UART 串口驱动开发文档

-------w83697/w83977 super I/O 串口驱动开发

文档作者: 侯辉华
部　　门:
时　　间: 2007/08/12

内容简介: 介绍了 Linux 下的串口驱动的设计层次及接口, 并指出串口与 TTY 终端之间的关联层次(串口可作 TTY 终端使用), 以及 Linux 下的中断处理机制/中断共享机制, 还有串口缓冲机制当中涉及的软中断机制; 其中有关 w83697/w83977 IC 方面的知识, 具体参考相关手册, 对串口的配置寄存器有详细介绍, 本文不再进行说明.

目录索引:

一. Linux 的串口接口及层次.

串口是使用已经非常广的设备了，因此在 linux 下面的支持已经很完善了，具有统一的编程接口，驱动开发者所要完整的工作就是针对不同的串口 IC 来做完成相应的配置宏，这此配置宏包括读与写，中断打开与关闭(如传送与接收中断)，接收状态处理，有 FIFO 时还要处理 FIFO 的状态. 如下我们就首先切入这一部分，具体了解一下与硬件串口 IC 相关的部分在驱动中的处理，这一部分可以说是串口驱动中的最基础部分，直接与硬件打交道，完成最底层具体的串口数据传输.

1. 串口硬件资源的处理.

W83697 及 W83977 在 ep93xx 板子上的映射的硬件物理空间如下:

W83697: 0x20000000 起 1K 空间.

W83977: 0x30000000 起 1K 空间.

因为串口设备的特殊性，可以当作终端使用，但是终端的使用在内核还未完全初始化之前(关于串口与终端的关联及层次在第四节中详细)，此时还没有通过 mem_init() 建立内核的虚存管理机制，所以不能通过 ioreamp 来进行物理内存到虚存的映射(物理内存必须由内核映射成系统管理的虚拟内存后才能进行读写访问)，这与先前所讲的 framebuffer 的物理内存映射是不同的，具体原因如下:

✓ 终端在注册并使用的调用路径如下:

start_kernel→console_init→uart_console_init→ep93xxuart_console_init→register_console→csambuart_console_write.

✓ FrameBuffer 显卡驱动中的物理内存映射调用路径如下:

start_kernel→ rest_init→init(内核初始线程)→ do_basic_setup→ do_initcalls→fbmem_init→lanrryfb_init

(Linux 下用__setup 启动初期初始机制与__initcall 系统初始化完成后的调用机制，这两个机制本质没有什么差别，主要是执行时所处的系统时段)

✓ 串口物理内存映射到虚存的时机:

依据上面所介绍的两条执行路径，再看内核的内存初始化的调用时期，只有完成这个初始化后才能进行物理内存到虚存的映射，内存的初始化主要是在 start_kernel 中调用的 mem_init，这个调用明显在 uart_console_init 之后，在 fbmem_init 之后，到此就全部说明了为何不能在对串口使用 ioremap 进行物理内存的映射了。那么究竟要在什么时机用什么方法进行串口物理内存的映射呢？

✓ 串口物理内存的映射方式:

参考 ep93xx 的板载 I/O 的映射处理，它的处理方式是一次性将所有的物理 I/O 所在的内存空间映射到虚存空间，映射的基址是 IO_BASE_VIRT，大小是 IO_SIZE.

```
/* Where in virtual memory the IO devices (timers, system controllers
 * and so on).  This gets used in arch/arm/mach-ep93xx/mm.c.*/
#define IO_BASE_VIRT            0xFF000000      // Virtual address of IO
#define IO_BASE_PHYS            0x80000000      // Physical address of IO
```

```
#define IO_SIZE                 0x00A00000      // How much?
```

完成映射的函数是 ep93xx_map_io，所有要进行映射内存都在 ep93xx_io_desc 结构当中描述，我们的串口映射也加在这个地方，基址分别如下:

文件: linux-2.4.21/include/asm-arm/arch-ep93xx/regmap.h

```
#define IO_W83697_UART_BASE     0x20000000
#define IO_W83697_UART_SIZE     0x1000
#define IO_W83977_UART_BASE     0x30000000
#define IO_W83977_UART_SIZE     0x1000
#define IO_SIZE_2               (IO_SIZE+0x100000)
#define IO_BASE83697_VIRT       IO_BASE_VIRT+IO_SIZE
#define IO_BASE83977_VIRT       IO_BASE_VIRT+IO_SIZE_2
```

ep93xx_map_io 完成是在 arch 初始化中赋值给 struct machine_desc mdesc 这个机器描述结构体,主要由位于 mach-ep93xx\arch.c 文件中如下宏完成此结构的初始化:

```
MACHINE_START(EDB9302, "edb9302")
    …..
    MAPIO(ep93xx_map_io)      //初始化. map_io= ep93xx_map_io….
MACHINE_END
```

