Create your own Spectrum game Workshop using *The MK2* Engine

The MK2 Engine is written by the Mojon Twins as an evolutionary upgrade of the MK1 Engine also know as the Churro Marker. The tutorials are written by Nathan to describe the exciting features of the MK2 engine.
The MK2 Engine is a natural progression of the Churro Maker Engine with many of the features used in the Churro Maker Engine. MK2 code is more modularized and 90% of the code has been rewritten for speed and lower memory constraints. There is more support for 128k available with the MK2 engine as well as sound. Most programs written in the Churro Engine can be used with the MK2 Engine with some minor changes. With that in mind, the MK2 Engine is stronger, faster and smaller. This document is more technical than the Churro Maker Engine, so it is best to read the Churro Maker document before tackling the MK2 engine.

Enjoy the reading about the advances of the MK2 Engine.

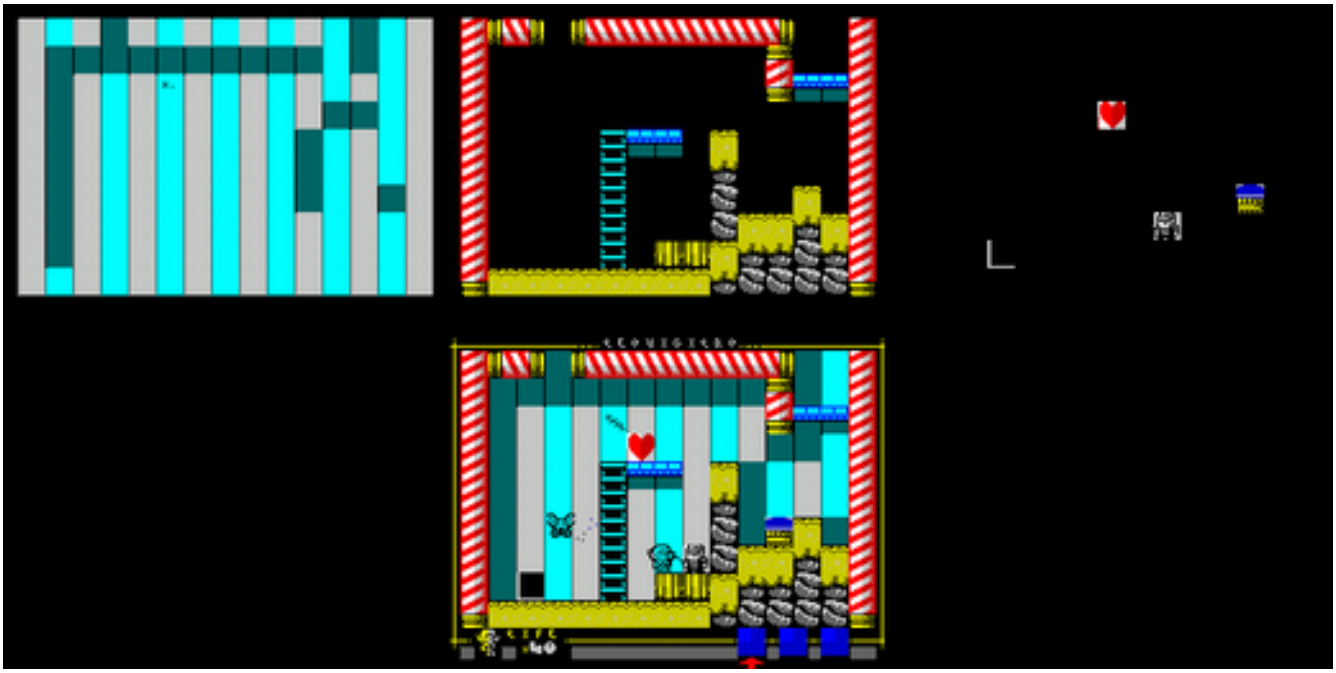
Spectrum games using the MK2 engine
Tutorial 1

Construction of screens in MK2

This tutorial applies to 0.88c (or later) version of MK2

MK2 has a construction engine greatly improved screens that allow us to get very good results without using a map "spread" of 48 tiles. To do this, the screen is divided into three logical levels: background, map, and decorations. Of course, you can continue to use normal maps of 16 or 48 tiles exactly as you did in La Churrera (well, almost the same: now the map converter will detect if your map is automatically packed or unpacked).

To see exactly how it works take as an example this screen of the third load of Leovigildo:



Background

To use a background, we will have to leave the empty tile 0 and designing our map considering that the fund will look through the boxes where we put this tile 0. It is as "transparent tile".

The background is programmatically generated by modifying a certain section of the draw_scr_background function defined in the /dev/engine/drawscr.h file. What is done is to detect if the tile that we extracted from the map is worth 0, in which case it is replaced by a tile of background. Basically, it's about adding code in this chunk:

```
// CUSTOM {
// Construction of assets.
      if (0 == gpd) {
      }
// } END OF CUSTOM
```

Here we can do practically what we want. Using gpx and gpy, which are the coordinates of the tile being drawn, we just have to write the correct value in gpd. For example, we can use a 15x10 tiles tilemap with a predesigned background, as done in Sir Ababol 2 48K, something like this (defined, for example, right at the beginning of drawscr.h):

```
// The tilemap of 15x10 tiles:

unsigned char backdrop [] = {
    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, 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, 0, 0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,19,21, 0,
    0, 0, 0,19,21, 0, 0,19,21, 0, 0,19,20,20,21,
    0, 0,19,20,20,21,19,20,20,21,19,20,20,20,20,
    0,19,20,20,20,20,21,20,20,19,20,20,20,20,20,
   19,20,20,20,20,20,20,20,21,19,20,20,20,20,20,20
};
```

In which case, our code building would be something as simple as:

```
// CUSTOM {
// Construction of assets.
      if (0 == gpd) {
         if (gpy >= 5) {
            gpd = backdrop [gpx + gpy * 15];
         }
      }
// } END OF CUSTOM
```

We can complicate the topic as much as we want, as long as the result ends in gpd (which indicates the tile number that will end up being painted, instead of 0). For example, Leovigildo III uses a code that paints the vertical stripes, adds imperfections randomly, and places shadows for the tiles on the map that are obstacles:

```
// CUSTOM {
// Construction of assets.
    if (0 == gpd) {
      gpd = (attr (gpx - 1, gpy) > 4 || attr (gpx, gpy - 1) > 4 || attr (gpx - 1, gpy - 1) > 4) ? 22 : 18 + (gpx & 1) + (gpjt ? 0 : 22);
    }
// } END OF CUSTOM
```

The possibilities are endless.

Map

The map layer is formed with tiles that are removed from the map you created in Mappy, neither more nor less. If you have added code to replace tile 0 with a background, these tiles will not be drawn.

If we are going to use the new screen layout engine, it is silly to use 48-tile maps, because in these we can do everything directly on the map. When drawing our map in mappy using only the first 16 tiles, the map converter will detect it and create a packed map of 16 tiles.

Decorations

The decorations are extra tiles that are printed on the two previous layers at the end of the process. This can be done by scripting or using the new /dev/engine/extraprints.h module if we are not using scripting in our game (as in the fourth Leovigildo load).

In any case, we will not have to enter the decorations by hand, but the map converter itself will detect them in our .map file and export them. For this, we just have to pass the parameter "forced". If we use this parameter when the converter detects a tile> = 16, instead of exporting a map in unpacked format, it will ignore those tiles in the exported map (which will, therefore, be "packed") and add them to A list which it will subsequently export separately:

```
..\utils\map2bin.exe ..\map\mapa.map 3 3 99 map.bin bolts.bin force
```

Decorations, using scripting

map2bin map.bin.spt generate a file (if you have specified that the output is written to map.bin to invoke) that must copy the folder / script, along with our main script. The map.bin.spt file will include an ENTERING SCREEN xx section for each screen with decorations. Inside, you will have an IF TRUE THEN ... END block with the new DECORATIONS ... END command in which you define a list of tiles that change. In this way, each decoration uses only 2 bytes of the script (compared to the 4 that occupied in the Churrera or in MK2 <0.8, for example), something like this:

```
[...]
ENTERING SCREEN 2
  IF TRUE
  THEN
    DECORATIONS
      7, 1, 16
      8, 1, 18
      7, 3, 22
      8, 3, 23
      7, 4, 25
      8, 4, 24
      1, 7, 27
      1, 8, 26
      7, 8, 19
      8, 8, 20
      9, 8, 20
      10, 8, 20
      11, 8, 21
    END
  END
END
[...]
```

(Each line of Decorations defines X, Y, T for a decoration tile and we can use it anywhere in the script that supports commands – this is great for modifying a torch part of the stage – it's like a sequence of SET_TILE X, Y) = T).

Once this is done, we will only need to add this line at the beginning of the script (in multi-level game scripts, we will have to add it at the beginning of the script section for the corresponding level):

```
INC_DECORATIONS map.bin.spt
```

When msc3 finds this line, it will include the code generated by map2bin and stored in map.bin.spt in the main script at the beginning of each ENTERING SCREEN block.

Decorations without using scripting

In the map.bin.spt file, in the end, be C code with definitions of arrays, something like this:

```
// If you use extraprints.h, trim this bit and use it!
const unsigned char ep_scr_00 [] = { 0x21, 16, 0x31, 17, 0x41, 18, 0xC1, 16, 0xD1, 18, 0x23, 22, 0x33, 23, 0x43, 22, 0xA3, 22, 0xB3, 23, 0x24, 25, 0x34, 24, 0xA4, 25, 0xB4, 24, 0x76, 28, 0x47, 29, 0x38, 19, 0x48, 21, 0xA8, 19, 0xB8, 20, 0xC8, 21 };
const unsigned char ep_scr_01 [] = { 0xC3, 18, 0x74, 16, 0x84, 18, 0x25, 27, 0xC5, 27, 0x26, 28, 0x76, 27, 0xC6, 26, 0x77, 29, 0xC7, 28, 0x88, 19, 0x98, 20, 0xA8, 20, 0xB8, 21 };
const unsigned char ep_scr_02 [] = { 0x71, 16, 0x81, 18, 0x73, 22, 0x83, 23, 0x74, 25, 0x84, 24, 0x17, 27, 0x18, 26, 0x78, 19, 0x88, 20, 0x98, 20, 0xA8, 20, 0xB8, 21 };
const unsigned char ep_scr_03 [] = { 0x42, 23, 0x33, 25, 0x43, 22, 0x53, 24, 0x73, 18, 0x44, 24, 0x84, 18, 0xA4, 16, 0x95, 17, 0x18, 19, 0x28, 21, 0x48, 19, 0x58, 20, 0x68, 21 };
const unsigned char *prints [] = { ep_scr_00, ep_scr_01, ep_scr_02, ep_scr_03, 0, 0, 0, 0, 0 };
```

If we want to use decorations with games without scripting, the first thing we have to do is configure this feature in our /dev/config.h file:

```
#define ENABLE_EXTRA_PRINTS
```

Once this is done, we will have to trim the code included in map.bin.spt and paste it to the beginning of /dev/engine/extraprints.h, right where it puts...

```
// [[[ PASTE HERE THE OUTPUT AT THE END OF THE .SPT FILE THAT MAP2BIN PRODUCES ]]]
```

You can see extraprints.h in action (and other things that will help you create simple video without scripting) in the fourth charge of Leovigildo.

Spectrum games using the MK2 engine
Tutorial 2

Modifying a hitter

This tutorial applies to 0.88c (or later) version of MK2

MK2 implements two hitters (for the moment, exclusive) that can be activated from /dev/config.h:

```
#define PLAYER_CAN_PUNCH
#define PLAYER_HAZ_SWORD
```

Activating one of these Two (only one) will be activating the hitter. By default, the first defines a punch that can be given to either side, and the second a punch that can be left, right or up (always in-side view, and we will curry something genital when necessary).

Of course, the graphics, travel, and behavior of hitters are easily modifiable. We will see here how to modify the sword, the cuffs are simpler and the process is practically the same.

Modifying the trajectory and number of frames

The main behavior hitters and, in this case, the sword is in the /dev/engine/hitter.h file. Almost at the beginning of it you will see something like this:

```
#ifdef PLAYER_HAZ_SWORD
unsigned char swoffs_x [] = {8, 10, 12, 14, 15, 15, 14, 13, 10};
unsigned char swoffs_y [] = {2,  2,  2, 3,  4,  4,  5,  6,  7};
#endif
```

These two arrays define the position of the sword (which is a sprite of 8x8) with respect to the sprite of the protagonist during all the pictures that lasts its animation. Specifically, if we know how to count, we see that there are nine steps of animation.

When the player shoots, the sword is activated. This activation lasts nine frames. Each frame, the sword is placed in the position indicated by the corresponding index of those two arrays, always taking as origin the position of the player.

When the player looks to the right or to the left:

- The "and" position where the sword is drawn will be gpy + swoffs_y [box], where gpy is the player's "y" position and frame goes from 0 to 9.
- If the player looks to the right, the "x" position will be gpx + swoffs_x [box], and if he looks to the left it will be gpx + 8 - swoffs_x [box], where gpx is the "x" position of the player and box goes From 0 to 8.

When the player throws the sword up, swoffs_x and swoffs_y are swapped to allow the movement to be vertical:

- The "y" position where the sword is drawn will be gpy + 6 - swoffs_x [box].
- The "x" position where the sword is drawn will be gpx + swoffs_y [box].

Modifying swoffs_x and swoffs_y we can modify the course of the sword. As it is, the sword makes a kind of curve.

We can also increase or decrease the number of frames in the animation. We will put a faster sword, we will remove all the even pairs and we will only be left with five pictures of animation. We do this, literally: we load each even value of the arrays. We would stay:

```
#ifdef PLAYER_HAZ_SWORD
unsigned char swoffs_x [] = {8, 12, 15, 14, 10};
unsigned char swoffs_y [] = {2,  2,  4,  5,  7};
#endif
```

Since we have changed the number of tables, we have to modify the code a bit. From line 88 is the code that detects that all the pictures of the sword have passed. Now we have 5 frames instead of 9, so we'll have to leave this like this:

```
#ifdef PLAYER_HAZ_SWORD
  if (hitter_frame == 5) {
#endif
```

Altering the "interval hot hitter"

call "hot hitter interval" to pictures of animation hitter in which is considered that the hitter is hitting. For example, in Ninjajar! The fist does not hit while retracting. With the sword, and considering that the paintings originally ranged from 0 to 8, the warm hitter interval was frames 3 to 6, both inclusive:

FRAME:	0	1	2	3	4	5	6	7	8
HIT :	NO	NO	NO	YES	YES	YES	YES	NO	NO

We will have to modify these intervals in two places: when it is detected that we hit a destructible tile (if we are using them), and when it is detected that we hit an enemy.

