目录

[串口子系统 2](#_Toc493167944)

[概念解释 2](#_Toc493167945)

[1.1 串行通信（Serial communication） 2](#_Toc493167946)

[1.2同步串行通信（Synchronous serial communication）和异步串行通信（Asynchronous serial communication） 2](#_Toc493167947)

[1.3 串口（Serial port）和RS-232 3](#_Toc493167948)

[1.4 UART(UniversalAsynchronous Receiver/Transmitter) 4](#_Toc493167949)

[1.5 TTL电平转换以及USB转串口 4](#_Toc493167950)

[波特率（baud rate）和比特率（bit rate） 4](#_Toc493167951)

[帧格式和波特率误差 5](#_Toc493167952)

[Linux serial framework 6](#_Toc493167953)

[软件架构 6](#_Toc493167954)

[serial core 7](#_Toc493167955)

[earlycon 11](#_Toc493167956)

[3.1 静态定义的方式（不需要串口驱动，无法通过device tree动态选择串口）。 11](#_Toc493167957)

[3.2 通过device tree动态选择的方式（和串口驱动配合使用，可以通过dts的stdout-path或者linux,stdout-path指定使用哪个串口）。 11](#_Toc493167958)

[基于serial framework编写串口驱动的方法 12](#_Toc493167959)

[8250例子详解 12](#_Toc493167960)

[8250串口的初始化 12](#_Toc493167961)

[串口打开流程 37](#_Toc493167962)

[串口发送流程 38](#_Toc493167963)

[串口接收流程 39](#_Toc493167964)

[console和串口的关系 40](#_Toc493167965)

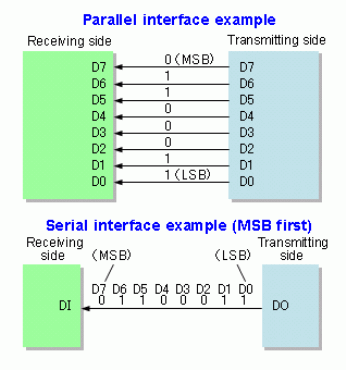
[参考资料 42](#_Toc493167966)

# 串口子系统

## 概念解释

### 1.1 串行通信（Serial communication）

在通信和计算机科学中，Serial communication是一个通用概念，泛指所有的串行的通信协议，如RS232、USB、I2C、SPI、1-Wire、Ethernet等。这里的串行（serial），是相对并行通信（parallel communication）来说的，如下图：



### 1.2同步串行通信（Synchronous serial communication）和异步串行通信（Asynchronous serial communication）

接收方接收到一长串的、表示0/1电平跳变的信号之后，怎么还原出有效的信息呢？有两种方法：

1）发送端在发送串行数据的同时，提供一个时钟信号，并按照一定的约定（例如在时钟信号的上升沿的时候，将数据发送出去）发送数据，接收端根据发送端提供的时钟信号，以及大家的约定，接收数据。这就是常说的同步串行通信（Synchronous serial communication），I2C、SPI等有时钟信号的协议，都属于这种通信方式。

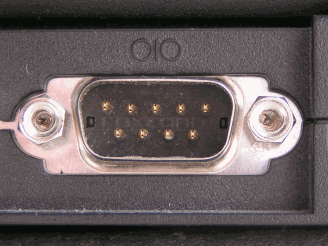
2）发送端在数据发送之前和之后，通过特定形式的信号（例如START信号和STOP信号），告诉接收端，可以开始（或者停止）接收数据了。与此同时，收发两方会约定一个数据发送的速度（就是大名鼎鼎的波特率），发送端在发送START信号之后，就按照固定的节奏发送串行数据，与此同时，接收端在收到START信号之后，也按照固定的节奏接收串行数据。这就是常说的异步串行通信（Asynchronous serial communication），我们本节的主角----串口通信，就是这种通信方式。

注1：严格意义上，START/STOP形式的异步串行通信，称作“asynchronous start-stop communication”，但由于这种形式在异步串行通信里面使用太广泛了，二者也就混为一谈、不加区分了。

注2：根据同步方式的不同，串口也包括异步串口和同步串口两种。本文所指的串口，都是只异步串口。

### 1.3 串口（Serial port）和RS-232

在计算机的世界里，“串口”是使用串行的方式进行数据传输的一种接口，它是相对于计算机的并行接口来说的。虽然串行通信可以通过多种技术实现（如USB、Ethernet等），但由于历史原因，我们所说的串口，特指符合RS-232规范的接口，下图就是我们比较常见的一种：



串口是用于实现串行通信的物理接口的一种统称，但只有名称还远远不够，通信的双方需要一些约定和规范，才能正确的进行数据传输，例如需要哪些信号线、信号的电平如何、接口的形状如何、数据的编码格式、数据传输的速率、等等。

1）RS-232（有时候也称作RS-232-C，C是版本）规范，定义“硬件相关”的特性，包括：

**电气信号特性，如信号电平（-3V～-15V表示逻辑1，+3～+15V表示逻辑0）等；**

**接口特性，如连接器（connectors）的定义（9-pin的 DE-9 Male/Female，25-pin的DB-25 Male/Female等）、管脚信号的定义、等等；**

**电缆（cable）的特性，如电缆的长度（RS-232没有显式的限制电缆的长度，但限制了电容，效果一样）等；**

**等等。**

2）串口硬件，定义“软件相关”的特性，包括：

**数据的编码格式（character encoding），就是上面提到的“asynchronous start-stop”格式，也即我们所熟知的的“起始位、数据位、校验位、停止位“；**

**数据传输的bit rates，也即我们熟悉的波特率（baud rate）。**

注3：说来奇怪，RS-232规范规定了串口通信有关的方方面面的特性，唯独没有规定数据传输相关的编码格式和bit rates。可能是硬件的差异太大，以至于该规范无法完全覆盖。

注4：虽然RS-232规范没有规定bit rates，但它做了一个要求----不能超过20Kbps，虽然从现在来看，这就是瞎扯，呵呵。

注5：RS-232定义了各种形态的串口接口，如DE-9、DB-25等等，但这大多和上古时代的通信有关场景以有关，随着数字通信的普及，以及USB等协议的蚕食，这些庞然大物已经越来越少见了。反而在嵌入式场景中，一些简化的形态，如4线（VCC/RX/TX/GND）串口等，反而使用的比较多。

### 1.4 UART(UniversalAsynchronous Receiver/Transmitter)

由于硬件的差异太大，RS-232并没有规定串口通信的编码格式和bit rates。因此，这一块的实现，完全由具体的硬件（或者对应的软件负责）。

在早期的产品中，大多使用软件的方式，以一定的速率（bit rates），产生出符合编码格式的bit流。但后来为了提升性能，很多产品（如IBM的PC）使用了专门的硬件模块，完成这个事情，这就是我们经常挂在嘴边的UART。

至此，串口通信有关的术语已经悉数登场，简单总结一下吧：

**串行通信（serial communication），泛指使用bit流的形式进行数据通信的方法。**

**同步串行通信（Synchronous serial communication）和异步串行通信（Asynchronous serial communication），串行通信的不同实现方法，主要区别在于同步的方式。**

**串口（serial port），特指使用“异步串行通信”的方法进行数据通信的接口，主要是从硬件的角度描述的。**

**RS-232，定义两个串口之间通信行为的一个规范，偏向于电平、连接器、电缆等硬件特性。**

**UART（Universal Asynchronous Receiver/Transmitter），一个硬件模块，根据串口硬件所规定编码格式、bit rate，产生处通信所需的bit流。**

### 1.5 TTL电平转换以及USB转串口

当前，串口应用最广泛的场景，就是嵌入式产品和PC之间的一些数据交互，如debug输出、firmware更新等。但这里有一个比较麻烦的事情：

**PC串口符合RS-232规范，其电平是“-3V～-15V/+3～+15V”。而大多数的嵌入式产品，都是使用TTL电平（如0v/3.3v）。因此，这两种电平无法直接连接。怎么办呢？增加一个电平转换电路就是了，于是嵌入式产品和PC之间的连接就变成了如下的形式：   
       PC(串口)<---------->电平转换电路<---------->嵌入式产品(串口)**

还有更麻烦的事情，随着技术的进步，串口这样的大杀器，渐渐地被PC抛弃了，怎么办呢？USB转串口应运而生了，连接方式变成了：

**PC(USB接口)<---------->USB转串口<---------->嵌入式产品(串口)**

### 波特率（baud rate）和比特率（bit rate）

理解了串口中这些既熟悉又陌生的术语之后，我们再来看看波特率（baud rate）。

说实话，在数据通信中，比特率（bit rate）比较容易理解，就是一定时间内，能够传输多少个bit。例如bps，就是bit per second的缩写。那什么事波特率呢？

在通信中，波特率也称作符号速率（symbol rate），指的的是“数据变化”的速率。说着很拗口，我们举个例子：

**在计算机系里，小杨和小李是一对好基友，不过小杨是学霸、小李是学渣。所以，期末考试到了，小杨决定“鼎力相助”。怎么办呢？**

**二人约定，考试时，小杨携带黑色和白色两支笔，根据两支笔出现的情况，表示A、B、C、D四种答案，即：   
    白色的笔没有出现    黑色的笔没有出现    A   
    白色的笔没有出现    黑色的笔出现          B   
    白色的笔出现          黑色的笔没有出现    C   
    白色的笔出现          黑色的笔出现           D**

**同时约定，在考试开始1小时之后，小杨从第1道选择题开始，以每分钟更换一次的速度，更换答案。小李按照这个速度，以及大家的约定，通过观察两支笔出现的情况，获得答案。**

确实是个好方法，不过仔细想想，这其实是一个典型的异步通信过程。通信的过程中，答案更新的速度（每分钟1次），就是我们所说的baud rate（或者symbol rate），即1 bd per minute（可以把bd看着baud的单位）。

与此同时，每次更新，传递了多少信息呢？表面上看是A、B、C、D，本质上是由白和黑所代表的两个bit，00、01、10或者11。因此，每次更新传递2个bit的信息，所以bit rate就是2 bits per minute。

上面的例子中，通信的波特率和比特率是不同的，分别为1和2（per minute），而有些通信系统，例如我们所熟知的串口通信，它们却是一样的，例如我们说115200的波特率，实际上的比特率也是115200。因为一次只传输1个bit（0或者1）。

### 帧格式和波特率误差

我们知道，串口通信的数据格式包括start bit、data bits、parity bit和stop bits，其中：

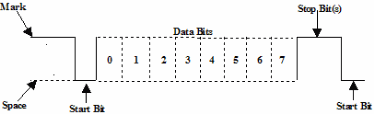
**start bit固定为1bit；**

**data bits可以为5、6、7、8或者9bits，不过常用的都是8bits；**

**parity bit是非必须的，一般为0bit；**

**stop bit可以是1bit和2bits两种，一般都是1bit。**

图示如下：



这种异步通信的方法，有一个坏消息，也有一个好消息：

**坏消息是，受系统时钟精度、分频系数等影响，波特率可能不精确，存在误差。而在一帧数据的传输过程中，误差会累积。**

**好消息是，如果误差累积的不多，受start/stop信号的矫正作用，下一帧数据中，可将累积误差清零。**

**关于误差，我们再稍微详细分析一下。**

根据我们的常识，信号是在中间点采样，因此，在一帧数据中，如果误差累积超过(1 / bard rate / 2)的时间，则数据会采样错误。假设理想的baud rate是BDi，实际的baud rate是BDr，则：

**每一个bit的误差时间是：1 / BDi – 1 / BDr。**

**10个bit的误差时间是：10 \* (1 / BDi – 1 / BDr)。**

**因此，安全的误差范围是：abs(10 \* (1 / BDi – 1 / BDr)) < 1 / BDi / 2。**

以115200的波特率为例，带入上面的公式：

**abs(10 \* (1 / 115200– 1 / BDr)) < 1 / 115200 / 2**

**-1 / 115200 / 2 < 10 \* (1 / 115200– 1 / BDr) < 1 / 115200 / 2**

**(20 / 21) \* 115200 < BDr < (20 / 19) \* 115200**

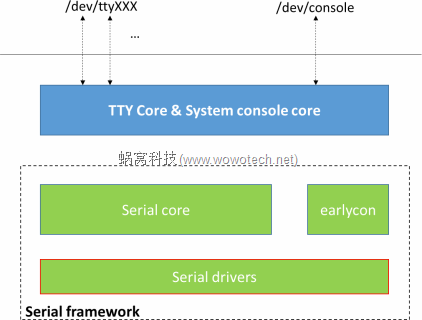
因此，115200波特率所容许的误差范围是20 / 21 ~ 20 / 19，即+/- 5%。其它波特率也可以带入公式去计算。另外，如果双方都有误差的话怎么办？嘿嘿，你懂的，除2吧。

