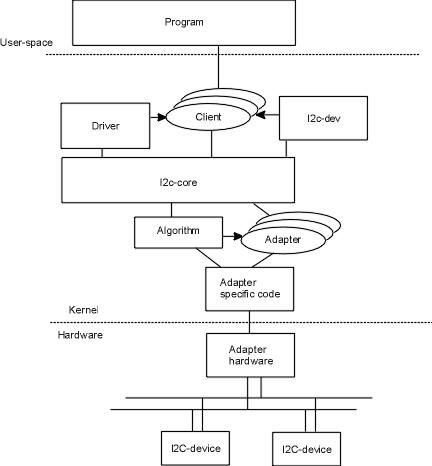
一:前言

I2c是philips提出的外设总线.I2C只有两条线,一条串行数据线:SDA,一条是时钟线SCL.正因为这样,它方便了工程人员的布线.另外,I2C是一种多主机控制总线.它和USB总线不同,USB是基于master-slave机制,任何设备的通信必须由主机发起才可以.而I2C是基于multi master机制.一同总线上可允许多个master.关于I2C协议的知识,这里不再赘述.可自行下载spec阅读即可.

二:I2C架构概述

在linux中,I2C驱动架构如下所示:



如上图所示,每一条I2C对应一个adapter.在kernel中,每一个adapter提供了一个描述的结构(struct i2c\_adapter),也定义了adapter支持的操作(struct i2c\_adapter).再通过i2c core层将i2c设备与i2c adapter关联起来.

这个图只是提供了一个大概的框架.在下面的代码分析中,从下至上的来分析这个框架图.以下的代码分析是基于linux 2.6.26.分析的代码基本位于: linux-2.6.26.3/drivers/i2c/位置.

三:adapter注册

在kernel中提供了两个adapter注册接口,分别为i2c\_add\_adapter()和i2c\_add\_numbered\_adapter().由于在系统中可能存在多个adapter,因为将每一条I2C总线对应一个编号,下文中称为I2C总线号.这个总线号的PCI中的总线号不同.它和硬件无关,只是软件上便于区分而已.

对于i2c\_add\_adapter()而言,它使用的是动态总线号,即由系统给其分析一个总线号,而i2c\_add\_numbered\_adapter()则是自己指定总线号,如果这个总线号非法或者是被占用,就会注册失败.

分别来看一下这两个函数的代码:

int i2c\_add\_adapter(struct i2c\_adapter \*adapter)

{

int id, res = 0;

retry:

if (idr\_pre\_get(&i2c\_adapter\_idr, GFP\_KERNEL) == 0)

return -ENOMEM;

mutex\_lock(&core\_lock);

/\* "above" here means "above or equal to", sigh \*/

res = idr\_get\_new\_above(&i2c\_adapter\_idr, adapter,

\_\_i2c\_first\_dynamic\_bus\_num, &id);

mutex\_unlock(&core\_lock);

if (res < 0) {

if (res == -EAGAIN)

goto retry;

return res;

}

adapter->nr = id;

return i2c\_register\_adapter(adapter);

}

在这里涉及到一个idr结构.idr结构本来是为了配合page cache中的radix tree而设计的.在这里我们只需要知道,它是一种高效的搜索树,且这个树预先存放了一些内存.避免在内存不够的时候出现问题.所在,在往idr中插入结构的时候,首先要调用idr\_pre\_get()为它预留足够的空闲内存,然后再调用idr\_get\_new\_above()将结构插入idr中,该函数以参数的形式返回一个id.以后凭这个id就可以在idr中找到相对应的结构了.对这个数据结构操作不太理解的可以查阅本站<< linux文件系统之文件的读写>>中有关radix tree的分析.

注意一下idr\_get\_new\_above(&i2c\_adapter\_idr, adapter,\_\_i2c\_first\_dynamic\_bus\_num, &id)的参数的含义,它是将adapter结构插入到i2c\_adapter\_idr中,存放位置的id必须要大于或者等于\_\_i2c\_first\_dynamic\_bus\_num,

然后将对应的id号存放在adapter->nr中.调用i2c\_register\_adapter(adapter)对这个adapter进行进一步注册.

看一下另外一人注册函数: i2c\_add\_numbered\_adapter( ),如下所示:

int i2c\_add\_numbered\_adapter(struct i2c\_adapter \*adap)

{

int id;

int status;

if (adap->nr & ~MAX\_ID\_MASK)

return -EINVAL;

retry:

if (idr\_pre\_get(&i2c\_adapter\_idr, GFP\_KERNEL) == 0)

return -ENOMEM;

mutex\_lock(&core\_lock);

/\* "above" here means "above or equal to", sigh;

\* we need the "equal to" result to force the result

\*/

status = idr\_get\_new\_above(&i2c\_adapter\_idr, adap, adap->nr, &id);

if (status == 0 && id != adap->nr) {

status = -EBUSY;

idr\_remove(&i2c\_adapter\_idr, id);

}

mutex\_unlock(&core\_lock);

if (status == -EAGAIN)

goto retry;

if (status == 0)

status = i2c\_register\_adapter(adap);

return status;

}

对比一下就知道差别了,在这里它已经指定好了adapter->nr了.如果分配的id不和指定的相等,便返回错误.

过一步跟踪i2c\_register\_adapter().代码如下:

static int i2c\_register\_adapter(struct i2c\_adapter \*adap)

{

int res = 0, dummy;

mutex\_init(&adap->bus\_lock);

mutex\_init(&adap->clist\_lock);

INIT\_LIST\_HEAD(&adap->clients);

mutex\_lock(&core\_lock);

/\* Add the adapter to the driver core.

\* If the parent pointer is not set up,

\* we add this adapter to the host bus.

\*/

if (adap->dev.parent == NULL) {

adap->dev.parent = &platform\_bus;

pr\_debug("I2C adapter driver [%s] forgot to specify "

"physical device\n", adap->name);

}

sprintf(adap->dev.bus\_id, "i2c-%d", adap->nr);

adap->dev.release = &i2c\_adapter\_dev\_release;

adap->dev.class = &i2c\_adapter\_class;

res = device\_register(&adap->dev);

if (res)

goto out\_list;

dev\_dbg(&adap->dev, "adapter [%s] registered\n", adap->name);

/\* create pre-declared device nodes for new-style drivers \*/

if (adap->nr < \_\_i2c\_first\_dynamic\_bus\_num)

i2c\_scan\_static\_board\_info(adap);

/\* let legacy drivers scan this bus for matching devices \*/

dummy = bus\_for\_each\_drv(&i2c\_bus\_type, NULL, adap,

i2c\_do\_add\_adapter);

out\_unlock:

mutex\_unlock(&core\_lock);

return res;

out\_list:

idr\_remove(&i2c\_adapter\_idr, adap->nr);

goto out\_unlock;

}

首先对adapter和adapter中内嵌的struct device结构进行必须的初始化.之后将adapter内嵌的struct device注册.

