The Making of Rem Pac

for XGameStation Micro Edition

by Rémi Veilleux

Preface

You want to know how to create a game for the mysterious yet wonderful XGameStation micro edition, this tutorial is for you. Rem-Pac is a pac-man clone written in SX52 assembly language, the CPU of the XGS ME. It was my first game for this system, and also the first game I wrote completely in assembly. It was also the most complex assembly challenge I've ever faced.

Obviously, in order to understand this document, you'll need a solid knowledge of XGS and assembly language. I greatly recommend you read all available documents and infos, and preferably have compiled and tried a couple of simpler demo programs. Of course, nothing beats writing and modifying a real game, so I won't stop you to read on and play with the code like crazy! Isn't it the whole point? Great challenge will lead you to great satisfaction and learning, and most of all, tons of fun!

Fun facts

- Rem-pac is made of over 2400 lines of code, a total of 69,137bytes
- When assembled, it fills almost entirely the 4096-word available ROM space
- It uses about 170 bytes of register RAM (out of a total 256 bytes)
- It doesn't use any external RAM
- It uses a 160x200 image resolution slightly stretched on the TV to a simulated 213x200 centered non-interlaced display (60 Hz)
- It was the first tile-based game on the XGS

Files

Rem-pac is made of 4 source files, which are:

- general_define.src: Small file with generic XGS defines and definition of 'global registers'
- general_macro.src: Useful general purpose macros
- general_function.src: General purpose functions (prefered over macros to reduce ROM space)
- rem_pac_01.src: The main source file, including bitmap graphics and all

Note of warning concerning source files:
Some comments in the code are not accurate because code has been updated and revised after and comments were not always updated accordingly. Sorry about that.

Chapter 1: Get on with it!

Alright, let's begin. First, here are the main aspect of the game:

- RAM address space repartition
- ROM address space repartition
- Video 'kernel'
- Main game logic and I/O
- Sound

The first 4 items are of extreme importance in order to succesfully write a XGS game. Remember: RAM and ROM space are so limited that they must be carefully assigned and respected. The other thing that soon become limitating and crucial is CPU clock-cycle usage, particularily in the video rendering part, which we call the 'video kernel'. Then of course you have the main game logic, which is not always as trivial as one could think, for various reason that will discuss in more detail. Last come sound, it is not considered a crucial part since a game could exist and work without it, but you might still need to consider it when you write a game, at least in order to leave some clock-cycle and space to add it later on.

RAM address

The XGS RAM is divided in 16 x 16-byte banks. RAM bank are numbered $00, $10, $20, ... $F0.
The RAM bank at $00 is reserved by the system and is called 'Global registers' since it can be accessed at any time, so we'll leave it alone for now.
By the way, let's make something clear: on the SX52, registers and variables mean the same thing.
Rem-Pac does not use RAM bank $10, so let's start at $20.

RAM bank	**Usage**
$20	General drawing, sound and backup variables.
$30	Pac and ghosts animation and screen position
$40	Score, highscore and pellet counter
$50	Pac and ghost move processing variables
$60-$87	Temporary scanline buffer, used by the video kernel to display one line of video
$90-$B5	Current level pellet status (1 bit for each pellet)

The global registers

As said before, global registers are accessible at any time so they are very precious, useful and fast. I have defined them in **general_define.src** at the bottom of the file:

```
org $0A                 ; Global register bank (8 bytes)
counter     ds      1   ; general counter, used in the DELAY macro
counter2    ds      1   ; general counter, used in the DELAY macro
counter3    ds      1   ; general counter, used in scanline overscan
counter4    ds      1   ; general counter, free
joystick2   ds      1   ; output status of joystick #2
joystick1   ds      1   ; output status of joystick #1
```

org is a keyword that tells the assembler that the next RAM variable will be placed at $0A, which is inside bank $00. If you look in your XGS documentation, you'll see that the SX52 has 8 free general purpose global registers, located from $0A to $0F. So here I basically assign each of these registers with a variable name. Please note that these variable name does absolutely not

reflect the usage of these registers in rem-pac. Since the program evolved and became more complex, these precious variables were used are reused for all purpose. They could be more adequately renamed to reg1, reg2, reg3, etc or something similar.

All other variables, starting at $20 and up, are kind of self-explanatory and slightly commented in the code itself. For example, **level_color** contains the color used to draw the current level. It starts with the value 231. Speaking of colors, if you want to see what '231' means, jump to the color chart in the appendix section.

