Computer Labs 2014/2015 - 1st Semester
Lab 5: The PC's Video Card in Graphics Mode

---

1. Objective

The objective of this lab is twofold:.

1. that you learn how to use the PC's video card in graphics mode, using the BIOS/VBE interface for its configuration
2. that you learn to do some animations

This lab is planned for 2 lab classes. You are expected to attend the lectures and in addition to spend about 5 hours to prepare each of the lab classes.

This handout is structured as follows:

2. What to Do
which summarizes what you are supposed to implement and what you have to do in order to prepare for the lab class
3. The VBE Standard
which provides a very brief overview of the VBE standard. More detailed information can be found in the standard document itself.
4. Use of the VBE Interface in Minix 3
which describes some Minix 3.1.8 aspects that are relevant for the lab.
5. Testing Code
which specifies the test functions that you must implement and that we will use to grade your work
6. Compiling your program
7. Configuring and running your program
8. Submission

2. What to Do

Write in C language several functions to use the PC's video card in graphics mode. The goal is to develop a generic module that will be used to create a library, which you will be able to use in the course's project.

Specifically, you shall develop the following functions whose prototypes are declared in test5.h:

```
1. void test_init(unsigned short mode, unsigned short delay)
2. int test_square(unsigned short x, unsigned short y, unsigned short size, unsigned long color)
3. int test_line(unsigned short xi, unsigned short yi, unsigned short xf, unsigned short yf, unsigned long color)
4. int test_xpm(unsigned short xi, unsigned short yi, char *xpm[])
5. int test_move(unsigned short xi, unsigned short yi, char *xpm[], unsigned short hor, short delta, unsigned short time)
6. int test_controller()
```

in a file whose name is `test5.c`. (Actually, you should use the file test5.c, which includes stubs for these functions as well as the implementation of other functions, without which you will not be able to test your code.) Section 5 describes what these functions should do.

2.1 Lab Preparation

This lab is planned for two lab classes.

First class

The goals for the first class are to implement:

1. `void test_init(unsigned short mode, unsigned short delay)`
2. `int test_square(unsigned short x, unsigned short y, unsigned short size, unsigned long color)`
3. `int test_line(unsigned short xi, unsigned short yi, unsigned short xf, unsigned short yf, unsigned long color)`

Therefore, for the first class you should read the material presented in the lecture of November 11th about the video card's in graphics mode and this handout.

Furthermore, you should:

1. Create a folder named `lab5` at the top level of your SVN repository
2. Create a file named `lab5.c` with the `main()` function that allows to test your code in a flexible way without having to compile it. You should also add this file, and the test files provided to your SVN repository.

Second class

The goals for the second class are to implement:

1. `int test_xpm(unsigned short xi, unsigned short yi, char *xpm[])`
2. `int test_move(unsigned short xi, unsigned short yi, char *xpm[], unsigned short hor, short delta, unsigned short time)`
3. `int test_controller()`

Therefore, for the first class you should read the material presented in the lecture of November 18th about sprites and their animation.

3. The VBE Standard

In the late 1980's, there was a large number of video card manufacturers offering video cards with higher resolutions than those specified in the VGA standard. In order to allow application programmers to develop portable video-based applications, the VESA (Video Electronics Standards Association) published in 1989 the VBE (VESA BIOS Extensions) standard. During the 1990's this standandard was revised several times. However, the last major version, version 3.0, dates from 1998. This is probably because by that time other standards had emerged, namely DirectX by Microsoft and OpenGL, originally developed by SGI.

The VBE standard defines a set of functions related to the operation of the video card, and specifies how they can be invoked. These functions were implemented by the video card manufacturers, usually in a ROM in the video card. Originally, the interface to these functions used only `INT 0x10`, i.e. the interrupt number used by the BIOS' video services, and had to be executed with the CPU in real-mode. Starting with version 2.0, VBE specifies also direct call of some time-critical functions from 32-bit applications. Version 3.0 specified a new interface for protected mode for most functions, but it is optional.

Because the VM Player video card's emulation supports only VBE 2.0, we will focus on that version of the standard.

In this lab we will use only a few basic functions that allow

1. to retrieve information regarding the video card's capabilities, including the modes supported and their characteristics and
2. to change the operating mode.