在这里注意一下adapter->dev的初始化.它的类别为i2c\_adapter\_class,如果没有父结点,则将其父结点设为platform\_bus.adapter->dev的名字为i2c + 总线号.

测试一下:

[eric@mochow i2c]$ cd /sys/class/i2c-adapter/

[eric@mochow i2c-adapter]$ ls

i2c-0

可以看到,在我的PC上,有一个I2C adapter,看下详细信息:

[eric@mochow i2c-adapter]$ tree

.

`-- i2c-0

|-- device -> ../../../devices/pci0000:00/0000:00:1f.3/i2c-0

|-- name

|-- subsystem -> ../../../class/i2c-adapter

`-- uevent

3 directories, 2 files

可以看到,该adapter是一个PCI设备.

继续往下看:

之后,在注释中看到,有两种类型的driver,一种是new-style drivers,另外一种是legacy drivers

New-style drivers是在2.6近版的kernel加入的.它们最主要的区别是在adapter和i2c driver的匹配上.

3.1: new-style 形式的adapter注册

对于第一种,也就是new-style drivers,将相关代码再次列出如下:

if (adap->nr < \_\_i2c\_first\_dynamic\_bus\_num)

i2c\_scan\_static\_board\_info(adap);

如果adap->nr 小于\_\_i2c\_first\_dynamic\_bus\_num的话,就会进入到i2c\_scan\_static\_board\_info().

结合我们之前分析的adapter的两种注册分式: i2c\_add\_adapter()所分得的总线号肯会不会小于\_\_i2c\_first\_dynamic\_bus\_num.只有i2c\_add\_numbered\_adapter()才有可能满足:

(adap->nr < \_\_i2c\_first\_dynamic\_bus\_num)

而且必须要调用i2c\_register\_board\_info()将板子上的I2C设备信息预先注册时才会更改\_\_i2c\_first\_dynamic\_bus\_num的值.在x86上只没有使用i2c\_register\_board\_info()的.因此,x86平台上的分析可以忽略掉new-style driver的方式.不过,还是详细分析这种情况下.

首先看一下i2c\_register\_board\_info(),如下:

int \_\_init

i2c\_register\_board\_info(int busnum,

struct i2c\_board\_info const \*info, unsigned len)

{

int status;

mutex\_lock(&\_\_i2c\_board\_lock);

/\* dynamic bus numbers will be assigned after the last static one \*/

if (busnum >= \_\_i2c\_first\_dynamic\_bus\_num)

\_\_i2c\_first\_dynamic\_bus\_num = busnum + 1;

for (status = 0; len; len--, info++) {

struct i2c\_devinfo \*devinfo;

devinfo = kzalloc(sizeof(\*devinfo), GFP\_KERNEL);

if (!devinfo) {

pr\_debug("i2c-core: can't register boardinfo!\n");

status = -ENOMEM;

break;

}

devinfo->busnum = busnum;

devinfo->board\_info = \*info;

list\_add\_tail(&devinfo->list, &\_\_i2c\_board\_list);

}

mutex\_unlock(&\_\_i2c\_board\_lock);

return status;

}

这个函数比较简单, struct i2c\_board\_info用来表示I2C设备的一些情况,比如所在的总线.名称,地址,中断号等.最后,这些信息会被存放到\_\_i2c\_board\_list链表.

跟踪i2c\_scan\_static\_board\_info():代码如下:

static void i2c\_scan\_static\_board\_info(struct i2c\_adapter \*adapter)

{

struct i2c\_devinfo \*devinfo;

mutex\_lock(&\_\_i2c\_board\_lock);

list\_for\_each\_entry(devinfo, &\_\_i2c\_board\_list, list) {

if (devinfo->busnum == adapter->nr

&& !i2c\_new\_device(adapter,

&devinfo->board\_info))

printk(KERN\_ERR "i2c-core: can't create i2c%d-%04x\n",

i2c\_adapter\_id(adapter),

devinfo->board\_info.addr);

}

mutex\_unlock(&\_\_i2c\_board\_lock);

}

该函数遍历挂在\_\_i2c\_board\_list链表上面的i2c设备的信息,也就是我们在启动的时候指出的i2c设备的信息.

如果指定设备是位于adapter所在的I2C总线上,那么,就调用i2c\_new\_device().代码如下:

struct i2c\_client \*

i2c\_new\_device(struct i2c\_adapter \*adap, struct i2c\_board\_info const \*info)

{

struct i2c\_client \*client;

int status;

client = kzalloc(sizeof \*client, GFP\_KERNEL);

if (!client)

return NULL;

client->adapter = adap;

client->dev.platform\_data = info->platform\_data;

device\_init\_wakeup(&client->dev, info->flags & I2C\_CLIENT\_WAKE);

client->flags = info->flags & ~I2C\_CLIENT\_WAKE;

client->addr = info->addr;

client->irq = info->irq;

strlcpy(client->name, info->type, sizeof(client->name));

/\* a new style driver may be bound to this device when we

\* return from this function, or any later moment (e.g. maybe

\* hotplugging will load the driver module). and the device

\* refcount model is the standard driver model one.

\*/

status = i2c\_attach\_client(client);

if (status < 0) {

kfree(client);

client = NULL;

}

return client;

}

我们又遇到了一个新的结构:struct i2c\_client,不要被这个结构吓倒了,其实它就是一个嵌入struct device的I2C设备的封装.它和我们之前遇到的struct usb\_device结构的作用是一样的.

首先,在clinet里保存该设备的相关消息.特别的, client->adapter指向了它所在的adapter.

特别的,clinet->name为info->name.也是指定好了的.

一切初始化完成之后,便会调用i2c\_attach\_client( ).看这个函数的字面意思,是将clinet关联起来.到底怎么样关联呢?继续往下看:

int i2c\_attach\_client(struct i2c\_client \*client)

{

struct i2c\_adapter \*adapter = client->adapter;

int res = 0;

//初始化client内嵌的dev结构

//父结点为所在的adapter,所在bus为i2c\_bus\_type

client->dev.parent = &client->adapter->dev;

client->dev.bus = &i2c\_bus\_type;

//如果client已经指定了driver,将driver和内嵌的dev关联起来

if (client->driver)

client->dev.driver = &client->driver->driver;

//指定了driver, 但不是newstyle的

if (client->driver && !is\_newstyle\_driver(client->driver)) {

client->dev.release = i2c\_client\_release;

client->dev.uevent\_suppress = 1;

} else

client->dev.release = i2c\_client\_dev\_release;

//clinet->dev的名称

snprintf(&client->dev.bus\_id[0], sizeof(client->dev.bus\_id),

"%d-%04x", i2c\_adapter\_id(adapter), client->addr);

//将内嵌的dev注册

res = device\_register(&client->dev);

if (res)

goto out\_err;

//将clinet链到adapter->clients中

mutex\_lock(&adapter->clist\_lock);

list\_add\_tail(&client->list, &adapter->clients);

mutex\_unlock(&adapter->clist\_lock);

dev\_dbg(&adapter->dev, "client [%s] registered with bus id %s\n",

client->name, client->dev.bus\_id);

//如果adapter->cleinet\_reqister存在,就调用它

if (adapter->client\_register) {

if (adapter->client\_register(client)) {

dev\_dbg(&adapter->dev, "client\_register "

"failed for client [%s] at 0x%02x\n",

client->name, client->addr);

}

}

return 0;

out\_err:

dev\_err(&adapter->dev, "Failed to attach i2c client %s at 0x%02x "

"(%d)\n", client->name, client->addr, res);

return res;

}

参考上面添加的注释,应该很容易理解这段代码了,就不加详细分析了.这个函数的名字不是i2c\_attach\_client()么?怎么没看到它的关系过程呢?

