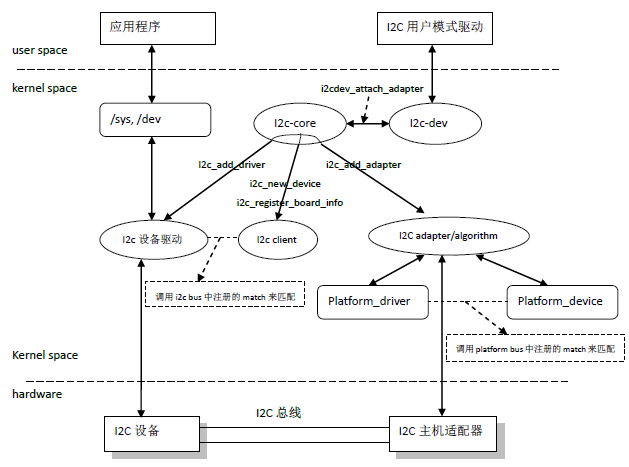
# Linux驱动\_I2C子系统

## 1. I2C子系统架构

### 1.1 系统架构图



### 1.2 三大组成部分

**I2C核心(i2c-core)**  
I2C核心提供了I2C总线驱动和设备驱动的注册、注销方法，I2C通信方法(algorithm)上层的、与具体适配器无关的代码以及探测设备、检测设备地址的上层代码等。

**I2C总线驱动(I2C adapter/Algo driver)**I2C总线驱动是I2C适配器的软件实现，提供I2C适配器与从设备间完成数据通信的能力。I2C总线驱动由i2c\_adapter和i2c\_algorithm来描述。

**I2C客户驱动程序(I2C client driver)**  
I2C客户驱动是对I2C从设备的软件实现，一个具体的I2C客户驱动包括两个部分：一部分是i2c\_driver，用于将设备挂接于i2c总线；另一部分是设备本身的驱动。  
I2C客户驱动程序由i2c\_driver和i2c\_client来描述。

所有的I2C驱动代码位于drivers/i2c目录下：

I2c**-**core**.**c    实现I2C核心的功能

I2c**-**dev**.**c   通用的从设备驱动，应用层操作i2c设备

Chips       特定的I2C设备驱动

Busses      I2C适配器的驱动

Algos       实现了一些I2C总线适配器的algorithm

从上面的图我们可以看到两种编写驱动方法，一种是利用系统提供的i2c-dev.c来实现一个i2c适配器的设备文件，然后通过在应用层操作I2C适配器来控制I2C设备；另一种是为I2C从设备独立编写一个设备驱动，不需要i2c-dev.c文件。

## 2. I2C设备代码编写

### 2.1 AT24c08的I2C代码

#### 2.1.1 i2c\_new\_device

##### 2.1.1.1 i2c\_client

创建一个i2c设备，该设备名称为at24c08，i2c地址为0x50。

//所支持的i2c设备的列表

static struct i2c\_board\_info at24cxx\_info **=** **{**

//一项代表一个支持的设备，它的名字叫做“at24c08”，器件地址是0x50

I2C\_BOARD\_INFO**(**"at24c08"**,** 0x50**),**

**};**

static struct i2c\_client **\***at24cxx\_client**;**

static int at24cxx\_dev\_init**(**void**)**

**{**

//分配一个适配器的指针

struct i2c\_adapter **\***i2c\_adap**;**

//调用core层的函数，获得一个i2c总线。这里我们已经知道新增的器件挂接在编号为0的i2c总线上

i2c\_adap **=** i2c\_get\_adapter**(**0**);**

//把i2c适配器和新增的I2C器件关联起来，这个用了i2c总线0，地址是0x50。这就组成了一个客户端。

at24cxx\_client **=** i2c\_new\_device**(**i2c\_adap**,** **&**at24cxx\_info**);**

//增加i2c适配器的引用

i2c\_put\_adapter**(**i2c\_adap**);**

**return** 0**;**

**}**

static void at24cxx\_dev\_exit**(**void**)**

**{**

i2c\_unregister\_device**(**at24cxx\_client**);**

**}**

module\_init**(**at24cxx\_dev\_init**);**

module\_exit**(**at24cxx\_dev\_exit**);**

MODULE\_LICENSE**(**"GPL"**);**

##### 2.1.1.2 i2c\_driver

创建了i2c设备，接下来需要创建设备对应的驱动程序。

static const struct i2c\_device\_id at24cxx\_id\_table**[]** **=** **{**

**{** "at24c08"**,** 0 **},**

**{}**

**};**

/\* 1. 分配/设置i2c\_driver \*/

static struct i2c\_driver at24cxx\_driver **=** **{**

**.**driver **=** **{**

**.**name **=** "100ask"**,**

**.**owner **=** THIS\_MODULE**,**

**},**

**.**probe **=** at24cxx\_probe**,**

**.**remove **=** \_\_devexit\_p**(**at24cxx\_remove**),**

**.**id\_table **=** at24cxx\_id\_table**,**

**};**

static int at24cxx\_drv\_init**(**void**)**

**{**

/\* 2. 注册i2c\_driver \*/

i2c\_add\_driver**(&**at24cxx\_driver**);**

**return** 0**;**

**}**

static void at24cxx\_drv\_exit**(**void**)**

**{**

i2c\_del\_driver**(&**at24cxx\_driver**);**

**}**

当i2c driver和i2c client的名称都为at24c08时，probe函数调用。

at24cxx\_probe函数创建了字符设备/dev/at24cxx，并把i2c\_client的信息保存在全局变量中。

static struct i2c\_client **\***at24cxx\_client**;**

static int \_\_devinit at24cxx\_probe**(**struct i2c\_client **\***client**,**

const struct i2c\_device\_id **\***id**)**

**{**

at24cxx\_client **=** client**;**

major **=** register\_chrdev**(**0**,** "at24cxx"**,** **&**at24cxx\_fops**);**

class **=** class\_create**(**THIS\_MODULE**,** "at24cxx"**);**

device\_create**(**class**,** **NULL,** MKDEV**(**major**,** 0**),** **NULL,** "at24cxx"**);**

/\* /dev/at24cxx \*/

**return** 0**;**

**}**

字符设备需要完成读写代码，这里对应i2c的读写过程。

/\* 传入: buf[0] : addr

\* 输出: buf[0] : data

\*/

static ssize\_t at24cxx\_read**(**struct file **\*** file**,** char \_\_user **\***buf**,** size\_t count**,** loff\_t **\***off**)**

**{**

unsigned char addr**,** data**;**

copy\_from\_user**(&**addr**,** buf**,** 1**);**

data **=** i2c\_smbus\_read\_byte\_data**(**at24cxx\_client**,** addr**);**

copy\_to\_user**(**buf**,** **&**data**,** 1**);**

**return** 1**;**

**}**

/\* buf[0] : addr

\* buf[1] : data

\*/

static ssize\_t at24cxx\_write**(**struct file **\***file**,** const char \_\_user **\***buf**,** size\_t count**,** loff\_t **\***off**)**

**{**

unsigned char ker\_buf**[**2**];**

unsigned char addr**,** data**;**

copy\_from\_user**(**ker\_buf**,** buf**,** 2**);**

addr **=** ker\_buf**[**0**];**

data **=** ker\_buf**[**1**];**

printk**(**"addr = 0x%02x, data = 0x%02x\n"**,** addr**,** data**);**

**if** **(!**i2c\_smbus\_write\_byte\_data**(**at24cxx\_client**,** addr**,** data**)**

**return** 2**;**

**else**

**return** **-**EIO**;**

**}**

static struct file\_operations at24cxx\_fops **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**read **=** at24cxx\_read**,**

**.**write **=** at24cxx\_write**,**

**};**

从代码上看，读写直接调用了i2c\_smbus\_read\_byte\_data和i2c\_smbus\_write\_byte\_data通用函数，通过i2c\_client中的i2c地址完成i2c的操作。

##### 2.1.1.3 应用程序测试代码

/\* i2c\_test r addr

\* i2c\_test w addr val

\*/

void print\_usage**(**char **\***file**)**

**{**

printf**(**"%s r addr\n"**,** file**);**

printf**(**"%s w addr val\n"**,** file**);**

**}**

int main**(**int argc**,** char **\*\***argv**)**

**{**

int fd**;**

unsigned char buf**[**2**];**

**if** **((**argc **!=** 3**)** **&&** **(**argc **!=** 4**))**

**{**

print\_usage**(**argv**[**0**]);**

**return** **-**1**;**

**}**

fd **=** open**(**"/dev/at24cxx"**,** O\_RDWR**);**

**if** **(**fd **<** 0**)**

**{**

printf**(**"can't open /dev/at24cxx\n"**);**

**return** **-**1**;**

**}**

**if** **(**strcmp**(**argv**[**1**],** "r"**)** **==** 0**)**

**{**

buf**[**0**]** **=** strtoul**(**argv**[**2**],** **NULL,** 0**);**

read**(**fd**,** buf**,** 1**);**

printf**(**"data: %c, %d, 0x%2x\n"**,** buf**[**0**],** buf**[**0**],** buf**[**0**]);**

