Pseudo-random ramblings about programming and other geeky stuff

Tuesday, 22 January 2013

Low-level Graphics on Raspberry Pi (part three)

So, after memmap'ing the framebuffer (see part two), it appears as a contiguous section of RAM - an array of bytes. That is why we declared the 'frame buffer pointer' variable **fbp** as:

```
char *fbp = 0;
```

and the memmap:

```
fbp = (char*)mmap(0,
                  screensize,
                  PROT_READ | PROT_WRITE,
                  MAP_SHARED,
                  fbfd, 0);
```

behaves about the same as if we allocated the buffer ourselves like:

```
#include <stdio.h>

int main(int argc, char* argv[])
{
    char *fbp = 0;
    char fb[100];

    fb[0] = 'a';

    fbp = fb; // similar to mmap ...

    printf("fb[0] = %c\n", fb[0]);
    printf("*fbp = %c\n", *fbp);
}
```

where the pointer variable **fbp** points to the first byte of the array **fb**. You might want to read more about C data-types and arrays vs pointers from your preferred source of C programming information...

Basically all RAM memory can be seen as an array of bytes - something like this:

0	1	2	3	4	5	6	7	8	9	...	n

The same could also be used to illustrate a particular array within the memory, where the array start byte would be the 'box zero' (fb[0] in the above example) and so on until the length of the allocated array at 'box n' (fb[99]).

As the framebuffer represents the two-dimensional display 'surface', we could illustrate the pixels as:

Blog Archive

- ► 2016 (6)
- ► 2015 (3)
- ► 2014 (9)
- ▼ 2013 (9)
  - ► April (2)
  - ► March (4)
  - ► February (1)
  - ▼ January (2)
    - Low-level Graphics on Raspberry Pi (part three)
    - Low-level Graphics on Raspberry Pi (part two)
- ► 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

0	1	2	3	4	5	6	7	8	9	...	w-1
w	w+1	w+2	w+3	...							
2w											
3w					3w+5						
4w											
...											
(h-1)w											

In case of a 8 bpp ('8 bits per pixel', 256 color, one byte per pixel) display mode, this would directly map to the framebuffer memory layout: so the "width'th" byte (fb[width]) in the array would represent the left-most pixel on the second (from the top) scanline and the 5th pixel on the 3rd row (highlighted in grey) would match the byte at '3 x width + 5' (fb[3 * width + 5]).

However, there are numerous other display modes generally used and in fact the default mode of RPi is 16 bpp (as seen in part one). This means that to be able to handle the framebuffer as 'just bytes', we need to switch to 8 bpp display mode.

This can be done from the command-line using the command:

```
fbset -depth 8
```

Now run the **fbtest** from the part one - it should output '..., 8 bpp'. If you try the fbtest2 from part two, you will most likely not get the same result as in 16 bpp mode - probably just a black screen (will have to verify this). To reset bit-depth back to original use '-depth 16' (or whatever fbtest said it was before).

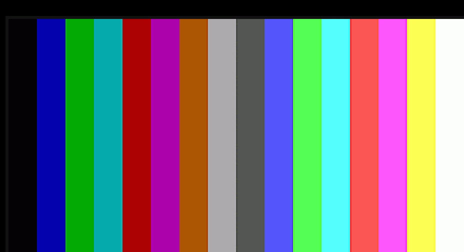
So what's with the drawing now? The reason to the different result is that in 16 bpp mode, the bytes (byte-pairs) in the buffer describe the actual pixel color - in 8 bpp mode, the byte value is an index to a palette. In 16 bpp a value of 0 (in both bytes) would yield black - in 8 bit the value 0 yields the color stored in the palette at index 0 and could basically be anything in the RGB colorspace (incidently, the color 0 in the default Linux framebuffer palette seems to be black ...but it could be set to anything).

Speaking of the default palette: why not take a look what colors it contains? This example (building on the previous examples) changes the display mode to 8 bpp, draws vertical bars of varying colors (the 16 colors of the default palette), pauses for 5 seconds and restores the display settings:

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <linux/fb.h>
#include <sys/mman.h>

// application entry point
int main(int argc, char* argv[])
{
  int fbfd = 0;
  struct fb_var_screeninfo orig_vinfo;
  struct fb_var_screeninfo vinfo;
  struct fb_fix_screeninfo finfo;
  long int screensize = 0;
  char *fbp = 0;


  // Open the file for reading and writing
  fbfd = open("/dev/fb0", O_RDWR);
  if (!fbfd) {
    printf("Error: cannot open framebuffer device.\n");
    return(1);
  }
  printf("The framebuffer device was opened successfully.\n");

  // Get variable screen information
  if (ioctl(fbfd, FBIOGET_VSCREENINFO, &vinfo)) {
    printf("Error reading variable information.\n");
  }
  printf("Original %dx%d, %dbpp\n", vinfo.xres, vinfo.yres,
```

```
        vinfo.bits_per_pixel );

  // Store for reset (copy vinfo to vinfo_orig)
  memcpy(&orig_vinfo, &vinfo, sizeof(struct fb_var_screeninfo));

  // Change variable info
  vinfo.bits_per_pixel = 8;
  if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &vinfo)) {
    printf("Error setting variable information.\n");
  }

  // Get fixed screen information
  if (ioctl(fbfd, FBIOGET_FSCREENINFO, &finfo)) {
    printf("Error reading fixed information.\n");
  }

  // map fb to user mem
  screensize = finfo.smem_len;
  fbp = (char*)mmap(0,
                    screensize,
                    PROT_READ | PROT_WRITE,
                    MAP_SHARED,
                    fbfd,
                    0);

  if ((int)fbp == -1) {
    printf("Failed to mmap.\n");
  }
  else {
    // draw...
    int x, y;
    unsigned int pix_offset;

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

        // calculate the pixel's byte offset inside the buffer
        // see the image above in the blog...
        pix_offset = x + y * finfo.line_length;

        // now this is about the same as fbp[pix_offset] = value
        *((char*)(fbp + pix_offset)) = 16 * x / vinfo.xres;

      }
    }

    sleep(5);
  }

  // cleanup
  munmap(fbp, screensize);
  if (ioctl(fbfd, FBIOPUT_VSCREENINFO, &orig_vinfo)) {
    printf("Error re-setting variable information.\n");
  }
  close(fbfd);

  return 0;

}
```

Save as fbtest3.c, 'make fbtest3' and run as './fbtest3' - this should produce vertical color bars at the upper half of the screen:



Hmm, so far there has been very little Raspberry Pi specific stuff in this series - all the code should work on most (if not all) Linux systems as is ...maybe I should have thought more about the title ;)

[Continues in part four]

Posted by Unknown at 16:44

Labels: C, graphics, Linux, Raspberry Pi

No comments:

Post a Comment

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

Enter your comment...

Comment as: Lhunden (Googl▾ Sign out

Publish Preview Notify me

Newer Post Home Older Post

Subscribe to: Post Comments (Atom)