VIDEOPAC G7000 BIOS

Introduction

Copyright © 1997-2007 by Sören Gust
<sgust@ithh.informationstheater.de>

February 2, 2007

Table of Contents

1 Introduction

Use this information at your own risk. The information in this document was compiled by disassembling the BIOS and some games for the Videopac G7000 and by running small programs on the machine and on the o2em emulator by Dan Boris that is now maintained by Andre de la Rocha. I also used a disassembly of the BIOS generated by Paul Robson. The PAL/NTSC detection program is based on an idea from René van den Enden, who also filled in some of the details of the Videopac+ G7400. This document is not fully complete and may not be always correct. English is not my first language, so please forgive me any errors.

This is not meant as a manual for 8048 assembler. In this document I assume you already know how to program the 8048. You can get information about the 8048 processor and the rest of the MCS-48 family at this FTP server. (ftp://ftp.armory.com/pub/user/rstevew/) You can also find several older 8048 assemblers running under DOS there.

All information about the hardware in this document is based on the "Odyssey² Technical Specs" at the home page (http://atarihq.com/danb/o2.shtml) from Daniel Boris.

If you find any bugs in the demo programs, mistakes in the text or have any other comments to this document please E-Mail me.

1.1 What is the Videopac G7000 / Odyssey²?

The Videopac G7000 is a video game console based on the Intel 8048 processor. In the United States it is known as the Odyssey². There are some differences between the machines especially in the video output, the Videopac G7000 outputs PAL video, the Odyssey² NTSC. For more details read the chapter about differences between PAL and NTSC machines (page 29) . In this document I try to describe how to use the built-in 1 KByte BIOS. This is identical to both machines as far as I know. For general information about the machine look at the Odyssey² FAQ (http://www.digitpress.com/faq/odyssey2.htm) written by Robert D. Kaiser.

The full source codes to my demo programs can be found on my web page (http://soeren.informationstheater.de/g7000/). There is also the latest version of this document in PDF and HTML format available. If you are seriously interested in programming the Videopac G7000 / Odyssey² I suggest you download the PDF version and print it.

1.2 Features provided by the BIOS

The BIOS contains a very sophisticated interrupt routine which can do complex VDC (Video Display Controller) register movements at the next VSYNC, checks for collisions and uses some kind of macro code interpreter to generate the built-in sounds. You can insert your own code at three different positions into this routine. The BIOS reads the keyboard and joystick, displays character graphics and can even manage a simple clock.

1.3 What resources are free to use

The BIOS uses RB0 for its interrupt routine. 03Dh-03Fh from the internal RAM are also used by the interrupt, as is F1. For some purposes the interrupt routine interprets structured data in the external RAM starting with 07Fh and counting down to the end of the structure. The bytes 01h-02h in external RAM are used by the clock routines.

2 Creating programs on the G7000 / Odyssey²

2.1 Which assembler to use

To assemble your programs you need an assembler for the Intel 8048 processor which is a member of the MCS-48 processor family. For the demo programs in this document I used the freeware assembler ASL (http://www.alfsembler.de/). I used the Linux version, but there are also DOS and OS/2 versions available there.

2.2 Using ASL

ASL does not generate a ROM file directly, it uses an intermediate file with the extension ".p". There are several tools included in the ASL package that can convert these ".p" files. Use p2bin to generate binary files. It needs to know which address range to extract from the ".p" file. The internal BIOS of the G7000 / Odyssey² occupies the first 1KByte. Nearly all commercial cartridges use only the next 2KByte, so to generate a ROM file for use with o2em the range you need is from 0400h to 0bffh. To mark parameters as hexadecimal p2bin uses the $ sign which needs to be escaped by most shells. So it is more portable to use decimal numbers. To assemble the hello.a48 demo program into a o2em ROM file you need to type:

```
asl hello.a48
p2bin hello.p hello.rom -r 1024-3071
```

This example is for Linux, the executable of the DOS version is called AS.EXE, so you have to substitute `asl` with `as` in the example when using DOS. For the G7000RAM you need to copy the range 1024-4095. Later versions of o2em support 3K per bank, so there is no longer a difference between o2em and the G7000RAM.

All these examples are for programs with only one G7000 / Odyssey² program bank. For bigger programs I can think of several methods to handle them. The method I currently prefer is to use one assembler file per program bank. It takes takes some time to fill the 2KBytes available with o2em. So if your code reaches that size you will be experienced enough to find the way which is best for your code.

2.3 How to get the binary into the machine

After assembling your programs you have to find a way to run them. You can run the programs on the o2em (http://o2em.sourceforge.net/) emulator. If you prefer to test your programs on a real machine you can use the G7000RAM cart. It is a 12K (4×3K) RAM cart with a serial port (RS232) to upload the programs. Find out how to build one on my home page. (http://soeren.informationstheater.de/g7000/g7000ram.html) If you can burn EPROMs you can build an EPROM cart, eg. by hacking an existing cart.

2.4 Conventions used in this document

All numbers in hex are written as 0ABCDh in this document. A means the A register of the 8048, R0-R7 the active register bank, most of the time this is RB1. When writing R0:5-7 I mean bit 5-7 of register R0. RB0 and RB1 are a specific register bank, MB0 and MB1 are the program memory banks. P0, P1 and P2 are the external ports. F0 and F1 are flags internal to the 8048. T is the timer register. 256 bytes together on a 256 byte boundary are one page. All symbolic labels for the BIOS routines in this document are explained in the summary chapter (page 49) and are included into g7000.h. (http://soeren.informationstheater.de/g7000/demoprograms/g7000.h) Symbolic names for VDC registers start with `vdc_`, those for internal RAM start with `iram_`. The names for external RAM start with `eram_`. Names starting with `col_` are names for colours.

2.5 The beginning

At power on or after reset the BIOS jumps directly to 0400h, the first address of the external ROM. Normally you put a `jmp selectgame = 02C3h` there, this initialises the VDC, internal and external RAM. Then it displays the "SELECT GAME" message and waits for a key. After that it jumps to 0408h. There you have to start your code with another `jmp`, because 040Ah is used by the interrupt. Which key was pressed is stored in A. All sprites, chars, quads and the grids are initialised, the VDC is enabled and the graphics are on.

2.6 Getting the interrupt chain right

If an external interrupt occurs, the BIOS jumps to 0402h. From there you can insert some code into the interrupt. After that do a `jmp 0009h`, I call this `irq`. The BIOS checks for VSYNC (Vertical sync, marks the beginning of a new frame) and jumps to 0406h, if VSYNC. Here you can insert again some code. Continue to `vsyn-cirq = 001Ah`. After processing the register transfer table the BIOS jumps to 040Ah, if there is a sound event to process. Again you can insert some code here. The sound event gets processed at `soundirq = 0044h`. So if you don't want to insert code anywhere into the interrupt the beginning of your program looks like:

```
        include "g7000.h"

        org     0400h

        jmp     selectgame
        jmp     irq
        jmp     timer
        jmp     vsyncirq
        jmp     start
        jmp     soundirq

timer   retr            ; timer interrupts are discussed later
start                   ; your code starts here
```

2.7 Displaying Characters: Hello World

In this section I will explain my hello.a48 (http://soeren.informationstheater.de/g7000/demoprograms/hello.a48) program. It is very important not to change the VDC registers while displaying graphics, at least not if you want predictable results. So there are two routines to turn the graphics off (`gfxoff = 11Ch`) and on (`gfxon = 0127h`). To display characters there is `printchar = 03EAh`. You put a pointer to the character to display into R0, for example `vdc_char0 = 010h` as the first character. In R3/R4 you put the X/Y position on the screen. R6 is the colour of the character, for example `col_char_white = 00Eh`. In R5 you tell which character to display. After calling printchar, R0 is set to the next character and R3 is advanced right for 8 pixels, so you can print several characters into one line. The following code prints the traditional "HELLO WORLD" onto the screen.

```
start
        call    gfxoff              ; switch the graphics off
        mov     r0,#vdc_char0       ; start char
        mov     r3,#20h             ; x-position
        mov     r4,#20h             ; y-position
        mov     r2,#0Bh             ; length
        mov     r1,#hellostr & 0FFh ; the string to print
                                    ; must be in the same page
loop    mov     a,r1                ; get pointer
        movp    a,@a                ; get char
        mov     r5,a                ; into to right register
        inc     r1                  ; advance pointer
        mov     r6,#col_chr_white   ; colour
        call    printchar           ; print it
        djnz    r2,loop             ; do it again
        call    gfxon               ; lets see what is written
stop    jmp     stop                ; Thats all

hellostr db     1Dh, 12h, 0Eh, 0Eh, 17h, 0Ch
         db     11h, 17h, 13h, 0Eh, 1Ah
```

3 Using interrupts to change VDC registers

In the standard interrupt routine there is a mechanism to automatically copy data into VDC registers at the next VSYNC. Simply put the number of bytes to copy into 07Fh at external RAM. Put a pointer to the first register at 07Eh. The bytes to put into the ascending registers are stored in descending order at 07Dh. After that begin again with a new byte count or put a 0 at the end. The table is activated when `irq_table = bit 7` of `iram_irqctrl = 03Fh` in internal RAM is set. At the next VSYNC interrupt the table is copied into the registers. The VDC and external RAM share the same address space, so there are two routines to switch to VDC (`vdcenable = 00E7h`) or to RAM (`extramenable = 00ECh`).

All routines using tables in the BIOS use R0 as a pointer to the next free byte of the table. The routine `tableprintchar = 0197h` puts the register contents to print a char or quad into the table, you have to fill in the number of bytes to copy and the register pointer. The other parameters are identical to those of `printchar`. The routine `tableend = 0132h` puts the end marker into the table and activates it. So here is another demo program (http://soeren.informationstheater.de/g7000/demoprograms/hellot.a48). It prints "HELLO WORLD", too, but this time with tables and in green.

```
start
        call    extramenable        ; the table is in extram
        mov     r0,#07Fh            ; begin of table
        mov     a,#02Ch             ; 4 bytes / char * 0Bh
        movx    @r0,a               ; into table
        dec     r0                  ; next byte in table
        mov     a,#vdc_char0        ; first register
        movx    @r0,a               ; into table
        dec     r0                  ; next byte in table
        mov     r3,#20h             ; x-position
        mov     r4,#20h             ; y-position
        mov     r2,#0Bh             ; length
        mov     r1,#hellostr & 0FFh ; the string to print
                                    ; must be in the same page
loop    mov     a,r1                ; get pointer
        movp    a,@a                ; get char
        mov     r5,a                ; into the right register
        inc     r1                  ; advance pointer
        mov     r6,#col_chr_green   ; colour green
        call    tableprintchar      ; put it into table
        djnz    r2,loop             ; do it again
        call    tableend            ; activate the table

stop    jmp     stop                ; Thats all

hellostr db     1Dh, 12h, 0Eh, 0Eh, 17h, 0Ch
         db     11h, 17h, 13h, 0Eh, 1Ah
```

4 A more complex example: joystick, sprites

In this section I explain a program (http://soeren.informationstheater.de/g7000/demoprograms/joystick.a48) which demonstrates the use of the joystick routines and how to display sprites. You can control a dot with the joystick. The dot changes into an arrow and moves if you move the joystick. By pressing fire you can double the size of the dot/arrow. This program is useful if you want to test a joystick, for example after some contact cleaning.

4.1 The routines the program uses

This program uses three new routines. The first one called `waitvsync` waits for the next VSYNC. I use it, because I only want to check for joystick movement once per frame. The routine `getjoystick` tests the position of one joystick. The number which joystick to test has to be put into R1. The outputs of `getjoystick` are 1/0/0FFh in register R2 for X axis and R3 for Y axis. All numbers are in 2's complement, so 0FFh is really a -1. If the button is pressed, F0 is set. This output is direct usable for changing positions, simply add R2/R3 to the X/Y position.

If your sprite has a front side which always points into the direction it moves, you can use `decodejoystick`. It takes the offset data in R2/R3 and converts them into a direction in R1. This direction can be used to change the shape of the sprite.

4.2 The program

Before I start to explain the program in details, some general remarks. All VDC accesses are done via tables. The program uses some variables in internal RAM, they are prefixed with `iram_`. All position data is first changed in internal RAM and then transferred into the sprite registers. There are no checks to make sure that the position is still on the screen. I don't explain every line of code here, only the main loop.

