RAPTOR Engine

User Manual

http://reboot.atari.org

Introduction

The RAPTOR Engine™ is a software package intended for use by developers on the Atari Jaguar 64bit Multimedia System. It provides a HAL (Hardware Abstraction Layer) to allow you to concentrate on your game rather than how the Jaguar hardware functions. Technology has been built into the RAPTOR Engine™ to simplify its use, whilst providing flexibility and minimizing main system bus access.

Licensing

Definitions

- "The Software" refers to the RAPTOR Engine, which is provided as an assembled binary object for use on the Atari jaguar 64bit Multimedia System.
- "U235SE" refers to the U235 Sound Engine by U235 (http://www.u-235.co.uk/developer/sound-engine/).
- "Author(s)" refers to the group REBOOT (http://reboot.atari.org).

Licence

This software is provided free of charge to anyone and everyone. REBOOT accepts no responsibility for damage or loss by its use or misuse. REBOOT grants you the right to use this software within your own works provided that:

- Clear identification of the use of this software is included within your own works, either by use of the approved logo provided, or textually.
- The identification of the use of this software must appear within the digital works in a manner that is visible to the end user and upon any physical packaging.
- The software may not be reverse engineered or modified without prior consent of the author(s).
- No source code forming any part of The Software is to be distributed without explicit permission from the author(s).

U235 Sound Engine

The RAPTOR Engine makes use of the U235 Sound Engine to provide audio functions. Use and inclusion of this library must also recognise and abide by the licence provided by U235 for their Sound Engine.

Features

- Entirely GPU based RISC core
- Kudos Ware Licensing model
- Text functions
- Up to 512 independent, fully programmable Objects with 16.16 sub-pixel positioning
- Animation Engine complete with scaling and mirroring functions
- Full control over Objects using a HAL
- Human Readable object composition lists
- Dynamic Branching Object Support
- Dynamic Object Culling
- CLUT/RGB/CRY support
- Collision handling
- Particle functions
- MemoryTrack and High Score management
- Tile Map Engine (with 16.16 sub-pixel accuracy)
- Full U235 Sound Engine integration

Disclaimer

Throughout this document you will find links to external websites. Although we make every effort to ensure these links are accurate, up to date and relevant, Reboot cannot take responsibility for pages maintained by external providers.

Before you start

Virtual Jaguar is an Atari Jaguar Emulator. The latest stable release can be obtained from https://icculus.org/virtualjaguar/ - however up to the minute releases are also available from: http://outrage.the-crow.co.uk/builds/

RMAC and RLN are the assembler and linker used to build the RAPTOR examples. The latest versions can be obtained from http://outrage.the-crow.co.uk/builds/

The latest versions of the above three binaries (as of the time of release) are included in the RAPTOR package with kind permission of their Author(s)

These files should be placed in the EXAMPLES\BIN folder.

Building the Examples

To help illustrate the operation and use of the RAPTOR Engine several demonstration programs are included. These examples will guide the user from first steps with a “Hello World” example through to a complete game using most of the RAPTOR functions. These examples will be documented at the end of this manual.

Inside each of the example folders is a BUILD.BAT. Running this batch file will build the example, link it to the RAPTOR and U235 SoundEngine objects and then execute them using VirtualJaguar (Providing it has been set up as detailed above)

Credits

Code and Documentation:	Cyrano Jones
Testing:	Matmook
Additional Testing:	Shamus, ggn, Zerosquare, SCPCD, Linkovitch
Logo and Artwork:	sh3-rg

Greetings and Thanks

In no special order:

U-235

Without the Sound Engine and permission to bundle with RAPTOR Engine this would be a very silent endeavour.

Reboot

ggn, sh3-rg, RemoWilliams – for sticking with me through thick and thin.

Jagware

Special thanks to SCPCD, ZeroSquare for the development and debugging assistance, to Matmook for developing this further (I hope to include RMOTION in a future update) and to GT-Turbo for his enthusiasm.

Shamus and the VirtualJaguar team

For fantastic a fantastic emulator that just keeps on giving. Here's to continuing to provide bug reports and getting fixes in the future!

Sinister Developments

For the Object Creation routines.

RAPTOR Functions, Equates and Variables

All RAPTOR functions, equates and variables are listed in the RAPTOR.INC file, which should be included in all your RAPTOR applications.

The EQUATES contain human readable values designed to make writing (and later, reading) your source code easier.

All RAPTOR functions start with ***RAPTOR_*** these will be detailed on the following pages.

All RAPTOR variables start with ***raptor_*** so that they can be distinguished from functions.

The .INC file can be found in RAPTOR\INCS.

RAPTOR Application Files

RAPTOR applications consist of the following files:

_RAPAPP.S	Your code will go into this file
_RAPINIT.S	Script file defining the composition of all display objects and lists
_RAPPIXL.S	Script file defining particle effect parameters
_RAPU235.S	Script file defining U235 Sound Effects
RAPTOR.O	The RAPTOR Library Object File
DSP.O	The U235 Sound Engine Object File
EXTERNAL FILES	External data files, eg, music Module, Audio files, Graphics

These files will be explored in the *Examples section* found later on in the manual.

RAPTOR Functions Calls

This section will describe all the function calls, along with their expected input parameters and their outputs. Example code will be provided in the *Examples section* found later on in the manual.

RAPTOR_HWinit (EX-01a)

This function is called at the start of every RAPTOR application. It will configure the system to the specified video mode, install the RAPTOR Core to the GPU local memory, create the initial database of objects from the *_RAPINIT.S* file, initialize and clear the Particle system and return the system in a usable state, ready for applications to run.

Expected Inputs:

raptor_vidmode	Video mode – CRY16, RGB16, RGB24, DIRECT16
raptor_videnable	Video Enable Mask
raptor_varmod	VARMOD On or Off
raptor_partbuf_x	Width of Particle & Text window in pixels
raptor_partbuf_y	Height of Particle& Text window in pixels
raptor_top_of_bss	Start of the BSS section in the binary
raptor_trashram	Start of work RAM (end of the BSS section)
raptor_MTwork	16k workspace for the MemoryTrack module
raptor_uvbi_jump	Address of the user VBI hook
raptor_poobjl	Address of the Pre Object List hook
raptor_probjl	Address of the Post Object List hook
raptor_8x8_addr	Address of the 8x8 font data
raptor_8x16_addr	Address of the 8x16 font data

raptor_16x16_addr	Address of the 16x16 font data
raptor_partipal	Address of the BMP file containing the font/particle palette data
raptor_pgfx	Address of the Particle & Text window bitmap
raptor_pgfxe	End address of the Particle & Text window bitmap
raptor_spritetab	Address of the RAPTOR Object data table
raptor_mtapp	Pointer to the MemoryTrack Application Name
raptor_mtfn	Pointer to the MemoryTrack File Name
raptor_inittab	Pointer to the data in the _RAPINIT.S file
raptor_samplebank_ptr	Pointer to the data in the _RAPU235.S file

If using the Tile Map functions, the following inputs are also expected:

raptor_maptop_obj	RAPTOR Object index for the first object in the map
raptor_tiles_x	Width in pixels of a single map tile
raptor_tiles_y	Height in pixels of s single map tile
raptor_tilesperx	Number of horizontal tiles to draw on screen
raptor_tilespery	Number of vertical tiles to draw on screen
raptor_mapwidth	Width (in tiles) of the map data
raptor_mapheight	Height (in tiles) of the map data
raptor_tilerem_mask	Mask for calculating position (tile width -1)
raptor_mapbmptiles	Pointer to the tile bitmap data

If using the Particle functions, the following inputs are also expected:

raptor_maxparts	Maximum number of particles to process
raptor_pdriftx	Drift (horizontal) to add to all particles every update
raptor_pdrifty	Drift (vertical) to add to all particles every update
raptor_partitab	Pointer to particle database

Outputs:

None

RAPTOR_start_video (EX-01a)

This function starts video generation. Before this function is called the screen will remain blank.

Expected Inputs:

None

Outputs:

None

RAPTOR_setlist (EX-01a / EX-07a)

This function selects a RAPTOR list to be displayed. The lists are defined in the *_RAPINIT.S* file and will be described later in the manual.

Expected inputs:

Do	List number to display

Outputs:

None

RAPTOR_particle_init

This function initializes the Particle Engine, which is also used for Text Output. It will configure the Particle Table database and clear the bitmap used for rendering particles and text. The particle and text bitmap will always use CLUT 15 (colours 240-256) in the 256 colour palette.

Its colour palette is defined in the Windows BMP file *partipal.bmp*, and the font data is defined in the three font files in the FONTS folder, *F_8x8.BMP*, *F_8x16.BMP* and *F_16x16.BMP*.

Different fonts for each size can be stacked vertically below each other in the bitmap files, as can be seen if they are opened in a picture viewer.

Expected inputs:

None

Outputs:

None

RAPTOR_particle_clear

This function will clear the particle and text layer bitmap.

Expected inputs:

None

Outputs:

None

RAPTOR_print (EX-01a)

This function allows text to be sent to the display window. Text can be one of three font sizes, and can use any number of fonts as defined in the font Windows BMP files.

Expected inputs:

A0	Pointer to the text string
D0	X position to print (Note: will be rounded to an even value)
D1	Y position to print
D2	Font size (0=8x8, 1=8x16, 2=16x16)
D3	Font index (0=1st font in BMP, 1=2nd, etc.)

Outputs:

None

Print Commands

The print string can contain additional commands beyond character graphics to manipulate the output string.

raptor_t_quit	Used to terminate the string
raptor_t_lf	Used to issue a line feed
raptor_t_font_idx	Used to change the font index
raptor_t_font_siz	Used to change the font size
raptor_t_pos_xy	Used to re-position the print position
raptor_t_home	Used to home the cursor to the top left
raptor_t_right	Used to subspace offset the text

RAPTOR_move_palette (EX-02b)

This function will move the palette from the last converted image to the specified location.

Expected inputs:

A1	Address to copy palette data to

Outputs:

None

RAPTOR_GFXConvert (EX-02a)

This function will convert the specified image (Windows BMP 4/8/16/24 bpp or TGA 16/24 bpp) to Jaguar Bitmap format.

Expected inputs:

A0	Pointer to image to convert
A1	Workspace RAM to use for conversion

Outputs:

None

RAPTOR_UPDATE_ALL (EX-02c)

This function will update all the RAPTOR and Particle objects with any changes made to their database fields.

Expected inputs:

None

Outputs:

None

RAPTOR_wait_frame

This function will halt the running application and wait until the vertical sync.

Expected inputs:

None

Outputs:

None

RAPTOR_wait_frame_UPDATE_ALL

This function will halt the running application and wait until the vertical sync, and then update all RAPTOR and Particle objects.

Expected inputs:

None

Outputs:

None

RAPTOR_UPDATE_SPRITES

This function will force an update of all RAPTOR objects.

Expected inputs:

None

Outputs:

None

RAPTOR_UPDATE_PARTICLES

This function will force an update of all Particle objects.

Expected inputs:

None

Outputs:

None

RAPTOR_U235init (EX-03b)

This function will install and start the U235 Sound Engine, which is a required module for both Audio generation and Jagpad reading.

Expected inputs:

None

Outputs:

None

RAPTOR_call_GPU_code (EX-04b)

This function will set the GPU PC to the specified address and start it running. Execution on the 68000 will not continue until the GPU halts.

Expected inputs:

A0	Address to set the GPU PC

Outputs:

None

RAPTOR_call_GPU_code_STOP

This function will set the GPU PC to the specified address and start it running, and then HALT the 68000 until the next VBI occurs.

Expected inputs:

A0	Address to set the GPU PC

Outputs:

None

RAPTOR_call_GPU_code_nowait

This function will set the GPU PC to the specified address and start it running, and then immediately continue execution on the 68000.

Expected inputs:

A0	Address to set the GPU PC

Outputs:

None

RAPTOR_GPU_COLLISION (EX-04b)

This function will call the RAPTOR collision module. It will perform hitpoint/damage on all colliding RAPTOR object(s) and flag a global 'collision occurred' variable, and also tag each colliding object.

Expected inputs:

raptor_sourcel	RAPTOR Object index value for source range (low)
raptor_sourceh	RAPTOR Object index value for source range (high)
raptor_targetl	RAPTOR Object index value for target range (low)
raptor_targeth	RAPTOR Object index value for source range (high)

Outputs:

raptor_result	Global flag indicating at least one collision occurred

This function will compare all objects between *raptor_sourcel* and *raptor_sourceh* against all objects between *raptor_targetl* and *raptor_targeth*.

If any of the objects have their *cant_hit* flag set they will be skipped.

If a collision occurs (defined by the target box values in the object definition list) the *damage* value of the source is subtracted from the *hitpoint value* of the target, and the *sprite_was_hit* flag is set in the target object.

If the target *hitpoints* go negative the object will obay it's *keep* or *remove* definition.

The *sprite_was_hit* flag needs to be reset before another collision test occurs.

RAPTOR_HEX_to_DEC (EX-04c)

This function will convert a hexadecimal value into ASCII decimal.

Expected inputs:

D1	Hexadecimal value to convert
D4	Number of digits in the output string -1
A0	Address to store the converted string

Outputs:

(A0)	String of characters (D4 long) with converted number

RAPTOR_particle_injection_GPU (EX-05a)

This function will update the particle database with the specified particle effect pattern.

Expected inputs:

raptor_part_inject_addr	Pointer to the particle effect structure to inject

Outputs:

None

Particle Effect Structure

The structure(s) for the particle effects are located in the *_RAPPIXL.S* file, and are defined as below:

X	X position to start the effect
Y	Y position to start the effect
Pixel count	Number of pixels in the effect

This is immediately followed by a list of pixel definitions, one for each specified by *Pixel count* above.

Angle	Angle particle will move at (0-511) Note: zero is EAST, 128 is SOUTH, 256 is WEST and 384=NORTH
Speed	The speed the particle will travel along the specified angle
Angular velocity	The rate of turn applied each update. Positive numbers will rotate clockwise, negative will rotate anti-clockwise
Initial colour	The initial colour (0-15) for the pixel. The palette used is specified in the *PARTIPAL.BMP* file, and is the same used for the fonts
Colour decay rate	Frame delay between each colour step downwards
Pixel life	How long the pixel will remain active (in frames)

RAPTOR_U235setmodule (EX-06a)

This function will call the U235 Sound Engine to set a module to play.

Expected inputs:

A0	Pointer to the module to play

Outputs:

None

RAPTOR_U235gomodule_stereo (EX-06a)

This function will start the specified module playing in stereo mode. Playback is at 16 khz.

Expected inputs:

None

Outputs:

None

RAPTOR_U235gomodule_mono

This function will start the specified module playing in mono mode. Playback is at 16 khz.

Expected inputs:

None

Outputs:

None

RAPTOR_U235playsample (EX-06b)

This function will call the U235 Sound Engine to play the specified sound effect.

Expected inputs:

D0	Sound effect to play
D1	Channel to play sound effect on (0-7). Note: Channels 0-3 are used for module playback.

Outputs:

None

The sound effects are defined in the *_RAPU235.S* file. Their format is as defined in the *U235 Sound Engine manual.*

RAPTOR_U235stopDSP

This function will halt the DSP and silence the channels. Doing so will shut down all U235 Sound Engine functions, including Jagpad reading and the random number generator.

Expected inputs:

None

Outputs:

None

RAPTOR_U235stopmodule

This function will halt any currently playing module and silence the audio channels used for music.

Expected inputs:

None

Outputs:

None

RAPTOR_chk_highscores (EX-09a)

This function compares a specified high score against the internal top ten list.

Expected inputs:

Do	Score to test

Outputs:

Do	Result. Negative = score not in top 10 Positive = score is a new top 10 entry

If the result is positive, the *RAPTOR_resort_score_table* function must be called.

RAPTOR supports a top ten list, storing both a 32-bit score and an 8 character name for each entry.

RAPTOR_resort_score_table (EX-09a)

This function will update the top 10 scoreboard with the last score, using the specified name.

Expected inputs:

A0	Pointer to 8 character name to enter into the table.

Outputs:

None

RAPTOR_mt_save (EX-09b)

This function will save the top 10 table to the MemoryTrack using the application name and filename specified in *_RAPINIT.S*. If no MemoryTrack is present, the function will exit cleanly.

The *RAPTOR_HWinit* function will auto-load the high score table if the MemoryTrack is present and the specified application name and file name exist.

Expected inputs:

None

Outputs:

None

Note: RAPTOR will use the following variables for saving, which are auto-loaded with a pointer to the high score data and its length during *RAPTOR_HWinit*.

RAPTOR_MT_start_address	Memory address to save from
RAPTOR_MT_save_length	Number of bytes to save

RAPTOR_mt_load

This function will load data from the MemoryTrack into main RAM.

The *RAPTOR_HWinit* function will auto-load the high score table if the MemoryTrack is present and the specified application name and file name exist.

Expected inputs:

RAPTOR_MT_start_address	Memory address to save from
RAPTOR_MT_save_length	Number of bytes to load

Outputs:

None

RAPTOR_init_map_objs (EX-10)

This function configures the Tile Map system. It requires the variables for the Tile Map engine to be configured before *RAPTOR_HWinit*.

It also requires that a special structure layout is present in *_RAPINIT.S* – which will be described in the *Examples* section (for EX-10)

Expected inputs:

raptor_mapindex	Pointer to the map data

Outputs:

None

RAPTOR_map_set_position (EX-10)

This function sets the X and Y co-ordinates (in pixels and sub-pixels) for the top-left of the map to be displayed on the screen.

Expected inputs:

raptor_map_position_x	16.16 map co-ordinate (x)
raptor_map_position_y	16.16 map co-ordinate (y)

Outputs:

None

RAPTOR_d_lz77

This function decompresses LZ77 compressed data.

Expected inputs:

A0	Pointer to compressed data
A1	Address to decompress data into

Outputs:

None

RAPTOR_ERROR

This function will effectively halt the application and flash the background colour. It will never return. It is useful for debugging.

Expected inputs:

None

Outputs:

None

RAPTOR_Version (EX-01b)

This function will return a pointer to a string containing the RAPTOR version number.

Expected inputs:

None

Outputs:

A0	Pointer to string containing version number

Non-RAPTOR Objects

This section will cover the hooks and function calls for creating and manipulating objects that are not managed by the RAPTOR Engine.

These routines are based on the ***Sinister Developments Object Library***.

RAPTOR_PRE_Object_List & RAPTOR_POST_Object_List

These two routines exist in the application code written by the user. During *RAPTOR_HWinit* these routines will be called before and after the creation of the RAPTOR objects defined in *_RAPINIT.S*.

If Non-RAPTOR controlled objects are not required, these two functions should contain a single RTS.

They are used to insert Non-RAPTOR objects into the Object List that can be managed directly, without RAPTOR updating them.

RAPTOR_CreateObject

This function will create a new object in the Object list.

Expected inputs:

D0	Object type: 0 – Bitmap Object 1 – Scaled Bitmap Object 2 – GPU Object 3 – Branch Object 4 – Stop Object
D1	Height (in pixels) of Object
D2	Width (in bytes) of a single line of the image
D3	Colour depth 0 - 1bpp 1 - 2bpp 2 - 4bpp 3 - 8bpp 4 - 16bpp 5 - 24bpp
D4	Transparency 0 – Opaque 1 – Transparent
D5	Image Width (in bytes) of the rendered image
A0	Address of the object
A1	Pointer to the bitmap data

Outputs:

None

RAPTOR_rmw

This function will convert the previous object made with *RAPTOR_CreateObject* into a CRY (Read/Modify/Write) Object.

Expected inputs:

None

Outputs:

None

RAPTOR_setpalt

This function will set the CLUT for the previous object made with *RAPTOR_CreateObject*

Expected inputs:

Do	CLUT (0-15) to set

Outputs:

None

RAPTOR_reflect

This function will set the *MIRROR* bit for the previous object made with *RAPTOR_CreateObject* – effectively flipping the bitmap horizontally.

Expected inputs:

None

Outputs:

None

RAPTOR_setup_object_xyg

This function will change the X, Y positions and Bitmap Address for a Non-RAPTOR Object.

Expected inputs:

A0	Pointer to a Non-RAPTOR Object
D0	New X-Position (Not Sub-Pixel)
D1	New Y-Position (Not Sub-Pixel)
D2	New Bitmap Address

Outputs:

None

RAPTOR_setup_object_xg

This function will change the X position and Bitmap Address for a Non-RAPTOR Object.

Expected inputs:

A0	Pointer to a Non-RAPTOR Object
D0	New X-Position (Not Sub-Pixel)
D2	New Bitmap Address

Outputs:

None

RAPTOR_setup_object_g

This function will change the Bitmap Address for a Non-RAPTOR Object.

Expected inputs:

A0	Pointer to a Non-RAPTOR Object
D2	New Bitmap Address

Outputs:

None

RAPTOR_setup_object_xy

This function will change the X and Y position for a Non-RAPTOR Object.

Expected inputs:

A0	Pointer to a Non-RAPTOR Object
D0	New X-Position (Not Sub-Pixel)
D1	New Y-Position (Not Sub-Pixel)

Outputs:

None

RAPTOR_setup_object_xyz

This function will change the X, Y and Scale Ratio for a Non-RAPTOR Object.

