# Linux驱动\_I2C

## 1. I2C驱动框架

### 1.1 I2C体系结构

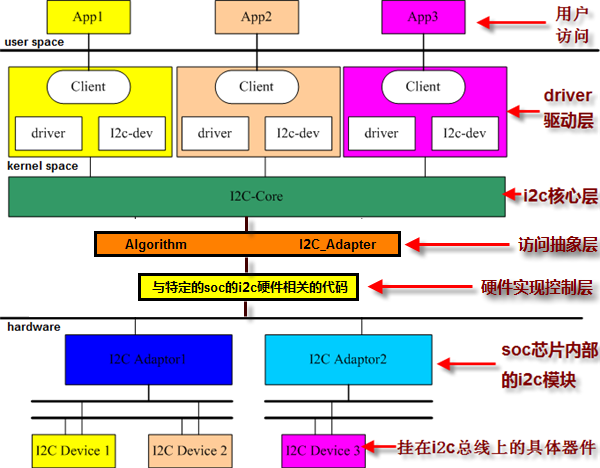
Linux的I2C**体系结构**分为3个组成部分：

**I2C核心：**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总线驱动是对I2C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至可以直接集成在CPU内部。

**I2C设备驱动：**I2C设备驱动(也称为客户驱动)是对I2C硬件体系结构中设备端的实现，设备一般挂接在受CPU控制的I2C适配器上，通过I2C适配器与CPU交换数据。

### 1.2 I2C架构层次



第一层：提供i2c adapter的硬件驱动，探测、初始化i2c adapter（如申请i2c的io地址和中断号），驱动soc控制的i2c adapter在硬件上产生信号（start、stop、ack）以及处理i2c中断。覆盖图中的硬件实现层。

第二层：提供i2c adapter的algorithm，用具体适配器的xxx\_xferf()函数来填充i2c\_algorithm的master\_xfer函数指针，并把赋值后的i2c\_algorithm再赋值给i2c\_adapter的algo指针。覆盖图中的访问抽象层、i2c核心层。

第三层：实现i2c设备驱动中的i2c\_driver接口，用具体的i2c device设备的attach\_adapter()、detach\_adapter()方法赋值给i2c\_driver的成员函数指针。实现设备device与总线（或者叫adapter）的挂接。覆盖图中的driver驱动层。

第四层：实现i2c设备所对应的具体device的驱动，i2c\_driver只是实现设备与总线的挂接，而挂接在总线上的设备则是千差万别的，所以要实现具体设备device的write()、read()、ioctl()等方法，赋值给file\_operations，然后注册字符设备（多数是字符设备）。覆盖图中的driver驱动层。

第一层和第二层又叫i2c总线驱动(bus)，第三第四属于i2c设备驱动(device driver)。

### 1.3 I2C体系文件架构

i2c-core.c这个文件实现了I2C核心的功能以及/proc/bus/i2c\*接口。  
i2c-dev.c实现了I2C适配器设备文件的功能，每一个I2C适配器都被分配一个设备。通过适配器访设备时的主设备号都为89，次设备号为0-255。I2c-dev.c并没有针对特定的设备而设计，只是提供了通用的read(),write(),和ioctl()等接口，应用层可以借用这些接口访问挂接在适配器上的I2C设备的存储空间或寄存器，并控制I2C设备的工作方式。  
busses文件夹这个文件中包含了一些I2C总线的驱动，如针对S3C2410，S3C2440，S3C6410等处理器的I2C控制器驱动为i2c-s3c2410.c.  
algos文件夹实现了一些I2C总线适配器的algorithm.

### 1.4 重要的I2C结构体和关系

**(1)i2c\_adapter与i2c\_algorithm**

i2c\_adapter对应与物理上的一个适配器，而i2c\_algorithm对应一套通信方法，一个i2c适配器需要i2c\_algorithm中提供的（i2c\_algorithm中的又是更下层与硬件相关的代码提供）通信函数来控制适配器上产生特定的访问周期。缺少i2c\_algorithm的i2c\_adapter什么也做不了，因此i2c\_adapter中包含其使用i2c\_algorithm的指针。

i2c\_algorithm中的关键函数master\_xfer()用于产生i2c访问周期需要的start stop ack信号，以i2c\_msg（即i2c消息）为单位发送和接收通信数据。

i2c\_msg也非常关键，调用驱动中的发送接收函数需要填充该结构体

**(2)** **i2c\_driver和i2c\_client**

i2c\_driver对应一套驱动方法，其主要函数是attach\_adapter()和detach\_client()，i2c\_driver实现设备与总线的挂接。

i2c\_client对应真实的i2c物理设备device，每个i2c设备都需要一个i2c\_client来描述。

i2c\_driver与i2c\_client的关系是一对多。一个i2c\_driver上可以支持多个同等类型的i2c\_client。

**(3)i2c\_adapter和i2c\_client**

i2c\_adapter和i2c\_client的关系与i2c硬件体系中适配器和设备的关系一致，即i2c\_client依附于i2c\_adapter,由于一个适配器上可以连接多个i2c设备，所以i2c\_adapter中包含依附于它的i2c\_client的链表。

