Kick Assembler Macros

Every now and then, someone (mostly Slammer), posts a nice Kick Assembler macro on CSDb. This page is intended to collect such tips and tricks that relate specifically to Kick Assembler. Please add your own macros here!

To call a Macro in Kick Assembler, just write something like:

```
:MacroName(Parameter1, Parameter2, Parameter3)
```

Graphics conversion

"Char"Pack straight from .PNG file (Bitmap Format)

By: TWW/CTR

This macro takes a .PNG file and converts it into a 2×2 tile based bitmap(koala) format.

You need to set the **Palette and the background color** inside the macro (make sure these are correct) and then call the macro with the filname and output location of the data in memory as follows:

```
    :BitmapPack("map.png", $8000)
```

Here follows the Macro:

```
    .macro BitmapPack (FileName,Destination) {

        .function RGB_Sum(argument1,argument2,argument3) {     // Function
to calculate RGB colors
            .return argument1 * 65536 + argument2 * 256 + argument3
        }

        .var RGB_Index = Hashtable().put(                      // Define the
palette
            RGB_Sum($00,$00,$00), 0,   // Black
            RGB_Sum($ff,$ff,$ff), 1,   // White
            RGB_Sum($92,$4a,$40), 2,   // Red
            RGB_Sum($00,$00,$01), 3,   // Cyan
            RGB_Sum($00,$00,$01), 4,   // Purple
            RGB_Sum($72,$b1,$4b), 5,   // Green
            RGB_Sum($48,$3a,$aa), 6,   // Blue
            RGB_Sum($00,$00,$01), 7,   // Yellow
            RGB_Sum($00,$00,$01), 8,   // Light Brown
            RGB_Sum($00,$00,$01), 9,   // Dark Brown
            RGB_Sum($00,$00,$01), 10,  // Light Red
            RGB_Sum($00,$00,$01), 11,  // Dark Grey
```

```
            RGB_Sum($8a,$8a,$8a), 12,  // Grey
            RGB_Sum($00,$00,$01), 13,  // Light Green
            RGB_Sum($00,$00,$01), 14,  // Light Blue
            RGB_Sum($00,$00,$01), 15   // Light Grey
        )

        .const BG_Color        = 0                              // User
selected Background Color
        .const BackgroundColor = RGB_Index.get(BG_Color)        // Set the
background RGB color
        .const BlockSizeX      = 16                             // X Size of
each block in Single Color Pixels
        .const BlockSizeY      = 16                             // Y Size of
each block in Single Color Pixels
        .const RawImage        = LoadPicture(FileName,List())   // Loads the
map into the list RawImage
        .const ImageBlocksX    = RawImage.width/BlockSizeX      // Sets the
map size in X
        .const ImageBlocksY    = RawImage.height/BlockSizeY     // Sets the
map size in Y

        .var void              = 0                              // Empty
value used to call funtions without any return
        .var PixelPosX         = 0                              // Global
Block Pointer X
        .var PixelPosY         = 0                              // Global
Block Pointer Y
        .var BlockNumber       = 0                              // Keeps
track of numbers for new blocks
        .var FinalBMP          = List()
        .var Final0400         = List()
        .var FinalD800         = List()
        .var FinalMap          = List()
        .var DiscoveredBlocks = Hashtable()                     // Keeps
track of earlier discovered blocks

        .for (n = 0 ; n < ImageBlocksY ; n++) {
            .for (j = 0 ; j < ImageBlocksX ; j++) {
                .var BlockDataBMP     = List()
                .var BlockData0400    = List()
                .var BlockDataD800    = List()
                .var BlockStructure   = List()
                .for (i = 0 ; i < 4 ; i++) {          // Start to decode 1
charblock
                    .var CharColors       = Hashtable()
                    .var CharColors2      = Hashtable()
                    .eval CharColors      .put(BackgroundColor, 0)
                    .eval CharColors2     .put(0, BackgroundColor)
                    .for (y = 0 ; y < 8 ; y++) {
                        .var BitMask = 0
                        .for (x = 0 ; x < 4 ; x++) {
```

```
                            .var RGB_Value =
RawImage.getPixel(PixelPosX+x*2,PixelPosY+y)
                            .if
([RGB_Value!=BackgroundColor]&&[CharColors.containsKey(RGB_Value) != true])
{
                                .eval
CharColors.put(RGB_Value,CharColors.keys().size())
                                .eval
CharColors2.put(CharColors2.keys().size(),RGB_Value)
                            }
                            .if(CharColors.get(RGB_Value) == 1) .eval
BitMask = BitMask | %10000000>>x*2
                            .if(CharColors.get(RGB_Value) == 2) .eval
BitMask = BitMask | %01000000>>x*2
                            .if(CharColors.get(RGB_Value) == 3) .eval
BitMask = BitMask | %11000000>>x*2
                        }
                        .eval BlockDataBMP.add(BitMask)
                    }
                    .if (CharColors.keys().size() == 1) {
                        .eval BlockData0400.add(BG_Color)
                        .eval BlockDataD800.add(BG_Color|BG_Color<<4)
                    }
                    .if (CharColors.keys().size() == 2) {
                        .eval
BlockData0400.add(RGB_Index.get(CharColors2.get(1))|BG_Color<<4)
                        .eval BlockDataD800.add(BG_Color)
                    }
                    .if (CharColors.keys().size() == 3) {
                        .eval
BlockData0400.add(RGB_Index.get(CharColors2.get(1))|RGB_Index.get(CharColors
2.get(2))<<4)
                        .eval BlockDataD800.add(BG_Color)
                    }
                    .if (CharColors.keys().size() == 4) {
                        .eval
BlockData0400.add(RGB_Index.get(CharColors2.get(1))|RGB_Index.get(CharColors
2.get(2))<<4)
                        .eval
BlockDataD800.add(RGB_Index.get(CharColors2.get(3)))
                    }
                    .if (i==0) { .eval PixelPosX = PixelPosX + 8 }
                    .if (i==1) { .eval PixelPosX = PixelPosX - 8 .eval
PixelPosY = PixelPosY + 8 }
                    .if (i==2) { .eval PixelPosX = PixelPosX + 8 }
                    .if (i==3) { .eval PixelPosX = PixelPosX + 8 .eval
PixelPosY = PixelPosY - 8 }
                }
                .eval
BlockStructure.add(BlockDataBMP,BlockData0400,BlockDataD800)
                .var DoesTheBlockExists =
```

```
DiscoveredBlocks.get(BlockStructure)
                .if (DoesTheBlockExists == null) {
                    .eval DiscoveredBlocks.put(BlockStructure,BlockNumber)
                    .eval DoesTheBlockExists = BlockNumber
                    .eval BlockNumber = BlockNumber + 1
                    .eval FinalBMP.addAll(BlockDataBMP)
                    .eval Final0400.addAll(BlockData0400)
                    .eval FinalD800.addAll(BlockDataD800)
                }
                .eval FinalMap.add(DoesTheBlockExists)
            }
            .eval PixelPosX = 0
            .eval PixelPosY = PixelPosY + 16
        }
        .pc = Destination "Level 1 - Bitmap Tile Graphics"
        .fill FinalBMP.size(),FinalBMP.get(i)
        .pc = * "Level 1 - Tile Screen Colors"
        .fill Final0400.size(),Final0400.get(i)
        .pc = * "Level 1 - Tile $d800 Colors"
        .fill FinalD800.size(),FinalD800.get(i)
        .pc = * "Level 1 - Map"
        .fill FinalMap.size(),FinalMap.get(i)
    }  // end macro
```

