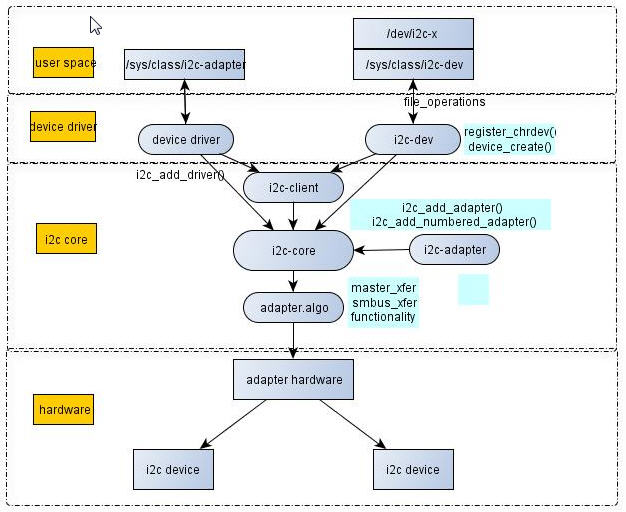
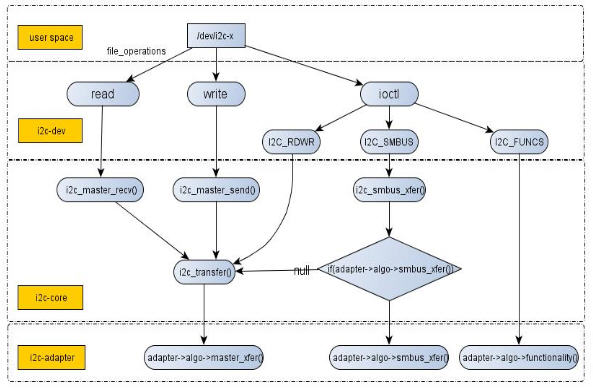
# 4. I2C 驱动

## 4.1 Linux I2C设备驱动编写

摘自 <http://blog.csdn.net/airk000/article/details/21345457>





### 4.1.1 I2C adapter

是CPU集成或外接的I2C适配器，用来控制各种I2C从设备，其驱动需要完成对适配器的完整描述，最主要的工作是需要完成i2c\_algorithm结构体。这个结构体包含了此I2C控制器的数据传输具体实现，以及对外上报此设备所支持的功能类型。i2c\_algorithm结构体如下：

struct i2c\_algorithm {

int (\*master\_xfer)(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs,

int num);

int (\*smbus\_xfer) (struct i2c\_adapter \*adap, u16 addr,

unsigned short flags, char read\_write,

u8 command, int size, union i2c\_smbus\_data \*data);

u32 (\*functionality) (struct i2c\_adapter \*);

};

如果一个I2C适配器不支持I2C通道，那么就将master\_xfer成员设为NULL。如果适配器支持SMBUS协议，那么需要去实现smbus\_xfer，如果smbus\_xfer指针被设为NULL，那么当使用SMBUS协议的时候将会通过I2C通道进行仿真。master\_xfer指向的函数的返回值应该是已经成功处理的消息数，或者返回负数表示出错了。functionality指针很简单，告诉询问着这个I2C主控器都支持什么功能。

在内核的drivers/i2c/i2c-stub.c中实现了一个i2c adapter的例子，其中实现的是更为复杂的SMBUS。

#### 4.1.1.1 SMBus 与 I2C的区别

通常情况下，I2C和SMBus是兼容的，但是还是有些微妙的区别的。

时钟速度对比：



在电气特性上SMBus要求的电压范围更低。

#### 4.1.1.2 I2C driver

具体的I2C设备驱动，如相机、传感器、触摸屏、背光控制器常见硬件设备大多都有或都是通过I2C协议与主机进行数据传输、控制。结构体如下：

struct i2c\_driver {

unsigned int class;

/\* Notifies the driver that a new bus has appeared or is about to be

\* removed. You should avoid using this, it will be removed in a

\* near future.

\*/

int (\*attach\_adapter)(struct i2c\_adapter \*) \_\_deprecated; //旧的与设备进行绑定的接口函数

int (\*detach\_adapter)(struct i2c\_adapter \*) \_\_deprecated; //旧的与设备进行解绑的接口函数

/\* Standard driver model interfaces \*/

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*); //现行通用的与对应设备进行绑定的接口函数

int (\*remove)(struct i2c\_client \*); //现行通用与对应设备进行解绑的接口函数

/\* driver model interfaces that don't relate to enumeration \*/

void (\*shutdown)(struct i2c\_client \*); //关闭设备

int (\*suspend)(struct i2c\_client \*, pm\_message\_t mesg); //挂起设备，与电源管理有关，为省电

int (\*resume)(struct i2c\_client \*); //从挂起状态恢复

/\* Alert callback, for example for the SMBus alert protocol.

\* The format and meaning of the data value depends on the protocol.

\* For the SMBus alert protocol, there is a single bit of data passed

\* as the alert response's low bit ("event flag").

\*/

void (\*alert)(struct i2c\_client \*, unsigned int data);

/\* a ioctl like command that can be used to perform specific functions

\* with the device.

\*/

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);

struct device\_driver driver; //I2C设备的驱动模型

const struct i2c\_device\_id \*id\_table; //匹配设备列表

/\* Device detection callback for automatic device creation \*/

int (\*detect)(struct i2c\_client \*, struct i2c\_board\_info \*);

const unsigned short \*address\_list;

struct list\_head clients;

};

#define to\_i2c\_driver(d) container\_of(d, struct i2c\_driver, driver) //一般编写驱动过程中对象常是driver类型，可以通过to\_i2c\_driver找到其父类型i2c\_driver

如同普通设备的驱动能够驱动多个设备一样，一个I2C driver也可以对应多个I2C client。

以重力传感器AXLL34X为例，其实现的I2C驱动为：

static const struct i2c\_device\_id adxl34x\_id[] = {

{ "adxl34x", 0 }, //匹配i2c client名为adxl34x的设备

{ }

};

MODULE\_DEVICE\_TABLE(i2c, adxl34x\_id);

static struct i2c\_driver adxl34x\_driver = {

.driver = {

.name = "adxl34x",

.owner = THIS\_MODULE,

.pm = &adxl34x\_i2c\_pm, //指定设备驱动的电源管理接口，包含suspend、resume

},

.probe = adxl34x\_i2c\_probe, //组装设备匹配时候的匹配动作

.remove = adxl34x\_i2c\_remove, //组装设备移除接口

.id\_table = adxl34x\_id, //制定匹配设备列表

};

module\_i2c\_driver(adxl34x\_driver);

这里要说明一下module\_i2c\_driver宏定义(i2c.h)：

#define module\_i2c\_driver(\_\_i2c\_driver) \

module\_driver(\_\_i2c\_driver, i2c\_add\_driver, \

i2c\_del\_driver)

