DATlib

Made in France.
Deal with it.

Rev 0.3

Table of contents

About DATlib

DATlib is a library designed for the NeoBitz/NeoDev environment.
It is designed to replace libvideo and libinput from the original kit.

Its goal is to provide easy functionality trough base elements (scroller, picture and animated sprite) which you are prone to use in your software, and allow better performance than basic libraries while writing less code.

Tools are also improved over standard ones to allow more colors (no longer limited to one palette per object), auto animation support, smaller data…

Combined tools and library allow easier coding while providing better performance and easy syncing between vblank and sprites update, greatly reducing tearing issues.

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.

DATlib will occupy about 10KB of the system ram.

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 evolve around 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 currently up to the user to manage sprites and palettes to make sure no overlaps occurs across different elements.

Installation

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

To install required files, merge the content of the archive's "NeoDev" folder with your current NeoDev installation.

DATlib is designed to supersede libvideo and libinput, make sure your remove those from your linker options in your project makefile and add DATlib library (remove `-lvideo` and `-linput`, add `-lDATlib`).

IE:

```
LIBS = -lvideo -linput -lprocess -lc –lgcc
```

becomes

```
LIBS = -lDATlib -lprocess -lc -lgcc
```

Add `<input.h>` and `<DATlib.h>` in your program includes.
If you use BuildChar to convert your data into tilemaps (most likely you will), also add the .h files made by it to your project.

Use the provided `common_crt0_cart.s` and `crt0_cart.s` file for your project, replacing older ones. (`common_crt0_cd.s` and `crt0_cd.s` for CD projects).

Features

Input management

As standard `libinput` is dropped when installing DATlib, input defines are provided in the new `input.h` include file. Input values can be read with the `volMEMBYTE()` macro from the BIOS memory region.

Support is provided for mahjong controllers as well as 4P adapters, check library reference section for available defines.

Example:

```
#include <input.h>


uchar p1;

p1=volMEMBYTE(P1_CURRENT);      /* get current status of P1 */

if(p1&JOY_A) {                  /* button A pressed ? */
      ...
}
```

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… */

SCClose(); //we done initializing
while(1) {
      waitVBlank(); // wait vblank
                    // screen updates will occur during vblank