ROM address

The XGS has 4096 word of address space. Note that we don't use the term 'byte' when talking about ROM since a byte is 8-bit, and the ROM memory of the SX52 is 12-bit wide. ROM address space spans from $000 to $FFF, giving a maximum total of 4096 instructions. This space is subdivided in page that are 512 word long, giving us a big total of 8 ROM pages, which are named $000,$200,$400,$600,$800,$A00,$C00 and $E00.
Why should we care about ROM? There is a couple of reason / limitation that we must know:

- You can only **call** functions that are located in the first 256 word of a page (also known as the lower half-page)
- When doing time critical stuff, staying 'inside' the same ROM page is faster and takes less instructions
- When accessing ROM graphic data, you need to know exactly where they are located.

So code ROM page are defined as follow in Rem-Pac:

Page	**Usage**
$000	Startup code and scanline rendering code
$200	Remaining rendering code and game logic
$400	Sprite position and level rendering
$600	Level rendering, score display
$800	Ghost 'AI'
$A00	Sprite rendering and sound

Now about ROM asset, they are located at these addresses:
Note: Since ROM graphic are not 'called' or jumped in, they are not restricted to page starting address.
Also of note, the SX52 is a 8-bit CPU. Since a ROM page is 512 words large, we often have to speak of "half-page" for ROM asset:

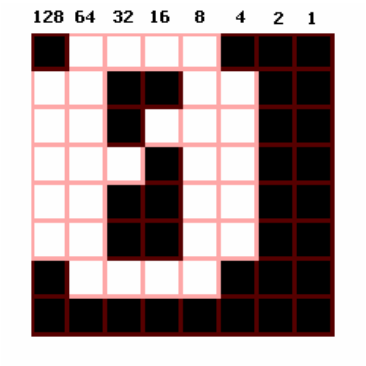
$B00	Tiles 0 to 31. Tiles are 8x8 pixel block, with 1 bit per pixel. (sprites, background tiles, numbers, etc)
$C00	Tiles 32 to 63
$D00	XGS Logo graphic data and display code. This code and graphic was taken from another XGS demo.
$E00	Tile map info for lines 0 to 11
$F00	Tile map info for lines 12 to 23

The tiles

As said above, Rem-Pac is a tile-based game using 8x8 pixel tiles with only 1 bit per pixel information. Each tile can more-or-less be assigned to any of the 150 available colors. To located them in **rem_pac_01.src**, search for **org $B00**. There is 64 tiles defined. Let's look at tile #1:

```
dw 120,204,220,236,204,204,120,000  ; tile 1 (address: 8)
```

This is what this tile looks like. Note that each column is identified with a bit. Each tile takes exactly 8 words.



So if we take for example the first row which has a value of 120 in decimal and convert it to binary, we obtain **01111000**, which in turn tells which pixel are on (1) and off (0). Repeat this 8 times to define a complete tile, and presto, you now know how to create and modify all tiles in Rem-Pac! It wasn't that hard?
You might wonder if I did enter all those number manually. Of course not. I created an image in photoshop, saved it in BMP format, then did a small Perl script to read in the BMP and convert it to this 1-bit text format ready to include in the source file. You'll find it included along with the source and BMP image if you want to look at it.

Here's the complete tileset used in Rem-Pac:

Note: Although some tiles are 'touching' each other, it doesn't matter since they are converted in separate 8x8 pixel block.
Also you want ask why is some tile colored. This is simply the way I created them, it has no influence since the convertion is done with only 1-bit per pixel (on/off) so all color information is lost anyway. Also if you look really closely (or zoom the image), you'll see that there are dark gray grid behind the tiles. This dark gray is ignored by the convertion script and was only used as alignment reference.
The exact convertion formula for each pixel is: (where Red, Green and Blue are integer values from 0 to 255)

average = (Red + Green + Blue) / 3
if average > 80 then Bit = 1 else Bit = 0

You can go on and modify all the tiles, change the number and text font, give the maze a cool new look, have fun! (I suggest you having a script or small program to do the math convertion for you).

The tilemap

The tilemap is what tells the video kernel what tile should be drawn at each location on the screen and what color they should be. To look at the tilemap, open up **rem_pac_01.src** and search for **org $E00**.