#define i2c\_add\_driver(driver) \

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

module\_driver():

#define module\_driver(\_\_driver, \_\_register, \_\_unregister, ...) \

static int \_\_init \_\_driver##\_init(void) \

{ \

return \_\_register(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \

} \

module\_init(\_\_driver##\_init); \

static void \_\_exit \_\_driver##\_exit(void) \

{ \

\_\_unregister(&(\_\_driver) , ##\_\_VA\_ARGS\_\_); \

} \

module\_exit(\_\_driver##\_exit);

理解上述宏定义后，将module\_i2c\_driver(adxl34x\_driver)展开就可以得到：

static int \_\_int adxl34x\_driver\_init(void)

{

return i2c\_register\_driver(&adxl34x\_driver);

}

module\_init(adxl34x\_driver\_init);

static void \_\_exit adxl34x\_driver\_exit(void)

{

return i2c\_del\_driver(&adxl34x\_driver);

}

module\_exit(adxl34x\_driver\_exit);

这一句宏就解决了模块module安装卸载的复杂代码。这样驱动开发者在实现I2C驱动时只要将i2c\_driver结构体填充进来就可以了，无需关心设备的注册与反注册过程。

#### 4.1.1.3 I2C client

即I2C设备。I2C设备的注册一般在板级代码中，在解析实例前还是先熟悉几个定义：

struct i2c\_client {

unsigned short flags; //I2C\_CLIENT\_TEN表示设备使用10bit从地址，I2C\_CLIENT\_PEC表示设备使用SMBus检错

unsigned short addr; //设备从地址，7bit。这里说一下为什么是7位，因为最后以为0表示写，1表示读，通过对这个7bit地址移位处理即可。addr<<1 & 0x0即写，addr<<1 | 0x01即读。

char name[I2C\_NAME\_SIZE]; //从设备名称

struct i2c\_adapter \*adapter; //此从设备依附于哪个adapter上

struct i2c\_driver \*driver; // 此设备对应的I2C驱动指针

struct device dev; // 设备模型

int irq; // 设备使用的中断号

struct list\_head detected; //用于链表操作

};

#define to\_i2c\_client(d) container\_of(d, struct i2c\_client, dev) //通常使用device设备模型进行操作，可以通过to\_i2c\_client找到对应client指针

struct i2c\_board\_info {

char type[I2C\_NAME\_SIZE]; //设备名，最长20个字符，最终安装到client的name上

unsigned short flags; //最终安装到client.flags

unsigned short addr; //设备从地址slave address，最终安装到client.addr上

void \*platform\_data; //设备数据，最终存储到i2c\_client.dev.platform\_data上

struct dev\_archdata \*archdata;

struct device\_node \*of\_node; //OpenFirmware设备节点指针

struct acpi\_dev\_node acpi\_node;

int irq; //设备采用的中断号，最终存储到i2c\_client.irq上

};

//可以看到，i2c\_board\_info基本是与i2c\_client对应的。

#define I2C\_BOARD\_INFO(dev\_type, dev\_addr) \

.type = dev\_type, .addr = (dev\_addr)

//通过这个宏定义可以方便的定义I2C设备的名称和从地址（别忘了是7bit的）

下面还是以adxl34x为例：

static struct i2c\_board\_info i2c0\_devices[] = {

{

I2C\_BOARD\_INFO("ak4648", 0x12),

},

{

I2C\_BOARD\_INFO("r2025sd", 0x32),

},

{

I2C\_BOARD\_INFO("ak8975", 0x0c),

.irq = intcs\_evt2irq(0x3380), /\* IRQ28 \*/

},

{

I2C\_BOARD\_INFO("adxl34x", 0x1d),

.irq = intcs\_evt2irq(0x3340), /\* IRQ26 \*/

},

};

...

i2c\_register\_board\_info(0, i2c0\_devices, ARRAY\_SIZE(i2c0\_devices));

这样ADXL34X的i2c设备就被注册到了系统中，当名字与i2c\_driver中的id\_table中的成员匹配时就能够出发probe匹配函数了。

带着问题去分析可能会更有帮助吧，通过对1. 的了解后，可能会产生以下的几点疑问：

* i2c\_adapter驱动如何添加？
* i2c\_client与i2c\_board\_info究竟是什么关系？

### 4.1.2 I2C子系统驱动模块的API

Linux内核的I2C子系统对驱动模块的API有以下：

// 对外数据结构

struct i2c\_driver — 代表一个I2C设备驱动

struct i2c\_client — 代表一个I2C从设备

struct i2c\_board\_info — 从设备创建的模版

I2C\_BOARD\_INFO — 创建I2C设备的宏，包含名字和地址

struct i2c\_algorithm — 代表I2C传输方法

struct i2c\_bus\_recovery\_info — I2C总线恢复信息？内核新加入的结构，不是很清楚。

//对外函数操作

module\_i2c\_driver — 注册I2C设备驱动的宏定义

i2c\_register\_board\_info — 静态声明（注册）I2C设备，可多个

i2c\_verify\_client — 如果设备是i2c\_client的dev成员则返回其父指针，否则返回NULL。用来校验设备是否为I2C设备

i2c\_lock\_adapter — I2C总线持锁操作，会找到最根源的那个i2c\_adapter。说明你的模块必须符合GPL协议才可以使用这个接口。后边以GPL代表。

i2c\_unlock\_adapter — 上一个的反操作，GPL

i2c\_new\_device — 由i2c\_board\_info信息声明一个i2c设备（client），GPL

i2c\_unregister\_device — 上一个的反操作，GPL。

i2c\_new\_dummy — 声明一个名为dummy（指定地址）的I2C设备，GPL

i2c\_verify\_adapter — 验证是否是i2c\_adapter

i2c\_add\_adapter — 声明I2C适配器，系统动态分配总线号。

i2c\_add\_numbered\_adapter — 同样是声明I2C适配器，但是指定了总线号，GPL

i2c\_del\_adapter — 卸载I2C适配器

i2c\_del\_driver — 卸载I2C设备驱动

i2c\_use\_client — i2c\_client引用数+1

i2c\_release\_client — i2c\_client引用数-1

\_\_i2c\_transfer — 没有自动持锁(adapter lock)的I2C传输接口

i2c\_transfer — 自动持锁的I2C传输接口

i2c\_master\_send — 单条消息发送

i2c\_master\_recv — 单条消息接收

i2c\_smbus\_read\_byte — SMBus “receive byte” protocol

i2c\_smbus\_write\_byte — SMBus “send byte” protocol

i2c\_smbus\_read\_byte\_data — SMBus “read byte” protocol

i2c\_smbus\_write\_byte\_data — SMBus “write byte” protocol

i2c\_smbus\_read\_word\_data — SMBus “read word” protocol

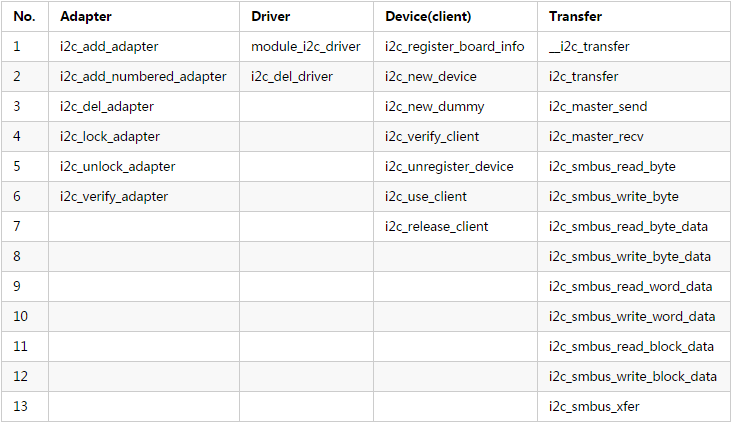
i2c\_smbus\_write\_word\_data — SMBus “write word” protocol