Let's leave the interval like this, now that we have fewer frames:

FRAME:	0	1	2	3	4
HIT :	NO	YES	YES	YES	NO

In /dev/engine/hitter.h, from line 72, it is where a destructive tile is detected and it is commanded to destroy if we are in the hot hitter range. If you look, the interval is defined if the box> 2 and box <7. We will change it so that tables 1, 2 and 3 are active, that is, if box> 0 and box <4. It looks like this:

```
#if defined (BREAKABLE_WALLS) || defined (BREAKABLE_WALLS_SIMPLE)
  if ((attr (gpxx, gpyy) & 16) && (hitter_frame > 0 && hitter_frame < 4))
    break_wall (gpxx, gpyy);
#endif
```

In /dev/engine/enemmods/hitter.h, on line 7, we have a similar range detection, this time to detect if the sword is "hot" when colliding with an enemy. In the same way, we change it to make it look like this:

```
if (hitter_frame > 0 && hitter_frame < 4) {
```

Changing the "hot spot"

Collision detection hitter with stage or enemies is based on a pixel in the sprite. By default, this pixel is:

- Sword up: (hitter_x + 4, hitter_y)
- Sword left: (hitter_x, hitter_y + 4)
- Sword right: (hitter_x + 7 hitter_y + 4)

To change this We will have to change the assignments to gpxx and gpyy that are in the block of code that governs the positioning of the sword with respect to the player in /dev/engine/hitter.h, that is, here:

```
#ifdef PLAYER_HAZ_SWORD
  if (p_up) {
    hitter_x = gpx + swoffs_y [hitter_frame];
    hitter_y = gpy + 6 - swoffs_x [hitter_frame];
    hitter_next_frame = sprite_sword_u;
#if defined (BREAKABLE_WALLS) || defined (BREAKABLE_WALLS_SIMPLE)
    gpxx = (hitter_x + 4) >> 4; gpyy = (hitter_y) >> 4;       // <<< AQUI (x, y espada hacia arriba)
#endif
  } else {
    hitter_y = gpy + swoffs_y [hitter_frame];
#if defined (BREAKABLE_WALLS) || defined (BREAKABLE_WALLS_SIMPLE)
    gpyy = (hitter_y + 4) >> 4;                          // <<< AQUI, (y espada horizontal)
#endif
    if (p_facing) {
      hitter_x = gpx + swoffs_x [hitter_frame];
      hitter_next_frame = sprite_sword_r;
#if defined (BREAKABLE_WALLS) || defined (BREAKABLE_WALLS_SIMPLE)
      gpxx = (hitter_x + 7) >> 4;                        // <<< AQUI, (x espada hacia la derecha)
#endif
    } else {
      hitter_x = gpx + 8 - swoffs_x [hitter_frame];
      hitter_next_frame = sprite_sword_l;
#if defined (BREAKABLE_WALLS) || defined (BREAKABLE_WALLS_SIMPLE)
      gpxx = (hitter_x) >> 4;                            // <<< AQUI, (x espada hacia la izquierda)
#endif
    }
  }
#if defined (BREAKABLE_WALLS) || defined (BREAKABLE_WALLS_SIMPLE)
  if ((attr (gpxx, gpyy) & 16) && (hitter_frame > 2 && hitter_frame < 7))
    break_wall (gpxx, gpyy);
#endif
#endif
```

The hot spot is located in a fairly convenient position, I doubt you have to change this unless you draw a very bizarre sword.

Changing the Graphic

The sword is defined in /dev/extrasprites.ha from the line 259. _sprite_sword_l labels, _sprite_sword_r and _sprite_sword_u contain the frames for the sword to the left, right and up, respectively, using the 4 blocks Of 8×8 pixels with usual masks.

Spectrum games using the MK2 engine
Tutorial 3

Building a 128K game with several levels (simple)

This tutorial applies to version 0.89 (or later) MK2

The issue of multi-level games are much given to personalization engine. The file included levels128.h and clevels.h handler contains two examples of how:

- The "Goku Evil" mode, in which each level is a compact entity formed map, bolts, behaviors, enemies, hotspots, and tileset Spriteset contained in a single binary.

- The "Ninjajar!" Mode, in which each level consists of a series of "assets" stored in the extra RAM that can be combined in any way, besides including a structure of sequence of levels in case we want to repeat a level several Times (for example, Ninjajar stores!, which are all on the same real level).

In this brief tutorial we will explain how to mount a 128K multilevel game with simple level handler (Goku Mal).

Configuration

In config.h have to establish certain policies to achieve what we want. The first are the basic ones, then I put some more than another that comes well (at least for me, you can use something else).

You have to activate these:

```
#define MODE_128K
#define COMPRESSED_LEVELS
```

And I think its is that if you use scripting (because you use scripting!!) activate this one:

```
#define SCRIPTED_GAME_ENDING
```

Notice we DO NOT activate this:

```
//#define EXTENDED_LEVELS
```

This is what he does is put all the ninjajar traya! And for now we will leave that quiet. Let's create a simple handler game.

Building the binary for each level

to build the binary for each level we will use the buildlevel.exe utility that we, like all others in the / utils. If we execute it to pelaco we obtain the parameters that it takes, that are a good billet:

```
D:\Dropbox\nicanor\desarrollo\dev>..\utils\buildlevel.exe
buildlevel v 0.2 [MK2]
usage:

$ buildlevel mapa.map map_w map_h lock font.png work.png spriteset.png enems.ene scr_ini x_ini y_ini max_objs
enems_life behs.txt level.bin

where:
    • mapa.map is your map from mappy .map
    • map_w, map_h are map dimmensions in screens
    • lock is 15 to autodetect lock, 99 otherwise
    • font.png is a 256x16 file with 64 chars ascii 32-95
    • work.png is a 256x48 file with your 16x16 tiles
    • spriteset.png is a 256x32 file with your spriteset
    • enems.ene enems/hotspots directly from colocador.exe
    • scr_ini, scr_x, scr_y, max_objs, enems_life general level data header
    • behs.txt is a tile behaviours file
    • level.bin is the output filename.
    • decorations.spt if specified, maps are forced to 16 tiles + decorations
```

We are going to use this convention not to mess with filenames (especially if we use many levels), where N is the number of the level:

- MapN.map: The map.
- WorkN.png: The tileset (256x48).
- SpritesN.png: The spriteset (256x32).
- EnemsN.ene: enemies / hotspots.
- BehsN.txt: Behaviors of tiles.
- LevelN.bin: The output file.
- DecorationsN.spt: Decorations to include in the script.

We see several things we need, therefore:

- Map, tileset, spriteset, enemies, as always.
- Behaviors: simply an ordered list of 48 values, like the one we used to put in config.h, something like this:

```
0,24, 8, 8, 8, 8, 8, 8, 1, 1, 8, 8, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 8, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

- font.png: It is a 256x16 file with 64 characters of text font (ascii 32-97). It's a bit of a waste of memory, especially if we're going to always use the same source (which is yours), but for now there's no other way to make a bundle by omitting a certain part of the charset. Nor will it involve much wasted space.

Once we have generated our binary and our decorations file (if applicable), we will have to compress the binary and put it in / bin for the librarian and move the decorations file to / script for later inclusion in the main script.

As the buildlevel commands are very long and you write bribes, and as we will have to generate the levels a lot of times during development, we will create a buildlevels.bat at / levels with calls to this command for each level.

```
@echo off
echo BUILDING LEVELS!

echo LEVEL 0
..\utils\buildlevel.exe ..\map\map0.map 4 4 99 ..\gfx\font.png ..\gfx\work0.png ..\gfx\sprites0.png ..\enems\enems0.ene 0
2 7 99 1 behs0.txt level0.bin decorations0.spt
..\utils\apack.exe level0.bin ..\bin\level0c.bin
move decorations0.spt ..\script

... (etc)
```

As you can see, by compressing it generated directly levelN.bin \ bin with the name of levelNc.bin

Changes to make.bat

First we must remove whatever is build the map, export enemies, graphics, etc. and include a call To our new buildlevels.bat that should look something like this:

```
cd ..\levels
call buildlevels.bat
cd ..\dev
```

We call so that when finished buildlevels.bat follow the execution of make.bat.

Librarian

Supposedly we would have to have the basics for the librarian. Remember that if you have pregenerated binaries for a specific RAM page (such as texts and script) you should place them in a preloadN.bin file where N is the RAM page where they should go so that librarian takes them into account. To the list of binary files /bin/list.txt we add our phases:

```
title.bin
marco.bin
ending.bin
level0c.bin
...
```

According to this configuration, the phase Nth resource will be in the N + 2-th: the first phase in the resource 3, the second at 4...

The script

This tutorial deserve if you own, but I will mention here: your Script file will contain scripts for all levels. These scripts are separated into the file with END_OF_LEVEL:

```
### level 0

ENTERING GAME
   IF TRUE
   THEN
      ...
   END
   ...
END

...

END_OF_LEVEL

### level 1

ENTERING GAME
   IF TRUE
   THEN
      ...
   END
   ...
END

...

END_OF_LEVEL

...
```

Now you have to tell the engine where to find the script for each level. Msc3.exe, in the file msc-config.h that generates, creates a few constants that will suit us very well:

```
#define SCRIPT_0 0x0000
#define SCRIPT_1 0x0179
#define SCRIPT_2 0x01CD
...
```

In addition, this make.bat line:

```
..\utils\sizeof.exe ..\bin\texts.bin 49152 "#define SCRIPT_INIT" >> msc-config.h
```

Add another SCRIPT_INIT constant that tells us where scripts start in memory. So, we will have the first script start at SCRIPT_INIT + SCRIPT_0, and so on. Well, these are the values that we have to put in the array of levels that appears at the end of levels128.h:

```
LEVEL levels [ ] = {
   {3, 3, SCRIPT_INIT + SCRIPT_0},
   {4, 4, SCRIPT_INIT + SCRIPT_1},
  ...
};
```

- The first number is the number of resource librarian that contains the bundle of the level that is. As you can see, we start at 3.
- The second number is the number of music within the OGT to sound of fondow.
- Finally, we have the expression that calculates where the concrete script to start for each phase begins.

With this and a sponge cake, the theme should work half what. And if not, as it is complicated the first time (and the second), there is always the forum to go slowly and step by step.

Spectrum games using the MK2 engine
Tutorial 4

New enemies module

This tutorial applies to version 0.89 (or later) MK2 in previous versions, the arrows are something "broken"

In Leovigildo completely rewrote the module enemies to make it better. Now you can choose which doll and what behavior each enemy has in multiple combinations (almost always), and define whether an enemy fires or not.

To place the "new" enemies you have to use the MK2 Positioner that is in enems. Go ahead with this, the old setter is still there. For the next version I think to smoke all the support to the traditional enemies, so their thing is to get accustomed.

When you give an enemy in MK2 Putter comes out a dialog with three things to fill: SPR, TYPE and SHOOTS.

1. Linear:
SPR: Number of sprite of our spriteset. 0, 1, 2 or 3.
TYPE: 1
SHOOTS: 0 or empty: not fire. 1: yes it shoots.

The bug will go from the source box to the destination box as usual. The speed gets in by selecting the destination box, as always. Must be 1, 2, 4.

2. Fantys:
SPR: Number of sprite. 0, 1, 2 or 3. Yes, you can now choose.
TYPE: 2
SHOOTS: the same.

It does not matter where we put the destination box and what speed we say. It is ignored.

3. Drops:
SPR: It is ignored. Leave it empty.
TYPE: 9
SHOOTS: is ignored.

The drop goes from the source box to the destination box. You have to put it vertically. Speed also affects.

4. Arrows:
SPR: It is ignored. Leave it empty.
TYPE 10
SHOOTS: If you put 1 the arrow will "trap" (only activated if the player touches any of the boxes where the path is defined). If you put 0, it will always leave.

The arrow goes from the source box to the destination box. You have to put it horizontally. It also affects speed.

The MK2 is very green. Edit an enemy already put is a pain, because the parameters SPR, TYPE and SHOOTs appear combined. I'll get better. It also does not take long to remove the bug and create it again. If you know binary and maybe you can understand those numbers yep some code dev / engine / enems.h

Oh, and you can not forget to remove the #define USE_OLD_ENEMS config.h, my god.

Scripting

Documentation "at least something is something" about msc3 and the scripting system Of MK2.

Target: your thing is that you control, at least, the Churrera and its scripting. It comes well to know MK2. At least a little bit.

This document is valid for MK2 v.0.88c and msc 3.91

DISCLAIMER: msc3 supports more checks / commands than do this document (but in motor-of-clauses.txt). They are not guaranteed to work. Some will not directly (they are from branch 4 of the Churrera).

DISCLAIMER 2: With scripting you can do incredible things. Things that you can not imagine We have done things that we never imagined were possible. Thousands of lucky accidents and things like that. Turn around, you'll probably find the solution. And if not, you can ask. But give him rounds before. We trust you. Dale, Fran.

Msc3

Msc3 stands for "Mojon Script Compiler 3".

Msc3 is the script compiler. It started as a simple solution for compile clause-based scripts in an easily interpretable bytecode by the engine, but it has become an incredible multi-monster tentacles and shifting shapes, like a little Nyarlathotep.

Msc3 generates a binary script.bin file with the bytecode of the script and a pair of .h files that contain the interpreter and its configuration (msc.h and msc-config.h).

Msc3 needs little to work: a script, the number of screens in total that has the global map of our game (*), and if you must prepare the interpreter to run from the extra RAM of 128K models.

Msc3 is run from the command line and its syntax is:

```
> Msc3.exe file.spt N [rampage]
```

Where file.spt is the file with the script to be compiled, N is the number of screens of our game (*), and [rampage] is an optional parameter which, if indicated, causes msc3 to generate an interpreter prepared for inter-prepend script.bin from an extra RAM page of the 128K models.

(*) In multi-level games where each level has a number of screens, refers to the maximum size on screens for which memory is reserved in RAM low, that is, the result of multiplying MAP_W by MAP_H.

Engine clauses

MK2 scripts are organized into sections. Each section will be executed in a precise moment and in a precise screen.

Mainly we have sections type ENTERING, that will be executed when entering on a screen, PRESS_FIRE sections, which will be executed when the key is pressed of action, and special sections that will respond to various events. These are the types of sections:

ENTERING SCREEN x

It is executed every time the player enters the screen x

ENTERING GAME

It runs when starting each game, and only this time.

ENTERING ANY

It is executed when entering each screen, just before ENTERING SCREEN x

PRESS_FIRE AT SCREEN x

It runs when the player presses the action key on the x screen.

PRESS_FIRE AT ANY

It is executed when the player presses the action key on any screen, just before PRESS_FIRE AT SCREEN x

ON_TIMER_OFF

It is executed when the timer reaches zero, as long as we have defined the TIMER_SCRIPT_0 directive in config.h

PLAYER_GETS_COIN

It runs when the player touches a TILE_GET tile. You need to have it activated and configured the TILE_GET functionality in config.h as well as the directive TILE_GET_SCRIPT.

PLAYER_KILLS_ENEMY

It is executed when the player kills an enemy, as long as we have defined the RUN_SCRIPT_ON_KILL directive in config.h

- The "PRESS_FIRE" scripts will run in more assumptions in addition to when the player press action:
- If we have #define ENABLE_FIRE_ZONE in config.h, we have defined a zone action with "SET_FIRE_ZONE" in our script, and the player enters said zone.
- If we push a block and have defined in config.h the directives ENABLE_PUSHED_SCRIPTING and PUSHING_ACTION.
- Each time a floating object is affected by gravity and falls, if we have defined ENABLE_FO_SCRIPTING and FO_GRAVITY.

In previous versions of MK2 and the Churrera also were thrown when killing an enemy, but has been replaced by the section PLAYER_KILLS_ENEMY.

Various levels

We can define script for several levels. The best current example to see it in action is Ninjajar !. In our script file we concatenamos the scripts of our levels one behind another separated by a line

```
END_OF_LEVEL
```

When compiling, msc3 will generate a constant to identify where each script for each level that we can use in our level_manager or as we'd better. The constant will be of type SCRIPT_X with X the number of order of the script within the .spt file.

However, from MK2 0.90 we can have more control over this. Yes, we included at the beginning of a section of our script (actually it can appear anywhere before END_OF_LEVEL, but let's be sorted) a line.

