Topic 3: (Part B)

I/O and Peripheral Devices Management

GRAPHICS in the Nintendo DS

Systèmes Embarqués Microprogrammés

Prof. David Atienza Alonso, SEL-STI

Content of Session

- Use of one background in NDS: extended rotoscale mode
  - NDS video modes for rotoscale mode
  - Palettes and backgrounds
  - Data storage in VRAM structure for rotoscale mode
  - Data transfer functions
  - Transformation matrix

- Drawing graphics in extended rotoscale mode
  - Examples of transformation matrix in the NDS
  - Converting images to bitmaps: grit tool

Configuration Sequence of Rotoscale Mode

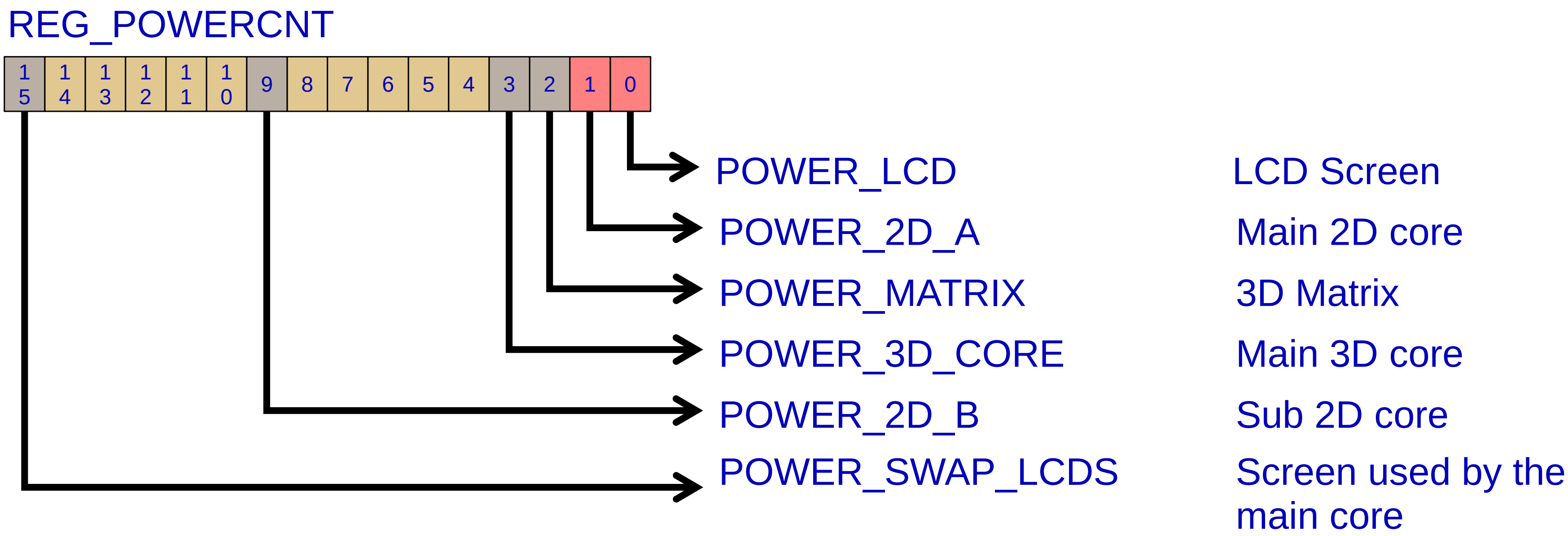
1. Power Manager configuration (REG_POWERCNT)
   - Performed with default settings in system boot-up
2. VRAM configuration (VRAM_x_CR)
   - Activate banks and configure them (depending on used graphical engines)
3. Graphical engines configuration (REG_DISPCNT)
   - Configure mode and backgrounds
4. Configure each active background (BGCTRL[n])
   - Set Bitmap base address, background size and pixel configuration (8 or 16 bits color mode).
5. Initialize palettes when using 8 bits mode (BG_PALETTE[0…255])

6. Optional: Adjust affine transformation matrix for each background

Ready to filling the bitmap with your graphics!

Revision: Graphics subsystem in the NDS
Activating screens and graphic coprocessors

- Control register for powering up I/O NDS devices: REG_POWERCNT
  - It is mapped on the memory address: 0x4000304



- LCD and engines activated by default during the boot-up process

- Possible to enable / disable them manually to save power at run-time

Activation: **REG_POWERCNT = POWER_LCD | POWER_2D_A;**

Deactivation: **REG_POWERCNT &= ~(POWER_LCD) & ~(POWER_2D_A);**

NDS screen modes: 2D engines configuration Choosing extended rotoscale mode

- Each engine has four backgrounds (or layers): BG0, BG1, BG2 and BG3
  - Final view on the screen will be their combination based on used graphic mode
  - Typically BG0 is the most external one and BG3 is the most internal one

- Multiple modes possible in 2D engines, select the most appropriate one
  - Main engine: 7 modes and framebuffer

Mode	BG0	BG1	BG2	BG3
0	Tiled/3D	Tiled	Tiled	Tiled
1	Tiled/3D	Tiled	Tiled	Rotoscale
2	Tiled/3D	Tiled	Rotoscale	Rotoscale
3	Tiled/3D	Tiled	Tiled	Ext. Rotoscale
4	Tiled/3D	Tiled	Rotoscale	Ext. Rotoscale
5	Tiled/3D	Tiled	Ext. Rotoscale	Ext. Rotoscale
6	3D	N/A	Large Bitmap	N/A
FrameBuf.	Direct VRAM display as a bitmap			

