Laboratory Exercise 6

Introduction to Graphics and Animation

The purpose of this exercise is to learn how to draw graphics and perform animation. You will create a Linux* character device driver that uses the video-out port on a DE-series board to display graphics. To do this exercise you need to know how to write Linux kernel modules and character device drivers. This exercise is meant to be done after Lab Exercise 3, which introduces character device drivers.

Background Information

You may want to familiarize yourself with the documentation for the DE1-SoC Computer that pertain to the use of the video-out port. The DE1-SoC Computer includes a video-out port with a VGA controller that can be connected to a standard VGA monitor. The VGA controller supports a screen resolution of 640 × 480. The image that is displayed by the VGA controller is derived from two sources: a *pixel* buffer, and a *character* buffer. The pixel buffer is described below, and the character buffer will be discussed later in the exercise.

Pixel Buffer

The pixel buffer for the video-out port holds the data (color) for each pixel that is displayed by the VGA controller. As illustrated in Figure 1, the pixel buffer in the DE1-SoC Computer provides an image resolution of 320 × 240 pixels, with the coordinate (0,0) being at the top-left corner of the image. Since the VGA controller supports the resolution 640 × 480, each of the pixel values in the pixel buffer is replicated in both the *x* and *y* dimensions when it is being displayed on the VGA screen.

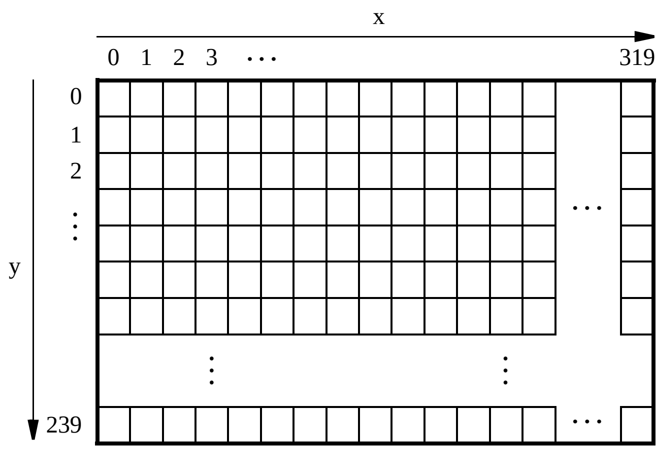
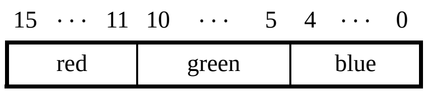
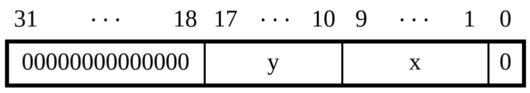


Figure 1: Pixel buffer coordinates in the DE1-SoC Computer.

Figure 2 shows how pixel colors and pixel addresses are specified for the DE1-SoC Computer. As illustrated in part a of the figure, each pixel color has 16 bits, with five bits for the blue and red components, and six bits for green. As depicted in part b of Figure 2, pixels are addressed in the pixel buffer by using the combination of a *base* address and an *x,y* offset. In the DE1-SoC Computer the default address of the pixel buffer is 0xC8000000, which corresponds to the starting address of the FPGA on-chip memory. Using this scheme, the pixel at coordinates $(0, 0)$ has the address 0xC8000000, the pixel $(1, 0)$ has the address *base* + $(00000000\ 000000001\ 0)_2$ = 0xC8000002, the pixel $(0, 1)$ has the address *base* + $(00000001\ 000000000\ 0)_2$ = 0xC8000400, and the pixel at coordinates $(319, 239)$ has the address *base* + $(11101111\ 100111111\ 0)_2$ = 0xC803BE7E.



(a) Pixel color



(b) Pixel (x,y) offset

Figure 2: Pixel values and addresses in the DE1-SoC Computer.