As the comment says, the tilemap is composed of a grid of 20x24 tiles. Since you remember that we have a total of 64 tiles, one word is well enough to cover them entirely, so we use 1 word per tile, giving us 20 words of data per row. But we also have to tell what color should be used to display these tiles. This is tricky since we don't have much ROM space left, so I've settled with a basic convention: Each row will use ONE color. (Of course when you try to game you can actually see several colors in the same row, but this will be explained in more detail in the video kernel section). As a simple test, start Rem-Pac on your XGS and don't press the button. If you look at the waiting screen, you'll see that there is in fact only one color defined for each row. Neat trick? (*Don't mind the colored XGS logo at the bottom, this one is displayed by a completely separate function and does not use the tile system we are describing)* Ok let's go on and take a look at the first row of tiles which is the "1UP HIGH SCORE" row:

```
        dw 255
        DEFINE_TILES 00,00,02,31,26,00,00,00,18,19,17,18,00,29,13,25,28,15,00,00 ; line 0
```

The first word is the color. So **dw 255** means this line will be displayed with the color **255**, and if you take a look at the color chart in the appendix section, you'll see that this is white. First row will be displayed in white. Good! You can already try to change this value to any other color, compile and run, and you should see the difference.

Now let's look at this **DEFINE_TILES** macro. This is a helper macro that I wrote to simplify the code. It is defined in **general_macro.src** and looks like this:

```
DEFINE_TILES MACRO
        REPT \0
              dw \%*8 + $B00
        ENDR
ENDM
```

What does this mean? Well you should make sure you understand assembler macros since there are **very** useful are used everywhere in the code. But let's summarize what it does in plain english:

For each parameters passed
    *Output a value which is **parameter * 8 + $B00***

Let's explain this. The values (or parameters) you see like 00,00,02,31,26,etc actually represent a tile number. For example if we look at the first 5 values:

00 = Empty space
00 = Empty space
02 = Character '1'
31 = Character 'U'
26 = Character 'P'

You can figure out that this is what will display **1UP** on the first row.

But since we want to help the video kernel doing its calculation the fastest possible, we mean to convert those simple 'tile number' to a more optimal and direct ROM address. This is where the **DEFINE_TILES** macro comes in. It takes a tile number and convert it to the ROM address of this

same tile. Since you remember that our tile takes 8 words each, and the first tile started at address **$B00**, the math is simple.

Note: You might remember that tiles from 0 to 31 are located in page $B00, but tiles 32 to 63 are in page $C00. So what about them? Don't worry: The math is already taking care of them: Let's take tile 32 for example. Its address will be calculated like this:
*32 * 8 + $B00*
which gives 256 + $B00 = $C00. Simple as that, no special case here.

This is good but wait a minute you'll say. If the color of a row is specified here in ROM, how can we make the whole level change color? You are right, there is a special color trick. Let's look at the first row of level data, row #3:

```
dw level_color+256
DEFINE_TILES 37,38,38,38,38,40,38,38,38,38,38,38,38,40,38,38,38,38,39 ; line 3
```

As you can see, the color word is not defined as a simple color number. It represents a variable name, plus a special bit value of 256 to tell the system that this row color should be taken on-the-fly from a RAM location, allowing us to change it whenever we please (in this case, when a level is completed, I'll set a new color in the variable **level_color**). This variable is defined in **org $20** (RAM bank $20) at the top of **rem_pac_01.src**.

Last thing: Since we require 1 word for color + 20 words for a row, this gives us 21 words of data per row. One half-page of ROM contains 256 words: this allows us to fit in 12 complete rows. Since our screen is composed of 24 rows, we'll have to use two half-pages: The first 12 rows are defined at **$E00**, the remaining 12 at **$F00**.

