

PETSCII ROBOTS in MMBASIC explained

Introduction

PETSCII ROBOTS is a game, originally developed by David Murray (aka “The 8bit Guy” on social media and Youtube) for the Commodore PET in 2021. Over the years several people have taken his code, and ported it to several retro computers. Some of these ports are from David’s hand, some are created by individuals or communities.

This document describes how the PETSCII ROBOTS computer game is implemented in MMBasic, a basic interpreter written by Geoff Graham and Peter Mather. This specific port is running on a Raspberry Pi Pico module with minimal additional hardware to transform it into a stand alone computer called PicoMiteVGA.

picture

The main driver behind this port was the fact that the game is a unique combination of puzzles and shooter. There are sufficient elements in the game to keep it exciting for hours and hours, clearly shown by these young beta testers.



Game essentials

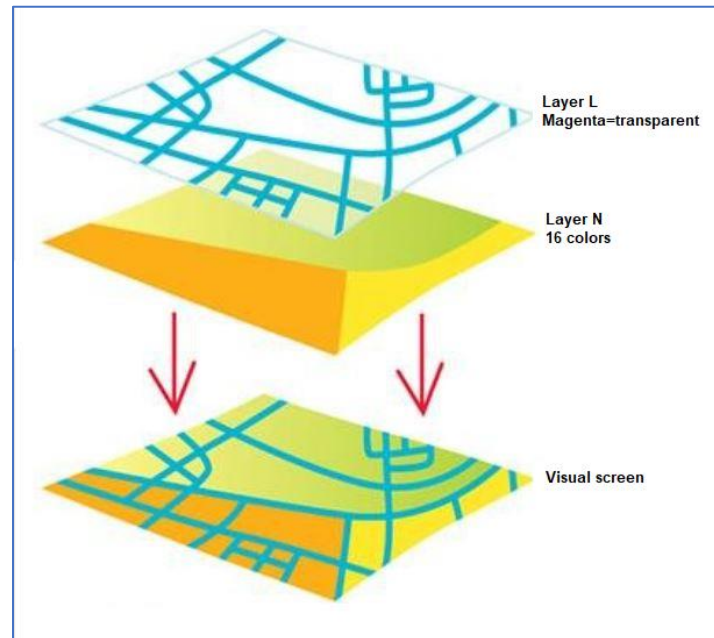
The difference between all known implementations and this version of PETSCII ROBOTS is the language. Other versions are written in 6502 assembler, or in C++, and compiled to machine language. This version is running on interpreted basic. It is written from scratch, not of copy of existing code.

To achieve resemblance with the original game, original graphical elements (tiles, sprites), as well as original music and sound effects are used and adapted to the PicoMite computer. Music re-sampling and 16 color graphics adaptations where made to fit it all on the PicoMite. The original game maps for the 14 levels are used.

This game works on a variety of PicoMite platforms, such as the Game*Mite (a “Gameboy” alike handheld), the PicoGameVGA (where it can be played using NES/SNES Controllers). Even a Wii-classic controller can be used. This description is based on the PicoMiteVGA with keyboard control.

The graphics system

The PicoMiteVGA video mode 2 is used with 320x240 resolution in 16 colors. Memory allowed for 2 layers.



Layer N: the normal video layer, 320x240 with 16 colors

In PicoMiteVGA the N layer is the default screen layer. No specific commands needed.

On this layer the tiles are shown. Tiles are the graphical elements that build the world where the player moves. Walls, floor, chairs, grass and trees. These are all part of the Tile Set, and are shown on layer N.

On this layer there are 5 elements

1. The frame: a static element that separates the playfield from player information.
2. Playfield static elements: these are walls, floors
3. Playfield uncontrolled animated elements, such as rotating fans, flowing water, flags
4. Playfield controlled animations, such as doors, elevators, raft
5. Player information: inventory, weapons, health, information screen

The playfield is a 11x7 size “window” on the 128x64 world map, with the player in central position.

In MAP mode, the playfield is replaced by a condensed map.

Layer L: on top of N, with transparency color “magenta”, 15 other colors in 320x240.

