声明

i2c 源代码情景分析（Beta2）

作者在 www.linuxforum.net 上的 ID 为 shrek2

欢迎补充，欢迎批评指正！

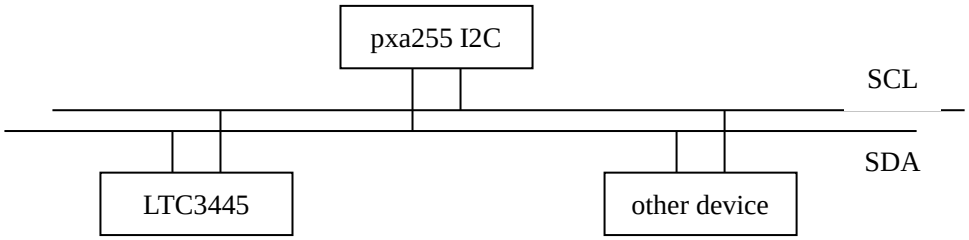
第 1 章 i2c 核心数据结构之间的关系 4
第 2 章 i2c-core.c 的初始化 10
i2c_init 函数 10
i2cproc_init 函数 11
第 3 章 安装、卸载 pxa255 的 i2c 适配器驱动程序 13
pxa_i2c 数据结构 13
i2c_adap_pxa_init 函数 15
i2c_add_adapter 函数 16
i2c_del_adapter 函数 18
i2c_adap_pxa_exit 函数 20
第 4 章 安装、卸载 ltc3445 驱动程序 21
ltc3445_init 函数 21
i2c_add_driver 函数 22
i2c_probe 函数 23
i2c_check_functionality 函数 26
i2c_smbus_xfer 函数 27
i2c_transfer 函数 29
ltc3445_detect_client 函数 30
i2c_attach_client 函数 31
ltc3445_cleanup 函数 32
i2c_del_driver 函数 33
ltc3445_detach_client 函数 35
i2c_detach_client 函数 35
第 5 章 与 pxa255 的 i2c 适配器相关的代码 37
i2c_pxa_reset 函数 37
i2c_pxa_abort 函数 38

i2c_pxa_xfer 函数.....38
i2c_pxa_do_xfer 函数.....39
i2c_pxa_start_message 函数.....41
i2c_pxa_handler 函数.....42
i2c_pxa_irq_txempty 函数.....42
i2c_pxa_irq_rxfull 函数.....47
第 6 章 i2c-dev 的初始化.....49
i2c_dev_init 函数.....49
i2cdev_attach_adapter 函数.....50
第 7 章 i2c 框架提供的设备访问方法.....52
i2cdev_open 函数.....52
i2cdev_ioctl 函数.....53
i2cdev_read 函数.....54
i2c_master_recv 函数.....55
对 i2cdev_read 和 i2c_master_recv 的修改.....56
i2cdev_release 函数.....59
第 8 章 编写 i2c 设备驱动程序模块的方法.....61
第 9 章 用户进程访问 i2c 设备的步骤.....64
讨论和总结.....65
i2c 操作中的同步问题.....65
总结各个模块初始化函数的作用.....65
对 i2c 框架代码的修改.....66
有关 i2c 设备私有数据结构的讨论.....68
遗留的问题.....68

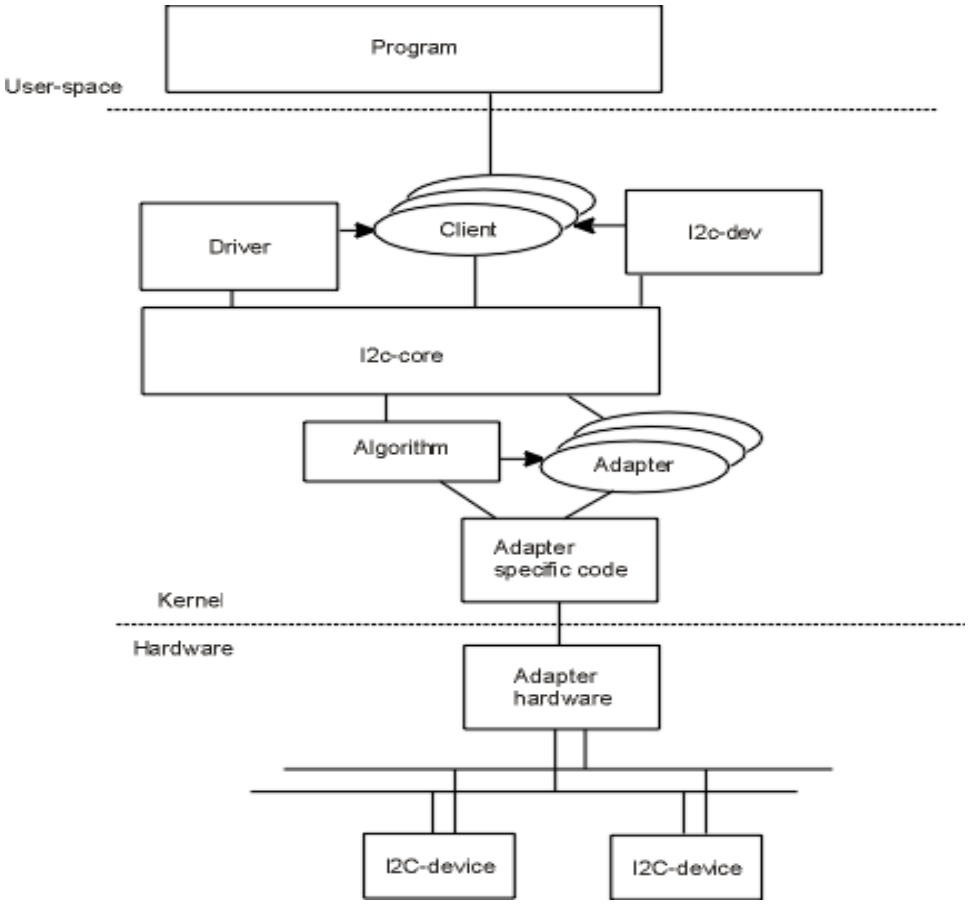
（注意：本文档中的源代码以 i2c-2.9.1 包及 www.arm.linux.org.uk 上下载的 pxa 的 i2c 适配器的补丁 2360-2 为准！）

第 1 章 i2c 核心数据结构之间的关系

i2c 总线适配器（adapter）就是一条 i2c 总线的控制器，在物理连接上若干 i2c 设备并联于该 i2c 总线的 SCL 和 SDA 线上，如下图所示：



那么相应软件数据结构的设计、数据结构之间的关系就至少应该描述硬件物理连接的这种组织关系。Linux 的 i2c 框架中各个部分的关系如下图所示：



内核中 i2c 相关代码可以分为三个层次：

1. i2c 框架：i2c.h 和 i2c-core.c 为 i2c 框架的主体，提供了核心数据结构的定义、i2c 适配器驱动和设备驱动的注册、注销管理，i2c 通信方法上层的、与具体适配器无关的代码、检测设备地址的上层代码等；i2c-dev.c 用于创建 i2c 适配器的/dev/i2c/%d 设备节点，提供 i2c 设备访问方法等。
2. i2c 总线适配器驱动：定义描述具体 i2c 总线适配器的 i2c_adapter 数据结构、实现在具体 i2c 适配器上的 i2c 总线通信方法，并由 i2c_algorithm 数据结构进行描述。
3. i2c 设备驱动：定义描述具体设备的 i2c_client 和可能的私有数据结构、借助 i2c 框架的 i2c_probe 函数实现注册设备的 attach_adapter 方法、提供设备可能使用的地址范围、以及设备地址检测成功后创建 i2c_client 数据结构的回调函数。

下面介绍 i2c 各核心数据结构的定义和它们之间的连接关系。

1. 一个 i2c 设备的驱动程序由 i2c_driver 数据结构描述，定义于 include/linux/i2c.h：

```
struct i2c_driver {
    char name[32];
    int id;
    unsigned int flags;
    int (*attach_adapter)(struct i2c_adapter *);
    int (*detach_client)(struct i2c_client *);
    int (*command)(struct i2c_client *client,unsigned int cmd, void *arg);
    void (*inc_use)(struct i2c_client *client);
    void (*dec_use)(struct i2c_client *client);
};
```

其中 name 为最大长度为 32 字节的字符串，id 可选 0xf000 到 0xffff 中的任一数值，flags 域可以直接设置为 I2C_DF_NOTIFY。attach_adapter 回调函数在安装 i2c 设备驱动程序模块时、或者在安装 i2c 适配器驱动程序模块时被调用，用于检测、认领设备并为设备分配 i2c_client 数据结构。detach_client 方法在卸载适配器或设备驱动程序模块时被调用，用于从总线上注销设备、并释放 i2c_client 及相应的私有数据结构。

inc_use 和 dec_use 所指向的函数用于改变 i2c 设备驱动程序模块的引用计数。注意不要直接调用 i2c_driver 数据结构中的这两个方法，而要通过如下函数调用路径：

```
i2c_use_client > i2c_inc_use_client > inc_use
i2c_release_client > i2c_dec_use_client > dec_use
```

通过最顶层的 i2c_use/release_client 函数来同时改变 i2c 设备和 i2c 适配器驱动程序模块的引用计数。另外，不能在 attach_adapter 函数检测到一个 i2c 设备时就增加驱动程序模块的引用计数，而应该在用户进程访问一个/dev/i2c/%d 设备节点时增加模块的引用计数，则关闭设备节点时减少引用计数（但在当前的应用中，适配器和设备的驱动程序都是静态地链接入内核映像的，所以在 pxa255 的 i2c 补丁中并没有使用控制引用计数的函数）。

2.　一个 i2c 设备由 i2c_client 数据结构进行描述：

```
struct i2c_client {
        char name[32];
        int id;
        unsigned int flags;             /* div., see below              */
        unsigned int addr;              /* chip address - NOTE: 7bit addresses are stored in the      */
                                        /* _LOWER_ 7 bits of this char     */
        struct i2c_adapter *adapter;    /* the adapter we sit on   */
        struct i2c_driver *driver;      /* and our access routines      */
        void *data;                     /* for the clients      */
        int usage_count;                /* How many accesses currently to the client*/
};
```

在安装适配器或者设备的驱动程序时通过设备驱动程序 i2c_driver 中的 attach_adapter 函数检测设备地址。如果检测成功则调用设备驱动程序提供的回调函数创建描述设备的 i2c_client 数据结构，并将其中的 driver 指针指向设备驱动程序的 i2c_driver 数据结构。这样将来就可以使用 i2c_driver 中的注销设备和控制引用计数的方法了。

由下文可见在描述 i2c 适配器的 i2c_adapter 数据结构中设计了指向该总线上所有 i2c 设备的 i2c_client 数据结构的指针数组 clients，而每个 i2c_client 又通过 adapter 指针回指 i2c_adapter。数据结构之间类似的组织关系在 Linux 内核中屡见不鲜，比如父子进程的 PCB 之间、父目录及子目录和子文件的 dentry 之间，等等。

每个 i2c 设备都有唯一的 7 位地址 addr。由于设备可能支持多个地址，所以在设备驱动程序模块中要指出需要检测的地址范围（由 i2c_client_address_data 二维数组指定），而设备实际使用的地址在检测成功并为之分配 i2c_client 数据结构时填入。以 i2c 设备 ltc3445 为例，硬件支持的地址为 1001011 或者 0101011，即 7 位地址的高 2 位由具体的布线方法决定（可以分别接到 VCC 或者 GND）。如果 ltc3445 驱动程序的开发者知道具体的布线方法，那么在驱动程序中就可以直接指定。否则可以指定地址检测范围为这两个地址，而在加载驱动程序模块时由软件进行地址检测。

需要说明的是，i2c 设备的 7 位地址是就当前 i2c 总线而言的，是“相对地址”。不同的 i2c 总线上的设备可以使用相同的 7 位地址，但是它们所在的 i2c 总线不同。所以在系统中一个 i2c 设备的“绝对地址”由二元组（i2c 适配器的 ID 和设备在该总线上的 7 位地址）表示。

i2c_client 数据结构为描述 i2c 设备的“模板”，而具体的 i2c 设备可能需要描述个性的私有数据。私有数据结构由 i2c_client 数据结构中的 data 域指向。设备驱动程序开发者可以设计合适的私有数据结构来描述硬件的特性。值得一提的是，目前在 Linux 内核中常用的表示与具体设备、对象等相关的私有数据结构的方法有两种，一种就是采用 void 类型的指针 data 来指向具体的私有数据结构，又比如 file 结构中的 private_data 域在设备驱动程序中往往被设置为指向具体的设备数据结构；第二种方法就是采用 union 域，比如 VFS 的 super_block、inode 数据结构。super_block 和 inode 数据结构本身集中描述了各种文件系统的共性，而具体文件系统的个性则放到 union 中进行描述，在挂载具体的文件系统时实例化为具体的 union 对象，比如 ext2_inode_union 或者 jffs2_inode_info。

有关设计私有数据的的讨论可参见本文末尾的讨论部分。

当不同进程访问同一 i2c 总线时，对 i2c 总线的互斥访问由 i2c_adapter 的 lock 信号量实现，系统调用执行流只有在获得该信号量期间才能调用 master_xfer，并且在阻塞期间不释放信号量（类似在读写正规文件期间必须持有 inode.i_sem，参见本文末尾的讨论部分）。而 usage_count 域为设备的使用引用计数，在 i2c_use_client 和 i2c_release_client 函数中控制 usage_count 域的值（但是当前 pxa255 的 i2c 补丁中并没有使用这两个函数，usage_count 的值自初始化后就一直为 0）。

3.　一个 i2c 适配器由 i2c_adapter 数据结构描述：

```
struct i2c_adapter {
    char name[32];
    unsigned int id;            /* == is algo->id | hwdep.struct->id, for registered values see below    */
    struct i2c_algorithm *algo; /* the algorithm to access the bus      */
    void *algo_data;
    void (*inc_use)(struct i2c_adapter *);
    void (*dec_use)(struct i2c_adapter *);
    int (*client_register)(struct i2c_client *);
    int (*client_unregister)(struct i2c_client *);
    void *data;                 /* private data for the adapter         */
    struct semaphore lock;
    unsigned int flags;         /* flags specifying div. data       */
    struct i2c_client *clients[I2C_CLIENT_MAX];
    int client_count;
    int timeout;
    int retries;
#ifdef CONFIG_PROC_FS
    /* No need to set this when you initialize the adapter              */
    int inode;
#if LINUX_VERSION_CODE < KERNEL_VERSION(2,1,29)
    struct proc_dir_entry *proc_entry;
#endif
#endif /* def CONFIG_PROC_FS */
};
```

在 i2c_adapter 数据结构中设计了 clients 指针数组，指向该总线上每个设备的 i2c_client 数据结构。由于一条 i2c 总线上最多只有 I2C_CLENT_MAX 个设备，所以可以使用静态数组（题外话，如果相关数据结构的个数是未知的，链表显然是更好的选择）。lock 信号量用于实现对 i2c 总线的互斥访问：在访问 i2c 总线上的任一设备期间当前进程必须首先获得该信号量，并且在阻塞等待 i2c 操作完成期间不释放。

一个 i2c 适配器上的 i2c 总线通信方法由其驱动程序提供的 i2c_algorithm 数据结构描述，由 algo 指针指向。i2c_algorithm 数据结构即为 i2c_adapter 数据结构与具体 i2c 适配器的总线通信方法的中间层，由下文可见正是这个中间层使得上层的 i2c 框架代码与与具体 i2c 适配器的总线通信方法无关，从而实现了 i2c 框架的可移植性和重用性。当安装具体 i2c 适配器的驱动程序时由相应驱动程序实现具体的 i2c_algorithm 数据结构，其中的函数指针指向操作具体 i2c 适配器的代码（换用面向对象的语言，就是当创建子类对象时将基

类中定义的函数调用接口实例化为与具体子类相关的代码。值得说明的是，在 Linux 内核层次中数据结构的设计大量地采用了面向对象的概念来实现框架的可移植性和重用性）。

inc_use 和 dec_use 方法可以用来控制适配器驱动程序的引用计数；client_register 和 client_unregister 函数可以用来完成适配器端的、额外的设备注册和注销工作。这些函数在当前 pxa255 的 i2c 补丁中都没有实现。最后 timeout 和 retries 用于超时重传机制。

4. 具体 i2c 适配器的通信方法由 i2c_algorithm 数据结构进行描述：

```
struct i2c_algorithm {
    char name[32];                  /* textual description     */
    unsigned int id;
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg msgs[], int num);
    int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,
                       unsigned short flags, char read_write,
                       u8 command, int size, union i2c_smbus_data * data);
    int (*slave_send)(struct i2c_adapter *,char*,int);
    int (*slave_recv)(struct i2c_adapter *,char*,int);
    int (*algo_control)(struct i2c_adapter *, unsigned int, unsigned long);
    u32 (*functionality) (struct i2c_adapter *);
};
```