You can create an image by writing color values into the pixel addresses as described above. A dedicated *pixel buffer controller* reads this pixel data from the memory and sends it to the VGA display. The controller reads the pixel data in sequential order, starting with the pixel data corresponding to the upper-left corner of the VGA screen and proceeding to read the whole buffer until it reaches the data for the lower-right corner. This process is then repeated, continuously. You can modify the pixel data at any time, by writing to the pixel addresses. Writes to the pixel buffer are automatically interleaved in the hardware with the read operations that are performed by the pixel buffer controller.

It is also possible to prepare a new image for the VGA display without changing the content of the pixel buffer, by using the concept of *double-buffering*. In this scheme two pixel buffers are involved, called the *front* and *back* buffers, as described below.

Double Buffering

As mentioned above, a pixel buffer controller reads data out of the pixel buffer so that it can be displayed on the VGA screen. This controller includes a programming interface in the form of a set of registers, as shown in Figure 3. The register at address 0xFF203020 is called the *Buffer* register, and the one at address 0xFF203024 is the *Backbuffer* register. Each of these registers stores the starting address of a pixel buffer. The Buffer register holds the address of the pixel buffer that is displayed on the VGA screen. As mentioned above, in the default configuration of the DE1-SoC Computer this Buffer register is set to the address 0xC8000000, which points to the start of the FPGA on-chip memory. The default value of the Backbuffer register is also 0xC8000000, which means that there is only one pixel buffer. But software can modify the address stored in Backbuffer, thereby creating a second pixel buffer. An image can be drawn into this second buffer by writing to its pixel addresses. This image is not displayed on the VGA monitor until a pixel buffer *swap* is performed, explained below.

A pixel buffer swap is caused by writing the value 1 to the Buffer register. This write operation does not directly modify the content of the Buffer register, but instead causes the contents of the Buffer and Backbuffer registers to be swapped. The swap operation does not happen right away; it occurs at the end of a VGA screen-drawing cycle, after the last pixel in the bottom-right corner has been displayed. This time instance is referred to as the *vertical synchronization* time, and occurs every 1/60 seconds. Software can poll the value of the S bit in the *Status* register, at address 0xFF20302C, to see when the vertical synchronization has happened. Writing the value 1 into the Buffer register causes S to be set to 1. Then, when the swap of the Buffer and Backbuffer registers has been completed S is reset back to 0. The programming interface includes a *Resolution* register, shown in the figure, which contains the X and Y resolution of the pixel buffer(s). Additional information is provided in the *Status* register. The m and n bits specify the number of y and x VGA address bits, respectively. The *BS* bits indicate pixel-size; if this value is set to 7 then a pixel is one byte, or if it is set to 15 then a pixel is two bytes.

In a typical application the pixel buffer controller is used as follows. While the image contained in the pixel buffer that is pointed to by the Buffer register is being displayed, a new image is drawn into the pixel buffer pointed to by the Backbuffer register. When this new image is ready to be displayed, a pixel buffer swap is performed. Then,

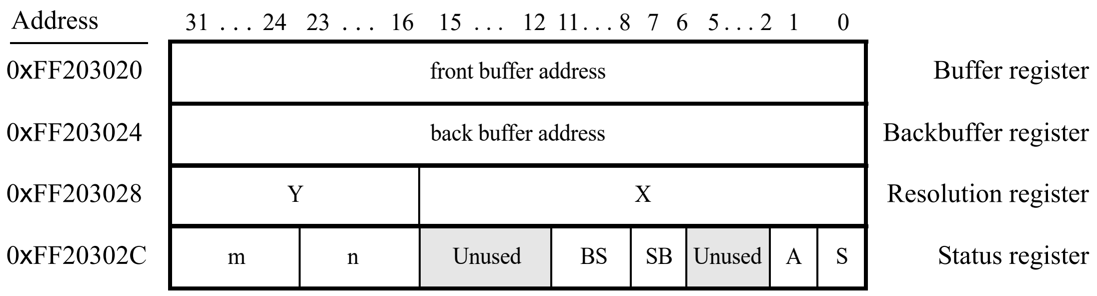


Figure 3: Pixel buffer controller registers.

the pixel buffer that is now pointed to by the Backbuffer register, which was already displayed, is cleared and the next new image is drawn. In this way, the next image to be displayed is always drawn into the "back" pixel buffer, and the "front" and "back" buffer pointers are swapped when the new image is ready to be displayed. Each time a swap is performed software has to synchronize with the VGA controller by waiting until the S bit in the Status register becomes 0.

