i2c_pxa_do_xfer()是传输msg[num]　array的core!

```
1.  static int i2c_pxa_do_xfer(struct pxa_i2c *i2c, struct i2c_msg *msg, int num)
2.  {
3.          long timeout;
4.          int ret;
5.
6.          /*
7.           * Wait for the bus to become free.
8.           */
9.          ret = i2c_pxa_wait_bus_not_busy(i2c);
10.         if (ret) {
11.                 dev_err(&i2c->adap.dev, "i2c_pxa: timeout waiting for bus free\n");
12.                 goto out;
13.         }
14.
15.         /*
16.          * Set master mode.
17.          */
18.         ret = i2c_pxa_set_master(i2c);
19.         if (ret) {
20.                 dev_err(&i2c->adap.dev, "i2c_pxa_set_master: error %d\n", ret);
21.                 goto out;
22.         }
23.
24.         if (i2c->high_mode) {
25.                 ret = i2c_pxa_send_mastercode(i2c);
26.                 if (ret) {
27.                         dev_err(&i2c->adap.dev, "i2c_pxa_send_mastercode timeout\n");
28.                         goto out;
29.                         }
30.         }
31.
32.         spin_lock_irq(&i2c->lock);
33.
34.         i2c->msg = msg;
35.         i2c->msg_num = num;
36.         i2c->msg_idx = 0;
37.         i2c->msg_ptr = 0;
38.         i2c->irqlogidx = 0;
39.
40.         i2c_pxa_start_message(i2c);        ①
41.
42.         spin_unlock_irq(&i2c->lock);
43.
44.         /*
45.          * The rest of the processing occurs in the interrupt handler.
46.          */
47.         timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);    ②
48.         i2c_pxa_stop_message(i2c);        ③
49.
50.         /*
51.          * We place the return code in i2c->msg_idx.
52.          */
53.         ret = i2c->msg_idx;
54.
55.         if (!timeout && i2c->msg_num) {
56.                 i2c_pxa_scream_blue_murder(i2c, "timeout");
```

```
57.                     ret = I2C_RETRY;
58.             }
59.
60.      out:
61.             return ret;
62.     }
```

①

i2c_pxa_start_message()发送START signal和i2c device address.

```
1.  static inline void i2c_pxa_start_message(struct pxa_i2c *i2c)
2.  {
3.          u32 icr;
4.
5.          /*
6.           * Step 1: target slave address into IDBR
7.           */
8.          writel(i2c_pxa_addr_byte(i2c->msg), _IDBR(i2c));    (A)
9.
10.         /*
11.          * Step 2: initiate the write.
12.          */
13.         icr = readl(_ICR(i2c)) & ~(ICR_STOP | ICR_ALDIE);    (B)
14.         writel(icr | ICR_START | ICR_TB, _ICR(i2c));        (C)
15. }
```

(A)要传输的内容是i2c device address

(B)去掉STOP signal传输

(C)ICR_START　is for sending START signal

ICR_TB真正启动硬件传输。

这一步就如下图

Question:

但这里并没有等待i2c device发送来的ACK / NAK signal,这一步在哪儿体现呢？

②

由于启动了中断，所以具体的发送/接收都是在interrupt handler中完成的。这里就是把当前thread放入等待queue，

直到传输完毕的条件满足后再resume。i2c->msg_num == 0即表示所有i2c_msg传输完毕。

可以想见在interrupt handler中应该有唤醒等待在wait queue上的thread的动作。

这个动作函数如下：

```
/*
 * i2c_pxa_master_complete - complete the message and wake up.
 */
static void i2c_pxa_master_complete(struct pxa_i2c *i2c, int ret)
{
        i2c->msg_ptr = 0;
        i2c->msg = NULL;
        i2c->msg_idx ++;
        i2c->msg_num = 0;        (A)
        if (ret)
                i2c->msg_idx = ret;     (B)
        if (!i2c->use_pio)
                wake_up(&i2c->wait);    (C)
}
```

