LibNG

(Le new version de DATlib)

Rev 1.1

Table of contents

About LibNG

LibNG is a library designed to expand the SGDK environment to allow NeoGeo development.

Its goal is to provide easy functionality trough base elements (scroller, picture and animated sprite) which are prone to be used in software, and allow good performance while writing C software.

Tools are provided to process and imports data, and animation tool allows more detailed animations.

Embedded banking support now allows for games wiith larger data sets.

Concepts and preliminary notices

First of all, there is a quick demo program provided in the archive with source code, which you can explore and play along with to get familiar with the library and tools, or use as a stepping stone for your project. Template project can be copied when starting a blank new project.

LibNG will occupy about 10KB of the system ram with default settings.

The main outline of how this library works is that graphic updates are queued into buffers (also called draw lists / command buffers) that are processed during vblank. There is currently four command buffers: tiledata commands buffer (`SC1` buffer, VRAM 0x0000 – 0x6FFF operations), sprites control commands buffer (`SC234` buffer, VRAM 0x8000- 0x85FF operations), palette jobs commands buffer (`PALJOBS` buffer, palette ram operations) and fix jobs command buffer (`FIXJOBS` buffer, fix layer operations). This means you can -for example- update a sprite position anywhere in your code, it will automatically be synced with and updated during next vblank.

Many components of this library use the concept of base sprite and base palette:

Base sprite designates the starting sprite to use for said element. As an example a 4 tiles width picture with base sprite set to 10 will therefore use sprites #10 #11 #12 #13 to display.

Base palette is basically the same concept as base sprite, applied to color palettes.

It is up to the user to manage sprites and palettes to make sure no overlaps occurs across different elements.

It is of course recommended to be somewhat familiar with the NeoGeo system. All the information can be found on the NeoGeo dev wiki (https://wiki.neogeodev.org/) and the official NeoGeo developper manual (http://www.neogeodev.org/NG.pdf).

Installation & build

Requirements: main SGDK package is required, make sure it is correctly installed and set up.

To install library, simply drop the files from the package into your SGDK folder. Building the library from the SGDK main folder:

```
make -f makelib.neo
```

NeoGeo projects can then be started by adding <neogeo.h> to the main file, and compiling them with:

```
make -f makefile.neo
```

Notes:

- First build will import `boot/neogeo.s` and `boot/neogeo_boot.s` files into the project folder, `neogeo.s` file can then be edited with setting for the current project (game ID, name, misc options…)
- There is no `main` function. `USER`, `PLAYER_START`, `DEMO_END` and `COIN_SOUND` functions must be provided. See provided template project.

Features

Entry points

NeoGeo program flow is a bit more complex than the average retro system, user needs to prvide 4 main functions for the bios to call:

- USER
- PLAYER_START
- DEMO_END
- COIN_SOUND

Please check the NEOGEO documentation and/or provided “NGtemplate” sample program for correct setup.

Exposed BIOS & hardware

Most bios data is now properly mapped and exposed to user through the BIOS structure, as well as some of the hardware registers via LSPC structure. This allows simpler code and better IDE integration. Check the NeoGeo dev wiki for detailed info on bios and hardware registers.

```
if(BIOS.P1.EDGE.A) // P1 button A positive edge?
{
	BIOS.CARD.COMMAND = CARDCMD_FORMAT;
	biosCall(BIOS_SUB_CARD);
}
...
LSPC.MODULO = 0x20;
```

Program loop

Using the library requires using a defined program flow for everything to work together.
While there are many ways to arrange code and use functionalities, here is a sample, basic program loop:

```
initGfx(); //initialize library components

/* initialize scroller, pictures etc… */

while(1) {
	SCClose();	//close Sprite Control lists
	waitVBlank();	// wait vblank
			// screen updates will occur during vblank

	/* do stuff */

}
```

Banking

Support for banking is provided by the library and toolchain, data can be arranged in various with provided tools (see buildchar tool section). Banking is automated in provided library functions, see `setBank` function for manual setting.
Base linker file comes with main ROM region and two data banks enabled. User can add or remove banks in the linker file as needed.

Bank # is encoded in addresses upper byte, IE address 0x01200400 designates data at offset 0x400 in PORT region (0x200000), with bank #1 set.

Note: animated sprite (`aSprite`) data is limited to banks 0-127.

Provided graphics types

LibNG provides three base graphical elements that should fulfill most needs:

picture

- simple picture type
- allows display, positioning and flipping of static pictures
- when setting picture position, you are setting top left pixel position
- uses picture tile width sprites (ie: 64px width picture = 4 tiles width = 4 used sprites)

scroller

- type used to display a scrolling plane
- 8 way scrolling ability
- no map size limit
- uses 21 sprites, regardless of plane dimensions
- can jump anywhere from any position in one frame

animatedSprite

- provides support for animated sprites
- allow display, positioning, flipping and animating sprites
- animation system supports repeats and animation linking
- up to 65536 animations, 65536 animation steps
- allocated mode / sprite pool mode
- used sprites depends on currently displayed frame. If using allocated mode, good practice is to plan enough sprites to fit the widest frame

Note: Animated sprites uses a different way to position themselves. Each frame location is relative to a fixed reference point. This is due to the nature of animations, often using a set of frames of different sizes and alignments (to avoid encasing a few pixels in a large picture frame, saving space and CPU time). Positioning operations on animated sprites refer to positioning the reference point (anchor point). Flipping animated sprites is done around the reference point. It is possible to revert back to a more classic cpprdinates system by using the strict coordinates flag.

Vblank handlers

Vblank handlers are interrupt handlers provided by the library, required for proper operation. Those have to be set up as your vertical interrupt (IRQ2) vector.

libNG_vblank

Standard vblank handler.

Operation:

- sets job meter to red
- process tiledata buffer
- process sprites control buffer
- process palette jobs buffer
- process fix jobs buffer
- sets job meter to orange
- resets draw lists, updates frame counter
- call `MESS_OUT` and `SYSTEM_IO`
- checks and process debug dips
- acknowledges IRQ, kicks watchdog, calls `SYSTEM_IO` (BIOS)
- sets job meter to green
- returns

Note: Job meter colors are only updated under select circumstances, see debug dips section.

libNG_vblankTI

Vblank handler with timer interrupt support.

Operation:

- Setup timer IRQ timings and data for next frame
- Branch to `libNG_vblank` for standard operations

Notes: When using timer interrupts, requested LSPC mode must be written to the `LSPCmode` variable (u16). This is due to the LSPC mode hardware register being manipulated to set timer values, therefore needing a reference value of requested settings to preserve them. If using standard vblank handler, the `LSPCmode` variable will be ignored and therefore you must write directly to the register when needed.

When using timer interrupts, user must use IRQ safe versions of functions when available. Thoses are slightly slower than the regular ones but are required to avoid VRAM corruption by interrupts.

Timer interrupt

Base functionality is provided for timer interrupts, allowing to change one or two VRAM value on every (or select set of) scanline.

To enable timer interrupt functionalities:

- set `libNG_vblankTI` as your vblank IRQ vector
- set `libNG_TIfunc` as your timer IRQ vector

Notes: make sure you set variable `TinextTable` (u32) to 0 before enabling IRQ when using timer interrupt. This is done in the default init code, but make sure to keep it if customizing files. Timer interrupt related code uses the USP register, make sure you code doesn't conflict.

Using timer interrupt:

To work with timer interrupt you need to prepare data in a WORD table, storing VRAM address and data combos.

Format for the data table is:

- VRAM address n (1x u16)
- VRAM data n (1x u16)
- VRAM address n+1 (1x u16)
- VRAM data n+1 (1x u16)
- (etc…)
- end marker (2x u16, 0x0000 0x0000)

For correct behavior it is required to use two alternating tables. One table for currently displaying frame, another one to prepare data for next frame.

Timer IRQ function must be set up with `loadTIirq()` prior to use.

Timer IRQ is available for single and dual data writes for each triggering. See `loadTIirq()` section.

Startup timer interrupt:

- set base and reload timers
- put pointer to data table for next frame in the `TinextTable` variable

Stop timer interrupt:

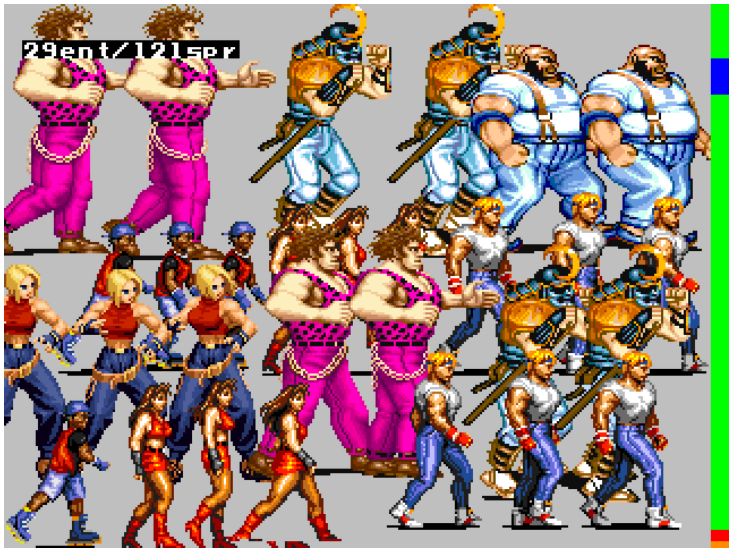
- set `TInextTable` to null (0)

Notes: When data last value is processed, the timer interrupt will be disabled for the rest of the frame until next vblank. This avoids triggering unnecessary IRQ, as they are CPU consuming. Default timer values are provided for first raster line triggering and each line repeat: `TI_ZERO` and `TI_RELOAD`. Timer interrupt will be disabled if TinextTable is null. Timer interrupt will be disabled if first table entry is end marker.

Job meter

Base job meter support is provided by the library.

Job meter allows basic profiling of your code, by having a visual representation of how much CPU time is used. Using different colors lets you observe CPU usage of every procedure, allowing targeting of things to optimize.



Job meter example:

Green color: free CPU time
Blue color: animation procedures
Red color: vblank sprites updates
Orange color: post vblank `SYSTEM_IO`

Note: Setting job meter colors during active display will issue a pixel of said color on screen (on real hardware). This is an issue with the hardware that can't be avoided, therefore make sure to use job meter in debug builds only.