      /* do stuff */

      SCClose(); //we done updating stuff
                 //loop for vblank sync and screen update
}
```

Provided graphics types

DATlib 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

animatedSprite

- provides support for animated sprites
- allow display, positioning, flipping and animating sprites
- animation system supports repeats and animation linking
- up to 65536 animations, unlimited 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. 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.

DAT_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
- 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.

DAT_vblankTI

Vblank handler with timer interrupt support.

Operation:

- Load base and reload timing values (if timer interrupt enabled)
- Branch to `DAT_vblank` for standard operations

Notes: When using timer interrupts, requested LSPC mode must be written to the `LSPCmode` variable (ushort). 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 `DAT_vblankTI` as your vblank IRQ vector
- set `DAT_TIfunc` as your timer IRQ vector

Notes: make sure you set variable `TinextTable` (uint) 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 (1 ushort)
- VRAM data n (1 ushort)
- VRAM address n+1 (1 ushort)
- VRAM data n+1 (1 ushort)
- (etc…)
- end marker (2 ushort, 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:

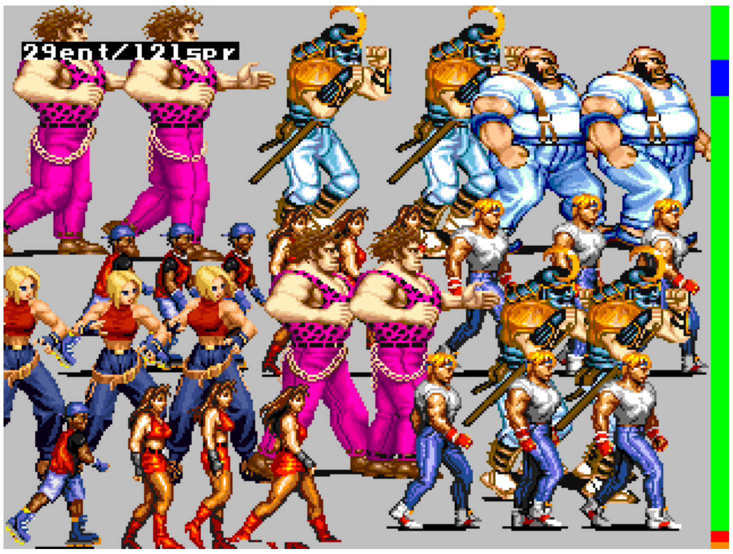
- set `TInextTable` to null (0)

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`

<u>Note:</u> 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 DATlib 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.

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

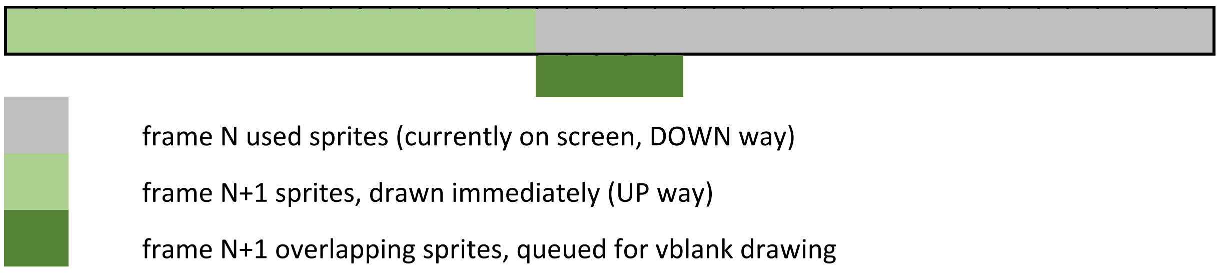
A spritePool entity must be initialized providing a pool size (# of sprites) and a starting position for this pool (baseSprite). Pool size should be aimed at twice the size of an average scene. If an average frame requires 80 sprites, ideally allocate 160.

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:


frame N used sprites (currently on screen, DOWN way)
frame N+1 sprites, drawn immediately (UP way)
frame N+1 overlapping sprites, queued for vblank drawing

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.

Note: As sprite pools are designed to update VRAM during active frame, this feature isn’t interrupt safe (using it alongside timer innterrupt can corrup VRAM info).

Vblank callbacks

Vblank callback function 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">
         <file>gfx\ffbg_c.png</file>
         <flips>xyz</flips>
   </pict>

   <sprt id="bmary_spr">
         <file>gfx\bmary.png</file>
         <flips>xyz</flips>
         <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”.
- `<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.
    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.
  - `<file>`
    PNG file of the picture.
  - `<flips>`
    Flip modes wanted for this picture (optional).
    X = horizontal flip
    Y = vertical flip
    Z = horizontal & vertical flip

- `<sprt>`
  Used to define an animated sprite
  - id (attribute)
    Literal name the animated sprite will be referenced by in C code.
  - `<file>`
    PNG file containing all animation frames.
  - `<flips>`
    Flip modes wanted for this animated sprite (optional).
    X = horizontal flip
    Y = vertical flip
