EMBEDDED LINUX KERNEL DEVELOPMENT REPORT 4

1 INTRODUCTION

This LAB's purpose is to learn how the Linux framebuffer works and use it through a userspace YUV video decoder application.

As the framebuffer uses RGBa format, we need to convert each YUV pixels to a RGB equivalent.

The format is YUV 4:2:0, meaning that for a 4x4 pixels square we have 4 Y, 2 U and 2 V data info.

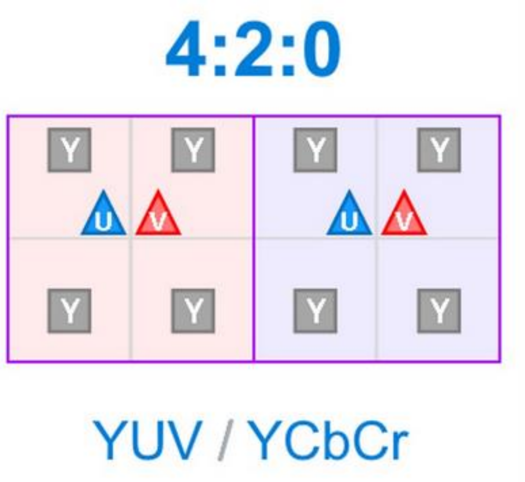


Figure 1- YUV 4:2:0 format (http://www.latelierducable.com)

The data in each frame is organized as follow.

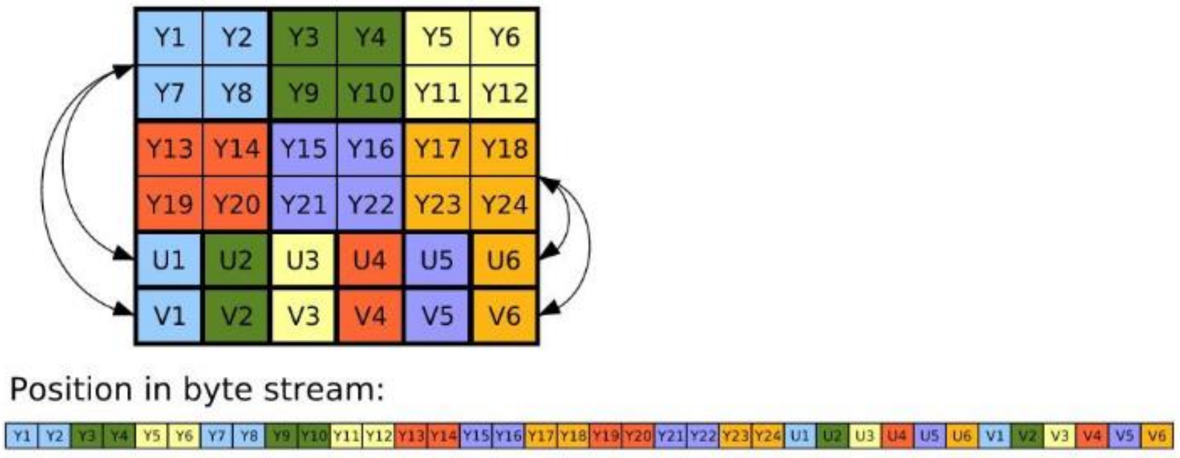


Figure 2- YUV data layout for one frame

2 FLOATING POINT CONVERSION

We first load the template code provided in LABS/TP_FRAMEBUFFER. We can test with the blueRectangle function that the writing is done correctly.

As the PC's screen is constantly refreshed by the system, we need to switch to console mode (**CTRL+ALT+F1**), then go back to GUI mode (**CTRL+ALT+F7**).