Load Multicolor Koala format straight from .png file

The routine is called as follows:

```
    :PNGtoKOALA("320x200.png", $2000, $5000, $6000, $7000)
```

First, the palette needs to defined as follows (One could consolidate this into one list like done below in the hires routine):

```
    // VICE C64 PALETTE
    .struct RGB {r,g,b}  // Based on standard vice colors
    .const black   = RGB(  0,  0,  0) // #$000000
    .const white   = RGB(255,255,255) // #$FFFFFF
    .const red     = RGB(137, 64, 54) // #$894036
    .const cyan    = RGB(122,191,199) // #$7ABFC7
    .const purple  = RGB(138, 70,174) // #$8A46AE
    .const green   = RGB(104,169, 65) // #$68A941
    .const blue    = RGB( 62, 49,162) // #$3E31A2
    .const yellow  = RGB(208,220,113) // #$D0DC71
    .const l_brown = RGB(144, 95, 37) // #$905F25
    .const d_brown = RGB( 92, 71,  0) // #$5C4700
    .const l_red   = RGB(187,119,109) // #$BB776D
    .const d_grey  = RGB( 85, 85, 85) // #$555555
    .const grey    = RGB(128,128,128) // #$808080
    .const l_green = RGB(172,234,136) // #$ACEA88
```

```
    .const l_blue  = RGB(124,112,218) // #$7C70DA
    .const l_grey  = RGB(171,171,171) // #$ABABAB

    .const Black   =   black.r * 65536 +   black.g * 256 +   black.b
    .const White   =   white.r * 65536 +   white.g * 256 +   white.b
    .const Red     =     red.r * 65536 +     red.g * 256 +     red.b
    .const Cyan    =    cyan.r * 65536 +    cyan.g * 256 +    cyan.b
    .const Purple  =  purple.r * 65536 +  purple.g * 256 +  purple.b
    .const Green   =   green.r * 65536 +   green.g * 256 +   green.b
    .const Blue    =    blue.r * 65536 +    blue.g * 256 +    blue.b
    .const Yellow  =  yellow.r * 65536 +  yellow.g * 256 +  yellow.b
    .const L_brown = l_brown.r * 65536 + l_brown.g * 256 + l_brown.b
    .const D_brown = d_brown.r * 65536 + d_brown.g * 256 + d_brown.b
    .const L_red   =   l_red.r * 65536 +   l_red.g * 256 +   l_red.b
    .const D_grey  =  d_grey.r * 65536 +  d_grey.g * 256 +  d_grey.b
    .const Grey    =    grey.r * 65536 +    grey.g * 256 +    grey.b
    .const L_green = l_green.r * 65536 + l_green.g * 256 + l_green.b
    .const L_blue  =  l_blue.r * 65536 +  l_blue.g * 256 +  l_blue.b
    .const L_grey  =  l_grey.g * 65536 +  l_grey.g * 256 +  l_grey.b
```