最终这个函数在调用路径如下:

start_kernel→setup_arch→paging_init→(mdesc->map_io())

至此完成串口物理内存的映射，这个过程在 console_init 调用之前，因此不会有问题，此种方法建立映射是直接创建物理内存页与虚存页的对应，大小为 4k 一页，最终调用的是 create_mapping()，建立页表映射是与具体的平台相关的，位于 mach_ep93xx/mm/ proc-arm920.S 文件中提供了与具体平台相关的页表建立函数，其中包括 TLB 表操作/Cache 操作/页表操作等：

在上层的 start_kernel→setup_arch→ setup_processor 调用下，会在 proc-arm920.S 文件中查找".proc.info"节的__arm920_proc_info，并从中找到配置的 process 相关的操作函数，具体的 arm 页表建立的详情须要参看 ARM 内存管理的相关手册.

```
.section    ".proc.info", #alloc, #execinstr
.type       __arm920_proc_info,#object
__arm920_proc_info:
.long       0x41009200
  ……
.long       arm920_processor_functions
.size       __arm920_proc_info, . - __arm920_proc_info
```

在 arm920_processor_functions 中包含的页表操作如下:

```
/* pgtable */
.word       cpu_arm920_set_pgd
```

```
.word       cpu_arm920_set_pmd
.word       cpu_arm920_set_pte
```

2. 与串口硬件相关的宏主.

如下，下面将详术如下，并指出其具体被使用的环境上下文:

<1>. 读写数据.

```
#define UART_GET_CHAR(p)            ((readb((p)->membase + W83697_UARTDR)) & 0xff)
#define UART_PUT_CHAR(p, c)         writeb((c), (p)->membase + W83697_UARTDR)
```

<2>. 接收发送状态.

```
#define UART_GET_RSR(p)             ((readb((p)->membase + W83697_UARTRSR)) & 0xff)
#define UART_PUT_RSR(p, c)          writeb((c), (p)->membase + W83697_UARTRSR)
```

<3>. 发送及接收中断状态.

```
#define UART_GET_CR(p)              ((readb((p)->membase + W83697_UARTCR)) & 0xff)
#define UART_PUT_CR(p,c)            writeb((c), (p)->membase + W83697_UARTCR)
#define UART_GET_INT_STATUS(p)      ((readb((p)->membase + W83697_UARTIIR)) & 0xff)
```

<4>. 以及其它的中断使能设置等，在传送时打开传送中断即会产生传送中断.

```
#define UART_PUT_ICR(p, c)      writeb((c), (p)->membase + W83697_UARTICR)
```

<5>. FIFO 的状态，是否读空/是否写满；每次读时必须读至 FIFO 空，写时必须等到 FIFO 不满时才能写(要等硬件传送完) .

接收中断读空 FIFO 的判断:

```
status = UART_GET_FR(port);
while (UART_RX_DATA(status) && max_count--) {
        ……
}
```

发送中断写 FIFO: 当发送缓冲区中有数据要传送时，置发送中断使能，随后即产生传送中断，此时 FIFO 为空，传送半个 FIFO 大小的字节，如果发送缓冲区数据传完,则关闭发送中断.

<6>. 传送时可直接写串口数据口，而不使用中断，但必须等待检测 FIFO 的状态

```
do {
        status = UART_GET_FR(port);
} while (!UART_TX_READY(status));       //wait for tx buffer not full...
```

3. 串口驱动的参数配置

串口的参数主要包括如下几个参数，全部都记录在 uart_port 结构上，为静态的赋值，本串口驱动支持 6 个设备，所以驱动中就包括了 6 个 port，一个串口对应一个 port 口，他们之间除了对应的中断号/寄存器起始基址/次设备号不同之外，其它的参数基本相同.

- ✓ 串口对应中断，这里六个串口，其中有 3 个串口使用的系统外部中断 0/1/2，其中另外几个中断用提 GPIO 中断，具体有关 GPIO 中断的内容可参见 EP93XX 芯片手册, GPIO 中断共享一个系统中断向量，涉及中断共享的问题，后面将详述 LINUX 中的中断共享支持.
- ✓ 串口时钟，串口时钟用来转换计算须要设置到配置寄存器当中的波特率比值，其计算方法为：quot = (port->uartclk / (16 * baud)); baud 为当前设置的波特率，可为 115200 等值，取决于所选的串口时钟源, quot 即为要设置到寄存器当中的比值.
- ✓ 串口基址，即串口所有配置寄存器基础址.
- ✓ 串口次设备号(由驱动中的最低次设备号依次累加)

前面已经讲过了六个串口中断，这里详细列出对应情况如下，方便查找:

w83697 的三个串口对应中断如下:

- ✓ uart 1: 读写数据寄存器偏移为 00x3F8，对应系统外部中断 INT_EXT[0].
- ✓ uart 2: 读写数据寄存器偏移为 00x2F8，对应系统外部中断 INT_EXT[1].
- ✓ uart 3: 读写数据寄存器偏移为 00x3e8，对应系统外部中断 INT_EXT[2].
- ✓ uart 4: 读写数据寄存器偏移为 00x3e8，对应 EGPIO[8].

w83977 的两个串口对应中断如下:

- ✓ uart 1: 读写数据寄存器偏移为 00x3F8，对应 EGPIO[1].
- ✓ uart 2: 读写数据寄存器偏移为 00x2F8，对应 EGPIO[2].