Debug dips

Some of LibNG features are enabled through debug dips. Enable dev mode into bios then set the requested dips to 1.

- Debug dip 2-1
  Enable vblank job meter color updates.
  Vblank interrupt will color draw buffers processing as red job, and post jobs like `SYSTEM_IO` in orange.

- Debug dip 2-2
  Displays current raster line # when draw buffers are done being processed.

- Debug dip 2-3
  Displays a rough usage meter for `SC1` and `SC234`, `FIXJOBS` and `PALJOBS` buffers

- Debug dip 2-4 ~2-8
  Unused / reserved future use / free to use by user.

Sprite Pools

Sprite pools are an alternate way to handle sprites rendering. It consists of a reserved sprites batch which is then used to display assets.
This technique is reminescent of double buffering, but using sprites.

It differs from the previous basic, “allocated” draw mode by many ways:

- Sprite tilemap/position data is written during active frame, alleviating vblank load
- Sprite tilemap/position data is fully rewritten every frame
- Removes the need to manage baseSprite from **aSprite** handles, they are drawn in the order they are submitted
- Submit order drawing allows for easier sprites sorting/priority change
- No baseSprite management means less sprite loss, when current frame is smaller than the maximum reserved space

Base operation sketch

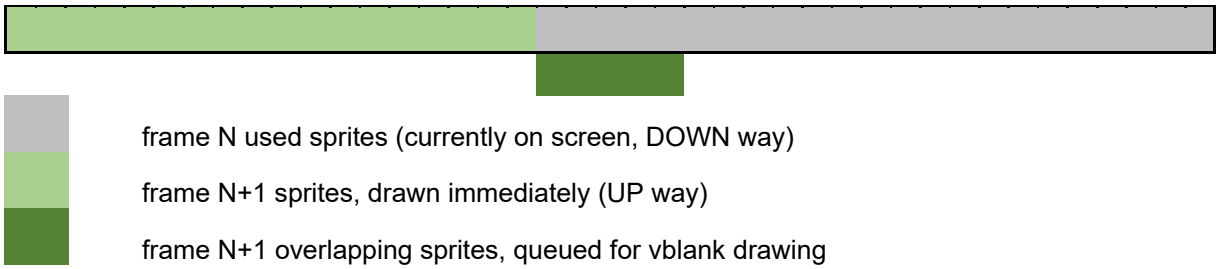
A spritePool entity must be initialized providing a pool size (# of sprites) and a starting position for this pool (baseSprite). Pool size should be aimed at twice the size of an average scene. If an average frame requires 80 sprites, ideally allocate 160.

To draw into the sprite pool, user must submit an array of pointers to **aSprite** handles, followed by a null pointer end marker.

Drawing in the sprite pool alternates way every frame (`WAY_UP`/`WAY_DOWN`). When going UP, pool uses sprites from pool start toward pool end, when going DOWN, from pool end toward pool start. User must supply the top or bottom end of the pointer array, to fit needs.

Tilemap and X position data is written into vram during active display, Y position is updated during vblank.

In case of heavy load, it is possible the sprite needs overlaps with the currently used sprites from previous frame. In this case overlapping sprite needs are queued for update during next vblank:



This provides a failsafe and user transparent operation in most scenarios, however exceeding the total pool side will lead to unexpected results and adjacent sprites corruption.

Vblank callbacks

Vblank callback functions are available by using the supplied pointers:

- `VBL_callBack`: callback pointer to function to be called after a regular Vblank
- `VBL_skipCallBack`: callback pointer to function to be called after a skipped frame Vblank

Callback functions are called at the very end of the Vblank interrupt procedure and after `SYSTEM_IO` occurred.

As all registers (except for A7) are restored when the callback function returns, user can trash them without caring about saving them.

Color streams

When creating a large background plane, an issue can arise with color palettes being too numerous to fit withing the available ressources.

Color streams are provided as a solution, allowing the streaming color data into palette RAM as scrolling advances.

When requesting a color stream from buildchar, orientation must be specified (horizontal/verttical) to indicate scan orientation. Scanning along the largest axis will usually provide the best results. IE a “landscape” orientation scroller should be using horizontal parameter.

When initializing a color stream, user can choose to load the start or end configuration, matching palettes state at start or end of scroller. It is advised to initialize streams with the configuration matching the scroller position the closest.

Once initialized user can request streams to advance to select position, required palette jobs will be buffered and processed on next Vblank.

Note: Be wary of large jumps in scrollers when using color streams, as it could induce a lot of palettes shuffling and possibly overflows the available palette jobs buffer.

Tools

Buildchar (character ROM)

Command line tool used to convert your graphics elements into tiles, tilemaps and palettes.

Input

- `chardata.xml`

  Contains description of assets to include into tile data.

  Example `chardata.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<chardata>
   <setup>
         <starting_tile fillmode="dummy">256</starting_tile>
   </setup>

   <scrl id="ffbg_b">
         <file>gfx\ffbg_b0.png</file>
         <auto1>gfx\ffbg_b1.png</auto1>
         <auto2>gfx\ffbg_b2.png</auto2>
         <auto3>gfx\ffbg_b3.png</auto3>
   </scrl>

   <pict id="ffbg_c" flips=”xyz”>
         <file>gfx\ffbg_c.png</file>
   </pict>

   <sprt id="bmary_spr" flips=”xyz”>
         <file>gfx\bmary.png</file>
         <frame>0,0:4,7</frame>
         <frame>4,0:4,7</frame>
         <frame>8,0:4,7</frame>
         <frame>12,0:4,7</frame>
         <frame>16,0:4,7</frame>
         <frame>20,0:4,7</frame>
         <frame>24,0:4,7</frame>
         <frame>28,0:4,7</frame>
         <frame>32,0:4,7</frame>
         <frame>36,0:4,7</frame>
         <frame>40,0:4,7</frame>
         <frame>44,0:4,7</frame>
   </sprt>
</chardata>
```

Nodes details:

- `<setup>`
  Contains general settings:
  - `<starting_tile>`
    Defines starting tile # (decimal). Used to leave blank tiles at the beginning of the char.bin file, useful if you need room to fit things like a character font at the beginning of the tileset. Additional parameter fillmode (none/dummy) defines if skipped tiles are to be filled or not.
  - `<charfile>`
    Defines output character file name. Optional, defaults to “char.bin”.
  - `<mapfile>`
    Defines output tilemaps data file name. Optional, defaults to “maps.s”.
  - `<palfile>`
    Defines output palettes data file name. Optional, defaults to “palettes.s”.
    It’s possible to use same name as `mapfile`, to merge data in the same file.
  - `<incfile>`
    Defines output include file name. Optional, defaults to “externs.h”.
  - `<incprefix>`
    Include prefix to add to include paths

- `<import>`
  Used to import binary data. Will copy the raw binary data into the output file.
  File size must be multiples of 128 bytes (single tile size).
  - `<file>`
    Binary file to import.

- `<scrl>`
  Used to declare a scroller
  - id (attribute)
    Literal name the scroller will be referenced by in C code.
  - section (attribute)
    Rom section to put data in. Defaults to `.text` if unspecified.
  - colorStream (attribute)
    Set this attribute to “horizontal” or “vertical” value to generate **colorStream** data for this scroller.
  - `<file>`
    PNG file of the display area.
  - `<auto1> to <auto7>`
    Additional pictures when using auto animation features.

- `<pict>`
  Used to declare a picture
  - id (attribute)
    Literal name the picture will be referenced by in C code.
  - section (attribute)
    Rom section to put data in. Defaults to `.text` if unspecified.
  - flips (attribute)
    Flip modes wanted for this picture (optional).
    X = horizontal flip
    Y = vertical flip
    Z = horizontal & vertical flip
  - `<file>`
    PNG file of the picture.

- `<sprt>`
  Used to define an animated sprite
  - id (attribute)
    Literal name the animated sprite will be referenced by in C code.
  - section (attribute)
    Rom section to put data in. Defaults to `.text` if unspecified.
  - flips (attribute)
    Flip modes wanted for this picture (optional).
    X = horizontal flip
    Y = vertical flip
    Z = horizontal & vertical flip
  - pal (attribute)
    Option to define how palette data is generated
    scan (default): automatic picture scan, no control over palettes / color indexes
    strips: will scan top left corner of picture for 16 color strips, allow color index palette & inclusion of color swaps
    keep: do not generate palette data, keep previous object data. Use if multiple objects share the same palette
  - opts (attribute)
    Misc advanced options, mostly useful if using a custom animation system and want to exclude some data. Can use multiple, separated by ‘,’
    noflips: don’t generate pointers for flips data
    noanim: don’t generate anim pointer & file include
    globalframes: generate global symbol for each frame
    forcesplit: force split frame format for all frames
  - `<file>`
    PNG file containing all animation frames.
  - `<frame>`
    Defines a frame, format is: top,left coordinate:width,height
    See Framer tool section to easily set up frames

About input files format:
Picture files used in chardata.xml must be PNG format, 32bppArgb. Define transparency by fuschia color (#ff00ff), or simply use transparency. Size must be multiples of 16.

About colors:
There is no limits color wise, as long as each tile is transparency + 15 colors max, you can use pics with hundreds of colors.
If your file is rejected for using too many colors per tile, erroneous tiles will be shown in a reject.png file.

About ID:
Each declared entity will generate an extern C object named <id>, as well as a palettes object named <id>_Palettes.

Output

- `char.bin`
  Your tile data, linear binary output.
  Convert to cart or CD format if needed by using the CharSplit tool.

- `maps.s`
  Tilemaps data, add to makefile to compile and link into your project.

- `palettes.s`

Palettes data, add to makefile to compile and link into your project.

- `externs.h`
  Extern definitions of your data. Include into your C program to use data.

Mixing auto4 and auto8 tiles

It is possible to mix up auto4 and auto8 tiles on the same file when using auto animation. To do so, use the supplied auto4 marker tile (auto4_tile.png) on your <auto4> file to designate an auto4 tile.

Tile distribution across mixed up files is as follow:

	<file>	<auto1>	<auto2>	<auto3>	<auto4>	<auto5>	<auto6>	<auto7>
Auto4	Tile #0	Tile #1	Tile #2	Tile #3	End marker	---- Ignored data ----		
Auto8	Tile #0	Tile #1	Tile #2	Tile #3	Tile #4	Tile #5	Tile #6	Tile #7

Note: Large data sets can become time consuming to process when new data is added. It is advised to split data across multiple xml files, using the `starting_tile` tile parameter to reserve areas for each file. That way, when adding data only one xml file needs to be reprocessed.

Buildchar (fix ROM)

Buildchar can also be used to generate FIX ROM character data.
Use the `fileType="fix"` tag inside the setup node to specify a FIX ROM file.

Fix data is split into 16 “banks” of 256 characters.
Input pictures must be sets of 256 characters forming a 128px*128px area bank.
Picture can contain multiple character sets, however layout must remain 128px height and 128px multiples width.

You can load multiple pictures in the same bank. As they will be merged together, make sure data doesn’t overlap.

Please note buildchar doesn’t optimize fix character data, as characters location is very often a programmer’s choice (fonts needing to be at set position, specific health bar setup, etc…).
In the same manner, there is no tilemap data output. You have to write the data fitting your needs (see 16bit strings format and `fixJobPut`, or the bios `MESS_OUT` function).

Input

- `fixdata.xml`

  Contains description of assets to include into fix data.

  Example `fixdatadata.xml` file:

```
<?xml version="1.0" encoding="UTF-8" ?>
<chardata>
   <setup fileType="fix">
         <charfile>out\fix.bin</charfile>
         <palfile>out\fixPals.s</palfile>
         <incfile>out\fixData.h</incfile>
   </setup>

   <import bank="0">
         <file>gfx\fix\systemFont.bin</file>
   </import>

   <fix bank="3" id="fix_font">
         <file>gfx\fix\font0.png</file>
   </fix>
</chardata>
```

Nodes details:

- `<setup>`
  Contains general settings:
  - `<charfile>`
    Defines output character file name. Optional, defaults to “char.bin”.
  - `<palfile>`
    Defines output palettes data file name. Optional, defaults to “palettes.s”.
    It’s possible to use same name as `mapfile`, to merge data in the same file.
  - `<incfile>`
    Defines output include file name. Optional, defaults to “externs.h”.
- `<import>`
  Used to import binary data. Will copy the raw binary data into the output file.
  This can be used to import a standard system font.
  File size must be multiples of 32 bytes (single character size).
  - Bank (attribute)
    Destination bank #.
  - `<file>`
    Binary file to import.
- `<fix>`
  Import a characters set
  - Bank (attribute)
    Destination bank # in the fix file.
  - id (attribute)
    Literal name for the palette data that will be referenced by in C code.
  - `<file>`
    PNG file of the characters data.

Charsplit

Command line tool used to convert raw character data issued by buildchar to either cart or CD format files.

Usage:

```
charSplit [input_file] <options> [output_file_prefix]

Options:
        -cart   Ouput to cart format ([output_file_prefix].c1 & .c2)
        -cd     Output to CD format ([output_file_prefix].cd)
```

Example:

```
charsplit char.bin –cart game
```

will split `char.bin` into `game.c1` and `game.c2` files for cart system use.

Note: raw files can be loaded directly into mame

```
<dataarea name="sprites" size="0x400000">

      <rom name="char.bin" offset="0x000000" size="0x400000" crc="12345678"
sha1="1234567890123456789012345678901234567890" />

</dataarea>
```

It is therefore unnecessary to use charsplit to format character data while in the testing / debugging phase.

Framer

Tool used to outline frames for an animated sprite object.
Each animated sprite must be assigned a set of frames before being processed by the buildchar tool.

A frame, or “metasprite” can be of various sizes and is made of various amount of actual hardware sprites.

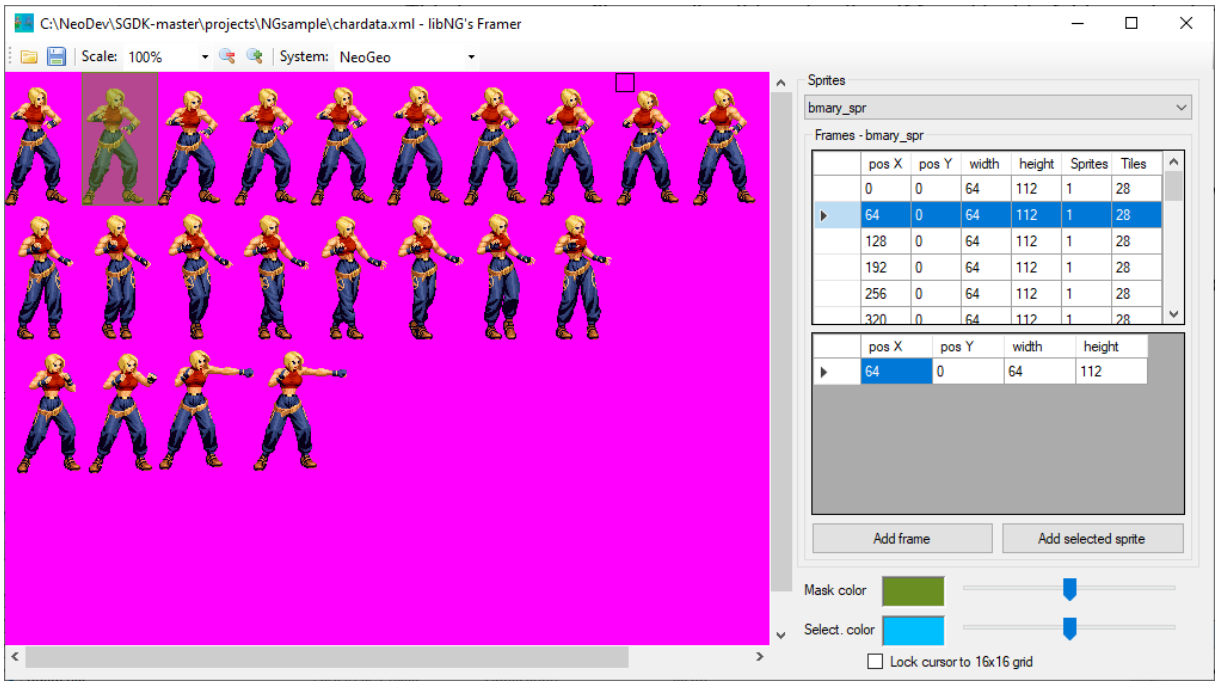
Input

- `chardata.xml`
  Click the open button and select the xml file containing reference to your animated sprites assets.

Output

- `chardata.xml`
  Click the save button to update xml file with the new/updated frames.

Usage:



Framer is very straightforward to use, open your xml file, then select the animated sprite object you want to work with from the drop down menu.
If the xml file already contains data, existing frames will be listed to be updated/removed.
To add a new frame, click the [Add frame] button (n key). To add sprites to newly selected frame, simply select shape with mouse and press the [Add selected sprite] button (space key).
When done, click save to update the xml file, which is then ready to use with buildchar for processing.

Frame formats:

libNG supports two frame formats, plain and split:

Plain frame is the classic rectangular shape, it fits most scenarios and scaling can be applied to thoses. Selecting a single rectangular shape will generate a plain frame.

Split frame allows more precise shapes by ass embling unaligned sprites together. This is useful for saving per-line sprite budget, however scaling cannot be applied to split frames. Adding two or mode hardware sprites with framer tool will automatically generate a split frame.

Plain vs split frame, split is 2 tiles narrower for most of the frame.

Frame format is decided by the buildchar tool when processing data, and is completely transparent to the user. Both formats can be mixed up in the same object at will.

Animator

Tool used to animate animated sprites.
Each animated sprite object processed by the buildchar tool must be assigned at least one animation with the animator tool for proper compilation and linking of your project.

Input

When defining an animated sprite, buildchar will output a subfolder containing frames cutouts. Open this folder in Animator.

- `animdata.xml`
  This is your save file regarding this animation. If found inside folder, animator will load it.

Output

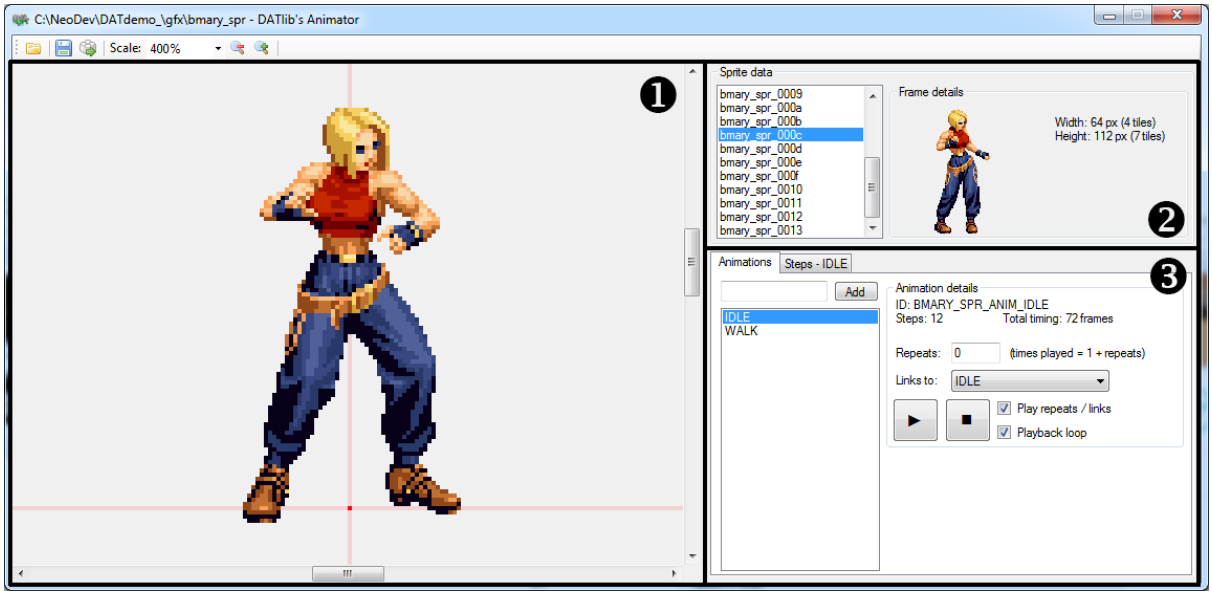
- `animdata.xml`
  This is your save file regarding this animation. Hit save button to save your work.

- `<id>_anims.s`
  Animations file, auto included in the `maps.s` file generated by the buildchar tool.

- `<id>.h`
  Contains animations C defines, auto included in the `externs.h` file generated by the buildchar tool.

Using Animator:

Main window is divided into 3 areas



❶ Preview area

This area allows you to visually align frames and preview animations. Change scale for better viewing. Reference point is visualized by the intersection of the two red axes. When setting position of animated sprites in your code, you are setting the position of this reference point.

❷ Sprite and frames data area

This area will list and provide a quick preview of all the frames you created using the framer and buildchar tools, making sure you exported them correctly.

❸ Animations area

This is the main section to edit and check animations

Adding an animation: Input animation name in text field and press the [Add] button, the new animation will appear in the animations list.

Edition an animation: Select the animation you want to edit in the animations list. Input repeat count and link data for selected animation. Repeats are the number of times the animation will be repeated after initial play. Link allows you to branch to another animation once current animation is done displaying (including repeats). You can link an animation to itself to create a loop. If no link is selected the last animation frame will remain of screen after animation is done.

From there on, double click on frames in frames list to add animation steps.
You will have to input frame position for each step (X & Y field, or arrow buttons) as well as step timing (T field). Timing is the number of display frames the selected step remains on screen. Mod all steps checkbox will allow you to edit all steps at the same time.
You can adjust steps order or delete steps by using buttons under the steps list.

Animations IDs

Each animation created with the animator tool will generate a C define that can therefore be used when setting animations.

Format is `<id>_ANIM_<animation name>` (all uppercase).
As an example, building animations named WALK and IDLE for animated sprite defined with ID “mycharacter” will generate `MYCHARACTER_ANIM_WALK` and `MYCHARACTER_ANIM_IDLE` defines.

Exporting data

Use the [Export] button in the toolbar to export animation data into your project for compilation/linking.

Keyboard shortcuts

Shift + arrow keys: move frame of currently selected step
Space bar: start/stop current animation playback

MAME setup

Some mame edits are recommanded for easier software testing / debugging. Mame compiling guide can be found at https://docs.mamedev.org/initialsetup/compilingmame.html.

Loading ROM

Add project ROM information into mame `hash/neogeo.xml`, code rom can be directly loaded without byteswapping and or splitting file:

