XGameStation Micro Edition
Plasma Effect
By Michael Ollanketo

In this lesson you will learn how to create an animated plasma effect on the XGameStation Micro Edition (XGSME). Differences between NTSC and PAL will be pointed out in this document.

Figure 1: NTSC version

Figure 2: PAL version

These images were grabbed through a TV-card, and the noise you see is because of the card and will not be present on an actual TV. The reason why the NTSC and PAL versions have different colors is that color is handled differently on NTSC and PAL devices, so getting the same results regardless of the format would require some extra work.

What this demo does is the following: a colorful animated plasma is drawn at the center of the screen, according to an algorithm which will be explained later. A text scroller is drawn on both sides of the plasma. The text scrolls vertically, and in opposite directions on either side of the screen. An XGS logo is drawn at the bottom of the screen, in the overscan area. The NTSC version of the demo also slowly fades the hue to give a wider range of colors. All of this will be explained throughout this tutorial.

Ok, so we're ready to start writing some SX52 assembly. First we start off with some common directives that you will see in all programs. We tell the IDE that we are using an SX52, that we want an HS3 oscillator, crystal drive and crystal feedback enabled, and the reset vector should be the label "Start".

Legend

`; n`	Instruction takes n cycles
`; m (n)`	Instruction takes m cycles, and the entire group it belongs to takes n cycles
`foobar`	NTSC-specific code
`foobar`	PAL-specific code

```
; MIC_PLASMA_04.SRC

      TITLE "Plasma demo v04"
; Plasma effect for the XGS ME
```

```
; Works only at 80 MHz

; This file needs 7 include files:
;     astaroth.mus            Music data
;     font5x7.src             Font data
;     general_define.src      Defines constants (EQU) and some system variables
;     general_macro.src       Useful general purpose macro definitions and
;                             system functions
;     sine.src                Sine table
;     xgslogo.src             XGS overscan logo
;     xgsmp.src               Music player
      DEVICE      SX52, OSCHS3, XTLBUFD, IFBD   ; Set everything for the XGS ME
      RESET Start
      FREQ  80_000_000  ; this is a directive to the ide only
                        ; if you want to put the XGS ME into RUN mode
                        ; you must make sure you go into the
                        ; device settings and make sure that
                        ; HS3 is enabled, and crystal drive and
                        ; feedback are disabled and then re-program
                        ; the chip in PGM mode and then switch it to RUN

      IRC_CAL IRC_FAST  ; Prevent a warning
      ID    "micplsma"  ; ID string

;################################################################################
```

Next, we include some common definitions and variables and declare some variables of our own for use in this program. The include file general_define.src contains some common variables, mainly counters, a `burst_phase` variable which is used to store the current color reference burst and a `phase_alt` variable for controlling the phase alternation in PAL mode. We place these variables in bank $20 and put our own variables in banks $30 and $40. The music player will use banks $A0, $B0 and $C0 for its variables. More on how to use the music player later.

```
; Global variables, starting at bank #2
      org   $20

; I'm using Remz' code here, I only modified it to use a black overscan
; instead of the blue.
include           "general_define.src"
      org $30
pixel       ds    1
horzPtr1    ds    1     ; Sine table "pointer"
horzPtr2    ds    1     ; Dito
temp        ds    1     ; Temporary
temp2       ds    1     ; ...
vertPtr1    ds    1     ; ...
vertPtr2    ds    1
cnt1        ds    1     ; Loop counter and temporary
cnt2        ds    1     ; ...
scroller1   ds    1     ; Scroller 1 position
scroller2   ds    1     ; Scroller 2 position
scroller1_2 ds    1     ; Backup
scroller2_2 ds    1     ; ...
xCnt        ds    1     ; Loop counter
yCnt        ds    1

      org $40
chroma_cnt  ds    1     ; Controls the frequency of burst phase changes
chroma_delta ds   1     ; Controls the amplitude of burst phase changes
```