## 2. at24cxx的i2c驱动

i2c的驱动程序可以由旧探测方法(adapt)和新探测方法(probe)这两种方法实现。

在i2c\_driver中，adapt老方法使用的是attach\_adapter作为探测函数，而在新方法中使用的是probe作为探测函数。

### 2.1 方法1(传统方法)

我们先从旧探测方法adapt开始学习实现存储芯片at24cxx的i2c驱动程序,注意本试验的实现必须在linux-2.6.22.6或者更低版本的内核上编译，原因是linux-2.6.30以后的内核i2c.c文件中已经取消了统一的i2c\_probe()这个函数了，所以本实验再高版本的内核中是无法实现的。

#### 2.1.1 at24cxx\_init

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

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

**.**name **=** "at24cxx"**,**

**},**

**.**attach\_adapter **=** at24cxx\_attach**,**

**.**detach\_client **=** at24cxx\_detach**,**

**};**

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

**{**

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

**return** 0**;**

**}**

当注册了i2c\_driver驱动后，系统会进入到attach\_adapter函数探测i2c设备是否支持。

#### 2.1.2 at24cxx\_attach

static int at24cxx\_attach**(**struct i2c\_adapter **\***adapter**)**

**{**

**return** i2c\_probe**(**adapter**,** **&**addr\_data**,** at24cxx\_detect**);**

**}**

attach\_adapter函数中主要的功能是在调用i2c\_probe函数，当系统检测到设备地址匹配时，就会进入i2c\_probe函数中干一些重要的事，接着就进入i2c-probe传入的at24cxx\_detect函数中实现我们自己的事。

addr\_data中定义了i2c设备地址。

static unsigned short ignore**[]** **=** **{** I2C\_CLIENT\_END **};**

static unsigned short normal\_addr**[]** **=** **{** 0x50**,** I2C\_CLIENT\_END **};**

/\* 地址值是7位 \*/

/\* 改为0x60的话, 由于不存在设备地址为0x60的设备, 所以at24cxx\_detect不被调用 \*/

static unsigned short force\_addr**[]** **=** **{**ANY\_I2C\_BUS**,** 0x60**,** I2C\_CLIENT\_END**};**

static unsigned short **\*** forces**[]** **=** **{**force\_addr**,** **NULL};**

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

**.**normal\_i2c **=** normal\_addr**,** /\* 要发出S信号和设备地址并得到ACK信号,才能确定存在这个设备 \*/

**.**probe **=** ignore**,**

**.**ignore **=** ignore**,**

//.forces = forces, /\* 强制认为存在这个设备 \*/

**};**

在设备地址数组中，I2C\_CLIENT\_END是指设备地址结束，一旦系统检测到这个值，就会停止扫描。

在这里at24cxx的i2c地址为0x50，如果某总线发送该设备地址，收到了ack响应，则认为探测到了该设备。

接下来要进入at24cxx\_detect函数。

#### 2.1.3 at24cxx\_detect

static int at24cxx\_detect**(**struct i2c\_adapter **\***adapter**,** int address**,** int kind**)**

**{**

printk**(**"at24cxx\_detect\n"**);**

/\* 构构一个i2c\_client结构体: 以后收改数据时会用到它 \*/

at24cxx\_client **=** kzalloc**(sizeof(**struct i2c\_client**),** GFP\_KERNEL**);**

at24cxx\_client**->**addr **=** address**;**

at24cxx\_client**->**adapter **=** adapter**;**

at24cxx\_client**->**driver **=** **&**at24cxx\_driver**;**

strcpy**(**at24cxx\_client**->**name**,** "at24cxx"**);**

i2c\_attach\_client**(**at24cxx\_client**);**

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

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

class\_device\_create**(**cls**,** **NULL,** MKDEV**(**major**,** 0**),** **NULL,** "at24cxx"**);** /\* /dev/at24cxx \*/

