The Frame Buffer Device

Last revised: May 10, 2001

0. Introduction

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer of some video hardware and allows application software to access the graphics hardware through a well-defined interface, so the software doesn't need to know anything about the low-level (hardware register) stuff.

The device is accessed through special device nodes, usually located in the /dev directory, i.e. /dev/fb*.

1. User's View of /dev/fb*

From the user's point of view, the frame buffer device looks just like any other device in /dev. It's a character device using major 29; the minor specifies the frame buffer number.

By convention, the following device nodes are used (numbers indicate the device minor numbers):

```
  0 = /dev/fb0      First frame buffer
  1 = /dev/fb1      Second frame buffer
      ...
 31 = /dev/fb31     32nd frame buffer
```

For backwards compatibility, you may want to create the following symbolic links:

```
/dev/fb0current -> fb0
/dev/fb1current -> fb1
```

and so on...

The frame buffer devices are also *normal* memory devices, this means, you can read and write their contents. You can, for example, make a screen snapshot by:

```
cp /dev/fb0 myfile
```

There also can be more than one frame buffer at a time, e.g. if you have a graphics card in addition to the built-in hardware. The corresponding frame buffer devices (/dev/fb0 and /dev/fb1 etc.) work independently.

Application software that uses the frame buffer device (e.g. the X server) will use /dev/fb0 by default (older software uses /dev/fb0current). You can specify an alternative frame buffer device by setting the environment variable $FRAMEBUFFER to the path name of a frame buffer device, e.g. (for sh/bash users):

```
export FRAMEBUFFER=/dev/fb1
```

or (for csh users):

```
setenv FRAMEBUFFER /dev/fb1
```

After this the X server will use the second frame buffer.

2. Programmer's View of /dev/fb*

As you already know, a frame buffer device is a memory device like /dev/mem and it has the same features. You can read it, write it, seek to some location in it and mmap() it (the main usage). The difference is just that the memory that appears in the special file is not the whole memory, but the frame buffer of some video hardware.

/dev/fb* also allows several ioctls on it, by which lots of information about the hardware can be queried and set. The color map handling works via ioctls, too. Look into <linux/fb.h> for more information on what ioctls exist and on which data structures they work. Here's just a brief overview:

- You can request unchangeable information about the hardware, like name, organization of the screen memory (planes, packed pixels, ...) and address and length of the screen memory.
- You can request and change variable information about the hardware, like visible and virtual geometry, depth, color map format, timing, and so on. If you try to change that information, the driver maybe will round up some values to meet the hardware's capabilities (or return EINVAL if that isn't possible).
- You can get and set parts of the color map. Communication is done with 16 bits per color part (red, green, blue, transparency) to support all existing hardware. The driver does all the computations needed to apply it to the hardware (round it down to less bits, maybe throw away transparency).

All this hardware abstraction makes the implementation of application programs easier and more portable. E.g. the X server works

completely on /dev/fb* and thus doesn't need to know, for example, how the color registers of the concrete hardware are organized. XF68_FBDev is a general X server for bitmapped, unaccelerated video hardware. The only thing that has to be built into application programs is the screen organization (bitplanes or chunky pixels etc.), because it works on the frame buffer image data directly.

For the future it is planned that frame buffer drivers for graphics cards and the like can be implemented as kernel modules that are loaded at runtime. Such a driver just has to call register_framebuffer() and supply some functions. Writing and distributing such drivers independently from the kernel will save much trouble...

3. Frame Buffer Resolution Maintenance

Frame buffer resolutions are maintained using the utility *fbset*. It can change the video mode properties of a frame buffer device. Its main usage is to change the current video mode, e.g. during boot up in one of your */etc/rc.** or */etc/init.d/** files.

Fbset uses a video mode database stored in a configuration file, so you can easily add your own modes and refer to them with a simple identifier.

4. The X Server

The X server (XF68_FBDev) is the most notable application program for the frame buffer device. Starting with XFree86 release 3.2, the X server is part of XFree86 and has 2 modes:

- If the *Display* subsection for the *fbdev* driver in the /etc/XF86Config file contains a:

  ```
  Modes "default"
  ```

  line, the X server will use the scheme discussed above, i.e. it will start up in the resolution determined by /dev/fb0 (or $FRAMEBUFFER, if set). You still have to specify the color depth (using the Depth keyword) and virtual resolution (using the Virtual keyword) though. This is the default for the configuration file supplied with XFree86. It's the most simple configuration, but it has some limitations.
- Therefore it's also possible to specify resolutions in the /etc/XF86Config file. This allows for on-the-fly resolution switching while retaining the same virtual desktop size. The frame buffer device that's used is still /dev/fb0current (or $FRAMEBUFFER), but the available resolutions are defined by /etc/XF86Config now. The disadvantage is that you have to specify the timings in a different format (but *fbset -x* may help).

