part04",,,15

' Main program loop
do

```

```

'merge pages 2 and 3 stepping 2 pixels at a time
for i=0 to MM.Hres step 2
  page stitch 2,3,6,i
  page copy 6 to 0,b
next i

' now page 2 is used up so merge pages 3 and 4
for i=2 to MM.Hres step 2
  page stitch 3,4,6,i
  page copy 6 to 0,b
next i

' finally merge pages 4 and 5
for i=2 to MM.Hres step 2
  page stitch 4,5,6,i
  page copy 6 to 0,b
next i

'we are now at the rightmost side of the composite image so we can start moving back left
for i= MM.Hres to 0 step -2
  page stitch 4,5,6,i
  page copy 6 to 0,b
next i
for i= MM.Hres-2 to 0 step -2
  page stitch 3,4,6,i
  page copy 6 to 0,b
next i
for i= MM.Hres-2 to 0 step -2
  page stitch 2,3,6,i
  page copy 6 to 0,b
next i

' Once all the way left then start again
loop

```

Here are the images used in this demo [Parts.zip](#)

Hopefully you can see from this short post that it is very easy to create and manipulate moving images on the screen. Of course, this needn't be anything to do with games. Scrolling graphs etc. are just as easy.

## 12-bit Mode Graphics

The 12-bit graphics modes are one of the most powerful capabilities of the Colour Maximize 2. They enable us to create semi-transparent objects floating over a background like the [ghost demo](#) or the effect of parallax like the [stars demo](#). Note in the latter how the nearer stars are moving faster relative to the spaceship than those further away.

The way that it works is using layers. The concept will be very familiar to those of you doing manipulations of images with any of the more capable graphics packages.

In 12-bit colour mode the CMM2 supports three layers. This is a property of the processor chip itself.

The bottom layer is just a single colour background and to set the colour we specify it in the MODE command.

```
MODE 3,12,RGB(blue)
```

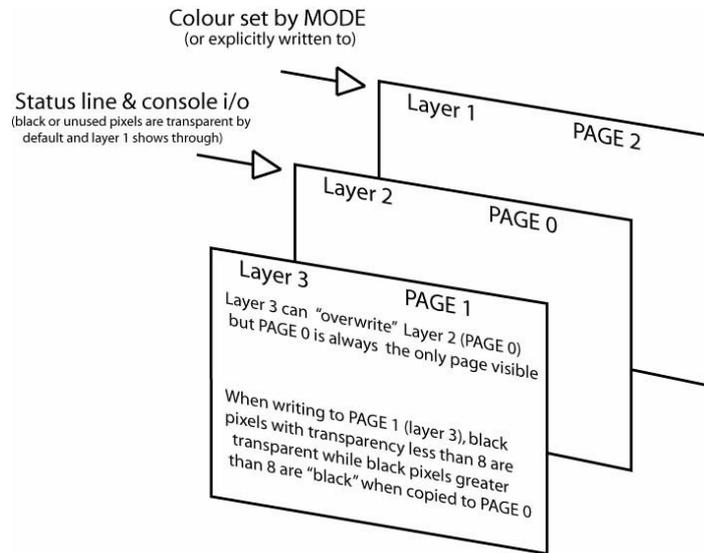


Will put a blue over the screen. Note how, as we are at the command prompt, the status line and command prompt overwrite the background, but it shows through the unset pixels in each character.

To change the background colour you can just use the mode command again with the new colour. Other information on the screen won't be affected (NB: V5.05.03 or greater).

The status line and command prompt are writing to the second layer of the three layers and the data for this layer is stored in page 0.

The data for the third and top layer is stored in page 1. Unless pixels are set to have some transparency (more on this later) any information written to layer 3 (page 1) will block out layers 1 and 2.

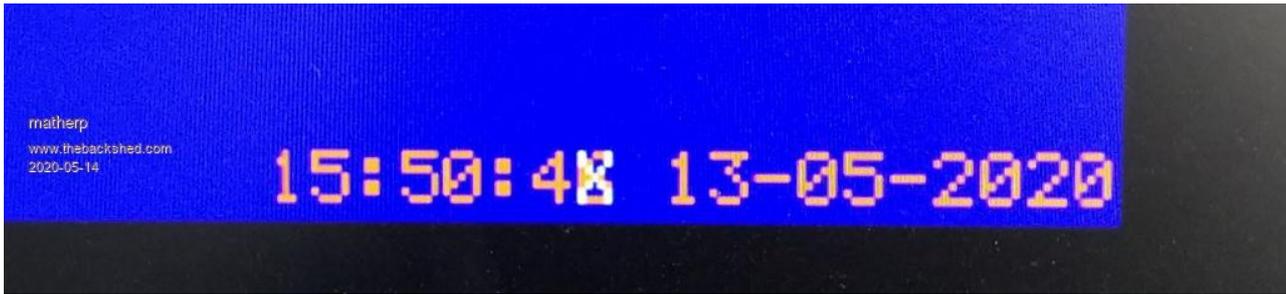


The 'layers' referred to above are NOT sprite layers - see further on how this applies to sprites.

We can prove this. Type:

```
PAGE WRITE 1: TEXT 247,190,"X":page write 0
```

We have to do the command as a single line to avoid the status updating while we are writing to page 1



Note how the "X" appears to have overwritten the brown seconds units but the seconds are still incrementing underneath and pixels that are not covered by the "X" show normally.

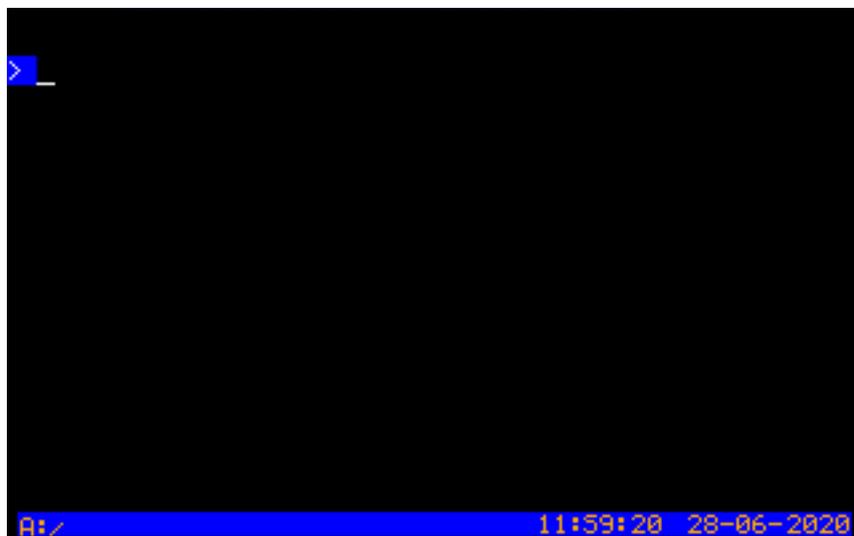
Now type:

```
PAGE WRITE 1: TEXT 247,190," ":page write 0
```

The "X" disappears and the seconds continue incrementing normally. Black, ie. the pixels in text that aren't set, defaults to being a completely transparent colour.

Now type

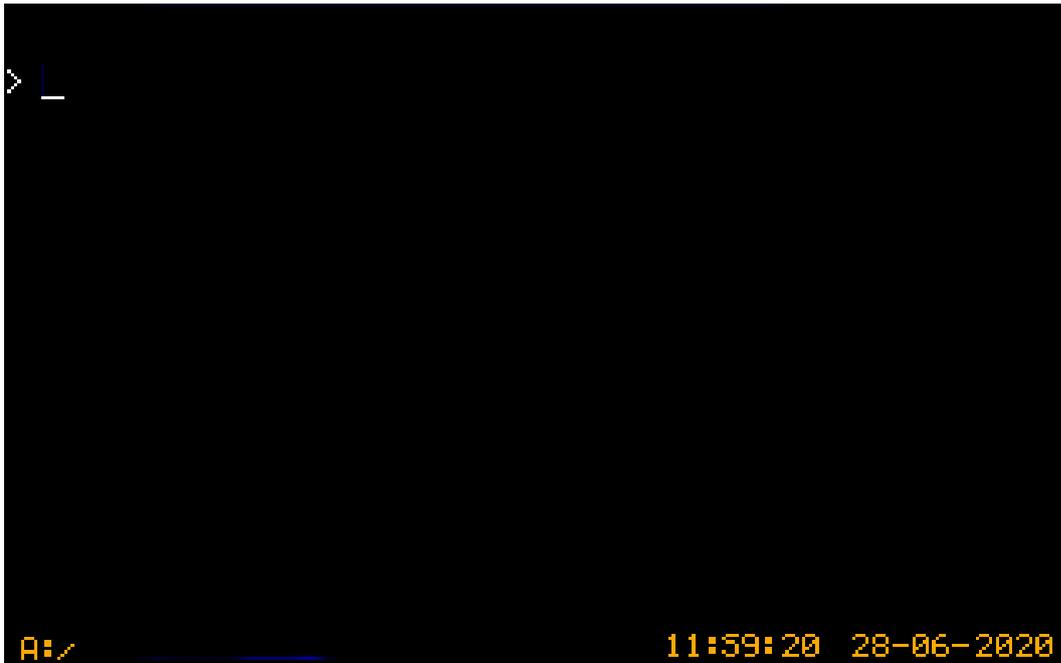
```
CLS RGB(black,15)
```



Check out the RGB function in the user manual. We have asked the system to clear the screen but to use a non-transparent black. Valid transparency values are from 0 to 15 where 15 is a solid colour. However, the default background colour is fully transparent black so the characters associated with the command prompt and the status line still show the blue colour.

Now try:

```
COLOUR RGB(WHITE), RGB(BLACK,15)
```



We have set the background colour to solid black rather than transparent black and now the blue colour on layer 1 has gone completely (NB: V5.05.03 or greater).

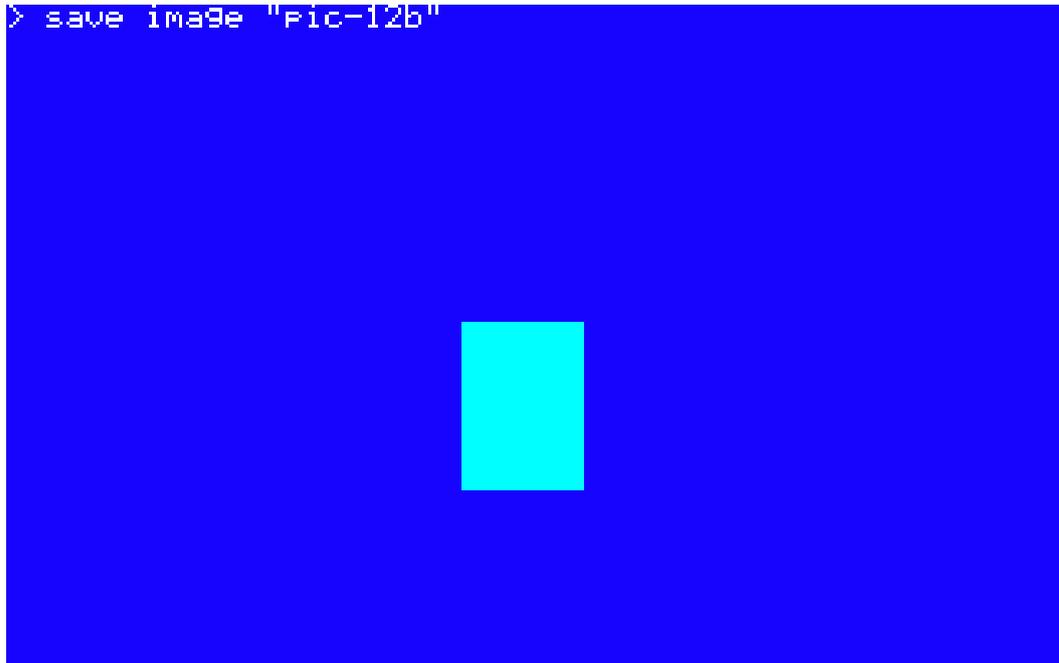
Lets play a bit more with transparency

```
colour rgb(white),rgb(black,15)
box 200,100,50,50,,rgb(blue),rgb(blue)
page write 1
box 225,100,50,50,,rgb(green,8),rgb(green,8)
```



The blue square on layer 2 (PAGE 0) is partially overwritten by the green square on layer 3 (PAGE 1) which is set to 50% transparency. Where the squares overlap we get CYAN.

We could do the same thing with layer 2 and the background layer 1



That should give you a good understanding of 12-bit mode (ARGB4444) and how the "A" transparency can be used to create interesting effects. The most important thing to note is that in 12-bit mode the CMM2 is displaying information from two pages at the same time (PAGE 0 and PAGE 1) plus a fixed background colour. Information on PAGE 0 will overwrite the background depending on the transparency of each individual pixel on PAGE 0. A transparency of 0 will have no effect. A transparency of 15 is a solid colour and will overwrite completely. Information on PAGE 1 will overwrite both the background and PAGE 0 depending on the transparency of each individual pixel.

We will revisit this with a more interesting example when we look at sprites.

However, the next post will look in more detail at loading and manipulating pictures.

Editors Note: Peter has posted some additional information on PAGEs and layers included here as follows:-

From Peter - *"There is clearly confusion here so I'll have another go at explaining."*

Pages are areas of memory that are sized as defined by the resolution. So each page for mode 1,8 is an area of memory 800 x 600 x one byte. Mode 1,16 is 800 x 600 x two bytes.

In all modes, PAGE 0 is shown on the display. In 12-bit modes PAGE 1 sits on top of PAGE 0 and is also displayed in which case PAGE 0 pixels show through to the extent that PAGE 1 pixels have some level of transparency (see the diagram on page 26 of this document.)

Use MM.INFO(MAX PAGES) to tell you the highest page number you can use for any resolution. There is 3.5Mbytes of RAM allocated to video pages and this is allocated in units of 128KBytes.

PAGE COPY simply takes the memory contents of a page and copies it to another page

All graphics output (including sprite commands) goes to the page set by the most recently executed PAGE WRITE command except for a small number of commands that explicitly allow you to define it (e.g. PAGE SCROLL)

When multiple pages are in use it is essential that all sprite commands that change the page act on the same page. For reasons that will become clear it makes sense to allocate a page other than 0 or 1 for sprite usage. I suggest you standardise on PAGE 2. This means before any sprite write operation you should set PAGE WRITE 2.

Sprites are entirely implemented in code. There is no supporting hardware. To move a sprite the firmware first redraws the stored background of the page where the sprite is drawn, then stores the background of where it will move to, then draws the sprite. If the sprites are drawn on PAGE 0 then unless you use a technique to control when you move a sprite relative to frame blanking (there are two ways of doing this documented in the manual) you will see flashing which is why it is better to use sprites on a non-displayed page.



\*\*\* So, a few rules to follow! \*\*\*

It is poor programming to show a sprite repeatedly in your process loop unless something has changed (position, orientation, sprite image changed by a READ).

Don't use **PAGE SCROLL** on the page that contains sprites. You must use the **SPRITE SCROLL** command as this understands sprites. It is however less efficient and a **PAGE COPY**, then **PAGE SCROLL** may be better.

Sprite layers have nothing to do with pages.

Sprite layers are used in just three ways:

1. Sprites on sprite layer 0 will move with the background of the page on which they are drawn when the **SPRITE SCROLL** command is used
2. A sprite move will only trigger a collision with a sprite on the same sprite layer or sprite layer 0
3. A sprite scroll can trigger a collision between a layer 0 sprite and a sprite on another layer
4. If you want to hide a sprite, it is down to you to know if there are any sprites that overlap it and have been drawn after it, in which case you must hide them first then redraw them after hiding the intended sprite.

So a typical algorithm for a sprite "killing" another is as follows:

```
sprite show weaponsprite, x,y,n 'this move triggers a collision

sub sprite interrupt
  local killedsprite=sprite(C,S,1)
  sprite hide weaponsprite
  sprite hide killedsprite
  sprite show weaponsprite, x, y, n
end sub
```

## IMAGES

In this post we will look at the pros and cons of the image formats supported by the CMM2: JPG, GIF, BMP and PNG.

We will start by putting the CMM2 into MODE 2,16 ie. 640x400 RGB565 and for testing purposes I have saved the same image in all 4 formats.

(Editors Note: The images shown are my own as I did not have access to those used in the original posts. The code examples however, refer to the original images. You could substitute your own images appropriately sized.)



Editors Note: File sizes and load times for the "caja" picture are from the original post as given below. I did not have access to the image file Peter used.

```
> list files "caja.*"
A:/
18:50 14-05-2020      768054  caja.bmp
18:50 14-05-2020      217064  caja.gif
18:49 14-05-2020      233823  caja.jpg
18:51 14-05-2020      506373  caja.png
0 directories, 4 files
>
```

And times to load in milliseconds

```
> mode 2,16
> timer=0:load bmp "caja":?timer
541.699
> timer=0:load gif "caja":?timer
728.09
> timer=0:load jpg "caja":?timer
216.345
> timer=0:load png "caja":?timer
509.873
```

Summarising the different file types then:-

JPG files - **JPGs are small and load much faster than anything else.**

The reasons for this are twofold; first there is less data to read from the SDcard, second the STM32H743 includes a hardware JPG decoder so the CPU isn't used.

So we use JPG s for everything? perhaps not.....

There are two main downsides with the JPG format on the CMM2:-

1. JPG s do not support transparency, and
2. JPG s cannot be bigger than the current display resolution.

If we tried to load an 800x600 JPG while in MODE 2 we would get an error. This is because the hardware decoder uses information from the way we have set up the page in LTDC to determine how to decode the image. We would also get an error even with a 640x400 JPG if we tried to load it with an x, y offset other than 0.

```
LOAD JPG file$ [, x, y]
```

One third limitation of the JPG support on the CMM2 is that it does not support progressive encoding. This is a technique that was developed for web use where a picture would appear fast but with low resolution and then the detail would "progressively" appear. This limitation is easily overcome by using any decent graphics program on your computer to save the image normally.

What about PNG files? - **The only format to support transparency so the only choice in some applications**

Reasonably fast to load but:-

1. Decoding PNG files eats memory.  
Even with 5MB available we cannot decode a 800x600 image so 640x400 is the biggest that can be practically supported;
2. PNG cannot be bigger than the current display resolution.  
If we tried to load a 640x400 JPG while in MODE 3 we would get an error.  
We would also get an error even with a 320x200 PNG in MODE 3 if we tried to load it with an x,y offset other than 0, and
3. The CMM2 only supports RGB888 and ARGB8888 formats - files are comparatively large

Next up, GIF files - **The only format to support [animations](#)**

GIF files can be bigger than the screen format or overlap the screen edge - GIF files are small but:-

1. slow, and
2. Only 256 colours used out of the palette of 65536

Finally BMP files - **The safe option**

BMP files image can be bigger than the screen format or overlap the screen edge

```
SAVE IMAGE 'saves BMP files
```

No practical limit on the image size, but of course only part of a very large image can be displayed, but:

full-colour BMP files are 3 bytes per pixel + header ie. large

A BMP image loads a line at a time from the bottom up which may not be what you want.

I'll add to this post tomorrow and look at IMAGE RESIZE and IMAGE ROTATE

I'll just finish off the previous post as promised.

IMAGE RESIZE and IMAGE ROTATE use some fairly complex math to scale and rotate images. They are therefore comparatively slow compared to actions like page copies but useful nevertheless. An example of rotation is available [here](#). In fact this was done on an early release of the code and various enhancements have speeded it up somewhat since then.

As an example of their use lets take an 800x600 image that is in landscape and scale and rotate it to a 600x450 image in portrait. This then gives us the worst case timings for the two functions - biggest image, highest colour depth.

#### [tigerside800.zip](#)

```
'Set to 800x600 16 bit mode
' NB to use this mode it must be unlocked first with OPTION MODES UNLOCKED
mode 1,16

'set output to page 1
page write 1

' lets see how long it all takes
timer=0

'load our 800x600 image to page 1
load jpg "tigerside800"
page write 0
print @(0,0)"Time to load JPG"
print timer/1000, "seconds"
timer=0
page write 1