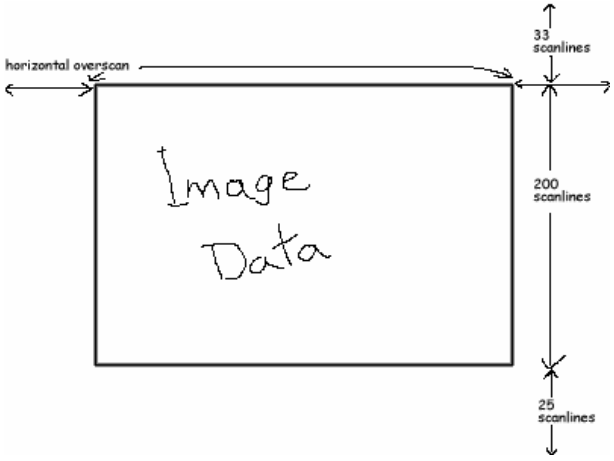
Chapter 2: The video kernel

The real challenge starts here. Now that we have all our tiles and tilemap defined and ready to use, we need to shoot it to the TV. This is not a simple task. The display code represent more than half of the total game code. For additional difficulty, all your code path must absolutely take an exact and fixed amount of clock cycle, or else the TV will display a wavy/poor signal or lose the syncronisation. This is why every line of code you'll find in this big portion will have a commented number displayed on its left: this represents the number of clock cycle this instruction takes. You'll often see DELAY macros to compensate where additional clock cycles need to be spent.

Let's begin with a simple overview of a typical rendered frame:

- Step 1: Prepare and send the horizontal blank and color burst signal to the TV
- Send one horizontal scanline of pixel data
- Wait horizontal overscan
- Loop to Step 1 for 192 lines
- Step 2: Display XGS Logo, 8 lines
- Step 3: Leave 25 empty lines for bottom overscan (black)
- Step 4: Image is completed, start counter to wait for vertical blank timer. Game logic will be performed here.
- Step 5: Delay of vertical blank timer elapsed, we need to start a new frame
- Step 6: Leave 33 empty lines for top overscan (black)

Of course, things are not as easy as they look, and there is a lot of critical timing delays that are calculated and spreaded all over the code. Let's show a graphical representation of a TV frame:



However, due to the fact that we have a lot of things to do for each scanline, we definitely need all available clock cycles. The right horizontal overscan would be entirely wasted time if we didn't use it, so in reality, we are going to start our preparation operation on the *previous scanline horizontal overscan!* Right!

Let's see this visually:

Blue: Right Horizontal overscan: Here we have 504 clocks available, we will prepare the scanline buffer.
Before the next scanline: We need to send the necessary video signal to the TV (including color burst, etc). This is done with the **PREPARE_VIDEO_HORIZ** function.
TV has now switch to the next scanline and is ready to receive the image signal.
Green: Left horizontal overscan: We have another 504 clocks, we'll use it to display the score, display sprites and hide eaten pellets.
Red: The image data for this scanline. In this step we simply transfer our fully constructed scanline buffer, one pixel at a time to the TV.
Blue: We are done for this scanline, loop back to prepare the next one.

Now if you want to look at the code that handles these steps, open up **rem_pac_01.src** and jump to the label **Begin_Raster**. I won't go over all timing since there is a lot of documentation and simple sample programs to explain NTSC delays and timing. Let's just cover the basic: A complete NTSC scanline lasts exactly 52.6 us. In clock-cycle term on the XGS (assuming we run at full speed 80Mhz), this means exactly 4208 clock cycles. In Rem-Pac, I settled for a 504 clock horizontal overscan on each side of the screen, leaving a nice and round 3200 clock cycles of image display. Since the game uses 20 tiles wide and each tile have 8 pixel giving 160 pixels of data, we get (3200 clock / 160 pixel) = 20 clock per pixel. This is a relatively big number. However when I looked at the result, it gave a wide but low-res image feeling, similar to an Atari 2600. This is why I've added an additional 400 clock cycle delay on each side on the display, and therefore reduced each pixel to 15 clocks. This still gives us the same total clock (400 + 400 + 15*160 = 3200), but the image will be centered and appear to be of higher horizontal resolution.
The part of code that takes care of the real image drawing start at line 307 in the code and ends at line 338.

Here what this loop does in pseudo-code:

```
For each tile (20)
      For each pixel/bit in this tile (8)
            If bit is set, output desired color to TV
            If bit is not set, output black to TV
      Loop pixel/bit
Loop tile
```

