Articles: DBPRO Memblock Reference

What are memblocks?

This reference guide demonstrates the use of memblock commands so that you can use them to manipulate 3D model data, images, sounds and instructions directly. Such information can be shared between DLLs or over a network. Find out what memblocks are...

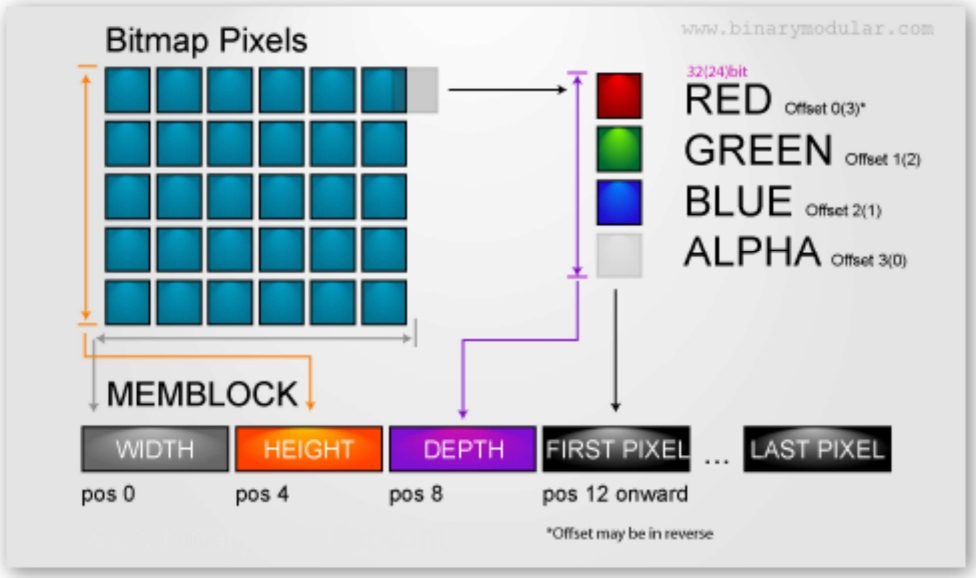
Make Memblock

Creates a block of data at a given size in physical memory used for storing information and building content using the listed memblock commands. This form of memblock creation requires a user specified data format per element.

Make Memblock From File

Creates a memblock at the size of an opened file; the memblock is populated with the data of the file specified.

Make Memblock From Bitmap

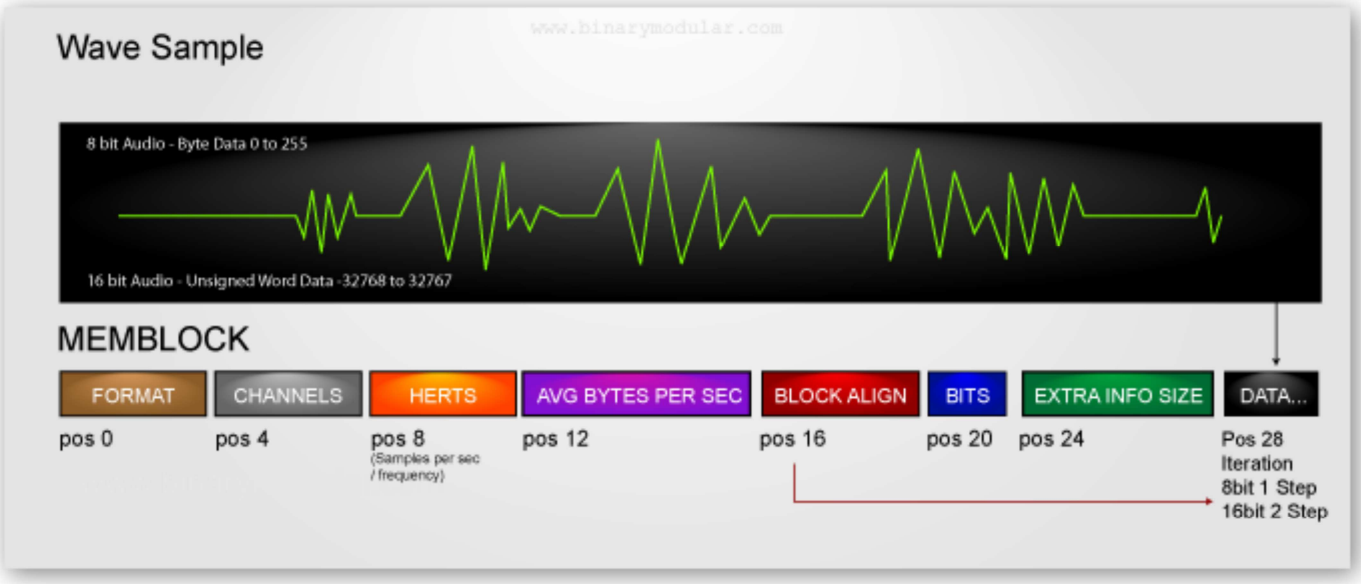


Creates a new memblock which consists of bitmap data. The data can be manipulated and displayed as a bitmap on the screen, or saved to file.

