Development of a Frame Buffer Driver for Embedded Linux Graphic System

Ga-Gue Kim, Woo-Chul Kang, Young-Jun Jung, and Hyung-Seok Lee

Embedded S/W Technology Center, ETRI, Daejeon, Korea
(Tel : +82-42-860-1123; E-mail: {ggkim, wchkang, jjing, hyslee@etri.re.kr)

Abstract: A frame buffer device is an abstraction for the graphic hardware. It allows application software to access the graphic hardware through a well-defined interface, so that the software doesn't need to know anything about the low-level interface stuff. We develop a frame buffer driver for VIA's CLE266 graphic system based on 'Qplus', an embedded linux operating system developed by ETRI. Then, it will be seen that our frame buffer system is applied to embedded solutions such as movie player and X server successfully.

Keywords: frame buffer, embedded linux, Qplus, abstract console

1. INTRODUCTION

Many machines (M68K, SPARC, PowerMac) use graphical console because either the hardware does not support (VGA) text mode, or because the firmware programs the hardware into a graphical mode. Thus the linux kernel needs to be aware of this and support graphical consoles on these architechtures.

Graphical consoles will also gain more importance on the Intel platform, as VGA compatibility will be phased out by many graphics chipset manufacturers in the near future. An early example of this the Cyrix MediaGX], which provides VGA compatibility through its BIOS only.

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 does not need to know about the low level (hardware register) stuff (except for hardware acceleration)[1].

Since years there has been minor support for graphical consoles in the linux kernel, but it differed a lot among the different platforms. Starting with kernel 2.1.107, the frame buffer device abstraction become completely integrated in the standard kernel sources and will become the standard way to access graphics hardware. But it is definitely not new: it originated from linux/M68K at the end of 1994 and has proven to fulfill its task during the previous years.

The frame buffer device abstraction has the following advantages:

(1) It provides a unified method to access graphics hardware across different platforms.

(2) Drivers can be shared among different architectures, which reduces code duplication. There were already three different drivers for the ATI Mach64 before. In the ideal case, a frame buffer device driver contains a chipset driver core, with machine and bus dependent probe code(Zorro/PCI/ISA/Open Firmware/…).

(3) It provides simple multi-head: currently up to 8 frame buffer devices (displays) are supported. Unfortunately the input part of the console subsystem is not ready for multi-head yet.

(4) On boot up, you get one or more penguin logos (with or without beer) The more CPUs you have, the more penguins you will see.

2. FRAME BUFFER INTERNAL API

Now that we understand the basic ideas behind video card technology and mode setting, we can now look at how the framebuffer devices abstract them. Also, we will see that fbdev actually handles most of the mode setting issues for you to make life much easier. In the older API, the console code was heavily linked to the framebuffer devices. The newer API has now moved nearly all console handling code into fbcon itself. Now, fbcon is a true wrapper around the video card's abilities. This allows for massive code reduction and easier driver development. A good example of a framebuffer driver is the virtual framebuffer (vfb). The vfb driver is not a true framebuffer driver. All it does is map a chunk of memory to userspace. It's used for demonstration purposes and testing.

2.1 Data Structures

The framebuffer drivers depend heavily on four data structures. These structures are declared in fb.h. They are *fb_var_screeninfo*, *fb_fix_screeninfo*, *fb_monospecs*, and *fb_info*. The first three can be made available to and from userland. First let me describe what each means and how they are used.

fb_var_screeninfo is used to describe the features of a video card you normally can set. With *fb_var_screeninfo*, you can define such things as depth and the resolution you want.

The next structure is *fb_fix_screeninfo*. This defines the properties of a card that are created when you set a mode and can't be changed otherwise. A good example is the start of the framebuffer memory. This can depend on what mode is set. Now while using that mode, you don't want to have the memory position change on you. In this case, the video hardware tells you the memory location and you have no say about it.

The third structure is *fb_monospecs*. In the old API, the importance of *fb_monospecs* was very little. This allowed for forbidden things such as setting a mode of 800x600 on a fix frequency monitor. With the new API, *fb_monospecs* prevents such things, and if used correctly, can prevent a monitor from being cooked.

The final data structure is *fb_info*. This defines the current state of the video card. *fb_info* is only visible from the kernel. Inside of *fb_info*, there exist a *fb_ops* which is a collection of needed functions to make fbdev and fbcon work.

