嵌入式Linux内核及其驱动开发

张海峰 *

(厦门大学，厦门， 361005)

摘 要 本文以在Intel StrongArm SA1110硬件环境中开发驱动程序为出发点，阐述了嵌入式LINUX驱动开发的关键技术。然后以 USB 、 GPIO 驱动程序为例描述了嵌入式LINUX下字符设备驱动程序的开发。

关键词 嵌入式LINUX USB GPIO ARM StrongArm SA1110 驱动程序开发

Linux起源于Internet，最初的核心程序是由芬兰赫尔辛基的大学生 Linus Torvalds 编写的。在1993年Linus把这一软件奉献给了自由软件基金会(FSF)的GNU计划，从此Linux在全世界计算机高手的扶持下成为计算机领域中任何人都无法忽视的力量。

本文是我在完成实时嵌入式操作系统UC/OS II驱动程序的开发后提出的又一篇有关嵌入式操作系统驱动程序开发的文章。USB 、GPIO是在StrongArm Sa1110开发环境下的驱动程序。该文章的提出希望能给一些想从事嵌入式Linux驱动程序开发的人员一点启发，同时也希望有共同爱好的开发者能一起探讨，以此发展中国的自由软件事业。

1 嵌入式Linux内核配置编译

Linux作为一个自由软件，在广大爱好者的支持下，内核版本不断更新。新的内核修订了旧内核的bug，并增加了许多新的特性。如果用户想要使用这些新特性，或想根据自己的系统定制一个更高效、更稳定的内核，就需要重新编译内核。为了正确、合理地设置内核编译配置选项，从而只编译系统需要的功能的代码，一般需要作下面四点考虑：

- 开发者可以根据自己产品的需求定制内核功能模块；
- 系统将拥有更多的内存；
- 将不需要的功能编译进入内核会增加系统软硬件的开销和降低系统的稳定性；
- 将某种功能编译为模块方式会比编译到内核方式速度要慢一些。

有了内核配置编译的基本指导方法还不够，我们需要详细了解自己产品的硬件配置和详细的硬件产品说明书。清楚硬件配置后，我们才可以开始配置内核。

下面就以Intel StrongArm SA1110开发板为例简单说明硬件构成，如图1所示。

(1) CPU

采用Intel的StrongARM SA1110处理器可运行于133MHz/206MHz(StrongARM core)

(2) 外设（与本篇文章无关的外设省略）

- 采用CS8900A芯片支持10-BaseT以太网接口；
- USB(Slave)接口(SA1110芯片集成)；
- 多个LED指示灯(在写GPIO驱动程序的时候使用)。