Make Memblock From Image

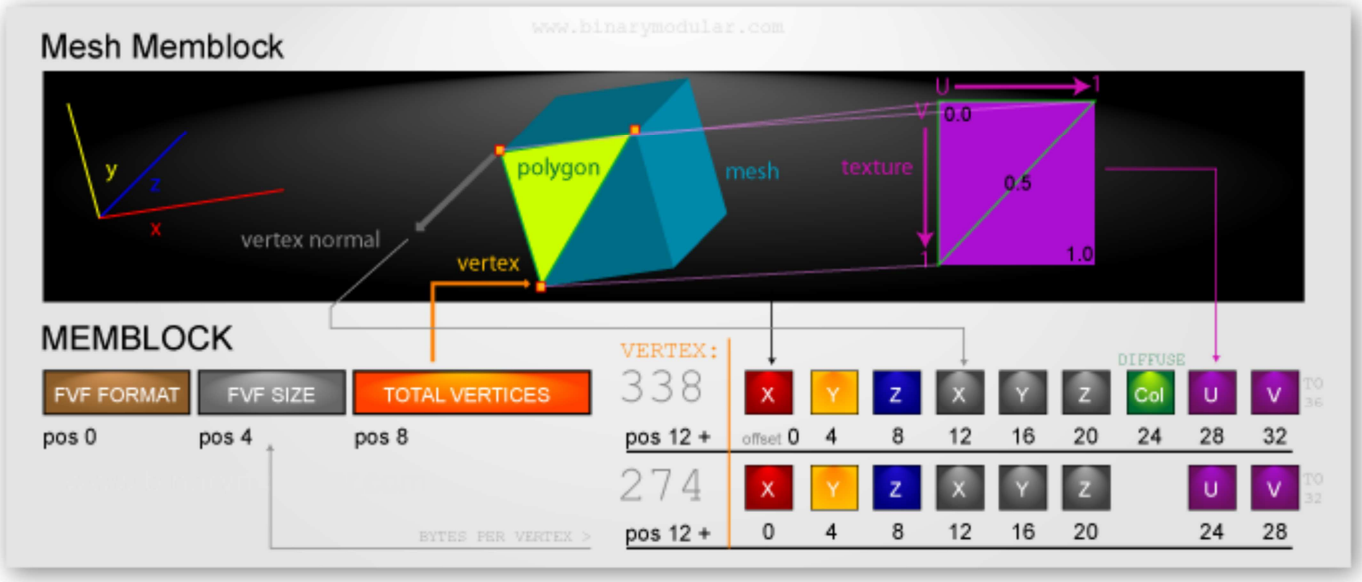
Creates a new memblock which consists of image data, similar to the bitmap equivalent. The data can be manipulated and displayed as a bitmap on the screen, or saved to file.

Make Memblock From Sound



Creates a new memblock which consists of sound data. The data can be manipulated and converted back into a sound to be played or saved to a wave file.

Make Memblock From Mesh



Creates a new memblock which consists of 3D object properties obtained from a mesh. The data can be manipulated and saved back into a mesh, which inturn can be appended to an existing 3D object, added as a new limb in a 3D object, used to replace an existing limb in a 3D object, added into vertexdata or saved as a direct X file.

What are memblocks?

Like directories which contain files, memblocks are a group of information stored in RAM, but the data here is made ready for speedy access compared to hard drive files.

Memblocks are like a block of text; this paragraph block contains a series of words at different lengths. Therefore, in a memblock where we treat each word individually, we make ourselves aware of where the words are, what order they are in and how long each word is. Furthermore, each word consists of individual letters represented by a small chunk of data called a byte.

We will briefly discuss what we can do with these bytes, then we will get into creating memblocks.

Data bytes

Bytes are by no means edible, they are 8-bit numbers. They are a series of 8 on or off, 1 or 0 values we call bits; therefore one byte is an 8 bit chunk of data and it counts from the number0 to the number 255*. We cannot store numbers greater than 255 or numbers lower than zero inside of a byte.

A series of bytes can be used to store text. The word 'UP' is said to be 2 bytes worth of data, this is because the system recognises the capital letter U by its ASCII code 85, and the uppercase P as the code 80.

Bytes can also be used to store 8 true or false values; also known as boolean values or flags, due to containing 8 bits. In the following code example, all of the if blocks except the last one will return true. Copy and paste into your editor for a better view:

```
` Decimal flags
#constant FOR_SALE = 1
#constant NEW_ITEM = 2
#constant SOLD = 4
#constant DAMAGED = 8
#constant SECOND_HAND = 16

` Binary values of 1,2,4 and 8 (set from right to left with 4 bits using 4 digits because the other 4 default to zero anyway
` [%0001 and %0001] is true because the first [1] is at the same position in the other
` [%0011 and %0010] is [3 and 2] and is true because the third [1] is also true in the second number
#constant IS_OLD = %0001
#constant IS_SPECIAL = %0010
#constant IS_CONSUMABLE = %0100
#constant IS_BOUND_TO_PLAYER = %1000

Gloves = FOR_SALE + NEW_ITEM
Helmet = IS_SPECIAL + IS_OLD
Certificate = IS_SPECIAL + IS_BOUND_TO_PLAYER

if Gloves and NEW_ITEM then print "The gloves are new"
if ( Gloves and NEW_ITEM ) or ( Gloves and FOR_SALE ) then print "The gloves are new or for sale"
if ( Helmet and IS_CONSUMABLE ) = 0 then print "The helmet is not consumable"
` Cannot write: if Helmet and IS_CONSUMABLE = 0 then, it would turn out to become
` if 3 and 4 = 0 which returns false because 4 is not 0. Parentheses ( ) was use to control
` logical precedence

` The following will return false
```

```
if Certificate and IS_OLD then print "The certificate is not old so this message will not display"

wait key
```

Knowing such information will come in handy when using switches or flags that determine whether something is switched on or off. How we store these bytes is simular to how we store any other information, it is written, just like in this document, in a certain format.

Formats

Data formats are like documents which specify what information they contain, where the information is and how long the information is. In Dark Basic Professional, memory blocks can be used to process information according your own format or a ready made one, such as the image format.

The CPU does not need to figure out what format the information is in or what types of data memblocks contain, it is down to you to tell the computer where the information is and how large each chuck of data is; this is due to memory blocks simply being a list of bytes. Unlike arrays, the data-types of elements in memblocks can be changed at runtime; you can store any kind of data almost anywhere in the memblock so as long as there is room for it.

This reference guide demonstrates the use of memblock commands so that you can use them to manipulate 3D model data, images, sounds and other content. Such information can be shared between DLLs or over a network.

Begin by learning how to create a memblock

** A datatype with a range that starts from zero is known as unsigned, such as a byte in DBPRO. An integer is a signed datatype, so it can store negative values; numbers below zero*

Make Memblock

Creates a block of data at a given size in physical memory used for storing information and building content using the memblock commands listed below.

Make Memblock *number byte*, *size dword*

COPY MEMBLOCK
DELETE MEMBLOCK
MAKE FILE FROM MEMBLOCK
MAKE BITMAP FROM MEMBLOCK
MAKE IMAGE FROM MEMBLOCK
MAKE SOUND FROM MEMBLOCK
MAKE SOUND FROM MEMBLOCK
MAKE MESH FROM MEMBLOCK
MAKE MESH FROM MEMBLOCK
CHANGE MESH FROM MEMBLOCK
MAKE ARRAY FROM MEMBLOCK
WRITE MEMBLOCK BYTE
WRITE MEMBLOCK WORD
WRITE MEMBLOCK DWORD
WRITE MEMBLOCK FLOAT

You also have access to the following expressions which return data stored in the memblock or information about it.