2.2 Driver Layout

Here I describe a clean way to code your drivers. A good example of the basic layout is vfb.c. In the example driver, we first present our data structures in the beginning of the file. Note that there is no *fb_monospecs* since this is handled by code in fbmon.c. This can be done since monitors are independent in behavior from video cards. First, we define our three basic data structures. For all the data structures I defined them static and declare the default values. The reason I do this is because it's less memory intensive than to allocate a piece of memory and filling in the default values. Note that drivers that support multihead (multiple video cards) of the same card, then the *fb_info* should be dynamically allocated for each card present. For *fb_var_screeninfo* and *fb_fix_screeninfo*, they still are declared static since all the cards can be set to the same mode.

2.3 Initialization and boot time parameter handling

There are two functions that handle the video card at boot time:

```
int xxfb_init(void);
int xxfb_setup(char*);
```

In the example driver as with most drivers, these functions are placed at the end of the driver. Both are very card specific. In order to link your driver directly into the kernel, both of these functions must add the above definition with extern in front to fbmem.c. Add these functions to the following in fbmem.c:

```
static struct {
    const char *name;

    int (*init)(void);
    int (*setup)(char*);
} fb_drivers[] __initdata = {
#ifdef CONFIG_FB_YOURCARD
    { "driver_name", xxxfb_init, xxxfb_setup },
#endif
```

Setup is used to pass card specific options from the boot prompt of your favorite boot loader. A good example is:

```
boot: video=matrox: vesa: 443
```

The basic setup function is:

```
int __init xxxfb_setup(char *options)
{
    char *this_opt;

    if (!options || !*options)
        return 0;
     for (this_opt = strtok(options, ","); this_opt;
        this_opt = strtok(NULL,          ","))
         if (!strcmp(this_opt, "my_option")) {
        /* Do your stuff. Usually set some static flags that the
driver later uses */
         } else if (!strncmp(this_opt, "Other_option:", 5))
            strcpy(some_flag_driver_uses, this_opt+5);
        } else ....
    }
}
```

The xxfb_init function sets the initial state of the video card. This function has to consider bus and platform handling since today most cards can exist on many platforms. For bus types we have to deal with, there are PCI, ISA, and zorro. Also, some platforms offer firmware that returns information about the video card. In this case, we often don't need to deal with the bus unless we need more control over the card. Let us look at Open Firmware that’s available on PowerPCs. If you are going to use Open Firmware to initialize your card, you need to add the following to offb.c.

```
#ifdef CONFIG_FB_YOUR_CARD
extern void xxxfb_of_init(struct device_node *dp);
#endif /* CONFIG_FB_YOUR_CARD */
```

Then in the function offb_init_driver, you add something similar to the following:

```
#ifdef CONFIG_FB_YOUR_CARD
if (!strncmp(dp->name,"Open Firmware number of your
card ", size_of_name)) {
    xxxfb_of_init(dp);
    return 1;
}
#endif /* CONFIG_FB_YOUR_CARD */
```

If Open Firmware doesn't detect your card, Open Firmware sets up a generic video mode for you. Now in your driver you really need two initialization functions.

The next major part of the driver is declaring the functions of *fb_ops* that are declared in *fb_info* for the driver.

The first two functions, *xxfb_open* and *xxfb_release*, can be called from both fbcon and fbdev. In fact, that's the use of the user flag. If user equals zero then fbcon wants to access this device, else it's an explicit open of the framebuffer device. This way, you can handle the framebuffer device for the console in a special way for a particular video card. For most drivers, this function just does a *MOD_INC_USE_COUNT* or *MOD_DEC_USE_COUNT*.

These are the functions that are at the heart of mode setting. There do exist a few cards that don't support mode changing. For these we have this function return an -EINVAL to let the user know he/she can't set the mode. Actually, set_var does more than just set modes. It can check them as well. In *fb_var_screeninfo*, there exists a flag called activate. This flag can take on the following values: *FB_ACTIVATE_NOW*, *FB_ACTIVATE_NXTOPEN*, and *FB_ACTIVATE_TEST*.