In PicoMiteVGA the L layer must be enabled, and requires 38kbyte free RAM, with command:

FRAMEBUFFER LAYER

This layer is used for the sprites. Default transparency color is black. In this game we needed the black color for sprites, therefore we changed the transparent color to magenta (color 9) with:

FRAMEBUFFER LAYER 9

On this layer sprites are shown. Player and robots are sprites as well as animations like explosions. Since they are overlayed over the world in layer N, a player or robot can walk on grass, or floor, without changing the sprite. The hoverbot can even hover over streaming water.

Note: there are a few tiles, that are used as sprites.

Despite the fact that MMBasic supports a third layer (F) that allows glitch free drawing of graphics, that layer is not used to conserve memory required for the game data. Resulting in occasional flickering of sprites when drawn.

Graphical elements

The graphical elements (tiles, sprites) are small bitmaps (BMP files) that are packed together in one binary file called “lib/pet_lib23.bin”. They are stored in MMBasic sprite format, consisting of a header that contains the sprite size (i.e. 24,0,24 for a 24x24 sprite or tile), and the binary data (4bit/pixel) in either plain format, or RLE compressed format.

The source material (16 bit rendered BMP’s) is evaluated sprite for sprite if compression is effective, and then the tile is added to the binary file, either compressed or plain.

To retrieve data from the binary file, and index file “lib/flash_index.txt” contains the start address of each tile/sprite in the binary file.

When the game is loading, the “lib/pet_lib23.bin” file is copied from SD card into PicoMite flash memory. PicoMite is equipped with 3 flash slots, each 100kbyte in size, that can be used for programs, or for (binary) data. The advantage of the flash slots is that data can be accessed very fast, much faster than SD card. In below command flash slot #3 is used, and content in it is overwritten (o).

```
FLASH DISK LOAD 3,"lib/pet_lib23.bin",o
```

Next to these graphical elements there are the graphical templates. Like the game frame, and start- and end screens. These are 320x240 size, so they fill the whole screen. The templates are stored as 16 color BMP’s on SD card.

Graphical templates

These are loaded from SD card from the folder "images/" when the game starts and ends. There is no need for frequent access, therefore loading from SD card is simple and efficient.

Start screen



Game screen



End screen



These elements are loaded onto the N layer by using

```
LOAD IMAGE fnam$
```

When the difficulty level is changes, small graphical elements are pasted over the start screen bmp, as visual candy, to show the robot being angry or nice.