Note that You have to put in the correct values here as this only serves as example. Putting wrong RGB collors will make the code below fail! If You don't like this approach, there is a routine below by Pantaloon to determine the collors or You could add line 201 to the image and read out the values directly from the 0 to 15th pixel. In short, feel free to modify.

This following snippet converts a 320×200 .png file directly to koala format and you can specify where you want the data to end up. The routine will automatically detect the Background Color and set the bitmap patterns accordingly. Made by TWW/Creators.

```
    .macro PNGtoKOALA(PNGpicture,BMPData,Chardata,D800data,BGC) {

        // Create RGB to C64 color index
        .var RGBtoC64 = Hashtable()
        .eval RGBtoC64.put(Black,0)
        .eval RGBtoC64.put(White,1)
        .eval RGBtoC64.put(Red,2)
        .eval RGBtoC64.put(Cyan,3)
        .eval RGBtoC64.put(Purple,4)
        .eval RGBtoC64.put(Green,5)
        .eval RGBtoC64.put(Blue,6)
        .eval RGBtoC64.put(Yellow,7)
        .eval RGBtoC64.put(L_brown,8)
        .eval RGBtoC64.put(D_brown,9)
        .eval RGBtoC64.put(L_red,10)
        .eval RGBtoC64.put(D_grey,11)
        .eval RGBtoC64.put(Grey,12)
        .eval RGBtoC64.put(L_green,13)
        .eval RGBtoC64.put(L_blue,14)
        .eval RGBtoC64.put(L_grey,15)

        // Hashtable for Storing all the colors.
```

```
        .var RGBColorTable  = Hashtable()

        // Create a list to hold all the Bitmap and collor data
        .var AllData = List(10000)

        // Load the picture into the data list Graphics
        .var Graphics = LoadPicture(PNGpicture,List())

        // Convert and return the Background Color
        .var BackgroundColor = FindBackgroundColor()

        .pc = BMPData "KOALA - Bitmap Graphics"
        .fill 8000,AllData.get(i)
        .pc = Chardata "KOALA - Character Color Data"
        .fill 1000,AllData.get(i+8000)
        .pc = D800data "KOALA - D800 Color Data"
        .fill 1000,AllData.get(i+9000)
        .pc = BGC "KOALA - Background Collor"
        .fill 1,BackgroundColor

    }

//=============================================================================
===
    .function FindBackgroundColor() {

        // Hastable for storing 4x8 block colordata
        .var BlockColors = Hashtable()

        // Hashtable for potential background Colors from inside one block
        .var BG = Hashtable()

        // List for keeping track of background color candidates from all
blocks
        .var BGCandidate  = List(4)

        // Declare some variables
        .var CurrentBlock = 0     // Keeps track of which block is being
checked
        .var BGRemaining  = 4     // Remaining backgound color candidates
(begins with 4 since the first block get's it's 4 colors copied into the
BGCandidate Hastable)
        .var ColorCounter = 0     // Counter for keeping track of how many
colors are found inside each block
        .var FirstMatch   = true  // Used to diferensiate between the first
4 color block (It has to contain the backgound color) and the rest of the
blocks

        // Loop for checking all 1000 blocks
        .for (CurrentBlock=0 ; CurrentBlock<1000 ; CurrentBlock++) {
```

```
            // Clear out any block colors from the hashtable
            .eval BlockColors = Hashtable()

            // Fetch 4x8 pixel block colors (32 total)
            .for (var Pixel = 0 ; Pixel < 32 ; Pixel++) {
                .var PixelColor =
Graphics.getPixel([8*CurrentBlock+[[Pixel<<1]&7]]-
[320*[floor(CurrentBlock/40)]], [8*floor(CurrentBlock/40)]+[Pixel>>2])
                .eval BlockColors.put(PixelColor,Pixel)
            }

            // Reset the block color counter
            .eval ColorCounter = 0

            // Store the block colors in BG
            .if (BlockColors.containsKey(Black)  ==true) { .eval
BG.put(ColorCounter,Black)   .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Black)   .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(White)  ==true) { .eval
BG.put(ColorCounter,White)   .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],White)   .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Red)    ==true) { .eval
BG.put(ColorCounter,Red)     .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Red)     .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Cyan)   ==true) { .eval
BG.put(ColorCounter,Cyan)    .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Cyan)    .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Purple) ==true) { .eval
BG.put(ColorCounter,Purple)  .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Purple)  .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Green)  ==true) { .eval
BG.put(ColorCounter,Green)   .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Green)   .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Blue)   ==true) { .eval
BG.put(ColorCounter,Blue)    .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Blue)    .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Yellow) ==true) { .eval
BG.put(ColorCounter,Yellow)  .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Yellow)  .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(L_brown)==true) { .eval
BG.put(ColorCounter,L_brown) .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],L_brown) .eval
ColorCounter=ColorCounter+1 }
```

```
            .if (BlockColors.containsKey(D_brown)==true) { .eval