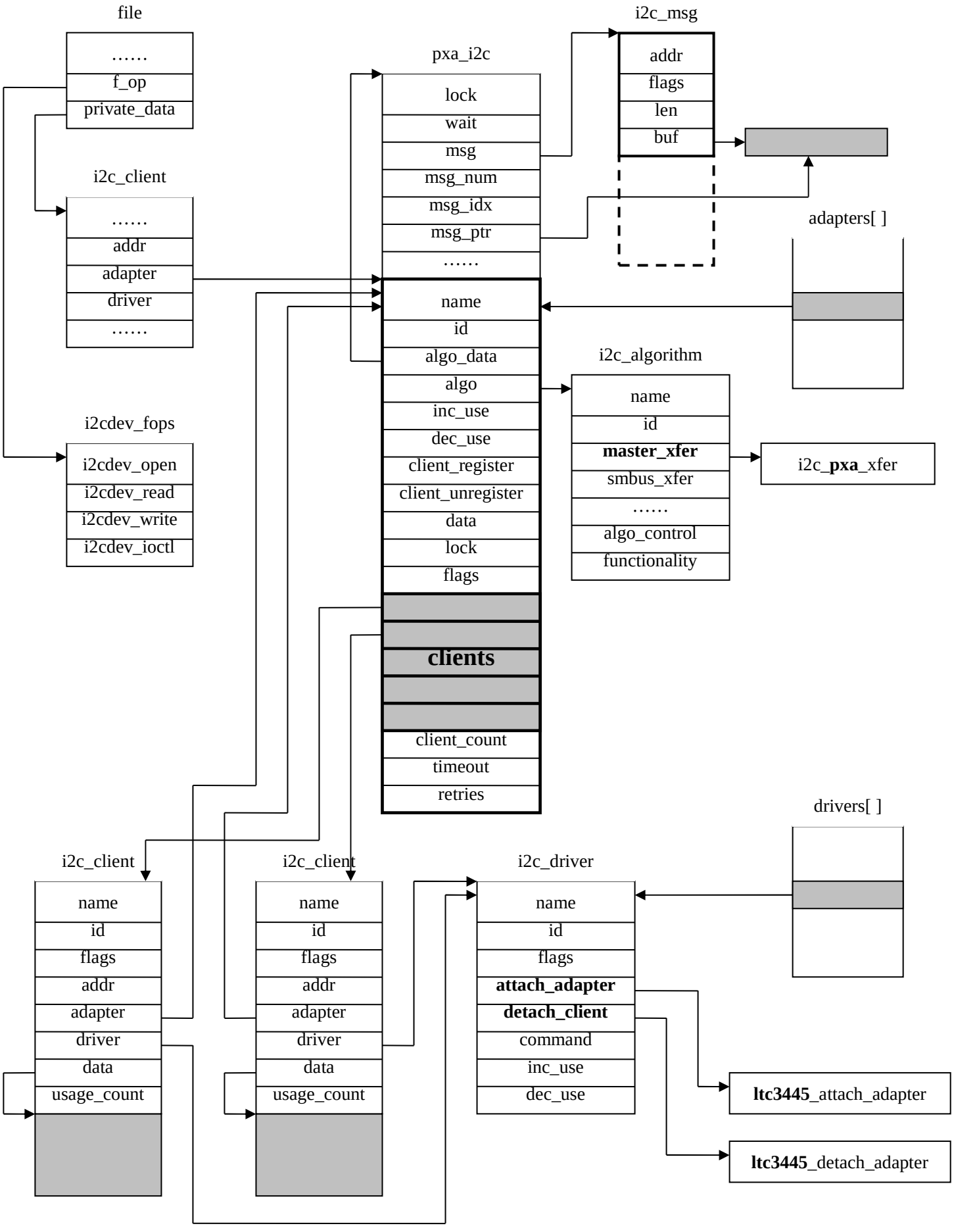
master_xfer/smbus_xfer 指针指向 i2c 适配器驱动程序模块实现的 i2c 通信协议或者 smbus 通信协议。由下文分析可见在用户进程通过 i2c-dev 提供的/dev/i2c/%d 设备节点访问 i2c 设备时，最终是通过调用 master_xfer 或者 smbus_xfer 指向的方法完成的。

slave_send/recv 函数用于实现当 i2c 适配器扮演 slave 角色时的传输方法。由于在 pxa255 的现有应用中其 i2c 适配器始终为主导 i2c 通信的 master，故补丁中这两个函数都没有实现。

i2c_algorithm 提供了 i2c 适配器的驱动，而 i2c 设备的驱动为 i2c_driver。内核中静态指针数组 adapters 和 drivers 分别记录所有已经注册的 i2c 总线设备和 i2c 设备驱动。从下文源代码分析可以看到，安装 i2c 总线驱动和 i2c 设备驱动的顺序不确定，因此在安装 i2c 设备驱动时必须遍历所有已注册的适配器上的 i2c 设备，以“认领”相应的设备；同理，在安装 i2c 适配器驱动时必须遍历所有已注册的 i2c 设备的驱动程序，让已有驱动程序“认领”新注册的适配器上的所有设备。

5. 假设一条 i2c 总线上有两个使用相同驱动程序的 i2c 设备，在打开该 i2c 总线的设备结点后相关数据结构之间的逻辑组织关系如下图所示。在阅读下文时请经常参照下图。

上层的 i2c 框架实现了控制策略，具体 i2c 适配器和设备的驱动实现了使具体设备可用的机制，上层策略和底层机制通过中间的函数调用接口联系。正是中间的函数调用接口使得上层策略与底层机制无关，从而使得上层策略具有良好的可移植性和重用性。阅读完全文后可以回过头来总结一下各数据结构的作用、创建时机、由谁创建等，品味这一点是如何通过这些数据结构实现的，进一步在自己的研发过程中积极实践这种思想并享受学以致用的乐趣：）


file
……
f_op
private_data
i2c_client
……
addr
adapter
driver
……
i2cdev_fops
i2cdev_open
i2cdev_read
i2cdev_write
i2cdev_ioctl
pxa_i2c
lock
wait
msg
msg_num
msg_idx
msg_ptr
……
name
id
algo_data
algo
inc_use
dec_use
client_register
client_unregister
data
lock
flags
clients
client_count
timeout
retries
i2c_msg
addr
flags
len
buf
adapters[ ]
i2c_algorithm
name
id
master_xfer
smbus_xfer
……
algo_control
functionality
i2c_pxa_xfer
drivers[ ]
i2c_client
name
id
flags
addr
adapter
driver
data
usage_count
i2c_client
name
id
flags
addr
adapter
driver
data
usage_count
i2c_driver
name
id
flags
attach_adapter
detach_client
command
inc_use
dec_use
ltc3445_attach_adapter
ltc3445_detach_adapter

第 2 章 i2c-core.c 的初始化

i2c-core.c 提供了 i2c 框架的主体代码，包括 i2c 适配器驱动和设备驱动的注册、注销管理的框架代码、i2c 通信方法的上层、与具体适配器无关的代码和检测设备地址的上层代码等。

i2c_init 函数

i2c-core 模块的初始化函数如下：

```
static int __init i2c_init(void)
{
    printk(KERN_DEBUG "i2c-core.o: i2c core module\n");
    memset(adapters,0,sizeof(adapters));
    memset(drivers,0,sizeof(drivers));
    adap_count=0;
    driver_count=0;

    init_MUTEX(&adap_lock);
    init_MUTEX(&driver_lock);
```

在 i2c-core.c 中定义了内核静态指针数组 adapters 和 drivers，用于注册描述 i2c 适配器及其驱动程序的 i2c_adapter 数据结构和描述设备及其驱动程序的 i2c_driver 数据结构。同时也定义了保护这两个全局指针数组的信号量：

```
static struct i2c_adapter *adapters[I2C_ADAP_MAX];
static struct i2c_driver *drivers[I2C_DRIVER_MAX];
struct semaphore adap_lock;
struct semaphore driver_lock;
```

其中表示适配器和驱动程序最大数量的宏在 linux/i2c.h 中均被定义为 16。由于安装 i2c 适配器驱动程序模块和 i2c 设备驱动程序模块的先后顺序不确定，所以必须通过全局数据结构（比如这里的指针数组或者链表等）来保存所有已安装的数据结构的地址，这样在后安装 i2c 适配器驱动程序时就可将 drivers 指针数组中记录的地址传递给 i2c_add_adapter 函数；在后安装 i2c 设备驱动程序时就可将 adapters 指针数组中记录的地址传递给 i2c_add_driver 函数（而这两个函数最终都是通过 i2c_driver.的 attach_adapter 函数“认领”设备的）。

这里首先初始化 adapters 和 drivers 指针数组及相关信号量 adap_lock 和 driver_lock，并将计数器清 0。

```
    i2cproc_init();
    return 0;
}
```

然后通过 i2cproc_init 函数创建相应的/proc/bus/i2c 文件，使得用户进程可以通过该文件得到当前系统上所有已注册的 i2c 总线信息。

i2cproc_init 函数

该函数用于创建描述系统中所有 i2c 总线的/proc/bus/i2c 文件。

```
int i2cproc_init(void)
{
    struct proc_dir_entry *proc_bus_i2c;

    i2cproc_initialized = 0;

    if (! proc_bus) {
        printk("i2c-core.o: /proc/bus/ does not exist");
        i2cproc_cleanup();
        return -ENOENT;
    }
```

在 i2c-core.c 文件中定义了表示 proc 接口初始化程度的静态变量 i2cproc_initialized，其初始值为 0。内核全局变量 proc_bus 定义于 fs/proc/root.c，声明于 include/linux/proc_fs.h，在 fs/proc/root.c 的 proc_root_init 函数中初始化：

```
struct proc_dir_entry *proc_net, *proc_bus, ……

void __init proc_root_init(void)
{
    int err = register_filesystem(&proc_fs_type);
    if (err)
        return;
    proc_mnt = kern_mount(&proc_fs_type);
    err = PTR_ERR(proc_mnt);
    if (IS_ERR(proc_mnt)) {
        unregister_filesystem(&proc_fs_type);
        return;
    }
    ……

    proc_bus = proc_mkdir("bus", 0);
}
```

可见在 proc_root_init 函数中注册、挂载 proc 文件系统，而 proc_bus 指向代表“/proc/bus”目录的 proc_dir_entry 类型的变量。在确保 proc_bus 不为 NULL 后，接下来通过 create_proc_entry 函数在/proc/bus 下创建文件

/proc/bus/i2c，相应的读回调函数为 read_bus_i2c。该函数很简单，就是遍历内核静态数组 adapters，向用户空间返回所有已注册的 i2c 适配器的信息。在此不再赘述，可参见内核文档 Documentation/i2c/proc-interface。函数的最后再增加 i2cproc_initialized 计数值。

```
    proc_bus_i2c = create_proc_entry("i2c",         /* name */
                                     0,             /* mode */
                                     proc_bus);     /* parent */
    if (!proc_bus_i2c) {
        printk("i2c-core.o: Could not create /proc/bus/i2c");
        i2cproc_cleanup();
        return -ENOENT;
    }
    proc_bus_i2c->read_proc = &read_bus_i2c;

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,27))
    proc_bus_i2c->owner = THIS_MODULE;
#else
    proc_bus_i2c->fill_inode = &monitor_bus_i2c;
#endif /* (LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,27)) */

    i2cproc_initialized += 2;
    return 0;
}
```

第 3 章 安装、卸载 pxa255 的 i2c 适配器驱动程序

pxa_i2c 数据结构

linux/i2c.h 中定义的 i2c_adapter 数据结构为描述各种 i2c 适配器的通用“模板”，它定义了注册总线上所有设备的 clients 指针数组、指向具体 i2c 适配器的总线通信方法 i2c_algorithm 的 algo 指针、实现 i2c 总线操作原子性的 lock 信号量等设施。但在适配器的驱动程序中可以根据具体适配器的需要“扩充”该数据结构。

在 pxa255 的 i2c 适配器驱动程序补丁中通过 pxa_i2c 数据结构来描述 pxa255 的 i2c 适配器：

```
struct pxa_i2c {
    spinlock_t              lock;
    wait_queue_head_t       wait;
    struct i2c_msg          *msg;
    unsigned int            msg_num;
    unsigned int            msg_idx;
    unsigned int            msg_ptr;
    struct i2c_adapter      adap;
    int                     irqlogidx;
    u32                     isrlog[32];
    u32                     icrlog[32];
};
```

该数据结构扩展了已有的 i2c_adapter 数据结构，其主体为 i2c_adapter 类型的域 adap。由于 i2c 操作为串行的，所以可以采用“阻塞－中断”的驱动模型，即读写 i2c 设备的用户进程在 i2c 操作期间进入阻塞状态，待 i2c 操作完成后总线适配器将引发中断，在相应的中断处理程序中唤醒受阻的用户进程。所以在 pxa_i2c 数据结构中设计了等待队列首部 wait 域。受阻的用户进程在阻塞期间仍然持有 i2c_adapter 中的 lock 信号量，从而实现上层用户进程对一条 i2c 总线访问的同步。

i2c 总线的同步问题还包括另一个方面：在读写 i2c 设备的系统调用执行流中最终将通过 i2c_pxa_do_xfer 函数初始化指向待传输信息的指针（msg，msg_idx，msg_ptr）并发起 i2c 通信（详见下文），而此期间可能发生 i2c 中断，而在 i2c 中断处理程序 i2c_pxa_handler 中也会步进指向待传输信息的指针。所以应该把 i2c_pxa_do_xfer 函数的相关操作放到中断禁区中执行，于是才设计 lock 自旋锁并通过 spin_lock/unlock_irq 函数来实现中断禁区。

自旋锁在嵌入式应用中往往都退化为实现中断禁区，另外当前应用所采用的 2.4 版本 Linux 中尚不支持内核态抢占，所以自旋锁的分配、释放操作也无需改变原子性计数 preempt_count。于是可以通过 spin_lock/unlock_irq 函数同步中断处理程序，或者通过 spin_lock/unlock_bh 同步下半部分（一定要分析清楚竞争条件的双方究竟是谁，从而采用适宜的同步机制。比如，如果当前内核控制路径可能与下半部分发生竞争条件，那么使用 spin_lock/unlock_irq 函数就有“扩大打击面”之嫌了，呵呵）。

一次 i2c 操作可以包含若干次交互，一个交互的传输方向是确定的，由一个 i2c_msg 数据结构进行描述。由于在 i2c 操作的开始必须由 master 将 slave 设备的地址“上传”到 i2c 总线上，所以一次读操作就至少包两个交互：第一次为写操作，由 master 在发送 START 分节后发出 slave 的地址及 WR 位段，等待 slave 回送 ACK 分节，然后 master 写入的第 2 个字节可以为待读出的寄存器编号；第二次为读操作，master 接收到 slave 回送的 ACK 分节后重新发送 START 分节，以告知 slave 设备可以开始发送数据了。所以一次读操作就至少需要两个 i2c_msg 数据结构进行描述，详见下文。

指针 msg 指向用于描述一组 i2c 交互的 i2c_msg 数据结构的数组：

```
struct i2c_msg {
    __u16           addr;                   /* slave address */
    unsigned short  flags;
    short           len;                    /* msg length  */
    char            *buf;                   /* pointer to msg data */
};
```

其中 addr 为设备的总线地址，buf 指向与 i2c 设备交互的数据缓冲区，其长度为 len。flags 中的标志位描述该 i2c 操作的属性，比如是读还是写。

一次 i2c 总线操作的 master 在获得 i2c 总线后可能与同一个设备进行读、写交互，也可能与多个 slave 设备交互，这些情况即 pxa255 开发者手册 9.3.3 节图 9-4 中所述的“Repeated Start”情形，使得 master 可以连续进行多个 i2c 交互而不放弃总线。

每完成一次交互，msg 指针和 msg_idx 计数器都会增加。而 msg_ptr 总是指向当前交互中要传送、接收的下一个字节在 i2c_msg.buf 中的偏移。最后 icrlog 和 isrlog 数组分别是 i2c 控制寄存器和状态寄存器的部分历史记录，可用作调试信息。

在 pxa255 的 i2c 适配器驱动程序模块中定义的 pxa_i2c 类型的变量为 i2c_pxa，注意其 algo 指针指向模块中定义的 i2c_algorithm 类型的静态变量 i2c_pxa_algorithm：

```
static struct pxa_i2c i2c_pxa = {
    .lock= SPIN_LOCK_UNLOCKED,
    .wait= __WAIT_QUEUE_HEAD_INITIALIZER(i2c_pxa.wait),
    .adap     = {
        .name       = "pxa2xx-i2c",
        .id         = I2C_ALGO_PXA,
        .algo       = &i2c_pxa_algorithm,
        .retries    = 5,
        .inc_use    = i2c_pxa_inc_use,      /* Added by shrek2 */
        .dec_use    = i2c_pxa_dec_use,      /* Added by shrek2 */
    },
};
```

```
static struct i2c_algorithm i2c_pxa_algorithm = {
    .name           = "PXA-I2C-Algorithm",
    .id             = I2C_ALGO_PXA,
    .master_xfer    = i2c_pxa_xfer,
    .smbus_xfer     = NULL,                 /* Added by shrek2 */
    .functionality  = i2c_pxa_functionality, /* Added by shrek2 */
};
```

i2c_pxa_algorithm 描述了 pxa255 的 i2c 适配器上的总线通信方法，其 ID 为 I2C_ALGO_PXA，定义于 linux/i2c-id.h。在 pxa255 上通过 i2c_pxa_xfer 函数全全执行 i2c 总线的通信任务。另外从这个数据结构也看到该 pxa255 的 i2c 驱动程序补丁尚不支持 smbus 协议，只支持 i2c 协议。

i2c_adap_pxa_init 函数

（注：从 www.arm.linux.org.uk 上下载的有关 PXA I2C 的补丁 2360-2 是针对 linux2.6.11 的，其模块的初始化函数利用了在 2.6 内核中才支持的 driver_register 函数及 device_driver 数据结构来进行注册。但是这个函数在当前我使用的 2.4.19 内核中还不存在，于是我就直接把 device_driver.probe 所指向的方法直接当作了模块的初始化函数。）

该函数用于 pxa255 的 i2c 适配器驱动程序的初始化：开启适配器时钟源、注册中断处理程序、初始化适配器、向内核注册描述适配器的 i2c_adapter 数据结构。

```
static int i2c_adap_pxa_init(void)
{
    struct pxa_i2c *i2c = &i2c_pxa;
    int ret;

    CKEN |= CKEN14_I2C;
    ret = request_irq(IRQ_I2C, i2c_pxa_handler, SA_INTERRUPT, "pxa2xx-i2c", i2c);
    if (ret)
        goto out;
```

首先将局部变量 i2c 指向静态创建的 pxa_i2c 类型的数据结构对象，然后开启 PXA255 的 i2c 适配器的时钟源，并注册 i2c 适配器的中断处理程序。宏 CKEN 和 CKEN14_I2C 均定义于 asm/arch/pxa-regs.h，将相应位置 1 即开启了 i2c 总线时钟。i2c 适配器使用的 IRQ 号为 IRQ_I2C，定义于 asm/irq.h，注册的中断处理程序为 i2c_pxa_handler。与中断管理有关的位掩码为 SA_INTERRUPT，表示该 IRQ 号被独占使用而不能被共享，而且是一个“快速”中断处理程序，即在执行中断处理期间 CPU 是关闭中断的。

该中断源的名字是“pxa2xx-i2c”。最后一个参数 void *dev_id 被赋值为 i2c_pxa 数据结构的地址。dev_id 指针一般指向设备私有数据结构，可以用在共享 IRQ 的中断处理程序中（当共享的 IRQ 上有中断发生时，共享该 IRQ 的所有中断处理程序都会被依次调用，dev_id 指针被原封不动地传递给相应的中断处理程序，那么在中断处理程序中就可以通过 dev_id 所指向的设备私有数据结构，或者直接通过设备的硬件状态寄存器来判断是否是这个设备真正引发了中断。可以参见《Linux 设备驱动程序》第 9 章）。现在当 i2c 适配器中断处理程序执行时，内核将传递 i2c_pxa 数据结构的地址。