4.2.1 Calling getjoystick and decodejoystick

This starts at the beginning of the main loop, I simply call `getjoystick`, test for fire, call `decodejoystick` and initialise my table pointer.

```
loop
        call    waitvsync       ; execute only once per frame
        mov     r1,#0           ; joystick 0
        call    getjoystick     ; get offsets

        ; test, if fire
        mov     r1,#iram_colctrl
        mov     a,#col_spr_white | spr_double
        jf0     firepressed     ; fire ?
        mov     a,#col_spr_white
firepressed
        mov     @r1,a           ; store color/control

        call    decodejoystick  ; get direction from offsets
        call    extramenable    ; enable extram
        mov     r0,#07Fh        ; start of table
```

4.2.2 Test, if I have to change the shape

Before I can test if the shape has changed, I have to test for the neutral position, because `decodejoystick` maps neutral to right. I map the neutral position to 8 if the X and Y offsets are both 0. Then I can test if the shape I need is already set.

```
        ; test, if joystick is in neutral position
        mov     a,r2            ; x-offset
        jnz     shapetest       ; left/right

        mov     a,r3            ; y-offset
        jnz     shapetest       ; up/down

        mov     r1,#8           ; shape: neutral

shapetest
        ; test, if shape has change since last frame
        mov     a,r1            ; we need r1 as pointer
        mov     r7,a            ; so put contents into r7
        mov     r1,#iram_shape  ; last shape
        mov     a,@r1           ; get it
        xrl     a,r7            ; compare
        jz      setpos          ; no need to set shape,
                                ; skip that part
```

4.2.3 Set the new shape

Now I set the new shape, which number I have to use is stored in R7. This part is skipped completely, if the shape is already set correctly. In a real program the part which copies the data should be put into a separate page together with the data, because the 8048 can only access data in ROM which is in the same page as the code. To maximise the space usable for shapes, the only code in the same page should be the one which copies the data. But this is only a small example, everything fits into one page, so this is not really necessary. Look at the program from the chapter about collision checks (page 10) to see how to handle this more correctly. When creating shape data you should know that the displayed data is mirrored by the VDC, bit 0 is displayed left.

```
        ; set new value of iram_shape
        mov     a,r7            ; the number of the shape
        mov     @r1,a           ; put into iram_shape

        ; init table to copy shape data
        mov     a,#8            ; copy 8 bytes
        movx    @r0,a
        dec     r0
        mov     a,#vdc_spr0_shape
        movx    @r0,a
        dec     r0

        ; now copy the data
        mov     a,r7            ; number of shape
        rl      a
        rl      a
        rl      a               ; 3*rl = a*8
        add     a,#spritedata & 0FFh
        mov     r1,a            ; start of shape data
        mov     r7,#8           ; 8 bytes
copyspriteloop
        mov     a,r1
        movp    a,@a            ; get byte
        movx    @r0,a           ; store in table
        dec     r0
        inc     r1
        djnz    r7,copyspriteloop
```

4.2.4 Adjusting the positions in internal RAM

Now I add the offsets to the old position. The Y position is no problem. But the X position is 9 bit wide, so I have to use the carry flag. Another problem is that the X offset is only 8 bit wide. I am dealing with signed numbers in 2’s complement, so to expand the X offset to 9 bits I have to reuse bit 7 as bit 8. Because the X offset is only +1/0/-1, bits 1-7 are always the same, so I can reuse bit 1 as bit 8.

```
setpos
        ; adjust sprite positions in iram
        ; y is simple
        mov     r1,#iram_y      ; y position
        mov     a,@r1           ; get it
        add     a,r3            ; add offset
        mov     @r1,a           ; store y position

        ; x is a 9 bit add using carry, we need to
        ; expand r2 to 9 bit also
        mov     r1,#iram_xl     ; low byte of x
        mov     a,@r1           ; get it
        add     a,r2            ; add offset, sets
                                ; carry if necessary
        mov     @r1,a           ; store low byte of x
        mov     r1,#iram_xh     ; high bit of x
        mov     a,@r1           ; get it
        addc    a,#0            ; add the carry
        mov     r7,a            ; we need this later
        mov     a,r2            ; get offset
        rr      a               ; reuse bit 1 of offset
                                ; as bit 8. we need it,
                                ; because we are dealing with
                                ; 2s complement if R2=-1=01ffh
        add     a,r7            ; add result from above to
                                ; bit 8 of offset
        anl     a,#001h         ; we only need 1 bit
        mov     @r1,a           ; store it as high bit
```

4.2.5 Copying the sprite position into the VDC

The only thing that is left to do is to put the sprite position into the VDC. For this I use the table routine in the VSYNC. First I have to prepare the table. The Y position can be put directly into the table. The X position is 9 bit wide, but this time the lowest bit is separated from the rest. In internal RAM the highest bit is separated from the rest. So I have to re-split the X position and combine the lowest bit with the color/control value stored at the beginning of the main loop when testing for fire.

```
        ; prepare table for sprite positions
        mov     a,#3            ; copy 3 bytes
        movx    @r0,a
        dec     r0
        mov     a,#vdc_spr0_ctrl
        movx    @r0,a
        dec     r0

        ; set sprite positions from iram using table
        mov     r1,#iram_y      ; y position
```

```
mov     a,@r1           ; get it
movx    @r0,a           ; put it into table
dec     r0

; x position: recombine xh and xl and split
; it into 8-1/0
mov     r1,#iram_xh     ; highest bit of x position
mov     a,@r1
rrc     a               ; highest bit of sprite_x
                        ; into carry
mov     r1,#iram_xl     ; low byte of x position
mov     a,@r1
rrc     a               ; lowest bit into carry,
                        ; highest bit into 7
movx    @r0,a           ; put bit 8-1 of sprite_x into
                        ; sprite control 1
dec     r0
mov     a,#0            ; we only need carry (lowest
                        ; bit of sprite_x)
rlc     a               ; lowest bit of sprite_x
                        ; into bit 0
mov     r7,a            ; store in r7
mov     r1,#iram_colctrl; color/control
mov     a,@r1           ; get it
orl     a,r7            ; put it together
movx    @r0,a           ; put it into sprite control 2
                        ; via table
dec     r0

call    tableend        ; thats all

jmp     loop
```

5 How to check for collisions

This program (http://soeren.informationstheater.de/g7000/demoprograms/collision.a48) demonstrates how to do collision checks between objects on the screen. To have something to collide with I put two horizontal grid lines, one vertical grid line, one ship shaped character and sprite 1 shaped as a triangle on the screen. To the right I display the contents of the collision register. It shows which objects collide with sprite 0 which is the joystick controlled ball. The joystick and sprite move routines are recycled from the previous chapter, (page 6) they are not explained again. Since the part of the program which displays the static objects mostly moves values into the correct VDC registers I don't explain it here.

5.1 Setting and reading the collision register

It is only legal to read the collision register during VBLANK. Fortunately the IRQ routine in the BIOS is doing this. The value is stored in internal RAM at 03Dh. But before I can check the value, I have to tell the VDC which object to test for collision. So at the beginning of every frame I write `vdc_coll_spr0 = 001h` into the register `vdc_collision`, which means I want to know what collides with sprite 0 in the frame currently drawn. Then I read the collision register stored in the internal RAM by the IRQ and display it as hexadecimal using a very simple routine not explained here. This value represents the situation as drawn in the last frame. If you move sprite 0 using the joystick you can see which bit of the collision register is responsible for which object. The external collision is used only on Videopac plus machines, there it flags a collision with an object drawn in bright colors. The demo program uses only normal Videopac graphics, so that feature is not demonstrated here.

```
call    waitvsync       ; wait for begin of frame

call    vdcenable       ; enable vdc

; tell the vdc to look for collisions with sprite 0
; in the next frame
mov     r0,#vdc_collision
mov     a,#vdc_coll_spr0
movx    @r0,a           ; activate collision checks

; display the contents of the collision register on
; the right side. this information reflects the
; situation of the last frame.
call    gfxoff
mov     r0,#iram_collision
mov     a,@r0
mov     r0,#vdc_char0
mov     r3,#040h        ; x position
mov     r4,#01Eh        ; y position
mov     r6,#col_chr_white
call    displayhex
call    gfxon
```

This is the first demo program which is longer than one page (256 bytes), so I put all code which uses data in program memory (accessed via `movp`) into an extra page, this includes the routine to set the sprite shape and the display a hex number routine. But this means there is some unused space at the end of page 4, in bigger projects it could be necessary to use this space. The only way to solve this is to shift routines internally until everything fits and every `movp` gets its data from the correct page.

6 How to play the built-in tunes

The collision.a48 (http://soeren.informationstheater.de/g7000/demoprograms/collision.a48) demo program shows another new feature: sound. Every time the fire button is pressed an explosion sound is played. There is a very sophisticated routine in the bios which should be called at every VSYNC. I have named this routine `soundirq`. It is possible to replace this routine completely, but for normal operation you have to put a `jmp soundirq` at 040Ah. This routine interprets a kind of macro code. There are commands for setting sound registers, waiting and jumping unconditionally. This routine reads the sound data from page 3 of the internal ROM, so to play tunes different from the built-in ones you have to replace the routine completely. To start the playing of one of the tunes put the number of the tune into A and call `playsound`. For a description of the tunes look at the summary chapter (page 56) .

```
        call    getjoystick     ; get offsets
        cpl     F0
        jf0     nofire          ; skip next part
                                ; if not fire

        ; start playing the built-in tune for "explode"
        ; the rest is done by the irq routines in ROM
        mov     a,#tune_explode
        call    playsound
nofire
```

7 The clock routines

The BIOS contains routines to manage and display a clock with minutes and seconds. The routines use quad0 and quad1 as display, the time is counted in the external RAM addresses 01h and 02h. This demo (http://soeren.informationstheater.de/g7000/demoprograms/clock.a48) program displays a clock which can be controlled with the joystick. Moving left stops the clock, moving right restarts it. The direction can be changed with moving up and down. To demonstrate how to read back the time the program plays a sound every full minute.

7.1 How does it work

The VSYNC IRQ counts the frames in bits 0-5 of `iram_clock = 03Eh`. If the counter reaches 60 it is reset to 0. This is true even for PAL machines, so the clock is too slow on PAL. Bit 6 = `clock_forward` of `iram_clock` controls the direction, by setting `clock_stop` = bit 7 the clock can be stopped. The routine `doclock` has to be called manually every frame. It tests if the counter is set to 59 and if the clock is active. Then it updates the time in external RAM and the display in quad0/1. Minutes and seconds are stored as BCD numbers. If the time hits 00:00 while counting backwards the clock stops.

7.2 Set start time and initialise quad0/1

To set the start time I simply put the values into external RAM. In this example I set the time to 1 minute and 1 second. This means the first visible time is 01:00, because later I set the frame counter to 58 so the first call to `doclock` will trigger a count and display the correct time in quad0 and quad1. After setting the time I call `initclock` which sets the position on screen and the color of the clock in the control registers of quad0 and 1. Then I call `waitvsync` to make sure that I have enough time between setting the frame counter and the first `doclock`. I then set the frame counter to 58. After that I call `gfxon`. The main loop starts with `waitvsync` so in the meantime the counter is set to 59. This also means that for one frame the displayed time is 01:01. The call to `doclock` then updates the display and decreases the time.

```
        ; need to initialise quad0/1
        call    gfxoff

        ; set the start time = 01:00
        call    extramenable
        mov     r0,#eram_minutes
        mov     a,#01h          ; BCD
        movx    @r0,a           ; 1 minutes
        mov     r0,#eram_seconds
        mov     a,#01h          ; BCD
        movx    @r0,a           ; 1 second, will roll over
                                ; immediately to 0 seconds

        ; initialise the display
        call    vdcenable
        mov     r3,#020h        ; x position
        mov     r4,#040h        ; y position
        mov     r6,#col_chr_white
        call    initclock

        ; activate the clock
        call    waitvsync       ; make sure there is no roll
                                ; over before we want it
        mov     r0,#iram_clock
        mov     a,#03Ah
        mov     @r0,a           ; start clock backwards
```

```
                                ; rolls over at first VSYNC
                                ; and initialises display in
                                ; first doclock
        call    gfxon
```

7.3 The main loop

As usual I use `waitvsync` to make sure the loop executes once per frame. When calling `doclock` it is important to set the Y position and the color to the same values as in `initclock`. If you want to move the clock or change color call `initclock` again.

```
loop
        call    waitvsync       ; once per frame

        mov     r4,#040h        ; y position, same as above !!
        mov     r6,#col_chr_white
        call    doclock         ; update clock, if necessary
```

7.3.1 Reading back the time

After updating the display I test if the seconds are 0. I only want to start the sound playing once although this part is executed every frame, so I have to test the frame counter. Because I don't want to play the sound again and again when the clock stops, I also check if `clock_stop` is reset. To ignore the direction bit I use `0FFh-clock_forward` and not the easier to read `~clock_forward` because ASL has problems with the latter, it is expanded to 16 or 32 bits which is too long to fit into a 8 bit register and ASL is too stupid to cut it to 8 bits.

```
        ; test the time and play sound every minute
        call    extramenable
        mov     r0,#eram_seconds
        movx    a,@r0
        jnz     nosound

        ; only play sound, if count=0 and still counting
        mov     r0,#iram_clock
        mov     a,@r0
        anl     a,#0FFh-clock_forward
        jnz     nosound

        ; now play the sound
        mov     a,#tune_select2
        call    playsound

nosound
```

7.3.2 Controlling the clock

Then I test the joystick. I only explain the test for left/right, up/down is the same only with R3 and `clock_forward`. First I test for left by increasing the X offset. If the joystick was turned left A is now 0. In this case I set the `clock_stop` bit and skip the test for right, because it is impossible to turn the joystick left and right. Now I decrease the X offset twice which means it is now 0 if the joystick is turned right. In this case I clear the `clock_stop` bit. Then I do the same for up/down. At last I put the new value for `iram_clock` back and jump to the beginning of the main loop.

```
        ; test the joystick
        mov     r1,#0
        call    getjoystick
        ; get value to manipulate
        mov     r0,#iram_clock
        mov     a,@r0
        mov     r7,a

        ; test left/right
        mov     a,r2                    ; x offset
        inc     a
        jnz     joy_noleft
        ; left=stop clock
        mov     a,#clock_stop
        orl     a,r7                    ; set clock_stop
        mov     r7,a
        jmp     joy_noright             ; left = not right
joy_noleft
        dec     a
        dec     a
        jnz     joy_noright
        ; right=start clock
        mov     a,#0FFh-clock_stop
        anl     a,r7                    ; clear clock_stop
        mov     r7,a
joy_noright
        ; ...

        ; up/down left out

        ; ...
joy_end
        ; put new value back into iram_clock
        mov     a,r7
        mov     @r0,a

        jmp     loop                    ; next frame
```

8 Bit fields and keyboard

This demo program (http://soeren.informationstheater.de/g7000/demoprograms/bitfield.a48) demonstrates how to manipulate single bits in external RAM or VDC registers. A bit field consists of several bytes in external memory accessed via `movx`. It can be used to mark some members of a group of objects. For example a card deck from which cards can be drawn randomly. To keep track which card is already drawn a bit field can be used.

