Systèmes Embarqués Microprogrammés,
Section d'Electricité (SEL), STI

Prof. David Atienza (ESL), EPFL

SUMMARY DOCUMENTATION FOR THE EE-310 COURSE

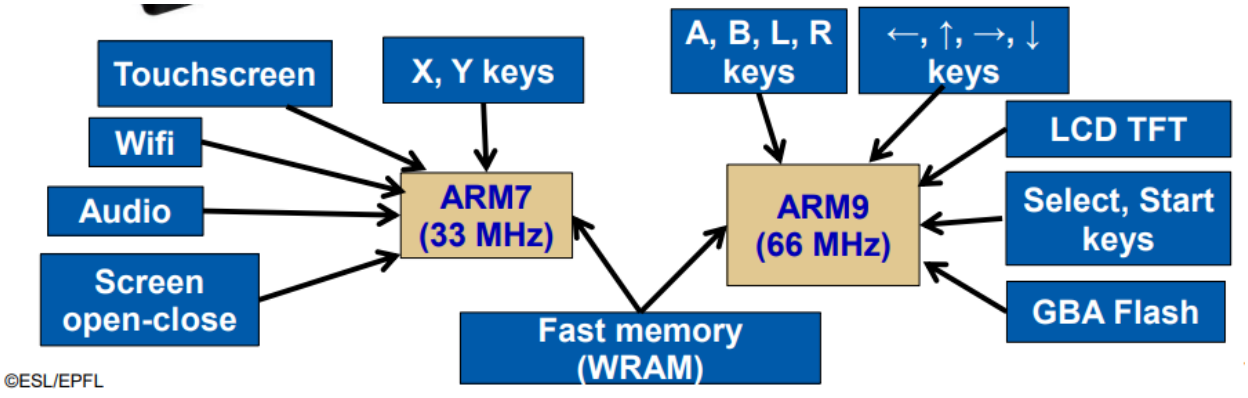
Introduction

This document is a summary of the complete EE-310 course material for the target embedded system used in the course, namely, the Nintendo DS Lite (NDS) platform. We have synthesized the most important information from all theoretical slides to summarize the key points to remember from each peripheral, as well as some typical code examples for configuring them.

This sheet **is ideal** for revision purposes and quick look-up of macros and common code snippets with configurations of peripherals. This **is not suitable** for in-depth understanding of concepts and peripheral usage.

For more detailed code explanations and functions information, you can check the official libnds documentation.

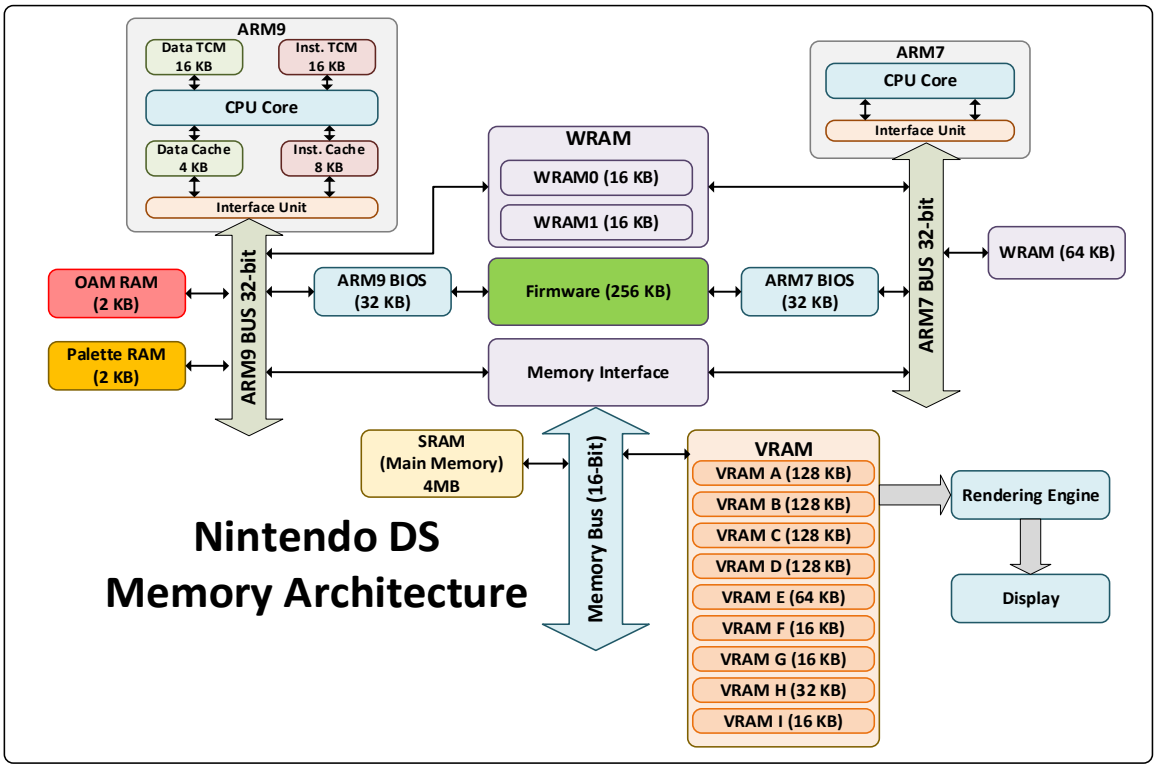
Overview - I/O & Peripheral Devices



Two 32-bit ARM cores manage the I/O and peripherals on the NDS, where each of them is connected to different peripherals:

- ARM9 (66 MHz) - ARM 946E-S
- ARM7 (33 MHz) - ARM 7TDMI-S

Overview – Memory Architecture



Main RAM: 0x02000000 - 0x023fffff (4MB)

VRAM: 0x06000000 - 0x068A0000 (656KB)

Interrupts

Common libnds macros

- Vertical blank:

  `IRQ_VBLANK`

- Horizontal blank:

  `IRQ_HBLANK`

- Timer 0:

  `IRQ_TIMER0`

- Keypad:

  `IRQ_KEYS`

- WiFi:

  `IRQ_WIFI`

API for managing interrupts

- Initialize interrupts subsystem

  ```
  void irqInit();
  ```

  This is the usual way to initialize peripherals (except when using sound).

  WARNING: This function disables the VBLANK interrupts (activated by default).

