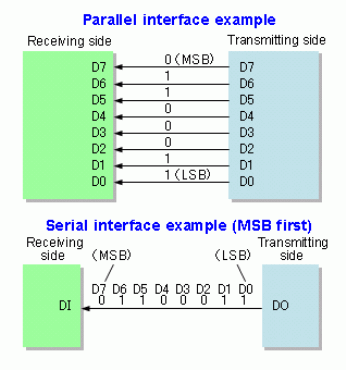
# 串口子系统

## 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。

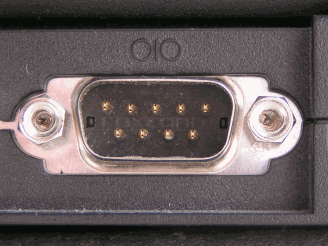
2）发送端在数据发送之前和之后，通过特定形式的信号（例如START信号和STOP信号），告诉接收端，可以开始（或者停止）接收数据了。与此同时，收发两方会约定一个数据发送的速度（就是大名鼎鼎的波特率），发送端在发送START信号之后，就按照固定的节奏发送串行数据，与此同时，接收端在收到START信号之后，也按照固定的节奏接收串行数据。这就是常说的异步串行通信（Asynchronous serial communication），我们本节的主角----串口通信，就是这种通信方式。

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。

## 1.4UART(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：

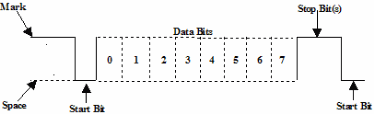
start bit固定为1bit；

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

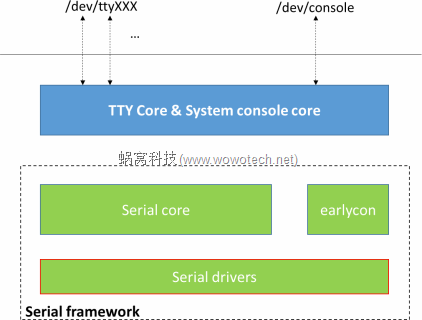
因此，115200波特率所容许的误差范围是20 / 21 ~ 20 / 19，即+/- 5%。其它波特率也可以带入公式去计算。另外，如果双方都有误差的话怎么办？嘿嘿，你懂的，除2吧。

## 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

#### 3.1 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

#### 3.2 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（具体可参考“include/linux/serial\_core.h”）。

**3.2.1 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

lock，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。

**3.2.2 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。

**3.2.3 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。

因为在历史上，串口设备是一种非常复杂的设备，因此，和struct uart\_port类似，struct uart\_ops结构也非常庞大，包罗万象，这里简单介绍一些常用的（其它在用到的时候再关注，或者可参考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）。

**3.2.4 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。

#### 3.3 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。

## 4. 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。

#### 4.1 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。

#### 4.2 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）。

这种方法稍显麻烦，有违earlycon的初衷，暂不介绍了，感兴趣的同学可以自行研读代码。

## 5. 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)

### 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，首先会有个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。

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