**}**

**else** **if** **((**strcmp**(**argv**[**1**],** "w"**)** **==** 0**)** **&&** **(**argc **==** 4**))**

**{**

buf**[**0**]** **=** strtoul**(**argv**[**2**],** **NULL,** 0**);**

buf**[**1**]** **=** strtoul**(**argv**[**3**],** **NULL,** 0**);**

**if** **(**write**(**fd**,** buf**,** 2**)** **!=** 2**)**

printf**(**"write err, addr = 0x%02x, data = 0x%02x\n"**,** buf**[**0**],** buf**[**1**]);**

**}**

**else**

**{**

print\_usage**(**argv**[**0**]);**

**return** **-**1**;**

**}**

**return** 0**;**

**}**

#### 2.1.2 i2c\_new\_probed\_device方式

i2c\_new\_device注册设备认为设备肯定存在，而i2c\_new\_probed\_device对于"已经识别出来的设备"(probed\_device)，才会创建("new")。相关过程伪代码如下：

i2c\_new\_probed\_device

probe(adap, addr\_list[i]) /\* 确定设备是否真实存在 \*/

info->addr = addr\_list[i];

i2c\_new\_device(adap, info);

使用i2c\_new\_probed\_device方式的i2c\_client注册代码如下：

static struct i2c\_client **\***at24cxx\_client**;**

static const unsigned short addr\_list**[]** **=** **{** 0x60**,** 0x50**,** I2C\_CLIENT\_END **};**

static int at24cxx\_dev\_init**(**void**)**

**{**

struct i2c\_adapter **\***i2c\_adap**;**

struct i2c\_board\_info at24cxx\_info**;**

memset**(&**at24cxx\_info**,** 0**,** **sizeof(**struct i2c\_board\_info**));**

strlcpy**(**at24cxx\_info**.**type**,** "at24c08"**,** I2C\_NAME\_SIZE**);**

i2c\_adap **=** i2c\_get\_adapter**(**0**);**

at24cxx\_client **=** i2c\_new\_probed\_device**(**i2c\_adap**,** **&**at24cxx\_info**,** addr\_list**,** **NULL);**

i2c\_put\_adapter**(**i2c\_adap**);**

**if** **(**at24cxx\_client**)**

**return** 0**;**

**else**

**return** **-**ENODEV**;**

**}**

static void at24cxx\_dev\_exit**(**void**)**

**{**

i2c\_unregister\_device**(**at24cxx\_client**);**

**}**

module\_init**(**at24cxx\_dev\_init**);**

module\_exit**(**at24cxx\_dev\_exit**);**

MODULE\_LICENSE**(**"GPL"**);**

#### 2.1.3 all\_adapter\_detect

前面的方法都要事先确定适配器(I2C总线,I2C控制器)，如果我事先并不知道这个I2C设备在哪个适配器上，怎么办？去class表示的所有的适配器上查找，如果有一些I2C设备的地址是一样，怎么继续分配它是哪一款？用detect函数。

static struct i2c\_driver at24cxx\_driver = {

.class = I2C\_CLASS\_HWMON, /\* 表示去哪些适配器上找设备 \*/

.driver = {

.name = "100ask",

.owner = THIS\_MODULE,

},

.probe = at24cxx\_probe,

.remove = \_\_devexit\_p(at24cxx\_remove),

.id\_table = at24cxx\_id\_table,

.detect = at24cxx\_detect, /\* 用这个函数来检测设备确实存在 \*/

.address\_list = addr\_list, /\* 这些设备的地址 \*/

};

去"class表示的这一类"I2C适配器，用"detect函数"来确定能否找到"address\_list里的设备",如果能找到就调用i2c\_new\_device来注册i2c\_client, 这会和i2c\_driver的id\_table比较，如果匹配，调用probe。

相关代码如下：

static int \_\_devinit at24cxx\_probe**(**struct i2c\_client **\***client**,**

const struct i2c\_device\_id **\***id**)**

**{**

printk**(**"%s %s %d\n"**,** \_\_FILE\_\_**,** \_\_FUNCTION\_\_**,** \_\_LINE\_\_**);**

**return** 0**;**

**}**

static int \_\_devexit at24cxx\_remove**(**struct i2c\_client **\***client**)**

**{**

printk**(**"%s %s %d\n"**,** \_\_FILE\_\_**,** \_\_FUNCTION\_\_**,** \_\_LINE\_\_**);**

**return** 0**;**

**}**

static const struct i2c\_device\_id at24cxx\_id\_table**[]** **=** **{**

**{** "at24c08"**,** 0 **},**

**{}**

**};**

static int at24cxx\_detect**(**struct i2c\_client **\***client**,**

struct i2c\_board\_info **\***info**)**

**{**

/\* 能运行到这里, 表示该addr的设备是存在的

\* 但是有些设备单凭地址无法分辨(A芯片的地址是0x50, B芯片的地址也是0x50)

\* 还需要进一步读写I2C设备来分辨是哪款芯片

\* detect就是用来进一步分辨这个芯片是哪一款，并且设置info->type

\*/

printk**(**"at24cxx\_detect : addr = 0x%x\n"**,** client**->**addr**);**

/\* 进一步判断是哪一款 \*/

strlcpy**(**info**->**type**,** "at24c08"**,** I2C\_NAME\_SIZE**);**

**return** 0**;**

**}**

static const unsigned short addr\_list**[]** **=** **{** 0x60**,** 0x50**,** I2C\_CLIENT\_END **};**

/\* 1. 分配/设置i2c\_driver \*/

static struct i2c\_driver at24cxx\_driver **=** **{**

**.**class **=** I2C\_CLASS\_HWMON**,** /\* 表示去哪些适配器上找设备 \*/

**.**driver **=** **{**

**.**name **=** "100ask"**,**

**.**owner **=** THIS\_MODULE**,**

**},**

**.**probe **=** at24cxx\_probe**,**

**.**remove **=** \_\_devexit\_p**(**at24cxx\_remove**),**

**.**id\_table **=** at24cxx\_id\_table**,**

**.**detect **=** at24cxx\_detect**,** /\* 用这个函数来检测设备确实存在 \*/

**.**address\_list **=** addr\_list**,** /\* 这些设备的地址 \*/

**};**

static int at24cxx\_drv\_init**(**void**)**

**{**

/\* 2. 注册i2c\_driver \*/

i2c\_add\_driver**(&**at24cxx\_driver**);**

**return** 0**;**

**}**

static void at24cxx\_drv\_exit**(**void**)**

**{**

i2c\_del\_driver**(&**at24cxx\_driver**);**

**}**

module\_init**(**at24cxx\_drv\_init**);**

module\_exit**(**at24cxx\_drv\_exit**);**

MODULE\_LICENSE**(**"GPL"**);**

#### 2.1.4 用户空间创建i2c设备

如果编写驱动之前确实无法得知i2c设备的地址（甚至连地址列表都不得而知），那就需要系统运行后从用户空间输入了。  
用户空间通过两个sysfs属性文件来建立和删除i2c\_client：new\_device和delete\_device。这两个文件都是只写的。  
new\_device有两个参数：i2c设备的名字（字符串）和地址（以0x开头的16进制数）。delete\_device只有一个参数，那就是设备的地址。

echo at24c08 0x50 > /sys/class/i2c-adapter/i2c-0/new\_device

echo 0x50 > /sys/class/i2c-adapter/i2c-0/delete\_device

#### 2.1.5 用户空间操作i2c设备

当然也可以在应用层创建i2c读写操作，示例代码如下：

//打开/dev/i2c-0

fd **=** open**(**"/dev/i2c-0"**,** O\_RDWR**);**

//设置i2c从机地址

ioctl**(**fd**,** I2C\_SLAVE**,** dev\_addr**)**

//读接口 addr为寄存器地址

i2c\_smbus\_read\_word\_data**(**fd**,** addr**)**

//写接口

i2c\_smbus\_write\_byte\_data**(**fd**,** addr**,** data**);**

//读写接口包含在linux/i2c-dev.h中

static inline \_\_s32 i2c\_smbus\_read\_word\_data**(**int file**,** \_\_u8 command**)**

