The Removers'libraries documentation (a work supported by Jagware)

Seb/The Removers*

June 25, 2006

Abstract

You will find here the documentation of the Removers'libraries. Their goal is to ease the game writing process on the Atari Jaguar.

Preamble

First of all, I would like to send warmful greetings to all the Jagware team (`http://www.jagware.org/`). I hope these libraries will really help in the creation of 2D games for this wonderful Atari console.

Licence

The Removers' libraries are distributed under the terms of the new BSD License. The following copyright notice applies to every file[1] of the libraries:

*`http://removers.atari.org`

[1] except `jaguar.inc`

1 Quick overview of the libraries

The Removers'libraries are composed of several components that are intended to help the game programmer. These libraries have been written essentially in 68k assembler but some crucial parts have been written in GPU assembler. It is to be used with `mac` assembler by *Brainstorm*.

The libraries include:

1. Initialisation routines
2. Memory manager
3. Object list manager
4. Sprite manager
5. Joypad manager
6. Pseudo-random numbers generator

We will now go through each of the library.

2 Initialisation Routines

Smile, this part is the less polished one. We intend to improve greatly this part thanks to the experiment made by Zerosquare with the video registers.

2.1 Defined procedures

2.1.1 `IntIni`

This procedure was given by Atari. It initialises interruptions. The procedure `VblRoutine` is used as generic VBL routine (see below).

2.1.2 `VideoIni`

Once again, this procedure was given by Atari. It initialises the video system and sets some variables: **height**, **a_vdb**, **a_vde**, **width**, **a_hdb**, **a_hde**.

2.1.3 `VblRoutine` (generic)

This is the generic routine which is called every VBL. If the variable **VblRoutineAddr** is not null, then this variable gives the address of a custom routine which will be called by `VblRoutine`. This customised routine *should preserve* every single register except maybe d0 and a0.

Moreover, this procedure clears the variable **VblFlag**.

2.1.4 `WaitVbl`

This procedure waits for the next VBL to occur. Of course, it uses the variable **VblFlag** to do this synchronisation. It is a just a boolean flag, not a VBL counter.

2.1.5 `copy_code`

This procedure is intended to copy *tiny* piece of data (typically a GPU routine). The register d0 contains the size in bytes of the data (which must be long-word aligned), the register a0 contains the address of the data to be copied (word aligned), and the register a1 contains the address where to copy these data. Beware that when copying code in the GPU memory, the target address should be long word aligned also.

2.2 Defined macros

2.2.1 `init_prog_state`

This macro expands to the code (given by Atari) which should begin every single Jaguar program. It is a macro because at this point, the stack is not initialised.

2.2.2 `installVblRoutine` (custom Vbl routine)

The first argument is used to fill the variable **VblRoutineAddr**.

2.2.3 `clearVblRoutine` (custom Vbl routine)

This macro clears the variable **VblRoutineAddr**.

2.3 Variables

- **height**
- **a_vdb**
- **a_vde**
- **width**
- **a_hdb**
- **a_hde**
- **VblFlag**
- **VblRoutineAddr**

3 The Memory Manager

The memory manager has been written in order to ease memory management on the Jaguar. Indeed, you have to know that most every piece of data should be at least phrase aligned, when not double-phrase or quad-phrase aligned. Thanks to the memory manager, you will not be annoyed with this technical detail because every piece of memory allocated with the memory manager is quad-phrase[2] aligned. This memory manager is quite rudimentary but it implements basic functions.

[2] a phrase is 64 bits long, a quad phrase is 4 phrases

3.1 Defined procedures

3.1.1 `mm_init`

This is where all begin. This initialises the memory manager. The register a0 contains the start address of the heap and the register a1 contains the end address of the heap.

3.1.2 `mm_reset`

This reinitialises the memory manager with the initial values (it is a fast way to free every allocated blocks).

3.1.3 `mm_alloc` and `mm_alloc_clear`

These procedures allocate a memory block in the heap. As previously said, the allocated block is quad phrase aligned. In input, the register d0 contains the desired size of the block and in output, the register a0 contains the address of the allocated block. If it is not possible to allocate such a block, the program is stopped by the `illegal`[3] instruction.

In addition, the procedure `mm_alloc_clear` (slowly) clears the allocated buffer.

3.1.4 `mm_free`

This procedure is called when one wants to free a previously allocated block. In input, the register a0 contains the address of the block. You have to be aware that no special sanity check is performed and thus, the behaviour is unpredictable if the given address is not the address of a previously (and still) allocated block.