  - Sub engine: 6 modes

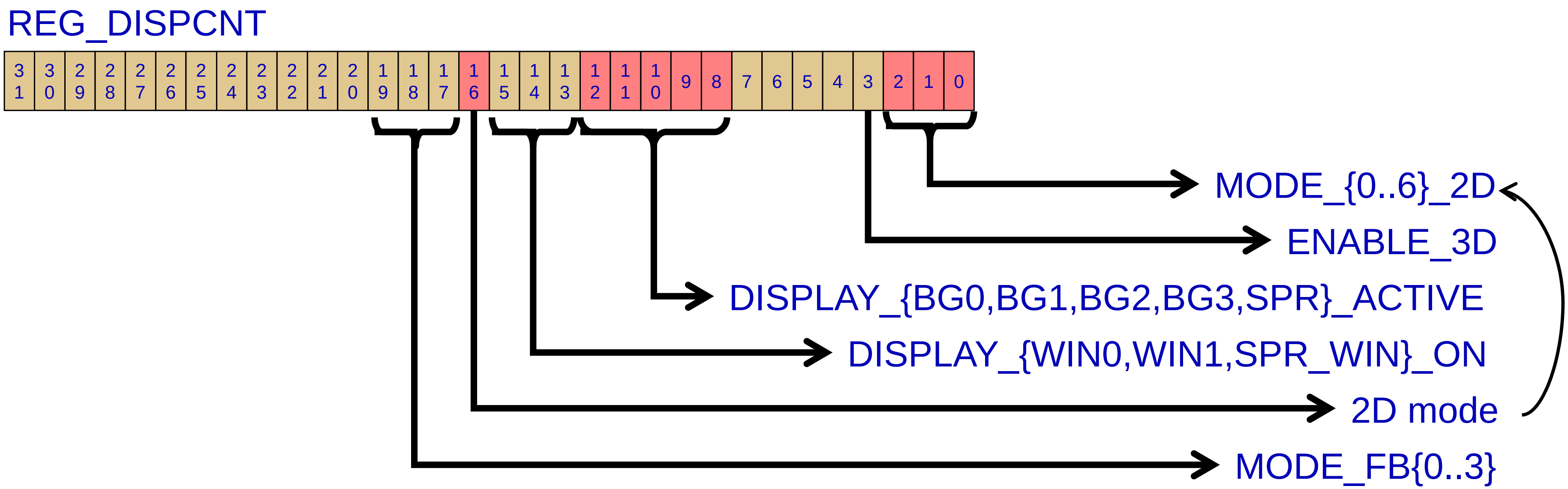
Mode	BG0	BG1	BG2	BG3
0	Tiled	Tiled	Tiled	Tiled
1	Tiled	Tiled	Tiled	Rotoscale
2	Tiled	Tiled	Rotoscale	Rotoscale
3	Tiled	Tiled	Tiled	Ext. Rotoscale
4	Tiled	Tiled	Rotoscale	Ext. Rotoscale
5	Tiled	Tiled	Ext. Rotoscale	Ext. Rotoscale

- In the case of extended rotation or extended rotoscale mode?
  - Configure REG_DISPCNT with one of the possible modes: 3, 4 and 5 (Extended)
  - Example: activate mode 5 and background 2 (BG2)

```
REG_DISPCNT = MODE_5_2D | DISPLAY_BG2_ACTIVE;
```

Reminder: Graphics Engine Control Register

- REG_DISPCNT: display register to control mode and active backgrounds



- Example: activate mode 0 and background 1 (BG1)

  REG_DISPCNT = MODE_0_2D | DISPLAY_BG1_ACTIVE;

Configuring Memory-Mapped I/O Peripheral Registers

- LibNDS library under /opt/devkitPro defines the registers ID

  #define REG_DISPCNT *(vu32*)0x04000000

  ***(vu32*)0x04000000**

  Volatile 32 bit
  Unsigned integer

  0000 0000 0100 0000 0000 0000 0000 0000 0000

- <u>Hint</u>: Use the following command in the virtual image terminal to find any macro's value:

  $: grep –rnw '/opt/devkitPro/libnds/include/nds' –e 'MACRO_NAME'

Example: Enabling and Assigning VRAM Block A to Main Screen in Mode 0

- C language operation meaning:

Non- exclusive binary OR

$$\underbrace{\texttt{REG\_DISPCNT}}_{\texttt{vu32}} = \underbrace{\texttt{MODE\_0\_2D}}_{\texttt{0x10000}} \mid \underbrace{\texttt{DISPLAY\_BG1\_ACTIVE}}_{\texttt{1<<9}}$$

0000 0000 0000 0001 0000 0000 0000 0000

OR

0000 0000 0000 0000 0000 0010 0000 0000

0000 0000 0000 0001 0000 0010 0000 0000

↓

0x00010200

The extended rotoscale mode: Drawing options and colours palette

- Drawing functionality
  - Transparency in pixels representation (not used in framebuffer mode)
  - Transformation matrix: rotate, scale and displacement of backgrounds

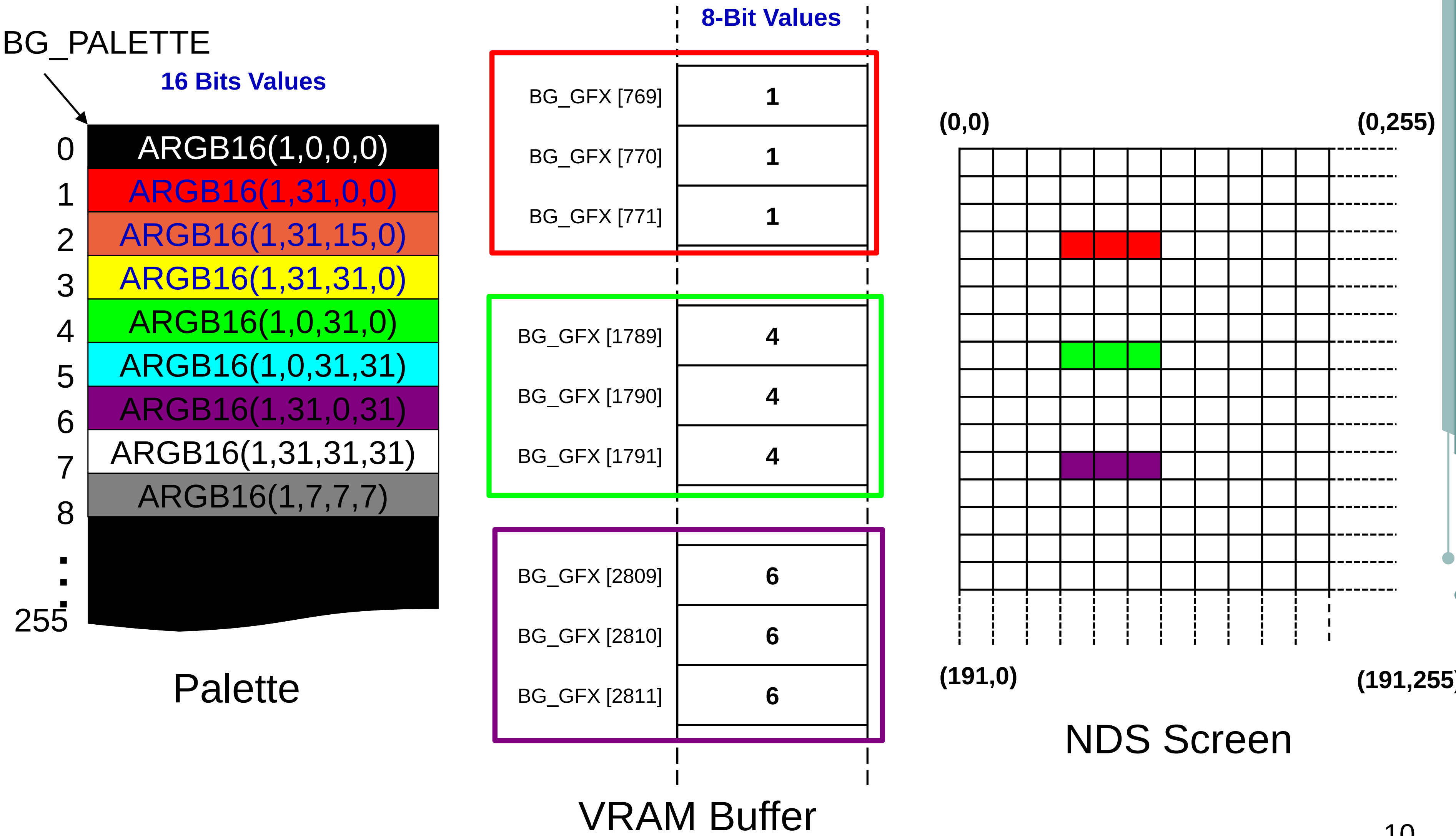
- Use of palettes: compact representation of colours
  - Collection of 256 colors in ARGB16
    - 5 bits per colour, and 1 bit of transparence
  - 8 bits per pixel: color index number in palette

- Video representation storage requirements?
  - Pixels: representation using palettes
  - Palette: representation of colours

Almost half memory size is used with respect to framebuffer format!