(A)
设置醒来的条件，i2c_msg传输完了
(B)
把传输结果通过i2c->msg_idx带回来。如果successful，则就是传输完成的i2c_msg number；
否则就是出错状态
(C)
唤醒等待在wait queue上的thread
③

```
static inline void i2c_pxa_stop_message(struct pxa_i2c *i2c)
{
        u32 icr;

        /*
         * Clear the STOP and ACK flags
         */
        icr = readl(_ICR(i2c));
        icr &= ~(ICR_STOP | ICR_ACKNAK);
        writel(icr, _ICR(i2c));
}
```

有点奇怪，函数名字叫"stop_message"，但实际上并不发送STOP signal。
该function只是把i2c control register的ICR_STOP bit clear。由于并没有设置ICR_TB，
所以并不真正启动hardware transfer。所以该function并没有时序。
那么STOP signal到底在哪儿发送呢?
一种可能是在interrupt handler里面在必要的时候(比如在transfer最后一个message的最后一个byte时)
已经发出了STOP signal.

```
1.  static irqreturn_t i2c_pxa_handler(int this_irq, void *dev_id)
2.  {
3.          struct pxa_i2c *i2c = dev_id;
4.          u32 isr = readl(_ISR(i2c));
5.
6.          if (!(isr & VALID_INT_SOURCE))
7.                  return IRQ_NONE;
8.
9.          if (i2c_debug > 2 && 0) {
10.                 dev_dbg(&i2c->adap.dev, "%s: ISR=%08x, ICR=%08x, IBMR=%02x\n",
11.                         __func__, isr, readl(_ICR(i2c)), readl(_IBMR(i2c)));
12.                 decode_ISR(isr);
13.         }
14.
15.         if (i2c->irqlogidx < ARRAY_SIZE(i2c->isrlog))
16.                 i2c->isrlog[i2c->irqlogidx++] = isr;
17.
18.         show_state(i2c);
19.
20.         /*
21.          * Always clear all pending IRQs.
22.          */
23.         writel(isr & VALID_INT_SOURCE, _ISR(i2c));
24.
25.         if (isr & ISR_SAD)
26.                 i2c_pxa_slave_start(i2c, isr);
27.         if (isr & ISR_SSD)
28.                 i2c_pxa_slave_stop(i2c);
29.
30.         if (i2c_pxa_is_slavemode(i2c)) {
31.                 if (isr & ISR_ITE)
32.                         i2c_pxa_slave_txempty(i2c, isr);
33.                 if (isr & ISR_IRF)
34.                         i2c_pxa_slave_rxfull(i2c, isr);
35.         } else if (i2c->msg && (!i2c->highmode_enter)) {
36.
37.                         i2c_pxa_irq_txempty(i2c, isr);
38.
39.                         i2c_pxa_irq_rxfull(i2c, isr);
40.         } else if ((isr & ISR_ITE) && i2c->highmode_enter) {
41.                 i2c->highmode_enter = false;
42.                 wake_up(&i2c->wait);
43.         } else {
44.                 i2c_pxa_scream_blue_murder(i2c, "spurious irq");
45.         }
46.
47.         return IRQ_HANDLED;
48. }
```

正常情况下就是标红的code在处理。
当一个byte被发送出去(write　operation)后，则调用i2c_pxa_irq_txempty()
当接收到一个byte(read operation)后，则调用i2c_pxa_irq_rxfull()

i2c_msg[] mesage之间的切换(一个message send / receieve 完了以后要切换到下一个message)