Because these functions are not time-critical, they must be accessed via the "`INT 0x10` interface".

3.1 Accessing the VBE Functions

As already mentioned, VBE functions are called using the interface used for standard BIOS video services. That is, the call is via the `INT 0x10` software interrupt instruction, in real mode, and the function parameters are passed in the CPU registers.

When invoking a VBE function, the `AH` register must be set to `0x4F`, thus distinguishing it from the standard VGA BIOS functions. The VBE function being called is specified in the `AL` register. The use of the other registers depends on the VBE function.

The `AX` register is also used to return a completion status value. If the VBE function called is supported, the `AL` register is set with value `0x4F`, i.e. the value passed in the `AH` register. A different value is returned if the function is not supported. If the VBE function completed successfully, value `0x00` is returned in the `AH`, otherwise it is set to indicate the nature of the failure, as shown in the table:

AH	Meaning
0x01	Function call failed
0x02	Function is not supported in current HW configuration
0x03	Function is invalid in current video mode

3.2 Setting the Graphics Mode

The VBE standard defines several operating modes that video cards may support. These modes have different attributes, for example whether they are text or graphic modes. Other attributes in the latter case include, the horizontal and vertical resolution, the number of bits per color. For yet more attributes, you can read the specification of function **`0x01` Return VBE Mode Information** in page 16 and following of the VBE 2.0 standard. In this lab, we are interested only in the modes with 256 colors:

Screen Resolution	Mode
640x480	0x101
800x600	0x103
1024x768	0x105
1280x1024	0x107

For a list of the modes specified in VBE, you can read Section 3 (pg. 6) of the VBE 2.0 standard. Video card manufacturers can also define other video modes. This is made possible by the VBE functions that allow an application to obtain the video card's capabilities. For example, the VBE implementation of the VMware Player used in the labs supports 100 video modes, of which only 10 are defined in the standard.

To initialize the controller and set a video mode, you must use function **`0x02` - Set VBE Mode**. The mode must be passed in the `BX` register. **Bit 14** of the `BX` register should be set to use a linear frame buffer model, which facilitates access to VRAM. You can find more details regarding this function in its specification in pg. 25 of the VBE2.0 standard.

3.3 Linear/Flat Frame Buffer Model

In graphics mode, the screen is abstracted as a matrix of points, **pixels** (from picture element). The number of pixels on the screen depends on the screen resolution. To each pixel is associated one or more bytes in VRAM whose value determines the color of the corresponding pixel. Thus we can abstract the screen as a set of `VRES` lines, each of which with `HRES` pixels.

In the linear/flat frame buffer model, the lines of the screen are located in VRAM one after the other, from the top line to the bottom line of the screen. Furthermore, in each line, the left most pixel comes first, then the pixel to its left, and so on until the right most pixel, which comes last. This is the model you have used in Lab1, the difference is that whereas in Lab 1 the lines were sequence of pairs (character,attribute), one for each symbol in that line, here a line is a sequence of colors, one for each pixel in that line.

Like the text mode VRAM, the graphics mode VRAM is not directly accessible by a user program. To make it accessible you need to map it in the process' address space. Of course, as we have already discussed in Lab1, this operation is privileged and your program needs to have the necessary permissions.

Before your process can map the graphics VRAM in its address space it needs to know the VRAM's physical address. This information can be obtained from the video controller using VBE function **`0x01` Return VBE Mode Information**, which takes as arguments the mode and a real-mode address of a buffer that will be filled in with the parameters of the specified mode. In addition to the linear buffer physical address, these parameters include the horizontal and vertical resolution, as well as the number of bits per pixel. Generally, all these parameters must be known to change the color of a specific pixel. You can find more details regarding this function in its specification in pg. 16 of the VBE2.0 standard.

Another useful function provided by the VBE standard is function **`0x00` Return VBE Controller Information**, which returns the capabilities of the controller, including a list of the video mode numbers supported by the controller. Like function `0x01`, this one also takes as an argument a real-mode address with a buffer that will be filled in with the controller information. This function and function `0x01` can be used by a graphics application to learn the capabilities of the video card, and set the video-mode that suits it better. You can find more details regarding this function in its specification in pg. 12 of the VBE2.0 standard.