然后，就是通过 i2c_pxa_reset 函数初始化 i2c 适配器了，详见后文。

```
    i2c_pxa_reset();
    i2c->adap.algo_data = i2c;
```

i2c_pxa 数据结构的主体是 i2c_adapter 数据结构，将其中的 algo_data 指针指向整个 i2c_pxa 数据结构的起始，这样通过 i2c_adapter.algo_data 就可以找到 i2c_pxa 数据结构了。

```
    ret = i2c_add_adapter(&i2c->adap);
    if (ret < 0) {
        pr_info("I2C: Failed to add bus\n");
        free_irq(IRQ_I2C, i2c);
        goto out;
    }
out:
    return ret;
}
```

最后就是利用 i2c 框架中的 i2c_add_adapter 函数注册 i2c_pxa 中所包含的 i2c_adapter 数据结构了。

i2c_add_adapter 函数

该函数用于向内核注册描述适配器及其驱动程序的 i2c_adapter 数据结构：

```
int i2c_add_adapter(struct i2c_adapter *adap)
{
    int i,j,res;

    ADAP_LOCK();
    for (i = 0; i < I2C_ADAP_MAX; i++)
        if (NULL == adapters[i])
            break;
    if (I2C_ADAP_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: register_adapter(%s) - enlarge I2C_ADAP_MAX.\n",
            adap->name);
        res = -ENOMEM;
        goto ERROR0;
    }
    adapters[i] = adap;
    adap_count++;
    ADAP_UNLOCK();
```

首先在内核静态数组 adapters 中找到一个未用项，将其指向这个 i2c_adapter 数据结构即完成注册，并增加

计数器。注意在访问全局数组期间要持有 adap_lock 信号量。

```
    /* init data types */
    init_MUTEX(&adap->lock);

#ifdef CONFIG_PROC_FS
    if (i2cproc_initialized) {
        char name[8];
        struct proc_dir_entry *proc_entry;
        sprintf(name,"i2c-%d", i);
        proc_entry = create_proc_entry(name,            /* name */
                                       0,               /* mode */
                                       proc_bus);       /* parent */
        if (! proc_entry) {
            printk("i2c-core.o: Could not create /proc/bus/%s\n", name);
            res = -ENOENT;
            goto ERROR1;
        }
```

由上文可知，在 i2c-core.c 文件的 i2c_init 函数中已经通过 i2cproc_init 函数创建/proc/bus/i2c 文件，如果创建成功则 i2cproc_initialized 变量大于 0。然后就可以通过 create_proc_entry 函数进一步在/proc/bus 目录下创建文件 i2c-%d（该文件用于描述一条具体 i2c 总线上的设备连接情况），其中%d 的数值正为刚刚注册的 i2c_adapter 数据结构的指针在内核静态数组 adapters 中的偏移。

```
#if (LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,48))
        proc_entry->proc_fops = &i2cproc_operations;
#else
        proc_entry->ops = &i2cproc_inode_operations;
#endif

#if (LINUX_VERSION_CODE >= KERNEL_VERSION(2,3,27))
        proc_entry->owner = THIS_MODULE;
#else
        proc_entry->fill_inode = &monitor_bus_i2c;
#endif
        adap->inode = proc_entry->low_ino;
    }
#endif /* def CONFIG_PROC_FS */
```

proc 文件的访问方法由 i2cproc_operations 结构体定义，其中只定义了读方法为 i2cproc_bus_read，这个函数通过访问 i2c_adapter 数据结构中的 clients 数组，向用户进程返回在该 i2c 总线上所有已注册的设备的信息（可以参见内核文档 Documentation/i2c/proc-interface）。

```
    /* inform drivers of new adapters */
```

```
    DRV_LOCK();
    for (j=0;j<I2C_DRIVER_MAX;j++)
        if (drivers[j]!=NULL && (drivers[j]->flags&(I2C_DF_NOTIFY|I2C_DF_DUMMY)))
            /* We ignore the return code; if it fails, too bad */
            drivers[j]->attach_adapter(adap);
    DRV_UNLOCK();

    DEB(printk("i2c-core.o: adapter %s registered as adapter %d.\n", adap->name, i));
    return 0;
```

最后遍历系统上所有已注册的 i2c 设备驱动程序，让它们都来尝试认领新注册的 i2c 适配器上的所有 i2c 设备，详见 ltc3445 设备驱动程序分析。

```
ERROR1:
    ADAP_LOCK();
    adapters[i] = NULL;
    adap_count--;
ERROR0:
    ADAP_UNLOCK();
    return res;
}
```

i2c_del_adapter 函数

与 i2c_add_adapter 函数相对，i2c_del_adapter 函数用于注销适配器的数据结构、删除其总线上所有设备的 i2c_client 数据结构、并减少其代表总线上所有设备的相应驱动程序数据结构的引用计数（如果到达 0，则卸载设备驱动程序）、删除/proc/bus/i2c-%d 文件。

```
int i2c_del_adapter(struct i2c_adapter *adap)
{
    int i,j,res;

    ADAP_LOCK();
    for (i = 0; i < I2C_ADAP_MAX; i++)
        if (adap == adapters[i])
            break;
    if (I2C_ADAP_MAX == i) {
        printk( "i2c-core.o: unregister_adapter adap [%s] not found.\n",
            adap->name);
        res = -ENODEV;
        goto ERROR0;
    }
```

首先在获得全局信号量 adap_lock 的情况下在内核静态数组 adapters[]中找到待注销的适配器驱动程序。

```
/* DUMMY drivers do not register their clients, so we have to
 * use a trick here: we call driver->attach_adapter to
 * *detach* it! Of course, each dummy driver should know about
 * this or hell will break loose...
 */
DRV_LOCK();
for (j = 0; j < I2C_DRIVER_MAX; j++)
    if (drivers[j] && (drivers[j]->flags & I2C_DF_DUMMY))
        if ((res = drivers[j]->attach_adapter(adap))) {
            printk("i2c-core.o: can't detach adapter %s "
                    "while detaching driver %s: driver not "
                    "detached!",adap->name,drivers[j]->name);
            goto ERROR1;
        }
DRV_UNLOCK();
```

以上代码与类型为 I2C_DF_DUMMY 的设备驱动有关。但是在我们的应用中目前只有 ltc3445 一个设备，而且其驱动程序类型为 I2C_DF_NOTIFY，所以暂时跳过这段代码。

```
/* detach any active clients. This must be done first, because
 * it can fail; in which case we give upp. */
for (j=0;j<I2C_CLIENT_MAX;j++) {
    struct i2c_client *client = adap->clients[j];
    if (client!=NULL)
        /* detaching devices is unconditional of the set notify
         * flag, as _all_ clients that reside on the adapter
         * must be deleted, as this would cause invalid states.
         */
        if ((res=client->driver->detach_client(client))) {
            printk("i2c-core.o: adapter %s not unregistered, because client at "
                "address %02x can't be detached. ", adap->name, client->addr);
            goto ERROR0;
        }
}
```

然后依次调用该 i2c 总线上所有设备的驱动程序的 detach_client 函数，该函数通过 i2c_detach_client 函数完成适配器端的注销工作，即从适配器的 clients[]数组中注销设备数据结构；并通过 kfree 释放设备的 i2c_client 及相应的私有数据结构，从而完成设备端的注销工作。详见 ltc3445 设备驱动程序分析。

```
#ifdef CONFIG_PROC_FS
    if (i2cproc_initialized) {
        char name[8];
        sprintf(name,"i2c-%d", i);
```

```
            remove_proc_entry(name,proc_bus);
        }
#endif /* def CONFIG_PROC_FS */
        adapters[i] = NULL;
        adap_count--;
```

最后，就是删除/proc/bus/i2c-%d 文件，并从内核静态数组 adapters[]中注销适配器驱动并减少计数值。

```
        ADAP_UNLOCK();
        DEB(printk("i2c-core.o: adapter unregistered: %s\n",adap->name));
        return 0;
ERROR0:
        ADAP_UNLOCK();
        return res;
ERROR1:
        DRV_UNLOCK();
        return res;
}
```

i2c_adap_pxa_exit 函数

该函数用于注销 pxa255 的 i2c 适配器驱动程序模块。

```
void i2c_adap_pxa_exit(void)
{
        struct pxa_i2c *i2c = &i2c_pxa;

        i2c_del_adapter(&i2c->adap);
        free_irq(IRQ_I2C, i2c);
        CKEN &= ~CKEN14_I2C;

        return 0;
}
```

既然在模块的初始化函数 i2c_adap_pxa_init 中使能了 i2c 适配器的时钟、注册了中断处理程序、注册了适配器驱动程序，那么在模块的清理函数中就反向地撤销这些操作。

第 4 章 安装、卸载 ltc3445 驱动程序

ltc3445 驱动程序模块定义了描述 ltc3445 的数据结构及相应的初始化方法，提供了地址线索、地址检测方法和向适配器注册、注销设备的方法。

ltc3445_init 函数

该函数为 ltc3445 设备驱动程序模块的初始化函数。

```
static int ltc3445_init(void)
{
        int res;
        if ((res = i2c_add_driver(&ltc3445_driver))){
            printk("ltc3445: Driver registration failed, module not inserted.\n");
            ltc3445_cleanup();
            return res;
        }
        ltc3445_initialized++;
        return 0;
}
```

它只需调用 i2c_add_driver 函数注册由 ltc3445_driver 数据结构描述的驱动程序：

```
static struct i2c_driver ltc3445_driver
{
    .name           = "LTC3445 I2C driver 0.1",
    .id             = I2C_DRIVERID_LTC3445,
    .flags          = I2C_DF_NOTIFY,
    .attach_adapter = ltc3445_attach_adapter,
    .detach_client  = ltc3445_detach_client,
    .command        = ltc3445_command,    /* May be NULL */
    .inc_use        = ltc3445_inc_use,    /* May be NULL */
    .dec_use        = ltc3445_dec_use     /* May be NULL */
};
```

其中 ltc3445_attach_adapter 函数在安装 i2c 适配器驱动程序模块或 i2c 设备驱动程序模块时用于认领设备；ltc3445_detach_client 函数用于在卸载适配器驱动程序或设备驱动程序时注销设备数据结构。由前文可见在 i2c_add_adapter 和 i2c_del_adapter 函数中调用的正是这两个方法。

i2c_add_driver 函数

该函数用于注册设备驱动程序的 i2c_driver 数据结构。

```
int i2c_add_driver(struct i2c_driver *driver)
{
    int i;
    DRV_LOCK();
    for (i = 0; i < I2C_DRIVER_MAX; i++)
        if (NULL == drivers[i])
            break;
    if (I2C_DRIVER_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: register_driver(%s) " "- enlarge I2C_DRIVER_MAX.\n",
            driver->name);
        DRV_UNLOCK();
        return -ENOMEM;
    }
```

首先要在 drivers 数组中找到一个未用项。注意在访问 drivers 指针数组期间要持有信号量 driver_lock。然后就可以将该数组元素指向这个 i2c_driver 数据结构即完成注册，并增加计数值：

```
    drivers[i] = driver;
    driver_count++;

    DRV_UNLOCK();       /* driver was successfully added */
    DEB(printk("i2c-core.o: driver %s registered.\n",driver->name));

    ADAP_LOCK();
    /* now look for instances of driver on our adapters    */
    if (driver->flags& (I2C_DF_NOTIFY | I2C_DF_DUMMY)) {
        for (i=0;i<I2C_ADAP_MAX;i++)
            if (adapters[i]!=NULL)
                /* Ignore errors */
                driver->attach_adapter(adapters[i]);
    }
    ADAP_UNLOCK();
    return 0;
}
```

使用该驱动程序的设备可能处在一条 i2c 总线上，也可能处在不同的 i2c 总线上。所以应该遍历当前所有的 i2c 总线“认领”其上可能存在的设备，即进行设备地址检测。这个任务通过一个循环，对 adapters 数组所指所有的 i2c_adapter 数据结构调用驱动程序的 attach_adapter 方法实现（这个循环是外层、针对 i2c 总线的循环，而设备驱动程序 attach_adapter 内的循环是针对一条 i2c 总线的内部循环）。

i2c_probe 函数

设备驱动程序模块必须实现的 attach_adapter 方法可以简单地通过调用 i2c 框架提供的 i2c_probe 函数实现：

```
int ltc3445_attach_adapter(struct i2c_adapter *adapter)
{
    return i2c_probe(adapter,&addr_data, ltc3445_detect_client);
}
```

其中参数 adapter 为内核指针数组 adapters[]的一个元素，代表当前被扫描的 i2c 总线。addr_data 是在 ltc3445.h 中由 I2C_CLIENT_INSMOD 宏创建的静态二维数组，表示使用该驱动程序的所有设备的所有可能地址的集合。第三个参数为一个在地址成功检测时被调用的回调函数，用于创建描述该设备的 i2c_client 数据结构并初始化。

i2c_probe 函数用于认领 adapter 所指适配器上的所有合适的设备。设备可能使用的地址的“线索”由 address_data 二元数组指出，如果检测到存在实际设备，则调用 found_proc 回调函数分配、初始化设备的 i2c_client 等数据结构：

```
int i2c_probe(struct i2c_adapter *adapter, struct i2c_client_address_data *address_data,
                    i2c_client_found_addr_proc *found_proc)
{
    int addr,i,found,err;
    int adap_id = i2c_adapter_id(adapter);

    /* Forget it if we can't probe using SMBUS_QUICK */
    if (! i2c_check_functionality(adapter,I2C_FUNC_SMBUS_QUICK))
        return -1;
```

adap_id 即为当前适配器在 adapters 数组中的索引。首先通过 i2c_check_functionality 函数确认当前适配器支持 I2C_FUNC_SMBUS_QUICK 方法。

```
    for (addr = 0x00; addr <= 0x7f; addr++) {
        /* Skip if already in use */
        if (i2c_check_addr(adapter,addr))
            continue;
```

每一个 i2c 设备在其所在 i2c 总线上都有惟一的 7 位地址，设备驱动程序模块的 addr_data 二维数组提供了设备可能使用的地址。那么检查设备实际使用地址就可以采用“穷举法”，即通过循环逐一将所有可能的地址与 addr_data 相比对。另外，在安装当前设备驱动程序模块时系统上可能已经存在已注册过的设备及其驱动程序了，所以在进行逐一比对时要跳过那些已经被使用的地址。

```
        /* If it is in one of the force entries, we don't do any detection at all */
        found = 0;
        for (i = 0; !found && (address_data->force[i] != I2C_CLIENT_END); i += 3) {
```

```
        if (((adap_id == address_data->force[i]) || (address_data->force[i] == ANY_I2C_BUS)) &&
              (addr == address_data->force[i+1])) {
            DEB2(printk("i2c-core.o: found force parameter for adapter %d, addr %04x\n",
                        adap_id,addr));
            if ((err = found_proc(adapter,addr,0,0)))
                return err;
            found = 1;
        }
    }
    if (found)
        continue;
```

address_data 中的 force 数组若不为空，则认为在指定地址上一定存在相应的设备，于是可以直接调用回调函数 found_proc。force 数组由二元数对组成，第一个数表示当前 i2c 适配器的编号，第二个数为设备在该 i2c 总线上的相对地址。如果 adap_id 和 addr 和 force 数组中的某两个连续的数值相符，则认为地址匹配了。如前文所述，在一个 i2c 总线上可能有使用同一个 i2c_driver 的多个设备，所以当一个地址匹配成功后还好继续寻找仍然可能匹配的地址，故当 found 为正时开始新的循环。
（另外，我觉得源代码中“i +=3”有误，应该是“i += 2”，因为 force 数组由二元数对组成）

```
    /* If this address is in one of the ignores, we can forget about it right now */
    for (i = 0; !found && (address_data->ignore[i] != I2C_CLIENT_END); i += 2) {
        if (((adap_id == address_data->ignore[i]) || ((address_data->ignore[i] == ANY_I2C_BUS))) &&
            (addr == address_data->ignore[i+1])) {
            DEB2(printk("i2c-core.o: found ignore parameter for adapter %d, "
                  "addr %04x\n", adap_id ,addr));
            found = 1;
        }
    }
    for (i = 0; !found && (address_data->ignore_range[i] != I2C_CLIENT_END);    i += 3) {
        if (((adap_id == address_data->ignore_range[i]) ||
            ((address_data->ignore_range[i]==ANY_I2C_BUS))) &&
            (addr >= address_data->ignore_range[i+1]) && (addr <= address_data->ignore_range[i+2])){
            DEB2(printk("i2c-core.o: found ignore_range parameter for adapter %d, "
                        "addr %04x\n", adap_id,addr));
            found = 1;
        }
    }
    if (found)
        continue;
```

如果当前地址落在 ignore 数组或者 ignore_range 数组元素内，则直接跳过该地址开始新的循环。

```
    /* Now, we will do a detection, but only if it is in the normal or probe entries */
    for (i = 0; !found && (address_data->normal_i2c[i] != I2C_CLIENT_END); i += 1) {
```

```
            if (addr == address_data->normal_i2c[i]) {
                found = 1;
                DEB2(printk("i2c-core.o: found normal i2c entry for adapter %d, "
                            "addr %02x", adap_id,addr));
            }
        }
        for (i = 0; !found && (address_data->normal_i2c_range[i] != I2C_CLIENT_END); i += 2) {
            if ((addr >= address_data->normal_i2c_range[i]) &&
                (addr <= address_data->normal_i2c_range[i+1])) {
                found = 1;
                DEB2(printk("i2c-core.o: found normal i2c_range entry for adapter %d, "
                            "addr %04x\n", adap_id,addr));
            }
        }
        for (i = 0; !found && (address_data->probe[i] != I2C_CLIENT_END); i += 2) {
            if (((adap_id == address_data->probe[i]) || ((address_data->probe[i] == ANY_I2C_BUS))) &&
                (addr == address_data->probe[i+1])) {
                found = 1;
                DEB2(printk("i2c-core.o: found probe parameter for adapter %d, "
                            "addr %04x\n", adap_id,addr));
            }
        }
        for (i = 0; !found && (address_data->probe_range[i] != I2C_CLIENT_END); i += 3) {
            if (((adap_id == address_data->probe_range[i]) ||
                (address_data->probe_range[i] == ANY_I2C_BUS)) &&
                (addr >= address_data->probe_range[i+1]) && (addr <= address_data->probe_range[i+2])) {
                found = 1;
                DEB2(printk("i2c-core.o: found probe_range parameter for adapter %d, "
                            "addr %04x\n", adap_id,addr));
            }
        }
        if (!found)
            continue;
```

address_data 二维数组提供了设备可能处在的地址的线索。如果当前地址不在 probe、probe_range、normal_i2c、normal 数组内，则跳过该地址。否则就应该真正检测该地址上是否存在设备：

```
        /* OK, so we really should examine this address. First check whether there is some client here at all! */
        if (i2c_smbus_xfer(adapter,addr,0,0,0,I2C_SMBUS_QUICK,NULL) >= 0)
            if ((err = found_proc(adapter,addr,0,-1)))
                return err;
    }
    return 0;
}
```

i2c_smbus_xfer 函数通过该适配器驱动程序实现的总线通信方法（由 i2c_algorithm 的 smbus_xfer 指向）检测该地址上是否存在设备。如果在当前地址上的确存在设备则调用 found_proc 回调函数。注意此时传递的最后一个参数为-1，而用 force 指定的地址时传递 0。