Expected inputs:

A0	Pointer to a Non-RAPTOR Object
D0	New X-Position (Not Sub-Pixel)
D1	New Y-Position (Not Sub-Pixel)
D3	Horizontal Scale value
D4	Vertical Scale value

Outputs:

None

RAPTOR_user_vbi (System Hook)

This routines exist in the application code written by the user. It is called by the RAPTOR core every VBI, and allows the user to synchronise code under interrupt.

No registers are saved or restored around this function. It is the user's responsibility to manage this.

Exiting the user routine is via an RTS command.

If this hook is not required it **must point to an RTS**.

RAPTOR Initialization File (_RAPINIT.S)

This file contains the human readable scripts (RAPTOR Lists) used to generate the visible screens generated by RAPTOR Engine. It also contains the MemoryTrack Application and Filenames.

The file is defined as below:

MemoryTrack configuration Info	Application and Filename used by MemoryTrack routines
>**RAPTOR**<	Denotes the start of the RAPTOR data
LIST	Denotes the start of a RAPTOR list
List Data	Per-object data for the items in the list
STOP	Denotes the end of a RAPTOR list
<**RAPTOR**>	Denotes the end of the RAPTOR data

RAPTOR List Data

This file contains the human readable definitions for the Objects used to generate a 'scene'

The LIST is defined as below:

REPEAT COUNTER	Create this many objects of this type (or 1 for a single object)
sprite_active	Active / Inactive flag
sprite_x	Object X co-ordinate in 16.16 format
sprite_y	Object Y co-ordinate in 16.16 format
sprite_xadd	Value to auto-add each call to sprite_x
sprite_yadd	Value to auto-add each call to sprite_y
sprite_width	Width of Object (in pixels)
sprite_height	Height of Object (in pixels)
sprite_flip	Flag for horizontally flipping in the image data
sprite_coffx	X offset from center for collision box center
sprite_coffy	Y offset from center for collision box center
sprite_hbox	Width of collision box
sprite_vbox	Height of collision box

sprite_gfxbase	Pointer to bitmap data
BIT_DEPTH	Bitmap depth (1/2/4/8/16/24)
CRY / RGB	Bitmap graphics type
TRANSPARENCY	Object transparency flag
sprite_framesz	Size (in bytes) of a single frame of sprite data
sprite_bytewid	Size (in bytes) of a single horizontal line of sprite data
sprite_animspd	Frame delay between animation changes
sprite_maxframe	Number of frames in animation chain
sprite_animloop	Play once / Repeat flag for animation chain
sprite_wrap	Flag for screen edge wrap or remove
sprite_timer	Frames sprite is active for (or -1 for infinite)
sprite_track	Flag for 16.16 updates, or pointer to an x/y table
sprite_tracktop	Loop point in x/y table (if used)
sprite_scaled	Flag for if Object is scaled or unscaled
sprite_scale_x	Horizontal scale factor
sprite_scale_y	Vertical scale factor

sprite_was_hit	Collision flag
sprite_CLUT	CLUT Index value
sprite_colchk	Flag defining if Object can collide with another
sprite_remhit	Flag to define if sprite is removed on collision or not
sprite_bboxlink	Flag for single bounding box, or pointer to box datalist*
sprite_hitpoint	Hitpoints for this Object
sprite_damage	Hitpoints deducted from colliding object
sprite_gwidth	Object bitmap width of data (in bytes)

*Multiple Bounding Box List

Multiple bounding box collision lists are defined as below

Number of Boxes	Number of bounding boxes for this Object
X-Offs	x-offset from the middle of the object for current box
Y-offs	y-offset from the middle of the object for current box
Width	Bounding box width /2
Height	Bounding box height /2

RAPTOR Examples

This section of the manual will describe the RAPTOR example code provided with this library. It will break down the example code, detailing what has been added between each example and describing the code's function.

It will start with a simple 'Hello World' example, moving through list manipulation to add graphics, sprites, animation, movement, collision management, particle effects, audio functions, multiple list manipulation, memory track and high score management and finally tile maps.

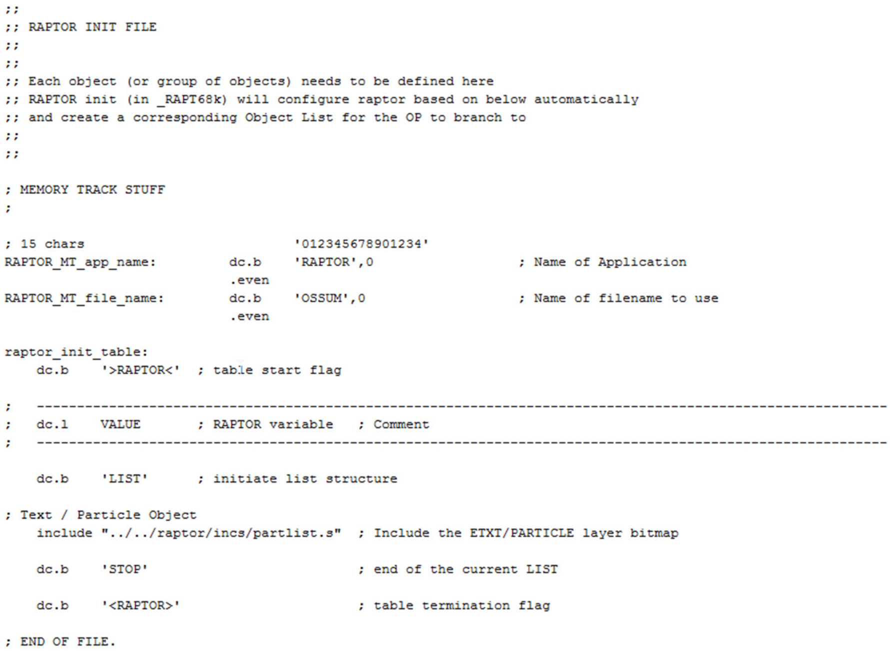
While basic explanation of 68000 instructions may be given to help explain a specific feature this manual is not intended as a 68000 primer. Some basic knowledge of the processor and its instruction set is assumed.

EX-01a – Hello World

The first example will demonstrate how to get a simple “Hello World” message on the screen.

Firstly we need to set up a screen buffer for us to write into. This is done by adding a bitmap object into the RAPTOR list, located in *_RAPINIT.S*.

Below is the *_RAPINIT.S* for EX-01a:



```
;;
;; RAPTOR INIT FILE
;;
;;
;; Each object (or group of objects) needs to be defined here
;; RAPTOR init (in _RAPT68k) will configure raptor based on below automatically
;; and create a corresponding Object List for the OP to branch to
;;
;;

; MEMORY TRACK STUFF
;

; 15 chars                          '012345678901234'
RAPTOR_MT_app_name:         dc.b    'RAPTOR',0                  ; Name of Application
                            .even
RAPTOR_MT_file_name:        dc.b    'OSSUM',0                   ; Name of filename to use
                            .even

raptor_init_table:
    dc.b    '>RAPTOR<'  ; table start flag

;   ----------------------------------------------------------------------------------------------------------
;   dc.l    VALUE       ; RAPTOR variable   ; Comment
;   ----------------------------------------------------------------------------------------------------------

    dc.b    'LIST'      ; initiate list structure

; Text / Particle Object
    include "../../raptor/incs/partlist.s"  ; Include the ETXT/PARTICLE layer bitmap

    dc.b    'STOP'                          ; end of the current LIST

    dc.b    '<RAPTOR>'                      ; table termination flag

; END OF FILE.
```

At the top we set up a dummy MemoryTrack application name and file name, we won’t be using these just yet, but RAPTOR expects them to be present.

Below this is the raptor_init_table – this is where all RAPTOR objects are defined. The RAPTOR table always starts with ‘>RAPTOR<’ and ends with ‘<RAPTOR>’.

Inside this table there can be a single RAPTOR List, or multiple RAPTOR Lists (Max 16). Each individual List starts with a ‘LIST’ command and ends with a ‘STOP’ command.

For example, a single List would look like this:

```
>RAPTOR<
LIST
        List Object Data
STOP
<RAPTOR>
```

Where a table with multiple Lists would look like this:

```
>RAPTOR<
LIST
        List #0 Object Data
STOP
LIST
        List #1 Object Data
STOP
LIST
        List #2 Object Data
STOP
<RAPTOR>
```

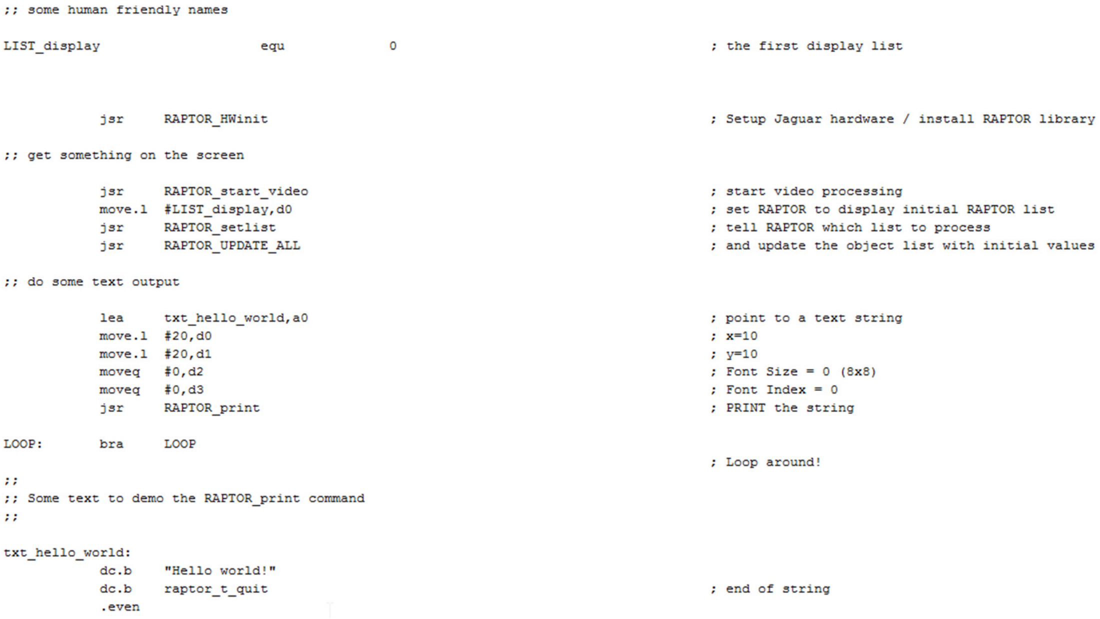
For this example all we require is a bitmap to draw some text into. The RAPTOR Engine uses the Particle Buffer for all text output.

There is a pre-written file already set up for the particle layer object, called *PARTLIST.S*, which is located in the RAPTOR\INCS folder. We will use that here as our only object.

This buffer uses CLUT 15 of the palette (Colours 240-256). The colour data for this buffer comes from the file *PARTIPAL.BMP* located in the RAPTOR\FONTS folder. All fonts (also located in this folder) will use the same palette data as this file, regardless of the palette data stored in their own font file.

This is all we need in the *_RAPINIT.S* file for this example.

Now that we have our display buffer configured lets write some code to get Hello World on the screen.



```
;; some human friendly names

LIST_display                    equu            0                              ; the first display list



          jsr     RAPTOR_HWinit                                                 ; Setup Jaguar hardware / install RAPTOR library

;; get something on the screen

          jsr     RAPTOR_start_video                                            ; start video processing
          move.l  #LIST_display,d0                                              ; set RAPTOR to display initial RAPTOR list
          jsr     RAPTOR_setlist                                                ; tell RAPTOR which list to process
          jsr     RAPTOR_UPDATE_ALL                                             ; and update the object list with initial values

;; do some text output

          lea     txt_hello_world,a0                                            ; point to a text string
          move.l  #20,d0                                                        ; x=10
          move.l  #20,d1                                                        ; y=10
          moveq   #0,d2                                                         ; Font Size = 0 (8x8)
          moveq   #0,d3                                                         ; Font Index = 0
          jsr     RAPTOR_print                                                  ; PRINT the string

LOOP:     bra     LOOP
                                                                                ; Loop around!
;;
;; Some text to demo the RAPTOR_print command
;;

txt_hello_world:
          dc.b    "Hello world!"
          dc.b    raptor_t_quit                                                 ; end of string
          .even
```

As you can see above, this is a very short application.

First we call *RAPTOR_HWinit* to set up the system, followed by *RAPTOR_start_video* which enables video generation.

We then have to tell RAPTOR which List to display, in this example we only have the one. Lists are numbered 0-15. In order to keep this human readable, an alias for this list, called 'LIST_display' has been created, and given a value of 0. This value is passed to the function *RAPTOR_setlist*.

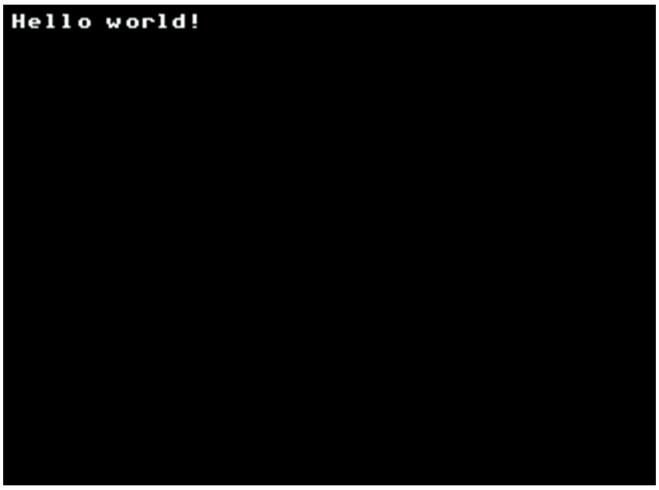
Now, in order to get the Objects in the List on the screen, we need to update RAPTOR. This is done with the function *RAPTOR_UPDATE_ALL*.

We now have a bitmap window on the screen ready for us to draw onto, so the next lines send some text to the screen. A pointer to the text is placed in the A0 register. X and Y co-ordinates are placed in D0-D1. The initial font size and index values are placed in D2 and D3, and finally *RAPTOR_print* is called.

After this, we simply loop around to infinity.

Note: *X values are rounded down to the nearest even value. Text cannot be plotted on odd horizontal values.*

EX-01a – Screen Output



```
Hello world!
```

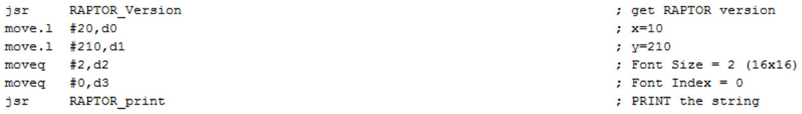
Functions Introduced

- RAPTOR_HWinit
- RAPTOR_start_video
- RAPTOR_setlist
- RAPTOR_print
- RAPTOR_UPDATE_ALL

EX-01b – Text Commands

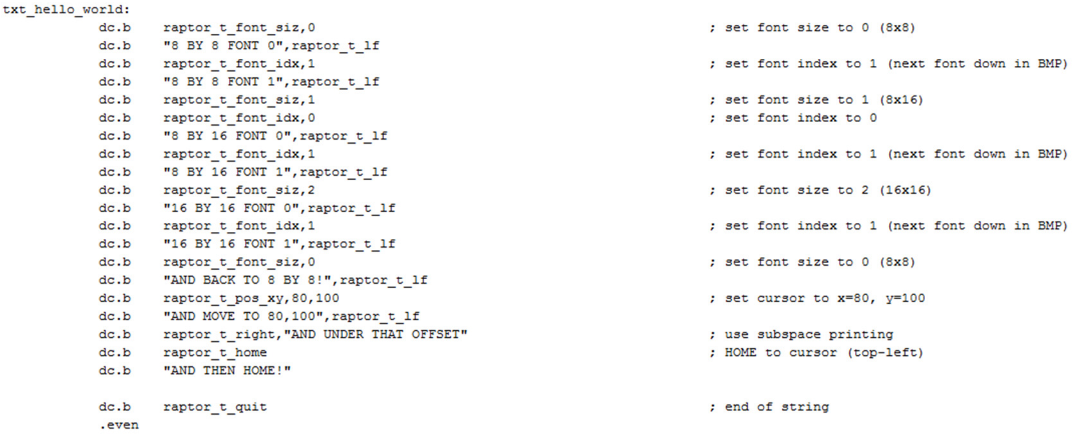
Expanding on the first example, we will now add some text commands to the string sent to the screen to demonstrate the text functions built into RAPTOR.

The _*RAPINIT.S* file remains the same.



```
            jsr     RAPTOR_Version                                                              ; get RAPTOR version
            move.l  #20,d0                                                                      ; x=10
            move.l  #210,d1                                                                     ; y=210
            moveq   #2,d2                                                                       ; Font Size = 2 (16x16)
            moveq   #0,d3                                                                       ; Font Index = 0
            jsr     RAPTOR_print                                                                ; PRINT the string
```

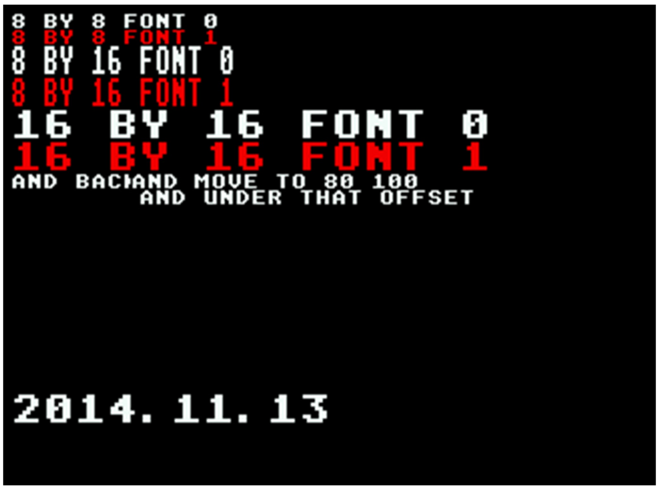
These lines call the RAPTOR function *RAPTOR_Version*, which returns a pointer to a string containing the RAPTOR version number. We then print this string at X (20), Y (210), using the topmost font (Index = 0) in the 16x16 (Size = 2) font bitmap.



```
txt_hello_world:
            dc.b    raptor_t_font_siz,0                                                         ; set font size to 0 (8x8)
            dc.b    "8 BY 8 FONT 0",raptor_t_lf
            dc.b    raptor_t_font_idx,1                                                         ; set font index to 1 (next font down in BMP)
            dc.b    "8 BY 8 FONT 1",raptor_t_lf
            dc.b    raptor_t_font_siz,1                                                         ; set font size to 1 (8x16)
            dc.b    raptor_t_font_idx,0                                                         ; set font index to 0
            dc.b    "8 BY 16 FONT 0",raptor_t_lf
            dc.b    raptor_t_font_idx,1                                                         ; set font index to 1 (next font down in BMP)
            dc.b    "8 BY 16 FONT 1",raptor_t_lf
            dc.b    raptor_t_font_siz,2                                                         ; set font size to 2 (16x16)
            dc.b    "16 BY 16 FONT 0",raptor_t_lf
            dc.b    raptor_t_font_idx,1                                                         ; set font index to 1 (next font down in BMP)
            dc.b    "16 BY 16 FONT 1",raptor_t_lf
            dc.b    raptor_t_font_siz,0                                                         ; set font size to 0 (8x8)
            dc.b    "AND BACK TO 8 BY 8!",raptor_t_lf
            dc.b    raptor_t_pos_xy,80,100                                                      ; set cursor to x=80, y=100
            dc.b    "AND MOVE TO 80,100",raptor_t_lf
            dc.b    raptor_t_right,"AND UNDER THAT OFFSET"                                      ; use subspace printing
            dc.b    raptor_t_home                                                               ; HOME to cursor (top-left)
            dc.b    "AND THEN HOME!"

            dc.b    raptor_t_quit                                                               ; end of string
            .even
```

These additional commands are described in the ***Print Commands*** section, earlier in this manual.