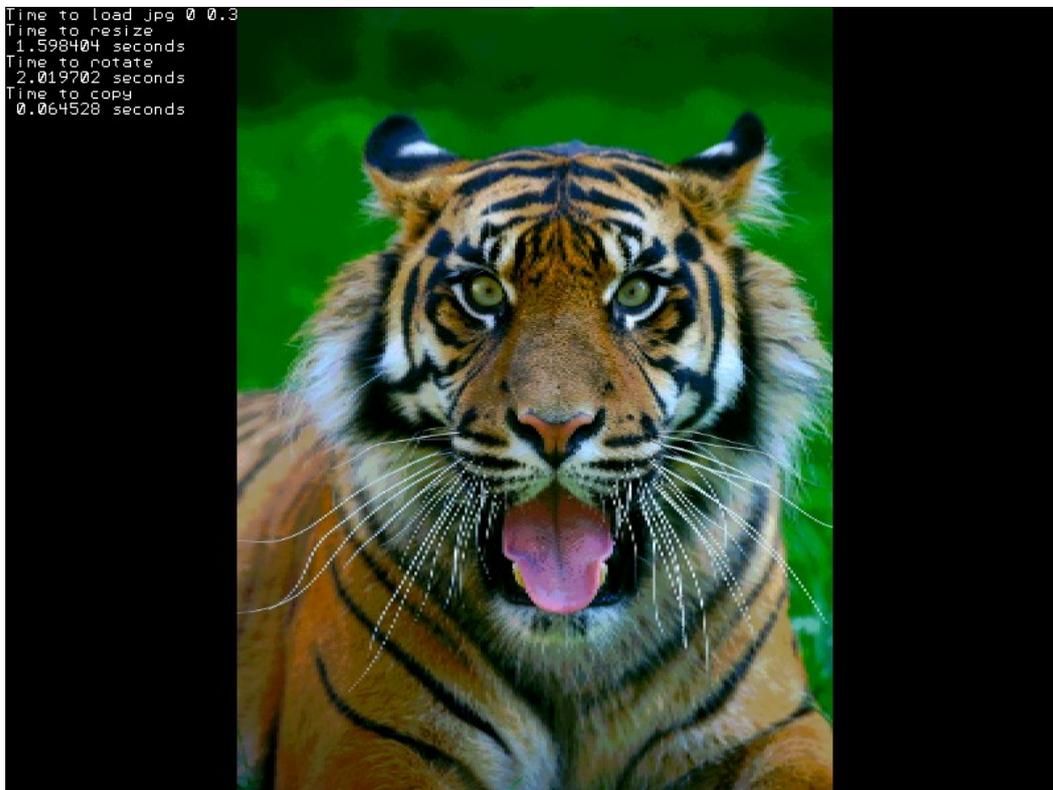
'When we rotate the image it can't be more than 600 pixels high so resize it
'Check the manual for syntax but we are specifying to resize it and place the resized image
centrally over the old image
image resize 0,0,800,600,100,75,600,450
page write 0
print "Time to resize"
print timer/1000, "seconds"
timer=0
page write 1

'Now we need to rotate the image
'In the case of the example this is a rotation of 270 degrees clockwise to get it the correct way up
' Note that the rotated image has to fit into a new area no bigger than the area to be rotated
'So we need to rotate a square area as big as the biggest dimension we need in the result
image rotate 100,0,600,600,100,0,270
page write 0
print "Time to rotate"
print timer/1000, "seconds"
timer=0

'Now we can go back to page 0
page write 0

' Copy the part of page 1 containing the rotated image to page 0 using BLIT
blit 175,0,175,0,450,600,1

print "Time to copy"
print timer/1000, "seconds"
do
loop
```



One important thing to note is that resizing and rotating by definition lose some information and accuracy. For this reason if you want to do something like the rotating image in the video you should do each rotation from the original rather than incrementally rotating the previous result otherwise the image will quickly degenerate.

Two variations on image rotate and resize are `IMAGE ROTATE_FAST` and `IMAGE_RESIZE_FAST`.

These commands have exactly the same syntax as the originals but use a nearest neighbour algorithm rather than bi-linear interpolation. As a result they are much faster (6x for rotate and 20x for resize) but, of course, the resulting image quality will not be as good. Additionally, `IMAGE ROTATE_FAST` and `IMAGE_RESIZE_FAST` both have a new optional parameter - `dontcopyblack`

```
IMAGE RESIZE x, y, width, height, new_x, new_y, new_width, new_height [,page_number]
[,dontcopyblack]
```

```
IMAGE ROTATE x, y, width, height, new_x, new_y, angle! [,page_number] [,dontcopyblack]
```

This option, `dontcopyblack` defaults to 0, i.e. black pixels are treated like any other. If it is set to 1 then any black pixels in the created image will not be written.

Next post we start on sprites by learning what is a sprite and the various ways to create them.

## SPRITES AND HOW TO CREATE THEM

Sprites are somewhat complex and in any case they are just something reserved for games programmers. Wrong on both counts as we will see.

The concept of a sprite is simple. It is an image stored in the computer memory. It can be written to the screen like any other image. However, what makes a sprite special is that any decent implementation of sprites will have the capability of storing what is on the screen in the area that the sprite will be displayed and this happens automatically without additional code from the user of the sprite.

What this means is that it is possible to write code that displays a sprite and then remove it and the display will be restored exactly like it was before. Perhaps more usefully you can display a sprite and then repeatedly move it on the screen and with each move the original screen image will be restored to create the illusion of the sprite moving across and in front of the original screen image. We will show simple examples of this later in the next post.

Before proceeding to load some sprites, it is important to understand how sprites are stored in the CMM2 memory. Each sprite when created has two areas of memory allocated to it. Both are the size needed to store a bounding rectangle with width and height equal to the maximum dimensions of the sprite. One of the memory areas holds the sprite image itself. The other is an empty slot ready to hold the image that was on the screen when the sprite is displayed. In both cases the number of bytes is also dependent on the current colour depth of the video mode in use - either one or two bytes per pixel.

The CMM2 supports up to 64 sprites numbered 1 to 64 and the number is used whenever we want to refer to a particular sprite. Setting up constants at the top of a program would be a good way to make your code more readable when using sprites

```
CONST Ogre = 1
CONST red_elf = 2
```

Now we know what a sprite is we need to know how to create one. On the CMM2 there are four methods of creating a sprite:-

1. Load a sprite from an ascii file in the same way as was done on the original Colour Maximite;
2. Load a sprite from a PNG (Portable Network Graphics) format file;
3. Create a sprite by reading from what is already displayed on one of the CMM2 video pages; or
4. Copy an existing sprite to create a new one.

We will look at each of these in turn:

### SPRITE LOAD

Load a sprite from an ascii file.

The CMM2 supports creating sprites by reading in an ascii file formatted as described in the Colour Maximite manual with a key enhancement: CMM2 sprites can be any width and any height.

Sprites are loaded from Maximite style **SPRITE** files using **SPRITE LOAD filename\$**. There is now has an optional parameter available in the first line of the sprite file. The definition of the new first line is:-

Horizontal size in pixels (need not be 16),  
number of sprites described in the file,  
vertical size of sprites in pixels (need not be 16)

If the vertical size parameter is omitted then the first parameter is used for both width and height allowing complete compatibility with CMM sprite files.

The example on the next page defines a red mouse pointer 13 pixels wide by 19 pixels high.



## SPRITE READ

Create a sprite by reading from what is already displayed on one of the CMM2 video pages

This can be the simplest way of creating a sprite. We simply specify that we want to read what is on one of the video pages and create a sprite from it. Given that, the syntax is obvious:

```
SPRITE READ [#]n, x , y, w, h [,page_number]
```

Added additional parameter to **SPRITE READ** `SPRITE READ [#]n, x , y, w, h [,page_number]`

This specifies which page to read from.

This says create sprite number `n` by reading what is on the screen in the rectangle with the top left coordinate of `x` and `y`, width `w` and height `h`. There is just one added component to this: **SPRITE READ** will read from the **PAGE** specified by `page_number` - if omitted it defaults to the **WRITE** page. This allows us to read in sprites without changing **PAGE 0** which is always being displayed.

Suppose we want a sprite which is simply a red box 10x10 pixels

```
PAGE WRITE 1
CLS RGB(RED)
SPRITE READ 1, 0, 0, 10, 10, 1
PAGE WRITE 0
SPRITE SHOW 1,100,100,1
```

would do the job and display the sprite on the display without having to see its creation.

Note: **BLIT READ** and **SPRITE READ** are identical and do exactly the same thing.

**BLIT** to and from the framebuffer is now supported for all graphics modes and with all orientation parameter support. To **BLIT** from the framebuffer to the current **WRITE** page, use **FRAMEBUFFER** as the page parameter e.g.

```
PAGE WRITE 0
BLIT 50,50,500,500,75,75,FRAMEBUFFER
```

To **BLIT** to the framebuffer from a normal page, just use the **BLIT** command as normal

```
PAGE WRITE FRAMEBUFFER
BLIT 500,500,50,50,75,75,0
```

**BLIT READ/SPRITE READ** both support reading from the **FRAMEBUFFER**

```
BLIT READ #1,100,100,100,100,FRAMEBUFFER
```

Note: See the chapter on **FRAMEBUFFER** for more information on using this feature.



Text in a tip

## SPRITE COPY

This command will copy an existing sprite to create a new one

As mentioned earlier, each sprite created uses two RAM buffers one for the sprite and one to store the display contents when the sprite is shown. This can use quite a lot of memory for big sprites. However, if we want many sprites that are based on the same image ( the blocks in [this demo](#) are all sprites) we can copy one sprite to many others and the "children" sprites will only take the one memory buffer needed to save the screen but share the image with the "parent".

The syntax for copying sprites is:

```
SPRITE COPY [#]n, [#]m, nbr
```

This makes a copy of sprite “`n`” to “`nbr`” of new sprites starting at number “`m`”.

The code from the demo uses both this and **SPRITE READ** to create the block sprites

```
' Draw one box of each colour
Box 401,1,18,48,2,RGB(green),RGB(Magenta)
Box 421,1,18,48,2,RGB(red),RGB(cyan)
Box 441,1,18,48,2,RGB(yellow),RGB(blue)
Box 461,1,18,48,2,RGB(white),RGB(red)
Box 481,1,18,48,2,RGB(Magenta),RGB(GREEN)

' read in each block as a parent sprite
SPRITE read 10+i,401,1,18,48
SPRITE read 18+i,421,1,18,48
SPRITE read 26+i,441,1,18,48
SPRITE read 34+i,461,1,18,48
SPRITE read 42+i,481,1,18,48

'copy each parent to 7 children
SPRITE copy 10,11,7
SPRITE copy 18,19,7
SPRITE copy 26,27,7
SPRITE copy 34,35,7
SPRITE copy 42,43,7
```

We now know how to create sprites and can think about the pros and cons of the various methods. Next post we had better start looking at how to use them.

## SPRITES AND HOW TO USE THEM

In this post I will look at some of the Basics of sprites and then the final sprite post will look at collisions. I might then do one more post on 3D graphics and try to de-mystify some of the maths behind them.

In the previous post we saw what a sprite is and how to create one so we can start here by getting rid of one or more sprites. This may be important to free up the 64 sprite slots for new sprites and the syntax is obvious.