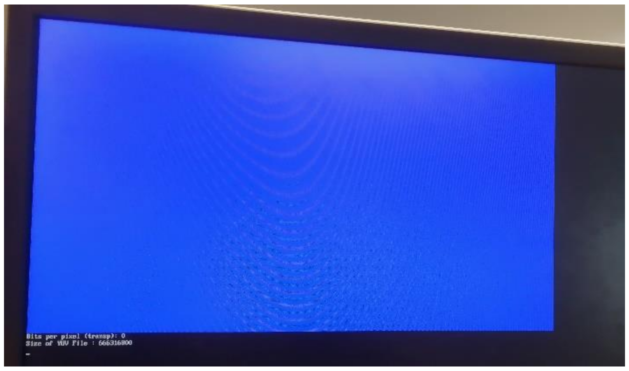
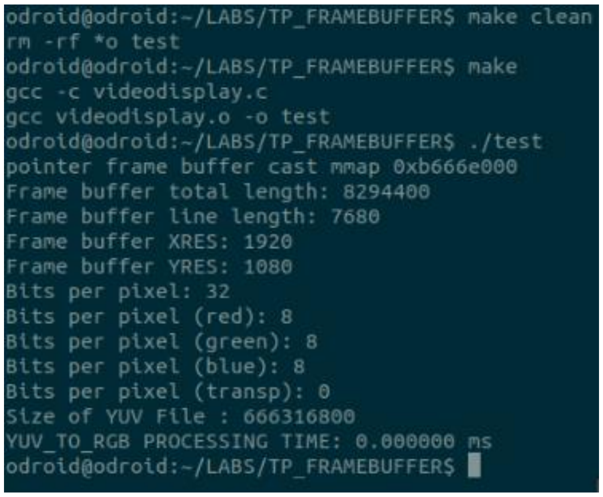


Figure 3- blue screen test code



```
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ make clean
rm -rf *o test
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ make
gcc -c videodisplay.c
gcc videodisplay.o -o test
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ ./test
pointer frame buffer cast mmap 0xb666e000
Frame buffer total length: 8294400
Frame buffer line length: 7680
Frame buffer XRES: 1920
Frame buffer YRES: 1080
Bits per pixel: 32
Bits per pixel (red): 8
Bits per pixel (green): 8
Bits per pixel (blue): 8
Bits per pixel (transp): 0
Size of YUV File : 666316800
YUV_TO_RGB PROCESSING TIME: 0.000000 ms
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ 
```

Figure 4- terminal output of testcode

This code uses the *mmap* method to map the framebuffer accessible through the /dev/fb0 file to the *fb_ptr* pointer. We can then directly write to the frame buffer using this pointer as the start address.

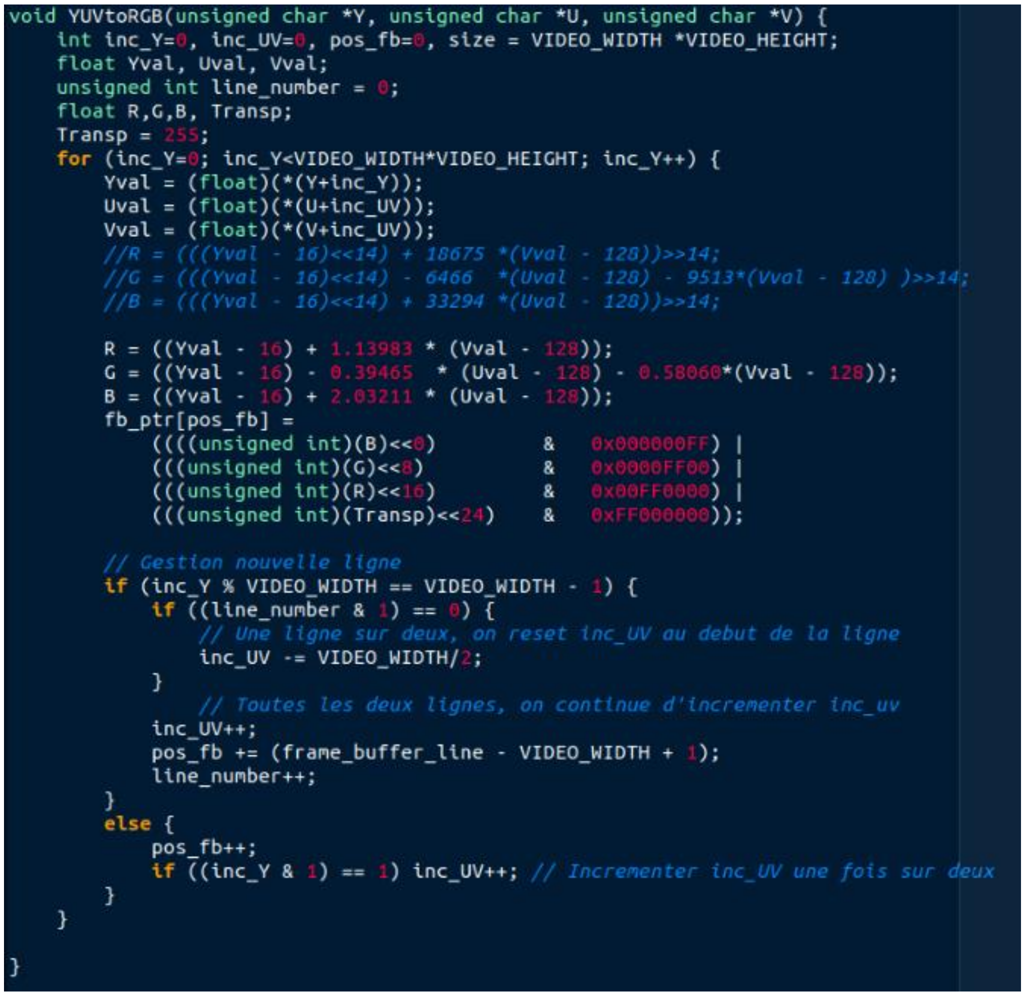
We can replace the blueRectangle method by the *YUVtoRGB* method and use the following formula to do the conversion.

```
R = (Y - 16) + 1,13983 * (V - 128)
G = (Y - 16) - 0,39465 * (U - 128) - 0,58060 * (V - 128)
B = (Y - 16) + 2,03211 * (U - 128)
```