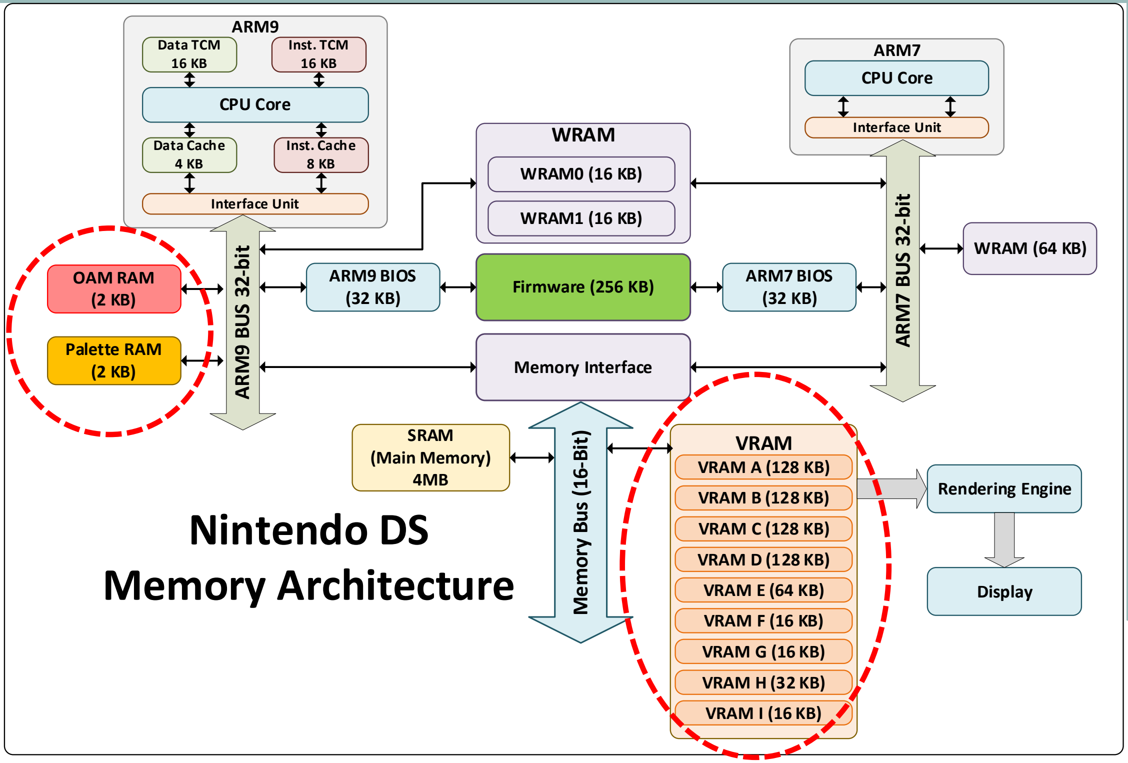
0	ARGB16(1,0,0,0)
1	ARGB16(1,31,0,0)
2	ARGB16(1,31,15,0)
3	ARGB16(1,31,31,0)
4	ARGB16(1,0,31,0)
5	ARGB16(1,0,31,31)
6	ARGB16(1,31,0,31)
7	ARGB16(1,31,31,31)
8	ARGB16(1,7,7,7)
⋮	
255	

The extended rotoscale mode: Palette example



Nintendo DS memory range: Storage of rotoscale video mode data

- Dedicated fast memories for caching
  - Object Attribute Memory (OAM)
    - Sprites or 2D images animations integrated into larger scene
  - Palette RAM
    - 1 x 256 colours
    - 16 x 16 colours

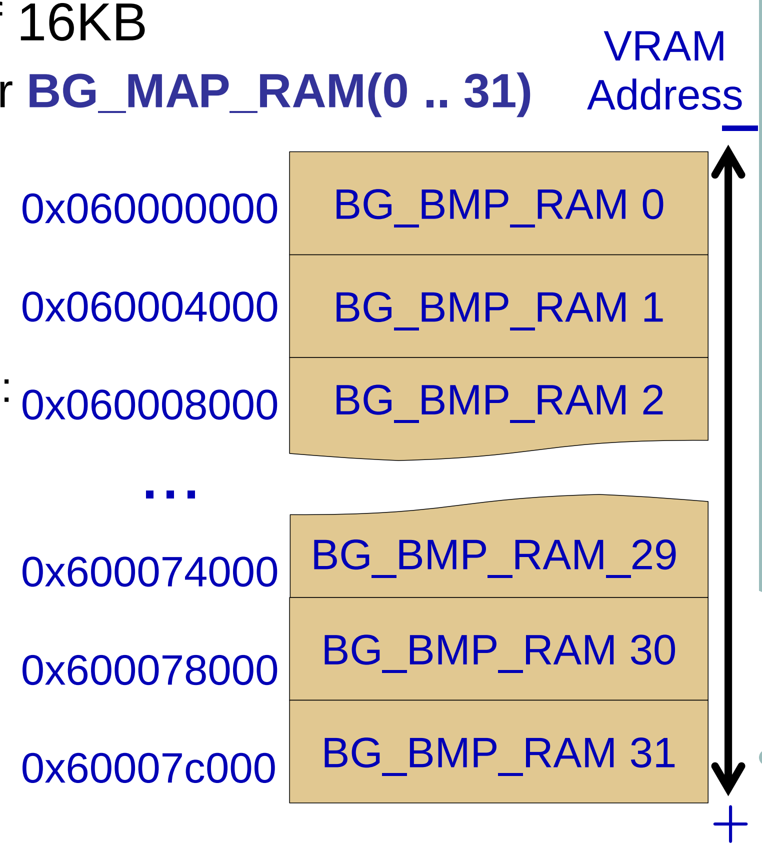


- Background memory: VRAM portions with pixels and palettes
  - Enable the necessary VRAM banks: **VRAM_?_CR**
    - Example: store image of 256x256 pixels with palette format (8 bits / pixel)
      64KB needed: enough with one bank (*bank A*), assign to *main 2D core*

VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;