We now know that each pixel has to take exactly 15 clock cycle. Let's have a look at this section in detail. I have removed the in-code comment to better show the timing. Each instruction takes a specific number of clock cycle, you can look them up in Quick Reference section 3.3.2 of the Parallax 'Programming the SX Microcontroller'.

```
        mov counter2, #20       ; Not counted because the djnz at the end
                                of this loop already takes account of
                                this.
draw_next_tile
        inc FSR                 ; 1
        mov counter4, IND       ; 2
        dec FSR                 ; 1
                                ; 1+2+1 = Total of 4 clocks spent yet

        REPT 8
            mov W, #BLACK       ; 1
            snb IND.7           ; 2
            mov W, counter4     ; Not counted because of 'snb' that
                                includes this
            mov RE, W           ; 1
            rl IND              ; 1
            IF (!(%=8))         ; Don't delay the last pixel
                iread           ; 4  Waste spare time for timing..
                jmp $+1         ; 3  Waste spare time for timing..
                jmp $+1         ; 3  Waste spare time for timing..
            ENDIF
; This REPT macro will take exactly 1+2+1+1+4+3+3 = 15 clocks for each
; of the 7 first pixels of this tile
; The last pixel will only take 1+2+1+1 = 5 clocks
        ENDR

        add FSR, #2             ; 2

        djnz counter2, draw_next_tile ; 4
; Add 2+4 = 6 and reloop to draw the next tile.
; Our last pixel which spent only 5 clocks, adding 6 here and 4 clocks
; at the top of the loop will give us 15.
```

So this is the 'heart' of the rendering portion of the video kernel. This not-too-complex loop is responsible for actually outputing all color pixels you see on the screen (excepting the special XGS logo at the bottom that is completely separated and is not using this tile-based system). But this loop alone won't do much if there was nothing else to feed it with the required data. In fact this loop assume and use a 'scanline buffer' that has been already prepared in RAM starting at address $60. This scanline buffer has the following structure:

1 Byte of pixel data
1 Byte of color

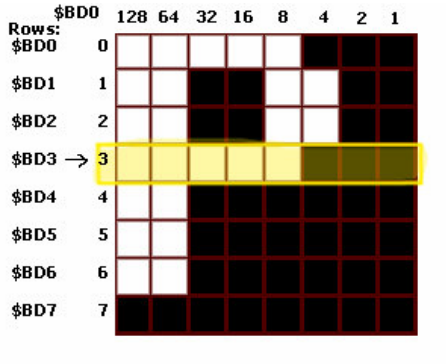
Repeat these 2 bytes 20 times. So the whole scanline buffer takes 40 bytes of RAM starting at $60 and ending at $87. The pixel data represents 8 pixels of data, so you've already guessed that it's 1 bit per pixel. The second byte tells the render loop what color should be used for those 8 pixels. This is what allows several colors to be displayed on the same line (remember that when we saw the tilemap definition, only one color was defined for the whole line! This means that something in-between is responsible for 'on-the-fly' colorisation, we will see this afterward).
Let's see this with a tangible example. We'll take the first 8 lines of display: If you remember correctly, these 8 lines correspond to the '1UP HIGHSCORE' text displayed at the top of the screen. So we know we have to loop for 8 successive scanlines and display the appropriate line of pixels from those 20 tiles (remember the screen has 20 tiles wide!). Let's see this in pseudo-code:

```
For 8 Scanlines
        For 20 Tiles
                TileAddress = Get Tile from Tilemap
                Copy 1 byte from TileAddress to Scanline Buffer
                TileColor = Get Color from Tilemap
                Copy TileColor to Scanline Buffer
        Next Tiles
Next Scanlines
```

There is a couple of things we must know that are being taken care of by this loop:

- Since a tile address byte will point to the top row of pixel, we must add the current scanline to this value to get the appropriate row. For example, to obtain the scanline #3 of the 'P' tile, we have to do this:

  'P' is tile 26
  Tile 26 starts at ROM address 26 * 8 + $B00 = $BD0
  Add 3 to get desired scanline: $BD0 + 3 = $BD3
  Data bits are 128+64+32+16+8 = 248



- The color is defined in the tilemap. If you remember or loop up in the tilemap definition, you'll see that this tile line has its color defined by `dw 255`
  This is what I call a 'ROM controlled color' since its value is fixed at determined in ROM. In this case, what our loop needs to do is simply take this '255' value and copy it 20 times (one time for each tile in the scanline buffer).
  But there is another case that I call a 'RAM controlled color'. An example would be any tile line defining the maze. The color word in the tilemap is defined like this:

  ```
  dw level_color+256
  ```

  The '256' value is a simple bit flag to tell the system that this value is not a straight ROM color (which range from 0 to 255). And 'level_color' is the RAM address of this variable, so the code will fetch the value contained by the variable and copy it 20 times, exactly like the ROM color counterpart.