```
<dataarea name="maincpu" width="16" endianness="big" size="0x300000">
	<rom name="rom.bin" offset="0x000000" size="0x300000" crc="0" sha1="0 />
</dataarea>
```

Buildchar output file can also be loaded directly without additional formatting:

```
<dataarea name="sprites" size="0x400000">
	<rom name="char.bin" offset="0x000000" size="0x400000" crc="0" sha1="0" />
</dataarea>
```

This allows faster iterations with less file operations during development.

Expanding screen

This modification is recommanded to have a glimpse of operations during vblank.

Standard view vs expanded view, vblank load now partly visible in bottom right corner

Screen definition can be found in `src/mame/neogeo/neogeo.cpp`, to expand 8 px both top and bottom, edit to:

```
m_screen->set_raw(NEOGEO_PIXEL_CLOCK, NEOGEO_HTOTAL, NEOGEO_HBEND, NEOGEO_HBSTART,
NEOGEO_VTOTAL, NEOGEO_VBEND - 8, NEOGEO_VBSTART + 8);
```

Message output

Another modification is to add message printing function to mame, so software can output messages through console or debugger log window.

Default library settings use bios area as write buffers, to avoid conflicts on real hardware (read only area):

```
	sprintf2(MAME_PRINT_BUFFER, "Console message\n");
	sprintf2(MAME_LOG_BUFFER, "Log message\n");
```

NeoGeo memory map must be edited in `src/mame/neogeo/neogeo.cpp`:

```
map(0xc00000, 0xc1ffff).mirror(0x0e0000).rw(FUNC(ngarcade_base_state::bios_r), FUNC(ngarcade_base_state::bios_w));
```

In the same file, custom handlers must also be added:

```
void ngarcade_base_state::bios_w(offs_t offset, uint16_t data, uint16_t mem_mask)
{
	static char printMsg[256], logMsg[256];
	static char *printPtr = printMsg;
	static char *logPtr = logMsg;

	char c = (mem_mask == 0xff ? data : (data >> 8));

	if(offset < 0x10000 / 2)
	{
		if(!(*printPtr++ = c))
			printf("%s", printPtr = printMsg);
	}
	else
	{
		if(!(*logPtr++ = c))
			logerror("%s", logPtr = logMsg);
	}
}

uint16_t aes_state::bios_r(offs_t offset)
{
	uint16_t* bios = (uint16_t*)m_region_mainbios->base();
	return bios[offset];
}
```

Library reference

Command buffers formats

SC1 buffer

Each `SC1` buffer entry is two u32:

31-24	23-15	15-0
Palette mod	Tile size x2 *	VRAM address
31-0		
Tile data address		

Buffer has a `0x00000000` end marker.

* When used to draw a scaled hardware sprite, use (32+Tile size)x2

SC234 buffer

Each `SC234` buffer entry is two u16:

15-0
VRAM address
15-0
VRAM data

Buffer has no end marker (size is computed from `SC234ptr` value).

PALJOBS buffer

Each `PALJOBS` buffer entry is two u32:

31-16	15-5	4-0
Palettes count-1	Palette number	00000
31-0		
Palette data address		

Buffer has a `0xffffffff` end marker.

FIXJOBS buffer

Each `FIXJOBS` buffer entry is two u32:

31-16	15-12	11-8	7-0
VRAM address	Palette #	0000	VRAM modulo
31-0			
FIX data address			

Buffer has a `0x00000000` end marker.

Library defines

A large set of defines is provided for library / system operation, please check include files.
Build options can be changed in `configNG.h` file.

Library variables

libNG will take some memory space for operation, about 10KB of RAM will be taken when using default buffers size settings. Scratchpads are internally used by text formatting / printing functions but can be reused outside of thoses functions if needed.

General variables

u32	libNG_frameCounter	frame counter
u32	libNG_droppedFrames	dropped (skipped) frames counter
u32	*VBL_callBack	VBlank callback function pointer
u32	*VBL_skipCallBack	VBlank callback function pointer (skipped frame)
u32	SC1[SC1_BUFFER_SIZE]	draw list for tilemap data
u32	*SC1ptr	pointer to tilemaps data draw list
u16	SC234[SC234_BUFFER_SIZE]	draw list for sprite control
u16	*SC234ptr	pointer to sprites control draw list
u32	PALJOBS[PAL_BUFFER_SIZE]	palette jobs buffer
u32	*palJobsPtr	pointer to palettes jobs buffer
u32	FIXJOBS[FIX_BUFFER_SIZE]	fix jobs buffer
u32	*fixJobsPtr	pointer to fix jobs buffer
char	libNG_scratchpad64[64]	64 bytes scratchpad
char	libNG_scratchpad16[16]	16 bytes scratchpad

Timer interrupt related variables

_LSPCMODE_W	LSPCmode	requested LSPC mode
u32	TIbase	timer interrupt timing to first trigger
u32	TIreload	timer interrupt reload timing
u16	*TInextTable	pointer to data table to use next frame

Sound buffer variables (internal use)

u8	sndBuffer[SOUNDBUFFER_SIZE];	command buffer
u16	sndBufferIndexRW;	command buffer R & W indexes

Color math variables

palette	palBuffer[256]	color buffer
palHandle	palHandles[256]	palette handles
u8	palTransferPending	set to 1 when color data is waiting for transfer
u8	palCommandPending	set to 0 when all fade commands are over
u8	palFlushOnFrameSkip	set to 1 to enable color transfer on frameskips

General purpose components

MEMBYTE, MEMWORD, MEMDWORD

Direct memory access macros.

Syntax

MEMBYTE(*address***)**
MEMWORD(*address***)**
MEMDWORD(*address***)**

Explanation

Macros that can be used to directly access a memory address or hardware register.
Available for 8, 16 and 32 bits operation.

Ex:

```
i=MEMWORD(0x3c0006);      /* reads LSPC mode register into i */

MEMBYTE(0x300001)=1;      /* kicks watchdog */
```

Note: 68000 requires even addresses when operating on word (16bit) and dword (32bit) data.
Read/write operation at an odd address for a word/long will crash the CPU.

Return value

N/A

volMEMBYTE, volMEMWORD, volMEMDWORD

Direct memory access macros, volatile declaration.

Syntax

volMEMBYTE(*address***)**
volMEMWORD(*address***)**
volMEMDWORD(*address***)**

Explanation

Macros that can be used to directly access a memory address or hardware register.
Available for 8, 16 and 32 bits operation.
Theses macros are defined with the volatile keyword.

Ex:

```
i=volMEMWORD(0x3c0006);   /* reads LSPC mode register into i */

volMEMWORD(0x300001)=1;   /* kicks watchdog */
```

Note: 68000 requires even addresses when operating on word (16bit) and dword (32bit) data.
Read/write operation at an odd address for a word/long will crash the CPU.

Return value

N/A

PRINTINFO, PRINTINFO_W

Print data formatting macros.

Syntax

PRINTINFO(

U8 *posX,*	Fix plane X pos
U8 *posY,*	Fix plane Y pos
U8 *pal,*	Palette #
U8 *bank***)**	Tile data bank/page

PRINTINFO_W(

U8 *posX,*	Fix plane X pos
U8 *posY,*	Fix plane Y pos
U8 *pal,*	Palette #
U8 *bank***)**	Tile data bank/page *address***)**

Explanation

Formats print parameters to provide print functions with sortened arguments count.

PRINTINFO is to be used for function handling 8bit string data, PRINTINFO_W is to be used with functions handling 16bit string data.

Return value

N/A

VRAM_SPR_ADDR, VRAM_FIX_ADDR, VRAM_SHRINK_ADDR, VRAM_SHRINK, VRAM_POSY_ADDR, VRAM_POSY, VRAM_POSX_ADDR, VRAM_POSX

Misc macros for VRAM address calculations and data formating.

Syntax

VRAM_SPR_ADDR1(*sprite_number***)**	Sprite tilemap address
VRAM_FIX_ADDR(*X_position, Y_position***)**	Fix address for character at posion x,y
VRAM_SHRINK_ADDR(*sprite_number***)**	Sprite shrink coefficient address
VRAM_SHRINK(*H_shrink, V_shrink***)**	Sprite shrink values
VRAM_POSY_ADDR(*sprite_number***)**	Sprite Y position address
VRAM_POSY(*Y_position, link, sprite_size***)**	Sprite Y position value
VRAM_POSX_ADDR(*sprite_number***)**	Sprite X position address
VRAM_POSX(*X_position***)**	Sprite X position value

Explanation

Eases up syntax when handling VRAM addresses and values.

Related defines (Y position link value):