Another feature explained in this chapter is the use of the keyboard. The program also uses VSYNC IRQ to blink and line interrupts to change colours mid screen, but this is discussed later in the next chapter. (page 18)

All the bit-field routines are not available on the Videopac+ G7400. Any program using them is not compatible with the Videopac+ G7400 and will crash. The keyboard routine is available on both machines.

The bit field used is the horizontal grid, so any changes are immediately visible on the screen. One grid line is active, this line blinks. With the keys "`+`" and "`-`" the next/previous grid line gets active. To set/clear the active grid line use the keys "`1`" and "`0`". So the program is a very simple grid editor, but many things are missing, there is no vertical grid, the last line of the horizontal grid is missing and there is no way to transfer the result to anything. As usual there are no range checks so leaving the screen crashes the system. If you want to finish it and mutate it into a real grid editor, please contact me first, I plan to do it myself sometime.

8.1 The routines

8.1.1 Bit test, bit set, bit clear

All the bit field routines need a pointer to the end of the bit field in R1. The number of the bit to manipulate has to be put into A. After calling one of the routines A contains the old status, R1 points to the byte in which the bit is in and R2 contains a bit mask with the bit set which was requested.

8.1.2 Waitforkey

The keyboard routine is very simple to use, it waits until a key is pressed and returns the key in A.

8.2 The program

As usual I leave out all the initialisations and start with the main loop. I wait until the user presses a key, store it for later use, test for "`+`" and "`-`", adjusting R6 accordingly and test if the active grid has changed.

```
main    ; begin of main loop
        call    waitforkey      ; let user press a key
        mov     r7,a            ; store it for later

        mov     r0,#iram_work
        mov     a,@r0           ; get active bit
        mov     r6,a            ; new bit, if any
        mov     r5,a            ; old bit

        mov     a,r7
        xrl     a,#010h         ; "+" key
        jnz     noplus
        inc     r6
noplus
        mov     a,r7
        xrl     a,#028h         ; "-" key
        jnz     nominus
        dec     r6
nominus

        ; test, if bit changed
        mov     a,r5
        xrl     a,r6
        jz      nonewbit        ; not changed
```

8.2.1 Changing the active grid line

I have to change the active grid line, but the routine which blinks runs in the IRQ, so to avoid problems I disable the interrupts. It is not possible to use `gfxoff` while the interrupts are disabled, because `gfxoff` needs to disable them too and enables them afterwards, so I call it before.

First I set `iram_work` to the new active bit, R0 still points to the right location. The old active grid line was blinking, so I have to set/reset it according to the old saved state in `iram_value`. So I call `bitset` or `bitclear` setting the old grid line to the value stored in `iram_value`. After that I save the contents of the new grid line in `iram_value`. Then I can enable the IRQ again to let the new current line blink.

```
        ; the user wants a new bit
        ; we play with the same registers as our irq
        ; so disable irq while changing bits
        ; we need access to VDC, gfxoff re-enables irq
        ; so turn gfx off before the irq
        call    gfxoff
        dis     i
        ; set new bit active, r0==iram_work
        mov     a,r6
        mov     @r0,a
        ; set old bit to value
        mov     r1,#vdc_gridh8+1
        mov     r0,#iram_value
        mov     a,@r0
        jnz     setoldbit
        mov     a,r5
        call    bitclear
        jmp     nooldbit
setoldbit
        mov     a,r5
        call    bitset
nooldbit

        ; set value to new bit
        mov     r1,#vdc_gridh8+1
        mov     a,r6
        call    bittest
        mov     r0,#iram_value
        mov     @r0,a

        en      i
        call    gfxon           ; outside of dis/en i
nonewbit
```

8.2.2 Set/clear the current bit

I still have to process the "`0`" and "`1`" keys, so I use the key number stored at the beginning, and set `iram_value` to 0 or 0FFh if one of these keys is pressed. The grid line is set to `iram_value` when changing the active grid line by the code above.

```
        ; now test for "0" and "1"
        ; the key which is pressed is still in r7
        mov     r0,#iram_value
        mov     a,r7
        xrl     a,#0            ; "0"
        jnz     nozero
        mov     @r0,a           ; a == 0, because jnz
nozero

        mov     a,r7
        xrl     a,#1            ; "1"
        jnz     noone
        cpl     a               ; a == 0, because jnz
        mov     @r0,a
noone

        jmp     main
```

9 VSYNC and line interrupts

The program bitfield.a48 (http://soeren.informationstheater.de/g7000/demoprograms/bitfield.a48) shows another powerful programming technique: how to extend the interrupt routines. As described in the second chapter (page 3) it is possible to insert code into the standard interrupt routines.

Here I put some code into the VSYNC interrupt which is called at the beginning of every new frame. This code counts the frames and toggles the active grid every 10 frames. It also initialises and enables the line IRQ which I use to change the colours in the middle of the screen.

9.1 VSYNC interrupt

When entering the VSYNC IRQ the main IRQ has already done some things, RB0 is the active register bank, the VDC is enabled and the old state (A and P1) is stored. The registers R3-R7 are used by the standard routine I call later, which leaves only R0-R2 for free use. It is possible to replace the routine completely but then there is no table, no collision check, no waitvsync, no clock and no sound, so I don't recommend doing this. But if you do, the routine should jump to `irqend` at the end, which restores P1 and A, selects RB1 and does a `retr`.

9.1.1 Check if blinking necessary

The code explained here starts after the line IRQ part, see below (page 19) for the first part of the VSYNC. At first I check if I should blink. The blinking can be turned off, because the VSYNC IRQ is always active, even on the "SELECT GAME" screen and I don't want any blinking there. After that I count frames, because I only change state every 10 frames, this is the blink frequency. If this is not the tenth VSYNC, I continue with the original VSYNC.

```
        ; do the blinking
        mov     r0,#iram_ictrl  ; control register
        mov     a,@r0
        jb7     myvsynccont     ; should we blink ?
        jmp     vsyncirq        ; thats all for now

myvsynccont
        mov     a,@r0           ; count the frames
        inc     a
        mov     @r0,a

        anl     a,#03Fh         ; mask out frame counter
        xrl     a,#00Ah         ; wait more frames ?
        jz      myvsyncblink
        jmp     vsyncirq        ; thats all for now
```

9.1.2 Do the blinking

Now that I have to blink, I reset my frame counter, leaving the two control bits intact. Then I have to turn the grid off, because changing the grid while it is turned on gives interesting results, but not what I want. I can't use the normal `gfxoff`, because that switches register banks and I have to restore `vdc_control` later. So I take the `vdc_control` register, store the old value and turn off the grid. Then I get the grid line I have to toggle and test its current state, but only to get a pointer to the corresponding byte in R1 and a bit mask in R2, so I can invert the bit very easy. The only thing left to do is restore the old `vdc_control` value and continue with the built-in routine.

```
myvsyncblink
        mov     a,@r0
        anl     a,#0C0h         ; restart frame counter
        mov     @r0,a
        ; turn grid gfx off, the bios-routine is not safe
        ; to use in irq, it leaves with RB1 and EN I
        mov     r1,#iram_vdcctrl
        mov     r0,#vdc_control
        movx    a,@r0
        mov     @r1,a           ; store old vdc_control
        anl     a,#0F6h
        movx    @r0,a

        mov     r0,#iram_work   ; get the bit to blink

        mov     r1,#vdc_gridh8+1
        mov     a,@r0
        call    bittest         ; test bit to blink
        movx    a,@r1           ; invert the bit we have
        xrl     a,r2
        movx    @r1,a

        ; restore old VDC status
        mov     r1,#iram_vdcctrl
        mov     r0,#vdc_control
        mov     a,@r1
        movx    @r0,a

        jmp     vsyncirq        ; continue vsyncirq
```

9.2 The line IRQ

The 8048 can count external events and produce a timer interrupt after an adjustable number of events have occurred. The timer interrupt is enabled with `en tcnti`, the event counter is activated with `strt cnt`. The timer interrupt occurs when the T register rolls over from 0FFh to 0. At every external event T is incremented. In the G7000 the external events are the HSYNC (Horizontal sync, marks the beginning of a new line) pulses, so T counts the scan-lines on the screen. So if I put eg. 0F8h into T and start the counting, the timer interrupt routine is called 8 scan lines later.

Here I use it the change the background and grid color in the middle of the screen. To do this I activate the line interrupt inside the VSYNC interrupt.

9.2.1 Activating the line interrupt

The first part of code is called at the beginning of the VSYNC IRQ. I test, if I should activate the line IRQ, because I don’t want to change colours on the "SELECT GAME" screen. Then I activate the line IRQ and set the background and grid colours for the top half of the screen.

```
        ; start line irq, if needed
        mov     r0,#iram_ictrl  ; control register
        mov     a,@r0
        cpl     a
        jb6     myvsyncnoline   ; should we start line irq ?
        mov     a,@r0
        mov     a,#088h         ; middle of the screen
        mov     t,a             ; set # of lines to wait
        strt    cnt             ; start line counting
        en      tcnti           ; enable timer irq
myvsyncnoline
```

```
        ; set grid+background color
        mov     r0,#vdc_color
        mov     a,#col_grd_white | col_grd_lum
        movx    @r0,a
```

9.2.2 The line interrupt

This is the timer IRQ part, it is reached via `0404h` when T overflows. At first I select RB0 and store A and P1. Then I enable the VDC and prepare to set the color register. But this preparations take too long and the VDC is in the visible section of the scan line. To prevent ugly flickering I wait until this scan line is completely displayed and change colours before the next one starts. I found the exact number of `nop`s by try and error until the flickering stopped. Due to the different timing you need a different number of `nop`s here for PAL or NTSC machines. The rest is just register restoring and finishing the IRQ.

```
timeirq
        sel     rb0
        mov     r5,a
        stop    tcnt

        in      a,P1
        mov     r6,a
        call    vdcenable

        ; set grid+background color
        mov     r0,#vdc_color
        mov     a,#col_grd_yellow | col_grd_lum | col_bck_blue

        ; wait until end of current scan line to avoid changing
        ; the background in the middle of the screen
        nop
        ; [... 15 nop deleted ...]
        nop
        movx    @r0,a

        mov     a,r6
        outl    P1,a

        mov     a,r5
        retr
```

10 Bank switching

In this chapter I describe both bank switching methods for normal carts: the one to access the full address space of 4K of the 8048 and the one to switch between 4 different address spaces implemented by the G7000. I also talk about more methods which use additional logic in the carts.

10.1 8048 switching: MB0 and MB1

To access the upper 2K of the 8048 address space you have to use the command `SEL MB1`. The next `JMP` or `CALL` command jumps to the address with the address bit A11 set. If you used `CALL` you return as normal with `RET`, but the next `JMP` and `CALL` commands have still bit 11 set. To reset it use `SEL MB0`. You need this mechanisms if your program is bigger than 1K, because the lower 1K is occupied by the BIOS.

Be careful if you use ASL, it automatically inserts switch commands if it thinks they are needed, see the documentation to ASL for details. This automatic fails before a `CALL` to a routine inside the other bank, the banks are switched, but stay switched afterwards. To disable this automatic clear bit 11 on all `CALL` addresses in MB0 and set it on all `CALL` addresses in MB1 and switch banks as needed by hand.

If the 8048 is executing an interrupt routine it is always in MB0, it is not possible to switch to MB1, `SEL MB1` is ignored.

If you want to use the full free address space of 3K remember that nearly all commercial carts don't use A10, so the upper and lower 1K of every bank are identical, which is no problem for MB0 since the lower 1K is ROM. But for MB1 pages 8-0Bh and pages 0Ch-0Fh are identical. This means that including version 0.70 the emulator o2em can't emulate your programs if you use the full 3K. Later versions do support 3K per bank, so this is no longer a problem any more.

