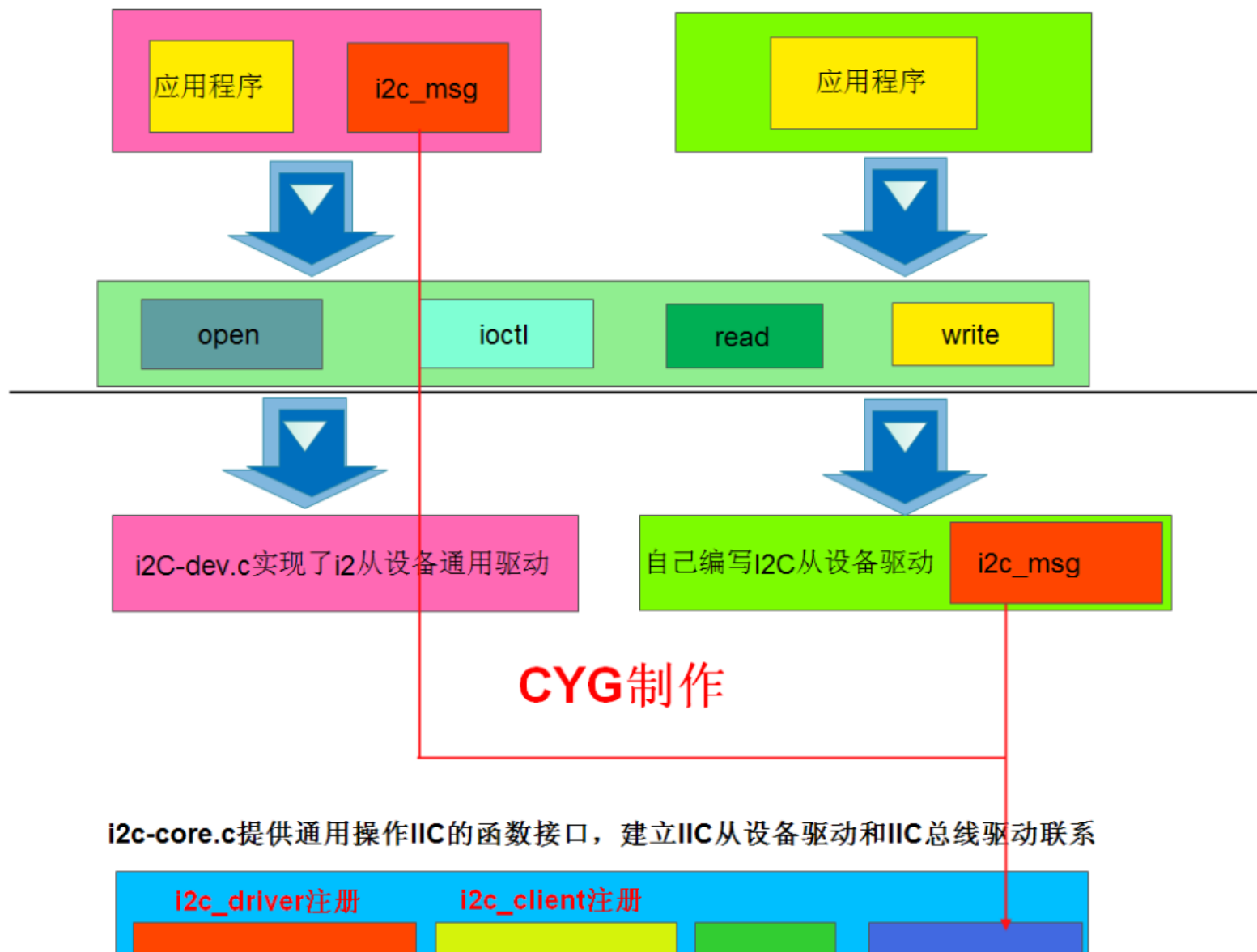