```
SPR_LINK            (0x0040)
SPR_UNLINK          (0x0000)
```

Ex: Moving sprite #16 to X position 120:

```
SC234Put(VRAM_POSX_ADDR(16), VRAM_POSX(120));
```

Return value

N/A

fixJobPut

Writes a command into fix jobs buffer. Macro.

Syntax

fixJobPut(

u16 *x,*	Target X position on fix layer
u16 *y,*	Target Y position on fix layer
u16 *mod,*	VRAM modulo
u16 *pal,*	Base palette
u16 **data* **)**	Pointer to fix data

Explanation

Macro allowing user to put a fix job into fix jobs buffer.

Return value

N/A

palJobPut

Writes a command into palette jobs buffer. Macro.

Syntax

palJobPut(

u16 *number,*	Destination palette number (0-255)
u16 *count,*	Number of palettes to write
u16 **data* **)**	Pointer to palette data start

Explanation

Macro allowing user to put a palette job into palette jobs buffer.

Return value

N/A

SC1Put

Writes a command into tilemap data draw buffer. Macro.

Syntax

SC1Put(

u16 *addr,*	Destination address in VRAM
u16 *size,*	Tile count
u16 *pal,*	Base palette
u16 **data* **)**	Pointer to tilemap data

Explanation

Macro allowing user to put a tilemap command into tilemap data draw buffer (VRAM sprite control block 1).

Maximum valid size is 32 tiles.

Return value

N/A

SC234Put

Writes to the sprite control draw buffer. Macro.

Syntax

SC234Put(

u16 *addr,*	Destination address in VRAM
u16 *data* **)**	Data

Explanation

Macro allowing user writes into the sprite control draw buffer (VRAM sprite control blocks 2 3 & 4).

Whilst designed for sprite control, the usage can be expanded to write any u16 data to any VRAM address.

Return value

N/A

addMessage

Queue a message to the BIOS MESS buffer.

Syntax

void addMessage(
u16 **message* **)**

Explanation

Add a message to the messages list, `MESS_OUT` is called during vblank to output pending messages. See NeoGeo dev wiki / programmer's manual for messages format. Supplied NGsample project also provide base mesasge macros / examples in the `messData.s` file.

Note: message system does not supoport banking, place messages data in main rom region

Return value

N/A

biosCall, biosCall2

Calls a BIOS function.

Syntax

void biosCall(
    u32 sub **)**

void biosCall2 (
    u32 *sub*,
    u32 *arg* **)**

Explanation

Protected call to designated BIOS function.
Registers are saved prior to call then restored up on return.

Most calls defines are provided:

```
BIOS_SUB_CREDIT_CHECK     (0xc00450)
BIOS_SUB_CREDIT_DOWN      (0xc00456)
BIOS_SUB_READ_CALENDAR    (0xc0045c)
BIOS_SUB_SETUP_CALENDAR   (0xc00462)
BIOS_SUB_CARD             (0xc00468)
BIOS_SUB_CARD_ERROR       (0xc0046e)
BIOS_SUB_HOW_TO_PLAY      (0xc00474)
BIOS_SUB_FIX_CLEAR        (0xc004c2)
BIOS_SUB_LSP_1ST          (0xc004c8)
BIOS_SUB_MESS_OUT         (0xc004ce)
BIOS_SUB_CONTROLLER_SETUP (0xc004d4)
// CD BIOS calls
CDBIOS_SUB_UPLOAD         (0xc00546)
CDBIOS_SUB_LOADFILE       (0xc00552)
CDBIOS_SUB_CDPLAYER       (0xc0055e)
CDBIOS_SUB_LOADFILEX      (0xc00564)
CDBIOS_SUB_CDDACMD        (0xc0056a)
CDBIOS_SUB_VIDEOEN        (0xc00570)
CDBIOS_SUB_PUSHCDOP       (0xc00576)
CDBIOS_SUB_SETCDDMODE     (0xc0057c)
CDBIOS_SUB_RESETGAME      (0xc00582)
```

Notes:

- **biosCall** must be user for BIOS functions not requiring arguments
- **biosCall2** is to be used for functions requiring an argument provided in `D0` or `A0`. See NeoGeo dev wiki for functions details

Return value

N/A

clearFixLayer, clearFixLayer2, clearFixLayer3

Clears the fix layer.

Syntax

void clearFixLayer()
void clearFixLayer2()
void clearFixLayer3()

Explanation

Clears the display fix layer.
Clearing is done with tile 0x0ff, make sure it is transparent in your fix data (it will be if using the standard system font).

Totally wipes the fix data, unlike bios FIX_CLEAR function which leaves black bars.

Notes:

- **clearFixLayer** operates immediately, not on next vblank. Performs VRAM operations and therefore isn't IRQ safe
- **clearFixLayer2** is an IRQ safe version of **clearFixLayer**
- **clearFixLayer3** uses fix command buffer, clear will be performed during next Vblank

Return value

N/A

clearSprites

Clears a set of sprites.

Syntax

void clearSprites(

u16 *spr,*	First sprite to clear
u16 *count* **)**	Number of sprites to clear, from starting sprite

Explanation

Clears a block of sprites from *spr* to *spr+count-1*. Clear is buffered for execution on next Vblank. Sprite clearing is done by unlinking it, setting a 0 size and position it offscreen. Tiledata, shrink values and X position aren't affected.

Return value

N/A

disableIRQ

Disables IRQ on the system.

Syntax

void disableIRQ()

Explanation

IRQ will no longer be triggered after calling this function.

Disables both IRQ1 and IRQ2.

Return value

N/A

enableIRQ

Enables IRQ on the system.

Syntax

void enableIRQ()

Explanation

IRQ will be active after calling this function.

Enables both IRQ1 and IRQ2.

Return value

N/A

initGfx

Initialize the library for graphics operations.

Syntax
void initGfx()

Explanation
Resets and sets up library for operation.
Calling this function is required before using the library.

The function notably resets frame counters and unloads timer interrupt function.

Return value
N/A

jobMeterSetup, jobMeterSetup2

Sets up the job meter.

Syntax

void jobMeterSetup(
bool *setDip* **)** Automatic soft dip setting

void jobMeterSetup2(
bool *setDip* **)** Automatic soft dip setting

Explanation

Draws the job meter on the fix layer, using fix tile `JOB_METER_CHARACTER` and palette `JOB_METER_PALETTE`. Make sure that the chosen tile is a plain tile of color `JOB_METER_COLOR` for proper display (it will be if using the standard system font).
Default values are character #0x0ff, from bank #0, using color #1.
Job meter takes place on the far right column of the fix layer.

For the job meter to be updated during vblank, devmode and soft dip 2-1 must be on.

Call function with *setDip* parameter set to *true* for the function to force bios devmode setting and soft dip 2-1 to on. This basically saves you from enabling them again manually on each boot.

jobMeterSetup2 is an IRQ safe variant of **jobMeterSetup**.

Note: Forcing bios setting is kind of a hack job, it isn't guaranteed to work on all bios (tested ok on debug bios and uinibios 3.2), try out and use accordingly. Do not use in release code.

Meter color can be changed via `jobMeterColor`, IE `jobMeterColor = 0x8000;`

Return value

N/A

loadTlirq

Loads timer interrupt handler.

Syntax

void loadTlirq(
u16 *mode***)** IRQ mode

Explanation

Loads the required code to process the timer interrupt.

Two modes are available:

- `TI_MODE_SINGLE_DATA:` One VRAM change per interrupt
- `TI_MODE_DUAL_DATA:` Two VRAM changes per interrupt

Return value

N/A

SCClose

Readies draw data for display.

Syntax

void SCClose()

Explanation

Closes sprite control draw lists and prepare system for next vblank.

`SCClose` will allow draw lists to be processed upon next VBlank and therefore need to be called before `waitVBlank`, or the library won't update display and will issue a frameskip.

Return value

N/A

setBank (macro)

Bring up bank of select object.

Syntax
setBank(u32 addr)

Explanation
This shortcut macro will load bank for provided address, effectively writing provided address upper byte to the banking register.

Return value
N/A

setBankNum (macro)

Bring up selected bank number.

Syntax
setBank(u8 bank)

Explanation
This shortcut macro will load provided bank number, effectively writing provided byte to the banking register.

Return value
N/A

setup4P

Initialize 4P input mode.

Syntax

bool setup4P()

Explanation

This function will check if a 4P adapter (NEO-FTC1B / NEO-4P) is hooked to the system.
It should enable 4 players mode on any bios if hardware is found.

Return value

`FALSE` - adapter was not found
`TRUE` - adapter was found

unloadTIirq

Unloads timer interrupt handler.

Syntax
void unloadTIirq()

Explanation
Unloads the required code to process the timer interrupts.

This actually loads a failsafe handler (acknowledge IRQ then return), shall a timer interrupt occur when unexpected.

Note: make sure you set `TinextTable` to 0, then wait for a VBlank to occur before using `unloadTIirq()` to avoid unstable behavior.

Return value
N/A

waitVBlank

Waits for next vblank.

Syntax

void waitVBlank()

Explanation

Holds program execution until next vblank is triggered.

Program will resume after the vblank interrupt has been processed.

Note: this function also automates some tasks:

- color math commands will be processed before VBL sync
- one sound code will be flushed from the sound buffer after VBL sync

Return value

N/A

String / Text components

About string formats

Text functions can handle two string formats: 8 or 16 bits.

8 bits format: Standard 8 bits character encoding for general purpose use. Ends with a 0x00 character.

16 bits format: 16 bits character encoding aimed for display on fix layer, using VRAM character format.

15-12	11-8	7-0
Palette number	Character code MSB *	Character code LSB

* Character code MSB is often referenced as “bank”.

16 bits strings ends with a 0x0000 character.

sprintf2

Formats a text string.

Syntax

u16 sprintf2(

char **dest,*	Pointer to destination buffer
char **format,*	Pointer to format string
…)	Extra arguments

Explanation

Will process the format string and arguments, writing the result into the dest buffer.
This is a streamlined and tweaked alternative to the standard **sprintf** function, allowing faster execution.

Available format tags:

- %d: prints an signed integer, decimal format
- %u: prints an unsigned integer, decimal format
- %x: prints an integer, hex format
- %c: prints a 8 bit character
- %s: prints a string
- %0: pads the following argument with zeros
  - Ex: %08x will print an integer in hex fomat, with a 8 characters size.
  - Valid sizes are 2-12, encoded as `23456789:;<` characters.
  - Weird but saves cycles :)
- %w: prints a 16 bit character (**sprintf3** only)

Return value

Total written characters count, excluding string termination character.

sprintf3

Formats a text string. 16bits fix character format.

Syntax

u16 sprintf3(

u32 printInfo,	print parameters (see `PRINTINFO`), only pal & bank are used
char **dest,*	Pointer to destination buffer
char **format,*	Pointer to format string (standard 8 bit characters format)
…)	Extra arguments

Explanation

Will process the format string and arguments, writing the result into the dest buffer.
Input format string is standard 8bits encoding. Output string is 16bits fix format encoding, using provided palette and bank setting.

This function is equivalent to **sprintf2**, aside the different output format.

Available format tags: see **sprintf2.**

Return value

Total written characters count, excluding string termination character.

fixPrint, fixPrint2, fixPrint3, fixPrint4

Displays a character string on the fix layer.

Syntax

void fixPrint(
- **u32** printInfo, — Print parameters (see `PRINTINFO` macro)
- **char** **buf***)** — String to print (8bit character format)

void fixPrint2(
- **u32** printInfo, — Print parameters (see `PRINTINFO` macro)
- **char** **buf***)** — String to print (8bit character format)

void fixPrint3(
- **u32** printInfo, — Print parameters (see `PRINTINFO` macro), bank ignored
- **char** **buf***)** — String to print (16bit character format)

void fixPrint4(
- **u32** printInfo, — Print parameters (see `PRINTINFO` macro), bank ignored
- **char** **buf***)** — String to print (16bit character format)

Explanation

Will print the supplied 8/16bit characters string to the fix layer, using supplied parameters.

Functions are not Vblank synced and will print the text immediately during active display. It is therefore advised to use carefuly and/or only for debug messages purpose.

fixPrint2 is an IRQ safe version of **fixPrint**.
fixPrint4 is an IRQ safe version of **fixPrint3**.

Return value

N/A

fixPrintf1, fixPrintf2, fixPrintf3

Formats and text string and displays it on fix layer, macros.

Syntax

void fixPrintf1(

u32 printInfo,	Print parameters (see `PRINTINFO` macro)
char **format,*	Format string pointer
…)	Extra arguments

void fixPrintf2(

* *Same as* ***fixPrintf1*** *

)

void fixPrintf3(

u32 printInfo,	Print parameters (see `PRINTINFO` macro)
u16 **buffer,*	Buffer pointer for formatted string (16bits format)
char **format,*	Format string pointer (8bit format)
…)	Extra arguments

Explanation

Will process the format string and arguments, displaying the result on the fix layer.
Available format tags: see **sprintf2.**

fixPrintf1 and **fixPrintf2** macros are using `libNG_scratchpad64` as a temporary string buffer before output. Make sure you do not exceed 64 characters.

fixPrintf1: Standard print.
fixPrintf2: Similar to **fixPrintf1**, but using the IRQ safe **fixPrint2** for display.
fixPrintf3: Uses the supplied buffer to store the resulting formatted 16bits string, and adds display command to the `FIXJOBS` buffer. Display is Vblank synced.

Return value

N/A

Pictures components

picture

Runtime handler for a picture.

Syntax

typedef struct picture {

u16 *baseSprite;*	Base sprite # used for this picture
u8 *basePalette;*	Base palette # used for this picture
u8 RFU*;*	Unused / padding.
s16 *posX;*	Current position, X axis
s16 *posY;*	Current position, Y axis
u16 *currentFlip;*	Current flip mode.
pictureInfo **info;*	Pointer to the pictureInfo struct of this picture

} picture;

Explanation

This is the base structure the library uses to handle picture type elements.
Has to be allocated in the ram section of your code.

Pictures require user to use the provided functions for proper operation (pictureSetPos etc…). Manually editing fields won”t show results on screen.

pictureInfo

Structure holding picture information.

Syntax