3.2 Variables

- **mm_free_blocks**
- **mm_start_address**
- **mm_end_address**

4 The Object List Manager

The object list manager defines some procedures and macros which eases the manipulation of object lists as used by the Object Processor (OP). For the moment, GPU objects and STOP objects cannot be dynamically created (the management of GPU objects is left as future work). Sprites are abstracted with a very rich data structure which should ease greatly the work of the programmer. The object list manager of course uses the memory manager to dynamically create objetcs. This library can be used as is (like in the Atomic game or the other early production of The Removers) or in conjunction with the sprite manager. Despite the refresh mecanisms offered by this library are quite rudimentary, it is sufficient for a game like Atomic. Note that some of the features described there can only be used through the sprite manager (in particular animated sprites or $y < 0$).

[3] it thus gives you hand back in the debugger

As the representation of sprites is central there, we first examine the data structure that abstracts sprites.

4.1 Sprite object representation

In the following, we use defined values in the Atari documentation.

```
        .offset 0
SPRITE_OBJ:                             ds.b    OBJECT_SIZEOF
SPRITE_RESTORE:                         ds.l    2
SPRITE_DATA:                            ds.l    1
SPRITE_PITCH:                           ds.l    1
SPRITE_RELEASE_TRANS_RMW_REFLECT:       ds.l    1
SPRITE_TYPE:                            ds.w    1
SPRITE_DEPTH:                           ds.w    1
SPRITE_IWIDTH:                          ds.w    1
        .long
SPRITE_XPOS:                            ds.w    1
SPRITE_YPOS:                            ds.w    1

SPRITE_DWIDTH:                          ds.w    1
SPRITE_HEIGHT:                          ds.w    1

SPRITE_INDEX:                           ds.w    1
SPRITE_FIRSTPIX:                        ds.w    1

        .long
SPRITE_ZOOM_DATA:                       ds.b    1
SPRITE_REMAINDER:                       ds.b    1
SPRITE_VSCALE:                          ds.b    1
SPRITE_HSCALE:                          ds.b    1

SPRITE_ANIM_ARRAY:                      ds.l    1
SPRITE_ANIM_DATA:
SPRITE_ANIM_COUNTER:                    ds.b    1
SPRITE_ANIM_SPEED:                      ds.b    1
SPRITE_ANIM_INDEX:                      ds.w    1
```

1. `SPRITE_OBJ` is the common header for every objects handled by the object list manager (private). This is in fact the objects as seen by the OP (and some more stuff to help the sprite manager in its task).
2. `SPRITE_RESTORE` is used by the object list manager to quickly refresh sprites (private)
3. `SPRITE_DATA` is the address of the graphics data (phrase aligned) of the sprite
4. `SPRITE_PITCH` defines how much data must be skipped (see TechRef Manual). The possible values are `O_NOGAP`, `O_1GAP`, `O_2GAP`, ..., `O_6GAP`
5. `SPRITE_RELEASE_TRANS_RMW_REFLECT` is the disjunction of the flags `O_RELEASE`, `O_TRANS`, `O_RMW` and/or `O_REFLECT`. The three interesting flags are `O_TRANS` for transparent sprites, `O_RMW` for saturated sprites (you can do blobs with this) and `O_REFLECT` for flipped sprites.
6. `SPRITE_TYPE` indicates the type of the sprite: either normal or scaled. The possible values are `BITOBJ` or `SCBITOBJ`.

7. `SPRITE_DEPTH` indicates the depth of the graphical data. The possible values are `O_DEPTH1`, `O_DEPTH2`, ..., `O_DEPTH32`.
8. `SPRITE_IWIDTH` gives the sprite width in phrases.
9. `SPRITE_XPOS` gives the X-coordinate of the sprite. It can be either negative or positive.
10. `SPRITE_YPOS` gives the Y-coordinate of the sprite. It can be either negative or positive (but negative Y-coordinate are best managed by the sprite manager (see below)).
11. `SPRITE_DWIDTH` gives the data width in phrases.
12. `SPRITE_INDEX` gives the start index in the CLUT for low-resolution sprites.
13. `SPRITE_FIRSTPIX` gives the first pair of pixels to display.
14. `SPRITE_REMAINDER` gives the Y-remainder for scaled sprites.
15. `SPRITE_VSCALE` gives the vertical scale for scaled sprites.
16. `SPRITE_HSCALE` gives the horizontal scale for scaled sprites.
17. `SPRITE_ANIM_ARRAY` gives, for animated sprites, the address of an array giving the address of the graphical data for each sprite of the animation (which must have all the same dimensions)
18. `SPRITE_ANIM_COUNTER` is a counter used to animate sprites. When it reaches 0, the next sprite in the array is used. The unit is a VBL.
19. `SPRITE_ANIM_SPEED` defines the speed of the animation.
20. `SPRITE_ANIM_INDEX` gives the index (15 lower bits) in the array of the next sprite to be displayed (the first sprite has index 0 or $n+1$ where n is the size of the array containing the animation). The higher bit is set when the animation loops.