```
SPRITE CLOSE [#]n
SPRITE CLOSE ALL
```

Executing these commands frees up the sprite slot and the memory used by the sprite. There is one important difference in their usage. Closing an individual sprite will also remove it from the screen if it is visible. Closing all sprites leaves the screen untouched as this is likely only to be used at the end of a program or when switching to a completely new context in a game. It is also important to note that you can't close a sprite from which copies have been made until all of the copies have been closed.

So we can create sprites and destroy them but we now need to know how to use them.

There is one overriding rule to be followed to make things work:

**\*\*\* While sprites are in use, all sprite commands must be done with the same PAGE WRITE set**



This need not be PAGE 0, the display page, but must be consistent for all SPRITE commands. You can move to other pages and use other graphics commands but if any sprite is active then all subsequent sprite commands must be to that page.

To keep things simple for the moment we will not change the page we are writing to and leave it at PAGE 0. I've attached the sprite file for the red mouse pointer we saw earlier and an image to use in the next example.

You need at least V5.05.03 to run this example due to a bug in earlier versions.



[mouse.zip](#)

```
'set to 320x200 16 bit colour
mode 3,16
' load the mouse cursor as sprite number 1
sprite load "mouse.spr",1
'load a background image
load bmp "tiger320"
'set up some variables to track the sprite position and show it
dim integer x=mm.hres/2, y=mm.vres/2
sprite show 1, x, y, 1
'loop forever
do
'use the keydown function with the USB keyboard to check for a keypress
'you can change this to use inkey$ to use a serial console
if keydown(0) then 'There is a key pressed so check for one of the arrow keys
  if keydown(1)=128 then y=y-1
  if keydown(1)=129 then y=y+1
  if keydown(1)=130 then x=x-1
  if keydown(1)=131 then x=x+1

' use the sprite function to check if the sprite needs to move
' by comparing its actual position to the new x,y position
' if it is different then display the sprite in the new position
' this will automatically restore the background overwritten by the original sprite
' and then rewrite it in the new position

  if x<> sprite(x,1) or y <> sprite(y,1) then sprite show 1, x, y, 1

endif
pause 10
loop
```



Try keeping the left arrow pressed and watch as the sprite leaves the screen

```
[12] Error: -13 is invalid (valid is -12 to 319)
```

Oops: sprites must have at least one pixel of their containing rectangle on the screen otherwise you will get an error. We can easily check for this in the code

```
if keydown(1)=128 and sprite(y,1) + sprite(h,1) > 1 then y=y-1
if keydown(1)=129 and sprite(y,1)+1 < MM.VRES then y=y+1
if keydown(1)=130 and sprite(x,1) + sprite(w,1) > 1 then x=x-1
if keydown(1)=131 and sprite(x,1)+1 < MM.HRES then x=x+1
```

The **SPRITE** function is very useful for telling us about the sprite. Check the manual for all the various **SPRITE** function parameters.

The important command we have used here is **SPRITE SHOW**. This command is used both to display a sprite and to move it. However it has two additional parameters that we haven't yet considered.

```
SPRITE SHOW [#n], x, y, layer [,orientation]
```

The optional orientation parameter acts to modify the sprite as it is displayed in the same way as we saw for the **BLIT** command, 0=normal, 1=mirrored left to right, 2=mirrored top to bottom, 3=rotated 180 degrees.

The mandatory (collision) layer parameter does two things for us relating to collisions and movement and can be set between 0 and 10.

Sprites shown on layer 0 will move with the background if we use **SPRITE SCROLL** or **SPRITE SCROLLR**. Sprites on all other layers will remain fixed on the screen during a scroll and the background will move underneath them.

Sprites collisions are detected when a sprite overlaps another sprite on the same layer or layer 0 so collisions can be caused by moving a sprite on any layer or scrolling the background such that a layer 0 sprite now overlaps a sprite on any of the other layers.

Collisions will be discussed in much more detail in the next post but there are other sprite commands we should look at first.

**SPRITE WRITE** and **BLIT WRITE** do exactly the same thing. In both case the **SPRITE** is written out but the background image isn't stored first and you can **WRITE** the same sprite as many times as you wish to different locations on the screen. In essence the command is just using the sprite as a in memory image that can be used whenever required.

**SPRITE HIDE** does what it says, it removes a sprite from the screen and restores the background we can add this to our example program and make the cursor flash.

```
mode 3,16
sprite load "mouse.spr",1
load bmp "tiger320"
dim integer x=mm.hres/2, y=mm.vres/2
settick 100,flash
sprite show 1, x, y, 1
do
  if keydown(0) then
    if keydown(1)=128 and sprite(y,1) + sprite(h,1) > 1 then y=y-1
    if keydown(1)=129 and sprite(y,1)+1 < MM.VRES then y=y+1
    if keydown(1)=130 and sprite(x,1) + sprite(w,1) > 1 then x=x-1
    if keydown(1)=131 and sprite(x,1)+1 < MM.HRES then x=x+1
  endif
  pause 20
loop

sub flash
  static a=1
  if a then
    sprite show 1,x,y,1
    a=0
  else
    sprite hide 1
    a=1
  endif
end sub
end sub
```

**SPRITE NEXT** and **SPRITE MOVE** commands allow you to create a sequence of sprite actions which will be executed as a single atomic transaction. This was developed for other platforms to avoid tearing and flashing effects but because on the CMM2 sprites can be moved on non-visible pages these commands are not as important.

The **SPRITE SCROLL** and **SPRITE SCROLLR** commands allow you to move the background image under the sprites for the whole screen or just part of it. **SPRITE SCROLL** is really identical to **PAGE SCROLL** that we discussed in detail above with the simple exception that all sprites on layers 1 to 10 are automatically hidden before the scroll and then replaced in their original positions afterwards. **SPRITE SCROLLR** does exactly the same thing but allows you to specify a part of the background image to scroll.

The last sprite command we will consider in this post is **SPRITE TRANSPARENCY** and this is used to create the [ghost demo](#)

The full code for this is attached and explained below. This uses many of the concepts we have seen in the posts on this thread and brings them together. What I hope will be clear from reading the code is how easy it is to create stunning effects with the CMM2 once you understand a little about the graphics are designed and work.

Image files for the ghost demo are [2020-05-08 234454 Ghost.zip](#)

```
'set to 640x400 mode with 2 video layers and 12-bit colour. Page 0 is the bottom layer and page 1 is
the top
mode 2,12
'clear the first three framebuffers (pages)
for i=0 to 2:page write i:cls:next i
'load a sprite which is the image of the ghost from a png file.
'The fact it is a png is important as png files encode transparency as well as solid colours
sprite loadpng 1,"ghost"
'set that we are going to write to the background layer
page write 0
'load the background image to the background layer - page 0
load png "part02"

'The image I am using is only 320x200 so I'm going to resize it to fit the screen
image resize 0,0,320,200,0,0,640,400

'initialise the display position of the ghost
x=100
y=50

'initialise the transparency of the ghost
'transparencies go from 1 (nearly invisible) to 15 (solid colour)
t=8

'set to write to page 2 which is not being displayed
page write 2

' output the ghost on page 2
sprite show 1,x,y,1
```

```

i=0

'start the main process loop
do

'do some silly maths to create a random walk of the ghost in both position and
'transparency while keeping it within the display bounds and the transparency
'within useful limits
i=i+1
if i mod 5 = 0 then c=rnd()-0.5
if i mod 3 = 0 then a=rnd()*8-4
if i mod 3 = 0 then b=rnd()*6-3
x=x+a
if x<0 then x=0
if x>MM.HRES-sprite(w,1) then x=mm.hres-sprite(w,1)
y=y+b
if y<0 then y=0
if y>MM.VRES-sprite(h,1) then y=mm.vres-sprite(h,1)
t=t+c
if t<3 then t=3
if t>12 then t=12

'display the sprite in the new position and with the new transparency
sprite transparency 1,t
sprite show 1,x,y,1

'now copy page 2 to the foreground layer during frame blanking
'this ensures that there are no tearing effects in the image

page copy 2 to 1,b
'slow things down a bit, the CMM2 is too fast
pause 100
loop

```

## New SPRITE sub-commands:

In version 5.05.05 of the firmware, a number of new SPRITE sub commands were introduced.

### SPRITE HIDE ALL

Hides all the sprites allowing the background to be manipulated.

The following commands **cannot** be used when all sprites are hidden:

```

SPRITE SHOW (SAFE)
SPRITE HIDE (SAFE, ALL)
SPRITE SWAP
SPRITE MOVE
SPRITE SCROLLR
SPRITE SCROLL

```

### SPRITE RESTORE

Restores all the sprites. NB that any position changes previously requested using SPRITE NEXT will be actioned by the RESTORE and collision detection will be run

### SPRITE HIDE SAFE n

Hides a sprite but automatically compensates for any other sprites that overlap it. This allows you to safely hide a sprite that is partially obscured but is obviously lower in performance than using the simple HIDE command

## SPRITE SHOW SAFE n, x, y, layer[, orientation] [,ontop]

Shows a sprite and automatically compensates for any other sprites that overlap it.

If the sprite is not already being displayed, the command acts exactly the same as `SPRITE SHOW`.

If the sprite is already shown it is moved and remains in its relative position relative to other sprites based on the order of writing. i.e. if sprite 1 was written before sprite 2 and it is moved to overlap sprite 2 it will display under sprite 2.

If the optional "ontop" parameter is set to 1 then the sprite moved will become the newest sprite and will sit on top of any other sprite it overlaps.