MEMBLOCK EXIST
GET MEMBLOCK PTR - For DLL access to the memblock
GET MEMBLOCK SIZE
MEMBLOCK BYTE
MEMBLOCK WORD
MEMBLOCK DWORD
MEMBLOCK FLOAT

and also the WRITE MEMBLOCK command to save memblocks to file, and the READ MEMBLOCK to load memblocks from file.

There are also memblock networking commands in Dark Net.

The [Make Memblock] command requires a user specified data format per element using commands such as write memblock byte or write memblock float.

Support

DBPRO and GDK

Technical Explanation

Like directories which contain files, memblocks contain dynamic information stored in RAM, ready for speedy access without having to access the hard drive.

Unlike arrays, the data-types of elements in memblocks are unlimited and can be specified at runtime; you can store any

kind of data almost anywhere in the memblock.

Read more..

You must specify a free memblock number used for reference. The size of memblock must be large enough to store what you need inside it, here are the datatypes along with their sizes that you can use:

BYTE : RANGE 0-255 : SIZE 1
WORD : RANGE 0-65535 : SIZE 2
DWORD : RANGE 0-4294967295 : SIZE 4 (Unsigned integer)
FLOAT: RANGE +/- 7 floating point digits : SIZE 4

Supposing you wanted to store a item at a location in 3D space, we could use a BYTE to store the item type, a WORD to store its identity and 3 FLOAT numbers to store the location. The following example demonstrates this:

```
MemblockNumber=1
#constant BAG = 1

type TItem
ItemType
ID
X#
Y#
Z#
endtype

global Sackbag as TItem

Sackbag.ItemType = BAG
Sackbag.Id = 10
Sackbag.x# = 10.2
Sackbag.y# = 25.5
Sackbag.z# = -20.124

// Create a memblock to store our item using a calculation to
` illustrate the size of the memblock against the size of the values
size = 1+2+4+4+4
make memblock MemblockNumber, size

write memblock byte MemblockNumber, 0, Sackbag.ItemType
write memblock word MemblockNumber, 1, Sackbag.Id
write memblock float MemblockNumber, 3, Sackbag.x#
write memblock float MemblockNumber, 7, Sackbag.y#
write memblock float MemblockNumber, 11, Sackbag.z#
print "Written data to physical memory (RAM)"
print "Press any key to save to disk"
wait key

// Save the memblock to file
`delete file "File.dat"
open to write 1, "File.dat"
write memblock 1, MemblockNumber
close file 1

delete memblock MemblockNumber
print "Memblock deleted. Memblock exists returned: "; memblock exist( MemblockNumber )
wait 2000
```

```
print "No"
wait 1000
cls

// Remove the memblock from the physical memory


// Open back the file
print "Load memblock from file"
open to read 1, "File.dat"
read memblock 1, MemblockNumber


// Output the information
print "Memblock size in bytes: "; get memblock size( MemblockNumber )
print "Item type from memblock: "; memblock byte( MemblockNumber, 0 )
print "Item ID from memblock: "; memblock word( MemblockNumber, 1 )
print "Item position from memblock: ";
print memblock float( MemblockNumber, 3 ); ", ";
print memblock float( MemblockNumber, 7 ); ", ";
print memblock float( MemblockNumber, 11 )
print
print "Press any key to end"
wait key
```

Related Articles

What are memblocks?

Beginner Notes

Memblocks can be used to create 3D models from code, but this is not the quickest way to go about it in terms of processing time. Consider using vertex data on existing objects or the Make Object New command in the Matrix Utils plugin by IanM.

Large Numbers and Strings
To store large numbers such as Double Floats, you will need to split the value in half and store each half in their own position in the memblock. It is easier to use an array to store such numbers in memory. Like memblocks, array lists can also be saved to file; and be sent over a network by submitting the bytes in the file.

To store strings, you may use the Mid$(SearchText, Tndex) function to return an ASCII coded byte value for each character in the search text, however this should only be used where it is necessary to work with memblocks; it is easier to store strings in arrays and variables.

Make Memblock From File

Creates a memblock at the size of an opened file; the memblock is populated with the data of the file specified.

Make Memblock From File memblocknumber *byte*, filenumber *(0-32)*

Support

DBPRO and GDK

Technical Explanation

You can use string, numeric and memblock expressions to obtain strings, numbers and memblocks from file respectively using [READ STRING], [READ BYTE/WORD/FLOAT etc] and [READ MEMBLOCK]. However, besides the [READ BYTE FROM FILE] expression, file read expressions will only extract information in sequencial order, memblock expressions can get information at any location in the memblock using an index; doing so allows you to obtain a variable amount of information in any order, without having to use the [SKIP BYTES] command.

Using this feature provides a useful method for reading your own file format that can contain data such as text, images, sounds and 3D models; but there are other ways to read and write such information to file.

Example

```
// Free a file to write a random landscape
if file exist("Land.dat") then delete file "Land.dat"
open to write 1, "Land.dat"

for heightX = 0 to 29
    for heightZ = 0 to 29
        // Save a random height, 60 random steps between 0 and 0.6
        write float 1, rnd(60) * 0.01
    next heightZ
next heightX

// Save and close the file
close file 1

// Load the file back from disk and create a memblock from it
open to read 2, "Land.dat"
make memblock from file 1, 2

// We can use an index to obtain information at any position in the memblock; but with files you cannot
print "Second tile height: "; memblock float(1, 4)
wait 2000

// Make matrix for the terrain
make matrix 1, 30, 30, 30, 30

// Store the last float (4 byte) index
```

```
i = ( get memblock size( 1 ) / 4 ) - 1

// Get the heights in what ever order we wish; we are going in reverse
// By using a memblock reference we do not need to go from the start to the end
for heightX = 0 to 29
    for heightZ = 0 to 29
        height# = memblock float(1, i*4)
        set matrix height 1, heightX, heightZ, height#
        dec i
    next heightZ
next heightX

update matrix 1
sync on : sync rate 60
position camera 0, 3, 0
rotate camera 0, 45, 0


do
    control camera using arrowkeys 0, 0.5, 8
    sync
loop
```