Structure of background memory: Organized in data blocks

- Background memory divided in 32 blocks of 16KB
  - Pointers in libnds: **BG_BMP_RAM(0 .. 31)** or **BG_MAP_RAM(0 .. 31)**
  - **BG_GFX** is an alias for **BG_BMP_RAM(0)**
- Two steps to configure the background controller: **BGCTRL[x]**
  - Indicate slot in dedicated memory for backgrounds: **BG_BMP_BASE(x)** or **BG_MAP_BASE(x)**
  - Indicate background size and format (multiple options exist in libnds)
    - Example of 256x256 pixels background with palette format: **BgSize_B8_256x256**

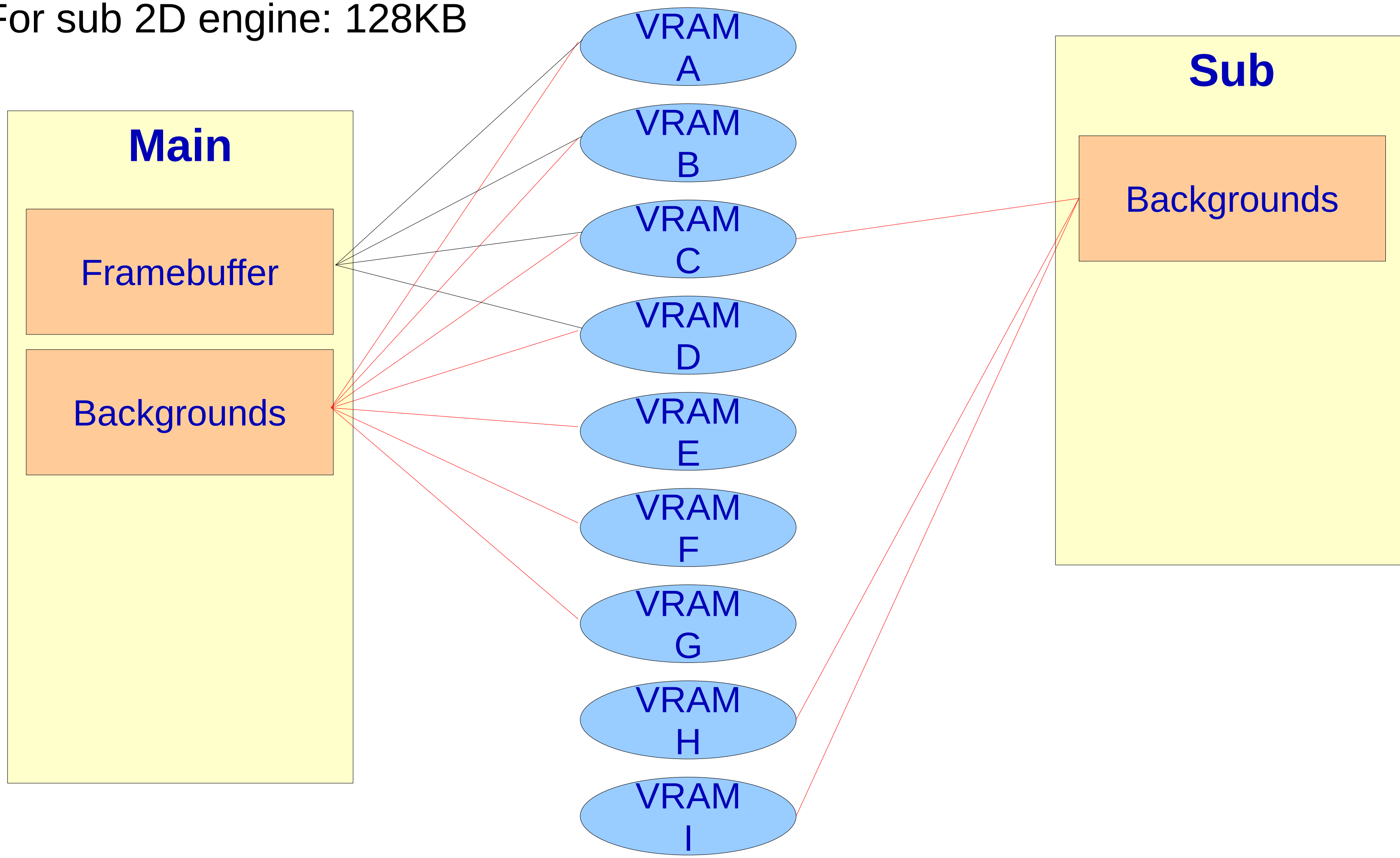


- Example with main 2D engine
  - Configure to use background 2
  - Map data starts at base map address 0 of background memory
  - Image size is 256x256 in palette format (8 bits per pixel index)

BGCTRL[2] = BG_BMP_BASE(0) | BgSize_B8_256x256;

Structure of background memory and VRAM banks mapping

- Background memory has a limited size, and fixed banks assignments
  - For main 2D engine: 512KB
  - For sub 2D engine: 128KB



Initializing the colours palette and the background memory content

- Palette memory without banks, colours stored linearly from base address
  - Pointer to indicate start in libnds: **BG_PALETTE**

```
uint 16* myPalette = BG_PALETTE;
```

- Initialization of palette as pixels in framebuffer mode: **ARGB16** function
  - Loop up to 256 values
  - Example of palette with all possible blue scales:

```
int i;
for (i=0; i<32; i++)
        myPalette[i] = ARGB16(1,0, 0, i);
```

- Background initialization with aux. variable: VRAM writes only of 16/32 bits
  - Declare an 8-bit variable for the screen line of NDS: **uint8 myLine[256];**
  - Initialize the variable with a double loop to cover the whole screen, and copy to memory after each line: use **BG_GFX** libnds macro for video memory

How can we copy large amount of data (instead of one by one) and with sizes of 8 bits instead of 16 or 32 bit?

Data transfers between CPU and I/O subsystems: Two methods

- Standard C function: **stdio.h / string.h**

  void memcpy (void * destination, const void * source, size_t num);*

  - **destination**: Destination memory address for data to transfer
  - **source**: Origin memory address of data to transfer
  - **num**: Data size in bytes to transfer

  - Example: copy line of data in extended rotoscale mode: **uint8 myLine[256];**

```
uint8* mymemory = (uint8*) BG_GFX;
memcpy (mymemory, myLine, 256);     // Copy 1st line of NDS screen
```

- <u>Alternative: Software Interrupt mode (SWI)</u> - Bios function: **nds/bios.h**

