Frame buffers and input events

Chris Simmonds

Embedded Linux Conference Europe 2011

Overview

- The Linux frame buffer
- The Linux input event layer

Linux frame buffer

- Provides functions to get and set display characteristics
  - mode
  - resolution
  - colour map
- And a method to access the raw frame buffer
- But, no methods for hardware acceleration

Configuring frame buffer

- Enable frame buffer support in kernel
  - You need to select CONFIG_FB and the driver for your hardware
  - Part of kernel or loaded as a module
- Create device node
  - mknod /dev/fb0 c 29 0
  - can be up 32 fb
  - minor nos 0..31
  - names fb0..fb31
- Test with fbset -i

```
fbset -i

mode "800x480-51"
    # D: 21.900 MHz, H: 24.746 kHz, V: 50.917 Hz
    geometry 800 480 800 600 16
    timings 45662 20 15 1 3 50 2
    rgba 5/11,6/5,5/0,0/0
endmode

Frame buffer device information:
    Name        : omapfb
    Address     : 0x13d00000
    Size        : 1048576
    Type        : PACKED PIXELS
    Visual      : TRUECOLOR
    XPanStep    : 0
    YPanStep    : 0
    YWrapStep   : 0
    LineLength  : 1600
    Accelerator : Unknown (49)
```

Programming the frame buffer

- The frame buffer has three types of interface
- A normal character device
  - You can open it and read or write video memory
  - E.g. To grab a screen shot, `cat /dev/fb0 > screen.dump`
  - see also `fbgrab` later on
- A set of ioctls
  - FB-specific API to get and set configuration
- A memory device
  - using the mmap function to map video memory directly into an application's address space

List of FB ioctls

```
#include <linux/fb.h>
```

`FBIOGET_VSCREENINFO`	(M) Get variable screen information
`FBIOPUT_VSCREENINFO`	(M) Set variable screen information
`FBIOGET_FSCREENINFO`	(M) Get fixed screen information
`FBIOGETCMAP`	Get colour palette entry
`FBIOPUTCMAP`	Set colour palette entry
`FBIOPAN_DISPLAY`	Pan actual display within a (larger) virtual display
`FBIOGET_CON2FBMAP`	Get mapping of console to frame buffer
`FBIOPUT_CON2FBMAP`	Set mapping of console to frame buffer
`FBIOBLANK`	Blank or un-blank the display
`FBIOGET_VBLANK`	Get current raster position (line, pixel, blanking)
`FBIO_ALLOC`	Allocate video memory, e.g. for dual buffering
`FBIO_FREE`	Free memory allocated with FBIO_ALLOC

(M) = mandatory. All others are optional.

Fixed screen information

- “Fixed information” = things you cannot change

```
#include <linux/fb.h>
ioctl (int fd, FBIOGET_FSCREENINFO, struct fb_fix_screeninfo* fb_fs);
```

Amongst other things `struct fb_fix_screeninfo` contains:

```
char id[16];        /* identification string eg "TT Builtin"    */
unsigned long smem_start;    /* Start of frame buffer mem (physical address)  */
__u32 smem_len;          /* Length of frame buffer mem          */
__u32 type;         /* see FB_TYPE_*                  */
__u32 type_aux;          /* Interleave for interleaved Planes        */
__u32 visual;       /* see FB_VISUAL_*                */
```

Variable screen information

- "Variable information" = things you can change
  - read using FBIOGET_FSCREENINFO
  - write using FBIOSET_FSCREENINFO

```
#include <linux/fb.h>
ioctl (int fd, FBIOGET_VSCREENINFO, struct fb_var_screeninfo* fb_vs);
```

Amongst other things `struct fb_var_screeninfo` contains:

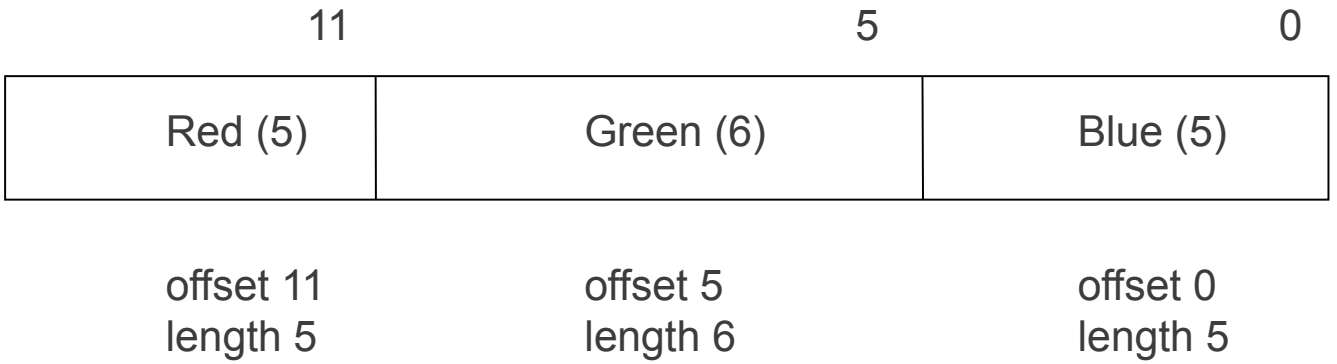
```
__u32 xres;         /* visible resolution            */
__u32 yres;
__u32 xres_virtual;     /* virtual resolution            */
__u32 yres_virtual;
__u32 xoffset;      /* offset from virtual to visible resolution     */
__u32 yoffset;
__u32 bits_per_pixel;   /* bits per pixel                */
__u32 grayscale;        /* != 0 Graylevels instead of colors         */
struct fb_bitfield red;  /* bitfield in fb mem if true color,        */
struct fb_bitfield green;    /* else only length is significant          */
struct fb_bitfield blue;
struct fb_bitfield transp;   /* transparency                  */
```

fb_bitfield

- Colour depth (bits per pixel) are defined as a bitfield

```
struct fb_bitfield {
     __u32 offset;  /* beginning of bitfield */
     __u32 length;  /* length of bitfield        */
     __u32 msb_right;    /* != 0 : Most significant bit is  right */
};
```

For example, RGB565 format looks like this:



Direct access of video memory

- Direct access is important for reasonable performance
- The POSIX mmap function does what is required

```
#include <sys/mman.h>
void *mmap (
    void *start,      usually NULL
    size_t length,    size of area to map in
    int prot,         PROT_READ | PROT_WRITE for read and write
    int flags,        MAP_SHARED
    int fd,           open file handle of frame buffer
    off_t offset      offset from the start of video memory, usually 0
);
```

Returns a pointer to the video memory or NULL mapping could not be done.
Reverse mapping with munmap ()

Frame buffer application (1)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <linux/fb.h>

int main(int argc, const char *argv[])
{
     int fd;
     struct fb_fix_screeninfo fb_fixinfo;
     struct fb_var_screeninfo fb_varinfo;

     fd = open ("/dev/fb0", O_RDWR);
     ioctl (fd, FBIOGET_FSCREENINFO, &fb_fixinfo);
     printf ("FB driver %s\n", fb_fixinfo.id);

     ioctl (fd, FBIOGET_VSCREENINFO, &fb_varinfo);
     printf ("resolution %dx%d  bpp %d\n",
          fb_varinfo.xres, fb_varinfo.yres, fb_varinfo.bits_per_pixel);

     fill_bg (fd, fb_fixinfo.smem_len);
     close (fd);
     return 0;
}
```

Example frame buffer application (2)