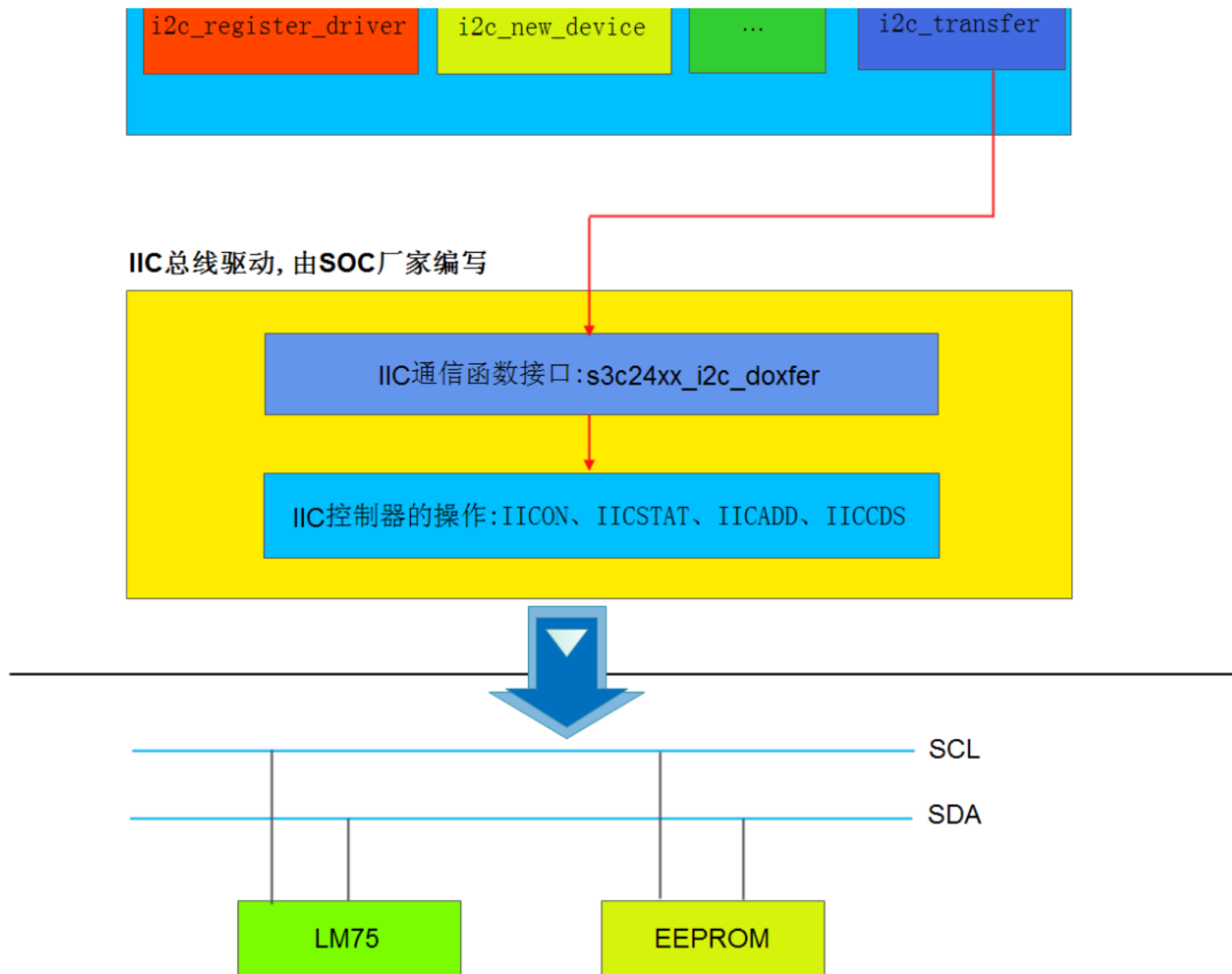