EX-01b – Screen Output



```
8 BY 8 FONT 0
8 BY 8 FONT 1
8 BY 16 FONT 0
8 BY 16 FONT 1
16 BY 16 FONT 0
16 BY 16 FONT 1
AND BACIAND MOVE TO 80 100
            AND UNDER THAT OFFSET




2014.11.13
```

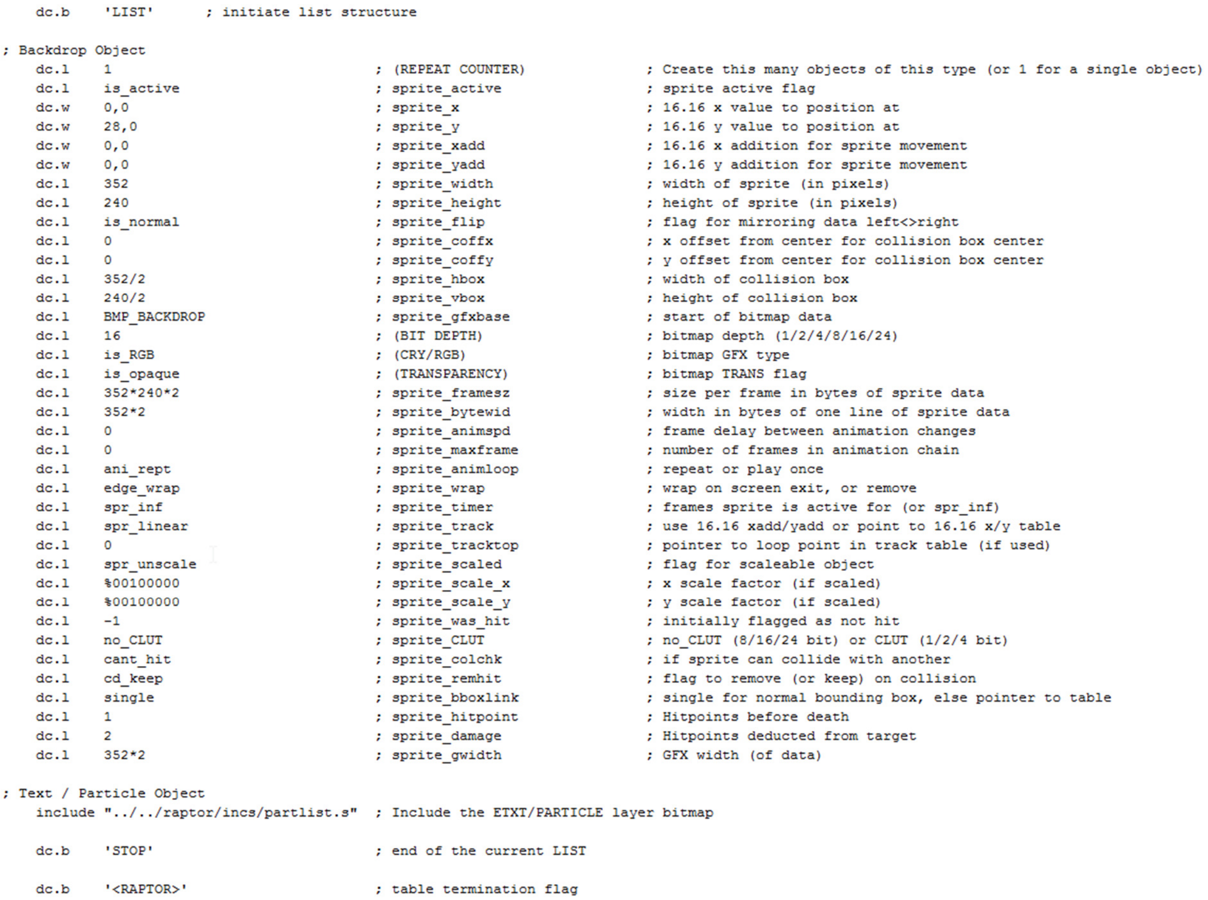
Functions Introduced

- RAPTOR_Version

EX-02a – List Objects (Backdrop)

Now we have some text on the screen, but that is pretty boring, so now we will add a backdrop image. We will also change the text to say something more game-like, such as ‘SCORE: 00000000’.

To do this we need to modify the *_RAPINIT.S* file to add another object, as shown below:



```
    dc.b    'LIST'      ; initiate list structure

; Backdrop Object
    dc.l    1                           ; (REPEAT COUNTER)              ; Create this many objects of this type (or 1 for a single object)
    dc.l    is_active                   ; sprite_active                 ; sprite active flag
    dc.w    0,0                         ; sprite_x                      ; 16.16 x value to position at
    dc.w    28,0                        ; sprite_y                      ; 16.16 y value to position at
    dc.w    0,0                         ; sprite_xadd                   ; 16.16 x addition for sprite movement
    dc.w    0,0                         ; sprite_yadd                   ; 16.16 y addition for sprite movement
    dc.l    352                         ; sprite_width                  ; width of sprite (in pixels)
    dc.l    240                         ; sprite_height                 ; height of sprite (in pixels)
    dc.l    is_normal                   ; sprite_flip                   ; flag for mirroring data left<>right
    dc.l    0                           ; sprite_coffx                  ; x offset from center for collision box center
    dc.l    0                           ; sprite_coffy                  ; y offset from center for collision box center
    dc.l    352/2                       ; sprite_hbox                   ; width of collision box
    dc.l    240/2                       ; sprite_vbox                   ; height of collision box
    dc.l    BMP_BACKDROP                ; sprite_gfxbase                ; start of bitmap data
    dc.l    16                          ; (BIT DEPTH)                   ; bitmap depth (1/2/4/8/16/24)
    dc.l    is_RGB                      ; (CRY/RGB)                     ; bitmap GFX type
    dc.l    is_opaque                   ; (TRANSPARENCY)                ; bitmap TRANS flag
    dc.l    352*240*2                   ; sprite_framesz                ; size per frame in bytes of sprite data
    dc.l    352*2                       ; sprite_bytewid                ; width in bytes of one line of sprite data
    dc.l    0                           ; sprite_animspd                ; frame delay between animation changes
    dc.l    0                           ; sprite_maxframe               ; number of frames in animation chain
    dc.l    ani_rept                    ; sprite_animloop               ; repeat or play once
    dc.l    edge_wrap                   ; sprite_wrap                   ; wrap on screen exit, or remove
    dc.l    spr_inf                     ; sprite_timer                  ; frames sprite is active for (or spr_inf)
    dc.l    spr_linear                  ; sprite_track                  ; use 16.16 xadd/yadd or point to 16.16 x/y table
    dc.l    0                           ; sprite_tracktop               ; pointer to loop point in track table (if used)
    dc.l    spr_unscale                 ; sprite_scaled                 ; flag for scaleable object
    dc.l    %00100000                   ; sprite_scale_x                ; x scale factor (if scaled)
    dc.l    %00100000                   ; sprite_scale_y                ; y scale factor (if scaled)
    dc.l    -1                          ; sprite_was_hit                ; initially flagged as not hit
    dc.l    no_CLUT                     ; sprite_CLUT                   ; no_CLUT (8/16/24 bit) or CLUT (1/2/4 bit)
    dc.l    cant_hit                    ; sprite_colchk                 ; if sprite can collide with another
    dc.l    cd_keep                     ; sprite_remhit                 ; flag to remove (or keep) on collision
    dc.l    single                      ; sprite_bboxlink               ; single for normal bounding box, else pointer to table
    dc.l    1                           ; sprite_hitpoint               ; Hitpoints before death
    dc.l    2                           ; sprite_damage                 ; Hitpoints deducted from target
    dc.l    352*2                       ; sprite_gwidth                 ; GFX width (of data)

; Text / Particle Object
    include "../../raptor/incs/partlist.s"  ; Include the ETXT/PARTICLE layer bitmap

    dc.b    'STOP'                      ; end of the current LIST

    dc.b    '<RAPTOR>'                  ; table termination flag
```

Objects are displayed on screen in a back-to-front order. The higher up the list an object is, the further back it is displayed. We want our text to appear on top of the background picture, so the particle layer has to come after the new background object.

This object in detail is shown in the following table:

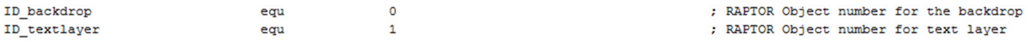
1	REPEAT COUNTER	There is a single instance of this object
is_active	sprite_active	Object is active (Will be displayed)
0,0	sprite_x	X position to display (in 16.16 sub-pixel format)
0,0	sprite_y	Y position to display (in 16.16 sub-pixel format)
0,0	sprite_xadd	Object is stationary in X direction
0,0	sprite_yadd	Object is stationary in Y direction
352	sprite_width	Object is 352 pixels wide
240	sprite_height	Object is 240 pixels high
is_normal	sprite_flip	Object is not horizontally flipped
0	sprite_coffx	X offset from center for collision box
0	sprite_coffy	Y offset from center for collision box
352/2	sprite_hbox	Width of the collision box from the collision center point
240/2	sprite_vbox	Height of the collision box from the collision center point
BMP_BACKDROP	sprite_gfxbase	Pointer to the bitmap data
16	BIT_DEPTH	Bitmap depth (1/2/4/8/16/24)
is_RGB	CRY/RGB	Specifies if object is CRY or RGB

is_opaque	TRANSPARENCY	Transparency
352*240*2	sprite_framesz	Size (in bytes) of a single frame of the image. In this case the image is 352*240 pixels, and 16 bpp (2 bytes per pixel)
352*2	sprite_bytewid	Size (in bytes) of a single line of the image.
0	sprite_animspd	Animation speed. As this is a single image, it is set to zero
0	sprite_maxframe	Number of frames in the animation chain (1st frame = 0)
ani_rept	sprite_animloop	Repeat (0) or play once (1)
edge_wrap	sprite_wrap	Screen edge condition. If sprite moved off the edge of the screen it will wrap to the other side
spr_inf	sprite_timer	The number of frames the sprite will be active for
spr_linear	sprite_track	This flag sets how the object is to be updated. If set to linear it means the object will use the X/Y positions, otherwise it is a pointer to a pre-determined set of x/y co-ordinates.
0	sprite_tracktop	If following a pre-determined path, this is a pointer to the loop position
spr_unscale	sprite_scaled	Object can be either scaled, or unscaled.
%00100000	sprite_scale_x	Object scaling value (horizontal)

%00100000	sprite_scae_y	Object scaling value (vertical)
-1	sprite_was_hit	Collision flag. -1 = not hit
no_CLUT	sprite_CLUT	CLUT to use for colour data. As this is a 16bpp image no CLUT is required
cant_hit	sprite_colchk	Flag to decide if object can collide. As this is the backdrop, there is no need for collision.
cd_keep	sprite_remhit	Condition to perform is a collision occurs. In this case, keep the object alive.
single	sprite_bboxlink	Flag for a single bounding box used for collision, else this is a pointer to a bounding box collision list
1	sprite_hitpoint	Object hit points
2	sprite_damage	Damage inflicted on colliding object
352*2	sprite_gwidth	Width of bitmap data. Keep this the same as sprite_bytewid for standard image. Used for offset effects.

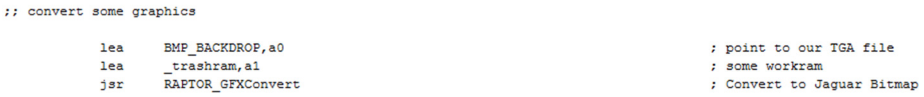
This defines the object in the list, now we need to add some code to the *_RAPAPP.S* file to configure this object.

So that we can identify the objects easily, we will set up some human readable names:



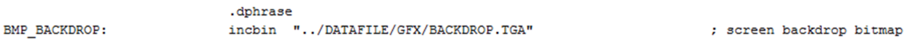
```
ID_backdrop                 equu         0                                   ; RAPTOR Object number for the backdrop
ID_textlayer                equ          1                                   ; RAPTOR Object number for text layer
```

We will be using a TGA file for the backdrop image. The Jaguar hardware does not know how to display images in this format, so we need to convert it to Jaguar Bitmap format before use.



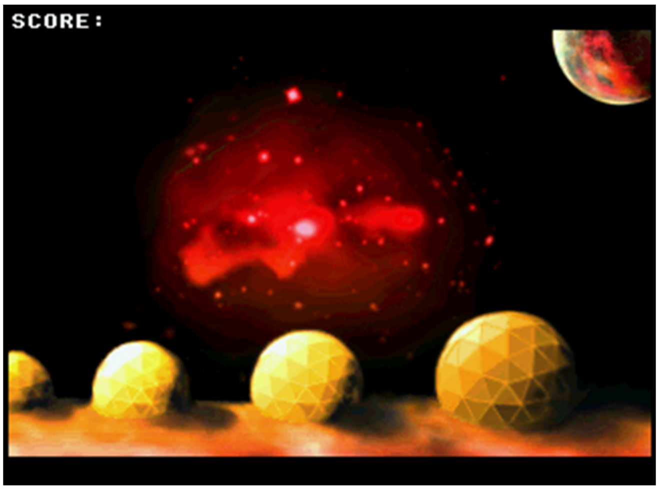
```
;; convert some graphics

            lea     BMP_BACKDROP,a0                                          ; point to our TGA file
            lea     _trashram,a1                                             ; some workram
            jsr     RAPTOR_GFXConvert                                        ; Convert to Jaguar Bitmap
```

And lastly, we need to include the image in the binary.



```
                            .dphrase
BMP_BACKDROP:               incbin  "../DATAFILE/GFX/BACKDROP.TGA"           ; screen backdrop bitmap
```

EX-02a – Screen Output



Functions Introduced

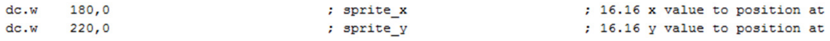
- RAPTOR_GFXConvert

EX-02b – List Objects (Player)

The next thing we will add is a player spaceship object, positioned at the bottom middle of the screen.

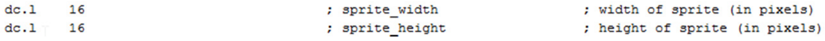
To do this we will need to add another object to the List structure. We will place this object after the background object, but before the text layer.

Most of the structure will be the same as the backdrop, however the following attributes will be changed:



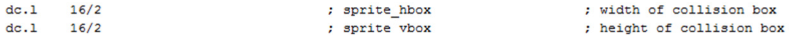
```
    dc.w    180,0                           ; sprite_x                      ; 16.16 x value to position at
    dc.w    220,0                           ; sprite_y                      ; 16.16 y value to position at
```

The X and Y positions will be set to 180 (X), 220 (Y)



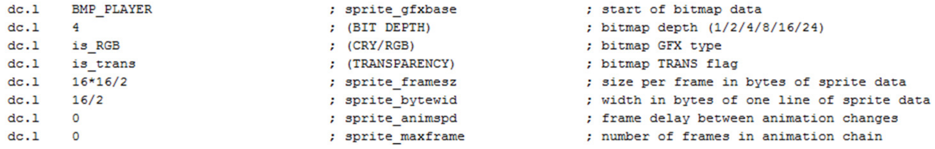
```
    dc.l    16                              ; sprite_width                  ; width of sprite (in pixels)
    dc.l    16                              ; sprite_height                 ; height of sprite (in pixels)
```

The width and height of the object are both set to 16 (16x16 sprite)



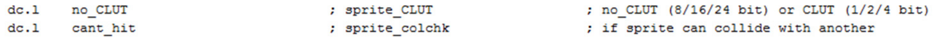
```
    dc.l    16/2                            ; sprite_hbox                   ; width of collision box
    dc.l    16/2                            ; sprite vbox                   ; height of collision box
```

The collision bounding box width and height are set to 8 (8 pixels from the center, in each direction)



```
    dc.l    BMP_PLAYER                      ; sprite_gfxbase                ; start of bitmap data
    dc.l    4                               ; (BIT DEPTH)                   ; bitmap depth (1/2/4/8/16/24)
    dc.l    is_RGB                          ; (CRY/RGB)                     ; bitmap GFX type
    dc.l    is_trans                        ; (TRANSPARENCY)                ; bitmap TRANS flag
    dc.l    16*16/2                         ; sprite_framesz                ; size per frame in bytes of sprite data
    dc.l    16/2                            ; sprite_bytewid                ; width in bytes of one line of sprite data
    dc.l    0                               ; sprite_animspd                ; frame delay between animation changes
    dc.l    0                               ; sprite_maxframe               ; number of frames in animation chain
```

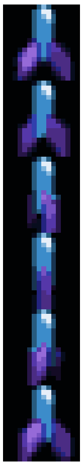
The bitmap pointer is set to point to our sprite data, is set to 4Bpp (16 colours), RGB, Transparent. The frame size is 16*16/2 (1 byte = 2 pixels), and the bytewidth is set to 16/2 (8 bytes per line). We have also set the animation frames to zero (only one frame)



```
    dc.l    no_CLUT                         ; sprite_CLUT                   ; no_CLUT (8/16/24 bit) or CLUT (1/2/4 bit)
    dc.l    cant_hit                        ; sprite_colchk                 ; if sprite can collide with another
```

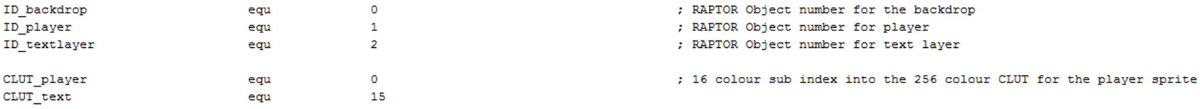
We have created a human readable alias (CLUT_player) to define the CLUT location for these 16 colours, and for now we'll tell RAPTOR that the object cannot be hit.

Our sprite data bitmap will look like this:



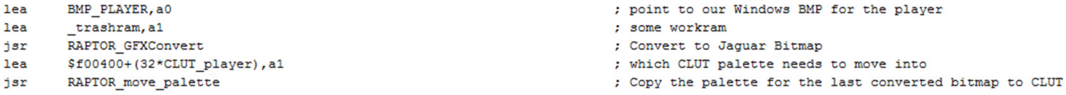
Sprites frames are numbered from zero (0) at the top, incrementing by 1 as they go down. The animation sequence is played *bottom upwards*.

Now we need to edit *_RAPAPP.S* to finish these changes off.



```
ID_backdrop             equu        0                       ; RAPTOR Object number for the backdrop
ID_player               equ         1                       ; RAPTOR Object number for player
ID_textlayer            equ         2                       ; RAPTOR Object number for text layer

CLUT_player             equ         0                       ; 16 colour sub index into the 256 colour CLUT for the player sprite
CLUT_text               equ         15
```

Another human readable alias is created for the player, along with a CLUT_ reference.

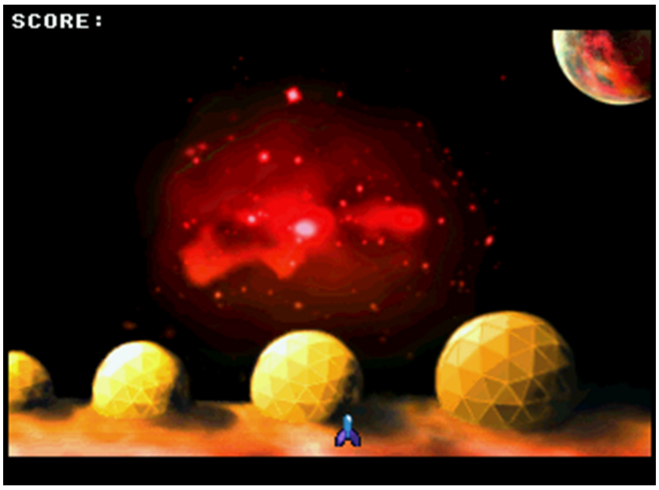


```
    lea     BMP_PLAYER,a0                               ; point to our Windows BMP for the player
    lea     _trashram,a1                                ; some workram
    jsr     RAPTOR_GFXConvert                           ; Convert to Jaguar Bitmap
    lea     $f00400+(32*CLUT_player),a1                 ; which CLUT palette needs to move into
    jsr     RAPTOR_move_palette                         ; Copy the palette for the last converted bitmap to CLUT
```

Again, we have to convert the bitmap (this time, a 4bb Windows BMP) to Jaguar Bitmap format. As this is a paletted image (16 colours) we also need to move the palette data into the correct position in the CLUT. The Jaguar palette data starts at hardware address $f00400, each CLUT is 16 colours, with 2 bytes per colour, as can be seen by the line setting the A1 register. Finally we call *RAPTOR_move_palette* to copy the converted palette to the display CLUT.

You will note that the first image in the file is displayed for the player ship, and that it does not animate.

EX-02a – Screen Output



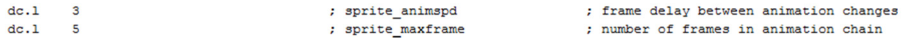
Functions Introduced

- RAPTOR_move_palette

EX-02c – List Objects (Player Animation)

Now we have a player object on the screen, but it isn't animating. The bitmap data file has 6 frames of animation, which we will now enable.

In the _*RAPINIT.S* file we make the following changes:

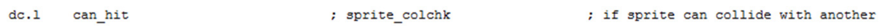


```
    dc.l    3                           ; sprite_animspd              ; frame delay between animation changes
    dc.l    5                           ; sprite_maxframe             ; number of frames in animation chain
```

The *sprite_animspd* field is set to 3, this means that every 3 vertical blanks the next frame will be swapped in.

The *sprite_maxframe* field is set to 5. Frames start at zero, so for this sprite 6 frames this is the correct number.

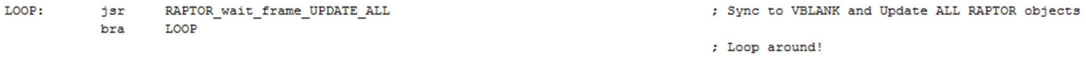
While we are here, we also make the following change:



```
    dc.l    can_hit                     ; sprite_colchk               ; if sprite can collide with another
```

This will allow the collision detection later on to collide with this object.