These elements are loaded onto the N layer at location (234,85) by using

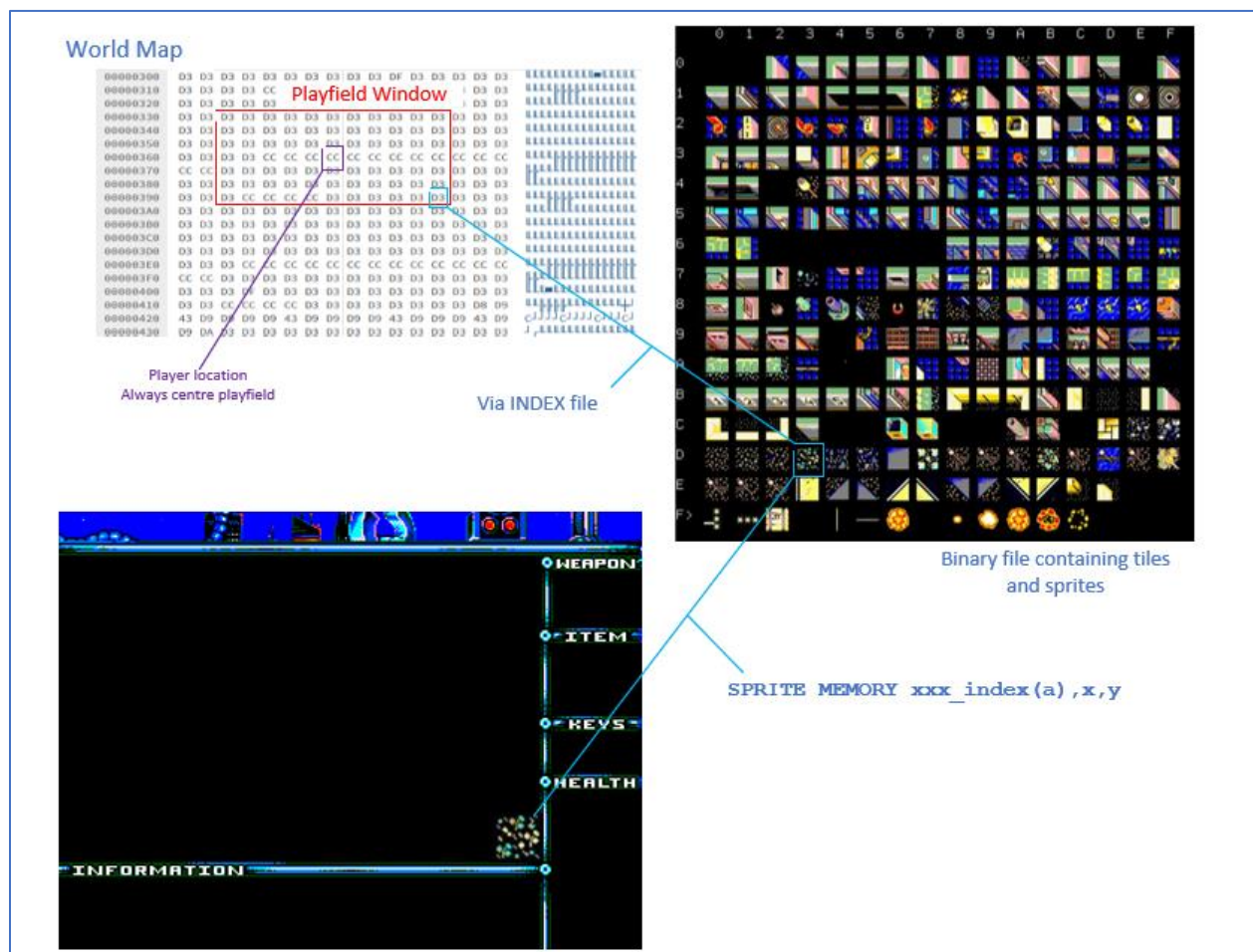
LOAD IMAGE fname\$,234,85

Tiles

All graphical elements in the binary flash file are explained in this section.

Playfield static elements

These elements are 24x24 pixels and use all 16 colors. They are copied from the binary file directly into graphical layer N. The world map, located in the "data/" folder, is used to place the correct tile to the correct location. Since the player is always in the centre of the playfield, it's coordinates (xp,yp) in the world map are plotted in the centre of the playfield (xs,ys).



The playfield size is 11(horizontal) x 7(vertical) tiles. The whole playfield is redrawn every 100ms. A redraw of layer N consumes roughly 10ms of CPU time.