```
typedef struct pictureInfo {
    u16 stripSize;      Bytesize of each sprite tilemap (basically tileHeight*4)
    u16 tileWidth;      Picture width, tiles unit
    u16 tileHeight;     Picture height, tiles unit
    paletteInfo *palInfo;   Pointer to related paletteInfo
    u16 *maps[4];       Pointers to tilemaps (standard, flipX, flipY, flipXY)
} pictureInfo;
```

Explanation

pictureInfo structures are generated by the buildchar tool. Holds basic info about the picture.

Tilemap pointers are always valid. IE if you did not request flipX for that picture in buildChar tool, maps[1] will point to the standard map.
Picture tilemaps bytesize is (tileWidth*tileHeight)*4, or StripSize*tileWidth.

pictureHide

Hide a picture.

Syntax

void pictureHide(
picture *p* **)** Pointer to picture structure to use

Explanation

Removes designated picture element from display.

Note: As hiding is done by altering Y position and sprite size, please be aware that changing Y pos of designated picture will revert it back to visible.

Return value

N/A

pictureInit

Initialize a picture structure for use.

Syntax

void pictureInit(

picture **p,*	Pointer to picture handler to use
pictureInfo **pi,*	Pointer to pictureInfo structure
u16 *baseSprite,*	Base sprite # to use
u16 *basePalette,*	Base palette # to use
s16 *posX,*	Picture initial X position
s16 *posY,*	Picture initial Y position
u16 *flip* **)**	Picture initial flip mode

Explanation

Initialize and prepare a picture element for use.
Picture will be set up with provided initial position/flip.
Picture object will be drawn on next Vblank.

Return value

N/A

pictureMove

Updates position of a picture object.

Syntax

void pictureMove(

picture **p*,	Pointer to picture handler to use
s16 *shiftX*,	X axis offset
s16 *shiftY* **)**	Y axis offset

Explanation

Change picture screen position.
New position is determined relatively to current position (new pos= current pos + shift).

Return value

N/A

pictureSetFlip

Sets flip mode of a picture entity.

Syntax

void pictureSetFlip(

picture **p,*	Pointer to picture handler to use
u16 *flip* **)**	Desired flip mode

Explanation

Change picture flip mode.
Flip modes most be specified in your chardata.xml file for the buildchar tool to make them available.
Will default to base orientation if requested flip mode isn't available.
Picture will be redrawn with the nex orientation on next Vblank.

Return value

N/A

pictureSetPos

Sets position of a picture entity.

Syntax

void pictureSetPos(

picture **p,*	Pointer to picture handler to use
s16 *toX,*	New X position
s16 *toY* **)**	New Y position

Explanation

Change picture screen position.
Position is set to supplied values.

Return value

N/A

pictureShow

Show a picture entity.

Syntax

void pictureShow(

picture *p **)**	Pointer to picture handler to use

Explanation

Put back a previously hidden picture on display.
Picture will be displayed at latest set position with latest set flip.

Return value

N/A

Scrollers components

scroller

Runtime handler for a scroller.

Syntax

```
typedef struct scroller {
    u16 baseSprite;          Base sprite # used for this scroller
    u16 basePalette;         Base palette # used for this scroller
    u16 scrlPosX;            Current scroll index, X axis
    u16 scrlPosY;            Current scroll index, Y axis
    scrollerInfo *info;      Pointer to the scrollerInfo struct of this scroller
    u16 config[32];          Scroller configuration data - internal use
} scroller;
```

Explanation

This is the base structure the library uses to handle scroller type objects.
Has to be allocated in the ram section of your code.

As operation on this datatype is managed by the library, it is strongly advised to use as read only in your code.

scrollerInfo

Structure holding scroller information.

Syntax

```
typedef struct scrollerInfo {
        u16 stripSize;          Bytesize of each sprite tilemap (basically mapHeight*4)
        u16 sprHeight;          Required sprite height to use (max 32)
        u16 mapWidth;           Scroller width, tiles unit
        u16 mapHeight;          Scroller height, tiles unit
        paletteInfo *palInfo;   Pointer to related paletteInfo
        colorStreamInfo *csInfo;        Pointer to related colorStreamInfo
        u16 *strips[0];         Tilemap data (size varies)
} scrollerInfo;
```

Explanation

scrollerInfo structures are generated by the buildchar tool. Holds basic info about the scroller.

Actual map data size (u16) is (mapWidth*mapHeight)*2.

Member *csInfo* wil be `0x00000000` if there is no **colorStream** related to the scroller.

scrollerInit

Initialize a **scroller** object handle for use.

Syntax

void scrollerInit(

scroller **s,*	Pointer to **scroller** handler to use
scrollerInfo **si,*	Pointer to **scrollerInfo** structure
u16 *baseSprite,*	Base sprite # to use
u16 *basePalette,*	Base palette # to use
short *posX,*	Scroller initial X position
short *posY* **)**	Scroller initial Y position

Explanation

Initialize and prepare a **scroller** handler for use.
scroller will be set up with provided initial scroll positions.

Return value

N/A

scrollerSetPos

Sets new position into scroll plane

Syntax

void scrollerInit(

scroller **s*,	Pointer to **scroller** handler
s16 *toX*,	Scroller X position
s16 *toY* **)**	Scroller Y position

Explanation

Sets scrolling position of designated **scroller** object handle.

Return value

N/A

Animated sprites components

aSprite

Runtime object handle for an animated sprite.

Syntax