下面列出其中一个具体的串口 port 的定义如下:

```
{
        .port  = {
                .membase   = (void *)W83697_UART4_BASE,
                .mapbase   = W83697_UART4_BASE,
                .iotype        = SERIAL_IO_MEM,
                .irq           = W83697_IRQ_UART4,     //串口中断号
                .uartclk       = 1846100,          //uart 时钟，默认.
                .fifosize      = 8,                //硬件 fifo 大小.
                .ops           = &amba_pops,  //底层驱动的硬件操作集，如开关中断等.
                .flags = ASYNC_BOOT_AUTOCONF,
                .line          = 3,                //串口在次设备数组中的索引号，须注意从 0 计起…
        },
        .dtr_mask   = 0,
        .rts_mask   = 0,
}
```

4. 串口驱动的底层接口函数

驱动文件:linux-2.4.21/drivers/serial/Ep93xx_w83697.c
相关文件: linux-2.4.21/drivers/serial/core.c 下面详述.

函数: w83697uart_rx_chars(struct uart_port *port, struct pt_regs *regs)
描述: 串口接收数据中断, 此函数中应当注意的要点如下:

- ✓ 接收数据时，要注意判断 FIFO 是否读空(参见上述 2 点中说明).
- ✓ 接收数据放入 flip 缓冲区，此缓冲区专供缓存中断中接收到的数据，是最原始的串口数据，为更上一层中各种终端处理模式的原始数据，可以进行各种加工处理。
- ✓ 接收数据到 flip 缓冲区中时，须根据硬件接收状态，置每一个接收到的字符的接收标志，放在 flag_buf_ptr 当中, 标志类型有 TTY_NORMAL/ TTY_PARITY/ TTY_FRAME 等，分别表示正常/校验出错/帧出错(无停止位)等.
- ✓ 每接收数据之后，会通过在退出中断前调用 tty_flip_buffer_push()来往 tq_timer 任务列表中加一个队列任务，串口的队列任务主要是负责将中断接收到 flip 缓冲区中的数据往上传输至终端终冲区, 队列任务的机制将在后面介绍，它是一种异步执行机制，在软中断中触发执行.

函数: static void w83697uart_tx_chars(struct uart_port *port)
描述: 串口发送数据中断, 发送中断中要做的事比较少，比起接收中断简单了好多，注意事项如下:

- ✓ 当上层要发送数据时，就会打开串口发送中断，此时 FIFO 为空，传送半个 FIFO 大小数据即退出, 通常打开中断是通过更上一层的 uart_flush_chars()调用，最终调用的是 w83697uart_start_tx().
- ✓ 检测当没有数据要传输的时候，关闭传送中断，在传送之前与传送完之后都有检测.
- ✓ 最重要的一点是如果传送缓冲区当中的字符数已经小于 WAKEUP_CHARS, 则可以唤醒当前正在使用串口进行传送的进程，这里是通过 tasklet 机制来完成，这也是一异步执行机制.

顺带介绍开关中断接口:
static void w83697uart_start_tx(struct uart_port *port, unsigned int tty_start)
static void w83697uart_stop_tx(struct uart_port *port, unsigned int tty_stop)

函数: static void w83697uart_int(int irq, void *dev_id, struct pt_regs *regs)
描述: 中断处理函数，为 3 个使用系统外部中断的的串口的中断入口，其中必须处理的中断状态分为如下几种, 注意必须在处理中断时根据手册中的说明来清除中断，通常是读或写某些寄存器即可。

- ✓ 接收中断.
- ✓ 传送中断.
- ✓ FIFO 超时中断.
- ✓ 其它不具体处理的中断，必须读相应寄存器清中断.

函数: static void w83697uart_int2(int irq, void *dev_id, struct pt_regs *regs)

描述：中断处理函数，为另外几个使用串口使用的 GPIO 中断入口，GPIO 中断共享同一个系统中断向量，必须根据 GPIO 的中断状态寄存器的相应位来判断对应的中断是属哪一个串口的，从而进行相应的处理，其实这个判断也是无所谓的，因为中断产生时传进来的参数已经含有了相应串口的参数，在判断完中断产生的 GPIO 口后立即调用 w83697uart_int2 完成具体的中断处理.