10.2 G7000 bank switching: P1.0 and P1.1

The G7000 cart slot has 2 pins which can be used to switch the full address space of the 8048, they are connected to P1 of the processor. When using both pins you have 4 times the address space of the 8048. Since the lower 1K is always the BIOS your programs can use up to 12K with G7000RAM or 8K with o2em.

When switching banks using this method everything is switched including the code that triggered the bank switch and the IRQ routines. To make switching easier the BIOS contains 4 routines for bank switching. After switching all routines jump to 0408h, this is the same address `selectgame` jumps to. The G7000 always starts in bank 3. There are routines to reset one of the bank switch bits called `bank01` and `bank02`, one to reset both bits called `bank0` and one to set both bits, this is called `bank3`. This means you can't switch from bank 1 to bank 2 or vice versa using the BIOS routines. In most cases you need to build your own routine dispatcher at 0408h in every bank to be able to call different routines. Be careful with the IRQ routines, either put them in every bank or disable the IRQ before using banks which don't contain the IRQ routines.

10.3 XROM: "Musician" and "4 in 1 Row": 1 KByte data and 3 KByte code

The Videopac games 31 and 40 use a 4K ROM connected 1:1 to the address lines A0-A11, including A10. So there is code from 0400h-0fffh. The range from 0000h-03ffh can't be used for code as the BIOS has preference there. But the cart also contains a 7400 chip which enables the ROM for read accesses if P11 is 1. This allows data access to the whole ROM from any code position with `movx`. The upper 4 address lines A8-A11 can be set by writing to P2. Both games only read from 0000h-03ffh, but it is possible to read all 4K of the ROM.

Both games do not work on the Videopac+. At reset P11 is already 1, so the XROM circuit is active. On the normal machines this does not matter as the BIOS jumps into the cart code before initialisation, so the program can clear P11. On the Videopac+ the additional graphics chip gets initialised before any code from the cart gets executed, so there is a bus collision when checking if the VPP is busy.

10.4 12/16 KBytes Videopac+ games

Videopac+ graphics data can get quite big, so there are some games which need more ROM space. The games Videopac 55 and 58 are 12K, the games Videopac 59 and 60 are 16K. All four use the same bankswitch method. There are two ROM chips, one 7432, one 7404 and one 74374 inside the cart. The address lines A0-A9 and A11 from the Videopac are connected to the address lines A0-A10 of the ROM chips which results in 2K per bank as usual for commercial carts. The range from 0c00h-0fffh is just a copy of 0800h-0bffh. Any write access to external RAM (P1.4=0, P1.6=0) at address 080h-0ffh stores D0-D2 in the 74374 latch. This is the bank number. The latch gets activated by setting P1.0 to 0. The lower 2 bits of the bank number are the address lines A11 and A12 of the ROM chips, bit 2 decides which ROM chip is used. The 4K ROM in the 12K carts is selected by a 0 in bit 2.

10.5 The MegaCART and FlashCART

Both bank switch methods with additional logic can be combined and easily extended to allow up to 1 Mbyte of ROM space. I use this in my MegaCART (http://soeren.informationstheater.de/g7000/megacart.html) design. The lower address lines A0-A11 are connected 1:1, this gives 3K code from 0400h-0fffh. You can set a bank number for code by writing to `ereg_codebank = 080h` in external RAM (P1.4=0, P1.6=0). Another bank number for the data bank can be written into `ereg_databank = 081h`. The bank numbers contain the address lines A12-A19. Which bank number is used depends on the type of ROM access. The code bank switching is activated by setting P1.0 to 0. Program code is then fetched from the code bank. The XROM read is activated by setting P1.1 to 0. Any `movx` read then fetches its data from the data bank. Additionally P20-P23 have to be set to A8-A11 of the read address. Usually the interrupts are disabled while the XROM read is active (P1.1=0), because the standard interrupt routines try access the VDC but fail because they don't know about the active XROM.

There are two more registers, one for data output `ereg_io_out = 082h` and one for data input `ereg_io_in = 083h`. They are connected to an 93CX6 serial EEPROM. On the FlashCART one of the output bits is used as TX line for the serial port to the PC. The RX line is connected to T0 like on the G7000RAM (http://soeren.informationstheater.de/g7000/g7000ram.html).

A read to any of the 3 output registers returns the last value written to it. All 4 registers are mirrored every 4 bytes on the current MegaCART/FlashCART hardware, but I have vague plans for other expansion hardware there in the future, so only use 080h-083h for access.

The activation of the XROM read with P1.1=0 is incompatible to Videopac 31 and 40, but allows to use the XROM read on Videopac+ machines. As a result you have 768 KByte for program code and 1 Mbyte for data which includes the code space. On the MegaCART all this is implemented by a single CPLD besides one 27C080 EPROM.

Here is a small code example from a slide show program for the Videopac+. It shows how easy it is to handle big data (the uncompressed char and attribute data) using the XROM read. First some useful macros to enable and disable the XROM. The explicit disable is needed, because the BIOS routines `extramenable`, `vdcenable` and `plusenable` don't know about XROM read and don't turn it off. Here I use another macro as replacement for `call plusenable`.

```
; Enable XROM (read and write)
;  P16=0 (enable cart signal), P11=0 (enable XROM), P15=1, P14=1, P13=1, P12=1
m_xromenable macro
        orl     P1,#03ch
        anl     P1,#0bdh
        endm

; Disable XROM reading: P11=1
m_xromdisable macro
        orl     P1,#002h
        endm

; Enable VPP chip
;  P16=0, P15=0, P14=1, P13=1, P12=1, P11=1
m_plusenable macro
        orl     P1,#01eh
        anl     P1,#09fh
        endm
```

This code copies the char and attribute data directly from XROM uncompressed to the screen. It assumes that the `ereg_databank` register is already set. The 25th line in ROM is mapped to the service row.

```
        dis     i               ; XROM reading used here
        anl     P2,#0f0h        ; char data starts at X000h
        mov     r2,#0           ; rom pointer
        mov     r3,#0           ; line number
.draws  mov     a,r3
        mov     r6,a
        ; map 24 into service row
        xrl     a,#24
        jnz     .setrow
        mov     r6,#31
.setrow mov     r7,#plus_cmd_brow
        call    pluscmd
        mov     r4,#40
.drawl  m_xromenable
        mov     a,r2
        mov     r0,a
        movx    a,@r0           ; read char from XROM
        mov     r7,a
        inc     r0
        movx    a,@r0           ; read attribute from XROM
        mov     r6,a
        m_plusenable
        call    plusdata
        ; advance ROM pointer
        inc     r2
        inc     r2
        mov     a,r2
        jnz     .skipp2
        in      a,P2
        inc     a
        outl    P2,a
.skipp2 djnz    r4,.drawl       ; loop until line complete
        inc     r3
        mov     a,r3
        xrl     a,#25
        jnz     .draws          ; loop until service row (mapped to 24)
        en      i
```

11 Other useful routines not yet explained

In this section I will explain several useful routines which I have not used in any of the demo programs.

11.1 Routines for characters

The first routine is very simple, it is called `clearchar` and clears all characters. This means all characters on the screen disappear. The grid, sprites and quads are not affected.

If you look at the description for the character registers you will see that byte 2 and 3 are not easy to calculate, but fortunately there are several routines for that in the BIOS. At first there is `calcchar23`, you feed it with the Y position in R4, the character to display in R5 and the color in R6. It tells you what to put into byte 2/3 in R5/R6. A very similar routine is `putchar23`, it is a front-end to `calcchar23` and puts the register data into the registers, so it needs a pointer to a character register byte 2 as additional parameter in R0. Another front-end for `calcchar23` is `tablechar23`, it puts the register data into a table, the order of the bytes is reversed.

11.2 Routines for tables

There are some more routines which deal with tables in the BIOS, the simplest one puts a 2 into the table at R0. It is called `tableput2`. Then there is `tablebcdnibble`. It puts the lower nibble of A as a BCD number into the table for use as a character. The length and start register have to be put into the table manually before calling `tablebcdnibble`. There is a front-end to `tablebcdnibble` called `tablebcdbyte`. It initialises the table, processes the parameters, ends the table and activates it. The BCD number to display has to be put into A, a pointer to the first character into R1. As usual the screen position is in R3/R4 and the color in R6.

11.3 Routines for mathematics

There is one routine for multiplication and one for division in the BIOS. They are called `multiply` and `divide`. Both use repeated addition/subtraction, so they are not very fast. The results are limited to 8 bit, there are no checks for overflow, so be careful with the results.

11.4 Routine for initialisation

There is one routine which initialises nearly everything called `init`. It can be used if you don't want to display the "SELECT GAME" message. Just make sure to switch to RB1 before using it if you want to use the standard IRQ routines or other routines which switch register banks, there are several. If you use the keyboard routines put 0FFh into R7 at RB0 first.

11.5 Routines for random number generation

There are two routines which can be used to generate random numbers. They use the T register for this purpose, so you have to start the timer with `STRT T` once at the beginning. This implies you can't use the line interrupt, the random numbers would depend on the screen position where you get them. The routine called `random` generates two random nibbles in R2 and R3=A. Before you call this routine you have to set R2 and R7 to the 1's complement of the biggest number you want to get. The number returned in R2 is less or equal to the 1's complement set in R7, the number returned in R3 is less or equal to the 1's complement set in R2. Note that the number in R2 does not depend on what you put in there before the call, thats R7.

There is a helper routine called `nibblemixer` which takes a nibble in R5 and mixes it until it is less or equal to the 1's complement of R7. For this purpose it uses a ROM table starting at `02B3h`. This routine does not depend on T, the result is always the same.

This routines are not very useful, you should better implement your own random number generator eg. using primitive polynomials (also known as LFSR).

This routines are not available on the Videopac+ G7400 which is just another argument against them. This incompatibility is the reason why Videopac 5 (Blackjack) hangs when shuffling the cards.

12 Custom sound player

Here in this chapter I explain routines for a VSYNC interrupt driven custom sound player (http://soeren.informationstheater.de/g7000/demoprograms/tuneplay.a48). This means that you can create your own sound effects that are still playable with `playsound`. It is possible to keep or to replace the built-in tunes. The piece of code shown here is used in several variants by a lot of the commercial games. The one I explain here supports loading the sound shift registers with any value which is a feature that is only used by a few of the commercial games. Most game leave out the code handling it. The demo tune included into the full example does not use it, I explain it just to be complete.

12.1 The sound hardware

The sound hardware is just a 24 bit shift register with one end connected to the sound output. This 24 bit register is split into the 3 separate byte wide registers `vdc_sound0/1/2`. The sound is controlled by the `vdc_soundctrl` register. The highest bit switches the sound output on or off. The next bit enables a loop mode, the sound output will be rotated back into the other end of the shift register when this bit is set. Otherwise the sound will stop after all 24 bits are sent. Bit 5 is used to select between two different shift frequencies. Another source of sound is a noise generator enabled with bit 4. The volume of the sound output is controlled by the last 4 bits inside the `vdc_soundctrl` register.

The number of sounds the hardware can create is very limited and creating a tone with a certain frequency is beyond the scope of the BIOS code. It is possible to generate a certain frequency by modulating a noise signal: Generate an exact timing with the timer and a software timing loop and toggle the sound enable bit, this is used by Videopac 31 (Musician). But here I explain tunes that the BIOS routines can generate.

12.2 The BIOS tune player

The tunes played by the BIOS use commands interpreted by the BIOS tune player as long as bit 6 of `iram_irqctrl` is set. The pointer to the current tune command is set by `playsound`. This pointer points into bank 3, so the tune for the "SELECT GAME" sound starts at 034Ah. The custom tune player just gets the data from another bank. It is obviously possible to overwrite the built-in tunes as desired, but this example keeps them intact. Every VSYNC interrupt the tune player is called, but most of the commands have a duration. Only after this duration is over a new command is loaded by a jump to 040Ah, which normally jumps to `soundirq`.

12.3 The tune player opcodes

There are 4 commands supported by the BIOS. Three of the commands have a parameter. The commands are differentiated by the highest bit that is set. The custom tune player here supports another sound opcode to remove a limit by the BIOS routines that is only relevant when creating custom tunes.

12.3.1 Play tone

If bit 7 of the command byte is set a new tone starts. The duration of this command is the command byte with bit 7 cleared. The parameter byte is a pointer to the new contents for the shift registers and `vdc_soundctrl`. They are always read from bank 3. There are 10 different tones from 0300h-0327h, but the BIOS tunes only use 8 of them. If you want more than those 10 tones you have to create your own opcode as shown later.

12.3.2 Load control

If bit 7 of the command byte is 0 and bit 6 is set the control register is loaded. The duration is the command byte with bit 6 cleared. The parameter byte is the new value for `vdc_soundctrl`. This command is used to change the sound volume and to play sound with noise.

12.3.3 Silence

If the bits 7 and 6 are 0 and bit 5 is set the sound is turned off. The duration is the command byte with bit 5 cleared. This command does not have a parameter byte.

