国嵌

www.enjoylinux.cn

# LINUX
# PCI、串口驱动程序

# Contents





# Contents

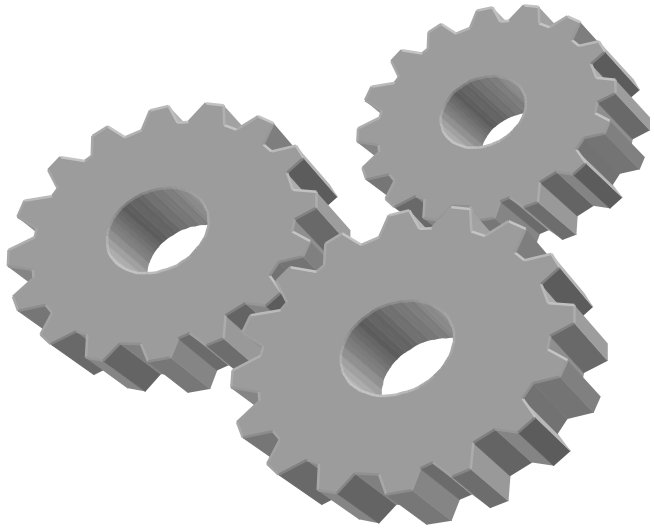




# 总线概念



总线是一种传输信号的信道;总线是连接一个或多个导体的电气连线。总线由电气接口和编程接口组成，我们重点关注编程接口。





# PCI概念



PCI是Peripheral Component Interconnect（外围设备互联）的简称，是在桌面及更大型的计算机上普遍使用的外设总线。



# PCI特点

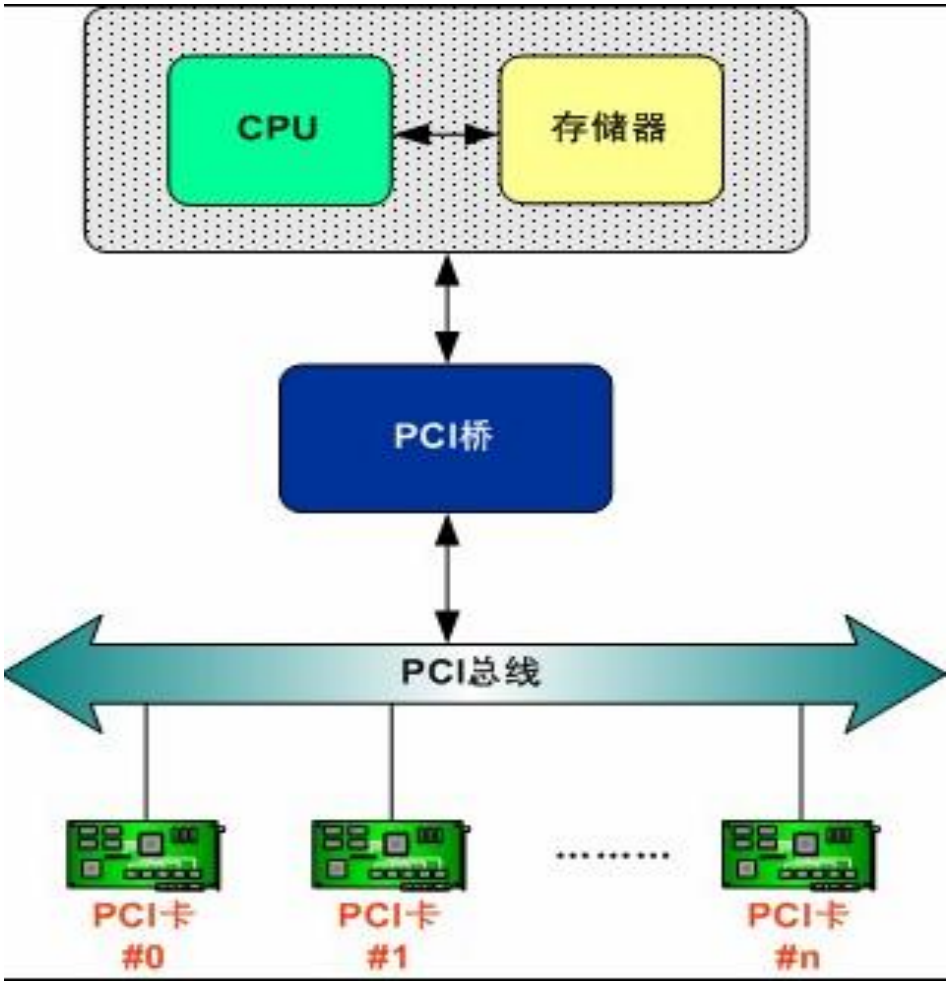


PCI总线具有三个非常显著的优点：

- 在计算机和外设间传输数据时具有更好的性能
- 能够尽量独立于具体的平台
- 可以方便地实现即插即用

# 体系结构-1





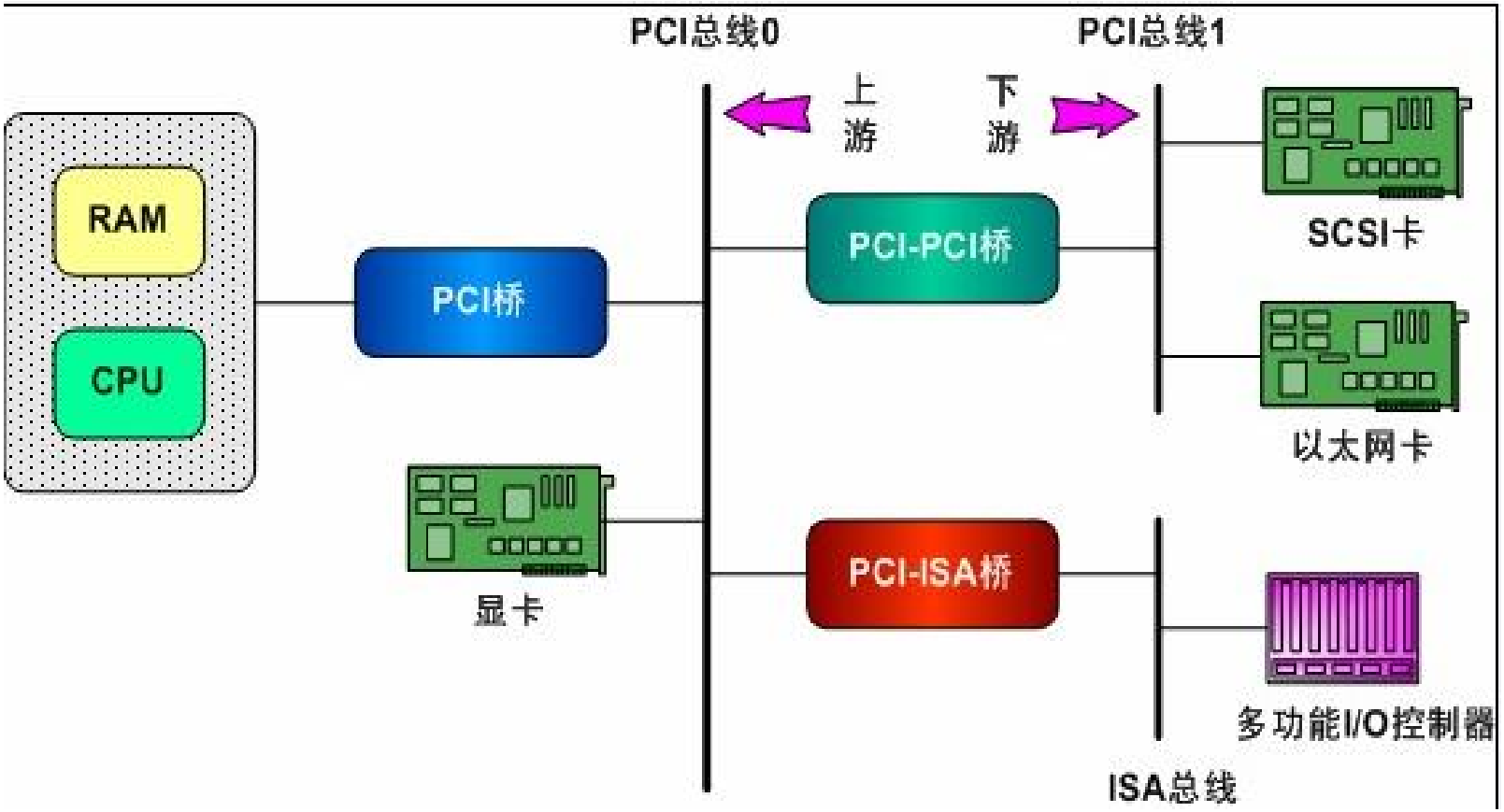
# 体系结构-1



从结构上看， PCI总线是一种不依附于某个具体处理器的局部总线，它是在CPU和原来的系统总线之间插入的一级总线，具体由一个桥接电路实现对这一层的管理，并实现上下之间的接口以协调数据的传送。

# 体系结构-2





# 体系结构-2



系统的各个部分通过PCI总线和PCI-PCI桥连接在一起。CPU和RAM通过PCI桥连接到PCI总线0（即主PCI总线），而具有PCI接口的显卡直接连接到主PCI总线上。PCI-PCI桥是一个特殊的PCI设备，它负责将PCI总线0和PCI总线1连接在一起。图中连接到 PCI 1号总线上的是SCSI卡和以太网卡。为了兼容旧的ISA总线标准，PCI总线还可以通过PCI-ISA桥来连接ISA总线，从而支持以前的ISA设备，图中ISA总线上连接着一个多功能I/O控制器，用于控制键盘、鼠标和软驱等。

# PCI设备寻址



每个PCI设备由一个总线号、一个设备号、和一个功能号确定。PCI规范允许一个系统最多拥有256条总线，每条总线最多带32个设备，但每个设备可以是最多8个功能的多功能板（如一个音频设备带一个CD-ROM驱动器）。

# PCI设备寻址



/proc/iomem描述了系统中所有的设备I/O在内存地址空间上的映射。我们来看地址从1G开始的第一个设备在/proc/iomem中是如何描述的:

40000000-400003ff : 0000:00:1f.1

这是一个PCI设备，40000000-400003ff是它所映射的内存空间地址，占据了内存地址空间 1024 bytes的位置，而0000:00:1f.1则是这个PCI外设的地址,它以冒号和逗号分隔为4个部分，第一个16位表示域，第二个8位表示一个总线号，第三个5位表示一个设备号，最后是3位，表示功能号。



# PCI设备寻址



因为PCI规范允许单个系统拥有最多256条总线，所以总线编号是8位。每个总线上可支持32个设备，所以设备号是5位，而每个设备上最多可有8种功能，所以功能号是3位。由此，由此可以得出上述的PCI设备的地址是0号总线上的31号设备上的1号功能。

# PCI寻址



使用lspci命令可以查看系统中的PCI设备，下面是使用lspci命令得到的一组输出：(练习：分析该数据后画出系统的PCI结构图)

00:00.0 Host bridge: Intel Corporation 82845 845

00:01.0 PCI bridge: Intel Corporation 82845 845

00:1d.0 USB Controller: Intel Corporation 82801CA/CAM USB

00:1d.1 USB Controller: Intel Corporation 82801CA/CAM USB

00:1e.0 PCI bridge: Intel 82801 Mobile PCI Bridge

00:1f.0 ISA bridge: Intel Corporation 82801CAM ISA Bridge

00:1f.1 IDE interface: Intel Corporation 82801CAM IDE U100



# PCI寻址



00:1f.3 SMBus: Intel 82801CA/CAM SMBus Controller

00:1f.5 Multimedia audio controller:Intel Corporation
82801CA/CAM AC'97 Audio Controller

00:1f.6 Modem: Intel 82801CA/CAM AC'97 Modem Controller

01:00.0 VGA compatible controller: nVidia Corporation NV17

02:00.0 FireWire: VIA Technologies. 1394 Host Controller

02:01.0 Ethernet controller: Realtek Semiconductor Co.

02:04.0 CardBus bridge: O2 Micro, Inc. Cardbus Controller

02:04.1 CardBus bridge: O2 Micro, Inc. Cardbus Controller



# PCI寻址


www.enjoylinux.cn

```
00:00.0(主桥) –00:01.0(PCI 桥) -----01:00:0(nVidia 显卡)
              |
              |---00:1d(USB 控制器)
              |              |-- 00:1d:0(USB1 号控制器)
              |              |
              |              |--00:1d:1(USB2 号控制器)
              |
              |-00:1e:0(PCI 桥)
              |              |--02:00.0(IEEE1394)
              |              |
              |              |-02:01.0(8139 网卡)
              |              |
              |              |-02:04(CardBus 桥)
              |                             |-02:04.0(桥 1)
              |                             |
              |                             |--02:04.1(桥 2)
              |
              |-00:1f(多功能板卡)
                             |
                             |--00:1f:1(IDE 接口)
                             |
                             |--00:1f:3(SMBus)
                             |
                             |--00:1f:5(多媒体声音控制器)
                             |
                             |--00:1f:6(调制解调器)
```