Linux 设备驱动之 IIC 驱动

—编写者:草根老师(程姚根)

一、Linux IIC子系统框架





二、基于i2c-dev通用驱动框架编写应用程序

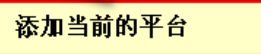
从上面的Linux IIC子系统框架可以看出,i2c-dev.c中已经实现好了通用的IIC从设备驱动。它会为SOC芯片上的每个IIC控制器生成一个主设备号为89的设备节点,并且实现了文件的操作函数接口。用户空间可以通过i2c设备节点,访问i2c控制器。每个i2c控制器的编号从0开始,对应i2c设备文件的次设备号。

注意:如果想看到这个设备节点,我们必须配置一下我们的Linux内核,让它包含我们的IIC控制器驱动。

第一步:在Linux内核中,找到平台的IIC控制器驱动

平台的IIC控制器驱动默认存放在Linux内核源码树:drivers/i2c/busses目录下存放。我们当前平台为s5pc100,但是这个目录并没有s5pc100相关的IIC驱动,但是我们会发现有个i2c-s3c2410.c的文件,这个是针对s3c2410平台编写的IIC控制器驱动。比较幸运的是s3c2410的IIC控制器操作方法和s5pc100的IIC控制器操作方法一样。**好了接下来我们修改一下这个目录下的Kconfig文件,让它支持我们的s5pc100平台。**修改如下:

```
config I2C_S3C2410
    tristate "S3C2410 I2C Driver"
    depends on ARCH_S3C2410 || ARCH_S3C64XX || ARCH_S5PC100
    help
        Say Y here to include support for I2C controller in the
        Samsung S3C2410 based System-on-Chip devices.
```



第二步:配置Linux内核,让内核包含IIC控制器驱动

```
Device Drivers --->
  <*> I2C support --->
    I2C Hardware Bus support --->
      <*> S3C2410 I2C Driver
```

需要注意的是,前面的Kconfig一定要修改正确,否则这里将看不到S3C2410 I2C Driver。配置好后重新编译Linux内核,然后重新启动开发板,家在新编译好的Linux内核,启动结束后,我们在/dev目录下可以看到i2c控制器驱动对应的设备节点。

```
[cyg@fsc100 ~]# ls /dev/i2c-* -l
crw-rw----  1 root    root      89,   0 Jan  1 00:00 /dev/i2c-0
crw-rw----  1 root    root      89,   1 Jan  1 00:00 /dev/i2c-1
[cyg@fsc100 ~]#
```

从上面的Linux IIC子系统框架可以看出,每层之间的数据交互是通过i2c_msg来打包传递的。如果我们想基于i2c-dev提

供的通用IIC从设备驱动框架编写应用程序，我们在应用层，必须要填充好i2c_msg。

好了，我们先来看看i2c_msg是如何定义的？

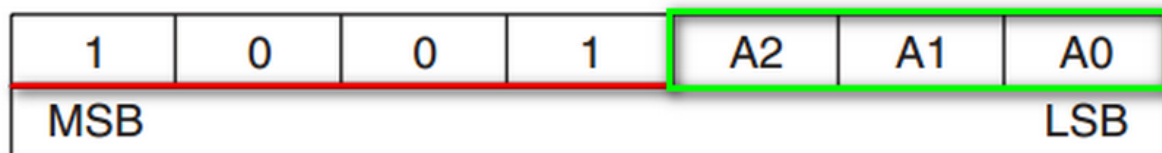
在/usr/include/linux/i2c.h文件中，可以看到i2c_msg结构体的定义，定义如下：

```
/**
 * struct i2c_msg - an I2C transaction segment beginning with START
 *
 * @addr: Slave address, either seven or ten bits. When this is a ten
 * bit address, I2C_M_TEN must be set in @flags and the adapter
 * must support I2C_FUNC_10BIT_ADDR.
 *
 * @flags: I2C_M_RD is handled by all adapters.
 *
 * @len: Number of data bytes in @buf being read from or written to the
 *       I2C slave address.
 *
 * @buf: The buffer into which data is read, or from which it's written.
 */
struct i2c_msg {
    __u16 addr;                /* slave address */
    __u16 flags;
#define I2C_M_TEN              0x0010 /* this is a ten bit chip address */
#define I2C_M_WR               0x0000 /* write data, from master to slave */
#define I2C_M_RD               0x0001 /* read data, from slave to master */
    __u16 len;                /* msg length */
    __u8 *buf;                /* pointer to msg data */
};
```