```
box 0,0,50,50,4,rgb(red),rgb(white)
box 0,50,50,50,4,rgb(green),rgb(yellow)
sprite read 1,0,0,50,50
sprite read 2,0,50,50,50
sprite copy 2,3,1
sprite show 1, 150,150,1
sprite show 2, 170,170,1
sprite show 3,210,210,1
pause 2000
sprite show safe 1,180,180,1,,1
pause 2000
sprite show safe 2,190,190,1
pause 2000
sprite show safe 1,200,200,1
pause 2000
sprite hide safe 3
do
loop
```

## The FRAMEBUFFER command

The framebuffer is an area of memory that can be up to 1600x1200 pixels in size irrespective of the current mode. The memory for the framebuffer is taken from the 5Mb of variable memory.



Note: The FRAMEBUFFER is a completely different and independent section of memory from that used for Pages.

Rather than PAGE COPY you use FRAMEBUFFER WINDOW to move data from the framebuffer to main memory. This allows you to have a playing field bigger than the display and the window that is displayed on the screen to be a different part of the field with a single command.

A simple example of using the framebuffer is shown [here](#)

This command allows you to create, use and remove a variable size framebuffer which should make many applications which need a working graphics area bigger than the screen easier to program. Moving a 640x400x16 image from the framebuffer to the display takes just over 4mSec. Moving a 320x200x8 image from the framebuffer to the display takes just less than 1mSec.

### FRAMEBUFFER CREATE HorizontalSize%, VerticalSize%

This command creates a framebuffer with the width and height specified in pixels. HorizontalSize $\geq$ MM.HRES and  $\leq$ 1600; VerticalSize $\geq$ MM.VRES and  $\leq$ 1200

### FRAMEBUFFER WRITE

#### PAGE WRITE FRAMEBUFFER

These commands are interchangeable and set all drawing commands to write to the framebuffer and inherit the width and height defined.

### FRAMEBUFFER BACKUP

This command creates a backup copy of the framebuffer. If a backup already exists it is overwritten. This allows the programmer to save the background before he/she starts writing non-static data to it. NB: It won't be possible to use this command if a very large framebuffer is specified in 12 or 16-bit colour depth. A sensible error will be given in this case.

### FRAMEBUFFER RESTORE [x, y, w, h

This command restores all or part of the framebuffer from the backup. This allows the programmer to “clean” all or part of the framebuffer before adding new non-static items.

### FRAMEBUFFER WINDOW x, y, pageno [,I or B]

This command copies an area MM.HRES by MM.VRES from the framebuffer with top left at x,y to the page specified. The optional parameter specifies if the copy is Immediate or during frame **B**lanking.

### FRAMEBUFFER CLOSE

This command releases the memory resources used by the framebuffer and backup allowing a new framebuffer to be created with a different size.

### MM.INFO(FrameH)

This function returns the horizontal size of the framebuffer in pixels.

### MM.INFO(FrameV)

This function returns the vertical size of the framebuffer in pixels.



Note: The following should be noted when using the FRAMEBUFFER command.

- Setting a different graphics mode that changes the colour depth will close and delete the framebuffer.
- JPG files cannot be loaded to the framebuffer and will error if tried.
- The framebuffer is deleted by Ctrl-C and by running a new program.
- Changing graphics mode such that MM.HRES or MM.VRES are bigger than the size of the framebuffer will cause it to be deleted.

Example use similar to the video:

```
option console serial
framebuffer create 1024,1024
page write framebuffer
load bmp "1024" 'load a 1024x1024 image to the framebuffer
memory
framebuffer backup 'back up the image
memory
cls rgb(red) 'clear the framebuffer to red
framebuffer restore 100,100,800,800 'restore the central area of the backup
'
line 0,0,1023,1023 'check simple graphics command
text 850,850,"hello",,,4,rgb(blue),-1 'check text output to the framebuffer
```

```

j=0
do
mode 4,16
for i=0 to 1023-mm.hres
framebuffer window i,j,0
next i
for j=0 to 1023-mm.vres
framebuffer window i,j,0
next j
for i=1023-mm.hres to 0 step -1
framebuffer window i,j,0
next i
i=0
for j=1023-mm.vres to 0 step -1
framebuffer window i,j,0
next j
j=0
framebuffer restore 'go back to the clean image
mode 2,16
for i=0 to 1023-mm.hres
timer=0
framebuffer window i,j,0
print timer
next i
for j=0 to 1023-mm.vres
framebuffer window i,j,0
next j
for i=1023-mm.hres to 0 step -1
framebuffer window i,j,0
next i
i=0
for j=1023-mm.vres to 0 step -1
framebuffer window i,j,0
next j
j=0
loop

```

**BLIT** to and from the framebuffer is now supported for all graphics modes and with all orientation parameter support. To **BLIT** from the framebuffer to the current **WRITE** page, use **FRAMEBUFFER** as the page parameter e.g.

```

PAGE WRITE 0
BLIT 50,50,500,500,75,75,FRAMEBUFFER

```

To **BLIT** from the framebuffer to a normal page just use the **BLIT** command as normal

```

PAGE WRITE FRAMEBUFFER
BLIT 500,500,50,50,75,75,0

```

**BLIT READ/SPRITE READ** both support reading from the **FRAMEBUFFER**

```

BLIT READ #1,100,100,100,100,FRAMEBUFFER

```

## QUATERNIONS

In this thread we look at using the mathematical technique of using QUATERNIONS to manipulate wire-frame objects in 3-dimensional space.

This is a demo program (all in Basic) of some interesting maths that underpins all sorts of technologies like attitude/heading systems and 3D graphics.

*Editors note: While not specific to the CMM2, this program demonstrates the power of the graphics in the CMM2.*



Quaternions are a mathematical technique that allows the rotation in any direction of any shape. The maths behind it are way beyond me and extend the concept of complex numbers such that:

$$i*i = j*j = k*k = i*j*k = -1$$

I find that equation completely bemusing 😊

However, quaternions are pretty easy to use in the real world. There are some gotchas: quaternions are not commutative e.g  $q1 * q2$  does not give the same result as  $q2 * q1$ . But overall the calculations are pretty simple.

The attached code draws a wireframe cube and then rotates it continuously in the directions specified by the rotation quaternion.

The cube is specified as a set of coordinates around a point 0,0,0 in the array "cube". Then the array cubelines% is used to store the coordinate pairs that make up the lines that draw the cube.

The array "cubelocation" defines the actual position of the cube in space. The x and y axes are as per the display and the z axis comes out of the display towards the viewer.

The code uses a projection method of perspective to visualise the cube in 2 dimensions and "viewplane" defines the location of the projection plane in the z axis.

All quaternion arithmetic is carried out with the quaternion normalised. The subroutine "create\_normalised\_quaternion" takes a rotation vector and an angle and converts it into a quaternion.

Then to rotate a point we just multiply the rotation quaternion by the normalised coordinates of the point and then multiply the answer by the inverse of the quaternion. Note how in normal math this would leave the point unchanged but because of the non-commutative nature of the quaternion somehow the sequence does what we want (look at Jorge Rodriguez excellent videos on Youtube [here](#) if you want to know more). Note that no trigonometry is needed other than in the initialisation of the rotation quaternion and that only uses the sine and cosine of a single angle.

The attached code includes all the routines necessary to rotate objects in 3D and then project them onto a 2D screen.

My interest is in using the math to understand what is happening inside an Attitude and Heading Reference System (AHRS), but the example code below is quite fun.

The code should run on any type of Micromite with a connected TFT display. The picture is of a 320x240 SSD1289 display driven by a 44-pin uM2.

Play with this line:

```
create_normalised_quaternion(2,1,0,0,q1())
```

to change the way the cube rotates. The parameters are:  
number of degrees to rotate about the rotation vector,  
x-coordinate of the end of the rotation vector,  
y-coordinate of the end of the rotation vector,  
z-coordinate of the end of the rotation vector, and  
array to receive the quaternion.

Enjoy 🍌

```
option explicit
option default FLOAT
cpu 48
' 3D wire frame graphics demo using quaternions to rotate the coordinates
cls

dim q1(4),v(4),vout(4),M_PI=3.14159265359,i%,x1,y1,z1,x2,y2,z2

' define the coordinates of a cube with the z-axis towards the viewer
dim cube(3,7)=(-100,-100,100,0, -100,100,100,0, 100,-100,100,0, 100,100,100,0, -100,-
100,-100,0, -100,100,-100,0, 100,-100,-100,0, 100,100,-100,0 ) 'coordinates of the cube

' define the coordinate pairs that make up the edges of the cube
DIM cubelocation(2)=(mm.hres/2,MM.VRES/2,-500) 'actual location of the centre of the cube
in space

' define the coordinate pairs that make up the edges of the cube
dim cubelines%(1,11)=(0,1, 0,2, 1,3, 2,3, 4,5, 4,6, 5,7, 6,7, 0,4, 1,5, 2,6, 3,7)

'define the projection plane
DIM viewplane = -200
dim lines%(3,11),oldlines%(3,11) 'storage for the coordinates for lines drawn

for i%=0 to 7 'convert coordinates to normalised form
  x1=cube(0,i%)
  y1=cube(1,i%)
  z1=cube(2,i%)
  create_vector(x1,y1,z1,v())
  cube(0,i%)=v(2)
  cube(1,i%)=v(3)
  cube(2,i%)=v(4)
  cube(3,i%)=v(0)
next i%

'create a quarternion to rotate 2 degrees about x axis

'play with the x,y,z vector which is the sxis of rotation
create_normalised_quaternion(2,1,0,0,q1())

do
for i%=0 to 11 'project the coordinates onto the viewplane
  z1=cube(2,cubelines%(0,i%))*cube(3,cubelines%(0,i%))+cubelocation(2)
  x1=(cube(0,cubelines%(0,i%))*cube(3,cubelines%(0,i%))*viewplane/z1 +cubelocation(0))
  y1=(cube(1,cubelines%(0,i%))*cube(3,cubelines%(0,i%))*viewplane/z1 +cubelocation(1))
  z2=cube(2,cubelines%(1,i%))*cube(3,cubelines%(1,i%))+cubelocation(2)
  x2=(cube(0,cubelines%(1,i%))*cube(3,cubelines%(1,i%))*viewplane/z2 +cubelocation(0))
  y2=(cube(1,cubelines%(1,i%))*cube(3,cubelines%(1,i%))*viewplane/z2 +cubelocation(1))
  lines%(0,i%)=x1 :lines%(1,i%)=y1 :lines%(2,i%)=x2 :lines%(3,i%)=y2
next i%

for i%=0 to 7 'rotate coordinates
  v(2)=cube(0,i%)
  v(3)=cube(1,i%)
  v(4)=cube(2,i%)
  v(0)=cube(3,i%)
  v(1)=0
  rotate_vector(vout(),v(),q1())
  cube(0,i%)=vout(2)
  cube(1,i%)=vout(3)
  cube(2,i%)=vout(4)
  cube(3,i%)=vout(0)
next i%

for i%=0 to 11 'delete the old lines
  line oldlines%(0,i%),oldlines%(1,i%),oldlines%(2,i%),oldlines%(3,i%),1,rgb(black)
next i%
```