这是因为:在代码中设置了client->dev所在的bus为i2c\_bus\_type .以为只需要有bus为i2c\_bus\_type的driver注册,就会产生probe了.这个过程呆后面分析i2c driver的时候再来详细分析.

3.2: legacy形式的adapter注册

Legacy形式的adapter注册代码片段如下:

dummy = bus\_for\_each\_drv(&i2c\_bus\_type, NULL, adap,

i2c\_do\_add\_adapter);

这段代码遍历挂在i2c\_bus\_type上的驱动,然后对每一个驱动和adapter调用i2c\_do\_add\_adapter().

代码如下:

static int i2c\_do\_add\_adapter(struct device\_driver \*d, void \*data)

{

struct i2c\_driver \*driver = to\_i2c\_driver(d);

struct i2c\_adapter \*adap = data;

if (driver->attach\_adapter) {

/\* We ignore the return code; if it fails, too bad \*/

driver->attach\_adapter(adap);

}

return 0;

}

该函数很简单,就是调用driver的attach\_adapter()接口.

到此为止,adapter的注册已经分析完了.

四:i2c driver注册

在分析i2c driver的时候,有必要先分析一下i2c架构的初始化

代码如下:

static int \_\_init i2c\_init(void)

{

int retval;

retval = bus\_register(&i2c\_bus\_type);

if (retval)

return retval;

retval = class\_register(&i2c\_adapter\_class);

if (retval)

goto bus\_err;

retval = i2c\_add\_driver(&dummy\_driver);

if (retval)

goto class\_err;

return 0;

class\_err:

class\_unregister(&i2c\_adapter\_class);

bus\_err:

bus\_unregister(&i2c\_bus\_type);

return retval;

}

subsys\_initcall(i2c\_init);

很明显,i2c\_init()会在系统初始化的时候被调用.

在i2c\_init中,先注册了i2c\_bus\_type的bus,i2c\_adapter\_class的class.然后再调用i2c\_add\_driver()注册了一个i2c driver.

I2c\_bus\_type结构如下:

static struct bus\_type i2c\_bus\_type = {

.name = "i2c",

.dev\_attrs = i2c\_dev\_attrs,

.match = i2c\_device\_match,

.uevent = i2c\_device\_uevent,

.probe = i2c\_device\_probe,

.remove = i2c\_device\_remove,

.shutdown = i2c\_device\_shutdown,

.suspend = i2c\_device\_suspend,

.resume = i2c\_device\_resume,

};

这个结构先放在这里吧,以后还会用到里面的信息的.

从上面的初始化函数里也看到了,注册i2c driver的接口为i2c\_add\_driver().代码如下:

static inline int i2c\_add\_driver(struct i2c\_driver \*driver)

{

return i2c\_register\_driver(THIS\_MODULE, driver);

}

继续跟踪:

int i2c\_register\_driver(struct module \*owner, struct i2c\_driver \*driver)

{

int res;

/\* new style driver methods can't mix with legacy ones \*/

//如果是一个newstyle的driver.但又定义了attach\_adapter/detach\_adapter.非法

if (is\_newstyle\_driver(driver)) {

if (driver->attach\_adapter || driver->detach\_adapter

|| driver->detach\_client) {

printk(KERN\_WARNING

"i2c-core: driver [%s] is confused\n",

driver->driver.name);

return -EINVAL;

}

}

/\* add the driver to the list of i2c drivers in the driver core \*/

//关联到i2c\_bus\_types

driver->driver.owner = owner;

driver->driver.bus = &i2c\_bus\_type;

/\* for new style drivers, when registration returns the driver core

\* will have called probe() for all matching-but-unbound devices.

\*/

//注册内嵌的driver

res = driver\_register(&driver->driver);

if (res)

return res;

mutex\_lock(&core\_lock);

pr\_debug("i2c-core: driver [%s] registered\n", driver->driver.name);

/\* legacy drivers scan i2c busses directly \*/

//遍历所有的adapter,对其都调用driver->attach\_adapter

if (driver->attach\_adapter) {

struct i2c\_adapter \*adapter;

down(&i2c\_adapter\_class.sem);

list\_for\_each\_entry(adapter, &i2c\_adapter\_class.devices,

dev.node) {

driver->attach\_adapter(adapter);

}

up(&i2c\_adapter\_class.sem);

}

mutex\_unlock(&core\_lock);

return 0;

}

这里也有两种形式的区分,对于第一种,只需要将内嵌的driver注册就可以了,对于legacy的情况,对每一个adapter都调用driver->attach\_adapter().

现在,我们可以将adapter和i2c driver关联起来考虑一下了:

1:如果是news style形式的,在注册adapter的时候,将它上面的i2c 设备转换成了struct client.struct client->dev->bus又指定了和i2c driver同一个bus.因为,它们可以发生probe.

2:如果是legacy形式,就直接找到对应的对象,调用driver->attach\_adapter().

五: i2c\_bus\_type的相关操作

I2c\_bus\_type的操作主要存在于new-style形式的驱动中.接下来分析一下对应的probe过程:

5.1:match过程分析

Match对应的操作函数为i2c\_device\_match().代码如下

static int i2c\_device\_match(struct device \*dev, struct device\_driver \*drv)

{

struct i2c\_client \*client = to\_i2c\_client(dev);

struct i2c\_driver \*driver = to\_i2c\_driver(drv);

/\* make legacy i2c drivers bypass driver model probing entirely;

\* such drivers scan each i2c adapter/bus themselves.

\*/

if (!is\_newstyle\_driver(driver))

return 0;

/\* match on an id table if there is one \*/

if (driver->id\_table)

return i2c\_match\_id(driver->id\_table, client) != NULL;

return 0;

}

如果该驱动不是一个new-style形式的.或者driver没有定义匹配的id\_table.都会匹配失败.