The music player contains two public functions; `XgsMpInit` which is called to initialize the player, and `XgsMpUpdate` which is called every frame to update the track positions and output any changes. The song was composed in *MML* (Music Macro Language) and converted to a suitable format with *XGSMC* (XGSME Music Compiler).

```
include     "general_macro.src"
include     "astaroth.mus"    ; Include song data
include     "xgsmp.src"       ; Include music player
;###############################################################################

            org   $0    ; Set the start of the program code

            org   $+2   ; leave 2 free word here for the debugger
```

Since functions have to reside on the lower half of a page, we create a jump table here that we can use later on. The `@` sign causes the assembler to insert a `PAGE` instruction to cope with jumps to other pages.

```
; Jump table
InitMusic   jmp @XgsMpInit
UpdateMusic jmp @XgsMpUpdate


Start       ; Our real code starts here

            ; Initialize variables
            bank $40
            mov chroma_cnt,#0
            mov chroma_delta,#1

            bank $30
            mov vertPtr1,#0
            mov vertPtr2,#0
            mov yCnt,#0
```

Registers `10` and `11` in bank 0 are used to pass the address of the song to `XgsMpInit`. They form a 16-bit address together, with `10` containing the lower 8 bits and `11` the upper 8 bits. Registers 10-15 can be used at all time as extra variables no matter what bank is selected. The `XgsMpInit` function will set bit 7 of *`FSR`* (the File Select Register) as it uses banks $A0-$C0, so we need to clear that bit ourselves after the call since the `BANK` instruction only modifies bits 4..6 of `FSR`. I've written a macro called `_bank` to enforce updating of `FSR` bit 7.

```
            mov 10,#$00
            mov 11,#$08
            call InitMusic          ; Initialize the music player
            _bank $20               ; Set bank $20, and ensure bit 7 of FSR is
                                    ; cleared
```

The `INITIALIZE_VIDEO` macro in general_macro.src sets up the video port (RE) for output and initializes the `burst_phase` variable. We then proceed to render our active scanlines. I've chosen to have 192 active scanlines in both versions of the demo, since they are guaranteed to be visible both on NTSC and PAL displays even though you could use a higher resolution in the PAL version.

```
            INITIALIZE_VIDEO  ; Initialize I/O controller for video
Begin_Raster                  ; Loop here for each scanline
            ; Use 192 active scanlines
            mov scanline, #192
```

On each side of he plasma there is a short text message scrolling vertically. Each of the 192 "active" scanlines are drawn in the following manner: left scroller, plasma, right scroller. The scroller consists of thirty two characters (`"NTSC PLASMA FOR THE XGS-ME   *    "`/`"PAL PLASMA FOR THE XGS-ME    *     "`). Each character is eight pixels high, giving a total of 256 pixels. So for every scanline we divide the scroller position (0..255) by eight to get an offset into the string. We then `AND` the scroller position with seven to get the row within the character. The below figure shows this process:

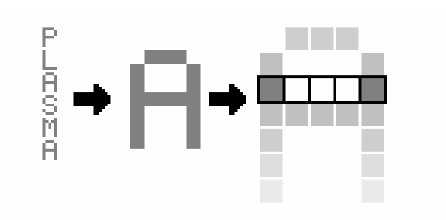

Figure 3: Scroller character row selection

The `M` register is used to select the page to read from, while `W` contains the 8-bit offset. After the read, `W` will contain the lower 8 bits of the read word and `M` will contain the upper 4 bits.

The setup code for each scanline is slightly different for NTSC and PAL, so I have two different macros to handle each format, called `PREPARE_VIDEO_HORIZ` (NTSC) and `PREPARE_VIDEO_HORIZ_PAL`. These macros send out the horizontal sync signal and the color reference burst among other things. `PREPARE_VIDEO_HORIZ_PAL` also takes care of the color phase alternation, in which the reference burst is shifted approximately 45 degrees every scanline. Both macros use variables in bank $20 so we need to be sure that bank $20 is selected before using the macros.