In the _*RAPAPP.S* file we have to make the following changes:



```
LOOP:       jsr     RAPTOR_wait_frame_UPDATE_ALL                        ; Sync to VBLANK and Update ALL RAPTOR objects
            bra     LOOP
                                                                        ; Loop around!
```

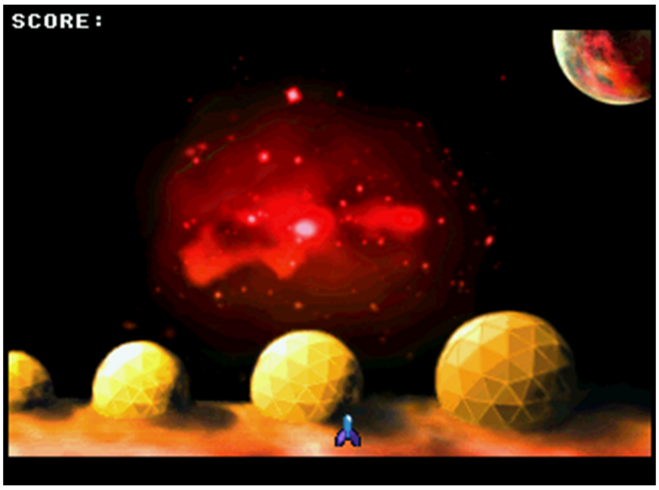
Previously nothing was moving or animating, so the main loop could happily keep repeating to infinity. Now, however, we want some animation to occur.

To accomplish this, we add the call to *RAPTOR_wait_frame_UPDATE_ALL*.

This command will synchronise to the vertical blank, and then update all active RAPTOR Objects and Particles.

Nothing else needs to be changed, we now have an animating sprite at the bottom of the screen.

EX-02c – Screen Output

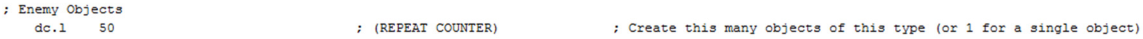


Functions Introduced

- RAPTOR_wait_frame_UPDATE_ALL

EX-02d – List Objects (Enemies)

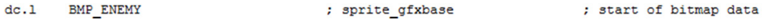
Next we need to introduce some enemies. We could define them, one at a time, as we have done the player. While that would work it would be extremely tedious, so instead we are going to use the object repeat counter. In the *_RAPINIT.S* file we add another object.



```
; Enemy Objects
    dc.l    50                              ; (REPEAT COUNTER)              ; Create this many objects of this type (or 1 for a single object)
```

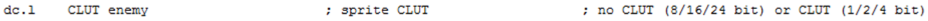
This will create 50 copies of the object defined below, giving us 50 enemy objects.

All the other fields are the same as the player, with the exception of:



```
    dc.l    BMP_ENEMY                       ; sprite_gfxbase                ; start of bitmap data
```

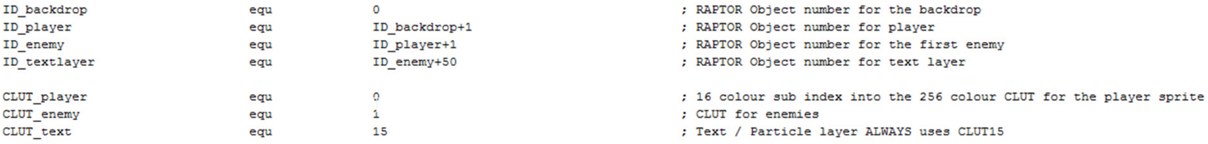
Which sets a different bitmap for the enemies, and:



```
    dc.l    CLUT_enemy                      ; sprite CLUT                   ; no CLUT (8/16/24 bit) or CLUT (1/2/4 bit)
```

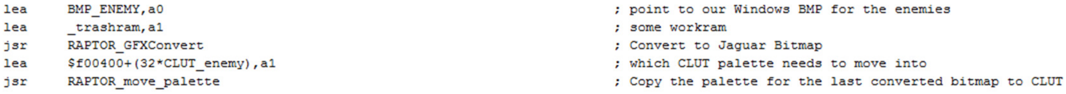
Which sets up a different CLUT for them, keeping their colour data separate from the player.

Now we need to make some changes to the *_RAPAPP.S* file.



```
ID_backdrop                     equu            0                               ; RAPTOR Object number for the backdrop
ID_player                       equ             ID_backdrop+1                   ; RAPTOR Object number for player
ID_enemy                        equ             ID_player+1                     ; RAPTOR Object number for the first enemy
ID_textlayer                    equ             ID_enemy+50                     ; RAPTOR Object number for text layer

CLUT_player                     equ             0                               ; 16 colour sub index into the 256 colour CLUT for the player sprite
CLUT_enemy                      equ             1                               ; CLUT for enemies
CLUT_text                       equ             15                              ; Text / Particle layer ALWAYS uses CLUT15
```

You will notice that the ID_textlayer is set to ID_enemy+50. By setting the current object to the previous value plus the number of repeats of the previous object, we can save ourselves the bother of having to calculate the index value when things change. We have also created another CLUT_ value for the enemies.

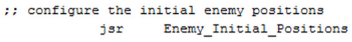


```
        lea     BMP_ENEMY,a0                                    ; point to our Windows BMP for the enemies
        lea     _trashram,a1                                    ; some workram
        jsr     RAPTOR_GFXConvert                               ; Convert to Jaguar Bitmap
        lea     $f00400+(32*CLUT_enemy),a1                      ; which CLUT palette needs to move into
        jsr     RAPTOR_move_palette                             ; Copy the palette for the last converted bitmap to CLUT
```

We also convert the enemy bitmap, exactly as we did for the player sprite.

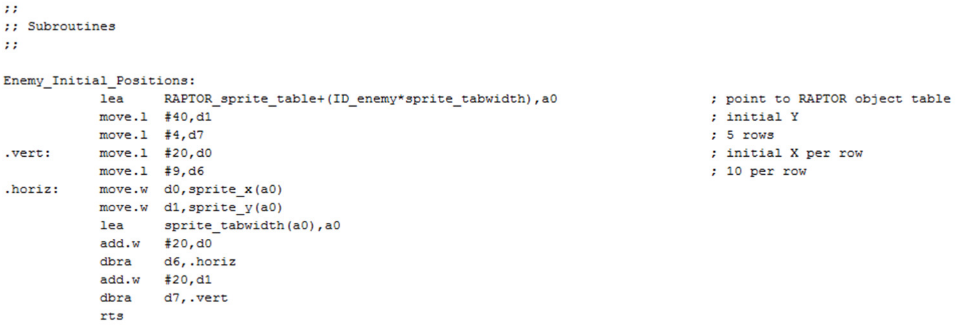
We are also going to create our first subroutine. This will be a routine that will reset the enemy positions back to the starting point.

We call the subroutine with the following line:



```
;; configure the initial enemy positions
            jsr     Enemy_Initial_Positions
```

And define it, below the main loop, as follows:

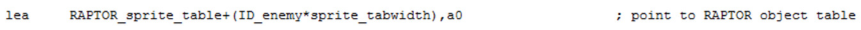


```
;;
;; Subroutines
;;

Enemy_Initial_Positions:
            lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a0                  ; point to RAPTOR object table
            move.l  #40,d1                                                             ; initial Y
            move.l  #4,d7                                                              ; 5 rows
.vert:      move.l  #20,d0                                                             ; initial X per row
            move.l  #9,d6                                                              ; 10 per row
.horiz:     move.w  d0,sprite_x(a0)
            move.w  d1,sprite_y(a0)
            lea     sprite_tabwidth(a0),a0
            add.w   #20,d0
            dbra    d6,.horiz
            add.w   #20,d1
            dbra    d7,.vert
            rts
```

We will now break this down further.

To make changes to the enemy objects we need to change their field values in the RAPTOR internal database. This is the main reason we have been adding the *ID_name* tags at the top of the source code listing.

The database starts at location *RAPTOR_sprite_table*, and each entry in the table is *sprite_tabwidth* wide (ie, it takes sprite_tabwidth bytes for one entry.) To calculate the position of the first enemy we use:



```
            lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a0                  ; point to RAPTOR object table
```

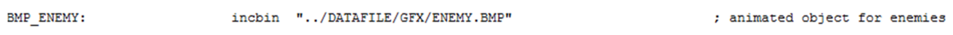
The next two lines set up an initial Y position, and a counter for the number of rows. There will be 5 rows of 10 (50 objects) so the row counter is set to 4 (68000 DBRA loops while positive, so it will loop on zero)

Next, we set the initial X position each row. This will get reset each time around the vertical loop, so each row will start at the same position, and we set the objects per row counter to 9 (10 objects).

The inner loop configures the actual objects. We move the X and Y positions into the fields, increment the object pointer to the next object, add 20 to the X position, and loop around.

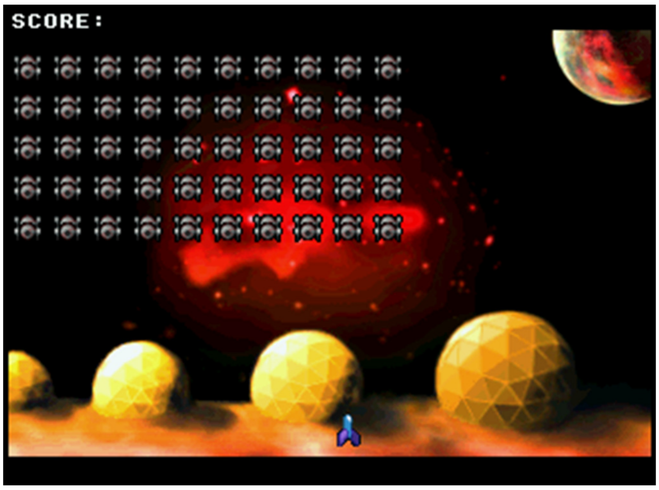
We add 20 to the Y position, and again loop around for each line. Finally we exit the routine with an RTS (Return Subroutine) instruction.

Finally, we include the bitmap for the enemy objects:



```
BMP_ENEMY:                  incbin  "../DATAFILE/GFX/ENEMY.BMP"                         ; animated object for enemies
```

EX-02d – Screen Output



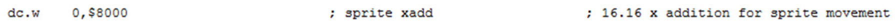
EX-03a – Movement (Automovement)

We now have a group of enemies animating away on the screen. But how can we make them move?

There are a few ways to make objects move using RAPTOR. We can directly update their X/Y positions in the database, we can use a pre-defined file of X/Y co-ordinates, or we can use auto-movement.

To accomplish this, we are only going to make one change to the previous example.

In *_RAPINIT.S* we will alter the following line in the enemy definition:



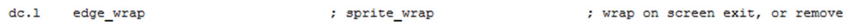
```
dc.w    0,$8000                         ; sprite xadd                   ; 16.16 x addition for sprite movement
```

This field (and the one for Y below it) tell RAPTOR what value to add to the X and Y co-ordinates every update. Like the co-ordinates themselves, they are 16.16 fractional values.

Each co-ordinate in RAPTOR is stored as a fractional value, composed of an integer part (the high 16 bits) and a fractional part (the low 16 bits) which allows RAPTOR to position Objects with sub-pixel accuracy.

You will notice the xadd value above is $0000,$8000. The integer part is set to zero, but the sub-pixel fractional part is set to $8000 – this is exactly 50% of a whole unit ($8000 is one-half of $10000). What this effectively means is that the enemies will move right half a pixel every update, or once every other frame.

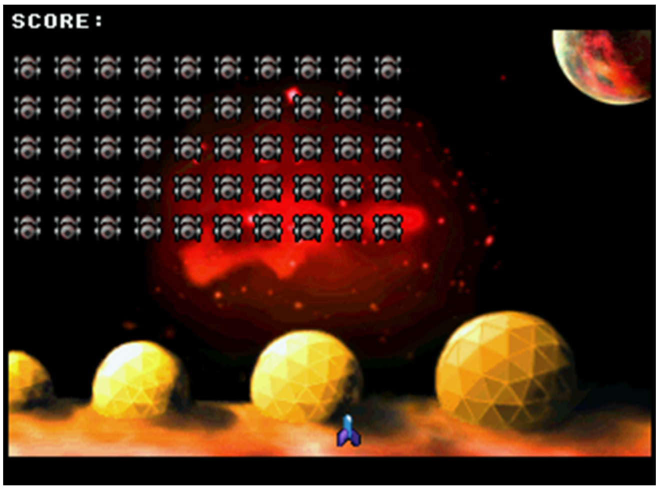
When they reach the right hand edge of the screen they will auto-wrap back to the left side, because the field…



```
dc.l    edge_wrap                       ; sprite_wrap                   ; wrap on screen exit, or remove
```

…is set.

EX-03a – Screen Output



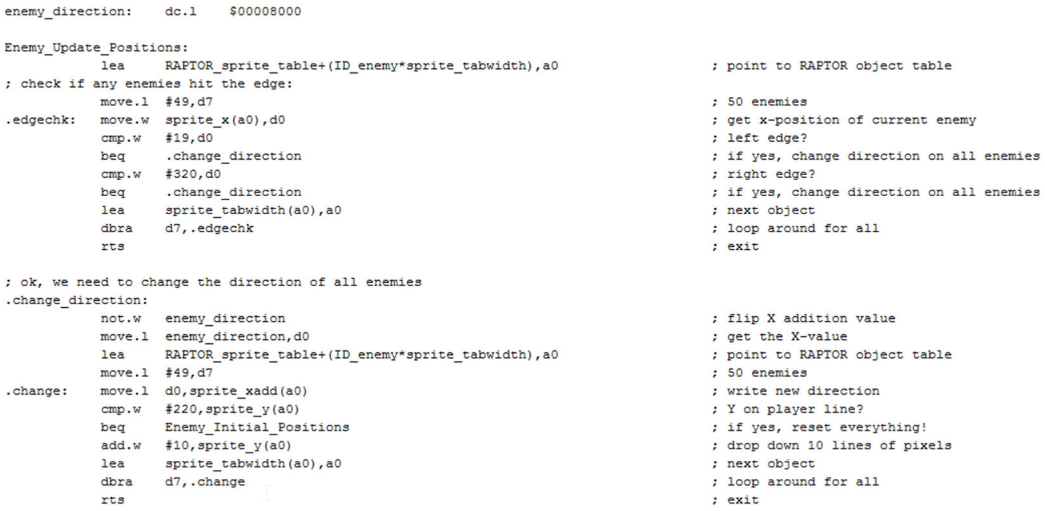
EX-03b – Movement (Code Controlled)

OK, so now we have movement, but it isn’t really space invaders. The enemies need to march down the screen, alternating direction when they hit the edge. To do that, we need to update them programmatically, and to do that we are going to use another subroutine.

In the main loop we add the following command:

```
            jsr     Enemy_Update_Positions                                                  ; Make them march!
```

To call the subroutine below:

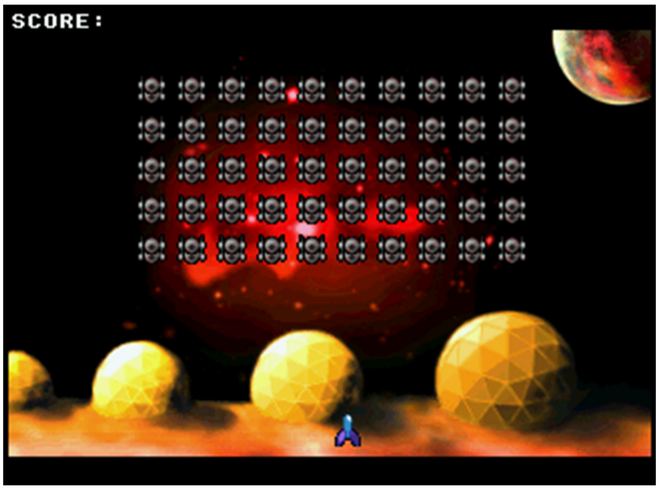


```
enemy_direction:    dc.l    $00008000

Enemy_Update_Positions:
            lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a0                       ; point to RAPTOR object table
; check if any enemies hit the edge:
            move.l  #49,d7                                                                  ; 50 enemies
.edgechk:   move.w  sprite_x(a0),d0                                                         ; get x-position of current enemy
            cmp.w   #19,d0                                                                  ; left edge?
            beq     .change_direction                                                       ; if yes, change direction on all enemies
            cmp.w   #320,d0                                                                 ; right edge?
            beq     .change_direction                                                       ; if yes, change direction on all enemies
            lea     sprite_tabwidth(a0),a0                                                  ; next object
            dbra    d7,.edgechk                                                             ; loop around for all
            rts                                                                             ; exit

; ok, we need to change the direction of all enemies
.change_direction:
            not.w   enemy_direction                                                         ; flip X addition value
            move.l  enemy_direction,d0                                                      ; get the X-value
            lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a0                       ; point to RAPTOR object table
            move.l  #49,d7                                                                  ; 50 enemies
.change:    move.l  d0,sprite_xadd(a0)                                                      ; write new direction
            cmp.w   #220,sprite_y(a0)                                                       ; Y on player line?
            beq     Enemy_Initial_Positions                                                 ; if yes, reset everything!
            add.w   #10,sprite_y(a0)                                                        ; drop down 10 lines of pixels
            lea     sprite_tabwidth(a0),a0                                                  ; next object
            dbra    d7,.change                                                              ; loop around for all
            rts                                                                             ; exit
```

The comments in the code for this section should be sufficient explanation of the code.

The Object positions are checked to see if they hit the edge (X=19, or X=320), and if so *change_direction* is called. This resets the X-direction of the auto-movement field, increments the objects Y-position, and checks if it has ‘landed’ at the bottom. If it has, it resets the formation.

EX-03b – Screen Output



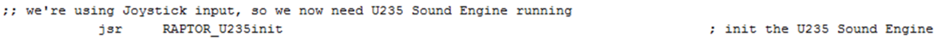
EX-03c – Movement (Jagpad Controlled)

Now we need to get our player moving. To do this we are going to need to use some functions external to RAPTOR, as Joypad input is provided by the U235 Sound Engine library.

RAPTOR contains ‘wrapper calls’ to make using the Sound Engine slightly easier to call from within your own application however there is no reason why you couldn’t make the design decision to not use these and directly access the U235 Sound Engine functions yourself.

For further details regarding the direct use of the Sound Engine please see the manual.pdf provided with that package, which is also included inside the RAPTOR zip file archive.

To initialize the sound engine we use the following line:

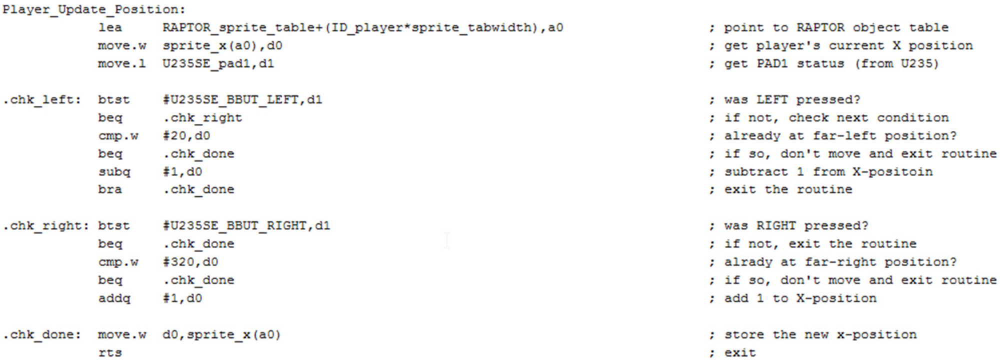


```
;; we're using Joystick input, so we now need U235 Sound Engine running
		jsr	RAPTOR_U235init								; init the U235 Sound Engine
```

In the main loop of the code we need another subroutine to manage the player movement:

```
		jsr	Player_Update_Position							; Move the player
```

Which we define as below:

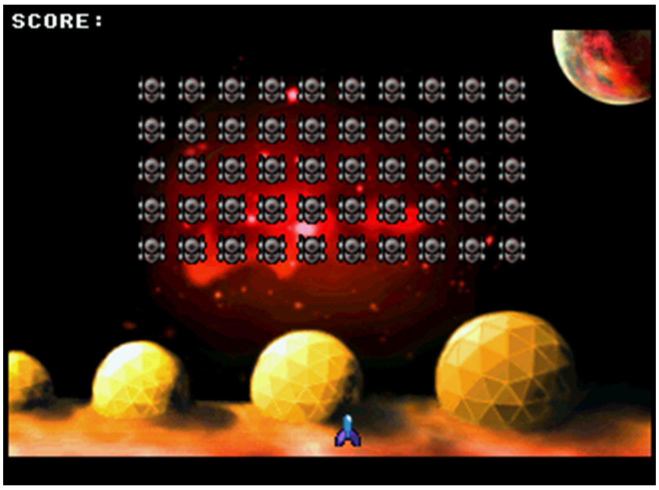


```
Player_Update_Position:
		lea	RAPTOR_sprite_table+(ID_player*sprite_tabwidth),a0			; point to RAPTOR object table
		move.w	sprite_x(a0),d0								; get player's current X position
		move.l	U235SE_pad1,d1								; get PAD1 status (from U235)

.chk_left:	btst	#U235SE_BBUT_LEFT,d1							; was LEFT pressed?
		beq	.chk_right								; if not, check next condition
		cmp.w	#20,d0									; already at far-left position?
		beq	.chk_done								; if so, don't move and exit routine
		subq	#1,d0									; subtract 1 from X-positoin
		bra	.chk_done								; exit the routine

.chk_right:	btst	#U235SE_BBUT_RIGHT,d1							; was RIGHT pressed?
		beq	.chk_done								; if not, exit the routine
		cmp.w	#320,d0									; alrady at far-right position?
		beq	.chk_done								; if so, don't move and exit routine
		addq	#1,d0									; add 1 to X-position

.chk_done:	move.w	d0,sprite_x(a0)								; store the new x-position
		rts										; exit
```

Again, the comments in the code for this section should be sufficient explanation.