```
LEVELID xxxxx
```

With "xxxxx" an alphanumeric literal, this literal will be used as a name of the generated constant. Instead of SCRIPT_X will be what we put, which will give us perhaps a little more control and legibility when it comes to mount our level handler.

Clauses

All sections described above will contain a list of clauses. Each clause consists of a checklist and a list of commands.

The interpreter will scroll through the checklist in order, making each testing. If any failure, it will stop processing the clause.

If all the checks have proved to be true, the list will be executed command in order.

The syntax is:

```
IF CHECK
...
THEN
COMMAND
...
END
```

All clauses in a section are executed in order, non-stop (less that you indicate it with a BREAK command or after some commands like WARP_TO, WARP_TO_LEVEL, REHASH, REPOSTN, or REENTER).

Many times you can save script and avoid using BREAK. Most of the times the execution time of a script is not critical and you can allow it.

Instead of:

```
IF FLAG 1 = 0
THEN
SET TILE (2, 4) = 2
        SET FLAG 1 = 1

        # If we do not put this break will run the following
        # Clause yes or yes, since FLAG 1 = 1.
        BREAK
END

IF FLAG 1 = 1
THEN
SET TILE (2, 4) = 3
END
```

You can do:

```
IF FLAG 1 = 1
THEN
SET TILE (2, 4) = 3
END

IF FLAG 1 = 0
THEN
SET TILE (2, 4) = 2
SET FLAG 1 = 1
END
```

And you save yourself a BREAK.

Flags

The scripting engine handles a set of flags or flags that can contain a value from 0 to 127 and are used as variables. The flags are often referenced as FLAG N with N from 0 to 127 in the script.

In almost all checks and commands that support immediate values you can use construction #N where N is a flag number, which means "the value of flag N".

For example:

```
IF FLAG 5 = # 3
```

It will be true if the value of flag 5 is equal to the value of flag 3, or:

```
WARP_TO # 3, # 4, # 5
```

It will jump to the screen in FLAG 3, in the position indicated by the flags 4 and 5. On the contrary:

```
WARP_TO 3, 4, 5
```

You will jump to screen 3, in position (4, 5).

Unchanged, MK2 allows 32 flags, but this number can be changed easily by editing the corresponding #define in definitions.h:

```
#define MAX_FLAGS 32
```

Alias

In order not to have to remember so many numbers of flags, alias. Each alias represents a flag number, and begins with the character "$". In this way, we can associate for example the alias $ KEY to flag 2 And use in the script this alias instead of 2:

```
IF FLAG $ KEY = 1
```

Aliases also work with construct #N, so if we write # $ NENEMS we will refer to the content of the flag whose alias is NENEMS:

```
IF FLAG $ TOTAL = # $ NENEMS
```

It is true if the value of the flag whose alias is $ TOTAL matches the value of the flag whose alias is $ NENEMS.

Aliases must be defined at the beginning of the script (they can actually defined in any part of the script, but will only be valid in the part of the script that comes later) in a DEFALIAS section that must have the following format:

```
DEFALIAS
$ ALIAS = N
...
END
```

Where $ ALIAS is an alias and N is a flag number. We will define an alias for each line. It is not necessary to define an alias for each flag.

As of version 3.92 of msc3, you can skip the word FLAG if you use alias. That is, the compiler will accept

```
SET $ KEY = 1
```

and also

```
IF $ KEY = 1
```

As of version 0.90 of MK2 (and its corresponding msc3.exe), we can define simple macros within DEFALIAS. Macros start with the character %. Everything that comes after the macro name is considered the replacement text. For example, if you define:

```
% CLS EXTERN 0
```

Each time you enter% CLS in your msc3 code it will be replaced by "EXTERN 0".

Decorations

Each script (and by that I mean each level script included in our script file) may include a set of decorations such as generates map2bin.exe. This must include the directive INC_DECORATIONS At the beginning of the script:

INC_DECORATIONS file.spt

Where .spt file is the .spt file that map2bin.exe generates

The dynamic interpreter

Msc3 generates an interpreter that will only be able to understand the checks and commands that you have entered in your script. This is done to save memory not generating code that will never be executed.

Sometimes there are several ways to get one thing in your script. If you have you choose, do not choose the one that produces a simpler script, but the one that makes you have to use less variety of checks or commands, since a little more script takes up much less than the necessary C code to execute a check or command.

Always true

There is a test that is always true and used to run commands in any case:

```
IF TRUE
```

Checks and commands related to flags

Much of your script will be checking flag values and modifying these values. For this there is a whole set of checks and commands.

Checks with flags

```
IF FLAG x = n Evaluate to TRUE if flag "x" is "n"
IF FLAG x <n Evaluate to TRUE if flag "x" <n
IF FLAG x> n Evaluate to TRUE if flag "x"> n
IF FLAG x <> n Evaluate to TRUE if flag "x" <> n
```

These four are legacy. They are here because it was the only way compare two flags in the early versions of the scripting system, but they can be avoided, in fact MUST be avoided, as this will save interpreter code.

IF FLAG x = FLAG and will evaluate to TRUE if the flag "x" = flag "y"
Equivalent to IF FLAG x = #y

IF FLAG x <FLAG and Evaluate to TRUE if the flag "x" <flag "and"
Equals IF FLAG x <#y

IF FLAG x> FLAG and evaluate to TRUE if the flag "x"> flag "y"
Equivalent to IF FLAG x> #y

IF FLAG x <> FLAG and evaluate to TRUE if the flag "x" <> flag "y"
Equivalent to IF FLAG x <> #y

One of the things in my ALL is to modify MSC to translate automatically constructs the "IF FLAG x <OP> FLAG and" a "IF FLAG X <OP> #y".

Commands with flags

SET FLAG x = n Give the value N to the X flag. Needless to say, SET FLAG x = #y will give the value of flag and flag x. But I have already said it.

INC FLAG x, n Increases the value of flag X in N.

DEC FLAG x, n Decrements the value of flag X in N

FLIPFLOP x If x is 0, it will be 1. If it is 1, 0 is worth. What comes to be a flip-flop, go.

SWAP x, y Swaps the value of flags x and y

These two are legacy as they can be simulated with INC / DEC FLAG. The same of before, they remain here from the first versions when it did not exist the #N construction.

ADD FLAGS x, y Make x = x + y.
Equivalent to INC FLAG x, #y

SUB FLAGS x, y Make x = x - y.
Equivalent to DEC FLAG x, #y

Engine items that modify flags

All this mess of flags would be much less useful if the engine could not modify them also giving status information. Certain config.h will cause certain flags to be updated with engine values:

Number of objects collected

Directive OBJECT_COUNT

With "objects" I mean the automatic collectibles that are they place using Type 1 hotspots in the Colocador.

If this policy is defined, the specified value will indicate which flag should update the engine with the number of objects the player has collected. For example

```
#define OBJECT_COUNT 1
```

It will make FLAG 1 contain at all times the number of objects it carries the player collected. In this way,

```
IF FLAG 1 = 5
```

It will be fulfilled when the player has collected 5 objects.

Number of enemies on the screen

COUNT_SCR_ENEMS_ON_FLAG directive

If this directive is defined, the engine will count how many active enemies there are on the screen at the time of entering it. For example

```
#define COUNT_SCR_ENEMS_ON_FLAG 1
```

Will make FLAG 1 contain the number of enemies on the screen to enter.

Number of enemies eliminated

BODY_COUNT_ON directive

If this directive is defined, the engine will always increase the specified flag that the player eliminates an enemy. For example,

```
#define BODY_COUNT_ON 2
```

It will cause FLAG 2 to increase each time the player kills an enemy. This can be very useful to use in conjunction with COUNT_SCR_ENEMS_ON_FLAG.

Following our example (enemy counts on screen in flag 1, and increase the flag 2 by killing an enemy), if entering a new screen we set the flag 2 to 0 (reset the count of killed enemies:

```
ENTERING ANY
IF TRUE
THEN
SET FLAG 2 = 0
END
END
```

We can control that we have killed all the enemies of the screen very easily, in the PLAYER_KILLS_ENEMY section, which is executed whenever the player kills an enemy:

```
PLAYER_KILLS_ENEMY
IF FLAG 2 = # 1
THEN
# We killed all the enemies on the screen!
END
END
```

Count TILE_GET

TILE_GET_FLAG directive

The engine allows you to define a map tile as "pickable". For example, coins. You can place coins on the map, or make them appear on the break a stone (see Ninjajar!). If you have soaked whatsnew.txt you will know that TILE_GET directive n causes tile "n" to be "collectible". This is used in conjunction with the TILE_GET_FLAG directive, so that when the player touch a tile TILE_GET number to increment that flag.

You can not use TILE_GET without TILE_GET_FLAG, of course.

For example, you can make your tile 10 a diamond and place these diamonds all over the map. Then you define TILE_GET to 10 and TILE_GET_FLAG to 1. Each time the player touches a diamond, it will disappear and FLAG 1 it will increase.

In Ninjajar it is used in a more complex way, in conjunction with the directive BREAKABLE_TILE_GET n. In ninjajar is defined as:

```
#define BREAKABLE_TILE_GET 12
#define TILE_GET 22
#define TILE_GET_FLAG 1
```

And it produces this behavior: If the player breaks tile 12, which is the box with stars (that in its "behaviors" has activated the flag "breakable"), the TILE_GET tile, which is 22, will appear. If the player touches a Tile 22 (one coin), it will disappear and FLAG 1, which is the money account we have collected.

Note that Ninjajar uses 16 tile maps - coins will only appear in the break star blocks.

All this works automatically, we just have to worry to examine the value of the FLAG defined in TILE_GET_FLAG from our script.

Blocks that push themselves

Policies MOVED_TILE_FLAG, MOVED_X_FLAG, and MOVED_Y_FLAG

If we have enabled the player to push boxes (tile 14 of the tileset, with "behavior" = 10) using #define PLAYER_PUSH_BOXES, and we also activate the ENABLE_PUSHED_SCRIPTING directive, each time the player moves a box (Or block, whatever we put in tile 14 ;-)), this will happen:

- The tile number that is "Treaded" will be copied into the MOVED_TILE_FLAG flag.
- The X coordinate to which the box moves is copied to the MOVED_X_FLAG flag.
- The Y coordinate to which the box is moved is copied to the MOVED_Y_FLAG flag.

If we also define PUSHING_ACTION, the sections PRESS_FIRE (ANY and corresponding to the current screen) and we can make the checks immediately.

This system is used in Cadaverion (although it is a game of the Churrera, this works exactly the same) to verify that we have placed the statues on their pedestals.

For example, if we want to open a door if a box is pushed to the position (7, 7) of the screen 4, we can define:

```
#define PLAYER_PUSH_BOXES
#define ENABLE_PUSHED_SCRIPTING
#define PUSHING_ACTION
#define MOVED_TILE_FLAG 1 // This is not going to be used in this example
# Define MOVED_X_FLAG 2
#define MOVED_Y_FLAG 3
```

And in our script...

```
PRESS_FIRE AT SCREEN 4
IF FLAG 2 = 7
IF FLAG 3 = 7
THEN
# The box we moved is in position 7,7
# open door...
END
END
```

If, for example, we want a door to be opened whenever a door is placed box on a tile "pushbutton", which we have drawn on the tile, say 3, (With the same defines) we would have:

```
PRESS_FIRE AT SCREEN 4
        IF FLAG 1 = 3
        THEN
                # The box we moved is on a tile 3
                # open door...
        END
END
```

Whether or not you can shoot

PLAYER_CAN_FIRE_FLAG directive

If we have our engine configured for a set of shots using the PLAYER_CAN_FIRE directive and its companions, we can define that the player can only fire if a given flag is worth 1 giving the number of that flag, Flag to the PLAYER_CAN_FIRE_FLAG directive. For example:

```
#define PLAYER_CAN_FIRE_FLAG 4
```

At the beginning of the game we can do:

```
ENTERING GAME
IF TRUE
THEN
SET FLAG 4 = 0
END
END
```

So the player can not shoot. Then, later, we can do that the player finds a gun and picks it up:

```
... (anywhere)
IF... (conditions of taking the gun)
THEN
SET FLAG 4 = 1
# Now the wrist can shoot
END
END
```

TODO: do the same for the hitter (fist / sword). Soon.

Struck by a floating floating object

CARRIABLE_BOXES_COUNT_KILLS directive

In fact, dying crushed by a box counts as an increase in the flag defined in BODY_COUNT_ON, but for some mysterious reason I do not remember (Probably, mental disorganization) I also put it separate.

If this directive is activated, the indicated flag number will count only the enemies who die struck by a dropable floating object.

(If you set BODY_COUNT_ON will also increase the flag set there ... no I know, maybe it works for you).

Floating objects that fall affected by gravity

Policies ENABLE_FO_SCRIPTING, FO_X_FLAG, FO_Y_FLAG, and FO_T_FLAG

If your game uses "floating objects" (Leovigildo 1, 2 and 3), if you activate the gravity with FO_GRAVITY objects will fall if they have no obstacle below.

In Leovigildo 3 we invented a puzzle in which we had to drop a Sartar buckle on Amador the Tamer to squeeze it. I needed this extend the engine to see where a floating object fell.

Enabling ENABLE_FO_SCRIPTING will execute the PRESS_FIRE sections the current screen whenever a floating object drops and drop a box

Just before calling the script, the position to which it has fallen will be copied in the flags indicated in FO_X_FLAG and FO_Y_FLAG, and the type of "floating object "will be copied to FO_T_FLAG.

It is interesting to know that the FO_T_FLAG flag will be set to 0 when entering a screen.

```
#define ENABLE_FO_SCRIPTING
#define FO_X_FLAG 1
#define FO_Y_FLAG 2
#define FO_T_FLAG 3
```

We already use alias here...

```
DEFALIAS
[...]
$ FO_X 1
$ FO_Y 2
$ FO_T 3
[...]
END
```

And in the PRESS_FIRE of the screen of Amador we find...