```

for i%=0 to 11 'draw the new lines
  line lines%(0,i%),lines%(1,i%),lines%(2,i%),lines%(3,i%),1,rgb(white)
  oldlines%(0,i%)=lines%(0,i%)
  oldlines%(1,i%)=lines%(1,i%)
  oldlines%(2,i%)=lines%(2,i%)
  oldlines%(3,i%)=lines%(3,i%)
next i%

loop
end

'
sub create_normalised_quaternion(theta,x,y,z,q())
  local radians = theta/180.0*M_PI
  local sineterm= sin(radians!/2)
  q(1)=cos(radians/2)
  q(2)=x* sineterm
  q(3)=y* sineterm
  q(4)=z* sineterm
  q(0)=sqr(q!(1)*q(1) + q(2)*q(2) + q(3)*q(3) + q(4)*q(4)) 'calculate the magnitude
  q(1)=q(1)/q(0) 'create a normalised quaternion
  q(2)=q(2)/q(0)
  q(3)=q(3)/q(0)
  q(4)=q(4)/q(0)
  q(0)=1
end sub
'
sub invert_quaternion(n(),q())
  n(0)=q(0)
  n(1)=q(1)
  n(2)=-q(2)
  n(3)=-q(3)
  n(4)=-q(4)
end sub
'
sub multiply_quaternion(n(),q1(),q2())
  local a1=q1(1),a2=q2(1),b1=q1(2),b2=q2(2),c1=q1(3),c2=q2(3),d1=q1(4),d2=q2(4)
  n(1)=a1*a2-b1*b2-c1*c2-d1*d2
  n(2)=a1*b2+b1*a2+c1*d2-d1*c2
  n(3)=a1*c2-b1*d2+c1*a2+d1*b2
  n(4)=a1*d2+b1*c2-c1*b2+d1*a2
  n(0)=q1(0)*q2(0)
end sub
'
sub create_vector(x,y,z,v())
  v(0)=sqr(x*x + y*y + z*z)
  v(1)=0
  v(2)=x/v(0)
  v(3)=y/v(0)
  v(4)=z/v(0)
end sub

sub rotate_vector(vnew(),v(),q())
  local n(4),iq(4)
  multiply_quaternion(n(),q(),v())
  invert_quaternion(iq(),q())
  multiply_quaternion(vnew(),n(),iq())
end sub

```

```

option explicit
option default none
mode 2
page write 1
const edgelenh=100 'set the length of the verticies of the ticosahedron
const zlocation=1000 'how far is the center of the ticosahedron away from us
const viewplane=800 'how far is the viewplane away from us
dim float x,y,d
dim float phi=(1+sqr(5))/2
dim float x1,y1,z1

```

```

dim integer col(11), sortorder(11)
dim float q1(4),depth(11),v(4), vout(4)
' data for location of verticies for ticosahedron of edge length 2
data 0,1,3*phi
data 0,1,-3*phi
data 0,-1,3*phi
data 0,-1,-3*phi
data 1,3*phi,0
data 1,-3*phi,0
data -1,3*phi,0
data -1,-3*phi,0
data 3*phi,0,1
data 3*phi,0,-1
data -3*phi,0,1
data -3*phi,0,-1
data 2,(1+2*phi),phi
data 2,(1+2*phi),-phi
data 2,-(1+2*phi),phi
data 2,-(1+2*phi),-phi
data -2,(1+2*phi),phi
data -2,(1+2*phi),-phi
data -2,-(1+2*phi),phi
data -2,-(1+2*phi),-phi
data (1+2*phi),phi,2
data (1+2*phi),phi,-2
data (1+2*phi),-phi,2
data (1+2*phi),-phi,-2
data -(1+2*phi),phi,2
data -(1+2*phi),phi,-2
data -(1+2*phi),-phi,2
data -(1+2*phi),-phi,-2
data phi,2,(1+2*phi)
data phi,2,-(1+2*phi)
data phi,-2,(1+2*phi)
data phi,-2,-(1+2*phi)
data -phi,2,(1+2*phi)
data -phi,2,-(1+2*phi)
data -phi,-2,(1+2*phi)
data -phi,-2,-(1+2*phi)
data 1,(2+phi),2*phi
data 1,(2+phi),-2*phi
data 1,-(2+phi),2*phi
data 1,-(2+phi),-2*phi
data -1,(2+phi),2*phi
data -1,(2+phi),-2*phi
data -1,-(2+phi),2*phi
data -1,-(2+phi),-2*phi
data (2+phi),2*phi,1
data (2+phi),2*phi,-1
data (2+phi),-2*phi,1
data (2+phi),-2*phi,-1
data -(2+phi),2*phi,1
data -(2+phi),2*phi,-1
data -(2+phi),-2*phi,1
data -(2+phi),-2*phi,-1
data 2*phi,1,(2+phi)
data 2*phi,1,-(2+phi)
data 2*phi,-1,(2+phi)
data 2*phi,-1,-(2+phi)
data -2*phi,1,(2+phi)
data -2*phi,1,-(2+phi)
data -2*phi,-1,(2+phi)
data -2*phi,-1,-(2+phi)
' 12 faces with 5 sides
data 0,28,36,40,32
data 1,29,37,41,33
data 2,30,38,42,34
data 3,31,39,43,35
data 4,12,44,45,13
data 5,14,46,47,15
data 6,16,48,49,17
data 7,18,50,51,19
data 8,20,52,54,22
data 9,21,53,55,23
data 10,24,56,58,26
data 11,27,59,57,25
' 20 faces with 6 sides

```

```

data 0,2,34,58,56,32
data 0,2,30,54,52,28
data 1,3,31,55,53,29
data 1,3,35,59,57,33
data 4,6,17,41,37,13
data 4,6,16,40,36,12
data 5,7,19,43,39,15
data 5,7,18,42,38,14
data 8,9,23,47,46,22
data 8,9,21,45,44,20
data 10,11,27,51,50,26
data 10,11,25,49,48,24
data 12,44,20,52,28,36
data 13,45,21,53,29,37
data 14,46,22,54,30,38
data 15,47,23,55,31,39
data 16,48,24,56,32,40
data 17,49,25,57,33,41
data 18,50,26,58,34,42
data 19,51,27,59,35,43
'
dim float zpos(31),zsort(31)
dim float ticos(2,59), ntricos(3,59)
dim integer i,j,k,l,m,n
dim integer xs(179),ys(179),xe(179),ye(179)
dim integer index(31),nnum(31)
' read in the coordinates of the verticies and scale
for j=0 to 59
  for i=0 to 2
    read ticos(i,j)
    ticos(i,j)=ticos(i,j)*edgelen/2
  next i
next j
'
dim integer xarr(179),yarr(179)
dim integer nv(31)=(5,5,5,5,5,5,5,5,5,5,5,5,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6,6)
dim integer np(31)=(0,1,2,3,4,5,6,7,8,9,10,11,0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19)
dim integer nd(31)
dim integer f5(4,11), f6(5,19)
dim integer ncol(31)
for j=0 to 11
  for i=0 to 4
    read f5(i,j)
  next i
next j
for j=0 to 19
  for i=0 to 5
    read f6(i,j)
  next i
next j

'convert coordinates to normalised form
for i=0 to 59
  x1=ticos(0,i): y1=ticos(1,i): z1=ticos(2,i)
  create_vector(x1,y1,z1,v())
  ntricos(0,i)=v(2): ntricos(1,i)=v(3): ntricos(2,i)=v(4): ntricos(3,i)=v(0)
next i

'create a quaternion to rotate 4 degrees about a chosen axis
'play with the x,y,z vector which is the axis of rotation
create_normalised_quaternion(4,1,0.5,0.25,q1())

do
  cls
  for i=0 to 59 'rotate coordinates
    v(2)=ntricos(0,i): v(3)=ntricos(1,i): v(4)=ntricos(2,i): v(0)=ntricos(3,i): v(1)=0
    rotate_vector(vout(),v(),q1())
    ntricos(0,i)=vout(2): ntricos(1,i)=vout(3): ntricos(2,i)=vout(4): ntricos(3,i)=vout(0)
  next i
'
' average the z positions for the five sided faces
for k=0 to 11
  zpos(k)=0
  for i=0 to 4
    zpos(k)=zpos(k)+ntricos(2,f5(i,k))
  next i
  zpos(k)=zpos(k)/5

```

