android
open source project

Display Drivers

In this document

This section describes how the display driver functions and offers a functional template designed to help you build your own device-specific driver.

Android relies on the standard frame buffer device (`/dev/fb0` or `/dev/graphics/fb0`) and driver as described in the `linux/fb.h` kernel header file. For more information regarding the standard Linux frame buffer, please see The Frame Buffer Device at http://kernel.org.

Functionality

In Android, every window gets implemented with an underlying Surface object, an object that gets placed on the framebuffer by SurfaceFlinger, the system-wide screen composer. Each Surface is double-buffered. The back buffer is where drawing takes place and the front buffer is used for composition.

When `unlockCanvas()` is called, the back buffer is posted, which means that it gets displayed and becomes available again. Android flips the front and back buffers, ensuring a minimal amount of buffer copying and that there is always a buffer for SurfaceFlinger to use for composition (which ensures that the screen never flickers or shows artifacts).

Android makes two requirements of the driver: a linear address space of mappable memory that it can write to directly and support for the rgb_565 pixel format. A typical frame display includes:

- accessing the driver by calling open on `/dev/fb0`
- using the `FBIOGET_FSCREENINFO` and `FBIOGET_VSCREENINFO` Input / Output Control (ioctl) calls to retrieve information about the screen
- using `FBIOPUT_VSCREENINFO` ioctl to attempt to create a virtual display twice the size of the physical screen and to set the pixel format to rgb_565. If this succeeds, double buffering is accomplished with video memory.

When a page flip is required, Android makes another `FBIOPUT_VSCREENINFO` ioctl call with a new y-offset pointing to the other buffer in video memory. This ioctl, in turn, invokes the driver's `.fb_pan_display` function in order to do the actual flip. If there isn't sufficient video memory, regular memory is used and is just copied into the video memory when it is time do the flip. After allocating the video memory and setting the pixel format, Android uses `mmap()` to map the memory into the process's address space. All writes to the frame buffer are done through this mmaped memory.

To maintain adequate performance, framebuffer memory should be cacheable. If you use write-back, flush the cache before the frame buffer is written from DMA to the LCD. If that isn't possible, you may use write-through. As a last resort, you can also use uncached memory with the write-bugger enabled, but performance will suffer.

Implementing Your Own Driver (Driver Template)