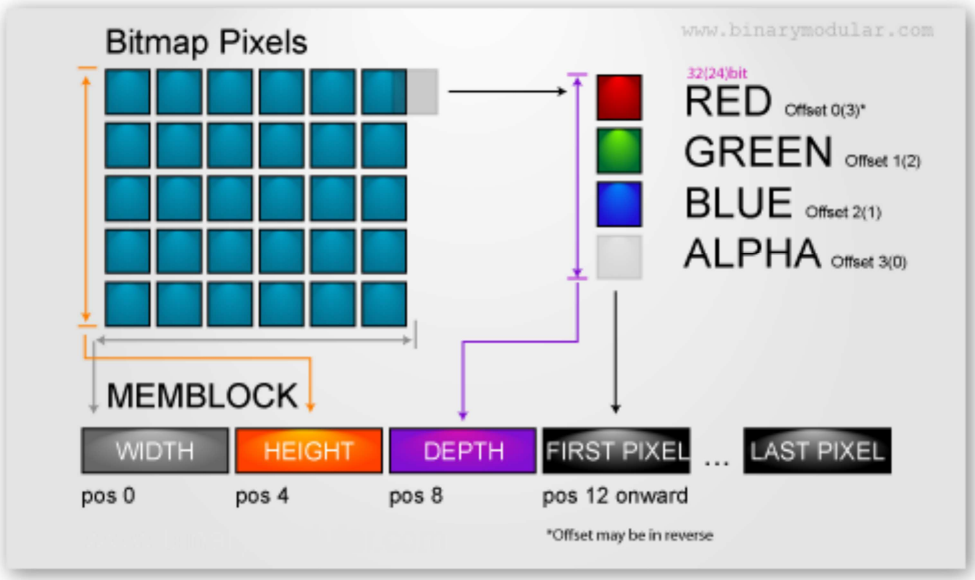
Related Article

Methods of writing and reading back files in DBPRO

Make Memblock From Bitmap

Creates a new memblock which consists of bitmap data. The data can be manipulated and displayed as a bitmap on the screen, or saved to file.

make memblock from bitmap
memblock ID *int*, bitmap ID *int*



Support

DBPRO and GDK

Technical Explanation

An image in programming terms is often called a bitmap because it contains a series of bits consisting of colour information. The dots are technically known as pixels and usually contain 3 to 4 bytes (or 24 to 32 bits, due to there being 8 bits per byte). A 32 bit image, contains a fourth byte that stores the transparency of the pixel. (Note that 24-Bit png images actually contain an additional 8 bits used for the alpha channel). When the alpha channel is set to zero, the pixel is transparent, and when set to 255 the pixel is opaque.

The other three bytes contain levels of red, green and blue to mix together to create the resulting hue and tone, where 0 adds nothing and 255 adds the maximum. When all three channels, RGB, are 0, the tone is black. When all three channels are 255 in value, the tone is white. When there is 255 in red and 127 in green, the hue is orange; and when there is 255 in blue and 127 in red the hue is Violet. The more equal the RGB channels are, the less saturated the result; thus a value of 127 in each channel will produce a medium grey.

When iterating through a memblock of pixels, you must start from position 12. The pixel at this location in the memblock is always 0,0; the top left of the image.

Example

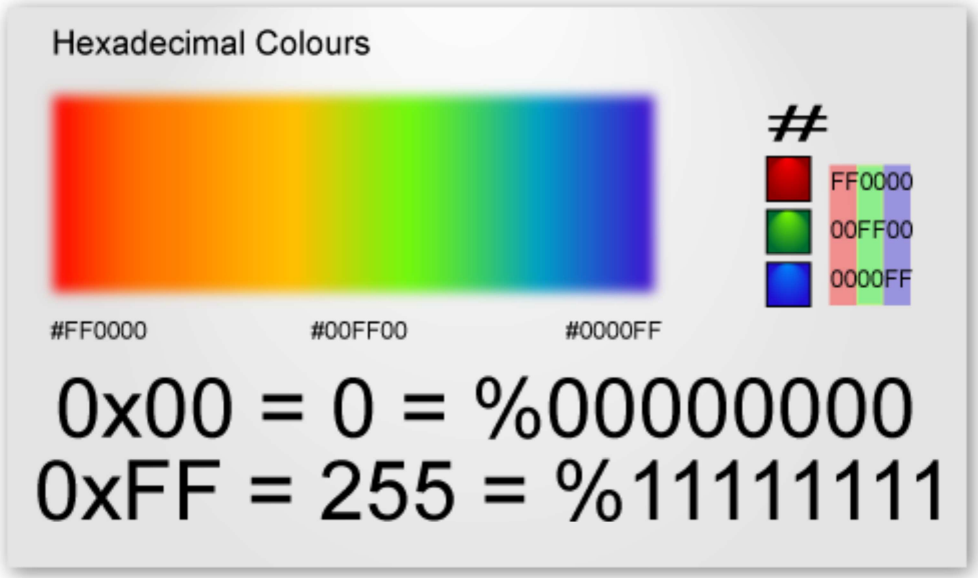
The following example demonstrates basic use of the bitmap memblock creation command.

```
` Basic bitmap memblock use
set display mode 800, 600, 32

` Create a small bitmap half filled with red
set bitmap format 21
box 1,1,6,4,rgb(255,5,2),rgb(255,5,2),rgb(255,5,2),rgb(255,5,2)
create bitmap 1, 6, 4

` Copy the red onto the current bitmap from the screen; which is bitmap zero
copy bitmap 0, 0, 0, 6, 4, 1, 0, 0, 6, 4

` Set the current bitmap back to the screen
set current bitmap 0
```

```
cls

` Create a new memblock from bitmap 1
MemblockNumber = 1
make memblock from bitmap MemblockNumber, 1

` Get the dimensions of the memblock and display them
BitmapWidth = memblock dword( MemblockNumber, 0 )
BitmapHeight = memblock dword( MemblockNumber, 4 )
BitmapDepth = memblock dword( MemblockNumber, 8 )
print "Bitmap width is "; BitmapWidth
print "Bitmap height is "; BitmapHeight
print "Bitmap depth is "; BitmapDepth


` Set pixel index step and co-ordinates to zero
s = 0
x = 0
y = 0

` Display the pixel data from position 12 to the end (-1) at increments of 4
for i = 12 to get memblock size( MemblockNumber ) - 1 step 4

    ` Increase the index step
    inc s

    ` After the last column, increase the row id and reset the column to zero
    if x => BitmapWidth
       inc y
       x = 0
    endif

    ` Get the pixel data, from blue, green, red to alpha; then display the result
    b = memblock byte( MemblockNumber, i)
    g = memblock byte( MemblockNumber, i+1)
    r = memblock byte( MemblockNumber, i+2)
    a = memblock byte( MemblockNumber, i+3)
    print "Step"; s; " X"; x; " Y"; y; " Index: "; i; " - Colour: A";a; ", R"; r; ", G"; g; ", B"; b


    if x < 2
       ` Change the first two columns of pixels on the X axis to green
       write memblock byte MemblockNumber, i+3, 128 ` Alpha
       write memblock byte MemblockNumber, i+2, 0 ` Red
       write memblock byte MemblockNumber, i+1, 255 ` Green
    write memblock byte MemblockNumber, i, 0 ` Blue
    endif

    ` Next column of pixels
    inc x

next i

make bitmap from memblock 3, MemblockNumber

print
print "Click to paste the bitmap"
```

```
` Save the image to see the semi transparent green zoomed in because the copy bitmap command will not
` show the effect well.
get image 2, 0, 0, 6, 4, 3
save image "Alpha.png", 2
delete image 2


do

    ` Copy the bitmap onto the screen when the mouse is clicked, at a larger size
    if mouseclick() = 1
        mx = mousex()
        my = mousey()
        bw = bitmap width( 3 )
        bh = bitmap height( 3 )
        copy bitmap 3, 0,0, bw, bh, 0, mx, my, mx+bw+60, my+bh+40
        set current bitmap 3
        wait 200
    endif

loop
```