The use of the VBE functions in general and of the functions `0x00` and `0x01` is somewhat tricky because they usually use real-mode addresses: these are physical addresses, and are composed of the base address of a segment, a 16 bit-value that should be shifted by 4 to create a 20 bit address, and a 16-bit offset, that should be added to the 20-bit address. However, Minix uses virtual addresses. Fortunately, Minix provides functions that allow to map virtual addresses into physical addresses, thus making it possible to use the VBE interface.

3.4 Returning to Text Mode

As already mentioned, in graphics mode you will not have access to the Minix VTs, and hence to the shell. Thus, before terminating your program, you should always reset the video controller to operate in text mode.

The mode used by Minix in text mode is a standard CGA mode, whose number is `0x03`. To set this mode, you should use the standard `INT 0x10` BIOS interface, namely function `0x00`. Thus you should set the `AH` register to `0x00` and the `AL` register to `0x03`.

4. Use of the VBE Interface in Minix 3

Accessing the video card via the VBE interface in Minix 3 raises a few issues:

Invocation of the `INT 0x10` instruction in real-mode
: Minix 3 executes in **protected mode**, however VBE requires the invocation of `INT 0x10` in real-mode. There is therefore a need for a change in the operating mode of the processor.

Allocation and access to memory in the lower 1 MByte of the physical address space
: VBE functions `0x00` (Return VBE Controller Information) and `0x01` (Return VBE Mode Information) require that a memory buffer be passed as an argument. This buffer will be filled in with the required information by these functions. Because they are executed in real-mode, the buffer must be allocated in the lower 1 MByte of the physical address space.

Access and processing of the information returned by VBE functions `0x00` and `0x01`
: Actually, there are two issues related to this. First, this information is stored in a sequence of memory positions without any concern for the alignment of the data according to their type. Second, most pointers, with the notable exception of the physical address of video VRAM, are real-mode far-pointers.

Mapping of VRAM into the process address space

Graphics mode VRAM, like text mode VRAM, is not directly accessible to a user process in Minix 3. So that a process can access VRAM, it must first map it into its address space.

Granting the necessary permissions to the "driver"

Addressing most of the issues in this list requires the execution of privileged operations. Therefore, the program you'll develop needs to be granted the necessary permissions, and therefore synchronize with the RS server.

The last two issue were already discussed in the first lab, thus here we will address only the other three.

4.1 Invocation of the `INT 0x10` instruction in real-mode

Minix 3 offers the `SYS_INT86` kernel call, whose description in the Minix 3 Developers Guide is as follows:

> *Make a real-mode BIOS on behalf of a user-space device driver. This temporarily switches from 32-bit protected mode to 16-bit real-mode to access the BIOS calls.*

The library function to make this kernel call is as follows:

```
int sys_int86(struct reg86u *reg86p);
```

It returns either `OK`, in the case of success, or `EFAULT` otherwise, and takes as arguments a value of type `struct reg86u *reg86p`, which allows to specify the values of the interrupt number and of the processor registers. This struct is defined in the header file `<machine/int86.h>`.

Function `vg_exit()` already provided in `video_gr.c`:

```
 /* Set default text mode */
int vg_exit() {
  struct reg86u reg86;

  reg86.u.b.intno = 0x10; /* BIOS video services */
  reg86.u.b.ah = 0x00;    /* Set Video Mode function */
  reg86.u.b.al = 0x03;    /* 80x25 text mode*/

  if( sys_int86(& reg86) != OK ) {
    printf("\tvg_exit(): sys_int86() failed \n");
    return 1;
  }
  return 0;
}
```

resets the video controller to operate in text mode, by calling function `sys_int86()` to invoke function `0x00` (Set Video Mode) of the BIOS video services (`INT 0x10`).

4.2 Allocation and access to memory in the lower 1 MByte of the physical address space

As usual, access to the lower 1 MByte of the physical address space requires mapping that region into the process' address space. However, functions `0x00` and `0x01` of the VBE standard, require also the allocation of a buffer in that region of the physical address space.

