PP3 - 2020

1. The LCV1

In this programming test you will have to develop a Minix privileged program that interfaces with your screen's computer. For that you will use the LVC1, a graphics controller. It comes with the LCOM Protected Video API, or just LPV-API, that you have used in December's programming test (PP2), but provides also direct access to some registers for time-critical operations.

1.1. The LPV-API

In this programming test you will use only the following function of the LPV-API:

`int lpv_get_mode_info(uint16_t mode, lpv_mode_info_t * lmi_p)`

Where `lpv_mode_info_t` is:

```
typedef struct {
    uint16_t x_res;          // horizontal resolution
    uint16_t y_res;          // vertical resolution
    uint8_t bpp;             // bits per pixel
    uint8_t r_sz;            // red component size
    uint8_t r_pos;           // red component LSB position
    uint8_t g_sz;            // green component size
    uint8_t g_pos;           // green component LSB position
    uint8_t b_sz;            // blue component size
    uint8_t b_pos;           // blue component LSB position
    phys_addr_t phys_addr;  // video ram base phys address
} lpv_mode_info_t;
```

and `lmi_p`, the second argument, is the address of a previously allocated `lpv_mode_info_t` struct that will be filled with the relevant information for the LPV-mode specified in the first argument, `mode`.

This function returns 0 upon success and non-zero otherwise.

Your solution must support the LPV-modes listed in the following table.

Mode	**Screen Resolution**	**Color Model**	**Bits per pixel (R:G:B)**
0	Text Mode (25 lines x 80 chars)	N/A	N/A
1	1024x768	Indexed	8 (N/A)
2	1152x864	Direct color	32 ((8:)8:8:8)

IMP.: Note that in indexed mode, i.e. mode 1, all the members relative to the RGB components of the `lpv_mode_info_t` struct, i.e. their size and the position of their LSB, have value 0.

Furthermore, you must use the following function, **already provided by the LCF**, to map physical memory on a process' virtual address space:

```
uint8_t *video_map_phys(uint32_t ph_addr, size_t len);
```

It returns the virtual address of the specified physical memory range.

1.2. The LVC1 Control Register

The LVC1 provides among other registers a **control register** at port **0x24** that is readable/writable. The relevant bits of the control register for this test are:

Bit	7	6	5	4	3	2	1	0
CTRL. REG.	EnableSVRInt	-	-	-	LPV-mode 3	LPV-mode 2	LPV-mode 1	LPV-mode 0

bits 7

the `EnableSVRInt` bit enables the Start Vertical Retrace interrupt when set to 1. If this bit is set, the LVCR1 generates an interrupt at the start of the vertical retrace interval, via the **IRQ-line 7**. This can be used to avoid visual artifacts. (See below.)

bit 0-3

the `LPV-mode` bits, which set the desired LPV-mode

By writing to the LVC1 control register it is possible to set the LPV-mode and to enable vertical retrace interrupts.

2. The Problem

This programming test has two parts. The second part subsumes the first one. However, for ease of testing (and to avoid breaking running code) I suggest that you solve each part in its own function (the file pp.c already provides two stubs for functions `part1()` and `part2()` that you should use for the respective part).

2.1. Part 1

In this part, you must develop the following function:

```
int pp_test_square(uint8_t mode, uint16_t x, uint16_t y, uint16_t len,
uint32_t color, uint32_t delay);
```

`pp_test_square()` shall configure the video card to operate in the mode specified in its first argument, by writing to the LVC1's control register, map its frame-buffer, and then draw a **filled** square whose top-left corner is at the screen coordinates (x, y) (the pixel on the top left position of the screen has coordinates (0,0)), with a side's length (in pixels) specified in the fifth argument, `len`, the sixth argument specifies the color to draw the square. Finally the last argument specifies the delay in seconds before switching back to Minix's text mode (you must use `sleep()` to measure the delay) and exit.

For direct colors modes, the color is encoded in 3 bytes, such that the LSB encodes the B component, the middle byte encodes the G component, whereas the MSB encodes the R component.

In this part, you **must not** use the LVC1's retrace interrupt.

IMP: You should always draw the visible part of the square, even if part of it may not be displayed because it is out of the screen bounds.

IMP: **you should develop your code in the `/home/exame/pp/` directory** that was created by the setup.sh script. The pp.c file in that directory already includes the `main()` function, which you must not change , and a skeleton for `pp_test_square()`, which you are supposed to complete.

2.2. Part 2

In this part, you shall use the **LVC1's vertical retrace interrupts** to ensure synchronization with the vertical retrace. Furthermore, after a first interval of `delay` seconds displaying the square with the top-left corner at coordinates (x, y), your program must redraw the square at a position with the coordinates of the top-left corner swapped. E.g., if the `x` and `y` arguments are 300 and 200, respectively, you must draw a square at coordinates (300, 200) first, and then at coordinates (200, 300). Again, this square should be displayed for `delay` seconds, after which your program should switch back to Minix's text mode and exit.

Hint - The LVC1 is configured to use a frame rate of 30 fps. You may use this information to measure the needed time intervals.

2.3. Building, running and testing

In order to build your program you should use the Makefile that is available in `/home/lcom/labs/pp/`, by executing in that directory:

```
minix$ make depend && make
```

Hint: Remember that you can always login remotely on Minix by running the following command on a terminal (in Linux):

```
$ ssh lcom@localhost -p 2222
```

and use the newly created shell to run (and build) your program.

To learn how to run your program, you should use the command `lcom_run` and specify no program arguments:

```
minix $ lcom_run pp
Usage:
    lcom_run pp "square <mode - one of 1, 2> <x - decimal>
<y - decimal> <len - decimal> <color - hexadecimal> <delay - decimal>
-t <test no.>"
    If mode different from 1, should use 2 hexadecimal symbols per
primary color in RGB order.
```

It will output a usage message that describes the command line arguments that you can use.

IMP: In mode 1, the color pallete is identical to that used by Minix in VBE mode 0x105. I.e. the pallete has only 64 colors, and color 0 is also black. Color 4 is red.

The `-t` option is **mandatory**. The `<test no.>` parameter is an integer between **1** and **6**. For all test no. 's, the behavior is deterministic, therefore you should get always the same results for the same set of arguments. The following table summarizes each test case:

Test No.	**Test description**	**Guaranteed score**
1	Tests mapping of the video RAM	5%
2	Tests graphics mode setting.	10%
3	Tests resetting to text mode	5%
4	Tests drawing of square in part 1.	About 10% per mode, if your code passes most tests for that mode.

5	Tests drawing of square for the first time in part 2.	This will be used instead of test 4 if you do part 2, because test 4 will fail.
6	Tests the drawing of the square in the first position in part 2.	About 5% per mode, if your code passes most tests for that mode.
7	Tests the drawing of the square in the second position in part 2. Note that the test fails if there are more than one square on the screen.	About 5% per mode, if your code passes most tests for that mode.

Note that all these tests are parametric. I.e., you can specify arbitrary values for the different arguments.

IMP: To choose other input parameters, you may wish to use the following helper function that displays the LPV-mode information:

```
int pp_display_lpv_mode_info(uint16_t mode);
```

If your program terminates normally, it will print either:

```
Test succeeded.
```

indicating success, or

```
Test FAILED!
```

indicating failure. In case of failure, you should examine carefully the output on the terminal or in the output.txt file and the trace with the calls your program performed in the trace.txt file. (Both text files are in the directory where you executed the `lcom_run` command.)