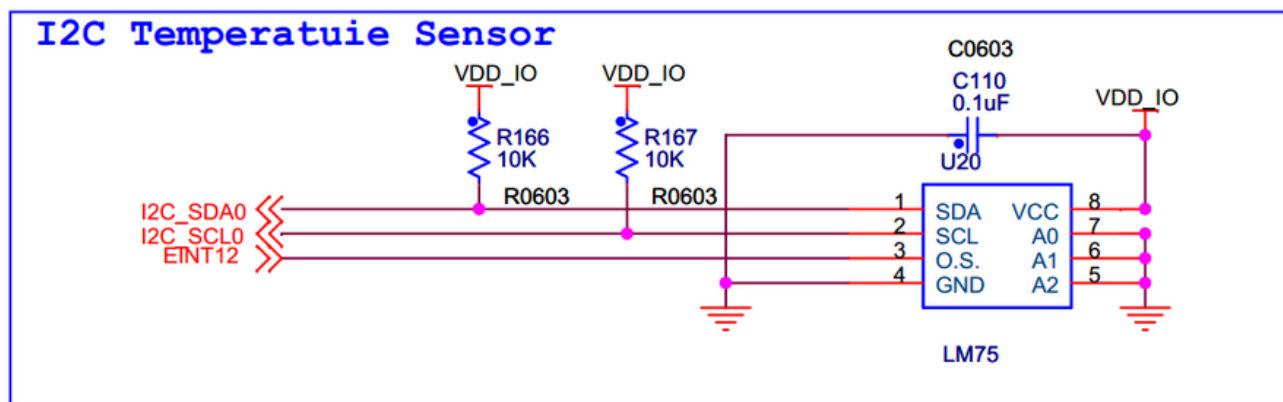
需要注意的是I2C_M_WR是我自己添加的，标准的头文件中并没有这一项。下面我们以lm75设备为例，来看看如何填充这个结构体。

(1)lm75 从机地址

从lm75手册中可以看到，lm75从机地址如下:



从硬件原理图中，我们可以看到lm75接线如下:



所以，我们可以确定lm75作为从机时，其从机地址为0x48。

(2) lm75的温度格式

从lm75的手册中我们可以查询到如下信息:

1.12 TEMPERATURE REGISTER

(Read Only):

D15	D14	D13	D12	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
MSB	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	LSB	X	X	X	X	X	X	X

D0–D6: Undefined

D7–D15: Temperature Data. One LSB = 0.5°C . Two's

可以看出，当我们从lm75中读取的温度用两个字节表示，第一个字节表示温度的整数部分，第二个字节表示小数部分。

第二个字节的最高位为1表示，小数部分为0.5，第二个字节的最高位为0表示，小数部分为0。

好了，有这些信息，我们就可以填充i2c_msg结构体了，填充如下：

```
struct i2c_msg msg[1]; 如果要发送的i2_msg有多个，则用数组
```

```
//填充i2c msg
msg[0].addr = LM75_SLAVE_ADDR; //从机地址
msg[0].flags = I2C_M_RD;        //读从机
msg[0].len = 2;                 //从机器中读取2个字节数据
msg[0].buf = temp;              //从从机中读取的数据存放的地址
```

嗯，我们在上层已经填充好了i2c_msg结构体，怎么告诉驱动层呢？

回顾一下，我们编写的字符驱动提供的接口有：open / read / write / ioctl。聪明的你一定猜到了，i2c-dev中已经实现了这些接口。我们只需要调用这些接口，就可以将我们的i2c_msg传递下去。虽然，我们可以通过read/write来传递i2c-msg，但是大神们更喜欢使用ioctl接口。嗯，既然是使用ioctl那肯定需要命令，需要什么命令呢？

我们先来看看，底层支持的命令：

大家可以从/usr/include/linux/i2c-dev.h文件中看到i2c-dev在底层支持的命令。这里只介绍一些常用的命令。

```
/*
 *dev/i2c-X ioctl commands.
 */
#define I2C_RETRIES 0x0701 /* number of times a device address should
                             be polled when not acknowledging */
#define I2C_TIMEOUT 0x0702 /* set timeout in units of 10 ms */
#define I2C_RDWR 0x0707 /* Combined R/W transfer (one STOP only) */
```

```
#define I2C_FUNCS    0x0705  /* Get the adapter functionality mask */
```