```
PRESS_FIRE AT SCREEN 19
# Throw FO over Amateur
# Amateur is in X, Y = (8, 7).
# The FO corkboard is tile 17
IF FLAG $ FO_T = 17
IF FLAG $ FO_X = 8
IF FLAG $ FO_Y = 7
THEN
SOUND 0
EXTERN 31
SET FLAG $ AMADOR = 5
REENTER
END
…
```

Position-related checks and commands

We also have several ways to check and change the position - even changing screen and level!

Position checks

IF PLAYER_TOUCHES x, and will evaluate to TRUE if the player is playing the tile (x, y). X and y can carry #.

IF PLAYER_IN_X x1, x2 Evaluate to TRUE if the player is horizontally
Between the co-ordinates in pixels x1 and x2.

IF PLAYER_IN_Y y1, y2 Evaluate to TRUE if the player is vertically
Between the coordinates in pixles y1 and y2.

Changing position

These commands are used to modify the position of the character. All are expressed at tiles (x from 0 to 14, and from 0 to 9).

SETX x Places the character in the tile x coordinate
(Modifies only the horizontal position)

SETY y will place the character in the tile coordinate and
(Modifies only the vertical position)

SETXY x, y Places the character in the tile coordinate (x, y)
(Modifies both coordinates, x and y).

Screen Checks

Although you can define scripts in ENTERING n and PRESS_FIRE AT n where n is the current screen and that only run when we are on that screen, there are times when it is necessary to know in which screen we are in one of the sections "general" (when the TIMER is finished, when we kill an enemy ...) For these cases we have:

IF NPANT n Evaluates to TRUE if the player is on screen n

IF NPANT_NOT n Evaluate to TRUE if player is NOT on screen n

Changing the screen

WARP_TO n, x, y Moves the player to position (x, y) of screen n. X and y at tiles level

Changing Level

Obviously, only if your game has several levels.

```
WARP_TO_LEVEL l, n, x, y, s
```

It does the following:

- Ends the current level.
- Load and initialize level l.
- Sets the active screen to n.
- Put the player in the coordinates (x, y) (at tiles level).
- If s = 1, it does not display a new level screen (*)

(*) This is very relative and very custom. In Ninjajar! There is a screen before start each level, which is only displayed if the variable silent_level is 0. The value of s is copied to silent_level, so...

If your level handler is different you can skip it, or use it for another thing.

Yes, make games of many levels and such is complicated. Life is like that.

Redraw the screen

It is useful if you do something that loads the screen, such as taking a picture Text with an EXTERN (see below) (all we have done for Ninjajar in forward use EXTERN mainly to extract text boxes that are loaded the play area). That's how you re-paint everything. Just run:

REDRAW

Eyelet: There is a screen size buffer where every thing that prints (Either by the routine that is executed when entering a new screen and that compose the scenario, either by a SET TILE (X, Y) = T of the scripting, etc.) is copied there. REDRAW simply dumps that buffer to the screen. If you have modified the screen with things from the script, REDRAW is not going to come back to its original state!

"Reenter" on the screen

Sometimes you need to re-run the entire ENTERING ANY script and / or ENTERING SCREEN n, or you need to reinitialize the enemies (if you change the type on the fly, see later), or god knows what.

There are several ways to re-enter the screen:

REENTER Re-enters the screen, exactly the same that if we came from another. It does everything: redraw, initialize everything, run the scripts...

REHASH The same, but no redraw. Nor does it show the screen "LEVEL XX" if your motor is configured for that. But if you initialize everything and execute the scripts.

Modify the screen

There are several ways to change the display:

Change tiles from the play area

Changing tiles from the playing area effectively modifies the playing area. I mean that the changes are persistent (they survive a REDRAW) and in addition the modified tiles are interchangeable. That is, if you change the screen by removing a wall with a transparent tile, the player may pass by there.

SET TILE (x, y) = t Put tile t in the coordinate (x, y).
The coordinates (x, y) are at the tiles level.

Of course, and this is very useful, both x and y as t can lead # To indicate the contents of a flag. To print on 4, 5 tile that says flag 2, we do

```
Set TILE (4, 5) = # 2.
```

To print a tile 7 on the coordinates stored in flags 2 (x) and 3 (y), we do:

```
SET TILE (# 2, # 3) = 7
```

And, remember, whenever we reference a flag we can use an alias. I'm not remembering all the time because I trust your intelligence, but

```
DEFALIAS
$ COORD_X 2
$ COORD_Y 3
END

[...]
```

```
SET TILE (# $ COORD_X, # $ COORD_Y) = 7
```

It's the same.

You can also use lists of decorations. I mentioned them above when I talked about INC_DECORATIONS. The lists of decorations you can put them in any section of commands, are as follows:

```
DECORATIONS
X, y, t
...
END
```

Where (x, y) is a tile-level coordinate, and t is a tile number.
For example:

```
DECORATIONS
12, 3, 18
7, 4, 16
8, 4, 18
2, 5, 27
12, 5, 27
2, 6, 28
7, 6, 27
12, 6, 26
7, 7, 29
12, 7, 28
8, 8, 19
9.8, 20
10, 8, 20
11, 8, 21
END
```

Change behavior only

Very, very silly. If you need it, it works just like SET TILE but only for the original behavior for which you specify:

```
SET BEH (x, y) = b
```

Print tiles anywhere

We can print a tile anywhere on the screen, either in the game or outside (for example, in a marker area). For this we use:

```
PRINT_TILE_AT (x, y) = n
```

Prints the time n e (x, y), with (x, y) eye! In CHARACTER coordinates (x = 0-30, y = 0-22).

This function prints only. Although the tile we paint is over the area of game will not affect it at all for nothing.

A very cool thing for what this can serve is to make passages secrets: on your map you make a corridor, but then on ENTERING SCREEN it covers of tiles with PRINT_TILE_AT... Since these tiles do not affect the area of game, it will seem that you can not pass by... but you can!

Show changes

All tile impressions in the engine are made to a buffer. In each frame of play, this buffer is drawn on the screen following a fun and magical process. However, during script execution, the buffer is not dumped to the screen.

If we change something and we want it to be seen immediately without having to wait to return to the game (for example, if we are doing an animation), we need tell the interpreter explicitly to paint the buffer on the screen. This is done with the command:

SHOW

Push blocks

We have already spoken, but we had not mentioned that:

IF JUST_PUSHED It will be true if we have reached FIRE_SCRIPT by have pushed a box.

This is VERY USEFUL. You have to keep in mind that the position of the object pushed (stored in flags defined in MOVED_X_FLAG and MOVED_Y_FLAG) and the tile you have overwritten (stored in the flag defined in MOVED_TILE_FLAG) are persistent - these flags will retain their value until you move another block.

In the PRESS_FIRE section you can enter in several ways, for example by pressing action. Maybe it is not convenient for you to carry out tests that involve the flags affected by the pushed blocks if we have not entered after pushing a block.

Cadaverion makes use of this.

In Cadaverion we must push a number of statues over a certain number of pedestals The configuration in config.h related to this is:

```
#define ENABLE_PUSHED_SCRIPTING
#define MOVED_TILE_FLAG 1
#define MOVED_X_FLAG 2
#define MOVED_Y_FLAG 3
#define PUSHING_ACTION
```

In the ENTERING of each screen, the flag number 9 is set to the number of statues / pedestals out there. That is, flag # 9 will contain the number of statues which we have to place on their pedestals.

On each screen there is a gate that must be opened by placing the statues on their pedestals. In the ENTERING we define its position in flags # 6 and # 7 (Resp., X and Y coordinates).

In flag # 10 we will go counting the statues that we placed in its place. When all the statues are in place, (ie, flag 9 and flag 10 contain the same value) we will open the gate that allows you to go to the next screen.

Let's use flag # 11 as a flag. If it is 0, the gate is closed. Yes worth 1, we have already opened it. Then, this code is the one that opens the gate:

```
IF FLAG 9 = # 10
IF FLAG 11 = 0
THEN
# We say that the gate is open:
SET FLAG 11 = 1

# Delete the gate by putting a tile 0 in its coordinates:
SET TILE (# 6, # 7) = 0

# Show the changes immediately.
SHOW

# Ruiditos
SOUND 0
SOUND 7

# More things that do not interest us.
[...]
END
```

We could have added JUST_PUSHED but it does not matter. It will not be fulfilled until we have not placed the last statue, so it does not matter that it is checked when it should not.

The statues correspond to tile 14, and have "behavior 10", that is, that the statues are the "pushable" block.

The pedestals are tile 13.

In the PRESS_FIRE AT ANY, which will be launched (together with the PRESS_FIRE AT SCREEN n of the current screen) every time we move a block, we will see that we have stepped on a pedestal. In other words, we will verify that:

```
IF JUST_PUSHED
IF FLAG 1 = 13
THEN
INC FLAG 10, 1
```

That is: we came here for pushing a statue (JUST_PUSHED) and the tile we have "stepped on" is 13 (a pedestal). Then, we add 1 to the account of statues placed.

What if we did not put JUST_PUSHED? Well imagine: you take, you move one statue to the pedestal. In FLAG 1, the tile you just stepped on is copied, which is 13. But that value stays there ... If you press ACTION, for example, the value it will remain there, and count as if we have stepped on another pedestal. No spring.

The complete code of this clause is this, because it is done very interesting:

```
IF JUST_PUSHED
IF FLAG 1 = 13
THEN
# We increased the number of statues
INC FLAG 10, 1

# Sound
SOUND 0

# We changed the statue to another tile!
SET TILE (# 2, # 3) = 22

SHOW
SOUND 0
END
```

If we did not change the statue for another tile, you could keep pushing it. As pushing a normal block is destructive (always erases with tile 0), this it would remain of the host of ugly.

When you change the tile in (# 2, # 3), which is the position of our statue according to config.h, other than 14 (22 is a rotated statue, was left Chuli) can not be pushed further.

Clever, huh?

The timer

The timer is a very cool thing in the MK2 engine (it was also in the Churrera, but now it's better). Basically it is a value that decreases every certain number of game frames. When it reaches 0, it can be done the player loses a life, or that there is a game over ... Or a game can be played special section of the script.

(There is a lot about the timer, it can run autonomously and do many things, but that does not concern us. Look at whatsnew.txt or question).

If we activate in config.h

```
#define TIMER_SCRIPT_0
```

Each time the timer reaches 0, the ON_TIMER_OFF section is triggered. There we can do things

In addition we have certain checks and some commands related to the Timer:

Timer checks

IF TIMER> = x True if the timer has a value> = x.

IF TIMER <= x True if the timer ... oh, come on!

Timer Commands

Actually, "command", because there is only this:

SET_TIMER v, r Sets the timer to a value v with "rate" r. It means that each r will be decremented game. Under normal conditions, the games go 22 and 33 fps per second ... 25 to 30 are good values if you want your timer to look like it counts in seconds. Try it.

EYE: SET_TIMER, with "_". Yes, I know there is a SET FLAG without "_". As ín is, Accept it Get over it. You'll be wrong a thousand times. I'm mistaken a thousand times. But I'm fucked. Seriously, I'll change it.

TODO: Change this.

TIMER_START Turn on the timer.

TIMER_STOP Turns off the timer.

Many things can be done. For example, we are worth what is done in Cadaverion ...

This is what happens when the time runs out.
In # 12 we save the screen to which we must return at the end of the time.
In # 13, # 14 the coordinates where we will appear when this occurs.

```
ON_TIMER_OFF
IF TRUE
THEN
SOUND 0
SOUND 0
SOUND 0
SOUND 0
SET_TIMER 60, 40
DEC LIFE 1
SET FLAG 8 = # 0
WARP_TO # 12, # 13, # 14
END
END
```

One thing that is usually done, as we have just seen, is to restart the timer. In this case a life is subtracted, the timer is reset, and changed to site character.

The inventory

This started out as a joke, it got worse in Leovigildo, and now it's a great cucumber which allows us to do many things with very little code. Ains, would have come well in Ninjajar! - would have saved a lot of script code and headaches.

Inventory is nothing more than an object container. We can define how many objects at most we will carry.

The inventory therefore has an N number of slots. Each slot can be empty or contain an object. Objects are referenced in the simplest way possible: by its tile number. "0" can not be used because it indicates "empty".

There is, in addition, and at all times, a "selected slot".

The value of the selected slot and, for convenience, the value of the object that is in that slot, are kept in two special flags that we can choose.

The position and configuration of the inventory is also quite configurable and can be quite adapted.

Let's talk first about the inventory "by hair" and then we talk about the "Floating objects" of type "container", which were created to make make games with objects and inventory is a fucking ride.

Defining our inventory.

The inventory is defined in a special section at the beginning of the script. At the different parameters necessary to start up the system are defined and has this form:

```
ITEMSET
SIZE n
LOCATION x, y
DISPOSITION disp, sep
SELECTOR col, c1, c2
EMPTY tile_empty
SLOT_FLAG slot_flag
ITEM_FLAG item_flag
END
```

We are describing each line one by one (yes, all are necessary).

SIZE n Indicates the size of our inventory, that is, the number of slots that will compose it.

LOCATION x, y Indicates the position (x, y) at the character level from the top corner of the inventory, which corresponds to where the first item will appear.

DISPOSITION disp, sep "disp" must be HORZ or VERT, to indicate if we want the slots in our inventory to be show next to each other (HORZ) or some below others (VERT). "Sep" defines the separation:

* If the inventory is HORIZONTAL (HORZ)
    - The first slot will be placed in (x, y)
    - The next, in (x + sep, y)
    - The next, in (x + sep + sep, y)

...

* If the inventory is VERTICAL (VERT)
    - The first slot will be placed in (x, y)
    - The next, in (x, y + sep)
    - The next, in (x, y + sep + sep)

...

Each slot occupies 2×2 characters (which occupies a tile) but must be left a space of 2×1 characters just below to paint the "selector" that indicates which slot is active.

SELECTOR col, c1, c2 Defines the selector. The selector is only one arrow or something that is placed just below the slot.

The selector will be painted col color and will consist of the characters c1 and c2 of the charset, In horizontal I usually use two characters of letters, to do not waste tiles. Look font.png at any Of the "Leovigildo" and you will see the arrow characters 62 and 63.

EMPTY tile_empty Contains the tile number of our tileset that will be used to represent an "empty slot". In Leovigildo we put a very blue square Cool, you can see it in the tileset in the position 31.

SLOT_FLAG slot_flag Says which flag contains the currently selected slot

ITEM_FLAG item_flag Says which flag contains the contents of the slot selected.

In order not to interfere, I usually define flags 30 and 31 (the last two if we use the 32 that come by default) for these two values. A) Yes, at any time, flag 30 (SLOT_FLAG 30) contains which slot is selected (a number from 0 to n - 1, if the inventory has 4 slots, may be 0, 1, 2 or 3 depending on which one is selected), and Flag 31 (ITEM_FLAG 31) contains the object in that slot.

Examples: Leovigildo 1, for example:

```
ITEMSET
SIZE 4
LOCATION 18, 21
DISPOSITION HORZ, 3
SELECTOR 66, 62, 63
EMPTY 31
SLOT_FLAG 30
ITEM_FLAG 31
END
```

We have an inventory that may contain four objects. We have left site in the marker for it, from the coordinates (18, 21). If will paint horizontally, with a new slot every 3 characters. The selector will be bright red (2 + 64 = 66) and will be painted with characters 62 and 63.
The tile that will represent an empty slot is the 31, which in the tileset appears like a blue box. The flag 30 will contain the selected slot, and the flag 31 will contain what object is in that slot (0 if there are none).

In Leovigildo 3 we also have an inventory:

```
ITEMSET
SIZE 3
LOCATION 21, 21
DISPOSITION HORZ, 3
SELECTOR 66, 62, 63
EMPTY 31
SLOT_FLAG 30
ITEM_FLAG 31
END
```

In this case the inventory is smaller, only 3 slots. The from the coordinates (21, 21), will also be horizontal and the slots will be drawn every 3 characters. The selector will be the same, and the configuration of empty tile and flags is the same.

Horrible examples, they are very similar.

Direct access to each slot

We can directly access each slot, remember, numbered 0 to n - 1,
Using "ITEM", in both conditions and commands:

IF ITEM n = t Evaluates TRUE if slot N is the item T

IF ITEM n <> t Evaluates TRUE if slot N is not the T
(I doubt this will serve anything ... but good)

SET ITEM n = t Puts the object represented by tile T in the
Slot N.

SET ITEM will serve to initialize the inventory, for example, or to make appear objects in it by art potagio. At the beginning of each game you can do:

```
SET ITEM 0 = 0
SET ITEM 1 = 0
SET ITEM 2 = 0
SET ITEM 3 = 0
```

(For an inventory of 4 slots). I use it also when I'm doing Debug, to put in the inventory objects that I need to test some thing.

As with impressions, the changes will not be visible until the next game box occurs. If for some reason you need to modify the inventory and it shows in the middle of a script, you can use

REDRAW_ITEMS

To refresh the on-screen inventory.

Selected Slot and Selected Object in Script

The above is enough for the inventory to appear when we run the game, but will have to make it work. For this we will use the defined flags for this purpose in SLOT_FLAG and ITEM_FLAG.

We have talked about aliases. We could define these aliases (counting following the example and we have defined SLOT_FLAG in 30 and ITEM_FLAG in 31):

```
ALIAS
$ SLOT_FLAG 30
$ ITEM_FLAG 31
END
```

We could even use flags 30 and 31 to hair ...

You can do everything with this, but there are certain conditions and certain commands which will make your script more readable ... and that will be translated in a way transparent to handle the flags defined in SLOT_FLAG and ITEM_FLAG – for which will not cost you "interpreter code".

IF SEL_ITEM = T Evaluates TRUE if the item in the slot selected is that represented by tile T. (Internally equals IF FLAG $ ITEM_FLAG = T)

IF SEL_ITEM <> T Evaluates TRUE if the item in the slot selected is NOT represented by tile T. (Internally equals IF FLAG $ ITEM_FLAG <> T)

In addition, the automatic alias is defined:

SLOT_SELECTED Equals the value of the selected slot.

Therefore, we can establish what ITEM we want in the current slot by doing

```
SET ITEM SLOT_SELECTED = T
```

This is used as well. It is assumed that when we press "action", it is to use the item that we have selected at a specific site on the screen.

Let's take a practical example to see the basic operation of the inventory. All this is greatly simplified with the use of "floating objects" of container type, but we will see it first in command plan.

Imagine that we arrived at a room where there is an object, a paper. This is tile 22 of our tileset. We have placed it in the position (5, 4) when entering the room in concrete, which is 6. We have a special flag, $ PAPER, that will be worth 0 if we have not yet caught it.

```
ENTERING SCREEN 5
IF FLAG $ PAPER = 0
THEN
SET TILE (5, 4) = 22
END
END
```

Now let's let the player take the paper. For this, the player (By touching the position (5, 4), that is) and pressing the action key. This will launch the corresponding PRESS_FIRE script.

In it we will detect that we are in the right place, we will eliminate the role of the screen, we will put it as ITEM in the inventory, and we will set the $ PAPER flag to 1 so that it does not reappear:

```
PRESS_FIRE AT SCREEN 6
        IF FLAG $PAPEL = 0
        IF PLAYER_TOUCHES 5, 4
        THEN
                SET TILE (5, 4) = 0
                SET ITEM SLOT_SELECTED = 22
                SET FLAG $PAPEL = 1
        END
END
```

The "SET ITEM SLOT_SELECTED = 22" line causes the paper to appear in the inventory, right in the slot that the user had selected.

Obviously, if there was something there will be crushed - to avoid it would have to mount a small pifostio, but for that we have the "floating objects" to see later. Right now it's just like crumbs, it's an example.

We go to another screen of our game, say that is the number 8. Let's say that we have in (7, 7) a character who expects us to give him a paper to write a letter. When we give it to you, you will be "content" and this will make things happen. The state of "contentment" will be expressed in a CONTENT flag, which will be worth 0 at first.

The player must select the item in the inventory and then go to right place and hit action. That will launch our PRESS_FIRE script:

```
PRESS_FIRE AT SCREEN 8
IF SEL_ITEM = 22
THEN
# Do things of oh, a papewl!
# With EXTERN, or whatever, sounds, such.
# Yasta, now ...
SET ITEM SLOT_SELECTED = 0
SET FLAG $ CONTENT = 1
END
END
```

As you can see, we have removed the object from the inventory simply by saying to the game that puts a 0 in the selected slot.

Should not we have done more checking? For example, that $ PAPER = 1 or $ CONTENT = 0. They could be put and would continue to work, but they are not necessary: keep in mind that, on the one hand, paper only can be taken once, and that once given to the uncle will disappear from the Inventory and, therefore, of the game. Therefore, SEL_ITEM can never back to 22. Savings!

"Floating Objects" of type "container".

"Floating Objects" of the type "container" are just "boxes" where we can put an object These boxes are placed on a screen from the script and can be assigned content.

The "Floating Objects" type "container" (from now on, we will to call containers) make handling the inventory very simple, since, when the user presses ACCION by touching one, they will make the object in the selected slot and the object contained in the exchange automatically, with no script in between. That serves us for:

- If the selected slot is empty, "we will take the object".
- If the selected slot is full, the object that was exchanged for the container, and you will not miss anything.
- If the container is empty but not the selected slot, "we will leave the object".

In order to use containers, you must enable them in config.h. Further, you have to let the engine know which flag in our script represents the slot (as defined in the ITEMSET section of the Script):

```
#define ENABLE_FO_OBJECT_CONTAINERS
```

With this we activate the conenedores and we tell the engine that the slot selected is stored in flag 30.

Oh, and do not forget to enable floating objects, in general!

```
#define ENABLE_FLOATING_OBJECTS
```

Creating a Container

Containers can be created from any clause body, but are usually created in the ENTERING sections in an IF TRUE.

Each container is assigned a flag. The value of this flag shall indicate object is in the container, and will be 0 if it is empty.

The containers are created with the following command:

```
ADD_CONTAINER FLAG, X, Y //Creates a container for the FLAG flag in (X, Y)
```

It is good practice to create an alias for each container and put them CONT_, to distinguish them easily. For example, we will create a container for the role of the previous example:

```
DEFALIAS
...
$ CONT_PAPEL 16
...
END
```

The paper came out on screen 6, so let's create that container in the ENTERING SCREEN section of the screen 6. Remember that before we had to print the tile and such (see previous section), but with the containers not it is necessary. Nor will we need a $ PAPER flag or dicks.

```
ENTERING SCREEN 5
IF TRUE
THEN
ADD CONTAINER $ CONT_PAPEL, 5, 4
END
END
```

But where do we say that in that container has to be the object that is represents by tile 22 (the sheet of paper) - Remember that the containers are in general abstractions of flags, so that the container that We have just created a sheet of paper, we must give that value to the flag when? At the start of the game:

```
ENTERING GAME
IF TRUE
THEN
SET FLAG $ CONT_PAPEL = 22
END
END
```

With this, in each game there will be a container in (5, 4) of the screen 6 that has a sheet of paper inside.

When the player reaches screen 6, touch the container (in (5, 4)) and press the action key, the engine will automatically swap content of the slot selected with the contents of the container. Thus, if the slot Selected was empty, it will contain the sheet of paper and the container will remain empty. If we had an object in that slot, it will be in the position (5, 4) and paper in our inventory.

And we will not have to do anything. Neither the script, nor anything.

At the time of going to screen 8 to give it to the jipi, everything remains the same:

```
PRESS_FIRE AT SCREEN 8
IF SEL_ITEM = 22
THEN
SET ITEM SLOT_SELECTED = 0
SET FLAG $ CONTENT = 1
END
END
```

Obviously, the contents of the containers do not have to be created from the beginning of the game. For example, in Leovigildo III the objects for the final puzzle are not available until we speak to Nicanor el Aguador on the last screen. Initially set to 0 and arrived at that point already they are given value.

Other floating objects

The engine supports other types of floating objects: boxes that are transported, for example. All this has to do with the configuration of the engine and such, but as they are placed from the script, I mention it here.

A floating object is represented by a tile number and its location initial. To create a floating object we have to execute the following command in the body of a clause (usually in the IF TRUE within an ENTERING SCREEN section):

```
ADD_FLOATING_OBJECT T, X, Y //Create a floating object with tile T in (X, Y).
```

As we said, the behavior of FO will depend on tile assigned to it and this in turn will depend on how we configured the game.

For example, in Leovigildo the boxes that we can carry and stack are a floating object and are represented by tile 16. For this, we have done the following configuration in config.h:

```
#define ENABLE_FLOATING_OBJECTS
#define ENABLE_FO_CARRIABLE_BOXES
#define FT_CARRIABLE_BOXES 16
#define CARRIABLE_BOXES_ALTER_JUMP 180
```

The first enables floating objects, in general. The second enables the floating objects of type "CARRIABLE_BOXES" (transportable boxes). Third says that these boxes are represented by tile 16. The last one has to see with the configuration of this shit: if defined, the force of the jump will look Altered when we carry a box (to skip less) and the maximum value of the vertical speed will be defined.

With this, we will add one of these nice boxes to the screen 4 in the position (5, 5) by putting this in the script:

```
ENTERING SCREEN 4
IF TRUE
THEN
ADD_FLOATING_OBJECT 16, 5, 5
END
END
```

Checks and commands related to character values

There is a whole set of checks and commands that have to do with the values of the character (eg, life).

Checks

These two tests are deprecated because I think it is much more comfortable define #define OBJECT_COUNT and assign it to a flag, and operate with that flag. They stay here for what I know.

```
IF PLAYER_HAS_OBJECTS //Evaluates to TRUE if the player has objects.
```

```
IF OBJECT_COUNT = n //Evaluates to TRUE if the player has N objects.
```

Commands

INC LIFE n Increases the value of life in n

DEC LIFE n Decreases the value of life in n

RECHARGE Recharges all life (puts it to the maximum)

FLICKER Causes the player to blink during a second and a peak, like when they take a life.

On these last two I say the same thing as before: using OBJECT_COUNT and one flag everything is simpler, but there they continue:

INC OBJECTS n Add n more objects.

DEC OBJECTS n Subtract n objects (if objects> = n; if not objects = 0).

Finish the game

Commands to finish the game from the scripting (it is necessary to activate, in config.h, #define WIN_CONDITION 2, in case we want to WIN from the Script - for GAME OVER Not required).

GAME OVER Finish the game with a GAME OVER.

WIN GAME Finish the game if there are not several levels. In games
With several levels ends the current level (and happens to the next, if your level handler works this way)

GAME_ENDING In multilevel games, the game ends completely and tells the engine to show the Final sequence.

In order for GAME_ENDING to work, it must be indicated in config.h with a nice #define SCRIPTED_GAME_ENDING.

Fire zone

The "fire zone" of a screen is a rectangular zone defined at the level of pixels that will launch the PRESS_FIRE section of the screen (and PRESS_FIRE AT ANY). If the player touches it. It serves us to throw script pieces when the player to touch something or enter somewhere.

You have to activate them in config.h

```
#define ENABLE_FIRE_ZONE
```

To define the active FIRE_ZONE of a screen we use this command from any section of commands (usually in an IF TRUE of ENTERING SCREEN)

```
SET_FIRE_ZONE x1, y1, x2, y2
```

That will define a rectangle from (x1, y1) to (x2, y2), in pixels.

If you want to disable the "fire zone" you just have to put a rectangle out of range or empty:

```
SET_FIRE_ZONE 0, 0, 0, 0
```

To make it less painful to work, and since most of the times fire zones must be calculated based on a range of tiles, msc3 will understand the command

```
SET_FIRE_ZONE_TILES tx1, ty1, tx2, ty2
```

Where the parameters define a range in tile coordinates (both boundaries inclusive) that msc3 will translate internally to a normal SET_FIRE_ZONE.

Modifying enemies

The simplest modification is the one that allows to activate or deactivate enemies. Enemies have, internally, a flag that activates or deactivates them for,. For example, kill them and do not come out anymore. From the script we can modify that flag to get several effects.

For example, in Ninjajar! We make mobile platforms appear as a result of more or less complex actions. For this, we created the platform normally in that screen, we deactivate it in the ENTERING SCREEN, and activate it later when the necessary action has been executed.

On the screens are three enemies, numbered 0, 1 and 2. These numbers correspond to the order in which they were placed in the Place. Must take this into account, because we need this number to reference them:

```
ENEMY n ON //Activates enemy "n".
```

```
ENEMY n OFF //Deactivates enemy "n".
```

Another thing we can do with enemies is to change their type. At type of enemy, with the new engine introduced in Leovigildo III, we have encoded the type of motion, whether or not it fires, and the sprite number (All this detailed in whatsnew.txt, I will not stop here), so with this we have a very powerful tool.

```
ENEMY n TYPE t //Set type "t" for enemy "n".
```

The problem is that we are modifying a basic parameter of the enemy. Yes our game manages several levels does not matter, since we always have one copy (compressed) of the original values that we can restore when we feel like it.

The problem comes in single level games. If we change the type of an enemy, we will lose the original type forever. In order to recover it we will need a "backup" of all enemy types. This copy occupies 3 bytes per screen and must be activated from config.h:

```
#define ENEMY_BACKUP
```

With the backup enabled, we can restore all enemies to its original value or only those of the current screen:

ENEMIES RESTORE Restores to its original values the enemies of the current screen.

ENEMIES RESTORE ALL Restores the type of ALL level enemies.

If you only need to restore them at the start of each game, you can go from using ENEMIES RESTORE ALL in the script and activate the RESTORE_ON_INIT directive in Config.h

```
#define BODY_COUNT_ON n
```

Where n is a flag number, causes the engine to count the deaths on the flag specified. This gives us enough control over the issue of deaths.