继续跟踪进i2c\_match\_id():

static const struct i2c\_device\_id \*i2c\_match\_id(const struct i2c\_device\_id \*id,

const struct i2c\_client \*client)

{

while (id->name[0]) {

if (strcmp(client->name, id->name) == 0)

return id;

id++;

}

return NULL;

}

由此可见.如果client的名字和driver->id\_table[]中的名称匹配即为成功.

5.2:probe过程分析

Probe对应的函数为: i2c\_device\_probe()

static int i2c\_device\_probe(struct device \*dev)

{

struct i2c\_client \*client = to\_i2c\_client(dev);

struct i2c\_driver \*driver = to\_i2c\_driver(dev->driver);

const struct i2c\_device\_id \*id;

int status;

if (!driver->probe)

return -ENODEV;

client->driver = driver;

dev\_dbg(dev, "probe\n");

if (driver->id\_table)

id = i2c\_match\_id(driver->id\_table, client);

else

id = NULL;

status = driver->probe(client, id);

if (status)

client->driver = NULL;

return status;

}

这个函数也很简单,就是将probe流程回溯到i2c driver的probe()

六:其它的扩展

分析完adapter和i2c driver的注册之后,好像整个架构也差不多了,其它,扩展的东西还有很多.

我们举一个legacy形式的例子,这个例子是在kernel中随便搜索出来的:

在linux-2.6.26.3/drivers/hwmon/ad7418.c中,初始化函数为:

static int \_\_init ad7418\_init(void)

{

return i2c\_add\_driver(&ad7418\_driver);

}

i2c\_driver ad7418\_driver结构如下:

static struct i2c\_driver ad7418\_driver = {

.driver = {

.name = "ad7418",

},

.attach\_adapter = ad7418\_attach\_adapter,

.detach\_client = ad7418\_detach\_client,

};

该结构中没有probe()函数,可以断定是一个legacy形式的驱动.这类驱动注册的时候,会调用driver的attach\_adapter函数.在这里也就是ad7418\_attach\_adapter.

这个函数代码如下:

static int ad7418\_attach\_adapter(struct i2c\_adapter \*adapter)

{

if (!(adapter->class & I2C\_CLASS\_HWMON))

return 0;

return i2c\_probe(adapter, &addr\_data, ad7418\_detect);

}

在这里我们又遇到了一个i2c-core中的函数,i2c\_probe().在分析这个函数之前,先来看下addr\_data是什么?

#define I2C\_CLIENT\_MODULE\_PARM(var,desc) \

static unsigned short var[I2C\_CLIENT\_MAX\_OPTS] = I2C\_CLIENT\_DEFAULTS; \

static unsigned int var##\_num; \

module\_param\_array(var, short, &var##\_num, 0); \

MODULE\_PARM\_DESC(var,desc)

#define I2C\_CLIENT\_MODULE\_PARM\_FORCE(name) \

I2C\_CLIENT\_MODULE\_PARM(force\_##name, \

"List of adapter,address pairs which are " \

"unquestionably assumed to contain a `" \

# name "' chip")

#define I2C\_CLIENT\_INSMOD\_COMMON \

I2C\_CLIENT\_MODULE\_PARM(probe, "List of adapter,address pairs to scan " \

"additionally"); \

I2C\_CLIENT\_MODULE\_PARM(ignore, "List of adapter,address pairs not to " \

"scan"); \

static const struct i2c\_client\_address\_data addr\_data = { \

.normal\_i2c = normal\_i2c, \

.probe = probe, \

.ignore = ignore, \

.forces = forces, \

}

#define I2C\_CLIENT\_FORCE\_TEXT \

"List of adapter,address pairs to boldly assume to be present"

由此可知道,addr\_data中的三个成员都是模块参数.在加载模块的时候可以用参数的方式对其赋值.三个模块参数为别为probe,ignore,force.另外需要指出的是normal\_i2c不能以模块参数的方式对其赋值,只能在驱动内部静态指定.

从模块参数的模述看来, probe是指"List of adapter,address pairs to scan additionally"

Ignore是指"List of adapter,address pairs not to scan "

Force是指"List of adapter,address pairs to boldly assume to be present"

事实上,它们里面的数据都是成对出现的.前面一部份表示所在的总线号,ANY\_I2C\_BUS表示任一总线.后一部份表示设备的地址.

现在可以来跟踪i2c\_probe()的代码了.如下:

int i2c\_probe(struct i2c\_adapter \*adapter,

const struct i2c\_client\_address\_data \*address\_data,

int (\*found\_proc) (struct i2c\_adapter \*, int, int))

{

int i, err;

int adap\_id = i2c\_adapter\_id(adapter);

/\* Force entries are done first, and are not affected by ignore

entries \*/

//先扫描force里面的信息,注意它是一个二级指针.ignore里的信息对它是无效的

if (address\_data->forces) {

const unsigned short \* const \*forces = address\_data->forces;

int kind;

for (kind = 0; forces[kind]; kind++) {

for (i = 0; forces[kind][i] != I2C\_CLIENT\_END;

i += 2) {

if (forces[kind][i] == adap\_id

|| forces[kind][i] == ANY\_I2C\_BUS) {

dev\_dbg(&adapter->dev, "found force "

"parameter for adapter %d, "

"addr 0x%02x, kind %d\n",

adap\_id, forces[kind][i + 1],

kind);

err = i2c\_probe\_address(adapter,

forces[kind][i + 1],

kind, found\_proc);

if (err)

return err;

}

}

}

}

/\* Stop here if we can't use SMBUS\_QUICK \*/

//如果adapter不支持quick.不能够遍历这个adapter上面的设备

if (!i2c\_check\_functionality(adapter, I2C\_FUNC\_SMBUS\_QUICK)) {

if (address\_data->probe[0] == I2C\_CLIENT\_END

&& address\_data->normal\_i2c[0] == I2C\_CLIENT\_END)

return 0;

dev\_warn(&adapter->dev, "SMBus Quick command not supported, "

"can't probe for chips\n");

return -1;

}

/\* Probe entries are done second, and are not affected by ignore

entries either \*/

//遍历probe上面的信息.ignore上的信息也对它是没有影响的

for (i = 0; address\_data->probe[i] != I2C\_CLIENT\_END; i += 2) {

if (address\_data->probe[i] == adap\_id

|| address\_data->probe[i] == ANY\_I2C\_BUS) {

dev\_dbg(&adapter->dev, "found probe parameter for "

"adapter %d, addr 0x%02x\n", adap\_id,

address\_data->probe[i + 1]);

err = i2c\_probe\_address(adapter,

address\_data->probe[i + 1],

-1, found\_proc);

if (err)

return err;

}

}

/\* Normal entries are done last, unless shadowed by an ignore entry \*/

//最后遍历normal\_i2c上面的信息.它上面的信息不能在ignore中.