Part I

You are to create a character device driver that controls the VGA display. Your driver should use the file */dev/video* to communicate with the user. Reading from this file should return the character string "`ccc rrr`", where the three-digit decimal number `ccc` is the number of columns in the VGA screen, and `rrr` is the number of rows. One way to read from the device driver is to type a command such as `cat /dev/video` on a Linux Terminal window. The device driver should support the following commands that can be written to the file */dev/video*: `clear`, and `pixel x,y color`. The `clear` command should erase the VGA screen by setting all pixels in the pixel buffer to the color 0 (*black*). The `pixel` command should set the pixel on the VGA screen at the coordinates (x, y) to the value `color`. One way to write to your video driver is to type a command such as `echo clear > /dev/video` on a Linux Terminal window. The command `echo "pixel 319,239 07E0" > /dev/video` would set the pixel at the bottom-right corner of the screen to the color green.

An outline of the required code for the character device driver is given in Figure 4. Lines 1 to 7 include header files that are needed for the driver. Global variables that are used to access the pixel buffer, which will be described later, are declared in lines 10 to 13. Line 16 is a placeholder for the declarations of function prototypes and variables that are needed for the character device driver. Prototypes have to be declared for the functions that are executed when opening, reading, writing, and closing the driver. Data structures of type `file_operations` and `miscdevice` have to be declared and initialized in the function *start_video*, shown in lines 19 to 41, which is executed when the video driver is inserted into the Linux kernel. Refer to Exercise 3 for a more detailed discussion about the functions and variables that are needed for character device drivers.

To provide the kernel module with access to the FPGA light-weight bridge, line 25 calls the `ioremap_nocache` function. Information about this function, and the FPGA light-weight bridge, can be found in the tutorial *Using Linux on DE-series Boards*. Line 30 computes the base address for the pixel buffer controller registers, which are illustrated in Figure 3. This address is then passed to the function named `get_screen_specs`, which reads the *Resolution* register in the pixel controller, so that it can use this information to set the global variables *resolution_x* and *resolution_y*.

In Line 34 `ioremap_nocache` is called again, to map the physical addresses of the pixel buffer into virtual addresses. This code assumes that the pixel buffer is in its default location, which is within the FPGA on-chip memory. The `clear_screen` function is then called, which is used to set all pixels in the pixel buffer to the color 0, which is *black*.

```
#include <linux/fs.h>           // struct file, struct file_operations
#include <linux/init.h>         // for __init, see code
#include <linux/module.h>       // for module init and exit macros
#include <linux/miscdevice.h>   // for misc_device_register and struct miscdev
#include <linux/uaccess.h>      // for copy_to_user, see code
#include <asm/io.h>             // for mmap
#include "address_map_arm.h"

// Declare global variables needed to use the pixel buffer
void *LW_virtual;           // used to access FPGA light-weight bridge
volatile int * pixel_ctrl_ptr;   // virtual address of pixel buffer controller
int pixel_buffer;           // used for virtual address of pixel buffer
int resolution_x, resolution_y;  // VGA screen size

// Declare variables and prototypes needed for a character device driver
... code not shown

/* Code to initialize the video driver */
static int __init start_video(void)
{
   // call misc_register (...) to create the device driver
   ... code not shown

   // generate a virtual address for the FPGA lightweight bridge
   LW_virtual = ioremap_nocache (0xFF200000, 0x00005000);
   if (LW_virtual == 0)
      printk (KERN_ERR "Error: ioremap_nocache returned NULL\n");

   // Create virtual memory access to the pixel buffer controller
   pixel_ctrl_ptr = (unsigned int *) (LW_virtual + 0x00003020);
   get_screen_specs (pixel_ctrl_ptr); // determine X, Y screen size

   // Create virtual memory access to the pixel buffer
   pixel_buffer = (int) ioremap_nocache (0xC8000000, 0x0003FFFF);
   if (pixel_buffer == 0)
      printk (KERN_ERR "Error: ioremap_nocache returned NULL\n");

   /* Erase the pixel buffer */
   clear_screen ( );
   return 0;
}
```