- Specify the handler to use for the given interrupt in the interrupt table

  ```
  void irqSet(IRQ_MASK irq, VoidFn handler)
  ```

- Remove the handler associated with the interrupt in the interrupt table

  ```
  void irqClear(IRQ_MASK irq)
  ```

- Allow the given interrupt to occur

  ```
  void irqEnable(uint32 irq)
  ```

- Prevent the given interrupt from occurring

  ```
  void irqDisable(uint32 irq)
  ```

Timers

Common libnds macros

- Return a de-referenced pointer to the timer control register x (0 to 3):

  ```
  TIMER_CR(x) or TIMERx_CR
  ```

- Enable the timer:

  ```
  TIMER_ENABLE
  ```

- Timer will count at (33.514 / VALUE) MHz, with VALUE = 1, 64, 256, or 1024:

  ```
  TIMER_DIV_[VALUE]
  ```

- Request interrupt on overflow:

  ```
  TIMER_IRQ_REQ
  ```

- Return a de-referenced pointer to the data register for timer x (0 to 3):

  ```
  TIMER_DATA(x) or TIMERx_DATA
  ```

- Set up the register value to start and overflow each 1/freq second, with VALUE = 64, 256, or 1024 (if no VALUE specified, 1):

  ```
  TIMER_FREQ[_VALUE](freq)
  ```

Choosing the correct divider

To choose the divider to use depending on the actual timer frequency we want, we need to use the smallest possible value in the table below that satisfies our constraints.

Divider	Max Time/Cycle	Min Freq
1	1.955 ms	510 Hz
64	125.151 ms	8 Hz
256	500.603 ms	2 Hz
1024	2.002 s	0.5 Hz

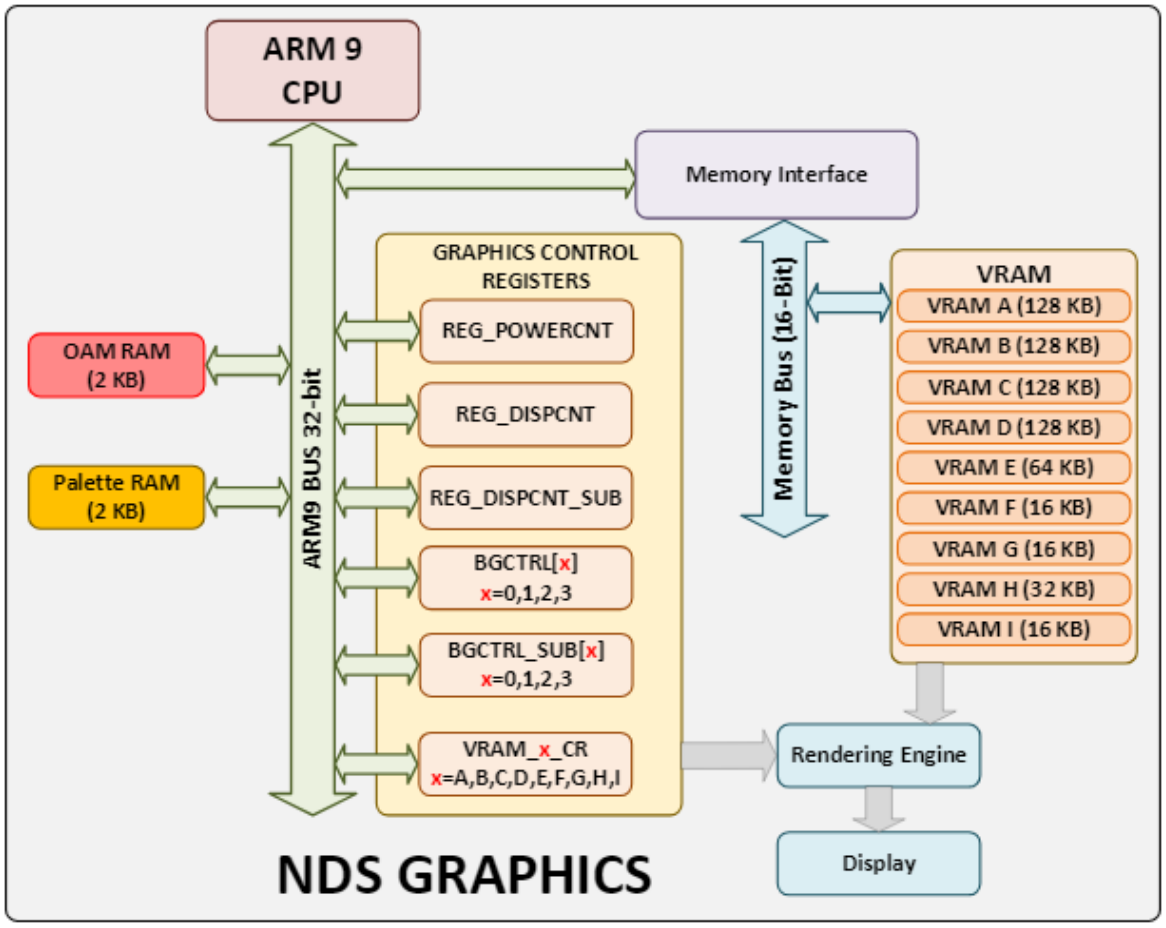
Configuring timer to run out every 20 Hz:

```
int frequency = 20;
TIMER0_CR = TIMER_DIV_64 | TIMER_ENABLE | TIMER_IRQ_REQ;
TIMER0_DATA = TIMER_FREQ_64(frequency);
```

Associate interrupt of timer 0 with `timer_isr` and enable this interrupt:

```
irqSet(IRQ_TIMER0, timer_isr);
irqEnable(IRQ_TIMER0);
```

Graphics



Power Manager

Control register for powering up I/O NDS devices: **REG_POWERCNT**

LCD and engines activated by default during the boot-up process. This means we usually won't need to manually set configurations for the power manager.

It is 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);
```

Video Memory

Control register for each bank to select and activate: `VRAM_[x]_CR`,
where x can be A, B, ..., H, I.

The VRAM is divided into 9 banks of different sizes:

VRAM_A	128KiB
VRAM_B	128KiB
VRAM_C	128KiB
VRAM_D	128KiB
VRAM_E	64KiB
VRAM_F	16KiB
VRAM_G	16KiB
VRAM_H	32KiB
VRAM_I	16KiB

Activate bank A and map it to the framebuffer:

```
VRAM_A_CR = VRAM_ENABLE | VRAM_A_LCD;
```

Activate bank A and map it to the main background:

```
VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;
```

Graphical Engine

Display register to control the mode and active backgrounds: **REG_DISPCNT**

The selection of the most suitable configurations for the graphical engine is done according to the existing screen modes for each engine, as well as according to the requirements of resolution and effects on the graphics to be shown on the screen.

Use **REG_DISPCNT** to choose configurations for the main graphical engine (the top screen of the NDS) and **REG_DISPCNT_SUB** to choose the configurations for the secondary graphical engine (the bottom screen of the NDS).

Activate mode 0 and background 1:

```
REG_DISPCNT = MODE_0_2D | DISPLAY_BG1_ACTIVE;
```

Pixels & Colors

Each pixel has an RGB (Red-Green-Blue) representation of 16 bits

- 5 bits for the intensity of each colour (0: none, 31: maximum value)
- 1 bit for transparency (0: pixel is transparent, 1: pixel is opaque)

Some important reference colors are the following ones:

Red	`RGB15(31,0,0)`
Green	`RGB15(0,31,0)`
Blue	`RGB15(0,0,31)`
Yellow	`RGB15(31,31,0)`
Black	`RGB15(0,0,0)`
White	`RGB15(31,31,31)`

With Transparency bit:

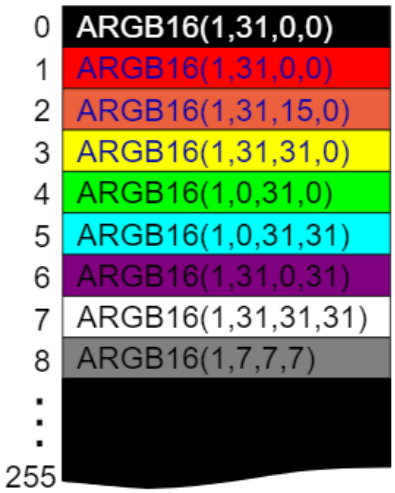
```
static uint16 shape_color = ARGB16(1, 31, 0, 0);
```

Without Transparency bit:

```
static uint16 shape_color = RGB15(31, 0, 0);
```

Palettes

A palette is a color collection used to reduce index sizes needed to represent pixel colors. Instead of colors being 16 bits per pixel (ARGB16), we use only 8 bits per pixel. Therefore, we predefine 256 colors (defined in ARGB16 and stored in a collection), and then use the index of these colors to set pixel colors. Since we have 256 available colors in our defined palette, this means each index is only 8 bits.



Palettes are stored in the palette RAM, which stores either one 256 color palette, or 16 palettes of 16 colors each. With 256 colors the color indexes go from 0x00 to 0xFF. When using 16 palettes of 16 colors, the first digit (0x0 to 0xF) is the palette and the second one (0x0 to 0xF) is the color.

Index 0 of a palette represents transparency: a pixel set at this color will display opaque pixels behind it from other backgrounds if there are any. If there is no opaque pixel behind it, it will appear as the actual RGB color stored at index 0.

Initializing a palette:

```
uint 16* myPalette = BG_PALETTE;
int i;
for (i=0; i < 32 ; i++) {
    myPalette[i] = ARGB16(1,0, 0, i);
}
```

Screen & Modes

Each of the screens have 49152 pixels: 192 rows of 256 points each. The screen draws pixels sequentially from left to right and up to down (the origin is always top left).

Two interrupts occure from the screen drawing:

`IRQ_HBLANK` Horizontal blank: after each line is drawn

`IRQ_VBLANK` Vertical blank: after all of the lines are drawn

The bitmap content to draw is changed after a vertical blank, before the start of the next redraw.

Each engine has four backgrounds (or layers): BG0, BG1, BG2 and BG3. Final view on the screen is their combination based on the graphic mode.

Each 2D engine has different sets of four possible modes:

- Tiled
- Rotoscale
- Extended Rotoscale
- Framebuffer

There are two 2D engines on the NDS

- Main: can display both video memory content or bitmaps of 256 colors (framebuffer mode).
- Sub: secondary display that can only use the video memory content.

Main engine modes

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 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

Backgrounds

Up to four superposed backgrounds can be used at a time, and each one can be used in a different mode. BG0 is the topmost background and BG3 the furthest. Therefore, BG0 is always visible as it is configured by default during booting, and the most internal (BG3) is only visible if the other backgrounds do not have any valid pixel to show.

To configure a background we need to set the corresponding background control register: **`BGCTRL[x] or BGCTRL_SUB[x],`** where x is the number of the background.

For (**ext.) Rotoscale mode**, we indicate to the background control register:

1. The memory slot where the map lies: i.e. **`BG_BMP_BASE(0),`**
   when the map lies at the start of the corresponding VRAM bank
1. Background size and format: i.e. **`BgSize_B8_256x256,`** for 256-color palette with 256x256 pixels map

For **Tiled mode**, we indicate to the background control register:

1. Grid size: i.e. **`BG_32x32`**, for 32x32 tiles
2. Color amount: i.e. **`BG_COLOR_256`**, for 256-color palette
3. Memory slot where the map lies: i.e. **`BG_MAP_BASE(0)`**
4. Memory slot where the tiles lie: i.e. **`BG_TILE_BASE(1)`**
   (careful not to overlap map with tiles → check VRAM management section)

Configuring main engine background 2 in Rotoscale mode emulating Framebuffer:

```
BGCTRL[2] = BG_BMP_BASE(0) | BgSize_B16_256x256;
```

Configuring sub engine background 2 in Rotoscale mode:

```
BGCTRL_SUB[2] = BG_BMP_BASE(0) | BgSize_B8_256x256;
```

Configuring main engine background 0 in tiled mode:

```
BGCTRL[0] = BG_32x32 | BG_COLOR_256 | BG_MAP_BASE(0) |
BG_TILE_BASE(1);
```

Framebuffer Mode

In the Framebuffer mode, **each pixel of the screen is directly mapped** to a portion of memory (192 x 256 bitmap) indicating the color of each pixel.

There are four different framebuffers, each linked to one of the four 128KiB VRAM banks.

FB0	VRAM_A
FB1	VRAM_B
FB2	VRAM_C
FB3	VRAM_D

The pixel format is **always RGB15** in the framebuffer mode: **pixels are always opaque**, so the transparency bit is not used.

Configuration when using Framebuffer FB0:

```
REG_DISPCNT = MODE_FB0;
VRAM_A_CR = VRAM_ENABLE | VRAM_A_LCD;
```

Making pixel (i, j) black:

```
u16* buff = (u16*) VRAM_A;
buff[i * 256 + j] = RGB(0, 0, 0);
```

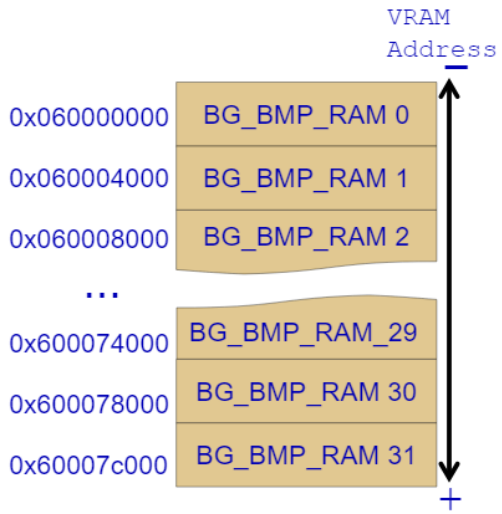
TIP - Framebuffer in bottom screen

To draw onto the bottom screen, we need the framebuffer mode, which is only possible on the main 2D engine. **To make the bottom screen use the main 2D engine, flip bit 15 in the `REG_POWERCNT` register**.

Rotoscale Mode

In the Rotoscale mode, **each pixel is mapped to an index** in the memory which indicates a position in the palette, hence the color to be used. Each pixel is represented with 8 bits, which are enough to index a 256-color palette (or 16 palettes of 16 colors).

The map of indexes is stored at **`BG_BMP_RAM(x)`** and we use **`BG_BMP_BASE(x)`** to indicate to the background register where to look for the map (**`x = 0, 1, … 31`**). The macros above allow the placement of the map in **multiple of 16 KiB** in the VRAM:



Configuration when using Rotoscale mode:

```
REG_DISPCNT = MODE_4_2D | DISPLAY_BG2_ACTIVE;
VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;

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

Making pixel (i, j) black:

```
u16* myPalette = (u16*) BG_PALETTE;
myPalette[1] = ARGB(1, 0, 0, 0);
u8* myMemory = (u8*) BG_GFX;
myMemory [i * 256 + j] = 1;
```

The **extended Rotoscale mode** allows the use of a **transformation matrix** to rotate, scale and displace the original background. Here are some well-known transformations:

$$\begin{pmatrix} hdx & hdy \\ vdx & vdy \\ dx & dy \end{pmatrix}$$ Transformation matrix format

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$ Identity

$$\begin{pmatrix} 1 & 0 \\ 0 & -1 \\ 0 & 192 \end{pmatrix}$$ Vertical Mirror

$$\begin{pmatrix} -1 & 0 \\ 0 & 1 \\ 256 & 0 \end{pmatrix}$$ Horizontal Mirror

$$\begin{pmatrix} -1 & 0 \\ 0 & -1 \\ 256 & 192 \end{pmatrix}$$ Vertical & Horizontal Mirror

$$\begin{pmatrix} 1 & 0 \\ 0 & 2 \\ 0 & 0 \end{pmatrix}$$ Vertical Shrink (divide height by 2)

$$\begin{pmatrix} 2 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$ Horizontal Shrink (divide width by 2)

$$\begin{pmatrix} \cos y & -\sin y \\ \sin y & \cos y \\ 0 & 0 \end{pmatrix}$$ Clockwise Rotation by y (in radians)

The transformation matrix is configured using **`bgTransform`**.

Although the fields are declared as signed integers, they use **fixed-point numbers**, with 8 bits for decimal part**. So, to write a number x we must actually use x * 256**

```
typedef struct {
    s16 hdx;
    s16 vdx;
    s16 hdy;
    s16 vdy;
    s32 dx;
    s32 dy;
} bg_transform;
```

Configure the transformation matrix to be the identity matrix (n is the background number):

```
bgTransform[n]->hdx = 256;
bgTransform[n]->vdx = 0;
bgTransform[n]->hdy = 0;
bgTransform[n]->vdy = 256;
bgTransform[n]->dx  = 0;
bgTransform[n]->dy  = 0;
```

Tiled Mode

In the tiled mode, the backgrounds are composed of smaller blocks. The minimum unit used to create tiled backgrounds are called **tiles and are blocks of 8x8 pixels**.

Backgrounds are maps of tiles: they reference tiles using their index. The tile collection can contain up to 1024 tiles, meaning their index is 10 bits.

Palettes are also used for pixels in tiled mode. Either one 256-color palette or 16 palettes of 16 colors are used.

Tiles

A tile is essentially a 8x8 matrix of color indexes from a palette. These color indexes are either 8 bits (256 color palette) or 4 bits (16 color palette).

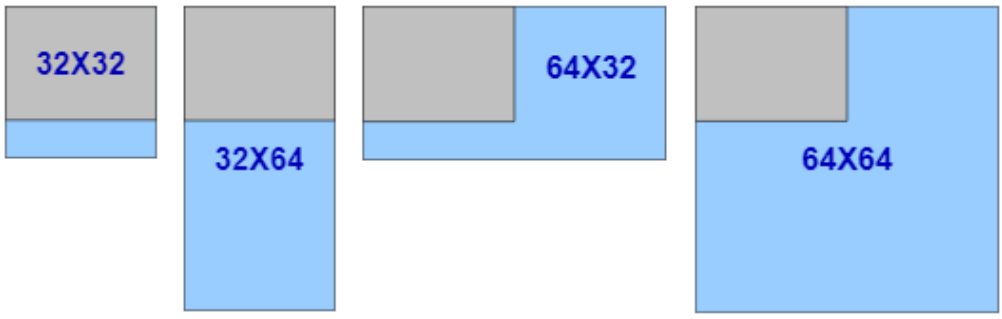
Tiles are stored from the base address **`BG_TILE_RAM(x)`**. Tile base addresses are **multiples of 16 KiB** and the maximum size is 256 KiB.

We use **`BG_TILE_BASE(x)`** to configure the background control register and **`BG_TILE_RAM(x)`** (or **`BG_TILE_RAM_SUB(x)`**) to access the tile set or modify it (**`x = 0, 1, … 15`**).

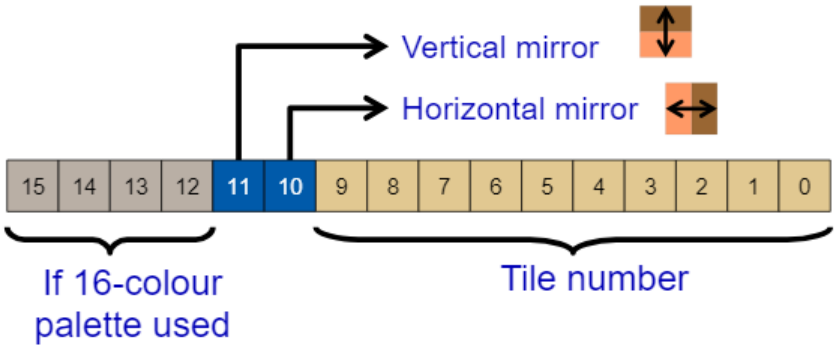
Tile maps

A map of tiles can be larger than the screen, and the visible area can then be adjusted with a scroll effect. **The screen has size 32x24 tiles** (256x192 pixels).

There are four possible map sizes on the NDS:

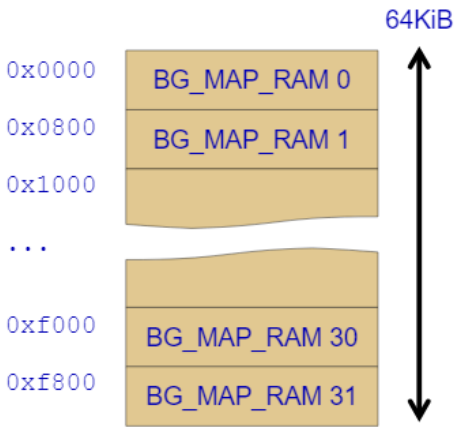


Each tile in a map is represented with 16 bits:



Maps are stored from the base address **`BG_MAP_RAM(x)`**. Map base addresses are **multiples of 2 KiB** and the maximum size is 64 KiB. The most used size of map is 32x32 tiles, which is exactly 2KiB (2B per tile), but other map sizes may use up more than one slot in RAM.