    Z = horizontal & vertical flip
  - `<frame>`
    Defines a frame, format is: top,left coordinate:width,height
    Unit is tile (16px)
    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 pink 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

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 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`).

<u>Input</u>

- `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 #.
  - 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.

Framer

Tool used to delimit animated sprites frames.
Each animated sprite must be assigned a set of frames before being processed by the buildchar tool.

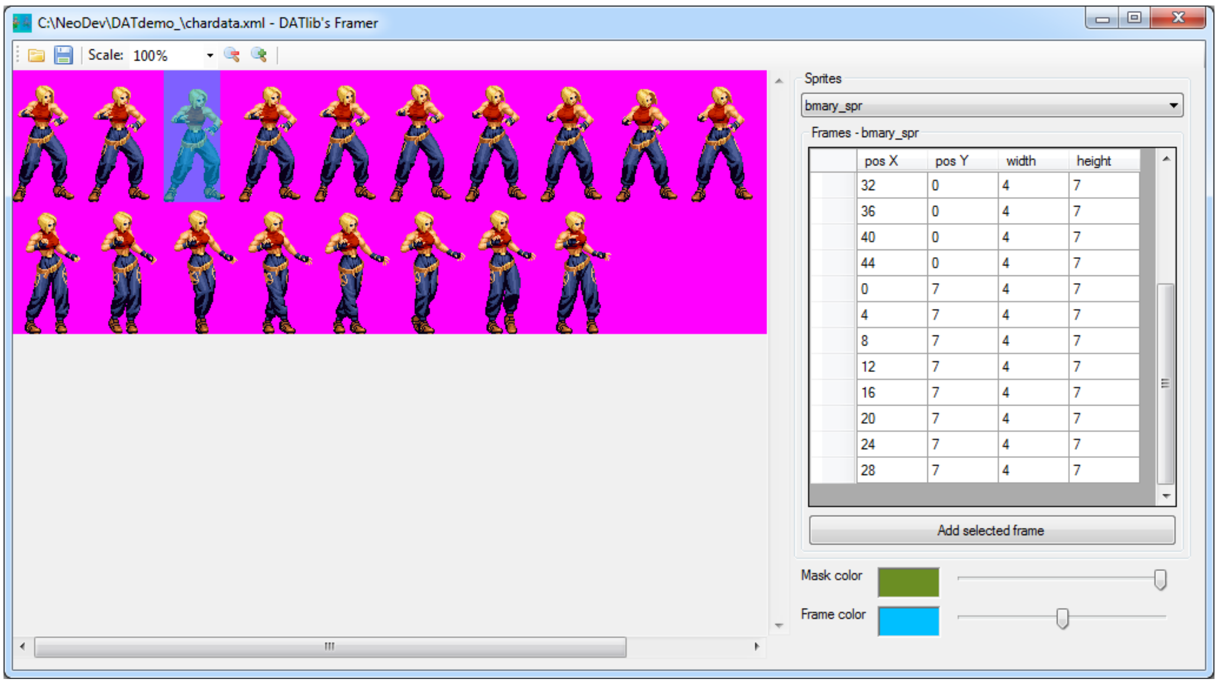
Input

- `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:



C:\NeoDev\DATdemo_\chardata.xml - DATlib's Framer

Scale: 100%

Sprites: bmary_spr

Frames - bmary_spr

pos X	pos Y	width	height
32	0	4	7
36	0	4	7
40	0	4	7
44	0	4	7
0	7	4	7
4	7	4	7
8	7	4	7
12	7	4	7
16	7	4	7
20	7	4	7
24	7	4	7
28	7	4	7

Framer is very straightforward to use. Open your xml file, then select the animated sprite 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, simply select it by clicking and dragging mouse, then click the add button, or press the space bar.
When done, click save to update the xml file, which is then ready to use with buildchar for processing.

Animator

Tool used to animate animated sprites.
Each animated sprite you process with 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

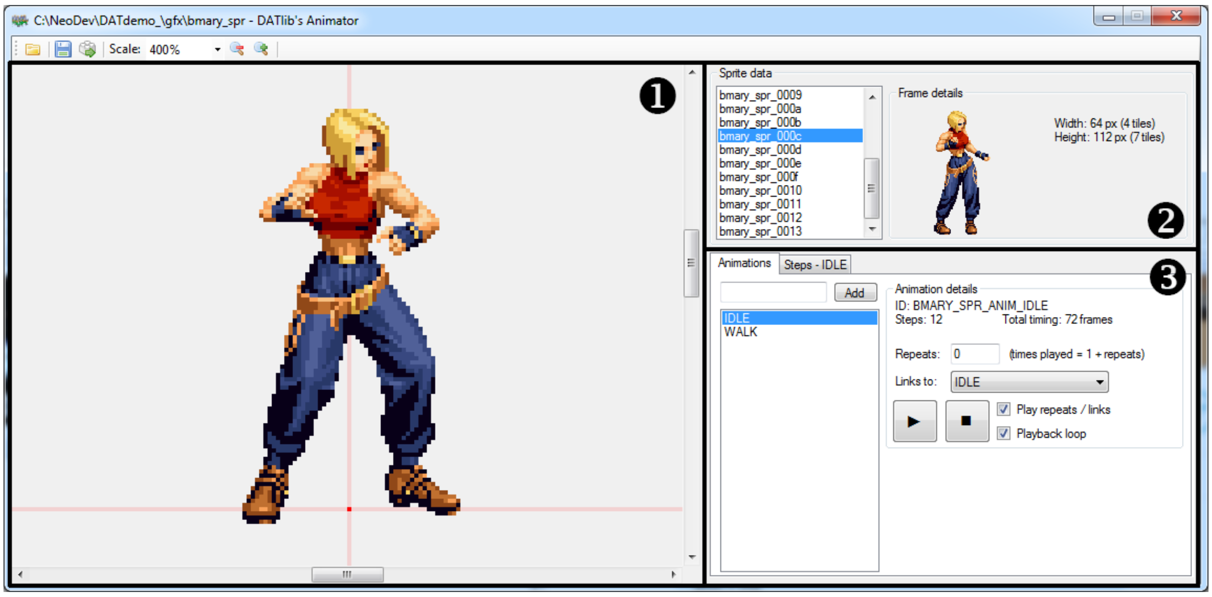
- `animdata.xml`
  This is your save file regarding this animation. Hit save button to save your work.

- `<id>_anims.s`
  Animations file, this should already be an include in your `maps.s` file by buildchar tool.

- `<id>.h`
  Contains animations C defines, should already be included in your `externs.h` file by buildchar tool.

Using Animator:

Main window is divided into 3 areas


C:\NeoDev\DATdemo_\gfx\bmary_spr - DATlib's Animator

Scale: 400%

Sprite data

bmary_spr_0009
bmary_spr_000a
bmary_spr_000b
bmary_spr_000c
bmary_spr_000d
bmary_spr_000e
bmary_spr_000f
bmary_spr_0010
bmary_spr_0011
bmary_spr_0012
bmary_spr_0013

Frame details

Width: 64 px (4 tiles)
Height: 112 px (7 tiles)

Animation details
ID: BMARY_SPR_ANIM_IDLE
Steps: 12 Total timing: 72 frames

Repeats: 0 (times played = 1 + repeats)

Links to: IDLE

[✓] Play repeats / links
[✓] Playback loop

IDLE
WALK

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

Library reference

Command buffers formats

SC1 buffer

Each `SC1` buffer entry is two 32bit uint:

31-24	23-18	17-16	15-0
Palette mod	Tile count	00	VRAM address

31-0
Tile data address

Buffer has a `0x00000000` end marker.

SC234 buffer

Each `SC234` buffer entry is two 16bit ushort:

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 32bit uint:

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 32bits uint:

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

Flip modes defines

Flip modes used for graphical elements to define orientation.

Flip modes	
`FLIP_NONE`	unflipped
`FLIP_X`	horizontal flip
`FLIP_Y`	vertical flip
`FLIP_XY`	horizontal and vertical flip
`FLIP_BOTH`	horizontal and vertical flip

Job meter defines

Basic colors used for job meter

Colors	
`JOB_BLACK`	
`JOB_WHITE`	
`JOB_LIGHTRED`	`JOB_LIGHTGREEN`
`JOB_RED`	`JOB_GREEN`
`JOB_DARKRED`	`JOB_DARKGREEN`
`JOB_LIGHTBLUE`	`JOB_LIGHTPURPLE`