更详细的硬件信息可以查阅相关芯片资料，这里不再赘述。

2 驱动程序开发关键技术

Linux系统的设备分为字符设备、块设备和网络设备三种。字符设备是指存取时具有较少

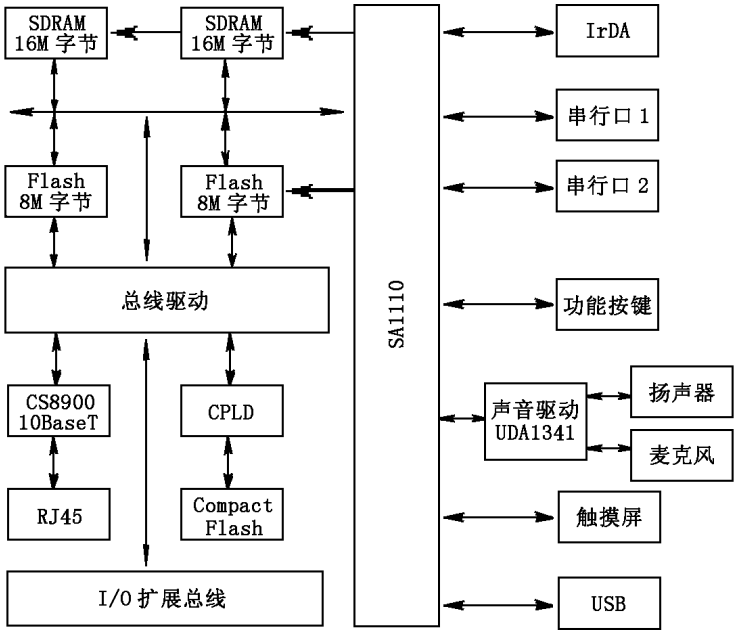

图 1 Intel StrongArm SA1110 开发板

缓存的设备。块设备的读写都由缓存来支持，并且块设备必须能够随机存取，字符设备则没有这个要求。

设备驱动程序必须向 Linux 核心或者它所在的子系统提供一个标准的接口。例如， USB 驱动程序向 Linux 核心提供了一个设备文件 I/O 接口， GPIO 设备驱动程序向 GPIO 子系统提供了 GPIO 设备接口。接着，向核心提供了文件 I/O 和缓冲区的接口。

设备驱动程序使用标准的核心服务例如内存分配、中断转发和等待队列来完成工作。大多数设备驱动程序可以在需要的时候作为核心模块加载，在不再需要的时候卸载。这使得核心对于系统资源非常具有适应性和效率。

2.1 核心数据结构

操作系统必须记录关于系统当前状态的许多信息。如果系统中发生了事情，这些数据结构就必须相应改变以反映当前的实际情况。这样的数据结构多数在物理内存中，而且只能由核心和它的子系统访问。数据结构包括数据和指针。理解 Linux 核心的关键在于理解它的数据结构和核心在处理这些数据结构时所用到的大量函数。

下面列举在几个设备驱动程序的开发过程中经常用到的几个数据结构。

- struct inode /* 设备文件的入口点，记录了文件的属性等信息 */
- struct file
- struct fileoperations
- struct usb_device
- struct usb_driver
- struct usb_fingerhid（自定义 USB 设备数据结构）

```
struct usb_fingerhid {
    struct usb_device *fingerhid_dev;
    unsigned int ifnum;              /* USB 设备的接口值 */
    kdev_t fingerhid_minor;          /* USB 设备的子设备号 */
    char isopen;                     /* 设备打开标记 */
    char present;                    /* 设备准备标记 */
    char *obuf, *ibuf;               /* 传输缓冲区 */
```

```
        char bulk_in_ep[2], bulk_out_ep[2];      /* 节点标记 */
        wait_queue_head_t rd_wait_q;             /* 读超时标记 */
        struct semaphore sem;                    /* 读写信号量, 实现对当前缓冲区读写访问加锁 */
        unsigned int rd_nak_timeout;             /* 读超时设定 */
};
• struct FingerPrint_Device(自定义 GPIO 设备数据结构)
  struct FingerPrint_Device {
        kdev_t FingerPrint_minor;                /* 子设备号 */
        char isopen;                             /* 设备打开标记 */
        char present;                            /* 设备准备标记 */
        char *obuf, *ibuf;                       /* 传输缓冲区 */
        struct semaphore sem;                    /* 读写信号量 */
};
• struct usb_device_id                           /* 设备的生产商和设备号 */
  static struct usb_device_id usb_fingerhid_id_table [ ] = {
        { USB_DEVICE(0x0471, 0x1001) },
        ...
        { }
};
```

2.2 字符设备

字符设备是 Linux 最简单的设备，可以像文件一样访问。应用程序使用标准系统调用打开、读取、写和关闭，这个设备就像是一个普通文件。当字符设备初始化的时候，它的设备驱动程序向 Linux 核心登记，在 chrdevs 向量表中增加一个 device_struct 数据结构条目。这个设备的主设备标识符 (例如对于 USB 设备是 180) 用作这个向量表的索引。一个设备的主设备标识符是固定的。 Chrdevs 向量表中的 device_struct 数据结构包括两个元素：一个是登记的设备驱动程序的名称的指针，另一个是指向一组文件操作的指针。这块文件操作本身位于这个设备的字符设备驱动程序中，每一个都处理特定的文件操作比如打开、读、写和关闭。 /proc/devices 中字符设备的内容来自 chrdevs 向量表。