The *`RTCC`* (Realtime Cycle Counter) is used for scanline timing, so we start off by setting it to count up every 32nd cycle.

```
            mov !OPTION, #%11000100          ; Sets prescaler to 1:32

Raster_Loop1

            PREPARE_VIDEO_HORIZ burst_phase     ; Prepare required video signal
                                                ; for a scanline
            PREPARE_VIDEO_HORIZ_PAL burst_phase
            clr RTCC                ; 1. Reset the realtime cycle counter
            bank $30                ; 1
            mov RE, #BLACK          ; 2. Output black

            mov xCnt,#100           ; 2. 100 "pixels"

            ; Copy pointers for use in the scanline loop
            mov cnt1,horzPtr1       ; 2
            mov cnt2,horzPtr2       ; 2
            bank $20                ; 1 (10)
            DELAY(492)              ; Delay a while to center the effect
            DELAY(495)              ; Delay a while to center the effect
```

```
; Convert the scroller position (0..255) to a string
; position (0..31) and character offset (0..7) and
; read a character from the string.
bank $30            ; 1
mov M,#TEXT_PAGE    ; 1. Set page to read from
mov W,scroller1     ; 1. Scroller position
mov temp,W          ; 1
and W,#7            ; 1
mov temp2,W         ; 1. Save character row (0..7)
clc                 ; 1. Divide by 8 (char height)
rr temp             ; 1  ...
clc                 ; 1  ...
rr temp             ; 1  ...
clc                 ; 1  ...
rr temp             ; 1  ...
mov W,temp          ; 1. String position (0..31)
iread               ; 4 (17)

; Multiply by 8 (height of char)
mov temp,W          ; 1. temp=x
add W,temp          ; 1. W=x*2
mov temp,W          ; 1. temp=x*2
add W,temp          ; 1. W=x*4
mov temp,W          ; 1. temp=x*4
add W,temp          ; 1. W=x*8

; Now read one row of character data
mov M,#FONT_PAGE    ; 1. Set page to read from
add W,temp2         ; 1. Offset = char*8 + (scroller&7)
iread               ; 4. Read one byte (=row)
mov pixel,W         ; 1. Save it
mov temp,#8         ; 2 (9). Number of pixels to draw
```

We have now read the correct byte, or row, from the character to draw. We now loop eight times, shifting out one bit each iteration, and draw either the desired color or black depending on whether the bit was set or not. The character rows are stored backwards, with the leftmost pixel in bit 0 which means that we rotate right every iteration to the the current pixel. This figure shows the process of drawing one row of the letter ‘S’:

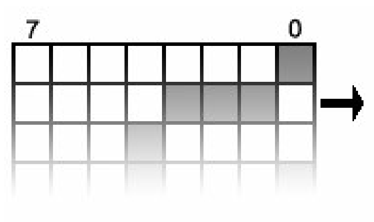

Figure 4: Representation of 1-bpp characters

```
            ; Draw the row
:Draw_char_line_1
            mov W,#(COLOR14+8) ; 1
            rr pixel            ; 1. Place lsb in C
            sc                  ; 2 / 1. Skip if bit is set
            mov W,#BLACK        ; 1. Bit was clear, should be black
            mov RE,W            ; 1. Output color
            DELAY(7)
            djnz temp,:Draw_char_line_1 ; 4 / 2
```

```
            mov RE,#BLACK      ; 2. Output black
            bank $20           ; 1
            DELAY(40)
```

The plasma effect is achieved by sampling three sine curves. See the below figure.