**{**

union i2c\_smbus\_data data**;**

**if** **(**i2c\_smbus\_access**(**file**,**I2C\_SMBUS\_READ**,**command**,**

I2C\_SMBUS\_WORD\_DATA**,&**data**))**

**return** **-**1**;**

**else**

**return** 0x0FFFF **&** data**.**word**;**

**}**

static inline \_\_s32 i2c\_smbus\_write\_byte\_data**(**int file**,** \_\_u8 command**,**

\_\_u8 value**)**

**{**

union i2c\_smbus\_data data**;**

data**.**byte **=** value**;**

**return** i2c\_smbus\_access**(**file**,**I2C\_SMBUS\_WRITE**,**command**,**

I2C\_SMBUS\_BYTE\_DATA**,** **&**data**);**

**}**

static inline \_\_s32 i2c\_smbus\_access**(**int file**,** char read\_write**,** \_\_u8 command**,** int size**,** union i2c\_smbus\_data **\***data**)**

**{**

struct i2c\_smbus\_ioctl\_data args**;**

args**.**read\_write **=** read\_write**;**

args**.**command **=** command**;**

args**.**size **=** size**;**

args**.**data **=** data**;**

**return** ioctl**(**file**,**I2C\_SMBUS**,&**args**);**

**}**

也可以使用i2c-tool工具。

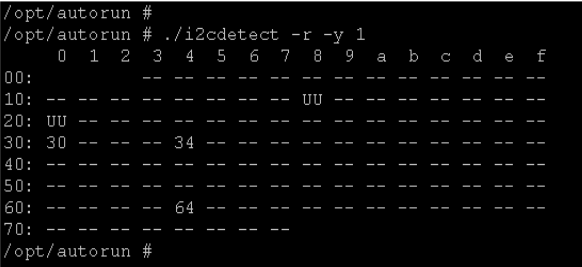
从开源网站http://dl.lm-sensors.org/i2c-tools/releases/下载i2c-tools，最有用的是tools/文件夹，接下来修改Makefile，主要是修改交叉编译器然后保存退出。直接make，进行编译。编译完成后在tools/文件夹下会有如下几个可执行文件，i2cdetect, i2cdump, i2cget, i2cset。

用i2cdetect检测有几组i2c总线在系统上，输入：./i2cdetect –l



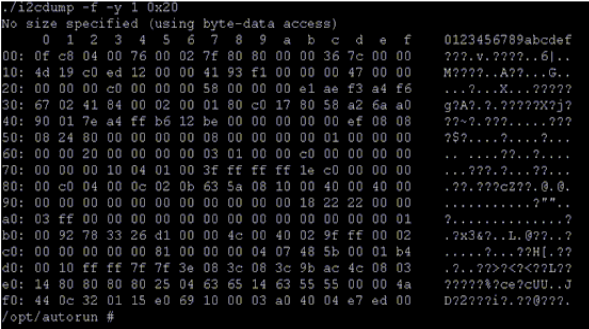
由上图可知，系统中存在两组总线分别i2c-1和i2c-2。

用i2cdetect检测挂载在i2c总线上器件，输入 ./i2cdetect -r -y 1（检测i2c-1上的挂载情况）



由上图可知，0x18,0x20，0x30,0x34， 0x64地址有挂载。

用i2cdump查看器件所有寄存器的值，这个很有用，输入 ./i2cdump -f -y 1 0x20  
该命令会到处0x00到0xFF地址范围内的所有数据。

用i2cset设置单个寄存器值，用i2cget读取单个寄存器值。

**./**i2cset **-**f **-**y 1 0x20 0x77 0x3f

//（设置i2c-1上0x20器件的0x77寄存器值为0x3f）

**./**i2cget **-**f **-**y 1 0x20 0x77

//（读取i2c-1上0x20器件的0x77寄存器值）

## 3. 简易的i2c adapter代码编写

### 3.1 i2c总线硬件初始化

s3c2440\_i2c\_regs **=** ioremap**(**0x54000000**,** **sizeof(**struct s3c2440\_i2c\_regs**));**

s3c2440\_i2c\_init**();**

request\_irq**(**IRQ\_IIC**,** s3c2440\_i2c\_xfer\_irq**,** 0**,** "s3c2440-i2c"**,** **NULL);**

init\_waitqueue\_head**(&**s3c2440\_i2c\_xfer\_data**.**wait**);**

static void s3c2440\_i2c\_init**(**void**)**

**{**

struct clk **\***clk**;**

clk **=** clk\_get**(NULL,** "i2c"**);**

clk\_enable**(**clk**);**

// 选择引脚功能：GPE15:IICSDA, GPE14:IICSCL

s3c\_gpio\_cfgpin**(**S3C2410\_GPE**(**14**),** S3C2410\_GPE14\_IICSCL**);**

s3c\_gpio\_cfgpin**(**S3C2410\_GPE**(**15**),** S3C2410\_GPE15\_IICSDA**);**

/\* bit[7] = 1, 使能ACK

\* bit[6] = 0, IICCLK = PCLK/16

\* bit[5] = 1, 使能中断

\* bit[3:0] = 0xf, Tx clock = IICCLK/16

\* PCLK = 50MHz, IICCLK = 3.125MHz, Tx Clock = 0.195MHz

\*/

s3c2440\_i2c\_regs**->**iiccon **=** **(**1**<<**7**)** **|** **(**0**<<**6**)** **|** **(**1**<<**5**)** **|** **(**0xf**);** // 0xaf

s3c2440\_i2c\_regs**->**iicadd **=** 0x10**;** // S3C24xx slave address = [7:1]

s3c2440\_i2c\_regs**->**iicstat **=** 0x10**;** // I2C串行输出使能(Rx/Tx)

**}**

### 3.2 分配i2c adapter

static u32 s3c2440\_i2c\_func**(**struct i2c\_adapter **\***adap**)**

**{**

**return** I2C\_FUNC\_I2C **|** I2C\_FUNC\_SMBUS\_EMUL **|** I2C\_FUNC\_PROTOCOL\_MANGLING**;**

**}**

static int s3c2440\_i2c\_xfer**(**struct i2c\_adapter **\***adap**,**

struct i2c\_msg **\***msgs**,** int num**)**

**{**

**...........**

**}**

static const struct i2c\_algorithm s3c2440\_i2c\_algo **=** **{**

// .smbus\_xfer = ,

//master\_xfer表示i2c读写数据的通用函数

**.**master\_xfer **=** s3c2440\_i2c\_xfer**,**

//functionality表示该i2c适配器具备的能力

**.**functionality **=** s3c2440\_i2c\_func**,**

**};**

static struct i2c\_adapter s3c2440\_i2c\_adapter **=** **{**

**.**name **=** "s3c2440\_100ask"**,**

**.**algo **=** **&**s3c2440\_i2c\_algo**,**

**.**owner **=** THIS\_MODULE**,**

**};**

### 3.3 注册i2c adapter

i2c\_add\_adapter**(&**s3c2440\_i2c\_adapter**);**

## 4. i2c子系统框架代码分析

### 4.1 i2c子系统初始化

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

**{**

bus\_register**(&**i2c\_bus\_type**);**

i2c\_adapter\_compat\_class **=** class\_compat\_register**(**"i2c-adapter"**);**

i2c\_add\_driver**(&**dummy\_driver**);**

**}**

#### 4.1.1 bus\_register

注册i2c总线。

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

**.**name **=** "i2c"**,**

**.**match **=** i2c\_device\_match**,**

**.**probe **=** i2c\_device\_probe**,**

**.**remove **=** i2c\_device\_remove**,**

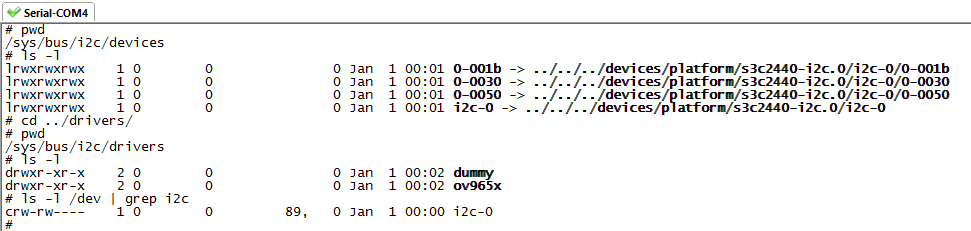
**.**shutdown **=** i2c\_device\_shutdown**,**