BG.put(ColorCounter,D_brown) .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],D_brown) .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(L_red)  ==true) { .eval
BG.put(ColorCounter,L_red)   .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],L_red)   .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(D_grey) ==true) { .eval
BG.put(ColorCounter,D_grey)  .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],D_grey)  .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(Grey)   ==true) { .eval
BG.put(ColorCounter,Grey)    .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],Grey)    .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(L_green)==true) { .eval
BG.put(ColorCounter,L_green) .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],L_green) .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(L_blue) ==true) { .eval
BG.put(ColorCounter,L_blue)  .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],L_blue)  .eval
ColorCounter=ColorCounter+1 }
            .if (BlockColors.containsKey(L_grey) ==true) { .eval
BG.put(ColorCounter,L_grey)  .eval
RGBColorTable.put([ColorCounter+[4*CurrentBlock]],L_grey)  .eval
ColorCounter=ColorCounter+1 }

            // Carry out a background color check when there are 4 colors in
a block
            .if (ColorCounter == 4 && BGCandidate.size()>1) {

                // Check if it is the first block with 4 collors
                .if (FirstMatch) {

                    // Copy the 4 collors as possible candidates
                    .eval BGCandidate.add(BG.get(0))
                    .eval BGCandidate.add(BG.get(1))
                    .eval BGCandidate.add(BG.get(2))
                    .eval BGCandidate.add(BG.get(3))
                    .eval FirstMatch = false
                } else {
                    .for (var i = 0 ; i < BGCandidate.size() ; i++) {
                        .if (BGCandidate.get(i) != BG.get(0)) {
                            .if (BGCandidate.get(i) != BG.get(1)) {
                                .if (BGCandidate.get(i) != BG.get(2)) {
                                    .if (BGCandidate.get(i) != BG.get(3)) {
                                        .eval BGCandidate.remove(i)
                                    }
                                }
```

```
                            }
                        }
                    }
                }
            }
        }


        .var BackgroundColor = BGCandidate.get(0)

        // Variable for keeping track of which byte is in use
        .var ByteNumber = 0

        // Create hashtable and ascociate bitmap patterns to RGB Colors (one
for bit patterns and one for collor referance)
        .var ColorIndex = Hashtable()
        .var ColorIndex2 = Hashtable()

        // Define the BG Color into the Color Indexes
        .eval ColorIndex.put(BackgroundColor,0)
        .eval ColorIndex2.put(0,BackgroundColor)

        .for (var BlockNumber = 0 ; BlockNumber < 1000 ; BlockNumber++) {

            // Variable for keeping track of which collor is to be used
inside the block
            .var colpos = 1

            // Place the RGB color data into the color indexes (Multicolor
Bit-combinations 01, 10 & 11 assigned the 3 colors)
            .for (var i = 0 ; i < 4 ; i++) {
                .if (RGBColorTable.get(i+[BlockNumber*4]) !=
BackgroundColor) {
                    .if (RGBColorTable.get(i+[BlockNumber*4]) != null) {
                        .eval
ColorIndex.put(RGBColorTable.get(i+[BlockNumber*4]),colpos)
                        .eval
ColorIndex2.put(colpos,RGBColorTable.get(i+[BlockNumber*4]))
                        .eval colpos = colpos+1
                    }
                }
            }

            // Read Pixel Collors in current block and fill in BMPData
accordingly
            .for (var Byte = 0 ; Byte < 8 ; Byte++) {

                // Temp Storage for bitmap byte, bitmap pattern and the
pixelcolor
                .var BMPByte = 0
                .var BMPPattern = 0
```

```
                .var PixelColor = 0

                // Find the pixel collors and cross ref. with the bit
patterns to create the BMP data
                .for (var Pixel = 0 ; Pixel < 4 ; Pixel++) {
                        .eval PixelColor =
Graphics.getPixel([[8*BlockNumber]+[[Pixel<<1]&7]]-
[320*[floor(BlockNumber/40)]], [8*floor(BlockNumber/40)]+Byte)
                        .eval BMPPattern = ColorIndex.get(PixelColor)
                        .eval BMPByte = BMPByte|[BMPPattern << [6 -
Pixel*2]]
                }

                // Set the done BMP data into final data storage
                .eval AllData.set(ByteNumber,BMPByte)
                .eval ByteNumber = ByteNumber+1
            }

            // Create the color data
            .var CharacterColor = 0
            .var D800Color = 0

            .if (RGBtoC64.get(ColorIndex2.get(1)) != null) { .eval
CharacterColor = [RGBtoC64.get(ColorIndex2.get(1))<<4] }
            .if (RGBtoC64.get(ColorIndex2.get(2)) != null) { .eval
CharacterColor = CharacterColor|RGBtoC64.get(ColorIndex2.get(2)) }
            .if (RGBtoC64.get(ColorIndex2.get(3)) != null) { .eval D800Color
= RGBtoC64.get(ColorIndex2.get(3)) }

            // Store the colors into final data storage
            .eval AllData.set(8000+BlockNumber,CharacterColor)
            .eval AllData.set(9000+BlockNumber,D800Color)
        }

    // Return background color:
    .return BackgroundColor

    }
```