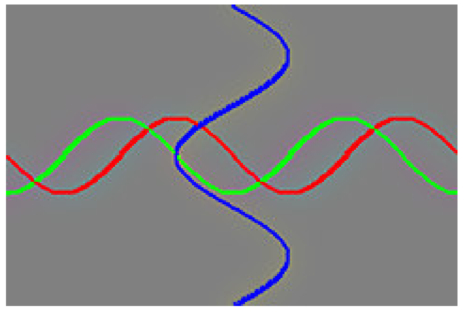

Figure 5: Sine curves

The "red" and "green" curves move in the opposite direction for every pixel, while the "blue" curve moves every scanline. This creates the impression of a number of expanding and shrinking blobs. All three curves are also moved for each frame to scroll the entire plasma across the screen.

In the scanline loop we sample each of these curves by reading bytes from the table contained in sine.src with three different offsets. These samples are added, and then we `OR` in some additional bits to ensure that we never get zero luma (which would send out an incorrect sync signal and disrupt our video output). Here `cnt1` and `cnt2` controls the "red" and "green" curves, while `vertPtr1` controls the "blue" curve.

Each element in the sine table has the page number stored in the upper 4 bits. That way we don't have to reload `M` for every `iread`.

```
            ; Set up the plasma rasterizer
            bank $30           ; 1. Set active register bank
            nop                ; 1. Delay for a few cycles..
            nop                ; 1
            nop                ; 1 (4)

            ; Draw one line of the plasma
draw_scanline
            add cnt2,#$FE      ; 2. Subtract 2 ($FE == -2)
            inc cnt1           ; 1 (3)

            ; Read from sine table
            mov M,#SINE_PAGE   ; 1
            mov W,cnt1         ; 1
            iread              ; 4
            mov pixel,W        ; 1 (7)

            ; Read another byte
            mov W,cnt2         ; 1
            iread              ; 4
            add pixel,W        ; 1 (6)

            ; ..and another
            mov W,vertPtr1     ; 1
            iread              ; 4
            add W,pixel        ; 1 (6)

            ; OR in some additional bits
```

```
            or W,#$8A        ; 1

            ; Output the pixel data
            mov RE,W         ; 1

            djnz xCnt,draw_scanline ;4 / 2
            ; Total: 3+7+6+6+1+1+4 = 28 clocks
```

This line of the plasma is done. We output black for a while and then draw the second scroller, which moves in the opposite direction of the first one. The code is very similar to that of the first scroller. We also change the vertical sine table offset (which moves the blue curve in figure 5).

```
            mov RE, #BLACK   ; 2. Output black
            dec vertPtr1     ; 1. Decrease vertical pointer

            bank $20         ; 1
            DELAY(39)

            ; Draw the second scroller. Same procedure as for
            ; the first one.
            bank $30         ; 1
            mov M,#TEXT_PAGE ; 1
            mov W,scroller2  ; 1
            mov temp,W       ; 1
            and W,#7         ; 1
            mov temp2,W      ; 1
            clc              ; 1
            rr temp          ; 1
            clc              ; 1
            rr temp          ; 1
            clc              ; 1
            rr temp          ; 1
            mov W,temp       ; 1
            iread            ; 4 (17)

            ; Multiply by 8
            mov temp,W       ; 1
            add W,temp       ; 1
            mov temp,W       ; 1
            add W,temp       ; 1
            mov temp,W       ; 1
            add W,temp       ; 1 (6)

            mov M,#FONT_PAGE ; 1
            add W,temp2      ; 1
            iread            ; 4
            mov pixel,W      ; 1
            mov temp,#8      ; 2 (9)

:Draw_char_line_2
            mov W,#(COLOR2+8) ; 1
            rr pixel         ; 1. Place lsb in C
            sc               ; 2 / 1. Skip if bit is set
            mov W,#BLACK     ; 1. Bit was clear, should be black
            mov RE,W         ; 1. Output color
            DELAY(7)
            djnz temp,:Draw_char_line_2 ; 4 / 2
            mov RE,#BLACK    ; 2

            inc scroller1    ; 1
```

```
            inc scroller2     ; 1
            bank $20          ; 1
            cjb RTCC,#131,$         ; Delay for the remainder of the scanline
            cjb RTCC,#130,$         ; Delay for the remainder of the scanline

            djnz scanline, Raster_Loop1   ; Loop for the next scanline

            ; page_200 trick: this clunky trick is to prevent involuntary jumps
            ; between a page boundary.
            jmp @page_200
IF $ > $200
ERROR "Page Spillage!"
ENDIF


;################################################################################

; Let's start a new page to avoid overflow..

org $200
page_200
            page $                        ; Set the new page
```