由于在 ltc3445 驱动程序模块中只定义了 normal_i2c[]数组，所以 i2c_probe 的这七个 for 循环只有处理 normal_i2c[]数组的那个会被执行，最终通过 i2c_smbus_xfer 检测设备真正使用的是 normal_i2c[]数组中的那些地址。

正是核心数据结构的设计使得 i2c 框架与具体的应用无关，从而提供了良好的可移植性和重用性，i2c_probe 函数就是一个很好的例子。无论具体的 i2c 适配器或者设备如何，其 i2c 总线的通信方法总由 i2c_adapter 数据结构的 algo 指针所指向的 i2c_algorithm 数据结构的 master_xfer 或 smbus_xfer 函数提供；另外无论安装驱动的先后顺序如何，认领设备即检测设备地址的过程总是相同的。正是因为 i2c_algorithm 这个中间层和其它数据结构的支持，具体应用的开发者只需在设备端用 addr_data 指明具体 i2c 设备实际可能使用的地址，并提供创建和初始化 i2c_client 的回调函数；在适配器端实现具体 i2c 适配器的总线控制方法，结合 i2c 框架代码就可以实现完整的 i2c 应用了。

i2c_check_functionality 函数

该函数用于判断指定适配器是否支持相应的方法：

```
int i2c_check_functionality (struct i2c_adapter *adap, u32 func)
{
    u32 adap_func = i2c_get_functionality (adap);
    return (func & adap_func) == func;
}
```

通过 i2c_get_functionality 函数得到该适配器所支持的所有方法的位图，如果参数所指方法的相应位有效，则表示支持该方法。

```
/* You should always define `functionality'; the 'else' is just for backward compatibility. */
u32 i2c_get_functionality (struct i2c_adapter *adap)
{
    if (adap->algo->functionality)
        return adap->algo->functionality(adap);
    else
        return 0xffffffff;
}
```

而 i2c_get_functionality 函数又是通过调用适配器所采用的 i2c_algorithm 数据结构中定义的 functionality 函数得到该适配器所支持的方法的位图的。在 pxa255 的 i2c 补丁中，该方法被投其所好地实现如下：

```
static u32 i2c_pxa_functionality(struct i2c_adapter *adap)
{
```

```
    return      I2C_FUNC_I2C | I2C_FUNC_SMBUS_QUICK;       /* needed by i2c_probe */
}
```

i2c_smbus_xfer 函数

该函数通过适配器驱动提供的总线访问方法（i2c_algorithm 的 smbus_xfer 方法）尝试访问处于 addr 地址上的设备。

```
s32 i2c_smbus_xfer(struct i2c_adapter * adapter, u16 addr, unsigned short flags,
                        char read_write, u8 command, int size, union i2c_smbus_data * data)
{
    s32 res;
    flags = flags & I2C_M_TEN;
    if (adapter->algo->smbus_xfer) {
        I2C_LOCK(adapter);
        res = adapter->algo->smbus_xfer(adapter, addr, flags, read_write, command, size, data);
        I2C_UNLOCK(adapter);
    } else
        res = i2c_smbus_xfer_emulated(adapter, addr, flags, read_write, command, size, data);
    return res;
}
```

由于目前 pxa255 的 i2c 补丁中没有实现 smbus_xfer 方法而只实现了 master_xfer 方法，所以相关的操作就通过 i2c_smbus_xfer_emulated 函数来模拟。这个函数较长，并且在 i2c_probe 函数中对其的调用为：

```
i2c_smbus_xfer(adapter,addr,0,0,0,I2C_SMBUS_QUICK,NULL)
```

所以这里只分析代码中 size 为 I2C_SMBUS_QUICK 的一段。第二个参数为需要检测的设备地址，第四个参数 read_write 为 0，表示进行写入操作。如果使用这个总线地址和设备通信成功，则说明设备使用的正是这个地址。

```
static s32 i2c_smbus_xfer_emulated(struct i2c_adapter * adapter, u16 addr,
                                   unsigned short flags,
                                   char read_write, u8 command, int size,
                                   union i2c_smbus_data * data)
{
    unsigned char msgbuf0[34];
    unsigned char msgbuf1[34];
    int num = read_write == I2C_SMBUS_READ?2:1;
    struct i2c_msg msg[2] = {   { addr, flags, 1, msgbuf0 },
                                { addr, flags | I2C_M_RD, 0, msgbuf1 }
                            };
    int i;
    msgbuf0[0] = command;
```

一次 i2c 操作可以包含若干次交互，一个交互的传输方向是确定的，由一个 i2c_msg 数据结构进行描述：

```
struct i2c_msg {
    __u16           addr;                   /* slave address */
    unsigned short  flags;
    short           len;                    /* msg length  */
    char            *buf;                   /* pointer to msg data */
};
```

其中 addr 为设备的总线地址，buf 指向与 i2c 设备交互的数据缓冲区，其长度为 len。flags 中的标志位描述该 i2c 操作的属性，比如是读还是写。

由于在 i2c 操作的开始必须由 master 将 slave 设备的地址“上传”到 i2c 总线上，所以一次读操作就至少包两个交互：第一次为写操作，由 master 在发送 START 分节后发出 slave 的地址及 WR 位段，等待 slave 回送 ACK 分节。另外 master 上传到 i2c 总线上的第二个字节可以是待读出的设备寄存器编号；第二次为读操作，master 接收到 slave 回送的 ACK 分节后重新发送 START 分节，以告知 slave 设备可以开始发送数据了（参见 ltc3445 的读规程）。所以读操作就至少需要两个 i2c_msg 数据结构进行描述，而写操作可以只需要一个 i2c_msg 数据结构。

局部变量 num 表示需要的 i2c_msg 数据结构的个数。宏 I2C_SMBUS_READ 在 i2c.h 中被定义为 1，而地址检测时由写操作完成，所以只需要一个 i2c_msg 数据结构即可，故上文调用该函数时传递的第 4 个参数 read_write 为 0，从而使得 num 等于 1。

```
    switch(size) {
    case I2C_SMBUS_QUICK:
        msg[0].len = 0;
        /* Special case: The read/write field is used as data */
        msg[0].flags = flags | (read_write==I2C_SMBUS_READ)?I2C_M_RD:0;
        num = 1;
        break;
        ……
    }
```

i2c_probe 函数中通过写操作进行设备检测的思路是：如果接收到了设备在地址匹配后回送的应答 ACK，则说明设备使用该地址。所以在检测设备时无需传递任何数据，故将 msg[0].len 设置为 0。另外，传递的参数 read_write 为 0，表示写入操作，则 msg[0].flags 中**没有**设置 I2C_M_RD 标志（这个标志只有读设备时才设置用于告知从设备当前 i2c 交互的方向。由上文可见，在 i2c_pxa_start_message 函数中如果判断该标志有效，则设置 IDBR 的末位为 1）。

```
        if (i2c_transfer(adapter, msg, num) < 0)
            return -1;

        if (read_write == I2C_SMBUS_READ)
```

```
            ……
        return 0;
    }
```

最后，就是通过 i2c_transfer 函数进行实际的 i2c 操作了。

i2c_transfer 函数

该函数几乎与 i2c_master_recv 和 i2c_master_send 函数一模一样，都是在持有 i2c_adapter 的 lock 信号量的情况下利用 i2c 适配器的 algo 所指的 i2c_algorithm 数据结构的 master_xfer 方法执行实际的 i2c 操作：

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg msgs[],int num)
{
    int ret;
    if (adap->algo->master_xfer) {
        DEB2(printk("i2c-core.o: master_xfer: %s with %d msgs.\n", adap->name,num));

        I2C_LOCK(adap);
        ret = adap->algo->master_xfer(adap,msgs,num);
        I2C_UNLOCK(adap);
        return ret;
    } else {
        printk("i2c-core.o: I2C adapter %04x: I2C level transfers not supported\n", adap->id);
        return -ENOSYS;
    }
}
```

pxa255 的 i2c 驱动采用“阻塞一唤醒”模型，由于地址检测在 master 收到 slave 发送的 ACK 分节后即可完成，但是此时 slave 已经处于地址匹配了的状态，所以 master 应该复位总线从而复位 slave 的状态。因此还需要在 i2c 的中断处理程序中编写对应的处理代码，比如在 i2c_pxa_irq_txempty 函数的末尾有如下代码：

```
else {
        if (i2c->msg->len == 0) {
            /*
             * Device probe's have a meesage length of zero and need
             * the bus to be reset before it can be used again.
             */
            pr_debug("i2c-pxa: resetting unit");
            i2c_pxa_reset();
        }
        i2c_pxa_master_complete(i2c, 0);
}
```

函数调用链 i2c_probe > i2c_smbus_xfer > i2c_smbus_xfer_emulated > i2c_transfer 最终会阻塞在 i2c 适配器驱

动的 master_xfer 方法上，此时当前执行流（执行 insmod 操作的用户进程）阻塞在 i2c_pxa.wait 等待队列上。master 在写入设备地址后如果收到设备的 ACK 应答，则引发 Transmit Empty 中断，在 i2c_pxa_irq_txempty 函数中如果发现当前交互的长度为 0，则说明当前交互用于进行设备检测，而且此时检测成功，所以通过 i2c_pxa_master_complete 函数唤醒等待的执行流。注意传递的第二个参数为返回码，0 表示成功。那么返回到 i2c_probe 中：

```
if (i2c_smbus_xfer(adapter,addr,0,0,0,I2C_SMBUS_QUICK,NULL) >= 0)
        if ((err = found_proc(adapter,addr,0,-1)))
            return err;
```

就可以接着调用 ltc3445 设备驱动提供的回调函数 ltc3445_detect_client 了。

由于检测设备时并没有传递任何数据，而设备的硬件状态机已经步进到“地址已匹配”状态了，所以必须重启 pxa255 的 i2c 适配器以给设备发送 STOP 使得设备硬件恢复初始状态，否则设备的逻辑将出现混乱。

ltc3445_detect_client 函数

当地址检测成功时将调用设备驱动程序模块中定义的 ltc3445_detect_client 方法。第一个参数指明设备所在的 i2c 总线，第二个参数指明设备的总线地址。第三个参数在认领设备的两种情景中都是 0。如果地址由 force 数组指定，则最后一个参数为 0，若该地址上经检测的确存在设备，则为-1。

```
static int ltc3445_detect_client(struct i2c_adapter *adapter, int address, unsigned short flags, int kind)
{
        int err = 0;
        struct i2c_client *new_client;
        const char *client_name = "LTC3445 I2C driver 0.1";

        /* Let's see whether this adapter can support what we need. */
        if (!i2c_check_functionality(adapter, I2C_FUNC_I2C | I2C_FUNC_SMBUS_QUICK))
            goto ERROR0;
```

首先检查 pxa255 的 i2c 适配器是否支持相应的方法。当前的 pxa255 的 i2c 补丁只实现了基本的 i2c 协议，所以一定支持 I2C_FUNC_I2C 标志。另外为了配合 i2c-core.c 文件中 i2c_probe 函数的需要，在 i2c 补丁中也支持了 I2C_FUNC_SMBUS_QUICK 标志。

```
    /* kmalloc i2c_client and private data */
    if (!(new_client = kmalloc(sizeof(struct i2c_client) + sizeof(struct ltc3445_data),
                                        GFP_KERNEL))){
            err = -ENOMEM;
            goto ERROR0;
    }
```

然后为设备分配一个 i2c_client 数据结构。如果定义了设备的私有数据结构则同时分配其空间，这里将私有数据结构直接放到了 i2c_client 数据结构之后。

```
    new_client->addr = address;
    new_client->data = (struct ltc3445_data *)(new_client + 1);
    new_client->adapter = adapter;
    new_client->driver = &ltc3445_driver;
    new_client->flags = I2C_CLIENT_ALLOW_USE;
    new_client->id = ltc3445_id++;          /* Automatically unique */
```

现在就可以将匹配的地址保存在 i2c_client 数据结构的 addr 域了，在下面注册该数据结构到 i2c 总线的 i2c_adapter 数据结构时还要进一步检查是否有地址冲突。由于 new_client 指针的类型是 struct i2c_client *，所以“new_client+1”操作将得到 i2c_client 数据结构的后继地址，这个地址正是设备的私有数据的起始地址，将其赋予 i2c_client.data。然后将 i2c_client 数据结构中的 adapter 指针指向 i2c 总线的 i2c_adapter 数据结构（以便下面调用 i2c_attach_client），并将 drivers 指针指向 ltc3445_driver 数据结构。

```
        /* Tell the i2c layer a new client has arrived */
        if ((err = i2c_attach_client(new_client)))
            goto ERROR1;

        /* Initialize private data and device */
        if ((err = ltc3445_init_device(new_client)) != 0)
            goto ERROR1;

        return 0;
ERROR1:
    kfree(new_client);
ERROR0:
        return err;
}
```

最后，就是通过 i2c 框架的 i2c_attach_client 函数注册这个 i2c_client 数据结构，并调用函数 ltc3445_init_device 初始化私有数据结构，并完成 ltc3445 设备的初始化（由于当前应用中没有为 ltc3445 设备设计私有数据结构，而且设备的初始化完全可以通过应用层读写/dev/i2c/%d 设备结点完成，所以这里的初始化函数可以为空）。

i2c_attach_client 函数

该函数向设备所在的总线注册设备数据结构。

```
int i2c_attach_client(struct i2c_client *client)
{
    struct i2c_adapter *adapter = client->adapter;
    int i;
```

在 ltc3445_detect_client 函数中已经将设备的 i2c_client 的 adapter 指针指向了其所在总线的 i2c_adapter 数据

结构，这只完成了设备端的注册工作，该函数则完成适配器端的另一半注册工作。

```
    if (i2c_check_addr(client->adapter,client->addr))
        return -EBUSY;
```

首先要检查这个新设备使用的地址不与总线上已有设备的地址冲突，否则返回-EBUSY。

```
    for (i = 0; i < I2C_CLIENT_MAX; i++)
        if (NULL == adapter->clients[i])
            break;
    if (I2C_CLIENT_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: attach_client(%s) - enlarge I2C_CLIENT_MAX.\n",
            client->name);
        return -ENOMEM;
    }
```

一条 i2c 总线上所有已注册设备的 i2c_client 数据结构被组织在 i2c_adapter 中的 clients 指针数组中。所谓注册，其实也就是在 clients 数组中找到第一个未使用的元素，并将其指向新的 i2c_client 数据结构：

```
    adapter->clients[i] = client;
    adapter->client_count++;

    if (adapter->client_register)
        if (adapter->client_register(client))
            printk("i2c-core.o: warning: client_register seems to have failed for client %02x at adapter %s\n",
                client->addr,adapter->name);
    DEB(printk("i2c-core.o: client [%s] registered to adapter [%s](pos. %d).\n",client->name, adapter->name,i));

    if(client->flags & I2C_CLIENT_ALLOW_USE)
        client->usage_count = 0;

    return 0;
}
```

最后，如果适配器驱动程序模块中提供了 client_register 方法则调用之，相信这个方法用于完成适配器端的额外的注册工作。而当前 pxa255 的 i2c 适配器驱动程序补丁中没有定义该方法（当前 i2c_adapter 和 i2c_client 数据结构已经能够通过 clients 指针数组和 adapter 指针相互指引了）。

ltc3445_cleanup 函数

在 ltc3445 驱动程序模块的初始化函数 ltc3445_init 中通过 i2c_add_driver 注册了驱动程序，那么在模块的清理函数中就得相应地调用 i2c_del_driver 注销驱动程序：

```
int ltc3445_cleanup(void)
```

```
{
        int res;

        if (ltc3445_initialized == 1) {
            if ((res = i2c_del_driver(&ltc3445_driver))) {
                printk("ltc3445: Driver registration failed, module not removed.\n");
                return res;
            }
        ltc3445_initialized--;
        }
        return 0;
}
```

i2c_del_driver 函数

使用同一设备驱动程序的若干 i2c 设备可能处在同一条 i2c 总线上，也可能处在不同的总线上。删除设备驱动程序首先要将其从内核静态数组 drivers[]中注销，而且，删除设备驱动程序意味着不再（无法继续）使用相应的设备，所以还应该同时删除相应设备的 i2c_client 等私有数据结构。

正因为使用同一驱动程序的设备可能不在同一条 i2c 总线上，所以这个函数采用类似地址检测的双层循环以达到遍历系统上所有 i2c 设备的目的：外层循环遍历系统上所有的 i2c 总线适配器，内层循环遍历一条总线上的所有已注册设备。如果设备使用的驱动程序就是这个待删除的驱动程序，则调用驱动程序的 detach_client 方法删除这个设备的 i2c_client 和其它私有数据结构，并从相应 i2c 总线上注销。

```
int i2c_del_driver(struct i2c_driver *driver)
{
    int i,j,k,res;

    DRV_LOCK();
    for (i = 0; i < I2C_DRIVER_MAX; i++)
        if (driver == drivers[i])
            break;
    if (I2C_DRIVER_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: unregister_driver: [%s] not found\n", driver->name);
        DRV_UNLOCK();
        return -ENODEV;
    }
```

在访问内核全局数据结构 drivers[]时必须首先获得 driver_lock 信号量。如果遍历该数组都没有找到相应的驱动程序，则释放信号量并退出。

```
    ADAP_LOCK(); /* should be moved inside the if statement... */
    for (k=0;k<I2C_ADAP_MAX;k++) {
        struct i2c_adapter *adap = adapters[k];
```

```
if (adap == NULL) /* skip empty entries. */
    continue;
```

外层循环，遍历系统上所有注册的 i2c 适配器。必须首先获得 adap_lock 信号量。