## Linux serial framework

串口设备（serial or uart，后面不再区分）是TTY设备的一种，Linux kernel为了方便串口驱动的开发，在TTY framework的基础上，封装了一层串口框架（serial framework）。该框架尽可能的屏蔽了TTY有关的技术细节（比较难懂），驱动工程师在编写串口驱动的时候，只需要把精力放在串口以及串口控制器本身即可。

### 软件架构

Linux kernel serial framework位于“drivers/tty/serial”目录中，其软件架构如图：



Serial core是Serial framework的核心实现，对上封装、屏蔽TTY的技术细节，对下为具体的串口驱动提供简单、统一的编程API。

earlycon（early console）是serial framework中比较新的一个功能，它基于Kernel system console的框架，提供了一种比较简单的控制台实现方式。

最后就是具体的串口驱动（Serial drivers）

### serial core

#### 2.1 功能介绍

serial core主要实现如下三类功能（任何一个framework的core模块都提供类似的功能）：

**1）将串口设备有关的物理对象（及其操作方法）封装成一个一个的数据结构，以达到用软件语言描述硬件的目的。**

**2）向底层driver提供串口驱动的编程接口。**

**3）基于TTY framework所提供的TTY driver的编写规则，将底层driver看到的serial driver，转换为TTY driver，并将所有的serial操作，转换为对应的tty操作。**

本文将重点介绍1）和2）两类功能，第3）类，属于TTY core的内部逻辑

#### 2.2 关键数据结构

数据结构是一个软件模块的灵魂和骨架，而对设备驱动来说，数据结构一般和具体的硬件实体对应，例如：

**假如一个soc中有5个串口控制器（也可称作uart控制器，后面我们不再区分），每个uart控制器都可引出一个串口（uart port）。那么：**

**每个uart控制器，都是一个platform device，它们可由同一个patform driver驱动；**

**相对于uart控制器实实在在的存在，我们更为熟悉的串口（uart port），可以看作虚拟的设备，serial core将它们抽象为“struct uart\_port”，并在platform driver的probe接口中，注册到kernel；**

**和platform device类似，这些虚拟的串口设备，也可由同一个虚拟的driver驱动，这就是serial core中的“struct uart\_driver”。**

下面我们将跟随上面的思路，介绍serial core提供的数据结构（具体“include/linux/serial\_core.h”）。

##### **2.2.1 struct uart\_port**

在serial framework中，**struct uart\_port抽象虚拟的串口设备**（具体的串口控制器，则为实实在在的硬件设备），这是一个庞大的数据结构，存放了五花八门的、各式各样的、有新有旧的、有用没有的和串口设备有关的信息，例如：

1）最基本、最必须的，需要驱动工程师根据实际的硬件自行填充的字段

**dev，父设备的指针，通常是串口控制器所对应的platform device；**

**type，该串口的类型，是以PORT\_为前缀的一个宏定义，可以根据需要在include/uapi/linux/serial\_core.h中定义；**

**ops，该串口的操作函数集（struct uart\_ops类型的指针），具体可参考3.2.3；**

**iotype，该串口的I/O类型，例如UPIO\_MEM32（常用的通过寄存器访问的uart控制器）；**

**mapbase，对应MEM类型的串口，保存它的寄存器基址（物理地址），一般是从DTS中解析得到的**

**membase，从mapbase ioremap得来（虚拟地址）；**

**irq、irqflags，该串口对应的中断号（以及相应的终端flags）， 一般是从DTS中解析得到的；**

**line，该串口的编号，和字符设备的次设备号等有关。**

2）和运行时状态有关的字段

l**ock，一个自旋锁（spinlock\_t类型），用于对该数据结构进行访问保护；**

**icount，一个struct uart\_icount类型变量，用于保存该串口的统计信息，例如收发数据的统计等；**

**cons，console指针，如果该串口被注册为system console的话，将对应的console指针保存在这里；**

**state，struct uart\_state指针，具体请参考3.2.2。**

3）一些有用的函数指针（driver实现，serial core调用），例如

**serial\_in，读取该串口的某个寄存器；**

**serial\_out，向该串口的某个寄存器写入某一value；**

**最后，serial core根据这两个函数指针，封装出两个公共的寄存器访问接口：serial\_port\_in和serial\_port\_out，以方便driver使用。**

注1：struct uart\_port中的内容非常多，因此在编写串口驱动的时候，要把握一个原则：不到万不得已的时候，不要新定义变量，多去struct uart\_port找找，很有可能就找到了你想要的东西。

##### **2.2.2 struct uart\_state**

**struct uart\_state中保存了串口使用过程中的动态信息**（它们的生命周期是串口被打开到被关闭的过程），包括：

**port，对应的struct tty\_port变量（uart port是tty port的一个特例）；**

**pm\_state，电源管理有关的状态，例如UART\_PM\_STATE\_ON、UART\_PM\_STATE\_OFF和UART\_PM\_STATE\_UNDEFINED三种；**

**xmit，用于保存TX数据的环形缓冲区（struct circ\_buf，具体可参考include/linux/circ\_buf.h）；**

**uart\_port，struct uart\_port类型的指针，指向所属的串口。**

##### **2.2.3 struct uart\_ops**

使用**struct uart\_port抽象串口**的同时，serial core将**串口有关的操作函数封装在struct uart\_ops**中，底层驱动根据实际硬件情况，填充这些函数，serial core在合适的时候，帮忙调用。

串口设备是一种非常复杂的设备， struct uart\_ops结构也非常庞大，这里简单介绍一些常用的（可参考kernel的帮助文档----Documentation/serial/driver）：

**startup，打开串口设备的时候，serial core会调用该接口，driver可以在这里进行串口的初始化操作，例如申请中断资源、使能clock、使能接收，等等；**

**shutdown，startup的反操作，在串口设备被关闭的时候调用；**

**start\_tx，每当有一笔新的数据需要通过串口发送出去的时候，serial core会先把数据保存在TX的buffer中（参考3.2.2的介绍），然后调用start\_tx通知driver。driver需要在该接口中，根据当前的状态（TX是否正在进行），决定是否需要发起一次传输；**

**stop\_tx，停止正在进行中的TX；**

**stop\_rx，停止RX；**

**tx\_empty，判断硬件的TX FIFO是否为空，如果是，则返回TIOCSER\_TEMT，否则返回0；**

**.set\_mctrl，设置modem的control line，可以留空；**

**.set\_termios，设置串口的termios（例如波特率、数据位、停止位等）。**

##### **2.2.4 struct uart\_driver**

上面介绍的几个数据结构，都在竭力描述串口设备，相应的，按照设备模型的惯例，需要一个和设备对应的、**抽象driver数据结构，就是struct uart\_driver**：

**struct uart\_driver {   
        struct module           \*owner;   
        const char              \*driver\_name;   
        const char              \*dev\_name;   
        int                      major;   
        int                      minor;   
        int                      nr;   
        struct console          \*cons;**

**/\*   
         \* these are private; the low level driver should not   
         \* touch these; they should be initialised to NULL   
         \*/   
        struct uart\_state       \*state;   
        struct tty\_driver       \*tty\_driver;   
};**

该数据结构非常简单，一般情况下，只要关注如下的字段：

**driver\_name，driver的名称；**

**dev\_name，对应的设备名，例如“ttyS”；**

**nr，该驱动可以支持的串口的数量（serial core会根据这个值为每一个串口分配一些内部使用资源）；**

**major、minor，主、次设备号，可以不指定（这样的话，TTY core会帮忙动态分配）；**

**cons，如果需要将某一个串口当作system console，可以在driver中定义struct console变量，并将它的指针保存在这里，serial core会在注册串口的时候帮忙将console注册到系统中。**

#### 2.3 向具体driver提供的用于编写串口驱动的API

数据结构抽象完毕后，serial core向下层的driver提供了方便的编程API，主要包括：

1）uart driver有关的API

**int uart\_register\_driver(struct uart\_driver \*uart);   
void uart\_unregister\_driver(struct uart\_driver \*uart);**

uart\_register\_driver，将定义并填充好的uart driver注册到kernel中，一般在驱动模块的init接口中被调用。

uart\_unregister\_driver，注销uart driver，在驱动模块的exit接口中被调用。

2）uart port有关的API

**int uart\_add\_one\_port(struct uart\_driver \*reg, struct uart\_port \*port);   
int uart\_remove\_one\_port(struct uart\_driver \*reg, struct uart\_port \*port);   
int uart\_match\_port(struct uart\_port \*port1, struct uart\_port \*port2);   
  
int uart\_suspend\_port(struct uart\_driver \*reg, struct uart\_port \*port);   
int uart\_resume\_port(struct uart\_driver \*reg, struct uart\_port \*port);   
  
static inline int uart\_tx\_stopped(struct uart\_port \*port)**

**extern void uart\_insert\_char(struct uart\_port \*port, unsigned int status,   
                 unsigned int overrun, unsigned int ch, unsigned int flag);**

uart\_add\_one\_port、uart\_remove\_one\_port，添加/删除一个uart port，一般在platform driver的probe/remove中被调用。

uart\_suspend\_port、uart\_resume\_port，suspend/resume uart port，在电源管理状态切换的时候被调用。

uart\_tx\_stopped，判断某一个uart port的tx是否处于停止状态。

uart\_insert\_char，驱动从串口接收到一个字符之后，可以调用该接口把该字符放到RX buffer中（相比tty\_insert\_flip\_char，可以进行一些overrun的处理）。

3）system console有关的API

**struct tty\_driver \*uart\_console\_device(struct console \*co, int \*index);   
void uart\_console\_write(struct uart\_port \*port, const char \*s,   
                        unsigned int count,   
                        void (\*putchar)(struct uart\_port \*, int));**

uart\_console\_device，通过console指针获取tty driver指针的帮助函数，通常情况下会把该函数赋值给串口console的.device指针，例如：

**static struct console xxx\_console = {   
        …   
        .device = uart\_console\_device,   
        …   
};**

uart\_console\_write，向串口发送一个字符串的辅助接口，通常在console的.write接口中被调用。

4）其它辅助类的API

**#define uart\_circ\_empty(circ)           ((circ)->head == (circ)->tail)   
#define uart\_circ\_clear(circ)           ((circ)->head = (circ)->tail = 0)**

**#define uart\_circ\_chars\_pending(circ)   \   
        (CIRC\_CNT((circ)->head, (circ)->tail, UART\_XMIT\_SIZE))**

**#define uart\_circ\_chars\_free(circ)      \   
        (CIRC\_SPACE((circ)->head, (circ)->tail, UART\_XMIT\_SIZE))**

为了方便driver操作环形缓冲区，serial core定义了一些状态判断的宏，例如是否为空（uart\_circ\_empty）、是否为初始状态（uart\_circ\_clear）、缓冲区中有效数据的个数（uart\_circ\_chars\_pending）、缓冲区中空闲空间的个数（uart\_circ\_chars\_free）、等等。

## earlycon

early console是linux serial framework提供的一个可在kernel启动早期使用的console，最早可以在“start\_kernel-->setup\_arch-->parse\_early\_param”之后就可使用，对于kernel早期的debug很有帮助。

根据复杂程度，serial framework提供了两种API，供底层的driver使用。

### 3.1 静态定义的方式（不需要串口驱动，无法通过device tree动态选择串口）。

这种方法很简单，相关的API包括：

1）EARLYCON\_DECLARE（定义一个early console）

#define EARLYCON\_DECLARE(\_name, fn)     OF\_EARLYCON\_DECLARE(\_name, "", fn)

该函数有两个参数：\_name是early console的名称，需要和后续在命令行参数中指定的一致；fn是一个原型为“int     (\*setup)(struct earlycon\_device \*, const char \*options);”的回调函数，用于在kernel启动的时候配置改console。

2）setup

driver需要在setup回调中做两件事情：

a）初始化early console有关的硬件，例如ioremap、初始化串口控制器、等等。

b）实现一个console->write[2]接口，并将函数指针保存在setup第一个参数的相应指针中（device->con->write）

3）命令行参数（earlycon=xxx）

完成上面1）和2）之后，一个earlycon driver就完成了，serial framework会在kernel启动的时候，调用我们提供的setup接口，对early console初始化，然后调用system console core提供的register\_console接口，帮忙注册console[2]。

如果需要使用这个console，可在kernel启动的时候传入命令行参数“earlycon=xxx”，xxx为early console的名称。

### 3.2 通过device tree动态选择的方式（和串口驱动配合使用，可以通过dts的stdout-path或者linux,stdout-path指定使用哪个串口）。

自行研读代码。

## 基于serial framework编写串口驱动的方法

可参考如下文章：

[X-012-KERNEL-serial early console的移植](http://www.wowotech.net/x_project/kernel_earlycon_porting.html)