As you can see, lots of stuff need to be defined. Fortunately, a macro and a procedure will help you to manage most common cases (and you are strongly advised to use them in order to stay compatible with future releases of the library).

4.2 Sprite creation

4.2.1 In data section

The macro `data_simple_sprite` defines a sprite in the DATA section of your program. The parameters are:

1. name of the label created to reference the sprite (in your code)
2. name of the label where the graphical data are located
3. width *in pixels* of the image (it is assumed that DWIDTH = IWIDTH)
4. height *in pixels* of the image
5. depth of the image (should be one of `O_DEPTH1`, `O_DEPTH2`, ..., `O_DEPTH32`)
6. optionnal: X-coordinate (default 0)
7. optionnal: Y-coordinate (default 0)

When this macro is expanded, some sanity checks are performed about the width.

4.2.2 Dynamically: in the heap

Thanks to the procedure `new_simple_sprite`, you can create dynamically in the heap a sprite. In input, the register a0 contains the address of the graphical data, d0 contains the data width in phrases (it is assumed that DWIDTH = IWIDTH), d1 contains the height, d2 contains the X-coordinate, d3 contains the Y-coordinate and d4 contains the data depth. In output, a0 contains the address of the freshly created sprite.

4.2.3 Other kinds of sprites

It is highly recommended to first create a sprite with the help of one of the two previously explained method and then change only the desired values.

4.3 Object list creation

4.3.1 Branch objects

The procedure `new_branch_object` allows to freshly create branch objects. A branch object can be seen as a "if-then-else" construction for object lists. The procedure takes as argument in a0 the address of the "then" branch, in a1 the address of the "else" branch, in d0 the Y-coordinate used by the test condition and in d1 the condition for the test (which can be `O_BREQ`, `O_BRGT`, `O_BRLT` or `O_BRHALF`). The register a0 will then contain the address of the freshly created branch object.

As a facility, there is also a routine to create the first branch object that should be at the beginning of every object lists. This procedure, which is `first_branch_object`, need as arguments in a0 the address of the stop object, in a1 the address of the object list, in d0 the value **a_vdb** and in d1 the value **a_vde**. As before, the address of the branch object is in a0 on return.

4.3.2 Sprite objects

Once a sprite is created using the previously explained methods, you need to convert it in a sprite object (scaled or not) and this is done with the help of the procedure `finalize_sprite_object`. This procedure allows to add a sprite object at the beginning of an object list. In a0 is the address of the sprite structure and in a1 is the address of the object list. You have to understand that a sprite structure is also a sprite object (in OO language, we would say that `sprite structure` inherit from `sprite object`).

4.3.3 Stop object

Once again, a macro is there to help you... With the macro `data_stop_object`, you insert in the data section a STOP object.

4.3.4 Final warning

Once again, you should exclusively use the procedures described there to stay compatible with next versions of this library (even for as dummy objects as STOP ones!)

4.4 Refreshing objects

As we mentionned earlier, the refresh mechanism offered by this library are rudimentary or not efficient. Thus, it is of course possible to recompute all the fields of sprite objects thanks to the routine `update_sprite_object` but you will probably prefer a faster refresh mechanism that just update the fields crashed by the OP or that only refresh in addition the coordinates of the sprites. All the refresh routines takes in a0 the address of the sprite structure. Here are the different refresh mechanism offered by the library:

- `update_sprite_object`: refresh all fields (very slow)
- `refresh_sprite_object`: refresh only the crashed fields (very fast)
- `refresh_coords_sprite_object`: refresh in addition the coordinates
- `refresh_gfx_sprite_object`: refresh in addition the DATA field

4.5 Animated sprites

If you use this library through the sprite manager, you will be able to have animated sprites almost for free. The format of animation handled by the sprite manager is simple but should be sufficient in most of the cases.

The field `SPRITE_ANIM_SPEED` represents the number of calls of the procedure `refresh_sprite_list` (see below) between the change of two images of the sprite.

The sprite is animated if the address indicated by `SPRITE_ANIM_ARRAY` is not null. In this case, it is the address of an array containing the address of the sequence of images to be used for the animation. The array ends by a null pointer.