  *void swiCopy (const void *source, void *dest, int flags)*

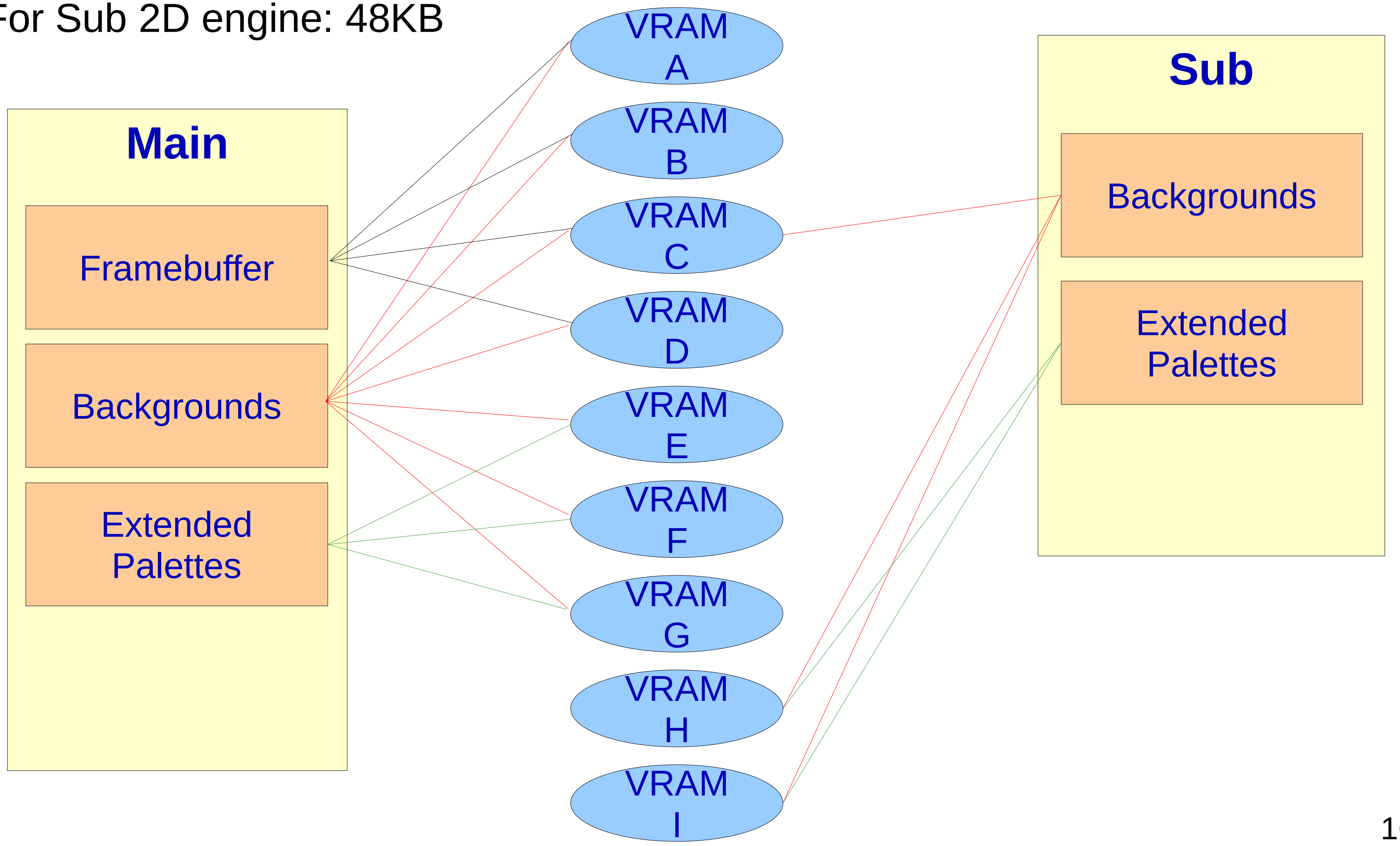
  - **source**: Origin memory address of data to transfer
  - **dest**: Destination memory address for data to transfer
  - **flags**: Data size in 2-byte words to transfer

  - Example: copy line of data in extended rotoscale mode using SWI: **uint8 myLine[256];**

```
uint8* mymemory = (uint8*) BG_GFX;
swiCopy(myLine, mymemory, 128);   // Copy 1st line NDS screen - BIOS
```

Structure of colours palette memory and VRAM banks mapping

- Palettes have a limited size, and fixed banks assignments
  - For main 2D engine: 96KB
  - For Sub 2D engine: 48KB



Complete display configuration in extended rotoscale mode

- A extended rotoscale configuration is characterized by three elements:
  - Displacement in VRAM for multiple backgrounds
    - Banks: BG_BMP_BASE(0) in VRAM_A, … BG_BMP_BASE (31) in VRAM_G, ...
  - Inform 2D engine of image size and pixel configuration (8 or 16 bits): libnds list of *BgSize* enumerator type options in ***background.h***
    - BgSize_B8_256x256 : 256 x 256 pixel 8-bit bitmap background
    - BgSize_B8_1024x512 : 1024 x 512 pixel 8-bit bitmap background
    - BgSize_B16_128x128 : 128 x 128 pixel 16-bit bitmap background
    - …
  - Configure the colour palette for 2D engine: BG_PALETTE
    - Initialize palette to use in BG_PALETTE
    - Use of auxiliary variable to write correctly in VRAM!
- Possible to configure background effects: **affine transformation matrix**

Affine transformation matrix

- Rotoscale mode allows the designer to apply a **Rotation** and **Scaling** operation on the different backgrounds
- Affine transformation matrix:
  - Special matrix that sets the transformation factors per background
- For a given pixel with coordinates *(x,y)* the transformation gives the following new coordinates

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} * \begin{pmatrix} x - x_0 \\ y - y_0 \end{pmatrix}$$

  - ***a, b, c** and **d*** are the factors to scale and rotate
  - (x_0, y_0) correspond to the coordinates of the origin of the system

Affine transformation matrix

- Graphical engines calculate the inverse operation of an affine transformation: for a given pair of coordinates, the corresponding pixel in the original image is obtained
  - They avoid making calculations for areas out of the screen
- The screen can be sequentially refreshed (pixel by pixel) from left to right horizontally and line by line from top to bottom
  - The engines calculate the coordinates of the pixel in the original image that corresponds to a specific pair of coordinates ***(x,y)***

$$(x, y) = (x', y', 1) * \begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix}$$

  – ***(x', y')*** are the coordinates of the pixel to render on the screen
  – ***(x, y)*** are the coordinates of the pixel value en the original image

Affine transformation matrix: Well-known transformations

- No transformation applied

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} 1, & 0 \\ 0, & 1 \\ 0, & 0 \end{pmatrix}$$

- Vertical mirror

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} 1, & 0 \\ 0, & -1 \\ 0, & 192 \end{pmatrix}$$

Affine transformation matrix: Well-known transformations

- Horizontal mirror

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} -1, & 0 \\ 0, & 1 \\ 256 & 0 \end{pmatrix}$$

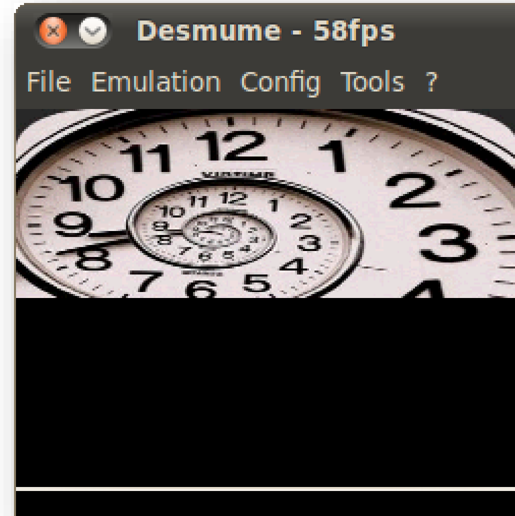
- Vertical and horizontal mirror

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} -1, & 0 \\ 0, & -1 \\ 256, & 192 \end{pmatrix}$$