We use **`BG_MAP_BASE(x)`** to configure the background control register and **`BG_MAP_RAM(x)`** (or **`BG_MAP_RAM_SUB(x)`**) to access the tile set or modify it (**`x = 0, 1, … 31`**).



NOTE:

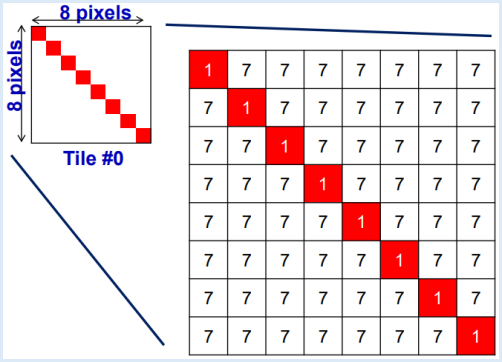
- 64x64 maps require the use of 4 map slots. They are not stored linearly, but by quadrant: each of the 4 used map bases contain a quarter of the total map.

Configuration when using Tiled mode:

```
REG_DISPCNT = MODE_0_2D | DISPLAY_BG1_ACTIVE;
VRAM_A_CR = VRAM_ENABLE | VRAM_A_MAIN_BG;

BGCTRL[1] = BG_32x32 | BG_COLOR_256 |
            BG_MAP_BASE(0) | BG_TILE_BASE(1);
```

Setting tile (i, j) to white pixels with red diagonal:




```
// Define palette colors
BG_PALETTE[1] = RGB(31, 0, 0);
BG_PALETTE[7] = RGB(31, 31, 31);

// Create and copy tile to correct VRAM slot
u8 tileRedDiagonal[64] = {
        1, 7, 7, 7, 7, 7, 7, 7,
        7, 1, 7, 7, 7, 7, 7, 7,
        7, 7, 1, 7, 7, 7, 7, 7,
        7, 7, 7, 1, 7, 7, 7, 7,
        7, 7, 7, 7, 1, 7, 7, 7,
        7, 7, 7, 7, 7, 1, 7, 7,
        7, 7, 7, 7, 7, 7, 1, 7,
        7, 7, 7, 7, 7, 7, 7, 1 };
dmaCopy(tile0, &BG_TILE_RAM(1)[1], 64);

// Set tile map (i, j) to point at the correct tile
BG_MAP_RAM(0)[32*i + j] = 1;
```

Sprites

Sprites are **small graphic objects that can be rendered on top of the backgrounds** and provide extended features:

- Different sizes are possible (8x8, 16x16, 32x32, 64x64, 16x8, 32x8, 32x16, 64x32, 8x32, 16x32)
- They can be rendered in any position, on or outside of the screen
- They can be rotated, scaled or flipped.
- The amount of sprites is fixed to 128 and they are hidden by default.
- A special mode has to be configured in a VRAM bank: this bank cannot be used for anything else (e.g.: for backgrounds).
- They use a special palette (different from the one used for backgrounds) and can use extended palettes

Sprite Configuration:

```
VRAM_B_CR = VRAM_ENABLE | VRAM_B_MAIN_SPRITE;

// Initialize sprite manager and the engine
oamInit(&oamMain, SpriteMapping_1D_32, false);

// Allocate space for the graphic to show in the sprite
gfx = oamAllocateGfx(&oamMain, SpriteSize_32x32,
SpriteColorFormat_256Color);

// Copy data for th graphic
swiCopy(imagePal, SPRITE_PALETTE, imagePalLen / 2);
swiCopy(imageTiles, gfx, imageTilesLen);
```

Moving a sprite with the directional keys:

```
// Position and keys
int x = 0, y = 0, keys;

while (1) {
    // Read held keys
    scanKeys();
    keys = keysHeld();

    // Modify position of the sprite accordingly
    if((keys & KEY_RIGHT) && (x < SCREEN_WIDTH-SPRITE_WIDTH))
        x+=2
    if((keys & KEY_DOWN) && (y < (SCREEN HEIGHT-SPRITE_HEIGHT))
        y+=2;
    if((keys & KEY_LEFT) && (x > 0))
        x-=2;
    if((keys & KEY_UP) && (y > 0))
        y-=2;

    oamSet (&oaMain ,         // oam handler
        0,                    // Number of sprite
        x, y,                 // Coordinates (top left)
        0,                    // Priority
        0,                    // Palette to use
        SpriteSize_32x32,     // Sprite size
        SpriteColorFormat_256Color, // Color format
        gfx,           // Loaded graphic to display
        -1,            // Affine rotation (-1 none)
        false,         // Double size if rotating
        false,         // Hide this sprite
        false, false,  // Horizontal or vertical flip
        false);        // Mosaic

     swiWaitForVBlank();

    //Update the sprites
    oamUpdate(&oamMain);}
```

Grit

Grit (GBA Raster Image Transmogrifier) is a **tool that transforms images from PNG format into an NDS readable format**.

It generates assembly code and a C header file (declaration of palettes, maps and graphic data).

Conversion is automated using a few rules included in the Makefile of the project and a configuration file (extension .grit). The configuration file simply contains some parameter flags. It must have the same name as the image.

- 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 grit configuration file may contain the following parameters :

-g | -g!	Include or do not include graphic data	Always Include!
-gb | -gt	Generate Bitmap or Tiles	Depends on Mode
-p | -p!	Include or do not include the palette	Usually Generate
-m | -m!	Include or do not include the map	For Tiled Mode
-pnX	Restrict the palette size to X colors	
-gBX	Sets the pixel size to X bits	X = 8 or 16
-gTC	Sets the color C to transparent	C = hex

Grit can be used from the terminal like this:

```
grit myImage.png -g -gB16
```

For more parameters use:

```
grit --help
```

- For tiles, image sizes must be multiples of 8
- Transparency color: should be a 16-bit BGR color or 24-bit RGB color in hex