Figure 5- YUV to RGB formula

Doing the conversion, we need to mind the inc_Y and inc_UV values so that they match the correct pixel in buffer.

For inc_Y we just need to increment by one, but for inc_UV we can refer to Figure1. We need to increment inc_UV by 1 for 2 pixels crossed, and reset the counter at the start of the line for each one of two new lines.



```
void YUVtoRGB(unsigned char *Y, unsigned char *U, unsigned char *V) {
    int inc_Y=0, inc_UV=0, pos_fb=0, size = VIDEO_WIDTH *VIDEO_HEIGHT;
    float Yval, Uval, Vval;
    unsigned int line_number = 0;
    float R,G,B, Transp;
    Transp = 255;
    for (inc_Y=0; inc_Y<VIDEO_WIDTH*VIDEO_HEIGHT; inc_Y++) {
        Yval = (float)(*(Y+inc_Y));
        Uval = (float)(*(U+inc_UV));
        Vval = (float)(*(V+inc_UV));
        //R = (((Yval - 16)<<14) + 18675 *(Vval - 128))>>14;
        //G = (((Yval - 16)<<14) - 6466  *(Uval - 128) - 9513*(Vval - 128) )>>14;
        //B = (((Yval - 16)<<14) + 33294 *(Uval - 128))>>14;

        R = ((Yval - 16) + 1.13983 * (Vval - 128));
        G = ((Yval - 16) - 0.39465  * (Uval - 128) - 0.58060*(Vval - 128));
        B = ((Yval - 16) + 2.03211 * (Uval - 128));
        fb_ptr[pos_fb] =
            ((((unsigned int)(B)<<0)        &  0x000000FF) |
            (((unsigned int)(G)<<8)         &  0x0000FF00) |
            (((unsigned int)(R)<<16)        &  0x00FF0000) |
            (((unsigned int)(Transp)<<24)   &  0xFF000000));

        // Gestion nouvelle ligne
        if (inc_Y % VIDEO_WIDTH == VIDEO_WIDTH - 1) {
            if ((line_number & 1) == 0) {
                // Une ligne sur deux, on reset inc_UV au debut de la ligne
                inc_UV -= VIDEO_WIDTH/2;
            }
                // Toutes les deux lignes, on continue d'incrementer inc_uv
            inc_UV++;
            pos_fb += (frame_buffer_line - VIDEO_WIDTH + 1);
            line_number++;
        }
        else {
            pos_fb++;
            if ((inc_Y & 1) == 1) inc_UV++; // Incrementer inc_UV une fois sur deux
        }
    }

}
```