**.**pm **=** **&**i2c\_device\_pm\_ops**,**

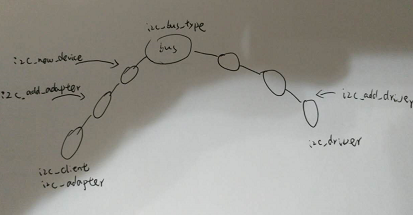
**};**

i2c总线中的match函数用于驱动和设备之间的匹配。

该函数执行完会在/sys/bus目录下创建i2c子目录，并在i2c子目录下创建devices和drivers两个目录，以后注册到i2c总线上的设备和驱动会分别放在这两个目录。



i2c\_bus\_type总线-设备-驱动模型如下图：



#### 4.1.2 class\_compat\_register

在/sys/class/目录下创建i2c-adapter子类目录。



#### 4.1.3 dummy\_driver

i2c\_add\_driver(&dummy\_driver)在i2c-bus上注册驱动，该函数执行成功后会在/sys/bus/i2c/drivers目录下创建.driver.name = "dummy"为名字的目录。

http://images2015.cnblogs.com/blog/780757/201612/780757-20161224231624964-1782586567.png

### 4.2 i2c adapter添加

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**);**

//找到一个合适的id对应adapter这个结构体

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**;**

**}**

//设置adpter结构体对应id

adapter**->**nr **=** id**;**

**return** i2c\_register\_adapter**(**adapter**);**

**}**

实际上i2c\_register\_adapter才是真正的注册函数，该函数会调用device\_register在i2c总线上注册device设备，并遍历i2c bus type上的driver，检查该driver中的地址列表中是否有地址会响应该i2c总线的操作，如果有，则会调用i2c\_new\_device在i2c bus type上创建i2c client设备。

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

**{**

int res **=** 0**;**

//检查总线是否创建成功

**if** **(**unlikely**(**WARN\_ON**(!**i2c\_bus\_type**.**p**)))** **{**

res **=** **-**EAGAIN**;**

**goto** out\_list**;**

**}**

//检查adpater的名称是否存在

**if** **(**unlikely**(**adap**->**name**[**0**]** **==** '\0'**))** **{**

pr\_err**(**"i2c-core: Attempt to register an adapter with "

"no name!\n"**);**

**return** **-**EINVAL**;**

**}**

//检查adapter的算法是否存在

**if** **(**unlikely**(!**adap**->**algo**))** **{**

pr\_err**(**"i2c-core: Attempt to register adapter '%s' with "

"no algo!\n"**,** adap**->**name**);**

**return** **-**EINVAL**;**

**}**

//初始化相关锁和链表

rt\_mutex\_init**(&**adap**->**bus\_lock**);**

mutex\_init**(&**adap**->**userspace\_clients\_lock**);**

INIT\_LIST\_HEAD**(&**adap**->**userspace\_clients**);**

/\* Set default timeout to 1 second if not already set \*/

**if** **(**adap**->**timeout **==** 0**)**

adap**->**timeout **=** HZ**;**

//platform\_device\_register 是向platform这条虚拟总线上注册一个设备

//device\_register是将设备注册到所在的物理总线上去

//sys/bus/i2c/devices/目录出现i2c-0目录，目录下有new\_device和delete\_device文件

dev\_set\_name**(&**adap**->**dev**,** "i2c-%d"**,** adap**->**nr**);**

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

adap**->**dev**.**type **=** **&**i2c\_adapter\_type**;**

//i2c总线上注册adapter设备

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

**if** **(**res**)**

**goto** out\_list**;**

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

#ifdef CONFIG\_I2C\_COMPAT

//在/sys/class/i2c-adapter/i2c-0链接，里面有new\_device和delete\_device文件

res **=** class\_compat\_create\_link**(**i2c\_adapter\_compat\_class**,** **&**adap**->**dev**,**

adap**->**dev**.**parent**);**

**if** **(**res**)**

dev\_warn**(&**adap**->**dev**,**

"Failed to create compatibility class link\n"**);**

#endif

//i2c\_register\_board\_info()函数用于往\_\_i2c\_board\_list这条链表添加一条i2c设备信息，

//在i2c adapter注册的时候，会扫描\_\_i2c\_board\_list链表，然后调用i2c\_new\_device()函数来注册i2c设备，

//注意，要先于i2c adapter注册之前就添加好i2c设备信息，

//否则会出现调用了i2c\_register\_board\_info()函数，而设备不能注册的情况。

//如果程序调用了i2c\_register\_board\_info注册i2c设备，那么\_\_i2c\_first\_dynamic\_bus\_num就会更新

//如果当前的adapter<\_\_i2c\_first\_dynamic\_bus\_num，则需要扫描\_\_i2c\_board\_list链表，将添加到链表的

//i2c设备调用i2c\_new\_device

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

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

//遍历i2c bus上的注册的i2c\_driver,并探测该driver是否在该总线上存在设备

mutex\_lock**(&**core\_lock**);**

bus\_for\_each\_drv**(&**i2c\_bus\_type**,** **NULL,** adap**,** \_\_process\_new\_adapter**);**

mutex\_unlock**(&**core\_lock**);**

**return** 0**;**

out\_list**:**

mutex\_lock**(&**core\_lock**);**

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

mutex\_unlock**(&**core\_lock**);**

**return** res**;**

**}**

如果在i2c adpter注册之前调用了i2c\_register\_board\_info，那么i2c\_scan\_static\_board\_info会扫描注册的board\_info并调用i2c\_new\_device创建i2c\_client。

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

**{**

struct i2c\_devinfo **\***devinfo**;**

down\_read**(&**\_\_i2c\_board\_lock**);**

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

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

**&&** **!**i2c\_new\_device**(**adapter**,**

**&**devinfo**->**board\_info**))**

dev\_err**(&**adapter**->**dev**,**

"Can't create device at 0x%02x\n"**,**

devinfo**->**board\_info**.**addr**);**

**}**

up\_read**(&**\_\_i2c\_board\_lock**);**

**}**

如上面说当i2c\_adapter注册的时候，会遍历i2c bus type中所有的driver，并调用\_\_process\_new\_adapter函数。

static int \_\_process\_new\_adapter**(**struct device\_driver **\***d**,** void **\***data**)**

**{**

**return** i2c\_do\_add\_adapter**(**to\_i2c\_driver**(**d**),** data**);**

**}**

static int i2c\_do\_add\_adapter**(**struct i2c\_driver **\***driver**,**struct i2c\_adapter **\***adap**)**

**{**

i2c\_detect**(**adap**,** driver**);**

**......**

**}**

如果driver中存在address\_list，则该i2c总线会一一检查该driver的i2c地址是否可以被访问到，i2c\_detect函数便完成了该功能。

static int i2c\_detect**(**struct i2c\_adapter **\***adapter**,** struct i2c\_driver **\***driver**)**

**{**

const unsigned short **\***address\_list**;**

struct i2c\_client **\***temp\_client**;**

int i**,** err **=** 0**;**

//获取i2c adapter的number

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

//检查该driver对应的i2c地址序列

address\_list **=** driver**->**address\_list**;**

//driver里面需要有detect函数,并且地址列表不能为空。

**if** **(!**driver**->**detect **||** **!**address\_list**)**

**return** 0**;**

/\* Stop here if the classes do not match \*/

**if** **(!(**adapter**->**class **&** driver**->**class**))**

**return** 0**;**

//分配一个临时的i2c\_client结构体

temp\_client **=** kzalloc**(sizeof(**struct i2c\_client**),** GFP\_KERNEL**);**

//设置i2c client对应的i2c adapter

temp\_client**->**adapter **=** adapter**;**

//轮询address\_list中的地址，并做探测，如果该地址有响应，则调用i2c\_new\_device创建i2c\_client

**for** **(**i **=** 0**;** address\_list**[**i**]** **!=** I2C\_CLIENT\_END**;** i **+=** 1**)** **{**

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

"addr 0x%02x\n"**,** adap\_id**,** address\_list**[**i**]);**

temp\_client**->**addr **=** address\_list**[**i**];**

err **=** i2c\_detect\_address**(**temp\_client**,** driver**);**

**if** **(**unlikely**(**err**))**

**break;**

**}**

kfree**(**temp\_client**);**

**return** err**;**

**}**

i2c\_detect\_address代码分析如下：

static int i2c\_detect\_address**(**struct i2c\_client **\***temp\_client**,**struct i2c\_driver **\***driver**)**