The joypad is scanned for left and right, and if the player is not already at a limit its position is updated.

EX-03c – Screen Output



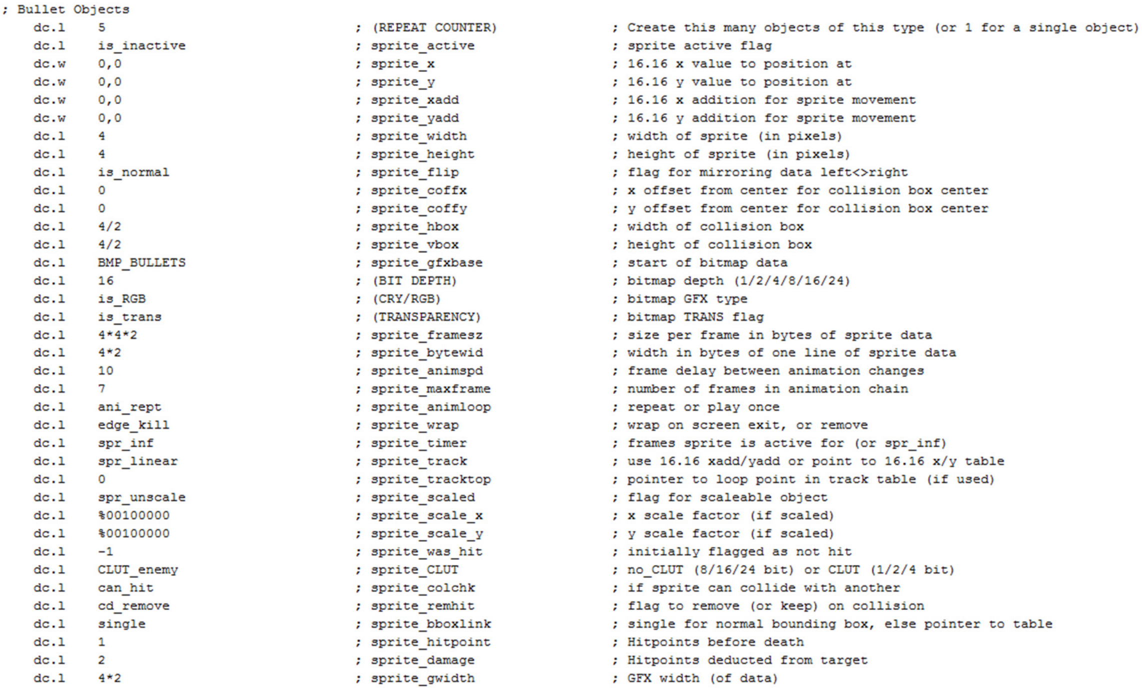
Functions Introduced

- RAPTOR_U235init

EX-04a – Object Spawning

We now have the player able to move, and the enemies marching around the screen. Next we will add a ‘bullet’ that the player can fire, shooting upwards. To do this we need to spawn new objects.

To start with, we need to define some bullet objects in the *_RAPINIT.S* file:

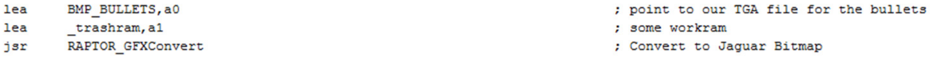


```
; Bullet Objects
    dc.l    5                                ; (REPEAT COUNTER)                ; Create this many objects of this type (or 1 for a single object)
    dc.l    is_inactive                      ; sprite_active                   ; sprite active flag
    dc.w    0,0                              ; sprite_x                        ; 16.16 x value to position at
    dc.w    0,0                              ; sprite_y                        ; 16.16 y value to position at
    dc.w    0,0                              ; sprite_xadd                     ; 16.16 x addition for sprite movement
    dc.w    0,0                              ; sprite_yadd                     ; 16.16 y addition for sprite movement
    dc.l    4                                ; sprite_width                    ; width of sprite (in pixels)
    dc.l    4                                ; sprite_height                   ; height of sprite (in pixels)
    dc.l    is_normal                        ; sprite_flip                     ; flag for mirroring data left<>right
    dc.l    0                                ; sprite_coffx                    ; x offset from center for collision box center
    dc.l    0                                ; sprite_coffy                    ; y offset from center for collision box center
    dc.l    4/2                              ; sprite_hbox                     ; width of collision box
    dc.l    4/2                              ; sprite_vbox                     ; height of collision box
    dc.l    BMP_BULLETS                      ; sprite_gfxbase                  ; start of bitmap data
    dc.l    16                               ; (BIT DEPTH)                     ; bitmap depth (1/2/4/8/16/24)
    dc.l    is_RGB                           ; (CRY/RGB)                       ; bitmap GFX type
    dc.l    is_trans                         ; (TRANSPARENCY)                  ; bitmap TRANS flag
    dc.l    4*4*2                            ; sprite_framesz                  ; size per frame in bytes of sprite data
    dc.l    4*2                              ; sprite_bytewid                  ; width in bytes of one line of sprite data
    dc.l    10                               ; sprite_animspd                  ; frame delay between animation changes
    dc.l    7                                ; sprite_maxframe                 ; number of frames in animation chain
    dc.l    ani_rept                         ; sprite_animloop                 ; repeat or play once
    dc.l    edge_kill                        ; sprite_wrap                     ; wrap on screen exit, or remove
    dc.l    spr_inf                          ; sprite_timer                    ; frames sprite is active for (or spr_inf)
    dc.l    spr_linear                       ; sprite_track                    ; use 16.16 xadd/yadd or point to 16.16 x/y table
    dc.l    0                                ; sprite_tracktop                 ; pointer to loop point in track table (if used)
    dc.l    spr_unscale                      ; sprite_scaled                   ; flag for scaleable object
    dc.l    %00100000                        ; sprite_scale_x                  ; x scale factor (if scaled)
    dc.l    %00100000                        ; sprite_scale_y                  ; y scale factor (if scaled)
    dc.l    -1                               ; sprite_was_hit                  ; initially flagged as not hit
    dc.l    CLUT_enemy                       ; sprite_CLUT                     ; no_CLUT (8/16/24 bit) or CLUT (1/2/4 bit)
    dc.l    can_hit                          ; sprite_colchk                   ; if sprite can collide with another
    dc.l    cd_remove                        ; sprite_remhit                   ; flag to remove (or keep) on collision
    dc.l    single                           ; sprite_bboxlink                 ; single for normal bounding box, else pointer to table
    dc.l    1                                ; sprite_hitpoint                 ; Hitpoints before death
    dc.l    2                                ; sprite_damage                   ; Hitpoints deducted from target
    dc.l    4*2                              ; sprite_gwidth                   ; GFX width (of data)
```

We’ll again use the REPEAT COUNTER to define 5 bullets at the same time. The second line sets the objects to inactive status. This means they will not be drawn on the screen. Their height and width is set to 4, the collision bounding box is set to 4/2, the bitmap pointer is set to point to our TGA file containing the bullets image and the animation, collision and size entries are all set to their relevant value.

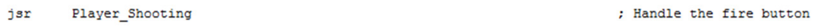
Other changes to previous objects are that *sprite_remhit* is set to cd_remove, meaning the object will deactivate if it collides with another object, and that *sprite_wrap* is set to edge_kill, meaning the object will be deactivated if it goes off the edge of the screen instead of jumping to the other side.

In *_RAPAPP.S* we once again convert the bitmap:



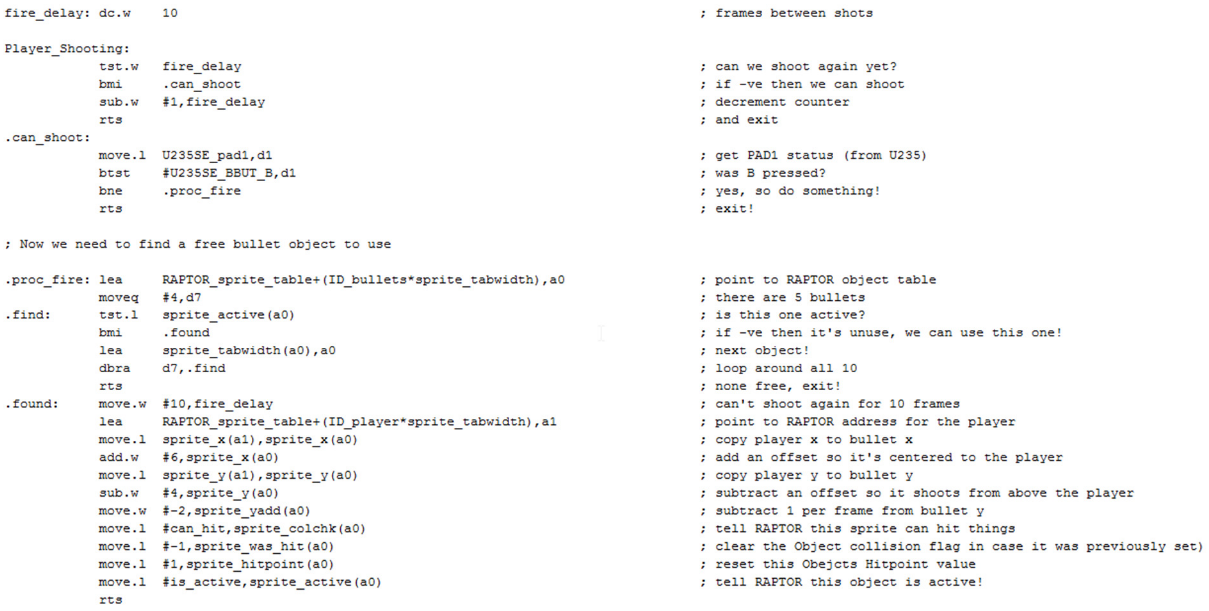
```
lea     BMP_BULLETS,a0                                            ; point to our TGA file for the bullets
lea     _trashram,a1                                              ; some workram
jsr     RAPTOR_GFXConvert                                         ; Convert to Jaguar Bitmap
```

We will also add another subroutine to the main loop to deal with shooting:



```
        jsr     Player_Shooting                                     ; Handle the fire button
```

Which is defined below as:



```
fire_delay: dc.w    10                                              ; frames between shots

Player_Shooting:
            tst.w   fire_delay                                      ; can we shoot again yet?
            bmi     .can_shoot                                      ; if -ve then we can shoot
            sub.w   #1,fire_delay                                   ; decrement counter
            rts                                                     ; and exit
.can_shoot:
            move.l  U235SE_pad1,d1                                  ; get PAD1 status (from U235)
            btst    #U235SE_BBUT_B,d1                               ; was B pressed?
            bne     .proc_fire                                      ; yes, so do something!
            rts                                                     ; exit!

; Now we need to find a free bullet object to use

.proc_fire: lea     RAPTOR_sprite_table+(ID_bullets*sprite_tabwidth),a0  ; point to RAPTOR object table
            moveq   #4,d7                                           ; there are 5 bullets
.find:      tst.l   sprite_active(a0)                               ; is this one active?
            bmi     .found                                          ; if -ve then it's unuse, we can use this one!
            lea     sprite_tabwidth(a0),a0                          ; next object!
            dbra    d7,.find                                        ; loop around all 10
            rts                                                     ; none free, exit!
.found:     move.w  #10,fire_delay                                  ; can't shoot again for 10 frames
            lea     RAPTOR_sprite_table+(ID_player*sprite_tabwidth),a1   ; point to RAPTOR address for the player
            move.l  sprite_x(a1),sprite_x(a0)                       ; copy player x to bullet x
            add.w   #6,sprite_x(a0)                                 ; add an offset so it's centered to the player
            move.l  sprite_y(a1),sprite_y(a0)                       ; copy player y to bullet y
            sub.w   #4,sprite_y(a0)                                 ; subtract an offset so it shoots from above the player
            move.w  #-2,sprite_yadd(a0)                             ; subtract 1 per frame from bullet y
            move.l  #can_hit,sprite_colchk(a0)                      ; tell RAPTOR this sprite can hit things
            move.l  #-1,sprite_was_hit(a0)                          ; clear the Object collision flag in case it was previously set)
            move.l  #1,sprite_hitpoint(a0)                          ; reset this Obejcts Hitpoint value
            move.l  #is_active,sprite_active(a0)                    ; tell RAPTOR this object is active!
            rts
```

The button will only be polled if 10 frames have passed since the last button press was registered. This prevents a constant stream of bullets being spawned.

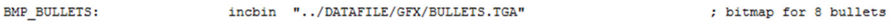
We then search through the available bullet objects to find an inactive one. Once found we set the repeat delay back to 10 frames and configure the new object.

We copy its X/Y position from the player and add an offset to center the object correctly. We then set the bullets *sprite_yadd* value to -2 so that it will track vertically up the screen on its own, with no intervention from us. Lastly we clear its collision flag and reset *sprite_hitpoints*, and finally we set *sprite_object* to active.

The bullet will spawn and move up the screen on its own, erasing itself when it goes off the top of the screen.

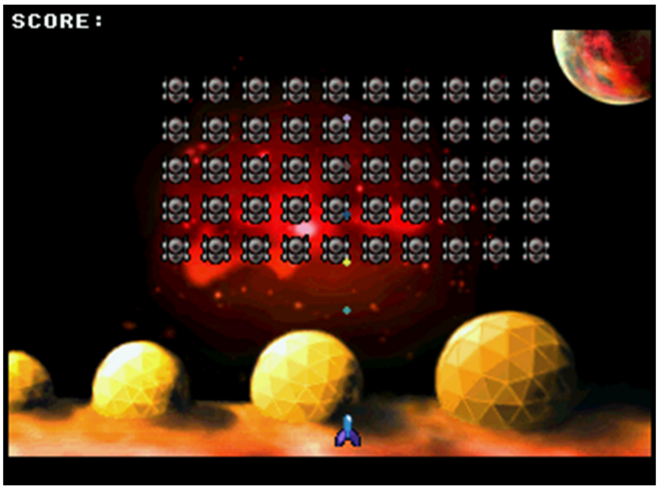
You will note that if passes through the enemies, even though it has its collision flags set. This is because no collision detection has been performed, which we will add next.

Finally, we include the bitmap for the bullets



```
BMP_BULLETS:            incbin  "../DATAFILE/GFX/BULLETS.TGA"       ; bitmap for 8 bullets
```

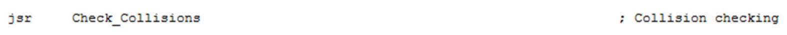
EX-04a – Screen Output



EX-04b – Object Collisions

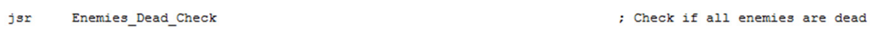
At this point, we have all our objects on the screen, their fields are set to enable collisions, hit points and damage, but nothing is happening when they collide. The reason for this is that we have not called the collision routine.

We will add another subroutine to the main loop:

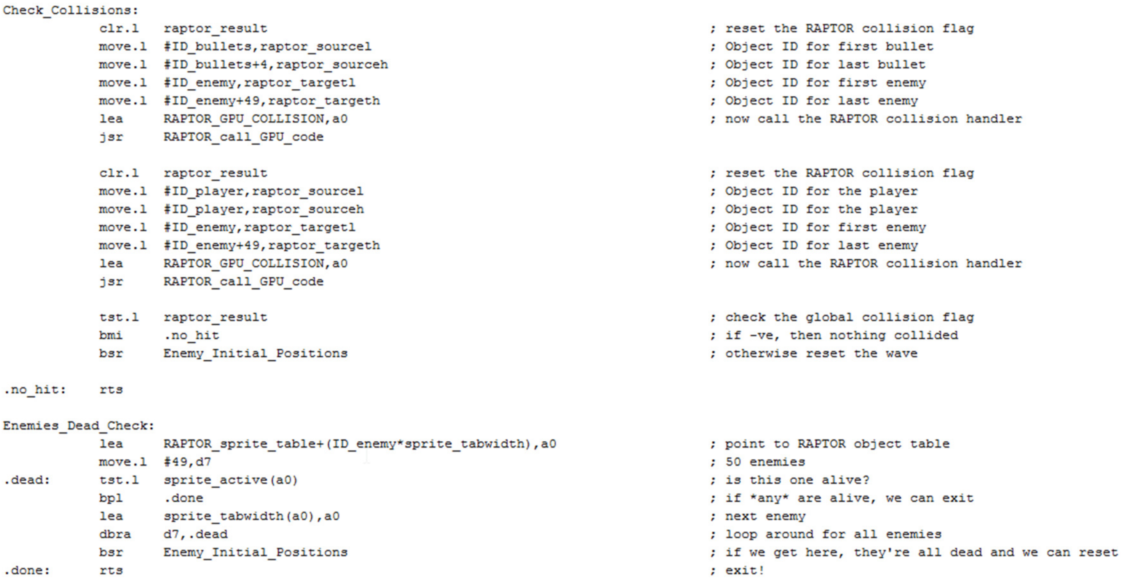


```
        jsr     Check_Collisions                                ; Collision checking
```

We will also add another subroutine to check if all the enemies are dead, and if so, we will reset the formation.



```
        jsr     Enemies_Dead_Check                              ; Check if all enemies are dead
```



```
Check_Collisions:
        clr.l   raptor_result                                   ; reset the RAPTOR collision flag
        move.l  #ID_bullets,raptor_sourcel                      ; Object ID for first bullet
        move.l  #ID_bullets+4,raptor_sourceh                    ; Object ID for last bullet
        move.l  #ID_enemy,raptor_targetl                        ; Object ID for first enemy
        move.l  #ID_enemy+49,raptor_targeth                     ; Object ID for last enemy
        lea     RAPTOR_GPU_COLLISION,a0                         ; now call the RAPTOR collision handler
        jsr     RAPTOR_call_GPU_code

        clr.l   raptor_result                                   ; reset the RAPTOR collision flag
        move.l  #ID_player,raptor_sourcel                       ; Object ID for the player
        move.l  #ID_player,raptor_sourceh                       ; Object ID for the player
        move.l  #ID_enemy,raptor_targetl                        ; Object ID for first enemy
        move.l  #ID_enemy+49,raptor_targeth                     ; Object ID for last enemy
        lea     RAPTOR_GPU_COLLISION,a0                         ; now call the RAPTOR collision handler
        jsr     RAPTOR_call_GPU_code

        tst.l   raptor_result                                   ; check the global collision flag
        bmi     .no_hit                                         ; if -ve, then nothing collided
        bsr     Enemy_Initial_Positions                         ; otherwise reset the wave

.no_hit:        rts

Enemies_Dead_Check:
        lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a0       ; point to RAPTOR object table
        move.l  #49,d7                                          ; 50 enemies
.dead:  tst.l   sprite_active(a0)                               ; is this one alive?
        bpl     .done                                           ; if *any* are alive, we can exit
        lea     sprite_tabwidth(a0),a0                          ; next enemy
        dbra    d7,.dead                                        ; loop around for all enemies
        bsr     Enemy_Initial_Positions                         ; if we get here, they're all dead and we can reset
.done:  rts                                                     ; exit!
```

First we reset the global collision flag. This flag will indicate if anything collided during the check. It will not tell us what collided, only that something did. For our purposed this will be sufficient at the moment.

The first call checks all the player bullets against all the enemies. We don't care if anything hit, because the collision engine will do hit-point deduction and remove any object that becomes 'dead'. As the damage on the bullet is set to 2, and the hit-point of the enemies are set to 1, a single hit will destroy an enemy. Because the flag on the enemy and the bullet is set to remove the object, RAPTOR will take it out of the visible list and set its status to '*is_inactive*'.