  You can find this scanline preparing loop in **rem_pac_01.src** starting at line 230, ending at line 275.

Well what's next? We have described a working rendering kernel that would display colored tiles. This is great, but it doesn't move much. Guess what? We still have TONS of things to do:

- Display the current score and high score. These tiles cannot be simply defined in ROM since they need to change while the game is running
- Display the pellets and the power-pills according to which one has been eaten and make power-pills blink
- Display the remaining lives at the bottom of the screen
- Display sprites, Pac and 3 ghosts.

Whew! I won't go into much detail on all of these since they are pretty huge and complex. To look them up in the source code, search for **page_400**. You'll first find a **JMP** instruction where it is actually called in the code. Keep searching and you'll reach this function which starts on page $**400** and goes up to **$809**. This is one heck of a big function! And it's not all, you'll have to add the **sprite_draw** function that starts at **$A00** and goes to **$ABD**. And guess again? All of this must be done in a (incredibly) short amount of time, in fact slightly less than 500 clock cycles! It has to fit completely inside the left horizontal overscan delay of each scanline.
Let's try to explain how this works.

Current score and high score display

Look for **score_line** in the code to find the function that does this job. By the way, the **DRAW_SCORE** macro defined here is no longer used (you can see that all lines are commented out). This is because it was taking too much valuable ROM space, so I replaced it with a slightly slower but extremely smaller loop. Since the code that prepared the scanline buffer has already been called, the scanline buffer more-or-less contains one scanline of tiles that would look like this:

```
00000          00000
```

So the score display loop does this:

```
For each 5 characters
        Overwrite scanline buffer score tile with adequate score digit
        Overwrite scanline buffer high score tile with adequate high score digit
Next characters
```

Note that in this case, we leave the scanline buffer color data intact (which will be white). We only overwrite the tile pixel data to replace it with the real current score and high score. Remember that this overwriting takes place in the temporary scanline buffer! Nothing is overwritten in the tilemap (In fact this couldn't happen: the tilemap is defined in ROM; there is no possible way to write anything in ROM at run-time).

Pellet and power-pill display

This function uses a similar approach to the score display function, except that since pellets cannot move, we only have to set their color to white in the pellet is still present, and black (to hide it) if it has been eaten. Clever?

This is done through the usage of two macros: **SET_PELLET_OFFSET** and **SET_TILE_COLOR**. The first macro, **SET_PELLET_OFFSET**, is used to set the starting RAM address location of the pellet. Pellets starts at RAM address **$90** and needs two bytes per tile line. Since there is 19 rows of pellet in the game, pellets will occupy RAM from **$90** to **$B5**. How is this information stored? Here's the trick:

The widest rows of pellet in the game have exactly 16 pellets. This means that using 1 bit per pellet, we need two full bytes of memory to hold the status of these pellets. (0=pellet is eaten 1=pellet is still present).

However, for the sake of simplicity, all rows will take two bytes, even if some rows have only 4 pellets. This is not really a mere 'waste' of RAM memory, it's a trade-off between code effiency and memory usage.

Now that you know all of this, the pseudo-code is pretty straight-forward:

```
For up to 16 pellets
        If bit = 0 then set corresponding tile color to black
        If bit = 1 then set corresponding tile color to white
Next Pellets
```

This is what **SET_TILE_COLOR** is taking care of. If you look at the macro, you'll see that there is no real 'loop', it's a repetition (using **REPT**) . As you can imagine, doing all rows and all pellets like this is going to take a lot of ROM code space. This is true, but again, it is a trade-off between run-time effiency and ROM space. A nicely done loop doing all necessary checks would be dangerously too slow. You have to keep in mind that we are running on a very tight budget of clock cycle here, so we have to keep everything as fast as possible.

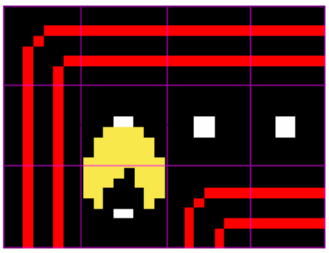
By the way for the curious, there is a small 'special case' for power-pill, you can find it by searching **line_5_common**. This '**frame_counter**' trick is what causes the power-pill to blink every 32 frames (by checking bit 4 of the variable frame_counter).