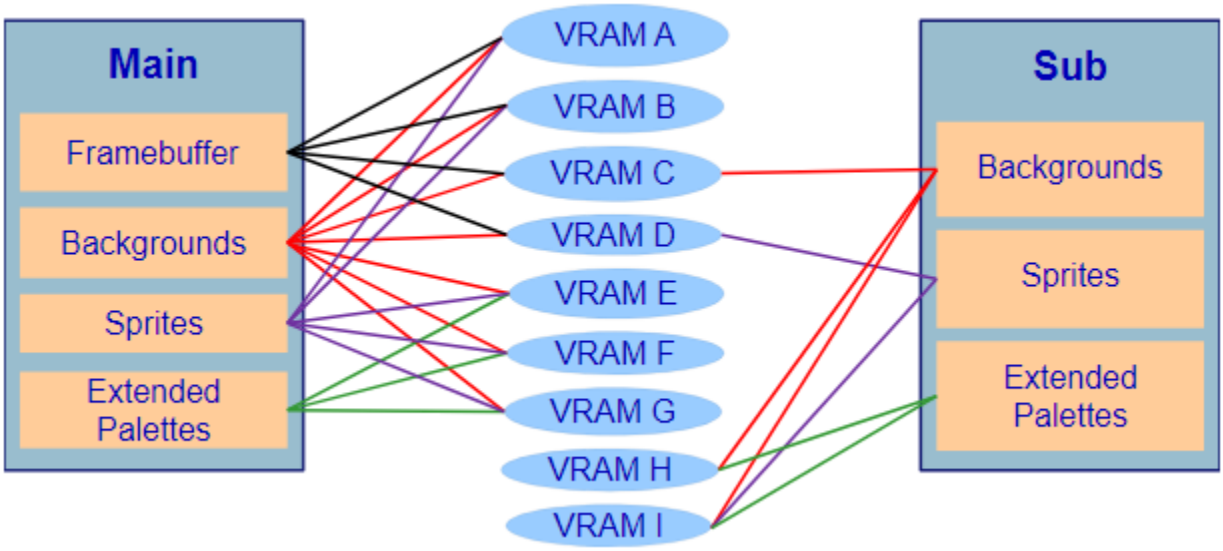
VRAM Management

Many different types of data may be stored in VRAM, like palettes, maps, tiles... Because all of these are stored in the same memory space we **must be careful to avoid overlaps**, or we may accidentally overwrite some parts of memory and loose data.

To do this we must take into consideration base addresses as well as storage size. For example `BG_TILE_BASE` and `BG_MAP_BASE` may overlap: both begin at the same place in memory. This is something we must take into account when assigning our slots.

Note that the VRAM banks reside in the background memory starting from 0x6000000. If you use the macro `VRAM_x_MAIN_BG` when activating `VRAM_x`, then all the VRAM banks will be continuous starting from `VRAM_A` at `0x6000000`.

Each type of data has a fixed assignment in VRAM banks:



For the main engine, the base indexes we use to set the background register look like this:

Addresses	Bitmap	Tiles	Maps
0x00000	BMP BASE 0	TILE BASE 0	MAP BASE 0
0x00800			MAP BASE 1
0x01000			MAP BASE 2
0x01800			MAP BASE 3
0x02000			MAP BASE 4
0x02800			MAP BASE 5
0x03000			MAP BASE 6
0x03800			MAP BASE 7
0x04000	BMP BASE 1	TILE BASE 1	MAP BASE 8
0x04800			MAP BASE 9
0x05000			MAP BASE 10
0x05800			MAP BASE 11
0x06000			MAP BASE 12
0x06800			MAP BASE 13
0x07000			MAP BASE 14
0x07800			MAP BASE 15
0x08000	BMP BASE 2	TILE BASE 2	MAP BASE 16
...	...	...	...
0x0F800			MAP BASE 31
0x10000	BMP BASE 4	TILE BASE 4	END
...	...	...	
0x3C000	BMP BASE 15	TILE BASE 15	
0x40000	BMP BASE 16	END	
...	...		
0x7C000	BMP BASE 31		
0x80000	END		

Data Transfer API

There are mutliple ways to transfer data between the CPU and I/O subsystems.

MemCopy

```
void *memcpy(void *dest, const void * src, size_t n)
```

dest	Pointer to the destination array where the content is to be copied
src	Pointer to the source of data to be copied
n	Number of bytes to be copied

SwiCopy

```
void swiCopy(const void *src, void *dest, int flags)
```

dest	Pointer to the destination array where the content is to be copied
src	Pointer to the source of data to be copied
flags	Data size in 2-byte words to transfer

You can use `swiCopy` as you would `memCopy`, only dividing n by 2 to obtain flags (`swiCopy` copies half words instead of bytes).

DMACopy

```
void dmaCopy(const void *src, void *dest, uint32 size)
```

dest	Pointer to the destination array where the content is to be copied
src	Pointer to the source of data to be copied
n	Number of bytes to be copied

`dmaCopy` uses Direct Memory Access to copy data. Using DMA is generally faster and more efficient than CPU copies (`memcpy, swiCopy`) because it offloads the task of data transfer to a dedicated hardware component, allowing the CPU to perform other tasks.

SOME TIPS

BG_GFX is simply a u16 pointer to the address 0x6000000, very useful for copying the bitmaps. This address is the start of VRAM.

BG_BMP_RAM and other similar pointers are often u16 pointers! Make sure to take this into account when incrementing your pointers.

For example:

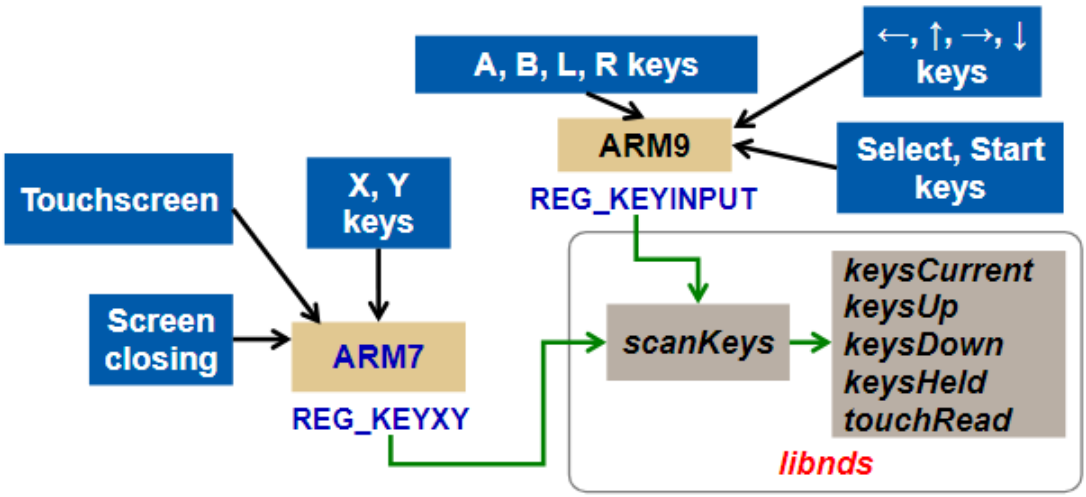
```
dmaCopy(RedTile, BG_TILE_RAM_SUB(1), 64);
dmaCopy(BlueTile, BG_TILE_RAM(1) + 32, 64);
dmaCopy(GreenTile, BG_TILE_RAM(1) + 32*2, 64);
```

Tiles are 64 bytes here but we only increment by 32 each time because our pointer is u16 (one increment is 2 bytes).