[X-016-KERNEL-串口驱动开发之驱动框架](http://www.wowotech.net/x_project/serial_driver_porting_1.html)

[X-017-KERNEL-串口驱动开发之uart driver框架](http://www.wowotech.net/x_project/serial_driver_porting_2.html)

[X-018-KERNEL-串口驱动开发之serial console](http://www.wowotech.net/x_project/serial_driver_porting_3.html)

[X-019-KERNEL-串口驱动开发之数据收发](http://www.wowotech.net/x_project/serial_driver_porting_4.html)

# 8250例子详解

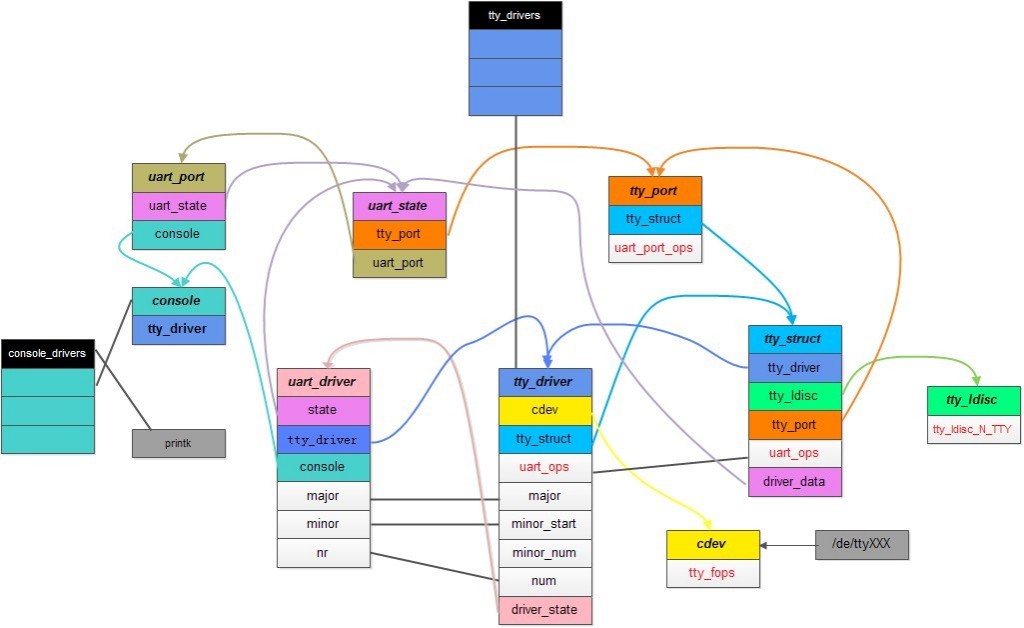
## http://images2015.cnblogs.com/blog/886842/201604/886842-20160419213308570-324396923.png8250串口的初始化

**8250串口的初始化：**

（1）定义uart\_driver、uart\_ops、uart\_port等结构体的实例并在适当的地方更具具体的硬件驱动情况初始化他们，当然具体设备xxx的驱动可以将这些结构体套在新定义的xxx\_uart\_driver、xxx\_uart\_ops、xxx\_uart\_port之内。

（2）在模块初始化调用uart\_register()和uart\_add\_one\_port()以注册UART驱动并添加端口，在模块卸载时调用uart\_unregister\_driver()和uart\_remove\_one\_port()以注销UART驱动以移除端口。

（3）根据具体硬件的uart\_ops中的成员函数，这些函数的实现成为UART驱动的主体工作。  
1.串口结构体,他们之间的关系如图所示：

  
//（1）串口驱动结构体  
struct uart\_driver {    
    struct module   \*owner; //模块所有者     
    const char  \*driver\_name;   //驱动名     
    const char  \*dev\_name;  //设备名     
    int  major; //主设备号     
    int  minor; //次设备号     
    int  nr;    //支持串口个数     
    struct console  \*cons;//控制台设备     
    struct uart\_state   \*state; //串口状态     
    struct tty\_driver   \*tty\_driver; //tty设备     
};

//（2）串口端口结构体  
struct uart\_port {    
    spinlock\_t  lock;    
    unsigned long   iobase; //io端口基地址     
    unsigned char \_\_iomem   \*membase; //内存端口基地址     
    unsigned int    (\*serial\_in)(struct uart\_port \*, int); //串口读函数   
    void    (\*serial\_out)(struct uart\_port \*, int, int); //串口写方法  
    void    (\*set\_termios)(struct uart\_port \*,struct ktermios \*new,struct ktermios \*old); //串口配置方法函数  
    void    (\*pm)(struct uart\_port \*, unsigned int state,unsigned int old);    
    unsigned int    irq;    //中断号     
    unsigned long   irqflags;   //中断标志     
    unsigned int    uartclk;   //串口时钟  
    unsigned int    fifosize;   //fifo大小         
    unsigned char   x\_char;    
    unsigned char   regshift;   //寄存器偏移值         
    unsigned char   iotype; //io访问类型     
    unsigned char   unused1;    
    unsigned int    read\_status\_mask;    
    unsigned int    ignore\_status\_mask;    
    struct uart\_state   \*state; //uart\_state结构体      
    struct uart\_icount  icount; //串口使用计数     
    struct console  \*cons;  //console控制台     
#if defined(CONFIG\_SERIAL\_CORE\_CONSOLE) || defined(SUPPORT\_SYSRQ)     
    unsigned long   sysrq;    
#endif     
    upf\_t   flags;    
    unsigned int    mctrl;    
    unsigned int    timeout;    
    unsigned int    type; //串口类型   
    const struct uart\_ops   \*ops;   //串口操作函数集     
    unsigned int    custom\_divisor;    
    unsigned int    line;   //端口号     
    resource\_size\_t mapbase; //串口寄存器基地址（物理地址）  
    struct device   \*dev;   //设备文件     
    unsigned char   hub6;    
    unsigned char   suspended;    
    unsigned char   irq\_wake;    
    unsigned char   unused[2];    
    void    \*private\_data;    
};

//（3）操作函数集  
struct uart\_ops {    
    unsigned int    (\*tx\_empty)(struct uart\_port \*);    //发送缓冲区为空     
    void    (\*set\_mctrl)(struct uart\_port \*, unsigned int mctrl);   //设置串口modem控制模式     
    unsigned int    (\*get\_mctrl)(struct uart\_port \*);   //获取串口modem控制模式     
    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);    
    void    (\*stop\_rx)(struct uart\_port \*); //停止接收     
    void    (\*enable\_ms)(struct uart\_port \*);   //使能modem状态信息     
    void    (\*break\_ctl)(struct uart\_port \*, int ctl);    
    int (\*startup)(struct uart\_port \*); //打开串口     
    void    (\*shutdown)(struct uart\_port \*);    //关闭串口     
    void    (\*flush\_buffer)(struct uart\_port \*);    
    void    (\*set\_termios)(struct uart\_port \*, struct ktermios \*new,struct ktermios \*old);  //设置串口参数     
    void    (\*set\_ldisc)(struct uart\_port \*, int new);    
    void    (\*pm)(struct uart\_port \*, unsigned int state,unsigned int oldstate);    
    int (\*set\_wake)(struct uart\_port \*, unsigned int state);    
    const char \*(\*type)(struct uart\_port \*);    
    void    (\*release\_port)(struct uart\_port \*);    //释放端口     
    int (\*request\_port)(struct uart\_port \*);    //请求端口     
    void    (\*config\_port)(struct uart\_port \*, int);    //配置端口     
    int (\*verify\_port)(struct uart\_port \*, struct serial\_struct \*); //校验端口     
    int (\*ioctl)(struct uart\_port \*, unsigned int, unsigned long);  //控制     
#ifdef CONFIG\_CONSOLE\_POLL     
    void    (\*poll\_put\_char)(struct uart\_port \*, unsigned char);    
    int (\*poll\_get\_char)(struct uart\_port \*);    
#endif     
};

//（4）uart\_state  
struct uart\_state {    
    struct tty\_port port;    
    int     pm\_state;    
    struct circ\_buf xmit;    
    struct tasklet\_struct   tlet;    
    struct uart\_port    \*uart\_port;//指向对应的串口结构    
};

2.串口初始化  
static int \_\_init serial8250\_init(void)  
{  
 int ret;

 if (nr\_uarts > UART\_NR)//UART\_NR =3  
  nr\_uarts = UART\_NR;//串口数量最多设为3个

 printk(KERN\_INFO "Serial: 8250/16550 driver, ""%d ports, IRQ sharing %sabled\n", nr\_uarts,share\_irqs ? "en" : "dis");

 serial8250\_reg.nr = UART\_NR;//串口数量  
 /\*  
  static struct uart\_driver serial8250\_reg = {  
  .owner   = THIS\_MODULE,  
  .driver\_name  = "serial",  
  .dev\_name  = "ttyS",  
  .major   = TTY\_MAJOR,//主设备号是4  
  .minor   = 64,  
  .cons   = SERIAL8250\_CONSOLE,//终端  
 };  
   
 #define SERIAL8250\_CONSOLE &serial8250\_console  
   
 static struct console serial8250\_console = {  
  .name  = "ttyS",  
  .write  = serial8250\_console\_write,  
  .device  = uart\_console\_device,  
  .setup  = serial8250\_console\_setup,//设置串口波特率，也就是设置串口。很重要，里面涉及到平台特性，波特率相关。  
  .early\_setup = serial8250\_console\_early\_setup,  
  .flags  = CON\_PRINTBUFFER | CON\_ANYTIME,  
  .index  = -1,  
  .data  = &serial8250\_reg,  
 };  
 \*/  
 //函数定义在serial\_core.c中  
 //注册uart串口驱动，完善uart\_driver结构serial8250\_reg的uart\_state成员及tty\_driver成员，并注册tty驱动  
 ret = uart\_register\_driver(&serial8250\_reg);  
#endif  
 if (ret)  
  goto out;  
   
 //创建一个platform\_device结构：serial8250\_isa\_devs  
 serial8250\_isa\_devs = platform\_device\_alloc("serial8250",PLAT8250\_DEV\_LEGACY);  
 if (!serial8250\_isa\_devs) {  
  ret = -ENOMEM;  
  goto unreg\_uart\_drv;  
 }  
   
  //将该结构serial8250\_isa\_devs注册到总线上  
 ret = platform\_device\_add(serial8250\_isa\_devs);  
 if (ret)  
  goto put\_dev;  
    
 //对uart\_8250\_port结构serial8250\_reg[]初始化,即对3个串口的uart\_port结构初始化，并添加端口  
 serial8250\_register\_ports(&serial8250\_reg, &serial8250\_isa\_devs->dev);  
   
 /\*  
 static struct platform\_driver serial8250\_isa\_driver = {  
  .probe  = serial8250\_probe,  
  .remove  = \_\_devexit\_p(serial8250\_remove),  
  .suspend = serial8250\_suspend,  
  .resume  = serial8250\_resume,  
  .driver  = {  
   .name = "serial8250",  
   .owner = THIS\_MODULE,  
  },  
 };  
 \*/  
 //注册设备,会调用serial8250\_probe()。怎样调用的serial8250\_probe？？？？？？？  
 ret = platform\_driver\_register(&serial8250\_isa\_driver);  
 if (ret == 0)  
  goto out;

 platform\_device\_del(serial8250\_isa\_devs);  
put\_dev:  
 platform\_device\_put(serial8250\_isa\_devs);  
unreg\_uart\_drv:  
#ifdef CONFIG\_SPARC  
 sunserial\_unregister\_minors(&serial8250\_reg, UART\_NR);  
#else  
 uart\_unregister\_driver(&serial8250\_reg);  
#endif  
out:  
 return ret;  
}

//注册串口驱动  
int uart\_register\_driver(struct uart\_driver \*drv)  
{  
 struct tty\_driver \*normal;  
 int i, retval;

 BUG\_ON(drv->state);

 //为串口的uart\_driver结构分配要指向的uart\_state结构的空间\*串口数量  
 drv->state = kzalloc(sizeof(struct uart\_state) \* drv->nr, GFP\_KERNEL);  
 if (!drv->state)  
  goto out;  
   