The following sample driver offers a functional example to help you build your own display driver. Modify `PGUIDE_FB...` macros as desired to match the requirements of your own device hardware.

```
/*
 *  pguidefb.c
 *
 *  Copyright 2007, Google Inc.
 *
```

```
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License version 2 as
 *  published by the Free Software Foundation.
 */


/*
 * ANDROID PORTING GUIDE: FRAME BUFFER DRIVER TEMPLATE
 *
 * This template is designed to provide the minimum frame buffer
 * functionality necessary for Android to display properly on a new
 * device.  The PGUIDE_FB macros are meant as pointers indicating
 * where to implement the hardware specific code necessary for the new
 * device.  The existence of the macros is not meant to trivialize the
 * work required, just as an indication of where the work needs to be
 * done.
 */

#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/errno.h>
#include <linux/string.h>
#include <linux/slab.h>
#include <linux/delay.h>
#include <linux/mm.h>
#include <linux/fb.h>
#include <linux/init.h>
#include <linux/platform_device.h>


/* Android currently only uses rgb565 in the hardware framebuffer */
#define ANDROID_BYTES_PER_PIXEL 2

/* Android will use double buffer in video if there is enough */
#define ANDROID_NUMBER_OF_BUFFERS 2

/* Modify these macros to suit the hardware */

#define PGUIDE_FB_ROTATE
        /* Do what is necessary to cause the rotation */

#define PGUIDE_FB_PAN
        /* Do what is necessary to cause the panning */

#define PGUIDE_FB_PROBE_FIRST
        /* Do any early hardware initialization */

#define PGUIDE_FB_PROBE_SECOND
        /* Do any later hardware initialization */

#define PGUIDE_FB_WIDTH 320
        /* Return the width of the screen */

#define PGUIDE_FB_HEIGHT 240
        /* Return the heighth of the screen */

#define PGUIDE_FB_SCREEN_BASE 0
        /* Return the virtual address of the start of fb memory */

#define PGUIDE_FB_SMEM_START PGUIDE_FB_SCREEN_BASE
        /* Return the physical address of the start of fb memory */

#define PGUIDE_FB_REMOVE
        /* Do any hardware shutdown */




struct pguide_fb {
        int rotation;
        struct fb_info fb;
        u32                     cmap[16];
```

```
};

static inline u32 convert_bitfield(int val, struct fb_bitfield *bf)
{
        unsigned int mask = (1 << bf->length) - 1;

        return (val >> (16 - bf->length) & mask) << bf->offset;
}


/* set the software color map.  Probably doesn't need modifying. */
static int
pguide_fb_setcolreg(unsigned int regno, unsigned int red, unsigned int green,
                 unsigned int blue, unsigned int transp, struct fb_info *info)
{
        struct pguide_fb  *fb = container_of(info, struct pguide_fb, fb);

        if (regno < 16) {
                fb->cmap[regno] = convert_bitfield(transp, &fb->fb.var.transp) |
                                  convert_bitfield(blue, &fb->fb.var.blue) |
                                  convert_bitfield(green, &fb->fb.var.green) |
                                  convert_bitfield(red, &fb->fb.var.red);
                return 0;
        }
        else {
                return 1;
        }
}

/* check var to see if supported by this device.  Probably doesn't
 * need modifying.
 */
static int pguide_fb_check_var(struct fb_var_screeninfo *var, struct fb_info *info)
{
        if((var->rotate & 1) != (info->var.rotate & 1)) {
                if((var->xres != info->var.yres) ||
                   (var->yres != info->var.xres) ||
                   (var->xres_virtual != info->var.yres) ||
                   (var->yres_virtual >
                    info->var.xres * ANDROID_NUMBER_OF_BUFFERS) ||
                   (var->yres_virtual < info->var.xres )) {
                        return -EINVAL;
                }
        }
        else {
                if((var->xres != info->var.xres) ||
                   (var->yres != info->var.yres) ||
                   (var->xres_virtual != info->var.xres) ||
                   (var->yres_virtual >
                    info->var.yres * ANDROID_NUMBER_OF_BUFFERS) ||
                   (var->yres_virtual < info->var.yres )) {
                        return -EINVAL;
                }
        }
        if((var->xoffset != info->var.xoffset) ||
           (var->bits_per_pixel != info->var.bits_per_pixel) ||
           (var->grayscale != info->var.grayscale)) {
                return -EINVAL;
        }
        return 0;
}


/* Handles screen rotation if device supports it. */
static int pguide_fb_set_par(struct fb_info *info)
{
        struct pguide_fb *fb = container_of(info, struct pguide_fb, fb);
        if(fb->rotation != fb->fb.var.rotate) {
                info->fix.line_length =
