# 实例解析 linux 内核 I2C 体系结构

一、概述

谈到在 linux 系统下编写 I2C 驱动，目前主要有两种方式，一种是把 I2C 设备当作一个普通的字符设备来处理，另一种是利用 linux I2C 驱动体系结构来完成。下面比较下这两种驱动。

第一种方法的好处（对应第二种方法的劣势）有：

- 思路比较直接，不需要花时间去了解 linux 内核中复杂的 I2C 子系统的操作方法。

第一种方法问题（对应第二种方法的好处）有：

- 要求工程师不仅要对 I2C 设备的操作熟悉，而且要熟悉 I2C 的适配器操作；
- 要求工程师对 I2C 的设备器及 I2C 的设备操作方法都比较熟悉，最重要的是写出的程序可移植性差；
- 对内核的资源无法直接使用。因为内核提供的所有 I2C 设备器及设备驱动都是基于 I2C 子系统的格式。I2C 适配器的操作简单还好，如果遇到复杂的 I2C 适配器（如：基于 PCI 的 I2C 适配器），工作量就会大很多。

本文针对的对象是熟悉 I2C 协议，并且想使用 linux 内核子系统的开发人员。

网络和一些书籍上有介绍 I2C 子系统的源码结构。但发现很多开发人员看了这些文章后，还是不清楚自己究竟该做些什么。究其原因还是没弄清楚 I2C 子系统为我们做了些什么，以及我们怎样利用 I2C 子系统。本文首先要解决是如何利用现有内核支持的 I2C 适配器，完成对 I2C 设备的操作，然后再过度到适配器代码的编写。本文主要从解决问题的角度去写，不会涉及特别详细的代码跟踪。

二、I2C 设备驱动程序编写

首先要明确适配器驱动的作用是让我们能够通过它发出符合 I2C 标准协议的时序。

在 Linux 内核源代码中的 drivers/i2c/busses 目录下包含着一些适配器的驱动。如 S3C2410 的驱动 i2c-s3c2410.c。当适配器加载到内核后，接下来的工作就要针对具体的设备编写设备驱动了。

编写 I2C 设备驱动也有两种方法。一种是利用系统给我们提供的 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 设备，独立编写一个设备驱动。注意：在后一种情况下，是不需要使用 i2c-dev.c 的。

1、利用 i2c-dev.c 操作适配器，进而控制 i2c 设备

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。

需要特别注意的是：i2c-dev.c 的 read()、write()方法都只适合于如下方式的数据格式（可查看内核相关源码）


图 1 单开始信号时序

所以不具有太强的通用性，如下面这种情况就不适用（通常出现在读目标时）。


图 2 多开始信号时序

而且 read()、write()方法只适用用于适配器支持 i2c 算法的情况，如：

```
static const struct i2c_algorithm s3c24xx_i2c_algorithm = {
        .master_xfer = s3c24xx_i2c_xfer,
        .functionality = s3c24xx_i2c_func,
};
```

而不适合适配器只支持 smbus 算法的情况，如：

```
static const struct i2c_algorithm smbus_algorithm = {
    .smbus_xfer = i801_access,
    .functionality = i801_func,
};
```

基于上面几个原因，所以一般都不会使用 i2c-dev.c 的 read()、write()方法。最常用的是 ioctl()方法。ioctl()方法可以实现上面所有的情况（两种数据格式、以及 I2C 算法和 smbus 算法）。

针对 i2c 的算法，需要熟悉 struct i2c_rdwr_ioctl_data 、struct i2c_msg。使用的命令是 I2C_RDWR。

```
struct i2c_rdwr_ioctl_data {
    struct i2c_msg __user *msgs; /* pointers to i2c_msgs */
    __u32 nmsgs; /* number of i2c_msgs */
};
struct i2c_msg {
    _ _u16 addr; /* slave address */
    _ _u16 flags; /*  标志（读、写）  */
```

```
    __u16 len; /* msg length */
    __u8 *buf; /* pointer to msg data */
};
```

