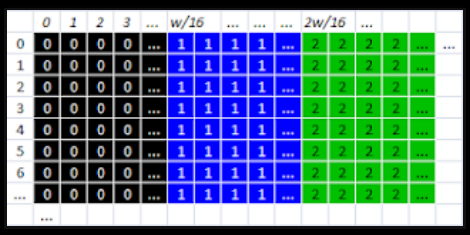
Raspberry Compote

Pseudo-random ramblings about programming and other geeky stuff

Friday, 8 March 2013

Low-level Graphics on Raspberry Pi [part six]

In the previous posts we have been plotting pixels using a 8 bit, 256 color, palette display mode. In this mode, every byte of the framebuffer (mmap'ed) memory block present one byte and the value of the byte is an index to the palette (see part three). So to get the color bars in the previous examples, we have plotted values of 0 (zero) to the first bar (black), values of 1 (one) to the second bar (blue) and so on... This picture illustrates the idea - each cell presents one pixel (some columns skipped for compacting), showing both the byte value and the resulting color:



Here is the pixel plotting function used:

```
void put_pixel(int x, int y, int c)
{
    // calculate the pixel's byte offset inside the buffer
    unsigned int pix_offset = x + y * finfo.line_length;

    // now this is about the same as 'fbp[pix_offset] = value'
    *((char*)(fbp + pix_offset)) = c;

}
```

So we are storing one byte (that `'char *'` there) ...obviously we could (/should) have defined the color parameter `c` as a `char` too, but... (the above code takes the lowest byte of the four byte integer variable, so works as is).

Well, how about other display modes? We have noticed already (part one) that the default mode on RPi is 16 bit and quite often one comes across mentions of 24 bit and 32 bit modes. I suppose the easiest of these to begin with would be the 24 bit mode: 3 bytes per pixel - one byte per each RGB (red, green, blue) value. The RGB values are very similar to the values in the palette for the 8 bit mode. To illustrate this, in the following image we have the two leftmost pixels of the two first color bars - the first pixel occupies three first bytes of the memory buffer. For the black pixels all three byte values are zeroes - for the blue pixels the 'R' and 'G' bytes are zero and 'B' byte is 255 (= full blue):

Blog Archive

- ► 2016 (6)
- ► 2015 (3)
- ► 2014 (9)
- ▼ 2013 (9)
  - ► April (2)
  - ▼ March (4)
    - Low-level Graphics on Raspberry Pi (part six)
    - Low-level Graphics on Raspberry Pi (part five)
    - Coding Gold Dust: How to break out from an infinit...
    - Low-level Graphics on Raspberry Pi (part four)
  - ► February (1)
  - ► January (2)
- ► 2012 (2)

Code Repository

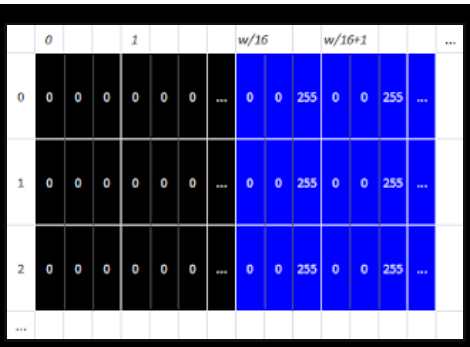
- Low-level Graphics on RPi

Discussion

- Low-level Graphics on RPi
- Python Programming on RPi
- Java Programming on RPi

Links

- Raspberry Pi
- Python

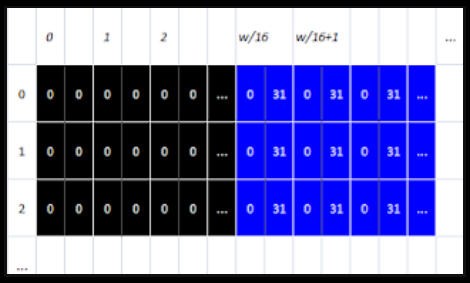


This could be implemented as the following pixel plotting code:

```
void put_pixel_RGB24(int x, int y, int r, int g, int b)
{

    // calculate the pixel's byte offset inside the buffer
    // note: x * 3 as every pixel is 3 consecutive bytes
    unsigned int pix_offset = x * 3 + y * finfo.line_length;

    // now this is about the same as 'fbp[pix_offset] = value'
    *((char*)(fbp + pix_offset)) = r;
    *((char*)(fbp + pix_offset + 1)) = g;
    *((char*)(fbp + pix_offset + 2)) = b;

}
```

The RPi default 16 bit RGB565 is slightly more complex - there are 2 bytes per pixel and the color components are encoded so that 5 first bits are for the red, 6 middle bits for green and 5 last bits for blue:

In the format similar to the above ones, the memory buffer would look something like this:



...the blue value `31` comes from the fact that there are `5` bits for blue and the binary value of `0b11111` is `31` in decimal. Full red would be `0b1111100000000000` (63488) so the bytes for full red would be `248` and `0` - full green would be `0b0000011111100000` (2016) so the bytes `7` and `224`.