for (i = 0; address\_data->normal\_i2c[i] != I2C\_CLIENT\_END; i += 1) {

int j, ignore;

ignore = 0;

for (j = 0; address\_data->ignore[j] != I2C\_CLIENT\_END;

j += 2) {

if ((address\_data->ignore[j] == adap\_id ||

address\_data->ignore[j] == ANY\_I2C\_BUS)

&& address\_data->ignore[j + 1]

== address\_data->normal\_i2c[i]) {

dev\_dbg(&adapter->dev, "found ignore "

"parameter for adapter %d, "

"addr 0x%02x\n", adap\_id,

address\_data->ignore[j + 1]);

ignore = 1;

break;

}

}

if (ignore)

continue;

dev\_dbg(&adapter->dev, "found normal entry for adapter %d, "

"addr 0x%02x\n", adap\_id,

address\_data->normal\_i2c[i]);

err = i2c\_probe\_address(adapter, address\_data->normal\_i2c[i],

-1, found\_proc);

if (err)

return err;

}

return 0;

}

这段代码很简单,结合代码上面添加的注释应该很好理解.如果匹配成功,则会调用i2c\_probe\_address ().这个函数代码如下:

static int i2c\_probe\_address(struct i2c\_adapter \*adapter, int addr, int kind,

int (\*found\_proc) (struct i2c\_adapter \*, int, int))

{

int err;

/\* Make sure the address is valid \*/

//地址小于0x03或者大于0x77都是不合法的

if (addr < 0x03 || addr > 0x77) {

dev\_warn(&adapter->dev, "Invalid probe address 0x%02x\n",

addr);

return -EINVAL;

}

/\* Skip if already in use \*/

//adapter上已经有这个设备了

if (i2c\_check\_addr(adapter, addr))

return 0;

/\* Make sure there is something at this address, unless forced \*/

//如果kind小于0.检查adapter上是否有这个设备

if (kind < 0) {

if (i2c\_smbus\_xfer(adapter, addr, 0, 0, 0,

I2C\_SMBUS\_QUICK, NULL) < 0)

return 0;

/\* prevent 24RF08 corruption \*/

if ((addr & ~0x0f) == 0x50)

i2c\_smbus\_xfer(adapter, addr, 0, 0, 0,

I2C\_SMBUS\_QUICK, NULL);

}

/\* Finally call the custom detection function \*/

//调用回调函数

err = found\_proc(adapter, addr, kind);

/\* -ENODEV can be returned if there is a chip at the given address

but it isn't supported by this chip driver. We catch it here as

this isn't an error. \*/

if (err == -ENODEV)

err = 0;

if (err)

dev\_warn(&adapter->dev, "Client creation failed at 0x%x (%d)\n",

addr, err);

return err;

}

首先,对传入的参数进行一系列的合法性检查.另外,如果该adapter上已经有了这个地址的设备了.也会返回失败.所有adapter下面的设备都是以adapter->dev为父结点的.因此只需要遍历adapter->dev下面的子设备就可以得到当前地址是不是被占用了.

如果kind < 0.还得要adapter检查该总线是否有这个地址的设备.方法是向这个地址发送一个Read的Quick请求.如果该地址有应答,则说明这个地址上有这个设备.另外还有一种情况是在24RF08设备的特例.

如果adapter上确实有这个设备,就会调用驱动调用时的回调函数.

在上面涉及到了IIC的传输方式,有疑问的可以参考intel ICH5手册的有关smbus部份.

跟踪i2c\_smbus\_xfer().代码如下:

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 &= I2C\_M\_TEN | I2C\_CLIENT\_PEC;

if (adapter->algo->smbus\_xfer) {

mutex\_lock(&adapter->bus\_lock);

res = adapter->algo->smbus\_xfer(adapter,addr,flags,read\_write,

command,size,data);

mutex\_unlock(&adapter->bus\_lock);

} else

res = i2c\_smbus\_xfer\_emulated(adapter,addr,flags,read\_write,

command,size,data);

return res;

}

如果adapter有smbus\_xfer()函数,则直接调用它发送,否则,也就是在adapter不支持smbus协议的情况下,调用i2c\_smbus\_xfer\_emulated()继续处理.

跟进i2c\_smbus\_xfer\_emulated().代码如下:

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)

{

/\* So we need to generate a series of msgs. In the case of writing, we

need to use only one message; when reading, we need two. We initialize

most things with sane defaults, to keep the code below somewhat

simpler. \*/

//写操作只会进行一次交互,而读操作,有时会有两次操作.

//因为有时候读操作要先写command,再从总线上读数据

//在这里为了代码的简洁.使用了两个缓存区,将两种情况统一起来.

unsigned char msgbuf0[I2C\_SMBUS\_BLOCK\_MAX+3];

unsigned char msgbuf1[I2C\_SMBUS\_BLOCK\_MAX+2];

//一般来说,读操作要交互两次.例外的情况我们在下面会接着分析

int num = read\_write == I2C\_SMBUS\_READ?2:1;

//与设备交互的数据,一般在msg[0]存放写入设备的信息,在msb[1]里存放接收到的

//信息.不过也有例外的

//msg[2]的初始化,默认发送缓存区占一个字节,无接收缓存

struct i2c\_msg msg[2] = { { addr, flags, 1, msgbuf0 },

{ addr, flags | I2C\_M\_RD, 0, msgbuf1 }

};

int i;

u8 partial\_pec = 0;

//将要发送的信息copy到发送缓存区的第一字节

msgbuf0[0] = command;

switch(size) {

//quick类型的,其它并不传输有效数据,只是将地址写到总线上,等待应答即可

//所以将发送缓存区长度置为0 .再根据读/写操作,调整msg[0]的标志位

//这类传输只需要一次总线交互

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;

case I2C\_SMBUS\_BYTE:

//BYTE类型指一次写和读只有一个字节.这种情况下,读和写都只会交互一次

//这种类型的读有例外,它读取出来的数据不是放在msg[1]中的,而是存放在msg[0]

if (read\_write == I2C\_SMBUS\_READ) {

/\* Special case: only a read! \*/

msg[0].flags = I2C\_M\_RD | flags;

num = 1;

}

break;

case I2C\_SMBUS\_BYTE\_DATA:

//Byte\_Data是指命令+数据的传输形式.在这种情况下,写只需要一次交互,读却要两次

//第一次将command写到总线上,第二次要转换方向.要将设备地址和read标志写入总线.