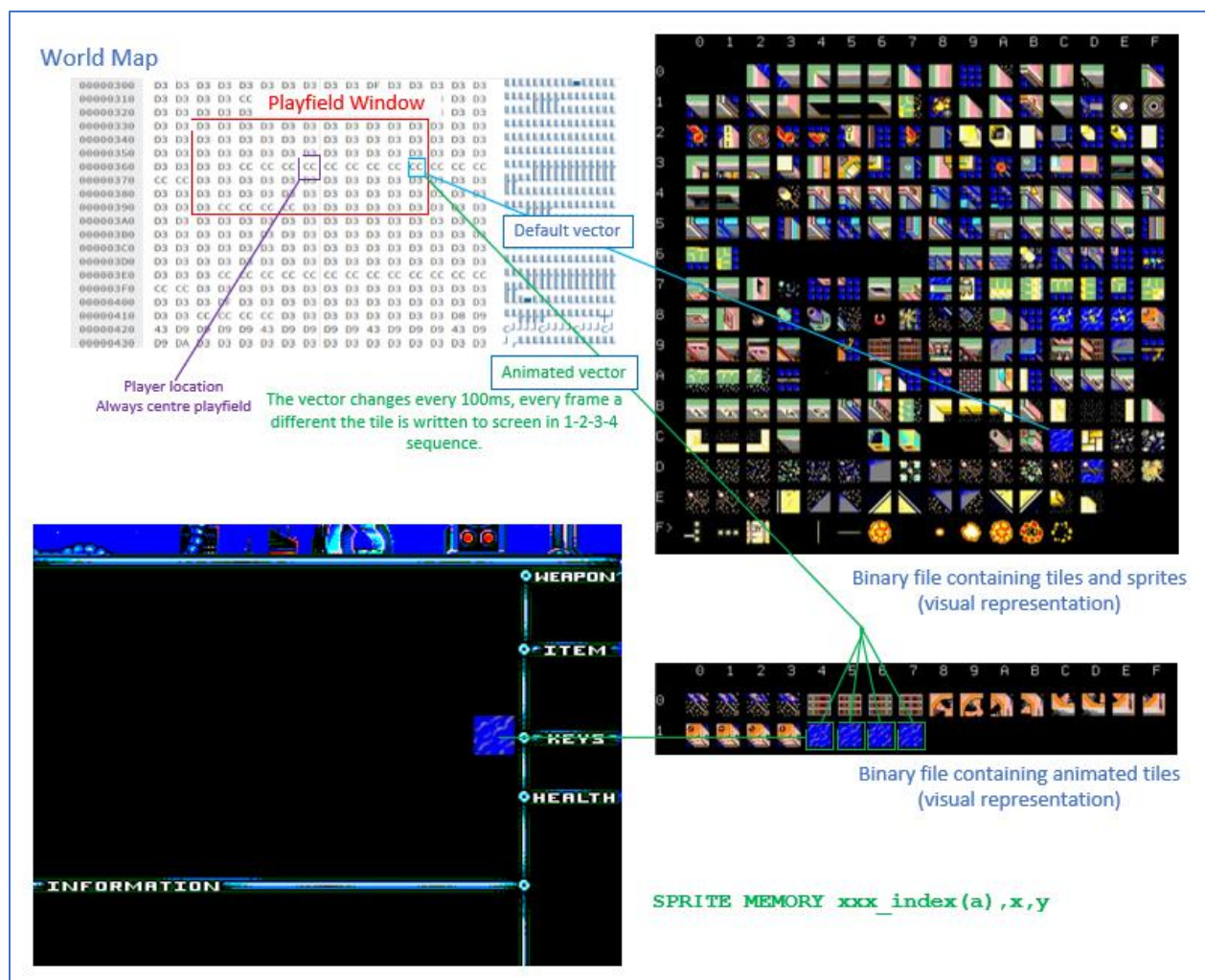

```
BLIT MEMORY index_array(sprite_number),x,y      `for picomite
```

Playfield uncontrolled animated elements

Uncontrolled elements on the playfield are just eye candy. Visually appealing, but non functional. When the playfield is drawn from the world map, some tiles (i.e. water, flag, AC's) are drawn in a never ending sequence of animated tiles. This is done by adapting the index file for that particular tile.

Instead of pointing to the static tile, the vector points to one of the animations of that tile. Since these animations are in the same binary, updating the pointer in the index file to a new tile is sufficient.

Since the tile on the world map itself does not change, also the tile capabilities (attributes) do not change. i.e. water has the ability that you cannot walk on it. And this capability is connected to the tile number. The tile number does not change, only the vector pointing to the displayed graphical element.