```

    index(k)=k
next k
'average the z positions for the 6 sided faces
for k=12 to 31
    zpos(k)=0
    for i=0 to 5
        zpos(k)=zpos(k)+nticos(2,f6(i,k-12))
    next i
    zpos(k)=zpos(k)/6
    index(k)=k
next k
' sort the z positions
sort zpos(),index()
'
j=0:m=0
for l=0 to 31
    k=index(l)
    m=np(k)
    nd(l)=nv(k)
    if nv(k)=5 then
        ncol(l)=rgb(red)
    else
        ncol(l)=rgb(white)
    endif
    for i=0 to nv(k)-1
        if nv(k)=5 then
            xarr(j)=nticos(0,f5(i,m))*viewplane/(nticos(2,f5(i,m))+zlocation)*nticos(3,f5(i,m))+MM.HRES/2
            yarr(j)=nticos(1,f5(i,m))*viewplane/(nticos(2,f5(i,m))+zlocation)*nticos(3,f5(i,m))+MM.VRES/2
        else
            xarr(j)=nticos(0,f6(i,m))*viewplane/(nticos(2,f6(i,m))+zlocation)*nticos(3,f6(i,m))+MM.HRES/2
            yarr(j)=nticos(1,f6(i,m))*viewplane/(nticos(2,f6(i,m))+zlocation)*nticos(3,f6(i,m))+MM.VRES/2
        endif
        j=j+1
    next i
next l
polygon nd(),xarr(),yarr(),rgb(black),ncol()
page copy 1 to 0
loop
'
sub create_normalised_quaternion(theta as float,x as float,y as float,z as float,q() as float)
    local float radians = theta/180.0*PI
    local float sineterm= sin(radians!/2)
    q(1)=cos(radians/2)
    q(2)=x* sineterm
    q(3)=y* sineterm
    q(4)=z* sineterm
    q(0)=sqr(q!(1)*q(1) + q(2)*q(2) + q(3)*q(3) + q(4)*q(4)) 'calculate the magnitude
    q(1)=q(1)/q(0) 'create a normalised quaternion
    q(2)=q(2)/q(0)
    q(3)=q(3)/q(0)
    q(4)=q(4)/q(0)
    q(0)=1
end sub
'
sub invert_quaternion(n() as float,q()as float)
    n(0)=q(0)
    n(1)=q(1)
    n(2)=-q(2)
    n(3)=-q(3)
    n(4)=-q(4)
end sub
'
sub multiply_quaternion(n() as float,q1() as float,q2() as float)
    local float a1=q1(1),a2=q2(1),b1=q1(2),b2=q2(2),c1=q1(3),c2=q2(3),d1=q1(4),d2=q2(4)
    n(1)=a1*a2-b1*b2-c1*c2-d1*d2
    n(2)=a1*b2+b1*a2+c1*d2-d1*c2
    n(3)=a1*c2-b1*d2+c1*a2+d1*b2
    n(4)=a1*d2+b1*c2-c1*b2+d1*a2
    n(0)=q1(0)*q2(0)
end sub
'
sub create_vector(x as float,y as float ,z as float,v() as float)
    v(0)=sqr(x*x + y*y + z*z)
    v(1)=0
    v(2)=x/v(0)
    v(3)=y/v(0)
    v(4)=z/v(0)

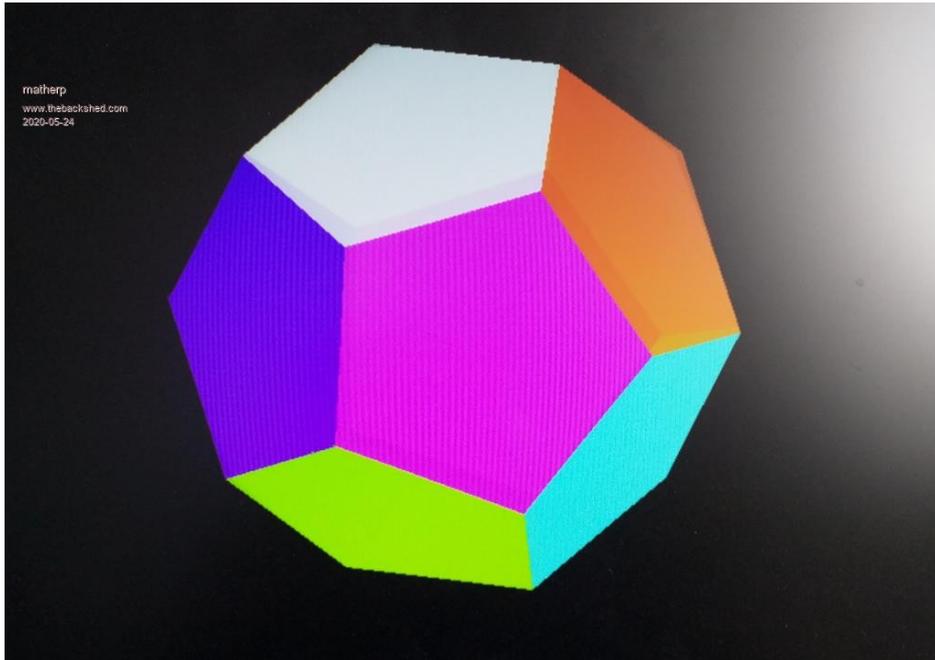
```

```
end sub

sub rotate_vector(vnew() as float,v() as float,q() as float)
  local float n(4),iq(4)
  multiply_quaternion(n(),q(),v())
  invert_quaternion(iq(),q())
  multiply_quaternion(vnew(),n(),iq())
end sub
```

Warning: this code is completely useless - [Watch this video](#) and [this one](#)

.....but I think it is interesting 😊



This post takes the discussion on quaternions further by using some of the techniques of 3D graphics processing to manipulate and display solid objects.

To do this we need to implement [Hidden Surface Determination](#) . In the code below we use [Painter's Algorithm](#) .

In general computer graphics uses triangles as the basic drawing object. In this example each face of the cube is split into two triangles and a standard fill-triangle algorithm used to draw them.

The vertices of the cube are manipulated in 3D in the same way as in the previous post. However, in this example we also manipulate the centroid (geometric centre) of each face of the cube. By doing this we can calculate the distance away of the centroid along the Z-axis after each rotation. This allows us to sort the Z-position of the centroids and display each face in turn, furthest away from the viewer first. This implements the Painter's algorithm which works extremely well except when faces overlap as shown in the Wikipedia article.

The code is all in Basic except the floating point sort and the triangle display functions.

Drawing has been deliberately slowed to see the hidden surfaces being overwritten by each face as they are drawn in increasing order of Z axis.

The code as listed runs on a SSD1963 800x480 display but will also run on a ILI9341 but you will need to adjust the "scale" to fit the screen (300 works well). It will run on uM2 or MM+

I hope this is interesting insight into the way objects can be manipulated in 3 dimensions and then displayed on a 2D screen

```
option explicit
option default FLOAT
' 3D solid graphics demo using quaternions to rotate the coordinates
'
cls
' Adjust the following to fit your display
'
dim scale=600 'length of the edge of the cube in "real" space
'define the projection plane in units from the observer
DIM viewplane = -500
DIM cubelocation(2)=(mm.hres/2,MM.VRES/2,-2000) 'actual location of the centre of the
cube in "real" 3D space
'
'
' define the coordinates of a cube with the z-axis towards the viewer and the centre at
0,0,0
dim cube(3,7)=(-1,-1,1,0, -1,1,1,0, 1,1,1,0, 1,-1,1,0, -1,-1,-1,0, -1,1,-1,0, 1,1,-1,0,
1,-1,-1,0 ) 'coordinates of the cube
dim centroid(3,5)=(0,0,1,0, -1,0,0,0, 0,1,0,0, 1,0,0,0, 0,0,-1,0, 0,-1,0,0) 'centre of
each face in space
```

```

'
' define the coordinate triples that make up two triangles for each face of the cube,
together with a face index number
' all drawing is done as filled triangles
' NB the triangles must be in the same order as the faces, two triangles/face
'
dim cubetriangles%(3,11)=(0,1,2,0, 0,2,3,0, 0,1,5,1, 0,4,5,1, 1,2,6,2, 1,5,6,2, 3,6,7,3,
3,2,6,3, 4,6,7,4, 4,5,6,4, 0,4,7,5, 0,3,7,5)
'
dim q1(4),v(4),vout(4),M_PI=3.14159265359,i%,j%,k%,x1,y1,z1,x2,y2,z2,x3,y3,z3
dim csort(5)
dim sortorder%(5)
DIM integer xx0(11), yy0(11), xx1(11), yy1(11), xx2(11), yy2(11), tcol(5) ,scol(11)
'arrays for calling the triangle draw routine
'
for i%=0 to 7 'scale the cube verticies up depending on the display size
for j%=0 to 2
cube(j%,i%)=cube(j%,i%)*scale/2
next j%
next i%
'
for i%=0 to 5 'scale the centroid positions up depending on the display size
for j%=0 to 2
centroid(j%,i%)=centroid(j%,i%)*scale/2
next j%
next i%
'
' define the colour of each face of the cube
tcol(0)=rgb(red)
tcol(1)=rgb(blue)
tcol(2)=rgb(magenta)
tcol(3)=rgb(yellow)
tcol(4)=rgb(cyan)
tcol(5)=rgb(green)

for i%=0 to 7 'convert coordinates to normalised form
x1=cube(0,i%): y1=cube(1,i%): z1=cube(2,i%)
create_vector(x1,y1,z1,v())
cube(0,i%)=v(2): cube(1,i%)=v(3): cube(2,i%)=v(4): cube(3,i%)=v(0)
next i%
'
for i%=0 to 5 'convert centroid coordinates to normalised form
x1=centroid(0,i%): y1=centroid(1,i%): z1=centroid(2,i%)
create_vector(x1,y1,z1,v())
centroid(0,i%)=v(2): centroid(1,i%)=v(3): centroid(2,i%)=v(4): centroid(3,i%)=v(0)
next i%

'create a quaternions to rotate 15 degrees about a chosen axis
'play with the x,y,z vector which is the axis of rotation
create_normalised_QUATERNION(15,1,0.5,0.25,q1())
'
do
for i%=0 to 7 'rotate coordinates
v(2)=cube(0,i%): v(3)=cube(1,i%): v(4)=cube(2,i%): v(0)=cube(3,i%): v(1)=0
rotate_vector(vout(),v(),q1())
cube(0,i%)=vout(2): cube(1,i%)=vout(3): cube(2,i%)=vout(4): cube(3,i%)=vout(0)
next i%
'
for i%=0 to 5 'rotate centroids
v(2)=centroid(0,i%): v(3)=centroid(1,i%): v(4)=centroid(2,i%): v(0)=centroid(3,i%):
v(1)=0
rotate_vector(vout(),v(),q1())
centroid(0,i%)=vout(2): centroid(1,i%)=vout(3): centroid(2,i%)=vout(4): centroid(3,i
%)=vout(0)
csort(i%)=vout(4)
sortorder%(i%)=i%
next i%
'

```

