

第 1 章

Linux 设备驱动概述及开发环境构建

本章导读

本章将介绍 Linux 设备驱动开发的基本概念，并对本书所基于的平台和开发环境进行讲解。

1.1 节阐明设备驱动的概念和作用。

1.2 节和 1.3 节分别讲解在无操作系统情况下和有操作系统情况下设备驱动的设计，通过对设计差异的分析，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，给出设备驱动与整个软硬件系统的关系，分析 Linux 设备驱动的重点、难点和学习方法。

1.5 节对本书所基于的 QEMU 模拟的 vexpress ARM Cortex-A9 四核开发板和开发环境的安装进行介绍。

本章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转都是系统中软硬件共同努力的结果，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，并完成用户的最终诉求。硬件较固定，软件则很灵活，可以适应各种复杂多变的应用。因此，计算机系统的软硬件相互成就了对方。

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入对方的领地。为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。譬如，应用软件工程师在调用套接字发送和接收数据包的时候，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者串口。

2 Linux设备驱动开发详解：基于最新的Linux 4.0内核

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，硬件必须透明地呈现给他。谁来实现硬件对应用软件工程师的隐形？这个光荣而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”。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，读写设备的寄存器，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等，最终让通信设备能收发数据，让显示设备能显示文字和画面，让存储设备能记录文件和数据。

由此可见，设备驱动充当了硬件和应用软件之间的纽带，应用软件时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()，对 Flash 定义 FlashWr()、FlashRd() 等。而在有操作系统的情况下，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驱动，这样，驱动才能良好地整合入操作系统的内核中。

驱动程序负责硬件和应用软件之间的沟通，而驱动工程师则负责硬件工程师和应用软件工程师之间的沟通。目前，随着通信、电子行业的迅速发展，全世界每天都会生产大量新芯片，设计大量新电路板，也因此，会有大量设备驱动需要开发。这些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，它们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有操作系统，在许多情况下，操作系统都不必存在。对于功能比较单一、控制并不复杂的系统，譬如 ASIC 内部、公交车的刷卡机、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良好地支持它们的工作。一个无限循环中夹杂着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             {
8                 ProcessSerialInt();      /* 处理串口中断 */
9                 serialInt = 0;        /* 中断标志变量清 0 */
10            }
11         if (keyInt == 1)
```

```
12     /* 有按键中断 */
13     {
14         ProcessKeyInt();           /* 处理按键中断 */
15         keyInt = 0;              /* 中断标志变量清 0 */
16     }
17     status = CheckXXX();
18     switch (status)
19     {
20         ...
21     }
22     ...
23 }
24 }
```

在这样的系统中，虽然不存在操作系统，但是设备驱动则无论如何都必须存在。一般情况下，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，后者进行驱动的具体实现。譬如，可以像代码清单1.2那样定义一个串口的驱动。

代码清单1.2 无操作系统情况下串口的驱动