12.3.4 Jump

If the bits 7, 6 and 5 are 0 and bit 4 is set the current tune position is set to the parameter byte and the tune player is restarted. It is possible to create endless sounds with this opcode. To play another sound just call `playsound` again.

12.3.5 Stop sound

If the BIOS does not recognise a command it turns the sound off and stops playing. The BIOS tunes use 0 for this.

12.3.6 Opcode 00Fh

If this custom tune player reads a command byte 00Fh it turns off any sound and copies the next 3 bytes into the sound shift register. After that the tune player is restarted, the next command executes immediately. This command should load `vdc_soundctrl`, because the sound is still turned off from loading the shift register.

12.4 The program

As usual I only show the relevant parts of the program, the full program can be downloaded as usual.

12.4.1 The IRQ vectors

To create custom sound tunes the first part of the BIOS sound IRQ has to be replaced, the place to do this is at 040Ah:

```
        jmp     selectgame      ; RESET
        jmp     irq             ; interrupt
        jmp     timer           ; timer
        jmp     vsyncirq        ; VSYNC-interrupt
        jmp     start           ; after selectgame
        jmp     mysoundirq      ; sound-interrupt
```

12.4.2 Feeding the BIOS player with new data

The 8048 is still in IRQ mode, so `SEL MB1` is not possible and the BIOS has switched to RB0. First we have to see if the tune currently played is one of the built-in tunes. Here I leave all built-in tunes intact, the new ones start at 076h. The current tune position is delivered in R4, if it is smaller than 076h the old BIOS routine is called.

```
mysoundirq
        ; check if BIOS sound or custom sound
        mov     a,r4
        add     a,#08ah         ; >= 076h
        jc      .custom
        jmp     soundirq        ; BIOS tune
```

Now I just read a byte as command byte and test for the new 00Fh opcode. If it is not the new opcode I read another byte as parameter byte. The rest is handled by the BIOS in `parsesnd`.

```
.custom ; custom sound handler, read sound opcodes from current page
        mov     a,r4
        movp    a,@a
        mov     r1,a            ; command byte
        inc     r4
        xrl     a,#0fh
        jz      .op0f           ; test for new opcode
        mov     a,r4
        movp    a,@a
        mov     r2,a            ; parameter byte
        jmp     parsesnd        ; let BIOS sound IRQ handle opcode
```

There is no BIOS routine for the new opcode 00Fh, so I have to do it all by hand. First the sound gets turned off, because I change the sound shift register. Then the next 3 bytes are copied into the shift register and the sound IRQ handler is started again to execute the next sound opcode.

```
        ; opcode 0F: sound off, copy next 3 bytes into A7/8/9
.op0f   mov     r0,#vdc_soundctrl
        clr     a
        movx    @r0,a           ; old sound off
        mov     r0,#vdc_sound0
        mov     r1,#3           ; number of bytes to copy
.loop   mov     a,r4
        movp    a,@a
        movx    @r0,a
        inc     r0
        inc     r4
        djnz    r1,.loop
        jmp     mysoundirq      ; restart sound handler
```

13 Differences between PAL and NTSC machines

13.1 Video standards: PAL and NTSC

In different areas of the world different video standards are used for broadcasting TV signals. The most common ones are NTSC and PAL. The main difference between them is the way the color information is encoded, but they also have different line resolutions and frame rates. When taking a simplified view a NTSC signal has 60 frames per second with 262 lines each, PAL has 50 frames with 312 lines each. It is a bit more complicated in reality, because normal TV broadcast is interlaced, only half of the picture is drawn at once, half of the lines are black and filled in by the next field. To differentiate between them one of the fields is one line longer than the other and a full frame of two fields has 525/625 lines. The Odyssey² and the Videopac G7000 both generate a non interlaced display, so I will use the 262/312 lines value. When broadcasting there are even several sub-standards dealing with the sound encoding into the video signal, but these are not relevant for programming. There is another video standard called SECAM which uses another way to encode color, but it uses 50 frames and 312 lines, so it is very similar to PAL. I think the Jopac consoles are the SECAM version of the Videopac G7000, but I don't have any more information about them. If you know something about them please E-Mail me.

13.2 Odyssey² and Videopac G7000

The Odyssey² generates a NTSC signal, the Videopac G7000 a PAL signal. The Odyssey² is the base model, the Videopac G7000 has a lot of additional logic to generate a PAL compatible signal. Both consoles use a single Quartz to generate several clock signals, including the CPU clock. This means all internal clocks differ slightly, which has a big impact on programming. The Videopac+ G7400 uses different clock sources for CPU and VDC, so there is no fixed relationship between CPU and VDC clock. The frequencies are the same as for the Videpac G7000.

13.3 The internal clocks

	Odyssey²	**Videopac**
Main clock	7.15909 MHz	17.734476 MHz
VDC divider	2	5
VDC clock	3.58 MHz	3.55 MHz
CPU divider	4/3	3
CPU clock	5.37 MHz	5.91 MHz
Instruction cycle	2.79 µs	2.54 µs
Lines per frame	262	312
Frames per second	60	50
Instructions per line	342/15 = 22.8	380/15 = 25.3
Lines of VBLANK	22	72
Instructions per VBLANK	≅500	≅1820

13.4 Consequences for the programmer

It is common to use the frame as basis for all timing information. This means that the same code runs 20 % faster on NTSC machines. But there are less CPU clocks available per frame, too. Even worse is the much shorter VBLANK time on NTSC machines. To be able to change VDC objects you have to turn the graphics off. So the only time to change graphics without disturbing the picture is inside VBLANK, but this is 3 times longer on PAL. Additionally the CPU runs slower on NTSC, making the situation worse again.

Coding only for PAL gives you much more CPU power, so it is possible to create more complex games which only work on PAL machines. But if the game is simple enough you should try to make it run on NTSC, too. So for development it is better to test your code on NTSC machines, if it has no timing problems there it won't have any on PAL. The only exception is code that relies on exact timing, like eg. changing the background on the fly. This code needs to be different on PAL and NTSC.

If you are writing code which only runs on the Videopac+ G7400 I think it is ok to ignore any NTSC issues. The Odyssey[3], the NTSC version of the Videopac+ G7400 only exists as several prototypes, and I think anybody owning one will have a Videopac+ G7400, too.

13.5 Detecting PAL or NTSC: VBLANK

This program (http://soeren.informationstheater.de/g7000/demoprograms/palntsc.a48) shows how to detect if your program is running on a PAL or NTSC machine. I only explain the detection routine here, the interpretion and drawing parts are left out. The program is based on an idea by from René van den Enden.

13.5.1 How does it work

This program checks the length of the VBLANK pulse. This pulse can be seen on the external input T1 starting at VSYNC. So this routine waits for VSYNC and counts in a loop until T1 is 0 again. The result is the loop counter, it is 0d6h on PAL and 034h on NTSC. This program uses the standard interrupt routines, adding code into the VSYNC or sound interrupt will influence the results. That is also the reason why it waits until all sounds have finished.

13.5.2 The program

```
start
        ; wait for end of keyclick
        mov     r0,#iram_irqctrl
        mov     a,@r0
        jb6     start

        call    waitvsync       ; wait until vsync

        ; count until pulse ends
count   clr     a               ; init counter
loop    inc     a
        jz      error           ; counter overflow
        nop                     ; prevent overflow on PAL
        nop
        nop
        jt1     loop
error   mov     r1,a

show    ; show result, PAL: D6, NTSC: 34
```

14 The Videopac+ G7400

To support the additional graphics and the new "SELECT GAME" screen the BIOS of the Videopac+ G7400 is modified at 153 bytes. To make room for this modifications the routines for bit-fields and random number generation were removed.

Here I present vpplus.a48 (http://soeren.informationstheater.de/g7000/demoprograms/vpplus.a48) to show how to use the Videopac+ G7400. Most of the code is quite simple and not explained further. It displays text with different attributes. Use the key "C" to toggle conceal mode, "B" to box mode, "L" to toggle blinking and "V" to display a VDC character over the boxed text.

14.1 The additional hardware

The new hardware consists of the EF9340/EF9341 display processor designed by Thomson Semiconductors. I have found a data-sheet at Alldatasheet.com (http://www.alldatasheet.com/). You have to enter EF9340 into the search field yourself, a direct link did not work for me. If you know other sources please let me know.

The chip is designed for Teletext like displays. It can display a 40×24 text mode with 8 colors. The size of the chars are 8×10 pixels. It is possible to redefine up to 96 chars with default background color and another 96 chars with explicit background color. The first line of the display is called service row and quite independent from the rest of the display. The chip supports 2 different kind of attributes, parallel and serial. Parallel attributes can be set with every char, changing serial attributes requires one char space. Parallel attributes are text color, double height and width, inverted display, blinking. Block graphic chars support background color as parallel, for normal chars the background color is a serial attribute. Other serial attributes are concealed, box mode and underlining. Concealed chars can be hidden by setting a control bit. In box mode it is possible to make parts of the display transparent. This is not used on the Videopac+ G7400, you can only use box mode to turn the screen black outside the box zone. If pin B of the cartridge is not grounded even the VDC graphics are blanked outside the box zone. This can be used to apply the black border that is normally around plus graphics to VDC graphics. On o2em you can emulate this effect by putting the ASCII string "OPNB" at 040Ch.

14.2 Discovering the machine type

To find out on what kind of machine your program is running call `plusselectgame` instead of `selectgame`. Due to a clever positioned `nop` both routines are identical on the non-plus Videopac. On a Videopac+ G7400 the program continues in bank 1, not in bank 3 as usual. There you can set up some background graphics just like the games which were re-released for the VP+. After that you can jump to `plusstart` which simulates the result of the normal `selectgame` routine. Instead of setting up just one background for the whole game you could also just set a flag in RAM and use this later to display VP+ graphics.

14.3 Mixing VDC and VP+ graphics

The VP+ graphics is always behind all other graphic objects. To be able to see the VP+ graphics it is possible to define any combination of the 8 VDC colors as transparent. This is done by a routine called `plusmode`. Every bit in R7 corresponds to a VDC color. If the bit is 0, the color will be transparent. With A you can set the luminance of the 8 VP+ colors in a similar way, if the bit is 0 the color gets brighter. Any object drawn in a bright color is active for collisions with normal videopac objects. The collisions are flagged as external, `vdc_coll_ext`. A shortcut to hide all VP+ graphics is called `plushide`.

14.4 Drawing characters

Before you can access the VP+ graphic registers you have to call `plusenable`, just like for VDC and external RAM. Then you can call `plusdata` to draw chars on the screen. In R7 you put the ASCII code of the char you want to draw. From 000h-01Fh there are some accented chars. The range from 020h-07Fh is mostly normal ASCII. The chars 0A0h-0FFh are user definable. This leaves 080h-09Fh for the serial attributes.

In R6 you have to put the parallel attribute for the char. Bits 0-2 define the color of the char. Setting bit 3 (`col_patr_stable`) prevents the char from blinking. If bit 4 (`col_patr_dhght`) is set the char is displayed with double height. To double the width set bit 5 (`col_patr_dwdth`). The EF9340/41 chips have some limitations when doubling chars, they can't display top and bottom parts in the same line and left and right parts in the same column. With bit 6 (`col_patr_invrt`) it is possible to invert the char.

Setting bit 7 (`col_patr_blck`) changes to block graphics. Bits 4-6 of the attribute are now the new background color for the rest of the current line, so you can neither double nor invert block gfx chars. Bit 3 still prevents blinking. The char byte has a new meaning, too. From 000h-07Fh bits 0-5 form a 2×3 bitmap. If bit 6 (`plus_blck_full`) is cleared there are small lines between the blocks, if it is set the blocks touch each other. 080h-09Fh are marked as ILLEGAL in the data-sheet, so do not use them. 0A0h-0FFh are the second set of user definable chars, they have the background color as parallel attribute.

14.5 Setting the cursor position

By calling `pluscmd` you can send several commands to the VP+ graphics engine. Among them are commands to move the cursor.

To set the cursor to the first column in any line set R7 to `plus_cmd_brow`. Put the line number into R6. The first line (service row) is special, its position is 01Fh. The second line is numbered 0 (or 020h). Similar is `plus_cmd_loady`, the y position is set, but this time the x position is unchanged. To set the x position of the cursor set R7 to `plus_cmd_loadx`. Set R6 to 0 for the first column, to 1 for the second column etc. The last command to move the cursor is `plus_cmd_incc`. This one advances the cursor one char.

14.6 Command load R: some global parameters

This command sets the R register. It has its own routine called `plusloadr`. It takes the display mode in A. This mode is composed by 8 single bits. Setting bit 7 (`plus_loadr_blnk`) enables the blink mode, everything where `col_patr_stable` is not set blinks. Bit 6 in the R register switches between 50 and 60Hz mode. This is handled by the BIOS, it gets always set on my machine which means 50Hz. Bit 5 should always be left to 0, it changes the width of the HSYNC pulse. Setting bit 4 (`plus_loadr_crsr`) enables a visible cursor. Bit 3 (`plus_loadr_srow`) enables the display of the service row (the first line), bit 0 (`plus_loadr_dspl`) enables the rest of the display. When setting bit 2 (`plus_loadr_conc`) all chars with the `col_satr_conc` attribute set are hidden. Setting bit 1 (`plus_loadr_box`) hides all chars except the ones with `col_satr_box` set.