1.设置重试次数----- I2C_RETRIES

```
ioctl(fd, I2C_RETRIES, count);
```

设置适配器收不到ACK时重试的次数为count。默认的重试次数为1。

2.设置超时----- I2C_TIMEOUT

```
ioctl(fd, I2C_TIMEOUT, time);
```

设置超时时间为time，单位为jiffies。

3.获取适配器功能-----I2C_FUNCS

```
ioctl(file, I2C_FUNCS, (unsignedlong *) &func)
```

获取的i2c适配器功能保存在func中。获取的是一个整数，这个整数的每一位表示i2c适配器支持的功能。

大家可以再/usr/include/linux/i2c.h头文件中，查询到每一位支持代表的功能。

常用的适配器功能查询位如下：

```
83 /* To determine what functionality is present */
84
85 #define I2C_FUNC_I2C                0x00000001
86 #define I2C_FUNC_10BIT_ADDR         0x00000002
87
```

4.读写数据-----I2C_RDWR

```
ioctl(file, I2C_RDWR, (struct i2c_rdwr_ioctl_data *)&xxx_ioctl_data);
```

这一行代码可以使用I2C协议和设备进行通信。它进行连续的读写，中间没有间歇。只有当适配器支持I2C_FUNC_I2C此命令才有效。参数是一个指针，指向一个结构体，它的定义如：

```
/* This is the structure as used in the I2C_RDWR ioctl call */
struct i2c_rdwr_ioctl_data {
    struct i2c_msg *msgs;    /* pointers to i2c_msgs */
    __u32 nmsgs;             /* number of i2c_msgs */
};
```

哈哈，到这里你应该这知道，怎么传递我们的i2_msg到底层了吧！

下面给出读取lm75温度的i2c应用程序：

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <linux/i2c.h>
4  #include <linux/i2c-dev.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8
9  #define LM75_SLAVE_ADDR 0x48
10
11 int read_lm75_temp(int fd,unsigned char *temp)
12 {
13     struct i2c_msg  msg[1];
14
15     //填充i2c msg
16     msg[0].addr  = LM75_SLAVE_ADDR;//从机地址
17     msg[0].flags = I2C_M_RD;      //读从机
18     msg[0].len   = 2;             //从机器中读取2个字节数据
19     msg[0].buf   = temp;         //从从机中读取的数据存放的地址
20
21
22     struct i2c_rdwr_ioctl_data  lm75_data;
23
24     //填充i2c ioctl data
25     lm75_data.msgs = msg;
26     lm75_data.nmsgs = sizeof(msg)/sizeof(msg[0]);
27
28     if(ioctl(fd,I2C_RDWR,&lm75_data) < 0)
29     {
30         perror("Fail to ioctl");
31         return -1;
32     }
33
34
35     return 0;
36 }
37
38 int main(int argc, const char *argv[])
39 {
40     int fd;
```



```

41     int ret;
42     unsigned char temp[2];
43
44     fd = open("/dev/i2c-0",O_RDONLY);
45     if(fd < 0){
46         perror("Fail to open");
47         exit(EXIT_FAILURE);
48     }
49     printf("fd = %d\n",fd);
50
51     while(1)
52     {
53         ret = read_lm75_temp(fd,temp);
54         if(ret < 0){
55             printf("Fail to read lm75\n");
56             return -1;
57         }
58
59         printf("temp : %d.%d\n",temp[0],(temp[1] >> 7) == 1 ? 5 : 0);
60         sleep(1);
61     }
62
63     return 0;
64 }
65

```

注意:上面的应用程序并没有去设置重试次数、设置超时、获取适配器功能等操作，大家在编写应用程序的时候可以把这些加上。

三、IIC从设备驱动编写

虽然我们可以基于i2c-dev提供的驱动编写应用程序来操作i2c从设备，但是这对上层的应用的工程师要求比较高。很多时候，我们都会自己编写i2c从设备驱动，向上层应用工程师提供更简单的接口。下面我们就来看看，如何基于Linux IIC子系统编写i2c 从设备驱动。

我们先简单回顾一下，前面我在讲设备模型时，提到总线，设备，驱动三个概念。在总线这一块，我们学习了如何基

于platform总线，编写驱动程序。

在注册驱动的时候，我们先填充好platform_driver结构体,然后通过platform_driver_register()注册驱动。

在注册设备的时候，我们先填充好platform_device结构体,然后通过platform_device_register()注册设备。

设备和驱动在注册的时候，会通过名字相互匹配，匹配成功后，会执行platform_driver提供的probe()函数。我们最终在probe()函数里面，注册了字符设备。

那基于IIC总线写驱动，是不是也有对应结构体和函数接口呢？

1.IIC总线上驱动注册

(1)驱动描述结构体