和普通文件或目录一样，每一个设备特殊文件都用 VFS I 节点表达。这个字符特殊文件的 VFS inode(实际上所有的设备特殊文件) 都包括设备的 major 和 minor 标识符。这个 VFS I 节点由底层的文件系统 (例如 EXT2) 在查找这个设备特殊文件的时候根据实际的文件系统创建。

每一个 VFS I 节点都联系着一组文件操作，依赖于 I 节点所代表的文件系统对象不同而不同。不管代表一个字符特殊文件的 VFS I 节点什么时候创建，它的文件操作被设置成字符设备的缺省操作。这只有一种文件操作： open 操作。当一个应用程序打开这个字符特殊文件的时候，通用的 open 文件操作使用设备的主设备标识符作为 chrdevs 向量表中的索引，取出这种特殊设备的文件操作块。它也建立描述这个字符特殊文件的 file 数据结构，让它的文件操作指向设备驱动程序中的操作。然后应用程序所有的文件系统操作都被映射到字符设备的文件操作。

2.3 设备文件系统 (Devfs)

Devfs 也叫设备文件系统，设计它的唯一目的就是提供一个新的 (更理性的) 方式，以管理通常位于 /dev 的所有块设备和字符设备。由于 devfs 技术还在发展阶段，这里给出实现函数。

Devfs 技术注册设备的时候克服了系统设备号的限制，采用名称注册的方法注册系统设备，驱动程序还可以制定设备名、所有者和权限位，而用户空间程序仍可以修改所有者和权限位。在系统启动设备初始化时，操作系统自动在 /dev 目录下创建相应名称的设备入口点，移除设备时将它删除。设备文件系统的调用函数是 devfs_register() 、 devfs_unregister() 。

USB 、GPIO 驱动的开发过程中，我分别采用 devfs 技术和“原来”的设备注册技术实现。

用户程序访问设备的整体工作情况如图 2 所示。上面的数据结构是在内核态工作，而内核通过对相应数据结构的赋值，以此记录了用户程序对外部设备的使用情况。 Inode 数据结构工作在 VFS 阶段，其定位了用户程序访问的设备文件，并根据相应文件属性满足用户程序对该设备文件的访问。在对设备数据的交换过程中， file 数据结构维护着缓冲区的数据。 Fileoperations 数据结构维护着设备与文件系统的接口。 usb_device 、 usb_fingerhid 数据结构维护着 USB 设备的枚举和地址分配。这些数据结构的有机结合完成了高层应用程序对设备的访问。

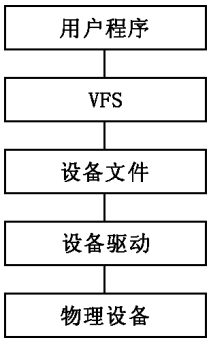

图 2　文件层次结构

3 USB 、 GPIO 设备驱动程序开发

本篇文章中提到的 USB 、GPIO 驱动可以归类为 Linux 设备驱动的字符设备驱动。所以，在介绍具体的字符设备开发方法的过程中，可以将两个不同设备驱动放在一起阐述，这样读者就能更容易地理解字符设备驱动的关键技术。

3.1 模块化驱动程序

不失 Linux 驱动开发一般性，在写字符设备的驱动程序的时候，也要遵守模块化编程的一般规范。设备模块在用户空间的初始化和终止：

Init_module() 向内核注册模块提供数据结构、局部和全局变量。

Cleanup_module() 取消所有 init_module 在内核中的注册。

3.2 设备模块在内核空间的内存申请和释放

Kmalloc() 函数分配一段内存，这样也就实现了 Chrdevs 向量表中指向设备驱动程序名称的指针。使用 kfree 释放内存。

fingerhid = kmalloc(sizeof(struct usb_fingerhid), GFP_KERNEL)

kfree(fingerhid);

3.3 字符设备主设备号和子设备号的分配

主设备号标志设备对应的驱动程序，内核利用主设备号将设备与相应的驱动程序对应起来。主设备号的取值空间是 0 ~ 255 ，如果不善加规划的话，很容易造成主设备号的冲突。主设备号的分配有静态和动态之分，从我开发设备的角度来看，我推荐静态设备号的分配。

