《LINUX 设备驱动开发详解》

作者：华清远见

第 1 章　设备驱动概述

本章简介

本章将带您走进 Linux 设备驱动的精彩世界。

1.1 节讲解了设备驱动的概念和作用。

1.2 节和 1.3 节分别讲述无操作系统和有操作系统情况下设备驱动的设计方法，通过分析讲解设备驱动与硬件和操作系统的关系。

1.4 节对 Linux 操作系统的设备驱动进行了概要性的介绍，讲解设备驱动与系统软硬件的关系，分析了 Linux 设备驱动的重点难点和学习方法。

本章的最后给出了一个设备驱动的“ Hello World ”实例，即最简单的 LED 驱动在无操作系统情况下和 Linux 操作系统下的实现。

1.1 设备驱动的作用

任何一个计算机系统的运行都是系统中软硬件协作的结果，没有硬件的软件是空中楼阁，而没有软件的硬件则只是一堆废铁。硬件是底层基础，是所有软件得以运行的平台，代码最终会落实为硬件上的组合逻辑与时序逻辑；软件则实现了具体应用，它按照各种不同的业务需求而设计，满足了用户的需求。硬件较固定，软件则很灵活，可以适应各种复杂多变的应用。可以说，计算机系统的软硬件互相成就了对方。

但是，软硬件之间同样存在着悖论，那就是软件和硬件不应该互相渗透到对方的领地。为了尽可能快速地完成设计，应用软件工程师不想也不必关心硬件，而硬件工程师也难有足够的闲暇和能力来顾及软件。例如，应用软件工程师在调用套接字发送和接收数据包的时候，他不必关心网卡上的中断、寄存器、存储空间、I/O 端口、片选以及其他任何硬件词汇；在使用 printf()函数输出信息的时候，他不用知道底层究竟是怎样把相应的信息输出到屏幕或串口。

也就是说，应用软件工程师需要看到一个没有硬件的纯粹的软件世界，硬件必须被透明地呈现给他们。谁来实现硬件对应用软件工程师的隐形？这个艰巨的任务就落在了驱动工程师的头上。

对设备驱动最通俗的解释就是“驱使硬件设备行动”。设备驱动与底层硬件直接打交道，按照硬件设备的具体工作方式读写设备寄存器，完成设备的轮询、中断处理、DMA 通信，进行物理内存向虚拟内存的映射，最终使通信设备能够收发数据，使显示设备能够显示文字和画面，使存储设备能够记录文件和数据。

由此可见，设备驱动充当了硬件和应用软件之间的纽带，它使得应用软件只需要调用系统软件的应用编程接口（API）就可让硬件去完成要求的工作。在系统中没有操作系统的情况下，工程师可以根据硬件设备的特点自行定义接口，如对串口定义 SerialSend()、SerialRecv()；对 LED 定义 LightOn()、LightOff()；以及对 Flash 定义 FlashWrite()、FlashRead()等。而在有操作系统的情况下，设备驱动的架构则由相应的操作系统定义，驱动工程师必须按照相应的架构设计设备驱动，这样，设备驱动才能良好地整合到操作系统的内核中。

驱动程序沟通着硬件和应用软件，而驱动工程师则沟通着硬件工程师和应用软件工程师。随着通信、电子行业的迅速发展，全世界每天都会有大量的新芯片被生产，大量的新电路板被设计，因此，也会有大量设备驱动需要开发。这些设备驱动，或运行在简单的单任务环境中，或运行在 VxWorks、Linux、Windows 等多任务操作系统环境中，发挥着不可替代的作用。

1.2 无操作系统时的设备驱动

并不是任何一个计算机系统都一定要运行操作系统，在许多情况下操作系统是不必要的。对于功能比较单一、控制并不复杂的系统，如公交车刷卡机、电冰箱、微波炉、简单的手机和小灵通等，并不需要多任务调度、文件系统、内存管理等复杂功能，用单任务架构完全可以很好地支持它们的工作。一个无限循环中夹杂对设备中断的检测或者对设备的轮询是这种系统中软件的典型架构，如代码清单 1.1 所示。

代码清单 1.1　单任务软件典型架构

```
1  int main(int argc, char* argv[])
2  {
3    while (1)
4    {
5      if (serialInt == 1)
6      /*有串口中断*/
7      {
```

```
    ProcessSerialInt(); /*处理串口中断*/
    serialInt = 0;          /*中断标志变量清零*/
  }
  if (keyInt == 1)
  /*有按键中断*/
  {
    ProcessKeyInt();        /*处理按键中断*/
    keyInt = 0;         /*中断标志变量清零*/
  }
  status = CheckXXX();
  switch (status)
  {
    ...
  }
  ...
 }
}
```