Here we draw the bottom overscan. This consists of eighteen black lines and a sixteen pixels high "XGS" logo, for a total of thirty four lines. The PAL version has another 23 lines of bottom overscan, which will be black.

The logo is contained in xgslogo.src and is 24*8 pixels with one bit per pixel. Each line of the logo is drawn twice to make it look taller, and a gradient is applied at runtime for better looking results. This figure shows the effect of applying the gradient and scaling the logo:

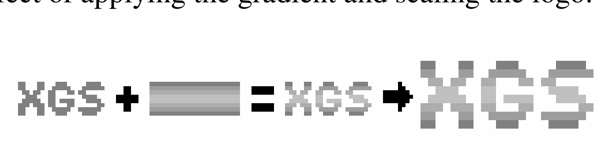

Figure 6: Overscan logo

The code for drawing the overscan logo is very similar to that for drawing the text scrollers earlier. The difference is that each row is twenty four bits instead of eight, so we have three times as much code.

Each NTSC frame is made up of 525 lines, while a PAL frame has 625 lines. This gives 262 and 312 lines per field, respectively, if we ignore the fractional part. I chose to spend the extra 50 lines in the PAL version on the top and bottom overscans. These are areas of the frame that may or may not be visible, depending on the device. If you look at the code you will notice that I have added a total of 23+28 lines of extra overscan for the PAL version, which doesn't add up to 50. This is because we have a longer vertical sync period in the NTSC version (20224 cycles compared to 15360).

```
            bank $30
            mov cnt2,#34                  ; Number of scanlines
            mov cnt2,#(34+23)             ; Number of scanlines
:Vblank_Loop1
            bank $20
            PREPARE_VIDEO_HORIZ burst_phase
            PERPARE_VIDEO_HORIZ_PAL burst_phase
            clr RTCC                      ; Reset realtime cycle counter
            bank $30
```

```
            cjae cnt2,#30, :Black_line1
            cjb  cnt2,#14, :Black_line2

            ; Draw XGS overscan logo

            mov RE,#BLACK     ; 2. Output black
            mov cnt1,#8       ; 2. Number of pixels
            mov M,#LOGO_PAGE  ; 1
            mov W,yCnt        ; 1
            iread             ; 4. Read first byte
            mov pixel,W       ; 1

            ; Calculate the logo gradient
            mov temp,cnt2     ; 2
            sub temp,#22      ; 2. 22 = first_line-char_height = 30-8
            sb temp.7         ; 1. Skip if the result is negative
            not temp          ; 1. Invert bits
            mov W,#7          ; 1. Keep lower 3 bits
            and temp,W        ; 1
            add temp,#(COLOR11+2)   ; 2. Add base color

            bank $20          ; 1
            DELAY(1614 - 2 - 14 - 8 - 1 - 10)
            bank $30          ; 1

:Draw_bits_0_7
            mov W,temp ;#(COLOR14+5)      ; 1
            rr pixel          ; 1. Place lsb in C
            sc                ; 2 / 1. Skip if bit is set
            mov W,#BLACK      ; 1. Bit was clear, should be black
            mov RE,W          ; 1. Output color
            DELAY(31)
            djnz cnt1,:Draw_bits_0_7 ; 4 / 2
            mov RE,#BLACK     ; 2

            ; Read the next byte
            mov cnt1,#8       ; 2
            mov M,#LOGO_PAGE  ; 1
            inc yCnt
            mov W,yCnt        ; 1
            iread             ; 4
            mov pixel,W       ; 1 (10)
:Draw_bits_8_15
            mov W,temp ;#(COLOR14+5)      ; 1
            rr pixel          ; 1
            sc                ; 2 / 1
            mov W,#BLACK      ; 1
            mov RE,W          ; 1
            DELAY(31)
            djnz cnt1,:Draw_bits_8_15 ; 4 / 2
            mov RE,#BLACK     ; 2

            ; ..and finally the last one
            mov RE,#BLACK     ; 2
            mov cnt1,#8       ; 2
            mov M,#LOGO_PAGE  ; 1
            inc yCnt
            mov W,yCnt        ; 1
            iread             ; 4
            mov pixel,W       ; 1 (12)
:Draw_bits_16_23
            mov W,temp ;#(COLOR14+5)      ; 1
            rr pixel          ; 1
            sc                ; 2 / 1
```

```
            mov W,#BLACK      ; 1
            mov RE,W          ; 1
            DELAY(31)
            djnz cnt1,:Draw_bits_16_23 ; 4 / 2
            nop               ; 1
            nop               ; 1

            mov RE,#BLACK     ; 2
            sub yCnt,#2       ; 2. Set yCnt back to its prior value
```

