# 驱动注册

串口驱动使用如下结构描述：

struct uart\_driver {

struct module \*owner;//一般为THIS\_MODULE

const char \*driver\_name;//驱动名称

const char \*dev\_name;//设备名称

int major;//主设备号

int minor;//此设备号

int nr;//支持的端口数目

struct console \*cons;//控制台

struct uart\_state \*state;

struct tty\_driver \*tty\_driver;//串口驱动是一种tty驱动

};

上述结构赋值完毕后就可向内核进行注册：

int uart\_register\_driver(struct uart\_driver \*drv);

相应的注销函数为：

void uart\_unregister\_driver(struct uart\_driver \*drv);

# 端口添加

一个串口控制器可以包含多个端口，uart\_port结构就描述了端口的中断、I/O内存地址、FIFO大小、端口类型等信息：

struct uart\_port {

spinlock\_t lock; /\* port lock \*/

unsigned long iobase; /\* in/out[bwl] \*/

unsigned char \_\_iomem \*membase; /\* read/write[bwl] \*/

unsigned int (\*serial\_in)(struct uart\_port \*, int);

void (\*serial\_out)(struct uart\_port \*, int, int);

void (\*set\_termios)(struct uart\_port \*, struct ktermios \*new, struct ktermios \*old);

unsigned int (\*get\_mctrl)(struct uart\_port \*);

void (\*set\_mctrl)(struct uart\_port \*, unsigned int);

int (\*startup)(struct uart\_port \*port);

void (\*shutdown)(struct uart\_port \*port);

void (\*throttle)(struct uart\_port \*port);

void (\*unthrottle)(struct uart\_port \*port);

int (\*handle\_irq)(struct uart\_port \*);

void (\*pm)(struct uart\_port \*, unsigned int state,

unsigned int old);

void (\*handle\_break)(struct uart\_port \*);

int (\*rs485\_config)(struct uart\_port \*, struct serial\_rs485 \*rs485);

unsigned int irq; /\* irq number \*/

unsigned long irqflags; /\* irq flags \*/

unsigned int uartclk; /\* base uart clock \*/

unsigned int fifosize; /\* tx fifo size \*/

unsigned char x\_char; /\* xon/xoff char \*/

unsigned char regshift; /\* reg offset shift \*/

unsigned char iotype; /\* io access style \*/

unsigned char unused1;

unsigned int read\_status\_mask; /\* driver specific \*/

unsigned int ignore\_status\_mask; /\* driver specific \*/

struct uart\_state \*state; /\* pointer to parent state \*/

struct uart\_icount icount; /\* statistics \*/

struct console \*cons; /\* struct console, if any \*/

upf\_t flags;

upstat\_t status;

int hw\_stopped; /\* sw-assisted CTS flow state \*/

unsigned int mctrl; /\* current modem ctrl settings \*/

unsigned int timeout; /\* character-based timeout \*/

unsigned int type; /\* port type \*/

const struct uart\_ops \*ops;

unsigned int custom\_divisor;

unsigned int line; /\* port index \*/

unsigned int minor;

resource\_size\_t mapbase; /\* for ioremap \*/

resource\_size\_t mapsize;

struct device \*dev; /\* parent device \*/

unsigned char hub6; /\* this should be in the 8250 driver \*/

unsigned char suspended;

unsigned char irq\_wake;

unsigned char unused[2];

struct attribute\_group \*attr\_group; /\* port specific attributes \*/

const struct attribute\_group \*\*tty\_groups; /\* all attributes (serial core use only) \*/

struct serial\_rs485 rs485;

void \*private\_data; /\* generic platform data pointer \*/

};

其中uart\_ops是操作端口的接口函数集，串口设备的文件操作最后都会转化为对端口的接口函数的调用，对此结构的详细描述可参考Documentation\driver-api\serial\driver.rst:

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 \*);

void (\*start\_tx)(struct uart\_port \*);

void (\*throttle)(struct uart\_port \*);

void (\*unthrottle)(struct uart\_port \*);

void (\*send\_xchar)(struct uart\_port \*, char ch);

void (\*stop\_rx)(struct uart\_port \*);

void (\*enable\_ms)(struct uart\_port \*);

void (\*break\_ctl)(struct uart\_port \*, int ctl);

int (\*startup)(struct uart\_port \*);

void (\*shutdown)(struct uart\_port \*);

void (\*flush\_buffer)(struct uart\_port \*);

void (\*set\_termios)(struct uart\_port \*, struct ktermios \*new,

struct ktermios \*old);

void (\*set\_ldisc)(struct uart\_port \*, struct ktermios \*);

void (\*pm)(struct uart\_port \*, unsigned int state,

unsigned int oldstate);

const char \*(\*type)(struct uart\_port \*);

void (\*release\_port)(struct uart\_port \*);

int (\*request\_port)(struct uart\_port \*);

void (\*config\_port)(struct uart\_port \*, int);

int (\*verify\_port)(struct uart\_port \*, struct serial\_struct \*);

int (\*ioctl)(struct uart\_port \*, unsigned int, unsigned long);