```
typedef struct aSprite {
    u16 baseSprite;             Base sprite # used for this animated sprite
    u8 basePalette;             Base palette # used for this animated sprite
    u8 bank;                    Data bank #
    s16 posX;                   Animated sprite current X position
    s16 posY;                   Animated sprite current Y position
    u16 animID;                 ID of last requested animation
    u16 currentAnim;            ID of current animation
    u16 stepNum;                Current step number
    animStep *anims;            Pointer to animations block
    animStep *steps;            Pointer to steps block of current animation
    animStep *currentStep;      Pointer to current step data
    sprFrame *currentFrame;     Pointer to current frame data
    u32 counter;                Internal frame update counter
    u16 repeats;                Number of repeats done
    u16 tileWidth;              Width of current frame, tiles unit
    u16 currentFlip;            Current flip mode
    union
    {
        u16         flags;                  Flags
        struct
        {
            u16     flag_noAnim : 1;        Animation disable
            u16     flag_padding : 7;
            u16     flag_noDisplay : 1;     Display disable
            u16     flag_strictCoords : 1;  Use strict coordinates
            u16     flag_none : 4;
            u16     flag_flipped : 1;       Flipped flag (internal)
            u16     flag_moved : 1;         Moved flag (internal)
        };
    };
    union
    {
        struct
        {
            u8      Xbig;           X axis scale factor
            u8      Ybig;           Y axis scale factor
        };
        u16     XYbig;          packed scale factor on both X and Y axis
    };
} aSprite;
```

Explanation

This is the base structure the library uses to handle animated sprites objects.
Has to be allocated in the ram section of your code.

When using animated sprites with the recommanded sprite pools, it is possible to manipulate fields directly. Otherwise use the provided functions

When animation has reached its end (when applicable), counter value will change to 0xffffffff.

Notes:

`.currentFlip` format is as follows:

15-2	1	0
00000000000000	Vertical flip	Horizontal flip

Related defines:

```
FLIP_NONE    (0)
FLIP_X       (1)
FLIP_Y       (2)
FLIP_XY      (3)
FLIP_BOTH    (3)
```

`.flags` format is as follows:

15	14-8	7	6	5-2	1	0
Anim stop	00000000	No display	Strict coords	0000	Flipped	Moved

Moved / Flipped flags are only relevant when using allocated sprite mode.

Related defines:

```
AS_FLAGS_DEFAULT          (0x0000)
AS_FLAG_MOVED             (0x0001)
AS_FLAG_FLIPPED           (0x0002)
AS_FLAG_STD_COORDS        (0x0000)
AS_FLAG_STRICT_COORDS     (0x0040)
AS_FLAG_DISPLAY           (0x0000)
AS_FLAG_NODISPLAY         (0x0080)
AS_FLAG_NOANIM            (0x8000)

AS_MASK_MOVED             (0xfffe)
AS_MASK_FLIPPED           (0xfffd)
AS_MASK_MOVED_FLIPPED     (0xfffc)
AS_MASK_STRICT_COORDS     (0xffbf)
AS_MASK_NODISPLAY         (0xff7f)
AS_NOSPRITECLEAR          (0x7fff)
```

spriteInfo, animStep, sprFrame

Structures holding animated sprites informations.

Syntax

```
typedef struct spriteInfo {
    u16 frameCount;          Total number of frames
    u16 maxWidth;            Maximum width, tiles unit (width of the largest frame)
    paletteInfo *palInfo;    Pointer to related paletteInfo
    animStep **anims;        Pointer array to animations
    sprFrame frames[0];      sprFrames array
} spriteInfo;

typedef struct animStep {
    sprFrame *frame;         Pointer to frame info
    s16 flipShiftX;          Frame X displacement from origin when X flipped
    s16 shiftX;              Frame X displacement from origin
    s16 flipShiftY;          Frame Y displacement from origin when Y flipped
    s16 shiftY;              Frame Y displacement from origin
    u16 duration;            Number of frame to display
} animStep;

typedef struct sprFrame {    Frame data, plain format
    u16 tileWidth;           Frame width, tiles unit. Value 0 means split frame format
    u16 tileHeight;          Frame height, tiles unit.
    u16 stripSize;           Bytesize of each sprite tilemap (basically tileHeight*4)
    u16 *maps[4];            Pointers to frame tilemaps (standard, flipX, flipY, flipXY)
} sprFrame;

typedef struct sprFrame2 {   Frame data, split format
    u16 key;                 Key to indicate split format, always 0.
    u16 sprCount;            Amount of hardware sprites required.
    u16 *maps[4];            Pointers to frame tilemaps (standard, flipX, flipY, flipXY)
} sprFrame2;
```

Explanation

spriteInfo, **animStep**, **sprFrame** and **sprFrame2** structures are generated by the buildchar and animator tools. Holds infos about animated sprite frames and animations.

Frame tilemap pointers are always valid. IE if you did not request flipX for that sprite in the buildchar tool, maps[1] will point to the standard map.

Plain frame tilemaps size (u16 count) are (tileWidth*tileHeight)*2.

aSpriteAnimate

Performs animation updates on an **aSprite** object handle.

Syntax

void aSpriteAnimate(
aSprite **as* **)** Pointer to **aSprite** handle to use

Explanation

Updates the **aSprite** handle animation.
Will apply position/flip/animation changes and queue required commands into draw buffers for update next VBlank.
This function must be called every frame for each animated sprite for proper animation.

Note: this function is for allocated sprites mode, See **spritePoolDrawList** for sprite pool use.

Return value

N/A

aSpriteHide

Hides an **aSprite** object (macro).

Syntax

void aSpriteHide(
aSprite **as* **)** Pointer to **aSprite** handler to use

Explanation

Flag the designated **aSprite** as no display.
When flagged as no display, animated sprites will no longer be displayed. This allows to keep animating an offscreen/hidden object without having to display it.

Note: If the aSprite is currently used in allocated mode, you must manually clear the sprites used by the current frame => `clearSprites(as->baseSprite, as->tileWidth);`

If using pool mode, you can change the flag setting directly.

Return value

N/A

aSpriteInit

Initialize an **aSprite** object for use.

Syntax

void aSpriteInit(

aSprite **as,*	Pointer to **aSprite** handler to use
spriteInfo **si,*	Pointer to **spriteInfo** structure
u16 *baseSprite,*	Base sprite # to use
u16 *basePalette,*	Base palette # to use
s16 *posX,*	aSprite initial X position
s16 *posY,*	aSprite initial Y position
u16 *anim,*	aSprite initial animation sequence
u16 *flip*	aSprite initial flip mode
u16 *flags* **)**	aSprite initial flags

Explanation

Initialize and prepare an **aSprite** handle for use.
aSprite will be set up with provided initial position, animation, flip mode and flags.
This function will not push frame to display, a call to **aSpriteAnimate** / **spritePoolDrawList** is required after aSpriteInit to push initial frame on display upon next VBlank.

If using pool mode, set baseSprite as `AS_USE_SPRITEPOOL`.
If using fixed allocation and want to disable sprite clear from init, set baseSprite as `AS_NOSPRITECLEAR` + <spr slot>

Return value

N/A

aSpriteMove

Updates position of an **aSprite** object.

Syntax

void aSpriteMove(

aSprite **as,*	Pointer to **aSprite** handler
s16 *shiftX,*	X axis offset
s16 *shiftY* **)**	Y axis offset

Explanation

Change **aSprite** handler screen position.
New position is determined relatively to current position (new pos= current pos + shift).
Will not update the display position directly, use **aSpriteAnimate** / **spritePoolDrawList** afterward to apply changes.

Note: When using sprite pools, you can freely increase or decrease the **aSprite** `.posX` and `.posY` fields, without the need of this function.

Return value

N/A

aSpriteSetAnim, aSpriteSetAnim2

Sets animation for an **aSprite** object.

Syntax

void aSpriteSetAnim(

aSprite **as,*	Pointer to **aSprite** handler
u16 *anim* **)**	Animation ID

void aSpriteSetAnim2(

aSprite **as,*	Pointer to **aSprite** handler
u16 *anim* **)**	Animation ID

Explanation

Change current animation.
Animation IDs are defines issued by the animator tool, see documentation for syntax.
Will not push frame to display, use **aSpriteAnimate** / **spritePoolDrawList** afterward to apply changes.
If requesting change to the animation sequence ID that is already running, nothing will be done.

About animation links:
When using linked animations (ie A > B > C (loop)) system will remember “A” as last requested animation ID.
This means if said animated sprite ran long enough to reach animation “C”, a request for animation ID “A” might be discarded as this is the last requence requested and running.

aSpriteSetAnim will discard animation requests of the same ID.
aSpriteSetAnim2 will set animation regardless of current state. If the same animation is already running, it will be rewinded/reset.

Return value

N/A

aSpriteSetStep, aSpriteSetStep2

Sets step number for an **aSprite** object.

Syntax

void aSpriteSetStep(
aSprite **as,* Pointer to **aSprite** handler
u16 *step* **)** Step number

void aSpriteSetStep2(
aSprite **as,* Pointer to **aSprite** handler
u16 *step* **)** Step number

Explanation

Moves current animation of the provided **aSprite** handler to selected step number.

aSpriteSetStep will discard request if current step is the same as requested.
aSpriteSetStep2 will set step regardless of current state. If the same step is already displayed, step timing will be reset.

Return value

N/A

aSpriteSetAnimStep, aSpriteSetAnimStep2

Sets animation and step number for an **aSprite** object.

Syntax

void aSpriteSetAnimStep(

aSprite **as,*	Pointer to **aSprite** handler
u16 *anim,*	Animation ID
u16 *step* **)**	Step number

void aSpriteSetAnimStep2(

aSprite **as,*	Pointer to **aSprite** handler
u16 *anim,*	Animation ID
u16 *step* **)**	Step number

Explanation

Changes current animation of privided **aSprite** handler, running from the choosen step number.

Animating rules applied are the same as `aSpriteSetAnim`.

aSpriteSetAnimStep will discard request if current animation and step is the same as requested.
aSpriteSetAnimStep2 will set animation and step regardless of current state. Step timing will be reset if parameters are same as current state.

Return value

N/A

aSpriteSetFlip

Sets flip mode of an **aSprite** object.

Syntax

void aSpriteSetFlip(

aSprite **as,*	Pointer to **aSprite** handler
u16 *flip* **)**	Desired flip mode

Explanation

Change **aSprite** handler flip mode.
Flip modes most be specified in your chardata.xml file for the buildchar tool to make them available.
Will default to base orientation if requested flip mode isn't available.

Note: When using sprite pools, you can freely set requested flip mode directly into the **aSprite** `.currentFlip` field, without the need of this function.

Return value

N/A

aSpriteSetPos

Sets position of an **aSprite** object.

Syntax

void aSpriteSetPos(

aSprite **as,*	Pointer to **aSprite** handler
s16 *newX,*	New X position
s16 *newY* **)**	New Y position

Explanation

Change **aSprite** handler screen position.
Will not update the display position directly, use **aSpriteAnimate** / **spritePoolDrawList** afterward to apply changes.

Note: When using sprite pools, you can freely set coordinates directly into the **aSprite** `.posX` and `.posY` fields, without the need of this function.

Return value

N/A

aSpriteShow

Reverts an hidden **aSprite** object to visible. (macro).

Syntax

void aSpriteShow(
aSprite **as* **)** Pointer to **aSprite** handler

Explanation

Removes the no display flag from the designated **aSprite**.
Returns the aSprite to its normal state, allowing it to be displayed again.

Has no effect if **aSprite** handler is already flaged as visible.

Note: When using sprite pools, you can freely set the visibility flag, without the need of this function.

Return value

N/A

Sprite Pools components

spritePool

Runtime handle for a sprite pool.

Syntax