Figure 4: An outline of the video-driver code (Part a).

```
void get_screen_specs(volatile int * pixel_ctrl_ptr)
{
   ... code not shown
}
void clear_screen( )
{
   ... code not shown
}
void plot_pixel(int x, int y, short int color)
{
   ... code not shown
}
static void __exit stop_video(void)
{
   /* unmap the physical-to-virtual mappings */
   iounmap (LW_virtual);
   iounmap ((void *) pixel_buffer);

   /* Remove the device from the kernel */
   ... code not shown
}
static int device_open(struct inode *inode, struct file *file)
{
   return SUCCESS;
}
static int device_release(struct inode *inode, struct file *file)
{
   return 0;
}
static ssize_t device_read(struct file *filp, char *buffer,
    size_t length, loff_t *offset)
{
   ... code not shown
}

static ssize_t device_write(struct file *filp, const char
    *buffer, size_t length, loff_t *offset)
{
   ... code not shown
}

MODULE_LICENSE("GPL");
module_init (start_video);
module_exit (stop_video);
```

Figure 4. An outline of the video-driver code (Part *b*).

Part *b* of Figure 4 gives an outline for the rest of the functions that are required for the character device driver. A detailed discussion of these functions can be found in Laboratory Exercise 3.

Perform the following:

1. Create a file named *video.c* and fill in the missing code from Figure 4. Create a Makefile for your character device driver. Compile the code to create the kernel module *video.ko*, and insert this module into the Linux kernel.

2. Connect a VGA monitor to the DE-series board, or use the available *VGA Emulator* via a *VNC* Viewer application.

3. Test your video character device driver by using a Terminal window. Reading from the file */dev/video* should return the string “320 240”, which provides the VGA screen size. Use `pixel` commands to color some pixels on the screen, and send the `clear` command to the video driver to erase the VGA display.

4. Create a user-level program called *part1.c* that reads and writes to your video device driver. A skeleton of an example program is given in Figure 5. Fill in the rest of the code so that it performs some operations using the device driver. For example, you could write a loop that uses `pixel` commands to fill the entire VGA screen with a certain color. Compile your program using a command such as `gcc -Wall -o part1 part1.c`. Test your program by trying various operations.

```c
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <fcntl.h>
#include <unistd.h>

#define video_BYTES 8         // number of characters to read from /dev/video

int screen_x, screen_y;

int main(int argc, char *argv[]){

   int video_FD;             // file descriptor
   char buffer[video_BYTES];  // buffer for data read from /dev/video
   char command[64];         // buffer for commands written to /dev/video
   int x, y;

   // Open the character device driver
   if ((video_FD = open("/dev/video", O_RDWR)) == -1)
   {
      printf("Error opening /dev/video: %s\n", strerror(errno));
      return -1;
   }
   // Set screen_x and screen_y by reading from the driver
   ... code not shown
   // Use pixel commands to color some pixels on the screen
   ... code not shown

   close (video_FD);
   return 0;
}
```

Figure 5: A program that communicates with */dev/video*.

Part II

In this part you will add a simple line-drawing algorithm to your video driver. Drawing a line on a screen requires coloring pixels between two coordinates (x_1, y_1) and (x_2, y_2), such that the pixels represent the desired line as closely as possible. Consider the example in Figure 6, where we want to draw a line between coordinates $(1, 1)$ and $(12, 5)$. The squares in the figure represent the location and size of pixels on the screen. As indicated in the figure, we cannot draw the line precisely—we can only draw a shape that is similar to the line by coloring the pixels that fall closest to the line’s ideal location on the screen.