FB_ACTIVATE_TEST tells us if the hardware can handle what the user requested. *FB_ACTIVATE_NXTOPEN* sets the values wanted on the next explicit open of fbdev. The final one *FB_ACTIVATE_NOW* checks the mode to see if it can be done and then sets the mode. You MUST check the mode before all things. Note that this function is very card specific, but I will attempt to give you the most general layout. The basic layout then for *xxxfb_set_var* is:

```
static int vfb_set_var(struct fb_var_screeninfo *var, struct
fb_info *info)
{
    int line_length;
    /* Basic setup test. Here we look at what the user passed
in that he/she wants.For example to test the fb_var_screeninfo
field vmode like its done in vfb.c.Here we see if the user has
FB_VMODE_YWARP. Also we should look to see if the user
tried to pass in invalid values like 17 bpp (bits per pixel) */
    /* Remember the above discussion on how monitors see a
mode. They don't care about bit depth.   So you can divide the
checking into two parts. One is to see if the user changed a
mode from say 640x480 at 8 bpp to 640x480 at 32
bpp.Remember the var in fb_info represents the current video
mode. Before we actually change any resolutions we have to
make sure the card has enough memory for the new mode.
Discovering how much memory a video card has varies from
card to card.   Also finding out how much memory we have is
done in xxxfb_init since this never changes unless you add
more memory to your card, which requires a reboot of the
machine anyway. You might have to do other tests depending
on make of your card. Note the par filed in fb_info. This is
used to store card specific data. This data can affect set_var.
```

```
Also it is present to allow other possible drivers that could
effect the framebuffer device such as a special driver for an
accel engine or memory mapping the Z buffer on a card */
   /* Summary. First look at any var fields to see if they are
valid. Next test hardware with these fields without setting the
hardware. An example of one is to find what the line_length
would be for the new mode. Then test the following: */
   if ((line_length * var->yres_virtual) > info->fix.smem_len)
      return -ENOMEM;
   if (info->var.xres != var->xres || info->var.yres !=
var->yres || info->var.xres_virtual != var->xres_virtual ||
info->var.yres_vitual != var->yres_virtual) {

            /* Resolution changed !!! */

            /* Next you must check to see if the monitor can
handle this mode. Don't want to fry your monitor or mess up
the display really badly */
   if (fbmon_valid_timings(u_int pixclock, u_int htotal, u_int
vtotal, const struct fb_info *fb_info))
   /* Can't handle these timings. */
      return -EINVAL;
   /* Timings are okay. Next we see if we really want to
change this mode */
   if ((activate & FB_ACTIVATE_MASK) ==
FB_ACTIVATE_NOW) {

   /* Now lets program the clocks on this card. Here the code
is very card specific. Remember to change any fields for fix in
info that might be affected by the changing of the resolution.
*/
      info->fix.line_length          = line_length;
   /* Now that we have dealt with the possible changing
resolutions lets handle a possible change of bit depth. */
   if (info->var.bits_per_pixel != var->bits_per_pixel) {
      if ((err = fb_alloc_cmap(&info->cmap, 0, 0)))
         return err;
   }
}
   /* We have shown that the monitor and video card can
handle this mode or have actually set the mode.  Next the
fb_bitfield structure in fb_var_screeninfo is filled in. Even if
you don't set the mode you get a feel of the mode before you
really set it. These are typical values but may be different for
your card. For truecolor modes all the fields matter. For
pseudocolor modes only the length matters. Thus all the
lengths should be the same (=bpp). */
   switch (var->bits_per_pixel) {
      case 1:
      case 8:
      /* Pseudocolor mode example */
         var->red.offset      = 0;
         var->red.length      = 8;
         var->green.offset    = 0;
         var->green.length    = 8;
         var->blue.offset     = 0;
         var->blue.length     = 8;
         var->transp.offset = 0;
         var->transp.length = 0;
         break;
      case 16:       /* RGB 565 */
         var->red.offset      = 0;
         var->red.length      = 5;
         var->green.offset    = 5;
         var->green.length    = 6;
         var->blue.offset     = 11;
         var->blue.length     = 5;
         var->transp.offset = 0;
         var->transp.length = 0;
         break;
      case 24:       /* RGB 888 */
         var->red.offset      = 0;
         var->red.length      = 8;
         var->green.offset    = 8;
         var->green.length    = 8;
         var->blue.offset     = 16;
         var->blue.length     = 8;
         var->transp.offset = 0;
         var->transp.length = 0;
         break;
      case 32:       /* RGBA 8888 */
         var->red.offset      = 0;
         var->red.length      = 8;
         var->green.offset    = 8;
         var->green.length    = 8;
         var->blue.offset     = 16;
         var->blue.length     = 8;
         var->transp.offset = 24;
         var->transp.length = 8;
         break;
   }
   /* Yeah. We are done !!! */
}
```