在这样的系统中，虽然不存在操作系统，但是设备驱动是必须存在的。一般情况下，对每一种设备驱动都会定义为一个软件模块，包含.h 文件和.c 文件，前者定义该设备驱动的数据结构并声明外部函数，后者进行设备驱动的具体实现。代码清单 1.2 定义了一个串口的驱动。

代码清单 1.2　无操作系统情况下串口的驱动

```
/**********************
 *serial.h 文件
 **********************/
extern void SerialInit(void);
extern void SerialSend(const char buf*,int count);
extern void SerialRecv(char buf*,int count);

/**********************
 *serial.c 文件
 **********************/
/*初始化串口*/
void SerialInit(void)
{
 ...
}
/*串口发送*/
void SerialSend(const char buf*,int count)
{
 ...
}
/*串口接收*/
void SerialRecv(char buf*,int count)
{
 ...
}
/*串口中断处理函数*/
void SerialIsr(void)
{
 ...
 serialInt = 1;
}
```

其他模块需要使用这个设备的时候，只需要包含设备驱动的头文件 serial.h，然后调用其中的外部接口函数即可。如我们要从串口上发送字符串“Hello World”，使用函数 SerialSend("Hello World ",11)即可。

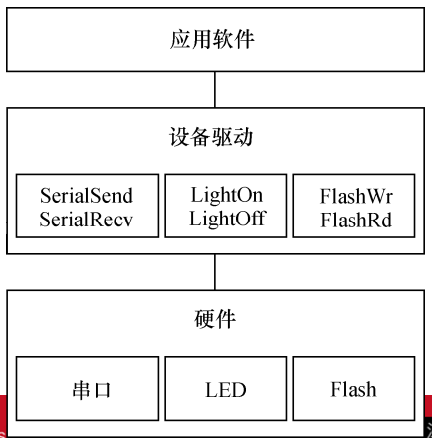

图 1.1　无操作系统时硬件、设备驱动和应用软件的关系

由此可见，在没有操作系统的情况下，设备驱动的接口被直接

提交给了应用软件工程师，应用软件没有跨越任何层次就直接访问了设备驱动的接口。设备驱动包含的接口函数也与硬件的功能直接吻合，没有任何附加功能。图 1.1 所示为无操作系统情况下硬件、设备驱动与应用软件的关系。

有的工程师把单任务系统设计成了如图 1.2 所示的结构，即设备驱动和具体的应用软件模块处于同一层次，这显然是不合理的，不符合软件设计中高内聚低耦合的要求。

另一种不合理的设计是直接在应用中操作硬件的寄存器，而不单独设计驱动模块，如图 1.3 所示。这种设计意味着系统中不存在或未能充分利用可被重用的驱动代码。

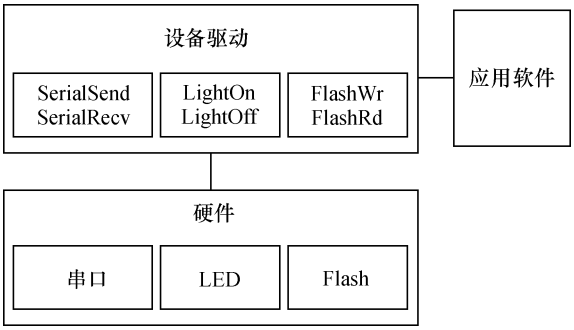

图 1.2　设备驱动与应用高耦合的不合理设计

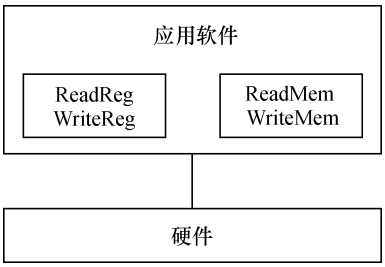

图 1.3　应用直接访问硬件的不合理设计

1.3 有操作系统时的设备驱动

1.2 节中的设备驱动直接运行在硬件之上，不与任何操作系统关联。当系统中包含操作系统后，设备驱动会变得怎样？

首先，无操作系统时设备驱动的硬件操作工作仍然是必不可少的，没有这一部分，设备驱动不可能与硬件打交道。

其次，我们还需要将设备驱动融入内核。为了实现这种融合，必须在所有的设备驱动中设计面向操作系统内核的接口，这样的接口由操作系统规定，对一类设备而言结构一致，独立于具体的设备。