函数: static int w83697uart_startup(struct uart_port *port)
描述：串口开启后的初始化函数，主要完成初始化配置，以及安装中断处理了函数，初始化配置包括打开中断使能标志。

函数: static void w83697uart_shutdown(struct uart_port *port)
描述：串口关闭函数，清除配置，半闭中断.

函数: static void w83697uart_change_speed(struct uart_port *port, unsigned int cflag, unsigned int iflag, unsigned int quot)
描述：配置函数，经由上次调用下来，主要配制串口的波特率比，以及各种容错处理，在串口打开初始化时会被调用，在必变串口波特率/校验方式/停止位/传送位数等参数时会被调用.

5. 串口驱动与上层的接口关联

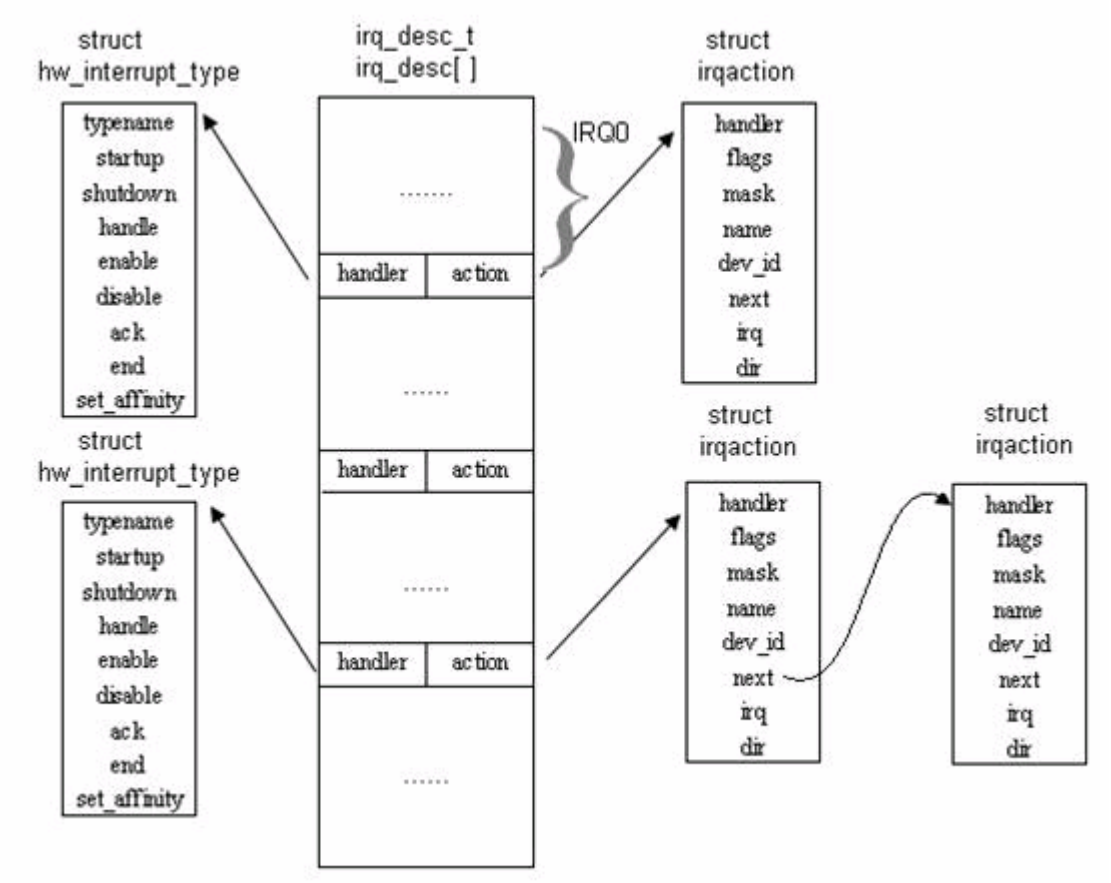
文件: linux-2.4.21/drivers/serial/core.c
这一层接口是串口驱动中的共用部分代码，核心结构为 struct uart_driver. 这一层上承 TTY 终端，下启串口底层，串口底层主要处理了与串口硬件相关的部分，并向上提供 uart 中间层向下的接口. Uart coar 向下与底层驱动的接口，通过一个 static struct uart_ops amba_pops 结构完成，这个结构直接赋值给串口 struct uart_amba_port amba_ports 的.ops 成员，最后将串口的 port 加入到 uart_driver 当中完成关联，通过 uart_add_one_port 加入.

```
static int __init w83697uart_init(void)
{
	int ret, i;
	ret = uart_register_driver(&amba_reg);
	if (ret == 0) {
		for (i = 0; i < UART_NR; i++)
			uart_add_one_port(&amba_reg, &amba_ports[i].port);
	}
	return ret;
}
```

二. Linux 的中断机制及中断共享机制.

