|  |  |
| --- | --- |
| |  | | --- | | **Framebuffer的配置及应用**  \*一、FrameBuffer的原理\*  FrameBuffer 是出现在 2.2.xx 内核当中的一种驱动程序接口。  Linux是工作在保护模式下，所以用户态进程是无法象DOS那样使用显卡BIOS里提供的中断调用来实现直接写屏，Linux抽象出FrameBuffer这 个设备来供用户态进程实现直接写屏。Framebuffer机制模仿显卡的功能，将显卡硬件结构抽象掉，可以通过Framebuffer的读写直接对显存进行操 作。用户可以将Framebuffer看成是显示内存的一个映像，将其映射到进程地址空间之后，就可以直接进行读写操作，而写操作可以立即反应在屏幕上。这种操 作是抽象的，统一的。用户不必关心物理显存的位置、换页机制等等具体细节。这些都是由Framebuffer设备驱动来完成的。      但Framebuffer本身不具备任何运算数据的能力,就只好比是一个暂时存放水的水池.CPU将运算后的结果放到这个水池，水池再将结果流到显示器.中间不会对数据做处理. 应用程序也可以直接读写这个水池的内容.在这种机制下，尽管Framebuffer需要真正的显卡驱动的支持，但所有显示任务都有CPU完成,因此CPU负担很重.  framebuffer的设备文件一般是 /dev/fb0、/dev/fb1 等等。  可以用命令: #dd if=/dev/zero of=/dev/fb 清空屏幕.  如果显示模式是 1024x768-8 位色，用命令：$ dd if=/dev/zero of=/dev/fb0 bs=1024 count=768 清空屏幕；  用命令: #dd if=/dev/fb of=fbfile  可以将fb中的内容保存下来；  可以重新写回屏幕: #dd if=fbfile of=/dev/fb；  在使用Framebuffer时，Linux是将显卡置于图形模式下的．       在应用程序中，一般通过将 FrameBuffer 设备映射到进程地址空间的方式使用，比如下面的程序就打开 /dev/fb0 设备，并通过mmap 系统调用进行地址映射，随后用 memset 将屏幕清空（这里假设显示模式是 1024x768-8 位色模式，线性内存模式）：  int fb;  unsigned char\* fb\_mem;  fb = open ("/dev/fb0", O\_RDWR);  fb\_mem = mmap (NULL, 1024\*768, PROT\_READ|PROT\_WRITE,MAP\_SHARED,fb,0);  memset (fb\_mem, 0, 1024\*76http://bbs.chinaunix.net/static/image/smiley/default/icon_cool.gif;       FrameBuffer 设备还提供了若干ioctl命令，通过这些命令，可以获得显示设备的一些固定信息（比如显示内存大小）、与显示模式相关的可变信息（比如分辨率、象素结构、每扫描线的字节宽度），以及伪彩 色模式下的调色板信息等等。      通过 FrameBuffer设备，还可以获得当前内核所支持的加速显示卡的类型（通过固定信息得到），这种类型通常是和特定显示芯片相关的。比如目前最新的内核（2.4.9）中，就包含有 对S3、Matrox、nVidia、3Dfx 等等流行显示芯片的加速支持。在获得了加速芯片类型之后，应用程序就可以将 PCI设备的内存I/O（memio）映射到进程的地址空间。这些memio  一般是用来控制显示卡的寄存器，通过对这些寄存器的操作，应用程序就可以控制特定显卡的加速功能。      PCI设备可以将自己的控制寄存器映射到物理内存空间，而后，对这些控制寄存器的访问，给变成了对物理内存的访问。因此，这些寄存器又被称为"memio"。一旦被映 射到物理内存，Linux的普通进程就可以通过 mmap 将这些内存 I/O 映射到进程地址空间，这样就可以直接访问这些寄存器了。      当然，因为不同的显示芯片具有不同的加速能力，对memio的使用和定义也各自不同，这时，就需要针对加速芯片的不同类型来编写实现不同的加速功能。比如大多数芯片都提供了对矩形填充的硬件加速支持，但不同的芯片实现方 式不同，这时，就需要针对不同的芯片类型编写不同的用来完成填充矩形的函数。      FrameBuffer 只是一个提供显示内存和显示芯片寄存器从物理内存映射到进程地址空间中的设备。所以，对于应用程序而言，如果希望在FrameBuffer 之上进行图形编程，还需要自己动手完成其他许多工作。  \*二、FrameBuffer在Linux中的实现和机制\* Framebuffer对应的源文件在linux/drivers/video/目录下。总的抽象设备文件为fbcon.c，在这个目录下还有与各种显卡驱动相关的源文件。  (一)、分析Framebuffer设备驱动      需要特别提出的是在INTEL平台上，老式的VESA1.2卡，如CGA/EGA卡，是不能支持Framebuffer的，因为Framebuffer要求显卡支持线性帧缓冲，即CPU可以访问显缓冲中的每一位， 但是VESA1.2 卡只能允许CPU一次访问64K的地址空间。  FrameBuffer设备驱动基于如下两个文件：  1) linux/include/linux/fb.h  2) linux/drivers/video/fbmem.c 下面分析这两个文件。  1、fb.h     几乎主要的结构都是在这个中文件定义的。