```
DEB2(printk("i2c-core.o: examining adapter %s:\n", adap->name));

if (driver->flags & I2C_DF_DUMMY) {
/* DUMMY drivers do not register their clients, so we have to
 * use a trick here: we call driver->attach_adapter to
 * *detach* it! Of course, each dummy driver should know about
 * this or hell will break loose...
 */
    if ((res = driver->attach_adapter(adap))) {
        printk("i2c-core.o: while unregistering "
                "dummy driver %s, adapter %s could "
                "not be detached properly; driver "
                "not unloaded!",driver->name,
                adap->name);
        ADAP_UNLOCK();
        return res;
    }
```

这部分代码与 DUMMY 类型的设备驱动有关。而 ltc3445 驱动的类型为 I2C_DF_NOTIFY，所以这里暂时跳过这段代码。

```
} else {
    for (j=0;j<I2C_CLIENT_MAX;j++) {
        struct i2c_client *client = adap->clients[j];
```

内层循环，遍历一条总线上的所有设备，即 i2c_adapter.clients[]数组的非空项。如果这个设备使用待删除的驱动，则调用驱动的 detach_client 函数从 i2c_adapter.clients[]数组中注销这个设备，并释放设备的私有数据结构：

```
        if (client != NULL && client->driver == driver) {
            DEB2(printk("i2c-core.o: detaching client %s:\n", client->name));
            if ((res = driver->detach_client(client)))
            {
                printk("i2c-core.o: while unregistering driver `%s', the client at address %02x of "
                        "adapter `%s' could not be detached; driver not unloaded!",
                        driver->name, client->addr, adap->name);
                ADAP_UNLOCK();
                return res;
            }
```

```
                    }
                }/*  内层循环  */
            }/* if –else */
        }/*  外层循环  */
        ADAP_UNLOCK();
        drivers[i] = NULL;
        driver_count--;
        DRV_UNLOCK();

        DEB(printk("i2c-core.o: driver unregistered: %s\n",driver->name));
        return 0;
}
```

函数的最后就是释放信号量，并从 drivers[]数组中注销设备驱动程序了。

ltc3445_detach_client 函数

该函数在卸载适配器或设备的驱动程序模块时被调用，用于从总线上注销设备，并释放设备的 i2c_client 数据结构和私有数据结构。

```
int ltc3445_detach_client(struct i2c_client *client)
{
        int err;
        /* Try to detach the client from i2c space */
        if ((err = i2c_detach_client(client))){
            printk("ltc3445.o: Client deregistration failed, client not detached.\n");
            return err;
        }
        /* free i2c_client and private data all together*/
        kfree(client);
        return 0;
}
```

与通过 i2c_attach_client 函数完成 i2c 适配器端的注册相对，i2c_detach_client 函数则完成 i2c 适配器端的注销。而 i2c 设备端的注销工作只需通过 kfree 释放 i2c_client 及其私有数据结构即可。

i2c_detach_client 函数

```
int i2c_detach_client(struct i2c_client *client)
{
    struct i2c_adapter *adapter = client->adapter;
    int i,res;
```

```
    for (i = 0; i < I2C_CLIENT_MAX; i++)
        if (client == adapter->clients[i])
            break;
    if (I2C_CLIENT_MAX == i) {
        printk(KERN_WARNING " i2c-core.o: unregister_client [%s] not found\n",
            client->name);
        return -ENODEV;
    }

    if( (client->flags & I2C_CLIENT_ALLOW_USE) && (client->usage_count>0))
        return -EBUSY;
```

如果相应的 i2c 设备仍被使用即其 usage_count 引用计数大于 0 则拒绝注销设备，并返回错误-EBUSY。usage_count 引用计数由 i2c_use_client 和 i2c_release_client 函数控制，在这两个函数中还分别调用 i2c_adapter 和 i2c_driver 中定义的 inc_use 和 dec_use 方法。但是在当前 pxa255 的 i2c 补丁中在访问 i2c 设备期间并没有使用 i2c_use_client 和 i2c_release_client 函数来控制 usage_count 计数。

```
    if (adapter->client_unregister != NULL)
        if ((res = adapter->client_unregister(client))) {
            printk(KERN_ERR "i2c-core.o: client_unregister [%s] failed, "
                    "client not detached",client->name);
            return res;
        }
```

进一步如果适配器驱动中实现了 client_register 方法，则调用之完成额外的注销工作（与在 i2c_attach_client 中调用适配器可能定义的 client_register 函数完成额外的注册工作相对）。在当前 pxa255 的 i2c 补丁中没有实现这个方法，所以适配器端的注销工作仅包括从 clients 指针数组中删除相应的指针并减少设备计数：

```
    adapter->clients[i] = NULL;
    adapter->client_count--;

    DEB(printk("i2c-core.o: client [%s] unregistered.\n",client->name));
    return 0;
}
```

第 5 章 与 pxa255 的 i2c 适配器相关的代码

i2c_pxa_reset 函数

该函数用于初始化 pxa255 的 i2c 适配器。

```
static void i2c_pxa_reset(void)
{
    /* abort any transfer currently under way */
    i2c_pxa_abort();
    /* reset according to 9.8 */
    ICR = ICR_UR;
    ISR = I2C_ISR_INIT;
    ICR &= ~ICR_UR;
```

根据 pxa255 开发者手册 9.8 节的内容，软件在重启 i2c 适配器前必须保证 i2c 总线是空闲的，而且在重启后必须保证 i2c 总线处于 idle 状态。首先通过 i2c_pxa_abort 函数放弃当前任何可能的 i2c 操作（从而保证 i2c 总线空闲），然后根据 9.8 节建议的三步，依次设置 i2c 适配器的控制寄存器的 ICR_UR 位并清除其余所有位，然后清除状态寄存器 ISR，最后清除 ICR_UR 位。清除状态寄存器 ISR 是通过向各个位写 1 完成的，宏 I2C_ISR_INIT 定义于 include/asm/arch/i2c.h。

```
    /* set control register values */
    ICR = I2C_ICR_INIT;

#ifdef CONFIG_I2C_PXA_FAST
    ICR |= ICR_FM;
#endif
    /* enable unit */
    ICR |= ICR_IUE;  // enables and defaults to slave-receive mode
    udelay(100);
}
```

重启 i2c 适配器后，将其控制寄存器初始化为 I2C_ICR_INIT，定义于 include/asm/arch/i2c.h：

```
#define I2C_ICR_INIT                    (ICR_BEIE | ICR_IRFIE | ICR_ITEIE | ICR_GCD | ICR_SCLE)
```

而宏 ICR_xxxx 定义于 include/asm/arch/pxa-regs.h，使得 i2c 适配器初始化后即使能了 Bus Error、Receive Full、Transmit Empty、General Call Detected 中断，并使能了 SCL 位（使得 i2c 适配器能够成为控制 i2c 总线的 master）。最后通过设置控制寄存器的 ICR_IUE 使能 i2c 适配器（处于缺省的 slave-receive 模式）。

i2c_pxa_abort 函数

该函数用于使 i2c 适配器放弃当前字节的操作。

```
#define PXA_ABORT_MA          /* Added by shrek2 */
static void i2c_pxa_abort(void)
{
    unsigned long timeout = jiffies + HZ/4;

#ifdef PXA_ABORT_MA
    while ((long)(timeout - jiffies) > 0 && (ICR & ICR_TB)) {
        msleep(1);
    }
    ICR |= ICR_MA;
    udelay(100);
```

如果宏 PXA_ABORT_MA 有效，则在指定间隔内（四分之一秒，即 250 毫秒内）等待当前传输操作的完成。根据 pxa 手册表 9-11 中关于 ICR_TB 位的介绍，ICR_TB 位在当前传输结束前一直有效，在收到应答传输结束后，由硬件清除 ICR_TB，并且硬件将 SCL 拉低，直到软件开始下一次字节传输时设置 ICR_TB 位。所以，如果当前操作没有完成，就等待 1 毫秒，直到结束或者超时。然后设置 ICR_MA 位，使得 pxa255 的 i2c 适配器处于 master 模式时发送 STOP。

```
#else
    while ((long)(timeout - jiffies) > 0 && (IBMR & 0x1) == 0) {
        i2c_pxa_transfer(1, I2C_RECEIVE, 1);
        msleep(1);
    }
#endif
    ICR &= ~(ICR_MA | ICR_START | ICR_STOP);
}
```

发送完 STOP 后，再清除 ICR_MA。
（问题：在原始补丁中怎么没有看到宏 PXA_ABORT_MA 的定义处？另外，如果没有定义该宏，那么 else 中采用方法的思路如何？）

i2c_pxa_xfer 函数

在安装了 i2c-dev 模块后用户就可以通过打开代表整条 i2c 总线的/dev/i2c/%d 设备节点来访问具体的 i2c 设备了：首先 open 该设备节点，然后通过 ioctl 设置相应设备的总线地址，进而就可以 read/write 设备节点了。read/write 系统调用最终会调用 i2c-core.c 文件中定义的 i2c_master_recv/send 方法，比如 i2c_master_recv 的片断如下：

```
    if (client->adapter->algo->master_xfer) {
```

```
        msg.addr = client->addr;
        msg.flags = client->flags & I2C_M_TEN;
        msg.flags |= I2C_M_RD;
        msg.len = count;
        msg.buf = buf;

        I2C_LOCK(adap);
        ret = adap->algo->master_xfer(adap,&msg,1);
        I2C_UNLOCK(adap);
        ……
    }
```

其中 client 指针指向在打开/dev/i2c/%d 设备节点时创建的临时 i2c_client 数据结构，其 addr 域保存有待访问设备的总线地址，adapter 域指向 i2c 总线适配器的 i2c_adapter 数据结构，而 i2c_adapter 数据结构的 algo 域又指向描述当前总线访问方法的 i2c_algorithm 数据结构。

由此可见，真正的 i2c 操作是在持有 i2c_adapter 的 lock 信号量的情况下通过 i2c_algorithm 的 master_xfer 方法实现的。对于 pxa255 正是上文提及的 i2c_pxa_algorithm 的 i2c_pxa_xfer 方法。

```
static int i2c_pxa_xfer(struct i2c_adapter *adap, struct i2c_msg msgs[], int num)
{
    struct pxa_i2c *i2c = adap->algo_data;
    int ret, i;

    for (i = adap->retries; i >= 0; i--) {
        ret = i2c_pxa_do_xfer(i2c, msgs, num);
        if (ret != I2C_RETRY)
            goto out;
        if (i2c_debug)
            pr_info("i2c-pxa: retrying transmission\n");
        udelay(100);
    }
    i2c_pxa_scream_blue_murder(i2c, "exhausted retries");
    ret = -EREMOTEIO;
out:
    return ret;
}
```

可见这个函数把具体操作交给 i2c_pxa_do_xfer 函数完成，参数 msgs 指向描述一组 i2c 操作的 i2c_msg 数据结构数组，操作的个数为 num。如果尝试全部失败则通过 i2c_pxa_scream_blue_murder 输出信息。

i2c_pxa_do_xfer 函数

该函数开启 i2c 操作，并将当前执行流阻塞在 pxa_i2c 数据结构的 wait 等待队列上。

```
static int i2c_pxa_do_xfer(struct pxa_i2c *i2c, struct i2c_msg *msg, int num)
{
    long timeout;
    int ret;

    spin_lock_irq(&i2c->lock);

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

    i2c_pxa_start_message(i2c);

    spin_unlock_irq(&i2c->lock);
```

由上文分析知，i2c_pxa 数据结构的核心成员为 i2c_adapter 数据结构，另外还有等待队列 wait、记录 i2c 操作的数组 msg 及其相关的其它域、以及保护整个数据结构的自旋锁 lock。设置 msg 和 msg_xxxx、调用 i2c_pxa_start_message 函数都必须放到由自旋锁操作实现的中断禁区中，从而与 i2c 中断处理程序同步（另见文档末尾的相关讨论）。

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

由上文 i2c_adap_pxa_init 函数完成 i2c 适配器的初始化、由 i2c_pxa_start_message 函数向 IDBR 中写入设备地址、操作性质，并启动了写入序列后，当前执行流就应该静待 Transmit Empty 中断的到来，故通过 wait_event_timeout 函数将自己挂起到 i2c_pxa.wait 等待队列上，直到等待的条件“i2c->msg_num == 0”为真，或者 5 秒钟超时后才被唤醒。注意一次 i2c 操作可能要涉及多个字节，只有第一个字节的发送是在当前进程的文件系统操作执行流中进行的，该字节操作的完成及后继字节的写入都由中断处理程序来完成。在此期间当前进程挂起在 i2c_pxa.wait 等待队列上。

```
    /*
     * We place the return code in i2c->msg_idx.
     */
    ret = i2c->msg_idx;
    if (timeout == 0)
        i2c_pxa_scream_blue_murder(i2c, "timeout");
    return ret;
}
```

i2c 操作的返回码由中断处理程序设置，保存在 i2c_pxa.msg_idx 域中并向上层函数返回（最终返回给发出文件系统调用的当前进程）。

i2c_pxa_start_message 函数

这个函数完成了 pxa255 开发者手册 9.6.2 的第 1、2 步的操作。

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

    IDBR = i2c_pxa_addr_byte(i2c->msg);

    icr = ICR & ~(ICR_STOP | ICR_ALDIE);
    ICR = icr | ICR_START | ICR_TB;
}
```

首先通过 i2c_pxa_addr_byte 函数把 msg 所指第一个 i2c_msg 结构中的设备地址、操作方式写入 i2c 适配器的数据缓冲区寄存器 IDBR：

```
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;
}
```

当 pxa255 的 i2c 适配器处于 master 模式时，由其发起 i2c 总线操作。无论是读、写设备，首先要将设备的 7 位总线地址及操作的性质写入 IDBR。IDBR[7:1]为 7 位设备地址，IDBR[0]为 master 的操作性质：如果 pxa255 的 i2c 适配器要读取指定设备，则将该位置 1，写入设备则清 0（而设备在检测到地址匹配时——类似于 pxa255 的 i2c 适配器的 Slave Address Detected 中断——将根据 R/nW 位的值来决定后继的收发状态）。

根据 pxa255 开发者手册 9.6.2 节的内容，在将设备地址和读写方式写入 IDBR 后，master-transmit 的第二步就是设置 ICR[START]、清除 ICR[STOP]、清除 ICR[ALDIE]、设置 ICR[TB]。其中 ICR[TB]位在 i2c 总线读写一个字节期间为 1，在读写完一个字节并发送、接收应答位 ACK/NAK 后由硬件自动清 0。此后硬件将 SCL 置为低，直到软件设置 ICR[TB]才开始驱动 SCL 线。

清除 ICR[ALDIE]禁止仲裁丢失检测中断（ALD）的原因正如手册 9.6.2 节第 5 步所述：当 i2c 控制器处于 master 发送 slave 设备的地址时，如果发生总线争用而发生 Arbitration Lost，禁止该中断可以使 i2c 适配器自动重新尝试发送设备地址。注意，只要 slave 设备地址发送结束就使能 ALD 中断。

i2c_pxa_handler 函数

当 pxa255 的 i2c 适配器处于 master 模式时，i2c 操作的第一步总是向 i2c 总线写入设备的地址及操作性质，这步在当前执行流的文件系统操作中最终由 i2c_pxa_do_xfer 函数完成。而收发的后继操作都是在 i2c 中断处理函数 i2c_pxa_handler 中完成，它在 i2c 适配器驱动程序模块的初始化函数 i2c_adap_pxa_init 中注册：

ret = request_irq(IRQ_I2C, **i2c_pxa_handler**, SA_INTERRUPT, "pxa2xx-i2c", **i2c**);

其中最后一个参数 void *dev_id 可用于指向设备私有数据结构，这里被传递的值为描述适配器驱动程序模块的 i2c_pxa 数据结构的地址，那么当 i2c 中断发生时该数据结构的地址就被传递给中断处理函数。

```
static irqreturn_t i2c_pxa_handler(int this_irq, void *dev_id, struct pt_regs *regs)
{
    struct pxa_i2c *i2c = dev_id;
    int status/*, wakeup = 0*/;
    status = (ISR);
    ……

    ISR = status & (ISR_SSD|ISR_ALD|ISR_ITE|ISR_IRF|ISR_SAD|ISR_BED);
```

该中断处理函数在所有使能的 i2c 中断发生时被操作系统调用。进入中断处理函数后，可以从 i2c 适配器的状态寄存器 ISR 中判断具体的中断源，并且软件应该清除相应位以清除中断（通过写 1 来完成）。否则相应中断将不断被触发（当发生中断的条件满足时 ISR 中的相应位就被设置。而且只要 ISR 中的标志位有效，则 i2c 适配器就不停地发出中断请求）。

```
    if (i2c->msg) {
        if (status & ISR_ITE)
            i2c_pxa_irq_txempty(i2c, status);
        if (status & ISR_IRF)
            i2c_pxa_irq_rxfull(i2c, status);
    } else {
        i2c_pxa_scream_blue_murder(i2c, "spurious irq");
    }
    return IRQ_HANDLED;
}
```

然后，通过检查中断发生时 ISR 中的标志位即可判断具体的中断源：Transmit Empty 或者 Receive Full，并相应调用 i2c_pxa_irq_txempty 或 i2c_pxa_irq_rxfull 函数进行处理。

i2c_pxa_irq_txempty 函数

```
static void i2c_pxa_irq_txempty(struct pxa_i2c *i2c, u32 isr)
{
```

```
u32 icr = ICR & ~(ICR_START|ICR_STOP|ICR_ACKNAK|ICR_TB);
```

pxa255 通过内部总线向 i2c 适配器的 IDBR 中写入待发送的字节，然后设置 ICR[TB]位恢复 SCL 输出，在每一个 SCL 的上升沿 IDBR 中的数据通过串行移位寄存器发送到 SDA 线上。在一个字节被发送完**并收到对方的应答位后** i2c 适配器发出 Transmit Empty 中断并自动清除 ICR[TB]使得 SCL 线一直置低，直到软件再次写入 IDBR 并由软件设置 ICR[TB]（参见手册 9.4.2 节）。

由手册 9.3.3 节 table 9-4 的内容可知，当收发数据时 ICR[START]和 ICR[STOP]位都应该被清 0。清除 ICR[ACKNAK]位使得在接收到一个字节后回送 ACK 应答（而不是 NAK 应答，该应答只有在收到最后一个字节后设置，所以 ICR[ACKNAK]位在准备接收最后一个字节前设置）。

```
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 handle them as I think
         * they should, we end up causing an I2C error, which
         * is painful for some systems.
         */
        return; /* ignore */
    }
```