针对 smbus 算法，需要熟悉 struct i2c_smbus_ioctl_data。使用的命令是 I2C_SMBUS。对于 smbus 算法，不需要考虑“多开始信号时序”问题。

```
struct i2c_smbus_ioctl_data {
    __u8 read_write; //读、写
    __u8 command; //命令
    __u32 size; //数据长度标识
    union i2c_smbus_data __user *data; //数据
};
```

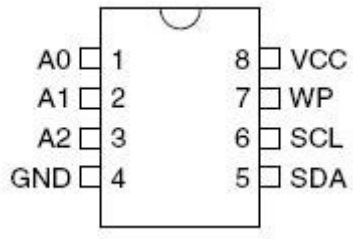
下面以一个实例讲解操作的具体过程。通过 S3C2410 操作 AT24C02 e2prom。实现在 AT24C02 中任意位置的读、写功能。

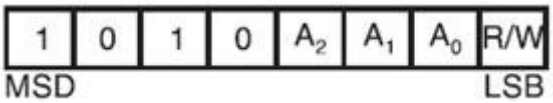
首先在内核中已经包含了对 s3c2410 中的 i2c 控制器驱动的支持。提供了 i2c 算法（非 smbus 类型的，所以后面的 ioctl 的命令是 I2C_RDWR）

```
static const struct i2c_algorithm s3c24xx_i2c_algorithm = {
    .master_xfer = s3c24xx_i2c_xfer,
    .functionality = s3c24xx_i2c_func,
};
```

另外一方面需要确定为了实现对 AT24C02 e2prom 的操作，需要确定 AT24C02 的地址及读写访问时序。

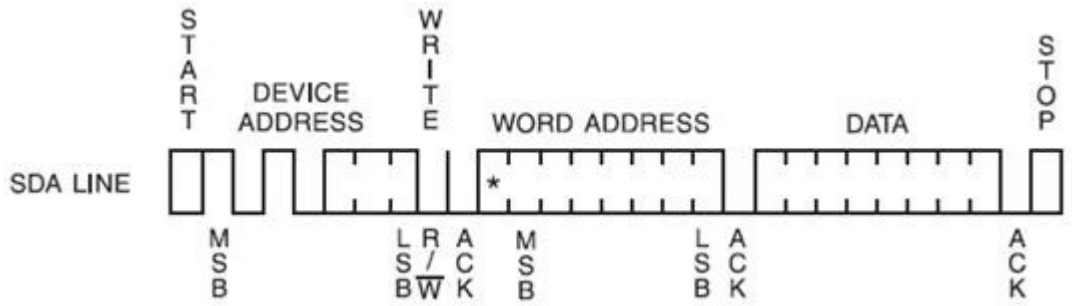
- AT24C02 地址的确定



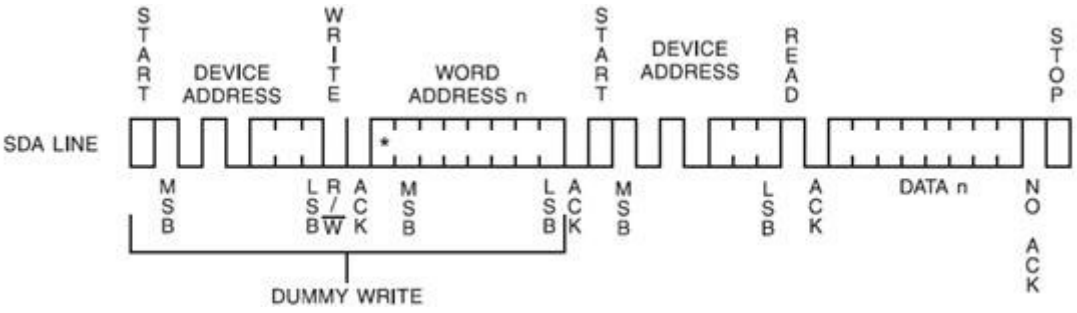

原理图上将 A2、A1、A0 都接地了，所以地址是 0x50。

- AT24C02 任意地址字节写的时序




可见此时序符合前面提到的“单开始信号时序”