也应该是这两个函数中处理的。

发送，也就是write operation

```
static void i2c_pxa_irq_txempty(struct pxa_i2c *i2c, u32 isr)
{
        u32 icr = readl(_ICR(i2c)) & ~(ICR_START|ICR_STOP|ICR_ACKNAK|ICR_TB);

 again:
        /*
         * If ISR_ALD is set, we lost arbitration.
         */
        if (isr & ISR_ALD) {
                /*
                 * Do we need to do anything here?  The PXA docs
                 * are vague about what happens.
                 */
                i2c_pxa_scream_blue_murder(i2c, "ALD set");

                /*
                 * We ignore this error.  We seem to see spurious ALDs
                 * for seemingly no reason.  If we handle them as I think
                 * they should, we end up causing an I2C error, which
                 * is painful for some systems.
                 */
                return; /* ignore */
        }

        if (isr & ISR_BED) {
                int ret = BUS_ERROR;

                /*
                 * I2C bus error - either the device NAK'd us, or
                 * something more serious happened.  If we were NAK'd
                 * on the initial address phase, we can retry.
                 */
                if (isr & ISR_ACKNAK) {
                        if (i2c->msg_ptr == 0 && i2c->msg_idx == 0)
                                ret = I2C_RETRY;
                        else
                                ret = XFER_NAKED;
                }
                i2c_pxa_master_complete(i2c, ret);
        } else if (isr & ISR_RWM) {        (A)
                /*
                 * Read mode.  We have just sent the address byte, and
                 * now we must initiate the transfer.
                 */
                if (i2c->msg_ptr == i2c->msg->len - 1 &&
                    i2c->msg_idx == i2c->msg_num - 1)
                        icr |= ICR_STOP | ICR_ACKNAK;

                icr |= ICR_ALDIE | ICR_TB;
        } else if (i2c->msg_ptr < i2c->msg->len) {    (B)
                /*
                 * Write mode.  Write the next data byte.
                 */
                writel(i2c->msg->buf[i2c->msg_ptr++], _IDBR(i2c));

                icr |= ICR_ALDIE | ICR_TB;
```

```
57.
58.                 /*
59.                  * If this is the last byte of the last message, send
60.                  * a STOP.
61.                  */
62.                 if (i2c->msg_ptr == i2c->msg->len &&
63.                     i2c->msg_idx == i2c->msg_num - 1)
64.                         icr |= ICR_STOP;
65.         } else if (i2c->msg_idx < i2c->msg_num - 1) {    (C)
66.                 /*
67.                  * Next segment of the message.
68.                  */
69.                 i2c->msg_ptr = 0;
70.                 i2c->msg_idx ++;
71.                 i2c->msg++;
72.
73.                 /*
74.                  * If we aren't doing a repeated start and address,
75.                  * go back and try to send the next byte.  Note that
76.                  * we do not support switching the R/W direction here.
77.                  */
78.                 if (i2c->msg->flags & I2C_M_NOSTART)
79.                         goto again;
80.
81.                 /*
82.                  * Write the next address.
83.                  */
84.                 writel(i2c_pxa_addr_byte(i2c->msg), _IDBR(i2c));
85.
86.                 /*
87.                  * And trigger a repeated start, and send the byte.
88.                  */
89.                 icr &= ~ICR_ALDIE;
90.                 icr |= ICR_START | ICR_TB;
91.         } else {                        (D)
92.                 if (i2c->msg->len == 0) {
93.                         /*
94.                          * Device probes have a message length of zero
95.                          * and need the bus to be reset before it can
96.                          * be used again.
97.                          */
98.                         i2c_pxa_reset(i2c);
99.                 }
100.                i2c_pxa_master_complete(i2c, 0);
101.        }
102.
103.        i2c->icrlog[i2c->irqlogidx-1] = icr;
104.
105.        writel(icr, _ICR(i2c));
106.        show_state(i2c);
107. }
```

(A)(B)(C)(D)是正常处理分支。

(A) branch

```
                /*
                 * Read mode.  We have just sent the address byte, and
                 * now we must initiate the transfer.
                 */
                if (i2c->msg_ptr == i2c->msg->len - 1 &&
                    i2c->msg_idx == i2c->msg_num - 1)
                        icr |= ICR_STOP | ICR_ACKNAK;

                icr |= ICR_ALDIE | ICR_TB;
```

在Programmer Guide对ISR_RWM bit的描述如下
Read/write Mode
0 = The TWSI is in master-transmit or slave-receive mode.
1 = The TWSI is in master-receive or slave-transmit mode.
This is the R/nW bit of the slave address. It is cleared
automatically by hardware after a Stop state.