`JOB_BLUE`	`JOB_PURPLE`
`JOB_DARKBLUE`	`JOB_DARKPURPLE`
`JOB_LIGHTCYAN`	`JOB_LIGHTYELLOW`
`JOB_CYAN`	`JOB_YELLOW`
`JOB_DARKCYAN`	`JOB_DARKYELLOW`
`JOB_LIGHTORANGE`	`JOB_LIGHTPINK`
`JOB_ORANGE`	`JOB_PINK`
`JOB_DARKORANGE`	`JOB_DARKPINK`
`JOB_LIGHTGREY`	
`JOB_GREY`	
`JOB_DARKGREY`	

Input related defines

Defines used to read controller data and check button presses.
All data registers are byte size.

Hardware registers	
P1_HW	hardware controller port 1 (negative logic)
P2_HW	hardware controller port 2 (negative logic)

Bios registers	
P1_STATUS	player 1 status
P1_PAST	player 1 previous frame data
P1_CURRENT	player 1 current data
P1_EDGE	player 1 active edge data
P1_REPEAT	player 1 repeat data
P1_TIMER	player 1 repeat timer
P2_STATUS	player 2 status
P2_PAST	player 2 previous frame data
P2_CURRENT	player 2 current data
P2_EDGE	player 2 active edge data
P2_REPEAT	player 2 repeat data
P2_TIMER	player 2 repeat timer
P1B_STATUS	player 3 status
P1B_PAST	player 3 previous frame data
P1B_CURRENT	player 3 current data
P1B_EDGE	player 3 active edge data
P1B_REPEAT	player 3 repeat data
P1B_TIMER	player 3 repeat timer
P2B_STATUS	player 4 status
P2B_PAST	player 4 previous frame data
P2B_CURRENT	player 4 current data
P2B_EDGE	player 4 active edge data
P2B_REPEAT	player 4 repeat data
P2B_TIMER	player 4 repeat timer
PS_CURRENT	current select/start data
PS_EDGE	active edge select/start data

Controller types (status byte value)	
CTRL_NOCONNECT	not connected
CTRL_STANDARD	standard controller
CTRL_EXPANDED	expanded controller (4P mode)
CTRL_KEYBOARD	keyboard
CTRL_MAHJONG	mahjong controller

Controller positions	
`JOY_UP`	lever up
`JOY_DOWN`	lever down
`JOY_LEFT`	lever left
`JOY_RIGHT`	lever right
`JOY_A`	A button
`JOY_B`	B button
`JOY_C`	C button
`JOY_D`	D button
`P1_START`	player 1 start button (select/start register)
`P1_SELECT`	player 1 select button (select/start register)
`P2_START`	player 2 start button (select/start register)
`P2_SELECT`	player 2 select button (select/start register)
`P1B_START`	player 3 start button (select/start register)
`P1B_SELECT`	player 3 select button (select/start register)
`P2B_START`	player 4 start button (select/start register)
`P2B_SELECT`	player 4 select button (select/start register)

Mahjong controller related	
`P1_JONG_A_G`	player 1 mahjong data, A-G buttons
`P1_JONG_H_N`	player 1 mahjong data, H-N buttons
`P1_JONG_BTN`	player 1 mahjong data, action buttons
`P2_JONG_A_G`	player 2 mahjong data, A-G buttons
`P2_JONG_H_N`	player 2 mahjong data, H-N buttons
`P2_JONG_BTN`	player 2 mahjong data, action buttons
`JONG_A`	A button
`JONG_B`	B button
`JONG_C`	C button
`JONG_D`	D button
`JONG_E`	E button
`JONG_F`	F button
`JONG_G`	G button
`JONG_H`	H button
`JONG_I`	I button
`JONG_J`	J button
`JONG_K`	K button
`JONG_L`	L button
`JONG_M`	M button
`JONG_N`	N button
`JONG_PON`	PON button
`JONG_CHI`	CHI button
`JONG_KAN`	KAN button
`JONG_RON`	RON button
`JONG_REACH`	REACH button

Library variables

General variables

uint	DAT_frameCounter	frame counter
uint	DAT_droppedFrames	dropped (skipped) frames counter
uint	*VBL_callBack	VBlank callback function pointer
uint	*VBL_skipCallBack	VBlank callback function pointer (skipped frame)
uint	SC1[760]	draw list for tilemap data
uint	*SC1ptr	pointer to tilemaps data draw list
ushort	SC234[2280]	draw list for sprite control
ushort	*SC234ptr	pointer to sprites control draw list