```
1  ****
2  *serial.h文件
3  ****
4  extern void SerialInit(void);
5  extern void SerialSend(const char buf*,int count);
6  extern void SerialRecv(char buf*,int count);
7
8  ****
9  *serial.c文件
10 ****
11 /* 初始化串口 */
12 void SerialInit(void)
13 {
14     ...
15 }
16 /* 串口发送 */
17 void SerialSend(const char buf*,int count)
18 {
19     ...
20 }
21 /* 串口接收 */
22 void SerialRecv(char buf*,int count)
23 {
24     ...
25 }
26 /* 串口中断处理函数 */
27 void SerialIsr(void)
28 {
29     ...
30     serialInt = 1;
31 }
```

4 Linux设备驱动开发详解：基于最新的Linux 4.0内核

其他模块想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。如要从串口上发送“Hello World”字符串，使用语句 SerialSend(“Hello World”, 11) 即可。

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应用软件工程师，应用软件没有跨越任何层次就直接访问设备驱动的接口。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之间平等，驱动中包含了业务层面上的处理，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重用的驱动代码。

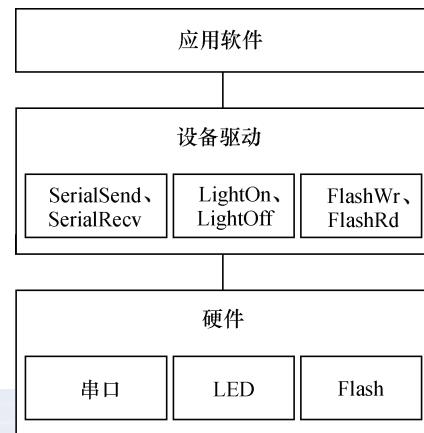


图 1.1 无操作系统时硬件、设备驱动和应用软件的关系

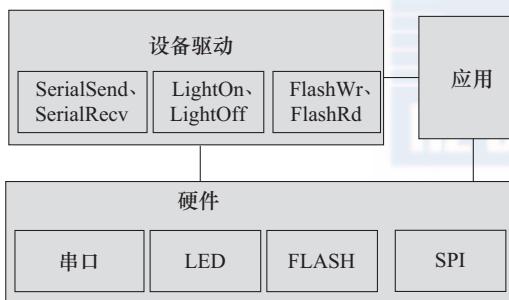


图 1.2 驱动与应用高耦合的不合理设计

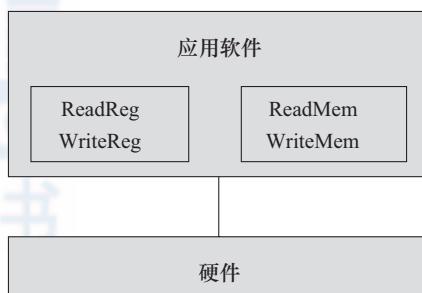


图 1.3 应用直接访问硬件的不合理设计

1.3 有操作系统时的设备驱动

在 1.2 节中我们看到一个清晰的设备驱动，它直接运行在硬件之上，不与任何操作系统关联。当系统包含操作系统时，设备驱动会变得怎样呢？

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，驱动不可能与硬件打交道。

其次，我们还需要将驱动融入内核。为了实现这种融合，必须在所有设备的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，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。

那么我们要问，有了操作系统之后，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32位处理器而言，Windows、Linux等操作系统可以让每个进程都可以独立地访问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实质性的好处？

简而言之，操作系统通过给驱动制造麻烦来达到给上层应用提供便利的目的。当驱动都按照操作系统给出的独立于设备的接口而设计时，那么，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，当应用程序通过write()、read()等函数读写文件就可访问各种字符设备和块设备，而不论设备的具体类型和工作方式，那将是多么便利。



图1.4 硬件、驱动、操作系统和应用程序的关系