根据手册当 i2c 适配器处于 master-transmit 模式时，在发送 slave 地址期间应该禁止 ALD 中断，使得当 ALD 发生时硬件可以在 i2c 总线空闲时自动重新尝试发送 slave 地址。然而在发送（或接收）后继数据期间应该重新使能 ALD 中断。根据作者的注释当发生 ALD 时什么也不做，而直接退出。
（问题：为什么把处理 ALD 中断的代码放到 i2c_pxa_irq_txempty 中？只有当 ALD 中断发生时同时发生 TE 中断这样做才有意义。）

```
    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);
```

当 master 发送完一个字节后立即释放 SDA 线（使 SDA 为高电平），期望在后继 SCL 为高时能够收到 slave 的 ACK 将 SDA 拉为低电平。如果没有收到 ACK 则引发 BED 中断。
（问题：如果发送完一个字节而没有收到 ACK，则肯定触发 BED 中断。那么此时 Transmit Empty 中断是否触发？这个源代码认为此时这两个中断都会发生，所以才把 BED 中断的处理放到了 i2c_pxa_irq_txempty 函数中。）

根据代码作者的注释，如果由没有收到 ACK 引起的 BED 中断发生在进行第一次交互发送 slave 的地址时，则可以返回错误码 I2C_RETRY，使得系统调用执行流可以重新尝试发起操作。回想在 i2c_pxa_xfer 函数中有如下循环：

```
    for (i = adap->retries; i >= 0; i--) {
        ret = i2c_pxa_do_xfer(i2c, msgs, num);
        if (ret != I2C_RETRY)
            goto out;
        if (i2c_debug)
            pr_info("i2c-pxa: retrying transmission\n");
        udelay(100);
    }
```

即如果操作的返回码为 I2C_RETRY，则可以最多尝试 retries 次。

无论 BED 中断是否由没有收到 ACK 引起，在中断处理程序中都要通过 i2c_pxa_master_complete 函数唤醒在 i2c_pxa.wait 上阻塞的用户进程：

```
static inline 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;
    wake_up(&i2c->wait);
}
```

在 i2c_pxa_do_xfer 函数中启动了传送后，当前进程就由 wait_event_timeout 函数阻塞到 i2c_pxa 的 wait 等

待队列上了。当发生 BED 错误后，在 i2c_pxa 的 msg_idx 中保存错误码，并用 wake_up 函数唤醒阻塞的执行流，使得用户进程能够沿着如下函数调用过程：

sys_write > i2cdev_write > i2c_master_send > i2c_pxa_xfer > i2c_pxa_do_xfer

反方向逐层返回。

在处理了 ALD 和 BED 中断后，我们接着分析 i2c_pxa_irq_txempty 的代码：

```
} 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;
```

无论处于 master-transmit 还是 master-receive 模式，只要处于 master 模式，则在发送 START 后首先要处于 master-transmit 模式发送 slave 设备的 addr。如果要从设备读，则由 master-transmit 模式切换到 master-receive 模式。如何判断是否从设备读呢？ISR_RWM 正好保存了 slave 地址的最低 R/nW 位。如果 ISR_RWM 为 0，则表示写入设备，则由后继代码处理。如果为 1，则表示本次 TE 中断由发送 slave 地址结束引起，那么就应该设置 ICR_TB 启动后继读过程了。

如果 if 中的两个条件都为真，则表示我们要读的字节为最后一个交互的最末字节（其实这里就是只需要读取一个字节的情况。因为如果需要读取多个字节，那么读取最后一个字节的准备工作是在 i2c_pxa_irq_rxfull 函数中完成的），读完这个字节后 master 就必须向 slave 回应 NAK 应答，使得 slave 停止发送字节，并向 slave 发送 STOP。根据手册 9.4.3 节的内容，在 master 准备读取最后一个字节前，必须首先设置 ICR_ACKNAK 位。

一言以蔽之，正如手册 9.3.3.3 节中所述，处在 master-receive 模式读取最后一个字节前，应该设置 ICR 的 ACKNAK、STOP 和 TB 位，以启动读取最后一个字节的操作。

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

    icr |= ICR_ALDIE | ICR_TB;

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

如上文介绍 i2c_pxa 数据结构时所述 msg 数组指向“Repeated Start”情形中的一组交互，msg_num 为交互的数量，msg_idx 为当前交互在数组中的偏移。每完成一个交互 msg 指针就指向数组中的下一个元素，msg_idx 就增 1。msg_ptr 指向当前交互中下一个要接收、发送的字节在 i2c_msg,bus 缓冲区中的偏移。i2c_msg.len 为缓冲区的长度。

如果 i2c->msg_ptr 小于 i2c->msg->len 即当前交互尚未完成，那么就把 i2c->msg->buf[i2c->msg_ptr]字节写入 IDBR，同时步进 msg_ptr。然后设置 ICR[TB]启动发送操作。另外，如果待发送的是最后一个交互的最末字节，则设置 ICR_STOP 使得在发送完最后一个字节后发送 STOP。

如果上一个 if 判断的条件不满足，即 i2c->msg_ptr 不小于 i2c->msg->len，则说明当前交互已完成。此时如果还有其它交互，那么就应该启动它们：

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

由此可见，msg 总是指向当前交互、msg_idx 为当前交互在数组中的偏移，都是在这里实现的。当开始新交互时，将 msg_ptr 清 0，即从新交互的数据缓冲区的首址开始。

```
    /*
     * 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;
```

从一次 i2c 操作的第二个交互开始，如果是操作同一 slave 设备，则无需重新发送 slave 地址，此时直接跳转到 again 标号出执行（实际上跳转到上一个 if 处开始执行）。如果新的交互涉及不同的 slave 设备，则应该重新发送 slave 地址。

（问题：虽然作者说这里不支持从 master-transmit 到 master-receive 的切换，但是，只要不指定 I2C_M_NOSTART 标志，那么下面将用 i2c_pxa_addr_byte 函数重新发送设备地址，使得 ICR_RWM 位被更新，那么在上面判断 isr&ISR_RWM 中就可以切换到 master-receive 模式了。）

若没有指定 I2C_M_NOSTART 标志，那么用 i2c_pxa_addr_byte 函数重新发送设备地址：

```
    /*
     * Write the next address.
     */
    IDBR = i2c_pxa_addr_byte(i2c->msg);
```

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

        } else {
            if (i2c->msg->len == 0) {
                /*
                 * Device probe's have a message length of zero and need
                 * the bus to be reset before it can be used again.
                 */
                pr_debug("i2c-pxa: resetting unit");
                i2c_pxa_reset();
            }
            i2c_pxa_master_complete(i2c, 0);
        }
```

最后的 else 分支用于设备地址检测。详见上文 i2c_smbus_xfer_emulated 函数分析的相关部分。

```
    i2c->icrlog[i2c->irqlogidx-1] = icr;
    ICR = icr;
    show_state();
}
```

i2c_pxa_irq_rxfull 函数

在看懂 i2c_pxa_irq_txempty 的代码后，i2c_pxa_irq_rxfull 的代码就很简单了：

```
static void i2c_pxa_irq_rxfull(struct pxa_i2c *i2c, u32 isr)
{
    u32 icr = ICR & ~(ICR_START|ICR_STOP|ICR_ACKNAK|ICR_TB);

    /*
     * Read the byte.
     */
    i2c->msg->buf[i2c->msg_ptr++] = IDBR;

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

```
            icr |= ICR_ALDIE | ICR_TB;
        } else {
            i2c_pxa_master_complete(i2c, 0);
        }

        i2c->icrlog[i2c->irqlogidx-1] = icr;
        ICR = icr;
}
```

当 Receive Full 中断发生时，直接从 IDBR 中读取接收的数据，保存到 i2c->msg->buf 的 i2c->msg_ptr 偏移处，然后设置 ICR[TB]启动后继字节的读取。如果即将读取最后一个字节，则同时设置 ICR[ACKNAK]使得 i2c 适配器在接收字节后回应 NAK，使得 slave 设备停止发送，并通过设置 ICR[STOP]发送 STOP。

如果接收完成，则通过 i2c_pxa_master_complete 唤醒等待的用户进程，返回的错误码为 0，即没有错误。

另外需要说明的是，在进行完当前 i2c 读操作后就立刻唤醒阻塞的当前进程了。也就是说，代码认为一个读操作只用一个 i2c_msg 数据结构进行描述，而不会拆分为多个。所以，进行读操作时 i2c_pxa.msg 所指 i2c_msg 数组中就只有一个元素。

第 6 章 i2c-dev 的初始化

i2c-dev 模块为系统中所有的 i2c 适配器创建相应的/dev/i2c/%d 字符设备节点，并注册设备访问方法，从而使得用户进程可以访问该 i2c 总线上的设备。

i2c_dev_init 函数

该函数为 i2c-dev 模块的初始化函数。

```
static int __init i2c_dev_init(void)
{
    int res;
    printk(KERN_DEBUG "i2c-dev.o: i2c /dev entries driver module\n");
    i2cdev_initialized = 0;
```

i2c-dev 模块的初始化函数为 i2c_dev_init。模块内静态变量 i2cdev_initilized 用于记录该模块初始化过程的完成进度。由下文可见，当其值为 1 时表示已创建/dev/i2c 目录并注册 i2cdev_fops 方法，当其值为 2 时表示已经为系统上所有已安装的 i2c 适配器创建了/dev/i2c/%d 设备节点。如果模块的初始化过程失败，那么在清理函数中就可以根据 i2cdev_initialized 的值执行相应的清理操作。

```
#ifdef CONFIG_DEVFS_FS
    if (devfs_register_chrdev(I2C_MAJOR, "i2c", &i2cdev_fops)) {
#else
    if (register_chrdev(I2C_MAJOR,"i2c",&i2cdev_fops)) {
#endif
        printk("i2c-dev.o: unable to get major %d for i2c bus\n",
               I2C_MAJOR);
        return -EIO;
    }
#ifdef CONFIG_DEVFS_FS
    devfs_handle = devfs_mk_dir(NULL, "i2c", NULL);
#endif
    i2cdev_initialized ++;
```

初始化工作的第一步就是通过 devfs_register_chrdev 函数向 devfs 注册设备方法 i2cdev_fops，并用 devfs_mk_dir 创建目录/dev/i2c。将来在用户进程打开/dev/i2c/%d 设备节点，对该设备文件进行文件系统操作时，最终将通过 i2cdev_fops 方法完成。

```
    if ((res = i2c_add_driver(&i2cdev_driver))) {
        printk("i2c-dev.o: Driver registration failed, module not inserted.\n");
        i2cdev_cleanup();
```

```
            return res;
        }
        i2cdev_initialized ++;
        return 0;
}
```

注册过程的第二步就是通过 i2c_add_driver 函数注册 i2cdev_driver 数据结构，其作用在于为系统中所有已安装的 i2c 适配器调用 i2cdev_driver 的 attach_adpter 方法，即 i2cdev_attach_adapter 函数，为所有已安装的适配器创建相应的/dev/i2c/%d 字符设备结点并注册设备访问方法。

i2cdev_attach_adapter 函数

由前文所知 i2c_add_driver 函数会针对每一个已安装的适配器调用当前 i2c_driver 的 attach_adapter 方法，在安装 i2c-dev 模块时该方法就是 i2cdev_attach_adapter 函数。

```
int i2cdev_attach_adapter(struct i2c_adapter *adap)
{
        int i;
        char name[8];
        if ((i = i2c_adapter_id(adap)) < 0) {
                printk("i2c-dev.o: Unknown adapter ?!?\n");
                return -ENODEV;
        }
        if (i >= I2CDEV_ADAPS_MAX) {
                printk("i2c-dev.o: Adapter number too large?!? (%d)\n",i);
                return -ENODEV;
        }
        sprintf (name, "%d", i);
        if (! i2cdev_adaps[i]) {
                i2cdev_adaps[i] = adap;
```

由前文知，i2c-core.c 中定义的内核静态指针数组 adapter 记录了所有已安装的适配器的 i2c_adapter 数据结构。而 i2c-dev.c 也定义了一个类似的静态指针数组 i2cdev_adaps：

```
static struct i2c_adapter *i2cdev_adaps[I2CDEV_ADAPS_MAX];
```

所以首先就是根据当前适配器的 i2c_adapter 数据结构在 adapters 数组中的偏移，将 i2cdev_adaps 数组相同偏移上也指向该适配器的 i2c_adapter 数据结构。

```
#ifdef CONFIG_DEVFS_FS
            devfs_i2c[i] = devfs_register (devfs_handle, name, DEVFS_FL_DEFAULT, I2C_MAJOR, i,
                S_IFCHR | S_IRUSR | S_IWUSR, &i2cdev_fops, NULL);
#endif
            printk("i2c-dev.o: Registered '%s' as minor %d\n",adap->name,i);
```

然后，就是在/dev/i2c 目录下创建相应的字符设备节点/dev/i2c/%d 了，数值%d 正是该适配器的 i2c_adapter 数据结构在 adapters/i2cdev_adaps 数组中的偏移。另外注册的设备访问方法为 i2cdev_fops 函数。（再次重申，一条 i2c 总线包括适配器及总线上的所有设备，只需要用一个设备节点/dev/i2c/%d 表示）

```
        } else {
                /* This is actually a detach_adapter call! */
#ifdef CONFIG_DEVFS_FS
                devfs_unregister(devfs_i2c[i]);
#endif
                i2cdev_adaps[i] = NULL;
#ifdef DEBUG
                printk("i2c-dev.o: Adapter unregistered: %s\n",adap->name);
#endif
        }
        return 0;
}
```

如果 i2cdev_adaps 数组中相应偏移上的指针不为空，则说明该函数被用来注销设备节点。

第 7 章 i2c 框架提供的设备访问方法

i2cdev_open 函数

模块 i2c-dev 安装后，就已经为系统上所有已安装的适配器创建了/dev/i2c/%d 设备节点了，并定义设备的访问方法由 i2cdev_fops 方法表提供。然后用户进程就可以 open 该设备节点创建文件对象，并对其进行文件系统操作了。

在用户进程打开 i2c 适配器的设备节点时 sys_open 将创建相关的 VFS 对象，并根据主号找到已注册的设备访问方法表，然后将该方法表的地址赋予文件对象的 f_op 指针。最后调用设备方法表中的 i2cdev_open 函数，它创建一个临时的 i2c_client 数据结构，并建立与 i2c 适配器 i2c_adapter 的单向联系：

```
int i2cdev_open (struct inode *inode, struct file *file)
{
    unsigned int minor = MINOR(inode->i_rdev);
    struct i2c_client *client;
    if ((minor >= I2CDEV_ADAPS_MAX) || ! (i2cdev_adaps[minor])) {
#ifdef DEBUG
        printk("i2c-dev.o: Trying to open unattached adapter i2c-%d\n", minor);
#endif
        return -ENODEV;
    }
```

首先进行基本的参数检查。设备的次号%d 正是该适配器的 i2c_adapter 数据结构在 adapters/i2cdev_adaps 数组中的偏移。所以次号必须有效，而且 i2cdev_adaps[minor]不应为空。

```
    /* Note that we here allocate a client for later use, but we will *not*
       register this client! Yes, this is safe. No, it is not very clean. */
    if(! (client = kmalloc(sizeof(struct i2c_client), GFP_KERNEL)))
        return -ENOMEM;
    memcpy(client,&i2cdev_client_template,sizeof(struct i2c_client));
    client->adapter = i2cdev_adaps[minor];
    file->private_data = client;
```

在打开 i2c 适配器的设备节点时创建一个临时的 i2c_client 数据结构，并用 i2cdev_client_template 初始化。然后将文件对象的 private_data 指向这个临时的 i2c_client。以后 i2cdev_fops 中的方法就可以根据 file 找到这个临时的 i2c_client，再由其 adapter 域找到适配器的 i2c_adapter 数据结构了，从而进一步通过 i2c_adapter 的 clients 指针数组访问该 i2c 总线上所有设备的 i2c_client 数据结构（参见数据结构之间的关系图）。

创建这个临时的 i2c_client 数据结构是为了保存设备地址、建立与 i2c 总线的 i2c_adapter 的联系（从而进一步获得其总线通信方法）、以及建立与 i2c 设备访问方法的联系。之所以称这个 i2c_client 数据结构为“临

时的”是因为它只在打开 i2c 设备结点期间存在，而不是真正存在于该 i2c 总线上的设备，所以才没有被进一步注册到 i2c_adapter 的 clients 指针数组中，因为设备操作只需要从该临时 i2c_client 到 i2c_adapter 的单向联系即可。另外如果用户进程需要访问一条总线上的多个设备，则只需打开适配器设备节点一次，然后用 ioctl 将该临时 i2c_client 的 addr 域设置为不同设备的地址即可。

```
if (client->adapter->inc_use)
        client->adapter->inc_use(client->adapter);
#if LINUX_KERNEL_VERSION < KERNEL_VERSION(2,4,0)
    MOD_INC_USE_COUNT;
#endif /* LINUX_KERNEL_VERSION < KERNEL_VERSION(2,4,0) */
```

最后，调用适配器的 inc_use 方法增加适配器驱动模块的引用计数（但在当前 pxa255 的 i2c 补丁中该方法为空）。

```
#ifdef DEBUG
    printk("i2c-dev.o: opened i2c-%d\n",minor);
#endif
    return 0;
}
```

i2cdev_ioctl 函数

由于 i2c 适配器的设备节点代表的是整条 i2c 总线，所以在对其进行具体的文件系统操作之前还必须指明待访问设备的总线地址。指明地址的操作通过 ioctl 系统调用完成的，它最终调用设备方法 i2cdev_ioctl。

用户进程可以通过 ioctl 系统调用发出 I2C_TMP_DEVICE_ADDR 命令来设置打开适配器设备结点时创建的临时 i2c_client 数据结构中 addr 域的值：（参见内核文档 Documentation/i2c/dev-interface）

```
    int addr = 0x4b;            /* The I2C address */
    if (ioctl(file,I2C_TMP_DEVICE_ADDR,addr) < 0) {
        err_sys("ioctl error\n");
    }
```