Hexadecimals



Sometimes you may have to deal with hues and tones expressed with hexadecimals, prefixed with a '#' hash in many programs or with '0x' in DBPRO. Counting from 0 to 15, hexadecimals are written as 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E and F. The hexadecimal of 16 is 10; multiply that by two and you have 32 which is written as 20 in hexadecimal form. 31 is written as 1F, 47 is written as 2F and 63 is written as 3F.

In the image illustration the first two digits represent red, the second two represent the green and the third represents the blue. The hexadecimal value of FF is 255, and the hexadecimal value of 7F is 127; therefore the colour 0xFF7F00 is the same as rgb(255,127,0)

Begineers Notes

There are also other ways to manipulate images or bitmaps. You can use the Basic 2D commands such as [point] and [dot]; inbetween [lock pixels] and [unlock pixels] for extra speed. There are also plugins such as Extendsand Sven's Image Kit that offer further image manipulation commands.

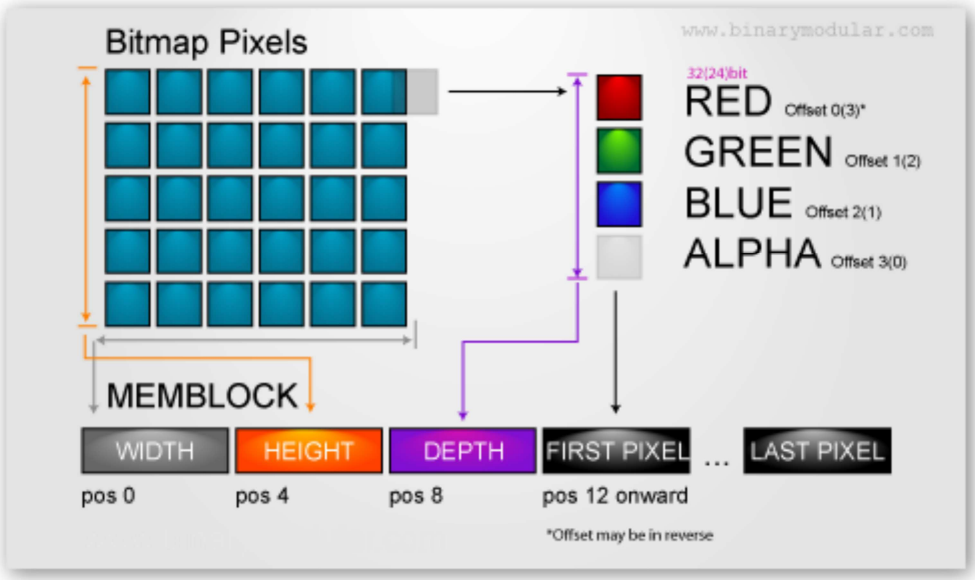
Example

The following example performs as wave effect on a bitmap which was generated, but can also be loaded from a file or extracted from the camera view. It also demonstrates interpreting hexadecimal colours.

```
` Set desired display mode
set display mode 640, 480, 32

` Display information
cls
print "Move the mouse left and right to adjust waviness"
wait 3000

` Setup the screen refresh mode
sync on
sync rate 60

` Draw a gradient background
Colour1 = rgb( 0x7F, 0, 0 ) ` Medium Red (#7F0000)
Colour2 = rgb( 0x7F, 0, 0x7F ) ` Medium Magenta (#7F007F)
Colour3 = rgb( 0, 0x7F, 0 ) ` Medium green (#007F00)
Colour4 = rgb( 0, 0, 0x7F ) ` Medium Blue (#00007F)
box 0, 0, screen width() - 1, screen height() - 1, Colour1, Colour2, Colour3, Colour4

` Set the ink to white (#FFFFFF)
ink rgb(0xFF,0xFF,0xFF), 0

` Setup text
set text font "Arial"
set text size 40
text$ = "Memblock Bitmaps"
LineSpacing = screen height() / 20
WordSpacing = text width( text$ ) + 5
TextEndX = screen width()
TextEndY = screen height()

` Repeat the text down and across the screen
for x = 0 to TextEndX step WordSpacing
    for y = 0 to TextEndY step LineSpacing
       text x, y, text$
    next y
next x

` Copy the drawing to two bitmaps
make memblock from bitmap 1,0
make memblock from bitmap 2,0



` Declare the step of the effect
LoopStep# = 0.0

` Declare the depth of the bitmap
pixelDepth = 4 ` ARGB (32bits)

` Declare the position of the last pixel in bytes * depth
```

```
pixelEnd = screen width() * pixelDepth

` Store the last position of colour data of the last pixel
lastByte = get memblock size( 2 )

` Calculate the position of the last line, once, and store for use in the loop
lastLine = screen height() - 1

do

    ` Multiply the ratio of the mouse position on the X axis by 100, and store the result
    WaveFreq# = 100 * ( ( mousex() + 1.0 ) / ( screen width() * 1.0 ) )

    ` Loop from the first horizontal line to the last line of pixels
    for i = 0 to lastLine

    sourcePixelStart = 12 + ( i * pixelEnd )
    LoopStep# = wrapvalue( LoopStep# + 0.7 )
    wave = WaveFreq# + ( sin( LoopStep# ) * WaveFreq# )
    destinationPixelStart = 12 + (i * pixelEnd) + ( wave * pixelDepth )
    if destinationPixelStart + pixelEnd < lastByte
       copy memblock 1, 2, sourcePixelStart, destinationPixelStart, pixelEnd
    endif

    next i

    ` Display the result of memblock 2
    make bitmap from memblock 0,2

    ` Update the screen
    sync

loop
```