前面讲到了有 6 个串口，除了 w83697 中的前三个串使用的是独立的系统外部中断之外，其它的在个串口是共享一个系统中断向量的，现在我们来看看多个中断是如何挂在一个系统中断向量表当中的，共享中断到底是什么样的一种机制?

进行分析代码可知，linux 下的中断采用的是中断向量的方式，每一个中断对应一个中断描述数组当中的一项，结构为 struct irqdesc，其当中对应一成员结构为 struct irqactionr 的成员 action，这个即表示此中断向量对应的中断处理动作，这里引用从网上下载的一幅图讲明中断向量表与中断动作之间的关系:



摘自IBM developerWorks
中国

```
struct irqaction {
        void (*handler)(int, void *, struct pt_regs *);
        unsigned long flags;
        unsigned long mask;
        const char *name;
        void *dev_id;
        struct irqaction *next;
```

};

从上面的结构体与图当中，我们就可以很清楚的看到，一个中断向量表可以对应一个 irqaction，也可能对应多个由链表链在一起的一个链表 irqaction，这当中主要在安装中断的时候通过中断的标志位来决定:

- ✓ 安装中断处理，不可共享：

```
retval = request_irq(port->irq, w83697uart_int, 0, "w83697_uart3", port);
```

- ✓ 安装中断处理，可共享：

```
retval = request_irq(port->irq, w83697uart_int2, SA_SHIRQ, "w83977_uart5", port);
```

由上即可知，安装共享中断时，只须指定安装的中断标志位 flag 为 SA_SHIRQ, 进入分析安装中断的处理可知，在安装时，会检测已经安装的中断是否支持共享中断，如果不支持，则新的中断安装动作失败；如果已经安装的中断支持共享中断，则还必须检测将要安装的新中断是否支持中断共享，如果不支持则安装还是会失败，如果支持则将此新的中断处理链接到此中断向量对应的中断动作处理链表当中.

在产生中断时，共享中断向量中对应的中断处理程序链表中的每一个都会被调用，依据链表的次序来，这样处理虽然会有影响到效率，但是一般情况下中断传到用户的中断处理服务程序中时，由用户根据硬件的状态来决定是否处理中断，所以能常情况下都是立即就返回了，效率的影响不会是大的问题.

三. Linux 的软中断机制.

前面已经简单讲过了 LINUX 下的硬中断处理机制，其实硬中断的处理都由 LINUX 底层代码具体完成了，使用者一般在处理硬中断时是相当简单的，只须要用 request_irq()简单的挂上中断即可，这里我们进一步介绍一下 LINUX 下的软中断机制，软中断机制相比起硬中断机制稍微复杂一些，而且在 LINUX 内核本身应用非常的广，它作为一种软性的异步执行机制，只有深入理解了它才能灵活的运用.

之所以提到内核的 softirq 机制，主要是因为在串口中断也使用了这些机制，理解了这些机制就能更加明白串口驱动一些问题，现在先提出几个问题如下:

- ✓ 前面提供到中断接收后数据，先放到 flip 缓冲区当中，这样让人很容易进一步想知道，中断处理的缓冲区的数据，用户进程读取串口时如何读到的？很明显中断处于内核空间，用户读取串口输入进程是在用户空间，中断缓冲区中的数据如何被处理到终端缓冲区中，供用户读取的？
- ✓ 另外写串口时，是向终端缓冲区当中写入，那么上层的写操作如何知道下层缓冲区中的的数据是否传送完成？用户空间的写串口进程处于什么样的状态？如果是写完缓冲区就睡眠以保证高效的 CPU 使用率，那么何时才应该醒过来？ 由谁负责醒过来？

1. 往 tq_timer 任务队列中添加一项任务.

根据以上这两个问题，我们来深入代码分析，首先看接收缓冲区中的数据如何上传，前面已经提到过，接收中断处理完成后，会调用 tty_flip_buffer_push()，这个函数完成的功能就是往一系统定义的任务队列当中加入一个任务，下面我们将详细的分析加入的任务最终是如何执行起来的.［任务：这里所讲的任务可以直接理解成为一个相应的回调函数，ＬＩＮＵＸ下术语称作 tasklet］

```
void tty_flip_buffer_push(struct tty_struct *tty)
{
    if (tty->low_latency)
        flush_to_ldisc((void *) tty);
    else
        queue_task(&tty->flip.tqueue, &tq_timer);
}
```

2. tq_timer 的执行路径分析.

tq_timer 是一个双链表结构任务队列,每项任务包含一个函数指针成员，它通过 run_task_queue 每次将当中的所有任务(其实是一些函数指针)全部调用一次，然后清空队列,最终的执行 tq_timer 的是在中断底半的 tqueue_bh 中执行，如下:

```
void tqueue_bh(void)
{
    run_task_queue(&tq_timer);
}
```