Lives display

This is a lot similar to the pellet display function. You can find it by searching for **line_23**. Since there is a maximum of 5 lives displayed at the bottom of the screen, the function will loop to 5, and set the tile color to yellow to display a life, or black to hide it.

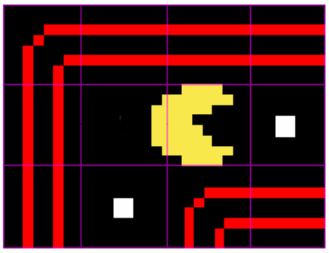
Sprites display

This is a neat and complex part of the game. The main function can be found by searching for **$A00** (function is called **sprite_draw**). The idea here is to replace a background tile at run-time with a sprite tile. Here's a closeup view of a typical tile background with a sprite stamped over it:



In this example, the sprite is 'perfectly aligned' horizontally. This is because its horizontal position correspond to the start position of a tile. The sprite will be overwriting only one tile per scanline. More precisly, it will only overwrite the 4 bottom scanlines of the upper tile and the top 4 scanlines of the lower tile. The label **sprite_0** in the code is what takes care of these cases.

Now look at this other example:



This sprite is not aligned horizontally. It is actually overwriting TWO tiles for all 8 scanlines. Since 2 pixels are located in the leftmost tile and 6 pixels in the rightmost, the label **sprite_6** would take care of this case.

If you look carefully at this second screenshot, one important detail can be observed. Where has gone the pellet that should be behind the Pac guy? This is a limitation of my sprite/tile system: Since the sprite is partially occluding the leftmost tile, the content of this tile is **entirely** replaced with the part of the sprite that is inside. When playing the game, you can 'see' this when eating pellets while moving horizontally: the pellet will mysteriously 'disappear' a tiny bit before you actually eat them. It is not very apparent, but a similar effect can be observed when a ghost is chasing you or is following another ghost very closely. You should be able to see a horizontal 'black box' surrounding the moving objects.

True purists might say "These are not sprites!". That's correct: What I did in Rem-Pac could be called a 'pseudo-sprite' system. True sprite would have been incredibly difficult (or even impossible) without complete redesign of the whole game code.

An interesting summary of the clock cycle situation can be found in the code, it says:

```
; the worst line (16 pellets) takes 107 clocks
; 482-24-4 = 454 clocks - 107 = 347
; 83 clocks per sprite, so: 347/83 = 4 maximum sprites possible
; (1 pac + 3 ghosts)
```

This means that since we have only 347 spare clock cycles, and that drawing a single sprite takes 83 clocks, we can only draw 4 sprites. This is to support the 'worst case' scenario, which is the rows that have 16 pellets, because it takes more clock cycles to process the color status of these pellets.

Finally when everything is done for the complete video frame, we are ready to start the 'Wait for vertical blank' delay that comes between each frame displayed on the TV. Search for **STARTVBL** to see where this happen. This macro is taking care of tidying up, displaying the XGS logo at the bottom, wait and display the bottom overscan, and set up the XGS hardware countdown timer before the next frame.

Chapter 3: Main game logic and I/O

Finally! The next most important part, right after the video kernel, is the game logic. Without this, all you would see on the screen would a static image with not joystick input and nothing moving. This part is probably the easiest and funniest since we don't have to care about clock timing anymore (maybe except for the fact that we have about 20,000 clock cycles before the next video frame, but in most pratictal case, this is enough for doing a lot of work).
The game logic code begins at the **page_200** label.

Here's a summary of the game logic operation:

- Read joystick input
- Execute one step of the current 'game state' (0 to 4)
- Play sound *(only if the current game needs it)*
- Increment frame counter
- Call the **ENDVBL** macro to tell the system that we are done for this frame
- Loop back to the start of the video kernel to display a new frame

Let's look at all the game states. You can find the appropriate code in the source file by searching for **game_state0, game_state1, game_state2, game_state3** and **game_state4.**

Game state 0: Starting state when the game is launched.

This state has two functions:

- Make the 'press button to start' message slowly blink on the screen
- Check if the user presses the button

If the user pressed the joystick button, we initialize all gameplay variables (score, pellet, sprites initial position, life counter), and transit to state 1.

Game state 1: 1 second delay before gameplay starts

This state is where a startup music could be played. But actually what it does it simply waits 60 frames (which on NTSC correspond to one second) before transiting to state 2.