Load SingleColor HiRes format straight from .png file

Macro to load those HiRES pictures with more than 2 colors into Kickass. by TWW/CTR

Read header for info. Make sure you update the RGB colors so they match your palete.

Sometimes the graphics can be inverted from 8×8 pixel block to the next but the result (in the color table) works anyway.

```
/*~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~~~
```

```
    PNGtoHIRES
    ~~~~~~~~~~

        By: TWW/CTR

    USAGE
    ~~~~~

        :PNGtoHIRES("Filename.png", BitmapMemoryAddress, ScreenMemoryColors)

        @SIGNATURE      void PNGtoHIRES (STR Filename.png ,U16
BitmapMemoryAddress, U16 ScreenMemoryColors)
        @AUTHOR         tww@creators.no

        @PARAM          Filename.png        - Filename & path to picture
file
        @PARAM          BitmapMemoryAddress - Memorylocation for output of
bmp-data
        @PARAM          ScreenMemoryColors  - Memorylocation for output of
Char-data


    EXAMPLES
    ~~~~~~~~

        :PNGtoHIRES("something.png", $2000, $2000+8000)


    NOTES
    ~~~~~

        For now, only handles 320x200


    IMPROVEMENTS
    ~~~~~~~~~~~~

        Add variable picture sizes
        Handle assertions if the format is unsupported (size, color
restrictions etc.)

    TODO
    ~~~~


    BUGS
    ~~~~


~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
~~*/
```

```
.macro PNGtoHIRES(PNGpicture,BMPData,ColData) {

    .var Graphics = LoadPicture(PNGpicture)

    // Graphics RGB Colors. Must be adapted to the graphics
    .const C64Black   = 000 * 65536 + 000 * 256 + 000
    .const C64White   = 255 * 65536 + 255 * 256 + 255
    .const C64Red     = 1 * 65536 + 0 * 256 + 0
    .const C64Cyan    = 1 * 65536 + 0 * 256 + 0
    .const C64Purple  = 1 * 65536 + 0 * 256 + 0
    .const C64Green   = 1 * 65536 + 0 * 256 + 0
    .const C64Blue    = 1 * 65536 + 0 * 256 + 0
    .const C64Yellow  = 1 * 65536 + 0 * 256 + 0
    .const C64L_brown = 111 * 65536 + 079 * 256 + 037
    .const C64D_brown = 067 * 65536 + 057 * 256 + 000
    .const C64L_red   = 1 * 65536 + 0 * 256 + 0
    .const C64D_grey  = 068 * 65536 + 068 * 256 + 068
    .const C64Grey    = 108 * 65536 + 108 * 256 + 108
    .const C64L_green = 1 * 65536 + 0 * 256 + 0
    .const C64L_blue  = 108 * 65536 + 094 * 256 + 181
    .const C64L_grey  = 149 * 65536 + 149 * 256 + 149

    // Add the colors neatly into a Hashtable for easy lookup reference
    .var ColorTable = Hashtable()
    .eval ColorTable.put(C64Black,0)
    .eval ColorTable.put(C64White,1)
    .eval ColorTable.put(C64Red,2)
    .eval ColorTable.put(C64Cyan,3)
    .eval ColorTable.put(C64Purple,4)
    .eval ColorTable.put(C64Green,5)
    .eval ColorTable.put(C64Blue,6)
    .eval ColorTable.put(C64Yellow,7)
    .eval ColorTable.put(C64L_brown,8)
    .eval ColorTable.put(C64D_brown,9)
    .eval ColorTable.put(C64L_red,10)
    .eval ColorTable.put(C64D_grey,11)
    .eval ColorTable.put(C64Grey,12)
    .eval ColorTable.put(C64L_green,13)
    .eval ColorTable.put(C64L_blue,14)
    .eval ColorTable.put(C64L_grey,15)

    .pc = BMPData "Hires Bitmap"

    .var ScreenMem = List()
    .for (var Line = 0 ; Line < 200 ; Line = Line + 8) {
        .for (var Block = 0 ; Block < 320 ; Block=Block+8) {
            .var Coll1 = Graphics.getPixel(Block,Line)
            .var Coll2 = 0
            .for (var j = 0 ; j < 8 ; j++ ) {
                .var ByteValue = 0
                .for (var i = 0 ; i < 8 ; i++ ) {
```

```
                        .if (Graphics.getPixel(Block,Line) !=
Graphics.getPixel(Block+i,Line+j)) .eval ByteValue = ByteValue + pow(2,7-i)
                        .if (Graphics.getPixel(Block,Line) !=
Graphics.getPixel(Block+i,Line+j)) .eval Coll2 =
Graphics.getPixel(Block+i,Line+j)
                    }
                .byte ByteValue
                }
            .var BlockColor =
[ColorTable.get(Coll2)]*16+ColorTable.get(Coll1)
            .eval ScreenMem.add(BlockColor)
            }
        }
        .pc = ColData "Hires Color Data"
    ScreenMemColors:
        .for (var i = 0 ; i < 1000 ; i++ ) {
            .byte ScreenMem.get(i)
        }
    }
```