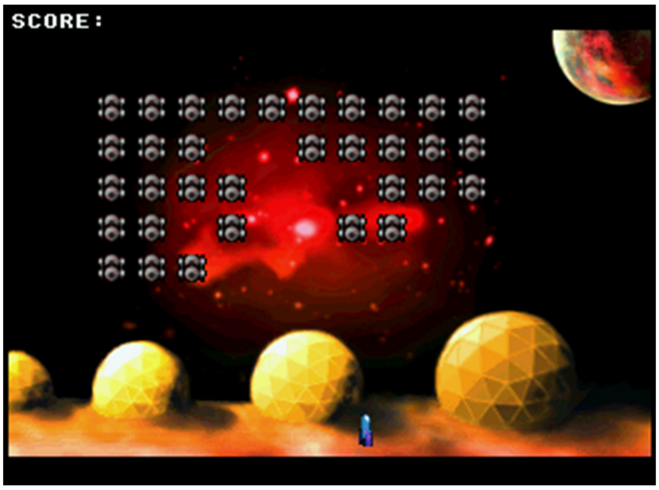
The next check compares all the enemies against the player. If the player was hit then the enemy formation is reset.

At this point we have a functional (if very simplistic) shoot ‘em up game without any code to manage:

- Enemy location
- Bullet location
- Player location
- Bullet movement
- Enemy Movement
- Enemy removal upon collision
- Bullet removal from the screen

This is where RAPTOR Engine puts the programmer in control of their game idea, rather than having them tied up with the complexity of the object management.

EX-04b – Screen Output



Functions Introduced

- RAPTOR_call_GPU_code
- RAPTOR_GPU_COLLISION

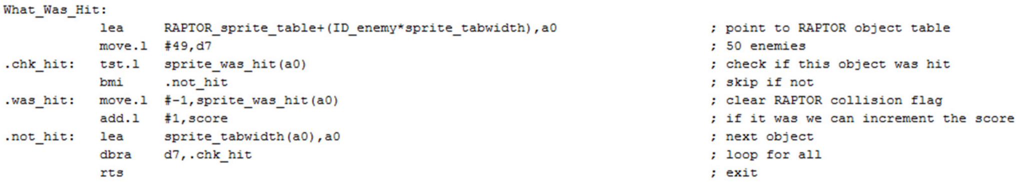
EX-04c – Identifying Collisions

For scoring purposes knowing that something was hit is not good enough. We need to know how many were hit so we can adjust the score accordingly. To do this we will identify what objects were hit during the collision detection function.

We will insert another subroutine call into the main loop:

```
          jsr     What_Was_Hit                                            ; Check what was hit
```

Which will execute the subroutine below:



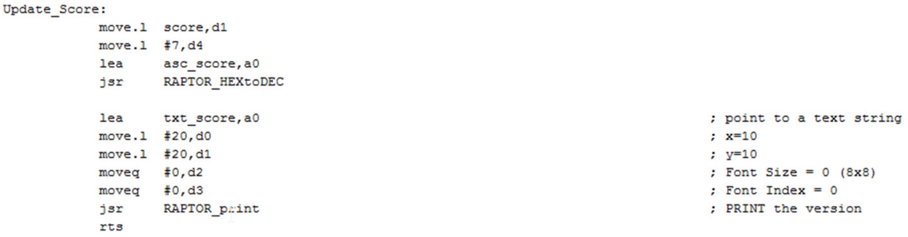
```
What_Was_Hit:
          lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a0       ; point to RAPTOR object table
          move.l  #49,d7                                                  ; 50 enemies
.chk_hit: tst.l   sprite_was_hit(a0)                                      ; check if this object was hit
          bmi     .not_hit                                                ; skip if not
.was_hit: move.l  #-1,sprite_was_hit(a0)                                  ; clear RAPTOR collision flag
          add.l   #1,score                                                ; if it was we can increment the score
.not_hit: lea     sprite_tabwidth(a0),a0                                  ; next object
          dbra    d7,.chk_hit                                             ; loop for all
          rts                                                             ; exit
```

This routine loops around the enemy table checking if the *sprite_was_hit* flag has been set. If it has, we reset it, and increment the score counter. This ensures we score 1 point for every enemy successfully hit by a bullet.

We now also need to update the score at the top of the screen, we will do this via another subroutine in the main loop

```
          jsr     Update_Score                                            ; Update the scoreboard
```

Which is defined as:

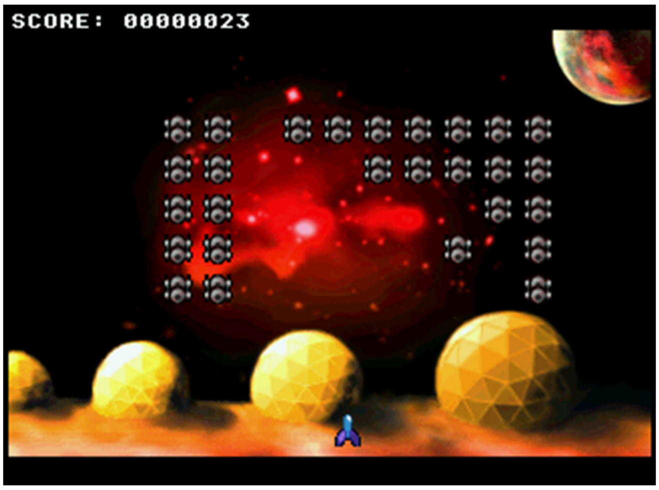


```
Update_Score:
          move.l  score,d1
          move.l  #7,d4
          lea     asc_score,a0
          jsr     RAPTOR_HEXtoDEC

          lea     txt_score,a0                                            ; point to a text string
          move.l  #20,d0                                                  ; x=10
          move.l  #20,d1                                                  ; y=10
          moveq   #0,d2                                                   ; Font Size = 0 (8x8)
          moveq   #0,d3                                                   ; Font Index = 0
          jsr     RAPTOR_print                                            ; PRINT the version
          rts
```

This introduced another command, *RAPTOR_HEXtoDEC* which converts a hexadecimal value (in this case the score) into an ASCII string.

D1 is loaded with the value to convert, D4 is set to the number of characters to display minus one. We are printing 8 characters in the score board ('00000000') so this is set to 7. Finally, A0 points to the output buffer and the routine is called.

EX-04c – Screen Output



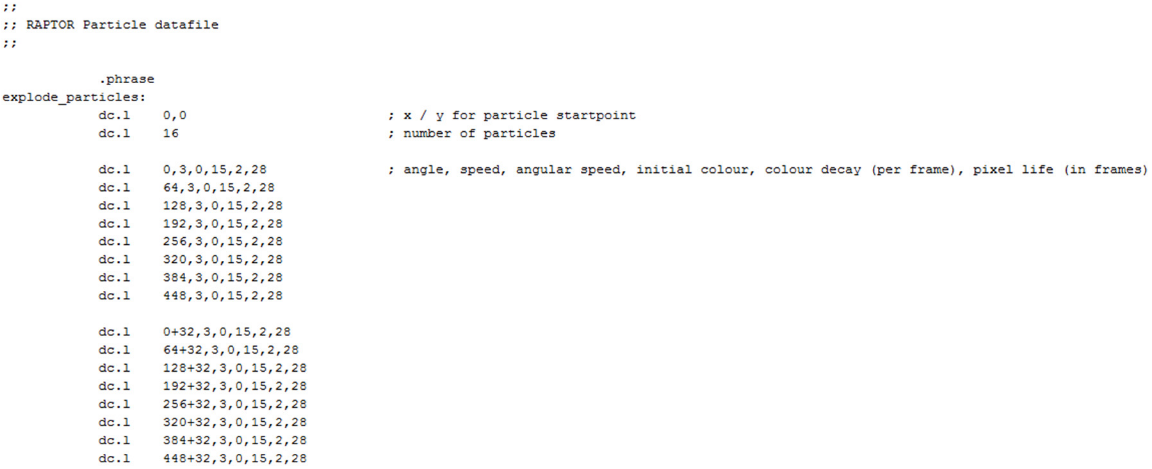
Functions Introduced

- RAPTOR_HEXtoDEC

EX-05 – Particle Effects

RAPTOR has a built in Particle (or Pixel) Engine that can inject and manage pixels on a per-pixel basis without any user intervention. We will use this in the example game to add an exploding effect when an enemy is hit.

First of all, we must define the particle effect in *_RAPPIXL.S*, as shown below:



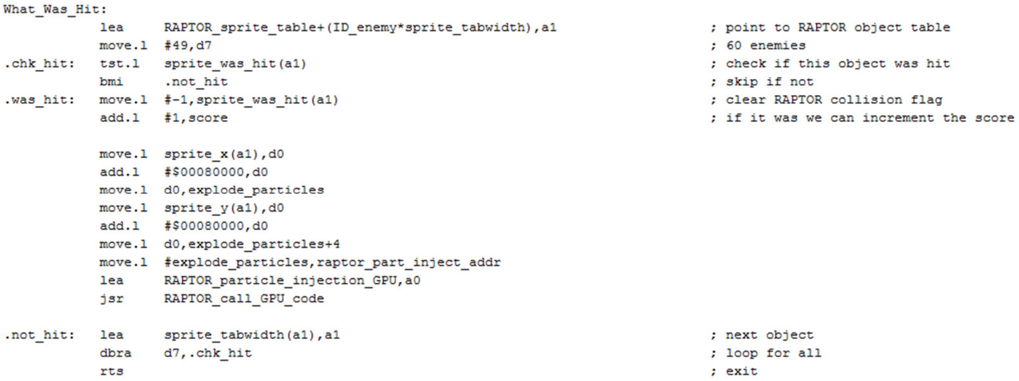
```
;;
;; RAPTOR Particle datafile
;;

            .phrase
explode_particles:
            dc.l    0,0                         ; x / y for particle startpoint
            dc.l    16                          ; number of particles

            dc.l    0,3,0,15,2,28               ; angle, speed, angular speed, initial colour, colour decay (per frame), pixel life (in frames)
            dc.l    64,3,0,15,2,28
            dc.l    128,3,0,15,2,28
            dc.l    192,3,0,15,2,28
            dc.l    256,3,0,15,2,28
            dc.l    320,3,0,15,2,28
            dc.l    384,3,0,15,2,28
            dc.l    448,3,0,15,2,28

            dc.l    0+32,3,0,15,2,28
            dc.l    64+32,3,0,15,2,28
            dc.l    128+32,3,0,15,2,28
            dc.l    192+32,3,0,15,2,28
            dc.l    256+32,3,0,15,2,28
            dc.l    320+32,3,0,15,2,28
            dc.l    384+32,3,0,15,2,28
            dc.l    448+32,3,0,15,2,28
```

This first two longwords contain the 16.16 X and Y co-ordinates that will act as the origin of the effect. We will later copy the co-ordinates of the destroyed enemy to this location before using the *RAPTOR_particle_injection_GPU function*.

Next, we store the number of particles (pixels) in this effect, in this case 16.

Then follows a line for each pixel detailing its initial angle, speed, angular speed, initial colour, colour decay rate and pixel lifetime. The pattern above will give a starburst effect, as the angles range from 0 to 512 and are evenly distributed.

We already have a '*what_was_hit*' routine, so we already know which enemies were hit by anything. We will modify this routine to add the explosion effect, as below:

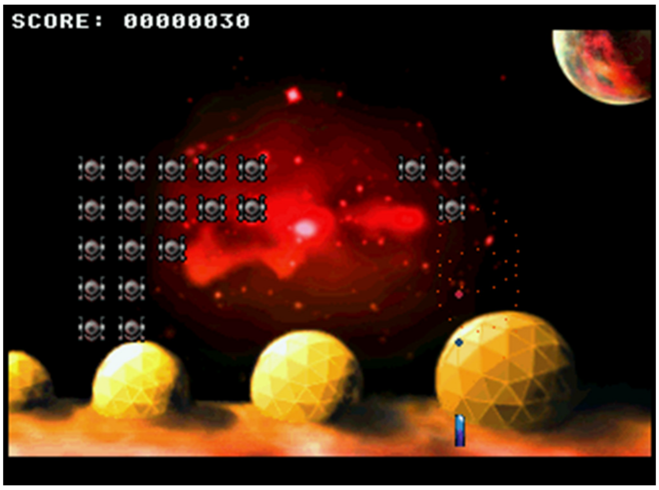


```
What_Was_Hit:
            lea     RAPTOR_sprite_table+(ID_enemy*sprite_tabwidth),a1                 ; point to RAPTOR object table
            move.l  #49,d7                                                            ; 60 enemies
.chk_hit:   tst.l   sprite_was_hit(a1)                                                ; check if this object was hit
            bmi     .not_hit                                                          ; skip if not
.was_hit:   move.l  #-1,sprite_was_hit(a1)                                            ; clear RAPTOR collision flag
            add.l   #1,score                                                          ; if it was we can increment the score

            move.l  sprite_x(a1),d0
            add.l   #$00080000,d0
            move.l  d0,explode_particles
            move.l  sprite_y(a1),d0
            add.l   #$00080000,d0
            move.l  d0,explode_particles+4
            move.l  #explode_particles,raptor_part_inject_addr
            lea     RAPTOR_particle_injection_GPU,a0
            jsr     RAPTOR_call_GPU_code

.not_hit:   lea     sprite_tabwidth(a1),a1                                            ; next object
            dbra    d7,.chk_hit                                                       ; loop for all
            rts                                                                       ; exit
```

If the enemy was hit we perform the following:

- Get the enemy X-position
- Add 8 pixels to it (remember, $00080000 is 8.0 in 16.16 sub-pixel format)
- Store this at the start of the defined particle effect in the pre-defined space for the X co-ordinate.
- Get the enemy Y-position
- Add 8 pixels to it
- Store this after the X position, completing the X/Y values
- Point the Particle Injection Routine at the effect table
- Call the *RAPTOR_particle_injection_GPU* function to start the effect

EX-05 – Screen Output



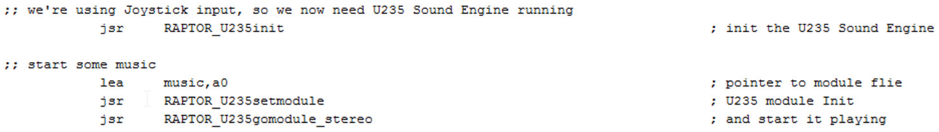
Functions Introduced

- RAPTOR_particle_injection_GPU

EX-06a – Audio (Music)

RAPTOR uses the U235 Sound Engine as its Audio subsystem. The Sound Engine can be accessed directly (see the U235 manual) or you can use the RAPTOR wrapper functions for audio output.

To start, we will get some music playing.

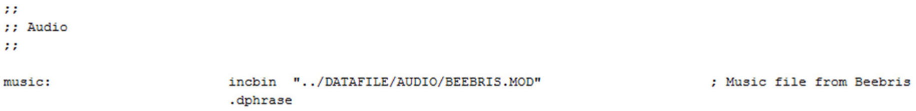


```
;; we're using Joystick input, so we now need U235 Sound Engine running
          jsr     RAPTOR_U235init                                              ; init the U235 Sound Engine

;; start some music
          lea     music,a0                                                     ; pointer to module flie
          jsr     RAPTOR_U235setmodule                                         ; U235 module Init
          jsr     RAPTOR_U235gomodule_stereo                                   ; and start it playing
```

Following the *RAPTOR_U235init* call we will set a pointer to our module file, and call *RAPTOR_U235setmodule* to initialize it.

We will then start it playing with *RAPTOR_U235gomodule_stereo* to start music playback.

In our data section at the bottom we will include the module file:



```
;;
;; Audio
;;

music:                   incbin  "../DATAFILE/AUDIO/BEEBRIS.MOD"                ; Music file from Beebris
                         .dphrase
```

Nothing else needs to be done, the music will play and loop to infinity.

Note: The RAPTOR wrappers will lock the replay frequency to 16khz. If you wish to use another frequency you will have to access the U235 Sound Engine directly.

EX-06a – Audio (Music)

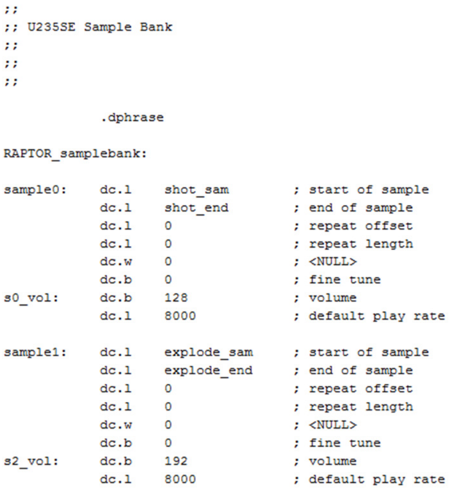
Functions Introduced

- RAPTOR_U235setmodule
- RAPTOR_U235gomodule_stereo

EX-06b – Audio (Sound Effects)

Now we will add some sound effects. To keep it simple we will just add effects for shooting and an explosion when something gets destroyed (an enemy or the player)

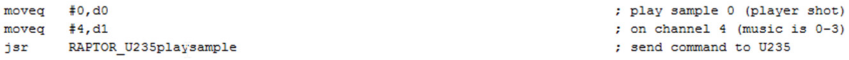
To do this we need to edit the *_RAPU235.S* file:



```
;;
;; U235SE Sample Bank
;;
;;
;;

            .dphrase

RAPTOR_samplebank:

sample0:    dc.l    shot_sam        ; start of sample
            dc.l    shot_end        ; end of sample
            dc.l    0               ; repeat offset
            dc.l    0               ; repeat length
            dc.w    0               ; <NULL>
            dc.b    0               ; fine tune
s0_vol:     dc.b    128             ; volume
            dc.l    8000            ; default play rate

sample1:    dc.l    explode_sam     ; start of sample
            dc.l    explode_end     ; end of sample
            dc.l    0               ; repeat offset
            dc.l    0               ; repeat length
            dc.w    0               ; <NULL>
            dc.b    0               ; fine tune
s2_vol:     dc.b    192             ; volume
            dc.l    8000            ; default play rate
```

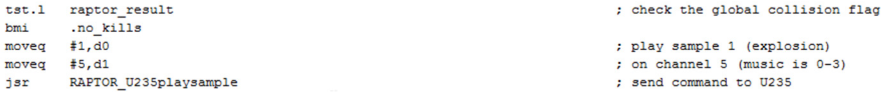
These definitions follow the exact same specification as described in the U235 Sound Engine manual.

At the end of the *Player_Shooting* subroutine we will add a call to play the sample:



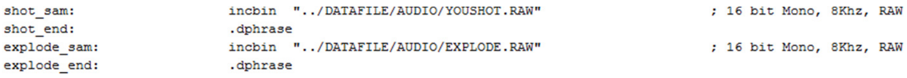
```
            moveq   #0,d0                                           ; play sample 0 (player shot)
            moveq   #4,d1                                           ; on channel 4 (music is 0-3)
            jsr     RAPTOR_U235playsample                           ; send command to U235
```

And in the *Check_Collisions* routine we will check the global flag *raptor_result* after the bullets vs enemies check, and all the following code:



```
            tst.l   raptor_result                                   ; check the global collision flag
            bmi     .no_kills
            moveq   #1,d0                                           ; play sample 1 (explosion)
            moveq   #5,d1                                           ; on channel 5 (music is 0-3)
            jsr     RAPTOR_U235playsample                           ; send command to U235
```

All that is left is to include the audio files into the binary:



```
shot_sam:                   incbin  "../DATAFILE/AUDIO/YOUSHOT.RAW"     ; 16 bit Mono, 8Khz, RAW
shot_end:                   .dphrase
explode_sam:                incbin  "../DATAFILE/AUDIO/EXPLODE.RAW"     ; 16 bit Mono, 8Khz, RAW
explode_end:                .dphrase
```

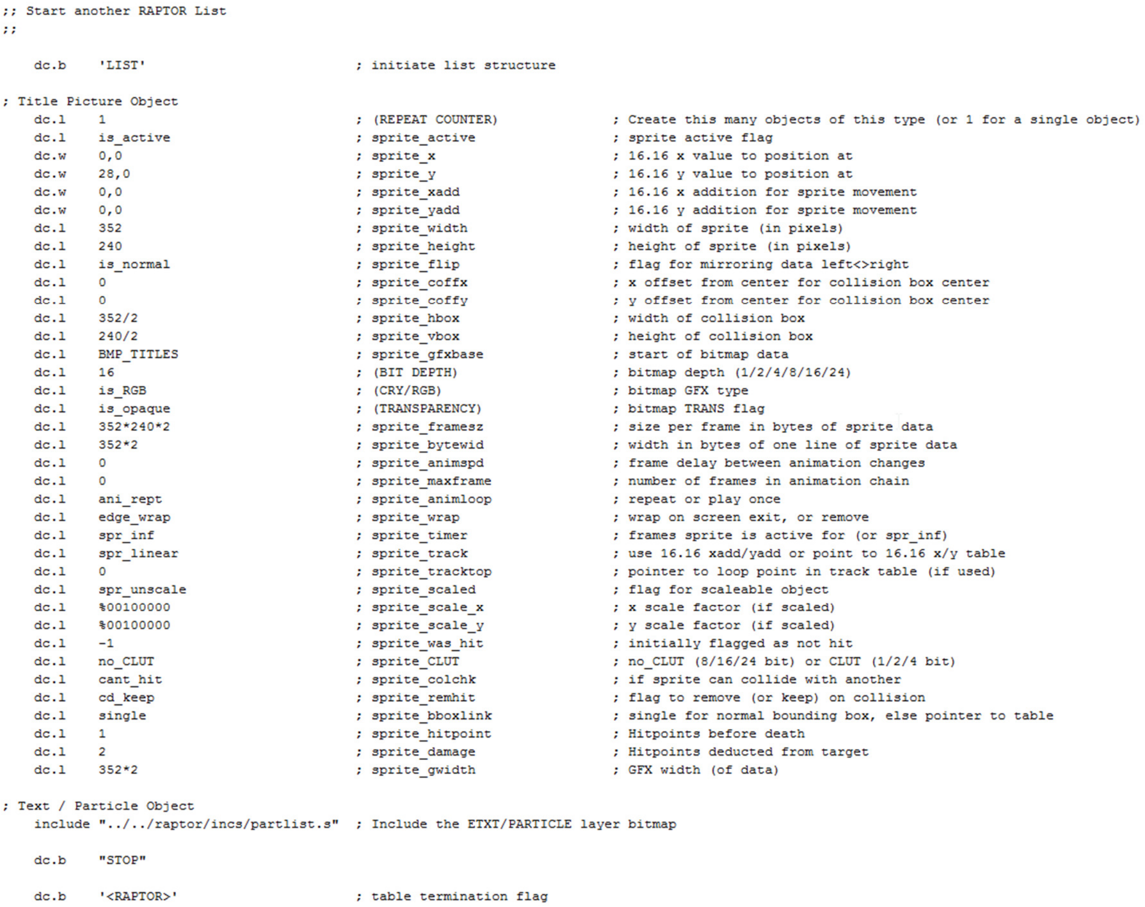
Functions Introduced

- RAPTOR_U235playsample

EX-07 – Multiple Lists

Now we have a playable (if simple game) – but what if we want to add a title screen? This is where multiple RAPTOR Lists are used.