The RGB565 pixe plotting function would be something along this:

```
void put_pixel_RGB565(int x, int y, int r, int g, int b)
{

    // calculate the pixel's byte offset inside the buffer
    // note: x * 2 as every pixel is 2 consecutive bytes
    unsigned int pix_offset = x * 2 + y * finfo.line_length;

    // now this is about the same as 'fbp[pix_offset] = value'
    // but a bit more complicated for RGB565
    unsigned short c = ((r / 8) << 11) + ((g / 4) << 5) + (b / 8);
    // or: c = ((r / 8) * 2048) + ((g / 4) * 32) + (b / 8);
    // write 'two bytes at once'
    *((unsigned short*)(fbp + pix_offset)) = c;
```

```
}
```

The red value has 5 bits, so can be in the range 0-31, therefore divide the original 0-255 value by 8. It is stored in the first 5 bits, so multiply by 2048 or shift 11 bits left. The green has 6 bits, so can be in the range 0-63, divide by 4, and multiply by 32 or shift 5 bits left. Finally the blue has 5 bits and is stored at the last bits, so no need to move.

The 32 bit mode is usually so called ARGB - where there are 4 bytes per pixel, A standing for 'alpha' = transparency and the rest just like in the RGB24. Modifying the `put_pixel_RGB24()` to `put_pixel_ARGB32()` should be trivial.

Note that there are several other display (and especially image and video) modes as well, but the ones covered here are the most standard ones for Linux framebuffers.

To test the above pixel plotting functions, we could try the following (continuing from the code in part five):

```
void draw() {

    int x, y;

    for (y = 0; y < (vinfo.yres / 2); y++) {
        for (x = 0; x < vinfo.xres; x++) {

            // color based on the 16th of the screen width
            int c = 16 * x / vinfo.xres;

            if (vinfo.bits_per_pixel == 8) {
                put_pixel(x, y, c);
            }
            else if (vinfo.bits_per_pixel == 16) {
                put_pixel_RGB565(x, y, def_r[c], def_g[c], def_b[c]);
            }
            else if (vinfo.bits_per_pixel == 24) {
                put_pixel_RGB24(x, y,  def_r[c], def_g[c], def_b[c]);
            }

        }
    }

}

...

// in main()

// comment out setting the bit depth

    // Change variable info
    /* use: 'fbset -depth x' to test different bpps
    vinfo.bits_per_pixel = 8;
    if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo)) {
      printf("Error setting variable information.\n");
    }
    */

// also can comment out setting the palette...

// change the calculation of required memory
    // map fb to user mem
    screensize = vinfo.xres * vinfo.yres * vinfo.bits_per_pixel / 8;

...
```

Save the new code to `fbtest6.c`, compile and then execute the following sequence:

```
fbset -depth 8
./fbtest6
fbset -depth 24
./fbtest6
fbset -depth 16
./fbtest6
```

...this should yield three times the exact same color bars. Full source code available in github.

[Continued in part seven]

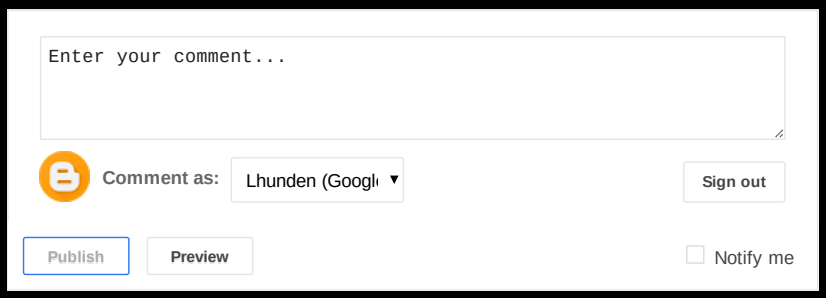
Posted by Unknown at 12:57

Labels: C, graphics, Linux, Raspberry Pi

No comments:

Post a Comment

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



Newer Post Home Older Post

Subscribe to: Post Comments (Atom)