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制作工艺的发展，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。譬如，相当多的ARM、PowerPC、MIPS等处理器都集成了UART、I²C控制器、SPI控制器、USB控制器、SDRAM控制器等，有的处理器还集成了GPU（图形处理器）、视频编解码器等。

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内核。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。

6 Linux设备驱动开发详解：基于最新的Linux 4.0内核

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。块设备可以按任意顺序进行访问，以块为单位进行操作，如硬盘、eMMC等。字符设备和块设备的驱动设计有出很大的差异，但是对于用户而言，它们都要使用文件系统的操作接口open()、close()、read()、write()等进行访问。

在Linux系统中，网络设备面向数据包的接收和发送而设计，它并不倾向于对应于文件系统的节点。内核与网络设备的通信与内核和字符设备、网络设备的通信方式完全不同，前者主要还是使用套接字接口。

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()等即可访问字符设备和块设备。所有字符设备和块设备都统一呈现给用户。Linux的块设备有两种访问方法：一种是类似dd命令对应的原始块设备，如“/dev/sdb1”等；另外一种方法是在块设备上建立FAT、EXT4、BTRFS等文件系统，然后以文件路径如“/home/barry/hello.txt”的形式进行访问。在Linux中，针对NOR、NAND等提供了独立的内存技术设备（Memory Technology Device，MTD）子系统，其上运行YAFFS2、JFFS2、UBIFS等具备擦除和负载均衡能力的文件系统。针对磁盘或者Flash设备的FAT、EXT4、YAFFS2、JFFS2、UBIFS等文件系统定义了文件和目录在存储介质上的组织。而Linux的虚拟文件系统则统一对它们进行了抽象。

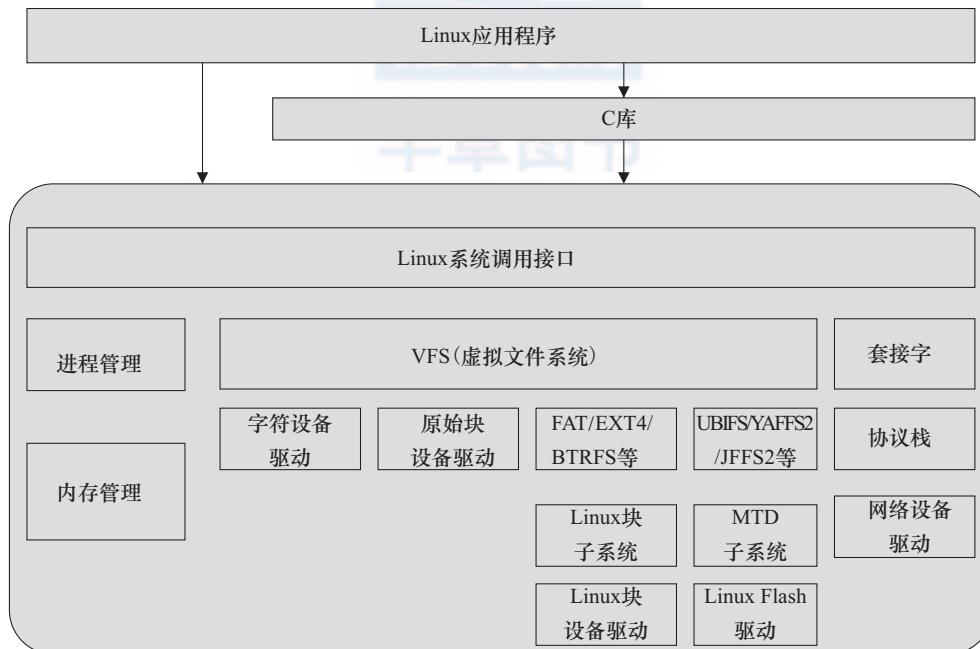


图1.5 Linux设备驱动与整个软硬件系统的关系

应用程序可以使用 Linux 的系统调用接口编程，但也可使用 C 库函数，出于代码可移植性的目的，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 `fopen()`、`fwrite()`、`fread()`、`fclose()` 分别会调用操作系统的 API `open()`、`write()`、`read()`、`close()`。

1.4.3 Linux 设备驱动的重点、难点

Linux 设备驱动的学习是一项浩繁的工程，包含如下重点、难点。

- 编写 Linux 设备驱动要求工程师有非常好的硬件基础，懂得 SRAM、Flash、SDRAM、磁盘的读写方式，UART、I²C、USB 等设备的接口以及轮询、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有非常好的 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，但至少要明白驱动与内核的接口。尤其是对于块设备、网络设备、Flash 设备、串口设备等复杂设备，内核定义的驱动体系结构本身就非常复杂。
- 编写 Linux 设备驱动要求工程师有非常好的多任务并发控制和同步的基础，因为在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。

上述经验值的获取并非朝夕之事，因此要求我们有足够的学习恒心和毅力。对这些重点、难点，本书都会在相应章节进行讲解。

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，本书使用的是通过 QEMU 模拟的 ARM vexpress 电路板，本书中的所有实例均可在该“电路板”上直接执行。

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。这是本书各章节讲解驱动的方法。

1.5 Linux 设备驱动的开发环境构建