uint	PALJOBS[514]	palette jobs buffer
uint	*palJobsPtr;	pointer to palettes jobs buffer
uint	FIXJOBS[129];	fix jobs buffer
uint	*fixJobsPtr;	pointer to fix jobs buffer
uchar	DAT_scratchpad64[64];	64 bytes scratchpad
uchar	DAT_scratchpad16[16];	16 bytes scratchpad

Timer interrupt related variables

ushort	LSPCmode	requested LSPC mode
uint	TIbase	timer interrupt timing to first trigger
uint	TIreload	timer interrupt reload timing
ushort	*TInextTable	pointer to data table to use next frame
ushort	Tivalues0[256]	timer interrupt data space 0
ushort	TIvalues1[256]	timer interrupt data space 1

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 byte, word and dword 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 (short - 16bit) and dword (in - 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 byte, word and dword 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 (short - 16bit) and dword (in - 32bit) data. Read/write operation at an odd address for a word/long will crash the CPU.

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 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(

ushort *x,*	Target X position on fix layer
ushort *y,*	Target Y position on fix layer
ushort *mod,*	VRAM modulo
ushort *pal,*	Base palette
ushort* *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(	
ushort *number,*	Destination palette number (0-255)
ushort *count,*	Number of palettes to write
ushort* *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(

ushort *addr,*	Destination address in VRAM
ushort *size,*	Tile count
ushort *pal,*	Base palette
ushort* *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(	
ushort *addr,*	Destination address in VRAM
ushort *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 ushort data to any VRAM address.

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
- **clearFixLayer** 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(	
ushort *spr,*	First sprite to clear
ushort *count* **)**	Number of sprites to clear, from starting sprite

Explanation

Clears a block of sprites from *spr* to *spr+count-1*.
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

jobMeterColor

Changes current jobmeter color.

Syntax

void jobMeterColor(

ushort *color* **)**	Requested color

Explanation

Macro used to change job meter color to differentiate code segments execution timing.

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 of the fix layer, using fix tile 0x000 and palette 0xf. Make sure that tile is a plain color #1 tile in your fix data for proper display (it will be if using the standard system font).
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.

Return value

N/A

loadTIirq

Loads timer interrupt handler.

Syntax

void loadTIirq(
    ushort *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 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

setup4P

Initialize 4P input mode.

Syntax
int 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
0 - adapter was not found
1 - 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 function has been processed.

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 can be referenced as “bank”.

16 bits strings ends with a 0x0000 character.

sprintf2

Formats a text string.

Syntax

ushort 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.
%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

ushort sprintf3(	