在 void __init sched_init(void)当中初始化底半的向量如, tqueue_bh 初始化在 bh_base 的 TIMER_BH 位置，bh_base 为一结构很简单的数组，在什么位置调用什么样的了函数基本已经形成默认的习惯：

```
init_bh(TIMER_BH, timer_bh);
init_bh(TQUEUE_BH, tqueue_bh);
init_bh(IMMEDIATE_BH, immediate_bh);
```

看看 init_bh 相当于初始底半的服务程序，非常简单：

```
void init_bh(int nr, void (*routine)(void))
{
    bh_base[nr] = routine;
    mb();
}
```

最终真正的执行 bh_base 中保存的函数指针的，在 bh_action()当中：

```
static void bh_action(unsigned long nr)
{
…
    if (bh_base[nr])
        bh_base[nr]();
…
}
```

关于这里所指出的 bh_base， 我们在后面就直接称作 bh，意即中断底半所做的事.

3. tq_timer 实现所依赖的 tasklet.

那么 bh_action 在什么时候执行呢？bh_action 被初始化成 bh_task_vec 这 32 个 tasklet 调用的任务，因此它的依赖机制是 tasklet 机制，后面将进行简单介绍.

```
void __init softirq_init()
{
    int i;
    for (i=0; i<32; i++)
        tasklet_init(bh_task_vec+i, bh_action, i);
    ….
}
```

至此已经把任务队列的执行流程及原理分析完成，tasklet 是须要激活的，这里我们先指出任务队列是如何激活的，在时钟中断的 do_timer()当中会调用 mark_bh(TIMER_BH)，来激时钟底半所依赖运行的 tasklet，其中 bh_task_vec 的所有成员的函数指针全部指向 bh_action.

```
static inline void mark_bh(int nr)
{
    tasklet_hi_schedule(bh_task_vec+nr);
}
```

tasklet_hi_schedule 的功能就是往 tasklet 当中加入一个新的 tasklet.

4. tasklet 的机制简单分析.

讲到 tasklet，我们才与我们真正要讲的 softirq 最近了，因为目前在软中断当中有主要的应用就是 tasklet，而且在所有 32 个软中断中仅有限的几个软中断如下：

```
enum{
        HI_SOFTIRQ=0,
        NET_TX_SOFTIRQ,
        NET_RX_SOFTIRQ,
        TASKLET_SOFTIRQ
};
struct softirq_action{
        void  (*action)(struct softirq_action *);
        void  *data;
};
static struct softirq_action softirq_vec[32] __cacheline_aligned;//软中断的中断向量表，实为数组.
```

[1]. 初始化软中断向量.

我们这里所要讲的，就是 HI_SOFTIRQ / TASKLET_SOFTIRQ 两项，据我理解这两项根本在实现机制上一样的，之所以分开两个名字叫主要是为了将不同的功能分开，就类似于虽然同是软中断，但是各处所完成的功能不一样，所以分在两个软中断完成，后面我们仅取其中用于执行时钟底半的任务队列 HI_SOFTIRQ 为例进行讲解，而且我们不讲及多个ＣＰＵ情况下的 tasklet 相关机制，这两项软中断的实始化如下:

```
void __init softirq_init()
{
        ....
        open_softirq(TASKLET_SOFTIRQ, tasklet_action, NULL);
        open_softirq(HI_SOFTIRQ, tasklet_hi_action, NULL);
}
```

open_softirq 下所做的事相当简单，即往软中断向量中赋值，相当于硬中断当中的 request_irq 挂硬件中断：

```
void open_softirq(int nr, void (*action)(struct softirq_action*), void *data)
{
        softirq_vec[nr].data = data;
        softirq_vec[nr].action = action;
}
```

[2]. 软中断中断服务程序

对于 HI_SOFTIRQ，相应的中断服务程序为 tasklet_hi_action，由上文所讲的初始化过程给出，这个函数目前完成的功能相当简单，它的任务就是遍历执行此中断所对应一个 tasklet

链表， NR_CPUS= 1.

```
struct tasklet_head tasklet_hi_vec[NR_CPUS] __cacheline_aligned;
```

[3]. 往软中断对应的 tasklet 链表中加入新的 tasklet, 加在尾部.

```
void __tasklet_hi_schedule(struct tasklet_struct *t)
{
…
        t->next = tasklet_hi_vec[cpu].list;
        tasklet_hi_vec[cpu].list = t;
        cpu_raise_softirq(cpu, HI_SOFTIRQ);
…
}
```

最重要的一点是，在安装了新的 tasklet 后，还必须将软中断设置为激活，告诉系统有软中断须要执行了，下面一点即提到系统如何检测是否有软中断须要处理:

```
#define __cpu_raise_softirq(cpu, nr) do { softirq_pending(cpu) |= 1UL << (nr); } while (0)
```

[4]. 软中断所依赖的执行机制.