- AT24C02 任意地址字节读的时序




可见此时序符合前面提到的“多开始信号时序”

下面开始具体代码的分析（代码在 2.6.22 内核上测试通过）：

```
/*i2c_test.c
* hongtao_liu <lht@farsight.com.cn>
*/
#include <stdio.h>
#include <linux/types.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/ioctl.h>
#include <errno.h>
#define I2C_RETRIES 0x0701
#define I2C_TIMEOUT 0x0702
#define I2C_RDWR 0x0707
/*********定义 struct i2c_rdwr_ioctl_data 和 struct i2c_msg，要和内核一致*******/
```

```
struct i2c_msg
	{
		unsigned short addr;
		unsigned short flags;
	#define I2C_M_TEN 0x0010
	#define I2C_M_RD 0x0001
		unsigned short len;
		unsigned char *buf;
	};

struct i2c_rdwr_ioctl_data
	{
		struct i2c_msg *msgs;
		int nmsgs;
	/* nmsgs 这个数量决定了有多少开始信号，对于“单开始时序”，取 1*/
	};

/***********主程序***********/
	int main()
	{
		int fd,ret;
		struct i2c_rdwr_ioctl_data e2prom_data;
		fd=open("/dev/i2c-0",O_RDWR);
	/*
	*/dev/i2c-0 是在注册 i2c-dev.c 后产生的，代表一个可操作的适配器。如果不使用
i2c-dev.c
	*的方式，就没有，也不需要这个节点。
	*/
		if(fd<0)
		{
			perror("open error");
		}
		e2prom_data.nmsgs=2;
	/*
	*因为操作时序中，最多是用到 2 个开始信号（字节读操作中），所以此将
	*e2prom_data.nmsgs 配置为 2
	*/
		e2prom_data.msgs=(struct
i2c_msg*)malloc(e2prom_data.nmsgs*sizeof(struct i2c_msg));
		if(!e2prom_data.msgs)
		{
			perror("malloc error");
			exit(1);
```

```
        }
        ioctl(fd,I2C_TIMEOUT,1);/*超时时间*/
        ioctl(fd,I2C_RETRIES,2);/*重复次数*/
        /***write data to e2prom**/

        e2prom_data.nmsgs=1;
        (e2prom_data.msgs[0]).len=2; //1 个  e2prom  写入目标的地址和 1 个数据
        (e2prom_data.msgs[0]).addr=0x50;//e2prom  设备地址
        (e2prom_data.msgs[0]).flags=0; //write
        (e2prom_data.msgs[0]).buf=(unsigned char*)malloc(2);
        (e2prom_data.msgs[0]).buf[0]=0x10;// e2prom  写入目标的地址
        (e2prom_data.msgs[0]).buf[1]=0x58;//the data to write

ret=ioctl(fd,I2C_RDWR,(unsigned long)&e2prom_data);
        if(ret<0)
        {
                perror("ioctl error1");
        }
        sleep(1);
/******read data from e2prom*******/
        e2prom_data.nmsgs=2;
        (e2prom_data.msgs[0]).len=1; //e2prom  目标数据的地址
        (e2prom_data.msgs[0]).addr=0x50; // e2prom  设备地址
        (e2prom_data.msgs[0]).flags=0;//write
        (e2prom_data.msgs[0]).buf[0]=0x10;//e2prom 数据地址
        (e2prom_data.msgs[1]).len=1;//读出的数据
        (e2prom_data.msgs[1]).addr=0x50;// e2prom  设备地址
        (e2prom_data.msgs[1]).flags=I2C_M_RD;//read
        (e2prom_data.msgs[1]).buf=(unsigned char*)malloc(1);//存放返回值的地址。
        (e2prom_data.msgs[1]).buf[0]=0;//初始化读缓冲

ret=ioctl(fd,I2C_RDWR,(unsigned long)&e2prom_data);
        if(ret<0)
        {
                perror("ioctl error2");
        }
        printf("buff[0]=%x\n",(e2prom_data.msgs[1]).buf[0]);
/***打印读出的值，没错的话，就应该是前面写的 0x58 了***/
        close(fd);
        return 0;
}
```