i2c\_smbus\_read\_block\_data — SMBus “block read” protocol

i2c\_smbus\_write\_block\_data — SMBus “block write” protocol

i2c\_smbus\_xfer — execute SMBus protocol operations

1. 对几个基本的结构体和宏定义也有了大概的解释，相信结合I2C的理论基础不难理解。对以上一些I2C的API进行分类：



发现在Linux I2C子系统中，最重要的要数i2c\_client，而最多样化的就是数据的传输。下边的顺序是由client到driver，再到adapter。

### 4.1.3 I2C client的注册

i2c\_client即I2C设备的注册接口有三个：

i2c\_register\_board\_info

i2c\_new\_device

i2c\_new\_dummy

而i2c\_new\_dummy在内部其实也就是将client的name指定为dummy后依旧执行的是i2c\_new\_device，所以就只分析前两个就可以了。首先看这两个函数的原型：

i2c\_register\_board\_info(int busnum, struct i2c\_board\_info const \*info, unsigned len)

//busnum 通过总线号指定这个（些）设备属于哪个总线

//info i2c设备的数组集合 i2c\_board\_info格式

//len 数组个数ARRAY\_SIZE(info)

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

//adap 此设备所依附的I2C适配器指针

//info 此设备描述，i2c\_board\_info格式，bus\_num成员是被忽略的

#### 4.1.3.1使用总线号声明设备i2c\_register\_board\_info

在内核的初始化中定义设备的信息。前提是内核编译的时候已经确定有哪些i2c设备和它们的地址，还要知道连接的总线的编号。

int \_\_init

i2c\_register\_board\_info(int busnum,

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

{

int status;

down\_write(&\_\_i2c\_board\_lock); //i2c设备信息读写锁，锁写操作，其他只读

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

if (busnum >= \_\_i2c\_first\_dynamic\_bus\_num) //与动态分配的总线号相关，动态分配的总线号应该是从已经现有最大总线号基础上+1的，这样能够保证动态分配出的总线号与板级总线号不会产生冲突

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

for (status = 0; len; len--, info++) { //处理info数组中每个成员

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; //组装总线号

devinfo->board\_info = \*info; //组装设备信息

list\_add\_tail(&devinfo->list, &\_\_i2c\_board\_list); //加入到\_\_i2c\_board\_list链表中（尾部）

}

up\_write(&\_\_i2c\_board\_lock); //释放读锁，其他可读可写

return status;

}

怎么将相关信息放到链表中就算完事了吗？不着急，来看下内核中已经给出的解释：

\* Systems using the Linux I2C driver stack can declare tables of board info

\* while they initialize. This should be done in board-specific init code

\* near arch\_initcall() time, or equivalent, before any I2C adapter driver is

\* registered. For example, mainboard init code could define several devices,

\* as could the init code for each daughtercard in a board stack.

\*

\* The I2C devices will be created later, after the adapter for the relevant

\* bus has been registered. After that moment, standard driver model tools

\* are used to bind "new style" I2C drivers to the devices. The bus number

\* for any device declared using this routine is not available for dynamic

\* allocation.

就是说关于集成的I2C设备注册过程应该在板级代码初始化期间，也就是arch\_initcall前后的时间，或者就在这个时候（board-xxx-yyy.c中），切记切记！！！一定要在I2C适配器驱动注册前完成！！！为什么说是静态注册，是因为真实的I2C设备是在适配器成功注册后才被生成的。如果在I2C适配器注册完后还想要添加I2C设备的话，就要通过新方式！（即i2c\_new\_device）

对于\_\_i2c\_board\_list链表中的信息是如何变成实际的i2c设备信息的过程放在之后adapter注册过程的分析中。记得，重点是i2c\_register\_board\_info方式一定要赶在I2C适配器的注册前，这样就没有问题。

#### 4.1.3.2枚举设备i2c\_new\_device或者i2c\_new\_probed\_device

方法1有诸多限制，必须必须在编译内核的时候知道i2c的总线编号和物理的连接。有时开发者面对的是一个已经存在的系统，无法修改内核。

或者内核开发者移植系统的时候也不知道有哪些i2c设备或者到底有多少i2c总线。

在这种情况下就需要用到i2c\_new\_device()了。它的原型是：

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

这个函数将会使用info提供的信息建立一个i2c\_client并与第一个参数指向的i2c\_adapter绑定。返回的参数是一个i2c\_client指针。

驱动中可以直接使用i2c\_client指针和设备通信了。这个方法是一个比较简单的方法。

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); //为即将注册的client申请内存

if (!client)

return NULL;

client->adapter = adap; //绑定指定的adapter适配器

client->dev.platform\_data = info->platform\_data; //保存设备数据

if (info->archdata) //代码上看是DMA相关操作数据

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

client->flags = info->flags; //类型，（一）中说过，或是10位地址，或是使用SMBus检错

client->addr = info->addr; //设备从地址

client->irq = info->irq; //设备终端

strlcpy(client->name, info->type, sizeof(client->name)); //从设备名

//（1.）中说过i2c\_board\_info中的信息是与i2c\_client有对应关系的

/\* Check for address validity \*/

status = i2c\_check\_client\_addr\_validity(client); //检测地址是否有效，10位地址是否大于0x3ff，7位地址是否大于0x7f或为0

if (status) { //非零（实际上为-22，无效参数Invalid argument

dev\_err(&adap->dev, "Invalid %d-bit I2C address 0x%02hx\n",

client->flags & I2C\_CLIENT\_TEN ? 10 : 7, client->addr);

goto out\_err\_silent;

}

/\* Check for address business \*/

status = i2c\_check\_addr\_busy(adap, client->addr); //检测指定适配器上该地址状态

if (status)

goto out\_err;

client->dev.parent = &client->adapter->dev; //建立从设备与适配器的父子关系

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

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

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

ACPI\_HANDLE\_SET(&client->dev, info->acpi\_node.handle);