To tune a video mode, you can use fbset or xvidtune. Note that xvidtune doesn't work 100% with XF68_FBDev: the reported clock values are always incorrect.

5. Video Mode Timings

A monitor draws an image on the screen by using an electron beam (3 electron beams for color models, 1 electron beam for monochrome monitors). The front of the screen is covered by a pattern of colored phosphors (pixels). If a phosphor is hit by an electron, it emits a photon and thus becomes visible.

The electron beam draws horizontal lines (scanlines) from left to right, and from the top to the bottom of the screen. By modifying the intensity of the electron beam, pixels with various colors and intensities can be shown.

After each scanline the electron beam has to move back to the left side of the screen and to the next line: this is called the horizontal retrace. After the whole screen (frame) was painted, the beam moves back to the upper left corner: this is called the vertical retrace. During both the horizontal and vertical retrace, the electron beam is turned off (blanked).

The speed at which the electron beam paints the pixels is determined by the dotclock in the graphics board. For a dotclock of e.g. 28.37516 MHz (millions of cycles per second), each pixel is 35242 ps (picoseconds) long:

```
1/(28.37516E6 Hz) = 35.242E-9 s
```

If the screen resolution is 640x480, it will take:

```
640*35.242E-9 s = 22.555E-6 s
```

to paint the 640 (xres) pixels on one scanline. But the horizontal retrace also takes time (e.g. 272 *pixels*), so a full scanline takes:

```
(640+272)*35.242E-9 s = 32.141E-6 s
```

We'll say that the horizontal scanrate is about 31 kHz:

```
1/(32.141E-6 s) = 31.113E3 Hz
```

A full screen counts 480 (yres) lines, but we have to consider the vertical retrace too (e.g. 49 *lines*). So a full screen will take:

```
(480+49)*32.141E-6 s = 17.002E-3 s
```

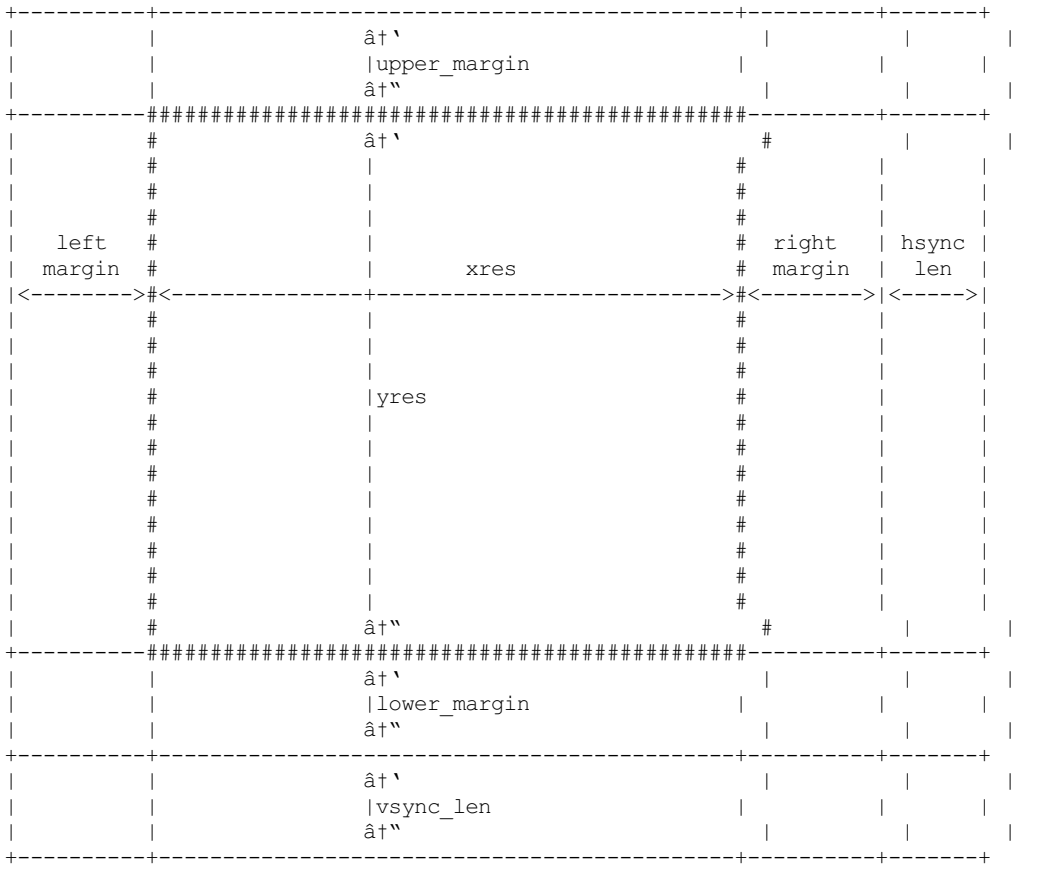
The vertical scanrate is about 59 Hz:

```
1/(17.002E-3 s) = 58.815 Hz
```