以上讲述了一种比较常用的利用 i2c-dev.c 操作 i2c 设备的方法，这种方法可以说是在应用层完成了对具体 i2c 设备的驱动工作。

计划下一篇总结以下几点：

（1）在内核里写 i2c 设备驱动的两种方式：

- Probe 方式（new style），如：

```
static struct i2c_driver pca953x_driver = {
        .driver = {
                .name = "pca953x",
        },
        .probe = pca953x_probe,
        .remove = pca953x_remove,
        .id_table = pca953x_id,
};
```

- Adapter 方式（LEGACY），如：

```
static struct i2c_driver pcf8575_driver = {
        .driver = {
                .owner = THIS_MODULE,
                .name = "pcf8575",
        },
        .attach_adapter = pcf8575_attach_adapter,
        .detach_client = pcf8575_detach_client,
};
```

（2）适配器驱动编写方法

（3）分享一些项目中遇到的问题

希望大家多提意见，多多交流

四、在内核里写 i2c 设备驱动的两种方式

前文介绍了利用/dev/i2c-0 在应用层完成对 i2c 设备的操作，但很多时候我们还是习惯为 i2c 设备在内核层编写驱动程序。目前内核支持两种编写 i2c 驱动程序的方式。下面分别介绍这两种方式的实现。这里分别称这两种方式为“Adapter 方式（LEGACY）”和“Probe 方式（new style）”。

（1） Adapter 方式（LEGACY）

（下面的实例代码是在 2.6.27 内核的 pca953x.c 基础上修改的，原始代码采用的是本文将要讨论的第 2 种方式，即 Probe 方式）

- 构建 i2c_driver

```
static struct i2c_driver pca953x_driver = {
```

```
        .driver = {
                .name= "pca953x", //名称
            },
        .id= ID_PCA9555,//id 号
        .attach_adapter= pca953x_attach_adapter, //调用适配器连接设备
        .detach_client= pca953x_detach_client,//让设备脱离适配器
};
```

- 注册 i2c_driver

```
static int __init pca953x_init(void)
    {
        return i2c_add_driver(&pca953x_driver);
    }
    module_init(pca953x_init);
```

- attach_adapter 动作

执行 i2c_add_driver(&pca953x_driver)后会，如果内核中已经注册了 i2c 适配器，则顺序调用这些适配器来连接我们的 i2c 设备。此过程是通过调用 i2c_driver 中的 attach_adapter 方法完成的。具体实现形式如下：

```
static int pca953x_attach_adapter(struct i2c_adapter *adapter)
    {
        return i2c_probe(adapter, &addr_data, pca953x_detect);
        /*
        adapter:适配器
        addr_data:地址信息
        pca953x_detect：探测到设备后调用的函数
        */
    }
```

地址信息 addr_data 是由下面代码指定的。

```
    /* Addresses to scan */
    static unsigned short normal_i2c[] = {0x20,0x21,0x22,0x23,0x24,0x25,0x26,0x27,I2C_CLIENT_END};
    I2C_CLIENT_INSMOD;
```

注意：normal_i2c 里的地址必须是你 i2c 芯片的地址。否则将无法正确探测到设备。而 I2C_CLIENT_INSMOD 是一个宏，它会利用 normal_i2c 构建 addr_data。