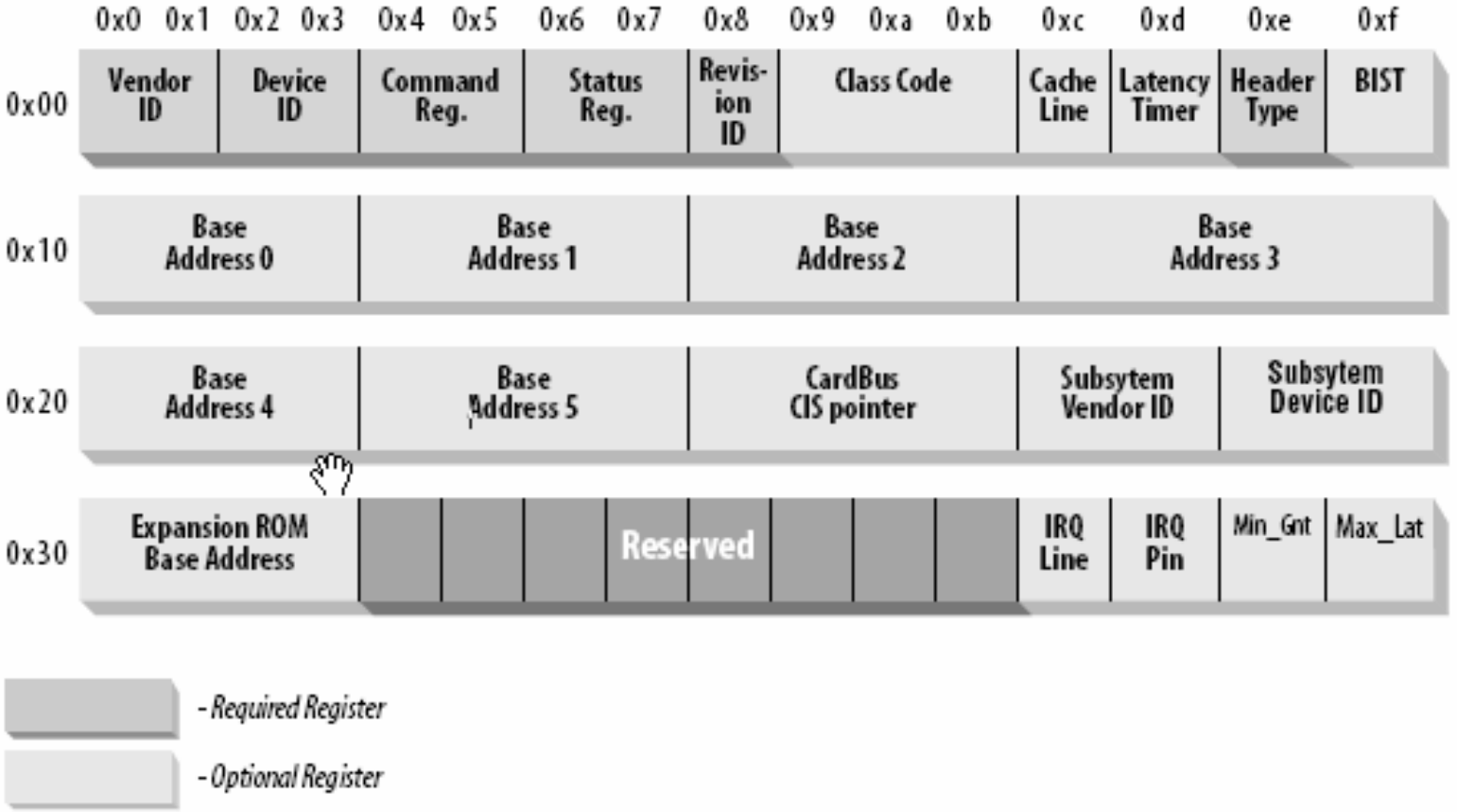
# 配置寄存器



每个PCI设备都有一组固定格式的寄存器，即配置寄存器，配置寄存器由Linux内核中的PCI初始化代码与驱动程序共同使用。内核在启动时负责对配置寄存器进行初始化，包括设置中断号以及I/O基址等。

# 配置寄存器





# 配置空间



00H—01H Vendor ID 制造商标识

02H—03H Device ID 设备标识

04H—05H Command 命令寄存器

06H—07H Status 状态寄存器

08H Revision ID 版本识别号寄存器

09H—0bH Class Code 分类代码寄存器

0cH Cache Line Size CACHE行长度寄存器

# 配置空间



0dH Latency Timer 主设备延迟时间寄存器

0eH Header Type 头标类型寄存器

0fH Bulit-in-teset Register 自测试寄存器

10H—13H Base Address Register 0 基地址寄存器0

14H—17H Base Address Register 1 基地址寄存器1

18H—1bH Base Address Register 2 基地址寄存器2

1cH—19H Base Address Register 3 基地址寄存器3



# 配置空间



20H—23H Base Address Register 4 基地址寄存器4

24H—27H Base Address Register 5 基地址寄存器5

28H—2bH Cardbus CIS Pointer 设备总线CIS指针寄存器

2cH—2dH Subsystem Vendor ID 子设备制造商标识

2eH—2fH Subsystem Device ID 子设备标识



# 配置空间



30H—33H Expasion ROM Base Address 扩展ROM基地址

34H—3bH ——— 保留

3cH Interrupt Line 中断线寄存器

3dH Interrupt Pin 中断引脚寄存器

3eH Min_Gnt 最小授权寄存器

3fH Max_Lat 最大延迟寄存器

# 配置空间



- 厂商标识(Vendor Id)

  用来标识PCI设备生产厂家的数值。Intel的厂商标识为0x8086，全球厂商标识由PCI Special Interest Group来分配。

- 设备标识(Device Id)

  用来标识设备的数值。Digital 21141快速以太设备的设备标识为0x0009



# 配置空间



- 基地址寄存器(Base Address Registers)记录此设备使用的I/O与内存空间的位置

- 中断连线(Interrupt Line)
  记录此设备使用的中断号

- 中断引脚(Interrupt Pin)
  记录此PCI设备使用的引脚号（A,B,C,D）

# Contents





# 驱动描述



在Linux内核中，PCI 驱动使用 struct pci_driver 结构来描述:

```
struct pci_driver {
    。。。。。。    。。。。。。
    const struct pci_device_id *id_table;
    int  (*probe)  (struct pci_dev *dev, const struct pci_device_id *id);
    void (*remove) (struct pci_dev *dev);
   /* Device removed (NULL if not a hot-plug capable driver) */
    。。。。。。    。。。。。。
}
```



# 注册驱动



注册 PCI 驱动, 使用如下函数:

pci_register_driver(struct pci_driver *drv)

# 使能设备



在 PCI 驱动使用PCI 设备的任何资源(I/O 区或者中断)之前，驱动必须调用如下函数来使能设备:

```
int pci_enable_device(struct pci_dev *dev)
```

# 获取基地址




一个PCI设备最多可以实现6个地址区域，大多数PCI设备在这些区域实现I/O寄存器。Linux提供了一组函数来获取这些区间的基地址:

**pci_resource_start(struct pci_dev *dev, int bar)**

返回指定区域的起始地址，这个区域通过参数 bar 指定,范围从 0-5，表示6个PCI区域中的一个。

**pci_resource_end(struct pci_dev *dev, int bar)**

返回指定区域的末地址。



# 中断



中断号存放于配置寄存器PCI_INTERRUPT_LINE中，驱动不必去检查它，因为从 PCI_INTERRUPT_LINE 中找到的值保证是正确的。如果设备不支持中断，寄存器 PCI_INTERRUPT_PIN 中的值是0，否则它是非零的值。但因为驱动开发者通常知道设备是否是支持终端，所以常常不需要访问 PCI_INTERRUPT_PIN。

# 实例分析



《国嵌PCI网卡驱动程序分析》



# Contents





# 数据通信



数据通信的基本方式可分为并行通信与串行通信两种：

- 并行通信：利用多条数据线将数据的各位同时传送。它的特点是传输速度快，适用于短距离通信。
- 串行通信：利用一条数据线将数据一位位地顺序传送。特点是通信线路简单，利用简单的线缆就可实现通信，低成本，适用于远距离通信。

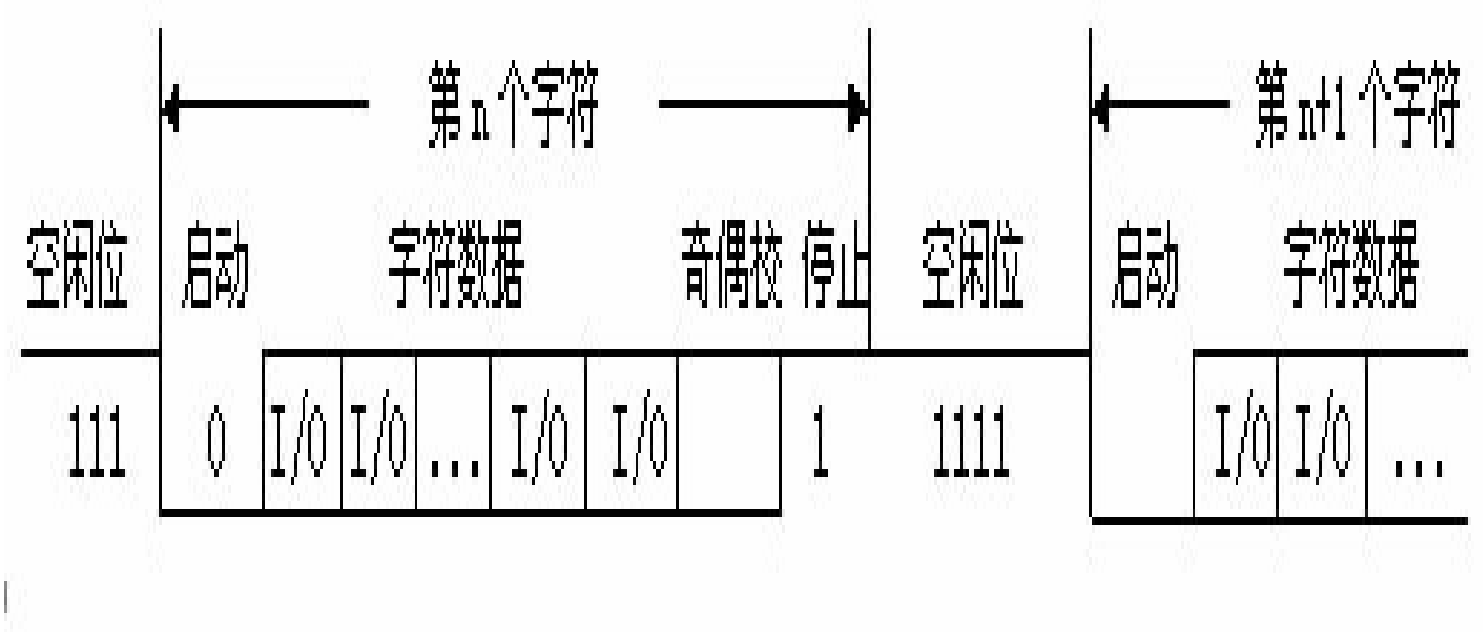
# 异步通信



- 异步通信以一个字符为传输单位，通信中两个字符间的时间间隔是不固定的，然而同一个字符中的两个相邻位之间的时间间隔是固定的。
- 通信协议：是指通信双方约定的一些规则。在使用异步串口传送一个字符的信息时，对数据格式有如下约定：规定有空闲位、起始位、资料位、奇偶校验位、停止位。

# 异步通信





# 异步通信



- 起始位：先发一个逻辑”0”信号，表示传输字符的开始
- 数据位：紧接在起始位之后。数据位的个数可以是4、5、6、7、8等，从最低位开始传送，靠时钟定位。
- 奇偶校验位：数据位加上这一位后，使得“1”的位数应为偶数(偶校验)或奇数(奇校验)，以此校验数据传送的正确性。
- 停止位：它是一个字符数据的结束标志。
- 空闲位：处于逻辑“1”状态，表示当前线路上没有数据传送。