Minix 3 provides the necessary mechanisms for that, but it is neither documented nor straightforward. Therefore, I have abstracted those mechanisms in a set of 3 functions, whose prototypes can be found in lmlib.h:

```
int lm_init();
void *lm_alloc(unsigned long size, mmap_t *map);
void lm_free(mmap_t *map);
```

which are implemented in the liblm.a library. For more details, you can read their documentation.

The `mmap_t` type, also defined in `lmlib.h`, contains the necessary information on the mapping of a physical memory region into the virtual address space of a process:

```
typedef struct {
  phys_bytes phys;          /* physical address */
  void *virtual;            /* virtual address */
  unsigned long size;       /* size of memory region */
} mmap_t;
```

The `phys` member may be useful when using the VBE interface, whereas the `virtual` field is useful in all other cases.

Because the amount of memory available in the lower 1 MBytes of the physical address space is very limited, it is important that you free a region of memory that you have allocated with `lm_alloc()` as soon as you don't need it, by calling `lm_free()`. Not doing it may lead to **memory leaks** and eventually to the depletion of the available memory in that region. On the other hand, using a memory region that may have already been freed, may lead to all sorts of problems, the least of which is the crash of the process. Thus you should be very careful with the use of these two functions.

Your code should not modify the value of the `mmap_t` struct initialized by `lm_alloc()`, as it may affect the correctness of the `lm_free()` call with that struct as argument.

4.3 Access and Processing to the Data Returned by VBE Functions `0x00` and `0x01`

VBE functions `0x00` and `0x01` return data in a memory buffer, as defined in their specification, in pages 12 to 24 of the VBE 2.0 standard. These data comprises several fields whose size is specified using one of 3 types: `db`, `dw` and `dd`, with sizes 1, 2 and 4 bytes, respectively. Because, memory space was at a premium, this data is stored sequentially in memory, without holes (except for fields defined in previous versions that were deprecated).

This layout creates a potential problem when one defines C language structs with the fields defined in the VBE standard and uses C types such as `uint8_t`, `uint16_t` and `uint32_t` defined in `<stdint.h>` corresponding to the "types" used in VBE's specification. The problem is that, for performance reasons, most compilers store the members of a C struct in positions whose addresses are aligned according to their types; this may lead to holes in the structure. In this case, access to a member of the struct in C, may actually access a memory position storing a different field, or a different piece of the same field.

The GNU C compiler installed in the Minix 3 image in the lab computers (and VMware Player) provides the `__attribute__ ((packed))` extension: when this annotation is used in the source code, the C compiler will not place the members of the C struct in memory positions whose addresses are aligned according to their types, but rather place them sequentially without any holes between them.

Because the definition of the `vbe_mode_info_t` type in `vbe.h` is already declared as "packed", you can access directly its members, without concern of their layout in memory.

4.4 Accessing Graphics Mode Video RAM in C

As we have already mentioned, with the linear frame buffer model, each pixel on the screen is mapped sequentially to video RAM from the left to the right, and from the top to the bottom. Thus, to access graphics mode VRAM in C, after mapping it in the process' address space, you can use C pointers, just like you have done in Lab1 to access text mode VRAM.

Note To keep the prototypes of the functions `vg_fill()`, `vg_set_pixel()`, etc. simple, they do not take the address on which VRAM is mapped nor the horizontal resolution and so on as arguments. Instead, we have declared 4 static variables in video_gr.c, which are inititialized by `vg_init()`:

```
static char *video_mem;         /* Process address to which VRAM is mapped */

static unsigned h_res;          /* Horizontal resolution in pixels */
static unsigned v_res;          /* Vertical resolution in pixels */
static unsigned bits_per_pixel; /* Number of VRAM bits per pixel */
```

Although the use of global variables is something you should avoid, there are two reasons why they are acceptable here:

1. These are **static** global variables, and thus their scope is limited to the file where they are declared, i.e. they are not visible in other files
2. We are structuring our code very much like in object oriented programming, and these variables are akin to the private member variables of a C++ class.