For the moment, there is only one kind of animation available: cyclic ones. We think it is sufficient to manage every kind of animation by using an array of data pointers that encodes other kinds of animation (reversed or ping-pong).

You have access to the index in the array of the currently used graphical data through the variable `SPRITE_ANIM_INDEX`. You can also use this field to test if the animation has looped or not.

To set a new animation, you can use the procedure `set_sprite_animation` which takes in d0 the speed of the animation, in a0 the address of the sprite and in a1 the address of the array.

To clear an animation, simply use the macro `clear_sprite_animation` which takes as first argument an address register which points to the sprite. In this case, do not forget to change the field `SPRITE_DATA` if necessary... (this field is indeed modified when animation is active)

You can also activate or deactivate an animation with the two macros `sprite_animation_on` and `sprite_animation_off` which take as first argument an address register which points to the sprite.

5 The Sprite Manager

Here comes the master piece of code! With the sprite manager, you can easily manage sprites by just adding them or removing them from the sprite manager structure. A fast refresh mechanism (written in GPU assembler) will allow you to have animated sprites very easily.

5.1 Initialisation of the sprite manager

The first thing to do is to initialise the sprite manager. That is the aim of the `init_sprite_manager` procedure. This procedure copies the GPU code and initialises the variable **y_origin**. The variable **y_origin** is simply twice the variable **a_vde**. It is used by the object list manager or the sprite manager to make the Y-coordinate 0 to be the top of the visible data. Negative Y-coordinate are handled by the sprite manager (only non-scaled sprites) offering some comfort to the game programmer.

5.2 Creating a new sprite list

Once the sprite manager is initialised, you can create a new sprite list with the procedure `new_sprite_list` which gives you back in a0 the address of the freshly created "list" (called after SM-structure).

The SM-structure is organized in 16 layers[4] (this can be changed by modifying the EQU in the library) so that sprites at layer 0 are under those at layer 1 which are under those at layer 2 and so on...

5.3 Deleting a sprite list

You will probably need at a moment or another to free the memory allocated by a SM-structure. This is what the procedure `delete_sprite_list` exactly does. The address of the SM-structure is given in register a0. Of course, the sprites contained in the SM-structure are not freed (since some of them could be in DATA segment and thus not in the heap).

5.4 Adding a new sprite

With the procedure `cons_sprite_at_depth`, you can add a sprite to the SM-structure. This procedure takes in a0 the address of the SM-structure, in a1 the address of the sprite and in d0 the layer where to add it in the SM-structure.

5.5 Removing a sprite

You can remove a sprite from the SM-structure it is attached to by using the procedure `remove_sprite` which takes as an argument in a0 the address of the sprite and that's all!

5.6 Installing the sprite list

You can install the sprite list corresponding to the SM-structure with the procedure `install_sprite_list` which takes in a0 the address of the SM-structure. Installing means setting OLP so that your data are displayed on screen.

5.7 Refreshing the list

The procedure `refresh_sprite_list` refreshes (with the help of the GPU) all the sprites of the currently displayed SM-structure. (the address of the current SM-structure is put in the variable **current_sprite_list**).

[4] we will also use the term depth or level to express this same concept

Since it could be harmful to add or remove a sprite while the SM-structure is refreshed, there is a mutex (lock) attached to each SM-structure. Your 68000 code can wait the SM-structure is refreshed with the procedure `wait_refresh_list` which takes in a0 the address of the SM-structure. Note that two consecutive calls to this synchronisation procedure will hang your program! In the actual state of the code, we strongly think that only sprite removal can lead to concurrency problems.

The refreshed fields are `DATA`, `XPOS`, `YPOS`, `HEIGHT`, `HSCALE`, `VSCALE` and `REMAINDER`.

5.8 Give me back the GPU, please!

Actually, the GPU is not completely stolen by the sprite manager (and as far as I can say, it is far from that!). There is a GPU driver included in the sprite manager library. This driver simply reads a parameter at variable **GPU_SUBROUT_ADDR** and if it is not null, this represents the address of a GPU subroutine to jump to. The routine should end by giving hand back to the GPU driver. This is simply achieved by reading the return address on the stack (r31) and jump to this address. The following code implements this:

```
load    (r31),r0
addq    #4,r31
jump    t,(r0)
nop
```

Registers need not to be preserved.

The refresh mechanism of the list is actually implemented by an interruption. The interruption that refreshes the list does not save registers (but actually, this could be changed very easily if needed). That is why the GPU is set to use bank 1 when running normal GPU code and to use bank 0 for the interrupt (so, to be more precise, only bank 0 is alterated by the refresh interrupt).