```
/* converts 24-bit 888 RGB value to 16-bit 555 RGB value */
static inline unsigned short rgb_888_to_555(int rgb888)
{
        return
       (((rgb888 & 0xff0000) >> 9) & 0x7c00) |
       (((rgb888 & 0x00ff00) >> 6) & 0x03e0) |
       (((rgb888 & 0x0000ff) >> 3) & 0x001f);
}

void fill_bg (int fd, int fb_size)
{
     unsigned short *fb_mem;
     unsigned short bg_colour = rgb_888_to_555 (0xff0000); /* red */
     unsigned short *outp;
     int i;

     fb_mem = mmap (0, fb_size, PROT_READ | PROT_WRITE, MAP_SHARED, fd, 0);
     outp = (unsigned short *)fb_mem;
     for (i = 0; i < fb_size/2; i++)
          *outp++ = bg_colour;
     munmap (fb_mem, fb_size);
     return 0;
}
```

The Linux input layer

- The input layer notifies applications of events such as keyboard key presses and mouse movement
- Files are in /dev/input
  - /dev/input/event* are the raw events
  - /dev/input/mouse* are translated mouse data

Input events

The event nodes, event0..event63, are character devices with major number 13 and minor number 64 .. 127

Reading an event node returns 16-bye event records. Each event is a `struct input_event` (defined in `linux/input.h`):

```
struct input_event {
    struct timeval time;  /* Time stamp */
    __u16 type;       /* Type of event, e.g. EV_KEY, EV_REL */
    __u16 code;       /* Key code, etc. */
    __s32 value;      /* For mice, the movement. For keyboards
                0 for release, 1 for press, 2 for
                auto-repeat */
};
```

Event types

Keyboard events

type	EV_KEY
code	key code defined in linux/input.h
value	1 for press, 0 for release

Pointer movement events

type	EV_REL (e.g. mouse) or EV_ABS (e.g. touch screen)
code	REL_X, REL_Y, ABS_X or ABS_Y
value	x or y value, relative or absolute

Pointer button click events

type	EV_KEY
code	BTN_LEFT, BTN_RIGHT, BTN_MIDDLE, etc
value	1 for press, 0 for release

“Sync” event, follows one or more other events to indicate report complete

type	EV_SYN
code	SYN_REPORT
value	0

Getting information about input drivers

Input drivers support 20 ioctl calls, including:

```
#include <linux/input.h>
```

`EVIOCGVERSION`	get driver version (== EV_VERSION)
`EVIOCGNAME`	get device name
`EVIOCGBIT`	get event or keyboard bits

Use EVIOCGBIT to find the capabilities of the device. The information is returned as a bit mask:

`EVIOCGBIT(0, EV_MAX)`	mask of event types it can generate
`EVIOCGBIT(EV_KEY, KEY_MAX)`	mask of key codes
`EVIOCGBIT(EV_REL, REL_MAX)`	mask of relative pointer events
`EVIOCGBIT(EV_ABS, ABS_MAX)`	mask of absolute pointer events

Example: information about event drivers

```
#include <linux/input.h>

int main (int argc, char **argv)
{
     char *event_name = "/dev/input/event0";
     int version;
     int  fd;
     char name [80];
     unsigned long evbit;

     open (event_name, O_RDWR);
     ioctl (fd, EVIOCGVERSION, &version);
     printf ("Event protocol version %x\n", version);
     ioctl (fd, EVIOCGNAME (sizeof (name) - 1), name);
     printf ("%s is %s\n", event_name, name);
     ioctl (fd, EVIOCGBIT(0, EV_MAX), &evbit);
     if (test_bit (EV_KEY, evbit))
          printf ("  Has keys or buttons\n");
     if (test_bit (EV_REL, evbit))
          printf ("  Relative pointer\n");
     if (test_bit (EV_ABS, evbit))
          printf ("  Absolute pointer\n");
     close (fd);
     return 0;
}
```

Get the event mask and test for key, relative pointer and absolute pointer events

Example: information about event drivers

Result of running the program with one event driver - a Microsoft mouse - loaded:

```
# event-info
Event protocol version 10000
/dev/input/event0 is Microsoft Basic Optical Mouse
  Has keys or buttons
  Relative pointer
```