**{**

struct i2c\_board\_info info**;**

struct i2c\_adapter **\***adapter **=** temp\_client**->**adapter**;**

int addr **=** temp\_client**->**addr**;**

int err**;**

//检查i2c地址是否在0x08~0x77之间

err **=** i2c\_check\_addr\_validity**(**addr**);**

//检查该地址是否已经注册

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

**return** 0**;**

//读或写一个字节，如果有响应，说明该地址存在

**if** **(!**i2c\_default\_probe**(**adapter**,** addr**))**

**return** 0**;**

//准备好i2c\_board\_info用于i2c\_new\_device

memset**(&**info**,** 0**,** **sizeof(**struct i2c\_board\_info**));**

//设置i2c\_board\_info的i2c地址

info**.**addr **=** addr**;**

//调用driver的detect函数，detect函数中会设置i2c\_board\_info中的i2c设备名字

err **=** driver**->**detect**(**temp\_client**,** **&**info**);**

//如果名字不为空,detect中一般设置，则调用i2c\_new\_device

**if** **(**info**.**type**[**0**]** **==** '\0'**)** **{**

dev\_err**(&**adapter**->**dev**,** "%s detection function provided "

"no name for 0x%x\n"**,** driver**->**driver**.**name**,**

addr**);**

**}** **else** **{**

struct i2c\_client **\***client**;**

/\* Detection succeeded, instantiate the device \*/

dev\_dbg**(&**adapter**->**dev**,** "Creating %s at 0x%02x\n"**,**

info**.**type**,** info**.**addr**);**

client **=** i2c\_new\_device**(**adapter**,** **&**info**);**

//将i2c\_client添加到driver->clients链表中

**if** **(**client**)**

list\_add\_tail**(&**client**->**detected**,** **&**driver**->**clients**);**

**else**

dev\_err**(&**adapter**->**dev**,** "Failed creating %s at 0x%02x\n"**,**

info**.**type**,** info**.**addr**);**

**}**

**return** 0**;**

**}**

### 4.3 i2c driver添加

i2c\_add\_driver用于添加driver到总线上，如果有匹配到总线上的device，则调用probe，同时会遍历driver中定义的address\_list，如果探测到设备，则创建i2c\_client。

#define i2c\_add\_driver(driver) \

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

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

**{**

int res**;**

//i2c bus type必须先初始化

**if** **(**unlikely**(**WARN\_ON**(!**i2c\_bus\_type**.**p**)))**

**return** **-**EAGAIN**;**

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

driver**->**driver**.**owner **=** owner**;**

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

//将该i2c driver添加到总线上并且从dev链表里取出能匹配的i2c\_client并调用probe

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

//初始化链表，用于保存连接在该driver上的client

INIT\_LIST\_HEAD**(&**driver**->**clients**);**

//遍历i2c bus type下的adpter device，并执行\_\_process\_new\_driver

i2c\_for\_each\_dev**(**driver**,** \_\_process\_new\_driver**);**

**return** 0**;**

**}**

int i2c\_for\_each\_dev**(**void **\***data**,** int **(\***fn**)(**struct device **\*,** void **\*))**

**{**

int res**;**

//遍历i2c bus type下所有的device设备

mutex\_lock**(&**core\_lock**);**

res **=** bus\_for\_each\_dev**(&**i2c\_bus\_type**,** **NULL,** data**,** fn**);**

mutex\_unlock**(&**core\_lock**);**

**return** res**;**

**}**

static int \_\_process\_new\_driver**(**struct device **\***dev**,** void **\***data**)**

**{**

//该device是否是adpter设备

**if** **(**dev**->**type **!=** **&**i2c\_adapter\_type**)**

**return** 0**;**

//分析该driver下的address\_list，如果address\_list中有addr被探测到了

//执行i2c\_new\_device操作。

**return** i2c\_do\_add\_adapter**(**data**,** to\_i2c\_adapter**(**dev**));**

**}**

### 4.4 i2c client添加

i2c\_new\_client添加device到i2c bus type，在添加时会进行driver匹配和probe操作。

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

**{**

struct i2c\_client **\***client**;**

int status**;**

//分配i2c\_client结构体

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

//设置client对应的adapter

client**->**adapter **=** adap**;**

//设置platform\_data

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

**if** **(**info**->**archdata**)**

client**->**dev**.**archdata **=** **\***info**->**archdata**;**

client**->**flags **=** info**->**flags**;**

client**->**addr **=** info**->**addr**;**

client**->**irq **=** info**->**irq**;**

//设置i2c client的名称

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

//判断i2c地址是否合法

status **=** i2c\_check\_client\_addr\_validity**(**client**);**

//检测该addr是否已经使用了

status **=** i2c\_check\_addr\_busy**(**adap**,** client**->**addr**);**

//设置i2c client dev对应的bus和client类型

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

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

client**->**dev**.**type **=** **&**i2c\_client\_type**;**

client**->**dev**.**of\_node **=** info**->**of\_node**;**

//设置i2c client的名称

dev\_set\_name**(&**client**->**dev**,** "%d-%04x"**,** i2c\_adapter\_id**(**adap**),**

client**->**addr **|** **((**client**->**flags **&** I2C\_CLIENT\_TEN**)**

**?** 0xa000 **:** 0**));**

//向i2c bus type注册device,添加的过程中会主动寻找对应的driver，并probe

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

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

client**->**name**,** dev\_name**(&**client**->**dev**));**

**return** client**;**

out\_err**:**

dev\_err**(&**adap**->**dev**,** "Failed to register i2c client %s at 0x%02x "

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

out\_err\_silent**:**

kfree**(**client**);**

**return** **NULL;**

**}**

### 4.5 device和driver的匹配函数

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

**{**

//获取dev对应的i2c\_client

struct i2c\_client **\***client **=** i2c\_verify\_client**(**dev**);**

struct i2c\_driver **\***driver**;**

**if** **(!**client**)**

**return** 0**;**

//通过驱动里定义了of\_device\_id项，则通过这一项来比对

**if** **(**of\_driver\_match\_device**(**dev**,** drv**))**

**return** 1**;**

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

//如果在平台驱动中定义了id\_table项，则通过对比id\_table来判断

**if** **(**driver**->**id\_table**)**

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

**return** 0**;**

**}**

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;**

**}**

### 4.6 i2c传输函数

#### 4.6.1 i2c\_transfer函数

struct i2c\_msg **{**

    \_\_u16 addr**;**     // 从机地址

    \_\_u16 flags**;**    // 标志

    \_\_u16 len**;**      // 数据长度

    \_\_u8 **\***buf**;**      // 数据指针

**};**

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

**{**

unsigned long orig\_jiffies**;**

int ret**,** **try;**

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

//i2c操作前需要上锁

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

ret **=** i2c\_trylock\_adapter**(**adap**);**

**if** **(!**ret**)**

/\* I2C activity is ongoing. \*/

**return** **-**EAGAIN**;**

**}** **else** **{**

i2c\_lock\_adapter**(**adap**);**

**}**

/\* Retry automatically on arbitration loss \*/

orig\_jiffies **=** jiffies**;**

**for** **(**ret **=** 0**,** **try** **=** 0**;** **try** **<=** adap**->**retries**;** **try++)** **{**

//实际还是调用master\_xfer函数

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

**if** **(**ret **!=** **-**EAGAIN**)**

**break;**

**if** **(**time\_after**(**jiffies**,** orig\_jiffies **+** adap**->**timeout**))**

**break;**

**}**

i2c\_unlock\_adapter**(**adap**);**

**return** ret**;**

**}** **else** **{**

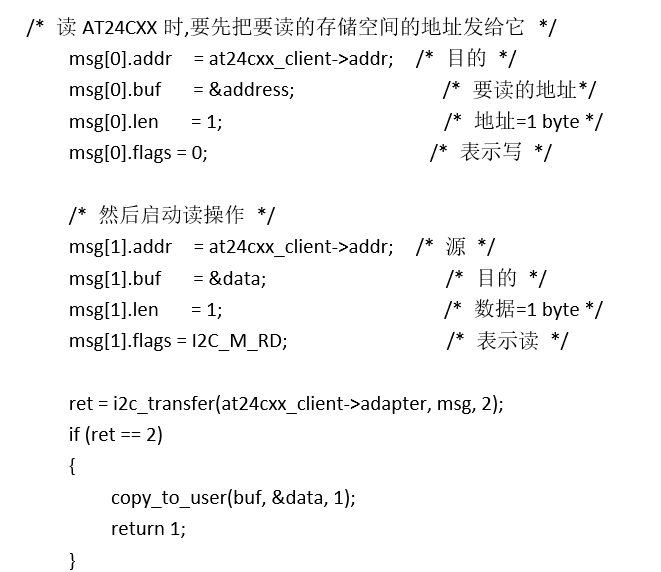
dev\_dbg**(&**adap**->**dev**,** "I2C level transfers not supported\n"**);**