Make Memblock From Image



Creates a new memblock which consists of image data, similar to the bitmap equivalent. The data can be manipulated and displayed as a bitmap on the screen, or saved to file.

make memblock from image*memblock ID int, image ID int*

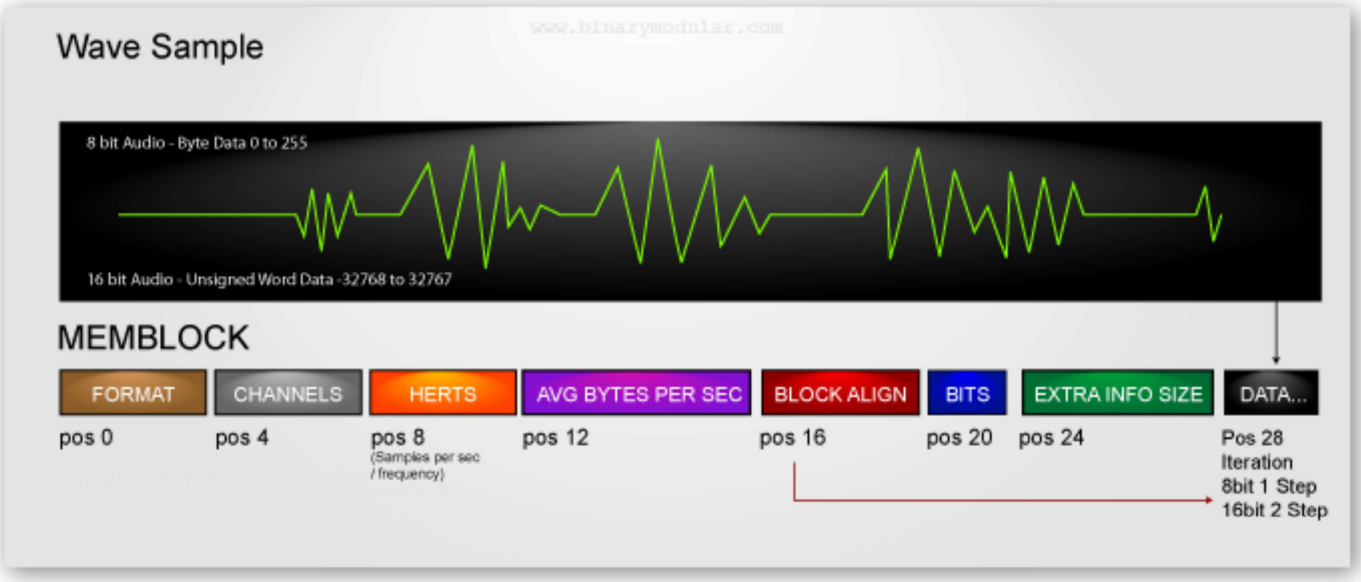
An image in programming terms is often called a bitmap because it contains a series of bits consisting of colour information. The dots are technically known as pixels and usually contain 3 to 4 bytes (or 24 to 32 bits, due to there being 8 bits per byte). A 32 bit image, contains a fourth byte that stores the transparency of the pixel. (Note that 24-Bit png images actually contain an additional 8 bits used for the alpha channel). When the alpha channel is set to zero, the pixel is transparent, and when set to 255 the pixel is opaque.

The other three bytes contain levels of red, green and blue to mix together to create the resulting hue and tone, where 0 adds nothing and 255 adds the maximum. When all three channels, RGB, are 0, the tone is black. When all three channels are 255 in value, the tone is white. When there is 255 in red and 127 in green, the hue is orange; and when there is 255 in blue and 127 in red the hue is Violet. The more equal the RGB channels are, the less saturated the result; thus a value of 127 in each channel will produce a medium grey.

When iterating through a memblock of pixels, you must start from position 12. The pixel at this location in the memblock is always 0,0; the top left of the image.

Read more information in the make memblock from bitmap page.

Make Memblock From Sound



make memblock from sound MemblockId *int*, SoundId *int*
Creates a new memblock which consists of sound data. The data can be manipulated and converted back into a sound to be played or saved to a wave file.

Technical Explanation

The wave file consists of header information as shown in the image above.

Format
Offset 0

The format of the wave file is generally the PCM (Pulse Code Modulation) format, which is represented by the return value of 1.

Channels
Offset 4

The number of channels generally determines whether the audio file is mono or stereo; respectively playing one audio stream in all speakers, or one audio stream for the left speaker and another for the right speaker.

Samples Per Second (Frequency Hz)
Offset 8

Measured in Hertz (Hz), the number of sample values that exist per second in the wave file are stored at offset 8 of the memblock. Usually, this will be 22,050 Hz or 44,100 Hz (or 22.050 KHz & 44.100 KHz)

Average Bytes Per Second
Offset 12

The average bytes per second value simply returns the average number of bytes used to store audio spanning one second in the wave file.

Block Alignment
Offset 16

The size of each data block in the wave file. Generally 1 byte for 8 bit audio and 2 bytes (Unsigned Word) for 16bit audio.

Bits per sample
Offset 20

Simply returns the number of bits used to store each data block; generally 8 or 16 bits.

Extra info (cbSize)
Offset 24

Generally used in non-PCM formats to specify an extended data offset size to accommodate more information. This will generally return zero when the PCM format is used.

