2. What to Do

Write in the C programming 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.

Like in Labs 3 and 4 you are not given the prototypes of the functions to implement: the specification of these functions is part of your job. However, to make the task of grading your assignment feasible you are required to implement the following testing functions:

1. `void video_test_init(unsigned short mode, unsigned short delay)`
2. `int video_test_rectangle(uint16_t mode, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint32_t color)`
3. `int video_test_pattern(uint16_t mode, uint8_t no_rectangles, uint32_t first)`
4. `int video_test_xpm(char *xpm[], unsigned short xi, unsigned short yi)`
5. `int video_test_move(char *xpm[], unsigned short xi, unsigned short yi, unsigned short xf, unsigned short yf, short speed, short frame_rate)`
6. `int video_test_controller()`

in a file whose name is `lab5.c`. (Actually, you should use the file lab5.c, which includes stubs for these functions as well as the implementation of `main()`, 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 video_test_init(unsigned short mode, unsigned short delay)`
2. `int video_test_rectangle(uint16_t mode, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint32_t color)`
3. `int video_test_pattern(uint16_t mode, uint8_t no_rectangles, uint32_t first)`

IMP- For the first lab class, you will be using the function `vbe_get_mode_info()` provided by the LCF to find the parameters of the VBE mode, e.g. the video RAM's (frame buffer) physical address.

Therefore, for the first class you should read the material presented in the lectures of November 5th and 9th about the video card's in graphics mode and this handout, **except** Sections 4.3 and 4.4.

Furthermore, you should:

1. Copy file `lab5.c` to the folder `/home/lcom/labs/lab5`, that already exists (in Minix's filesystem). You should also copy to this folder a `makefile` you have used in one of the previous labs. (Do not forget to make appropriate changes to this `makefile` so that it can be used for this lab.)
2. Import the directory tree rooted at `~/labs/lab5` to the SVN repository of your project on Redmine, as described in Lab 0's Section 6.1.
3. Transform `~/labs/lab5/` in a working copy of the directory tree rooted at `lab5`, as described in Lab 0's Section 6.2.
4. Write `video_test_init()`

Second class

The goals for the second class are to implement:

1. `int video_test_xpm(char *xpm[], unsigned short xi, unsigned short yi, )`
2. `int video_test_move(char *xpm[], unsigned short xi, unsigned short yi, unsigned short xf, unsigned short yf, short speed, short frame_rate)`
3. `int video_test_controller()`

You should also develop your own implementation of `vbe_get_mode_info()` and you should use it instead of the implementation provided by the LCF. Do not forget to put the name of the function between parenthesis in its definition.

Therefore, for the second class you should read Sections 4.3 and 4.4, and the material presented in the lectures of November 12th, 16th and 19th, including the material about sprites and their animation and implement `video_test_xpm()`.

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 standard 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 protected mode. Version 3.0 specified a new interface for protected mode for most functions, but their implementation is optional: i.e. an implementation can claim conformance with VBE 3.0, even though it does not support the protected mode interface.

Because the VirtualBox 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`. 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 want you to play with modes with different parameters, e.g.:

Mode	Screen Resolution	Color Model	Bits per pixel (R:G:B)
0x105	1024x768	Indexed	8
0x110	640x480	Direct color	15((1:)5:5:5)
0x115	800x600	Direct color	24 (8:8:8)
0x11A	1280x1024	Direct color	16 (5:6:5)
0x14C	1152x864	Direct color	32 ((8:)8:8:8)

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 VirtualBox supports many modes that are not defined in the standard, e.g. mode `0x14C` referred to in the table above.

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, in order to set the 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 right, and so on until the right most pixel, which comes last.

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, 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 segment 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. (Anyway, the best is for you to use `ssh` from "remote" terminal.)

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. Minix 3 Notes

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.

Mapping of video RAM (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.

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.

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

Minix 3.1.x 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.

IMP- Although Minix has dropped this kernel call in release 3.2.0, Pedro Silva added support for this kernel call to the Minix 3.4.0rc6 version we are using in LCOM, by porting `libx86emu` to Minix.