You can also cast your pointer to a u8 or void pointer instead.

Controls

Controls are handled on either the ARM9 or ARM7 CPU, depending on the button.



Two methods exist to use the I/O: 1. **Polling**, 2. **I/O interrupts**

1. **Polling:** programmed I/O with active waiting, using LibNDS methods

   Controls linked to ARM7 can only be used with polling (touchscreen, X and Y Keys) and not with interrupts.

   `void scanKeys()` : scans and stores pressed keys
   `uint32 keysHeld()` : returns keys pressed and held
   `uint32 keysDown()` : returns keys just pressed
   `uint32 keysUp()` : returns keys just released
   `void touchRead(touchPosition* pos)` : where was the touch screen pressed?

2. **I/O interrupts:** directly managing the REG_KEYCNT register (setup interrupts using bits manipulation of the register using C code).

Whether a key is currently pressed is stored in the REG_KEYINPUT register for the ARM9 and the REG_KEYXY register for the ARM7. **A 0 means the key is pressed, not a 1.**

Control interrupts are configured in the REG_KEYCNT register. The lower 14 bits control which bits can trigger interrupts. These also correspond to the bits in REG_KEYINPUT and REG_KEYXY.

Bit	LibNDS Name	Description
0	KEY_A	The A button
1	KEY_B	The B button
2	KEY_SELECT	The Select button
3	KEY_START	The Start button
4	KEY_RIGHT	Right arrow D-Pad
5	KEY_LEFT	Left arrow D-Pad
6	KEY_UP	Up arrow D-Pad
7	KEY_DOWN	Down arrow D-Pad
8	KEY_R	Right shoulder button
9	KEY_L	Left shoulder button
10	KEY_X	The X button
11	KEY_Y	The Y button
12	KEY_TOUCH	Touchscreen pressed
13	KEY_LID	Lid closed

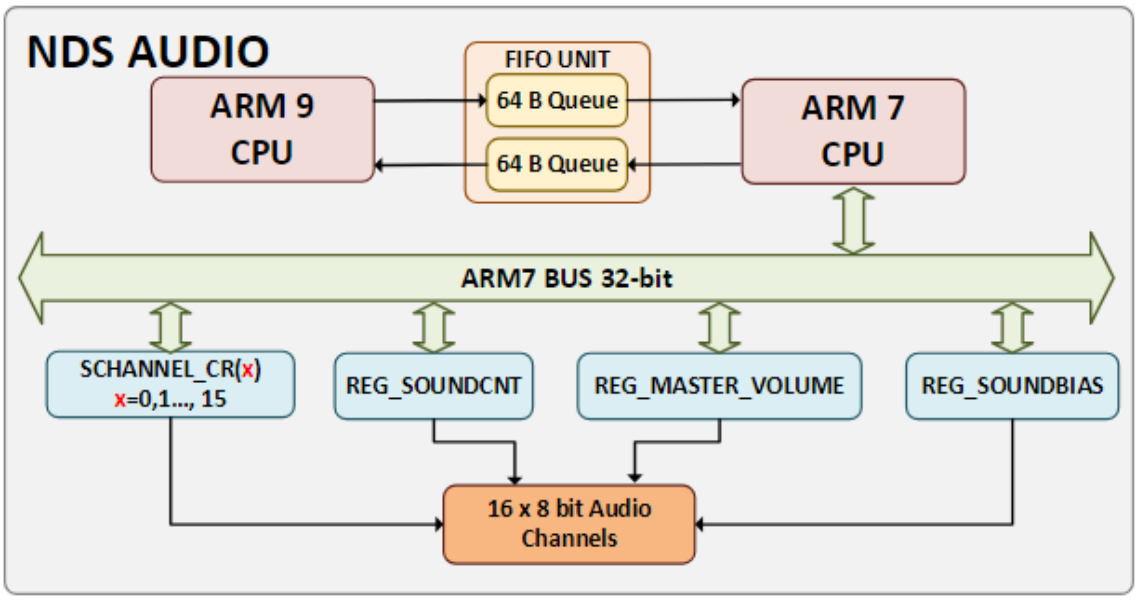
In the REG_KEYCNT register, bit 14 enables the key button interrupt (i.e. when any button whose KEYCNT bit is set is pressed, it triggers an interrupt). Bit 15 makes it so all buttons whose KEYCNT bits are set need to be pressed at the same time to trigger an interrupt.

To read where the touchscreen is pressed, call `touchRead(touchPosition* pos)` to write into pos the touchscreen data.

`touchPosition`:

```
typedef struct {
    u16 rawx; // Raw x value
    u16 rawy; // Raw y value
    u16 px; // Pixel X value
    u16 py; // Pixel Y value
    u16 z1; // Raw cross panel resistance
    u16 z2; // Raw cross panel resistance
}
```

Sound



Power on the sound system :

```
powerON(POWER_SOUND);
```

Set the sound configurations :

```
SOUND_CR= SOUND_ENABLE | SOUND_VOL(0x7F);
```

The volume goes from 0x00 (silent) to 0x7F (full).

Each channel can then be configured independently through `SCHANNEL_CR(n)`, and additional properties can be configured using `SCHANNEL_property(n)`.

EXAMPLE

```
SCHANNEL_CR(0) = SCHANNEL_ENABLE | SOUND_ONE_SHOT | SOUND_8BIT;
SCHANNEL_TIMER(0) = SOUND_FREQ(11127);
SCHANNEL_SOURCE(0) = (uint32)sound1;
SCHANNEL_LENGTH(0) = ((int)sound1_end - (int)sound) >> 2;
```

We use the MaxMod API to deal with sound. We can use two types of sound: module files (.mod, .s3m, .it or .xm) for background music, and sound effects (.wav) to be played on demand.
Sound effects can also be a module sound played only once instead of looping, but this is not recommended. Wav file generate large binaries, so they must be used sparingly.

Some useful resources:

- The Mod Archive (music modules)
- WavSource (wav sounds)

Initializing the library pointers to sounds and internal buffers :

```
void mmInitDefaultMem(mm_addr soundbank);
```

The input parameter is the name of the sound-bank binary object (by default it is **soundbank_bin**).

Load music modules :

```
void mmLoad(mm_word module_ID);
```

The input parameter is the 32-bit index with the module identifier (by default all identifiers are defined in **soundbank.h**).

Load sound effect :

```
void mmLoadEffect(mm_word sample_ID);
```

The input parameter is the 32-bit sample index with the effect identifier (by default all identifiers are defined in **soundbank.h**).