ushort *palette,*	Palette number to encode characters with (4 bits)
ushort *bank,*	Fix “bank” (character code MSB (4 bits))
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.

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(

ushort *x,*	X opsition on the fix layer
ushort *y,*	Y opsition on the fix layer
ushort *pal,*	Palette # to use
ushort *bank,*	Fix characer “bank” (character MSB, 4 bits)
char **buf***)**	String to print (8bit character format)

void fixPrint2(

ushort *x,*	X opsition on the fix layer
ushort *y,*	Y opsition on the fix layer
ushort *pal,*	Palette # to use
ushort *bank,*	Fix characer “bank” (character MSB, 4 bits)
char **buf***)**	String to print (8bit character format)

void fixPrint3(

ushort *x,*	X opsition on the fix layer
ushort *y,*	Y opsition on the fix layer
ushort *pal,*	Palette mod (will be added to character palette in string)
char **buf***)**	String to print (16bit character format)

void fixPrint4(

ushort *x,*	X opsition on the fix layer
ushort *y,*	Y opsition on the fix layer
ushort *pal,*	Palette mod (will be added to character palette in string)
char **buf***)**	String to print (16bit character format)

Explanation

Will print the supplied 8/16bit characters string to the fix layer, using supplied coordinates, palette and/or characters bank.

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

fixPrintf, fixPrintf1, fixPrintf2, fixPrintf3

Formats and text string and displays it on fix layer.

Syntax

void fixPrintf(

ushort *x,*	Palette number to encode characters with (4 bits)
ushort *y,*	Fix "bank" (character code MSB (4 bits))
ushort *pal,*	Pointer to destination buffer
ushort *bank,*	Pointer to format string (standard 8 bit characters format)
char **format,*	Format string pointer
…)	Extra arguments

void fixPrintf1(
** Same as **fixPrintf** **
)

void fixPrintf2(
** Same as **fixPrintf** **
)

void fixPrintf3(

ushort *x,*	Palette number to encode characters with (4 bits)
ushort *y,*	Fix "bank" (character code MSB (4 bits))
ushort *pal,*	Pointer to destination buffer
ushort *bank,*	Pointer to format string (standard 8 bit characters format)
char **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.**

fixPrintf: standard legacy function from the original neoDev archive.
fixPrintf1: Similar to **fixPrintf**, but internally using the faster **sprintf2.**
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 {	
ushort *baseSprite;*	Base sprite # used for this picture
ushort *basePalette;*	Base palette # used for this picture
short *posX;*	Current position, X axis
short *posY;*	Current position, Y axis
ushort *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.

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

pictureInfo

Structure holding picture information.

Syntax

typedef struct pictureInfo {

ushort *stripSize;*	Bytesize of each sprite tilemap (basically tileHeight*4)
ushort *tileWidth;*	Picture width, tiles unit
ushort *tileHeight;*	Picture height, tiles unit
paletteInfo ******palInfo;*	Pointer to related **paletteInfo**
**ushort ****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
ushort *baseSprite,*	Base sprite # to use
ushort *basePalette,*	Base palette # to use
short *posX,*	Picture initial X position
short *posY,*	Picture initial Y position
ushort *flip* **)**	Picture initial flip mode

Explanation

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