Load sprite data straight from .gif file

First example assumes that each sprite is saved in a separate .gif file:

```
//Example by Slammer
.pc = $3000 "Sprite Data"
spriteData:

    :LoadSpriteFromPicture("sprite1.gif")
    :LoadSpriteFromPicture("sprite2.gif")
    :LoadSpriteFromPicture("sprite3.gif")
// etc

.macro LoadSpriteFromPicture(filename) {
    .var picture = LoadPicture(filename, List().add($000000,
$ffffff,$6c6c6c,$959595))
    .for (var y=0; y<21; y++)
        .for (var x=0; x<3; x++)
            .byte picture.getMulticolorByte(x,y)
    .byte 0
}
```

Second example is an adaption of the above script, which loads a single picture with 8 sprites organized horizontally:

```
//Slammer's example, but adapted by Cruzer
.var spriteData = $3000
.pc = spriteData "spriteData"
.var spritePic = LoadPicture("sprites.png",
```

```
List().add($000000,$ffffff,$6c6c6c,$959595))
.for (var i=0; i<8; i++)
    :getSprite(spritePic, i)

.macro getSprite(spritePic, spriteNo) {
    .for (var y=0; y<21; y++)
        .for (var x=0; x<3; x++)
            .byte spritePic.getMulticolorByte(x + spriteNo * 3, y)
    .byte 0
}
```

The values in the second argument of the LoadPicture function is the rgb values of the colors asigned to the four bit combinations (00, 01, 10 and 11). To find which colors are used in your picture you can use a paint program. Another posibility is to use the script below which will print the colors used in the picture 'MyLogo.gif'.

```
.var logo = LoadPicture("MyLogo.gif")

.var colors = Hashtable()
.for (var x=0; x<logo.width; x++)
    .for (var y=0; y<logo.height; y++)
        .eval colors.put(logo.getPixel(x,y),logo.getPixel(x,y))

.var colorSet = colors.keys()
.for (var i=0; i<colorSet.size(); i++)
    .print "$"+toHexString(colors.get(colorSet.get(i)))
```

Convert picture into a charset

This is a so called “equal char packer” that uses a hash table to detect repeated chars. It was originally written by Cruzer but has been updated/extended by others. It currently packs hires pics of x/y sizes that are multiples of 8 (to fit evenly into chars), but it should be easy to change it to multicolor...

```
.macro equalCharPack(filename, screenAdr, charsetAdr) {
    .var charMap = Hashtable()
    .var charNo = 0
    .var screenData = List()
    .var charsetData = List()
    .var pic = LoadPicture(filename)

    // Graphics should fit in 8x8 Single collor / 4 x 8 Multi collor blocks
    .var PictureSizeX = pic.width/8
    .var PictureSizeY = pic.height/8

    .for (var charY=0; charY<PictureSizeY; charY++) {
        .for (var charX=0; charX<PictureSizeX; charX++) {
            .var currentCharBytes = List()
            .var key = ""
```

```
            .for (var i=0; i<8; i++) {
                .var byteVal = pic.getSinglecolorByte(charX, charY*8 + i)
                .eval key = key + toHexString(byteVal) + ","
                .eval currentCharBytes.add(byteVal)
            }
            .var currentChar = charMap.get(key)
            .if (currentChar == null) {
                .eval currentChar = charNo
                .eval charMap.put(key, charNo)
                .eval charNo++
                .for (var i=0; i<8; i++) {
                    .eval charsetData.add(currentCharBytes.get(i))
                }
            }
            .eval screenData.add(currentChar)
        }
    }
    .pc = screenAdr "screen"
    .fill screenData.size(), screenData.get(i)
    .pc = charsetAdr "charset"
    .fill charsetData.size(), charsetData.get(i)
}
```

It is used simply like this:

```
:equalCharPack("pic.png", $2800, $2000)
```

Match RGB colors with the C64 palette

By Pantaloon/FLT.