```
/**
 * struct i2c_driver - represent an I2C device driver
 * @probe: Callback for device binding
 * @remove: Callback for device unbinding
 * @id_table: List of I2C devices supported by this driver
 */
struct i2c_driver {

    /* Standard driver model interfaces */
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
    int (*remove)(struct i2c_client *);

    struct device_driver driver;
    const struct i2c_device_id *id_table;
    ...
};
```

(2)驱动注册函数

```
static inline int i2c_add_driver(struct i2c_driver *driver)
{
    return i2c_register_driver(THIS_MODULE, driver);
}
```

(3)驱动注销函数

```
/**
 * i2c_del_driver - unregister I2C driver
 * @driver: the driver being unregistered
 * Context: can sleep
 */
void i2c_del_driver(struct i2c_driver *driver)
{
    mutex_lock(&core_lock);
    bus_for_each_dev(&i2c_bus_type, NULL, driver, __process_removed_driver);
    mutex_unlock(&core_lock);

    driver_unregister(&driver->driver);
    pr_debug("i2c-core: driver [%s] unregistered\n", driver->driver.name);
}
```

2.IIC总线上从设备注册

(1)IIC总线上从设备的描述结构体

```
/**
 * struct i2c_board_info - template for device creation
 * @type: chip type, to initialize i2c_client.name
 * @flags: to initialize i2c_client.flags
 * @addr: stored in i2c_client.addr
 * @platform_data: stored in i2c_client.dev.platform_data
 * @archdata: copied into i2c_client.dev.archdata
 * @irq: stored in i2c_client.irq
 */
```

```

struct i2c_board_info {
    char        type[I2C_NAME_SIZE];
    unsigned short  flags;
    unsigned short  addr;
    void          *platform_data;
    struct dev_archdata *archdata;
    int           irq;
};

```

一般我们只需要在这个结构体中填充好type(从设备的名字)和addr(从设备地址)两个成员即可。

例如:针对我们的lm75设备, 我们需要做如下事情

```

1  修改arch/arm/mach-s5pc100/mach-smdkc100.c
2
3  static struct i2c_board_info i2c_devs0[] __initdata = {
4  };
5  为:
6  static struct i2c_board_info i2c_devs0[] __initdata = {
7      { I2C_BOARD_INFO("lm75", 0x48) },
8  };
9

```

```

/**
 * I2C_BOARD_INFO - macro used to list an i2c device and its address
 * @dev_type: identifies the device type
 * @dev_addr: the device's address on the bus.
 */
#define I2C_BOARD_INFO(dev_type, dev_addr) \
    .type = dev_type, .addr = (dev_addr)

```

(2)向系统中注册设备信息

```

40 /**
41  * i2c_register_board_info - statically declare I2C devices
42  * @busnum: identifies the bus to which these devices belong
43  * @info: vector of i2c device descriptors

```

```

44  * @len: how many descriptors in the vector;
45  */
46  int __init
47  i2c_register_board_info(int busnum, struct i2c_board_info const *info, unsigned len)
48  {
49      int status;
50      for (status = 0; len; len--, info++) {
51          struct i2c_devinfo *devinfo;
52
53          devinfo = kzalloc(sizeof(*devinfo), GFP_KERNEL);
54          if (!devinfo) {
55              pr_debug("i2c-core: can't register boardinfo!\n");
56              status = -ENOMEM;
57              break;
58          }
59          devinfo->busnum = busnum;
60          devinfo->board_info = *info;
61          list_add_tail(&devinfo->list, &__i2c_board_list);
62      }
63      return status;
64  }

```

drivers/i2c/i2c-boardinfo.c

63,6-9

注意:如果他地方如果需要使用这里注册的设备结构信息，只需要遍历链表__i2c_board_list，通过总线号即可找到相应的设备信息。

问题：什么时候会用到这些信息呢？

系统启动的时候，会把从设备信息，注册到__i2c_board_list。

I2C总线驱动(控制器驱动)在匹配到I2C控制器设备的时候，会调用 (我们可以从drivers/i2c/busses/i2c-s3c2410.c的 s3c24xx_i2c_probe函数中查看)

i2c_add_numbered_adapter(&i2c->adap);

|

i2c_register_adapter(adap);

|