Return value

N/A

pictureMove

Updates position of a picture entity.

Syntax

void pictureMove(	
picture* *p,*	Pointer to picture handler to use
short *shiftX,*	X axis offset
short *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
ushort *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.

Return value

N/A

pictureSetPos

Sets position of a picture entity.

Syntax

void pictureSetPos(	
picture* *p,*	Pointer to picture handler to use
short *toX,*	New X position
short *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 {	
ushort *baseSprite;*	Base sprite # used for this scroller
ushort *basePalette;*	Base palette # used for this scroller
ushort *scrlPosX;*	Current scroll index, X axis
ushort *scrlPosY;*	Current scroll index, Y axis
scrollerInfo* *info;*	Pointer to the scrollerInfo struct of this scroller
ushort *config[32];*	Scroller configuration data - internal use
} scroller;	

Explanation

This is the base structure the library uses to handle scroller type 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.

scrollerInfo

Structure holding scroller information.

Syntax

typedef struct scrollerInfo {	
ushort strip*Size;*	Bytesize of each sprite tilemap (basically mapHeight*4)
ushort *sprHeight;*	Required sprite height to use (max 32)
ushort *mapWidth;*	Scroller width, tiles unit
ushort *mapHeight;*	Scroller height, tiles unit
paletteInfo **palInfo;*	Pointer to related **paletteInfo**
colorStreamInfo **csInfo;*	Pointer to related **colorStreamInfo**
ushort **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 (ushort) is (mapWidth*mapHeight)*2.

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

scrollerInit

Initialize a Scroller entity for use.

Syntax

void scrollerInit(

scroller* *s,*	Pointer to **scroller** handler to use
scrollerInfo* *si,*	Pointer to **scrollerInfo** structure
ushort *baseSprite,*	Base sprite # to use
ushort *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

Initialize a **scroller** handler for use.

Syntax

void scrollerInit(	
scroller* *s*,	Pointer to **scroller** handler
short *toX*,	Scroller X position
short *toY* **)**	Scroller Y position

Explanation

Sets scrolling position of designated **scroller** handler.

Return value

N/A

Animated sprites components

aSprite

Runtime handler for an animated sprite.

Syntax

typedef struct aSprite {

ushort *baseSprite;*	Base sprite # used for this animated sprite
ushort *basePalette;*	Base palette # used for this animated sprite
short *posX;*	Animated sprite current X position
short *posY;*	Animated sprite current Y position
ushort *animID;*	ID of last requested animation
ushort currentAnim*;*	ID of current animation
ushort 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
uint *counter;*	Internal frame update counter
ushort *repeats;*	Number of repeats done
ushort *tileWidth;*	Width of current frame, tiles unit
ushort *currentFlip;*	Current flip mode
ushort *flags;*	Flags

} aSprite;

Explanation

This is the base structure the library uses to handle animated sprites 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.

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:

15-8	7	6	5-2	1	0
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_MASK_MOVED             (0xfffe)
AS_MASK_FLIPPED           (0xfffd)
AS_MASK_MOVED_FLIPPED     (0xfffc)
AS_MASK_STRICT_COORDS     (0xffbf)
AS_MASK_NODISPLAY         (0xff7f)
```