Affine transformation matrix: Well-known transformations

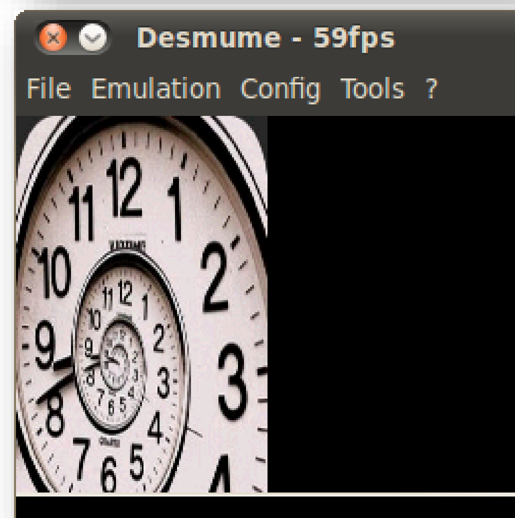
- Vertical shrink

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} 1, & 0 \\ 0, & 2 \\ 0, & 0 \end{pmatrix}$$



- Horizontal shrink

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} 2, & 0 \\ 0, & 1 \\ 0, & 0 \end{pmatrix}$$

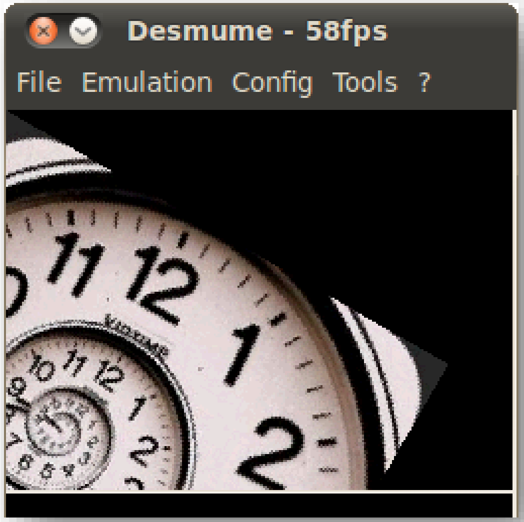


Affine transformation matrix: Well-known transformations

- Clockwise angle ‘Y’

$$\begin{pmatrix} xdx, xdy \\ ydx, ydy \\ dx, dy \end{pmatrix} = \begin{pmatrix} \cos(Y), -\sin(Y) \\ \sin(Y), \quad \cos(Y) \\ 0, \quad 0 \end{pmatrix}$$

  - The angle ‘Y’ must be in radians *(rad)*



The affine transformation matrix in the NDS

- Transformations configured using ***bgTransform*** structure in libnds
  - Although fields declared as signed integers, they use **fixed-point** numbers with 8 bits for decimal part (format: $Q_{1.8}$)
    - Set 1:1 scale => **1.00000000**, so in reality it is: **256 (1<<8)**
  - A number ***x*** in NDS fixed-point arithmetic equals: ***256 * x***

```
typedef struct {
    s16 xdx;
    s16 ydx;
    s16 xdy;
    s16 ydy;
    s32 dx;
    s32 dy;
} bg_transform;
```

- One matrix per background: ***bgTransform[background_nr]***
  - Example: set 1:1 scale and no rotation or translation for BG2
  - Note: in Latest NDS libs, it changed to:
    - **bgTransform[2]->hdx**
    - **bgTransform[2]->vdx**

```
bgTransform[2]->xdx =256;
bgTransform[2]->ydx = 0;
bgTransform[2]->xdy = 0;
bgTransform[2]->ydy = 256;
bgTransform[2]->dx  = 0;
bgTransform[2]->dy  = 0;
```

Images for the NDS - **grit**

- DevkitPro provides a useful tool to transform images from **Portable Network Graphics (png)** format into a readable format for NDS

- **grit : “G**BA **R**aster **I**mage **T**ransmogrifier”
  - Converts the images into assembly code that can be introduced in the final program as a data segment
    - Assembly code generated with user-defined configurable parameters
    - The generated code corresponds to the declaration of: palettes, maps and graphic data
  - Apart from assembly code, it generates a header file to be included C code with all necessary declarations
    - **Link between C code and Assembly Code**

Images for the NDS - **grit**

- ***grit*** can be used from a terminal
  - *Example:* *grit myImage.png –g –gB16*
    - *2 output files will be obtained: " myImage.h" and "myImage.s"*

- Conversion is automated for NDS using few rules included in the **Makefile** of the Eclipse project and a configuration file (extension *.grit*)
  - Place image to transform ("*myImage.png*") and configuration file ("*myImage.grit*") in the project folder ***data***
  - During the compilation process grit will be called and the output files will be placed automatically in the temporary building folder ***build***

- The configuration file must have the same name of the image to transform and contains the necessary formatting parameters
  - In the previous example: ***-g -gB16***

Images for the NDS – **grit**: Conversion parameters

- Main parameters for the grit configuration file
  - -g | -g!   = Include or do not include graphic data (**include always!**)
  - -gb | -gt   = Generate Bitmap or Tiles (**depends on video mode**)
  - -p | -p!   = Include or do not include the palette (**usually generate**)
  - -m | -m! = Include or do not include the map **(used in tiled graphics)**
  - -pnX   = Restrict the palette size to X components
  - -gBX   = Sets the pixel size to X bits. X can be 1, 2, 4, 8 or 16
    - for NDS: –gB8 or –gB16 are used
- Example of a configuration file

```
-g
-gb
-gB8
-m
-p
```

- Complete list of the parameters
  - Type: ***grit --help*** in a terminal

Images for the NDS – **grit** Example

- PNG Image ***(spiral.png)*** + Config file ***(spiral.grit)***

```
-g
-gb
-gB16
```

- Output header file ***(spiral.h)***

```
//{{BLOCK(spiral)

//======================================
//
//  spiral, 256x192@16,
//  + bitmap not compressed
//  Total size: 98304 = 98304
//
//  Time-stamp: 2011-04-06, 22:13:25
//  Exported by Cearn's GBA Image Transmog
//  ( http://www.coranac.com/projects/#gri
//
//======================================

#ifndef GRIT_SPIRAL_H
#define GRIT_SPIRAL_H

#define spiralBitmapLen 98304
extern const unsigned int spiralBitmap[245

#endif // GRIT_SPIRAL_H

//}}BLOCK(spiral)
```