1.5.1 PC 上的 Linux 环境

本书配套资源提供了一个 Ubuntu 的 VirtualBox 虚拟机映像，该虚拟机上安装了本书涉及的所有源代码、工具链和各种开发工具，读者无须再安装和配置任何环境。该虚拟机可运行于 Windows、Ubuntu 等操作系统中，运行方法如下。

1) 安装 VirtualBox。

如果主机为 Windows 系统，请安装 VirtualBox WIN 版本：

VirtualBox-4.3.20-96997-Win.exe

如果主机为 Ubuntu 系统，请安装 VirtualBox DEB 版本：

virtualbox-4.3_4.3.20-96996~Ubuntu~precise_i386.deb

2) 安装 VirtualBox extension。

Oracle_VM_VirtualBox_Extension_Pack-4.3.20-96996.vbox-extpack

3) 准备虚拟机镜像。

解压 Baohua_Linux.vmdk.rar 为 Baohua_Linux.vmdk

4) 新建虚拟机。

运行第 1 步安装的 Oracle VM VirtualBox，单击“新建 (N)”图标创建虚拟机，“类型”选择 Linux，“版本”选择 Ubuntu (32 bit)，名称可以取名为“linux-training”，如图 1.6 所示。

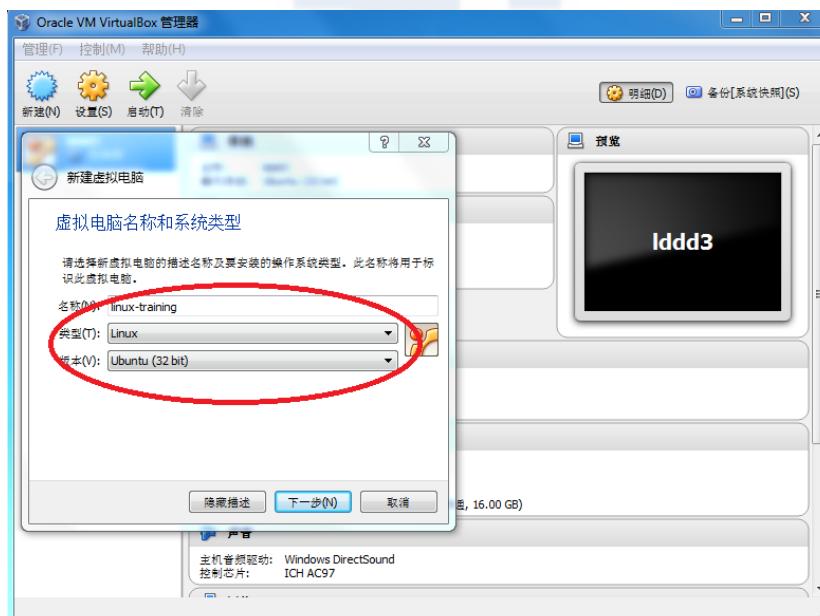


图 1.6 新建 Ubuntu 32 位虚拟机

单击“下一步 (N)”按钮，设置内存，如图 1.7 所示。

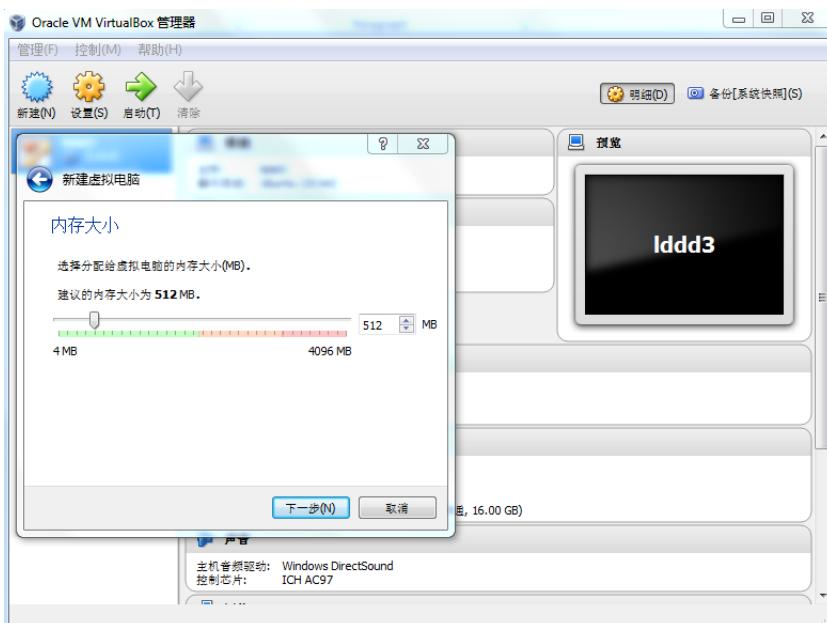


图 1.7 设置虚拟机的内存

继续单击“下一步 (N)”按钮。设置硬盘，注意选择“使用已有的虚拟硬盘文件 (U)”单选按钮，虚拟硬盘文件是第 3 步解压之后的“Baohua_Linux.vmdk”，如图 1.8 所示。



图 1.8 设置虚拟机硬盘镜像

10 Linux设备驱动开发详解：基于最新的Linux 4.0内核

最后，单击“创建”按钮以完成虚拟机的构建工作。

5) 启动虚拟机。

在 VirtualBox 上选择先前创建的“linux-training”虚拟机并单击“启动”图标，如图 1.9 所示。

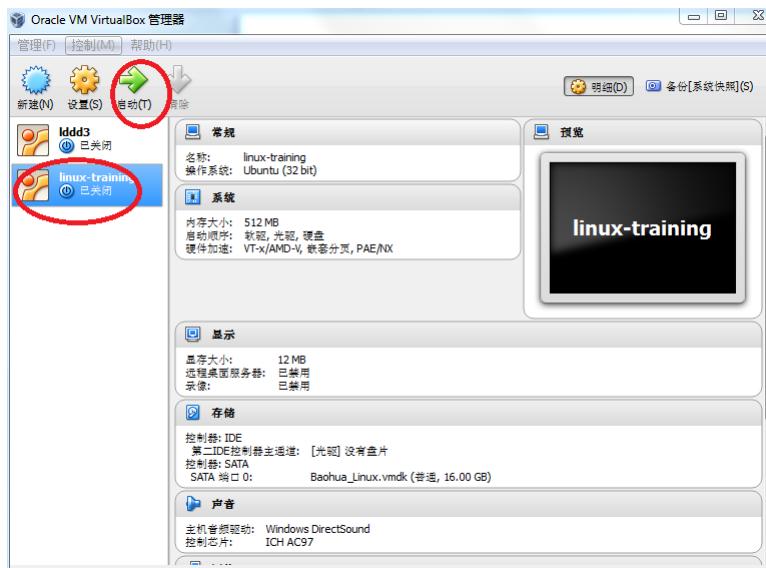


图 1.9 启动虚拟机

虚拟机的账号和密码都是“baohua”，如果要执行特权命令，sudo 密码也是“baohua”，如图 1.10 所示。

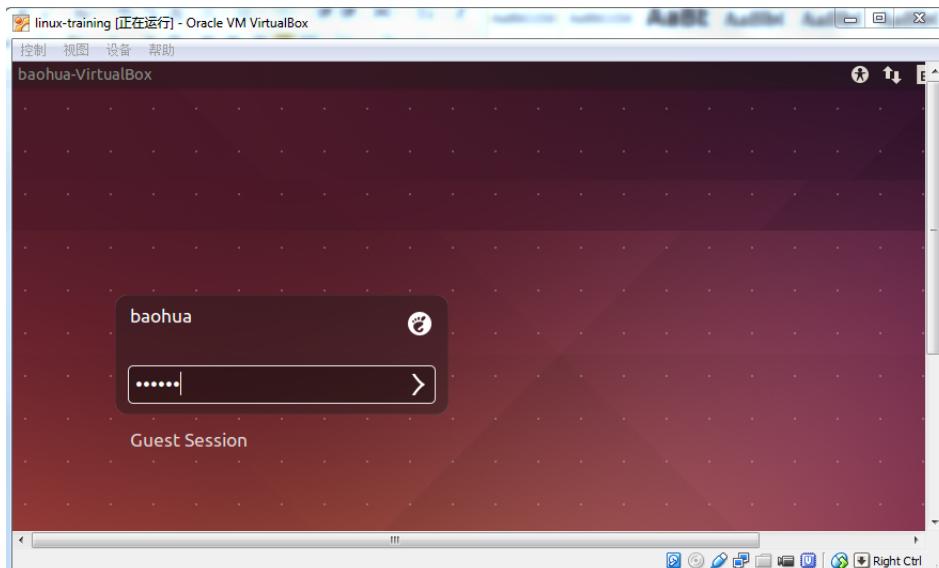


图 1.10 虚拟机登录界面

本书配套的 Ubuntu 版本是 14.04，但是内核版本升级到了 4.0-rc1，以保证和本书讲解内容的版本一致。

注意事项：

如果发现 VirtualBox 不稳定或者有兼容性问题（经过测试，有极少数 PC 存在此问题），也可以安装 VMware（Baohua_Linux.vmdk 也是支持 VMware 的）。

如果光盘不小心损坏，可以从链接：<http://pan.baidu.com/s/1c08gzi4>（密码为 puki）处提取网盘上的文件。

1.5.2 QEMU 实验平台

QEMU 模拟了 vexpress Cortex-A9 SMP 四核处理器开发板，板上集成了 Flash、SD、I²C、LCD 等。ARM 公司的 Versatile Express 系列开发平台提供了超快的环境，用于为下一代片上系统设计方案建立原型，比如 Cortex-A9 Quad Core。

在 <http://www.arm.com/zh/products/tools/development-boards/versatile-express/index.php> 上可以发现更多关于 Versatile Express 系列开发平台的细节。

本书配套虚拟机映像中已经安装好了工具链，包含 arm-linux-gnueabihf-gcc 和 arm-linux-gnueabi-gcc 两个版本。

Linux 内核在 /home/baohua/develop/linux 目录中，在该目录下面，包含内核编译脚本：