 //为uart\_driver结构分配要指向的tty\_driver结构的空间  
 /\*struct tty\_driver \*alloc\_tty\_driver(int lines)  
 {  
  struct tty\_driver \*driver;  
   
  driver = kzalloc(sizeof(struct tty\_driver), GFP\_KERNEL);//分配空间  
  if (driver) {  
   kref\_init(&driver->kref);  
   driver->magic = TTY\_DRIVER\_MAGIC;//tty\_driver的魔数  
   driver->num = lines;//串口数量3个  
  }  
  return driver;  
 }\*/  
 normal = alloc\_tty\_driver(drv->nr);  
 if (!normal)  
  goto out\_kfree;

 drv->tty\_driver = normal;//赋值给uart\_driver结构的tty\_driver成员  
 //对tty\_driver成员指向的结构初始化  
 normal->owner  = drv->owner;//THIS\_MODULE  
 normal->driver\_name = drv->driver\_name;//"serial"  
 normal->name  = drv->dev\_name;//"ttyS"  
 normal->major  = drv->major;//主设备号：TTY\_MAJOR=4  
 normal->minor\_start = drv->minor;//次设备号：64  
 normal->type  = TTY\_DRIVER\_TYPE\_SERIAL;//#define TTY\_DRIVER\_TYPE\_SERIAL 0x0003  
 normal->subtype  = SERIAL\_TYPE\_NORMAL;//#define SERIAL\_TYPE\_NORMAL  1(定义在include/linux/tty\_driver.h)  
 normal->init\_termios = tty\_std\_termios;//终端的标准配置（drivers/tty/tty\_io.c）  
 normal->init\_termios.c\_cflag = B9600 | CS8 | CREAD | HUPCL | CLOCAL;//配置c\_cflag  
 normal->init\_termios.c\_ispeed = normal->init\_termios.c\_ospeed = 9600;//波特率设为9600  
 normal->flags  = TTY\_DRIVER\_REAL\_RAW | TTY\_DRIVER\_DYNAMIC\_DEV;  
 normal->driver\_state    = drv;//指向相应的uart\_driver  
 tty\_set\_operations(normal, &uart\_ops);//将tty\_driver结构的normal操作指向uart\_ops  
 /\*void tty\_set\_operations(struct tty\_driver \*driver,const struct tty\_operations \*op)  
 {  
  driver->ops = op;  
 };  
 static const struct tty\_operations uart\_ops = {  
  .open  = uart\_open,  
  .close  = uart\_close,  
  .write  = uart\_write,  
  .put\_char = uart\_put\_char,  
  .flush\_chars = uart\_flush\_chars,  
  .write\_room = uart\_write\_room,  
  .chars\_in\_buffer= uart\_chars\_in\_buffer,  
  .flush\_buffer = uart\_flush\_buffer,  
  .ioctl  = uart\_ioctl,  
  .throttle = uart\_throttle,  
  .unthrottle = uart\_unthrottle,  
  .send\_xchar = uart\_send\_xchar,  
  .set\_termios = uart\_set\_termios,  
  .set\_ldisc = uart\_set\_ldisc,  
  .stop  = uart\_stop,  
  .start  = uart\_start,  
  .hangup  = uart\_hangup,  
  .break\_ctl = uart\_break\_ctl,  
  .wait\_until\_sent= uart\_wait\_until\_sent,  
 #ifdef CONFIG\_PROC\_FS  
  .proc\_fops = &uart\_proc\_fops,  
 #endif  
  .tiocmget = uart\_tiocmget,  
  .tiocmset = uart\_tiocmset,  
  .get\_icount = uart\_get\_icount,  
 #ifdef CONFIG\_CONSOLE\_POLL  
  .poll\_init = uart\_poll\_init,  
  .poll\_get\_char = uart\_poll\_get\_char,  
  .poll\_put\_char = uart\_poll\_put\_char,  
 #endif  
 };  
 \*/

 for (i = 0; i < drv->nr; i++) {//根据串口数量依次扫描drv->nr=3  
  struct uart\_state \*state = drv->state + i;//找到每个串口的uart\_state地址  
  struct tty\_port \*port = &state->port;//每个串口有一个tty\_port结构  
    
  //初始化每个串口的tty\_port结构  
  tty\_port\_init(port);  
  port->ops = &uart\_port\_ops;  
  port->close\_delay     = 500; /\* .5 seconds \*/  
  port->closing\_wait    = 30000; /\* 30 seconds \*/  
  tasklet\_init(&state->tlet, uart\_tasklet\_action,(unsigned long)state);  
 }  
   
 //注册tty驱动  
 retval = tty\_register\_driver(normal);  
 if (retval >= 0)  
  return retval;

 put\_tty\_driver(normal);  
out\_kfree:  
 kfree(drv->state);  
out:  
 return -ENOMEM;  
}

//初始化串口的uart\_port结构  
static void \_\_init serial8250\_register\_ports(struct uart\_driver \*drv, struct device \*dev)  
{  
 int i;  
   
 //对uart\_8250\_port结构的serial8250\_ports进行初始化  
 for (i = 0; i < nr\_uarts; i++) {//共3个串口  
  struct uart\_8250\_port \*up = &serial8250\_ports[i];  
  up->cur\_iotype = 0xFF;//先初始化为0xFF，在set\_io\_from\_upio（）会赋值成uart\_port->iotype（即UPIO\_MEM=2）  
 }  
   
 //对uart\_8250\_port结构的serial8250\_ports[]接着进行初始化,主要是设置uart\_port字段ops的操作  
 //由于此函数在初始化console时被调用过，所以现在调用的话会直接返回。即串口的uart\_port的ops字段已经赋过值了  
 serial8250\_isa\_init\_ports();

 for (i = 0; i < nr\_uarts; i++) {//共3个串口  
  struct uart\_8250\_port \*up = &serial8250\_ports[i];

  up->port.dev = dev;//指向相应的struct device结构

  if (up->port.flags & UPF\_FIXED\_TYPE)//因为未进行serial8250\_probe(),所以此标志未设置，不进入下边函数  
   serial8250\_init\_fixed\_type\_port(up, up->port.type);//设置串口type=PORT\_AR7，即18  
    
  //向设备添加端口,是在uart\_driver增加一个port，在未进行serial8250\_probe()之前，这个函数的串口配置操作会失败  
  //进行过serial8250\_probe()后，还会再调用此函数，那时就可以配置好串口。  
  uart\_add\_one\_port(drv, &up->port);  
 }  
}

static void \_\_init serial8250\_isa\_init\_ports(void)  
{  
 struct uart\_8250\_port \*up;  
 static int first = 1;  
 int i, irqflag = 0;

 if (!first)//静态变量，serial8250\_console\_init（）第一次进入这个函数，之后serial8250\_init()再进入这个函数就会直接返回  
  return;  
 first = 0;  
   
 //对三个串口的uart\_8250\_port结构serial8250\_ports结构体进行初始化  
 for (i = 0; i < nr\_uarts; i++) {  
  struct uart\_8250\_port \*up = &serial8250\_ports[i];

  up->port.line = i;//0代表串口0,1代表串口1  
  spin\_lock\_init(&up->port.lock);

  init\_timer(&up->timer);//初始化定时器  
  up->timer.function = serial8250\_timeout;//初始化定时器的超时函数

  //ALPHA\_KLUDGE\_MCR needs to be killed.  
  up->mcr\_mask = ~ALPHA\_KLUDGE\_MCR;  
  up->mcr\_force = ALPHA\_KLUDGE\_MCR;  
    
  //初始化uart\_8250\_port指向的uart\_port字段port的操作  
  up->port.ops = &serial8250\_pops;  
  /\*  
  static struct uart\_ops serial8250\_pops = {  
   .tx\_empty = serial8250\_tx\_empty,  
   .set\_mctrl = serial8250\_set\_mctrl,  
   .get\_mctrl = serial8250\_get\_mctrl,  
   .stop\_tx = serial8250\_stop\_tx,  
   .start\_tx = serial8250\_start\_tx,  
   .stop\_rx = serial8250\_stop\_rx,  
   .enable\_ms = serial8250\_enable\_ms,  
   .break\_ctl = serial8250\_break\_ctl,  
   .startup = serial8250\_startup,  
   .shutdown = serial8250\_shutdown,  
   .set\_termios = serial8250\_set\_termios,  
   .set\_ldisc = serial8250\_set\_ldisc,  
   .pm  = serial8250\_pm,  
   .type  = serial8250\_type,  
   .release\_port = serial8250\_release\_port,  
   .request\_port = serial8250\_request\_port,  
   .config\_port = serial8250\_config\_port,  
   .verify\_port = serial8250\_verify\_port,  
  #ifdef CONFIG\_CONSOLE\_POLL  
   .poll\_get\_char = serial8250\_get\_poll\_char,  
   .poll\_put\_char = serial8250\_put\_poll\_char,  
  #endif  
  };  
  \*/  
 }

 if (share\_irqs)//中断是否共享（这里设置成不共享）  
  irqflag = IRQF\_SHARED;  
   
 //条件不满足，不会进来初始化  
 for (i = 0, up = serial8250\_ports;i < ARRAY\_SIZE(old\_serial\_port) && i < nr\_uarts;i++, up++) {  
/\* up->port.iobase   = old\_serial\_port[i].port;  
  up->port.irq      = irq\_canonicalize(old\_serial\_port[i].irq);  
  up->port.irqflags = old\_serial\_port[i].irqflags;  
  up->port.uartclk  = old\_serial\_port[i].baud\_base \* 16;  
  up->port.flags    = old\_serial\_port[i].flags;  
  up->port.hub6     = old\_serial\_port[i].hub6;  
  up->port.membase  = old\_serial\_port[i].iomem\_base;  
  up->port.iotype   = old\_serial\_port[i].io\_type;  
  up->port.regshift = old\_serial\_port[i].iomem\_reg\_shift;  
  set\_io\_from\_upio(&up->port);  
  up->port.irqflags |= irqflag;  
  if (serial8250\_isa\_config != NULL)  
   serial8250\_isa\_config(i, &up->port, &up->capabilities);  
\*/  
 }  
}

int uart\_add\_one\_port(struct uart\_driver \*drv, struct uart\_port \*uport)  
{  
 struct uart\_state \*state;  
 struct tty\_port \*port;  
 int ret = 0;  
 struct device \*tty\_dev;

 BUG\_ON(in\_interrupt());//不能在中断中调用  
   
 //Uart\_port->line就是对uart设备文件序号.它对应的也就是uart\_driver->state数组中的uart\_port->line项.  
 if (uport->line >= drv->nr)//在serial8250\_isa\_init\_ports()已经初始化过，代表端口号，0代表串口0。  
  return -EINVAL;  
   
 //根据串口号找到每个串口对应的uart\_state结构  
 state = drv->state + uport->line;  
 port = &state->port;//通过uart\_state结构找到每个串口的tty\_port结构

 mutex\_lock(&port\_mutex);  
 mutex\_lock(&port->mutex);  
 if (state->uart\_port) {  
  ret = -EINVAL;  
  goto out;  
 }  
   
 //将uart\_state和uart\_port结构相关联起来  
 state->uart\_port = uport;   
 state->pm\_state = -1;

 uport->cons = drv->cons;//将uart\_driver的serial8250\_console成员赋值给uart\_port成员  
 uport->state = state;//uart\_port的state成员指向相应的uart\_state

  //If this port is a console, then the spinlock is already initialised.  
  //检查这个串口是否就是终端，并且此终端是否已经注册完毕  
 if (!(uart\_console(uport) && (uport->cons->flags & CON\_ENABLED))) {  
  spin\_lock\_init(&uport->lock);  
  lockdep\_set\_class(&uport->lock, &port\_lock\_key);  
 }  
   
 //进行port的自动配置，在未进行serial8250\_probe()，串口的port->iobase、port->mapbase、port->membase都为空，所以函数进去会立即返回，即未配置成功。  
 //当进行完serial8250\_probe()函数时，还会调用uart\_add\_one\_port()，再到这个函数配置时就会配置成功  
 uart\_configure\_port(drv, state, uport);

  //然后注册tty\_device.如果用户空间运行了udev或者已经配置好了hotplug.就会在/dev下自动生成设备文件了.  
 tty\_dev = tty\_register\_device(drv->tty\_driver, uport->line, uport->dev);  
 if (likely(!IS\_ERR(tty\_dev))) {//设置设备的唤醒状态  
  device\_init\_wakeup(tty\_dev, 1);  
  device\_set\_wakeup\_enable(tty\_dev, 0);  
 } else  
  printk(KERN\_ERR "Cannot register tty device on line %d\n",uport->line);

  //Ensure UPF\_DEAD is not set.  
 uport->flags &= ~UPF\_DEAD;

 out:  
 mutex\_unlock(&port->mutex);  
 mutex\_unlock(&port\_mutex);

 return ret;  
}

static void uart\_configure\_port(struct uart\_driver \*drv, struct uart\_state \*state,struct uart\_port \*port)  
{  
 unsigned int flags;

 if (!port->iobase && !port->mapbase && !port->membase)//未调用serial8250\_probe()之前，会从这里直接返回。  
  return;  
   
