通过 USB 总线实现可控制的点对点通信*

谢　兵

(重庆交通大学 信息科学与工程学院,重庆　400074)

摘要:通过对 USB 驱动程序的修改,实现了在应用层可控制的、USB device 端到 USB host 端的点到点数据传输. 基于 Linux 2.6.18 内核,重点描述了 USB 驱动程序部分的设计和实现.

关键词:USB;点对点通信;驱动;Linux

中图分类号:TP336　　**文献标识码**:A　　**文章编号**:1006 - 0707(2009)12 - 0049 - 03

1　Linux USB 驱动层次结构

Linux 内核支持 2 种主要类型的 USB 驱动程序:宿主(host)系统上的驱动程序和设备(device)上的驱动程序. 从宿主的观点来看(1 个普通的 USB 宿主是 1 个桌面计算机),宿主系统的 USB 驱动程序控制插入其中的 USB 设备,而 USB 设备的驱动程序则控制该设备如何作为 1 个 USB 设备与主机通信.

图 1 为 USB 总体结构. 从图中可以看出,USB 驱动程序存在于不同的内核子系统和 USB 硬件控制器之中. USB 核心为 USB 驱动程序提供了 1 个用于访问和控制 USB 硬件的接口,而不必考虑系统当前存在的各种不同类型的 USB 硬件控制器.

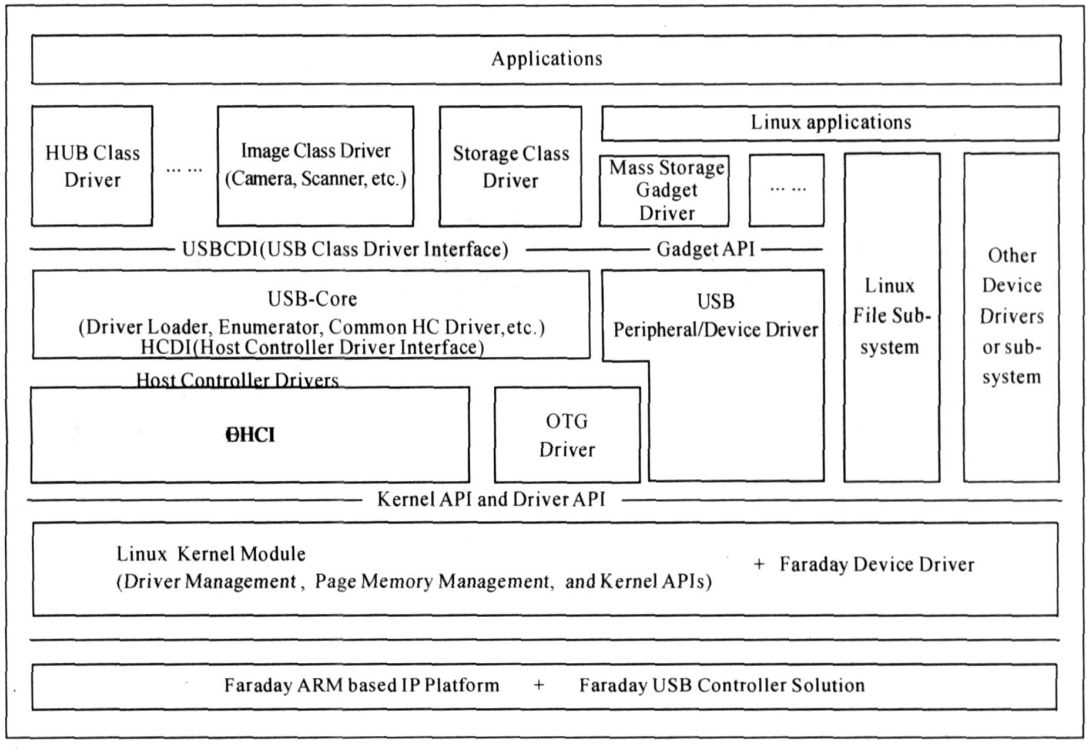

图 1　USB 总体结构

* **收稿日期**:2009 - 10 - 20

作者简介:谢兵(1980 —),男,重庆人,硕士,高级工程师,主要从事嵌入式系统及驱动研究.