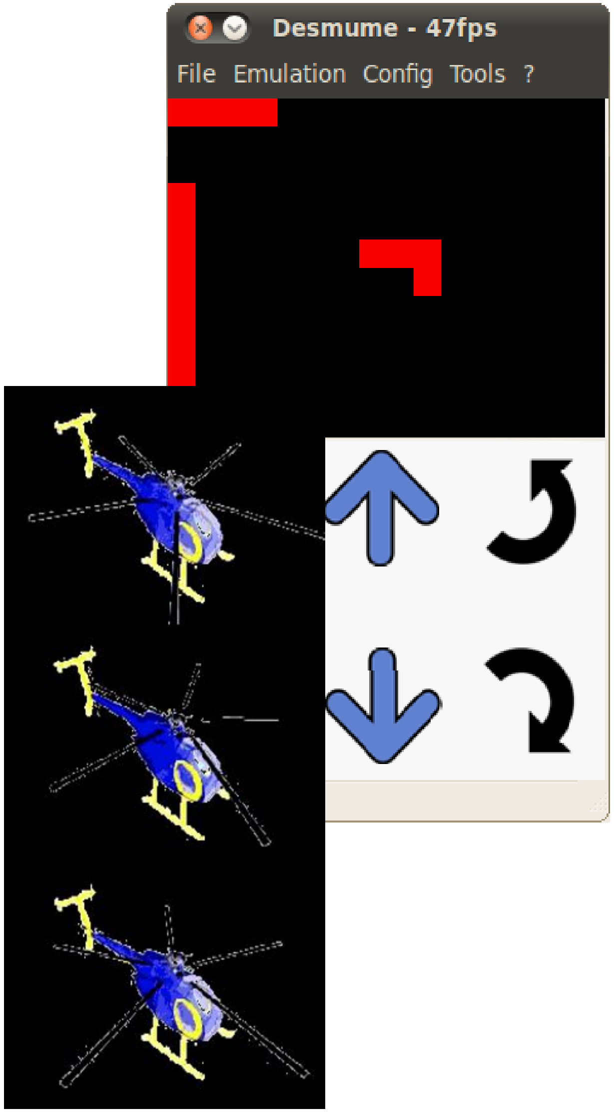
- Output assembly file ***(spiral.s)***

```
@=============================================

    .section .rodata
    .align  2
    .global spiralBitmap        @ 98304 unsigned
spiralBitmap:
    .word 0x14A514A5,0x14A514A5,0x14A514A5,0x14A5
    .word 0x14A514A4,0x108414A5,0x14A514A5,0x10A5
    .word 0x4A533DF0,0x673A5AD7,0x6F7C6B5B,0x739D
    .word 0x0C64252A,0x14A60001,0x5EF84210,0x673A
    .word 0x00000000,0x00000000,0x00000000,0x0000
    .word 0x00000000,0x00000000,0x00010000,0x0422
    .word 0x04010401,0x00000000,0x00000000,0x0000
    .word 0x00000000,0x00000000,0x00000000,0x0000

    .word 0x00000000,0x00000000,0x00000000,0x0000
    .word 0x04210401,0x08430842,0x08640864,0x1086
    .word 0x00010002,0x00010001,0x00010001,0x0000
    .word 0x00000000,0x00000000,0x00000000,0x0000
    .word 0x6F7D739E,0x5AD8737D,0x5275671A,0x0C44
    .word 0x739D739D,0x739D739D,0x739D739D,0x739E
    .word 0x20E92D6C,0x18A618C7,0x14A514A5,0x14A5
    .word 0x14A514A4,0x14A514A5,0x14A514A5,0x14A5
```

Practical Work 7: Graphics (Part 2): Extended rotoscale mode

- Exercises (and homework)
  - Exercise 1 – Changing conditional flag
  - Exercise 2 – Configuring the main engine
  - Exercise 3 – Configuring the background
  - Exercise 4 – Configuring sub engine and background
  - Exercise 5 – Transforming image
  - Exercise 6 – Displaying transformed image

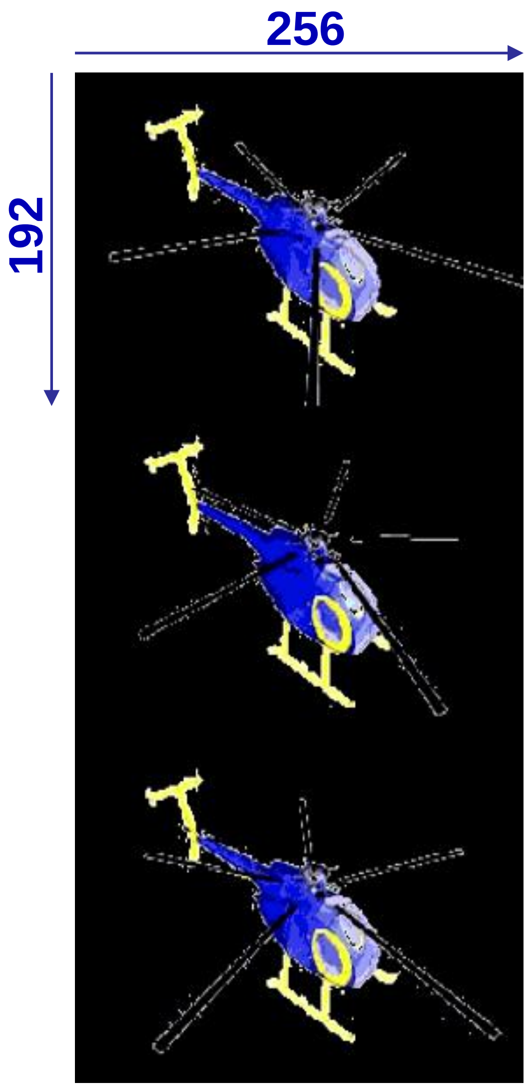
  - *Exercise 8 – Color degradation with rotoscale
  - *Exercise 8 – Color degradation with palette in rotoscale mode
  - *Exercise 9 – Helicopter animation



* Additional exercises

Practical Work 7: Additional Exercise 9
Helicopter animation

- How can we create an animation of a rotating helicopter blade from the image on the right?
  - Solution: Alternate the three image frames fast enough to create the rotating blade effect



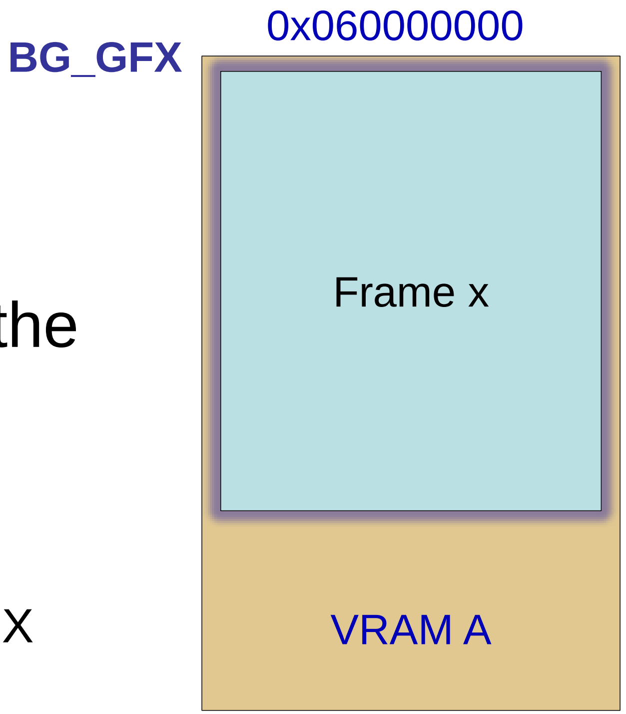
- Provided image: 256x586 pixels
  - Bitmap size: 144 kB for 8-bit pixel depth
  - The image consists of 3 different frames
    - Each frame is 256x192 pixels
    - Bitmap size per frame: 48 kB