Game state 2: Normal gameplay occurs here.

This is obviously the most complex game state. This state is responsible for:

- Checking joystick direction to move the Pac guy accordingly
- Move and update all 3 ghosts
- If a ghost collides with Pac:
- If a power-pill was in effect, ghost is eaten
- If no power-pill was active, Pac dies, transit to state 4
- Check if Pac is 'walking' over a pellet to eat it, if so play a sound
- If the pellet was a power-pill, turn the power-pill effect on
- Count the remaining pellets. If there is none, this level is finished: Transit to state 3
- Play the background 'siren' sound

Game state 3: Level is finished.

This state perform a 3 seconds delay and make the whole maze blinks to show the player he did it. Since no joystick check and no update are performed, all sprites appear to be 'frozen' in this state.

After the 3 seconds delay, set a new maze color, re-initialize pellets and stuff, then transit back to state 1.

Game state 4: Pac is dead.

This state waits for 200 frame (200/60 = 3.33 seconds) and display an 'exploded Pac', and makes the ghost celebrate by animating quickly. After this delay, decrease the number of life. If the player still has a remaining life, reset some status and sprite position then transit back to state 1; the player can play again.

If there was no remaining life, verify if the current score is higher than the high score. If so, copy the current score to the high score; we have a new high score!
Finally transit back to state 0, the game is over.

This is it! Of course there is quite a lot of code behind those explanation, especially for state 2 with all collision detection code and ghost movement decision. I hope that you will be able to navigate in the code and figure out how it works and have fun.

Chapter 4: Sound

There isn't much to say about sound in Rem-Pac. I'll try to cover the essential:

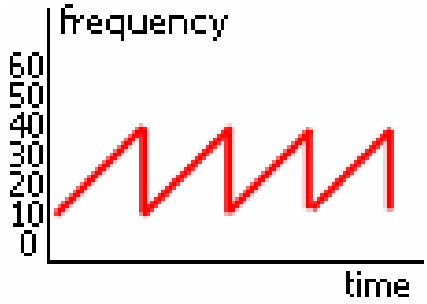
Sound is only played when the game is in state 2. All other states have absolutely no sound. The game uses two of of the three available sound channels. You can see where the sound system is being initialized by searching for **init_sound_system**. Just below this function, there is two sound functions:

play_sound: This function plays a simple 'ding' sound on channel 1; it is called when the Pac guy eats a pellet

background_sound: This plays a 'siren' sound on channel 2 at all time while in state 2. **scfreq** is a RAM variable that is used to set the background sound frequency. The formula is as follow, updated 60 times per second:

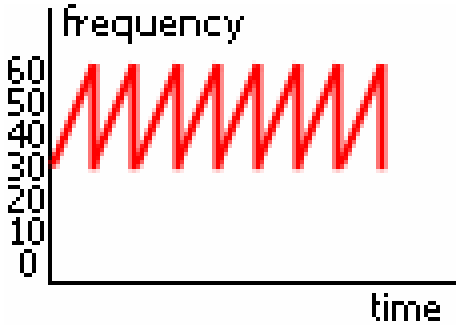
If no power-pill are active:

Add 1 to **scfreq**
If **scfreq** is greater or equal than **40**, set it back to **10**.
This will output a sound with this shape:



If a power-pill is active, then the formula is:

Add 3 to **scfreq**
If **scfreq** is greater or equal than **60**, set it back to **30**.
This will output a higher pitched sound with this shape:



Final words
Well that covers Pac from A to Y, the remaining part being putting all of this together and fully understanding it. I hope this document was of some use, and I am glad if this game can help you learn and have fun on the XGS micro edition.

Appendix

Color chart reference

Included here is a color chart reference that I've created to help me when selecting color on the XGS for Rem-Pac.

Note: Since Rem-Pac uses color 0 as reference for its color burst signal, these are the only 150 available colors in the game.

This chart was done 'manually' so it's not perfect and TVs will differ slightly.

Horizontal is the Chroma, from 0 to 15
Vertical is the Luminance, from 5 to 15

To get a color number, the formula is:
Chroma * 16 + Luminance

As an example, the starting level color in Rem-Pac is 231.
Chroma = 231 / 16 = 14
Luminance = 231 mod 16 = 7
So if you look in column chroma 14, row luminance 7, you should see approximately the light blueish tint of the starting level.