2 实现思路

由 USB 总体结构图可以看出,USB device 端到 USB host 端的点到点数据传输驱动程序开发主要集中在 class driver 这 1 层,即在 USB-Core 和 Applications 之间的这 1 层. Linux USB 子系统提供了 1 个 USB host 端的框架驱动程序,即 Usb-skeleton.c 文件. 通过对这个文件做适当的增减即可实现 USB host 端的驱动程序. 对于 USB device 端的驱动可通过修改 Mass Storage Gadget Driver 来实现,其在 USB 协议栈中的位置见图 1,具体修改的文件是 File_storage.c.

USB 最基本的通信通过端点(endpoint)来实现. USB 端点只能往一个方向传送数据,从主机到设备(称为输出端点)或从设备到主机(称为输入端点). 通常的 USB device 端包含 3 个端点,分别是控制端点、输入端点和输出端点. 根据要实现的功能,可用控制端点来进行参数等少量数据的传输,而用输入端点实现纯数据流的传输. 通过 USB host 端向控制端点发送数据,USB device 端在控制端点上接收数据就可实现参数的传输. 同理,通过 USB device 端向输入端点发送数据,USB host 端在输入端点上接收数据就可实现纯数据流的传输.

3 具体实现过程及相关文件的主要修改点

3.1 USB host 端 Usb-skeleton.c 文件的修改

修改下面代码的宏定义值,使之与 File_storage.c 中的定义一致,这样 USB host 端就可以识别 USB device 端. 其相关程序代码如下.

```
# define USB_SKEL_VENDOR_ID 0x2310
# define USB_SKEL_PRODUCT_ID 0x6688
/ * table of devices that work with this driver */
static struct usb_device_id skel_table [ ] = {
{ USB_DEVICE(USB_SKEL_VENDOR_ID,USB_SKEL_PRODUCT_ID) },
{ } / * Terminating entry */
};
```

在 skel_read() 函数中调用下面的函数在输入端点读取数据,可以设置适当的超时值. 其相关程序代码如下.

```
/ * do a blocking bulk read to get data from the device */
retval = usb_bulk_msg(dev->udev,
usb_rcvbulkpipe(dev->udev,
dev->bulk_in_endpointAddr),
dev->bulk_in_buffer,
min(dev->bulk_in_size,count),
&bytes_read,1);//10000
/ * if the read was successful,
copy the data to userspace */
if (!retval) {
if (copy_to_user(buffer,dev->bulk_in_buffer,
bytes_read))
retval = -EFAULT;
else
retval = bytes_read;
}
```

根据应用需要可增加和修改 static int skel_ioctl(struct inode *ioctl,struct file *file,unsigned int cmd,unsigned long arg) 函数,在用户空间通过调用标准 ioctl() 函数,来实现 host 端向 device 端的数据传输.

通过上面的简单修改,就有了 1 个定制的 USB host 端驱动程序,并可根据实际的应用需要来进一步完善. skel_ioctl() 和 skel_read() 就是开放给用户空间的 2 个接口函数. 将 Usb-skeleton.c 文件编译成模块 usb-skeletong.ko,用 insmod usb-skeletong.ko 命令装载驱动,并用 mknod /dev/usb-skel c 180 192 建立设备文件,这样在用户空间就可以使用 open、read、ioctl 等标准函数来访问 USB device.

3.2 USB device 端 File_storage.c 文件的修改

首先将 USB device 修改为定制的设备,通过修改下面的数据结构来实现(主要关注修改点).

```
static struct {
int num_filenames;
int num_ros;
unsigned int nluns;
int removable;
int can_stall;
char *transport_parm;
char *protocol_parm;
unsigned short vendor;
unsigned short product;
unsigned short release;
unsigned int buflen;
int transport_type;
char *transport_name;
int protocol_type;
char *protocol_name;
} mod_data = { // Default values
.transport_parm = "BBB",
.protocol_parm = "private",/ *修改点:原来为"SCSI",现在将标准 SCSI 改为自定义"private"*/
# ifdef CONFIG_GM_USB_DEVICE
.removable = 1,
.can_stall = 0,
# else
.removable = 0,
.can_stall = 1,
# endif
.vendor = DRIVER_VENDOR_ID,
.product = DRIVER_PRODUCT_ID,
.release = 0xffff,// Use controller chiptype
.buflen = 16384,
};
```

1) 在控制端点接收参数的函数.