指示当前是在read还是write。

```
                if (i2c->msg_ptr == i2c->msg->len - 1 &&
                    i2c->msg_idx == i2c->msg_num - 1)
                        icr |= ICR_STOP | ICR_ACKNAK;
```

上面的判断是check是否当前是最后一个message的最后一个byte,如果是，则要发送STOP signal。
这里ICR_ACKNAK的作用?
The positive/negative acknowledge control bit
Defines the type of acknowledge pulse sent by the TWSI
when in master receive mode: 0 = Send a positive
acknowledge (ACK) pulse after receiving a data byte. 1 =
Send a negative acknowledge (NAK) pulse after receiving a
data byte.

本分支就是在master receive mode。ISR_RWM为１，就表示in master-receive mode。
这里ICR_ACKNAK置１，表示要发送NAK signal。

```
icr |= ICR_ALDIE | ICR_TB;
```

启动硬件传输。
i2c controller怎么知道是什么mode呢？read or write ?
发送START signal必然伴随着的是发送(write)i2c slave device address,而在设置address时
必须指定本次START signal开启的transfer是read还是write，自然i2c controller也就知道了。

```
static inline void i2c_pxa_start_message(struct pxa_i2c *i2c)
{
        u32 icr;

        /*
         * Step 1: target slave address into IDBR
         */
        writel(i2c_pxa_addr_byte(i2c->msg), _IDBR(i2c));

        /*
         * Step 2: initiate the write.
         */
        icr = readl(_ICR(i2c)) & ~(ICR_STOP | ICR_ALDIE);
        writel(icr | ICR_START | ICR_TB, _ICR(i2c));
}
```

```
static inline unsigned int i2c_pxa_addr_byte(struct i2c_msg *msg)
{
        unsigned int addr = (msg->addr & 0x7f) << 1;

if (msg->flags & I2C_M_RD)
                addr |= 1;

        return addr;
}
```

传输的 8 位的bit 1 to bit 7的7 bit是i2c slave device的address，而bit 0就告诉i2c controller后面的传输是read还是write!也就是处于什么mode.

(B) branch

```
        } else if (i2c->msg_ptr < i2c->msg->len) {
                /*
                 * Write mode.  Write the next data byte.
                 */
                writel(i2c->msg->buf[i2c->msg_ptr++], _IDBR(i2c));    ①

                icr |= ICR_ALDIE | ICR_TB;        ②

                /*
                 * If this is the last byte of the last message, send
                 * a STOP.
                 */
                if (i2c->msg_ptr == i2c->msg->len &&        ③
                    i2c->msg_idx == i2c->msg_num - 1)
                        icr |= ICR_STOP;
```

本分支是Write mode (ISR_RWM为 0)

i2c->msg_ptr < i2c->msg->len

表示当前i2c_msg还没有send完。

①

把当前message中下一个byte填入data register

②
设置transfer control bit
③
如果这是最后一个message的最后一个要发送的byte，则显然还要发送STOP signal

(C) branch

```
        } else if (i2c->msg_idx < i2c->msg_num - 1) {
                /*
                 * Next segment of the message.
                 */
                i2c->msg_ptr = 0;     ①
                i2c->msg_idx ++;      ②
                i2c->msg++;           ③

                /*
                 * If we aren't doing a repeated start and address,
                 * go back and try to send the next byte.  Note that
                 * we do not support switching the R/W direction here.
                 */
                if (i2c->msg->flags & I2C_M_NOSTART)    ④
                        goto again;

                /*
                 * Write the next address.
                 */
                writel(i2c_pxa_addr_byte(i2c->msg), _IDBR(i2c));    ⑤

                /*
                 * And trigger a repeated start, and send the byte.
                 */
                icr &= ~ICR_ALDIE;            ⑥
                icr |= ICR_START | ICR_TB;
```