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

```
#include <machine/int86.h>

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 by the LCF:

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

  memset(&reg86, 0, sizeof(reg86));     /* zero the
structure */

  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`).

IMP.- In order to avoid unexpected behavior by the `libx86emu`, whenever you make a `INT 0x10` call, you should clear the ununsed registers of the `struct reg86u` before passing it to `sys_int86()`. See the example above for `vg_exit()`.

4.2 Mapping VRAM in a Process's Address Space

In Minix 3, so that a user-process can access VRAM it has to map it in its own address space using MINIX 3 kernel call:

```
void *vm_map_phys(endpoint_t who, void *phaddr, size_t
len);
```

The first argument is a value that identifies the process on whose address space the physical memory region should be mapped. Of course `phaddr` specifies the physical address of the first memory byte in that region and `len` the region's length. This call returns the virtual address (of the first physical memory position) on which the physical address range was mapped. Subsequently, a process can use the returned virtual address to access the contents in the mapped physical address range.

`vm_map_phys()` is a **MINIX 3 kernel call** and a process must have not only the necessary permissions to execute that call, but it must also have the permission to map the desired physical address range. To grant a process the permission to map a given address range you can use MINIX 3 kernel call:

```
int sys_privctl(endpoint_t proc_ep, int request, void *p)
```

Again, the first argument specifies the process whose privileges will be affected, and the other two arguments depend on the privileges to change.

The following code snippet shows how to map the video RAM in a process' address space:

```
#include <minix/driver.h>
#include <sys/mman.h>

[...]

static void *video_mem;          /* frame-buffer VM address
                        (static global variable*/
[...]

minix_mem_range mr;
unsigned int vram_base;  /* VRAM's physical addresss */
```

```
unsigned int vram_size;  /* VRAM's size, but you can use
                         the frame-buffer size, instead */
int r;

[...]
/* Allow memory mapping */

mr.mr_base = (phys_bytes) vram_base;
mr.mr_limit = mr.mr_base + vram_size;

if( OK != (r = sys_privctl(SELF, SYS_PRIV_ADD_MEM, &mr)))
   panic("sys_privctl (ADD_MEM) failed: %d\n", r);

/* Map memory */

video_mem = vm_map_phys(SELF, (void *)mr.mr_base,
vram_size);

if(video_mem == MAP_FAILED)
   panic("couldn't map video memory");
```

As in other labs, you need not include `<minix/drivers.h>` nor `<minix/drivers.h>`, as long as you include `<lcom/lcf.h>`, which includes several other header files.

4.2.1 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.

Note To keep the prototypes of the functions you will have to develop simple, they need not take the address on which VRAM is mapped nor the horizontal resolution and so on as arguments. Instead, you can use static variables in the source code file that changes the video RAM's content, as shown in the following code snippet:

```
static char *video_mem;        /* Process (virtual) 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 static member variables of a C++ class.

Of course, you can add other "static member variables" as you see fit.

4.3 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. Furthermore, 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 that are declared in `<liblm.h>`:

```
int lm_init(bool enable_logging);
void *lm_alloc(size_t 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 doxygen 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 *virt;           /* virtual address */
  size_t 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.

4.4 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.

To prevent the C compiler from placing the members of a C struct in memory positions whose addresses are aligned according to their types, but rather place them consecutively without any holes in between, you can enclose the relevant data structures inside the following pair of `#pragma` directives: 1) `#pragma pack(1)` and 2) `#pragma options align=reset`, as illustrated in the following code snippet with the definition of `vbe_mode_info_t` type in `vbe.h`:

```
#pragma pack(1)

typedef struct {
  /*  Mandatory information for all VBE revisions */
  uint16_t ModeAttributes;    /**< @brief mode attributes
*/
  uint8_t WinAAttributes;     /**< @brief window A
attributes */
  uint8_t WinBAttributes;     /**< @brief window B
attributes */
  uint16_t WinGranularity;    /**< @brief window
granularity */
  uint16_t WinSize;      /**< @brief window size */
  uint16_t WinASegment;       /**< @brief window A start
segment */
  uint16_t WinBSegment;       /**< @brief window B start
segment */
  phys_bytes WinFuncPtr; /**< @brief real mode/far pointer
to window function */
  uint16_t BytesPerScanLine;  /**< @brief bytes per scan
line */