讲到最后还没有指出软中断是如何触发执行的，其实很简单:

- ✓ 在系统处理所有硬中断信号时，他们的入口是统一的，在这个入口函数当中除了执行 do_IRQ()完成硬件中断的处理之外，还会执行 do_softirq()来检测是否有软中断须要执行，所以软中断所依赖的是硬件中断机制;
- ✓ 另外还有一个专门处理软中断内核线程 ksoftirqd()，这个线程处理软中断级别是比较低的，他是一个无限 LOOP 不停的检测是否有软中断须要处理，如果没有则进行任务调度.

在 do_softirq()中有如下的判断，以决定是否有软中断须要执行，如果没有就直接退出，在[3]中提到的激活软中断时，要将相应软中断位置 1，软中断有 32 个，因此一个整型数即可以表示 3 2 个软中断，即可判断有什么样的软中断须要处理，代码如下:

```
        pending = softirq_pending(cpu);
        if (pending) {
        }
        ….
        do {  //检测 32 个软中断位标志中是否有为 1 的…
                if (pending & 1)
                        h->action(h);
                h++;
                pending >>= 1;
        } while (pending);
```

[4]. 软中断所依赖的执行时期问题.

之所以将这个问题单独列开来讲，是因为他特别的重要，上面我已经讲过了软中断是依赖硬中断触发执行的，但是产生如下疑问：

- ✓ 是不是一有硬中断发生就会触发软中断的执行？
- ✓ 软中断的执行会不会影响到系统的性能？
- ✓ 会不会影响到硬中断的处理效率？也就是说会不会导致在处理软中断时而引起硬中断无法及时响应呢？

再看 do_softirq 的代码当中有如下判断:

```
if (in_interrupt())
    return;
```

这个条件就是能否进行软中断处理的关键条件，因此由此也可以了解到软中断是一种优先级低于硬中断的软性机制，具体来看看这个判断条件是什么:

```
/*Are we in an interrupt context? Either doing bottom half
 * or hardware interrupt processing?*/
#define in_interrupt() ({ const int __cpu = smp_processor_id(); \
    (local_irq_count(__cpu) + local_bh_count(__cpu) != 0); })
/* softirq.h is sensitive to the offsets of these fields */
typedef struct {
    unsigned int __softirq_pending;
    unsigned int __local_irq_count;
    unsigned int __local_bh_count;
    unsigned int __syscall_count;
    struct task_struct * __ksoftirqd_task; /* waitqueue is too large */
} ____cacheline_aligned irq_cpustat_t;
#define irq_enter(cpu,irq)    (local_irq_count(cpu)++)
#define irq_exit(cpu,irq)     (local_irq_count(cpu)--)
```

看到这里，不得不再多注意一个结构，那就是 irq_cpustat_t, 先前我们讲是否有软中断产生的标志位，但没有提到__softirq_pending，这个变量就是记载 32 个软中断是否产生的标志，每一个软中断对应一个位；在中断执行的 do_softirq 中有如下几个重要的动作，说明如下:

- ✓ in_interrupt 判断是否可以进行软中断处理，判断的条件就是没有没处在硬件中断环境中，而且还没有软中断正在执行（即不允许软中断嵌套），软中断的嵌套避免是通过 local_bh_disable()/local_bh_enable()实现，至于带有 **bh，其意也即指 softirq 是中断底半(bh),** 在处理硬件中断时，一进行即会调用 irq_enter 来表示已经进入硬件中断处理程序，处理完硬件中断后再调用 irq_exit 表示已经完成处理;
- ✓ pending 判断是否有软中断须要处理, 每个位用作当作一个软中断是否产生的标志.
- ✓ 清除所有软中断标志位，因为下面即将处理; 但清除之前先缓存起来, 因为下面还要使用这个变量一次.
- ✓ 在进入软中断处理后，会关闭 bh 功能的执行，执行完后才打开，这样在 in_interrupt 判断当中就会直接发现已经有 bh 在执行，不会再次进入 bh 执行了，这严格保证了 bh 执行的串行化.
- ✓ 打开硬件中断，让软中断在有硬件中断的环境下执行.
- ✓ 处理完软中断后关闭硬中断，再次检测是否有新的软中断产生，如果有的话，却只须立即处理本次软中断过程未发生过的软中断向量. 之所以会有新的软中断产生，那是因为软中断是在开硬件中断的情况下执行，硬件中断处理是可能又产生了新的软中断. 之所以只处理本次软中断未发生的软中断向量，依据我自己的理解，其目的是为了不加重软

中断处理的负担而不马上处理，只是相应的唤醒一个 wakeup_softirqd 线程，这是专门处理软中断的，这样虽然延误了软中断的处理，但避免了在硬中断服务程序中拖延太长的时间.[关于软中断的处理在后绪版本变化也很大，可以进一步学习研究，如何使软中断不至影响中断处理效率]