This routine is only safe when inside the VBLANK period. Otherwise parts of the screen memory get trashed. So always call `waitvsync` immediately before using `plusloadr`.

14.7 Scrolling with plus_cmd_loady0

This command sets the line number which gets displayed as second line. So you can scroll lines very easily, but the first line (service row) always stays as the first line. Set the line number into bits 0-4 of the parameter byte and set bit 5 (`plus_loady0_zom`) to display every line (except the first) with double height. Characters which have `col_patr_dhght` set are doubled again, so they are now 4 times as high as normal.

14.8 Using plus_cmd_loadm: user definable chars

There are 2 different ranges for user definable chars. One with doubling and inverting as parallel attributes and one with background color. To define a char you have to put it under the cursor. Normally the cursor advances to the next screen position when printing a char. To turn this off send the command `plus_cmd_loadm` with the parameter `plus_loadm_wrni`. Then print the char you want to redefine. Now use `plus_cmd_loadm` to switch to `plus_loadm_wrsl`. The next 10 bytes written with `plusdata` define the char. To redefine the next char switch back to `plus_loadm_wrni`, print it, switch to `plus_loadm_wrsl` and send the next 10 char bytes. To put the cursor back to auto advance use `plus_loadm_wr`. Here is some code to redefine the char 0A0h into a smiling face:

```
        mov     r7,#plus_cmd_loadm
        mov     r6,#plus_loadm_wrni
        call    pluscmd         ; do not move cursor
        mov     r6,#000h
        mov     r7,#0A0h
        call    plusdata        ; print char to define
        mov     r7,#plus_cmd_loadm
        mov     r6,#plus_loadm_wrsl
        call    pluscmd         ; char bitmap follows
        mov     r2,#chardata & 0FFh
        mov     r3,#10
.char   call    getchar
        mov     r6,a
        mov     r7,a
        call    plusdata
        inc     r2
        djnz    r3,.char

        ; back to normal
        mov     r7,#plus_cmd_loadm
        mov     r6,#plus_loadm_wr
        call    pluscmd         ; draw normal chars

        [...]

; get char data
getchar mov     a,r2
        movp    a,@a
        ret

chardata
        db      00111100b
        db      01000010b
        db      10000001b
        db      10100101b
        db      10000001b
        db      10100101b
        db      10011001b
        db      10000001b
        db      01000010b
        db      00111100b
```

If you have to define a lot of chars it is faster to turn off the display by clearing `plus_loadr_dspl` and `plus_loadr_srow`. That way the chip has more bandwidth available on its bus.

14.9 More plus_cmd_loadm: reading data

For any of the aforementioned `plus_loadm_` write modes there is a corresponding read mode, but there is no routine to read the data in the BIOS. So you have to read it manually. The data-sheet is not very clear is this aspect, it seems that the chip needs a dummy read from its TB register to fetch the data. Additionally you have to check the busy flag by calling `plusready`. Here is some example code how to use `plus_loadm_rdni`. This is just my test code, it is not included into any demo program.

```
        mov     r7,#plus_cmd_loadm
        mov     r6,#plus_loadm_rdni
        call    pluscmd
        call    plusready       ; wait until mode switched
        mov     r0,#vpp_tb_rd
        movx    a,@r0           ; trigger read
        call    plusready       ; wait until read finished
        mov     r0,#vpp_ta_rd
        movx    a,@r0           ; TA
        mov     r2,a
        inc     r0
        movx    a,@r0           ; TB
        mov     r3,a
```

14.10 Writing directly to the VPP

If you need the access the VPP as fast as possible you can access the registers directly. This avoids the `call` and `ret` overhead. First you have to wait until the chip is ready. Then you can send one command with parameters to the chip. The command/parameter combination is 16 bits wide. The VPP executes the command when the TB register part is written. So you need to write to the corresponding TA register first. The chip is busy until the command is finished, but you only need to check this before trying the next command. The following piece of code draws an A at the position R3/R4:

```
        ; position the cursor at the beginning of line R4
        mov     r0,#vpp_busy            ; busy flag at bit 7
.busy1  movx    a,@r0
        jb7     .busy1                  ; wait for mailbox ready
        mov     r1,#vpp_ta_cmd          ; command register
        mov     a,r4
        movx    @r1,a
        inc     r1
        mov     a,#plus_cmd_loady
        movx    @r1,a                   ; vpp_tb_cmd
        dec     r1

        ; position the cursor at column R3
.busy2  movx    a,@r0
        jb7     .busy2                  ; wait for mailbox ready
        mov     a,r3
        movx    @r1,a                   ; vpp_ta_cmd
        inc     r1
        mov     a,#plus_cmd_loadx
        movx    @r1,a                   ; vpp_tb_cmd

        ; write a white ’A’
.busy3  movx    a,@r0
        jb7     .busy3                  ; wait for mailbox ready
        mov     r1,#vpp_ta_wr           ; write to data register
        mov     a,#col_plus_white       ; attribute byte
        movx    @r1,a
```

```
inc     r1
mov     a,#041h                 ; char ’A’
movx    @r1,a                   ; vpp_tb_wr
dec     r1
```

14.11 Questions answered so far

Some things were still unclear, but with the help of René van den Enden and some more experimentations I can now answer them:

14.11.1 Is the VDC display really always in front or can the box mode be used to put it behind?

The VDC is always in front. VDC graphics outside box mode can be blanked if pin B of the cartridge is not grounded. To use this feature with o2em you have to put "OPNB" at 040Ch into bank 1.

14.11.2 Why is there 6 KByte RAM when only 4 KByte are necessary?

Only 4 KByte are used, two of the RAM chips are only used as 1 KByte chips. The reason is probably that 2 KByte SRAM chips are much more common than 1 KByte.

14.11.3 How does A in plusmode affect the interrupts?

It does **not**, it was a very stupid programming error on my side.

15 Tips and tricks

Here I document some useful tricks I discovered while learning to program the G7000 / Odyssey² with asl. The code examples shown here are not directly taken from a real program, although I have used them all while experimenting with the machine. So there are no demo programs, just these code snippets.

15.1 Argh! Intel forgot the subtraction opcodes

Yes, the 8048 doesn't have any commands for subtraction. This is not a big problem, just basic knowledge about computer arithmetics. But it took me some time to figure it out after I saw it in the BIOS. The trick is to calculate -R2 + R1 instead of R1 - R2. For -R2 I have to use 2's complement: invert all bits and add 1. So the full code to calculate R1 - R2 is:

```
        mov     a,r2    ; second operand
        cpl     a       ; invert bits
        inc     a       ; add 1: a contains -R2
        add     a,r1
```

15.2 BCD arithmetic

One feature of the 8048 is quite useful for numbers which need to be displayed: The 8048 can use BCD arithmetics. All you need to do is interpret numbers in hex as decimal: so 010h represents 10. This way it is much easier to display the numbers, just take the upper and lower 4 bits and display them separately, you no longer have to divide by 10. To calculate in BCD first add the numbers as usual. The 8048 has an auxiliary carry flag that shows overflows from the lower to the higher nibble. This information is now used by the `DA A` opcode to correct any non-BCD range nibbles. The A register contains the correct result in BCD afterwards, the carry flag is set if necessary. So this is code to calculate 18 + 89:

```
        mov     a,#018h
        add     a,#089h ; now in a: 0a1h
        da      a       ; now in a: 007h, carry set
```

15.3 Not all commands change the carry flag

This trick works on nearly all microprocessors, not only on the 8048. The carry flag is only changed by some commands, mainly the addition commands. So you can insert some code statements between the carry changing command and the `jc`/`jnc`. For example:

```
        add     a,#010h         ; set carry, when a>=0f0h
        mov     r0,#020h        ; carry not changed
        mov     a,@r0           ; carry not changed
        jc      somewhere
```

This trick only works with the carry flag. There is no zero flag, `jz` jumps when A is 0.

15.4 Saving pointers: aligning internal & external RAM

If you run out of internal RAM and have to put some data into external RAM you should try to put related data into the same addresses. That way you only need one pointer register:

```
iram_x  equ 020h                ; x position
eram_y  equ 020h                ; y position

        mov     r0,#iram_x      ; pointer to position data
        mov     a,@r0           ; x position in internal ram
        mov     r3,a
        movx    a,@r0           ; y position in external ram
        mov     r4,a
```

15.5 Cutting off the top of chars

It is possible to only display the bottom parts of characters. To achieve this you have to fill in the char bytes per hand instead of calling `printchar`. Bytes 0 and 1 are simply just the y and x position. For bytes 2 and 3 you can call `calcchar23`. For every line you want to cut off from the character you have to increase the char-map pointer in r5 by 1. You may have to manipulate bit 0 in r6 also as this is the 9th bit of the char pointer. This depends on the exact y position on the screen and can be omitted if the char does not move around as in this example, which displays a vertical bar by cutting off the horizontal part of a T:

```
        mov     r4,#GROUNDLEVEL - 6 ; y position
        mov     a,r4
        movx    @r0,a
        inc     r0
        mov     a,r3
        movx    @r0,a
        inc     r0
        mov     r5,#014h            ; T
        mov     r6,#col_chr_yellow
        call    calcchar23          ; calculate byte 2/3
        mov     a,r5                ; get char-pointer
        inc     a                   ; advance one line...
        movx    @r0,a               ; ...to cut off first line of T
        inc     r0
        mov     a,r6                ; get byte 3
        movx    @r0,a               ; store it
        inc     r0
```

This trick is used in many of the commercial games, e.g. the enemies in Killer Bees are constructed by combining the lower parts of 8 and the feet of the running mans.

15.6 Cutting off for quads

Advancing the char-map pointer does work for quads, too. The number of lines that are drawn are controlled by the last sub-quad. The other 3 sub-quads draw exactly the same number of lines, the normal character ends are ignored. So you can cut off the bottom of the first 3 sub-quads, just set the last sub-quad to a space with the number of lines to display. The first 3 sub-quads are now cut off at the bottom when used without offset. With offset you can move freely inside the whole char bitmap, if one character is finished the next one is drawn with a one pixel empty line between them.

This trick does not work in o2em version 1.01.

15.7 Using P1.7 for bright background colors

If you clear port 1 bit 7 dark colors are made brighter. This means that the colors for background and dark grid are now identical to the colors for chars, quads, sprites and bright grid. Clearing P1.7 is not easy, all BIOS routines which touch P1 set it. Among these routines are `vdcenable`, `extramenable`, `getjoystick` and the interrupt routines. So to keep P1.7 cleared you have to replace a lot of BIOS routines with your own versions.

15.8 Sections and local labels

One feature of asl is the possibility to divide programs into sections. They start with `section name` and end with `endsection name`. All labels inside a section are local to the section, so you can reuse common names like loop etc:

```
        section code1
        mov     r2,#3
loop    ; do something here
        djnz    r2,loop
        endsection code1

        section code2
        mov     r2,#4
loop    ; do something different here
        djnz    r2,loop
        endsection code2
```

More info about this can be found in the documentation of asl. This reuse of local labels is something that I have found convenient. One could argue that having labels with the same name is a source of confusion, but I like them.

15.9 Using asl features to prevent page misses

One of the problems any programmer on the G7000 / Odyssey² has to fight is the page alignment of ROM data and code which accesses it. One way to provoke error messages when assembling works by using subtracting the high byte of the current ROM position multiplied by 256 from the start of the table to get the low-byte. But this only keeps the start of the table in the same page. So at the end of the table I need to compare the high-bytes of the first and last table entry. For that I use conditional assembly to generate an error message if they are not equal.

```
addscore section addscore
        add     a,#scoretable - hi($)*0100h
        movp    a,@a
        ; ... lines snipped
        ret
        endsection addscore

scoretable
        db      5,15,10

        if (hi(scoretable) <> hi($))
        error "scoretable crosses page border"
        endif
```

15.10 Defining and using an assert macro

A more versatile way to achieve the same result is the usage of an assert macro. It takes an expression included into "" as argument. At assemble time this argument is evaluated. If it is not true an error message is generated.

```
assert macro expr
        if (~~val(expr))
        error expr
        endif
        endm
```

Here is the example from above, this time using the assert macro.

```
scoretable
        db      5,15,10

        assert "hi(scoretable) == hi($)"
```

16 About the hardware

In this chapter I give a short overview about the video and sound hardware of the Videopac G7000 / Odyssey². For a much more complete overview see the "Odyssey² Technical Specs" at the home page (http://atarihq.com/danb/o2.shtml) from Daniel Boris.

16.1 P1: selection for `movx` access

The most important thing the 8048 port P1 controls is which chip is accessed by `movx`. It also supplies 2 address lines used for bank-switching (page 21) and one signal to control the luminance of the video output.