由此可见，当系统中存在操作系统的时候，设备驱动变成了连接硬件和内核的桥梁。如图 1.4 所示，操作系统的存在势必要求设备驱动附加更多的代码和功能，把单一的“驱使硬件设备行动”变成了操作系统内与硬件交互的模块，它对外呈现为操作系统的 API，不再给应用软件工程师直接提供接口。

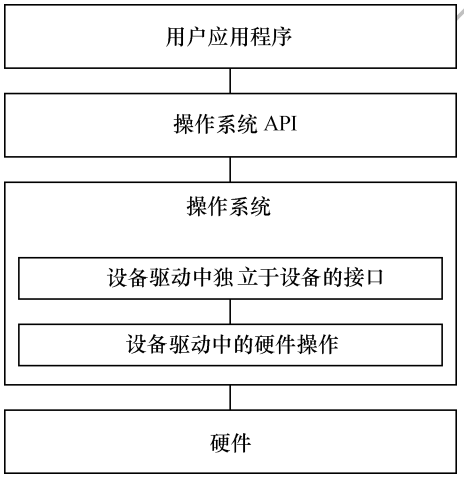

图 1.4　硬件、设备驱动、操作系统和应用程序的关系

有了操作系统之后，设备驱动反而变得复杂，那要操作系统干什么？

首先，一个复杂的软件系统需要处理多个并发的任务，没有操作系统，想完成多任务并发是很困难的。

其次，操作系统给我们提供内存管理机制。一个典型的例子是，对于多数含 MMU 的处理器而言，Windows、Linux 等操作系统可以让每个进程都独立地访问 4GB 的内存空间。

上述优点似乎并没有体现在设备驱动身上，操作系统的存在给设备驱动究竟带来了什么好处呢？

简而言之，操作系统通过给设备驱动制造麻烦来达到给上层应用提供便利的目的。如果设备驱动都按照操作系统给出的独立于设备的接口而设计，应用程序将可使用统一的系统调用接口来访问各种设备。对于类 UNIX 的 VxWorks、Linux 等操作系统而言，应用程序通过 write()、read()等函数读写文件就可以访问各种字符设备和块设备，而不用管设备的具体类型和工作方式，是非常方便的。

1.4 Linux 设备驱动

1.4.1 设备的分类及特点

计算机系统的硬件主要由 CPU、存储器和外设组成。随着 IC 制造工艺的发展，目前，芯片的集成度越来越高，往往在 CPU 内部就集成了存储器和外设适配器。ARM、PowerPC、MIPS 等处理器都集成了 UART、I^2C 控制器、USB 控制器、SDRAM 控制器等，有的处理器还集成了片内 RAM 和 Flash。

驱动针对的对象是存储器和外设（包括 CPU 内部集成的存储器和外设），而不是针对 CPU 核。Linux 将存储器和外设分为 3 个基础大类：

- 字符设备；
- 块设备；
- 网络设备。

字符设备指那些必须以串行顺序依次进行访问的设备，如触摸屏、磁带驱动器、鼠标等。块设备可以用任意顺序进行访问，以块为单位进行操作，如硬盘、软驱等。字符设备不经过系统的快速缓冲，而块设备经过系统的快速缓冲。但是，字符设备和块设备并没有明显的界限，如 Flash 设备符合块设备的特点，但是我们仍然可以把它作为一个字符设备来访问。

字符设备和块设备的驱动设计呈现出很大的差异，但是对于用户而言，他们都使用文件系统的操作接口 open()、close()、read()、write()等函数进行访问。

在 Linux 系统中，网络设备面向数据包的接收和发送而设计，它并不对应于文件系统的节点。内核与网络设备的通信和内核与字符设备、块设备的通信方式完全不同。

另外，TTY 驱动、I^2C 驱动、USB 驱动、PCI 驱动、LCD 驱动等本身大体可归纳入 3 个基础大类，但是对于这些复杂的设备，Linux 系统还定义了独特的驱动体系结构。

1.4.2 Linux 设备驱动与整个软硬件系统的关系

如图 1.5 所示，除网络设备外，字符设备与块设备都被映射到 Linux 文件系统的文件和目录，通过文件系统的系统调用接口 open()、write()、read()、close()等函数即可访问字符设备和块设备。所有的字符设备和块设备都被统一地呈现给用户。块设备比字符设备复杂，在它上面会首先建立一个磁盘/Flash 文件系统，如 FAT、Ext3、YAFFS、JFFS 等。FAT、Ext3、YAFFS、JFFS 规范了文件和目录在存储介质上的组织。