So, to sum up, you can have your own GPU routines cohabiting with the sprite manager refresh routine provided it only uses the active bank of registers (i.e. bank 1) and ends with the code given above. Of course, you should take care of not erasing the sprite manager and GPU driver code. The available GPU ram for your custom routines starts at **GPU_FREE_RAM**. The only other limitation is that you cannot use GPU interrupts anymore.

You have also to know that stacks (user and interrupt ones) allocated are really short (the default size is one long word). You might want to change this but we do not see the reasons why you would want to.

5.9 Tell me more about the refresh mechanism...

The sprite manager can run in different modes. The default one is to use an OP interruption for refreshing the list (yes, GPU objects can be useful!). In this case, it is not even necessary to call thee routine `refresh_current_list`.

Unfortunately, this mode is not compatible with the Jaguar emulator Project Tempest (GPU objects are not correctly emulated). Thus, if you develop your project using Project Tempest, you will probably find comfortable to be able to still use it. You can configure the sprite manager so that it uses a CPU interrupt instead. In this case, you will need to call the routine `refresh_current_list`.

For testing purpose, it is also possible to use the GPU driver to refresh the list but we strongly recommend to not use this mode.

6 Joypad manager

The joypad manager offers you only the procedure `read_joypads` which reads the state of the 8 joypads ports in the variable **joypad_1**, ..., **joypad_8**. Defined constants through EQU directives then help the programmer to test the current state of each joypad. Note that a pushed button is indicated by a cleared bit and conversely.

7 Pseudo-random Numbers Generator

The pseudo-random number generator included is a ranrot generator. It is based on the article *Chaotic Random Number Generators with Random Cycle Lengths* of Agner Fog.

You have first to initialise the PRNG with the help of the procedure `random_init`. Then, to get a new 32 bits pseudo-random number in d0, simply call the procedure `random_next`. Note that there is also a macro `gpu_random_next` that expands to GPU code that compute the next pseudo-random number. This macro takes four arguments: the first should be either r14 or r15, the second, third and fourth register are temporary registers used to compute the new value which is then put in the fourth argument.