Here is some code to find the closest c-64 color index from an RGB value, useful when doing image conversion. There are of course better ways to find the best match but this works ok if you have images with c-64 colors already. I use it for my kickasm image converters.

```
    .struct RGB {r,g,b}
    .var s_palette = List().add(
            RGB(0,0,0),        // black 0
            RGB(255,255,255),    // white 1
            RGB(104,55,43),        // red 2
            RGB(131,240,220),    // cyan 3
            RGB(111,61,134),    // purple 4
            RGB(89,205,54),        // green 5
            RGB(65,55,205),        // blue 6
            RGB(184,199,111),    // yellow 7
            RGB(209,127,48),    // orange 8
            RGB(67,57,0),        // brown 9
            RGB(154,103,89),    // light_red 10
            RGB(91,91,91),        // dark_gray 11
            RGB(142,142,142),    // gray 12
```

```
            RGB(157,255,157),    // light_green 13
            RGB(117,161,236),    // light_blue 14
            RGB(193,193,193)    // light_gray 15
            );
    .function colorDistance(c1,c2)
    {
        .var cr = c1.r-c2.r
        .var cg = c1.g-c2.g
        .var cb = c1.b-c2.b
        .return sqrt([cr*cr] + [cg*cg] + [cb*cb])
    }

    .function getClosestColorIndex(rgb)
    {
        .return getClosestColorIndex(
            rgb, s_palette
            )
    }

    .function getClosestColorIndex(rgb, palette)
    {
        .var distance = colorDistance(rgb, palette.get(0))
        .var closestColorIndex = 0

        .for (var index = 1; index < palette.size(); index++)
        {
            .var d = colorDistance(rgb, palette.get(index))
            .if (d < distance)
            {
                .eval distance = d
                .eval closestColorIndex = index
            }
        }

        .return closestColorIndex
    }
```

Pseudo Commands

Pseudo commands are macros that take assembler arguments. This means they know the difference between #00, $10 and ($10),y. This section will show some examples of how to create pseudo commands.

Repetition Commands

Every body who have programmed c64 assembler language have tried to repeat the same command several time. If you want to divide by 8 you type 3 lsr commands or if you want to create a pause you can do alot of nop's. To save alot of typing, you can create repetitive pseudo commands.

```
:lsr #3  // divide by 8
:nop #8  // Do 8 nops
```

The commands (+ some extra) are defined like this:

```
//---------------------------------
// repetition commands
//---------------------------------
.macro ensureImmediateArgument(arg) {
    .if (arg.getType()!=AT_IMMEDIATE)    .error "The argument must be
immediate!"
}
.pseudocommand asl x {
    :ensureImmediateArgument(x)
    .for (var i=0; i<x.getValue(); i++) asl
}
.pseudocommand lsr x {
    :ensureImmediateArgument(x)
    .for (var i=0; i<x.getValue(); i++) lsr
}
.pseudocommand rol x {
    :ensureImmediateArgument(x)
    .for (var i=0; i<x.getValue(); i++) rol
}
.pseudocommand ror x {
    :ensureImmediateArgument(x)
    .for (var i=0; i<x.getValue(); i++) ror
}

.pseudocommand pla x {
    :ensureImmediateArgument(x)
    .for (var i=0; i<x.getValue(); i++) pla
}

.pseudocommand nop x {
    :ensureImmediateArgument(x)
    .for (var i=0; i<x.getValue(); i++) nop
}
```

Creating a Pause Command

Often you want to time your code precisely. To make things a little easier you can create a pause command that pause a given amount of cycles. Eg:

```
:pause #10  // Waits 10 cycles
:pause2 #63 // Waits 63 cycles
```

Here you will see two versions of the pause command. The first one only uses bit and nops. Both are based on some of the pseudo commands defined in earlier sections:

```
.pseudocommand pause cycles {
    :ensureImmediateArgument(cycles)
    .var x = floor(cycles.getValue())
    .if (x<2) .error "Cant make a pause on " + x + " cycles"

    // Take care of odd cyclecount
    .if ([x&1]==1) {
        bit $00
        .eval x=x-3
    }
    // Take care of the rest
    .if (x>0)
        :nop #x/2
}
```

The second one uses a loop. This makes it take less memory, however the timing will not be correct if the conditional jump crosses a page boundary:

```
.pseudocommand pause2 cycles {
    :ensureImmediateArgument(cycles)
    .var x = floor(cycles.getValue())
    .if (x<2) .error "Cant make a pause on " + x + " cycles"

    // Make a delay loop
    .if (x>=11) {
        .const cfirst = 6    // cycles for first loop
        .const cextra = 5    // cycles for extra loops
        .var noOfLoops = 1+floor([x-cfirst]/cextra)
        .eval x = x - cfirst - [noOfLoops-1]*cextra
        .if (x==1){
            .eval x=x+cextra
            .eval noOfLoops--
        }
        ldy #noOfLoops
        dey
        bne *-1
    }

    // Take care of odd cyclecount
    .if ([x&1]==1) {
        bit $00
        .eval x=x-3
    }
    // Take care of the rest
    .if (x>0)
        :nop #x/2
}
```

