先从sci_receive_chars讲起（讲解的顺序就是一行一行的来，遇到函数就跳进去看，出来了继续下一行），这个函数被中断处理函数调用，当寄存器SCFRDR寄存器接收到数据后，产生中断，SCFRDR寄存器最多存储16个字节数据，已经存储了几个字节数据就触发中断是由SCFCR寄存器的RTRG位指定。在sci_receive_chars函数开始的status = serial_port_in(port, SCxSR);既判断当前SCFRDR寄存器是否已满（就是达到RTRG指定的个数），否则退出。

接下来的while(1) 循环，即开始接收数据，复制SCFRDR寄存器中的数据到tty_buffer。

讲tty_buffer_request_room函数之前，先了解一下tty_buffer的管理;

```
struct tty_bufhead {



        struct tty_buffer *head;          /* 指向接收buffer的头*/

        struct work_struct work;

        struct mutex       lock;

        atomic_t           priority;

        struct tty_buffer sentinel;

        struct llist_head free;              /* 空闲tty_buffer队列，当head中的数据被读取之后，会把该节点插入free队列 */

        atomic_t           mem_used;    /*所有tty_buffer已经使用了的内存总和 */

        int                mem_limit;     /*tty_buffer所能够使用的内存最大值(64k) */

        struct tty_buffer *tail;              /* Active buffer */

};
```

```
struct tty_buffer {



        union {

                struct tty_buffer *next;

                struct llist_node free;

        };            //链表的使用方式，直接next，还是list_node。  head使用next方式，free使用list_node。

        int used;     //该buffer已经使用了的大小

        int size;           //该buffer的大小

        int commit;   //该buffer已经写入了多少数据

        int read;           //该buffer已经有多少数据被读

        int flags;          //数据标识，标识当前的data中的数据是否含有标志位

        /* Data points here */

        unsigned long data[0];   //存储数据

};
```

我们可以看到tty_bufhead结构用来管理tty_buffer，head指向一个tty_buffer结构链表的表头，tail指向这个链表的尾部，并且每次读取到的数据都是向tail所指向的tty_buffer节点中插入数据，tail数据满了之后就新添加一个tty_buffer向后移动，上层读取数据的时候就从head开始读取，从tty_buffer的read处开始读取，读到commit位置就说明这个tty_buffer中的数据已经被读完了，此时，head = head->next,并切把之前的head添加到free空闲列表。上面说的tail寻找新的tty_buffer就是从free中获取，如果没有，就malloc。这里需要特

别说明一下tty_buffer中的这个data成员，data在使用的时候转化为char指针(因为读取数据都是一个一个字节的读取），tty_buffer在申请的时候大小为sizeof(struct tty_buffer) + 2 * size，这里是 2 倍的size，这是因为接收到数据都是有一个标识位的，即表明每一个字节是否有效，所以一个size大小的数据，需要有一个size大小的空间存储标识位，当然如果你在flags中表明了不需要标示位，那么你就有 2 ×size大小的空间存储数据了。

在调用__tty_buffer_request_room之前，先调用sci_rxfill(port)计算了SCFRDR寄存器中已经有多少个字节的数据了。

好了，现在来看看__tty_buffer_request_room函数源码：

```
1.  static int __tty_buffer_request_room(struct tty_port *port, size_t size,
2.                                      int flags)
3.  {
4.          struct tty_bufhead *buf = &port->buf;
5.          struct tty_buffer *b, *n;
6.          int left, change;
7.
8.          b = buf->tail;      //当前正在使用的tty_buffer
9.          if (b->flags & TTYB_NORMAL)     //当前的tty_buffer是否需要存储标识位
10.                 left = 2 * b->size - b->used;  //不需要存储标识位，那么你就有2倍的size空间来存储数据，计算出剩余的空间
11.         else
12.                 left = b->size - b->used;
13.
14.         change = (b->flags & TTYB_NORMAL) && (~flags & TTYB_NORMAL);     //当前的tty_buffer是否和将要存储的数据类型一样(是否需要存储标识位，如果需要应重新获取一个tty_buffer)
15.         if (change || left < size) {    //改变了存储类型，  或者当前的tty_buffer剩余的空间不够本次数据的存取
16.                 /* This is the slow path - looking for new buffers to use */
17.                 n = tty_buffer_alloc(port, size);     //get一个新的tty_buffer


18.                 if (n != NULL) {
19.                         n->flags = flags;
20.                         buf->tail = n;
21.                         /* paired w/ acquire in flush_to_ldisc(); ensures
22.                          * flush_to_ldisc() sees buffer data.
23.                          */
24.                         smp_store_release(&b->commit, b->used);
25.                         /* paired w/ acquire in flush_to_ldisc(); ensures the
26.                          * latest commit value can be read before the head is
27.                          * advanced to the next buffer
28.                          */
29.                         smp_store_release(&b->next, n);
30.                 } else if (change)
31.                         size = 0;
32.                 else
33.                         size = left;
34.         }
35.         return size;
36. }
```