- 构建 i2c_client，并注册字符设备驱动

i2c_probe 在探测到目标设备后，后调用 pca953x_detect，并把当时的探测地址 address 作为

参数传入。

```
static int pca953x_detect(struct i2c_adapter *adapter, int address, int kind)
        {
                struct i2c_client *new_client;
                struct pca953x_chip *chip; //设备结构体
                int err = 0,result;
                dev_t pca953x_dev=MKDEV(pca953x_major,0);//构建设备号，根据具体情
况设定，这里我只考虑了 normal_i2c 中只有一个地址匹配的情况。
                if   (!i2c_check_functionality(adapter,   I2C_FUNC_SMBUS_BYTE_DATA|
I2C_FUNC_SMBUS_WORD_DATA))//判定适配器能力
                goto exit;
                if (!(chip = kzalloc(sizeof(struct pca953x_chip), GFP_KERNEL))) {
                        err = -ENOMEM;
                        goto exit;
                }
                /****构建 i2c-client****/
                chip->client=kzalloc(sizeof(struct i2c_client),GFP_KERNEL);
                new_client = chip->client;
                i2c_set_clientdata(new_client, chip);
                new_client->addr = address;
                new_client->adapter = adapter;
                new_client->driver = &pca953x_driver;
                new_client->flags = 0;
                strlcpy(new_client->name, "pca953x", I2C_NAME_SIZE);
                if ((err = i2c_attach_client(new_client)))//注册 i2c_client
                goto exit_kfree;
                if (err)
                goto exit_detach;
                if(pca953x_major)
                {
                        result=register_chrdev_region(pca953x_dev,1,"pca953x");
                }
                else{
                        result=alloc_chrdev_region(&pca953x_dev,0,1,"pca953x");
                        pca953x_major=MAJOR(pca953x_dev);
                }
                if (result < 0) {
                        printk(KERN_NOTICE "Unable to get pca953x region, error %d\n",
result);
                        return result;
                }
                pca953x_setup_cdev(chip,0); //注册字符设备，此处不详解
                return 0;
```

```
        exit_detach:
                i2c_detach_client(new_client);
exit_kfree:
        kfree(chip);
exit:
        return err;
}
```

i2c_check_functionality 用来判定设配器的能力，这一点非常重要。你也可以直接查看对应设配器的能力，如

```
static const struct i2c_algorithm smbus_algorithm = {
        .smbus_xfer= i801_access,
        .functionality= i801_func,
};
static u32 i801_func(struct i2c_adapter *adapter)
{
                return I2C_FUNC_SMBUS_QUICK | I2C_FUNC_SMBUS_BYTE
|
                I2C_FUNC_SMBUS_BYTE_DATA                        |
I2C_FUNC_SMBUS_WORD_DATA |
                I2C_FUNC_SMBUS_BLOCK_DATA                       |
I2C_FUNC_SMBUS_WRITE_I2C_BLOCK
                        | (isich4 ? I2C_FUNC_SMBUS_HWPEC_CALC : 0);
}
```

- 字符驱动的具体实现

```
struct file_operations pca953x_fops = {
        .owner = THIS_MODULE,
        .ioctl= pca953x_ioctl,
        .open= pca953x_open,
        .release =pca953x_release,
};
```

字符设备驱动本身没有什么好说的，这里主要想说一下，如何在驱动中调用 i2c 设配器帮我们完成数据传输。

目前设配器主要支持两种传输方法：smbus_xfer 和 master_xfer。一般来说，如果设配器支持了 master_xfer 那么它也可以模拟支持 smbus 的传输。但如果只实现 smbus_xfer，则不支持一些 i2c 的传输。

```
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);

master_xfer 中的参数设置，和前面的用户空间编程一致。现在只是要在驱动中构建相关的参数然后调用 i2c_transfer 来完成传输既可。

int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num)

smbus_xfer 中的参数设置及调用方法如下：

```
static int pca953x_write_reg(struct pca953x_chip *chip, int reg, uint16_t val)
        {
                int ret;
                ret = i2c_smbus_write_word_data(chip->client, reg << 1, val);
                if (ret < 0) {
                                dev_err(&chip->client->dev, "failed writing register\n");
                                        return -EIO;
                                }
                return 0;
        }
```

上面函数完成向芯片的地址为 reg 的寄存器写一个 16bit 的数据。i2c_smbus_write_word_data 的实现如下：

```
s32 i2c_smbus_write_word_data(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);
        }
```

从中可以看出 smbus 传输一个 16 位数据的方法。其它操作如：字符写、字符读、字读、块操作等，可以参考内核的 i2c-core.c 中提供的方法。

- 注销 i2c_driver