The function *xxxfb_setcolreg* is used to set a single color register for a video card. To use this properly, you must understand colors, which is described above. This routine sets a color map entry. The regno passed into the routine represents the color map index which is equal to the color that's composed of the amount of red, green, blue, and even alpha that are also passed into the function. For pseudocolor modes, this color map index (regno) represents the pixel value. So if you place a pixel value of regno in video memory, you get the color that's made of the red, green, blue that you passed into *xxxfb_setcolreg*. Now for truecolor and directcolor mode, it's a little different. In this case, we simulate a pseudo color map. The reason for this is the console system always has a color map, which has 16 entries. In *fb_info*, there exist the pseudo_palette, which gives a mapping from a non-color map mode to a color map based system. The pseudo_palette always has 17 entries. The first 16 is for the console colors and the last one for the cursor. So if we wanted to display the 4 entry in the color map of the console, we would place the value of info->psuedo_palette[4] directly into the video memory. This is, of course, taken care of by fbcon. You just need to code the "formula" that does this translation. An example follows for 32-bit mode:

```
red >>= 8;
green >>= 8;
blue >>= 8;
info->pseudo_palette[regno] =
   (red   << info->var.red.offset)       |
   (green << info->var.green.offset)     |
   (blue  << info->var.blue.offset);
```

Here, we first scale down the color components. Each color passed to set_colreg is 16 bits in size. For 32-bit mode, each color is 8 bits in size. Next, we OR the colors together after we have offseted them. The offset is used because the pixel layout in 32 bits could be RBGA, ARGBA, etc. In setcol_reg of vfb.c, is the standard way to deal with packed pixel format of various image depths. Regno is the index to get this particular color.

That does it for required functions besides the set of needed accel functions, which has not been discussed yet. If the video card doesn't support the function, then we just place a NULL in fb_ops. The next function in fb_ops is *xxxfb_blank*. This function provides support for hardware blanking. For *xxxfb_blank*, the first parameter represents the blanking modes available. They are *VESA_NO_BLANKING*, *VESA_VSYNC_SUSPEND*, *VESA_HSYNC_SUSPEND*, and *VESA_POWERDOWN*. *VESA_NO_BLANKING* powers up the display again. *VESA_POWERDOWN* turns off the display. This is a great power saving feature on a laptop.

The next optional function is *xxxfb_pan_display*. This function enables panning. Panning is often used for scrolling.

The ioctl function gives you the power to take advantage of special features other cards don't have. If your card is nothing special then just give this *fb_ops* function a NULL pointer. The sky is the limit for defining your ioctl calls.

There exists a default memory map function for fbdev, but sometimes it just doesn't have the power you truly need. A good example of this is video cards that work in sparc workstations that need their own mmap functions because of the way sparcs handle memory is different from other platforms. This is true even for sparcs with PCI buses.

Now here is the next class of functions that are optional -- *xxxfb_accel_init* and *xxfb_accel_done*. *xxxfb_accel_init* really depends on the card. It is intended to initialize the engine or set the accel engine into a state so that you can use the acceleration engine. It also ensures that the framebuffer is not accessed at the same time as the accel engine. This can lock a system. Usually, there exists a bit to test to see if an accel engine is idle or if the card generates an interrupt. For cards that used the old fb_rasterimg, this function replaces it. Some cards have separate states for 3D and 2D. This function insures that the card goes into a 2D state. Just in case a previous application set the accel engine into a 3D state or made the accel engine very unhappy. The next function that encompasses this set is *xxxfb_accel_done*. This function sets the video card in a state such that you can write to the framebuffer again. You should provide both functions if your driver uses even one hardware accelerated function. The reason being is to ensure that the framebuffer is not accessed at the same time as the accel engine.

Finally, the third class of fb_op functions. Like the first, they are required. If your card does not support any of these accelerated functions, there exist default functions for packed pixel framebuffer formats. They are *cfba_fillrect*, *cfba_copyarea*, and *cfba_imgblit*. If your driver supports some but not all of the accels available, you can still use some of these software emulated accels. Each software-emulated accel is stored in a separate file. Now lets describe each accel function. Before we discuss these functions we need to note not to draw in areas pass the video boundaries. If it does, you need to adjust the width and height of the areas to avoid this problem.

The first function just fills in a rectangle starting at x1 and y1 of some width and height with a pixel value of packed pixel format. If the video memory mapping is not a direct mapping from the pixel value (not *FB_TYPE_PACKED_PIXEL*), you will have to do some translating. There are two ways to fill in the rectangle, *FBA_ROP_COPY* and *FBA_ROP_XOR*. *FBA_ROP_XOR* exclusive ORs the pixel value with the current pixel value. This allows things like quickly erasing a rectangular area. The other function just directly copies the data.