8 Example of use

8.1 Making blobs with RMW mode

The piece of code below illustrates the usage of all these libraries (except the joypad reading library). It implements shrinking and growing blobs. The saturation is made by the OP thanks to the RMW bit in CRY mode. The images have been converted with the image converter we have written and which is available on our website.

```
.68000
.text

;; example of use of The Removers libraries
;; we do NB_BLOBS blobs with help of RMW mode of sprites

BLOB_W equ 48
BLOB_H equ 48
BLOB_RADIUS equ BLOB_W/2

;; unfortunately, the OP cannot manage more than that
NB_BLOBS equ 27

SIZE_STACK equ 4*1024

include "prelude.s"

;; init state
```

```
init_prg_state
;; memory init
;; the heap starts at the end of the BSS section
move.l #Bss_end,a0
move.l #INITSTACK-SIZE_STACK,a1
bsr mm_init
;; video init
bsr VideoIni

;; for the moment, nothing to display
move.l #stop_object,d0
swap d0
move.l d0,OLP

;; changing video mode
move.w #CRY16|CSYNC|BGEN|PWIDTH4|VIDEN,VMODE

;; interrupts init
bsr IntIni

;; background color (BLACK in CRY)
move.w #$8800,BG
main:
;; init the PRNG
bsr random_init
;; init the sprite manager
bsr init_sprite_manager
;; init the blobs
bsr init_blobs
;; install the list
move.l sm_list,a0
bsr install_sprite_list
;; we put the refresh routine in VBL
;; we can instead also call it after WaitVbl
installVblRoutine #refresh_sprite_list
.loop:
* move.w #$8888,BG ; to see available 68k time!
bsr WaitVbl
* move.w #$8800,BG
;; computing next frame
bsr do_blobs
bra.s .loop
rts

init_blobs:
;; compute center coordinates and clipping coordinates
move.l #blob_array,a3
move.w width,d0
lsr.w #2,d0 ; PWIDTH4
move.w d0,d1
sub.w #2*BLOB_RADIUS,d1
move.w d1,xmax
lsr.w #1,d0
sub.w #BLOB_RADIUS,d0
```

```
move.w d0,xcenter
move.w height,d0
move.w d0,d1
sub.w #2*BLOB_RADIUS,d1
move.w d1,ymax
lsr.w #1,d0
sub.w #BLOB_RADIUS,d0
move.w d0,ycenter
;; allocate the blobs sprites
moveq #NB_BLOBS-1,d7
.create_blob:
move.l #blob48_0_gfx,a0
move.w #BLOB_W/4,d0
move.w #BLOB_H,d1
move.w xcenter,d2
move.w ycenter,d3
move.w #0_DEPTH16,d4
bsr new_simple_sprite
move.l #0_TRANS|0_RMW,SPRITE_RELEASE_TRANS_RMW_REFLECT(a0)
move.l #0_RMW,SPRITE_RELEASE_TRANS_RMW_REFLECT(a0)
move.l a0,(a3)+
move.l #blob48_animation,a1
bsr random_next
move.l -4(a3),a0
and.w #%11,d0 ; random speed
bsr set_sprite_animation ; animate each blob
dbf d7,.create_blob
;; we now build the list through the sprite manager
;; get a new SM-structure
bsr new_sprite_list
move.l a0,sm_list ; save the address
;; inserts every blob
move.l #blob_array,a3
moveq #NB_BLOBS-1,d7
.create_list:
move.l (a3)+,a1 ; blob sprite
moveq #0,d0 ; at level 0
bsr cons_sprite_at_depth ; add it!
dbf d7,.create_list
rts

do_blobs:
;; compute next frame
move.l #blob_array,a3
moveq #NB_BLOBS-1,d7
moveq #0,d6
.refresh:
;; get a 32 bit random number
bsr random_next
move.w d0,d1
;; modulo 8
and.w #$7,d0
;; divide by two
lsr.w #1,d0
```

```
;; add the rotated bit
addx.w d6,d0
;; we have then a random number between 0 and 5 in d0
;; we get another random number between 0 and 5 in d1
lsr.w #8,d1
and.w #$7,d1
lsr.w #1,d1
addx.w d6,d1
;; here it is
;; we now translate them in -2 .. 3
subq.w #2,d0
subq.w #2,d1
;; get the blob sprite
move.l (a3)+,a0
;; get x
move.w SPRITE_XPOS(a0),d2
;; add random number
add.w d0,d2
;; and clip it
ble.s .reset_x
cmp.w xmax,d2
ble.s .ok_x
.reset_x:
move.w xcenter,d2
.ok_x:
;; save x
move.w d2,SPRITE_XPOS(a0)
;; same for y
move.w SPRITE_YPOS(a0),d3
add.w d1,d3
ble.s .reset_y
cmp.w ymax,d3
ble.s .ok_y
.reset_y:
move.w ycenter,d3
.ok_y:
move.w d3,SPRITE_YPOS(a0)
;; that's it!
dbf d7,.refresh
rts

.data

.phrase
blob48_0_gfx: incbin "blob48_0.cry"
.phrase
blob48_1_gfx: incbin "blob48_1.cry"
.phrase
blob48_2_gfx: incbin "blob48_2.cry"
.phrase
blob48_3_gfx: incbin "blob48_3.cry"
.phrase
blob48_4_gfx: incbin "blob48_4.cry"
.phrase
```

```
blob48_5_gfx: incbin "blob48_5.cry"
.phrase
blob48_6_gfx: incbin "blob48_6.cry"
.phrase
blob48_7_gfx: incbin "blob48_7.cry"
.phrase
blob48_8_gfx: incbin "blob48_8.cry"
.phrase
blob48_9_gfx: incbin "blob48_9.cry"
.phrase
blob48_10_gfx: incbin "blob48_10.cry"
.phrase
blob48_11_gfx: incbin "blob48_11.cry"
.phrase
blob48_12_gfx: incbin "blob48_12.cry"

blob48_animation:
;; ping-pong animation for the blobs
dc.l blob48_0_gfx
dc.l blob48_1_gfx
dc.l blob48_2_gfx
dc.l blob48_3_gfx
dc.l blob48_4_gfx
dc.l blob48_5_gfx
dc.l blob48_6_gfx
dc.l blob48_7_gfx
dc.l blob48_8_gfx
dc.l blob48_9_gfx
dc.l blob48_10_gfx
dc.l blob48_11_gfx
dc.l blob48_12_gfx
dc.l blob48_11_gfx
dc.l blob48_10_gfx
dc.l blob48_9_gfx
dc.l blob48_8_gfx
dc.l blob48_7_gfx
dc.l blob48_6_gfx
dc.l blob48_5_gfx
dc.l blob48_4_gfx
dc.l blob48_3_gfx
dc.l blob48_2_gfx
dc.l blob48_1_gfx
dc.l 0

.bss
.even
xmax: ds.w 1
xcenter: ds.w 1
ymax: ds.w 1
ycenter: ds.w 1
;;
.long
sm_list: ds.l 1
;; blobs
```

```
.long
blob_array: ds.l NB_BLOBS
Bss_end:
```

8.2 Affine transformation with the blitter

In the official distribution of The Removers'libraries, you can also find an example of use of the blitter. It uses the joypad library and shows you how to set common variables for making affine transformations (such as rotations for example) with the blitter.