- Solution: Use **timer + interrupt** to change the displayed frame periodically
  - Display a different frame in each timer interrupt

Helicopter animation - Solution 1: Copy frame content each interrupt

- Idea: Load only the desired frame in VRAM
  - Only 48 kB needed per frame (8-bit pixel depth)
  - One VRAM bank is enough (i.e., VRAM_A)

- Configure a timer (fast enough to hide the animation: 100ms) and every timer interrupt:
  - Select which frame to display
  - Copy the bitmap of the desired frame to BG_GFX



Helicopter animation - Solution 1
Implementation code

1. Engine, Background, VRAM configuration

```
//Engine and background configuration
REG_DISPCNT = MODE_5_2D | DISPLAY_BG2_ACTIVE;
VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;
BGCTRL[2] = BG_BMP_BASE(0) | BgSize_B8_256x256;
```

2. Copy bitmap and palette (generated by grit)
   - start with Frame 1
   - copy **first 256x192 pixels**

```
//Transfer of the image and the palette to the engine
memcpy(BG_GFX, helicopterBitmap, 256*192);
memcpy(BG_PALETTE, helicopterPal, helicopterPalLen);
```

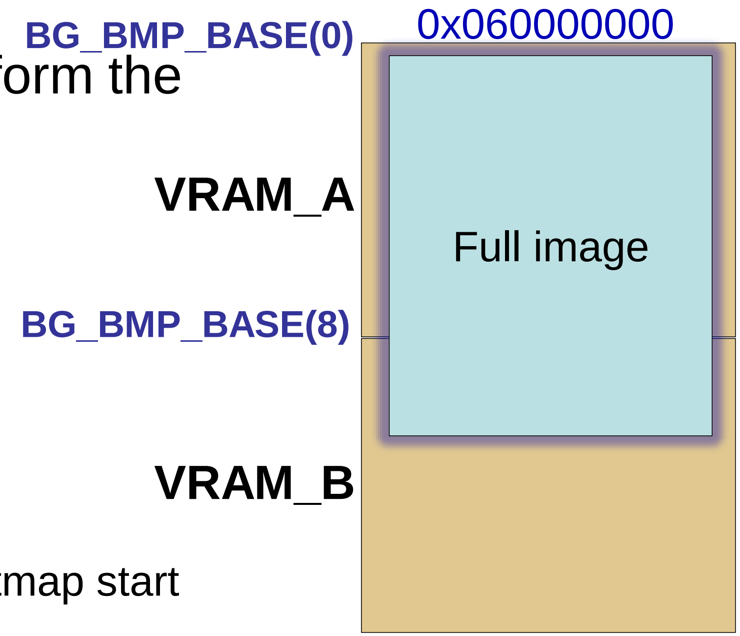
3. Setup timer with interrupt and associate ISR
   - use global variable "image" to select frame
4. - **copy the bitmap** of the selected frame

```
//This variable tells us which image to render (0,1 or 2)
int image = 0;

//Timer0 ISR for changing the image every 100 ms
void Timer0ISR()
{
    //Change image and copy it to the engine memory
    image = (image + 1) % 3;
    int offset = image*256*192/4;
    memcpy(BG_GFX, &helicopterBitmap[offset], 256*192);
}
```

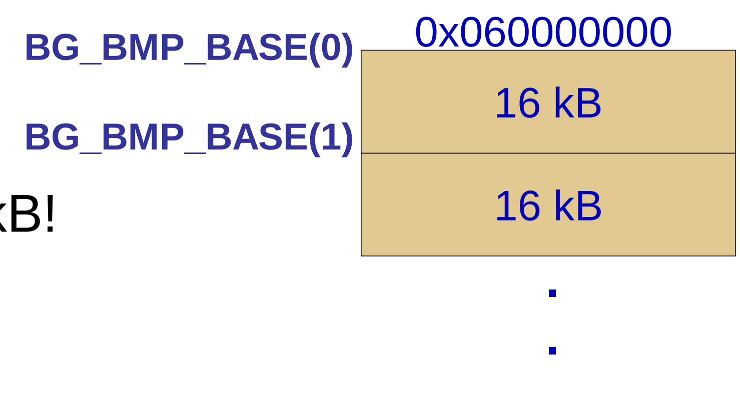
Intuitive solution, but requires copying 48 kB to VRAM per timer interrupt!
How can we avoid the overhead of memory copies?

Helicopter animation - Solution 2: Changing the map base

- <u>Idea</u>: Load the full image in VRAM and inform the background where to look for the bitmap
  - Load all 144 kB in VRAM
  - Modify **BGCTRL** register accordingly
  - Remember **BG_BMP_BASE** indicates where the bitmap starts
- In each timer interrupt:
  - Select which frame to display
  - **Modify the BGCTRL register** to change the bitmap start location
  - Use **BG_BMP_BASE(x)**



- But can we fit the full image in a VRAM?
- Image is 144 kB, VRAM_A has only 128 kB!
  - Activate **both VRAM_A and VRAM_B**
  - Continuous memory of 128 + 128 = 256 kB



Helicopter animation - Solution 2
Implementation code

1. Engine, Background, VRAM configuration
   - **VRAM_A and VRAM_B**
   - VRAM_x_MAIN_BG to ensure VRAM_A and B are assigned to main engine

```c
//Engine and background configuration
REG_DISPCNT = MODE_5_2D | DISPLAY_BG2_ACTIVE;
VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;
VRAM_B_CR = VRAM_ENABLE | VRAM_B_MAIN_BG;
BGCTRL[2] = BG_BMP_BASE(0) | BgSize_B8_256x256;
```

2. Copy bitmap and palette (generated by grit)
   - copy **full image**

```c
//Transfer of the image and the palette to the engine
memcpy(BG_GFX, helicopterBitmap, helicopterBitmapLen);
memcpy(BG_PALETTE, helicopterPal, helicopterPalLen);
```

3. Setup timer with interrupt and associate ISR
   - ISR logic:
   modify BGCTRL and change BG_BMP_BASE
   - use global variable "image" to select frame

```c
//This variable tells us which image to render (0,1 or 2)
int image = 0;

//Timer0 ISR for changing the image every 100 ms
void Timer0ISR()
{
    //Change image and copy it to the engine memory
    image = (image + 1) % 3;
    BGCTRL[2] = BG_BMP_BASE(image * 3) | BgSize_B8_256x256;
}
```

Questions?

Let’s use the screen of the NDS!