Example

The following example loads a mono wave file and displays its wave form. A wave file needs to be supplied under the given filename for the example to work.

```
` Wave file required
load sound "Audio.wav",1

memblockId = 1
make memblock from sound memblockId,1

global x, y, width, height, time
x = 0 : y = 240 : width = screen width() : height = 100
time = 4000

` Get total bits and samples per Second and display some information
Bits = memblock dword(memblockId,5*4)
BPS = Bits/8
Herts = memblock dword(memblockId,2*4)
print "Bits per Sample: "; Bits; "bit"
print "Herts (Samples per second): "; herts; "hz"

` Get the duration of the samples and display more information
Duration = get memblock size(memblockId)-(7*4)
herts# = herts
seconds# = time/1000.0
ExtraInfoBytes = memblock dword(memblockId,7*4)
Channels as byte : Channels = memblock dword(memblockId,4)

if Channels = 1
    print "Mono"
else
    print "Stereo"
endif

TotalDuration# = ( Duration * 1.0 ) / ( Channels * Herts * Bits / 8.0 )
print "----------------------------------------"
```

```
print "Total Duration: "; str$( TotalDuration#, 1 ); " seconds"
print "Display Duration: "; str$( seconds#, 1 ); " seconds"

MaxDuration=duration
Duration = seconds#*herts#
if duration > maxduration then duration = maxduration

` Gather information for loop
SamplePTR = 7*4
TimeSpan# = (duration/BPS) - 1.0
steps = TimeSpan# / width
print "Drawing step count: "; steps
lx#=x
ly#=y
SampleB as byte
Sample as word


` Draw the wave data

for i=0 to TimeSpan# step steps

    if BPS=1

       SampleB = memblock byte(memblockId,SamplePTR)
       inc SamplePTR, steps
       Percent#=Sampleb
       Percent#=(Percent#-128)/255.0
    endif

    if BPS=2
       Sample = memblock word(memblockId,SamplePTR)
       SampleSign = Sample && 32768
       inc SamplePTR, steps * 2
       Percent#=Sample
       if SampleSign<>0 then Percent#=( 32767 - ( Sample && 32767 ) ) * -1
       Percent#=Percent# / 32768.0
    endif

    ` Draw the next line
    Percent#=Percent#*height
    x# = width
    x# = x# / TimeSpan#
    x# = x# * i
    line lx#, ly#, x+x#, y+Percent#
    lx# = x + x#
    ly# = y+ Percent#

next i

print "Press any key to exit"

wait key
```

Example

The following example demonstrates altering sounds using an effect

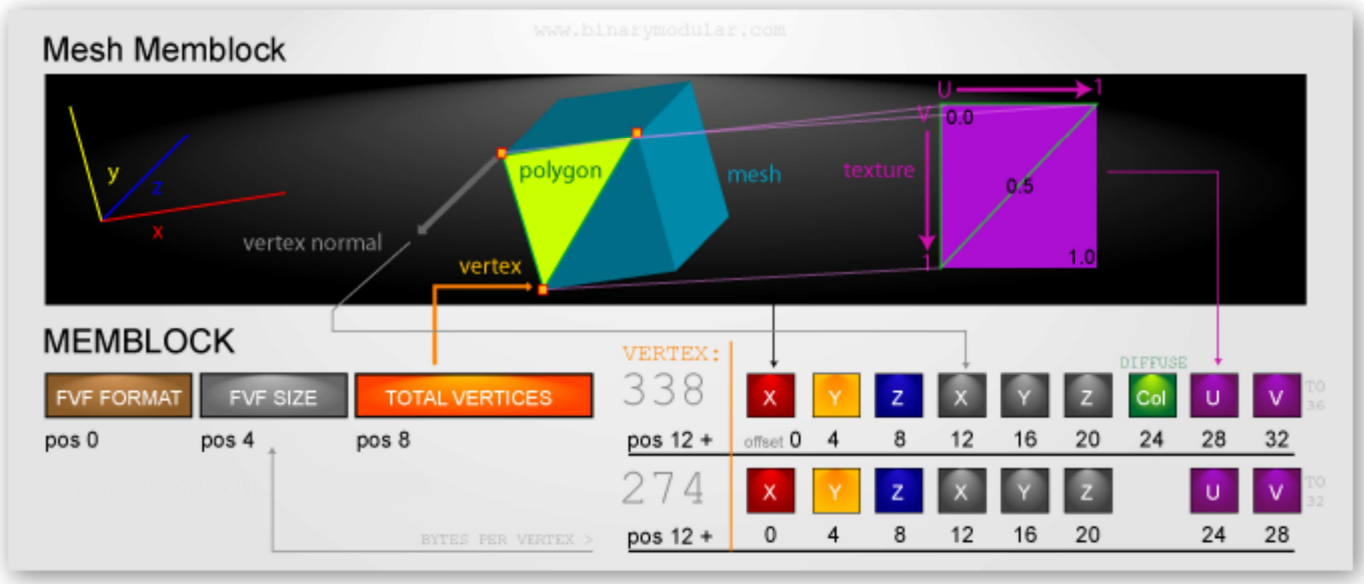
```
` Wave file required
`load sound "Audio.wav",1
load sound "C:\Documents and Settings\Binary Modular\Desktop\Binary
Modular\Source\Games\MathsTower\Data\MTXOperator.wav", 1
memblockId = 1
make memblock from sound memblockId,1
make memblock from sound memblockId+1,1

global x, y, width, height, time
x = 0 : y = 240 : width = screen width() : height = 100
time = 4000

` Get total bits and samples per Second and display some information
Bits = memblock dword(memblockId,5*4)
BPS = Bits/8
Herts = memblock dword(memblockId,2*4)
print "Bits per Sample: "; Bits; "bit"
print "Herts (Samples per second): "; herts; "hz"

` Get the duration of the samples and display more information
Duration = get memblock size(memblockId)-(7*4)
LastPosition = get memblock size(memblockId)-1
herts# = herts
seconds# = time/1000.0
ExtraInfoBytes = memblock dword(memblockId,7*4)
Channels as byte : Channels = memblock dword(memblockId,4)

if Channels = 1
    print "Mono"
else
    print "Stereo"
endif

TotalDuration# = ( Duration * 1.0 ) / ( Channels * Herts * Bits / 8.0 )
print "----------------------------------------"
print "Total Duration: "; str$( TotalDuration#, 1 ); " seconds"
print "Display Duration: "; str$( seconds#, 1 ); " seconds"

MaxDuration=duration
Duration = seconds#*herts#
if duration > maxduration then duration = maxduration

` Gather information for loop

ModulationStep# = 0
TimeSpan# = (duration/BPS) - 1.0
steps = TimeSpan# / width
SampleB as byte
Sample as word
ink rgb( 0, 255, 128 ), 0
```

```
for Process = 1 to 2

    SamplePTR = 7*4

    for i=0 to TimeSpan# step steps

        ModulationStep# = wrapvalue( ModulationStep# + 0.001 )

        if BPS=1

            SampleB = memblock byte(Process,SamplePTR)
            if Process = 2
                SampleB = AddNoiseToSample( SampleB, BPS, ModulationStep#, 1 )
                write memblock byte Process, SamplePTR + rnd(20), SampleB
            endif
            inc SamplePTR, steps
            Percent#=SampleB
            Percent#=(Percent#-128)/255.0

        endif

        if BPS=2

            Sample = memblock word(Process,SamplePTR)
            if Process = 2 and SamplePTR < LastPosition - 40
                Sample = AddNoiseToSample( SampleB, BPS, ModulationStep#, 1 )
                write memblock word Process, SamplePTR + ( rnd(20) * 2 ), Sample
            endif

            SampleSign = Sample && 32768
            inc SamplePTR, steps * 2
            Percent#=Sample
            if SampleSign<>0 then Percent#=( 32767 - ( Sample && 32767 ) ) * -1
            Percent#=Percent# / 32768.0

        endif

        if Process = 1
            ` Draw the next line
            Percent#=Percent#*height
            x# = width
            x# = x# / TimeSpan#
            x# = x# * i
            line lx#, ly#, x+x#, y+Percent#
            lx# = x + x#
            ly# = y+ Percent#
        endif

    next i

next Process
print "Press any key to exit"
make sound from memblock 2, MemblockId+1
loop sound 2
wait key
end
```

```
//=============================================================
// Noise effect
function AddNoiseToSample( Sample, BPS, ModulationStep#, Frequency# )

    if BPS = 1
       if Sample > 32 and Sample < 128 then Sample = Sample + ( sin( ModulationStep# ) * Frequency# )
       if Sample < 0 then Sample = 0
       if Sample > 255 then Sample = 255
    else
       if Sample > 4096 and Sample < 16384 then Sample = Sample + ( sin( ModulationStep# ) * Frequency# )
       if Sample < 0 then Sample = 0
       if Sample > 32767 then Sample = 32767
    endif

endfunction Sample
```