Pseudocommands for 16 bit operations

The following pseudocommands uses the function _16bit_nextArgument(arg), and it must be declared previously:

```
.function _16bit_nextArgument(arg) {
    .if (arg.getType()==AT_IMMEDIATE)
    .return CmdArgument(arg.getType(),>arg.getValue())
    .return CmdArgument(arg.getType(),arg.getValue()+1)
}
```

Increment a 16bit variable:

```
.pseudocommand inc16 arg {
        inc arg
            bne over
        inc _16bit_nextArgument(arg)
over:
}
```

Decrement a 16 bit variable:

```
.pseudocommand dec16 arg {
        lda arg
                bne skip
                dec _16bit_nextArgument(arg)
skip:       dec arg
}
```

Move or Load 16 bit values from/to 16 bit variables:

```
.pseudocommand mov16 src;tar {
        lda src
        sta tar
        lda _16bit_nextArgument(src)
        sta _16bit_nextArgument(tar)
}
```

Addition of 16 bits variables:

```
.pseudocommand add16 arg1 ; arg2 ; tar {
.if (tar.getType()==AT_NONE) .eval tar=arg1
        lda arg1
            adc arg2
            sta tar
            lda _16bit_nextArgument(arg1)
        adc _16bit_nextArgument(arg2)
        sta _16bit_nextArgument(tar)
}
```

Equal char packer

Cruzer made an equal char packer using a hash table. It packs a 320×200 hires pic, but it should be easy to change it to multicolor and extend the size...

```
.macro equalCharPack(filename, screenAdr, charsetAdr) {
    .var charMap = Hashtable()
    .var charNo = 0
    .var screenData = List()
    .var charsetData = List()
    .var pic = LoadPicture(filename)
    .for (var charY=0; charY<25; charY++) {
        .for (var charX=0; charX<40; charX++) {
            .var currentCharBytes = List()
            .var key = ""
            .for (var i=0; i<8; i++) {
                .var byteVal = pic.getSinglecolorByte(charX, charY*8 + i)
                .eval key = key + toHexString(byteVal) + ","
                .eval currentCharBytes.add(byteVal)
            }
            .var currentChar = charMap.get(key)
            .if (currentChar == null) {
                .eval currentChar = charNo
                .eval charMap.put(key, charNo)
                .eval charNo++
                .for (var i=0; i<8; i++) {
                    .eval charsetData.add(currentCharBytes.get(i))
                }
            }
            .eval screenData.add(currentChar)
        }
    }
    .pc = screenAdr "screen"
    .fill screenData.size(), screenData.get(i)
    .pc = charsetAdr "charset"
    .fill charsetData.size(), charsetData.get(i)
}

:equalCharPack("pic.png", $2800, $2000)
```

Set Screen and Char Location

```
.macro SetScreenAndCharLocation(screen, charset) {
    lda    #[[screen & $3FFF] / 64] | [[charset & $3FFF] / 1024]
    sta    $D018
}
```

Clear Screen

```
.macro ClearScreen(screen, clearByte) {
    lda #clearByte
    ldx #0
!loop:
    sta screen, x
    sta screen + $100, x
    sta screen + $200, x
    sta screen + $300, x
    inx
    bne !loop-
}
```

Clear Color RAM

```
.macro ClearColorRam(clearByte) {
    lda #clearByte
    ldx #0
!loop:
    sta $D800, x
    sta $D800 + $100, x
    sta $D800 + $200, x
    sta $D800 + $300, x
    inx
    bne !loop-
}
```

Set Various VIC Banks

```
// $DD00 = %xxxxxx11 -> bank0: $0000-$3fff
// $DD00 = %xxxxxx10 -> bank1: $4000-$7fff
// $DD00 = %xxxxxx01 -> bank2: $8000-$bfff
// $DD00 = %xxxxxx00 -> bank3: $c000-$ffff
.macro SetVICBank0() {
    lda $DD00
    and #%11111100
    ora #%00000011
    sta $DD00
}

.macro SetVICBank1() {
    lda $DD00
    and #%11111100
    ora #%00000010
    sta $DD00
```

```
}

.macro SetVICBank2() {
    lda $DD00
    and #%11111100
    ora #%00000001
    sta $DD00
}

.macro SetVICBank3() {
    lda $DD00
    and #%11111100
    ora #%00000000
    sta $DD00
}
```

Set Color Modes, Colors and Scroll

```
.macro SetBorderColor(color) {
    lda #color
    sta $d020
}

.macro SetBackgroundColor(color) {
    lda #color
    sta $d021
}

.macro SetMultiColor1(color) {
    lda #color
    sta $d022
}

.macro SetMultiColor2(color) {
    lda #color
    sta $d023
}

.macro SetMultiColorMode() {
    lda    $d016
    ora    #16
    sta    $d016
}

.macro SetScrollMode() {
    lda $D016
    eor #%00001000
    sta $D016
```

```
}
```