应用程序可以使用 Linux 的系统调用接口编程，也可以使用 C 库函数，出于代码可移植性的考虑，后者更值得推荐。C 库函数本身也通过系统调用接口而实现，如 C 库函数中的 fopen()、fwrite()、fread()、fclose()分别会调用操作系统 API 的 open()、write()、read()、close()函数。

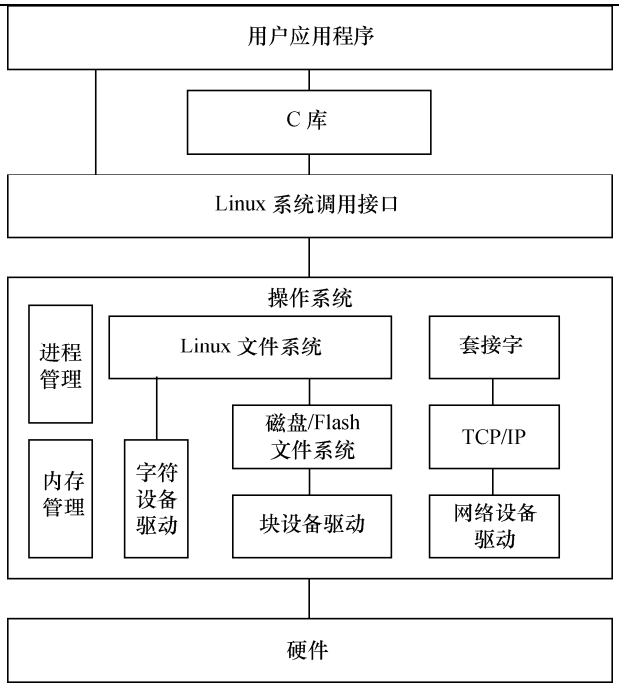

图 1.5　Linux 设备驱动与整个软硬件系统的关系

1.4.3　编写 Linux 设备驱动的技术基础

Linux 设备驱动的学习是一项浩大的工程，读者需要首先掌握以下基础。

- 编写 Linux 设备驱动要求工程师具有良好的硬件基础，懂得 SRAM、Flash、SDRAM、磁盘的读写方式，UART、I^2C、USB 等设备的接口，轮询、中断、DMA 的原理，PCI 总线的工作方式以及 CPU 的内存管理单元（MMU）等。
- 编写 Linux 设备驱动要求工程师具有良好的 C 语言基础，能灵活地运用 C 语言的结构体、指针、函数指针及内存动态申请和释放等。
- 编写 Linux 设备驱动要求工程师具有一定的 Linux 内核基础，虽然并不要求工程师对内核各个部分有深入的研究，但至少要了解设备驱动与内核的接口，尤其是对于块设备、网络设备、Flash 设备、串口设备等复杂设备。
- 编写 Linux 设备驱动要求工程师具有良好的多任务并发控制和同步的基础，因为在设备驱动中会大量使用自旋锁、互斥、信号量、等待队列等并发与同步机制。

本书对以上内容都进行了详细的讲解，以使读者快速掌握编写 Linux 设备驱动的基础。

1.4.4　Linux 设备驱动的学习方法

动手实践永远是学习任何软件开发的最好方法，学习 Linux 设备驱动也不例外。因此，您最好有一块可以实际练手的电路板来构造嵌入式开发环境。如果您暂时没有，则可以用 VmWare 搭建两台虚拟机，两台虚拟机上都运行 Linux 操作系统，一台作为开发主机，另一台作为目标机。

目前的 PC 上往往只有 1 个串口，但是调试要求主机和目标机之间使用串口通信，这要求 2 个串口。在虚拟机中我们可以用管道虚拟串口。在主机端设置“终端是客户机”，并选择“其他终端是一个虚拟机”。在目标机端设置“终端是服务器”，同样选择“其他终端是一个虚拟机”，但是要启用轮询。主机和目标机的串口设置分别如图 1.6 和图 1.7 所示。