```
export ARCH=arm
export CROSS_COMPILE=arm-linux-gnueabi-
make LDD3_vexpress_defconfig
make zImage -j8
make modules -j8
make dtbs
cp arch/arm/boot/zImage extra/
cp arch/arm/boot/dts/*ca9.dtb    extra/
cp .config extra/
```

由此可见，我们用的默认内核配置文件是 LDD3_vexpress_defconfig。上述脚本也会自动将编译好的 zImage 和 dtbs 复制到 extra 目录中。

extra 目录下的 vexpress.img 是一张虚拟的 SD 卡，将作为根文件系统的存放介质。它能以 loop 的形式被挂载（mount），譬如在 /home/baohua/develop/linux 目录下运行。

```
sudo mount -o loop,offset=$((2048*512)) extra/vexpress.img extra/img
```

可以把 vexpress.img 的根文件系统分区挂载到 extra/img，这样我们可以在目标板的根文件系统中放置我们喜欢的内容。

/home/baohua/develop/linux 目录下面有个编译模块的脚本 module.sh，它会自动编译内核模块并安装到 vexpress.img 中，其内容如下：

```
make ARCH=arm CROSS_COMPILE=arm-linux-gnueabi- modules
```

12 Linux设备驱动开发详解：基于最新的Linux 4.0内核

```
sudo mount -o loop,offset=$((2048*512)) extra/vexpress.img extra/img
sudo make ARCH=arm modules_install INSTALL_MOD_PATH=extra/img
sudo umount extra/img
```

运行 extra 下面的 run-nolcd.sh 可以启动一个不含 LCD 的 ARM Linux。run-nolcd.sh 的内容为 qemu-system-arm -nographic -sd vexpress.img -M vexpress-a9 -m 512M -kernel zImage -dtb vexpress-v2p-ca9.dtb -smp 4 -append "init=/linuxrc root=/dev/mmcblk0p1 rw rootwait earlyprintk console=ttyAMA0" 2>/dev/null，运行结果为：