 //调用serial8250\_probe()之后，会接着往下进行  
 flags = 0;  
 if (port->flags & UPF\_AUTO\_IRQ)//未设置此标志  
  flags |= UART\_CONFIG\_IRQ;  
    
 if (port->flags & UPF\_BOOT\_AUTOCONF) {//经过probe()函数，此标志已配置  
  if (!(port->flags & UPF\_FIXED\_TYPE)) {//已设置该标志，下边的不会进入  
   port->type = PORT\_UNKNOWN;//不会进入  
   flags |= UART\_CONFIG\_TYPE;  
  }  
  port->ops->config\_port(port, flags);//调用设备的自动配置函数，即serial8250\_config\_port()  
 }

 if (port->type != PORT\_UNKNOWN) {  
  unsigned long flags;  
    
  //打印串口的信息  
  uart\_report\_port(drv, port);

  /\* Power up port for set\_mctrl() \*/  
  uart\_change\_pm(state, 0);////改变端口的电源状态，上电

  spin\_lock\_irqsave(&port->lock, flags);  
  port->ops->set\_mctrl(port, port->mctrl & TIOCM\_DTR);////设置串口modem控制，调用serial8250\_set\_mctrl()  
  spin\_unlock\_irqrestore(&port->lock, flags);

   //注册终端,配置终端的信息，若此端口有cons字段，并且console的注册不成功。注册成功CON\_ENABLED标志为1  
  if (port->cons && !(port->cons->flags & CON\_ENABLED))  
  {  
   /\*serial8250\_console\_init()函数会比serial8250\_probe()先调用,所以调用register\_console的时候,port还没有初始化,所以当  
    register\_console调用serial8250\_console\_setup()设置buad,parity bits的时候,  
    serial8250\_console\_setup()会检测port->iobase和port->membase是否是有效值,如果不是就返回,  
    放弃初始化console,所以实际上,console不是在serial8250\_console\_init()里边初始化。  
      
    当serial8250\_probe()调用uart\_add\_one\_port->uart\_configure\_port:又会调用register\_console()，  
    在这里会将真正的console注册掉。  
    该函数会检查console有没有初始化,如果没有初始化,则调用register\_console来初始化.  
    所以console放在这里初始化也是比较好一些.  
   \*/  
   register\_console(port->cons);//将该console注册到console\_drivers链表上,最后调用release\_console\_sem,将printk缓冲的数据打印到ttyS2上  
  }

  //检查此串口是否是终端，除了我们使用作为console的串口，其余的进行断电  
  if (!uart\_console(port))//#define uart\_console(port) ((port)->cons && (port)->cons->index == (port)->line)  
   uart\_change\_pm(state, 3);//除了我们使用的console，其余的进行断电  
 }  
}

//注册设备serial8250\_isa\_driver时，会调用此函数  
static int \_\_devinit serial8250\_probe(struct platform\_device \*dev)  
{  
 //传入的参数就是da8xx\_serial\_device  
 /\*struct platform\_device da8xx\_serial\_device = {//platform\_device就描述了设备对象。  
  .name = "serial8250",  
  .id = PLAT8250\_DEV\_PLATFORM,  
  .dev = {  
   .platform\_data = da8xx\_serial\_pdata,//这个platform\_device对象的私有数据指成员向一个plat\_serial8250\_port类型的数组。在这里该数组描述了三个串口接口的基本信息。  
                     //当8250驱动检测到这个platform\_device对象后，就分析该对象的私有数据成员指向的那个plat\_serial8250\_port类型的数组。  
                     //然后根据该数组的每个成员描述的信息生成一个串口对象设备。  
  },  
 };  
   
 static struct plat\_serial8250\_port da8xx\_serial\_pdata[] = {  
  {  
   .mapbase = DA8XX\_UART0\_BASE,//串口接口寄存器物理地址的基地址,#define DAVINCI\_UART0\_BASE (IO\_PHYS + 0x20000)（见/arch/arm/mach-davinci/include/mach/serial.h）  
   .irq  = IRQ\_DA8XX\_UARTINT0,//该串口接口使用的中断号, #define IRQ\_DA8XX\_UARTINT0  25,（在/arch/arm/mach-davinci/include/mach/irqs.h）  
   .flags  = UPF\_BOOT\_AUTOCONF | UPF\_SKIP\_TEST | UPF\_IOREMAP | UPF\_FIXED\_TYPE,  
   .type  = PORT\_AR7,//串口0类型  
   .iotype  = UPIO\_MEM,//成员表示该串口接口寄存器的地址类型，8位的内存地址   
   .regshift = 2,//在访问该串口接口的某个寄存器时，需把该寄存器的号左移多少位然后加基地址（不管是物理或虚拟地址）才能得能到这个寄存器的址址  
  },  
  {  
   .mapbase = DA8XX\_UART1\_BASE, //DAVINCI\_UART1\_BASE (IO\_PHYS + 0x20400)  
   .irq  = IRQ\_DA8XX\_UARTINT1, //中断号是53  
   .flags  = UPF\_BOOT\_AUTOCONF | UPF\_SKIP\_TEST |UPF\_IOREMAP | UPF\_FIXED\_TYPE,  
   .type  = PORT\_AR7,  
   .iotype  = UPIO\_MEM,  
   .regshift = 2,  
  },  
  {  
   .mapbase = DA8XX\_UART2\_BASE, //#define DAVINCI\_UART2\_BASE (IO\_PHYS + 0x20800)  
   .irq  = IRQ\_DA8XX\_UARTINT2, //中断号是61  
   .flags  = UPF\_BOOT\_AUTOCONF | UPF\_SKIP\_TEST |UPF\_IOREMAP | UPF\_FIXED\_TYPE,  
   .type  = PORT\_AR7,  
   .iotype  = UPIO\_MEM,  
   .regshift = 2,  
  },  
  {  
   .flags = 0,  
  },  
 };\*/  
 struct plat\_serial8250\_port \*p = dev->dev.platform\_data;  
 struct uart\_port port;  
 int ret, i, irqflag = 0;

 memset(&port, 0, sizeof(struct uart\_port));

 if (share\_irqs)  
  irqflag = IRQF\_SHARED;  
    
 // 会 将 dev->dev.platform\_data 所代表的 port 添加到 uart\_driver 中  
 for (i = 0; p && p->flags != 0; p++, i++) {//遍历三个串口  
  port.iobase  = p->iobase;  
  port.membase  = p->membase;  
  port.irq  = p->irq;//中断号  
  port.irqflags  = p->irqflags;  
  port.uartclk  = p->uartclk;//时钟  
  port.regshift  = p->regshift;//寄存器偏移量 2  
  port.iotype  = p->iotype;//IO类型  
  port.flags  = p->flags;//标志  
  port.mapbase  = p->mapbase;//串口接口寄存器物理地址的基地址  
  port.hub6  = p->hub6;  
  port.private\_data = p->private\_data;  
  port.type  = p->type;//PORT\_AR7,串口类型  
  port.serial\_in  = p->serial\_in;  
  port.serial\_out  = p->serial\_out;  
  port.set\_termios = p->set\_termios;  
  port.pm   = p->pm;  
  port.dev  = &dev->dev;  
  port.irqflags  |= irqflag;  
  if (p->clk)  
   serial8250\_ports[i].clk = p->clk;  
    
  //再重新注册串口  
  ret = serial8250\_register\_port(&port);  
  if (ret < 0) {  
   dev\_err(&dev->dev, "unable to register port at index %d ""(IO%lx MEM%llx IRQ%d): %d\n", i,p->iobase, (unsigned long long)p->mapbase,p->irq, ret);  
  }  
 }  
 return 0;//到这里串口的初始化就结束!!!!  
}

int serial8250\_register\_port(struct uart\_port \*port)  
{  
 struct uart\_8250\_port \*uart;  
 int ret = -ENOSPC;

 if (port->uartclk == 0)  
  return -EINVAL;

 mutex\_lock(&serial\_mutex);  
   
 //查找在serial8250\_ports[]数组中是否已有记录  
 uart = serial8250\_find\_match\_or\_unused(port);  
 if (uart) {//都会查到有记录  
  uart\_remove\_one\_port(&serial8250\_reg, &uart->port);//把原来的串口移除掉，再重新添加  
    
  //串口的uart\_port再赋值  
  uart->port.iobase       = port->iobase;  
  uart->port.membase      = port->membase;  
  uart->port.irq          = port->irq;  
  uart->port.irqflags     = port->irqflags;  
  uart->port.uartclk      = port->uartclk;  
  uart->port.fifosize     = port->fifosize;  
  uart->port.regshift     = port->regshift;  
  uart->port.iotype       = port->iotype;  
  uart->port.flags        = port->flags | UPF\_BOOT\_AUTOCONF;  
  uart->port.mapbase      = port->mapbase;  
  uart->port.private\_data = port->private\_data;  
  if (port->dev)  
   uart->port.dev = port->dev;

  if (port->flags & UPF\_FIXED\_TYPE)//probe()进行时，此标志已设置  
   serial8250\_init\_fixed\_type\_port(uart, port->type);//设置串口type=PORT\_AR7，即18

  set\_io\_from\_upio(&uart->port);//设置串口的读写函数  
    
  //如果传进来的参数此成员有值，用原来的。实际时原来此成员变量为空，da8xx\_serial\_pdata变量中未赋值  
  if (port->serial\_in)  
   uart->port.serial\_in = port->serial\_in;  
  if (port->serial\_out)  
   uart->port.serial\_out = port->serial\_out;  
  if (port->set\_termios)  
   uart->port.set\_termios = port->set\_termios;  
  if (port->pm)  
   uart->port.pm = port->pm;

  if (serial8250\_isa\_config != NULL)  
   serial8250\_isa\_config(0, &uart->port,&uart->capabilities);  
    
  //再重新添加串口，这时配置串口就能成功  
  ret = uart\_add\_one\_port(&serial8250\_reg, &uart->port);  
  if (ret == 0)  
   ret = uart->port.line;

  ret = serial8250\_cpufreq\_register(uart);  
  if (ret < 0)  
   printk(KERN\_ERR "Failed to add cpufreq notifier\n");  
 }  
 mutex\_unlock(&serial\_mutex);

 return ret;  
}

static void serial8250\_config\_port(struct uart\_port \*port, int flags)  
{  
 struct uart\_8250\_port \*up = (struct uart\_8250\_port \*)port;  
 int probeflags = PROBE\_ANY;  
 int ret;

 if (cpu\_is\_davinci\_da850())//是否为达芬奇平台  
  up->bugs |= UART\_BUG\_NOMSR;

 ret = serial8250\_request\_std\_resource(up);//分配内存资源,IO资源  
 if (ret < 0)  
  return;

 ret = serial8250\_request\_rsa\_resource(up);//会返回失败  
 if (ret < 0)  
  probeflags &= ~PROBE\_RSA;//清除PROBE\_RSA标志  
   
 //up->cur\_iotype在serial8250\_register\_ports（）设置成0xFF，  
 //而up->port.iotype在serial8250\_isa\_init\_ports()被设置为UPIO\_MEM，即2  
 if (up->port.iotype != up->cur\_iotype)  
  set\_io\_from\_upio(port);//设置uart\_port结构的serial\_in函数和serial\_out函数

 if (flags & UART\_CONFIG\_TYPE)//此标志没有被设置  
  autoconfig(up, probeflags);//不会调用

 if (up->port.type == PORT\_16550A && up->port.iotype == UPIO\_AU)  
  up->bugs |= UART\_BUG\_NOMSR;

 if (up->port.type != PORT\_UNKNOWN && flags & UART\_CONFIG\_IRQ)//此标志UART\_CONFIG\_IRQ未设置  
  autoconfig\_irq(up);//不会调用  
   