In the file *_RAPINIT.S* we will add a second list:

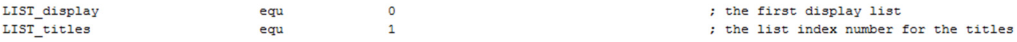


```
;; Start another RAPTOR List
;;

    dc.b    'LIST'                          ; initiate list structure

; Title Picture Object
    dc.l    1                               ; (REPEAT COUNTER)              ; Create this many objects of this type (or 1 for a single object)
    dc.l    is_active                       ; sprite_active                 ; sprite active flag
    dc.w    0,0                             ; sprite_x                      ; 16.16 x value to position at
    dc.w    28,0                            ; sprite_y                      ; 16.16 y value to position at
    dc.w    0,0                             ; sprite_xadd                   ; 16.16 x addition for sprite movement
    dc.w    0,0                             ; sprite_yadd                   ; 16.16 y addition for sprite movement
    dc.l    352                             ; sprite_width                  ; width of sprite (in pixels)
    dc.l    240                             ; sprite_height                 ; height of sprite (in pixels)
    dc.l    is_normal                       ; sprite_flip                   ; flag for mirroring data left<>right
    dc.l    0                               ; sprite_coffx                  ; x offset from center for collision box center
    dc.l    0                               ; sprite_coffy                  ; y offset from center for collision box center
    dc.l    352/2                           ; sprite_hbox                   ; width of collision box
    dc.l    240/2                           ; sprite_vbox                   ; height of collision box
    dc.l    BMP_TITLES                      ; sprite_gfxbase                ; start of bitmap data
    dc.l    16                              ; (BIT DEPTH)                   ; bitmap depth (1/2/4/8/16/24)
    dc.l    is_RGB                          ; (CRY/RGB)                     ; bitmap GFX type
    dc.l    is_opaque                       ; (TRANSPARENCY)                ; bitmap TRANS flag
    dc.l    352*240*2                       ; sprite_framesz                ; size per frame in bytes of sprite data
    dc.l    352*2                           ; sprite_bytewid                ; width in bytes of one line of sprite data
    dc.l    0                               ; sprite_animspd                ; frame delay between animation changes
    dc.l    0                               ; sprite_maxframe               ; number of frames in animation chain
    dc.l    ani_rept                        ; sprite_animloop               ; repeat or play once
    dc.l    edge_wrap                       ; sprite_wrap                   ; wrap on screen exit, or remove
    dc.l    spr_inf                         ; sprite_timer                  ; frames sprite is active for (or spr_inf)
    dc.l    spr_linear                      ; sprite_track                  ; use 16.16 xadd/yadd or point to 16.16 x/y table
    dc.l    0                               ; sprite_tracktop               ; pointer to loop point in track table (if used)
    dc.l    spr_unscale                     ; sprite_scaled                 ; flag for scaleable object
    dc.l    %00100000                       ; sprite_scale_x                ; x scale factor (if scaled)
    dc.l    %00100000                       ; sprite_scale_y                ; y scale factor (if scaled)
    dc.l    -1                              ; sprite_was_hit                ; initially flagged as not hit
    dc.l    no_CLUT                         ; sprite_CLUT                   ; no_CLUT (8/16/24 bit) or CLUT (1/2/4 bit)
    dc.l    cant_hit                        ; sprite_colchk                 ; if sprite can collide with another
    dc.l    cd_keep                         ; sprite_remhit                 ; flag to remove (or keep) on collision
    dc.l    single                          ; sprite_bboxlink               ; single for normal bounding box, else pointer to table
    dc.l    1                               ; sprite_hitpoint               ; Hitpoints before death
    dc.l    2                               ; sprite_damage                 ; Hitpoints deducted from target
    dc.l    352*2                           ; sprite_gwidth                 ; GFX width (of data)

; Text / Particle Object
    include "../../raptor/incs/partlist.s"  ; Include the ETXT/PARTICLE layer bitmap

    dc.b    "STOP"

    dc.b    '<RAPTOR>'                      ; table termination flag
```

As is shown above, we start the 2nd list with 'LIST' and end it with another 'STOP' before finally closing the list with '<RAPTOR>'.

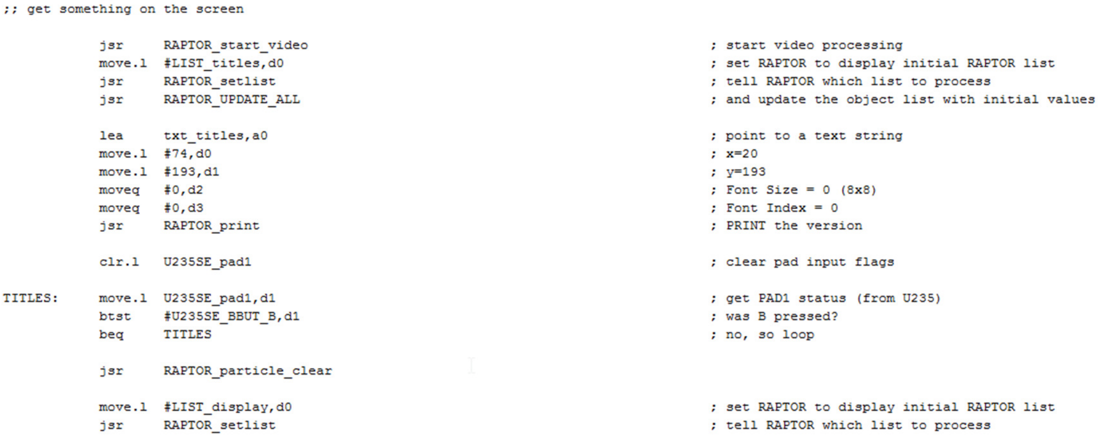
You will also note that we have again included the same file for a text layer. The text layer (or any object) can be included in multiple lists which is very useful for saving memory. It should be noted, however, that setting fields on the text/particle layer in this second list will not change any of the values for the text/particle later in other lists. While they share a common bitmap memory address they are two completely different RAPTOR objects.

We will also add a human friendly name for this list at the top of _RAPAPP.S as below:



```
LIST_display                equu        0                           ; the first display list
LIST_titles                 equ         1                           ; the list index number for the titles
```

And the following code to add our title screen:

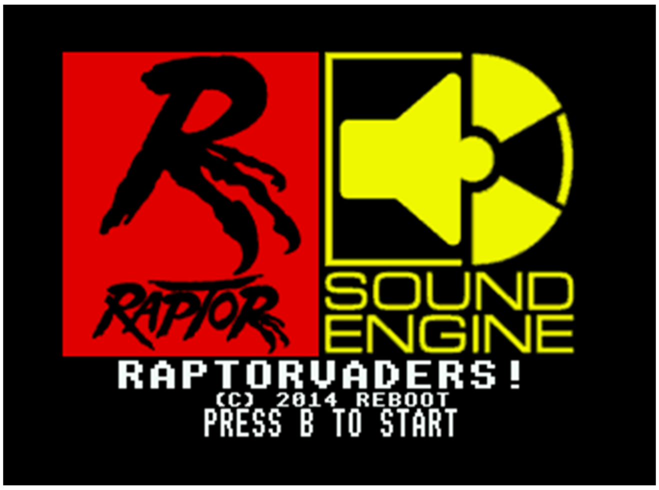


```
;; get something on the screen

            jsr     RAPTOR_start_video                                  ; start video processing
            move.l  #LIST_titles,d0                                     ; set RAPTOR to display initial RAPTOR list
            jsr     RAPTOR_setlist                                      ; tell RAPTOR which list to process
            jsr     RAPTOR_UPDATE_ALL                                   ; and update the object list with initial values

            lea     txt_titles,a0                                       ; point to a text string
            move.l  #74,d0                                              ; x=20
            move.l  #193,d1                                             ; y=193
            moveq   #0,d2                                               ; Font Size = 0 (8x8)
            moveq   #0,d3                                               ; Font Index = 0
            jsr     RAPTOR_print                                        ; PRINT the version

            clr.l   U235SE_pad1                                         ; clear pad input flags

TITLES:     move.l  U235SE_pad1,d1                                      ; get PAD1 status (from U235)
            btst    #U235SE_BBUT_B,d1                                   ; was B pressed?
            beq     TITLES                                              ; no, so loop

            jsr     RAPTOR_particle_clear

            move.l  #LIST_display,d0                                    ; set RAPTOR to display initial RAPTOR list
            jsr     RAPTOR_setlist                                      ; tell RAPTOR which list to process
```

We set the list, print some text and then loop around until B is pressed on the Jagpad. At that point we clear the text screen (we're using the same buffer in the game, so we have to clear it or the text will remain in the game screen), set the game display list and continue on with the game.

RAPTOR can support up to 16 Lists, numbered 0-15.

EX-07 – Screen Output



Functions Introduced

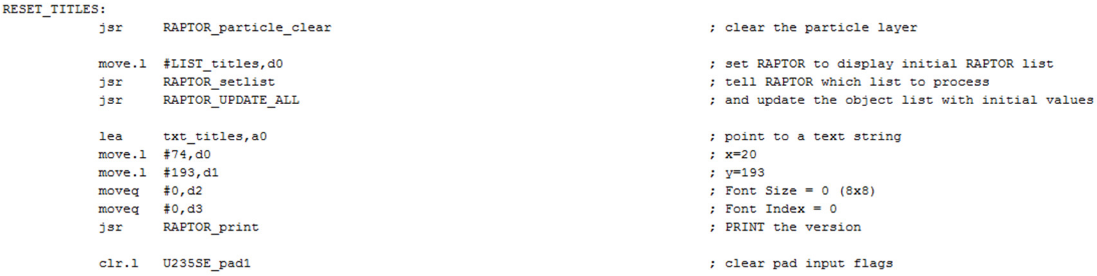
- RAPTOR_setlist

EX-08 – Interlude (Some Game Stuff)

This chapter will not introduce any new RAPTOR functions but will instead add some extra code to the game so that when the player dies the game returns to the title screen.

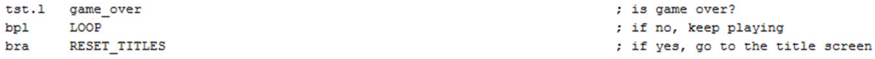
This is to allow us to demonstrate the MemoryTrack and Highscore routines in the next chapter.

Firstly, we modify the title screen code so that there is a label we can jump to which will display the screen:



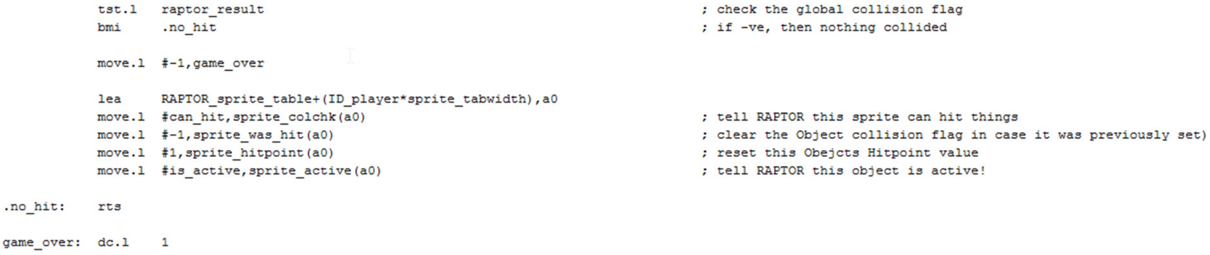
```
RESET_TITLES:
            jsr     RAPTOR_particle_clear                                               ; clear the particle layer

            move.l  #LIST_titles,d0                                                     ; set RAPTOR to display initial RAPTOR list
            jsr     RAPTOR_setlist                                                      ; tell RAPTOR which list to process
            jsr     RAPTOR_UPDATE_ALL                                                   ; and update the object list with initial values

            lea     txt_titles,a0                                                       ; point to a text string
            move.l  #74,d0                                                              ; x=20
            move.l  #193,d1                                                             ; y=193
            moveq   #0,d2                                                               ; Font Size = 0 (8x8)
            moveq   #0,d3                                                               ; Font Index = 0
            jsr     RAPTOR_print                                                        ; PRINT the version

            clr.l   U235SE_pad1                                                         ; clear pad input flags
```

Next we will modify the main loop. Up to now, the loop has executed to infinity. We will now add a conditional loop such that it only repeats if the '*game_over*' flag isn't set:



```
            tst.l   game_over                                                           ; is game over?
            bpl     LOOP                                                                ; if no, keep playing
            bra     RESET_TITLES                                                        ; if yes, go to the title screen
```

And finally, in the *Check_Collisions* routine we will add some code to set this flag if the player is hit. We will also play the sample for the explosion.



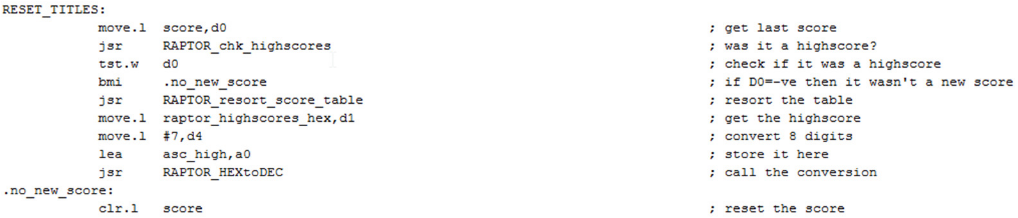
```
            tst.l   raptor_result                                                       ; check the global collision flag
            bmi     .no_hit                                                             ; if -ve, then nothing collided

            move.l  #-1,game_over

            lea     RAPTOR_sprite_table+(ID_player*sprite_tabwidth),a0
            move.l  #can_hit,sprite_colchk(a0)                                          ; tell RAPTOR this sprite can hit things
            move.l  #-1,sprite_was_hit(a0)                                              ; clear the Object collision flag in case it was previously set)
            move.l  #1,sprite_hitpoint(a0)                                              ; reset this Obejcts Hitpoint value
            move.l  #is_active,sprite_active(a0)                                        ; tell RAPTOR this object is active!

.no_hit:    rts

game_over:  dc.l    1
```

Now we have a version where you start on the title screen, and can play the game until you die, at which point you will return to the title screen.

EX-09a – High Scores

RAPTOR has its own internal high score management system. It is capable of managing a top 10 score list, stored as a 32-bit integer, and an 8 character ASCII name for each entry.

We will add the code to manage the high scores to the start of the title screen code, as below:



```
RESET_TITLES:
            move.l  score,d0                                                    ; get last score
            jsr     RAPTOR_chk_highscores                                       ; was it a highscore?
            tst.w   d0                                                          ; check if it was a highscore
            bmi     .no_new_score                                               ; if D0=-ve then it wasn't a new score
            jsr     RAPTOR_resort_score_table                                   ; resort the table
            move.l  raptor_highscores_hex,d1                                    ; get the highscore
            move.l  #7,d4                                                       ; convert 8 digits
            lea     asc_high,a0                                                 ; store it here
            jsr     RAPTOR_HEXtoDEC                                             ; call the conversion
.no_new_score:
            clr.l   score                                                       ; reset the score
```

To check if a score is in the top 10, load that score into D0 and call the function *RAPTOR_chk_highscores*. This will return a result in D0. If D0 is negative the score did not equal or beat any score in the high score table and nothing further needs to be done.

If, however, D0 is positive the scoreboard will need to be updated. The reason this is not done automatically with the single function is that you might want to test the score, and if it is a new entry you might want to ask the player to enter their name before sorting the table.

If you wish to submit a name to the function as well as a score then A0 must be loaded with a pointer to an 8 character string. In this example we are just storing the score, so nothing is loaded into A0 before calling the update function, *RAPTOR_resort_score_table*.

We then convert the new high score to ASCII ready for display and continue.

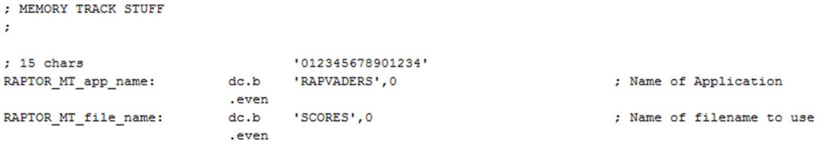
Functions Introduced

- RAPTOR_chk_highscores
- RAPTOR_resort_score_table

EX-09b - MemoryTrack

During the *RAPTOR_HWinit* function RAPTOR will check for the presence of a MemoryTrack device.

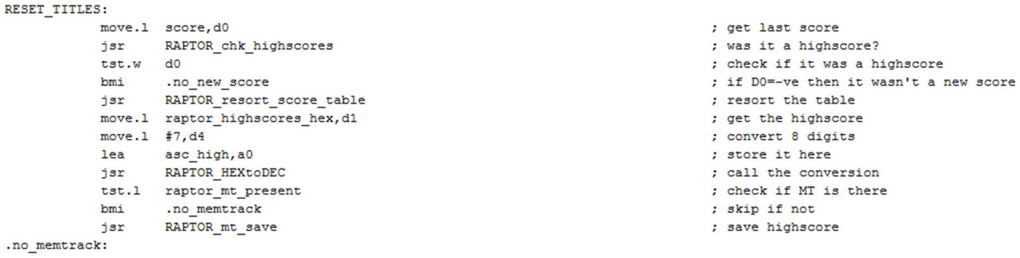
If one is detected it will attempt to load previously saved data from the Application and Filename specified in *_RAPINIT.S*, in our example as below:



```
; MEMORY TRACK STUFF
;

; 15 chars                          '012345678901234'
RAPTOR_MT_app_name:         dc.b    'RAPVADERS',0                   ; Name of Application
                            .even
RAPTOR_MT_file_name:        dc.b    'SCORES',0                      ; Name of filename to use
                            .even
```

This means that the application only has to take care of saving the data, and can assume any previously saved data is loaded at runtime.

To save the high score to MemoryTrack we will modify the routine we added for the high score as below:



```
RESET_TITLES:
            move.l  score,d0                                        ; get last score
            jsr     RAPTOR_chk_highscores                           ; was it a highscore?
            tst.w   d0                                              ; check if it was a highscore
            bmi     .no_new_score                                   ; if D0=-ve then it wasn't a new score
            jsr     RAPTOR_resort_score_table                       ; resort the table
            move.l  raptor_highscores_hex,d1                        ; get the highscore
            move.l  #7,d4                                           ; convert 8 digits
            lea     asc_high,a0                                     ; store it here
            jsr     RAPTOR_HEXtoDEC                                 ; call the conversion
            tst.l   raptor_mt_present                               ; check if MT is there
            bmi     .no_memtrack                                    ; skip if not
            jsr     RAPTOR_mt_save                                  ; save highscore
.no_memtrack:
```

Before we call *RAPTOR_mt_save* we test if the MemoryTrack was present by checking the *raptor_mt_present* variable, if it wasn't then we skip over the save function.

Functions Introduced

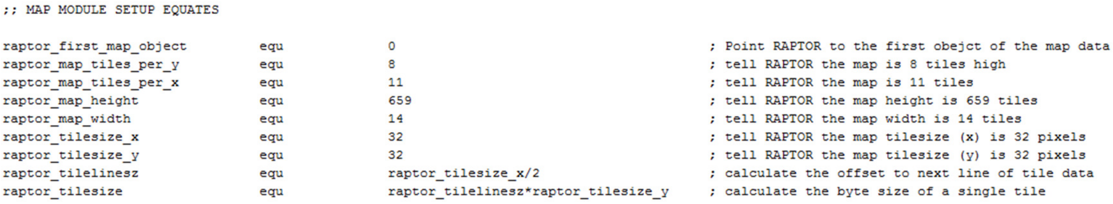
- RAPTOR_mt_save

EX-10 – Tile Maps

The example game does not use Tile Maps, so this is a completely different example to the previous ones.