//应回答之后再进行read操作

//写操作占两字节,分别是command+data.读操作的有效数据只有一个字节

//交互次数用初始化值就可以了

if (read\_write == I2C\_SMBUS\_READ)

msg[1].len = 1;

else {

msg[0].len = 2;

msgbuf0[1] = data->byte;

}

break;

case I2C\_SMBUS\_WORD\_DATA:

//Word\_Data是指命令+双字节的形式.这种情况跟Byte\_Data的情况类似

//两者相比只是交互的数据大小不同

if (read\_write == I2C\_SMBUS\_READ)

msg[1].len = 2;

else {

msg[0].len=3;

msgbuf0[1] = data->word & 0xff;

msgbuf0[2] = data->word >> 8;

}

break;

case I2C\_SMBUS\_PROC\_CALL:

//Proc\_Call的方式与write 的Word\_Data相似,只不过写完Word\_Data之后,要等待它的应答

//应该它需要交互两次,一次写一次读

num = 2; /\* Special case \*/

read\_write = I2C\_SMBUS\_READ;

msg[0].len = 3;

msg[1].len = 2;

msgbuf0[1] = data->word & 0xff;

msgbuf0[2] = data->word >> 8;

break;

case I2C\_SMBUS\_BLOCK\_DATA:

//Block\_Data:指command+N段数据的情况.

//如果是读操作,它首先要写command到总线,然后再读N段数据.要写的command已经

//放在msg[0]了.现在只需要将msg[1]的标志置I2C\_M\_RECV\_LEN位,msg[1]有效长度为1字节.因为

//adapter驱动会处理好的.现在现在还不知道要传多少段数据.

//对于写的情况:msg[1]照例不需要.将要写的数据全部都放到msb[0]中.相应的也要更新

//msg[0]中的缓存区长度

if (read\_write == I2C\_SMBUS\_READ) {

msg[1].flags |= I2C\_M\_RECV\_LEN;

msg[1].len = 1; /\* block length will be added by

the underlying bus driver \*/

} else {

//data->block[0]表示后面有多少段数据.总长度要加2是因为command+count+N段数据

msg[0].len = data->block[0] + 2;

if (msg[0].len > I2C\_SMBUS\_BLOCK\_MAX + 2) {

dev\_err(&adapter->dev, "smbus\_access called with "

"invalid block write size (%d)\n",

data->block[0]);

return -1;

}

for (i = 1; i < msg[0].len; i++)

msgbuf0[i] = data->block[i-1];

}

break;

case I2C\_SMBUS\_BLOCK\_PROC\_CALL:

//Proc\_Call:表示写完Block\_Data之后,要等它的应答消息它和Block\_Data相比,只是多了一部份应答而已

num = 2; /\* Another special case \*/

read\_write = I2C\_SMBUS\_READ;

if (data->block[0] > I2C\_SMBUS\_BLOCK\_MAX) {

dev\_err(&adapter->dev, "%s called with invalid "

"block proc call size (%d)\n", \_\_func\_\_,

data->block[0]);

return -1;

}

msg[0].len = data->block[0] + 2;

for (i = 1; i < msg[0].len; i++)

msgbuf0[i] = data->block[i-1];

msg[1].flags |= I2C\_M\_RECV\_LEN;

msg[1].len = 1; /\* block length will be added by

the underlying bus driver \*/

break;

case I2C\_SMBUS\_I2C\_BLOCK\_DATA:

//I2c Block\_Data与Block\_Data相似,只不过read的时候,数据长度是预先定义好了的.另外

//与Block\_Data相比,中间不需要传输Count字段.(Count表示数据段数目)

if (read\_write == I2C\_SMBUS\_READ) {

msg[1].len = data->block[0];

} else {

msg[0].len = data->block[0] + 1;

if (msg[0].len > I2C\_SMBUS\_BLOCK\_MAX + 1) {

dev\_err(&adapter->dev, "i2c\_smbus\_xfer\_emulated called with "

"invalid block write size (%d)\n",

data->block[0]);

return -1;

}

for (i = 1; i <= data->block[0]; i++)

msgbuf0[i] = data->block[i];

}

break;

default:

dev\_err(&adapter->dev, "smbus\_access called with invalid size (%d)\n",

size);

return -1;

}

//如果启用了PEC.Quick和I2c Block\_Data是不支持PEC的

i = ((flags & I2C\_CLIENT\_PEC) && size != I2C\_SMBUS\_QUICK

&& size != I2C\_SMBUS\_I2C\_BLOCK\_DATA);

if (i) {

/\* Compute PEC if first message is a write \*/

//如果第一个操作是写操作

if (!(msg[0].flags & I2C\_M\_RD)) {

//如果只是写操作

if (num == 1) /\* Write only \*/

//如果只有写操作,写缓存区要扩充一个字节,用来存放计算出来的PEC

i2c\_smbus\_add\_pec(&msg[0]);

else /\* Write followed by read \*/

//如果后面还有读操作,先计算前面写部份的PEC(注意这种情况下不需要

//扩充写缓存区,因为不需要发送PEC.只会接收到PEC)

partial\_pec = i2c\_smbus\_msg\_pec(0, &msg[0]);

}

/\* Ask for PEC if last message is a read \*/

//如果最后一次是读消息.还要接收到来自slave的PEC.所以接收缓存区要扩充一个字节

if (msg[num-1].flags & I2C\_M\_RD)

msg[num-1].len++;

}

if (i2c\_transfer(adapter, msg, num) < 0)

return -1;

/\* Check PEC if last message is a read \*/

//操作完了之后,如果最后一个操作是PEC的读操作.检验后面的PEC是否正确

if (i && (msg[num-1].flags & I2C\_M\_RD)) {

if (i2c\_smbus\_check\_pec(partial\_pec, &msg[num-1]) < 0)

return -1;

}

//操作完了,现在可以将数据放到data部份返回了.

if (read\_write == I2C\_SMBUS\_READ)

switch(size) {

case I2C\_SMBUS\_BYTE:

data->byte = msgbuf0[0];

break;

case I2C\_SMBUS\_BYTE\_DATA:

data->byte = msgbuf1[0];

break;

case I2C\_SMBUS\_WORD\_DATA:

case I2C\_SMBUS\_PROC\_CALL:

data->word = msgbuf1[0] | (msgbuf1[1] << 8);

break;

case I2C\_SMBUS\_I2C\_BLOCK\_DATA:

for (i = 0; i < data->block[0]; i++)

data->block[i+1] = msgbuf1[i];

break;

case I2C\_SMBUS\_BLOCK\_DATA:

case I2C\_SMBUS\_BLOCK\_PROC\_CALL:

for (i = 0; i < msgbuf1[0] + 1; i++)

data->block[i] = msgbuf1[i];

break;

}

return 0;

}

在这个函数添上了很详细的注释,配和intel的datasheet,应该很容易看懂.在上面的交互过程中,调用了子函数i2c\_transfer().它的代码如下所示:

int i2c\_transfer(struct i2c\_adapter \* adap, struct i2c\_msg \*msgs, int num)

{

int ret;

if (adap->algo->master\_xfer) {

#ifdef DEBUG

for (ret = 0; ret < num; ret++) {

dev\_dbg(&adap->dev, "master\_xfer[%d] %c, addr=0x%02x, "

"len=%d%s\n", ret, (msgs[ret].flags & I2C\_M\_RD)

? 'R' : 'W', msgs[ret].addr, msgs[ret].len,

(msgs[ret].flags & I2C\_M\_RECV\_LEN) ? "+" : "");

}

#endif

if (in\_atomic() || irqs\_disabled()) {

ret = mutex\_trylock(&adap->bus\_lock);

if (!ret)

/\* I2C activity is ongoing. \*/

return -EAGAIN;

} else {

mutex\_lock\_nested(&adap->bus\_lock, adap->level);

}

ret = adap->algo->master\_xfer(adap,msgs,num);

mutex\_unlock(&adap->bus\_lock);

return ret;

} else {

dev\_dbg(&adap->dev, "I2C level transfers not supported\n");

return -ENOSYS;

}

}

因为在这里的同步用的是mutex.首先判断判断是否充许睡眠,如果不允许,尝试获锁.如果获锁失败,则返回,这样的操作是避免进入睡眠,我们在后面也可以看到,实际的传输工作交给了adap->algo->master\_xfer()完成.