在函数 standard_setup_req() 中可接收来自 USB host 端的参数,在此函数中可根据需要添加自己的私有协议.

2) 在数据输入端点输入数据的函数.

用函数 start_transfer() 向 USB host 端发送数据,通过前

面介绍的函数,USB host 端就能接收到数据.

3) 为 USB device 提供 1 套内核空间和用户空间的通讯接口.

使用下面的函数动态注册 device 设备和访问该设备的 1 套接口,其中 fsg_driver_open、fsg_driver_read、fsg_driver_write 等函数根据实际应用需要进行修改.

```
struct file_operations scull_fops = {
.owner = THIS_MODULE,
.open = fsg_driver_open,
.read = fsg_driver_read,
.write = fsg_driver_write,
};
result = register_chrdev(fsg_driver_major,"fsgDrivertst",
&scull_fops);
if (result < 0)
    return result;
if (fsg_driver_major == 0)
    fsg_driver_major = result; /* dynamic */
return 0;
```

3.3 编译 USB device 驱动模块

在虚拟机开发环境中,进入目录/usr/src/arm-linux-2.6/linux-2.6.14-fa,在此目录下直接运行命令 make modules,将驱动模块 drivers/usb/gadget/g_file_storage.ko 提取到 FIE8180 开发板中,用 insmod g_file_storage.ko 命令加载驱动,用 cat /proc/devices 会看到如下信息.

```
Character devices:
1 mem
2 pty
3 ttyp
4 ttyS
5 /dev/tty
5 /dev/console
5 /dev/ptmx
10 misc
13 input
14 sound
29 fb
81 video4linux
89 i2c
90 mtd
108 ppp
116 alsa
128 ptm
136 pts
180 usb
189 usb_device
254 fsgDrivertst
```

其中,最后 1 条 254 fsgDrivertst 就是上面注册的 USB device 设备,使用命令 mknod /dev/usbdtst c 254 0 来为 USB device 设备创建 1 个设备文件,这样就可在用户空间使用 read、write 等标准函数来访问 USB device 设备了.

4 实际测试平台环境和测试代码

采用 FIE8180 评估板及配套的 linux SDK 开发包,作为 USB device 设备端;USB host 端使用 PC 平台,在其上安装 Vmware linux 虚拟机;USB device 设备端采用 mini-B usb 接口,host 端就用 PC 机的 USB 接口.

USB host 端测试程序如下.

```
int main()
{
    char fname[50];
    int fd;
    char buf[1024];
    int i;
    sprintf(fname,"/dev/usbskel");
    fd = open(fname,O_RDWR,0);
    if (fd == -1)
    {
      printf("open file: %s fail\n",fname);
      exit(1);
    }
    memset(buf,0,1024);
    for (i=0; i<3000; i++)
    {
      read(fd,buf,50);
      printf("read data: %s\n",buf);
    }
}
```

USB device 端测试程序如下.

```
int main()
{
    char fname[50];
    int fd;
    char buf[1024];
    int i;

    sprintf(fname,"/dev/usbdtst");

    fd = open(fname,O_RDWR,0);
    if (fd == -1)
    {
      printf("open file: %s fail\n",fname);
      exit(1);
    }
    memset(buf,0,1024);
    for (i=0; i<3000; i++)
    {
      write(fd,msg_info,msg_len);
      sleep(1);
    }
    return 0;

    }
```

(下转第 71 页)

5 试验技术对比

1) HALT和DVT

DVT(Design Verification Testing)设计验证试验,其目的是验证装备是否满足技术要求的试验,装备必须无故障地通过试验才能被认可. HALT的目的则是通过"过应力"快速地诱发装备故障,从而暴露出装备潜在的故障模式以及最薄弱的设计环节. 最终,通过改进这些环节达到提高装备可靠性的目的.