We will have new *_RAPAPP.S* and *_RAPINIT.S* files, and will also introduce RAPTOR List Branch Objects.

First, we must set the *equates* at the top of *_RAPAPP.S* to specify the parameters of the tile map to be used:

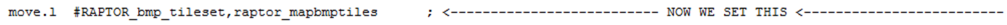


```
;; MAP MODULE SETUP EQUATES

raptor_first_map_object        equu            0                                   ; Point RAPTOR to the first obejct of the map data
raptor_map_tiles_per_y         equ             8                                   ; tell RAPTOR the map is 8 tiles high
raptor_map_tiles_per_x         equ             11                                  ; tell RAPTOR the map is 11 tiles
raptor_map_height              equ             659                                 ; tell RAPTOR the map height is 659 tiles
raptor_map_width               equ             14                                  ; tell RAPTOR the map width is 14 tiles
raptor_tilesize_x              equ             32                                  ; tell RAPTOR the map tilesize (x) is 32 pixels
raptor_tilesize_y              equ             32                                  ; tell RAPTOR the map tilesize (y) is 32 pixels
raptor_tilelinesz              equ             raptor_tilesize_x/2                 ; calculate the offset to next line of tile data
raptor_tilesize                equ             raptor_tilelinesz*raptor_tilesize_y ; calculate the byte size of a single tile
```

Let's break these down in detail:

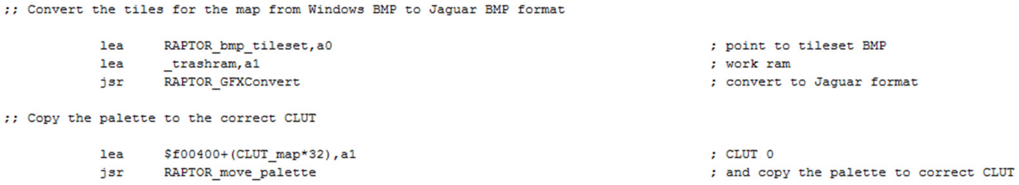
raptor_first_map_object	0	This is the RAPTOR List Object ID for the first object in the tilemap
raptor_map_tiles_per_y	8	How many tiles to render on screen vertically
raptor_map_tiles_per_x	11	How many tiles to render on screen horizontally
raptor_map_height	659	The height of the map in tiles
raptor_map_width	14	The width of the map in tiles
raptor_tilesize_x	32	The pixel width of a single tile
raptor_tilesize_y	32	The pixel height of a single tile
raptor_tilelinesz	Calculated	Tile image offset in bytes
raptor_tilesize	Calculated	Size in bytes of a single tile

We also need to set the pointer to the tile bitmap data before calling *RAPTOR_HWinit*.



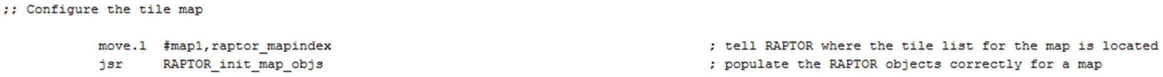
```
        move.l  #RAPTOR_bmp_tileset,raptor_mapbmptiles      ; <------------------------- NOW WE SET THIS <------------------------
```

The tiles for the tile map are restricted to 16 colour, 4bpp, and are stored in Windows BMP format, which means we need to convert them into Jaguar bitmap format and also move the palette into the correct CLUT.



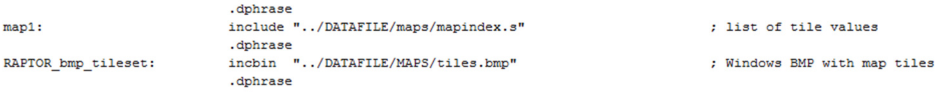
```
;; Convert the tiles for the map from Windows BMP to Jaguar BMP format

        lea     RAPTOR_bmp_tileset,a0                           ; point to tileset BMP
        lea     _trashram,a1                                    ; work ram
        jsr     RAPTOR_GFXConvert                               ; convert to Jaguar format

;; Copy the palette to the correct CLUT

        lea     $f00400+(CLUT_map*32),a1                        ; CLUT 0
        jsr     RAPTOR_move_palette                             ; and copy the palette to correct CLUT
```

The RAPTOR objects for the tile map must then be initialized by calling *RAPTOR_init_map_objs*.



```
;; Configure the tile map

        move.l  #map1,raptor_mapindex                           ; tell RAPTOR where the tile list for the map is located
        jsr     RAPTOR_init_map_objs                            ; populate the RAPTOR objects correctly for a map
```

All that is left to do now to complete the setup in *_RAPAPP.S* is to include the tile bitmap data and the map index values:

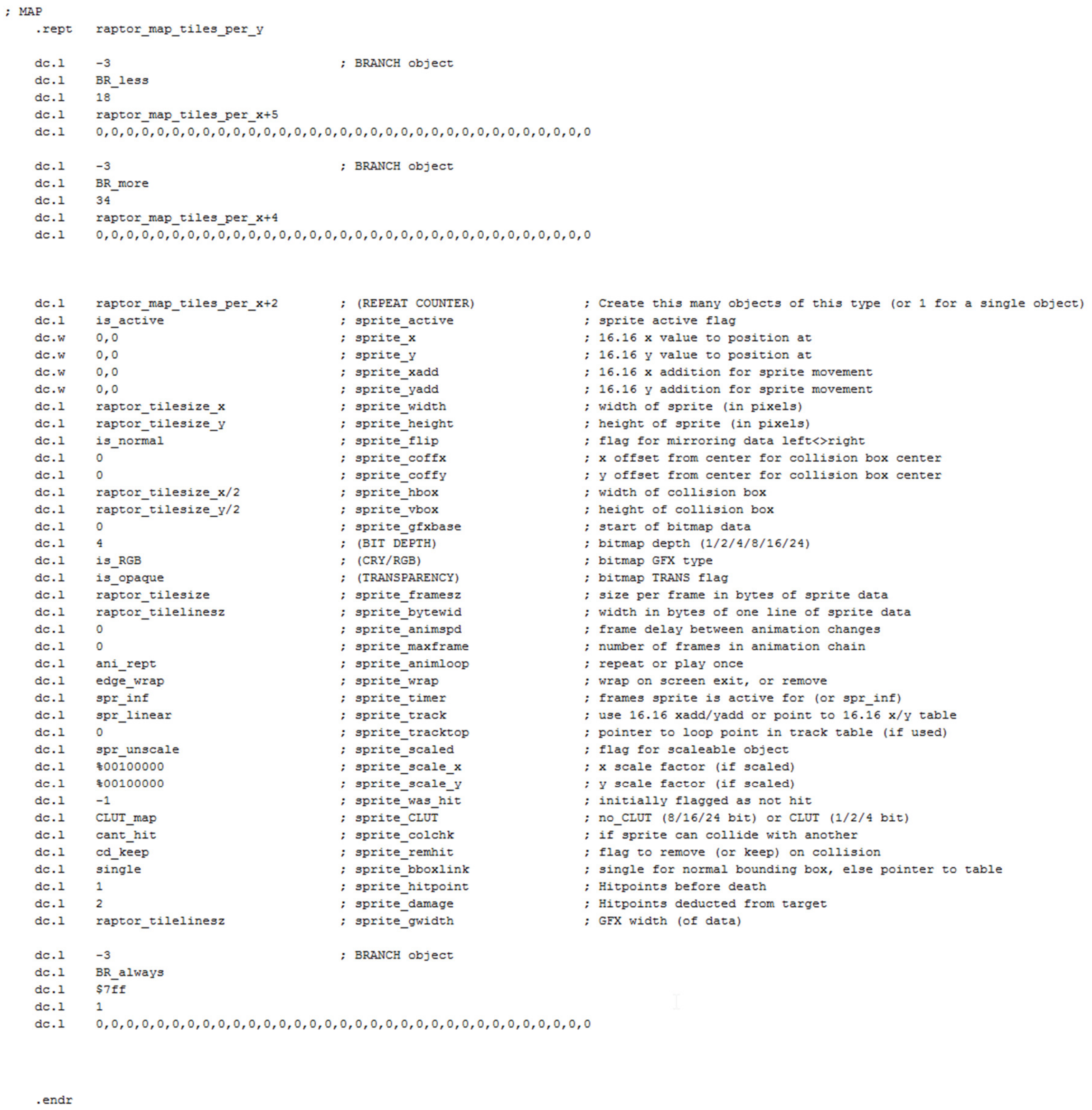


```
                        .dphrase
map1:                   include "../DATAFILE/maps/mapindex.s"           ; list of tile values
                        .dphrase
RAPTOR_bmp_tileset:     incbin  "../DATAFILE/MAPS/tiles.bmp"            ; Windows BMP with map tiles
                        .dphrase
```

The tiles are stored vertically, with tile index 0 at the top. The index data is stored as a string of words, with one word representing the index offset into the bitmap data for that tile.

The remaining setup is completed in the *_RAPINIT.S* file.

The List for a tile map must conform to the following structure:



```
; MAP
    .rept   raptor_map_tiles_per_y

    dc.l    -3                              ; BRANCH object
    dc.l    BR_less
    dc.l    18
    dc.l    raptor_map_tiles_per_x+5
    dc.l    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0

    dc.l    -3                              ; BRANCH object
    dc.l    BR_more
    dc.l    34
    dc.l    raptor_map_tiles_per_x+4
    dc.l    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0


    dc.l    raptor_map_tiles_per_x+2        ; (REPEAT COUNTER)              ; Create this many objects of this type (or 1 for a single object)
    dc.l    is_active                       ; sprite_active                 ; sprite active flag
    dc.w    0,0                             ; sprite_x                      ; 16.16 x value to position at
    dc.w    0,0                             ; sprite_y                      ; 16.16 y value to position at
    dc.w    0,0                             ; sprite_xadd                   ; 16.16 x addition for sprite movement
    dc.w    0,0                             ; sprite_yadd                   ; 16.16 y addition for sprite movement
    dc.l    raptor_tilesize_x               ; sprite_width                  ; width of sprite (in pixels)
    dc.l    raptor_tilesize_y               ; sprite_height                 ; height of sprite (in pixels)
    dc.l    is_normal                       ; sprite_flip                   ; flag for mirroring data left<>right
    dc.l    0                               ; sprite_coffx                  ; x offset from center for collision box center
    dc.l    0                               ; sprite_coffy                  ; y offset from center for collision box center
    dc.l    raptor_tilesize_x/2             ; sprite_hbox                   ; width of collision box
    dc.l    raptor_tilesize_y/2             ; sprite_vbox                   ; height of collision box
    dc.l    0                               ; sprite_gfxbase                ; start of bitmap data
    dc.l    4                               ; (BIT DEPTH)                   ; bitmap depth (1/2/4/8/16/24)
    dc.l    is_RGB                          ; (CRY/RGB)                     ; bitmap GFX type
    dc.l    is_opaque                       ; (TRANSPARENCY)                ; bitmap TRANS flag
    dc.l    raptor_tilesize                 ; sprite_framesz                ; size per frame in bytes of sprite data
    dc.l    raptor_tilelinesz               ; sprite_bytewid                ; width in bytes of one line of sprite data
    dc.l    0                               ; sprite_animspd                ; frame delay between animation changes
    dc.l    0                               ; sprite_maxframe               ; number of frames in animation chain
    dc.l    ani_rept                        ; sprite_animloop               ; repeat or play once
    dc.l    edge_wrap                       ; sprite_wrap                   ; wrap on screen exit, or remove
    dc.l    spr_inf                         ; sprite_timer                  ; frames sprite is active for (or spr_inf)
    dc.l    spr_linear                      ; sprite_track                  ; use 16.16 xadd/yadd or point to 16.16 x/y table
    dc.l    0                               ; sprite_tracktop               ; pointer to loop point in track table (if used)
    dc.l    spr_unscale                     ; sprite_scaled                 ; flag for scaleable object
    dc.l    %00100000                       ; sprite_scale_x                ; x scale factor (if scaled)
    dc.l    %00100000                       ; sprite_scale_y                ; y scale factor (if scaled)
    dc.l    -1                              ; sprite_was_hit                ; initially flagged as not hit
    dc.l    CLUT_map                        ; sprite_CLUT                   ; no_CLUT (8/16/24 bit) or CLUT (1/2/4 bit)
    dc.l    cant_hit                        ; sprite_colchk                 ; if sprite can collide with another
    dc.l    cd_keep                         ; sprite_remhit                 ; flag to remove (or keep) on collision
    dc.l    single                          ; sprite_bboxlink               ; single for normal bounding box, else pointer to table
    dc.l    1                               ; sprite_hitpoint               ; Hitpoints before death
    dc.l    2                               ; sprite_damage                 ; Hitpoints deducted from target
    dc.l    raptor_tilelinesz               ; sprite_gwidth                 ; GFX width (of data)

    dc.l    -3                              ; BRANCH object
    dc.l    BR_always
    dc.l    $7ff
    dc.l    1
    dc.l    0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0


    .endr
```

We will create *raptor_map_tiles_per_y* number of copies of the above, defining one for each required row in the display, which is achieved using the .rept and .endr assembler directives.

In order to optimise the tile map for speed, branch objects are used to skip over the list objects for rows that do not need to be displayed. These branch objects are dynamically updated by the GPU when the *RAPTOR_map_set_position* function is called.

Each horizontal row of tiles to be displayed will be padded with two branches at the start, and a branch at the end. The first branch will skip over the row if the current Y position is less than the line the tiles need to be displayed on. The second branch will skip over the row if the gap between the first and the second is more than the pixel height of the tiles.

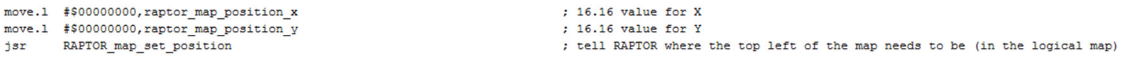
The final branch at the end will skip to the end of the map objects as we will know no other tiles need to be checked if the current line was displayed.

The format of a List Branch Object is as below:

-3	Denotes that this is a Branch Object
BR_(type)	Where (type) is **less**, **more** or **always**
Y comparator	Y value to use as a comparison for the previous less or more types
Number of objects to skip	Number of objects to skip if branch is taken
Padding	Padding to make a Branch Object the same size as a normal object

Note: RAPTOR tile maps can be 8,16,32 or 64 pixels wide. Any size below 32x32 will need a smaller display window as the number of used obejcts significantly increases.

In order to set the map co-ordinates we need to use the *RAPTOR_map_set_position* function:



```
move.l  #$00000000,raptor_map_position_x                        ; 16.16 value for X
move.l  #$00000000,raptor_map_position_y                        ; 16.16 value for Y
jsr     RAPTOR_map_set_position                                 ; tell RAPTOR where the top left of the map needs to be (in the logical map)
```

The *raptor_map_position_x* and *raptor_map_positoin_y* values are 16.16 pixel offsets (not tile offsets) from the top left of the map.

Also in the EX-10 folder is a subfolder called *MAPPY*.

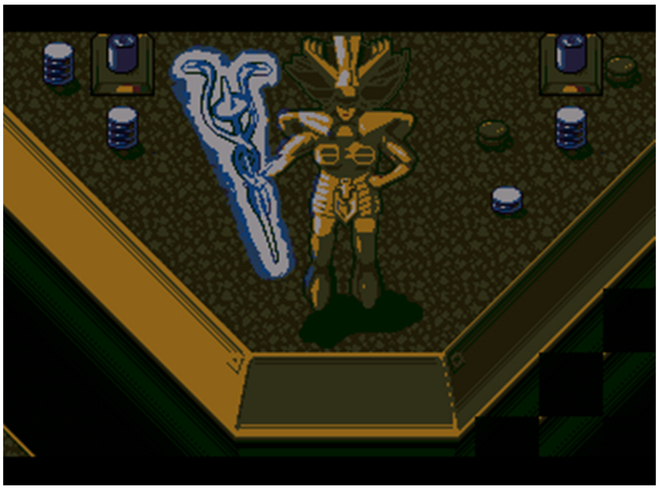
Mappy is a tile map editing package that can be downloaded from:

http://tilemap.co.uk/index.html

The map for Project One (Reboot's first release) has been provided in mappy format, along with a general purpose LUA script that will export the index data from any mappy map file in RAPTOR Index format, ready for including in a RAPTOR project.

Please note that the example uses a version of the Project One map that has been converted to 16 colours. The original map from the game uses 256 colours.

EX-10 – Tile Maps



Functions Introduced

- RAPTOR_init_map_objs
- RAPTOR_map_set_position

RAPTOR Example Games

The following games were all written with various versions of RAPTOR Engine:

Degz:

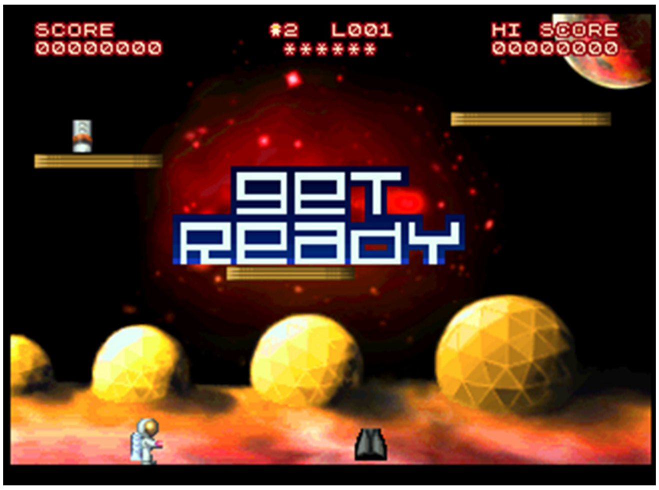
http://reboot.atari.org/new-reboot/degz.html

Expressway:

http://reboot.atari.org/new-reboot/expressway.html

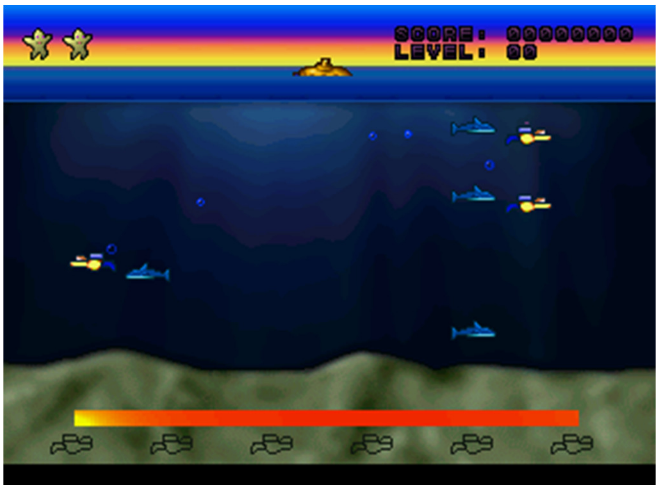
Full Circle: Rocketeer:

http://reboot.atari.org/new-reboot/rocketeer.html



HMS Raptor:

http://reboot.atari.org/new-reboot/hmsr.html

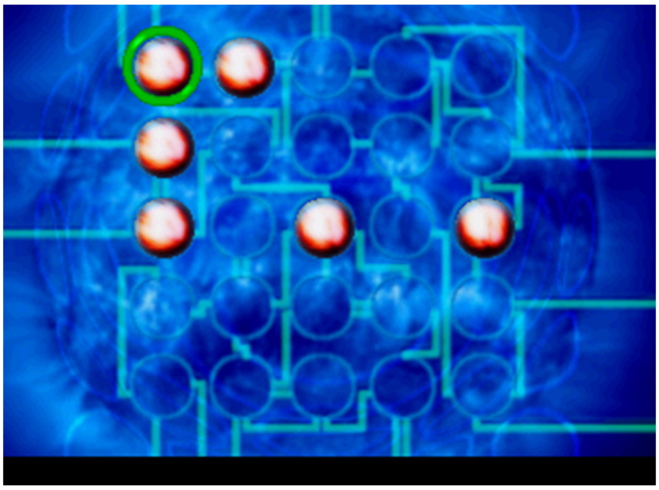


Kobayashi Maru:

http://reboot.atari.org/new-reboot/kobayashi.html

Rocks Off!:

http://reboot.atari.org/new-reboot/rocksoff.html



Rebooteroids:

http://reboot.atari.org/new-reboot/rebooteroids.html

II:

http://rgcd.bigcartel.com/product/jagware-collection-1-0-atari-jaguar-cd

Changes and Updates

- Typo: obejct change to object in all _RAPAPP.S files
- Comments in RAPTOR_print in all _RAPAPP.S files corrected
- L7ZZ replaced with LZ77 on page 46 of the manual
- raptor_liststart exposed – holds the base address of current list
- RAPTOR_particle_trigtable exposed – start of the internal angle table