/\* For 10-bit clients, add an arbitrary offset to avoid collisions \*/

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

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

? 0xa000 : 0)); //如果是10位地址设备，那名字格式与7bit的会有不同

status = device\_register(&client->dev); //注册了！注册了！！！

if (status)

goto out\_err;

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;

}

i2d\_new\_device没什么好多说的，由于有i2c\_register\_board\_info的铺垫，相信也很好理解了。而i2c\_new\_device不但印证了i2c\_client与i2c\_board\_info的对应关系，还顺便体现了10bit地址设备与7bit地址设备的略微不同。通过两者的对比，可以再总结出几点区别，从而更好的理解I2C设备的注册方法：

* i2c\_register\_board\_info的形参需要的是总线号
* i2c\_new\_device的形参需要的直接是适配器的指针

我想这也正好能完美的说明两者的根本区别，对于板级设备更在乎适配器的总线号，因为这是固定的，没有异议的。而对于可插拔设备，因为其适配器可能非板级集成的，所以不能在乎其总线号，反而只要寻求其适配器指针进行绑定即可。后边adapter注册的分析能更好的证明这一点。

* i2c\_register\_board\_info可以同时注册多个I2C设备
* i2c\_new\_device只能一次注册一个I2C设备

这也是其根本区别决定的，板级代码中常包含有许多I2C设备，所以i2c\_register\_board\_info需要有同时注册多个I2C设备的能力也可以说是刚需。而i2c\_new\_device既然是用来给可插拔设备用的，想必设备数量并不多，而常可能只是一个两个而已，所以一次一个就够了，需求量并不大。使用实例如下：

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

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

};

static struct i2c\_client \*at24cxx\_client;

static int at24cxx\_dev\_init(void)

{

struct i2c\_adapter \*i2c\_adap;

i2c\_adap = i2c\_get\_adapter(2);

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

i2c\_put\_adapter(i2c\_adap);

return 0;

}

如果连i2c设备的地址都是不固定的，甚至在不同的板子上有不同的地址，可以提供一个地址列表供系统探测。

此时应该使用的函数是i2c\_new\_probe\_device.。用法如下：

static const unsigned short addr\_list[] = { 0x50, 0x57, 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(2);

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;

}

说明：获取i2c\_adapter指针的函数是：

struct i2c\_adapter\* i2c\_get\_adapter(int id)；//它的参数是i2c总线编号。

获得总线号后，要将注册的i2c设备挂在总线 上。

使用完要释放：

void i2c\_put\_adapter(struct i2c\_adapter \*adap)；

#### 4.1.3.3从用户空间初始化I2C设备

这种方法添加i2c设备又称之为“i2c设备的实例化”。

在一般情况下，内核应该知道哪些I2C设备被连接以及他们的地址是什么。然而，在某些情况下，它没有，所以sysfs接口用作让用户提供信息。这接口是由属性组成，在每一个I2C总线上创建的文件目录：new\_device和delete\_device。这两个文件只写而且为了完成初始化、分别删除一个I2C设备你必须写正确的参数，给他们以正确的实例。

文件new\_device需要2个参数：I2C设备的名称和I2C器件的地址。

文件delete\_device需要一个参数：I2C设备地址。由于没有两个设备可以定义在同一地址上，地址是足以唯一标识一个需要删除的设备。

例如：

创建设备：

#echo at24c08 0x50 > /sys/bus/i2c/devices/i2c-2/new\_device

删除设备：

# echo 0x50 > /sys/bus/i2c/devices/i2c-2/delete\_device

虽然这个接口只用于当内核中的设备声明不能生效时，有各种不同的情况下它是非常有用的：

* I2C驱动程序通常会检测设备（方法3），但总线的设备存在不适当的位组，  因此检测不会触发。
* I2C驱动程序通常会检测设备，但您的​​设备在一个未知的地址。
* I2C驱动程序通常会检测设备，但您的​​设备没有被检测到。 要么因为过于严格的检测程序，或者是因为您  设备不能正常支持，但你知道它是兼容的。
* \*你正在开发一个自己焊接在测试板上的I2C设备驱动程序。

此接口是I2C驱动工具FORCE\_ \*模块的一些参数的替代品。在i2c-core中执行胜过每个设备驱动单独执行，更改设置时你不必重新加载驱动程序，这样做更有效，也更有优势。还可以在设备加载或可用之前初始化的设备的驱动程序，你不需要知道驱动设备的需求是什么。

**总结：**

添加i2c设备的方法很灵活。根据Linux的官方文档添加i2c设备的方法总结有4种：

1） i2c\_register\_board\_info：根据总线编号、设备名字（“at24c08”）、设备地址（0x50）注册一个字符驱动。这种方法最简单、最粗暴，最贴近平时在开片机上开发i2c器件的。

2）. i2c\_new\_device：根据i2c总线的编号，声明一个i2c设备：这种方法就是上面例子用的方法。这种方法也简单，但是需要事先知道器件挂接在哪条总线上。对于设备，还实现知道了设备地址0x50，总线适配器也支持名字为“at24c08”的设备

3） i2c\_new\_probed\_device：根据地址列表通过系统探测出设备地址，这种方法最灵活通用。

4）从用户空间实例化一个器件：这个方法相当智能快速，如下输入指令，即可增加一个i2c设备，同时增加了对应的设备文件。

### 4.1.4 I2C driver

#### 4.1.4.1 I2C设备驱动

Linux内核给出的接口只有两个，一个是注册，另一个就是卸载。在（一）也分析过module\_i2c\_driver这个宏定义，因为有它的存在，I2C设备驱动的开发可以不用在意你的I2C驱动需要如何注册以及如何卸载的，全部的精力都放在i2c\_driver的完善上就可以了。

通过最开始的表单能明显察觉到，I2C子系统中I2C driver的开放接口最少，说白了就是需要驱动编写者完成完了i2c\_driver放入module\_i2c\_driver宏中即可，而正因为如此，也恰恰说明，i2c\_driver的灵活性是最高的。通常驱动会首先在意在用户空间的打开、关闭、读写等接口，但是对于i2c\_driver来说，这些工作是I2C子系统已经做好的，关于常用的读写最终也是通过adapter实现的i2c\_algorithm达到目的。好吧，再次说明了I2C子系统的完善程度，对于I2C设备及驱动开发来说是极其方便的。那么I2C驱动要实现什么呢？

再次回顾一下i2c\_driver结构体，不过现在要剔除一些不常用的成员：

struct i2c\_driver {

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*); //现行通用的与对应设备进行绑定的接口函数

int (\*remove)(struct i2c\_client \*); //现行通用与对应设备进行解绑的接口函数

void (\*shutdown)(struct i2c\_client \*); //关闭设备

int (\*suspend)(struct i2c\_client \*, pm\_message\_t mesg); //挂起设备，与电源管理有关，为省电

int (\*resume)(struct i2c\_client \*); //从挂起状态恢复

struct device\_driver driver; //I2C设备的驱动模型

const struct i2c\_device\_id \*id\_table; //匹配设备列表

...