```
#define RUN_SCRIPT_ON_KILL
```

Run the PLAYER_KILLS_ENEMY section of the script whenever the player kills an enemy, whatever.

```
#define EXTERN_E
```

It does that instead of EXTERN n in the script we have to use EXTERN_E n, m, and which extern.h is replaced by extern_e.h; Furthermore, do_extern_action now take two parameters n and m instead of one. This serves to multiply by 256 the number of externs available from the script.

External code

There are many things we can not do directly from the script and by the system allows executing external code, which is nothing more than a function defined in the MK2 code.

To use external code it will have to include this function by activating in config.h

Directive

```
#define ENABLE_EXTERN_CODE
```

This will cause the extern.h file to be included in the compilation. This file can contain the code that wins us whenever the entry point be the function

```
Void do_extern_action (unsigned char n);
```

In our script we have the command EXTERN:

EXTERN n Make a call to do_extern_action by passing "n" to it, where n is a number from 0 to 255. Can not be used buildings #.

In the extern.h of almost all games from Ninjajar you will see code to print compressed texts with textstuffer.exe and included in a text.bin. To us it seems very convenient, but you can always use this Characteristic for what you want.

It is possible that 256 possible external actions are few (for example, if you will use a lot of text in your game and you need to do more things – Ninjajar!

Uses 226 lines of text, we are almost without values). In that case we can activate the "extended extern" in config.h. In addition to the above, you have to define

```
#define EXTERN_E
```

Note that this disables EXTERN and does not include the extern.h file. Instead includes extern_e.h and activates the EXTERN_E command. Now the point of entry, inside extern_e.h, must be

Void do_extern_action (unsigned char n, unsigned char m);

And in our script we must use the command:

EXTERN_E n, m Make a call to do_extern_action by passing n and m, where n and m are numbers from 0 to 255. Can not be used buildings #.

Safe Spot

If you define #define DIE_AND_RESPAWN in config.h, the player, upon dying, goes to reappear in the "last safe point". If DIE_AND_RESPAWN is active, the engine saves the position (screen, x, y) each time we Tile nontransferable (other than a floating object).

We can control the definition of the "safe spot" from our script. If we decide to do this (for example, to define a "checkpoint" manually), it is convenient to deactivate that the motor stores the safe spot automatically with:

```
#define DISABLE_AUTO_SAFE_SPOT
```

Whether or not we do this, we can define the safe spot from the script with these two commands:

SET SAFE HERE Sets the "safe spot" to the current position of the player.

SET SAFE n, x, y Sets the "safe spot" to screen n in the coordinates (of tile) (x, y).

Miscellaneous Commands

I did not know where to put these...

SOUND n Play the sound n. It will depend on what sound n.

TEXT text Prints a text in the text line if we have it defined in config.h with the #define LINE_OF_TEXT, LINE_OF_TEXT_X, and LINE_OF_TEXT_ATTR. The text goes to hair, without quotes, and instead of spaces you have that put "_". We have not used this for eons ...

PAUSE n Wait n frames, that is, n = 50 is a second. Eye which only works on 128K since it uses HALT. 48K games have interrupts off so using PAUSE will pause the game FOREVER.

MUSIC n Play music n. Obviously, only in 128K games.

Version Differences

Version 3.99.2

Come on, the churreras are going out like hotcakes. We're breaking it, and we can think of new things every day. We'll get them in as we can think of games that take them.

These are the new things that are in this version of the churrera:

Timers

Added to the churrera a timer that we can use automatically or from the script. The timer takes an initial value, counts toward Down, can be recharged, can be set every how many frames is decremented or decide what to do when it runs out.

```
#define TIMER_ENABLE
```

TIMER_ENABLE includes the code required to operate the timer. This code will need some other directives that specify the form of function:

```
#define TIMER_INITIAL 99
#define TIMER_REFILL 25
#define TIMER_LAPSE 32
```

TIMER_INITIAL specifies the initial value of the timer. The time, which are set with the setter as type 5 hotspots, will recharge the value specified in TIMER_REFILL. The maximum value of the timer, both for the as recharging, is 99. To control the amount of time elapses between each decrement of the timer, we specify in TIMER_LAPSE the number of frames that must pass.

```
#define TIMER_START
```

If TIMER_START is set, the timer will be active from the beginning.

We also have some directives that define what will happen when the timing to zero. It is necessary to uncomment those that apply:

```
#define
TIMER_SCRIPT_0
```

Defining this, when it reaches zero the timer will execute a section script special, ON_TIMER_OFF. It is ideal for carrying all the control of the timer by scripting, as it happens in Cadàveriön.

```
// # define TIMER_GAMEOVER_0
```

Defining this, the game will end when the timer reaches zero.

```
// # define TIMER_KILL_0
// # define TIMER_WARP_TO 0
// # define TIMER_WARP_TO_X 1
// # define TIMER_WARP_TO_Y 1
```

If TIMER_KILL_0 is set, a life will be subtracted when the timer reaches zero. If, in addition, TIMER_WARP_TO is defined, it will also be changed to the screen the player appears in the coordinates TIMER_WARP_TO_X and TIMER_WARP_TO_Y.

```
// # define TIMER_AUTO_RESET
```

If this option is set, the timer will return to maximum after reaching zero automatically. If you are going to perform the control by scripting, better leave it commented

```
#define SHOW_TIMER_OVER
```

If this is defined, in the case that we have defined either TIMER_SCRIPT_0 or well TIMER_KILL_0, a "TIME'S UP!" Poster will be displayed. When the timer reaches zero.

Scripting:

As we have said, the timer can be administered from the script. Is interesting that, if we decided to do this, let's activate TIMER_SCRIPT_0 so that when the timer reaches zero, the ON_TIMER_OFF section of our script and that control is total.

In addition, these checks and commands are defined:

Checks:

```
IF TIMER> = x
IF TIMER <= x
```

Which will be fulfilled if the value of the timer is greater than or equal to or less or equal than the specified value, respectively.

Commands:

```
SET_TIMER a, b
```

It is used to set the TIMER_INITIAL and TIMER_LAPSE values from the script.

```
TIMER_START
```

It is used to start the timer.

```
TIMER_STOP
```

It is used to stop the timer.

Control of push blocks

We have improved the engine so that more can be done with the tile 14 of type 10 (pushable tile) that simply push it or stop the trajectory of the enemies. Now we can tell the engine to launch the PRESS_FIRE section of the current screen just after pushing a pushable block. Besides, the number of the tile that is "stepped" and the final coordinates are stored in three Flags that we can configure, to be able to use them from the script to do checks.

This is the system that is used in the script of Cadàveriön to control that put the statues on the pedestals, to give an example.

Recall what we had so far:

```
#define PLAYER_PUSH_BOXES
#define FIRE_TO_PUSH
```

The first one is necessary to activate the pushable tiles. The second obliges the player to press FIRE to push and therefore is not mandatory. Let's see now the new directives:

```
#define ENABLE_PUSHED_SCRIPTING
#define MOVED_TILE_FLAG 1
#define MOVED_X_FLAG 2
#define MOVED_Y_FLAG 3
```

Enabling ENABLE_PUSHED_SCRIPTING, the tile to be pressed and its coordinates will store in the flags specified by the MOVED_TILE_FLAG, MOVED_X_FLAG and MOVED_Y_FLAG. In the code shown, the tile is will store in flag 1, and its coordinates in flags 2 and 3.

```
#define PUSHING_ACTION
```

If we define this, in addition, the PRESS_FIRE AT ANY and PRESS_FIRE of the current screen.

We recommend to study the Cadàveriön script, which, besides being a good example of the use of the timer and the pushbutton control, results be a rather complex script that employs a lot of advanced techniques.

Check if we get off the map

It is advisable to put limits on your map so that the player can not Exit, but if your map is narrow you may want to take advantage of the screen. In that case, you can activate:

```
#define PLAYER_CHECK_MAP_BOUNDARIES
```

It will add checks and will not let the player leave the map. eye! If you can avoid using it, the better: you will save space.

Type of enemy "custom" gift

Until now we had left the type 6 enemies without code, but we have thought that it is not difficult for us to put one, for example. It behaves like the bats of Cheril the Goddess. To use them, place them in the of enemies as type 6 and uses these directives:

```
#define ENABLE_CUSTOM_TYPE_6
#define TYPE_6_FIXED_SPRITE 2
#define SIGHT_DISTANCE 96
```

The first one activates them, the second defines which sprite to use (minus 1, if you want the sprite of enemy 3, put a 2. Sorry for the slut, but saving bytes). The third one says how many pixels you see from far away. If he sees you, he follows you. If not, return to your site (where you've put it with the setter).

This implementation, in addition, uses two directives of the enemies of type 5 to operate:

```
#define FANTY_MAX_V 256
#define FANTY_A 12
```

Define there the acceleration and the maximum speed of your type 6. If you go to also use type 5 and you want other values, be a man and modify the engine.

Keyboard / joystick configuration for two buttons

There are side view games that are best played with two buttons. If you activate this directive:

```
#define USE_TWO_BUTTONS
```

The keyboard will be the following, instead of the usual one:

- A = Left
- D = Right
- W = Up
- S = Down
- N = jump
- M = shot

If joystick is selected, FIRE and M fire, and N skips.

Shooting up and diagonally for side view

Now you can let the player shoot up or diagonally. To do this define this:

```
#define CAN_FIRE_UP
```

This configuration works best with USE_TWO_BUTTONS, since this separates "Top" of the jump button.

If you do not hit "up", the character will fire to where he is looking. Yes Press "up" while shooting, the character will shoot up. Yes, In addition, you are pressing an address, the character will shoot on the diagonal indicated.

Masked bullets

For speed, the bullets do not wear masks. This works fine if the background on which they move is dark (few active INK pixels). But nevertheless, there are situations where this does not happen and looks bad. In that case, we can activate masks for bullets:

```
#define MASKED_BULLETS
```

Version 3.99.2mod

This was a special version with a thing that Radastan asked us, the...

Animated Tiles

Everything is based on tilanim.h. This file is included if config.h is defined in ENABLE_TILANIMS directive. In addition, the value of this directive is what defines The number of smaller tile that is considered animated.

In tilanim.h there are, in addition to the definition of data, two functions:

Void add_tilanim (unsigned char x, unsigned char y, unsigned char t) is called from the function that paints the current screen if it detects that the tile that you are going to paint is> = ENABLE_TILANIMS. Add an animated tile to the list tiles.

Void do_tilanims (void) is called from the main loop. Basically select a random animated tile among all the stored ones, change the Frame (from 0 to 1, from 1 to 0) and draws it.

To use it, you just have to define the ENABLE_TILANIMS directive in config.h with the smaller animated tile. For example, if your last four tiles (8 in total) are animated, put the value 40. Then, on the map, you have to put the smaller tile of the pair, that is, tile 40 for 40-41, the 42 to 42-43... If you do not do that, funny things will happen. The code is (It has to be) minimal, do not check anything, so take care.

By the way, this has not been proven. If you put it in your game and peta, damages one touch.

Version 3.99.3

Animated Tiles

If you define:

```
#define ENABLE_TILANIMS 32 // If defined, animated tiles are enabled.
// the value specifies firt animated tile pair.
```

In config.h, tiles> = that specified index are considered animated. In the tileset, they come in pairs. If, for example, "46" is defined, then the only pair of tiles 46 and 47 will be animated. The engine will detect them and each frame will cause one of the tiles 46 to change state.

There can be up to 64 animated tiles on the same screen. If you put more, it will not work.

128K Mode

You have to do a lot of manual work with this. I'm sorry, but it's like that. First you will have to create a make.bat that will build everything you need. For that you can rely on the file spare / make128.bat and adapt it to your project.

The 128K mode is the same as the 48K but use WYZ Player and also supports several levels. You can not have longer levels, but you can have several levels.

To use it, you need to activate three things in config.h:

```
#define MODE_128K // Experimental!
#define COMPRESSED_LEVELS // use levels.h instead of map.h and enems.h (!)
#define MAX_LEVELS 4 // # of compressed levels
```

In MAX_LEVELS you have to specify the number of levels you are going to use.

In churromain.c you have to change the position of the pile and place it below of the main binary:

```
#pragma output STACKPTR = 24299
```

Then you have to modify levels128.h, which is where the level structure is defined and is included in 128K mode. There you will see an array levels, with information about the
Levels. In principle, very little information is included:

```
// Level struct
LEVEL levels [MAX_LEVELS] = {
3,2,
(4.3),
{5.4},
{6.5}
};
```

The first value is the resource number (see below) that contains the level. The second value is the song number in WYZ PLAYER that should ring while it is played at level.

To prepare a level, you have to use the new buildlevel.exe utility in / utils. This utility takes the following parameters:

```
$ Buildlevel map.map map_w map_h lock font.png work.png spriteset.png
Extrasprites.bin enems.ene scr_ini x_ini y_ini max_objs enems_life behs.txt level.bin
```

- Map.map Is mappy mappy
- Map_w, map_h Are the dimensions of the map on screens.
- Lock 15 for autodetect locks, 99 if no locks
- Font.png is a 256x16 file with 64 ascii characters 32-95
- Work.png is a 256x48 file with the tileset
- Spriteset.png is a 256x32 file with the spriteset
- Extrasprites.bin you find it in / levels
- Enems.ene the file with the enemies / hotspots of colocador.exe
- Scr_ini, scr_x, scr_y, max_objs, enems_life level values
- Behs.txt a file with tile types, separated by commas
- Level.bin is the output file name.

When we have all levels built, we have to compress them with apack:

```
$ /utils/apacke.exe level1.bin level1c.bin
...
```

When we have all levels compressed, we will have to create the images binary files that will be loaded into the extra RAM pages. For that we use the utility Librarian that is in the folder / bin. In fact, it is a good idea to work in Folder / bin for this.

The librarian utility uses a list list with the compressed binaries that should be getting into the binary images that will go on the extra pages of RAM. The first thing we will have to put in is the title.bin, marco.bin and ending.bin, in that order. If you do not have .bin you should use a Length 0, but you must specify it. Then we will add our levels. By example:

Title.bin
Marco.bin
Ending.bin
Level1c.bin
Level2c.bin
Level3c.bin
Level4c.bin

There we added four compressed levels.

When you run librarian, you will be filling in 16K images destined to go in the extra RAM. First it will create ram3.bin, then ram4.bin and finally ram6.bin, according to I need more space.

It will also generate the file librarian.h, which we will have to copy to / dev. Here we can see the resource number associated with each binary:

```
RESOURCE resources [] = {
        {3, 49152}, // 0: title.bin
        {3, 50680}, // 1: marco.bin
        {3, 50680}, // 2: ending.bin
        {3, 52449}, // 3: level1c.bin
        {3, 55469}, // 4: level2c.bin
        {3, 58148}, // 5: level3c.bin
        {3, 60842} // 6: level4c.bin
};
```

These resource numbers are the ones we will have to specify in the array levels mentioned above. In particular, resources 3, 4, 5 and 6 are those containing the four levels.

With all this done and prepared, we will have to mount the tape. For this there are which create a suitable loader.bas (you can see an example in /spare/loader.bas) and build a .tap with each block of RAM (again, the example in /spare/make.bat builds the tape with binaries in RAM3 and RAM4).

You will also need RAM1.BIN to build RAM1.TAP, containing the player of WYZ with songs. For this you have to modify /mus/WYZproPlay47aZX.ASM in / mus to include your songs. You have an example in / spare.

As you can see, it's a bit tedious. I recommend that you build mini-projects in 48K as you make the levels, and finally you build a 128K version with all.

In addition, you can use extra space to push more compressed screens, or even code to use passwords to jump directly to levels. You can see examples of all this in Goku Mal 128.

Type 3 Hotspots

We have made this modification, proposed in the forum, fixed to blow of directive. If you define

```
#define USE_HOTSPOTS_TYPE_3 // Alternate logic for recharges.
```

The recharges will appear only and exclusively where you place them, using the type 3 hotspot.

Pause / Abort

If it is defined

```
#define PAUSE_ABORT // Add h = PAUSE, y = ABORT
```

Code is added to enable the "h" key to pause the game and the key "And" to interrupt the game. If you want to change the assignment you will have to touch the code in mainloop.h

Message catching objects

If it is defined

```
#define GET_X_MORE // Shows "get X more"
```

A message will appear with the items you have left each time you take one.

3.99.3b

Minimal revision. It is arranged to be able to have 128K games with Only 1 level (ie use MODE_128K without COMPRESSED_LEVELS).

Right now there are two examples that can help you if you want to make a 128K game:

- Goku Mal: 128K with compressed levels. See this doc and the sources of the game.
- The new adventures of Dogmole Tuppowsky: 128K with only one level, more info in
The forum of mojonia.

Also, in spare I added the file extern-texts.h whose contents you can use In extern.h if you want an easy way to display text on the screen using the EXTERN command n of the script.

3.99.3c

```
#define PLAYER_CAN_FIRE_FLAG 1
```

If set, the indicated flag controls whether the player can (1) or not (0) fire.

Items Engine

We want to make sure there is a small inventory on screen and can Select an object from it, and we also want the objects that make up The inventory are not fixed and we can know, from the script, which object Is selected.

- In an initial section of the script, we will define "the itemset" (we must Put names to things, even if they are names as dull as this): how many Spaces have, where they are placed, and how objects are distributed. Something like that:

```
Code:
ITEMSET
# Number of holes:
SIZE 3

# Position x, y
LOCATION 2, 21

# Horizontal / vertical, spaced
DISPOSITION HORZ, 3

# Color and characters to paint the selector
SELECTOR 66, 82, 83

# (If defined) which tile represents the empty tile
EMPTY 31

# Flag containing which gap is selected
SLOT_FLAG 14

# Flag containing what object is in the selected hole
ITEM_FLAG 15
END
```

- An object is represented by its tile. If we have a crown on tile 10, The crown object will be 10. If in an inventory hole is 10, It means that in that hollow is the crown. A value of 0 will always represent An empty space. This simplifies the code an awful lot.

- There will be changes in the script. ITEM n = t means that in slot "n" is The object represented by tile t. We therefore define the following terms:

```
Code:
IF ITEM n = t
IF ITEM n <> t
```

They verify that in the space "n" is or not the object of tile "t".

```
IF SEL_ITEM = t
```

Check that in the space selected by the selector is the object of Tile "t"

And the following commands:

```
SET ITEM n = t
```

Sets tile n in tile t. Obviously, to remove an object from the gap N, we will set a 0.

There is a limitation, therefore, in the number of objects that the Character at a time. With a little head, as I said, it can be managed This very well, and with a minimum of code added to the engine we have a Quite powerful tool. All this has to be combined with the flags for Have full functionality. With ITEM's, we can only know if we have Not an ITEM in the inventory, but not if it has already been used. For that we need The flags.

How is this used? Let us give an example.

Imagine that on screen 6 we have a "crown" object, represented by the Tile 33, and we have it in (7, 7). In addition, the flag indicating its status is 3, Which will be worth 0 when we have not yet caught it or anything, to paint it on the screen.

```
Code:
ENTERING SCREEN 6
IF FLAG 3 = 0
THEN
SET TILE (7, 7) = 33
END
END
```

We will manage to take it. We can do it in basic mode or mode Virguero. Let's look at the basic mode first. In the basic mode we assign "by hand" A fixed gap for each item. The crown will be placed in the hole 2:

```
PRESS_FIRE AT SCREEN 6
IF PLAYER_TOUCHES (7, 7)
IF FLAG 3 = 0
THEN
SET FLAG 3 = 1
SET TILE (7, 7) = 0
SET ITEM 2 = 33
END
END
```

The game with flag 3 is simply so it does not re-draw. When the Flag 3 valga 1 will not re-paint the object when re-entering the Screen, or try to pick it up again. Otherwise, what is done is
Make the object 33 in the gap 2.

Imagine that in screen 12 we have to use it in coordinate 5, 8. Well You have to check that the selected item is 33:

```
PRESS_FIRE AT SCREEN 12
IF SEL_ITEM = 33
THEN
SET ITEM 2 = 0
# more things
END
END
```

If the selected object is 33 (which can only occur if We put it in the slot 2), we remove it from the inventory (putting a 0 in the slot 2) and then we do more things.

The virgin mode is for the object to go to the selected hole. For that we use The indirection allowed by the scripting engine with the # operator. Remember That we are using the flag 10 to represent the selected hole. Let's play with that. Also, check that the gap is free!

```
PRESS_FIRE AT SCREEN 6
IF PLAYER_TOUCHES (7, 7)
IF FLAG 3 = 0
IF FLAG 10 <> 0
THEN
# Evil! The gap is not free!
SOUND 2
END

IF PLAYER_TOUCHES (7, 7)
IF FLAG 3 = 0
IF FLAG 10 = 0
THEN
SET FLAG 3 = 1
SET TILE (7, 7) = 0
SET ITEM # 10 = 33
END
END
```

What do we do? Then place the object of tile 33 (our crown) in space Selected, which is nothing more than the one stored in flag 10 (remember Which # 10 means "the value of flag 10").

To prove that we have it, then the same thing.

What do you think? Doubts? Something to comment? If it does, I'll do exactly as I have described.

MT Engine MK2 v 0.8

We rewrote the almost whole engine. The enemy module is still missing We want to reorganize. That's why we're not in 1.0 yet.

The new engine works almost the same as the Churrera 3.X, but it works Faster and takes up less memory. There are also a lot of new things, like The "hitters" of Ninjajar (for now to give hosts, but soon to do Swords), better multi-phase support in scripting, able to jump from one phase To another, improved item engine...

Floating Objects

```
#define ENABLE_FLOATING_OBJECTS
```

They are interactive tiles that are not part of the map. They are placed from the Script. For now the engine can handle two types.

The floating objects are placed on each screen from the script:

```
ADD_FLOATING_OBJECT t, x, y
```

Carriable boxes

```
#define ENABLE_FO_CARRIABLE_BOXES
#define CARRIABLE_BOXES_ALTER_JUMP 180
#define FT_CARRIABLE_BOXES  16
```

They are boxes that can be transported. You need us to reserve 10 blocks of More sprites (see main.c, at first) for an extra sprite. The boxes are taken And deposit by pressing DOWN. Boxes are affected by gravity and are Stacking each other.

You can define the tile they use with the FT_CARRIABLE_BOXES directive. The Objects use the behavior defined for this tile. It will be the value That you have to give "t" at the time of placing them from the script, for example To place one of these boxes in position 7, 8 having defined that its Tile is the 16 we do:

```
ADD_FLOATING_OBJECT 16, 7, 8
```

If you define CARRIABLE_BOXES_ALTER_JUMP, the maximum value of the speed of Jump will change to the specified value when we carry a box.

Item containers

```
#define ENABLE_FO_OBJECT_CONTAINERS
#define FT_FLAG_SLOT 30
```

These are intended to be used with scripting inventory engine. Each container actually represents a flag of the scripting system. To the Paint the screen will paint the tile whose number is stored in the flag corresponding.

To create them from the script:

```
ADD_FLOATING_OBJECT 128 + f, x, y
```

Where f is the flag we want to represent. For example, if we are going to use the Flag 10 to represent a container in position 4, 4 of the screen, We should create it like this:

```
ADD_FLOATING_OBJECT 138, 4, 4
```

As this is a bit confusing, we have added an alias. The same can be done calling to:

```
ADD_CONTAINER f, x, y
```

The previous example would be:

```
ADD_CONTAINER 10, 4, 4
```

The engine reacts to these blocks by exchanging the selected object of the Inventory with the one in the container.

Alias in script

Because doing Ninjajar we wanted to go crazy with so much flag, we have Added aliases. We define a block at the beginning of the script like this:

```
DEFALIAS
        $ ALIAS N
        ...
END
```

From then on, we can substitute "N" for "$ ALIAS". For example if We use flag 2 to open a green door and flag 3 to see if we have Spoken with the ogre we do:

```
DEFALIAS
        $ PUERTA_VERDE 2
        $ HABLA_OGRO 3
END
```

In the script we can use the alias instead of the numeric:

```
...
IF FLAG $ HABLA_OGRO = 0
THEN
EXTERN 10
SET FLAG $ HABLA_OGRO = 1
END
…
```

MT Engine MK2 v 0.85

Changes and additions for the second charge of Leovigildo. They are a lot to see If I remember:

Throwing Carriable Boxes

You can launch the CARRIABLE_BOXES boxes by pressing FIRE. Boxes kill The bugs and count them in a flag:

```
#define CARRIABLE_BOXES_THROWABLE       // If defined, carriable boxes are throwable!
#define CARRIABLE_BOXES_COUNT_KILLS 2 // If defined, count # of kills and store in flag 2.
```

Single-display mode.

You can only change the screen by without scripting. Detects the change of screens of a lifetime when the child sticks to the edge. This is for making screen games on screen.

```
#define PLAYER_CANNOT_FLICK_SCREEN // If defined, automatic screen flicking is disabled.
```

Counting Enemies

Count how many enemies are on the screen and put them in a flag. To do Games of "kill them all to pass" or things like that, of the stick "si You kill all the bugs something happens ".

```
#define COUNT_SCR_ENEMS_ON_FLAG 1 // If defined, count # of enemys on screen and store in flag #
```

Show level

Each time you change the screen, display the screen number +1. In plan number of level. For screen-to-screen games.

```
#define SHOW_LEVEL_ON_SCREEN  // If defined, show level # whenever we enter a new screen
```

Enemies Module

Let's change the enemy module, so let's get ready.

Old Style Enemies

As the old module I will not erase it, you can continue using it if you specify

```
#define USE_OLD_ENEMS  // If defined, use old enems (like in Churrera)
```

Disable Platform

Disable mobile platforms in platform games. Now you can have Four different enemies.

```
#define DISABLE_PLATFORMS // If defined, type 4 are enemies in side-view mode
```

Resurrecting Enemies

Enemies resurrect as they enter the screen. Okay, this was it, but Now has some changes:

```
#define RESPAWN_ON_ENTER // Enemies respawn when entering screen
#define RESPAWN_ON_REENTER // Respawn even on a REENTER in the script
// (by default REENTER does not respawn enemies!)
```

If you activate the first one, when entering with the doll in a screen the enemies, they will come back to life.

If you activate, in addition, the second, the enemies will come back to life also after the REENTER command in the script.

We have improved a lot of things, including the timer, which was something broken (there will still be many things of the Churrera that are broken, let's go little To little).

MT Engine MK2 v 0.86

Phantomas Engine Edition. Now you can make Phantomas games – this Opens the possibility of adding more easily movement engines Purely linear (without inertia).

```
// Phantomas Engine
//
// Comment everything here for normal engine
#define PHANTOMAS_ENGINE  1           // Which phantomas engine:
                                              // 1 = Phantomas 1
                                              // 2 = Phantomas 2
                                              // 3 = LOKOsoft Phantomas
                                              // 4 = Abu Simbel Profanation

#define PHANTOMAS_FALLING4          // Falling speed (pixels / frame)
#define PHANTOMAS_WALK     2        // Walking speed

#define PHANTOMAS_INCR_1   2        // Used for jumping
#define PHANTOMAS_INCR_2   4
#define PHANTOMAS_JUMP_CTR          16        // Total jumping steps up & down


// Most things from now on will not apply if PHANTOMAS_ENGINE is on ...
// Try ... And if you need something, just ask us ... Maybe it's possible to add.

// For example, BOUNDING_BOX_8_BOTTOM works for PHANTOMAS / PROFANANTION engines.
```

It's simple (or not). There are four engine types, as seen in the code. Then there are configuration parameters.

As it is, by selecting engine 1 the movement will be as in Phantomas 1, engine 2 will do as Phantomas 2, and engine 4 as Abu Simbel Profanation... But playing with values you will get others things.

Engines 1, 2 and 4 are based on two types of jumps. At engine 1 we have high jump (2 high tiles, 1 wide) and long jump (1 tile high, 4 wide). In engine 2 we have a long jump (slightly more than 2 tiles of high, 2 tiles of width) and short (1 tile x 1 tile), and in addition We can change the direction of the jump in the middle of the air. In the engine 4 we have jumps just like in engine 2, but we can not change the Direction and also if we press only jump the doll will jump to arrive (It is necessary to press jump + left or right to jump sideways).

In addition we have used to add support for enemies 100% custom To the enemy module, and we have included a pair as "addons": drops and Arrows, which use their own sprites. In addition, there is a new utility To convert sprites for these goings-on.

```
#define ENABLE_DROPS            // Enemy type 9 = drops
#define ENABLE_ARROWS // Enemy type 10 = arrows
```

These enemies are placed with the setter. Look at the files

```
\ Dev \ addons \ drops \ move.h
\ Dev \ addons \ arros \ move.h
```

To see how your values are specified. Or the question in the forum.

The best thing is that if you want to use this we put a message or something asking The example micro-game where everything is seen in action. I do not think so Let's get it out of the way, I do not have the time or the energy.

MT Engine MK2 v 0.87

Edition Leovigildo III. It has few purely new things, but it brings a bunch of internal improvements, bug fixes, optimizations...

The new module of enemies.

To use it, make sure to comment #define USE_OLD_ENEMS.

Now enemies are much more flexible. Each defines several things:

- What sprite he uses, from 0 to 3.
- What kind of movement does it take: linear, flying..
- Whether he fires or not.

It is also prepared to make it very easy to get more behaviors.

Everything is specified in the enemy type, which is divided into several fields at bit level.

7	6	5	4	3	2	1	0
X	B	B	B	B	F	S	S

Where:

- X is reserved to mark if an enemy is crushed.
- BBBB is the type of movement. For now there are implemented these:

0001 (1) - round-trip linear, as always.
0010 (2) - flying. Like the fantys type 6.
0011 (3) - tracker. The bum type 7 always.
1000 (8) - mobile platform. As "1" but mobile platform.

If you use type 2, you have to enable ENABLE_FLYING_ENEMIES. Yes. Use type 3, enable ENABLE_PURSUE_ENEMIES.

- F is whether it triggers (1) or not (0). If you activate it for some enemy, remember that you have to enable ENABLE_SHOOTERS and set MAX_COCOS and other things.

- SS is the sprite number, according to spriteset, from 0 to 3 (00, 01, 10, 11).

The "type" of enemy is calculated, therefore, using this formula calculation:

S + 4 * F + 8 * B, where S is the sprite, F is fired, and B is the behavior

To place them you can calculate the value of the type of enemy using The above formula (is binary) or use the MK2 Putter there In / enems, where you can set the values separately.

Modifications to floating objects

- You can now better control your behavior with respect to the Gravity - if we want to use them in games of genital view, more than nothing:

```
#define FO_GRAVITY
#define FO_SOLID_FLOOR
```

Activating the first, the FO will fall if there is no floor underneath. With the They will stop when they reach the bottom sign of the screen instead To disappear.

Floating object that hurt

FOs "kill" while you carry them on their backs. This is for the nonsense Of this game, I do not know if it will serve for anything else... Anyway I do not It cost to put it in the config instead of a custom paranoia...