  /* Mandatory information for VBE 1.2 and above */
```

```
  uint16_t XResolution;       /**< @brief horizontal
resolution in pixels/characters */
  uint16_t YResolution;       /**< @brief vertical
resolution in pixels/characters */
  uint8_t XCharSize;          /**< @brief character cell
width in pixels */
  uint8_t YCharSize;          /**< @brief character cell
height in pixels */
  uint8_t NumberOfPlanes;     /**< @brief number of memory
planes */
  uint8_t BitsPerPixel;  /**< @brief bits per pixel */
  uint8_t NumberOfBanks; /**< @brief number of banks */
  uint8_t MemoryModel;        /**< @brief memory model type
*/
  uint8_t BankSize;      /**< @brief bank size in KB */
  uint8_t NumberOfImagePages; /**< @brief number of images
*/
  uint8_t Reserved1;          /**< @brief reserved for page
function */

  /* Direct Color fields (required for direct/6 and YUV/7
memory models) */

  uint8_t RedMaskSize;        /* size of direct color red
mask in bits */
  uint8_t RedFieldPosition;   /* bit position of lsb of red
mask */
  uint8_t GreenMaskSize; /* size of direct color green mask
in bits */
  uint8_t GreenFieldPosition; /* bit position of lsb of
green mask */
  uint8_t BlueMaskSize;  /* size of direct color blue mask
in bits */
  uint8_t BlueFieldPosition;  /* bit position of lsb of
blue mask */
  uint8_t RsvdMaskSize;       /* size of direct color
reserved mask in bits */
  uint8_t RsvdFieldPosition;  /* bit position of lsb of
reserved mask */
  uint8_t DirectColorModeInfo;     /* direct color mode
attributes */

  /* Mandatory information for VBE 2.0 and above */
  phys_bytes PhysBasePtr;        /**< @brief physical
address for flat memory frame buffer */
  uint8_t Reserved2[4];   /**< @brief Reserved - always set
to 0 */
  uint8_t Reserved3[2];   /**< @brief Reserved - always set
to 0 */

  /* Mandatory information for VBE 3.0 and above */
```

```
  uint16_t LinBytesPerScanLine;  /* bytes per scan line for
linear modes */
  uint8_t BnkNumberOfImagePages; /* number of images for
banked modes */
  uint8_t LinNumberOfImagePages; /* number of images for
linear modes */
  uint8_t LinRedMaskSize;      /* size of direct color red
mask (linear modes) */
  uint8_t LinRedFieldPosition;   /* bit position of lsb of
red mask (linear modes) */
  uint8_t LinGreenMaskSize;    /* size of direct color
green mask (linear modes) */
  uint8_t LinGreenFieldPosition; /* bit position of lsb of
green mask (linear  modes) */
  uint8_t LinBlueMaskSize;     /* size of direct color blue
mask (linear modes) */
  uint8_t LinBlueFieldPosition;  /* bit position of lsb of
blue mask (linear modes ) */
  uint8_t LinRsvdMaskSize;     /* size of direct color
reserved mask (linear modes) */
  uint8_t LinRsvdFieldPosition;     /* bit position of lsb
of reserved mask (linear modes) */
  uint32_t MaxPixelClock;      /* maximum pixel clock (in
Hz) for graphics mode */
  uint8_t Reserved4[190];      /* remainder of
ModeInfoBlock */
} vbe_mode_info_t;

#pragma options align=reset
```

Because `vbe.h` is already included by `<lcom/lcf.h>`, you can access directly the members of `vbe_mode_info_t` without concern of their layout in memory. For more detailed information regarding these members, you can read the definition of the `ModeInfoBloc struct` in pg. 23 of the VBE2.0 standard.

An alternative, is to use the `__attribute__ ((packed))` extension, of the GNU C compiler. Although some other compilers, e.g. `clang`, also support this extension, it is less portable than the above alternative.

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. The LCF function `vbe_get_mode_info()` can be used to obtain the relevant mode parameters. Furthermore, the graphics mode VRAM must be mapped into 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()`

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 LCF Requirements

Your implementation must use the LCF. Therefore, all your C source code files must include the following line:

```
#include <lcom/lcf.h>
```

This should be the first header file to be included in all your C source code files. You may create your own header files and use the `#include` directive to include them. However, note that you **must not** declare any of the functions that are specified for this lab in your own header files: they are already declared in header files included via `<lcom/lcf.h>`.

Another requirement of the LCF is that, in the definition of each of the functions specified in this Section, the name of the function must appear between parenthesis. For example, for the first function, you should do as done in the respective stub in lab5.c:

```
#include <lcom/lcf.h>


int (video_test_init)(uint16_t mode, uint8_t delay) {
  /* To be completed */
  printf("%s(0x%03x, %u): under construction\n", __func__, mode,
delay);


  return 1;
}
```

Because lab5.c already has a stub for each of the main functions that you have to implement and the name of the function of each of these stubs is already between parentheses, we suggest that you write these functions by editing lab5.c.

5.2 `video_test_init(uint16_t mode, uint8_t delay)`

The purpose of this function is that you learn how 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 `delay` seconds, it should go back to Minix's default text mode.

5.3 `video_test_rectangle(uint16_t mode, uint16_t x, uint16_t y, uint16_t width, uint16_t height, uint32_t color)`

The purpose of this function is that you learn how to change the color of pixels on the screen and to use the relevant graphics mode parameters.

This function shall change the video mode to that in its argument, map the video memory to the process' address space and draw a rectangle. Finally, upon receiving the break code of the ESC key (0x81), it should reset the video mode to Minix's default text mode and return.

The arguments `x` and `y` specify the coordinates of the rectangle's top-left vertex. (The pixel at the top-left corner of the screen has coordinates `(0,0)`.) The arguments `width` and `height` specify the width and height of the rectangle in pixels. The `color` argument specifies the color to fill the rectangle.

You can check the details in its doxygen documentation.

To allow us to give you partial credit, if there is some problem with your code, you should use the function:

```
int vg_draw_hline(uint16_t x, uint16_t y, uint16_t len, uint32_t
color)
```

which is documented here, and that you need to implement.

So that your code supports different graphics modes, it should use the `vbe_get_mode_info()` function that is provided by the LCF:

```
#include <lcom/lcf.h>


int vbe_get_mode_info(uint16_t mode, vbe_mode_info_t *vmi_p);
```

This function essentially calls the VBE function **`0x01` - Return VBE Mode Information** and is documented here, together with the `vbe_mode_info_t` struct. It provides a means to get the parameters relevant for the input graphics mode. You can then use these parameters to change the value of the appropriate location of the video RAM. For example, if the mode is `0x105`, then the resolution is `1024 x 768`, color mode is indexed and each pixel takes 8 bits, i.e. 1 byte; thus the argument color must not be larger than 255. Furthermore, to change a pixel you must write this byte to the byte in VRAM corresponding to that pixel. On the other hand, if the mode is `0x115`, then the resolution is `800 x 600`, the color mode is direct and each pixel takes 24 bits, 8 for each of the RGB components. Therefore, you should consider only the 24 less significant bits, i.e. the 3 least significant bytes, of the `color` argument. Furthermore, to change a pixel you must write these 3 bytes to the 3 bytes in VRAM corresponding to that pixel.

5.4 `int video_test_pattern(uint16_t mode, uint8_t no_rectangles, uint32_t first, uint8_t step)`

The purpose of this function is that you learn further how to to use the graphics mode parameter, namely to handle the different components of a pixel color in direct mode.

This function shall change the video mode to that in its argument. Then it should draw a pattern of colored (filled) rectangles on the screen. Finally, upon receiving the break code of the ESC key (0x81), it should reset the video mode to Minix's default text mode and return.

You can check the details in its doxygen documentation.

The pattern to be drawn is a matrix of `no_rectangles` by `no_rectangles`. All rectangles shall have the same size, i.e. the same width and the same height. If the horizontal (vertical) resolution is not divisible by `no_rectangles` then you should leave a black stripe with minimum width (height) on the right (bottom) of the screen.

The color of each rectangle depends on its coordinates in the rectangle matrix, `(row, col)` and on all the arguments.

If the color mode is indexed (or packed pixel, in VBE jargon), then the index of the color to fill the rectangle with coordinates `(row, column)` is given by the expression:

```
     index(row, col) = (first + (row * no_rectangles + col) *
step) % (1 << BitsPerPixel)
```

where `BitsPerPixel` is the value of the member of the VBEInfoBlock struct with the same name (check the `vbe_mode_info_t` in Section 4.4) for the input mode.

If the color mode is direct, then the RGB components of the color to fill the rectangle with coordinates `(row, column)` are given by the expressions:

```
     R(row, col) = (R(first) + col * step) % (1 <<
RedScreeMask)

     G(row, col) = (G(first) + row * step) % (1 <<
GreenScreeMask)

     B(row, col) = (B(first) + (col + row) * step) % (1 <<
BlueScreeMask)
```

where `RedScreenMask`, `GreenScreenMask` and `BlueScreenMask`, are the values of the members of the VBEInfoBlock struct with the same name (check the `vbe_mode_info_t` in Section 4.4) for the input mode.

To allow us to give you partial credit, if there is some problem with your code, you should use the function:

```
int vg_draw_rectangle(uint16_t x, uint16_t y, uint16_t width,
uint16_t height, uint32_t color)
```

which is documented here, and that you need to implement.

5.5 `video_test_xpm(const char *xpm[], uint16_t x, uint16_t y)`

The purpose of this function is that you learn how to draw a pixmap that is provided as an XPM image.

This function should change to video mode `0x105` and display the pixmap provided as an XPM at the specified coordinates (upper left corner of pixmap). When the user releases the ESC key (scancode `0x81`), it should reset the video mode to Minix's default text mode and return.

You can check the details in its doxygen documentation.

You can use the function `load_xpm()` that is provided by the LCF to convert an XPM into a pixmap. For more details, you can check its doxygen documentation. For further information about XPMs you can read the notes about sprites by Prof. João Cardoso ("Velho").

Note: The input XPM provided as the first argument of `video_test_xpm()` is one of the XPMs of the `pixmap.h` file. Which one, depends on the second argument (the one after the `xpm` string) of the `lab5` Minix service as specified by the following table:

First argument of `video_test_xpm()`	
Second argument of `lab5`	**XPM of the `pixmap.h` file**
0	`pic1`
1	`pic2`
2	`pic3`
3	`cross`
4	`penguin`

5.6 `video_test_move(const char *xpm[], uint16_t xi, uint16_t yi, uint16_t xf, uint16_t yf, int16_t speed, uint8_t fr_rate)`

The purpose of this function is that you learn how to move a pixmap that is provided as an XPM image.

When this function is invoked, your program should change to video mode `0x105` and display the pixmap provided as an XPM at `(xi,yi)` (upper left corner). Then it should move that sprite until its upper left corner is at position `(xf,yf)`. This movement should be done at the specified `speed` with the specified `frame_rate` in frames per second (fps). When the user releases the ESC key (scancode `0x81`), it should reset the video mode to Minix's default text mode and return, **even if the movement has not completed.**

IMP: You need only consider movements along either the horizontal or the vertical directions. I.e. either `xf` is equal to `xi` or `yf` is equal to `yi`.

IMP: If `speed` is positive it should be understood as the displacement in pixels between consecutive frames. If the `speed` is negative it should be understood as the number of frames required for a displacement of one pixel.

IMP: You must always ensure that the final position of the pixmap is the one specified in the arguments to `video_test_move()`. Thus if `speed` is positive and the length of the movement is not a multiple of `speed`, the last displacement of the pixmap in its movement will be smaller than `speed`.

You can check the details in its doxygen documentation.

Note: The input XPM provided as the first argument of `video_test_move()` is specified as described for `video_test_xpm()`.

5.7 `video_test_controller()`

The purpose of this function is that you learn how 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 call VBE function 0x0, parse the VBE controller information returned and display it on the console by calling the `vg_display_vbe_contr_info()` function, which is provided by the LCF:

```
int vg_display_vbe_contr_info(vg_vbe_contr_info_t *info_p);
```

You can check the details in its doxygen documentation. The argument of this function is the address of a variable of type `vg_vbe_contr_info_t`:

```
typedef struct {
      char VBESignature[4];
      BCD VBEVersion[2];
      char *OEMString;
      uint16_t *VideoModeList;
      uint32_t TotalMemory;
      char *OEMVendorNamePtr;
      char *OEMProductNamePtr;
      char *OEMProductRevPtr;
} vg_vbe_contr_info_t;
```

whose members you must have previously initialized, with the information returned by VBE function 0x0, Return VBE Controller Information. Please check the doxygen documentation for more detailed information. You should also read the specification of VBE Function 0x0 in the VBE 2.0 specification, pg. 19.