};

如果有可能的话，我还想再精简一下：

struct i2c\_driver {

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*); //现行通用的与对应设备进行绑定的接口函数

int (\*remove)(struct i2c\_client \*); //现行通用与对应设备进行解绑的接口函数

struct device\_driver driver; //I2C设备的驱动模型

const struct i2c\_device\_id \*id\_table; //匹配设备列表

...

};

好了，精简到这种程度，为什么把电源管理相关也干掉了呢？实际上没有，通常实际的I2C驱动喜欢在drivers中完成这个动作（以mpu3050为例）：

static UNIVERSAL\_DEV\_PM\_OPS(mpu3050\_pm, mpu3050\_suspend, mpu3050\_resume, NULL);

static const struct i2c\_device\_id mpu3050\_ids[] = {

{ "mpu3050", 0 },

{ }

};

MODULE\_DEVICE\_TABLE(i2c, mpu3050\_ids);

static const struct of\_device\_id mpu3050\_of\_match[] = {

{ .compatible = "invn,mpu3050", },

{ },

};

MODULE\_DEVICE\_TABLE(of, mpu3050\_of\_match);

static struct i2c\_driver mpu3050\_i2c\_driver = {

.driver = {

.name = "mpu3050",

.owner = THIS\_MODULE,

.pm = &mpu3050\_pm,

.of\_match\_table = mpu3050\_of\_match,

},

.probe = mpu3050\_probe,

.remove = mpu3050\_remove,

.id\_table = mpu3050\_ids,

};

module\_i2c\_driver(mpu3050\_i2c\_driver);

可以看到，实际驱动中喜欢将电源管理集成在i2c\_driver的driver成员中。

UNIVERSAL\_DEV\_PM\_OPS这个名字很犀利，貌似是“宇宙终极驱动电源管理大法”的样子：

#define UNIVERSAL\_DEV\_PM\_OPS(name, suspend\_fn, resume\_fn, idle\_fn) \

const struct dev\_pm\_ops name = { \

SET\_SYSTEM\_SLEEP\_PM\_OPS(suspend\_fn, resume\_fn) \

SET\_RUNTIME\_PM\_OPS(suspend\_fn, resume\_fn, idle\_fn) \

}

#define SET\_SYSTEM\_SLEEP\_PM\_OPS(suspend\_fn, resume\_fn) \

.suspend = suspend\_fn, \

.resume = resume\_fn, \

.freeze = suspend\_fn, \

.thaw = resume\_fn, \

.poweroff = suspend\_fn, \

.restore = resume\_fn,

#define SET\_RUNTIME\_PM\_OPS(suspend\_fn, resume\_fn, idle\_fn) \

.runtime\_suspend = suspend\_fn, \

.runtime\_resume = resume\_fn, \

.runtime\_idle = idle\_fn,

结合MPU3050的驱动将其完整展开可以得到：

static const struct dev\_pm\_ops mpu3050\_pm = {

.suspend = mpu3050\_suspend,

.resume = mpu3050\_resume,

.freeze = mpu3050\_suspend,

.thaw = mpu3050\_resume,

.poweroff = mpu3050\_suspend,

.restore = mpu3050\_resume,

.runtime\_suspend = mpu3050\_suspend,

.runtime\_resume = mpu3050\_resume,

.runtime\_idle = NULL,

}

对电源管理有兴趣的可以去查阅pm.h，其中对电源管理有详尽的说明了，这里不做分析。可以看到，在电源管理中，有很多成员实际上是一样的，在现实驱动中这样的情况也经常出现，所以会有“终极电源管理大法”宏的出现了。

上边说过，i2c\_driver的多样化最多，从mpu3050的驱动注册中也可以发现，其注重实现的为probe与电源管理，其中probe最为重要（好像是废话，哪个驱动这个都是最重要的-。-）。因为主要是从驱动的角度看待I2C子系统，所以这里不详尽分析mpu3050的代码，只以其为例说明I2C驱动大体框架。在mpu3050的probe主要对此传感器进行上电、工作模式初始化、注册INPUT子系统接口、关联中断处理程序（在中断处理线程中上报三轴参数）等工作。

#### 4.1.4.2 关于I2C设备驱动的小总结

I2C设备驱动通常只是需要挂载在I2C总线（即依附于I2C子系统），I2C子系统对于设备驱动来说只是一个载体、基石。许多设备的主要核心是建立在其他子系统上，如重力传感器、三轴传感器、触摸屏等通常主要工作集中在INPUT子系统中，而相机模块、FM模块、GPS模块大多主要依附于V4L2子系统。这也能通过I2C设计理念证明，I2C的产生正是为了节省外围电路复杂度，让CPU使用有限的IO口挂载更多的外部模块。假设CPU的扩展IO口足够多，我想I2C也没什么必要存在了，毕竟直接操作IO口驱动设备比I2C来的更简单。

### 4.1.5 I2C adapter的注册

#### 4.1.5.1 注册方法

如上表所示，对于I2C adapter的注册有两种途径：i2c\_add\_adapter 或i2c\_add\_numbered\_adapter，两者的区别是后者在注册时已经指定了此I2C适配器的总线号，而前者的总线号将由系统自动分配。

其各自的声明格式为：

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

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

在i2c\_add\_numberd\_adapter使用前必须制定adap->nr，如果给-1，说明还是叫系统去自动生成总线号的。

#### 4.1.5.2 使用场景

之所以区分开两种I2C adapter的注册方式，是因为他们的使用场景有所不同。

* i2c\_add\_adapter的使用经常是用来注册那些可插拔设备，如USB PCI设备等。主板上的其他模块与其没有直接联系，说白了就是现有模块不在乎新加入的I2C适配器的总线号是多少，因为他们不需要。反而这个可插拔设备上的一些模块会需要其注册成功的适配器指针。回看一开始就分析的i2c\_client，会发现不同场景的设备与其匹配的适配器有着这样的对应关系：

1. i2c\_register\_board\_info需要指定已有的busnum，而i2c\_add\_numbered\_adapter注册前已经指定总线号；

2. i2c\_new\_device需要指定adapter指针，而i2c\_add\_adapter注册成功后恰好这个指针就有了。

* i2c\_add\_numbered\_adapter用来注册CPU自带的I2C适配器，或是集成在主板上的I2C适配器。主板上的其他I2C从设备(client)在注册时候需要这个总线号。

通过简短的代码分析看一看他们的区别究竟如何，以及为什么静态注册的i2c\_client必须要在adapter注册前（此处会精简部分代码，只留重要部分）：

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

{

int id, res = 0;

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

\_\_i2c\_first\_dynamic\_bus\_num, &id); //动态获取总线号

adapter->nr = id;

return i2c\_register\_adapter(adapter); //注册adapter

}

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

{

int id;

int status;

if (adap->nr == -1) /\* -1 means dynamically assign bus id \*/

return i2c\_add\_adapter(adap);

status = i2c\_register\_adapter(adap);

return status;