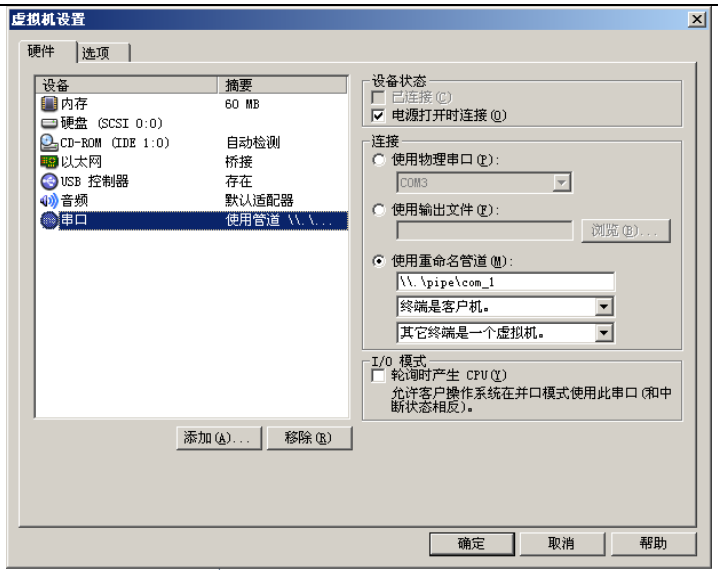

图 1.6　VmWare 中主机串口设置

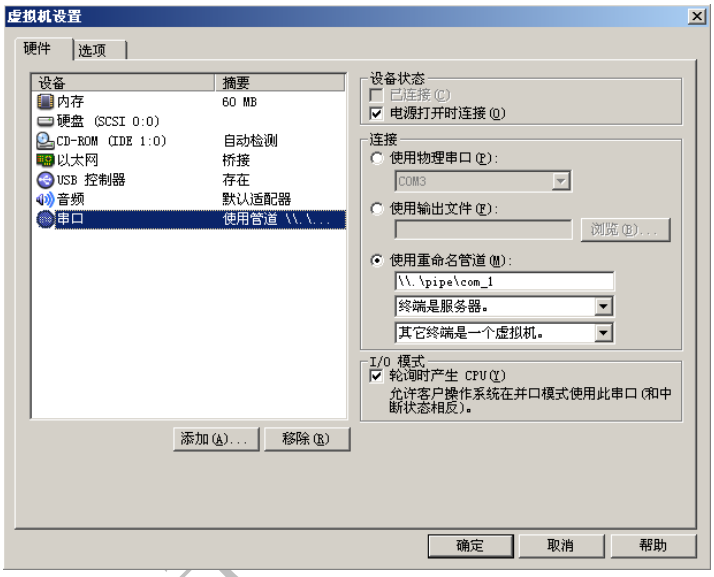

图 1.7　VmWare 中目标机串口设置

另外，为了使 VmWare 中安装的多个 Linux 系统之间以及 Linux 与 Windows 系统之间的网络互通，可以设置 VmWare 中的 Linux 系统使用 VMnet1（仅主机），如图 1.8 所示。打开 VMnet1 虚拟网卡的连接共享，将 Linux 系统的网关设置为 VMnet1 的 IP 地址“192.168.0.1”，Linux 系统将通过这个地址连接外网。

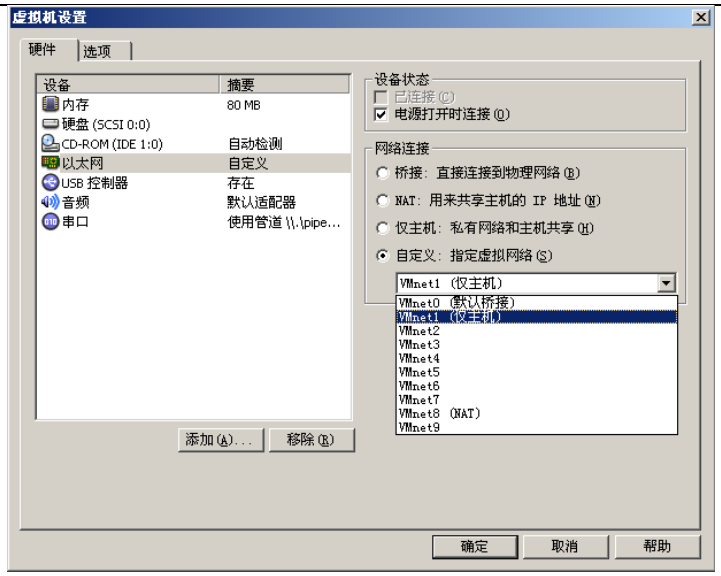

图 1.8　VmWare 中虚拟机网卡的设置

为了使 VmWare 中的 Linux 系统具有较高的显示分辨率，并能与主机上的 Windows 之间通过 hgfs 文件系统共享文件，最好在 Linux 中安装 VmWare tools。

源代码是学习 Linux 驱动的最权威资料，阅读 Linux 源代码的最佳工具是 Source Insight，在其中建立一个工程，并将 Linux 内核的所有源代码加入该工程，同步这个工程之后，我们将可以非常方便地在代码之间进行关联阅读，如图 1.9 所示。