```

floatindexsort(csort(),sortorder%,6) 'sort the faces in the order of the depth
,
for i%=0 to 5 'project the coordinates onto the viewplane
j%=sortorder%(i%)*2 'get the furthest, as yet undisplayed, face from the viewer
k%=i%*2
z1=cube(2,cubetriangles%(0,j%))*cube(3,cubetriangles%(0,j%))+cubelocation(2)
x1=(cube(0,cubetriangles%(0,j%))*cube(3,cubetriangles%(0,j%))*viewplane/z1
+cubelocation(0))
y1=(cube(1,cubetriangles%(0,j%))*cube(3,cubetriangles%(0,j%))*viewplane/z1
+cubelocation(1))
z2=cube(2,cubetriangles%(1,j%))*cube(3,cubetriangles%(1,j%))+cubelocation(2)
x2=(cube(0,cubetriangles%(1,j%))*cube(3,cubetriangles%(1,j%))*viewplane/z2
+cubelocation(0))
y2=(cube(1,cubetriangles%(1,j%))*cube(3,cubetriangles%(1,j%))*viewplane/z2
+cubelocation(1))
z3=cube(2,cubetriangles%(2,j%))*cube(3,cubetriangles%(2,j%))+cubelocation(2)
x3=(cube(0,cubetriangles%(2,j%))*cube(3,cubetriangles%(2,j%))*viewplane/z3
+cubelocation(0))
y3=(cube(1,cubetriangles%(2,j%))*cube(3,cubetriangles%(2,j%))*viewplane/z3
+cubelocation(1))
xx0(k%)=x1 :yy0(k%)=y1 :xx1(k%)=x2 :yy1(k%)=y2 :xx2(k%)=x3 :yy2(k%)=y3 :scol(k%)=tcol(j%\
2)
k%=k%+1
z1=cube(2,cubetriangles%(0,j%+1))*cube(3,cubetriangles%(0,j%+1))+cubelocation(2)
x1=(cube(0,cubetriangles%(0,j%+1))*cube(3,cubetriangles%(0,j%+1))*viewplane/z1
+cubelocation(0))
y1=(cube(1,cubetriangles%(0,j%+1))*cube(3,cubetriangles%(0,j%+1))*viewplane/z1
+cubelocation(1))
z2=cube(2,cubetriangles%(1,j%+1))*cube(3,cubetriangles%(1,j%+1))+cubelocation(2)
x2=(cube(0,cubetriangles%(1,j%+1))*cube(3,cubetriangles%(1,j%+1))*viewplane/z2
+cubelocation(0))
y2=(cube(1,cubetriangles%(1,j%+1))*cube(3,cubetriangles%(1,j%+1))*viewplane/z2
+cubelocation(1))
z3=cube(2,cubetriangles%(2,j%+1))*cube(3,cubetriangles%(2,j%+1))+cubelocation(2)
x3=(cube(0,cubetriangles%(2,j%+1))*cube(3,cubetriangles%(2,j%+1))*viewplane/z3
+cubelocation(0))
y3=(cube(1,cubetriangles%(2,j%+1))*cube(3,cubetriangles%(2,j%+1))*viewplane/z3
+cubelocation(1))
xx0(k%)=x1 :yy0(k%)=y1 :xx1(k%)=x2 :yy1(k%)=y2 :xx2(k%)=x3 :yy2(k%)=y3 :scol(k%)=tcol(j%\
2)
next i%
cls
triangles(12,xx0(),yy0(),xx1(),yy1(),xx2(),yy2(),scol(),100000)
pause 1000
loop
end
,
sub create_normalised_QUATERNION(theta,x,y,z,q())
local radians = theta/180.0*M_PI
local sineterm= sin(radians!/2)
q(1)=cos(radians/2)
q(2)=x* sineterm
q(3)=y* sineterm
q(4)=z* sineterm
q(0)=sqr(q!(1)*q(1) + q(2)*q(2) + q(3)*q(3) + q(4)*q(4)) 'calculate the magnitude
q(1)=q(1)/q(0) 'create a normalised QUATERNION
q(2)=q(2)/q(0)
q(3)=q(3)/q(0)
q(4)=q(4)/q(0)
q(0)=1
end sub
,
sub invert_QUATERNION(n(),q())
n(0)=q(0)
n(1)=q(1)
n(2)=-q(2)
n(3)=-q(3)
n(4)=-q(4)

```

```

end sub
'
sub multiply_QUATERNION(n(),q1(),q2())
local a1=q1(1),a2=q2(1),b1=q1(2),b2=q2(2),c1=q1(3),c2=q2(3),d1=q1(4),d2=q2(4)
n(1)=a1*a2-b1*b2-c1*c2-d1*d2
n(2)=a1*b2+b1*a2+c1*d2-d1*c2
n(3)=a1*c2-b1*d2+c1*a2+d1*b2
n(4)=a1*d2+b1*c2-c1*b2+d1*a2
n(0)=q1(0)*q2(0)
end sub
'
sub create_vector(x,y,z,v())
v(0)=sqr(x*x + y*y + z*z)
v(1)=0
v(2)=x/v(0)
v(3)=y/v(0)
v(4)=z/v(0)
end sub

sub rotate_vector(vnew(),v(),q())
local n(4),iq(4)
multiply_QUATERNION(n(),q(),v())
invert_QUATERNION(iq(),q())
multiply_QUATERNION(vnew(),n(),iq())
end sub

```

Csub triangles 'draws multiple triangles with a delay between if required

```

00000000
8C820004 27BDFFA0 AFBF005C AFBE0058 AFB70054 AFB60050 AFB5004C AFB40048
AFB30044 AFB20040 AFB1003C AFB00038 AFA40060 AFA50064 AFA60068 184000EE
AFA7006C AFA00030 3C159D00 8FA30030 8FA50068 8FA80070 000310C0 00A22021
01021821 8C840000 8C630000 8FA90064 AFA40018 01223821 AFA3001C 8FA40074
8FA3006C 8FA80078 8FA9007C 00623021 00822821 01022021 01221021 8FA80018
8FA9001C 8CC60000 8CA50000 0128182A 8CF30000 8C9E0000 AFA60028 AFA5002C
10600006 8C540000 02601021 AFA90018 00C09821 AFA8001C AFA20028 8FA5001C
03C5102A 10400008 8FA90018 8FA20028 8FA8002C AFBE001C AFA80028 00A0F021
AFA2002C 8FA90018 8FA3001C 0069102A 10400008 8FA50018 8FA4001C 02601021
AFA40018 8FB30028 AFA9001C AFA20028 8FA50018 10BE008C 8FA4001C 109E00A4
8FA50018 2483FFFF 0065102A 1440002E 8FB00018 8FA90028 8FA2002C 8FA8001C
01334823 00531023 03C5B023 AFBE0034 0105B823 0280F021 AFA90020 AFA20024
00A08021 00009021 00008821 0060A021 0237001A 02E001F4 8FA30024 8FA20020
02002821 02003821 02228821 26100001 00002012 0256001A 02C001F4 00932021
02439021 00003012 00D33021 00C4182A 10600003 00801021 00C02021 00403021
8EA20048 AFBE0010 8C420000 0040F809 00000000 0290102A 1040FFE5 00000000
03C0A021 8FBE0034 03D0102A 14400030 8FA8001C 8FA90018 8FA4002C 8FA50028
0093B023 0085B823 02089023 02098823 03C91823 03C81023 72579002 72368802
AFB40018 AFB6001C AFB70020 0260B021 00A0B821 00609821 0040A021 0254001A
028001F4 8FA3001C 8FA20020 02002821 02003821 02429021 26100001 00002012
0233001A 026001F4 00972021 02238821 00003012 00D63021 00C4182A 10600003
00801021 00C02021 00403021 8FA80018 8EA20048 AFA80010 8C420000 0040F809
00000000 03D0102A 1040FFE4 00000000 8FA90080 8D240000 8D220004 00821025
5440001D 8EA20004 8FA20030 8FA40060 24420001 8C830004 AFA20030 000217C3
0043202A 5480FF52 8FA30030 14620006 8FA50060 8FA80030 8CA20000 0102102B
1440FF4B 8FA30030 8FBF005C 8FBF0058 8FB70054 8FB60050 8FB5004C 8FB40048
8FB30044 8FB20040 8FB1003C 8FB00038 03E00008 27BD0060 0040F809 00000000
1000FFE2 8FA20030 8FA80028 0113102A 1440001D 02603821 0268102A 1440001C
8FA70028 02603821 02602021 8FA9002C 0124102A 54400003 8FA4002C 00E9102A
0122380B 8EA20048 8FA30018 AFB40010 00E33821 8C420000 00602821 00803021
0040F809 00E43823 1000FFC2 8FA90080 8FA3001C 0065102A 1040FF5F 8FA90028
1000FF89 8FB00018 1000FFE8 8FA40028 1000FFE6 02602021 1440FFCC 8FBF005C
8C820000 5440FF10 AFA00030 1000FFC8 8FBF0058
End Csub

```

Csub floatindexsort ' sorts a floating point array but keeps an index so you know the original order

```

00000000
8CC60000 27BDFF98 AFB00040 AFBF0064 AFBE0060 AFB7005C AFB60058 AFB50054
AFB40050 AFB3004C AFB20048 AFB10044 AFA6001C AFA40068 AFA5006C 00C08021

```

```
3C039D00 001017C2 00508021 00108043 12000068 8FBF0064 8FA4001C 0204102A
1040FFF9 001017C2 8FA60068 00102080 00C42021 00102823 AFA50010 AFA40028
8FA5001C 8FA4006C 001010C0 AFA6002C 8FA60010 00821021 00B02823 AFA20024
AFA50034 AFA00020 0006F080 0006B8C0 8FA20020 8FA50028 0202B021 8FA20024
8CA60000 AFA50014 8C450000 02D0202A AFA60018 1480002A AFA50030 8FA60020
8FA2006C 000690C0 00529021 8FB50024 8FB40028 8FB1002C 1000000F 00C09821
8E220000 8E440000 8E450004 0213B021 02D0302A AE820000 023E8821 02401021
AEA40000 AEA50004 02579021 029EA021 14C00013 02B7A821 8FA40010 8E250000
8C620068 02649821 8FA40018 AFA30038 0040F809 AFB10014 2405FFFF 1045FFE8
8FA30038 8FA60068 00161080 8FA4006C 00C21021 0016B0C0 AFA20014 00961021
8FA60020 8FA50030 24C60001 AFA60020 000527C3 8FA60018 8FA50014 ACA60000
8FA50028 8FA60030 24A50004 AFA50028 AC460000 AC440004 8FA20024 8FA4002C
8FA50020 8FA60034 24420008 24840004 AFA20024 14A6FFB6 AFA4002C 001017C2
00508021 00108043 5600FF9C 8FA4001C 8FBF0064 8FBE0060 8FB7005C 8FB60058
8FB50054 8FB40050 8FB3004C 8FB20048 8FB10044 8FB00040 03E00008 27BD0068
End Csub
```

## APPENDIX

---

This page intentionally blank.