需要说明的是，在 Documentation/i2c/dev-interface 一文中的通过 I2C_SLAVE 命令来指明待访问的设备地址的说法是不正确的，而必须增加一个额外的命令用于指明设备地址：（增加的部分如加粗所示）

```
    switch ( cmd ) {
    case I2C_TMP_DEVICE_ADDR:
        /* Make sure the addr is used by one device */
        if (-EBUSY == i2c_check_addr(client->adapter,arg)){
#ifdef DEBUG
            printk(KERN_INFO "set tmp i2c_client.addr to 0x%2x\n", arg);
#endif
            client->addr = arg;
```

```
            return 0;
        }else{
#ifdef DEBUG
            printk(KERN_INFO "No device on adapter(%s) has the addr of 0x%2x\n",
                                client->adapter->name, arg);
#endif
            return -EINVAL;
        }

    case I2C_SLAVE:
    case I2C_SLAVE_FORCE:
        if ((arg > 0x3ff) ||
            (((client->flags & I2C_M_TEN) == 0) && arg > 0x7f))
            return -EINVAL;
        if ((cmd == I2C_SLAVE) && i2c_check_addr(client->adapter,arg))
            return -EBUSY;
        client->addr = arg;
        return 0;
```

由此可见 I2C_SLAVE 命令语义是改变已有设备的地址，所以必须通过 i2c_check_addr 函数检查由 addr 指明的新地址是否与其所在 i2c 总线上的设备地址相冲突。如果冲突则返回-EBUSY，表明该地址已经被占用。而与我们指定设备地址的逻辑正好相反，我们这里需要将临时 i2c_client 的 addr 域设置为待访问设备的地址，因此在新增加的部分中只有当 i2c_check_addr 返回-EBUSY 时才说明指定的地址是有效的，才能设置临时 i2c_client 的 addr 域。

i2cdev_read 函数

该函数用于读取 i2c 设备。

```
static ssize_t i2cdev_read (struct file *file, char *buf, size_t count, loff_t *offset)
{
    char *tmp;
    int ret;
#ifdef DEBUG
    struct inode *inode = file->f_dentry->d_inode;
#endif /* DEBUG */

    struct i2c_client *client = (struct i2c_client *)file->private_data;
```

如 i2cdev_open 分析所述，在打开适配器设备节点时相应 file 对象的 private_data 域指向创建的那个临时的 i2c_client 数据结构，而其 adapter 域指向适配器的 i2c_adapter 数据结构。

```
    /* copy user space data to kernel space. */
    tmp = kmalloc(count,GFP_KERNEL);
```

```
    if (tmp==NULL)
        return -ENOMEM;
#ifdef DEBUG
    printk("i2c-dev.o: i2c-%d reading %d bytes.\n",MINOR(inode->i_rdev),
            count);
#endif

    ret = i2c_master_recv(client, tmp, count);
    if (ret >= 0)
        ret = copy_to_user(buf, tmp, count)? –EFAULT : ret;
    kfree(tmp);
    return ret;
}
```

然后，就是创建一个大小为 count 的临时缓冲区 tmp，并调用 i2c-core 中定义的 i2c_master_recv 方法从设备中读取 count 个字节，并通过 copy_to_user 拷贝回进程地址空间。

i2c_master_recv 函数

该函数通过调用 i2c 适配器驱动提供的 master_xfer 方法执行 i2c 读操作：

```
int i2c_master_recv(struct i2c_client *client, char *buf ,int count)
{
    struct i2c_adapter *adap=client->adapter;
    struct i2c_msg msg;
    int ret;
    if (client->adapter->algo->master_xfer) {
        msg.addr   = client->addr;
        msg.flags = client->flags & I2C_M_TEN;
        msg.flags |= I2C_M_RD;
        msg.len = count;
        msg.buf = buf;
```

从临时的 i2c_client 的 adapter 指针即可得到 i2c 适配器的 i2c_adapter 数据结构，另外其 addr 域在用户进程中在发出 read 系统调用之前实现已经通过 ioctl 指定为待访问的设备地址 addr 了（而用户可以从/proc/bus/i2c 和/proc/bus/i2c-%d 得到具体 i2c 总线上特定设备的地址）

i2c_msg 数据结构用于描述一次 i2c 交互，根据临时的 i2c_client.addr 设置其 addr 域。对于读操作设置 flags 中的 I2C_M_RD 标志，然后将 buf 指向先前在 i2cdev_read 中分配的内核缓冲区，len 为其长度。

```
        DEB2(printk("i2c-core.o: master_recv: reading %d bytes on %s.\n",
             count,client->adapter->name));

        I2C_LOCK(adap);
```

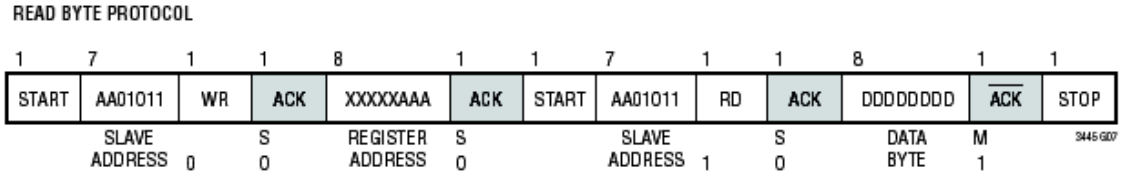
```
        ret = adap->algo->master_xfer(adap, &msg, 1);
        I2C_UNLOCK(adap);

        DEB2(printk("i2c-core.o: master_recv: return:%d (count:%d, addr:0x%02x)\n",
             ret, count, client->addr));

        /* if everything went ok (i.e. 1 msg transmitted), return #bytes
        * transmitted, else error code.
        */
        return (ret == 1 )? count : ret;
    } else {
        printk("i2c-core.o: I2C adapter %04x: I2C level transfers not supported\n",
                client->adapter->id);
        return -ENOSYS;
    }
}
```

接下来就是通过 i2c 适配器驱动实现的 i2c 总线通信方法，即 i2c_algorithm.master_xfer 函数执行实际的 i2c 操作。第一个参数为 i2c_adapter 数据结构的地址，由于一次 i2c 操作可能设计多个交互，所以第二个参数传递 i2c_msg 数据结构数组的地址，第三个参数指明 i2c_msg 数据结构的个数。

另外需要说明的是，在调用适配器驱动程序 master_xfer 开始 i2c 操作前必须首先通过 I2C_LOCK 宏获得当前适配器的 i2c_adapter 数据结构的 lock 信号量，在操作完成后才释放信号量。而当前进程在 i2c 操作期间是要被阻塞的，也就是说**当前进程在阻塞期间是一直持有该信号量的**，从而实现对 i2c 总线访问的同步。

对 i2cdev_read 和 i2c_master_recv 的修改

根据 ltc3445 的手册，读一个寄存器的值需要通过 2 次 i2c 交互，使用 Repeated Start 来实现：



其中第一次交互 pxa255 的 i2c 适配器为 master-transmit 模式，写入的数据为待读出的寄存器编号，写入长度为 1 个字节。第二次交互 pxa255 的 i2c 适配器为 master-receive 模式，读出相应寄存器的值。

由此可见一次 ltc3445 的读操作需要两次交互才能完成：第一次为写操作，写入待读出的寄存器编号；第二次为读操作，读出相应寄存器的值。所以在 i2c_master_recv 函数中必须使用两个 i2c_msg 数据结构，而且必须得到用户态指定的待读出寄存器的编号。

由此可以设计用户进程读取 ltc3445 的过程如下：

1. 用户进程打开/dev/i2c/%d 设备结点，并用 ioctl 指明待访问的设备的总线地址；
2. 用户进程在局部变量 buf 中保存寄存器的值，但首先将 buf 设置为待读出寄存器的编号。即 buf 为“输入－输出”参数；
3. i2c 的设备访问方法由 i2c-dev 模块的 i2cdev-fops 方法表提供。其中的 i2cdev_read 函数应该首先从用户态的 buf 中拷贝待读出寄存器的编号，然后再调用 i2c_master_recv 函数；
4. 在 i2c_master_recv 函数中使用两个 i2c_msg 数据结构。第一次为写操作，写入的数据即为用户态指定的待读寄存器编号；第二个为读操作，从设备中读出指定寄存器的值

需要说明的是，由于 ltc3445 上只有 5 个长度为 8 位的寄存器，所以读操作只需使用一个 unsigned char 类型变量来指明寄存器编号并保存结果。如果一次读操作可以读出一组字节流，则需要足够大的字节数组来保存结果。

综上，对 i2cdev_read 函数的修改如下：（增加的部分以加粗表示）

```
static ssize_t i2cdev_read (struct file *file, char *buf, size_t count, loff_t *offset)
{
    char *tmp;
    int ret;
#ifdef DEBUG
    struct inode *inode = file->f_dentry->d_inode;
#endif /* DEBUG */

    struct i2c_client *client = (struct i2c_client *)file->private_data;

    if (count > 8192)
        count = 8192;
    /* copy user space data to kernel space. */
    tmp = kmalloc(count,GFP_KERNEL);
    if (tmp==NULL)
        return -ENOMEM;

    if (copy_from_user(tmp, buf, 1))        /*  从用户态的局部变量中得到一个字节的寄存器编号  */
        return –EFAULT;

#ifdef DEBUG
    printk(KERN_DEBUG "i2c-dev.o: i2c-%d reading %d bytes.\n",minor(inode->i_rdev),
            count);
#endif
    ret = i2c_master_recv(client,tmp,count);
    if (ret >= 0)
        ret = copy_to_user(buf,tmp,count)?-EFAULT:ret;
    kfree(tmp);
    return ret;
```

```
}
```

而对 i2c_master_recv 函数的修改如下：（增加的部分以加粗表示，删掉的部分放到#if 0 #endif 中）

```
int i2c_master_recv(struct i2c_client *client, char *buf ,int count)
{
    struct i2c_adapter *adap=client->adapter;
#if 0
    struct i2c_msg msg;
#endif
    struct i2c_msg msg[2];
    int ret;
    if (client->adapter->algo->master_xfer) {
        msg[0].addr     = client->addr;
        msg[0].flags = client->flags & I2C_M_TEN;
        msg[0].len = 1;                 /*  只需一个字节指明待读出寄存器的编号  */
        msg[0].buf = buf;               /*  这里为在 i2cdev_read 中从用户态得到的待读出的寄存器编号  */

        msg[1].addr     = client->addr;
        msg[1].flags = client->flags & I2C_M_TEN;
        msg[1].flags |= I2C_M_RD;
        msg[1].len = count;             /*  使用用户态指定的待读出的字节量  */
        msg[1].buf = buf;

#if 0
        msg.addr    = client->addr;
        msg.flags   = client->flags & I2C_M_TEN;
        msg.flags |= I2C_M_RD;
        msg.len = count;
        msg.buf = buf;
#endif
        down(&adap->bus);
#if 0
        ret = adap->algo->master_xfer(adap,&msg,1);
#endif
        ret = adap->algo->master_xfer(adap,&msg,2);
        up(&adap->bus);

        DEB(printk(KERN_DEBUG "i2c-core.o: master_recv: return:%d (count:%d, addr:0x%02x)\n",
            ret, count, client->addr));

        /* if everything went ok (i.e. 1 msg transmitted), return #bytes
        * transmitted, else error code.
        */
```

```
#if 0
        return (ret == 1) ? count : ret;
#endif
        return (ret == 2) ? count : ret;
    } else {
        DEB(printk(KERN_ERR "i2c-core.o: I2C adapter %04x: I2C level transfers not supported\n",
                client->adapter->id));
        return -ENOSYS;
    }
}
```

其中 msg[0].flags 没有设置 I2C_M_RD 标志，表示这是一次写操作，待写入的数据即为 buf 中的内容。从上面对 i2cdev_read 函数的修改可见，buf 中是从用户空间读取的待访问的寄存器的编号。msg[1].flags 设置了 I2C_M_RD 标志，表示这是一次读操作，而且**没有**设置 I2C_M_NOSTART 标志，则从源代码分析可见没有设置该标志就可实现 Repeated Start，即在后继交互时重新发送 START。

另外，第一个 i2c_msg.len 一定要设置为 1，第二个 i2c_msg.len 设置为 count。这是因为第一次 i2c 交互写入待读出的寄存器编号，而寄存器编号使用一个字节足矣！而第二次 i2c 交互读出的数据量由用户进程指定，与具体的 i2c 设备相关。

值得说明的是，必须清醒地意识到如此这般根据具体的 i2c 设备对 i2cdev_read 函数和 i2c_master_recv 函数进行的修改，的确对于具体应用而言是必须的。**但是这样同时也损害了 i2c 框架代码的可移植性，因为这样使得这两个函数的代码与具体的设备相关了！**（进一步的讨论另见文档末尾的相关部分）

i2cdev_release 函数

这个函数在关闭设备节点时被调用：释放临时的 i2c_client 数据结构，并减小适配器驱动模块的引用计数。

```
static int i2cdev_release (struct inode *inode, struct file *file)
{
    unsigned int minor = MINOR(inode->i_rdev);
    kfree(file->private_data);
    file->private_data=NULL;

#ifdef DEBUG
    printk("i2c-dev.o: Closed: i2c-%d\n", minor);
#endif

#if LINUX_KERNEL_VERSION < KERNEL_VERSION(2,4,0)
    MOD_DEC_USE_COUNT;
#else
    lock_kernel();
#endif
```

```
        if (i2cdev_adaps[minor]->dec_use)
            i2cdev_adaps[minor]->dec_use(i2cdev_adaps[minor]);

#if LINUX_KERNEL_VERSION >= KERNEL_VERSION(2,4,0)
        unlock_kernel();
#endif
        return 0;
}
```

第 8 章 编写 i2c 设备驱动程序模块的方法

（题外话）在编写任何设备驱动程序代码之前都必须首先明确一点：为了不弄脏内核名字空间，该设备驱动程序模块代码中的所有导出的、不导出的符号都应该冠以相同的、与该驱动模块相关的前缀。比如，在 ltc3445 设备的驱动程序模块中，所有函数（无论导出与否）和全局变量都以“ltc3445_”开头。另外，对于所有无需导出内核的函数或全局量，都要用“static”声明。

1. ltc3445 设备驱动程序模块的实现代码定义在 ltc3445.c 文件中，它提供的设备驱动由一个 ltc3445_driver 数据结构描述：

```
static struct i2c_driver ltc3445_driver
{
    .name           = "LTC3445 I2C driver 0.1",
    .id             = I2C_DRIVERID_LTC3445,
    .flags          = I2C_DF_NOTIFY,
    .attach_adapter = ltc3445_attach_adapter,
    .detach_client  = ltc3445_detach_client,
    .command        = ltc3445_command,    /* May be NULL */
    .inc_use        = ltc3445_inc_use,    /* May be NULL */
    .dec_use        = ltc3445_dec_use     /* May be NULL */
};
```

其中设备的 id I2C_DRIVERID_LTC3445 定义于 ltc3445.h 头文件中，i2c_driver 数据结构中的函数也放到该头文件中声明。

2. 在 ltc3445.h 头文件中定义可能需要的私有数据结构 ltc3445_data：

```
/* private data owned by ltc3445, see writing-clients for more hints
*/
struct ltc3445_data {
    spinlock_t update_lock;          /* protect the whole data structure from race condition */
    unsigned char initialized;       /* != 0 if the following fields are valid. */
    unsigned long last_updated;      /* In jiffies */
    /* Add the read information here too */
    unsigned char regs[LTC3445_REG_NUM];
};
```

除了用 i2c_client 数据结构描述设备外，还可以设计私有数据结构 ltc3445_data 来缓存从 i2c 设备读取的结果。在 ltc3445 上有 5 个 8 位的控制/状态寄存器，对于那些只由软件设置的控制寄存器，可以将上一次设置的结果保存在 regs 数组中。这样当 i2c 适配器在开始 i2c 写操作前，可以将本次待写入的控制字与上一次写入设备的控制字相比较，只有在二者不同时才开始真正的 i2c 通信（当然这一点并不是必须的，只有在需要频繁地进行 i2c 操作时缓存数据才可能有意义。另外私有数据结构的使用可能损害 i2c 框架代码的可移植性和重用性，可参见文档末尾的相关讨论及 Linux 内核文档 Documentation/i2c/writing-clients）。

3. 在 ltc3445.h 头文件中还要定义使用该驱动程序的所有设备的可能使用的地址范围。在当前应用中在 pxa255 的 i2c 总线上只有 ltc3445 一个设备。而 ltc3445 可能采用的 i2c 地址为 0x4B 或者 0x2B：

```
#define LTC3445_ADDRESS1     0x4B      /* 100,1011 */
#define LTC3445_ADDRESS2     0x2B      /* 010,1011 */

/* probing device
 * The possible i2c address is LTC3445_ADDRESS1 or LTC3445_ADDRESS2
 */
static unsigned short normal_i2c[] = {LTC3445_ADDRESS1,
                                      LTC3445_ADDRESS2,
                                      I2C_CLIENT_END };
/* Magic definition of all other variables and things */
I2C_CLIENT_INSMOD;
```

其中 normal_i2c 数组指明设备可能使用的地址。另外还可以指定 normal_i2c_range[]、probe[]、probe_range、ignore[]、ignore_range、force[]等数组，其含义详见“writing-clients”文档。然后，由 I2C_CLIENT_INSMOD 宏声明该模块导出的与地址区间有关的可变参数，并创建静态变量 addr_data。

根据 ltc3445 手册，其 7 位 i2c 总线地址的低 5 位总是“01011”，高两位 ADD7 和 ADD6 由硬件布线决定，可以分别接到 VCC 或者 GND，即逻辑上分别为 1 或 0，所以 ltc3445 所有可能使用的地址也就只有这两个了。另外在当前应用中只有 ltc3445 这一个 i2c 设备，所以在 normal_i2c[]数组中只需声明 ltc3445 的这两个地址即可。

尤其需要说明的是，一个驱动程序可能适用于多个设备，而这些设备可能在同一条 i2c 总线上，也可能在不同的 i2c 总线上。驱动程序模块的 addr_data 二维数组提供了使用该驱动的**所有**设备的可能的地址的线索！（正是因为这样，在安装驱动程序模块、检进行地址检测时需要进行两层循环：外层针对每一个已注册的 i2c_adapter，即每一条总线，内层针对一条总线上所有可能的 7 位地址。详见下文源代码分析。）

4. 在 ltc3445.c 文件中定义方法 ltc3445_attach_adapter：

```
int ltc3445_attach_adapter(struct i2c_adapter *adapter)
{
    return i2c_probe(adapter, &addr_data, ltc3445_detect_client);
}
```