To make the logo look twice as high we only increase the row counter every other scanline.

```
            ; yCnt is increased by 3 on even scanlines
            ; (logo is 24 pixels wide = 3 bytes/row)
            mov W,cnt2        ; 1
            not W
            and W,#1          ; 1
            add yCnt,W        ; 1
            add yCnt,W        ; 1
            add yCnt,W        ; 1

            jmp :Next_line          ; 3
; This line should apparently be black.
:Black_line1
            mov RE, #BLACK          ; ( 2 cycles ) sync
            jmp :Next_line
:Black_line2
            mov RE, #BLACK          ; ( 2 cycles ) sync
            jmp :Next_line
```

We use the RTCC to control the length of each scanline. In PAL mode we have 32 cycles less because the line setup is slightly longer.

```
:Next_line
            bank $30
            cjb RTCC,#131,$         ; Wait for this scanline to end
            cjb RTCC,#130,$
            djnz cnt2, :Vblank_Loop1
            ; END BOTTOM SCREEN OVERSCAN
```

We are done with the bottom overscan. Now we send out the sync signal to the video port and reset the realtime cycle counter.

```
            ; VERTICAL SYNC PULSE
:Begin_Blank
            mov RE, #SYNC           ; Send sync signal
            mov !OPTION, #%11000111 ; Turns off interrupts, sets prescaler
                                    ; to 1:256
            mov RTCC, #0            ; Start RTCC counter
```

Here we have about 20000 cycles (about 15000 in PAL mode) to do anything we want. Typical things to do here could be collision detection, enemy A.I, music player updates etc.
In the NTSC version I am changing the color reference burst every 128th frame so that the hue of the plasma will change slowly with time. This is not done in the PAL version, since color works different for PAL.

```
;###############################################################################

            ; The idea here is to change the color burst phase every
            ; 128th frame (when chroma_cnt hits zero). The burst phase
            ; is either incremented or decremented depending on the value
            ; in chroma_delta. The delta is inverted (negated) when the
            ; color burst is 0 or 13.
            bank $40
            mov W,chroma_delta
            dec chroma_cnt
            decsz chroma_cnt
            jmp no_chroma_change
            bank $20
            swap burst_phase        ; Swap nibbles
            add burst_phase,W       ; Add delta
            and burst_phase,#15     ; Only four bits of chroma
            cje burst_phase,#13,invert_delta
            cje burst_phase,#0,invert_delta
            jmp dont_invert
invert_delta
            bank $40
            not chroma_delta
            inc chroma_delta        ; not+inc == neg
            bank $20
dont_invert
            swap burst_phase        ; Swap nibbles back to original order
            or burst_phase,#BLACK_LEVEL   ; OR in some base luma
no_chroma_change
            bank $30
            ; Update sine table pointers
            add horzPtr1,#$FE
            inc horzPtr2
            add vertPtr2,#2
            mov vertPtr1,vertPtr2

            mov yCnt,#0             ; Reset yCnt

            ; Update text scroller positions
            inc scroller1_2
            dec scroller2_2
            mov scroller1,scroller1_2
            mov scroller2,scroller2_2
            bank $20
```

