Pseudo-random ramblings about programming and other geeky stuff

Sunday, 16 March 2014

Low-level Graphics on Raspberry Pi [part X+1]

NOTE 2015-01-27: On later Raspbian builds this approach may lead to a lock up! It is adviced to use the 'pure fb' approach - code available in this post.

In previous part we tried a simple animation - with not so perfect results...

Obviously with so many different avenues to tackle for the Raspberry Pi driver developers it is not possible for us users to get everything 'for free' and at instant. I sure wish I had the time and/or the drive to attempt extending the fb driver myself ...or maybe the legendary 'Somebody Else' could do it ;) As the thing is, it does not seem that big of a job: there is already support for the 'page flipping' using double size buffer and panning the display in the Raspberry Pi firmware Mailbox interface.

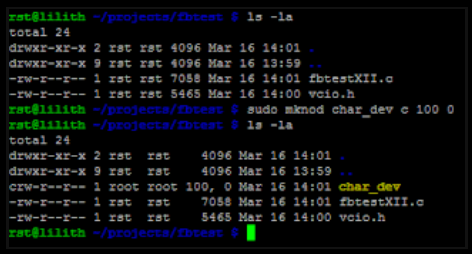
In the meantime, we could take a stab at trying this out. Raspberry Pi forum users hacking on the 'bare metal' (thanks guys) pointed me before to look at the `arch/arm/mach-bcm2708/include/mach/vcio.h` in the RPi firmware GitHub sources and to the way to talk to the mailbox:

```
...
#include "vcio.h"
...
int mboxfd = 0;
...

// helper function to talk to the mailbox interface
static int mbox_property(void *buf)
{
   if (mboxfd < -1) return -1;

   int ret_val = ioctl(mboxfd, IOCTL_MBOX_PROPERTY, buf);

   if (ret_val < 0) {
      printf("ioctl_set_msg failed:%d\n", ret_val);
   }

   return ret_val;
}

// helper function to set the framebuffer virtual offset == pan
static unsigned set_fb_voffs(unsigned *x, unsigned *y)
{
   int i=0;
   unsigned p[32];
   p[i++] = 0; // size
   p[i++] = 0x00000000; // process request

   p[i++] = 0x00048009; // get physical (display) width/height
   p[i++] = 0x00000008; // buffer size
```

```
    p[i++] = 0x00000000; // request size
    p[i++] = *x; // value buffer
    p[i++] = *y; // value buffer 2

    p[i++] = 0x00000000; // end tag
    p[0] = i*sizeof *p; // actual size

    mbox_property(p);
    *x = p[5];
    *y = p[6];
    return p[1];
}
...
void draw() {
...

        // switch page
        /*
        vinfo.yoffset = cur_page * vinfo.yres;
        vinfo.activate = FB_ACTIVATE_VBL;
        if (ioctl(fbfd, FBIOPAN_DISPLAY, &vinfo)) {
            printf("Error panning display.\n");
        }
        */
        vx = 0;
        vy = cur_page * vinfo.yres;
        set_fb_voffs(&vx, &vy);

        //usleep(1000000 / fps);

}

...
// main

    // open a char device file used for communicating with kernel mbox driver
    mboxfd = open(DEVICE_FILE_NAME, 0);
    if (mboxfd < 0) {
        printf("Can't open device file: %s\n", DEVICE_FILE_NAME);
        printf("Try creating a device file with: mknod %s c %d 0\n", DEVICE_FILE_NAME, MAJOR_NUM);

    }
...
```

From the `"vcio.h"` we are using the two defines: `DEVICE_FILE_NAME` `"char_dev"` and . This `char_dev` is a special file for communicating with the mailbox. The file must be created using the command `mknod` (see man):

```
$ sudo mknod char_dev c 100 0
```



Save the code as say `fbtestXII.c` (full source in GitHub), download the vcio.h file to the same directory, build with:

```
gcc -lrt -o fbtestXII fbtestXII.c
```

(as the code uses the clock functions from librt) and run with `./fbtestXII`. This should display the same gliding and bouncing rectangle, but this time with no tearing and with minimal flicker.