由于(B) branch的条件是i2c->msg_ptr < i2c->msg->len，那么运行到(C) branch的条件实际上是两个。
1. i2c->msg_ptr == i2c->msg->len
2. i2c->msg_idx < i2c->msg_num - 1
也就是当前i2c_msg已经处理完了，同时下面还有i2c_msg要处理(即当前message不是最后一个message)。

①②③
指向下一个message
④
如果即将处理的message设置了`I2C_M_NOSTART`，即在发送当前`message`时，不要发送`START signal`。跳转到`again`，实际上是进入`(B)branch`，即开始发送`data`。
⑤
如果在新message中并没有设置`I2C_M_NOSTART`，那么就要完全重新开始启动一次新的`data`传输那样，如下图

Start Condition

先发送i2c device address及START signal

⑥

ICR_TB启动硬件transfer

(D) branch

进入该分支的条件如下：

1. i2c->msg_ptr == i2c->msg->len

2. i2c->msg_idx = i2c->msg_num - 1

最后一个message的最后一byte

```
1.  if (i2c->msg->len == 0) {    ①
2.                          /*
3.                           * Device probes have a message length of zero
4.                           * and need the bus to be reset before it can
5.                           * be used again.
6.                           */
7.                          i2c_pxa_reset(i2c);        ②
8.                  }
9.                  i2c_pxa_master_complete(i2c, 0);    ③
```

①②

如果最后一个message中传输的data长度为0，这里要reset i2c，why?

③

结束所有message data的transfer，返回到i2c_pxa_do_xfer()中

in i2c_pxa_do_xfer()

```
1.          /*
2.           * The rest of the processing occurs in the interrupt handler.
3.           */
4.          timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);
5.          i2c_pxa_stop_message(i2c);
6.
7.          /*
8.           * We place the return code in i2c->msg_idx.
9.           */
10.         ret = i2c->msg_idx;
```

i2c->msg_idx会作为i2c_pxa_do_xfer()返回值返回。

```
/*
 * i2c_pxa_master_complete - complete the message and wake up.
 */
static void i2c_pxa_master_complete(struct pxa_i2c *i2c, int ret)
{
        i2c->msg_ptr = 0;
        i2c->msg = NULL;
        i2c->msg_idx ++;
        i2c->msg_num = 0;
        if (ret)
                i2c->msg_idx = ret;
        if (!i2c->use_pio)
                wake_up(&i2c->wait);
}
```

在正常transfer的情况下，都是i2c_pxa_master_complete(i2c, 0);

即`i2c->msg_idx是i2c_pxa_do_xfer()处理的i2c_msg个数。如果 < 0，则带回的是出错状态，比如i2c_pxa_master_complete(i2c, I2C_RETRY);`

```
        if (isr & ISR_BED) {
                int ret = BUS_ERROR;

                /*
                 * I2C bus error - either the device NAK'd us, or
                 * something more serious happened.  If we were NAK'd
                 * on the initial address phase, we can retry.
                 */
                if (isr & ISR_ACKNAK) {
                        if (i2c->msg_ptr == 0 && i2c->msg_idx == 0)
                                ret = I2C_RETRY;
                        else
                                ret = XFER_NAKED;
                }
                i2c_pxa_master_complete(i2c, ret);
```

下面基于上面的code来分析write多个message的情况。

应用一场景分析：

```
        int                             ret;
        struct i2c_adapter      *adap = adapter;
        struct i2c_msg          msg[3];

        msg0.addr = addr;
        msg0.flags = 0;
        msg0.len = sndcount0;
        msg0.buf = (char *)sndbuf0;

        msg1.addr = addr;
        msg1.flags = I2C_M_NOSTART;
        msg1.len = sndcount1;
        msg1.buf = (char *)sndbuf1;

        msg2.addr = addr;
        msg2.flags = I2C_M_NOSTART;
        msg2.len = sndcount2;
        msg2.buf = (char *)sndbuf2;

        ret = i2c_transfer(adap, msg, 3);
```