Another frame has been drawn so it is time to update the music track position, and output any new sounds. Again, we pass the address of the song as a 16-bit value in the global registers `10` and `11`.

```
            mov 10,#$00
            mov 11,#$08
            call @UpdateMusic       ; Update music each frame
            _bank $20
```

Now we wait for the vertical sync period to end. The realtime cycle counter will increment every 256th cycle, according to the setting in the `OPTION` register. The `$` sign simply states that we want to jump to the instruction itself, which saves us from creating a label for each such loop. When the vertical sync period is over we draw the top overscan and start all over again with the next frame.

```
            ; We are done, now wait for the remaining time to finish this video
            ; frame and prepare overscan (bottom and top) for the next!
```

```
            cjb RTCC, #79, $  ; Wait for remaining vsync (79*256 cycles = 20224,
                              ; close enough)
            cjb RTCC, #60, $  ; Wait for remaining vsync (60*256 cycles = 192us)
            ; END VERTICAL SYNC PULSE


            ; TOP SCREEN OVERSCAN
            mov !OPTION, #%11000100           ; Sets prescaler to 1:32
            mov scanline, #32
            mov scanline, #(32+28)
:Vblank_Loop3
            PREPARE_VIDEO_HORIZ burst_phase     ; Prepare video signal
            PREPARE_VIDEO_HORIZ_PAL burst_phase ; Prepare video signal
            clr RTCC
            mov RE, #BLACK                    ; Output black
            cjb RTCC,#131,$
            cjb RTCC,#130,$
            djnz scanline, :Vblank_Loop3      ; Repeat..

            jmp @Begin_Raster       ; Loop back for the next frame
```

The code for the demo is done. Hopefully, at this point you have grasped the theory behind the effect and also picked up a few things about the SX52 and the rest of the XGSME's hardware. The realtime cycle counter is a very handy feature which can be used in a lot of cases, and is much more elegant than nested loops or other cycle counting constructions. The music player and the sound chip will be explained in more detail in another tutorial, so don't worry about them for now. If you feel uncertain about something, try reading the tutorial again and look at the code. And remember that the best way to get better at something is to practice.

```
; Code ends here. Data follows below

;##############################################################################


; Sine table
; Calculated using the following formulae:
;
;     floor(cos(i*PI/64)*15.7 + 15.7)
;
; SINE_PAGE*$100 is added to each entry to avoid having to
; set the M register for each read since (M:W)->M:W
;
SINE_PAGE   EQU   $0A
org SINE_PAGE*$100
include "sine.src"

;##############################################################################

; XGS overscan logo
LOGO_PAGE   EQU   $0B
org LOGO_PAGE*$100
include "xgslogo.src"

;##############################################################################

; 5x7 font
; Each character is packed into 8 bytes (1 bit per pixel)
; First char is space, capital letters A..Z follow.
FONT_PAGE   EQU   $0C
org FONT_PAGE*$100
include "font5x7.src"
```

```
;################################################################################

; Scroller text. 32 chars
TEXT_PAGE   EQU   $0D
org TEXT_PAGE*$100

dw    $0E
dw    $14
dw    $13
dw    $03
dw    $00
dw    $10
dw    $0C
dw    $01
dw    $13
dw    $0D
dw    $01
dw    $00
dw    $06
dw    $0F
dw    $12
dw    $00
dw    $14
dw    $08
dw    $05
dw    $00
dw    $18
dw    $07
dw    $13
dw    $1B
dw    $0D
dw    $05
dw    $00
dw    $00
dw    $1C
dw    $00
dw    $00
dw    $00
```