4.5 Summary

To summarize, the sequence of operations of the program to develop in this lab is as follows:

1. Initialize the video graphics mode module, by calling `vg_init()`, which should call `vbe_get_mode_info()` to obtain the mode parameters, and map the graphics mode VRAM in the process' address space
2. Modify video RAM, by calling the functions you'll develop for this lab.
3. Switch back to text mode, by calling `vg_exit()`

IMP. Before executing this sequence you must call the function `sef_startup()` to synchronize with the RS.

5. Testing Code

So that we can grade your work, you are required to implement the test functions described in this section. We will develop the code that will call them, so make sure that your implementation matches their prototypes.

Rather than implement the required functionality directly in these functions, you should design and implement functions that may be useful to interface with the video adapter in your integration project, i.e. functions that configure the video adapter, that change the color of a pixel, etc. We will grade not only how you structure the required functionality in functions, but also how do you group these functions in compilation modules, i.e. in the source files.

5.1 `test_init(unsigned short mode, unsigned short delay)`

The purpose of this function is to test that your code is able to switch the video adapter to the graphics mode specified in its argument, using the VBE interface, and then back again to the default text mode.

When this function is invoked, your program should change to the video mode specified in its `mode` argument.

After a delay of `delay` seconds, it shoul go back to Minix's default text mode. Then, just before exiting, it should print the **physical address** of the Video RAM (VRAM) in the specified mode to the console, i.e.

using `printf()`. To determine this address you **must** use VBE function 0x01, Return VBE Mode Information.

Note: Using VBE function 0x01 requires you to read data from the lower 1 MB as described in Section 4. This requires you to understand some idiosyncrasies of the IA-32 architecture that stem from its compatibility with earlier Intel processors. This may require some effort. Actually, the issues here are similar to those of `test_controller` below. Therefore, my suggestion is that you implement the printing of the VRAM's physical address only after you have implemented the remaining functionality (except `test_controller`).

Note: You may wish to develop the following function whose prototype is declared in vbe.h:

```
int vbe_get_mode_info(unsigned short mode, vbe_mode_info_t *vmi_p)
```

in a file whose name is `vbe.c`. (You should use the file vbe.c, which includes a stub for this function as well several preprocessor directives that may facilitate your implementatiaon.) For more information about these functions please read their documentation.

5.2 `test_square(unsigned short x, unsigned short y, unsigned short size, unsigned long color)`

The purpose of this function is to test whether your code is able to change the color of pixels on the screen.

When this function is invoked, your program should change to video mode `0x105` and draw a square whose side in pixels is `size` and with the `color` specified in its last argument. The arguments `x` and `y` specify the coordinates of the square's upper left corner.

When the user releases the ESC key (scancode `0x81`), your program should go back to Minix's default text mode and exit, even though the movement has not completed.

Note Unless you have implemented `vbe_get_mode_info()` you do not know what is the physical address of the VRAM in mode `0x105`. I suggest that you use a program that solves an earlier version of this lab, including executing function VBE 0x01 to find out that address. Then, you can use hard-coded values for the parameters (screen resolution, number of colors per pixel and physical address of graphics mode VRAM) of the graphics mode used for testing (`0x105`) that are defined as symbolic constants in video_gr.c. In the final version, i.e. the one you should submit at the end of the class, your code should call the function `vbe_get_mode_info()`, which you should implement, and that uses the VBE function **`0x01` - Return VBE Mode Information** to obtain the values of these parameters.

5.3 `int test_line(unsigned short xi, unsigned short yi, unsigned short xf, unsigned short yf, unsigned long color)`

The purpose of this function is to test whether your program is able to draw a line segment on the screen.

When this function is invoked, your program should change to video mode `0x105` and draw a line segment with the `color` specified in its last argument, starting in the pixel with coordinates `(xi,yi)` and ending in the pixel with coordinates `(xf,yf)`.

When the user releases the ESC key (scancode `0x81`), your program should go back to Minix's default text mode and exit, even though the movement has not completed.

5.4 `test_xpm(unsigned short xi, unsigned short yi, char *xpm[])`

The purpose of this function is to test whether your program is able to draw a sprite that is provided as an XPM image.

When this function is invoked, your program should change to video mode `0x105` and display the sprite provided as an XPM at the specified coordinates (upper left corner of XPM).