The program outputs the timing info - the (most likely) 16 seconds (and some 700 ms) comes from the `fps = 100` and `secs = 10` ...it is quite obvious that since the screen refresh is now tied to the vertical sync, we 'only' get 60 fps and 100 * 10 = 1000 loops takes 1000 / 60 = 16.6 s.

Now if we change the code a bit:

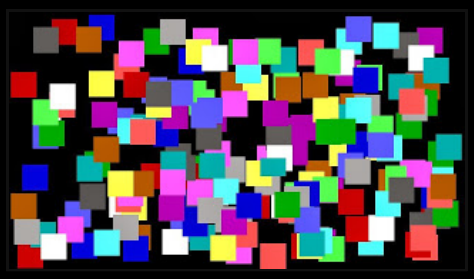
```
...
#define NUM_ELEMS 200
int xs[NUM_ELEMS];
int ys[NUM_ELEMS];
int dxs[NUM_ELEMS];
int dys[NUM_ELEMS];
...
void draw() {

    int i, x, y, w, h, dx, dy;
    struct timespec pt;
    struct timespec ct;
    struct timespec df;

    // rectangle dimensions
    w = vinfo.yres / 10;
    h = w;

    // start position (upper left)
    x = 0;
    y = 0;
    int n;
    for (n = 0; n < NUM_ELEMS; n++) {
        int ex = rand() % (vinfo.xres - w);
        int ey = rand() % (vinfo.yres - h);
        //printf("%d: %d,%d\n", n, ex, ey);
        xs[n] = ex;
        ys[n] = ey;
        int edx = (rand() % 10) + 1;
        int edy = (rand() % 10) + 1;
        dxs[n] = edx;
        dys[n] = edy;
    }

    // move step 'size'
    dx = 1;
    dy = 1;

    int fps = 60;
    int secs = 10;

    int vx, vy;

    clock_gettime(CLOCK_REALTIME, &pt);

    // loop for a while
    for (i = 0; i < (fps * secs); i++) {

        // change page to draw to (between 0 and 1)
        cur_page = (cur_page + 1) % 2;

        // clear the previous image (= fill entire screen)
        clear_screen(0);

        for (n = 0; n < NUM_ELEMS; n++) {
            x = xs[n];
            y = ys[n];
            dx = dxs[n];
            dy = dys[n];

            // draw the bouncing rectangle
            fill_rect(x, y, w, h, (n % 15) + 1);

            // move the rectangle
            x = x + dx;
            y = y + dy;

            // check for display sides
            if ((x < 0) || (x > (vinfo.xres - w))) {
                dx = -dx; // reverse direction
                x = x + 2 * dx; // counteract the move already done above
            }
            // same for vertical dir
```

```
            if ((y < 0) || (y > (vinfo.yres - h))) {
                dy = -dy;
                y = y + 2 * dy;
            }

            xs[n] = x;
            ys[n] = y;
            dxs[n] = dx;
            dys[n] = dy;
        }

        // switch page
        vx = 0;
        vy = cur_page * vinfo.yres;
        set_fb_voffs(&vx, &vy);

    }

    clock_gettime(CLOCK_REALTIME, &ct);
    df = timediff(pt, ct);
    printf("done in %ld s %5ld ms\n", df.tv_sec, df.tv_nsec / 1000000);
}
...
```

(full source)...and build with (optimisation on):

```
gcc -O2 -lrt -o fbtestXIII fbtestXIII.c
```

...we should get 200 colorful, bouncing 'sprites' going all over the screen:



Using the "char_dev" (especially as it has to be created as root) is not the most elegant way, but so far the only solution I know (if we want to stick to the fb) and at least for some uses this may be quite enough.

[Continued in part X+2>

Posted by Unknown at 15:37

Labels: C, graphics, Linux, Raspberry Pi

No comments:

Post a Comment

Note: only a member of this blog may post a comment.

Subscribe to: Post Comments (Atom)