```
#define CARRIABLE_BOXES_DRAIN 7
```

Floating objects with Springs

Couplings. FOs can be springs. If you fall on them, you will rebound. You can set the maximum bouncing speed.

```
#define CARRIABLE_BOXES_CORCHONETA
#define CARRIABLE_BOXES_MAX_C_VY 1024
```

For this to work you have to give it the "bouncing" behavior To the tile that represents the corkboard. This means that in a Future version we can use bouncing tiles that are not FO, only Defining the behavior... I believe.

```
// 0 = Walkable (no action)
// 1 = Walkable and kills.
// 2 = Walkable and hides.
// 4 = Platform (only stops player if falling on it)
// 8 = Full obstacle (blocks player from all directions)
// 10 = special obstacle (pushing blocks OR locks!)
// 16 = Breakable (#ifdef BREAKABLE_WALLS)
// 32 = Conveyor
// 64 = CUSTOM FO -> SHEET!
```

Scripting FO

This can be useful for many things but you have to use it carefully. HE Used in Leovigildo III to detect that we threw a The head to the tamer.

Basically, if activated, when a FO "falls", its type and position is stored In three flags (configurable) and PRESS_FIRE is called in the script of that screen.

```
#define ENABLE_FO_SCRIPTING
#define FO_X_FLAG 1
#define FO_Y_FLAG 2
#define FO_T_FLAG 3
```

This is starting to be very scary. I was already scared in Ninjajar. Now it's scary.

MT Engine MK2 v 0.88

Edited by Leovigildo F. WTF? He's an EASTER EGG, but I've expanded The engine in various directions. Some may be useful to others Games, and others do not. Let's see if I remember everything:

Scripting stuff without scripting

They are things that can be done with scripting, but being simple and You can store the pots in an array, if you were just going to use scripting For this, you save it and you can fit more. Namely:

```
- engine / levelnames.h
```

Allows you to name each screen. Names must have A fixed length and defined all in the same chain, all followed And in order. You can see it in levelnames.h itself, where you can also Set up location and color and such. To activate:

```
#define ENABLE_LEVEL_NAMES
```

```
- engine / extraprints.h
```

Additional Tiles Impressions

Allows you to define extra tile impressions for each screen. There is a lot People who have used scripting only for this (which seems to me a Shame, if you ask me). You do not need to activate the scripting, the Code takes up very little and each print only 2 bytes. To activate:

```
#define ENABLE_EXTRA_PRINTS
```

To set what will be printed, edit extraprints.h. There Defines an array for each screen with extra prints. Each extra print is It consists of 2 bytes: "xy" and "tile". "Xy" uses 4 bits for X and 4 for Y. It is Very easy to manage in hexadecimal, "x" goes from 0 to F and "y" from 0 to 9. The Byte "tile" is simply the tile number. The list ends with a 0xff (value 255).

For example, to print a tile 17 at the position X = 10, Y = 2, the two Bytes would be 0xA2, 17. To print a tile 33 at the position X = 5, Y = 7 The two bytes would be 0×57, 33.

Simple Item Manager (SIM)

Finally, there is another array * prints with an entry for each screen of the Map. If there are no extra prints on a screen, a 0 is set. If yes There are, the array of prints of the corresponding screen is set.

```
- engine / sim.h
```

SIM stands for "Simple Item Manager", and is used to manage items and Without requiring scripting for simple games in which there is X Objects in Y containers across the map, and the game is terminated when The X objects have been placed in other places. No more.

```
#define ENABLE_SIM
```

Activates the SIM. This will put redundant code with the system Of scripting, so both systems are NOT COMPATIBLE. If you use Scripting, manage your objects by hand.

The SIM has a few directives to configure it:

```
// General
#define SIM_MAXCONTAINERS 6
#define SIM_DOWN
// # define SIM_KEY_M
// # define SIM_KEY_FIRE
```

The first, SIM_MAXCONTAINERS, defines the maximum number of containers (Other than objects) that will be in the game. A container may be Empty or contain an object. Games in which you have to put three Objects in three different sites, for example, will need six Containers: the 3 that will contain the objects, and 3 empty ones with the "final destination".

The next three, define which key is used to interact (pick / Leave object). Respectively, below, M or FIRE. Define only one, like With scripting.

A component of the SIM is inventory. The inventory is exactly the Same that you get when you use scripting and you define it in your script. HE Configured with the following directives:

```
// Display:
#define SIM_DISPLAY_HORIZONTAL
#define SIM_DISPLAY_MAXITEMS      2
#define SIM_DISPLAY_X             24
#define SIM_DISPLAY_Y             21
#define SIM_DISPLAY_ITEM_EMPTY    31
#define SIM_DISPLAY_ITEM_STEP     3
#define SIM_DISPLAY_SEL_C         66
#define SIM_DISPLAY_SEL_CHAR1     62
#define SIM_DISPLAY_SEL_CHAR2     63
```

If SIM_DISPLAY_HORIZONTAL is set, the inventory will be displayed in a horizontal line. If not defined, it will be displayed in a vertical line.

SIM_DISPLAY_MAXITEMS defines the number of slots in the inventory.

SIM_DISPLAY_X and SIM_DISPLAY_Y indicate the coordinate of the screen where The inventory is displayed. SIM_DISPLAY_ITEM_STEP defines how many Character cells will draw a new slot from the coordinates initials.

SIM_DISPLAY_ITEM_EMPTY specifies which tile represents the empty slot.

SIM_DISPLAY_SEL_C specifies the color of the selector, and SIM_DISPLAY_SEL_CHAR1 And SIM_DISPLAY_SEL_CHAR2 which two characters of your charset to use for To draw it.

Once all this is defined, we will have to open engine / sim.h to finish To configure our game.

In sim.h, two arrays are defined: sim_initial and sim_final. The first defines The location of the containers on the map and its initial contents; he Second defines a final state that will win the game if it is reached.

Sim_initial is an array of structures. An entry is defined for each Container of the game (in total, SIM_MAXCONTAINERS entries). Each entry Has a format {n_pant, XY, tile}, where n_pant is the screen where Find, XY are the coordinates (4 bits X, 4 bits Y, as in extraprints) And tile is the tile that represents the object contained in the container at Principle of the game.

For example {10, 0×54, 32} will cause the display 10 to have a container At the position X = 5, Y = 4, initially having the object 32.

{3, 0×B3, 0} will cause the screen 3, at the position X = 11, Y = 3, to have an empty container.

Sim_final is an array of numbers. It simply specifies which item should Have in each container to finish the game.

For example, imagine a silly game where there is an object in the screen 0 And another on screen 1, and you have to swap them to win. The objects They will both appear in X = 7, Y = 4, and will be represented with tiles 20 and 21.

In that case, SIM_MAXCONTAINERS would be worth 2 and our arrays would be:

```
SIM_CONTAINER sim_initial [SIM_MAXCONTAINERS] = {
        {0, 0×74, 20},
        {1, 0×74, 21}
};
```

```
Unsigned char sim_final [SIM_MAXCONTAINERS] = {21, 20};
```

As simple as this. At first, the containers contain the objects 20 and 21, and at the end must contain the objects 21 and 20.

IMPORTANT NOTE: SIM needs to activate Floating Objects of type Container in config.h

```
#define ENABLE_FO_OBJECT_CONTAINERS
```

In addition: if you have not tangled should be the same, but make sure everyone That the value of FT_FLAG_SLOT in config.hy of FLAG_SLOT_SELECTED in Yes they correspond. It looks like a sandbox, but it's like this for possible future Expansions.

Improvements in JETPAC

The Jetpac occurred to us in the Churrera 1.0 and we program it, but without try. We did not use it until the Churrera 3.1, when we did Cheril the Goddess, and it was very rawro. Then we use it on Jet Paco.

From the beginning we had thought of giving chicha with refills, fuel That is over, and things like that, but even NOW has not been done.

```
#define PLAYER_HAS_JETPAC
#define JETPAC_DEPLETES          4
#define JETPAC_FUEL_INITIAL      25
#define JETPAC_FUEL_MAX          25
#define JETPAC_AUTO_REFILLS      2
// # define JETPAC_REFILLS
// # define JETPAC_FUEL_REFILL   25
```

PLAYER_HAS_JETPAC activates this system for a lifetime. If you only define This, you will have a jetpac as in jetpaco. Cool.

If you activate JETPAC_DEPLETES with value "X", the jetpac will have fuel that is Will exhaust each X frames, with X a power of 2 (2, 4, 8, 16 ...). Having this active, we can define more behaviors.

JETPAC_FUEL_INITIAL and JETPAC_FUEL_MAX are required in any case if Active

JETPAC_DEPLETES. Specify the fuel value at the beginning of the game And the maximum value that can be reached.

With this we have made the fuel to be spent. Now we must decide how Recover it

If you activate JETPAC_AUTO_REFILLS with value "Y", the jetpac will recharge only When not being used, each Y frames, with Y a power of 2.

If, instead, you activate JETPAC_REFILLS, you will see refills placed Such as type 6 hotspots in the colander. Each recharge will recharge the number of Units specified in JETPAC_FUEL_REFILL.

Whoa

Yes, that's what I say. For an easter egg. I have also fixed a few bugs That I've seen out there and cleaned up some things.

MT Engine MK2 v 0.88c

Warning: As of this version the classic module of handling of enemies (#define USE_OLD_ENEMS) will no longer be supported. It will still be there, but it is likely that there will be many things that are obsolute. It should not be used.

TODO: Write an application that changes a format.ene file Old to the new format, in case you want to do some other rehash of An old game of the Churrera.

This release integrates some improvements in scripting, such as strings Of decorations (see below) or the sword of Sir Ababol as type of HITTER.

There is no game with this, but the Espadewr demo.

Decorations

Basically it is to do the engine / extraprints.h but from the script. Until now we decorated the screens with extra tiles putting strings Of SET TILE (x, y) = t. That's a pain in writing and maintaining and Also occupied 4 bytes per tile.

Now we can define tiles of tiles in which the whole set of tiles Tiles will occupy 2 * n + 2, where n is the number of tiles. A good improvement Compared to the original, which occupied 4 * n bytes.

The theme is like this: simply enter this in the ENTERING_SCREEN (or in Any site that supports commands: this serves to change good Bits of the screen from the script and can come great to implement Puzzles that modify the stage:

```
IF TRUE
THEN
        DECORATIONS
                X, y, t
                X, y, t
                ...
        END
END
```

Each line x, y, t defines the position and the number of an extra tile of decor. You can put all you want.

This serves to save a lot of memory and have well-decorated screens. Generally you do NOT need the 48 tiles of a tileset extended to the time. There is always a group of tiles that repeats more. The idea is Set that tileset as the main tileset (0-15) and place the tileset Others as decorations.

In addition, the new map2bin is responsible for detecting them automatically and Generate the necessary script lines.

REENTER, REDRAW, REHASH

REDRAW has been redesigned to work better and faster. Redraw Again the screen from the buffer, and this includes everything we have Placed before with SET TILE (x, y) = t.

REHASH re-enters the screen to, among other things, initialize the enemies. It is necessary if an enemy is changed from linear type to type Flying, since you have to initialize some variables.

REENTER, as always: it's REDRAW + REHASH.

Change enemies and backup enemies

In addition to turning them off and on (Ninjajar!), We can now change the type Of the enemies. The type contains whether or not it fires, what pattern of movement Follow, and what sprite he has.

```
ENEMY n TYPE t
```

Sets type "t" for enemy "n".

However, it must be borne in mind that this is destructive: if we change the Type of an enemy, the original type will be lost forever.

In multi-level games this is no problem because with decompressing New level let's get ready.

For single-level games, we have introduced an "enemy backup" that We can activate from config.h using:

```
#define ENEMY_BACKUP
```

The backup takes up 3 bytes per screen and saves the original type of all Enemies.

From scripting, we can do:

ENEMIES RESTORE
 It restores to its original values the enemies of the current screen.

ENEMIES RESTORE ALL
Restore the type of ALL level enemies.

If you only need to restore them at the start of each game, you can go from using ENEMIES RESTORE ALL in the script and activate the RESTORE_ON_INIT directive in Config.h

Sword

We have added the sword of Sir Ababol 2 as a new type of hitter (interna-mind). For the moment, it is not possible to have a fist and a sword in it Game, although we will do something to make it possible in the future.

```
#define PLAYER_HAZ_SWORD
```

The screen graphic is defined where all others: in extrasprites.h

More things

We have changed everything but now my memory fails... To see, things misce-Laneas that I remember:

```
#define PLAYER_WRAP_AROUND
```

Use it only if #define PLAYER_CANNOT_FLICK_SCREEN is set. With Both active, in addition to not being able to leave the screen, if we approach a Lateral end we will leave on the contrary.

MT Engine MK2 v 0.89 (Nicanor Edition)

This version corresponds to the game "Nicanor the Profaner". Includes one New version of msc3, 3.92

Easier Scripting with Alias

As of version 3.92 of msc3, you can skip the word FLAG if you use alias. That is, the compiler will accept

```
SET $ KEY = 1
```

and also

```
IF $ KEY = 1
```

Safe Spot

If you define #define DIE_AND_RESPAWN in config.h, the player, upon dying, goes to Reappear in the "last safe point". If DIE_AND_RESPAWN is active, the Engine saves the position (screen, x, y) each time we Tile nontransferable (other than a floating object).

We can control the definition of the "safe spot" from our Script. If we decide to do this (for example, to define a "checkpoint" Manually), it is convenient to deactivate that the engine stores the safe Spot automatically with:

```
#define DISABLE_AUTO_SAFE_SPOT
```

Whether or not we do this, we can define the safe spot from the script with these two commands:

SET SAFE HERE	Sets the "safe spot" to the current position of the player.
SET SAFE n, x, y	Sets the "safe spot" to screen n in the Coordinates (of tile) (x, y).