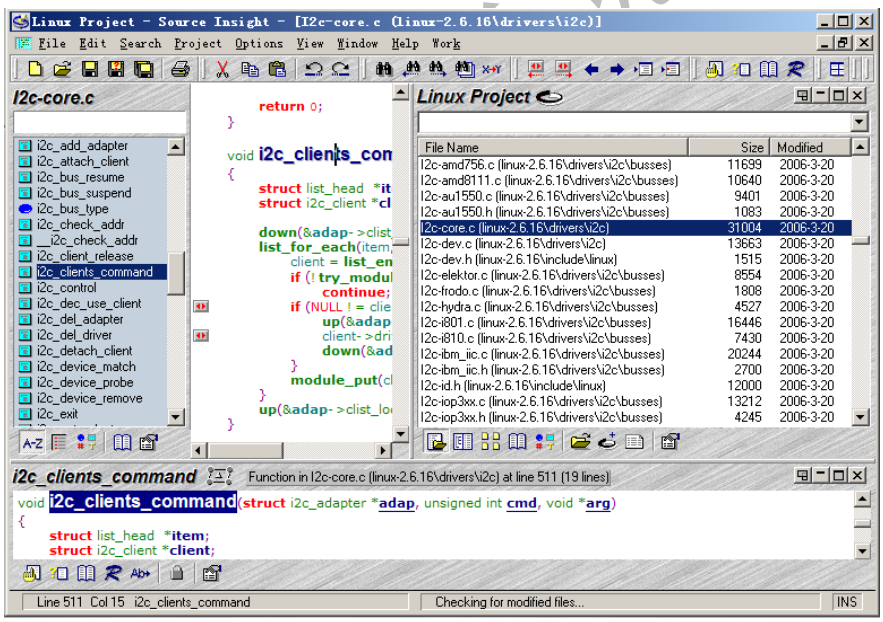

图 1.9　在 Source Insight 中阅读 Linux 源代码

网站 http://lxr.linux.no/ident 提供了 1.0.9 、1.2.13 、2.0.40、2.2.26、2.4.18 、2.4.20、2.4.28、2.6.10、2.6.11、2.6.17.13 、2.6.18、2.6.20.1 版本内核和 i386、Alpha、ARM、IA-64、m68k、MIPS、MIPS64、PowerPC、IBMS/390、SH、SPARC、SPARC64、x86-64 体系结构下 Linux 源代码的“Cross-Referencing”，在其中输入 Linux 内核中的函数、数据结构或变量的名称就可以直接得到以超链接形式给出的定义和引用它的所有位置。如搜索 2.6.20.1 内核、i386 体系结构下的 sys_write()函数，得到如下信息：

```
sys_write
Defined as a function in:

    * fs/read_write.c, line 374
```

```
Defined as a function prototype in:

    * include/linux/syscalls.h, line 380

Referenced (in 11 files total) in:

    * arch/mips/kernel/irixioctl.c, line 236
    * arch/mips/kernel/sysirix.c:
        o line 1046
        o line 1531
    * arch/s390/kernel/compat_linux.c, line 955
    * arch/sparc/kernel/sys_sunos.c, line 1093
    * arch/sparc64/kernel/sys_sunos32.c, line 1239
    * fs/read_write.c, line 374
    * include/asm-x86_64/unistd.h, line 18
    * include/asm-xtensa/unistd.h, line 48
    * include/linux/syscalls.h, line 380
    * init/do_mounts_rd.c:
        o line 239
        o line 373
    * init/initramfs.c:
        o line 305
        o line 311
        o line 552
```

除此之外，阅读经典书籍和参与 Linux 社区的讨论也是非常好的学习方法。Linux 内核源代码中包含了一个 Documentation 目录，其中包含了一批内核设计的文档，全部是文本文件。很遗憾，这些文档的组织不太好，内容也不够细致。

学习 Linux 设备驱动的一个注意事项是要避免管中窥豹、只见树木不见森林，因为各类 Linux 设备驱动都从属于一个 Linux 设备驱动的架构，单纯而片面地学习几个函数、几个数据结构是不可能理清设备驱动中各组成部分之间的关系的。因此，Linux 驱动的分析方法是点面结合，将对函数和数据结构的理解放在整体架构的背景之中，而这正是本书各章节讲解驱动的方法。