}

最终他们都是通过i2c\_register\_adapter注册适配器：

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

{

int res = 0;

/\* Can't register until after driver model init \*/ //时序检查

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

res = -EAGAIN;

goto out\_list;

}

/\* Sanity checks \*/

if (unlikely(adap->name[0] == '\0')) { //防御型代码，检查适配器名称

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

"no name!\n");

return -EINVAL;

}

if (unlikely(!adap->algo)) { //适配器是否已经完成了通信方法的实现

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

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

return -EINVAL;

}

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;

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

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

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

res = device\_register(&adap->dev); //注册设备节点

if (res)

goto out\_list;

/\* create pre-declared device nodes \*/ //创建预-声明的I2C设备节点

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

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

//如果adapter的总线号小于动态分配的总线号的最小那个，说明是板级adapter。

//因为通过i2c\_add\_adapter加入的适配器所分配的总线号一定是比\_\_i2c\_first\_dynamic\_bus\_num大的。

...

}

对于i2c\_add\_numbered\_adapter来说会触发i2c\_scan\_static\_board\_info：

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

{

struct i2c\_devinfo \*devinfo;

down\_read(&\_\_i2c\_board\_lock); //持有读写锁的读，有用户读的时候不允许写入

list\_for\_each\_entry(devinfo, &\_\_i2c\_board\_list, list) { //又见\_\_i2c\_board\_list，这不是通过i2c\_register\_board\_info组建起来的那个链表吗！

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

&& !i2c\_new\_device(adapter,

&devinfo->board\_info)) //找到总线号与刚注册的这个adapter相同的并通过i2c\_new\_device进行注册

dev\_err(&adapter->dev,

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

devinfo->board\_info.addr);

}

up\_read(&\_\_i2c\_board\_lock); //释放读写锁

}

而i2c\_board\_info成员与i2c\_client的对应动作也是在i2c\_new\_device中进行的，这一点在上边已经分析过了。看到这里，对adapter与client的微妙关系应该了解程度就比较深了，为什么说i2c\_register\_board\_info与i2c\_add\_numbered\_adapter对应而不是i2c\_add\_adapter也可以说得通。

那么，最终回答开篇提出的那两个问题：

* i2c\_adapter驱动如何添加？

板级适配器（CPU自带、主板集成）要通过i2c\_add\_numbered\_adapter注册，注册前要指定总线号，从0开始。假如板级I2C适配器注册了3个，那么第一个动态总线号一定是3，也就是说可插拔设备所带有的I2C适配器需要通过i2c\_add\_adapter进行注册，其总线号由系统指定。

* i2c\_client与i2c\_board\_info究竟是什么关系？

i2c\_client与i2c\_board\_info的对应关系在i2c\_new\_device中有完整体现。

i2c\_client->dev.platform\_data = i2c\_board\_info->platform\_data;

i2c\_client->dev.archdata = i2c\_board\_info->archdata;

i2c\_client->flags = i2c\_board\_info->flags;

i2c\_client->addr = i2c\_board\_info->addr;

i2c\_client->irq = i2c\_board\_info->irq;

#### 4.1.5.3物理i2c总线的编号查询

[root@Loongson:/]#cat /sys/class/i2c-adapter/i2c-1/name

loongson1

[root@Loongson:/]#cat /sys/class/i2c-adapter/i2c-0/name

loongson1

[root@Loongson:/]#cat /sys/class/i2c-adapter/i2c-2/name

loongson1

[root@Loongson:/]#

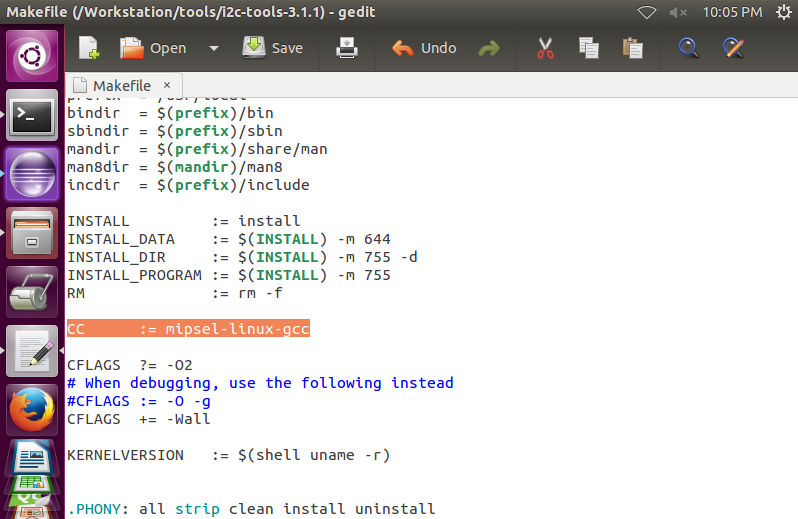
### 4.1.6 I2C tools使用

### 4.1.6.1下载安装

<http://packages.debian.org/search?keywords=i2c-tools>

在虚拟机中解压源码包：

     tar -xvf i2c-tools-3.1.0.tar.bz2  
解压后，编译：修改Makefile文件中编译工具链



最后在tools下生成i2cdetect i2cdump i2cset i2cget

root@ubuntu:/Workstation/tools/i2c-tools-3.1.1# make

mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cdetect.c -o tools/i2cdetect.o

mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cbusses.c -o tools/i2cbusses.o

mipsel-linux-gcc -o tools/i2cdetect tools/i2cdetect.o tools/i2cbusses.o

mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cdump.c -o tools/i2cdump.o

mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/util.c -o tools/util.o

mipsel-linux-gcc -o tools/i2cdump tools/i2cdump.o tools/i2cbusses.o tools/util.o

mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cset.c -o tools/i2cset.o

mipsel-linux-gcc -o tools/i2cset tools/i2cset.o tools/i2cbusses.o tools/util.o

mipsel-linux-gcc -O2 -Wall -Wstrict-prototypes -Wshadow -Wpointer-arith -Wcast-qual -Wcast-align -Wwrite-strings -Wnested-externs -Winline -W -Wundef -Wmissing-prototypes -Iinclude -c tools/i2cget.c -o tools/i2cget.o

mipsel-linux-gcc -o tools/i2cget tools/i2cget.o tools/i2cbusses.o tools/util.o

root@ubuntu:/Workstation/tools/i2c-tools-3.1.1#

#### 4.1.6.2 I2C总线扫描

将6.1编译好的 i2cdetect , i2cdump , i2cget , i2cset   拷备到开发板的新建的一个目录下：

[root@Loongson:/i2ctools]#ls

i2cdetect i2cdump i2cget i2cset

通过i2cdetect -l指令可以查看设备上的I2C总线。

[root@Loongson:/i2ctools]#./i2cdetect -l

i2c-0 i2c i2c-ls1x I2C adapter

i2c-1 i2c i2c-ls1x I2C adapter

i2c-2 i2c i2c-ls1x I2C adapter

#### 4.1.6.3 I2C设备查询

 将设备的SDA连接到GPIO50，SCL连接到GPIO51，若总线上挂载I2C从设备，可通过i2cdetect扫描某个I2C总线上的所有设备。可通过控制台输入i2cdetect -y 2，结果如下所示。