在这里,我们终于把i2c\_probe\_address()的执行分析完了,经过这个分析,我们也知道了数据是怎么样传输的.我们接着i2c\_probe()往下看.如果i2c\_probe\_address()成功.说明总线上确实有这样的设备.那么就会调用驱动中的回调函数.在ad7148的驱动中,如下所示:

return i2c\_probe(adapter, &addr\_data, ad7418\_detect);

也就是说,要调用的回调函数是ad7418\_detect().这个函数中我们只分析和i2c框架相关的部份.代码片段如下所示:

static int ad7418\_detect(struct i2c\_adapter \*adapter, int address, int kind)

{

struct i2c\_client \*client;

……

……

client->addr = address;

client->adapter = adapter;

client->driver = &ad7418\_driver;

i2c\_set\_clientdata(client, data);

……

……

if ((err = i2c\_attach\_client(client)))

goto exit\_free;

……

……

}

结合上面关于new-style形式的驱动分析.发现这里走的是同一个套路,即初始化了client.然后调用i2c\_attach\_client().后面的流程就跟上面分析的一样了.只不过,不相同的是,这里clinet已经指定了驱动为ad7418\_driver.应该在注册clinet->dev之后,就不会走bus->match和bus->probe的流程了.

七:i2c dev节点操作

现在来分析上面架构图中的i2c-dev.c中的部份.这个部份为用户空间提供了操作adapter的接口.这部份代码其实对应就晃一个模块.它的初始化函数为:

module\_init(i2c\_dev\_init);

i2c\_dev\_init()代码如下:

static int \_\_init i2c\_dev\_init(void)

{

int res;

printk(KERN\_INFO "i2c /dev entries driver\n");

res = register\_chrdev(I2C\_MAJOR, "i2c", &i2cdev\_fops);

if (res)

goto out;

i2c\_dev\_class = class\_create(THIS\_MODULE, "i2c-dev");

if (IS\_ERR(i2c\_dev\_class))

goto out\_unreg\_chrdev;

res = i2c\_add\_driver(&i2cdev\_driver);

if (res)

goto out\_unreg\_class;

return 0;

out\_unreg\_class:

class\_destroy(i2c\_dev\_class);

out\_unreg\_chrdev:

unregister\_chrdev(I2C\_MAJOR, "i2c");

out:

printk(KERN\_ERR "%s: Driver Initialisation failed\n", \_\_FILE\_\_);

return res;

}

首先为主册了一个主设备号为I2C\_MAJOR(89),操作集为i2cdev\_fops的字符设备.然后注册了一个名为”i2c-dev”的class.之后再注册了一个i2c的driver.如下所示:

res = i2c\_add\_driver(&i2cdev\_driver);

if (res)

goto out\_unreg\_class;

i2cdev\_driver定义如下:

static struct i2c\_driver i2cdev\_driver = {

.driver = {

.name = "dev\_driver",

},

.id = I2C\_DRIVERID\_I2CDEV,

.attach\_adapter = i2cdev\_attach\_adapter,

.detach\_adapter = i2cdev\_detach\_adapter,

.detach\_client = i2cdev\_detach\_client,

};

也就是说,当它注册或者有新的adapter注册后,就会它的attach\_adapter()函数.该函数代码如下:

static int i2cdev\_attach\_adapter(struct i2c\_adapter \*adap)

{

struct i2c\_dev \*i2c\_dev;

int res;

i2c\_dev = get\_free\_i2c\_dev(adap);

if (IS\_ERR(i2c\_dev))

return PTR\_ERR(i2c\_dev);

/\* register this i2c device with the driver core \*/

i2c\_dev->dev = device\_create(i2c\_dev\_class, &adap->dev,

MKDEV(I2C\_MAJOR, adap->nr),

"i2c-%d", adap->nr);

if (IS\_ERR(i2c\_dev->dev)) {

res = PTR\_ERR(i2c\_dev->dev);

goto error;

}

res = device\_create\_file(i2c\_dev->dev, &dev\_attr\_name);

if (res)

goto error\_destroy;

pr\_debug("i2c-dev: adapter [%s] registered as minor %d\n",

adap->name, adap->nr);

return 0;

error\_destroy:

device\_destroy(i2c\_dev\_class, MKDEV(I2C\_MAJOR, adap->nr));

error:

return\_i2c\_dev(i2c\_dev);

return res;

}

这个函数也很简单,首先调用get\_free\_i2c\_dev()分配并初始化了一个struct i2c\_dev结构,使i2c\_dev->adap指向操作的adapter.之后,该i2c\_dev会被链入链表i2c\_dev\_list中.再分别以I2C\_MAJOR, adap->nr为主次设备号创建了一个device.如果此时系统配置了udev或者是hotplug,那么就么在/dev下自动创建相关的设备节点了.

刚才我们说过,所有主设备号为I2C\_MAJOR的设备节点的操作函数是i2cdev\_fops.它的定义如下所示:

static const struct file\_operations i2cdev\_fops = {

.owner = THIS\_MODULE,

.llseek = no\_llseek,

.read = i2cdev\_read,

.write = i2cdev\_write,

.ioctl = i2cdev\_ioctl,

.open = i2cdev\_open,

.release = i2cdev\_release,

};

7.1:i2c dev的open操作

Open操作对应的函数为i2cdev\_open().代码如下:

static int i2cdev\_open(struct inode \*inode, struct file \*file)

{

unsigned int minor = iminor(inode);

struct i2c\_client \*client;

struct i2c\_adapter \*adap;

struct i2c\_dev \*i2c\_dev;

//以次设备号从i2c\_dev\_list链表中取得i2c\_dev

i2c\_dev = i2c\_dev\_get\_by\_minor(minor);

if (!i2c\_dev)

return -ENODEV;

//以apapter的总线号从i2c\_adapter\_idr中找到adapter

adap = i2c\_get\_adapter(i2c\_dev->adap->nr);

if (!adap)

return -ENODEV;

/\* This creates an anonymous i2c\_client, which may later be

\* pointed to some address using I2C\_SLAVE or I2C\_SLAVE\_FORCE.