spriteInfo, animStep, sprFrame

Structures holding animated sprites informations.

Syntax

typedef struct spriteInfo {	
ushort *frameCount;*	Total number of frames
ushort *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
short *shiftX;*	Frame X displacement from origin
short *shiftY;*	Frame Y displacement from origin
u**short** *duration;*	Number of frame to display
} animStep;	

typedef struct sprFrame {	
ushort *tileWidth;*	Frame width, tiles unit
ushort *tileHeight;*	Frame height, tiles unit.
ushort strip*Size;*	Bytesize of each sprite tilemap (basically tileHeight*4)
ushort **maps[4];*	Pointers to frame tilemaps (standard, flipX, flipY, flipXY)
} sprFrame;	

Explanation

spriteInfo, **animStep** and **sprFrame** 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.

Frame tilemaps size (ushort count) are (tileWidth*tileHeight)*2.

aSpriteAnimate

Performs animation updates on an **aSprite** entity.

Syntax

void aSpriteAnimate(

aSprite* *as* **)**	Pointer to **aSprite** handler to use

Explanation

Updates the **aSprite** handler 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** entity (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);`

Return value

N/A

aSpriteInit

Initialize an **aSprite** entity for use.

Syntax

void aSpriteInit(

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

Explanation

Initialize and prepare an **aSprite** handler 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.

Return value

N/A

aSpriteMove

Updates position of an **aSprite** entity.

Syntax

void aSpriteMove(

aSprite* *as,*	Pointer to **aSprite** handler
short *shiftX,*	X axis offset
short *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.

<u>**Note:**</u> 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** entity.

Syntax

void aSpriteSetAnim(	
aSprite* *as,*	Pointer to **aSprite** handler
ushort *anim* **)**	Animation ID
void aSpriteSetAnim2(	
aSprite* *as,*	Pointer to **aSprite** handler
ushort *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** entity.

Syntax

void aSpriteSetStep(

aSprite* *as,*	Pointer to **aSprite** handler
ushort *step* **)**	Step number

void aSpriteSetStep2(

aSprite* *as,*	Pointer to **aSprite** handler
ushort *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** entity.

Syntax

void aSpriteSetAnimStep(

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

void aSpriteSetAnimStep2(

aSprite* *as,*	Pointer to **aSprite** handler
ushort *anim,*	Animation ID
ushort *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** entity.

Syntax

void aSpriteSetFlip(

aSprite* *as,*	Pointer to **aSprite** handler
ushort *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** entity.

Syntax

void aSpriteSetPos(	
aSprite* *as,*	Pointer to **aSprite** handler
short *newX,*	New X position
short *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** entity 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.

Return value

N/A

Sprite Pools components

spritePool

Runtime handler for a sprite pool.

Syntax

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

ushort 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

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

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.

<u>Notes:</u> 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 <u>last+1</u> element, list will be read downward until null is found

SpritePoolDrawList isn't IRQ safe.
SpritePoolDrawList2 is an IRQ safe variant of **spritePoolDrawList**.

Return value

N/A

spritePoolInit

Initialize the supplied sprite pool handler.

Syntax

void spritePoolInit(

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

Explanation

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

If *clearSprites* is set to true, **spritePoolInit** will buffer a sprite clear of sprites *baseSprite* to *baseSprite+poolSize-1*.

Return value

N/A

Color steam components

colorStream

Runtime handler for a color stream.

Syntax

typedef struct colorStream {	
ushort *basePalette;*	Base palette # used for this color stream
ushort *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 {

ushort *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 {

ushort *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 handler.

Syntax

void colorStreamInit(

colorStream **cs,*	Pointer to **colorStream** handler
**colorStreamInfo ****csi,*	Pointer to related **colorStreamInfo** structure
ushort *basePalette,*	Base palette # to use
ushort *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 handler.

Syntax

void colorStreamSetPos(

colorStream **cs,*	Pointer to **colorStream** handler
ushort *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