[root@Loongson:/i2ctools]#./i2cdetect -y 2

     0  1  2  3  4  5  6  7  8  9  a  b  c  d  e  f

00:          -- -- -- -- -- -- -- -- -- -- -- -- --

10: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

20: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

30: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

40: -- -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

50: 50 -- -- -- -- -- -- -- -- -- -- -- -- -- -- --

60: -- -- -- -- -- -- -- -- 68 -- -- -- -- -- -- --

70: -- -- -- -- -- -- -- --

说明1：-y为一个可选参数，如果有-y参数的存在则会有一个用户交互过程z，意思是希望用户停止使用该I2C总线。如果写入该参数，则没有这个交互过程，一般该参数在脚本中使用。

说明2：此处I2C总线共挂载两个设备——DS3231和AT24C02，从机地址0x68为DS3231，从机地址0x50为AT24C02。

#### 4.1.6.4 寄存器内容导出

    通过i2cdump指令可导出I2C设备中的所有寄存器内容，例如输入i2cdump -y 1 0x68，可获得以下内容:

[root@Loongson:/i2ctools]#./i2cdump -y 2 0x68

No size specified (using byte-data access)

0 1 2 3 4 5 6 7 8 9 a b c d e f 0123456789abcdef

00: 21 45 12 02 07 04 17 00 00 00 00 00 00 00 1c 88 !E?????.......??

10: 00 1c 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .?..............

20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

30: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

40: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

90: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

b0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

c0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

d0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................

#### 4.1.6.5 寄存器内容写入

    如果向I2C设备中写入某字节，可输入指令/i2cset -y 2 0x50 0x01 0x15

    -y        代表曲线用户交互过程，直接执行指令

    2         代表I2C总线编号

    0x50    代表I2C设备地址，此处选择AT24C02

    0x01    代表存储器地址

    0x15    代表存储器地址中的具体内容

[root@Loongson:/i2ctools]#./i2cset -y 2 0x50 0x01 0x15

[root@Loongson:/i2ctools]#./i2cdump -y 2 0x50

No size specified (using byte-data access)

0 1 2 3 4 5 6 7 8 9 a b c d e f 0123456789abcdef

00: ff 15 ff ff ff ff ff ff ff ff ff ff ff ff ff ff .?..............

10: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

20: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

30: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

40: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

50: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

60: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

70: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

80: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

90: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

a0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

b0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

c0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

d0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

e0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

f0: ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ................

#### 4.1.6.6 寄存器内容读出

    如果从I2C从设备中读出某字节，可输入执行i2cget -y 1 0x50 0x00，可得到以下反馈结果

    -y        代表曲线用户交互过程，直接执行指令

    1         代表I2C总线编号

    0x50    代表I2C设备地址，此处选择AT24C04的低256字节内容

    0x00    代表存储器地址

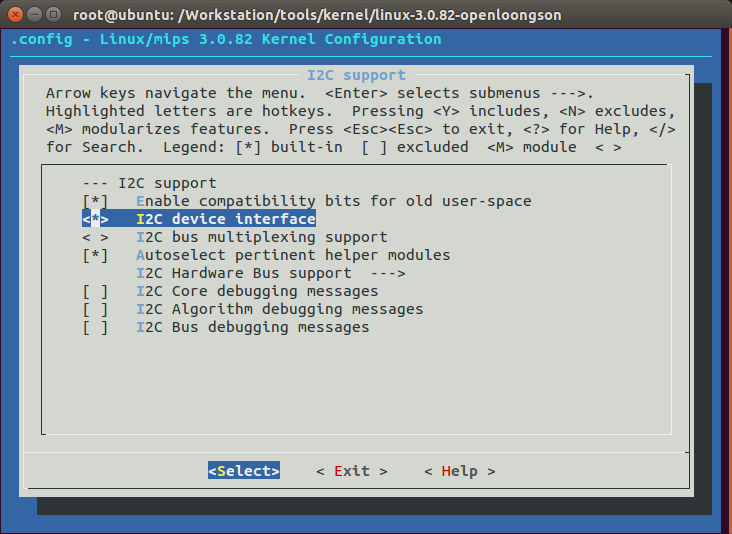
[root@Loongson:/i2ctools]#./i2cget -y 2 0x50 0x01