```
baohua@baohua-VirtualBox:~/develop/linux/extra$ ./run-nolcd.sh
Uncompressing Linux... done, booting the kernel.
Booting Linux on physical CPU 0x0
Initializing cgroup subsys cpuset
Linux version 3.16.0+ (baohua@baohua-VirtualBox) (gcc version 4.7.3 (Ubuntu/
Linaro 4.7.3-12ubuntu1) ) #3 SMP Mon Dec 1 16:53:04 CST 2014
CPU: ARMv7 Processor [410fc090] revision 0 (ARMv7), cr=10c53c7d
CPU: PIPT / VIPT nonaliasing data cache, VIPT aliasing instruction cache
Machine model: V2P-CA9
bootconsole [earlycon0] enabled
Memory policy: Data cache writealloc
PERCPU: Embedded 7 pages/cpu @9fbcd000 s7232 r8192 d13248 u32768
Built 1 zonelists in Zone order, mobility grouping on. Total pages: 130048
Kernel command line: init=/linuxrc root=/dev/mmcblk0p1 rw rootwait earlyprintk
console=ttyAMA0
PID hash table entries: 2048 (order: 1, 8192 bytes)
Dentry cache hash table entries: 65536 (order: 6, 262144 bytes)
Inode-cache hash table entries: 32768 (order: 5, 131072 bytes)
Memory: 513088K/524288K available (4583K kernel code, 188K rwdata, 1292K rodata,
247K init, 149K bss, 11200K reserved)
Virtual kernel memory layout:
  vector : 0xfffff0000 - 0xfffff1000 ( 4 kB)
  fixmap : 0xfffc00000 - 0xffe00000 (2048 kB)
  vmalloc : 0xa0800000 - 0xff000000 (1512 MB)
  lowmem : 0x80000000 - 0xa0000000 ( 512 MB)
  modules : 0x7f000000 - 0x80000000 ( 16 MB)
    .text : 0x80008000 - 0x805c502c (5877 kB)
    .init : 0x805c6000 - 0x80603c40 ( 248 kB)
    .data : 0x80604000 - 0x80633100 ( 189 kB)
    .bss : 0x80633108 - 0x806588a8 ( 150 kB)
SLUB: HWalign=64, Order=0-3, MinObjects=0, CPUs=4, Nodes=1
Hierarchical RCU implementation.
RCU restricting CPUs from NR_CPUS=8 to nr_cpu_ids=4.
RCU: Adjusting geometry for rcu_fanout_leaf=16, nr_cpu_ids=4
NR_IRQS:16 nr_irqs:16 16
L2C: platform modifies aux control register: 0x02020000 -> 0x02420000
L2C: device tree omits to specify unified cache
L2C: DT/platform modifies aux control register: 0x02020000 -> 0x02420000
L2C-310 enabling early BRESP for Cortex-A9
L2C-310 full line of zeros enabled for Cortex-A9
L2C-310 dynamic clock gating disabled, standby mode disabled
L2C-310 cache controller enabled, 8 ways, 128 kB
```

```
L2C-310: CACHE_ID 0x410000c8, AUX_CTRL 0x46420001
smp_twd: clock not found -2
sched_clock: 32 bits at 24MHz, resolution 41ns, wraps every 178956969942ns
Console: colour dummy device 80x30
...
```

运行 extra 下面的 run-lcd.sh 可以启动一个含 LCD 的 ARM Linux，运行结果如图 1.11 所示。

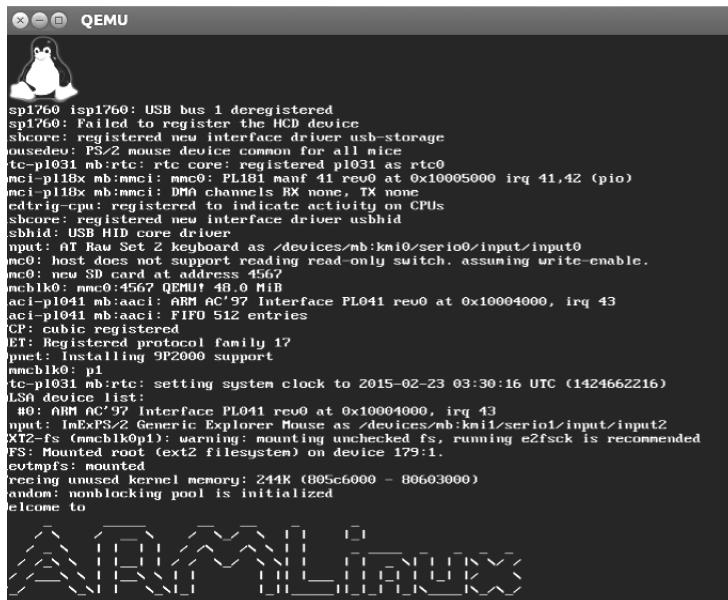


图 1.11 含 LCD 的 ARM Linux

除了 QEMU 模拟的 ARM 电路板以外，本书配套的 Ubuntu 中还包含一些直接可以在 Ubuntu 上运行的案例，譬如 /home/baohua/develop/training/kernel 中就包含了 globalfifo、globalmem 等，这些目录的源代码都包含了 Makefile，在其中直接 make，生成的 .ko 可以直接在 Ubuntu 上运行。

1.5.3 源代码阅读和编辑

源代码是学习 Linux 的权威资料，在 Windows 上阅读 Linux 源代码的最佳工具是 Source Insight，在其中建立一个工程，并将 Linux 的所有源代码加入该工程，同步这个工程之后，我们将能非常便捷地在代码之间进行关联阅读，如图 1.12 所示。

类似 <http://lxr.free-electrons.com/>、<http://lxr.oss.org.cn/> 这样的网站提供了 Linux 内核源代码的交叉索引，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。还有一些网站也提供了 Linux 内核中函数、变量和数据结构的搜索功能，在 google 中搜索“linux identifier search”可得。

14 Linux设备驱动开发详解：基于最新的Linux 4.0内核

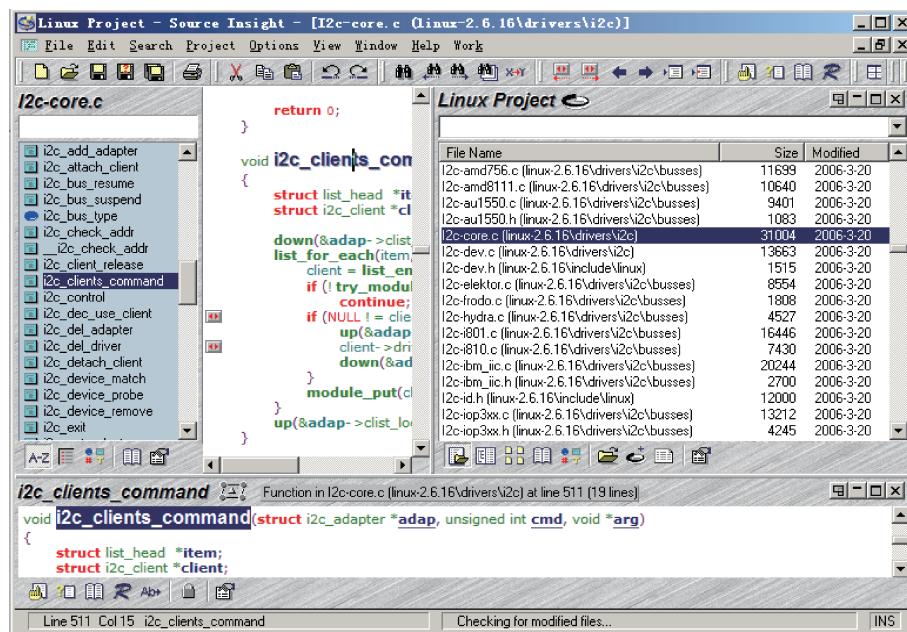


图 1.12 在 Source Insight 中阅读 Linux 源代码

在 Linux 主机上阅读和编辑 Linux 源码的常用方式是 vim + cscope 或者 vim + ctags，vim 是一个文本编辑器，而 cscope 和 ctags 则可建立代码索引，建议读者尽快使用基于文本界面全键盘操作的 vim 编辑器，如图 1.13 所示。