这些结构包括：  1）fb\_var\_screeninfo     这个结构描述了显示卡的特性：  struct fb\_var\_screeninfo  {  \_\_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; /\* guess what \*/  \_\_u32 grayscale; /\* != 0 Gray levels 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 \*/  \_\_u32 nonstd; /\* != 0 Non standard pixel format \*/  \_\_u32 activate; /\* see FB\_ACTIVATE\_\* \*/  \_\_u32 height; /\* height of picture in mm \*/  \_\_u32 width; /\* width of picture in mm \*/  \_\_u32 accel\_flags; /\* acceleration flags (hints) \*/  /\* Timing: All values in pixclocks, except pixclock (of course) \*/  \_\_u32 pixclock; /\* pixel clock in ps (pico seconds) \*/  \_\_u32 left\_margin; /\* time from sync to picture \*/  \_\_u32 right\_margin; /\* time from picture to sync \*/  \_\_u32 upper\_margin; /\* time from sync to picture \*/  \_\_u32 lower\_margin;  \_\_u32 hsync\_len; /\* length of horizontal sync \*/  \_\_u32 vsync\_len; /\* length of vertical sync \*/  \_\_u32 sync; /\* see FB\_SYNC\_\* \*/  \_\_u32 vmode; /\* see FB\_VMODE\_\* \*/  \_\_u32 reserved[6]; /\* Reserved for future compatibility \*/  };    2) fb\_fix\_screeninfon  这个结构在显卡被设定模式后创建，它描述显示卡的属性，并且系统运行时不能被修改；比如FrameBuffer内存的起始地址。它依赖于被设定的模式，当一个模 式被设定后，内存信息由显示卡硬件给出，内存的位置等信息就不可以修改。  struct fb\_fix\_screeninfo {  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\_\* \*/  \_\_u16 xpanstep; /\* zero if no hardware panning \*/  \_\_u16 ypanstep; /\* zero if no hardware panning \*/  \_\_u16 ywrapstep; /\* zero if no hardware ywrap \*/  \_\_u32 line\_length; /\* length of a line in bytes \*/  unsigned long mmio\_start; /\* Start of Memory Mapped I/O \*/  /\* (physical address) \*/  \_\_u32 mmio\_len; /\* Length of Memory Mapped I/O \*/  \_\_u32 accel; /\* Type of acceleration available \*/  \_\_u16 reserved[3]; /\* Reserved for future compatibility \*/  };    3) fb\_cmap  描述设备无关的颜色映射信息。可以通过FBIOGETCMAP 和 FBIOPUTCMAP 对应的ioctl操作设定或获取颜色映射信息.  struct fb\_cmap {  \_\_u32 start; /\* First entry \*/  \_\_u32 len; /\* Number of entries \*/  \_\_u16 \*red; /\* Red values \*/  \_\_u16 \*green;  \_\_u16 \*blue;  \_\_u16 \*transp; /\* transparency, can be NULL \*/  };    4) fb\_info  定义当显卡的当前状态；fb\_info结构仅在内核中可见，在这个结构中有一个fb\_ops指针， 指向驱动设备工作所需的函数集。  struct fb\_info {  char modename[40]; /\* default video mode \*/  kdev\_t node;  int flags;  int open; /\* Has this been open already ? \*/  #define FBINFO\_FLAG\_MODULE 1 /\* Low-level driver is a module \*/  struct fb\_var\_screeninfo var; /\* Current var \*/  struct fb\_fix\_screeninfo fix; /\* Current fix \*/  struct fb\_monspecs monspecs; /\* Current Monitor specs \*/  struct fb\_cmap cmap; /\* Current cmap \*/  struct fb\_ops \*fbops;  char \*screen\_base; /\* Virtual address \*/  struct display \*disp; /\* initial display variable \*/  struct vc\_data \*display\_fg; /\* Console visible on this display \*/  char fontname[40]; /\* default font name \*/  devfs\_handle\_t devfs\_handle; /\* Devfs handle for new name \*/  devfs\_handle\_t devfs\_lhandle; /\* Devfs handle for compat. symlink \*/  int (\*changevar)(int); /\* tell console var has changed \*/  int (\*switch\_con)(int, struct fb\_info\*);  /\* tell fb to switch consoles \*/  int (\*updatevar)(int, struct fb\_info\*);  /\* tell fb to update the vars \*/  void (\*blank)(int, struct fb\_info\*); /\* tell fb to (un)blank the screen \*/  /\* arg = 0: unblank \*/  /\* arg > 0: VESA level (arg-1) \*/  void \*pseudo\_palette; /\* Fake palette of 16 colors and  the cursor's color for non  palette mode \*/  /\* From here on everything is device dependent \*/  void \*par;  };    5) struct fb\_ops  用户应用可以使用ioctl()系统调用来操作设备，这个结构就是用一支持ioctl()的这些操作的。  struct fb\_ops {  /\* open/release and usage marking \*/  struct module \*owner;  int (\*fb\_open)(struct fb\_info \*info, int user);  int (\*fb\_release)(struct fb\_info \*info, int user);  /\* get non settable parameters \*/  int (\*fb\_get\_fix)(struct fb\_fix\_screeninfo \*fix, int con,  struct fb\_info \*info);  /\* get settable parameters \*/  int (\*fb\_get\_var)(struct fb\_var\_screeninfo \*var, int con,  struct fb\_info \*info);  /\* set settable parameters \*/  int (\*fb\_set\_var)(struct fb\_var\_screeninfo \*var, int con,  struct fb\_info \*info);  /\* get colormap \*/  int (\*fb\_get\_cmap)(struct fb\_cmap \*cmap, int kspc, int con,  struct fb\_info \*info);  /\* set colormap \*/  int (\*fb\_set\_cmap)(struct fb\_cmap \*cmap, int kspc, int con,  struct fb\_info \*info);  /\* pan display (optional) \*/  int (\*fb\_pan\_display)(struct fb\_var\_screeninfo \*var, int con,  struct fb\_info \*info);  /\* perform fb specific ioctl (optional) \*/  int (\*fb\_ioctl)(struct inode \*inode, struct file \*file, unsigned int cmd,  unsigned long arg, int con, struct fb\_info \*info);  /\* perform fb specific mmap \*/  int (\*fb\_mmap)(struct fb\_info \*info, struct file \*file, struct  vm\_area\_struct \*vma);  /\* switch to/from raster image mode \*/  int (\*fb\_rasterimg)(struct fb\_info \*info, int start);  };    6) structure map  struct fb\_info\_gen | struct fb\_info | fb\_var\_screeninfo  | | fb\_fix\_screeninfo  | | fb\_cmap  | | modename[40]  | | fb\_ops ---|--->ops on var  | | ... | fb\_open  | | | fb\_release  | | | fb\_ioctl  | | | fb\_mmap  | struct fbgen\_hwswitch -|-> detect  | | encode\_fix  | | encode\_var  | | decode\_fix  | | decode\_var  | | get\_var  | | set\_var  | | getcolreg  | | setcolreg  | | pan\_display  | | blank  | | set\_disp  [编排有点困难，第一行的第一条竖线和下面的第一列竖线对齐，第一行的第二条竖线和下面的第二列竖线对齐就可以了]  这个结构 fbgen\_hwswitch抽象了硬件的操作.