P1.0 and P1.1	These signals are directly connected to the cart port pins 13 and 12. On most of the carts they are used as address lines A12 and A13. Some carts use them as enable signals for special bank switch methods.
P1.2	This signal is used to enable the keyboard decoder.
P1.3	This is the select signal for VDC accesses. Setting it to 0 selects the VDC. Read accesses to the VDC need P1.6 to 0, too.
P1.4	This is the select signal for RAM accesses. Setting it to 0 selects RAM as external memory. For write accesses to RAM P1.6 has to be 0, too. This signal is also available at the cart port on pin 11.
P1.5	This signal is not used by the normal Videopac G7000 / Odyssey², but it is used as select signal for the EF9340/41 VPP chipset in the Videopac G7400. Setting it to 0 enables the VPP.
P1.6	This signal is used to be able to enable both RAM and VDC at the same time. Setting it to 1 turns off VDC read and RAM write accesses. Write accesses to the cart port get disabled, too. This allows you to read from RAM and write to VDC at the same time. The bios routines using the extram tables use this feature.
P1.7	This signal controls the luminance output of the VDC. See the chapter about tips and tricks (page 37) for details.

16.2 VDC control, status and color

16.2.1 `0A0h`: `vdc_control`

This is the main control register of the VDC. It is a write-only register.

Bit 0	This bit enables an interrupt for every line on the screen.
Bit 1	This bit enables the position counters in the `vdc_scanline` and `vdc_scanrow` registers.
Bit 2	This bit enables an interrupt for sound.
Bit 3	If this bit is 1 the grid is displayed. If it is 0 the grid can be changed. If you change the grid registers while this is 1 the grid is **not** set to the correct values.
Bit 4	Unused.
Bit 5	If this bit is 1 sprites, chars and quads are displayed. If it is 0 they are turned off and can be changed. Changing sprites, chars and quads while this bit is 1 seems to have no effect.
Bit 6	This bit can be used to display a fixed dot grid.
Bit 7	If this bit is set the vertical grid lines are extended to the right until the next grid line starts. So instead of lines you get boxes.

16.2.2 0A1h: `vdc_status`

This is the VDC status register. It is read-only.

Bit 0	This bit shows if the VDC is currently drawing a line or if it is in the HBLANK phase.
Bit 1	Shows if the scan counters `vdc_scanrow` and `vdc_scanline` are active.
Bit 2	Shows if the sound register is empty.
Bit 3	This bit show the start of the VBLANK.
Bit 4	Unused.
Bit 5	Unused.
Bit 6	Unused.
Bit 7	This bit is set when 2 or more chars/quads are overlapping each other.

16.2.3 0A2h: `vdc_collision`

This read/write register is used for collision detection. At VBLANK you need to set the bits for the objects you want collisions reported for. One frame later you can read what types of objects have collided. Here is an example: If you set only bit 0 and sprite 0 touches one of the chars one frame later the bits 0 and 7 are set.

Bit 0	This is the collision bit for sprite 0.
Bit 1	This is the collision bit for sprite 1.
Bit 2	This is the collision bit for sprite 2.
Bit 3	This is the collision bit for sprite 3.
Bit 4	This is the collision bit for the vertical grid lines.
Bit 5	This is the collision bit for the horizontal grid lines.
Bit 6	Unused.
Bit 7	This is the collision bit for the chars/quads.

16.2.4 0A3h: `vdc_color`

This register is used to set the background and grid color. The VDC generates a digital RGBI signal, one bit each for red, green, blue and intensity. The intensity bit for the background is always 0, so you can use normally only the 8 dark colors for the background. See the chapter about tips and tricks (page 37) for a trick to use the 8 bright colors for the background.

Bit 0	This bit is the blue component for the grid color.
Bit 1	This bit is the green component for the grid color.
Bit 2	This bit is the red component for the grid color.
Bit 3	This bit is the blue component for the background color.
Bit 4	This bit is the green component for the background color.
Bit 5	This bit is the red component for the background color.
Bit 6	This bit is the intensity component for the grid color.
Bit 7	Unused.

16.2.5 0A4h: `vdc_scanline` and 0A5h: `vdc_scanrow`

If bit 1 of `vdc_control` is 1 the position of the currently generated pixel can be read from here. If `vdc_control`:1 is changed from 1 to 0 a snapshot of the current pixel position gets stored in `vdc_scanline` and `vdc_scanrow`. The X position in `vdc_scanrow` increments on every 0 to 1 transition of the VDC pixel clock input. The `vdc_scanline` register is incremented when the `vdc_scanrow` register changes from 0CEh to 0CFh. The `vdc_scanrow` register wraps around from 0E3h to 0.

16.3 0C0h-0E9h: VDC grid

The VDC can generate a grid with 10 vertical and 9 horizontal grid lines. There are separate register sets for the vertical and horizontal grid. Every column of grid parts has its own register. For every bit that is set the matching grid part is drawn. Bit 0 is the top most grid part. The registers 0C0h-0C8h (`vdc_gridh0`-`vdc_gridh8`) form the top 8 horizontal grid lines. The registers 0D0h-0D8h (`vdc_gridi0`-`vdc_gridi8`) control the last horitontal grid line, only bit 0 of this registers is used. The vertical grid is generated by the registers 0E0h-0E9h (`vdc_gridv0`-`vdc_gridv9`). The vertical grid lines can be displayed as blocks by setting a bit in `vdc_control`. It is also possible to display just dots on all positions where the horizonal and vertical grids cross. Before the grid can be changed it has to be turned off. Otherwise the changes are not correctly done.

16.4 VDC sprites

The VDC can generate 4 sprites of 8x8 pixels. The color for each sprite can be any of the 8 brighter colors, the intensity bit is always set. The sprite registers can only be changed when the foreground graphics is turned off.

16.4.1 080h-09fh: `vdc_sprX_shape`

These are the bitmap registers for the sprites, 8 for each sprite. When the sprites are displayed bit 0 is the left-most pixel. So the look of the sprite is mirrored when written as binary number.

16.4.2 000h-00Fh: `vdc_sprX_ctrl`

These are the control registers for each sprite. Only 3 of them are used for each sprite, the fourth one is unused. There is no way to turn off the sprites individually, but when the position registers are set to 0F8h they are not displayed. This is the position used by the BIOS and I strongly recommend you use them too. Do not move the sprites just outside the visible area of your TV, they may still be visible on other TV sets and especially TV cards for PCs.

Sprite control 0	This register holds the Y position of the sprite.
Sprite control 1	This registers holds the 8 highest bits of the X position.
Sprite control 2	This register is used bitwise.
Bit 0	This is the lowest bit of the X position.
Bit 1	Setting this bit shifts the even rows of the sprite one pixel to the right.
Bit 2	If this bit is 1 the size of the sprite doubles.
Bit 3	This bit is the red component for the sprite color.
Bit 4	This bit is the green component for the sprite color.
Bit 5	This bit is the blue component for the sprite color.
Bit 6	Unused.
Bit 7	Unused.

16.5 VDC chars and quads

The VDC can generate up to 12 characters coming from a fixed character set of 64 different shapes. The bitmaps are 8x7 pixels big, the possible colors are the 8 bright colors, the intensity bit is always set. Officially the characters can't overlap each other. Ignoring this leads to a distorted display. A small overlap of less than 2 pixels in X direction seems ok. This means the X positions have to be at least 6 pixels apart. In the overlapping area the rightmost char gets displayed. The overlapping does not depend on the shape of the char, even spaces can't overlap. The charset pointer does not have to start at a character boundary, so it is possible to cut off the top part of a char, see tips and tricks (page 37) for details.

Quads are very similar to chars. Every quad has 4 sub-quads. The relative positions of the sub-quads are fixed, they are 16 pixels apart, so there is a free space of 8 pixels between 2 sub-quads. The charset pointer and color registers are the same for chars and sub-quads, but there is only one set of position registers for the whole quad. All accesses on relative addresses 0, 4, 8 and 0Ch control the Y position, the addresses 1, 5, 9 and 0Dh control the X position. The overlapping rules apply to the quads, too. As the whole quad drawing is controlled by the last sub-quad it is possible to cut off the bottom of the first 3 sub-quads. See tips and tricks (page 37) for details.

The char and quad registers can only be accessed when the foreground graphics is turned off. Any changes done when they are still active are ignored. Reading them does **not** return any sane values if the foreground graphics are enabled.

16.5.1 010h-07Fh: `vdc_char`*X*, `vdc_quad`*X*

Every char (and sub-quad) has a set of 4 control registers. Calculation of the charset pointer is a bit strange, to display the shape `c` in line `Y` you need to put `c*8-Y/2` into the charset pointer. To turn off one char individually move it to position 0F8h/0F8h, just like a sprite. The X position resolution of the sprites is twice as fine as the X position resolution of the chars.

Char control 0	This register holds the Y position of the char.
Char control 1	This registers holds the X position of the char.
Char control 2	This register holds the lowest 8 bits of the charset pointer.
Char control 3	This register is used bitwise.

Bit 0	This is bit 8 of the charset pointer, the highest bit.
Bit 1	This bit is the red component for the char color.
Bit 2	This bit is the green component for the char color.
Bit 3	This bit is the blue component for the char color.
Bit 4	Unused.
Bit 5	Unused.
Bit 6	Unused.
Bit 7	Unused.

16.6 Sound

The sound is generated by a 24 bit shift register. One bit at a time is shifted to the audio output. It is also possible to generate noise. The VDC supports 2 different sample rates. The higher frequency shifts out a bit every 4 lines, the lower every 16 lines. So the following sample shift rates are possible:

Machine	**0AAh:5**	**Sample shift rate**
Videopac	0	972 Hz
Videopac	1	3889 Hz
Odyssey²	0	981 Hz
Odyssey²	1	3925 Hz

16.6.1 0A7h-0A9h: `vdc_sound`*X*

This set of 3 registers form the 24 bit shift register.

16.6.2 0AAh: `vdc_soundctrl`

This is the sound control register, it is organized bitwise:

Bits 0-3	These 4 bits control the volume of the sound output.
Bit 4	This bit enables the noise generator.
Bit 5	This bit controls the shift frequency.
Bit 6	If this bit is 1 the shift register is rotated, the last bit from the output is shift back into the other end.
Bit 7	If this bit is 0 the sound is turned off.

17 Summary of the BIOS routines

17.1 bank0: 0387h

Switches banks to bank 0, continues at `0408h` in new bank.

Input	None
Output	None
Alters	P1
First used	Chapter about bank-switching (page 21)

17.2 bank01: 0383h

Switches banks from bank 2 to bank 0, bank 3 to bank 1, continues at `0408h` in new bank.

Input	None
Output	None
Alters	P1
First used	Chapter about bank-switching (page 21)

17.3 bank02: 037Fh

Switches banks from bank 1 to bank 0, bank 3 to bank 2, continues at `0408h` in new bank.

Input	None
Output	None
Alters	P1
First used	Chapter about bank-switching (page 21)

17.4 bank3: 038Bh

Switches banks to bank 3, continues at `0408h` in new bank.

Input	None
Output	None
Alters	P1
First used	Chapter about bank-switching (page 21)

17.5 bitclear: 0280h (G7000 only)

Clears a bit in a VDC or external RAM bit field.

This routine is not available on the Videopac+ G7400.

Input	A=bit number
	R1=pointer to end of bit field
Output	A=old bit
	R1=pointer to byte containing bit
	R2=bit mask
Alters	None
First used	Chapter about bit fields (page 15)

17.6 bitset: 028Ah (G7000 only)

Sets a bit in a VDC or external RAM bit field.

This routine is not available on the Videopac+ G7400.

Input	A=bit number
	R1=pointer to end of bit field
Output	A=old bit
	R1=pointer to byte containing bit
	R2=bit mask
Alters	None
First used	Chapter about bit fields (page 15)

17.7 bittest: 026Ah (G7000 only)

Tests, if a bit is set in VDC or external RAM bit field.

This routine is not available on the Videopac+ G7400.

Input	A=bit number
	R1=pointer to end of bit field
Output	A=bit
	R1=pointer to byte containing bit
	R2=bit mask
Alters	None
First used	Chapter about bit fields (page 15)

17.8 calcchar23: 014Bh

Calculates which bytes to put into character control register offset 2 and 3.

Input	R4=y
	R5=char
	R6=color
Output	R5/R6=byte 2/3
Alters	A
First used	Chapter about other useful routines (page 24)

17.9 clearchar: 016Bh

Clears all characters.

Input	None
Output	None
Alters	A, R0, R2
First used	Chapter about other useful routines (page 24)

17.10 decodejoystick: 03B1h

Converts the delta x/y from `getjoystick` into a direction.

Input	R2=delta x
	R3=delta y
Output	R1=direction: 0=right, 1=right+up, etc.
Alters	A, R0
First used	Chapter about other joystick (page 6)

17.11 divide: 03CFh

Calculates R4=R2/R3, R5=R2%R3.

Input	R2=dividend
	R3=divisor
Output	R4=R2/R3
	A=R5=R2%R3 (modulo operator, so R3*R4+R5=R2)
Alters	R6
First used	Chapter about other useful routines (page 24)

17.12 doclock: 01B0h

Updates the clock display if necessary.