**return** **-**EOPNOTSUPP**;**

**}**

**}**

i2c transfer使用例子如下：



#### 4.6.2 i2c\_master\_send和i2c\_master\_recv

在i2c\_transfer的基础上封装了一层。

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

**{**

int ret**;**

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

struct i2c\_msg msg**;**

msg**.**addr **=** client**->**addr**;**

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

msg**.**len **=** count**;**

msg**.**buf **=** **(**char **\*)**buf**;**

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

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

**}**

int i2c\_master\_recv**(**const 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**);**

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

**}**

#### 4.6.3 i2c\_smbus\_xfer

smbs是基于i2c总线的，但同i2c总线还是稍微有点区别的。可以借用smbus系列的函数，快速进行i2c的读写操作。

i2c\_smbus\_xfer是整个smbus操作的核心函数，如果硬件上支持smbus\_xfer则调用其smbus\_xfer函数，否则使用i2c模拟而成，这里使用i2c模拟。

s32 i2c\_smbus\_xfer**(**struct i2c\_adapter **\***adapter**,** u16 addr**,** unsigned short flags**,**char read\_write**,** u8 command**,** int protocol**,**union

i2c\_smbus\_data **\***data**)**

**{**

unsigned long orig\_jiffies**;**

int **try;**

s32 res**;**

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

//如果adapter支持smbus\_xfer，则直接使用smbus\_xfer，否则使用i2c模拟。

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

i2c\_lock\_adapter**(**adapter**);**

/\* Retry automatically on arbitration loss \*/

orig\_jiffies **=** jiffies**;**

**for** **(**res **=** 0**,** **try** **=** 0**;** **try** **<=** adapter**->**retries**;** **try++)** **{**

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

read\_write**,** command**,**

protocol**,** data**);**

**if** **(**res **!=** **-**EAGAIN**)**

**break;**

**if** **(**time\_after**(**jiffies**,**

orig\_jiffies **+** adapter**->**timeout**))**

**break;**

**}**

i2c\_unlock\_adapter**(**adapter**);**

**}** **else**

res **=** i2c\_smbus\_xfer\_emulated**(**adapter**,** addr**,** flags**,** read\_write**,**command**,** protocol**,** data**);**

**return** res**;**

**}**

i2c\_smbus\_xfer\_emulated是i2c模拟smbus函数，最终调用i2c\_transfer完成数据操作。所以i2c操作可以调用smbus系列函数，会方便一些。

s32 i2c\_smbus\_read\_byte\_data**(**const struct i2c\_client **\***client**,** u8 command**)**

**{**

union i2c\_smbus\_data data**;**

int status**;**

status **=** i2c\_smbus\_xfer**(**client**->**adapter**,** client**->**addr**,** client**->**flags**,**I2C\_SMBUS\_READ**,** command**,**I2C\_SMBUS\_BYTE\_DATA**,** **&**data**);**

**return** **(**status **<** 0**)** **?** status **:** data**.**byte**;**

**}**

s32 i2c\_smbus\_write\_byte\_data**(**const struct i2c\_client **\***client**,** u8 command**,**u8 value**)**

**{**

union i2c\_smbus\_data data**;**

data**.**byte **=** value**;**

**return** i2c\_smbus\_xfer**(**client**->**adapter**,** client**->**addr**,** client**->**flags**,**I2C\_SMBUS\_WRITE**,** command**,**I2C\_SMBUS\_BYTE\_DATA**,** **&**data**);**

**}**

s32 i2c\_smbus\_read\_word\_data**(**const struct i2c\_client **\***client**,** u8 command**)**

**{**

union i2c\_smbus\_data data**;**

int status**;**

status **=** i2c\_smbus\_xfer**(**client**->**adapter**,** client**->**addr**,** client**->**flags**,**I2C\_SMBUS\_READ**,** command**,**I2C\_SMBUS\_WORD\_DATA**,** **&**data**);**

**return** **(**status **<** 0**)** **?** status **:** data**.**word**;**

**}**

s32 i2c\_smbus\_write\_word\_data**(**const struct i2c\_client **\***client**,** u8 command**,** u16 value**)**

**{**

union i2c\_smbus\_data data**;**

data**.**word **=** value**;**

**return** i2c\_smbus\_xfer**(**client**->**adapter**,** client**->**addr**,** client**->**flags**,**I2C\_SMBUS\_WRITE**,** command**,**I2C\_SMBUS\_WORD\_DATA**,** **&**data**);**

**}**

s32 i2c\_smbus\_read\_block\_data**(**const struct i2c\_client **\***client**,** u8 command**,** u8 **\***values**)**

**{**

union i2c\_smbus\_data data**;**

int status**;**

status **=** i2c\_smbus\_xfer**(**client**->**adapter**,** client**->**addr**,** client**->**flags**,** I2C\_SMBUS\_READ**,** command**,** I2C\_SMBUS\_BLOCK\_DATA**,** **&**data**);**

**if** **(**status**)**

**return** status**;**

memcpy**(**values**,** **&**data**.**block**[**1**],** data**.**block**[**0**]);**

**return** data**.**block**[**0**];**

**}**

s32 i2c\_smbus\_write\_block\_data**(**const struct i2c\_client **\***client**,** u8 command**,** u8 length**,** const u8 **\***values**)**

**{**

union i2c\_smbus\_data data**;**

**if** **(**length **>** I2C\_SMBUS\_BLOCK\_MAX**)**

length **=** I2C\_SMBUS\_BLOCK\_MAX**;**

data**.**block**[**0**]** **=** length**;**

memcpy**(&**data**.**block**[**1**],** values**,** length**);**

**return** i2c\_smbus\_xfer**(**client**->**adapter**,** client**->**addr**,** client**->**flags**,** I2C\_SMBUS\_WRITE**,** command**,** I2C\_SMBUS\_BLOCK\_DATA**,**

**&**data**);**

**}**

## 5. 应用层的i2c操作

可以在应用层完成i2c的读写操作，下面分析下内核中是如何支持该操作。

i2c\_dev\_init创建了i2c总线在应用层的操作。

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

**{**

int res**;**

//创建字符设备，主设备号为123

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

//创建/sys/class/i2c-dev

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

//注册总线通知链

res **=** bus\_register\_notifier**(&**i2c\_bus\_type**,** **&**i2cdev\_notifier**);**

//遍历i2c bus上的adapter，并调用i2cdev\_attach\_adapter

i2c\_for\_each\_dev**(NULL,** i2cdev\_attach\_adapter**);**

**return** res**;**

**}**

bus\_register\_notifier设置i2c bus的通知链函数为i2cdev\_notifier。当有设备添加时，会发出BUS\_NOTIFY\_ADD\_DEVICE通知，当设备移除时，发出BUS\_NOTIFY\_DEL\_DEVICE通知。

blocking\_notifier\_call\_chain**(&**dev**->**bus**->**p**->**bus\_notifier**,**BUS\_NOTIFY\_ADD\_DEVICE**,** dev**);**

blocking\_notifier\_call\_chain**(&**dev**->**bus**->**p**->**bus\_notifier**,**BUS\_NOTIFY\_DEL\_DEVICE**,** dev**);**

i2cdev\_notifier代码如下：

static int i2cdev\_notifier\_call**(**struct notifier\_block **\***nb**,** unsigned long action**,**void **\***data**)**

**{**

struct device **\***dev **=** data**;**

**switch** **(**action**)** **{**

**case** BUS\_NOTIFY\_ADD\_DEVICE**:**

**return** i2cdev\_attach\_adapter**(**dev**,** **NULL);**

**case** BUS\_NOTIFY\_DEL\_DEVICE**:**

**return** i2cdev\_detach\_adapter**(**dev**,** **NULL);**

**}**

**return** 0**;**

**}**

static struct notifier\_block i2cdev\_notifier **=** **{**

**.**notifier\_call **=** i2cdev\_notifier\_call**,**

**};**

当有总线上有设备添加时，调用i2cdev\_attach\_adapter,当总线上有设备移除时，调用i2cdev\_detach\_adapter。

i2cdev\_attach\_adapter为每个i2c adapter创建字符设备。

static int i2cdev\_attach\_adapter**(**struct device **\***dev**,** void **\***dummy**)**

**{**

struct i2c\_adapter **\***adap**;**

struct i2c\_dev **\***i2c\_dev**;**

int res**;**

//判断该device是否是adapter设备

**if** **(**dev**->**type **!=** **&**i2c\_adapter\_type**)**

**return** 0**;**

//获取i2c\_adapter结构体

adap **=** to\_i2c\_adapter**(**dev**);**

//分配一个i2c\_dev结构体，并将其添加到i2c\_dev\_list

//i2c\_dev中设置了adapter结构体 i2c\_dev->adap = adap;