**return** 0**;**

**}**

at24cxx\_detect传入的参数有i2c\_adapter和address，也就是i2c bus的信息和探测到的i2c地址。

这个函数里创建了一个i2c\_client结构体，该结构体负责后续的i2c操作，最后调用i2c\_attach\_client注册i2c\_client。

接下来就可以注册一个字符设备用于读写操作。

#### 2.1.4 at24cxx\_exit

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

**{**

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

**}**

当卸载i2c\_driver时，会自动调用i2c\_driver中定义的卸载函数at24cxx\_detach进行卸载设备，将对应的i2c\_client删除掉。

static int at24cxx\_detach**(**struct i2c\_client **\***client**)**

**{**

printk**(**"at24cxx\_detach\n"**);**

class\_device\_destroy**(**cls**,** MKDEV**(**major**,** 0**));**

class\_destroy**(**cls**);**

unregister\_chrdev**(**major**,** "at24cxx"**);**

i2c\_detach\_client**(**client**);**

kfree**(**i2c\_get\_clientdata**(**client**));**

**return** 0**;**

**}**

#### 2.1.5 at24cxx\_read和at24cxx\_write

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

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

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

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

**};**

写操作如下：

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

**{**

unsigned char val**[**2**];**

struct i2c\_msg msg**[**1**];**

int ret**;**

/\* address = buf[0]

\* data = buf[1]

\*/

**if** **(**size **!=** 2**)**

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

copy\_from\_user**(**val**,** buf**,** 2**);**

/\* 数据传输三要素: 源,目的,长度 \*/

msg**[**0**].**addr **=** at24cxx\_client**->**addr**;** /\* 目的 \*/

msg**[**0**].**buf **=** val**;** /\* 源 \*/

msg**[**0**].**len **=** 2**;** /\* 地址+数据=2 byte \*/

msg**[**0**].**flags **=** 0**;** /\* 表示写 \*/

ret **=** i2c\_transfer**(**at24cxx\_client**->**adapter**,** msg**,** 1**);**

**if** **(**ret **==** 1**)**

**return** 2**;**

**else**

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

**}**

i2c读写函数由i2c\_transfer完成，可以看到i2c bus为at24cxx\_client->adapter。i2c\_msg的flags定义了数据方向。

读操作：

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

**{**

unsigned char address**;**

unsigned char data**;**

struct i2c\_msg msg**[**2**];**

int ret**;**

/\* address = buf[0]

\* data = buf[1]

\*/

**if** **(**size **!=** 1**)**

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

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

/\* 数据传输三要素: 源,目的,长度 \*/

/\* 读AT24CXX时,要先把要读的存储空间的地址发给它 \*/

msg**[**0**].**addr **=** at24cxx\_client**->**addr**;** /\* 目的 \*/

msg**[**0**].**buf **=** **&**address**;** /\* 源 \*/

msg**[**0**].**len **=** 1**;** /\* 地址=1 byte \*/

msg**[**0**].**flags **=** 0**;** /\* 表示写 \*/

/\* 然后启动读操作 \*/

msg**[**1**].**addr **=** at24cxx\_client**->**addr**;** /\* 源 \*/

msg**[**1**].**buf **=** **&**data**;** /\* 目的 \*/

msg**[**1**].**len **=** 1**;** /\* 数据=1 byte \*/

msg**[**1**].**flags **=** I2C\_M\_RD**;** /\* 表示读 \*/

ret **=** i2c\_transfer**(**at24cxx\_client**->**adapter**,** msg**,** 2**);**

**if** **(**ret **==** 2**)**

**{**

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

**return** 1**;**

**}**

**else**

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

**}**

读操作由一个写操作和一个读操作共同完成。

#### 2.1.6 测试

测试代码流程：

//打开设备

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

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

//读操作,传入adress，然后读取一个字节的数据

buf**[**0**]** **=** address**;**

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

//写操作，传入address和data

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

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

write**(**fd**,** buf**,** 2**);**

测试结果：

# insmod at24cxx.ko

at24cxx\_detect

# ls -l /dev/at24cxx

crw**-**rw**----** 1 0 0 252**,** 0 Jan 1 00**:**04 **/**dev**/**at24cxx

# ./i2c\_test w 00 01

# ./i2c\_test r 0

data**:** **,** 1**,** 0x 1

#### 2.1.7 force属性

不知道大家注意到没有，在i2c\_client\_address\_data结构体中还有一个我们没有使用到的属性.forces，它是用于强制找到设备地址的。当使用了这个属性，无论总线是否会探测

### 2.2 其他方法

由于新的linux内核i2c驱动写法有一些改变，我们将在新的内核中分析i2c驱动的写法。

i2c bus查询 i2c slave i2c tools i2c的4种写法 smbus 模拟i2c 时钟clk\_get

/dev/i2c-0