Images and Memblocks

Editor's note: Tutorial assumes you're working with 32-bit images, and not 16-bit.

What is a memblock?

Memblocks are blocks of memory allocated by the user to store any type of date. For this discussion, a memblock will be a chunk of memory containing an image.

Why use memblocks?

Memblocks allow you to access data which is stored directly in memory. This direct access is very quick, and that is useful when you need real-time effects and speed is an issue.

How do you use memblocks?

Because we will be using image data to demostrate memblocks, we will use the command make memblock from image. It's syntax is as follows:

```
make memblock from image memblock number, image number
make memblock from image 1, 5
```

The memblock number is used just like an object number. The image number is the image that the memblock will be constructed of. Now that we have a memblock containing the data of the image, how do we access that data? Whether you load a jpg, pcx, or bmp image, it's all stored the same way inside the memblock. There's no messy headers to mess with, which simplifies things greatly. First thing you should know is that a dword is 4 bytes. The first 12 bytes of data in the memblock contain info about the image.

Bytes:
0-3 = image width
4-7 = image height
8-11 = image depth

To retrieve this data, use the memblock dword() command.

```
width = memblock dword(1, 0)
height = memblock dword(1, 4)
depth = memblock dword(1, 8)
```

From this point on, the rest of the data in the memblock is the pixel information. Each color value is one byte. We typically think of the three primary colors are red, green, blue. But in the memblock, the colors are in reverse order to what we know. So starting at byte 12 (first byte after depth) we can extract the pixel's color information.

```
B = memblock byte(1,12)
G = memblock byte(1,13)
R = memblock byte(1,14)
A = memblock byte(1, 15)
```

These color components make up the color value for the first pixel in the image.(upper left corner) The "A" value is the alpha level of the pixel. B, G, R are the blue, green, and red channels, respectfully. Since we know the image's width and height, we can create a simple nested FOR loop to read in colors of the entire image.

```
for x = 1 to width
  for y = 1 to height
```

```
    location = ((y-1)*width + x - 1)*4 + 12
    B = memblock byte(1,location)
    G = memblock byte(1,location+1)
    R = memblock byte(1,location+2)
    A = memblock byte(1, location+3)
  next y
next x
```

Now obviously, this loop would serve no real purpose, because you're not doing anything with the values after you read them in. This is just an example to show you how you can easily read the data of the entire memblock of an image. Now for an explanation of the "**location**" variable. Given the X and Y coordinate for a pixel in the image, the correct memory location is calculated. Each pixel is every 4 bytes, because it takes 4 bytes of memory to store the color data of one pixel. 12 is added to the end of the equation as an offset, because the first actual 12 bytes of data isn't pixel information, its the width, height, depth data.
Now lets say you want to edit the image. For example, let's set the first pixel to a pink color. For this we use write memblock byte or write memblock dword.

```
write memblock byte 1, 12, 128 : `blue channel
write memblock byte 1, 13, 28 : `green channel
write memblock byte 1, 14, 255 : `red channel
write memblock byte 1, 15, 0
```

As far as I've been able to tell, the alpha value has shown no meaning. But if it does, it'll have an affect with the image's transparency. An alternative way of writing the pixel data is:

```
write memblock dword 1, 12, rgb(255,28,128)
```

The command, rgb(), returns a dword size value of a color made up of the red, green, and blue channel values specified. This will automatically write the proper numbers of each color channel to the appropriate bytes for you. The alpha value will be sent to either 0 or 255.

And finally, after we've done all the changes we want to the image's data, we must create a new image from that data using the command make image from memblock.

*note the order of the image number and memblock number. The DBP manual states that the first parameter is the memblock number. This is not true, the command expects the image number first, then the source memblock number.

```
make image from memblock image number, memblock number
make image from memblock 3, 1
```

You can overwrite the existing image number you created the memblock from, or create a whole new image. The source code below will load an image, create a memblock from it, change half the pixels of the image in the memblock to a random color, then create a new image. (You can overwrite the original image if you wish. This does not permanately overwrite your loaded image, the effect is only in memory. No actual data is ever written to the existing image file itself)

```
sync on

REM load an image
load image "fog.bmp", 4

REM create a memblock from the image
make memblock from image 1, 4
```

```
REM get the image's width and height
width = memblock dword(1,0)
height = memblock dword(1,4)

for x = 1 to width/2
  for y = 1 to height
    location = ((y-1)*width + x - 1)*4 + 12
    REM set each pixel in the image to a random color
    write memblock dword 1, location, rgb(rnd(255),rnd(255),rnd(255))
  next y
next x

REM create a new image from the memblock data
make image from memblock 3, 1

REM display the new image
repeat


  sprite 1,mousex(), mousey(), 4
  sprite 2,mousex()+width+5, mousey(), 3

  sync
until spacekey()
```