\*

\* This client is \*\* NEVER REGISTERED \*\* with the driver model

\* or I2C core code!! It just holds private copies of addressing

\* information and maybe a PEC flag.

\*/

//分配并初始化一个i2c\_client结构

client = kzalloc(sizeof(\*client), GFP\_KERNEL);

if (!client) {

i2c\_put\_adapter(adap);

return -ENOMEM;

}

snprintf(client->name, I2C\_NAME\_SIZE, "i2c-dev %d", adap->nr);

client->driver = &i2cdev\_driver;

//clinet->adapter指向操作的adapter

client->adapter = adap;

//关联到file

file->private\_data = client;

return 0;

}

注意这里分配并初始化了一个struct i2c\_client结构.但是没有注册这个clinet.此外,这个函数中还有一个比较奇怪的操作.不是在前面已经将i2c\_dev->adap指向要操作的adapter么?为什么还要以adapter->nr为关键字从i2c\_adapter\_idr去找这个操作的adapter呢?注意了,调用i2c\_get\_adapter()从总线号nr找到操作的adapter的时候,还会增加module的引用计数.这样可以防止模块意外被释放掉.也许有人会有这样的疑问,那 i2c\_dev->adap->nr操作,如果i2c\_dev->adap被释放掉的话,不是一样会引起系统崩溃么?这里因为,在i2cdev\_attach\_adapter()间接的增加了一次adapter的一次引用计数.如下:

tatic int i2cdev\_attach\_adapter(struct i2c\_adapter \*adap)

{

．．．．．．

i2c\_dev->dev = device\_create(i2c\_dev\_class, &adap->dev,

MKDEV(I2C\_MAJOR, adap->nr),

"i2c-%d", adap->nr);

．．．．．．

}

看到了么，i2c\_dev内嵌的device是以adap->dev为父结点,在device\_create()中会增次adap->dev的一次引用计数.

好了,open()操作到此就完成了.

7.2:read操作

Read操作对应的操作函数如下示:

static ssize\_t i2cdev\_read (struct file \*file, char \_\_user \*buf, size\_t count,

loff\_t \*offset)

{

char \*tmp;

int ret;

struct i2c\_client \*client = (struct i2c\_client \*)file->private\_data;

if (count > 8192)

count = 8192;

tmp = kmalloc(count,GFP\_KERNEL);

if (tmp==NULL)

return -ENOMEM;

pr\_debug("i2c-dev: i2c-%d reading %zd bytes.\n",

iminor(file->f\_path.dentry->d\_inode), count);

ret = i2c\_master\_recv(client,tmp,count);

if (ret >= 0)

ret = copy\_to\_user(buf,tmp,count)?-EFAULT:ret;

kfree(tmp);

return ret;

}

首先从file结构中取得struct i2c\_clinet.然后在kernel同分配相同长度的缓存区,随之调用i2c\_master\_recv()从设备中读取数据.再将读取出来的数据copy到用户空间中.

I2c\_master\_recv()代码如下:

int i2c\_master\_recv(struct i2c\_client \*client, char \*buf ,int count)

{

struct i2c\_adapter \*adap=client->adapter;

struct i2c\_msg msg;

int ret;

msg.addr = client->addr;

msg.flags = client->flags & I2C\_M\_TEN;

msg.flags |= I2C\_M\_RD;

msg.len = count;

msg.buf = buf;

ret = i2c\_transfer(adap, &msg, 1);

/\* If everything went ok (i.e. 1 msg transmitted), return #bytes

transmitted, else error code. \*/

return (ret == 1) ? count : ret;

}

看完前面的代码之后,这个函数应该很简单了,就是为读操作初始化了一个i2c\_msg.然后调用i2c\_tanster().代码中的client->flags & I2C\_M\_TEN表示adapter是否采用10位寻址的方式.在这里就不再详细分析了.

另外,有人可能看出了一个问题.这里clinet->addr是从哪来的呢?对,在read之前应该还要有一步操作来设置clinet->addr的值.这个过程是ioctl的操作.ioctl可以设置PEC标志,重试次数,超时时间,和发送接收数据等,我们在这里只看一下clinet->addr的设置.代码片段如下示:

static int i2cdev\_ioctl(struct inode \*inode, struct file \*file,

unsigned int cmd, unsigned long arg)

{

．．．．．．

．．．．．．

switch ( cmd ) {

case I2C\_SLAVE:

case I2C\_SLAVE\_FORCE:

/\* NOTE: devices set up to work with "new style" drivers

\* can't use I2C\_SLAVE, even when the device node is not

\* bound to a driver. Only I2C\_SLAVE\_FORCE will work.

\*

\* Setting the PEC flag here won't affect kernel drivers,

\* which will be using the i2c\_client node registered with

\* the driver model core. Likewise, when that client has

\* the PEC flag already set, the i2c-dev driver won't see

\* (or use) this setting.

\*/

if ((arg > 0x3ff) ||

(((client->flags & I2C\_M\_TEN) == 0) && arg > 0x7f))

return -EINVAL;

if (cmd == I2C\_SLAVE && i2cdev\_check\_addr(client->adapter, arg))

return -EBUSY;

/\* REVISIT: address could become busy later \*/

client->addr = arg;

return 0;

．．．．．．

．．．．．．

}

由此可见,调用I2C\_SLAVE或者I2C\_SLAVE\_FORCE的Ioctl就会设置clinet->addr.另外,注释中也说得很清楚了.如果是I2C\_SLAVE的话,还会调用其所长i2cdev\_check\_addr().进行地址检查,如果adapter已经关联到这个地址的设备,就会检查失败.

7.2:write操作

Write操作如下所示:

static ssize\_t i2cdev\_write (struct file \*file, const char \_\_user \*buf, size\_t count,

loff\_t \*offset)

{

int ret;

char \*tmp;

struct i2c\_client \*client = (struct i2c\_client \*)file->private\_data;

if (count > 8192)

count = 8192;

tmp = kmalloc(count,GFP\_KERNEL);

if (tmp==NULL)

return -ENOMEM;

if (copy\_from\_user(tmp,buf,count)) {

kfree(tmp);

return -EFAULT;

}

pr\_debug("i2c-dev: i2c-%d writing %zd bytes.\n",

iminor(file->f\_path.dentry->d\_inode), count);

ret = i2c\_master\_send(client,tmp,count);

kfree(tmp);

return ret;

}

该操作比较简单,就是将用户空间的数据发送到i2c 设备.

八:小结

在本节中,分析了i2c的框架设计.这个框架大体上沿用了Linux的设备驱动框架,不过之中又做了很多变通.在之后的分析中,会分别举一个adapter和i2c device的例子来详细描述一下有关i2c driver的设计.