该函数在 ltc3445 设备驱动程序模块的初始化时调用，针对系统上每个 i2c 总线都被调用一次，尝试认领该总线上所有使用这个驱动的 i2c 设备。另外，这个函数只需直接调用在 i2c-core.c 文件中实现的 i2c_probe 函数即可，传递的第三个参数为由驱动程序模块提供的回调函数 ltc3445_detect_client。

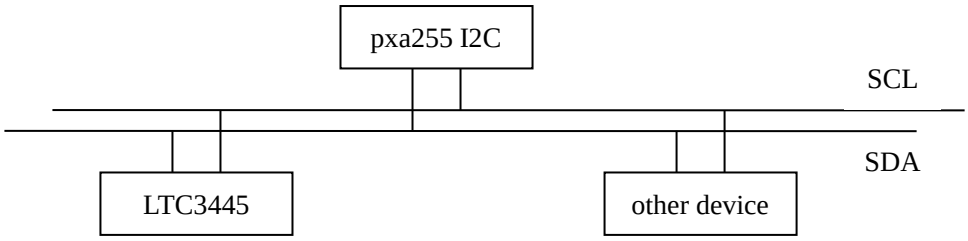
5. 在 ltc3445.c 文件中定义回调函数 ltc3445_detect_client。在检测到设备时调用该函数，传递的第一个参数为当前 i2c 适配器的 i2c_adapter 数据结构的地址，第二个参数为设备地址。该函数创建设备的 i2c_client 数据结构并直接调用 i2c-core.c 中实现的 i2c_attach_client 函数向 i2c 适配器数据结构

i2c_adapter.clients 指针数组注册，最后调用与具体 i2c 设备相关的代码初始化设备，并完成 i2c_client 中私有数据结构的初始化。

6. 最后，在 ltc3445.c 文件中还要定义增加设备驱动程序模块引用计数的方法，以及模块的初始化、清理函数。

第 9 章 用户进程访问 i2c 设备的步骤

在物理连接上，i2c 适配器（adapter）就是一条 i2c 总线的控制器，若干 i2c 设备并联于该 i2c 总线上，如下图所示：



pxa255 的 i2c 适配器的寄存器组的地址是 0x4038xxxx，位于 pxa255 的内存映射寄存器（memory-mapped register）范围内。而 i2c 设备是不需要内存地址的，只需使用 7 位总线地址即可，因为它们只通过 SCL 和 SDA 两根线 i2c 适配器相连。

一条 i2c 总线（i2c 适配器及总线上的所有设备）只用“/dev/i2c/%d”一个设备节点表示。该设备节点的主号为 89，次号等于“%d”，为该总线的适配器在系统中的编号（即，指向其 i2c_adapter 数据结构的指针在内核静态数组 adapters 中的偏移）。一条 i2c 总线上的若干设备则通过 7 位总线地址寻址。注意这个总线地址是相对地址，即如果存在多个 i2c 适配器（多条 i2c 总线），则不同总线上的设备可以使用相同的地址。即一个具体 i2c 设备的“地址”是其所在 i2c 总线的适配器的设备节点和其总线地址的二元组。

系统中所有已安装的 i2c 总线（i2c 适配器及其驱动）的信息可由/proc/bus/i2c 文件得到。该文件的内容由4列信息组成，第一列为“i2c-%d”，即适配器编号，第二列为总线类型（i2c/smbus），三、四列为适配器及其驱动（algorithm）的名称。

每一条 i2c 总线上的设备信息可由/proc/bus/i2c-%d 文件得到。该文件的内容由 3 列信息组成，第一列为一个 i2c 设备的在该总线上的相对地址，第二、三列为设备及其驱动的名称。

所以，用户进程访问 ltc3445 设备的步骤如下：

1. 找到包含设备名称的/proc/bus/i2c-%d 文件，并得到设备在总线上的地址 addr。
2. %d 即为设备所在总线在系统内的编号，该条总线由/dev/i2c/%d 节点表示。
3. 用 open 系统调用打开/dev/i2c/%d 节点（该节点代表 i2c 适配器，即整条 i2c 总线）。
4. 用 ioctl(file, I2C_TMP_DEVICE_ADDR, addr)系统调用设定待访问的设备地址。
5. 如果读出寄存器，则必须首先将待读出寄存器的编号在局部变量 buf 中指明，然后使用 buf 保存 read 操作的结果。即 buf 是一个“输入－输出”参数。
6. 如果写入寄存器，则使用 unsigned char buf[2]，其中 buf[0]为待写入寄存器的编号，buf[1]为待写入的值。

讨论和总结

i2c 操作中的同步问题

与操作磁盘相类比：用户进程对磁盘的**写入操作是一般是异步**的（除非执行同步写操作）：用户进程的 write 系统调用执行流只需向磁盘请求队列中添加相应的请求，系统调用执行流就可以退出了，而用户进程无需等待写入操作的完成。真正写入的磁盘操作由后台内核线程完成。用户进程对磁盘的**读操作一般为同步**的：用户进程需要阻塞在相应缓冲区首部的等待队列上，由磁盘低级驱动操作磁盘，在磁盘操作完成后引起中断而激活一个下半部分，而在下半部分中唤醒阻塞与缓冲区首部中的等待进程。

用户进程读写 i2c 总线的操作时**始终是同步**的：用户进程在操作期间将挂起在 i2c_pxa.wait 等待队列上，直到在中断中判断 i2c 操作完成时才将其唤醒。由 i2c_master_recv 和 i2c_master_send 函数可知，当前进程在调用 i2c 适配器的 master_xfer 方法进行 i2c 操作前，必须首先通过 I2C_LOCK 宏获得 i2c_adapter 的 lock 信号量，而且**在当前进程阻塞时一直持有该信号量**，从而实现对 i2c 适配器访问的同步。由此可见 i2c 操作是原子的。

i2c 操作的同步问题还包括另一个方面：在读写 i2c 设备的系统调用执行流中最终将通过 i2c_pxa_do_xfer 函数初始化指向待传输信息的指针并发起 i2c 通信，而此期间可能发生 i2c 中断，在 i2c 中断处理程序 i2c_pxa_handler 中也会步进指向待传输信息的指针。所以应该把 i2c_pxa_do_xfer 函数的相关操作放到中断禁区中执行。于是才在 pxa_i2c 数据结构中设计了 lock 自旋锁并通过 spin_lock/unlock_irq 函数来实现中断禁区。

另外，自旋锁在嵌入式应用中往往都退化为实现中断禁区。当前应用所采用的 2.4.19 版本 Linux 中尚不支持内核态抢占，所以自旋锁的分配、释放操作也无需改变原子性计数 preempt_count。于是可以通过 spin_lock/unlock_irq 函数同步中断处理程序，或者通过 spin_lock/unlock_bh 同步下半部分（一定要分析清楚竞争条件的双方究竟是谁,从而采用适宜的同步机制。比如，如果当前内核控制路径可能与下半部分发生竞争条件，那么使用 spin_lock/unlock_irq 函数开关中断就有“扩大打击面”之嫌了，呵呵）。

总结各个模块初始化函数的作用

通读全文后，现在是跳出源代码细节、站到高处总结各个模块初始化函数所注册的设施的时候了。我们还是假设按照如下的顺序安装各个模块：

安装 i2c-core 模块 > 安装 pxa255 的 i2c 适配器驱动程序模块 > 安装 ltc3445 设备驱动程序模块 > 安装 i2c-dev 模块 > 用户进程访问 ltc3445 设备（open > ioctl > read/write）

1. 安装 i2c-core 模块：
- 初始化内核静态数组 drivers[]和 adapters[]，及相关的计数和信号量；
- 创建/proc/bus/i2c 文件。

2. 安装 pxa255 的 i2c 适配器驱动程序模块：

- 向 adapters[]数组注册适配器的 i2c_adapter 数据结构
- 创建/proc/bus/i2c-%d 文件
- 对所有的已注册设备驱动程序，调用设备驱动程序的 attach_adapter 方法。

设备驱动程序的 attach_adapter 方法将利用适配器驱动程序模块提供的 i2c 总线访问方法（master_xfer 或 smbus_xfer），利用设备驱动程序模块中提供的地址线索信息，检测可能存在的设备及其地址。如果成功发现设备，则利用设备驱动程序提供的回调函数 detect_client 创建设备数据结构 i2c_client 并向该适配器的数据结构注册。

但是，此时尚未安装 ltc3445 设备驱动程序，所以这一工作只好推迟了。

3. 安装 ltc3445 设备驱动程序模块：

- 向 drivers[]数组注册设备驱动程序的 i2c_driver 数据结构
- 为 adapters[]数组中每一个已注册的适配器数据结构，调用设备驱动程序的 attach_adapter 方法。

现在，终于可以进行 ltc3445 设备的检测工作了，并为之分配 i2c_client 数据结构，并注册到 i2c_pxa.adap.clients[]中。

4. 安装 i2c-dev 模块：

- 向系统中注册字符设备访问方法表 i2cdev_fops
- 创建/dev/i2c/目录
- 为 adapters[]数组中每一个已注册的适配器数据结构，创建/dev/i2c/%d 设备节点。设备节点的文件系统操作方法正是 i2cdev_fops。

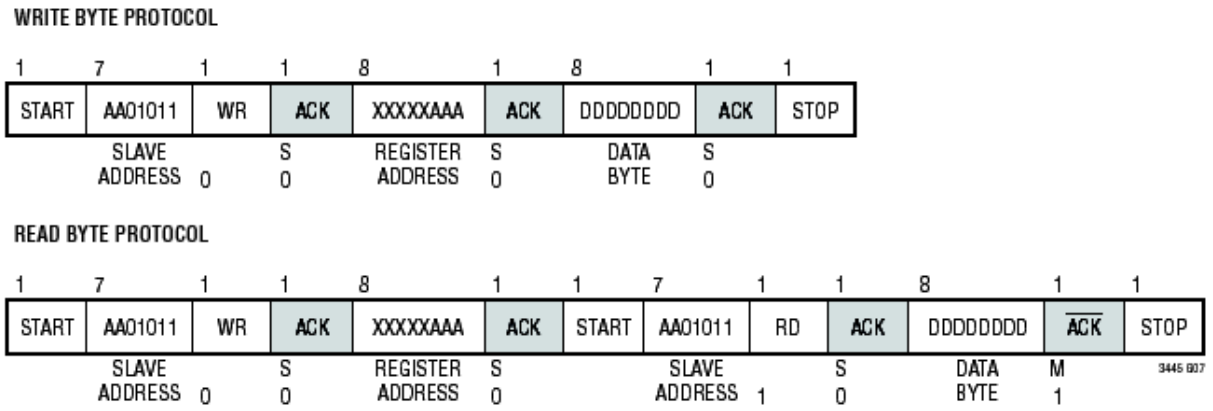
由于在第 2 步已经安装了 pxa255 的 i2c 适配器驱动程序模块，所以这里将为该条 i2c 总线创建相应的/dev 设备节点。从此，用户进程就可以通过文件系统调用来访问 i2c 设备了。

对 i2c 框架代码的修改

下图是 ltc3445 的读写规程。在写操作时因为只有一次 i2c 交互，所以只需要一个 i2c_msg 数据结构即可。用户进程在写入前准备好两个字节的数组，第一个数组元素为待写入的寄存器编号，第二个字节为写入的数值。而无需对 i2c_master_send 函数做任何修改。

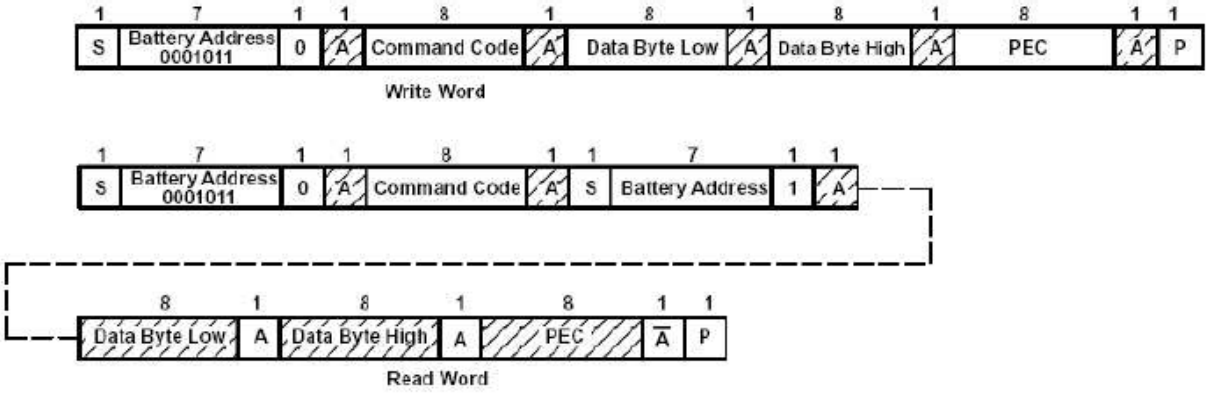
而在读操作时需要两次交互，而且以“Repeated Start”的形式出现。所以就不得不使用两个 i2c_msg 数据结构：第一个 i2c_msg 数据结构完成写操作，写入的数值为待读出的寄存器编号；第二个 i2c_msg 数据结构完成读操作，从先前指定的寄存器中读出数据。

所以修改 i2c_master_recv 函数使用两个 i2c_msg 数据结构（而在 i2c-2.9.1 中 i2c_master_send/recv 总是使用一个 i2c_msg 数据结构），第一次 i2c 交互写入用户准备好的寄存器编号，第二次 i2c 交互读出寄存器的内容，**到先前保存有寄存器编号的那个地址空间中**。所以用户进程在读出前只需准备好一个字节的数组，数组元素为待读出的寄存器编号，从读操作返回后，这个数组元素即含有读出的寄存器值。



需要说明的是，如果只有一个 i2c 设备，那么这样修改 i2c_master_recv 函数是没有问题的。**可是如果有两个以上 i2c 设备，而这些 i2c 设备的读规程不尽相同，那么，岂不要对 i2c_master_recv 函数做相互矛盾的修改了吗？！**

还好，估计这种情况不会出现。比如下图是另一款 i2c 设备，TI bq2084 i2c 芯片，的读写规程：



从图中可以看到，写操作也是由一次 i2c 交互即可完成，所以只需一个 i2c_msg 数据结构。用户进程在写操作前准备好 3 个字节的数组：第 1 个字节指明待写入的寄存器值，第 2、3 个字节为写入的数值（bq2084 的寄存器长度为 16 位）。而无需对 i2c_master_send 函数做任何修改。

读操作也是由两次 i2c 操作完成的，而且也是“Repeated Start”的形式。

目前已经把 i2c_master_recv 函数按照 ltc3445 读规程的要求改为了使用两个 i2c_msg 数据结构，而且用户进程传递一个字节的数组：第一次 i2c 交互写入该字节的数据，第二此 i2c 交互将读出的结果写回该字节，那么用户进程在读操作完成后就从该字节得到返回值了。

幸运的是，针对 ltc3445 的读规程修改的 i2c_master_recv 函数同样可以很好地为 bq2084 工作：用户进程只需要在读操作前准备好 2 个字节的数组：第 1 个字节为待读出的寄存器编号，第 2 字节空置。那么在读操作后就可以从这两个字节得到返回值了。

总结对 i2c_master_recv 函数的修改：
在 i2c-2.9.1 中 i2c_master_recv 函数只使用一个 i2c_msg 数据结构，因此只能完成一次 i2c 交互。而读操作首先需要写入待读出的寄存器编号，然后第二次 i2c 操作才能读出值。所以 i2c-2.9.1 中的 i2c_master_recv 是不正确的。

修改 i2c_master_recv 使用两个 i2c_msg 数据结构，第一个 i2c_msg.len = 1，第二个 i2c_msg.len = count。第一个 i2c_msg 的 len 域一定设置为 1，因为用一个字节表示寄存器编号足矣！**第二个 i2c_msg 的 len 域一定要设置为用户空间缓冲区的长度 count！这是因为不同 i2c 设备的寄存器长度可能不同**！比如 ltc3445 的寄存器长度为 1 个字节，所以 count =1，而 bq2084 的寄存器长度为 2，所以 count = 2。

另外，还需要相应地修改 i2cdev_read 函数：在调用 i2c_master_recv 前必须事先通过 copy_from_user 从用户空间拷贝待读出的寄存器编号！

有关 i2c 设备私有数据结构的讨论

上一小节中针对 ltc3445 的读规程对 i2c_master_recv 函数的修改可能具有普适性，因为其它的 i2c 芯片的读规程也很可能使用“Repeated Start”。使用修改的 i2c_master_recv 函数，用户进程在读取不同的 i2c 设备时只需准备不同长度的数组即可：读 ltc3445 时准备一个字节的数组，数组元素为待读出的寄存器编号；读 bq2084 时准备 2 个字节的数组，第一个数组元素为待读出的寄存器编号。那么，在读操作完成后，用户进程都可以从数组中得到寄存器的值。

但是对于私有数据结构就没有这么幸运了！

不同 i2c 芯片的特性不同，因此私有数据结构也不同，很难统一。而为了使用私有数据结构就至少需要修改 i2c_master_send 和 i2c_master_recv 函数。显然针对一个 i2c 设备私有数据结构的修改往往不适用于其它的 i2c 设备，导致破坏了 i2c 框架代码的与具体应用无关的特性，从而损害其可移植性和重用性。

还好，即使不使用私有数据结构也可以有其它的解决方法：由访问 i2c 设备的用户进程在静态或全局变量中保存私有数据结构的内容，同时提供读写 i2c 设备的封装函数，在封装函数中利用用户态的私有数据结构来决定是否真的发出 read、write 系统调用！**即把私有数据结构从设备驱动层提高到用户层，从而使得设备驱动层的代码于设备的特性无关！**

遗留的问题

1. 在当前的适配器驱动程序模块的初始化函数中并不创建设备结点，这一任务只能由 i2c-dev 模块完成。所以必须首先安装适配器驱动程序模块，然后再安装 i2c-dev 模块。这样岂不缺乏灵活性？
2. 补充有关引用计数的说明：由于现在 i2c 框架代码和所有驱动都静态地链接入内核，而不是通过模块的方式加载，所以尚未深究适配器和设备驱动程序模块的引用计数。
3. usage_count 引用计数由 i2c_use_client 和 i2c_release_client 函数控制，在这两个函数中还分别调用 i2c_adapter 和 i2c_driver 中定义的 inc_use 和 dec_use 方法。但是在 pxa255 的 i2c 补丁中在访问 i2c 设备期间并没有使用 i2c_use_client 和 i2c_release_client 函数来控制 usage_count 计数。

（全文完）