这种情况下，相当与把msg0,msg1和msg3中的 3 个buffer合并成一个大buffer，用一个i2c_msg来指向这个大buffer进行传输。

msg1和msg2设置

了I2C_M_NOSTART，这样其实msg1和msg2中的地址完全被忽略了。只有在传输msg0开始前才会发送START signal，并且在msg2的最后一个byte后发送STOP signal。

Note: msg0, msg1和msg2中的addr必须相同，否则就是错的！

应用二场景分析：

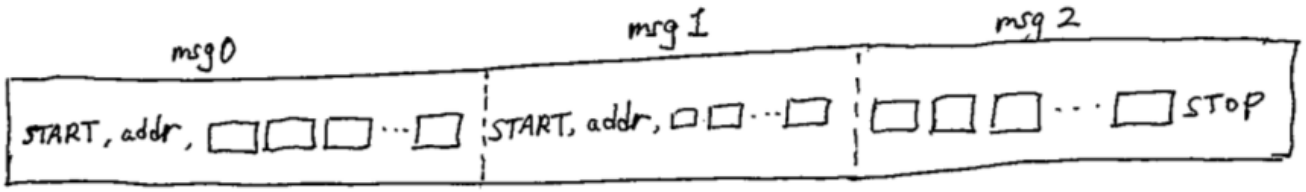
```
        int                             ret;
        struct i2c_adapter      *adap = adapter;
        struct i2c_msg          msg[3];

        msg0.addr = addr;
        msg0.flags = 0;
        msg0.len = sndcount0;
        msg0.buf = (char *)sndbuf0;

        msg1.addr = addr;
        msg1.flags = 0;
        msg1.len = sndcount1;
        msg1.buf = (char *)sndbuf1;

        msg2.addr = addr;
        msg2.flags = I2C_M_NOSTART;
        msg2.len = sndcount2;
        msg2.buf = (char *)sndbuf2;

        ret = i2c_transfer(adap, msg, 3);
```



msg2的addr完全是无用的，msg1和msg2可以合并成一个message。

应用三场景分析：

```
        int                             ret;
        struct i2c_adapter      *adap = adapter;
        struct i2c_msg          msg[3];

        msg0.addr = addr
        msg0.flags = 0;
        msg0.len = sndcount0;
        msg0.buf = (char *)sndbuf0;

        msg1.addr = addr2;
        msg1.flags = 0;
        msg1.len = sndcount1;
        msg1.buf = (char *)sndbuf1;

        msg2.addr = addr3;
        msg2.flags = I2C_M_NOSTART;
        msg2.len = sndcount2;
        msg2.buf = (char *)sndbuf2;

        ret = i2c_transfer(adap, msg, 3);
```

msg2的设置是错误的，因为msg2的flags设置了I2C_M_NOSTART，那么它的addr就不能与前面的message(msg1)的地址不同。
实际上本来想发往addr3的sndbuf3中的data发送给了addr2 i2c device!

应用四场景分析：

```
        int                             ret;
        struct i2c_adapter      *adap = adapter;
        struct i2c_msg          msg[3];

        msg0.addr = addr
        msg0.flags = I2C_M_NOSTART;
        msg0.len = sndcount0;
        msg0.buf = (char *)sndbuf0;

        msg1.addr = addr2;
        msg1.flags = I2C_M_NOSTART;
        msg1.len = sndcount1;
        msg1.buf = (char *)sndbuf1;

        msg2.addr = addr2;
        msg2.flags = I2C_M_NOSTART;
        msg2.len = sndcount2;
        msg2.buf = (char *)sndbuf2;

        ret = i2c_transfer(adap, msg, 3);
```

这里的msg0的I2C_M_NOSTART是没必要的。msg0中的START signal与addr是必发的。
另外msg1中设置了I2C_M_NOSTART及addr2不同于msg0中的addr，这是自相矛盾的！
设置了I2C_M_NOSTART，表示msg1的data其实是发往钱前一个msg0的地址的；而msg1的地址addr2不同与msg0的地址addr，就表示不应该设置I2C_M_NOSTART，因为发送START signal和addr2对msg1来说是必须的！