 //以下不会调用  
 if (up->port.type != PORT\_RSA && probeflags & PROBE\_RSA)  
  serial8250\_release\_rsa\_resource(up);  
 if (up->port.type == PORT\_UNKNOWN)  
  serial8250\_release\_std\_resource(up);  
}

static int serial8250\_request\_std\_resource(struct uart\_8250\_port \*up)  
{  
 unsigned int size = serial8250\_port\_size(up);  
 int ret = 0;

 switch (up->port.iotype) {//up->port.iotype == UPIO\_MEM  
 case UPIO\_AU:  
 case UPIO\_TSI:  
 case UPIO\_MEM32:  
 case UPIO\_MEM:  
 case UPIO\_DWAPB:  
  if (!up->port.mapbase)  
   break;

  if (!request\_mem\_region(up->port.mapbase, size, "serial")) {  
   ret = -EBUSY;  
   break;  
  }

  if (up->port.flags & UPF\_IOREMAP) {//如果前边映射过了，UPF\_IOREMAP被清零了，这里就不需再映射了  
   up->port.membase = ioremap\_nocache(up->port.mapbase,size);  
   if (!up->port.membase) {  
    release\_mem\_region(up->port.mapbase, size);  
    ret = -ENOMEM;  
   }  
  }  
  break;

 case UPIO\_HUB6:  
 case UPIO\_PORT:  
  if (!request\_region(up->port.iobase, size, "serial"))  
   ret = -EBUSY;  
  break;  
 }  
 return ret;  
}

static void set\_io\_from\_upio(struct uart\_port \*p)  
{  
 struct uart\_8250\_port \*up = (struct uart\_8250\_port \*)p;  
 switch (p->iotype) {  
 case UPIO\_HUB6:  
  p->serial\_in = hub6\_serial\_in;  
  p->serial\_out = hub6\_serial\_out;  
  break;

 case UPIO\_MEM://p->iotype=2，采用这里的读写函数  
  p->serial\_in = mem\_serial\_in;  
  p->serial\_out = mem\_serial\_out;  
  break;

 case UPIO\_RM9000:  
 case UPIO\_MEM32:  
  p->serial\_in = mem32\_serial\_in;  
  p->serial\_out = mem32\_serial\_out;  
  break;

 case UPIO\_AU:  
  p->serial\_in = au\_serial\_in;  
  p->serial\_out = au\_serial\_out;  
  break;

 case UPIO\_TSI:  
  p->serial\_in = tsi\_serial\_in;  
  p->serial\_out = tsi\_serial\_out;  
  break;

 case UPIO\_DWAPB:  
  p->serial\_in = mem\_serial\_in;  
  p->serial\_out = dwapb\_serial\_out;  
  break;

 default:  
  p->serial\_in = io\_serial\_in;  
  p->serial\_out = io\_serial\_out;  
  break;  
 }  
 /\* Remember loaded iotype \*/  
 up->cur\_iotype = p->iotype;//注意要赋值  
}

static inline void uart\_report\_port(struct uart\_driver \*drv, struct uart\_port \*port)  
{  
 char address[64];

 switch (port->iotype) {//port->iotype == UPIO\_MEM  
 case UPIO\_PORT:  
  snprintf(address, sizeof(address), "I/O 0x%lx", port->iobase);  
  break;  
 case UPIO\_HUB6:  
  snprintf(address, sizeof(address),"I/O 0x%lx offset 0x%x", port->iobase, port->hub6);  
  break;  
 case UPIO\_MEM:  
 case UPIO\_MEM32:  
 case UPIO\_AU:  
 case UPIO\_TSI:  
 case UPIO\_DWAPB:  
  snprintf(address, sizeof(address), "MMIO 0x%llx", (unsigned long long)port->mapbase);  
  break;  
 default:  
  strlcpy(address, "\*unknown\*", sizeof(address));  
  break;  
 }  
 /\*打印出的信息如下：  
 serial8250.0: ttyS0 at MMIO 0x1c42000 (irq = 25) is a AR7  
 serial8250.0: ttyS1 at MMIO 0x1d0c000 (irq = 53) is a AR7  
 serial8250.0: ttyS2 at MMIO 0x1d0d000 (irq = 61) is a AR7  
 \*/  
 printk(KERN\_INFO "%s%s%s%d at %s (irq = %d) is a %s\n",  
        port->dev ? dev\_name(port->dev) : "",  
        port->dev ? ": " : "",  
        drv->dev\_name,  
        drv->tty\_driver->name\_base + port->line,  
        address, port->irq, uart\_type(port));  
}

//下边再次调用register\_console()注册serial8250\_console真正的console终端  
void register\_console(struct console \*newcon)  
{  
 int i;  
 unsigned long flags;  
 struct console \*bcon = NULL;  
 /\*  
 现在是注册一个serial8250\_console，即  
 static struct console serial8250\_console = {  
  .name  = "ttyS",  
  .write  = serial8250\_console\_write,//写方法  
  .device  = uart\_console\_device,//tty驱动  
  .setup  = serial8250\_console\_setup,//设置串口波特率，也就是设置串口。很重要，里面涉及到平台特性，波特率相关。  
  .early\_setup = serial8250\_console\_early\_setup,  
  .flags  = CON\_PRINTBUFFER | CON\_ANYTIME,  
  .index  = -1,  
  .data  = &serial8250\_reg,  
 };  
 \*/  
 if (console\_drivers && newcon->flags & CON\_BOOT) {//注册的是serial8250\_console，CON\_BOOT没有置位，不是引导控制台。下边不会进去遍历  
  for\_each\_console(bcon) {////遍历全局console\_drivers数组     
   if (!(bcon->flags & CON\_BOOT)) {//判断是否已经有引导控制台了,有了的话就直接退出  
    printk(KERN\_INFO "Too late to register bootconsole %s%d\n",newcon->name, newcon->index);  
    return;  
   }  
  }  
 }  
   
 if (console\_drivers && console\_drivers->flags & CON\_BOOT)//如果注册的是引导控制台，serial8250\_console不是引导控制台  
  bcon = console\_drivers;//这里不执行

 if (preferred\_console < 0 || bcon || !console\_drivers)  
  preferred\_console = selected\_console;//设置preferred\_console为uboot命令选择的selected\_console(即在Uboot传入的参数“console=ttyS2,115200n8”在console\_cmdline[]数组中的索引)     
                     //这里preferred\_console =0  
 if (newcon->early\_setup)//serial8250\_console初始化early\_setup字段  
  newcon->early\_setup();//调用serial8250\_console\_early\_setup()

 if (preferred\_console < 0) {//由于preferred\_console =0，不会进入下边  
  if (newcon->index < 0)  
   newcon->index = 0;  
  if (newcon->setup == NULL ||newcon->setup(newcon, NULL) == 0) {  
   newcon->flags |= CON\_ENABLED;  
   if (newcon->device) {  
    newcon->flags |= CON\_CONSDEV;  
    preferred\_console = 0;  
   }  
  }  
 }

  //传给内核参数:  
  //Kernel command line: console=ttyS2,115200n8 rw root=/dev/ram0 initrd=0xc2000000,20M mem=128M ip=192.168.1.220::192.168.1.1:255.255.255.0::eth0:off  
  //所以这里将根据传参console=ttyS2,115200来配置作为console的ttyS2串口  
 for (i = 0; i < MAX\_CMDLINECONSOLES && console\_cmdline[i].name[0];i++) {//遍历全局console\_cmdline找到匹配的，i=0就是匹配的“ttyS2”  
  if (strcmp(console\_cmdline[i].name, newcon->name) != 0)//比较终端名称“ttyS”  
   continue;  
  if (newcon->index >= 0 &&newcon->index != console\_cmdline[i].index)//console\_cmdline[i].index=2。//比较次设备号    
   continue;  
  if (newcon->index < 0)  
   newcon->index = console\_cmdline[i].index;//将终端号赋值给serial8250\_console->index，这里是2  
     
  //console\_cmdline[i].options = "115200n8"，对于serial8250\_console而言setup字段已初始化  
  if (newcon->setup && newcon->setup(newcon, console\_cmdline[i].options) != 0)//调用serial8250\_console\_setup()对终端进行配置，未probe()前调用不成功，probe()后调用成功。  
   break;  
  //在这里注册serial8250\_console时，调用serial8250\_console\_setup()由于port->iobase和port->membase不是有效值，  
  //故返回错误，这样下边的操作不会执行，直接break跳出，从flag1出跳出函数。即在这里serial8250\_console没有注册成功  
  //由于内核在下边的操作队串口进行初始化时，还会调用register\_console（）来注册serial8250\_console，在那时注册就会成功  
    
  newcon->flags |= CON\_ENABLED; //设置标志为CON\_ENABLE,表示console使能(这个在printk调用中使用到)   
  newcon->index = console\_cmdline[i].index;//设置索引号     
  if (i == selected\_console) { //索引号和uboot指定的console的一样   
   newcon->flags |= CON\_CONSDEV;//设置标志CON\_CONSDEV(全局console\_drivers链表中靠前)   
   preferred\_console = selected\_console;  
  }  
  break;  
 }//for循环作用大致是查看注册的console是否是uboot知道的引导console，是则设置相关标志和preferred\_console

  //flag1：  
 if (!(newcon->flags & CON\_ENABLED))//若前边没有设置CON\_ENABLED标志，就退出。若进行过probe（）,CON\_ENABLED置位，这里就往下接着注册console  
  return;

 if (bcon && ((newcon->flags & (CON\_CONSDEV | CON\_BOOT)) == CON\_CONSDEV))//防止重复打印     
  newcon->flags &= ~CON\_PRINTBUFFER;

 acquire\_console\_sem();  
 if ((newcon->flags & CON\_CONSDEV) || console\_drivers == NULL) {//如果是preferred控制台，此标志CON\_CONSDEV在前边已设置过  
  newcon->next = console\_drivers;  
  console\_drivers = newcon;//添加进全局console\_drivers链表前面位置（printk中会遍历该表调用合适的console的write方法打印信息）  
  if (newcon->next)  
   newcon->next->flags &= ~CON\_CONSDEV;  
 } else {//如果不是preferred控制台   
  newcon->next = console\_drivers->next;  
  console\_drivers->next = newcon; //添加进全局console\_drivers链表后面位置  
 }  
   
 //主册console主要是刷选preferred\_console放置在全局console\_drivers链表前面,剩下的console放置链表靠后的位置,并设置相应的flags,  
 //console\_drivers最终会在printk函数的层层调用中遍历到,并调用console的write方法将信息打印出来

 if (newcon->flags & CON\_PRINTBUFFER) {  
  spin\_lock\_irqsave(&logbuf\_lock, flags);  
  con\_start = log\_start;  
  spin\_unlock\_irqrestore(&logbuf\_lock, flags);  
 }  
 release\_console\_sem();

 if (bcon && ((newcon->flags & (CON\_CONSDEV | CON\_BOOT)) == CON\_CONSDEV)) {  
  printk(KERN\_INFO "console [%s%d] enabled, bootconsole disabled\n",newcon->name, newcon->index);  
  for\_each\_console(bcon)  
   if (bcon->flags & CON\_BOOT)  
    unregister\_console(bcon);  
 } else {//调用这里  
  printk(KERN\_INFO "%sconsole [%s%d] enabled\n",(newcon->flags & CON\_BOOT) ? "boot" : "" ,newcon->name, newcon->index);  
 }  
}

//serial8250\_console\_early\_setup()-->serial8250\_find\_port\_for\_earlycon()  
int serial8250\_find\_port\_for\_earlycon(void)  
{  
 struct early\_serial8250\_device \*device = &early\_device;//early console初始化时对early\_device结构的初始化  
 struct uart\_port \*port = &device->port;  
 int line;  
 int ret;

 if (!device->port.membase && !device->port.iobase)//early\_device结构初始化时已经配置好  
  return -ENODEV;  
 //early console注册时不会调用此函数。  
 //当真正的console初始化时，会调用此函数。  
 //真正的console初始化时，会查找early console注册时用的是哪一个串口号，从serial8250\_ports[]中根据uart\_port->mapbase地址来比对  
 line = serial8250\_find\_port(port);//根据uart\_port结构找到串口号，比对没有找到串口号，line返回负值  
 if (line < 0)  
  return -ENODEV;//从这里返回，下边的不再执行  
   
 //若找到early console用的串口号，更新当初传入内核参数使用的console\_cmdline[i]，名称改成ttyS。。。。  
 ret = update\_console\_cmdline("uart", 8250, "ttyS", line, device->options);  
 if (ret < 0)  
  ret = update\_console\_cmdline("uart", 0,"ttyS", line, device->options);

 return ret;  
}

static int \_\_init serial8250\_console\_setup(struct console \*co, char \*options)  
{  
 struct uart\_port \*port;  
 int baud = 9600;  
 int bits = 8;  
 int parity = 'n';  
 int flow = 'n';

 if (co->index >= nr\_uarts)//console的索引，这里是2，即ttyS2  
  co->index = 0;  
 port = &serial8250\_ports[co->index].port;//找到对应的ttyS2的uart\_port结构  
   
 //由于console\_init在注册serial8250\_console时调用的register\_console()函数调用serial8250\_console\_setup()  
 //进入这个函数时，由于ttyS2的uart\_port结构没有初始化，port->iobase 和port->membase值都未设置，所以直接从下边返回  
 //当进行串口初始化时，还会回来注册serial8250\_console，再调用到这里，由于设置了ttyS2的uart\_port结构，所以下边的配置就会成功  
 if (!port->iobase && !port->membase)//第一次注册时，由于未设置，从这里直接返回  
  return -ENODEV;

 if (options)//如果options不为空，就将options里的数值写给baud, &parity, &bits, &flow  
  uart\_parse\_options(options, &baud, &parity, &bits, &flow);  
 //没有配置options，则使用缺省值，否则使用传下来的的参数options里的串口配置  
 return uart\_set\_options(port, co, baud, parity, bits, flow);  
}

void uart\_parse\_options(char \*options, int \*baud, int \*parity, int \*bits, int \*flow)  
{  
 char \*s = options;