```
static void __exit pca953x_exit(void)
        {
```

```
        i2c_del_driver(&pca953x_driver);
    }
    module_exit(pca953x_exit);
```

- detach_client 动作

顺序调用内核中注册的适配器来断开我们注册过的 i2c 设备。此过程通过调用 i2c_driver 中的 attach_adapter 方法完成的。具体实现形式如下：

```
static int pca953x_detach_client(struct i2c_client *client)
    {
        int err;
        struct pca953x_chip *data;
        if ((err = i2c_detach_client(client)))//断开 i2c_client
        return err;
        data=i2c_get_clientdata(client);
        cdev_del(&(data->cdev));
        unregister_chrdev_region(MKDEV(pca953x_major, 0), 1);
        kfree(data->client);
        kfree(data);
        return 0;
    }
```

（2） Probe 方式（new style）

- 构建 i2c_driver

和 LEGACY 方式一样，也需要构建 i2c_driver，但是内容有所不同。

```
static struct i2c_driver pca953x_driver = {
        .driver = {
            .name= "pca953x",
            },
            .probe= pca953x_probe, //当有 i2c_client 和 i2c_driver 匹配时调用
            .remove= pca953x_remove,//注销时调用
            .id_table= pca953x_id,//匹配规则
    };
```

- 注册 i2c_driver

```
static int __init pca953x_init(void)
    {
        return i2c_add_driver(&pca953x_driver);
    }
```

```
module_init(pca953x_init);
```

在注册 i2c_driver 的过程中，是将 driver 注册到了 i2c_bus_type 的总线上。此总线的匹配规则是：

```
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;
	}
```

可以看出是利用 i2c_client 的名称和 id_table 中的名称做匹配的。本驱动中的 id_table 为

```
static const struct i2c_device_id pca953x_id[] = {
		{ "pca9534", 8, },
		{ "pca9535", 16, },
		{ "pca9536", 4, },
		{ "pca9537", 4, },
		{ "pca9538", 8, },
		{ "pca9539", 16, },
		{ "pca9554", 8, },
		{ "pca9555", 16, },
		{ "pca9557", 8, },
		{ "max7310", 8, },
		{ }
	};
```

看到现在我们应该会有这样的疑问，在 Adapter 模式中，i2c_client 是我们自己构造出来的，而现在的 i2c_client 是从哪来的呢？看看下面的解释

- 注册 i2c_board_info

对于 Probe 模式，通常在平台代码中要完成 i2c_board_info 的注册。方法如下：

```
static struct i2c_board_info __initdata test_i2c_devices[] = {
		{
			I2C_BOARD_INFO("pca9555", 0x27),//pca9555 为芯片名称，0x27 为芯片地址
```

```
                .platform_data = &pca9555_data,
        }, {
                I2C_BOARD_INFO("mt9v022", 0x48),
                .platform_data = &iclink[0], /* With extender */
        }, {
                I2C_BOARD_INFO("mt9m001", 0x5d),
                .platform_data = &iclink[0], /* With extender */
        },
};
i2c_register_board_info(0, test_i2c_devices,ARRAY_SIZE(test_i2c_devices)); //注册
```

i2c_client 就是在注册过程中构建的。但有一点需要注意的是 i2c_register_board_info 并没有 EXPORT_SYMBOL 给模块使用。

- 字符驱动注册

在 Probe 方式下，添加字符驱动的位置在 pca953x_probe 中。

```
static int __devinit pca953x_probe(struct i2c_client *client,const struct i2c_device_id *id)
{
                ……
                /****字符设备驱动注册位置****/
                ……
                return 0;
}
```

- 注销 i2c_driver

```
static void __exit pca953x_exit(void)
{
        i2c_del_driver(&pca953x_driver);
}
module_exit(pca953x_exit);
```

- 注销字符设备驱动

在 Probe 方式下，注销字符驱动的位置在 pca953x_remove 中。

```
static int __devinit pca953x_remove (struct i2c_client *client)
{
        ……
        /****字符设备驱动注销的位置****/
        ……
        return 0;
```

}

- I2C 设备的数据交互方法(即：调用适配器操作设备的方法)和 Adapter 方式下相同。