次设备号由驱动程序使用，内核的其它部分并不使用它，仅将它传递给驱动程序。一个驱动程序控制若干个设备，次设备号提供了一种区分它们的方法。

3.4 设备模块在内核空间的注册与注销

字符设备的注册有 2 种方法，一种是常用注册方法，通过系统函数 register_chrdev() 将设备加入到系统设备列表中；另外一种方式是 devfs 技术，通过系统函数 devfs_register() 实现设备的注册。注销与注册相反，分别调用 unregister_chrdev() 和 devfs_unregister() 。

3.5 设备模块在内核空间提供系统调用的函数

设备驱动程序在注册成功以后，用户可以通过访问设备特殊文件 (一般情况在 /dev 目录下) 实现系统调用。实现 USB 设备驱动程序系统调用函数的数据结构表示如下所示：

```
static struct usb_driver usb_fingerhid_driver = {
    name:        " FingerHID ",                  /* 驱动程序加载名称 */
    probe:       usb_fingerhid_probe,            /* 配置 USB 端口 */
```

```
    disconnect:   usb_fingerhid_disconnect,        /* USB 断开连接处理 */
    fops:         &usb_fingerhid_fops,             /* 提供文件系统接口数据结构 */
    minor:        FINGERHID_BASE_MNR,              /* 最小的子设备号 */
    id_table:     usb_fingerhid_id_table,          /* 设备 ID 识别码定义 */
};
static struct file_operations usb_fingerhid_fops = {
    owner:        THIS_MODULE,                     /* 该设备模块的所有者 */
    read:         read_fingerhid,                  /* 读设备的系统调用函数 */
    write:        write_fingerhid,                 /* 写设备的系统调用函数 */
    open:         open_fingerhid,                  /* 打开设备的系统调用函数 */
    release:      close_fingerhid,                 /* 关闭设备的系统调用函数 */
};
```

usb_fingerhid_fops 数据结构是驱动程序在注册的时候提供给文件系统调用的接口。usb_fingerhid_driver 数据结构是驱动程序提供给 USB 的一组调用函数。

因为 USB 是即插即用设备，所以在子设备插入到 USB 集线器中时，驱动程序便开始调用 usb_fingerhid_probe() 函数，以此实现 USB 设备的枚举、地址分配和配置。当 USB 移除集线器的时候，驱动程序调用 usb_fingerhid_disconnect() 函数处理移除设备的变量和数据结构的赋值。 open_fingerhid() 、 read_fingerhid() 、 write_fingerhid() 、 close_fingerhid() ，分别代表应用程序在使用标准系统调用打开、读取、写和关闭设备特殊文件的时候所对应的处理函数。

GPIO 驱动程序只需要一个 file_operations 数据结构体就可以了。这是因为 GPIO 本身是在设计阶段就定制好了接入的物理设备，接入到 GPIO 端口的物理设备并不需要即插即用。因此可以在开发阶段定制该接口的物理设备，也可以把这个接口的设备当做该产品的标配外设发布。这样，我们在编写驱动程序的过程中，只要写好应用程序使用标准系统调用打开、读取、写和关闭等的处理函数就可以完成驱动程序的开发。其中标准系统调用处理函数在所有的字符设备里面具有同样的功能。

3.6 内核中的并发处理

由于驱动程序是运行在内核系统态、特权级，因此编写驱动程序的时候需要考虑进程并发处理。当一个进程请求内核驱动程序模块服务时，如果此时内核模块正在忙，那么可以将进程放入睡眠状态直到驱动程序模块空闲。

在一个运行的进程中用到的设备驱动程序必须满足时间片分配、内存分配管理等原则，因此必须引入并发处理的方法。 Linux 并发处理方法调用宏函数如下：

```
DECLARE_MUTEX (fingerhid_mutex);
/* 在驱动程序初始化时，打开进程访问控制权 */
```