Input	R4=y
	R6=color (both must be the same as in `initclock`)
Output	None
Alters	A, R0, R1, R2, R5, R6, P1, table
First used	Chapter about clock routines (page 12)

17.13 extramenable: 00ECh

Enables the external RAM, disables the VDC.

Input	None
Output	None
Alters	P1
First used	Chapter about table routines (page 5)

17.14 getjoystick: 038Fh

Gets the joystick position.

Input	R1=which joystick to test
Output	R2=delta x
	R3=delta y
	F0=button
Alters	A, R0, R1, P2
First used	Chapter about other joystick (page 6)

17.15 gfxoff: 011Ch

Turns the graphics off, so you can change the VDC registers.

Input	None
Output	None
Alters	A, interrupt enable
First used	Chapter about "HELLO WORLD" (page 3)

17.16 gfxon: 0127h

Turns the graphics on, you should not change VDC registers any more.

Input	None
Output	None
Alters	A, interrupt enable
First used	Chapter about "HELLO WORLD" (page 3)

17.17 init: 00F1h

Initialises VDC, internal and external RAM.

Input	None
Output	None
Alters	Everything
First used	Chapter about other useful routines (page 24)

17.18 initclock: 023Ah

Initialises quad0/1 to display a clock.

Input	R3=x
	R4=y
	R6=color
Output	None
Alters	A, R0, R1, R5, R6
First used	Chapter about clock routines (page 12)

17.19 irq: 0009h

Begin of IRQ routine.

Input	None	
Output	if VSYNC:	RB0 active, VDC enabled
		R5=old A
		R6=old P1
		`jmp 406h`
	if not VSYNC:	`retr`
Alters	A, R0, P1 when VSYNC	
First used	Chapter about "HELLO WORLD" (page 3)	

17.20 irqend: 0014h

Restores the old state before the interrupt and does a `retr`.

Input	RB0 active
	R5=old A
	R6=old P1
Output	`retr`
Alters	A, P1
First used	Chapter about interrupts (page 18)

17.21 multiply: 03DDh

Multiplies R2 and R3.

Input	R2=factor1
	R3=factor2
Output	A=R2*R3
Alters	R2
First used	Chapter about other useful routines (page 24)

17.22 nibblemixer: 02A4h (G7000 only)

Mixes a nibble using a ROM table at `02B3h` with selectable maximum.

This routine is not available on the Videopac+ G7400.

Input	R5=nibble to mix
	R7=complement of maximum
Output	A=R3=new nibble<=`CPL R7`
Alters	R5
First used	Chapter about other useful routines (page 24)

17.23 parsesnd: 04bh

Parse one sound opcode, part of sound irq handler.

Input	R1=command byte
	R2=parameter byte
Output	`retr` or `jmp 040Ah`
First used	Chapter about custom sound player (page 26)

17.24 playsound: 01A2h

Tells the IRQ routines to start playing one of the built-in tunes.

Input	A=pointer to tune in page 3
Output	None
Alters	A, interrupt enable
First used	Chapter about sound (page 11)

17.24.1 The built-in tunes

028h	`tune_beep_error`	An error occurred
02Eh	`tune_explode`	Something explodes
03Ch	`tune_alarm`	Something went wrong
04Ah	`tune_select`	Tune from "SELECT GAME"
056h	`tune_keyclick`	The keyboard click
05Ah	`tune_buzz`	A short buzzing sound
05Eh	`tune_select2`	`tune_select` backwards
06Ah	`tune_shoot`	A shot followed by an explosion

17.25 pluscmd: 0288h (G7400 only)

Send a command to the Videopac+ G7400.

Input	R6=Parameter (A)
	R7=Command (B)
Output	None
Alters	A, R0, R1
First used	Chapter about Videopac+ G7400 (page 32)

17.25.1 The command byte

000h	`plus_cmd_brow`	Set the y position, set x position to 0
020h	`plus_cmd_loady`	Set the y position
040h	`plus_cmd_loadx`	Set the x position
060h	`plus_cmd_incc`	Advance cursor position
080h	`plus_cmd_loadm`	Set read/write mode
0A0h	`plus_cmd_loadr`	Set the Videopac+ G7400 display mode
0C0h	`plus_cmd_loady0`	Choose which line is displayed as second line, useful for scrolling

17.25.2 The register M

000h	`plus_loadm_wr`	Normal write, advance cursor
020h	`plus_loadm_rd`	Normal read, advance cursor
040h	`plus_loadm_wrni`	Write without moving cursor
060h	`plus_loadm_rdni`	Read without moving cursor
080h	`plus_loadm_wrsl`	Write slice, for redefining chars
0A0h	`plus_loadm_rdsl`	Read slice, for redefining chars

17.26 plusdata: 028Ch (G7400 only)

Send data to the Videopac+ G7400.

Input	R6=Parallel attributes
	R7=Character
Output	None
Alters	A, R0, R1
First used	Chapter about Videopac+ G7400 (page 32)

17.26.1 The parallel attributes

080h	`col_patr_blck`	Enable block gfx, set background color
040h	`col_patr_invrt`	Invert the current char
020h	`col_patr_dwdth`	Double the width of the current char
010h	`col_patr_dhght`	Double the height of the current char
008h	`col_patr_stable`	Do not blink
000h-007h	`plus_col_color`	Foreground color

17.26.2 The serial attributes

080h	`col_satr_enable`	Enable serial attributes
004h	`col_satr_line`	Underline chars
002h	`col_satr_box`	Draw in boxed mode
001h	`col_satr_conc`	Draw concealed chars

17.27 plusenable: 02A1h (G7400 only)

Enables the Videopac+ G7400 graphics, disables the external RAM and VDC.

Input	None
Output	None
Alters	P1
First used	Chapter about Videopac+ G7400 (page 32)

17.28 plushide: 0296h (G7400 only)

Hides all Videopac+ G7400 graphics.

Input	None
Output	None
Alters	A, R7
First used	Chapter about Videopac+ G7400 (page 31)

17.29 plusloadr: 0283h (G7400 only)

Send the command plus_cmd_loadr to the Videopac+, handles 50/60Hz.

Input	A=Parameter
Output	None
Alters	A, R0, R1, R6, R7
First used	Chapter about Videopac+ G7400 (page 32)

17.29.1 The register R

080h	`plus_loadr_blnk`	Enable blink mode
040h	`plus_loadr_tt`	50 or 60 Hz, handled by BIOS
020h	`plus_loadr_tl`	Monitor mode, always 0 for VP+ G7400
010h	`plus_loadr_crsr`	Show cursor
008h	`plus_loadr_srow`	Display first line (called service row)
004h	`plus_loadr_conc`	Hide concealed chars
002h	`plus_loadr_box`	Enable box mode
001h	`plus_loadr_dspl`	Enable display of other lines

17.30 plusmode: 0299h (G7400 only)

Sets the Videopac+ G7400 display parameters.

Input	A=Luminance for Videopac+ colors
	R7=Transparency for VDC colors
Output	None
Alters	A
First used	Chapter about Videopac+ G7400 (page 31)

17.31 plusready: 027Dh (G7400 only)

Polls the busy flag of the EF9340/41, waits until it is ready.

Input	None
Output	None
Alters	A, R0
First used	Chapter about Videopac+ G7400 (page 34)

17.32 plusselectgame: 02C2h

Fills screen with "SELECT GAME", initialises VDC, internal and external RAM. Then waits until a key is pressed. This routine gives identical results to `selectgame` on non Videopac+ G7400 machines and can be used to detect the machine type.

Input	None
Output	VDC enabled, gfx on, A=key number
	G7000: `jmp 040Ch` in bank 3
	G7400: `jmp 040Ch` in bank 1
Alters	Everything
First used	Chapter about Videopac+ G7400 (page 31)

17.33 plusstart: 02ABh (G7400 only)

Initialises VDC and continues as if `selectgame` was called.

Input	None
Output	VDC enabled, gfx on, A=key number
	`jmp 040Ch` in bank 3
Alters	Everything
First used	Chapter about Videopac+ G7400 (page 31)

17.34 printchar: 03EAh

Prints a character or quad onto the screen.

Input	R0=pointer to char/quad
	R3=x
	R4=y
	R5=char
	R6=color
Output	R0=pointer to next char/quad
	R3=next x
	R5/R6=byte 2/3
Alters	A
First used	Chapter about "HELLO WORLD" (page 3)

17.35 putchar23: 0261h

Calculates which bytes to put into character data offset 2 and 3 and puts it into VDC.

Input	R0=pointer to char/quad byte 2
	R4=y
	R5=char
	R6=color
Output	R0=pointer to next char/quad
	R5/R6=byte 2/3
Alters	A
First used	Chapter about other useful routines (page 24)

17.36 random: 0293h (G7000 only)

Returns 2 random nibbles with selectable maximums using the T register.

This routine is not available on the Videopac+ G7400.

Input	R2=compl. max. 1st nibble
	R7=compl. max. 2nd nibble
Output	A=R3=1st nibble<=`CPL R2`
	R2=2nd nibble<=`CPL R7`
Alters	R4, R5, R7
First used	Chapter about other useful routines (page 24)

17.37 selectgame: 02C3h

Initialises VDC, internal and external RAM. Displays "SELECT GAME". Then waits until a key is pressed.

Input	None
Output	VDC enabled, gfx on
	A=key number
Alters	Everything
First used	Chapter about "HELLO WORLD" (page 3)

17.38 soundirq: 0044h

Processes sound-events in IRQ.

Input	Output from `vsyncirq`
Output	`retr` or `jmp 040Ah`
Alters	Restores everything stored in `irq`
First used	Chapter about "HELLO WORLD" (page 3)

17.39 tablebcdbyte: 017Ch

Displays a BCD byte via table, starts the table and activates it.

Input	A=byte
	R1=pointer to char/quad
	R3=x
	R4=y
	R6=color
Output	R0=pointer to end of table
Alters	A, R1, R2, R5, R6, P1
First used	Chapter about other useful routines (page 24)

17.40 tablebcdnibble: 0229h

Displays a BCD nibble via table.

Input	A=nibble
	R0=pointer to table
Output	R0=pointer to end of table
Alters	A, R5, R6
First used	Chapter about other useful routines (page 24)

17.41 tablechar23: 022Ch

Calculates which bytes to put into character data offset 2 and 3 and stores them into the table.

Input	R0=pointer to table
	R4=y
	R5=char
	R6=color
Output	R0=pointer to end of table
	R5/R6=byte 2/3
Alters	A
First used	Chapter about other useful routines (page 24)

17.42 tableend: 0132h

Puts an end marker into the table and activates it.

Input	R0=pointer to table
Output	None
Alters	A, R0, P1
First used	Chapter about table routines (page 5)

17.43 tableprintchar: 0197h

Prints a character or quad onto the screen using the table, it's up to you to set the first bytes of the table.

Input	R0=pointer to table
	R3=x
	R4=y
	R5=char
	R6=color
Output	R0=pointer to end of table
	R3=next x
	R5/R6=byte 2/3
Alters	A
First used	Chapter about table routines (page 5)

17.44 tableput2: 0235h

Puts a 2 into the table.

Input	R0=pointer to table
Output	R0=pointer to end of table
Alters	A
First used	Chapter about other useful routines (page 24)

17.45 vdcenable: 00E7h

Enables the VDC, disables the external RAM.

Input	None
Output	None
Alters	P1
First used	Chapter about table routines (page 5)

17.46 vsyncirq: 001Ah

VSYNC routine: checks for collisions, counts the clock, copies the table, processes first part of sound event.

Input	Output from `irq`
Output	`retr` or `jmp 40Ah` when sound is playing
Alters	A, R0, R1, F1
First used	Chapter about "HELLO WORLD" (page 3)

17.47 waitforkey: 013Dh

Waits until a key is pressed, plays the keyclick tune and returns which key was pressed.

Input	None
Output	A=key number
Alters	P1, P2, F1, interrupt enable
First used	Chapter about bit fields (page 15)

17.48 waitvsync: 0176h

Waits until the next frame starts.

Input	None
Output	None
Alters	F1, interrupt enable
First used	Chapter about other joystick (page 6)

17.49 Resources used by the BIOS

17.49.1 Register Bank 0

R7	The last key that was pressed
R6	Stores P1 while in interrupt
R5	Stores A while in interrupt
R4	Pointer to the next sound event in page 3
R3	Frames to wait for the next sound event
R2-R0	Scratch

17.49.2 Internal RAM

8048 Processor	
000h-007h	Register Bank 0
008h-017h	Stack
018h-01fh	Register Bank 1
BIOS	
03Fh:7	1=Table copy active
03Fh:6	1=Sound active
03Fh:5-0	unused
03Eh:7	1=Stop clock
03Eh:6	1=Count clock up
03Eh:5-0	Frame counter (0-03Ch)
03Dh	Collision register

17.49.3 External RAM

07Fh	How many registers to copy
07Eh	Start register
07Dh-X	Contents of registers
X-1	How many registers to copy
...	...
02h	Clock: seconds
01h	Clock: minutes