We can use algebra to determine which pixels to color. This is done by using the end points and the slope of the line. The slope of our example line is $slope = (y_2 - y_1)/(x_2 - x_1) = 4/11$. Starting at point $(1, 1)$ we move along the x axis and compute the y coordinate for the line as follows:

$$y = y_1 + slope \times (x - x_1)$$

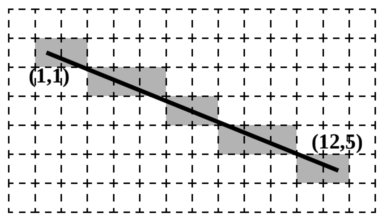


Figure 6: Drawing a line between coordinates $(1, 1)$ and $(12, 5)$.

Thus, for column $x = 2$, the y location of the pixel is $1 + \frac{4}{11} \times (2 - 1) = 1\frac{4}{11}$. Since pixel locations are defined by integer values we round the y coordinate to the nearest integer, and determine that in column $x = 2$ we should color the pixel at $y = 1$. For column $x = 3$ we perform the calculation $y = 1 + \frac{4}{11} \times (3 - 1) = 1\frac{8}{11}$, and round the result to $y = 2$. Similarly, we perform such computations for each column between x_1 and x_2.

The approach of moving along the x axis has drawbacks when a line is steep. A steep line spans more rows than it does columns, and hence has a slope with absolute value greater than 1. In this case our calculations will not produce a smooth-looking line. Also, if the line is vertical we cannot use the slope to make a calculation. To address this problem, we can alter the algorithm to move along the y axis when a line is steep. With this change, we can implement a line-drawing algorithm known as *Bresenham's algorithm*. A key property of this algorithm is that all variables are *integers*. Pseudo-code for this algorithm is given in Figure 7. The first 15 lines of the algorithm make the needed adjustments depending on whether or not the line is steep. Then, in lines 17 to 22 the algorithm increments the *x* variable 1 step at a time and computes the *y* value. The *y* value is incremented when needed to stay as close to the ideal location of the line as possible. Bresenham's algorithm calculates an *error* variable to decide whether or not to increment each *y* value. The *error* variable takes into account the relative difference between the width of the line (*deltax*) and height of the line (*deltay*) in deciding how often *y* should be incremented. The version of the algorithm shown in Figure 7 uses only integers to perform all calculations.

```
 1  draw_line(x0, x1, y0, y1)
 2     is_steep = (abs(y1 - y0) > abs(x1 - x0))
 3     if is_steep then
 4        swap(x0, y0)
 5        swap(x1, y1)
 6     if x0 > x1 then
 7        swap(x0, x1)
 8        swap(y0, y1)
 9
10     deltax = x1 - x0
11     deltay = abs(y1 - y0)
12     error = -(deltax / 2)
13     y = y0
14     if y0 < y1 then y_step = 1 else y_step = -1
15
16     for x from x0 to x1
17        if is_steep then draw_pixel(y,x) else draw_pixel(x,y)
18        error = error + deltay
19        if error ≥ 0 then
20           y = y + y_step
21           error = error - deltax
```

Figure 7: Pseudo-code for a line-drawing algorithm.

Perform the following:

1. Using your character device driver from Part I it would be possible to draw lines by repeatedly sending `pixel` commands to the driver. However, this would be an inefficient approach due to the large number of writes that may have to be made to the file */dev/video*. A better approach is to implement a line-drawing algorithm within the video driver. Augment your driver by adding a function called *draw_line* that implements Bresenham's algorithm. Also, add a new command `line x1,y1 x2,y2 color` that invokes this function.
2. Remove your kernel module from Part I, recompile the new version which includes the `line` command, and then re-insert the module back into the kernel.
3. Use the `echo` command in a Terminal window to test your line-drawing function.
4. Part of a user-level program that draws a few lines on the VGA screen is depicted in Figure 8. Fill in the missing parts of this program. Compile and test it.

```
#include <stdio.h>
... other include statements not shown

#define video_BYTES 8   // number of characters to read from /dev/video

int screen_x, screen_y;

int main(int argc, char *argv[]){

   int video_FD;                  // file descriptor
   char c, video_buffer[video_BYTES];  // buffer for video char data
   char command[64];              // buffer for command data

   // Open the character device driver
   if ((video_FD = open("/dev/video", O_RDWR)) == -1)
   {
      printf("Error opening /dev/video: %s\n", strerror(errno));
      return -1;
   }

   // Read VGA screen size from the video driver
   ... code not shown

   /* Draw a few lines */
   sprintf (command, "line %d,%d %d,%d %X\n", 0, screen_y - 1, screen_x - 1,
       0, 0xFFE0);  // yellow
   write (video_FD, command, strlen(command));
   sprintf (command, "line %d,%d %d,%d %X\n", 0, screen_y - 1,
      (screen_x >> 1) - 1, 0, 0x07FF); // cyan
   write (video_FD, command, strlen(command));
      (screen_x >> 2) - 1, 0, 0x07E0); // green
   write (video_FD, command, strlen(command));
   ... etc.

   close (video_FD);
   return 0;
}
```

Figure 8: A user-level program that draws a few lines.

Part III

Animation is an exciting part of computer graphics. Moving a displayed object is an illusion created by showing this same object at different locations on the screen. A simple way to "move" an object is to first draw the object at one position, and then after a short time erase the object and draw it again at another nearby position.

To realize animation it is necessary to move objects at regular time intervals. The VGA controller in the DE1-SoC Computer redraws the screen every $1/60^{th}$ of a second. Since the image on the screen cannot change more often than that, it is reasonable to control an animation using this unit of time.

To ensure that you change the image only once every $1/60^{th}$ of a second, use the pixel buffer controller to synchronize with the vertical synchronization cycle of the VGA controller. As we discussed in the background section of this exercise, synchronizing with the VGA controller can be accomplished by writing the value 1 into the *Buffer* register in the pixel buffer controller, and then waiting until bit S of the *Status* register becomes equal to 0. For this part of the exercise you do not need to use a back buffer, so ensure that the *Buffer* and *Backbuffer* addresses in the pixel buffer controller are the same. In this approach, a pixel buffer "swap" can be used as a way of synchronizing with the VGA controller via the *S* bit in the *Status* register.

Perform the following:

1. Augment your video driver so that it includes the ability to perform a pixel buffer swap, as described above. Also add a new command to the video driver, called `sync`, that allows a user-level program to synchronize with the VGA controller. Remove the previous version of the driver from the Linux kernel, and compile and insert the new version.
2. Write a user-level program that creates a simple animation using your video driver. The animation should "move" a horizontal line up and down on the screen and "bounce" the line off the top and bottom edges of the display. Your program should first clear the screen and draw the line at a starting row on the screen. Then, in an endless loop you should perform a VGA synchronization, erase the line (by drawing the line using black), and redraw it one row above or below the last one. When the line reaches the top, or bottom, of the screen it should start moving in the opposite direction.
3. Compile and execute your code. Notice how long it takes for the horizontal line to move through the 240 lines of the VGA display. It should take $240 \times 1/60 = 4$ seconds.

Part IV

Having gained the basic knowledge, you can now create a more interesting animation. You are to create an animation of eight small filled rectangles on the screen. These rectangles should appear to be moving continuously and "bouncing" off the edges of the screen. The rectangles should be connected with lines to form a chain. An illustration of the animation is given in Figure 9. Part a of the figure shows one position of the rectangles with arrows that indicate the directions of movement, and Figure 9b shows a subsequent position of the rectangles. In each step of your animation each of the rectangles should appear to "move" on a diagonal line: up/left, up/right, down/left, or down/right. Move the rectangles one row and one column at a time on the VGA screen.