# 波特率

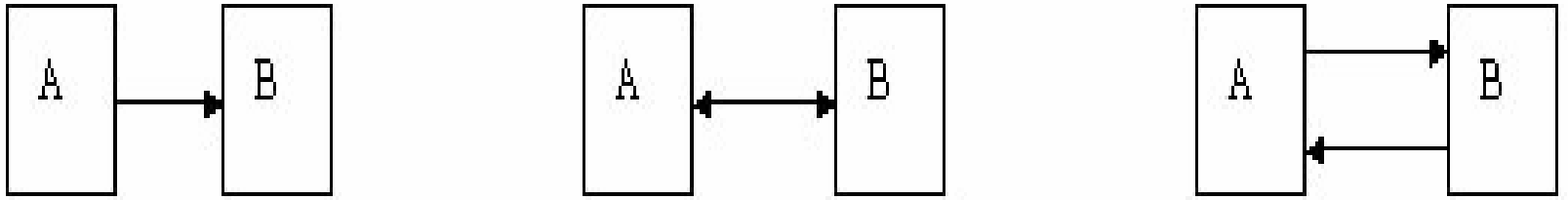


是衡量数据传送速率的指针。表示每秒钟传送的二进制位数。例如数据传送速率为120字符/秒，而每一个字符为10位，则其传送的波特率为10 × 120 = 1200位/秒 = 1200波特。

注：异步通信是按字符传输的，接收设备在收到起始信号之后只要在一个字符的传输时间内能和发送设备保持同步就能正确接收。

# 传送方式






- 单工方式：数据始终是从A设备发向B设备。
- 半双工方式：资料能从A设备传送到B设备，也能从B设备传送到A设备。但在任何时候都不能在两个方向上同时传送，即每次只能有一个设备发送，另一个设备接收。
- 全双工方式：允许通信双方同时进行发送和接收。这时，A设备在发送的同时也可以接收，B设备也一样。



# 终端概述



在Linux中，TTY(终端)是一类字符设备的统称，包括了3种类型：控制台，串口和伪终端。

# 控制台



供内核使用的终端为控制台。控制台在Linux启动时，通过命令console=…指定，如果没有指定控制台，系统把第一个注册的终端(tty)作为控制台。

# 控制台



1. 控制台是一个虚拟的终端，它必须映射到真正的终端上。
2. 控制台可以简单的理解为printk输出的地方。
3. 控制台是个只输出的设备，功能很简单，只能在内核中访问。

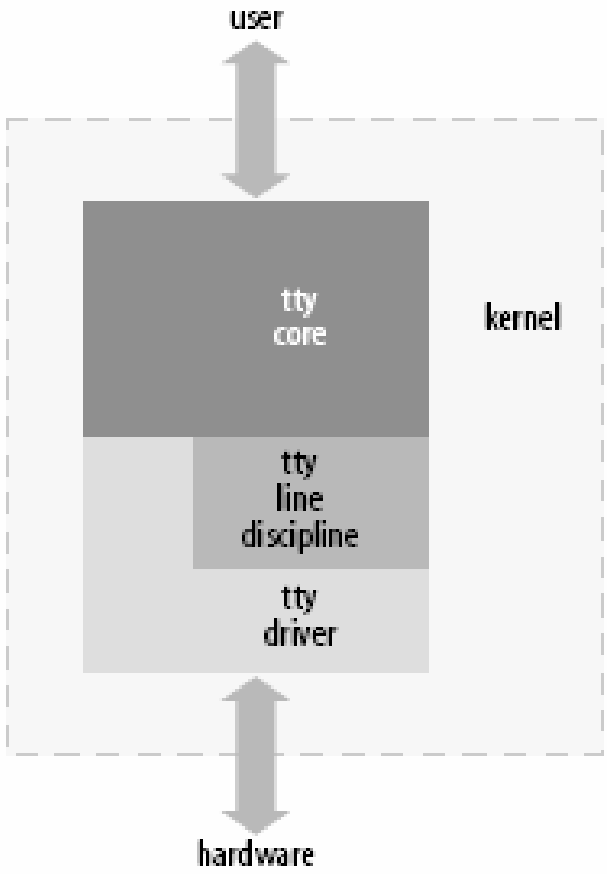
# 伪终端




伪终端设备是一种特殊的终端设备，由主-从两个成对的设备构成，当打开主设备时，对应的从设备随之打开，形成连接状态。输入到主设备的数据成为从设备的输出，输入到从设备的数据成为主设备的输出，形成双向管道。伪终端设备常用于远程登录服务器来建立网络和终端的关联。当通过telnet远程登录到另一台主机时，telnet进程与远程主机的telnet服务器相连接. telnet服务器使用某个主设备并通过对应的从设备与telnet进程相互通信。

# 终端体系





# 终端体系




在Linux中，TTY体系分为：TTY核心，TTY线路规程，TTY驱动3部分。TTY核心从用户获取要发送给TTY设备的数据，然后把数据传递给TTY线路规程，它对数据进行处理后，负责把数据传递到TTY驱动程序，TTY驱动程序负责格式化数据，并通过硬件发送出去。