Figure 6- YUVtoRGB with floating point solution

3 FIXED POINT

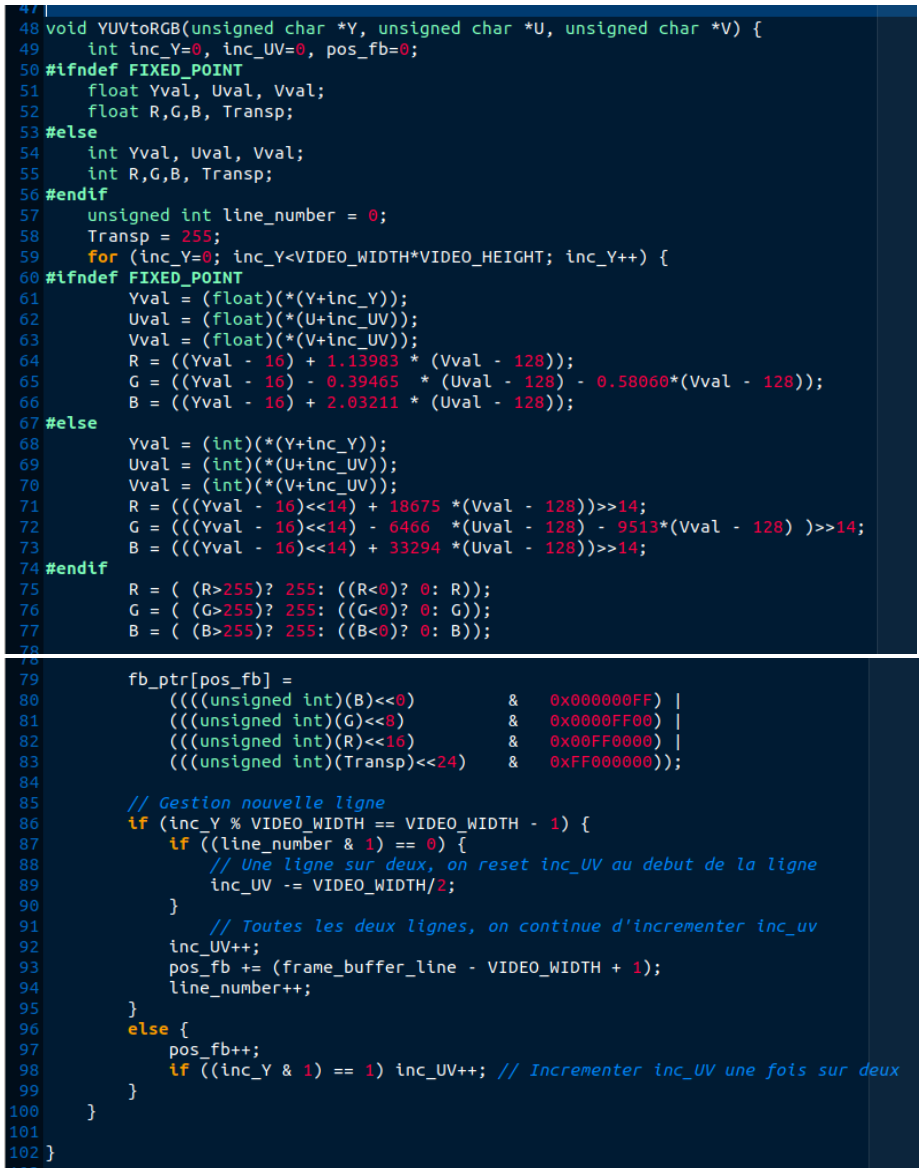
As the floating point calculus can be resource heavy, we switch to fixed point values.

By doing a 14bits shift on the previous formula, we can avoid *float*.

```
R = ((Y - 16)<<14 + 18675 * (V - 128)) >>14
G = ((Y - 16)<<14 - 6466 * (U - 128) - 9513 * (V - 128)) >>14
B = ((Y - 16)<<14 + 33294 * (U - 128)) >>14
```