2) HALT和AST

AST加速应力试验,按S-N原理使用强应力进行试验,并通过长时间试验来计算装备寿命,是HALT技术的前驱和理论基础. 应用在设计和早期制造阶段. HALT目的是找出装备的Margin和计算装备寿命期,因此使用应力可能会使样品受到不可恢复性的故障,但却缩短了试验时间并得到了更多的故障数据,如无损害的应力范围、各类失效模式等.

3) HALT和FMECA

FMECA是一种"纸上谈兵"的强有力的辅助分析手段. 目的是让设计师自我剖析,通过对装备潜在故障模式影响及危害的分析,从而对危害大故障率高的故障模式加以预防或改进(试验前应进行). HALT则是利用试验手段将装备中的潜在故障模式实实在在地暴露出来. 特别是在应力加大过程中"顺序逐个"地暴露. 为故障分析提供的有力的支持.

4) HALT和可靠性预计

可靠性预计是利用元器件的各种影响其工作的因子,结合Arrhenius模型计算装备的故障发生概率. 定量地掌握了装备故障的分布情况. HALT依靠试验来发现装备的故障,并结合相应的理论推算装备的寿命和可靠性(更加真实),且能明确装备各种故障模式发生的机理和次序.

5) HASS和Burn-in

Burn-in热老化试验,采用热应力作用在装备上并辅助以电应力对装备进行老化筛选. 目的:为去除装备的早期故障. HASS利用多种综合的应力施加在装备上进行筛选,目的:去除装备生产过程中引入的缺陷和早期故障,并使装备迅速地进入偶发(随机)故障区.

6) HASS和ESS

ESS环境应力筛选. 利用装备的技术要求应力值,通过快速而多次地温度循环、振动等使装备进入偶发(随机)故障区,并剔除装备生产过程中引入的缺陷. HASS则利用HALT试验结果,采用"极限"应力(非破坏性质)来进行高强度筛选,一般进行一到两个循环即可将生产过程中引入的缺陷剔除掉.

6 结束语

HALT和HASS技术是20世纪90年代由美国Gregg K. Hobbs博士所提出的试验技术,并快速而广泛地被应用在商业、工业以及国防的设计制造领域. 其发展速度之快,以至于现在发达国家的设计制造业被要求必须掌握这一技术.

但由于不同装备有不同的工作极限和损坏极限,因此,对所有装备不存在统一的HALT和HASS应力. 因此,它还不能形成标准,只能作为指导. 不同装备筛选的应力类型是不一样的,只能根据装备各自的特点,选择装备最敏感的应力进行筛选,希望对现行的试验技术能有所启示.

参考文献:

[1] Gregg K. Hobbs, Ph. D., P. E. What HALT and HASS Can Do For Your Products[Z]. [S.l.]:[s.n.],2000.

[2] Acme Electronics, LLC. Highly Accelerated Life Testing (HALT) and Highly Accelerated Stress Screening (HASS) [Z]. [S.l.]:[s.n.],2000.

[3] Neill Doertenbach. Highly Accelerated Life Testing Testing With a Different Purpose[z]. [S.l.]:[s.n.],2000.

[4] Christine Hans. A beginner's guide to HALT[Z]. [S.l.]:[s.n.],2000.

[5] RADC-TR-88-110 不工作状态的可靠性、维修性、测试性设计.

(上接第51页)

参考文献:

[1] JONATHAN CORBET, ALESSANDRO RUBINI, GREG KROAH. HARTMAN. Linux设备驱动程序[M]. 3版. 魏永明,耿岳,钟书毅,译. 北京:中国电力出版社,2006.

[2] BACH MAURICE J. UNIX操作系统设计[M]. 陈葆珏,王旭,柳纯录,等,译. 北京:机械工业出版社,2007.