1.5 设备驱动的 Hello World：LED 驱动

1.5.1 无操作系统时的 LED 驱动

在嵌入式系统的设计中，LED 一般直接由 CPU 的 GPIO（通用可编程 I/O 口）控制。GPIO 一般由两组寄存器控制，即一组控制寄存器和一组数据寄存器。控制寄存器可设置 GPIO 口的工作方式为输入或输出。当引脚被设置为输出时，向数据寄存器的对应位写入 1 和 0 会分别在引脚上产生高电平和低电平；当引脚设置为输入时，读取数据寄存器的对应位可获得引脚上相应的电平信号。

在本例子中，我们屏蔽具体 CPU 的差异，假设在 GPIO_REG_CTRL 物理地址处的控制寄存器处的第 n 位写入 1 可设置 GPIO 为输出，在 GPIO_REG_DATA 物理地址处的数据寄存器的第 n 位写入 1 或 0 可在引脚上产生高或低电平，则在无操作系统的情况下，设备驱动代码如清单 1.3 所示。

代码清单 1.3　无操作系统时的 LED 驱动

```
1  #define reg_gpio_ctrl *(volatile int *)(ToVirtual(GPIO_REG_CTRL))
2  #define reg_gpio_data *(volatile int *)(ToVirtual(GPIO_REG_DATA))
3  /*初始化 LED*/
```

```
4  void LightInit(void)
5  {
6    reg_gpio_ctrl |= (1 << n); /*设置 GPIO 为输出*/
7  }
8
9  /*点亮 LED*/
10 void LightOn(void)
11 {
12   reg_gpio_data |= (1 << n); /*在 GPIO 上输出高电平*/
13 }
14
15 /*熄灭 LED*/
16 void LightOff(void)
17 {
18   reg_gpio_data &= ~(1 << n); /*在 GPIO 上输出低电平*/
19 }
```

上述程序中的 LightInit()、LightOn()、LightOff()等函数都将作为 LED 驱动提供给应用程序的外部接口函数。程序中 ToVirtual()等函数的作用是当系统启动了硬件 MMU 之后，根据物理地址和虚拟地址的映射关系，将寄存器的物理地址转化为虚拟地址。

1.5.2 Linux 系统下的 LED 驱动

在 Linux 操作系统下编写 LED 设备的驱动时，操作硬件的 LightInit()、LightOn()、LightOff()这些函数仍然需要，但是，需要遵循 Linux 编程的命名习惯，重新将其命名为 light_init()、light_on()、light_off()。这些函数将被 LED 驱动中独立于设备的针对内核的接口进行调用，代码清单 1.4 给出了 Linux 系统下的 LED 驱动，现在读者并不需要能读懂这些代码。