The next function is *xxxfb_copyarea*. It just copies one area of the framebuffer at source x and source y of some width and height to some destination x and y.

The final function is *xxxfb_imageblt*. This function copies an image from system memory to video memory. You can get really fancy here but this is fbdev, which has the purpose of mode setting only. All the image blit function does is draw bitmaps, image made of a foreground and background color, and a color image of the same color depth as the framebuffer. The second part is used to draw the little penguins. The drawing of bitmaps is used to draw our fonts.

3. QPLUS TARGET BUILDER

Current embedded systems increasingly demand the services of a sophisticated, state-of-the-art operating system. Many such systems require advanced capabilities like: high resolution and user-friendly graphical user interface(GUIs); TCP/IP connectivity; substitution of reliable (and low power) flash memory; support for 32-bit high-speed CPUs.

These needs led many embedded system developer to look Linux as a convenient and low-cost way to solve their problems. Linux is open-source and has many modern OS functionalities required for those systems, and it supports many platforms and devices also.

But embedded Linux lacks of convenient development tools which help developer configure, build and deploy the system. Linux and most application on it are open-source and that means you must do every chore to build a working embedded Linux system.

If you do embedded Linux development, First, you need to install cross tool-chain, libraries and header files. And the next step is creating kernel and root image. In this step, you must configure and build every needed components. Then, you should create bunch of boot scripts and configuration files (usually placed in /etc directory). Finally, you must create and transfer bootable image to the target board. Figure 1 shows general development sequence of embedded Linux.

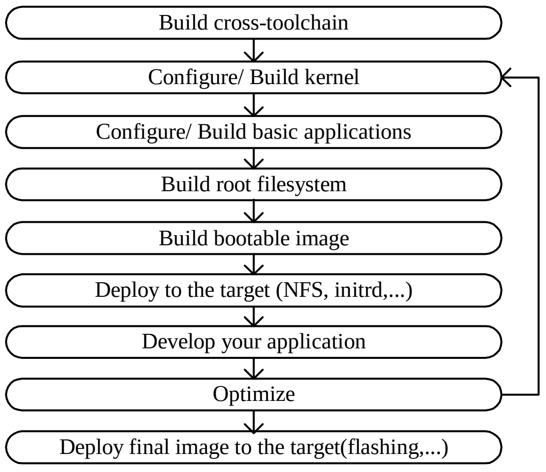
Build cross-toolchain
Configure/ Build kernel
Configure/ Build basic applications
Build root filesystem
Build bootable image
Deploy to the target (NFS, initrd,...)
Develop your application
Optimize
Deploy final image to the target(flashing,...)

Fig. 1 Development procedure Embedding Linux.

Most of these procedures have to be done repeatedly until the final deployment. Configuration, build and deployment to the target has been done manually and separately without assistance of integrated tools. It is highly error-prone and time-consuming job. Worst of all, Linux is not well documented. It made embedded Linux development require long learning time and high experience, and it lead to delayed time-to-market. Embedded system developers want their operating environment to just work so they can get to work right away on developing the actual application.

Tools to automate and assist this development process can make developer to save much time and effort. Recent survey showed 69% of the Embedded Linux developers is willing to pay for development tools.

In particular, the user demands a system where the tools are closely integrated and have a consistent user interface and help facility.

Target Builder is a toolkit to assist and to automate all development process. It shows all configurable options of kernel, applications and target environment in a tree. So developers can navigate all those options and set them to proper values. Target Builder shows on-line help for each option and check dependencies among them. If dependencies are violated, Target Builder informs the user what caused the violation. This dependency checking can dramatically reduce errors in configuration stage. It automates all other process; root file system generation, optimizations and deployment to the target[3].

REFERENCES

[1] Geert Uytterhoeven, “The Linux Frame Buffer Device Subsytem,” *http://www.cs.kuleuven.ac.be/~geert*

[2] James Simmons, “Linux Framebuffer Driver Writing HOWTO,” *http://metalab.unc.edu/LDP*, 1999.

[3] W. C. Kang, H. C. Yun, H. N. Kim, “Target Builder: An Embedded Linux Development Toolkit,” *Proceedings of the IASTED Software Engineering and Appplication(SEA ‘02’)*, pp. 163-167, Boston, USA, 2002.

[4] CLE266 Chipset Datasheet, VIA Technologies, inc. and S3 Graphics, inc., 2003.