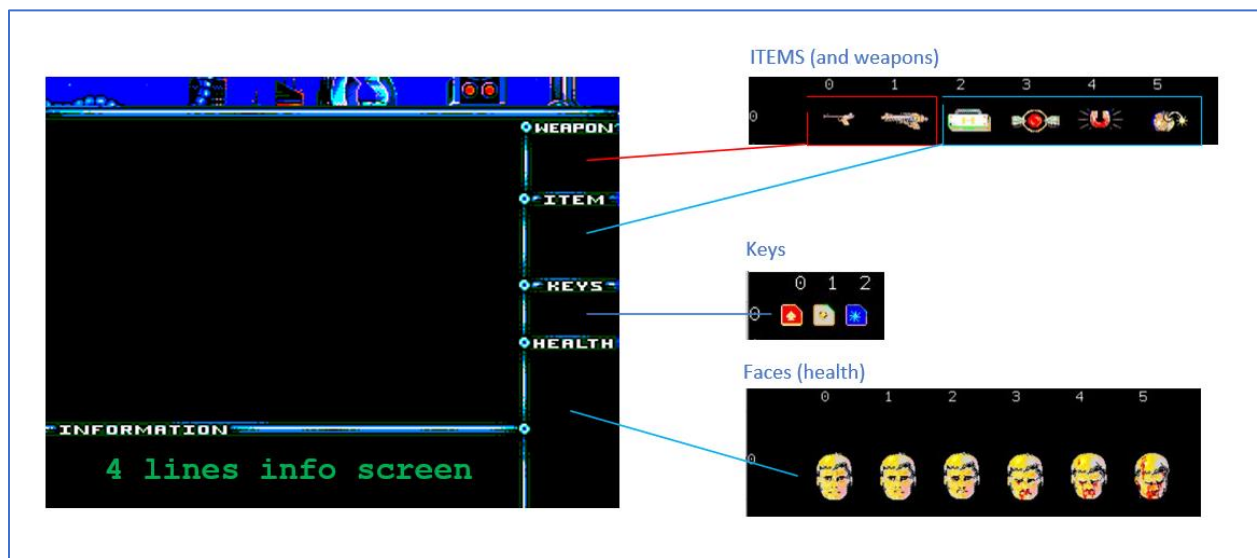


Playfield controlled animated elements

The playfield contains graphical elements that change capabilities (and visual appearance) as result from player actions. Elements such as doors. You cannot walk through a closed door, but you can when it is open. Or you can blow up a canister, search a box, move an object. Changes like these are performed by replacing the tile number in the world map. With the changed tile number, also the attributes (capabilities) can change. See chapter “Tile Attributes”

Player information

Player information in the game is stored in integer values (i.e. pl_it, pl_wp, pl_ky) and UNIT attributes (the player is UNIT 0) and the information is visualized by showing graphical elements on layer N.



```
SPRITE MEMORY index_array(sprite_number),x,y      `for picomite VGA
```

Tile attributes

Each of the 256 tiles has capabilities. These capabilities are listed in the file “amiga.tiles”. Associated to a tile are 8 bits (1 byte) that determine what the player and robots can do at that location.