代码清单 1.4 Linux 系统下的 LED 驱动

```
1  #include .../*包含内核中的多个头文件*/
2
3  /*设备结构体*/
4  struct light_dev
5  {
6    struct cdev cdev; /*字符设备 cdev 结构体*/
7    unsigned char value; /*LED 亮时为 1，熄灭时为 0，用户可读写此值*/
8  };
9
10 struct light_dev *light_devp;
11 int light_major = LIGHT_MAJOR;
12
13 MODULE_AUTHOR("Song Baohua");
14 MODULE_LICENSE("Dual BSD/GPL");
15
16 /*打开和关闭函数*/
17 int light_open(struct inode *inode, struct file *filp)
18 {
19   struct light_dev *dev;
20   /* 获得设备结构体指针 */
21   dev = container_of(inode->i_cdev, struct light_dev, cdev);
```

```
22    /* 让设备结构体作为设备的私有信息 */
23    filp->private_data = dev;
24    return 0;
25  }
26  int light_release(struct inode *inode, struct file *filp)
27  {
28    return 0;
29  }
30
31  /*写设备:可以不需要 */
32  ssize_t light_read(struct file *filp, char _ _user *buf, size_t count,
33    loff_t*f_pos)
34  {
35    struct light_dev *dev = filp->private_data; /*获得设备结构体 */
36
37    if (copy_to_user(buf, &(dev->value), 1))
38    {
39      return  - EFAULT;
40    }
41    return 1;
42  }
43
44  ssize_t light_write(struct file *filp, const char _ _user *buf, size_t count,
45    loff_t *f_pos)
46  {
47    struct light_dev *dev = filp->private_data;
48
49    if (copy_from_user(&(dev->value), buf, 1))
50    {
51      return  - EFAULT;
52    }
53    /*根据写入的值点亮和熄灭 LED*/
54    if (dev->value == 1)
55      light_on();
56    else
57      light_off();
58
59    return 1;
60  }
61
62  /* ioctl 函数 */
63  int light_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
64    unsigned long arg)
65  {
66    struct light_dev *dev = filp->private_data;
67
68    switch (cmd)
```

```
69    {
70      case LIGHT_ON:
71        dev->value = 1;
72        light_on();
73        break;
74      case LIGHT_OFF:
75        dev->value = 0;
76        light_off();
77        break;
78      default:
79        /* 不能支持的命令 */
80        return  - ENOTTY;
81    }
82
83    return 0;
84  }
85
86  struct file_operations light_fops =
87  {
88    .owner = THIS_MODULE,
89    .read = light_read,
90    .write = light_write,
91    .ioctl = light_ioctl,
92    .open = light_open,
93    .release = light_release,
94  };
95
96  /*设置字符设备 cdev 结构体*/
97  static void light_setup_cdev(struct light_dev *dev, int index)
98  {
99    int err, devno = MKDEV(light_major, index);
100
101   cdev_init(&dev->cdev, &light_fops);
102   dev->cdev.owner = THIS_MODULE;
103   dev->cdev.ops = &light_fops;
104   err = cdev_add(&dev->cdev, devno, 1);
105   if (err)
106     printk(KERN_NOTICE "Error %d adding LED%d", err, index);
107 }
108
109 /*模块加载函数*/
110 int light_init(void)
111 {
112   int result;
113   dev_t dev = MKDEV(light_major, 0);
114
115   /* 申请字符设备号*/
```

```
116   if (light_major)
117     result = register_chrdev_region(dev, 1, "LED");
118   else
119   {
120     result = alloc_chrdev_region(&dev, 0, 1, "LED");
121     light_major = MAJOR(dev);
122   }
123   if (result < 0)
124     return result;
125
126   /* 分配设备结构体的内存 */
127   light_devp = kmalloc(sizeof(struct light_dev), GFP_KERNEL);
128   if (!light_devp)   /*分配失败*/
129   {
130     result =  - ENOMEM;
131     goto fail_malloc;
132   }
133   memset(light_devp, 0, sizeof(struct light_dev));
134   light_setup_cdev(light_devp, 0);
135   light_init();
136   return 0;
137
138   fail_malloc: unregister_chrdev_region(dev, light_devp);
139   return result;
140 }
141
142 /*模块卸载函数*/
143 void light_cleanup(void)
144 {
145   cdev_del(&light_devp->cdev); /*删除字符设备结构体*/
146   kfree(light_devp); /*释放在 light_init 中分配的内存*/
147   unregister_chrdev_region(MKDEV(light_major, 0), 1); /*删除字符设备*/
148 }
149
150 module_init(light_init);
151 module_exit(light_cleanup);
```

代码清单 1.4 的行数与代码清单 1.3 相比多了很多，除了代码清单 1.3 中的硬件操作函数仍然需要外，代码清单 1.4 中还包含了大量读者陌生的元素，如结构体 file_operations、cdev，Linux 内核模块声明用的 MODULE_AUTHOR、 MODULE_LICENSE、module_init、module_exit，以及用于字符设备注册、分配和注销用的函数 register_chrdev_region()、alloc_chrdev_region()、unregister_chrdev_region()等。此外设驱动中也增加了 light_init ()、light_cleanup ()、light_read()、light_write()等这样的函数。

此时，我们只需要有一个感性认识，那就是，上述元素都是 Linux 驱动与内核的接口。Linux 对各类设备的驱动都定义了类似的数据结构和函数。

联系方式

集团官网：www.hqyj.com　　嵌入式学院：www.embedu.org　　移动互联网学院：www.3g-edu.org

企业学院：www.farsight.com.cn　　物联网学院：www.topsight.cn　　研发中心：dev.hqyj.com

集团总部地址：北京市海淀区西三旗悦秀路北京明园大学校内 华清远见教育集团

北京地址：北京市海淀区西三旗悦秀路北京明园大学校区，电话：010-82600386/5

上海地址：上海市徐汇区漕溪路银海大厦 A 座 8 层，电话：021-54485127

深圳地址：深圳市龙华新区人民北路美丽 AAA 大厦 15 层，电话：0755-22193762

成都地址：成都市武侯区科华北路 99 号科华大厦 6 层，电话：028-85405115

南京地址：南京市白下区汉中路 185 号鸿运大厦 10 层，电话：025-86551900

武汉地址：武汉市工程大学卓刀泉校区科技孵化器大楼 8 层，电话：027-87804688

西安地址：西安市高新区高新一路 12 号创业大厦 D3 楼 5 层，电话：029-68785218