接收，也就是read operation

```
1.  static void i2c_pxa_irq_rxfull(struct pxa_i2c *i2c, u32 isr)
2.  {
3.          u32 icr = readl(_ICR(i2c)) & ~(ICR_START|ICR_STOP|ICR_ACKNAK|ICR_TB);
4.
5.          /*
6.           * Read the byte.
7.           */
8.          i2c->msg->buf[i2c->msg_ptr++] = readl(_IDBR(i2c));    ①
9.
10.         if (i2c->msg_ptr < i2c->msg->len) {    ②
11.                 /*
12.                  * If this is the last byte of the last
13.                  * message, send a STOP.
14.                  */
15.                 if (i2c->msg_ptr == i2c->msg->len - 1)    ③
16.                         icr |= ICR_STOP | ICR_ACKNAK;
17.
18.                 icr |= ICR_ALDIE | ICR_TB;    ④
19.         } else {
20.                 i2c_pxa_master_complete(i2c, 0);    ⑤
21.         }
22.
23.         i2c->icrlog[i2c->irqlogidx-1] = icr;
24.
25.         writel(icr, _ICR(i2c));
26. }
```

①
从data register读取接收到的byte,i2c->msg_ptr总是指向下一个空闲slot
②
i2c->msg_ptr < i2c->msg->len
还未接收满指定的bytes
③
i2c->msg_ptr == i2c->msg->len - 1
这是要接收指定的最后一个byte,那样需要设置发送STOP signal,同时也要发送NAK给i2c slave device。在非最后一个byte情况下，ICR_ACKNAK未置位，则i2c master在接收到一个byte后发送ACK signal给i2c slave device.
④
启动hardware transfer
⑤
i2c->msg_ptr == i2c->msg->len
即指定的byte已经接收满了，回到i2c_pxa_do_xfer()。

由于i2c_pxa_irq_rxfull()并不处理多message的情况，所以i2c read operation必须是i2c-pxa master处理的最后一个message!这也就是redmine task #3256的issue.

gr2 I2C driver does not have read followed by write repeated start support

We currently have a customer that uses I2C to communicate between the 6270 and a separate custom board using both write followed by read and read followed by write repeated start sequences.

目前的i2c-pxa driver对后者(read followed by write)是支持的，但对前者(write followed by read)是不支持的.

假设场景如下：

```
        int                             ret;
        struct i2c_adapter      *adap = adapter;
        struct i2c_msg          msg[3];

        msg0.addr = addr
        msg0.flags = 0;
        msg0.len = sndcount0;
        msg0.buf = (char *)sndbuf0;

        msg1.addr = addr;
        msg1.flags |= |= I2C_M_RD;
        msg1.len = recvcount1;
        msg1.buf = (char *)recvbuf1;

        msg2.addr = addr;
        msg2.flags = 0;
        msg2.len = sndcount2;
        msg2.buf = (char *)sndbuf2;

        ret = i2c_transfer(adap, msg, 3);
```

i2c_transfer(adap, msg, 3) --> i2c_pxa_do_xfer(adap, msg, 3)

```
        i2c->msg = msg;
        i2c->msg_num = num;
        i2c->msg_idx = 0;
        i2c->msg_ptr = 0;
        i2c->irqlogidx = 0;

        i2c_pxa_start_message(i2c);     ①

        spin_unlock_irq(&i2c->lock);

        /*
         * The rest of the processing occurs in the interrupt handler.
         */
        timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);     ②
        i2c_pxa_stop_message(i2c);

        /*
         * We place the return code in i2c->msg_idx.
         */
        ret = i2c->msg_idx;     ③
```