 \*baud = simple\_strtoul(s, NULL, 10);  
 while (\*s >= '0' && \*s <= '9')  
  s++;  
 if (\*s)  
  \*parity = \*s++;  
 if (\*s)  
  \*bits = \*s++ - '0';  
 if (\*s)  
  \*flow = \*s;  
}

int uart\_set\_options(struct uart\_port \*port, struct console \*co,int baud, int parity, int bits, int flow)  
{  
 struct ktermios termios;  
 static struct ktermios dummy;  
 int i;

 //Ensure that the serial console lock is initialised early.  
 spin\_lock\_init(&port->lock);  
 lockdep\_set\_class(&port->lock, &port\_lock\_key);

 memset(&termios, 0, sizeof(struct ktermios));  
 termios.c\_cflag = CREAD | HUPCL | CLOCAL;

 //Construct a cflag setting.  
 for (i = 0; baud\_rates[i].rate; i++)  
  if (baud\_rates[i].rate <= baud)  
   break;

 termios.c\_cflag |= baud\_rates[i].cflag;

 if (bits == 7)  
  termios.c\_cflag |= CS7;  
 else  
  termios.c\_cflag |= CS8;

 switch (parity) {  
 case 'o': case 'O':  
  termios.c\_cflag |= PARODD;  
  /\*fall through\*/  
 case 'e': case 'E':  
  termios.c\_cflag |= PARENB;  
  break;  
 }

 if (flow == 'r')  
  termios.c\_cflag |= CRTSCTS;

 /\*  
  \* some uarts on other side don't support no flow control.  
  \* So we set \* DTR in host uart to make them happy  
  \*/  
 port->mctrl |= TIOCM\_DTR;

 port->ops->set\_termios(port, &termios, &dummy);//调用serial8250\_set\_termios()对串口进行配置

 //Allow the setting of the UART parameters with a NULL console too:  
 if (co)  
  co->cflag = termios.c\_cflag;

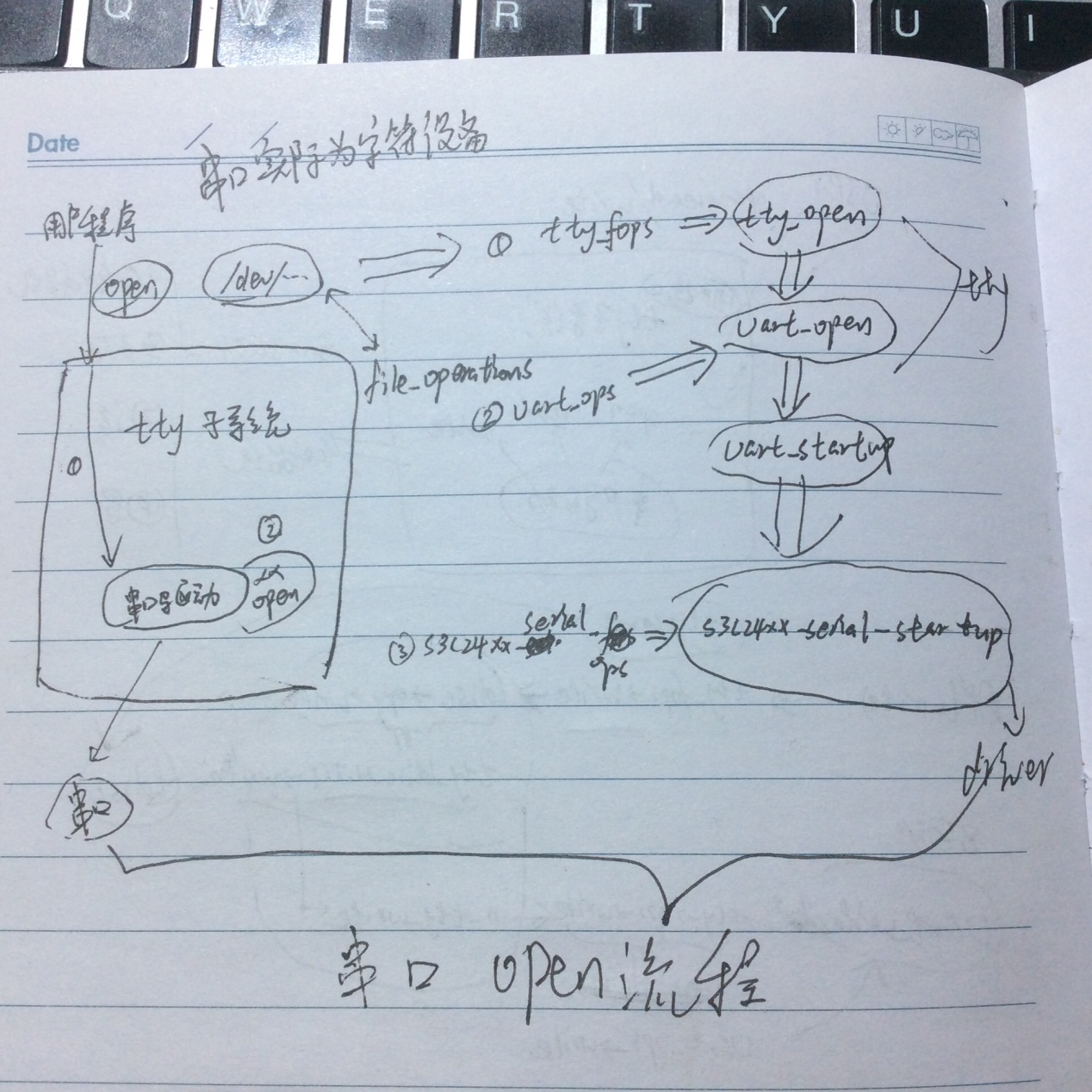
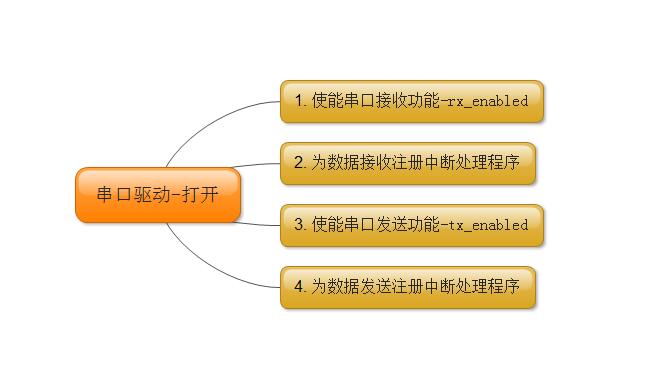
 return 0;  
}

struct device \*tty\_register\_device(struct tty\_driver \*driver, unsigned index,struct device \*device)  
{  
 char name[64];  
 dev\_t dev = MKDEV(driver->major, driver->minor\_start) + index;//得到设备号

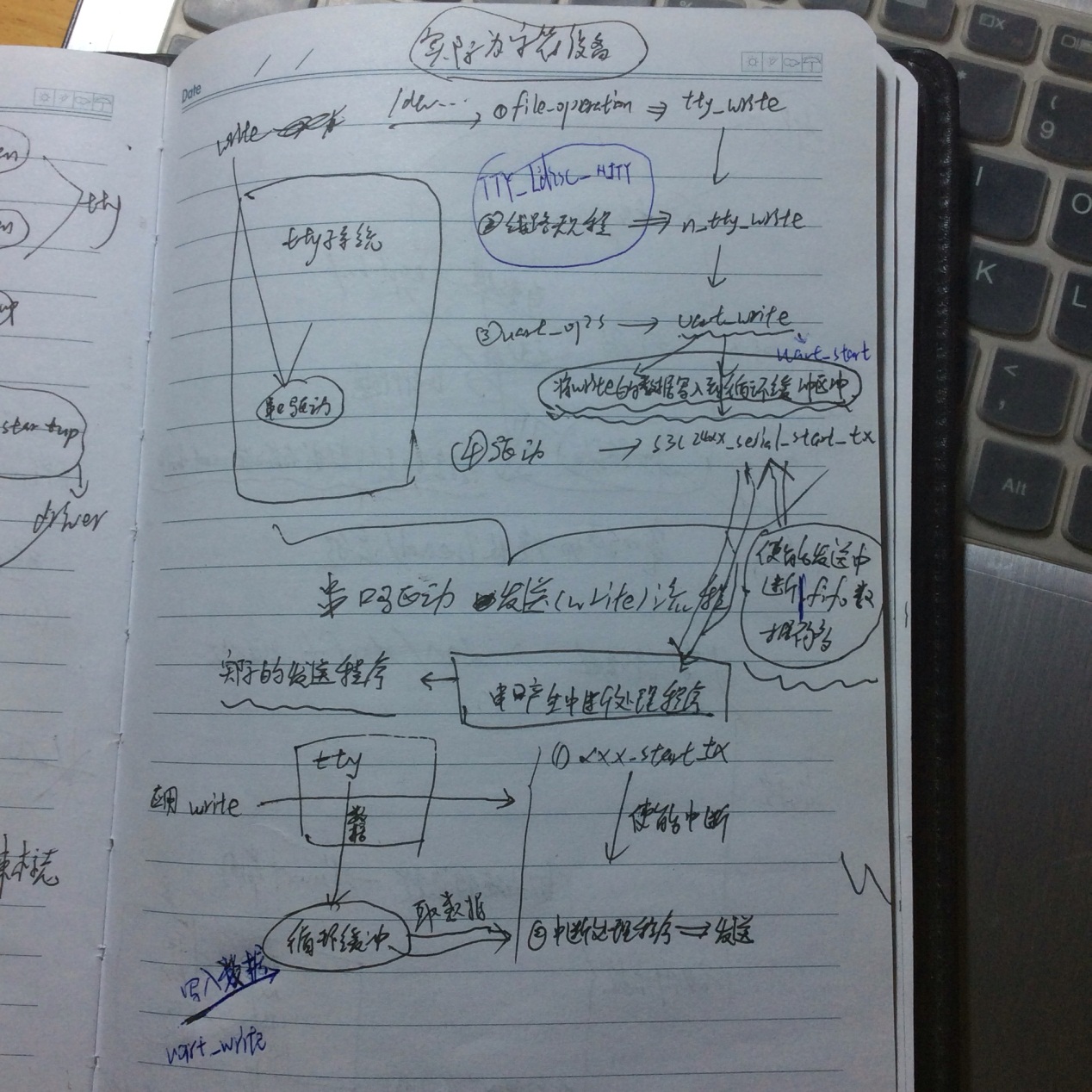
 if (index >= driver->num) {  
  printk(KERN\_ERR "Attempt to register invalid tty line number "" (%d).\n", index);  
  return ERR\_PTR(-EINVAL);  
 }

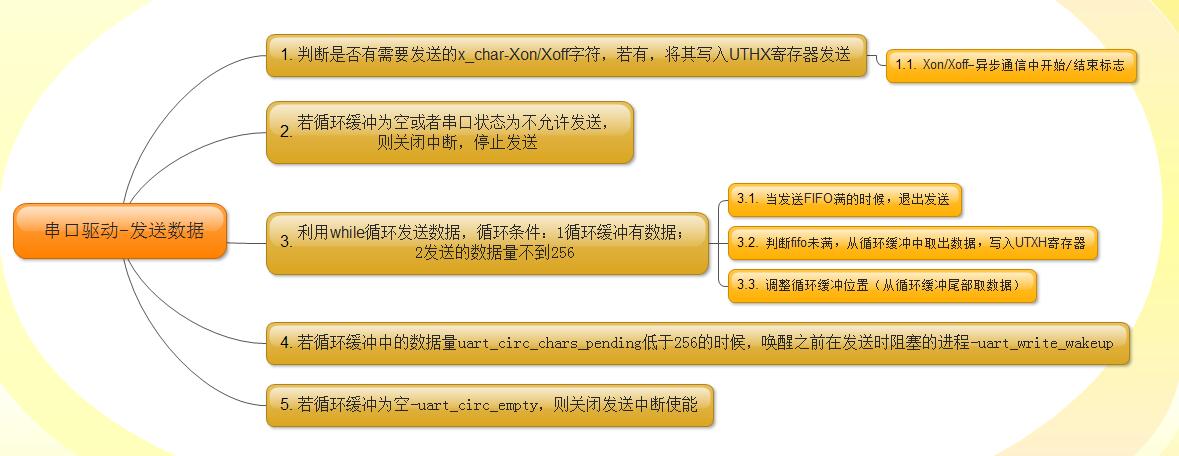
 if (driver->type == TTY\_DRIVER\_TYPE\_PTY)  
  pty\_line\_name(driver, index, name);  
 else  
  tty\_line\_name(driver, index, name);//得到串口设备名称ttyS0，ttyS1，ttyS2  
   