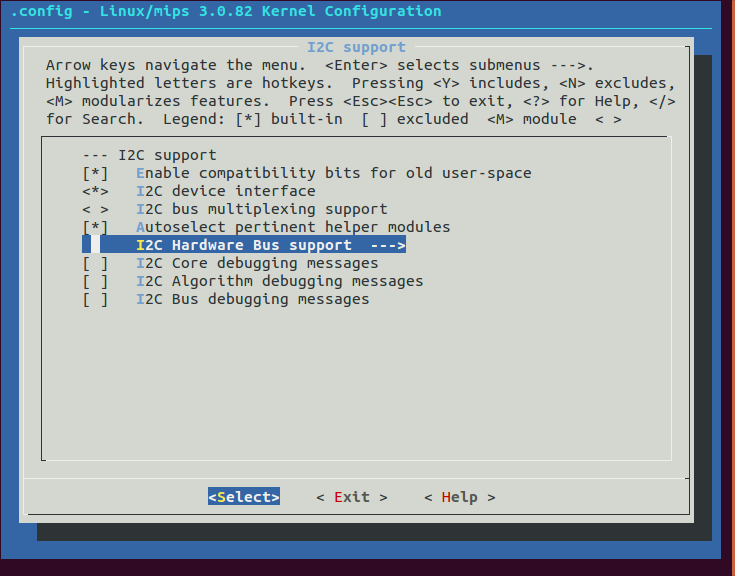
0x15

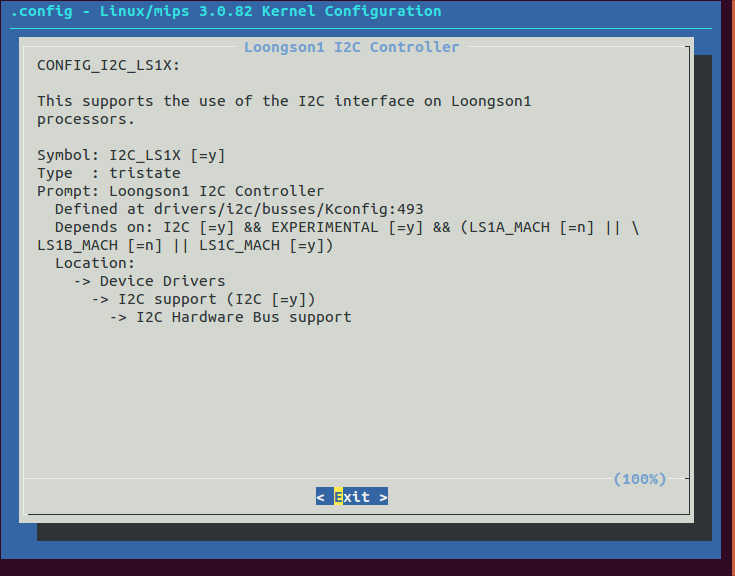
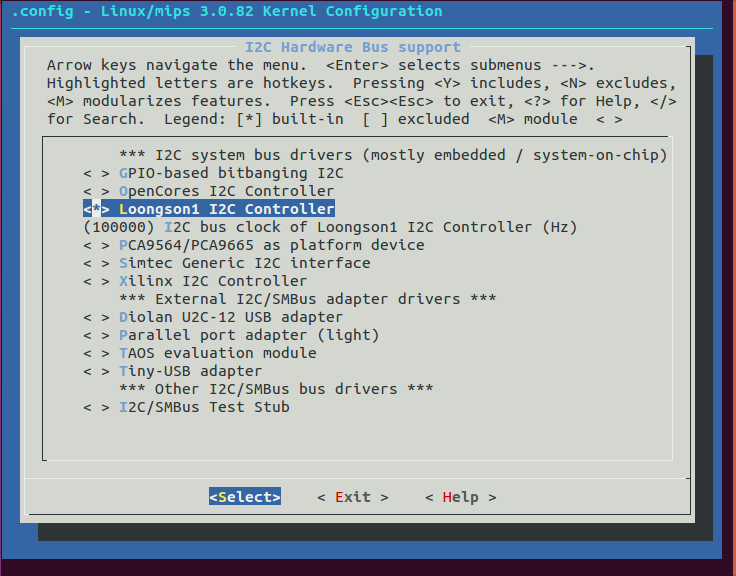
i2ctools是一个简单好用的工具，该工具使得I2C设备的调试更加方便。

### 4.1.7内核模块分析

内核中配置相关I2C的选项已经打开如下





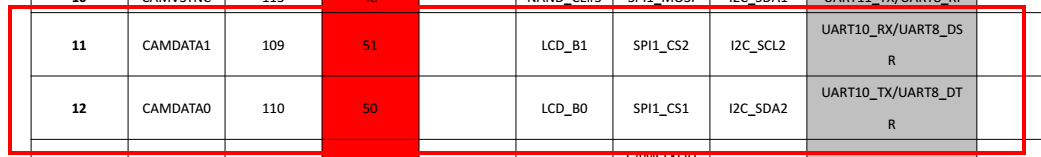


以上编译总共得到了3个驱动文件i2c-core.ko，i2c-dev.ko，i2c\_LS1X.ko, 这三个模块已经编译到内核中，会自动创建/dev/ i2c-0设备节点，然后就可以直接调用/dev/ i2c-0文件节点进行访问设备了。之后应用程序员可以利用下面两种ioctl函数ioctl(fd, I2C\_RDWR, (unsigned long)&work\_queue);或者octl(file,I2C\_SMBUS,&args);进行与i2c设备通信了。

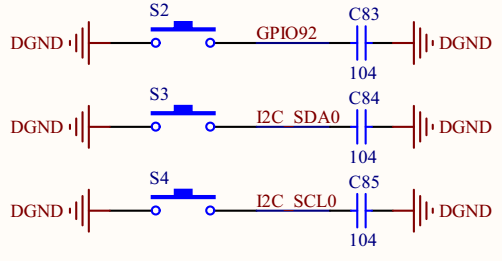
一句话概括之：platform设备是针对硬件资源的分配，platform驱动是软件对已分配的硬件资源的使用。

## 4.2 实例分析at24cxx

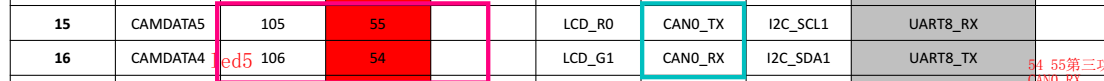
硬件连接：使用I2C2 ，



引脚 GPIO85 86 I2C0，用在了按键上



引脚 GPIO55,54 I2C1，用在了CAN上



步骤如下：

1、注册一个设备，里面要有设备名，以及该设备id，并挂载到这条i2c总线设备链表上

2、注册一个驱动，该设备驱动需要有设备名，probe函数、id\_table设备地址，然后挂载到驱动链表上

3、比较两条链表上的设备名字是否相同，如果有相同的话，就去到相关的驱动设备上的probe函数，进行操作

4、probe函数里面就是正常的驱动程序的编程程序了，主入口、出口、file\_operation结构体的构造等等操作；最终应用程序会进入这里操作硬件对象

### 4.2.1 注册新设备

文件at24cxx\_dev.c:注册新设备， 采用方法i2c\_new \_device。

文件at24cxx\_dev1.c:注册新设备， 采用方法i2c\_new\_probed\_device。

### 4.2.2 注册新驱动

at24cxx\_drv.c: 注册一个新驱动

### 4.2.3 对i2c驱动的操作

完成设备的打开、读、写、关闭等。填充at24cxx\_drv.c中的函数。

### 4.2.4 编译用的Makefile

KERN\_DIR = /Workstation/tools/kernel/linux-3.0.82-openloongson

all:

make -C $(KERN\_DIR) M=`pwd` modules ARCH=mips CROSS\_COMPILE=mipsel-linux-

clean:

make -C $(KERN\_DIR) M=`pwd` modules clean

rm -rf modules.order

#obj-m += at24cxx\_dev.o

#obj-m += at24cxx\_dev1.o

obj-m += at24cxx\_drv.o

### 4.2.5 测试应用编程 test\_at24cxx.c

#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

/\* 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;

}

## 4.3 实例分析DS3231

参考从长计议的《智龙V2通过I2C连接DS3231时钟模块》

<http://www.openloongson.org/forum.php?mod=viewthread&tid=149&extra=page%3D1>

作了改动，将静态注册修改成了动态注册，这样就不需要重新编译内核，原来是：

/\*\*

在platform.c中入加

ls1x\_i2c0\_devs加入成员

{ I2C\_BOARD\_INFO("ds3231", 0x68),},

\*/

修改成：

static struct i2c\_board\_info ds3231\_info = {

I2C\_BOARD\_INFO("ds3231", 0x68),

};

。。。。。。

。。。。。。

static int \_\_init ds3231\_drv\_init(void)

{

。。。。。。

ds3231\_client = i2c\_new\_device(i2c\_adap, &ds3231\_info);

i2c\_put\_adapter(i2c\_adap);

。。。。。。

}