};

.tx\_empty：测试端口的发送缓冲区是否为空，为空时则返回TIOCSER\_TEMT，否则返回0，如果端口不支持此操作则应返回TIOCSER\_TEMT，此调用不能睡眠。

.stop\_tx：停止发送操作，一般为关闭发送中断，此调用不能睡眠。

.start\_tx：开始发送操作，一般为使能发送中断，具体的发送过程在中断处理中完成，此调用不能睡眠。

.stop\_rx：停止接收操作，此调用不能睡眠。

.startup：启用端口，可在此函数中完成中断注册、初始化状态、启动接收等。

.set\_termios：用于设置端口参数，包括字长、波特率、奇偶校验位、停止位等。

uart\_port定义完成后需要与uart\_driver关联，使用如下函数可将uart\_port添加到uart\_driver中：

int uart\_add\_one\_port(struct uart\_driver \*drv, struct uart\_port \*uport);

相应的反函数为：

int uart\_remove\_one\_port(struct uart\_driver \*drv, struct uart\_port \*uport)

# API说明

向uart层插入字符：

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

.port：端口

.status：RX buffer的状态

.overrun:status中overrun未的掩码

.flag：插入字符的标志：TTY\_NORMAL、TTY\_BREAK、TTY\_FRAME、TTY\_PARITY、TTY\_OVERRUN

唤醒上层因向端口写数据而阻塞的进程，通常在串口发送中断处理函数中调用此函数：

void uart\_write\_wakeup(struct uart\_port \*port);

circ相关宏：

/\* 缓冲区是否为空 \*/

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

/\* 清除缓冲区 \*/

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

/\* 缓冲区中有效数据的个数 \*/

#define uart\_circ\_chars\_pending(circ) \

(CIRC\_CNT((circ)->head, (circ)->tail, UART\_XMIT\_SIZE))

/\* 缓冲区中空闲空间的个数 \*/

#define uart\_circ\_chars\_free(circ) \

(CIRC\_SPACE((circ)->head, (circ)->tail, UART\_XMIT\_SIZE))

# 数据收发过程

在应用层操作tty设备时，需经过tty驱动层才能来到串口驱动层，串口驱动的核心serial\_core.c在我们调用uart\_register\_driver()注册串口驱动时已经自动填充了tty的操作函数集tty\_operations和tty\_port\_operations并完成了tty驱动的注册，并且在这些操作函数中又会执行我们在端口添加时提供的uart\_ops中的各个回调。

## 设备打开过程



## 数据发送过程

数据发送时一般都为在start\_tx中开启发送中断，在中断回调中则完成数据的发送并完成唤醒。下图中的数据发送过程并没有画出使用dma的情况，如果使用dma，则是在start\_tx中开启发送dma传输，在发送dma的完成回调中完成唤醒。



## 数据接收过程

同样下图中并没有画出使用dma的情况，如果使用dma，则会在接收dma的回调函数中完成uart\_insert\_char()以及tty\_flip\_buffer\_push()。



# 示例说明

设备树节点：

virt\_uart0: serial@0 {

compatible = "xm,virt-uart";

port-index = <0>;

status = "okay";

};

virt\_uart1: serial@1 {

compatible = "xm,virt-uart";

port-index = <1>;

status = "okay";

};

serial\_uart.c完成了一个虚拟的串口驱动，可以完成回环功能，即接收到发送的数据，此虚拟串口包含两个端口，insmod serial-virt.ko后就会出现/dev/vttyS0和/dev/vttyS1设备，两个设备的功能完全一致。

测试程序test\_virt\_uart.c则用于测试，向设备写入自加数然后完成数据接收。

当然也可以通过如下方式测试，但是注意要先运行一下上面的测试程序，因为串口默认打开回显功能，会将收到的数据再发出去，而虚拟串口驱动又会接收到发送的数据，就会造成开始发送后就停不下来了，测试程序则会关闭回显。