```
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);  
}
```

```
    |  
i2c_new_device()  
{  
    1.产生了i2c_client (填充了从设备信息)  
    2.注册了client->device  
}
```

问题:驱动中是如何获得i2c从设备信息呢?

i2c_driver的probe函数在调用的时候, 会将i2c_client结构体传递过去, i2c_client包含了从设备的信息

```
int xxx_probe(struct i2c_client *client, const struct i2c_device_id *id)
```

```
{  
}
```

问题:最终IIC总线上的从设备是不是用*i2c_client*结构体描述的呀?

嗯, 是的呢!, 它的定义如下:

```
struct i2c_client {  
    unsigned short flags;      /* div., see below      */  
    unsigned short addr;      /* chip address - NOTE: 7bit */  
                                /* addresses are stored in the */  
                                /* _LOWER_ 7 bits             */  
    char name[I2C_NAME_SIZE];  
    struct i2c_adapter *adapter; /* the adapter we sit on */  
    struct i2c_driver *driver; /* and our access routines */  
    struct device dev;          /* the device structure */  
    int irq;                   /* irq issued by device */  
    struct list_head detected;  
};
```

3.i2c_msg 传递

```
/**  
 * i2c_transfer - execute a single or combined I2C message  
 * @adap: Handle to I2C bus  
 * @msgs: One or more messages to execute before STOP is issued to  
 *        terminate the operation; each message begins with a START.  
 * @num: Number of messages to be executed.  
 *  
 * Returns negative errno, else the number of messages executed.  
 *  
 * Note that there is no requirement that each message be sent to  
 * the same slave address, although that is the most common model.  
 */  
int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
```

好了，最后给出lm75的设备驱动。

```
1  #include <linux/init.h>
2  #include <linux/module.h>
3  #include <linux/platform_device.h>
4  #include <linux/i2c-dev.h>
5  #include <linux/i2c.h>
6  #include <linux/cdev.h>
7  #include <linux/fs.h>
8  #include <asm/uaccess.h>
9  #include <linux/slab.h>
10
11
12
13
14  struct lm75_device
15  {
16      dev_t devno;
17      struct cdev cdev;
18      struct class *cls;
19      struct i2c_client *client;
20  };
21
22  int s5pc100lm75_open(struct inode *inode, struct file *file)
23  {
24      file->private_data = container_of(inode->i_cdev, struct lm75_device, cdev);
25      return 0;
26  }
27
28
29  int s5pc100lm75_release(struct inode *inode, struct file *file)
30  {
31      return 0;
32  }
33
34  ssize_t s5pc100lm75_read(struct file *file, char __user *buf, size_t len, loff_t *offset)
35  {
36      int ret;
37      unsigned char temp[2];
38      struct i2c_msg msgs[1];
39      struct lm75_device *lm75 = file->private_data;
40      struct i2c_client *client = lm75->client;
41
42      msgs[0].addr = client->addr;
```



