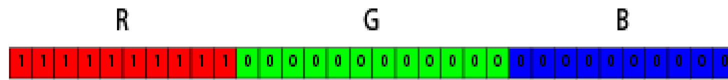
Introduction to Colour

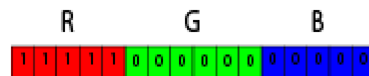
A brief side trip to explain the use of colour in the CMM2 may be in order, particularly for newcomers. This is a collection and perhaps a little expansion on details provided elsewhere in this manual.

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).



in this example, RGB(255,0,0)

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.



in this example, solid red, no green or blue.

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 **C**olour **L**ook**U**p **T**able. There is a detailed description of this in a later chapter titled CLUT.

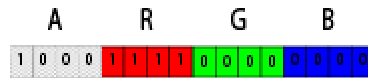
again, solid red, no green or blue.

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 imaging 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! (-:



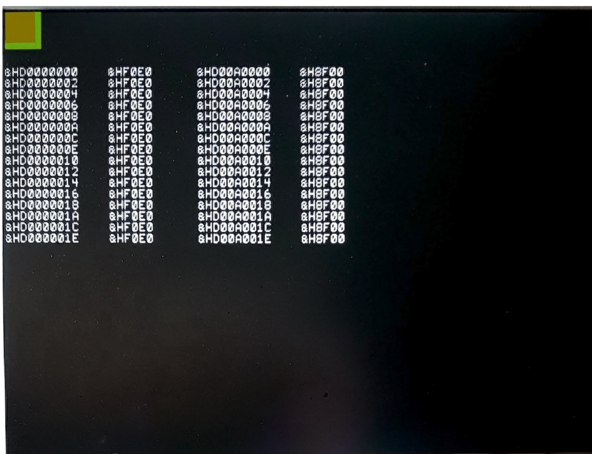
The way in which 12 bit colour depth appears to work is as follows:-

Set default to integer, put something on PAGE 0, eg. a small green box, change to PAGE 1 andwrite directly to PAGE 1 memory that will overlap the green box on PAGE 0 with transparency 8 (50%).

Now we dump some of the display memory for both pages, and finish sitting in a loop to see what we did.as follows;

The program below demonstrates.

```
option default integer

mode 7,12

‘ now write to PAGE 0 (layer 2)

box 0,0,40,40,,rgb(0,255,0,15),rgb(0,255,0,15) ‘ solid green on layer 2

page write 1 ‘ the top most, layer 3 (PAGE 1)

for row = 0 to 15
  for column = 0 to 40)
    poke short mm.info(page address 1)+(row*mm.hres+column)*2
  next column
next row

‘ any object would do, I just wanted to go to specific locations
‘ to make checking the result easier (-:
for k = 0 to 32 step 2
  a1$ = “&H”+hex$(mm.info(page address 0)  ‘ address
  a2$ = “   &H”+hex$(peek(byte mm.info(page address 0) +k + 1),2)  ‘ contents – high byte
  a3$ = hex$(peek(byte mm.info(page address 0) +k),2)  ‘ contents – low byte
  a4$ = “&H”+hex$(mm.info(page address 1)  ‘ address
  a5$ = “   &H”+hex$(peek(byte mm.info(page address 1) +k + 1),2)  ‘ contents – high byte
  a6$ = hex$(peek(byte mm.info(page address 1) +k),2)  ‘ contents – low byte
  text 0,60+k*6,a1$+a2$+a3$+”    “+a4$+a5$+a6$


do
loop
```



The memory dump in the code above shows (in part) the following:-

In PAGE 0 memory, there is &HF0E0 – in ARGB4444 format this means a (near) solid green.

…

In PAGE 1 memory, there is &H8F00 – in ARGB4444 format this means a solid red but a 50% transparency.

Looking at the display, we can see there that the end result is a dull yellow and in fact equates to 50% of the solid red area overlaying the solid green such that only 50% green area shines through. The visible effect is to halve the intensity of both layers then combine the result. (try it, display a box with colour rgb(127,127,0)).

You will also notice that the PAGE 0 memory is unchanged from the original green box– this is because the application of the transparency is carried out by the LTDC at actual output to the display monitor (ie. on the fly and not using any of MMBasic’s time and effort!).

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.