You can use the function `read_xpm()` provided in `read_xpm.c` to convert an XPM into a pixmap. You can find some XPMs for testing in `pixmap.c`. For further information about XPMs you can read the notes about sprites by Prof. João Cardoso (Velho).

When the user releases the ESC key (scancode `0x81`), your program should go back to Minix's default text mode and exit.

5.5 `int test_move(unsigned short xi, unsigned short yi, char *xpm[], unsigned short hor, short delta, unsigned short time)`

The purpose of this function is to test whether your program is able to move a sprite that is provided as an XPM image.

When this function is invoked, your program should change to video mode `0x105` and display the sprite provided as an XPM at the specified coordinates (upper left corner of XPM), and then move that sprite along the specified direction, for the number of pixels specified in `delta` (a positive value is in the direction of increasing the respective coordinate). This movement should take the number of seconds specified in `time`.

IMP: We will give you credit if you use Timer 0 interrupts to measure the time between "frames".

When the user releases the ESC key (scancode `0x81`), your program should go back to Minix's default text mode and exit, even though the movement has not completed.

5.6 `int test_controller()`

The purpose of this function is to test whether you are able to use VBE function 0x0, Return VBE Controller Information.

When this function is invoked, your program needs not change to a different mode, rather it should display on the console, i.e. using `printf()`:

1. The controller capabilities
2. List of mode numbers (in hexadecimal) supported
3. Size of VRAM memory

and then exit.

6. Compiling your Program

Follow the procedures described for the previous labs (of course now you should use directory `lab5` instead of directory `lab4`, `lab3` or `lab2`), but do not forget to modify the `Makefile`. In particular, do not forget to add the library `liblm.a`. If you do not remember how to do it, check the `Makefile` provided for Lab1

7. Configuring and Running your Program

Your program invokes functions that are privileged. Thus, before you can run it, you need to add to `/etc/system.conf.d/` a file named `lab5`, with the following contents:

```
service lab5
{
        system
                UMAP
                INT86
                PRIVCTL
                READBIOS
                DEVIO
```

```
                IRQCTL
                ;
        ipc
                SYSTEM
                rs
                vm
                pm
                vfs
                ;
        io
                40:4
                60
                64
                ;
        irq
                0         # TIMER 0 IRQ
                1         # KBD IRQ
                12        # AUX/MOUSE IRQ
                ;
        uid     0
                ;
};
```

I suggest that you add the new entries (actually only the first 4 entries under `system`) to the configuration file you have used in previous labs.

IMP. Do not forget to change the name of the service in the first line to `lab5`.

Once you have created the file with the permissions, you can execute your program using the `service` command. Given that your program should be privileged, you must run it as **user `root`**. Thus, assuming that you are logged in as user `lcom` and your shell's current directory is `/home/lcom/lab5`, you can issue the following commands:

```
$ su
# service run `pwd`/lab5 [-args "< list of space separated arguments >"]
```

Here we assume that your `main()` is similar to the one we provided in lab1.c. (You'll score points for this.)

Note The `` `pwd` `` expression evaluates to the output of the command `pwd`, which returns the present working directory (`/home/lcom/lab5`, according to our assumptions). This is simpler than having to type the full path:

```
# service run /home/lcom/lab5/lab5 [-args "< list of space separated arguments >"]
```

8. Submission

Like in previous labs, you should submit your code to the SVN repository as you develop and test it. This code should be under directory **`lab5`**, which should be at the top level of the SVN repository of your Redmine project.

The deadline for the submission is at the end of your second class of this lab. Because, some sections cannot stay in the lab after the end of their classes, you have only a 10 minutes tolerance. That is we will grade the latest submission as of 10 minutes after the end of your 2nd class. Please make sure that you are sufficiently familiar with SVN to avoid last minute problems. Unless these problems are not your responsibility, we will penalize you heavily for having to handle submissions in a different way.

IMP. You should add to the SVN repository all files needed, including the `Makefile`, so that compilation of your code can be done by checking out the `lab5` directory, adding the `liblm.a` library to that directory, and executing:

```
make
```

under that directory in the working copy. You'll score points for this.

Acknowledgments

This lab is based on a lab by João Cardoso and Miguel P. Monteiro for DJGPP running on Windows98.