```

43     msgs[0].flags = I2C_M_RD;
44     msgs[0].len   = 2;
45     msgs[0].buf   = temp;
46
47     printk("i2c_transfer : ok!\n");
48
49     ret = i2c_transfer(client->adapter, msgs, sizeof(msgs)/sizeof(msgs[0]));
50     if(ret < 0){
51         printk("Failed to i2c transfer!\n");
52         return ret;
53     }
54
55     ret = copy_to_user(buf, temp, sizeof(temp));
56     if (ret) {
57         ret = -EFAULT;
58         goto exit;
59     }
60     ret = 2;
61
62     return ret;
63
64 exit:
65     return ret;
66 }
67
68
69 struct file_operations s5pc100lm75_fops = {
70     .owner    = THIS_MODULE,
71     .open     = s5pc100lm75_open,
72     .release  = s5pc100lm75_release,
73     .read     = s5pc100lm75_read,
74
75 };
76
77 int register_lm75_device(struct lm75_device *lm75)
78 {
79     int ret;
80     struct device *device;
81
82     //初始化cdev
83     cdev_init(&lm75->cdev, &s5pc100lm75_fops);
84     lm75->cdev.owner = THIS_MODULE;
85
86     //动态申请设备号
87     ret = alloc_chrdev_region(&lm75->devno, 0, 1, "s5pc100-lm75");
88     if(ret < 0){

```

```

89     printk("Cannot alloc dev num");
90     return ret;
91 }
92
93 //添加字符设备
94 ret = cdev_add(&lm75->cdev, lm75->devno, 1);
95 if(ret < 0){
96     printk("Failed to add cdev");
97     goto err_cdev_add;
98 }
99
100 //创建led类
101 lm75->cls = class_create(THIS_MODULE, "lm75");
102 if(IS_ERR(lm75->cls)){
103     printk("Failed to class_create\n");
104     ret = PTR_ERR(lm75->cls);
105     goto err_class_create;
106 }
107
108 //导出设备信息到用户空间,由udev来创建设备节点
109 device = device_create(lm75->cls, NULL, lm75->devno, NULL, "lm75");
110 if(IS_ERR(device)){
111     printk("Failed to device_create");
112     ret = PTR_ERR(device);
113     goto err_device_create;
114 }
115
116 return 0;
117
118 err_device_create:
119     device_destroy(lm75->cls, lm75->devno);
120     class_destroy(lm75->cls);
121
122 err_class_create:
123     cdev_del(&lm75->cdev);
124
125 err_cdev_add:
126     unregister_chrdev_region(lm75->devno, 1);
127
128     return ret;
129 }
130
131 static int lm75_probe(struct i2c_client *client, const struct i2c_device_id *id)
132 {
133     int ret;
134     struct lm75_device *lm75;
135

```

```

136     printk("probe success!\n");
137
138     lm75 = kmalloc(sizeof(struct lm75_device), GFP_KERNEL);
139     if(lm75 == NULL){
140         printk("No free memory\n");
141         return -ENOMEM;
142     }
143     lm75->client = client;
144
145     i2c_set_clientdata(client, lm75);
146
147     //注册led_device到系统
148     ret = register_lm75_device(lm75);
149     if(ret < 0){
150         goto err_register_lm75_device;
151     }
152
153     return 0;
154
155 err_register_lm75_device:
156     kfree(lm75);
157
158     return ret;
159 }
160
161
162 static int lm75_remove(struct i2c_client *client)
163 {
164     struct lm75_device *lm75;
165
166     lm75 = i2c_get_clientdata(client);
167     device_destroy(lm75->cls, lm75->devno);
168     class_destroy(lm75->cls);
169     cdev_del(&lm75->cdev);
170     unregister_chrdev_region(lm75->devno, 1);
171     kfree(lm75);
172
173     return 0;
174 }
175
176 // 2. 创建 & 初始化驱动
177 const struct i2c_device_id lm75_ids[] = {
178     // 设备名 0-i2c0
179     {"lm75", 0},
180     {"lm75a", 1},
181 };

```

```
182
183 struct i2c_driver lm75_driver = {
184     .id_table = lm75_ids,
185     .probe = lm75_probe,
186     .remove = __devexit_p(lm75_remove),
187     .driver = {
188         .name = "lm75",
189     },
190 };
191
192
193 int __init lm75_driver_init(void)
194 {
195     int ret;
196
197     printk("Register lm75 driver to i2c bus!\n");
198
199     ret = i2c_add_driver(&lm75_driver);
200
201     return ret;
202 }
203
204 void __exit lm75_driver_exit(void)
205 {
206     printk("Remove lm75 driver from i2c bus!\n");
207
208     i2c_del_driver(&lm75_driver);
209
210     return;
211 }
212
213 module_init(lm75_driver_init);
214 module_exit(lm75_driver_exit);
215
```