# 终端体系

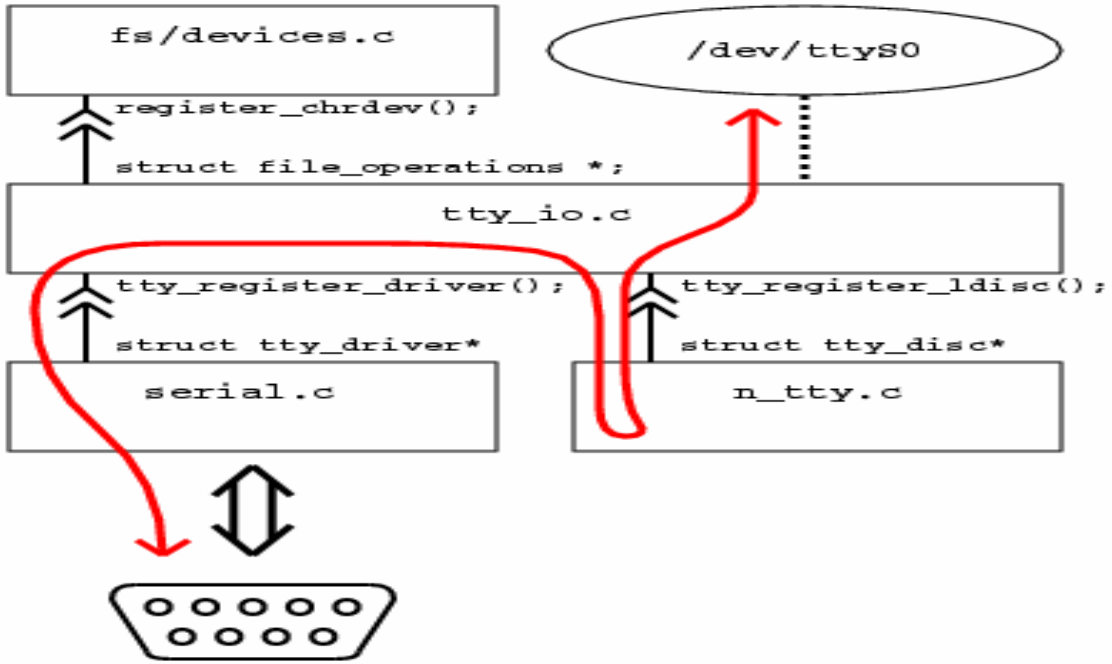


从硬件收到的数据向上通过TTY驱动, 进入TTY线路规程, 再进入TTY核心, 最后被用户获取。TTY驱动可以直接和TTY核心通讯, 但是通常TTY线路规程会修改在两者之间传送的数据。TTY驱动不能直接和线路规程通信，甚至不知道它的存在，线路规程的工作是格式化从用户或者硬件收到的数据. 这种格式化常常实现为一个协议, 如 PPP或 Bluetooth。