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

//创建目录/sys/class/i2c-dev/i2c-x

//创建设备/dev/i2c-x

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

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

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

//创建/sys/class/i2c-dev/i2c-x/name接口，可以显示i2c总线的名字

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

**return** 0**;**

**}**

现在有了字符设备/dev/i2c-x，那么如何操作这个设备呢？

static const struct file\_operations i2cdev\_fops **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**llseek **=** no\_llseek**,**

**.**read **=** i2cdev\_read**,**

**.**write **=** i2cdev\_write**,**

**.**unlocked\_ioctl **=** i2cdev\_ioctl**,**

**.**open **=** i2cdev\_open**,**

**.**release **=** i2cdev\_release**,**

**};**

### 5.1 i2cdev\_open

i2cdev\_open打开/dev/i2c-x设备，并创建一个i2c\_client结构体。

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

**{**

//获取设备对应的从设备号

unsigned int minor **=** iminor**(**inode**);**

struct i2c\_client **\***client**;**

struct i2c\_adapter **\***adap**;**

struct i2c\_dev **\***i2c\_dev**;**

//根据从设备号从i2c\_dev\_list找到i2c\_dev结构体

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

//使用idr机制，获取对应的i2c\_adapter结构体

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

//创建i2c\_client结构体

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

//设置i2c client的名字

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

//设置i2c client对应的adapter

client**->**adapter **=** adap**;**

//设置打开文件的私有数据为i2c\_client

file**->**private\_data **=** client**;**

**return** 0**;**

**}**

### 5.2 i2cdev\_ioctl

下面对ioctl相关的命令进行分析。

static long i2cdev\_ioctl**(**struct file **\***file**,** unsigned int cmd**,** unsigned long arg**)**

**{**

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

**switch** **(**cmd**)** **{**

//设置操作的i2c地址

**case** I2C\_SLAVE**:**

client**->**addr **=** arg**;**

**break;**

//i2c读写

**case** I2C\_RDWR**:**

i2cdev\_ioctl\_rdrw**(**client**,** arg**);**

**break;**

//smbus读写

**case** I2C\_SMBUS**:**

i2cdev\_ioctl\_smbus**(**client**,** arg**);**

**break;**

**}**

**}**

i2c/smbus读写传入的数据arg定义如下：

struct i2c\_smbus\_ioctl\_data **{**

\_\_u8 read\_write**;**

\_\_u8 command**;**

\_\_u32 size**;**

union i2c\_smbus\_data \_\_user **\***data**;**

**};**

struct i2c\_rdwr\_ioctl\_data **{**

struct i2c\_msg \_\_user **\***msgs**;** /\* pointers to i2c\_msgs \*/

\_\_u32 nmsgs**;** /\* number of i2c\_msgs \*/

**};**

### 5.3 i2cdev\_write/i2cdev\_read

读写函数比较简单，代码如下：

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 **=** file**->**private\_data**;**

**if** **(**count **>** 8192**)**

count **=** 8192**;**

tmp **=** memdup\_user**(**buf**,** count**);**

**if** **(**IS\_ERR**(**tmp**))**

**return** PTR\_ERR**(**tmp**);**

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

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

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

kfree**(**tmp**);**

**return** ret**;**

**}**

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 **=** 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 %zu 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**;**

**}**

### 5.4 i2c-tool原理分析

（1）扫描总线

//尝试从/proc/bus/i2c读取i2c总线信息

f **=** fopen**(**"/proc/bus/i2c"**,** "r"**)**

//如果读取不到,从/sys/class/i2c-dev中读取

//首先打开/proc/mounts，获取sys的地址

//sysfs /sys sysfs rw,nosuid,nodev,noexec,relatime 0 0

fopen**(**"/proc/mounts"**,** "r"**)**

//接下来读取/sys/class/i2c-dev，从中获取i2c-x的内容

strcat**(**sysfs**,** "/class/i2c-dev"**);**

（2）扫描总线下的i2c地址

//打开/dev/i2c-x

open\_i2c\_dev**(**i2cbus**,** filename**,** **sizeof(**filename**),** 0**);**

//设置i2c地址

ioctl**(**file**,** I2C\_SLAVE**,** i**+**j**)**

//尝试操作下

i2c\_smbus\_read\_byte

## 6. 2440下i2c总线的注册

### 6.1 platform driver和device匹配

platform driver的注册：

static int \_\_init i2c\_adap\_s3c\_init**(**void**)**

**{**

**return** platform\_driver\_register**(&**s3c24xx\_i2c\_driver**);**

**}**

#ifdef CONFIG\_OF

static const struct of\_device\_id s3c24xx\_i2c\_match**[]** **=** **{**

**{** **.**compatible **=** "samsung,s3c2410-i2c" **},**

**{** **.**compatible **=** "samsung,s3c2440-i2c" **},**

**{},**

**};**

static struct platform\_device\_id s3c24xx\_driver\_ids**[]** **=** **{**

**{**

**.**name **=** "s3c2410-i2c"**,**

**.**driver\_data **=** TYPE\_S3C2410**,**

**},** **{**

**.**name **=** "s3c2440-i2c"**,**

**.**driver\_data **=** TYPE\_S3C2440**,**

**},** **{** **},**

**};**

static struct platform\_driver s3c24xx\_i2c\_driver **=** **{**

**.**probe **=** s3c24xx\_i2c\_probe**,**

**.**remove **=** s3c24xx\_i2c\_remove**,**

**.**id\_table **=** s3c24xx\_driver\_ids**,**

**.**driver **=** **{**

**.**owner **=** THIS\_MODULE**,**

**.**name **=** "s3c-i2c"**,**

**.**pm **=** S3C24XX\_DEV\_PM\_OPS**,**

**.**of\_match\_table **=** s3c24xx\_i2c\_match**,**

**},**

**};**

platform device的注册：

smdk2440\_machine\_init

platform\_device\_register**(&**s3c\_device\_i2c0**);**

static struct resource s3c\_i2c0\_resource**[]** **=** **{**

**[**0**]** **=** DEFINE\_RES\_MEM**(**S3C\_PA\_IIC**,** SZ\_4K**),**

**[**1**]** **=** DEFINE\_RES\_IRQ**(**IRQ\_IIC**),**

**};**

struct platform\_device s3c\_device\_i2c0 **=** **{**

**.**name **=** "s3c2410-i2c"**,**

#ifdef CONFIG\_S3C\_DEV\_I2C1

**.**id **=** 0**,**

#else

**.**id **=** **-**1**,**

#endif

**.**num\_resources **=** ARRAY\_SIZE**(**s3c\_i2c0\_resource**),**

**.**resource **=** s3c\_i2c0\_resource**,**

**};**

3.4.2和2.6.26的内核在匹配的方式上有一些不同。

//匹配有两种方式，一种通过of\_match\_table，另外一种通过id\_table

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

**{**

    struct platform\_device **\***pdev **=** to\_platform\_device**(**dev**);**

    struct platform\_driver **\***pdrv **=** to\_platform\_driver**(**drv**);**

    /\* Attempt an OF style match first \*/

    if **(**of\_driver\_match\_device**(**dev**,** drv**))**

        return 1**;**

    /\* Then try to match against the id table \*/

    if **(**pdrv**->**id\_table**)**

        return platform\_match\_id**(**pdrv**->**id\_table**,** pdev**)** **!=** NULL**;**

    /\* fall-back to driver name match \*/

    return **(**strcmp**(**pdev**->**name**,** drv**->**name**)** **==** 0**);**

**}**

### 6.2 s3c24xx\_i2c\_probe

struct s3c2410\_platform\_i2c default\_i2c\_data \_\_initdata **=** **{**

**.**flags **=** 0**,**

**.**slave\_addr **=** 0x10**,**

**.**frequency **=** 100**\***1000**,**

**.**sda\_delay **=** 100**,**

**};**

static int s3c24xx\_i2c\_probe**(**struct platform\_device **\***pdev**)**

**{**

struct s3c24xx\_i2c **\***i2c**;**

struct s3c2410\_platform\_i2c **\***pdata **=** **NULL;**

struct resource **\***res**;**

int ret**;**

//of\_node用于dts树，这里未使用

**if** **(!**pdev**->**dev**.**of\_node**)** **{**

//如果未使用dts树，则获取platform\_data

pdata **=** pdev**->**dev**.**platform\_data**;**

**}**

//devm\_kzalloc() 是具有资源管理的 kzalloc()。使用资源管理（resource-managed）函数分配的内存

//是会与所属设备相关联。当设备从系统中分离或者设备驱动被卸载，该内存会被自动释放

i2c **=** devm\_kzalloc**(&**pdev**->**dev**,** **sizeof(**struct s3c24xx\_i2c**),** GFP\_KERNEL**);**

i2c**->**pdata **=** devm\_kzalloc**(&**pdev**->**dev**,** **sizeof(\***pdata**),** GFP\_KERNEL**);**