软中断处理这个函数虽然不长，但是相当的关键，每一句代码都很重要，结合上面所说的几点，与源码交互起来理解才能根本理解软中断的设计机制：

```
asmlinkage void do_softirq()
{
        int cpu = smp_processor_id();
        __u32 pending;
        unsigned long flags;
        __u32 mask;
        if (in_interrupt())         return;
        local_irq_save(flags);
        pending = softirq_pending(cpu);
        if (pending) {
                struct softirq_action *h;
                mask = ~pending;
                local_bh_disable();
restart:
                /* Reset the pending bitmask before enabling irqs */
                softirq_pending(cpu) = 0;
                local_irq_enable();
                h = softirq_vec;
                do {
                        if (pending & 1)
                                h->action(h);
                        h++;
                        pending >>= 1;
                } while (pending);
                local_irq_disable();
                pending = softirq_pending(cpu);
                if (pending & mask) {
                        mask &= ~pending;
                        goto restart;
                }
                __local_bh_enable();
                if (pending)
                        wakeup_softirqd(cpu);
        }
        local_irq_restore(flags);
}
}
```

四. TTY 与串口的具体关联.

串口设备可以当作 TTY 终端来使用，这又使串口设备比一般的设备稍微复杂一些，因为他还必须与终端驱动关联起来，虽然这部分与 TTY 的关联已经是属于公用部分的代码，并不须要驱动编写者特别做些什么来进行支持，但对它与 TTY 的层次关联的了解有助于理解整个串口的数据流向.

串口要能够成为终端，必须客外加入终端注册及初始化的代码，这部分很简单，基本上所有的串口驱动都是固定的模式，并无什么修改，主要包括如下结构:

```
static struct console cs_amba_console = {
        .name           = "ttyBM",
        .write          = w83697uart_console_write,
        .device         = w83697uart_console_device,
        .setup          = w83697uart_console_setup,
        .flags          = CON_PRINTBUFFER,
        .index          = -1,
};
```

串口终端的注册通过下面的函数，将 cs_amba_console 注册成为终端，这个函数调用路径是:

```
start_kernel()→console_init()→ep93xxuart_w83697_console_init()
void __init ep93xxuart_w83697_console_init(void)
```

终端会对应一种具体设备的 driver，相对于串口这个结构是 uart_driver，在驱动中我们已经提供了一个这样的结构. static struct uart_driver amba_reg, uart_register_driver 会将它注册成为终端对应的 driver，因此真正串口与终端的关联就在此处建立.

函数: static int __init w83697uart_init(void)
描述: 调用 uart_register_driver()完成串口与终端的关联，将串口注册成为一种 TTY 设备，在 uart_register_driver()当中调用 tty_register_driver()完成 TTY 设备注册; 其次是完成串口 port 口的注册，将静态描述的所有串口 port(结构为 struct uart_port)注册到 uart_driver 当中.

特别说明: 注册串口 TTY 设备时，由于历史的原因会注册两个 TTY 设备，一个是 normal，另一个是 callout，是两个设备来的，在我们这里两者没有什么差别，请看源码中的注解:

```
.normal_name        = "ttyBM",
.callout_name       = "cuaam",
/*
 * The callout device is just like the normal device except for
 * the major number and the subtype code.
 */
```

函数: static void __exit w83697uart_exit(void)
描述: 卸截设备，卸截 port 口，因为我编译的驱动是与内核绑定在一起的，因此实际上根本不会调用此函数.

五. 串口测试使用示例说明

为了测试串口驱动，提供一个测试示例，基本使用如下：

示例文件：uarttest-2.c
位　　置：与文档一起提供. ☺

1. 本示例支持同时打开多个串口进行测试，可以测试 ep93xx 所支持的八个串口，其中包括本身提供的两个串口.
2. 可以进行串口速度/校验/停止位等设置，但打开多个串口时设置只能设置所有串口相同.
3. 测试过程中，会自动统计每个串口收到的数据，可以使用“串口调试助手 2.2”协助测试，并对比发送的与接收数据是否相等.
4. 如要停止以查看结果，须等待 15 秒程序即会打印出各个串口的收发统计信息.
5. 具体操作:
   [1]. 首先选择须要打开的串口，用数字 0~7 代表如下的八个串口，如果选择完成输入数字 8.
   [2]. 接着选择须要设置的串口速度，按提示来.
   [3]. 其次设置奇偶性.

6. ep93xx 所支持的所有串口设备名称列表:

```
{
"/dev/ttyAM0",      //ep93xx 自带的两个串口…
"/dev/ttyAM1",
"/dev/ttyBM0",      //以后六个为 w83697 支持的四个串口…
"/dev/ttyBM1",
"/dev/ttyBM2",
"/dev/ttyBM3",
"/dev/ttyBM4",      //以后为 w83977 支持的两中串口…
"/dev/ttyBM5",
};
```