Attribute bytes

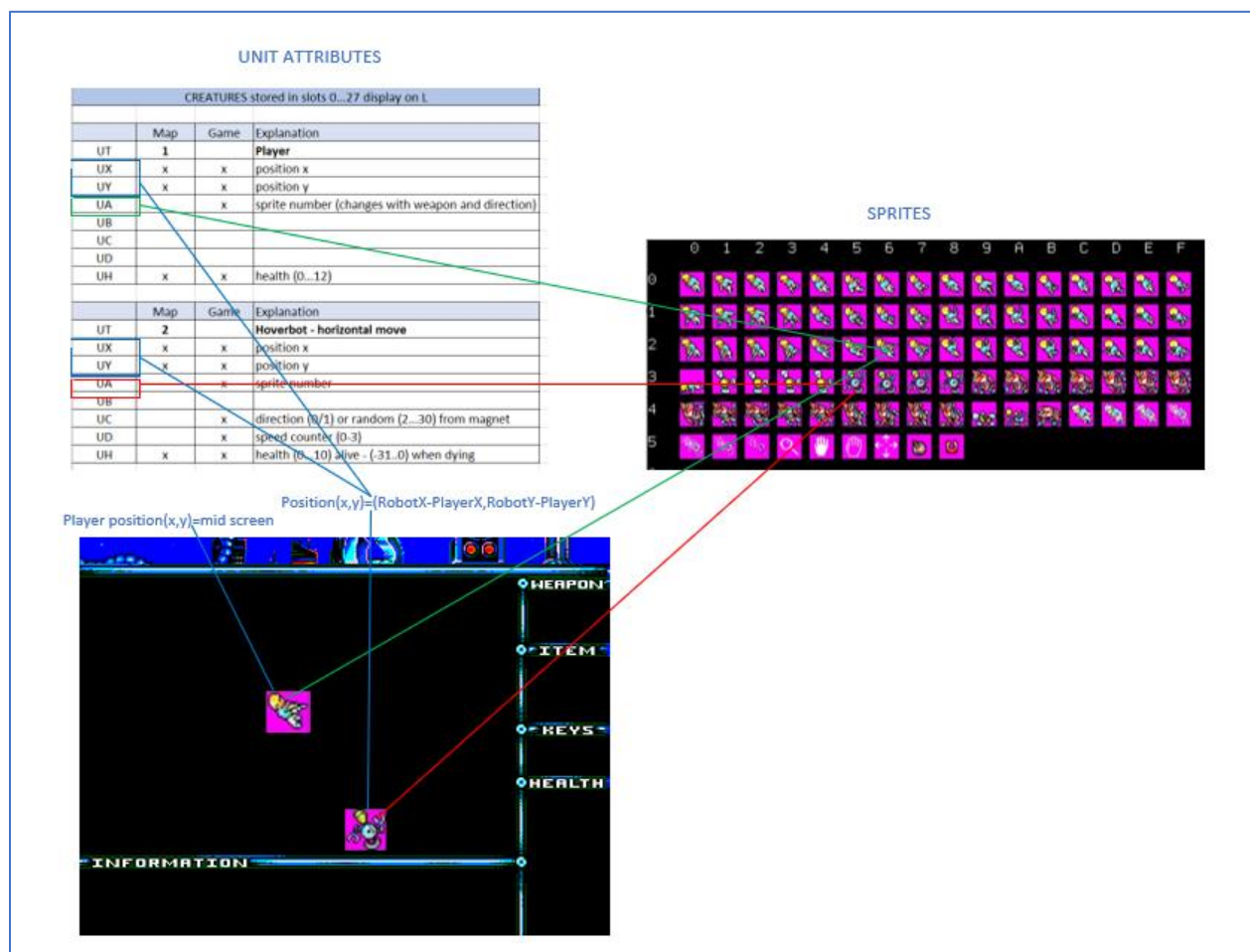
Bit	Function
0	Player/robot can walk over this tile (i.e. grass/floor)
1	(Player)/robot can hover over this tile (i.e. water)
2	This tile can be moved (i.e. a chair)
3	This tile can be damaged (i.e. a bridge, a plant, a PI sign)

- 4 You can see through this tile (i.e. a window, a plant)
- 5 This tile can be pushed (not used here)
- 6 This tile contain a hidden item (not used here, see UNITS)

Sprites

Graphical elements discussed in previous chapter reside in the normal screen (N) layer. Overlaid is the L layer that is used for sprites. The L-layer uses RGB(MAGENTA) as transparent color.

The player, and the robots, are shown on the L layer to allow the player/robots to walk on any surface (grass, floor, stairs). Each game loop the L layer is erased, and completely redrawn with all sprites at the new location in respect to the player location. All sprite locations are taken from the UNIT attributes.



Each game loop (100ms) every single UNIT coordinates are checked if they are within the visual playfield, and when they are, the associated sprites is shown on the L layer.