To make the animation look slightly different each time you run it, you can use the C library function *rand ()* to help calculate initial positions for each of the rectangles, and to determine their directions of movement.

Perform the following:

1. Augment your video driver to add a new command: `box x1,y1 x2,y2 color`. Remove the previous version of the driver from the Linux kernel, and compile and insert the new version.
2. Write a user-level program to implement your animation. Use both a front and back buffer in your program, so that you can avoid making changes to the image while it is being displayed by the pixel buffer controller. If you are using the DE1-SoC Computer, then in addition to placing one pixel buffer in the FPGA

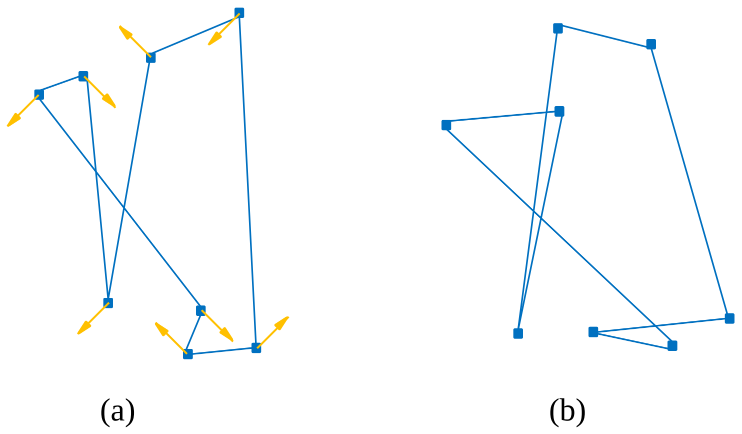


Figure 9: Two instants of the animation.

on-chip memory, you can place the other buffer in the SDRAM memory, which has the starting address `0xC0000000`. This choice is also available in the DE10-Standard Computer. If using the DE10-Nano Computer, then place both pixel buffers in the FPGA on-chip memory; if the first buffer starts at address `0xC8000000`, then the other buffer can begin at `0xC8000000` $+ (512 \times 256)$. An outline of a suitable program is illustrated in Figure 10. Compile, execute, and test your program.

3. Experiment with your code by modifying it to use just a single pixel buffer (simply change the address of the back buffer to be the same as the front buffer). Notice how the animation differs with this change.

Part V

For this part of the exercise you are to enhance the animation from Part IV so that during the animation the following changes can take place:

1. The speed of movement of the rectangles can be increased or decreased
2. The number of rectangles can be increased or decreased
3. The lines between rectangles can be drawn or not drawn

In Part IV the animation speed was set by the 1/60 seconds VGA vertical synchronization time. One way to control the speed of animation is to make use of a timer. You can use the *nanosleep* timer that is provided by the Linux kernel. In this scheme, the main program would draw the next step of the animation each time the timer expires. Lengthening the timeout would produce a slower animation, and shortening the timeout would speed up the animation. The maximum speed of the animation would be limited by the 1/60 seconds VGA synchronization time, as it was in Part IV. To cause the animation to appear to move more quickly than in Part IV, you have to increase the screen-distance that the rectangles move in each step of the animation.

Perform the following:

1. Implement the animation control discussed above. All of your code changes should be done in a user-level program; you do not need to modify your video driver. The user should be able to control the animation by using the KEY pushbuttons and SW slide switches. Implement the actions shown in Table 1. When any of the SW switches is set to the "on" position the lines between objects should not be drawn; only when all SW switches are set to the "off" position should the lines appear. Access both the SW and KEY switches by using their respective character device drivers.
2. Compile and test your animation code.

```c
#include <stdio.h>
... other include statements not shown

/* Declare global variables needed for the animation */
... code not shown

void draw (int);

volatile sig_atomic_t stop;
void catchSIGINT(int signum){
    stop = 1;
}

int main(int argc, char *argv[]){

   int i, video_FD;  // file descriptor
   ... other declarations not shown

   // catch SIGINT from ^C, instead of having it abruptly close this program
   signal(SIGINT, catchSIGINT);

   ... code to open the video driver shown

   // set random initial position and "delta" for all rectangle boxes
   ... code not shown

   while (!stop)
   {
      draw (video_FD);
      // sync with VGA display
      write (video_FD, "sync", 5);
   }
   close (video_FD);
   return 0;
}

/* Code that draws the animation objects, and updates their positions */
void draw(int fd)
{
   ... code not shown
}
```