# 终端体系—串口

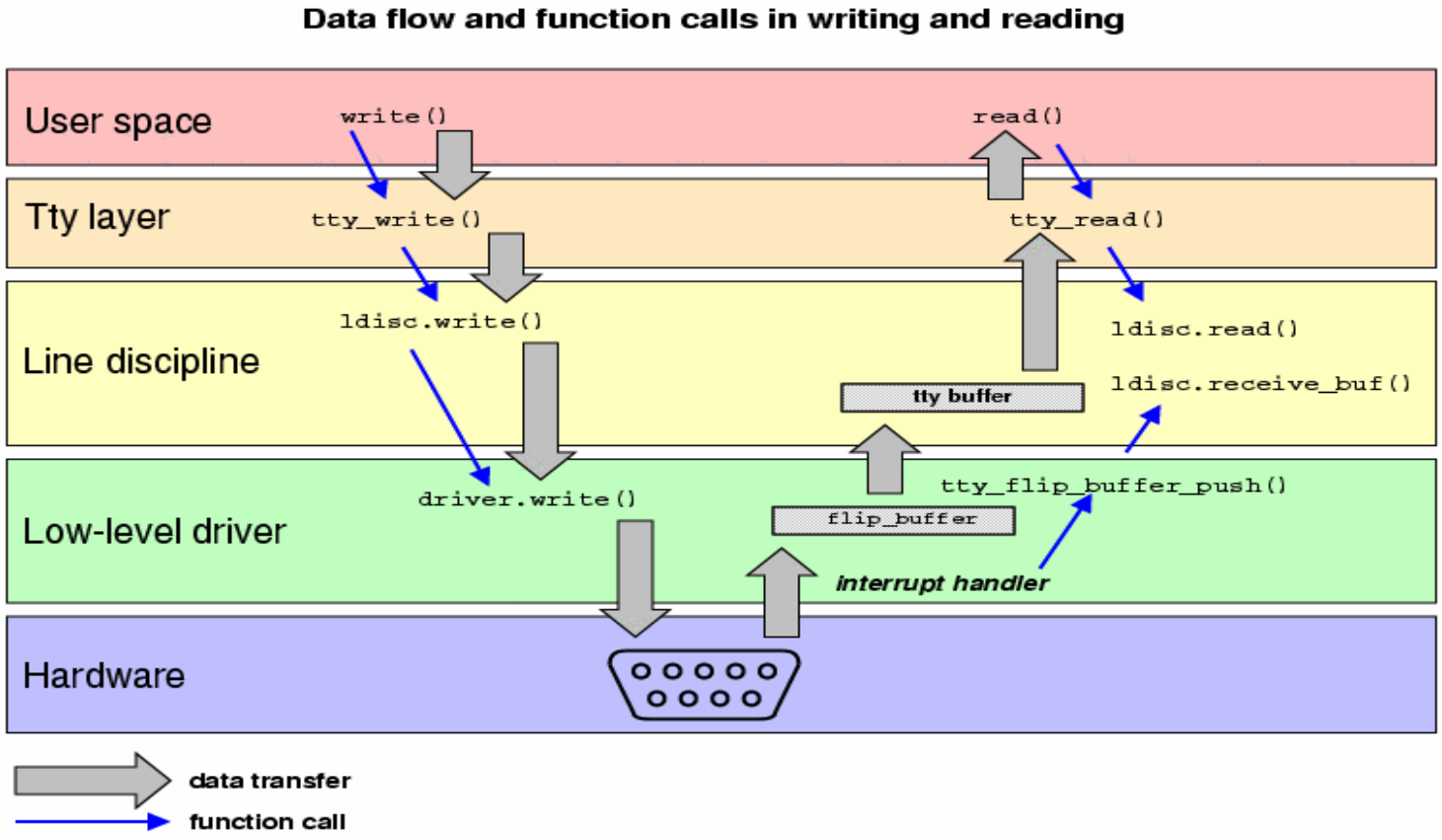


**Source files involved in serial management, how they are connected and how data flows.**



# 数据流





# 读操作



TTY驱动从硬件收到数据后，负责把数据传递到TTY 核心，TTY核心将从TTY驱动收到的数据缓存到一个tty_flip_buffer 类型的结构中 。该结构包含两个数据数组。从TTY设备接收到的数据被存储于第一个数组，当这个数组满，等待数据的用户将被通知。当用户从这个数组读数据时，任何从TTY驱动新来的数据将被存储在第2个数组。当第二个数组存满后，数据再次提交给用户，并且驱动又开始填充第1个数组，以此交替。

# 驱动描述



Linux内核使用uart_driver描述串口驱动，它包含串口设备的驱动名、设备名、设备号等信息。

```
struct uart_driver {
    struct module       *owner;
    const char          *driver_name;  //驱动名
    const char          *dev_name;   //设备名
    int                  major;   //主设备号
    int                  minor;   //起始次设备号
    int                  nr;      //设备数
    struct console                *cons;

    struct uart_state    *state;
    struct tty_driver    *tty_driver;
};
```

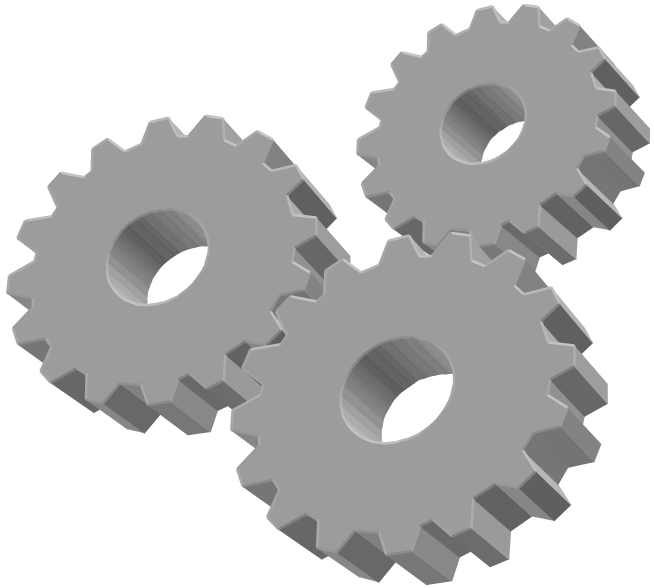


# 注册驱动



Linux为串口驱动注册提供了如下接口：

```
int uart_register_driver(struct uart_driver *drv)
```





# 端口描述




uart_port用于描述一个UART端口（一个串口）的地址、FIFO大小、端口类型等信息

```
struct uart_port
{
    spinlock_t lock; /* 端口锁 */
    unsigned int iobase; /* IO端口基地址 */
    unsigned char __iomem *membase; /* IO内存基地址 */
    unsigned int irq; /* 中断号 */
    unsigned char fifosize; /* 传输fifo大小 */
    const struct uart_ops     *ops;
    ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... ..... .....
}
```



# 操作串口




**uart_ops** 定义了针对串口的一系列操作，包括发送、接收及线路设置等。

```
struct uart_ops
{
    unsigned int(*tx_empty)(struct uart_port*);
    void(*set_mctrl)(struct uart_port *, unsigned int mctrl);
    unsigned int(*get_mctrl)(struct uart_port*);
    void(*stop_tx)(struct uart_port*);    //停止发送
    void(*start_tx)(struct uart_port*);   //开始发送
    void(*send_xchar)(struct uart_port *, char ch); //发送xchar
    void(*stop_rx)(struct uart_port*);   //停止接收
。。。。。。      。。。。。。。      。。。。。。
}
```



# 添加端口



串口核心层提供如下函数来添加1个端口：

**int uart_add_one_port(struct uart_driver *drv, struct uart_port *port)**



# 操作流程



1. 定义一个uart_driver的变量，并初始化;
2. 使用uart_register_driver来注册这个驱动;
3. 初始化uart_port和ops函数表;
4. 调用uart_add_one_port()添加初始化好的uart_port。



# 实例分析



## mini2440串口驱动程序