This means the screen data is refreshed about 59 times per second. To have a stable picture without visible flicker, VESA recommends a vertical scanrate of at least 72 Hz. But the perceived flicker is very human dependent: some people can use 50 Hz without any trouble, while I'll notice if it's less than 80 Hz.

Since the monitor doesn't know when a new scanline starts, the graphics board will supply a synchronization pulse (horizontal sync or hsync) for each scanline. Similarly it supplies a synchronization pulse (vertical sync or vsync) for each new frame. The position of the image on the screen is influenced by the moments at which the synchronization pulses occur.

The following picture summarizes all timings. The horizontal retrace time is the sum of the left margin, the right margin and the hsync length, while the vertical retrace time is the sum of the upper margin, the lower margin and the vsync length:



The frame buffer device expects all horizontal timings in number of dotclocks (in picoseconds, 1E-12 s), and vertical timings in number of scanlines.

6. Converting XFree86 timing values info frame buffer device timings

An XFree86 mode line consists of the following fields:

```
"800x600"     50      800  856  976 1040    600  637  643  666
< name >     DCF       HR  SH1  SH2  HFL     VR  SV1  SV2  VFL
```

The frame buffer device uses the following fields:

- pixclock: pixel clock in ps (pico seconds)
- left_margin: time from sync to picture
- right_margin: time from picture to sync
- upper_margin: time from sync to picture
- lower_margin: time from picture to sync
- hsync_len: length of horizontal sync
- vsync_len: length of vertical sync

1. Pixelclock:

   xfree: in MHz

   fb: in picoseconds (ps)

   pixclock = 1000000 / DCF

2. horizontal timings:

   left_margin = HFL - SH2

   right_margin = SH1 - HR

hsync_len = SH2 - SH1

3. vertical timings:

   upper_margin = VFL - SV2

   lower_margin = SV1 - VR

   vsync_len = SV2 - SV1

Good examples for VESA timings can be found in the XFree86 source tree, under "xc/programs/Xserver/hw/xfree86/doc/modeDB.txt".

7. References

For more specific information about the frame buffer device and its applications, please refer to the Linux-fbdev website:

http://linux-fbdev.sourceforge.net/

and to the following documentation:

- The manual pages for fbset: fbset(8), fb.modes(5)
- The manual pages for XFree86: XF68_FBDev(1), XF86Config(4/5)
- The mighty kernel sources:
  - linux/drivers/video/
  - linux/include/linux/fb.h
  - linux/include/video/

8. Mailing list

There is a frame buffer device related mailing list at kernel.org: linux-fbdev@vger.kernel.org.

Point your web browser to http://sourceforge.net/projects/linux-fbdev/ for subscription information and archive browsing.

9. Downloading

All necessary files can be found at

ftp://ftp.uni-erlangen.de/pub/Linux/LOCAL/680x0/

and on its mirrors.

The latest version of fbset can be found at

http://www.linux-fbdev.org/

10. Credits

This readme was written by Geert Uytterhoeven, partly based on the original *X-framebuffer.README* by Roman Hodek and Martin Schaller. Section 6 was provided by Frank Neumann.

The frame buffer device abstraction was designed by Martin Schaller.