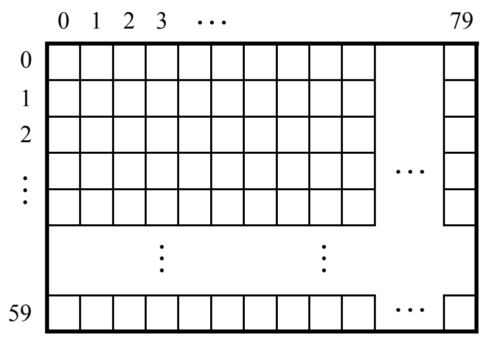
Figure 10: A user-level program that makes an animation.

Table 1: Animation control for the DE1-SoC and DE10-Standard boards.

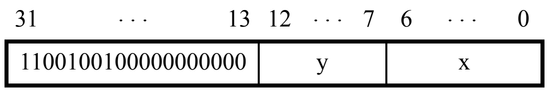
KEY	**Action**
KEY_0	The speed of animation should be increased by some noticeable amount
KEY_1	The speed of animation should be decreased by some noticeable amount
KEY_2	The number of displayed objects should be increased by one
KEY_3	The number of displayed objects should be decreased by one

Part VI

We mentioned previously that the image displayed by the VGA controller can be derived from both a *pixel* buffer, and a *character* buffer. For this part of the exercise you are to enhance your video driver so that it supports the display of text characters. The character buffer is stored in FPGA on-chip memory in the DE1-SoC Computer. Figure 11a depicts the character buffer for the VGA display, which has a resolution of 80 $\times$ 60 characters. Each character occupies an 8 $\times$ 8 block of pixels on the screen. Characters are stored in each of the locations shown in Figure 11a using their ASCII codes; when you store an ASCII character into the buffer, a corresponding pattern of pixels is automatically generated and displayed using a built-in font. Part b of Figure 11 shows that characters are addressed in the memory by using the combination of a *base* address, which has the value $(\text{C9000000})_{16}$, and an *x,y* offset. Using this scheme, the character at coordinates $(0,0)$ has the address $(\text{C9000000})_{16}$, $(1,0)$ has the address *base* + $(000000\ 0000001)_2$ = $(\text{C9000001})_{16}$, $(0,1)$ has the address *base* + $(000001\ 0000000)_2$ = $(\text{C9000080})_{16}$, and the character at location $(79,59)$ has the address *base* + $(111011\ 1001111)_2$ = $(\text{C9001DCF})_{16}$.



(a) Character buffer coordinates



(b) Character buffer addresses

Figure 11: Character buffer coordinates and addresses.

The character buffer has an associated controller, with a register interface like the one shown in Figure 3. The base address of this controller is 0xFF203030. You can read from the *Resolution* register to obtain the number of text columns (X) and rows (Y) on the screen. For the VGA screen, the values will be 80 columns $\times$ 60 rows.

Perform the following:

1. Enhance your kernel module from Part V by adding two new commands: `erase`, and `text x,y string`. The `erase` command should clear all characters from the VGA screen (by filling the screen with the ' ' space character). The `text` command should store the ASCII characters in the *string* into coordinates starting at `(x, y)` in the character buffer. Remove the previous version of your video driver from the Linux kernel, and compile and insert the new version.

2. Augment your user-level program from Part V so that it displays, in the upper-left corner of the screen, the number of video frames that have been drawn. Using the default animation speed, the frame counter should increment at the rate of 60 times per second.

3. Compile and test your animation.