The equation of the affine transformation displayed is

$$\begin{pmatrix} X \\ Y \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & e \\ c & d & f \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} X' \\ Y' \\ 1 \end{pmatrix}$$

where X, Y are the source coordinates and X', Y' are the target coordinates.

See the source code for more details.

9 Summary

Here we sum up every (public) procedure/macro and their parameters. When nothing is written for output registers, this means that the routine does not save them nor restore them but use them.

9.1 init.s

1. | | |
   |---|---|
   | Name | `init_prg_state` |
   | Input | none |
   | Output | none |
   | Note | macro |

2. | | |
   |---|---|
   | Name | `copy_code` |
   | Input | d0 size (long word aligned)
a0 source address
a1 target address |
   | Output | d0
a0
a1 |
   | Note | |

3. | | |
   |---|---|
   | Name | `VideoIni` |
   | Input | none |
   | Output | **a_vde**
a_vdb
a_hde
a_hdb
width
height
y_origin |
   | Note | **y_origin** is used by the sprite manager to handle negative Y-coordinates |

4. | | |
   |---|---|
   | Name | `IntIni` |
   | Input | none |
   | Output | none |
   | Note | |

5.

Name	`installVblRoutine`
Input	1st arg address of custom VBL routine
Output	**VblRoutineAddr**
Note	macro

6.

Name	`clearVblRoutine`
Input	none
Output	**VblRoutineAddr**
Note	macro

7.

Name	`VblRoutine`
Input	none
Output	none
Note	generic VBL routine

8.

Name	`WaitVbl`
Input	none
Output	none
Note	VBL synchronisation

9.2 memory.s

1.

Name	`mm_init`
Input	a0 heap address start a1 heap address end
Output	d0 MM_OK or MM_ERROR d1 d2 a0 a1
Note	

2.

Name	`mm_reset`
Input	none
Output	see `mm_init`
Note	

3.

Name	`mm_alloc`
Input	d0 desired size of the block
Output	d0 d1 d2 a0 address of the freshly allocated block in the heap (or null if impossible) a1 a2
Note	actually, it will result in an illegal instruction if memory allocation is impossible

4.

Name	`mm_alloc_clear`
Input	same as `mm_alloc`
Output	same as `mm_alloc`
Note	in addition, the block is cleared

5.

Name	`mm_free`
Input	a0 address of the block to free
Output	d0 d1 d2 a0 a1 a2
Note	the given address should be valid

9.3 object_list.s

1.

Name	`data_simple_sprite`
Input	1^{st} arg sprite label name (created) 2^{nd} arg graphical data label 3^{rd} arg width (in pixels) 4^{th} arg height (in pixels) 5^{th} arg depth of data 6^{th} arg X-coordinate (optionnal) 7^{th} arg Y-coordinate (optionnal)
Output	none
Note	macro expanded in DATA segment

2.

Name	`data_stop_object`
Input	1^{st} arg stop object label name (created)
Output	none
Note	macro expanded in DATA segment

3.

Name	`new_branch_object`
Input	d0 YPOS d1 condition code a0 “then” object a1 “else” object
Output	d0 d1 a0 address of the freshly created branch object a1
Note	

4.

Name	`first_branch_object`
Input	d0 **a_vdb** d1 **a_vde** a0 address of STOP object a1 address of object list
Output	d0 d1 a0 address of the freshly created branch object a1
Note	

5.

Name	`update_sprite_object`
Input	a0 sprite address
Output	none
Note	full refresh of sprite fields

6.

Name	`finalize_sprite_object`
Input	a0 sprite address a1 link address
Output	none
Note	

7.

Name	`fast_refresh_nonscaled_sprite_object`
Input	1st arg address register (sprite address)
Output	1st arg
Note	macro

8.

Name	`fast_refresh_scaled_sprite_object`
Input	1st arg address register (sprite address)
Output	1st arg
Note	macro

9.

Name	`refresh_sprite_object`
Input	a0 sprite address
Output	a0
Note	fast refresh of scaled sprites

10.