虽然它不是必需的，但有时候很有用.  2、 fbmem.c  fbmem.c 处于Framebuffer设备驱动技术的中心位置.它为上层应用程序提供系统调用也为下一层的特定硬件驱动提供接口；那些底层硬件驱动需要用到这儿的接口来向 系统内核注册它们自己.  fbmem.c 为所有支持FrameBuffer的设备驱动提供了通用的接口，避免重复工作.  1) 全局变量  struct fb\_info \*registered\_fb[FB\_MAX];  int num\_registered\_fb;  这两变量记录了所有fb\_info 结构的实例，fb\_info 结构描述显卡的当前状态，所有设备对应的fb\_info结构都保存在这个数组中，当一个FrameBuffer设备驱动向系统注册自己时，其对应的fb\_info结构就会添加到这个结构中，同时num\_registered\_fb 为自动加1. static struct {  const char \*name;  int (\*init)(void);  int (\*setup)(void);  } fb\_drivers[] \_\_initdata= { ....};    如果FrameBuffer设备被静态链接到内核，其对应的入口就会添加到这个表中；如果是动态加载的，即使用insmod/rmmod,就不需要关心这个表。  static struct file\_operations fb\_ops ={  owner: THIS\_MODULE,  read: fb\_read,  write: fb\_write,  ioctl: fb\_ioctl,  mmap: fb\_mmap,  open: fb\_open,  release: fb\_release  };    这是一个提供给应用程序的接口.  2）fbmem.c 实现了如下函数.  register\_framebuffer(struct fb\_info \*fb\_info);  unregister\_framebuffer(struct fb\_info \*fb\_info);  这两个是提供给下层FrameBuffer设备驱动的接口，设备驱动通过这两函数向系统注册或注销自己。几乎底层设备驱动所要做的所有事情就是填充fb\_inf o结构然后向系统注册或注销它。  （二）一个LCD显示芯片的驱动实例      以Skeleton LCD控制器驱动为例，在LINUX中存有一个/fb/skeleton.c的skeleton的Framebuffer驱动程序，很简单，仅仅是填充了fb\_info结构，并且注册/注销自己。设备驱动是向用户程序提供系统调用接口，所以我们需要实现底层硬件操作并且定义file\_operations结构来向系统提供系统调用接口，从而实现更有效的LCD控制器驱动程序。  1）在系统内存中分配显存  在fbmem.c文件中可以看到,file\_operations结构中的open()和release()操作不需底层支持，但read()、write()和 mmap()操作需要函数fb\_get\_fix()的支持.因此需要重新实现函数fb\_get\_fix()。另外还需要在系统内存中分配显存空间，大多数的LCD控制器都没有自己的显存空间，被分配的地址空间的起 始地址与长度将会被填充到fb\_fix\_screeninfo结构的smem\_start 和smem\_len 的两个变量中.被分配的空间必须是物理连续的。  2）实现 fb\_ops 中的函数  用户应用程序通过ioctl()系统调用操作硬件，fb\_ops 中的函数就用于支持这些操作。（注： fb\_ops结构与file\_operations结构不同，fb\_ops是底层操作的抽象,而file\_operations是提供给上层系统调用的接口，可以直接调用ioctl()系统调用在文件fbmem.c中实现，通过观察可以发现ioctl()命令与fb\_ops's 中函数的关系:  FBIOGET\_VSCREENINFO fb\_get\_var  FBIOPUT\_VSCREENINFO fb\_set\_var  FBIOGET\_FSCREENINFO fb\_get\_fix  FBIOPUTCMAP fb\_set\_cmap  FBIOGETCMAP fb\_get\_cmap  FBIOPAN\_DISPLAY fb\_pan\_display 如果我们定义了fb\_XXX\_XXX 方法，用户程序就可以使用FBIOXXXX宏的ioctl()操作来操作硬件。  文件linux/drivers/video/fbgen.c或者linux/drivers/video目录下的其它设备驱动是比较好的参考资料。在所有的这 些函数中fb\_set\_var()是最重要的，它用于设定显示卡的模式和其它属性，下面是函数fb\_set\_var()的执行步骤：  1)检测是否必须设定模式  2)设定模式  3)设定颜色映射  4) 根据以前的设定重新设置LCD控制器的各寄存器。  第四步表明了底层操作到底放置在何处。在系统内存中分配显存后，显存的起始地址及长度将被设定到LCD控制器的各寄存器中（一般通过fb\_set\_var()函数），显存中的内容将自动被LCD控制器输出到屏幕上。另一方面，用户程序通过函数mmap()将显存映射到用户进程地址空间中，然后用户进程向映射空间发送 的所有数据都将会被显示到LCD显示器上。 | |