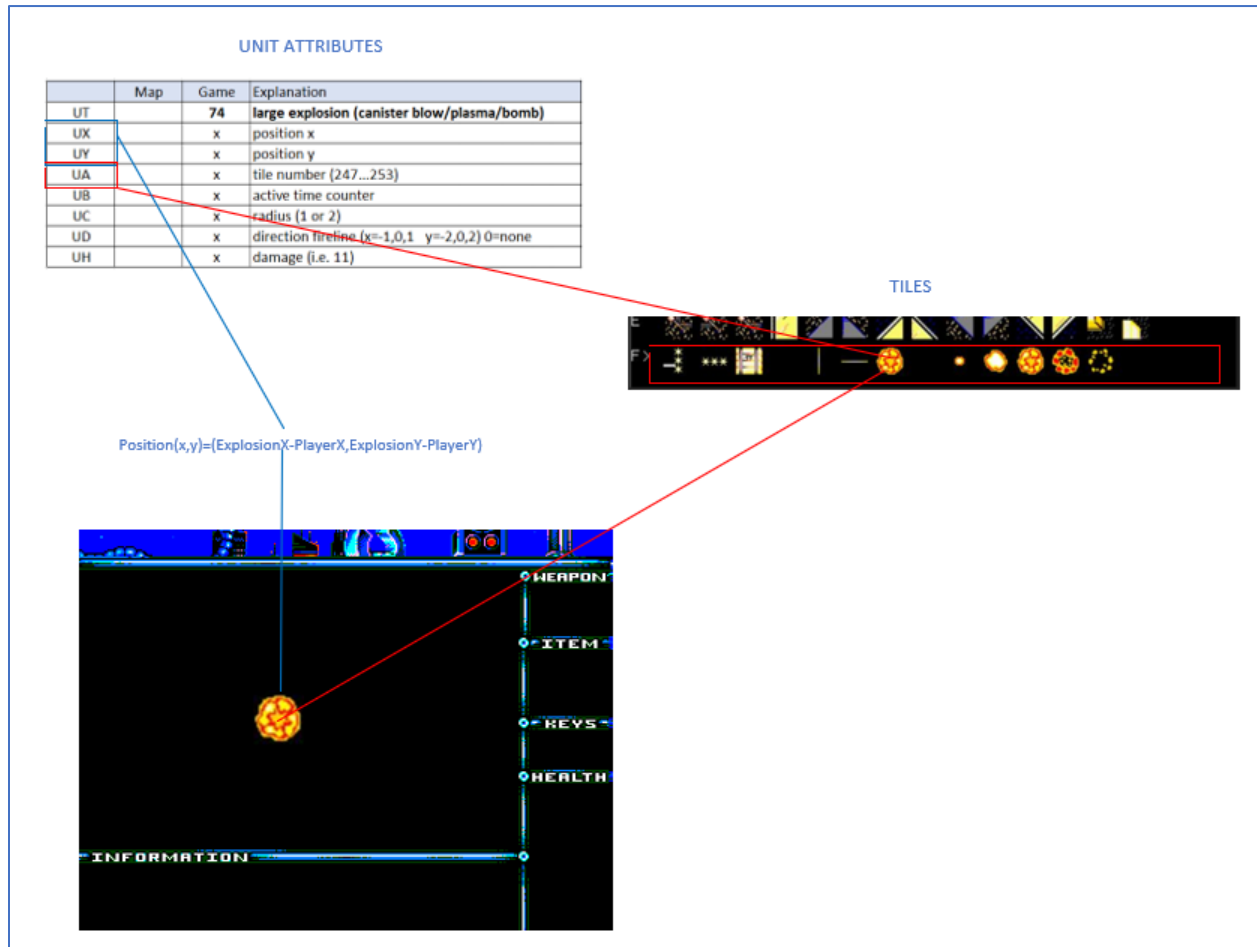
SPRITE MEMORY `index_array(sprite_number),x,y` ``for picomite VGA`

Apart from the robots and player, there are also animations that are printed on layer L, sometimes even on top of the robots or player: the explosions. These are taken from the tile set (there is no magenta background) but are used on the L layer. To make this possible the tile background can not be copied (they would be opaque). Tiles have a background color that is black (color 0).

These graphical elements can only use 14 colors. (not black, not magenta).

SPRITE MEMORY `index_array(tile_number),x,y,0` 'color 0 not copied

Each game loop the tile number is adapted, as to create a live animation.



How to create a the compressed binary from 300+ sprite files.