 return device\_create(tty\_class, device, dev, NULL, name);//在/dev下创建设备文件  
}

## 串口打开流程

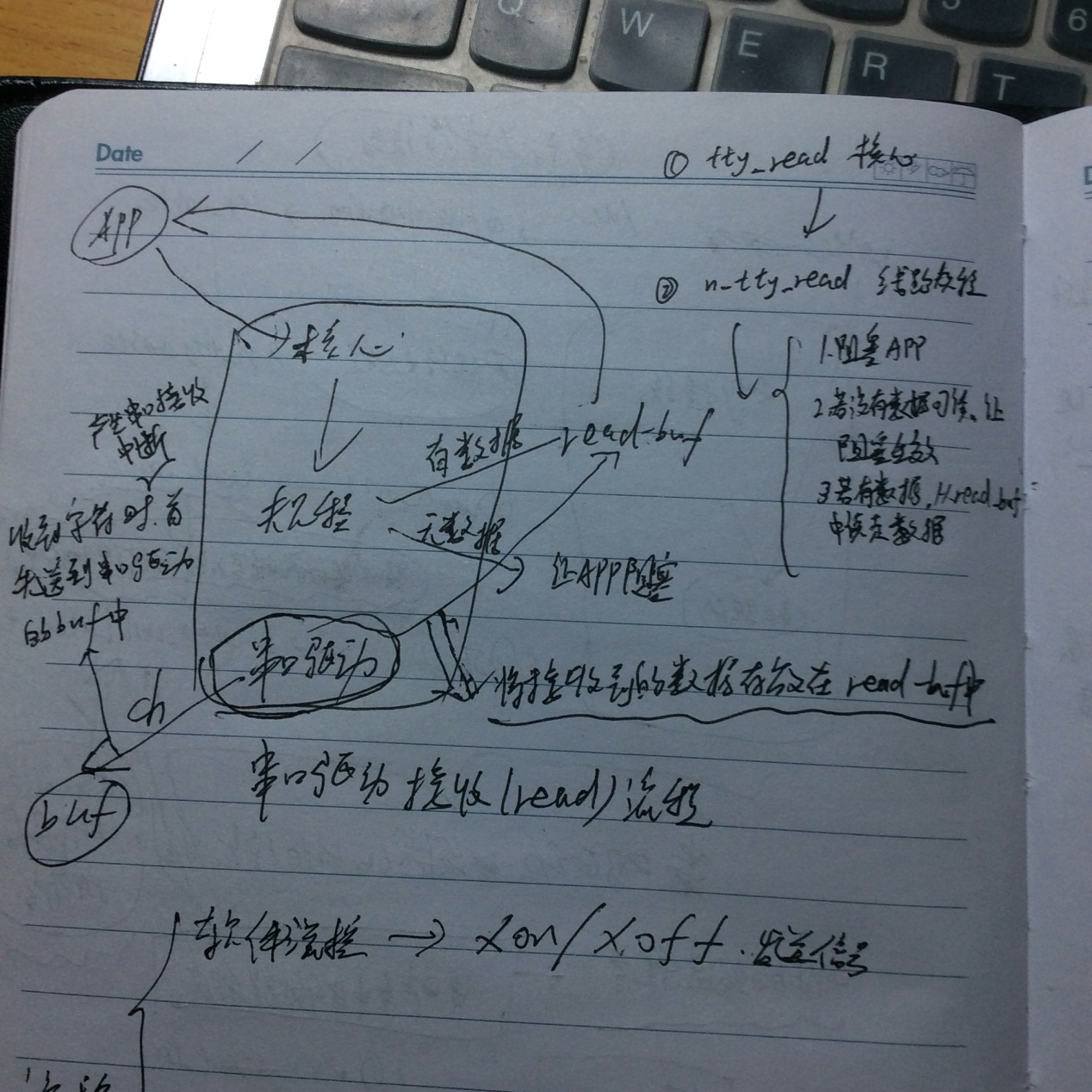
****

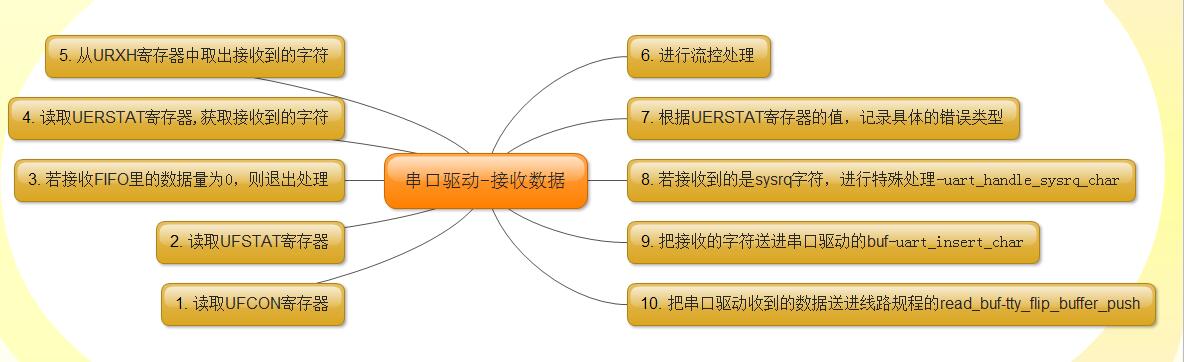
## 串口发送流程





## 串口接收流程





## console和串口的关系

Linux内核中的console是虚拟的，它依赖于串口设备；在系统启动的时候，首先会有earlyconsole用来打印log，这个时候内核中的串口设备还没有probe，通过printascii工作；等到串口初始化结束之后就通过串口设备打印，这个时候prink这类接口也可以用了。

在Linux系统中，终端是一类字符型设备，它包括多种类型，通常使用tty来简称各种类型的终端设备。

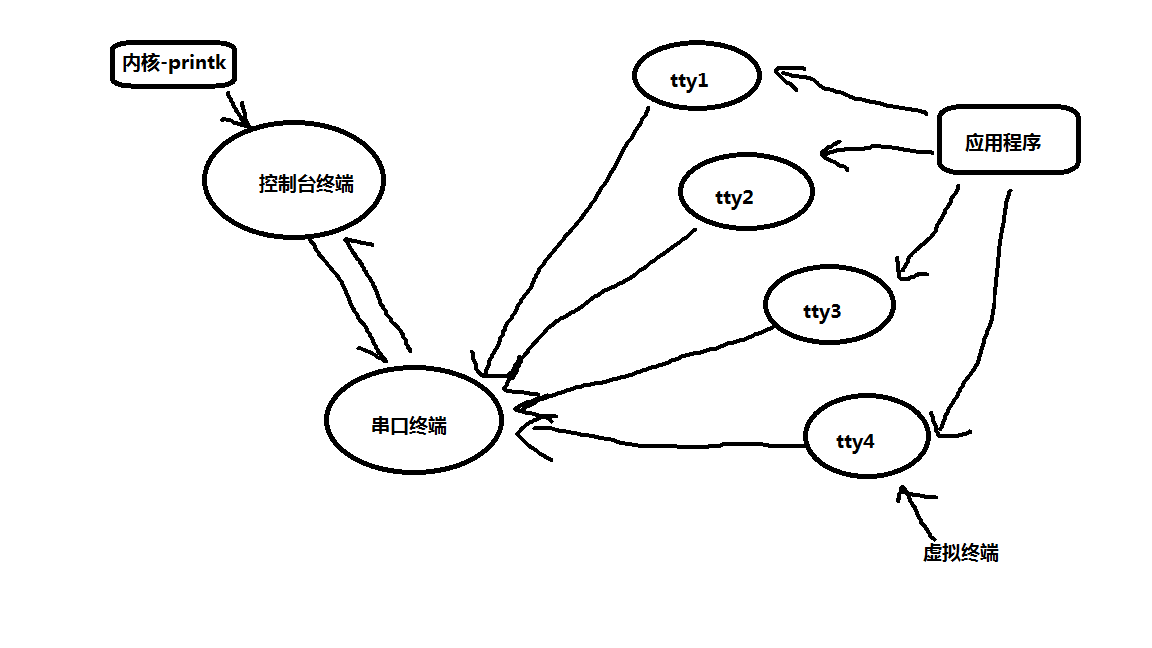
• 串口终端（/dev/ttyS\*）

　　串口终端是使用计算机串口连接的终端设备。Linux把每个串行端口都看作是一个字符设备。这些串行端口所对应的设备名称是/dev/ttySAC0;/dev/ttySAC1……

• 控制台终端（/dev/console）

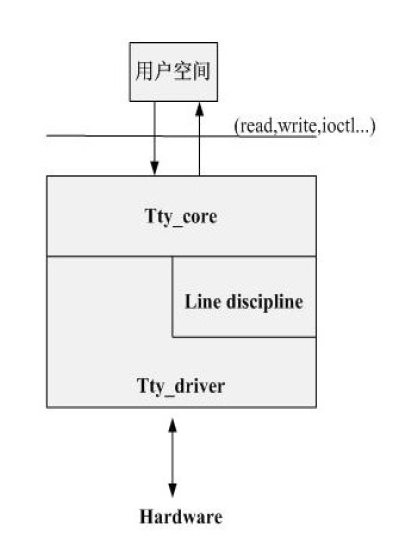
　　在Linux系统中，计算机的输出设备通常被称为控制台终端(Console),这里特指printk信息输出到的设备。/dev/console是一个虚拟的设备，它需要映射到真正的tty上，比如通过内核启动参数” console=ttySAC0”就把console映射到了串口0

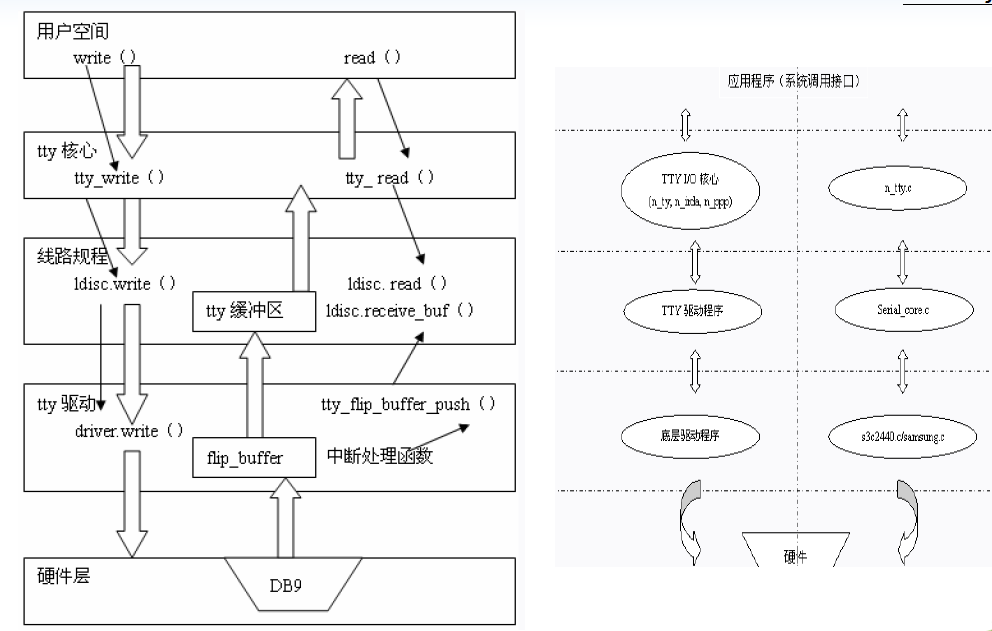
• 虚拟终端（/dev/tty\*）

　　当用户登录时，使用的是虚拟终端。使用Ctcl+Alt+[F1—F6]组合键时，我们就可以切换到tty1、tty2、tty3等上面去。tty1–tty6等称为虚拟终端，而tty0则是当前所使用虚拟终端的一个别名. 

**Ps:tty子系统**

**TTY架构分析：Linux tty子系统包含：tty核心，tty线路规程和tty驱动。tty核心是对整个tty设备的抽象，对用户提供统一的接口，tty线路规程是对传输数据的格式化，tty驱动则是面向tty设备的硬件驱动。**

****

****

**在linux内核中，一般可使用dump\_stack()函数回溯，重新编译内核，启动之后可看到回溯信息。**

待完成

## 参考资料

<http://blog.csdn.net/yyyks/article/details/7242267#comments>

<http://blog.csdn.net/tianxiawuzhei/article/details/7264873>

<http://dl.linux-sunxi.org/A20/A20%20User%20Manual%202013-03-22.pdf>

<http://www.wowotech.net/basic_tech/serial_intro.html>

<http://www.wowotech.net/comm/serial_overview.html>

http://cocboso.com/subject/about/28692.html

<http://www.cnblogs.com/chd-zhangbo/p/5409959.html>

http://www.cnblogs.com/chd-zhangbo/p/5410336.html