在对设备特殊文件和驱动程序设备数据结构访问的时候，驱动程序需要对这些访问做并发处理。独占和释放设备驱动程序控制权的调用宏函数如下：

```
down(&fingerhid_mutex); /* 锁定设备驱动程序访问 */
up(&fingerhid_mutex);   /* 打开设备驱动程序访问权 */
```

通过 down() 、 up() 两个宏函数的调用实现多个进程同时运行的时候只有一个进程独占访问设备驱动程序。下面以 open_fingerhid() 中获得子设备号为例说明：

当几个用户进程同时打开同一个主设备号的几个设备特殊文件时，系统会根据主设备号注册的设备驱动程序首先调用函数 open_fingerhid() 。在取得子设备号之前通过调用 down() 宏函数独占访问。取得子设备号后调用 up() 恢复设备驱动程序的访问权。实现代码如下所示：

```
down(&fingerhid_mutex);          /* 独占访问设备驱动程序 */
```

```
fingerhid_minor = USB_FINGERHID_MINOR(inode);   /* 取得子设备号 */
fingerhid = p_fingerhid_table[fingerhid_minor];  /* 取得子设备号对应的设备数据结构 */
up(&fingerhid_mutex);                            /* 恢复其它进程对驱动程序的访问 */
```

3.7 设备模块驱动程序内部数据结构的互斥方法

信号量是一个可以用来控制多个进程存取共享资源的计数器。它经常作为一种锁定机制来防止当一个进程正在存取共享资源时，另一个进程也存取同一资源。Linux 实现驱动程序内部数据结构互斥，可以在设备数据结构中加入信号量数据结构体的定义。如下所示：

```
struct usb_fingerhid {
    ...
    struct semaphore sem;          /* 读写信号量，实现对当前缓冲区读写访问加锁 */
    ...
};
```

还是以 open_fingerhid() 函数为例来说明，如何实现保护设备数据结构体的内容修改不受其他进程的影响，这也就是我们所说的互斥方法。具体的实现代码如下：

```
down(&(fingerhid → sem));                    /* 保护 fingerhid 数据结构体不受其它进程干扰 */
init_waitqueue_head(&fingerhid → rd_wait_q);  /* 初始化等待时间 */
fingerhid → isopen = 1;                      /* 设备打开标志置位 */
file → private_data = fingerhid;             /* 为读写函数提供设备数据结构体的指针 */
up(&(fingerhid → sem));                      /* 释放对 fingerhid 数据结构体的独占访问 */
```

3.8 设备模块驱动程序常用宏定义

- 设备驱动程序模块设备信息的输出
  MODULE_DEVICE_TABLE (usb, usb_fingerhid_id_table);
- 在目标文件中添加关于模块的文档信息
  MODULE_AUTHOR(DRIVER_AUTHOR);
  MODULE_DESCRIPTION(DRIVER_DESC);
  MODULE_LICENSE("GPL");
- 模块加载变量调整宏定义
  MODULE_PARM(variable,type)
  MODULE_PARM_DESC(variable,description)

4 结束语

本篇文章是在我开发完 USB、GPIO 字符设备驱动程序后总结的。USB 驱动程序正确实现了指纹设备的数据采集，GPIO 驱动程序实现了 GPIO–17 端口控制的发光二极管状态变化。开发工作者通过对本文的理解可以减少工作量。

参考文献

[1] Alessandro Rubini & Jonathan Corbet, Linux 设备驱动程序，中国电力出版社． 2002. 11.

[2] 张念淮，江浩． USB 总线接口开发指南，国防工业出版社． 2001. 3.

[3] 周巍松． Linux 系统分析与高级编程技术.

[4] http://mailman.ucLinux.org/mailman/listinfo/ucLinux-dev, 嵌入式 Linux 论坛.

[5] Robert Baruch. Writing Character Device Driver for Linux, 1993.

[6] Ori Pomerantz. Linux Kernel Module Programming Guide, 1999.

[7] 毛德操，胡希德． Linux 内核源代码情景分析，浙江大学出版社． 2001. 5.

[8] Intel StrongArm SA1110 开发板资料 (www. intel. com 下载).

[9] Bbs.cst.sh.cn,bbs.tsinghua.edu.cn