```
baohua@baohua-VirtualBox: ~/develop/linux
file Edit View Search Terminal Tabs Help
baohua@baohua-VirtualBox: ~/develop/linux          x baohua@baohua-VirtualBox: ~/develop/linux/extr
4 * Copyright (C) 2014 Barry Song (baohua@kernel.org)
5 *
6 * Licensed under GPLv2 or later.
7 */
8
9 #include <linux/module.h>
10 #include <linux/types.h>
11 #include <linux/sched.h>
12 #include <linux/init.h>
13 #include <linux/cdev.h>
14 #include <linux/slab.h>
15 #include <linux/poll.h>
16
17 #define GLOBALFIFO_SIZE 0x1000
18 #define FIFO_CLEAR 0x1
19 #define GLOBALFIFO_MAJOR 249
20
21 static int globalfifo_major = GLOBALFIFO_MAJOR;
22 module_param(globalfifo_major, int, S_IRUGO);
23
24 struct globalfifo_dev {
25     struct cdev cdev;
26     unsigned int current_len;
27     unsigned char mem[GLOBALFIFO_SIZE];
28     struct mutex mutex;
29     wait_queue_head_t r_wait;
30     wait_queue_head_t w_wait;
31 }
```

图 1.13 vim 编辑器

1.6 设备驱动 Hello World：LED 驱动

1.6.1 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者输出。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的电平为高或低。

在本例子中，我们屏蔽具体CPU的差异，假设在GPIO_REG_CTRL物理地址中控制寄存器处的第n位写入1可设置GPIO口为输出，在地址GPIO_REG_DATA物理地址中数据寄存器的第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，设备驱动见代码清单1.3。

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()都直接作为驱动提供给应用程序的外部接口函数。程序中ToVirtual()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。

1.6.2 Linux下的LED驱动

在Linux下，可以使用字符设备驱动的框架来编写对应于代码清单1.3的LED设备驱动（这里仅仅是为了方便讲解，内核中实际实现了一个提供sysfs节点的GPIO LED驱动，位于drivers/leds/leds-gpio.c中），操作硬件的LightInit()、LightOn()、LightOff()函数仍然需要，

16 Linux设备驱动开发详解：基于最新的Linux 4.0内核

但是，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设备驱动中独立于设备并针对内核的接口进行调用，代码清单1.4给出了Linux下的LED驱动，此时读者并不需要能读懂这些代码。

代码清单1.4 Linux操作系统下的LED驱动

```
1 #include .../* 包含内核中的多个头文件 */  
  
2 /* 设备结构体 */  
3 struct light_dev {  
4     struct cdev cdev;          /* 字符设备cdev结构体 */  
5     unsigned char value;      /* LED亮时为1，熄灭时为0，用户可读写此值 */  
6 };  
  