```
typedef struct spritePool {
    u16 poolStart;      Fist sprite # to be used for this sprite pool
    u16 poolEnd;        Last sprite # to be used for this sprite pool
    u16 poolSize;       Sprite pool size
    u16 way;            Current draw direction
    u16 currentUp;      Current spr index - internal use
    u16 currentDown;    Current spr index - internal use
} spritePool;
```

Explanation

This is the base structure the library uses to handle sprite pools elements.
Has to be allocated in the ram section of your code.

As operation on this datatype is managed by the library, it is advised to manipulate only using provided functions.

Related defines:

```
WAY_UP      (0)
WAY_DOWN    (1)
```

spritePoolClose

Finalize sprite pool operations for display.

Syntax

u16 spritePoolClose(
spritePool **sp* **)** Pointer to **spritePool** handler

Explanation

Prepares a spritePool for next VBlank.
Needs to be called before each VBlank, will switch pool direction and queue the necessary sprite clears for correct display.

Note: Sprite pool passed to this function is not to be used before next Vblank has occurred.

Return value

Will return 1 when draw operations exceeded total pool size, 0 otherwise.

spritePoolDrawList, spritePoolDrawList2, spritePoolDrawList3

Draws the supplied animated sprites list into sprite pool.

Syntax

void spritePoolDrawList(
spritePool **sp* Pointer to **spritePool** handler
void **list* **)** Pointer to draw list

void spritePoolDrawList2(
spritePool **sp* Pointer to **spritePool** handler
void **list* **)** Pointer to draw list

void spritePoolDrawList3(
spritePool **sp* Pointer to **spritePool** handler
void **list* **)**

Explanation

Utilize the supplied **spritePool** to render the **aSprite** entities in the supplied list.
This function takes care of updating the **aSprite** animation state, then display the updated entity.

Notes: User must supply a list pointer according to the current direction of the sprite pool :

- `WAY_UP`: list must point to the first item, list will be read upward until null is found
- `WAY_DOWN`: list must point to the last+1 element, list will be read downward until null is found

SpritePoolDrawList isn't IRQ safe.
SpritePoolDrawList2 is an IRQ safe variant of **spritePoolDrawList**.
SpritePoolDrawList3 isn't IRQ safe, supports scaling and split frame format.

Return value

N/A

spritePoolInit

Initialize the supplied sprite pool handle.

Syntax

void spritePoolInit(

spritePool **sp,*	Pointer to **spritePool** handler
u16 *baseSprite,*	Startig sprite of sprite pool
u16 *poolSize,*	Sprite pool size
bool *clearSprites* **)**	Sprites clear flag

Explanation

Sets up the supplied **spritePool** handle for use.

If *clearSprites* is set to true, **spritePoolInit** will buffer a sprite clear of all sprites withing the pool range.

Return value

N/A

Color steam components

colorStream

Runtime handle for a color stream.

Syntax

```
typedef struct colorStream {
    u16     palMod;          Base palette offset for this stream (basically basePal *32)
    u16     position;        Holds current position in stream – internal use
    colorStreamInfo *info;   Pointer to related colorStreamInfo
    colorStreamJob *fwJob;   Pointer to next job, forward way – internal use
    colorStreamJob *bwJob;   Pointer to next job, backward way – internal use
} colorStream;
```

Explanation

This is the base structure the library uses to handle color streams elements.
Has to be allocated in the ram section of your code.

As operation on this datatype is managed by the library, it is strongly advised to use as read only in your code.

colorStreamInfo, colorStreamJob

Structures holding color stream informations and data.

Syntax

typedef struct colorStreamInfo {

u16 *palSlots;*	Number of palettes required to operate the **colorStream**
void **startConfig;*	Pointer to start configuration data
void **endConfig;*	Pointer to end configuration data
void **fwData;*	Pointer to forward stream data
void **fwDataEnd;*	Pointer to end of forward stream data
void **bwData;*	Pointer to backward stream data
void **bwDataEnd;*	Pointer to end of backward stream data

} colorStreamInfo;

typedef struct colorStreamJob {

u16 *coord;*	Stream update coordinate
void **data;*	Pointer to update data

} colorStreamJob;

Explanation

colorStreamInfo and **colorStreamJob** structures are generated by the buildchar tool. Holds informations about color streams.

Configurations and jobs format are as follows:

```
.word  0x0012       ; palette slot #
.long  0x00123456   ; pointer to palette data
...
.word  0xffff       ; end marker
```

colorStreamInit

Initialize the supplied color stream handle.

Syntax

void colorStreamInit(

colorStream **cs,*	Pointer to **colorStream** handler
colorStreamInfo ***csi,**	Pointer to related **colorStreamInfo** structure
u16 *basePalette,*	Base palette # to use
u16 *config* **)**	Start configuration

Explanation

Sets up the supplied **colorStream** handler for use.

Will buffer the required palette jobs to set up the requested start *config*.

Related defines:

```
COLORSTREAM_STARTCONFIG   (0)
COLORSTREAM_ENDCONFIG     (1)
```

Return value

N/A

colorStreamSetPos

Updates the stream position of supplied color stream handle.

Syntax

void colorStreamSetPos(

colorStream **cs,*	Pointer to **colorStream** handler
u16 *pos* **)**	New stream position

Explanation

Advances or rewinds the supplied **colorStream** to the requested position.

colorStreamSetPos will buffer the required palette commands to update the color stream up to the designated position.

Return value

N/A

Sound components

sndReset

Resets the sound commands ring buffer.

Syntax

void sndReset (
    bool *sendResetCode* Send reset code to sound CPU flag
)

Explanation

This will empty the ring buffer holding the sound commands.
If called with *sendResetCode* set to ture, reset command (3) will be issued to the sound CPU. Make sure to leave enough time for Z80 to reset before sending codes again.

Return value

N/A

sndAddCode

Adds a command to ring buffer.

Syntax

void sndAddCode (
    u8 *code* Sound code to be added to buffer
)

Explanation

Puts a sound code into the ring buffer to be sent to sound CPU.

When ring buffer is full, last code is overwritten.

Sound codes are dispatched automatically by the waitVBlank function. By default 1 code is dispatched each frame, build option `SOUNDBUFFER_DISPATCH_TWICE` can be used to change to 2 codes per frame.

Return value

N/A

sndDispatch

Dispatch one sound code to sound CPU

Syntax

void sndDispatch ( )

Explanation

Picks pending sound code from ring buffer and sends it to sound CPU.

If the ring buffer is empty, function will return without anything being sent.

Note: This function is called automatically by the `waitVBlank` function, there is therefore no need to call it under normal circumstances.

Return value

N/A

Color math components

cMathLoadPalette

Setup palette data.

Syntax

void cMathLoadPalette (

u16 *slot,*	Palette slot # to load data in
u16 *count*	Number of consecutive palettes to load
u16 *cmd*	Color command to apply to loaded data
palette **pal*	Pointer to color data

)

Explanation

Setup color data in palette handles for use with color math commands.

For a standard palette setup + color RAM transfer, use the FADE_RESET command.

Return value

N/A

cMathSetCommand

Issue a color math command.

Syntax

void cMathLoadPalette (

u16 *slot,*	Palette slot # to load data in
u16 *count,*	Number of consecutive palettes to load
u16 *cmd*	Color command to apply to loaded data

)

Explanation

Issue a command to select palette slot(s).
Commands are composed of 3 parts: effect type, effect colors and effect speed.

Effect types are:

- `FADE_TO` base palette colors > selected effect color fade
- `FADE_FROM` base palette colors < selected effect color fade
- `FADE_FILL` fill with selected effect colors
- `FADE_RESET` reset palette to base colors

Effect colors are:

- `FADE_BLACK` (no color)
- `FADE_RED`
- `FADE_GREEN`
- `FADE_BLUE`
- `FADE_YELLOW` (red + green)
- `FADE_PURPLE` (red + blue)
- `FADE_CYAN` (green + blue)
- `FADE_WHITE` (red + green + blue)

Effect speeds are:

- `FADE_SPEED0` slowest speed
- `FADE_SPEED1`
- `FADE_SPEED2`
- `FADE_SPEED3` fastest speed

IE a moderate speed fade to white command would be: `(FADE_TO | FADE_WHITE | FADE_SPEED2)`.

Note: color math is CPU intensive (~ 4 raster lines timing per palette), if issuing fading commands over a large number of palettes, consider setting `palFlushOnFrameSkip` to true, this will allow partial transfer while computing, reducing sync delays between color data transfer and commands processing, diminushing slowdown perception for the user.

Return value

N/A

cMathPalEffect

Perform color math operation on palette data.

Syntax

void cMathPalEffect (

u16 **srcPal,*	Source palette data
u16 **dstPal,*	Destination palette buffer
u32 *effect_count,*	Effect type and palette count (use CMATH_EFFECT helper macro)
u32 *effectColor*	Color to apply effect with (use CMATH_COLOR helper macro)

)

Explanation

Transforms palette data by applying selected effect with set color.
Source and destination can be the same, allowing consecutive commands over the same data.

Effect types are:

- `CMATH_EFFECT_XOR`	performs xor operation on RGB components, most common use would be with color 0x1F1F1F for negative color effect
- `CMATH_EFFECT_ADD`	add supplied RGB values to color data
- `CMATH_EFFECT_ADD_HALF`	add supplied RGB values to color data then half the result, this is the “transparency” effect you are looking for
- `CMATH_EFFECT_SUB`	substract supplied RGB values to color data
- `CMATH_EFFECT_SUB_HALF`	substract supplied RGB values to color data then half the result
- `CMATH_EFFECT_DESATURATE`	desaturates color data, production black & white data This command ignores supplied effectColor

Note: will take ~2 raster lines timing per palette to process, be wary of excessive use during gameplay. Also avoid directly operating from/to palette ram during active display, this will cause onscreen color dots.

Return value

N/A