PicoMite MMBasic has program memory to store it's MMBasic program. Additional to that it has 3 flash memory slots (100kbyte size) that can be used to store programs that can be "chained" together. One of these flash slots (flash slot #3) can be converted to a "library". A library is an extension to the program memory. The library can contain valid MMBasic commands and functions, but it can also contain CSUB's. CSUB's are compiled C-code (read: ARM instructions) that can be executed by the MMBasic program. The CSUB's contain in essence binary data.

In PETSCII ROBOTS we use the CSUB's, not to store ARM executable binary code, but to store compressed sprite binary code. Only these are cloaked as CSUB's. We use the library function to create a 85kbyte compressed binary, hidden as a CSUB. Once we have created the compressed binary, we store it on SD card.

Then delete the library again (it has fulfilled it's task).

The compressed binary file can be loaded into one of the flash slots for use by the MMBasic program.

Create compressed library file from sprites (you need development version "petrobot24" since it contains all the sprite files and tools. These are stripped from later versions to compact size). The procedure is:

In each of below folders

/sprites/health
/sprites/keys
/sprites/items
/sprites/spr-files
/tiles/tile0_3f
/tiles/tile40_7f
/tiles/tile80_bf
/tiles/tilec0_ff
/tiles/tla

RUN "spr2csub2.bas"

This generates index files and compressed CSUB files.

Now we use the PicoMite library to combine the CSUB's into one binary file.

if needed **LIBRARY DELETE**
in folder /tiles
LOAD "tile0_csub.bas"
LIBRARY SAVE
LOAD "tile1_csub.bas"
LIBRARY SAVE
LOAD "tile2_csub.bas"
LIBRARY SAVE
LOAD "tile3_csub.bas"
LIBRARY SAVE
LOAD "tla_csub.bas"
LIBRARY SAVE

in folder /sprites
LOAD "hlt_csub.bas"
LIBRARY SAVE
LOAD "key_csub.bas"
LIBRARY SAVE
LOAD "item_csub.bas"
LIBRARY SAVE
LOAD "spr_csub.bas"

LIBRARY SAVE

Now we save the created library to disk

```
LIBRARY DISK SAVE "lib/pet_lib23.bin"
```

Now we create an index file for the newly created binary file

In folder /tools

```
RUN "make_index.bas"
```

This generates a relative master index file /lib/flash_index.txt

We do not need the library anymore, so delete it.

LIBRARY DELETE

Result: all sprites and tiles are compressed to

/lib/pet_lib23.bin

And the relative pointers to the tiles are in

/lib/flash_index.txt

This binary can be loaded in a flash slot (i.e. slot 3)

```
FLASH DISK LOAD 3,"lib/pet_lib23.bin",o
```

And the relative index can be made an absolute index by adding an offset equal to **MM.INFO(flash address 3)** . This conversion is performed by the robots.bas program.

Now we have absolute pointers to memory addresses that contain valid compressed sprite info (size info + binary bitmap)

UNITS

There are 64 units in this game. A UNIT can be the player, a robot, but also a door, or a hidden item.

David Murray has pre-defined the 64 units into following groups.

UNIT number	What is it
0	The player
1-27	Up to 26 robots
28-31	Visual Effects (explosions). These UNITS are used as a dynamic FIFO
32-47	doors, raft, elevator
48-63	Hidden items (weapons, items)

Each of these units has 8 attributes (1 byte each) to determine it's functions in the game. Positions are positions on the world map ($0 < UX(n) < 127$, $0 < UY(n) < 63$).

Attribute	Function
UT(n)	Unit type (1=player, 2=hoverbot, 10=door etc...)
UX(n)	Unit position in the map, X coordinate
UY(n)	Unit position in the map, Y coordinate
UA(n)	

UB(n)	Function depends on Unit type, for a door this involves what key is needed to open the door, for the raft what direction it moves. These are also used dynamically (i.e. what sprite number to is visual on the L-layer, how fast a player walks).
UC(n)	
UD(n)	
UH(n)	Unit Health (i.e. Robot, Player)

There is an overview of the way these attributes are used in the game in appendix A.

Appendix A

UNIT attributes are summarized in this overview