该函数先求出了当前(tail)tty_buffer中剩余空间大小，并判断了一下当前需要存储的数据和当前buffer类型(flags)是否一样（即一个是需要存储标识位，一个不需要存储位），如果不一样或者当前buffer空间不够，那么就需要新拿一个tty_buffer，最后获得了新的tty_buffer后，初始化他。下面看看这个tty_buffer_alloc函数：

```
static struct tty_buffer *tty_buffer_alloc(struct tty_port *port, size_t size)


{
        struct llist_node *free;
        struct tty_buffer *p;

        /* Round the buffer size out */
        size = __ALIGN_MASK(size, TTYB_ALIGN_MASK);

        if (size <= MIN_TTYB_SIZE) {
                free = llist_del_first(&port->buf.free);
                if (free) {
                        p = llist_entry(free, struct tty_buffer, free);
                        goto found;
                }
        }

        /* Should possibly check if this fails for the largest buffer we
           have queued and recycle that ? */
        if (atomic_read(&port->buf.mem_used) > port->buf.mem_limit)
                return NULL;
        p = kmalloc(sizeof(struct tty_buffer) + 2 * size, GFP_ATOMIC);
        if (p == NULL)
                return NULL;

found:
        tty_buffer_reset(p, size);
        atomic_add(size, &port->buf.mem_used);
        return p;
}
```

这个函数上来就对size来了个向上对齐，因为传进来的size是每次需要存储到tty_buffer中的数据的个数，都不一样，所以这里来个向上对其，规范一下tty_buffer链表的大小。接下来做了一个比较if (size <= MIN_TTYB_SIZE)　这是因为free链表中存储的tty_buffer大小都是等于TTYB_ALIGN_MASK大小的。因为把head节点转移到free中的时候，会把size > TTYB_ALIGN_MASK的给释放掉。如果调用llist_del_first函数返回了tty_buffer结构，那么就返回，如果没有找到，那么就计算当前tty_buffer剩余的空间是否满足size大小，如果还够，那么就malloc一个新的tty_buffer结构，反之，return NULL.

至此，tty_buffer_request_room函数结束。总的来说tty_buffer_request_room函数为将要接收的数据准备存储空间并返回将要存储的数据大小。

因为我们是scif串口，所以走else，进入for循环，char c = serial_port_in(port, SCxRDR)　读取SCFRDR寄存器（因为有很多串口，他们共用一个驱动，所以你看到的是SCxRDR）中一个字符，并检查一下读入的这个字符的状态（记得上面提到的，data中存数据的时候，你可以选择存或不存数据的状态）。然后调用tty_insert_flip_char(tport, c, flag)保存字符到tty_buffer。

看一下tty_insert_flip_char的源码：

```
static inline int tty_insert_flip_char(struct tty_port *port,


                                       unsigned char ch, char flag)
{
        struct tty_buffer *tb = port->buf.tail;
        int change;

        change = (tb->flags & TTYB_NORMAL) && (flag != TTY_NORMAL);
        if (!change && tb->used < tb->size) {
                if (~tb->flags & TTYB_NORMAL)
                        *flag_buf_ptr(tb, tb->used) = flag;
                *char_buf_ptr(tb, tb->used++) = ch;
                return 1;
        }
        return tty_insert_flip_string_flags(port, &ch, &flag, 1);
}
```

这个函数就是简单的存储字符（和flags）到tty_buffer, tty_insert_flip_string_flags和__tty_buffer_request_room相似，只是tty_insert_flip_string_flags寻找到新的buffer以后就填充内容了。

for循环count次后，退出for循环，进入while循环，因为在做字符存储的时候，从串口中来的数据会陆续的写入SCFRDR寄存器，所以在去执行sci_rxfill(port)仍然会返回不为 0 个。将继续存储一次（实际上处理一个字符的时间只需要1um，而scif rx脚1ms只能接

收14.4个字节，所以这里是不会循环的，并且如果你设置的是 1 个字节触发一次中断，那么该函数就只会处理一个字节，因为它太快了）。