//如果未用dts树，则直接将platform\_data拷贝

**if** **(**pdata**)**

memcpy**(**i2c**->**pdata**,** pdata**,** **sizeof(\***pdata**));**

**else**

s3c24xx\_i2c\_parse\_dt**(**pdev**->**dev**.**of\_node**,** i2c**);**

//设置i2c adapter的名字

strlcpy**(**i2c**->**adap**.**name**,** "s3c2410-i2c"**,** **sizeof(**i2c**->**adap**.**name**));**

i2c**->**adap**.**owner **=** THIS\_MODULE**;**

//设置i2c adapter的操作函数

i2c**->**adap**.**algo **=** **&**s3c24xx\_i2c\_algorithm**;**

i2c**->**adap**.**retries **=** 2**;**

i2c**->**adap**.**class **=** I2C\_CLASS\_HWMON **|** I2C\_CLASS\_SPD**;**

i2c**->**tx\_setup **=** 50**;**

spin\_lock\_init**(&**i2c**->**lock**);**

//传输完成唤醒阻塞

init\_waitqueue\_head**(&**i2c**->**wait**);**

//打开i2c时钟

i2c**->**dev **=** **&**pdev**->**dev**;**

i2c**->**clk **=** clk\_get**(&**pdev**->**dev**,** "i2c"**);**

clk\_enable**(**i2c**->**clk**);**

//获取platform device注册时的resource

res **=** platform\_get\_resource**(**pdev**,** IORESOURCE\_MEM**,** 0**);**

//申请虚拟内存空间

i2c**->**ioarea **=** request\_mem\_region**(**res**->**start**,** resource\_size**(**res**),**

pdev**->**name**);**

i2c**->**regs **=** ioremap**(**res**->**start**,** resource\_size**(**res**));**

i2c**->**adap**.**algo\_data **=** i2c**;**

i2c**->**adap**.**dev**.**parent **=** **&**pdev**->**dev**;**

//i2c硬件初始化

ret **=** s3c24xx\_i2c\_init**(**i2c**);**

//获取resource的irq信息

i2c**->**irq **=** ret **=** platform\_get\_irq**(**pdev**,** 0**);**

//设置中断

ret **=** request\_irq**(**i2c**->**irq**,** s3c24xx\_i2c\_irq**,** 0**,**

dev\_name**(&**pdev**->**dev**),** i2c**);**

//当cpu频率变化时，i2c时钟需要做相应的改变

ret **=** s3c24xx\_i2c\_register\_cpufreq**(**i2c**);**

//设置adapter number，这里为0

i2c**->**adap**.**nr **=** i2c**->**pdata**->**bus\_num**;**

i2c**->**adap**.**dev**.**of\_node **=** pdev**->**dev**.**of\_node**;**

//添加adapter，这里指定了number

ret **=** i2c\_add\_numbered\_adapter**(&**i2c**->**adap**);**

//of\_i2c\_register\_devices对dtsi上所描述的设备进行实例化。并创建相应的sys文件

of\_i2c\_register\_devices**(&**i2c**->**adap**);**

//设置driver\_data，remove的时候方便free i2c

platform\_set\_drvdata**(**pdev**,** i2c**);**

pm\_runtime\_enable**(&**pdev**->**dev**);**

pm\_runtime\_enable**(&**i2c**->**adap**.**dev**);**

//使用的时候再打开i2c时钟

clk\_disable**(**i2c**->**clk**);**

**return** 0**;**

**}**

这里补充一下dts设备树的相关i2c操作。

在dtsi中有：

/\*@后面是设备的起始地址\*/

**&**i2c**-**0@fe **{**

/\* i2c\_client的name = "hall-i2c" \*/

compatible **=** "qcom, hall-i2c"**;**

reg **=** **<**fe**>;**

interrupts **=** **<**70**>;**

/\* 如果设置成disabled，在初始化的时候就不会被实例化，可以在linux内置文档查看更多 \*/

status **=** "disabled"**;**

**};**

其中：i2c-0中的0是总线编号，reg是设备地址，interrupts是中断号。  
of\_i2c\_register\_devices对dtsi上所描述的设备进行实例化，并创建相应的sys文件：sys/bus/i2c/devices/0-00fe。

## 7. 多个i2c client一个i2c driver情形

i2c client的platform\_data：

struct i2c\_board\_info **{**

char type**[**I2C\_NAME\_SIZE**];**

unsigned short flags**;**

unsigned short addr**;**

void **\***platform\_data**;**

struct dev\_archdata **\***archdata**;**

struct device\_node **\***of\_node**;**

int irq**;**

**};**

struct i2c\_client **\***

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

**{**

**....**

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

**....**

**}**

多个i2c client，一个i2c driver依靠id\_table的名字来进行匹配。

static const struct i2c\_device\_id at24\_ids**[]** **=** **{**

/\* needs 8 addresses as A0-A2 are ignored \*/

**{** "24c00"**,** AT24\_DEVICE\_MAGIC**(**128 **/** 8**,** AT24\_FLAG\_TAKE8ADDR**)** **},**

/\* old variants can't be handled with this generic entry! \*/

**{** "24c01"**,** AT24\_DEVICE\_MAGIC**(**1024 **/** 8**,** 0**)** **},**

**{** "24c02"**,** AT24\_DEVICE\_MAGIC**(**2048 **/** 8**,** 0**)** **},**

/\* spd is a 24c02 in memory DIMMs \*/

**{** "spd"**,** AT24\_DEVICE\_MAGIC**(**2048 **/** 8**,**

AT24\_FLAG\_READONLY **|** AT24\_FLAG\_IRUGO**)** **},**

**{** "24c04"**,** AT24\_DEVICE\_MAGIC**(**4096 **/** 8**,** 0**)** **},**

/\* 24rf08 quirk is handled at i2c-core \*/

**{** "24c08"**,** AT24\_DEVICE\_MAGIC**(**8192 **/** 8**,** 0**)** **},**

**{** "24c16"**,** AT24\_DEVICE\_MAGIC**(**16384 **/** 8**,** 0**)** **},**

**{** "24c32"**,** AT24\_DEVICE\_MAGIC**(**32768 **/** 8**,** AT24\_FLAG\_ADDR16**)** **},**

**{** "24c64"**,** AT24\_DEVICE\_MAGIC**(**65536 **/** 8**,** AT24\_FLAG\_ADDR16**)** **},**

**{** "24c128"**,** AT24\_DEVICE\_MAGIC**(**131072 **/** 8**,** AT24\_FLAG\_ADDR16**)** **},**

**{** "24c256"**,** AT24\_DEVICE\_MAGIC**(**262144 **/** 8**,** AT24\_FLAG\_ADDR16**)** **},**

**{** "24c512"**,** AT24\_DEVICE\_MAGIC**(**524288 **/** 8**,** AT24\_FLAG\_ADDR16**)** **},**

**{** "24c1024"**,** AT24\_DEVICE\_MAGIC**(**1048576 **/** 8**,** AT24\_FLAG\_ADDR16**)** **},**

**{** "at24"**,** 0 **},**

**{** /\* END OF LIST \*/ **}**

**};**

static struct i2c\_driver at24\_driver **=** **{**

**.**driver **=** **{**

**.**name **=** "at24"**,**

**.**owner **=** THIS\_MODULE**,**

**},**

**.**probe **=** at24\_probe**,**

**.**remove **=** \_\_devexit\_p**(**at24\_remove**),**

**.**id\_table **=** at24\_ids**,**

**};**

那么在probe的时候就很容易获取到client和i2c\_device\_id的私有数据。

static int at24\_probe**(**struct i2c\_client **\***client**,** const struct i2c\_device\_id **\***id**)**

**{**

client**->**dev**->**platform\_data

id**->**driver\_data**;**

**}**

在i2c probe的时候，一般会创建一个较大的结构体，用于保存一些私有数据和i2c\_client，当然也需要保存device和cdev，可以借用container\_of反向得到i2c\_client。可以调用i2c\_set\_clientdata将该结构体设置为i2c\_client对应的driver\_data。