Name	`fast_refresh_coords_nonscaled_sprite_object`
Input	1st arg address register (sprite address) 2nd arg data register 3rd arg data register
Output	1st arg 2nd arg 3rd arg
Note	macro

11.

Name	`fast_refresh_coords_scaled_sprite_object`
Input	1st arg address register (sprite address) 2nd arg data register 3rd arg data register
Output	1st arg 2nd arg 3rd arg
Note	macro

12.

Name	`refresh_coords_sprite_object`
Input	a0 sprite address
Output	a0 d0 d1
Note	refresh coords of scaled sprites

13.

Name	`refresh_gfx_sprite_object`
Input	a0 sprite address
Output	d0 d1 a0 a1
Note	refresh DATA of scaled sprites

14.

Name	`new_simple_sprite`
Input	d0 width (in phrases) d1 height (in pixels) d2 X-coordinate d3 Y-coordinate d4 depth of graphical data a0 graphical data address
Output	a0 address of freshly created sprite
Note	

15.

Name	`set_sprite_animation`
Input	d0 animation speed a0 sprite address a1 animation array address
Output	none
Note	

16.

Name	`clear_sprite_animation`
Input	1st arg address register
Output	none
Note	macro

17.

Name	`sprite_animation_on`
Input	1st arg address register
Output	none
Note	macro

18.

Name	`sprite_animation_off`
Input	1st arg address register
Output	none
Note	macro

9.4 sprite_manager.s

1.

Name	`init_sprite_manager`
Input	none
Output	d0 a0 a1
Note	

2.

Name	`new_sprite_list`
Input	none
Output	a0 address of freshly created SM-structure
Note	

3.

Name	`wait_refresh_list`
Input	a0
Output	a0
Note	two consecutive calls (without a refresh between) will hang your program

4.

Name	`cons_sprite_at_depth`
Input	d0 level (also called depth) a0 SM-structure address a1 sprite address
Output	none
Note	

5.

Name	`remove_sprite`
Input	a0 sprite address
Output	none
Note	

6.

Name	`delete_sprite_list`
Input	a0 SM-structure address
Output	none
Note	*sprites are (of course) not freed*

7.

Name	`install_sprite_list`
Input	a0 SM-structure address
Output	none
Note	

8.

Name	`refresh_sprite_list`
Input	none
Output	none
Note	

9.

Name	`JumpGPUSubRoutine`
Input	1st arg address of the subroutine in GPU ram
Output	none
Note	a macro that tells the GPU driver to execute a GPU subroutine

9.5 joypad.s

1.

Name	`read_joypads`
Input	none
Output	**joypad_1** **joypad_2** **joypad_3** **joypad_4** **joypad_5** **joypad_6** **joypad_7** **joypad_8**
Note	use defined EQUs to test joypad states after!

9.6 random.s

1.

Name	`random_init`
Input	none
Output	none
Note	call this before using the PRNG

2.

Name	`random_next`
Input	none
Output	d0 new 32 bits pseudo-random number d1 d2 a0
Note	

3.

Name	`gpu_random_next`
Input	1^{st} arg r14 or r15
	2^{nd} arg a register
	3^{rd} arg a register
	4^{th} arg a register
Output	1^{st} arg
	2^{nd} arg
	3^{rd} arg
	4^{th} arg new 32 bits pseudo-random number
Note	GPU macro

9.7 prelude.s

Just include prelude.s at the beginning of your program. It will include all the libraries (and jaguar.inc) at the very beginning of the text segment and a label **start** is created after all the included code (this is necessary to jump over the included code). See the example to see how this is intended to be used.

10 Future work

Of course, these libraries are not perfect and maybe still contain bugs. For these reasons, we are interested in feedback concerning the use of these libraries. You can join me on the Jagware forum (or see my email on my website).

One improvement which could be great for game developper is the full support of scaled sprites (instead of limited support as in the current library).

Another improvement we could envisage is to add some kind of support for collision tests. For the moment, it is not clear to us how to integrate this in the library (but we have a GPU routine that do the collision test).

Finally, we envisage also to add double buffering to our sprite manager but obviously it is quite a big amount of additionnal work.

Your comments, questions and suggestions are welcome. Feel free to write to `SebRmv_@_jagware.org` (remove `_` to get the right email address).

11 Final words

I would like to thank particularly GT Turbo for the interesting discussions we had and the ideas that have outcome from them. I would like also to thank SCPCD, Zerosquare, Mariaud, Azrael, Fredifredo, Orion_, MK, Fadest, Mathias Domin and *of course* Stabylo.