Figure 7- new fixed point formula

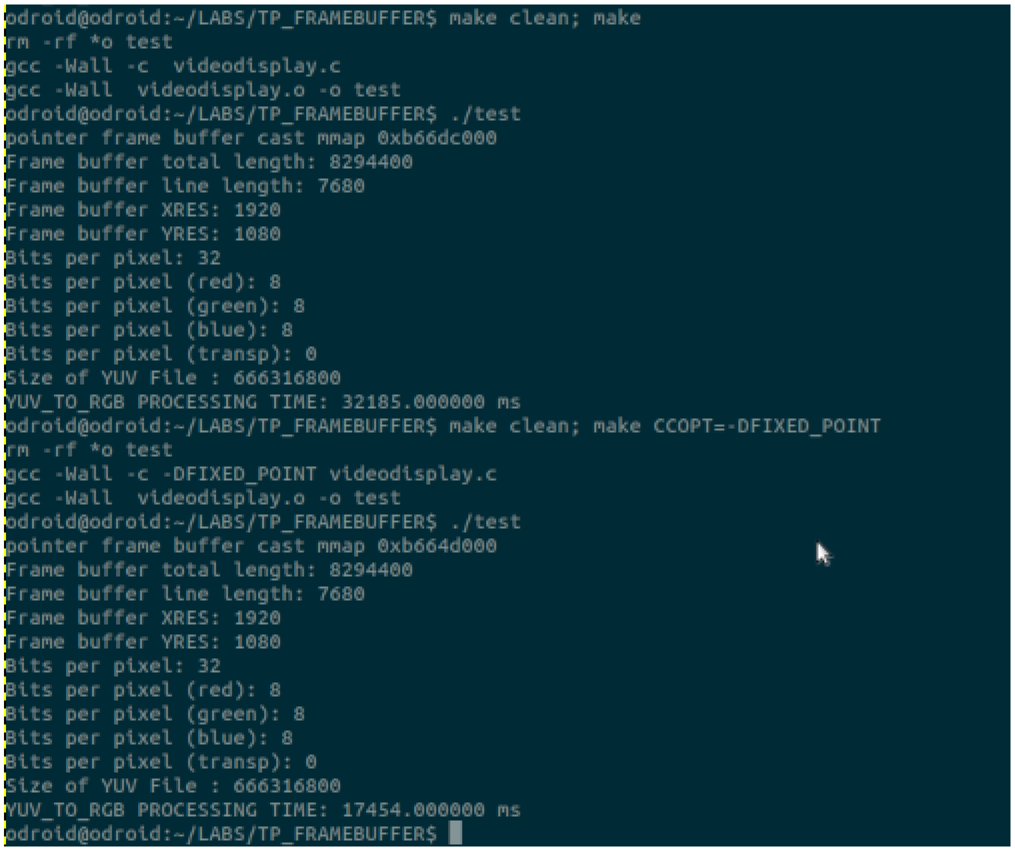
The code is updated as follow to easily compare performance between each mode. The mode switch appends in the compile options where we can add the -DFIXED_POINT option to run in fixed point mode.



```c
void YUVtoRGB(unsigned char *Y, unsigned char *U, unsigned char *V) {
    int inc_Y=0, inc_UV=0, pos_fb=0;
#ifndef FIXED_POINT
    float Yval, Uval, Vval;
    float R,G,B, Transp;
#else
    int Yval, Uval, Vval;
    int R,G,B, Transp;
#endif
    unsigned int line_number = 0;
    Transp = 255;
    for (inc_Y=0; inc_Y<VIDEO_WIDTH*VIDEO_HEIGHT; inc_Y++) {
#ifndef FIXED_POINT
        Yval = (float)(*(Y+inc_Y));
        Uval = (float)(*(U+inc_UV));
        Vval = (float)(*(V+inc_UV));
        R = ((Yval - 16) + 1.13983 * (Vval - 128));
        G = ((Yval - 16) - 0.39465  * (Uval - 128) - 0.58060*(Vval - 128));
        B = ((Yval - 16) + 2.03211 * (Uval - 128));
#else
        Yval = (int)(*(Y+inc_Y));
        Uval = (int)(*(U+inc_UV));
        Vval = (int)(*(V+inc_UV));
        R = (((Yval - 16)<<14) + 18675 *(Vval - 128))>>14;
        G = (((Yval - 16)<<14) - 6466  *(Uval - 128) - 9513*(Vval - 128) )>>14;
        B = (((Yval - 16)<<14) + 33294 *(Uval - 128))>>14;
#endif
        R = ( (R>255)? 255: ((R<0)? 0: R));
        G = ( (G>255)? 255: ((G<0)? 0: G));
        B = ( (B>255)? 255: ((B<0)? 0: B));

        fb_ptr[pos_fb] =
            ((((unsigned int)(B)<<0)         &   0x000000FF) |
            (((unsigned int)(G)<<8)          &   0x0000FF00) |
            (((unsigned int)(R)<<16)         &   0x00FF0000) |
            (((unsigned int)(Transp)<<24)    &   0xFF000000));

        // Gestion nouvelle ligne
        if (inc_Y % VIDEO_WIDTH == VIDEO_WIDTH - 1) {
            if ((line_number & 1) == 0) {
                // Une ligne sur deux, on reset inc_UV au debut de la ligne
                inc_UV -= VIDEO_WIDTH/2;
            }
                // Toutes les deux lignes, on continue d'incrementer inc_uv
            inc_UV++;
            pos_fb += (frame_buffer_line - VIDEO_WIDTH + 1);
            line_number++;
        }
        else {
            pos_fb++;
            if ((inc_Y & 1) == 1) inc_UV++; // Incrementer inc_UV une fois sur deux
        }
    }

}
```