①
启动transfer，首先write msg0
in i2c_pxa_irq_txempty()

```
        } else if (i2c->msg_idx < i2c->msg_num - 1) {
                /*
                 * Next segment of the message.
                 */
                i2c->msg_ptr = 0;
                i2c->msg_idx ++;
                i2c->msg++;

                /*
                 * If we aren't doing a repeated start and address,
                 * go back and try to send the next byte.  Note that
                 * we do not support switching the R/W direction here.
                 */
                if (i2c->msg->flags & I2C_M_NOSTART)
                        goto again;

                /*
                 * Write the next address.
                 */
                writel(i2c_pxa_addr_byte(i2c->msg), _IDBR(i2c));

                /*
                 * And trigger a repeated start, and send the byte.
                 */
                icr &= ~ICR_ALDIE;
                icr |= ICR_START | ICR_TB;
        } else {
```

write msg0到最后一个byte时，运行到上面的code。

i2c->msg_ptr = 0;

i2c->msg_idx ++;

i2c->msg++;

指向msg1

writel(i2c_pxa_addr_byte(i2c->msg), _IDBR(i2c));

```
static inline unsigned int i2c_pxa_addr_byte(struct i2c_msg *msg)
{
        unsigned int addr = (msg->addr & 0x7f) << 1;

        if (msg->flags & I2C_M_RD)
                addr |= 1;

        return addr;
}
```

由于msg1设置了I2C_M_RD,所以发送的是read operation的address。

/*

 * And trigger a repeated start, and send the byte.

 */

icr &= ~ICR_ALDIE;

icr |= ICR_START | ICR_TB;

把i2c slave device address发送出去。

接下来收到发送完成的中断，即ISR_ITE置位(tx buffer empty,把address已经发送出去了，data regsiter空了)
再次进入i2c_pxa_irq_txempty()的如下branch

```
        } else if (isr & ISR_RWM) {
                /*
                 * Read mode.  We have just sent the address byte, and
                 * now we must initiate the transfer.
                 */
                if (i2c->msg_ptr == i2c->msg->len - 1 &&
                    i2c->msg_idx == i2c->msg_num - 1)
                        icr |= ICR_STOP | ICR_ACKNAK;

                icr |= ICR_ALDIE | ICR_TB;
```

这是在read mode，所以只是令ICR_TB置位，即启动hardware开始接收slave device发来的data.
当data到来后，产生ISR_IRF　interrupt (rx buffer full)，进入i2c_pxa_irq_rxfull()。
在msg1中接收到该message的最后一个byte后会向i2c slave device发送STOP signal and NAK signal。
同时通过
i2c_pxa_master_complete(i2c, 0);
来返回到i2c_pxa_do_xfer()中。

```
static void i2c_pxa_master_complete(struct pxa_i2c *i2c, int ret)
{
        i2c->msg_ptr = 0;
        i2c->msg = NULL;
        i2c->msg_idx ++;
        i2c->msg_num = 0;
        if (ret)
                i2c->msg_idx = ret;
        if (!i2c->use_pio)
                wake_up(&i2c->wait);
}
```

这时i2c->msg_idx在++以后是2，即成功完成了2个message的传输，msg0 and msg1。
②
返回到i2c_pxa_do_xfer()的

```
timeout = wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5);
```

由于i2c->msg_num已经被赋值为 0 ,所以wait的条件满足，即调用i2c_pxa_do_xfer()的thread成功resume(不是由于timeout)!
③
i2c_pxa_do_xfer()返回值是 2 ,表示成功传输了 2 个message，msg0 and msg1。而msg2则被完全遗忘了！

所以在上面假设的场景中，
ret = i2c_transfer(adap, msg, 3);
返回值ret = 2。在msg1后面的所有message(无论是read operation还是write operation message)都丢失

了！

Notes:
static int i2c_pxa_do_xfer(struct pxa_i2c *i2c, struct i2c_msg *msg, int num)
The function可以传输多个message，但如果有read operation message，则只能有一个read operation message，并且这个read operation message必须是最后一个message!

write-operation-message + ... (多个write-operation-message) + read-operation-message是支持的
read-operation-message + ... (read or write operation message)是不支持的！