Make Memblock From Mesh

Creates
a new



memblock which consists of 3D object properties obtained from a mesh. The data can be manipulated and saved back into a mesh, which inturn can be appended to an existing 3D object, added as a new limb in a 3D object, used to replace an existing limb in a 3D object, added into vertexdata or saved as a direct X file.

make memblock from mesh memblock ID *int*, mesh ID *int*

Support

DBPRO and GDK

Technical Explanation

3D models are formed with triangles, which we often call polygons. Each triangle has one vertex on each corner. The vertices contain information about their position; and often their colour as well as other appearance properties which determine how they are rendered via the Direct3D pipeline.

The information in the mesh memblock is based on a given Flexible Vertex Format (FVF); which is much like a document with a variable amount of detail depending on the format used. This provides you with the ability to choose how much detail you wish to specify about your 3D models and allows you to design your own vertex format.

Converting a mesh into a memblock is often used for defining your own file format and extracting it into an object, or saving it into a file. This method is not the easiest way to manipulate an object, there are easier ways of manipulating objects using vertexdata commands and the object creation commands in matrix1 utils.

The first DWORD at position 0 in the memblock stores the FVF format, which is usually 274 or 338. The size of each FVF vertex is in the second DWORD at position 4, this will be based on how much information is available per vertex due to the given FVF format. The third DWORD is at position 8 and stores the number of vertices in your mesh. From position 12 onwards lies the vertex data, and every 3 vertices produces a triangle polygon.

By iterating through the vertices, the first set of 3 floating point values (4 bytes each) are the X, Y, and Z positions of the vertex. In format 338, the second 3 float values of the vertex element would be the vertex normals, which describe how the vertex is shaded in relation to light sources. The normals can be set yourself or they can be calculated using the [set object normals] command once an extracted mesh has been converted into an object.

The next FVF 338 vertex data is a DWORD that contains the diffuse colour of the vertex, this is missing from FVF 274.

The last two FLOATS of FVF 338 are UV texture coordinates for the vertex; when U is 1.0, the vertex texture X coordinate is the width of that texture; when V is 1.0, the vertex texture Y coordinate is the height of the texture. A value of 0.5 is half the length of the texture, and 0.0 is texture coordinate 0. It is this information that is manipulated by the [scroll object texture] command; note that the values are repeated, therefore position 1.5 in the texture is half way through it; thus setting the UV coordinates of vertices on the left most edge of a triangle to position 1.0 will render the same texture coordinates as position 0.0. (This feature is often used to create animated scrolling textures.)

The bytes used by a FVF 338 vertex is 36. Multiply 36 by the number of vertices in the mesh and you get the overall size of the mesh data. FVF 274 contains 32 bytes per vertex due to not containing a diffuse colour DWORD in the vertex entry. Adding the vertex index to position 12 gives you the memblock position of its data.

```
rem Output the first two vertices in a cube

rem Create a cube and convert it into a memblock
make object cube 1, 1
scroll object texture 1, 0.3, 0.3
make mesh from object 1, 1
make memblock from mesh 1, 1

rem Get the size of each vertex in bytes
FVFSize = memblock dword(1, 4)

do

    i = 0
    for p = 12 to 12+(FVFSize*2) step FVFSize

        text 0,i, "First two vertices of the cube:" : inc i, 25
        x# = memblock float(1, p)
        y# = memblock float(1, p+4)
        z# = memblock float(1, p+8)
        text 0,i, "Position: " + str$(x#) + ", " + str$(y#) + ", " + str$(z#) : inc i, 25

        nx# = memblock float(1, p+12)
        ny# = memblock float(1, p+16)
        nz# = memblock float(1, p+20)
        text 0,i, "Normals: " + str$(nx#) + ", " + str$(ny#) + ", " + str$(nz#): inc i, 25

        u# = memblock float(1, p+24)
        v# = memblock float(1, p+28)
        text 0,i, "Normals: " + str$(u#) + ", " + str$(v#): inc i, 25

    next p

loop
```