Figure 8- updated code for performance analysis

The fixed point method is 2 times quicker than the floating point method (~3sec for floating point and ~17sec for fixed point).2



```
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ make clean; make
rm -rf *o test
gcc -Wall -c  videodisplay.c
gcc -Wall  videodisplay.o -o test
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ ./test
pointer frame buffer cast mmap 0xb66dc000
Frame buffer total length: 8294400
Frame buffer line length: 7680
Frame buffer XRES: 1920
Frame buffer YRES: 1080
Bits per pixel: 32
Bits per pixel (red): 8
Bits per pixel (green): 8
Bits per pixel (blue): 8
Bits per pixel (transp): 0
Size of YUV File : 666316800
YUV_TO_RGB PROCESSING TIME: 32185.000000 ms
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ make clean; make CCOPT=-DFIXED_POINT
rm -rf *o test
gcc -Wall -c -DFIXED_POINT videodisplay.c
gcc -Wall  videodisplay.o -o test
odroid@odroid:~/LABS/TP_FRAMEBUFFER$ ./test
pointer frame buffer cast mmap 0xb664d000
Frame buffer total length: 8294400
Frame buffer line length: 7680
Frame buffer XRES: 1920
Frame buffer YRES: 1080
Bits per pixel: 32
Bits per pixel (red): 8
Bits per pixel (green): 8
Bits per pixel (blue): 8
Bits per pixel (transp): 0
Size of YUV File : 666316800
YUV_TO_RGB PROCESSING TIME: 17454.000000 ms
odroid@odroid:~/LABS/TP_FRAMEBUFFER$
```

Figure 9- fixed point and floating point performance comparison

The resulting video is still correctly displayed in the console mode:

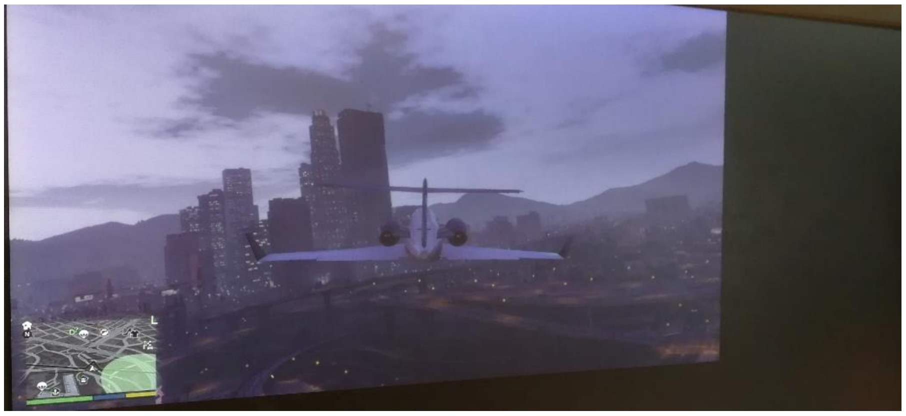


Figure 10- YUV video extract

4 CONCLUSION

Interacting with Linux framebuffer is fairly easy even when we're in the userspace. But as it's an existing Driver, some processes are already using it when in GUI mode; To see our application working we first need to stop the GUI from refreshing the screen.

When developing applications using the framebuffer, we need to mind the resources used as it can heavily impact the application performance.