7 struct light_dev *light_devp;  
8 int light_major = LIGHT_MAJOR;  
  
9 MODULE_AUTHOR("Barry Song <21cnbao@gmail.com>");  
10 MODULE_LICENSE("Dual BSD/GPL");  
11 /* 打开和关闭函数 */  
12 int light_open(struct inode *inode, struct file *filp)  
13 {  
14     struct light_dev *dev;  
15     /* 获得设备结构体指针 */  
16     dev = container_of(inode->i_cdev, struct light_dev, cdev);  
17     /* 让设备结构体作为设备的私有信息 */  
18     filp->private_data = dev;  
19     return 0;  
20 }  
  
21 int light_release(struct inode *inode, struct file *filp)  
22 {  
23     return 0;  
24 }  
  
25 /* 读写设备：可以不需要 */  
26 ssize_t light_read(struct file *filp, char __user *buf, size_t count,  
27     loff_t *f_pos)  
28 {  
29     struct light_dev *dev = filp->private_data; /* 获得设备结构体 */  
30     if (copy_to_user(buf, &(dev->value), 1))  
31         return -EFAULT;  
  
32     return 1;  
33 }  
  
34 ssize_t light_write(struct file *filp, const char __user *buf, size_t count,  
35     loff_t *f_pos)
```

```
36  {
37      struct light_dev *dev = filp->private_data;
38
39      if (copy_from_user(&(dev->value), buf, 1))
40          return -EFAULT;
41
42      /* 根据写入的值点亮和熄灭 LED */
43      if (dev->value == 1)
44          light_on();
45      else
46          light_off();
47
48      return 1;
49  }
50
51  /* ioctl 函数 */
52  int light_ioctl(struct inode *inode, struct file *filp, unsigned int cmd,
53                  unsigned long arg)
54  {
55      struct light_dev *dev = filp->private_data;
56
57      switch (cmd) {
58      case LIGHT_ON:
59          dev->value = 1;
60          light_on();
61          break;
62      case LIGHT_OFF:
63          dev->value = 0;
64          light_off();
65          break;
66      default:
67          /* 不能支持的命令 */
68          return -ENOTTY;
69  }
70
71  return 0;
72
73  struct file_operations light_fops = {
74      .owner = THIS_MODULE,
75      .read = light_read,
76      .write = light_write,
77      .ioctl = light_ioctl,
78      .open = light_open,
79      .release = light_release,
80  };
81
82  /* 设置字符设备 cdev 结构体 */
```

18 Linux设备驱动开发详解：基于最新的Linux 4.0内核

```
76 static void light_setup_cdev(struct light_dev *dev, int index)
77 {
78     int err, devno = MKDEV(light_major, index);
79     cdev_init(&dev->cdev, &light_fops);
80     dev->cdev.owner = THIS_MODULE;
81     dev->cdev.ops = &light_fops;
82     err = cdev_add(&dev->cdev, devno, 1);
83     if (err)
84         printk(KERN_NOTICE "Error %d adding LED%d", err, index);
85 }

86 /* 模块加载函数 */
87 int light_init(void)
88 {
89     int result;
90     dev_t dev = MKDEV(light_major, 0);
91     /* 申请字符设备号 */
92     if (light_major)
93         result = register_chrdev_region(dev, 1, "LED");
94     else {
95         result = alloc_chrdev_region(&dev, 0, 1, "LED");
96         light_major = MAJOR(dev);
97     }
98     if (result < 0)
99         return result;

100    /* 分配设备结构体的内存 */
101    light_devp = kmalloc(sizeof(struct light_dev), GFP_KERNEL);
102    if (!light_devp) {
103        result = -ENOMEM;
104        goto fail_malloc;
105    }
106    memset(light_devp, 0, sizeof(struct light_dev));
107    light_setup_cdev(light_devp, 0);
108    light_gpio_init();
109    return 0;

110 fail_malloc:
111     unregister_chrdev_region(dev, light_major);
112     return result;
113 }

114 /* 模块卸载函数 */
115 void light_cleanup(void)
116 {
117     cdev_del(&light_devp->cdev);           /* 删除字符设备结构体 */
118     kfree(light_devp);                     /* 释放放在 light_init 中分配的内存 */
119     unregister_chrdev_region(MKDEV(light_major, 0), 1); /* 删除字符设备 */
```

```
120 }  
  
121 module_init(light_init);  
122 module_exit(light_cleanup);
```

上述代码的行数与代码清单1.3已经不能相比了，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中还包含了大量暂时陌生的元素，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、分配和注销的函数register_chrdev_region()、alloc_chrdev_region()、unregister_chrdev_region()等。我们也不能理解为什么驱动中要包含light_init()、light_cleanup()、light_read()、light_write()等函数。

此时，我们只需要有一个感性认识，那就是，上述暂时陌生的元素都是Linux内核为字符设备定义的，以实现驱动与内核接口而定义的。Linux对各类设备的驱动都定义了类似的数据结构和函数。