                  info->var.xres * ANDROID_BYTES_PER_PIXEL;
                fb->rotation = fb->fb.var.rotate;
                PGUIDE_FB_ROTATE;
        }
```

```
        return 0;
}


/* Pan the display if device supports it. */
static int pguide_fb_pan_display(struct fb_var_screeninfo *var, struct fb_info *info)
{
        struct pguide_fb *fb    __attribute__ ((unused))
            = container_of(info, struct pguide_fb, fb);

        /* Set the frame buffer base to something like:
           fb->fb.fix.smem_start + fb->fb.var.xres *
           ANDROID_BYTES_PER_PIXEL * var->yoffset
        */
        PGUIDE_FB_PAN;

        return 0;
}


static struct fb_ops pguide_fb_ops = {
        .owner          = THIS_MODULE,
        .fb_check_var   = pguide_fb_check_var,
        .fb_set_par     = pguide_fb_set_par,
        .fb_setcolreg   = pguide_fb_setcolreg,
        .fb_pan_display = pguide_fb_pan_display,

        /* These are generic software based fb functions */
        .fb_fillrect    = cfb_fillrect,
        .fb_copyarea    = cfb_copyarea,
        .fb_imageblit   = cfb_imageblit,
};


static int pguide_fb_probe(struct platform_device *pdev)
{
        int ret;
        struct pguide_fb *fb;
        size_t framesize;
        uint32_t width, height;

        fb = kzalloc(sizeof(*fb), GFP_KERNEL);
        if(fb == NULL) {
                ret = -ENOMEM;
                goto err_fb_alloc_failed;
        }
        platform_set_drvdata(pdev, fb);

        PGUIDE_FB_PROBE_FIRST;
        width = PGUIDE_FB_WIDTH;
        height = PGUIDE_FB_HEIGHT;


        fb->fb.fbops            = &pguide_fb_ops;

        /* These modes are the ones currently required by Android */

        fb->fb.flags            = FBINFO_FLAG_DEFAULT;
        fb->fb.pseudo_palette   = fb->cmap;
        fb->fb.fix.type         = FB_TYPE_PACKED_PIXELS;
        fb->fb.fix.visual = FB_VISUAL_TRUECOLOR;
        fb->fb.fix.line_length = width * ANDROID_BYTES_PER_PIXEL;
        fb->fb.fix.accel        = FB_ACCEL_NONE;
        fb->fb.fix.ypanstep = 1;

        fb->fb.var.xres         = width;
        fb->fb.var.yres         = height;
        fb->fb.var.xres_virtual = width;
        fb->fb.var.yres_virtual = height * ANDROID_NUMBER_OF_BUFFERS;
        fb->fb.var.bits_per_pixel = 16;
        fb->fb.var.activate     = FB_ACTIVATE_NOW;
        fb->fb.var.height       = height;
        fb->fb.var.width        = width;
```

```
        fb->fb.var.red.offset = 11;
        fb->fb.var.red.length = 5;
        fb->fb.var.green.offset = 5;
        fb->fb.var.green.length = 6;
        fb->fb.var.blue.offset = 0;
        fb->fb.var.blue.length = 5;

        framesize = width * height *
          ANDROID_BYTES_PER_PIXEL * ANDROID_NUMBER_OF_BUFFERS;
        fb->fb.screen_base = PGUIDE_FB_SCREEN_BASE;
        fb->fb.fix.smem_start = PGUIDE_FB_SMEM_START;
        fb->fb.fix.smem_len = framesize;

        ret = fb_set_var(&fb->fb, &fb->fb.var);
        if(ret)
                goto err_fb_set_var_failed;

        PGUIDE_FB_PROBE_SECOND;

        ret = register_framebuffer(&fb->fb);
        if(ret)
                goto err_register_framebuffer_failed;

        return 0;


err_register_framebuffer_failed:
err_fb_set_var_failed:
        kfree(fb);
err_fb_alloc_failed:
        return ret;
}

static int pguide_fb_remove(struct platform_device *pdev)
{
        struct pguide_fb *fb = platform_get_drvdata(pdev);

        PGUIDE_FB_REMOVE;

        kfree(fb);
        return 0;
}


static struct platform_driver pguide_fb_driver = {
        .probe          = pguide_fb_probe,
        .remove         = pguide_fb_remove,
        .driver = {
                .name = "pguide_fb"
        }
};

static int __init pguide_fb_init(void)
{
        return platform_driver_register(&pguide_fb_driver);
}

static void __exit pguide_fb_exit(void)
{
        platform_driver_unregister(&pguide_fb_driver);
}

module_init(pguide_fb_init);
module_exit(pguide_fb_exit);

MODULE_LICENSE("GPL");
```

Troubleshooting

Both of the following problems have a similar cause:

- **Number keys**: In the dialer application, when a number key is pressed to dial a phone number, the number doesn't display on the screen until after the next number has been pressed.
- **Arrow keys**: When an arrow key is pressed, the desired icon doesn't get highlighted. For example, if you browse through icons in the Applications menu, you might notice that icons aren't highlighted as expected when you use the arrow key to navigate between options.

Both problems are caused by an incorrect implementation of the frame buffer's page flipping. Key events are captured, but the graphical interface appears to drop every other frame.

Android relies on a double buffer to smoothly render page flips (please see Functionality for details).

↑ Go to top