Play music :

```
void mmStart(mm_word module_ID, mm_pmode mode);
```

The first input parameter is the module identifier. The second parameter specifies whether the music has to be played once (`MM_PLAY_ONCE`) or in an infinite loop (`MM_PLAY_LOOP`)

Pause, resume or stop music :

```
void mmPause();
void mmResume();
void mmStop();
```

Play sound effect :

```
mm_sfxhand mmEffect(mm_word sample_ID);
```

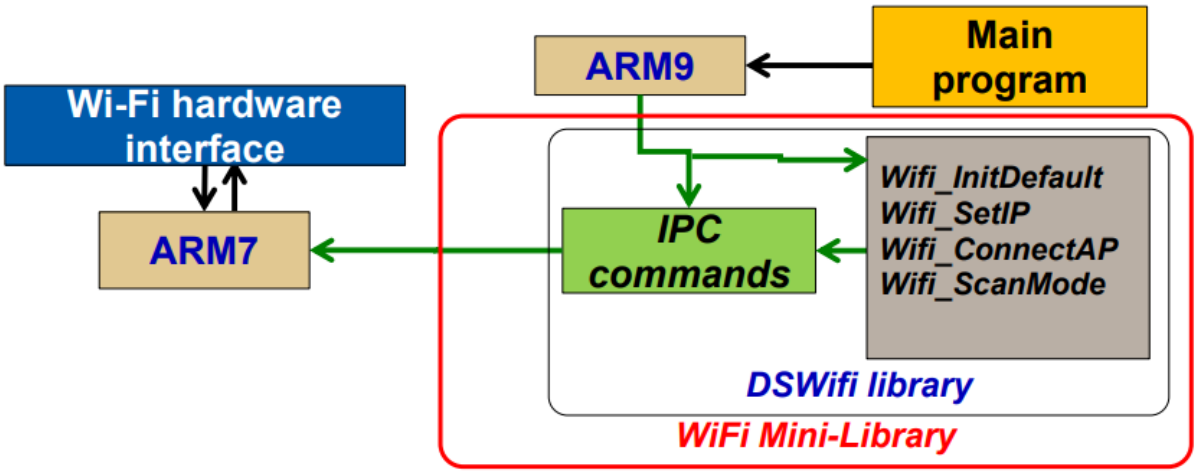
The input parameter specifies sound effect identifier of effect to play. The effect will be played without modifying the sound configuration.

Play sound effect with specific sound configuration :

```
mm_sfxhand mmEffectEx(mm_sound_effect* sound);
```

The input parameter is a structure with the effect identifier (id) and three main parameters: Volume, Panning, and Rate (frequency).

WiFi



WiFi Mini-Library

Simple library available in the Moodle Site for this course

It illustrates the use of a basic Wi-Fi connectivity functionality:

- Wi-Fi initialization / deactivation
  - Assuming access point is named "MES-NDS"
- Socket management and data transfer
  - Based in UDP datagrams (packets) broadcasting

TIP

You can use your smartphone as access point to connect your NDS devices. Just enable Mobile hotspot with:

- No password
- Hotspot name: "MES-NDS"

Wi-Fi Management

```
int initWiFi();
```

- Initializes the Wi-Fi library and tries to connect to the stored access point using the WFC
- Returns 1 on success and 0 otherwise
- The SSID of our access point is: "MES-NDS" (defined in "WiFi_minilib.h")

```
void disconnectFromWiFi();
```

- Closes the Wi-Fi connection by disassociating from the access point

Socket Management

```
int openSocket();
```

- Opens a bidirectional UDP channel in the local port 8888 (defined in the library)
- Returns 1 in case of success and 0 otherwise

```
void closeSocket();
```

- Closes the UDP bidirectional channel on port 8888 if it is opened

Data transmission

```
int receiveData(char* data_buff, int n);
```

- Opens Tries to read n bytes from the socket (if it is correctly opened) and writes them into the array `data_buff`
- Returns the number of bytes read or -1 if it failed

```
void sendData(char* data_buff, int bytes);
```

- Tries to send n bytes of data from the array `data_buff` through the bidirectional UDP channel
- Data is broadcasted and no confirmation is delivered by the sender

EXAMPLE

```
char msg[1];

consoleDemoInit();

// Initialize WiFi
if (initWiFi())
     printf("WiFi OK!\n");
else
    printf("Error WiFi\n");

// Open Socket
if (openSocket())
    printf("Socket OK!\n");
else
    printf("Error Socket\n");

while (1)       {
    // Poll the keypad
    scanKeys();
    unsigned short keys = keysDown();

    // If key A was pressed → send character "A" through WiFi
    if (keys == KEY_A) {
         msg[0] = "A";
         sendData(msg, 1);
    }

     // Listen to messages from other on WiFi
    if (receiveData(msg, 1) > 0) {
         if (msg[0] == "A")
             printf("Other pressed A\n");
    }
}
```

Acronyms

- NDS: Nintendo DS
- GBA: Game Boy Advance
- LCD: Liquid Crystal Display
- TFT: Thin-Film Transistor
- ARM: Advanced RISC Machine
- RISC: Reduced Instruction Set Computer
- CISC: Complex Instruction Set Computer
- CPU: Central Processing Unit
- ALU: Arithmetic Logic Unit
- PC: Program Counter
- CR: Control Register
- SR: State Register
- IR: Instruction Register
- MAR: Memory Address Register
- MDR: Memory Data Register
- I/O: Input/Output
- MM: Main Memory
- CM: Cache Memory
- SPM: Scratch-Pad Memory
- RAM: Random Access Memory
- VRAM: Video Random Access Memory
- WRAM: Window Random Access Memory

- REQ: REQuest
- FIQ: Fast Interrupt Request
- IRQ: Interrupt ReQuest
- FREQ: FREQuency
- SVC: SuperVisor Call
- INTR: INTeRrupt
- DMA: Direct Memory Access
- SPI: Serial Peripheral Interface
- ASCII: American Standard Code for Information Interchange
- GFX: Graphics
- FIFO: First-In, First-Out
- API: Application Programming Interface
- RGB: Red Green Blue
- ARGB: Alpha Red Green Blue
- BG: BackGround
- FB: FrameBuffer
- BMP: BitMap Pointer
- SWI: SoftWare Interrupt
- GRIT: GBA Raster Image Transmogrifier
- CNT/CTRL: CoNTRoLler
- A2D: Analog TO Digital
- ISR: Interrupt Service Routine
- IPC: Inter-Processor Communication

Acknowledgements

We want to sincerely thanks Ms. Sidonie Bouthors and Mr. Lucas Jung for sharing their notes with us! A big part of the original content they provided has been reused in this final document.

Also, we would like to thank all the students of the EE-310 course of the last years, who provided valuable feedback about the key elements to include in this final documentation.