原 Linux内核I2C子系统驱动(三)

标签：c linux内核 struct module algorithm table

2011-08-02 21:07 4108人阅读 评论(0) 收藏 举报

分类： Linux设备驱动开发(9)

I2C子系统驱动（三）

当适配器加载到内核后，就针对具体设备编写I2C设备驱动。编写设备驱动有两种方法，一种是利用系统提供的i2c-dev.c实现，另一种为i2c编写一个独立的设备驱动。

一、i2c-dev.c控制i2c设备

i2c-dev.c没有针对具体设备来设计，提供了通用i2cdev_read()、i2cdev_write()函数来对应用户空间要使用的read()和write()文件操作接口，这两个函数分别调用I2C 核心的i2c_master_recv()和i2c_master_send()函数来构造一条I2C 消息并引发适配器algorithm 通信函数的调用，完成消息的传输。但是i2c_read()、i2c_write()对于一些复杂消息(eeprom字节读ReStart模式)并不太适用，针对以上情况，可以使用i2cdev_ioctl()实现消息组的传输，通过I2C_RDWR IOCTL命令实现。

常用的IOCTL包括包括I2C_SLAVE（设置从设备地址）、I2C_RETRIES（没有收到设备ACK 情况下的重试次数，默认为1）、I2C_TIMEOUT（超时）以及I2C_RDWR等。应用程序通过i2c_rdwr_ioctl_data结构体来给内核传递消息，结构体定义如下

```
struct i2c_rdwr_ioctl_data {
    struct i2c_msg __user *msgs; /* pointers to i2c_msgs */
    __u32 nmsgs; /* number of i2c_msgs */
};
```

下面是利用i2c_dev.c操作i2c设备AT24C02具体的代码，这种方法是从应用层完成i2c设备驱动，代码摘自 http://blog.csdn.net/hongtao_liu/article/details/4964244

```cpp
#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
/*********参数定义与内核一致，路径include/linux/i2c-dev.h*******/

struct i2c_msg
{
    unsigned short addr;
    nsigned 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*/
};

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
```

个人资料

liaozc

+ 加关注 发私信

访问：53685次
积分：785
等级：BLOG 3
排名：千里之外

原创：23篇
转载：2篇
译文：0篇
评论：19条

文章搜索

文章分类

FFMPEG (4)
Linux内核根文件 (2)
Linux应用开发 (2)
Linux开发环境 (3)
Linux设备驱动开发 (10)
Qt (1)
U-BOOT (3)

文章存档

2011年08月 (4)
2011年07月 (5)
2011年04月 (4)
2011年02月 (1)
2010年12月 (8)

展开

阅读排行

AVPicture中data与linesi (6310)
Linux内核I2C子系统驱动 (4108)
Ffmpeg移植S3C2440 (3974)
ARM根文件系统制作 (3374)
嵌入式memdev驱动试验 (2812)
Linux内核I2C子系统驱动 (2574)
FFMPEG第三方库编译 (2543)
Linux内核I2C子系统驱动 (2531)
S3C2440驱动篇—Linux (2063)
S3C2440驱动篇—触摸屏 (1976)

评论排行

嵌入式memdev驱动试验 (6)
AVPicture中data与linesi (6)
FFMPEG第三方库编译 (3)
U-Boot1.2.0移植YC2440 (2)
Linux内核I2C子系统驱动 (2)
U-Boot关键代码分析 (0)
Ubuntu9.10构建ARM-Lir (0)
ARM Linux 交叉编译工具 (0)
Linux2.6.24内核移植 (0)
Linux内核I2C子系统驱动 (0)

推荐文章

* 2016 年最受欢迎的编程语言是什么？
* Chromium扩展(Extension)的页面(Page)加载过程分析
* Android Studio 2.2 来啦
* 手把手教你做音乐播放器(二)技术原理与框架设计
* JVM 性能调优实战之：使用阿里开源工具 TProfiler 在海量业务代码中精确定位性能代码

最新评论

AVPicture中data与linesize关系
wzw88486969: linesize 的只分别为:352 176 176 0我测试的是linesize 的只分...

快速回复
返回顶部

```
65.     (e2prom_data.msgs[0]).buf[1]=0x58;//the data to write
66.
67.         ret=ioctl(fd,I2C_RDWR,(unsigned long)&e2prom_data);
68.     if(ret<0)
69.     {
70.         perror("ioctl error1");
71.     }
72.     sleep(1);
73.
74.         /******read data from e2prom*******/
75.     e2prom_data.nmsgs=2;
76.     (e2prom_data.msgs[0]).len=1; //e2prom 目标数据的地址
77.     (e2prom_data.msgs[0]).addr=0x50; // e2prom 设备地址
78.     (e2prom_data.msgs[0]).flags=0;//write
79.     (e2prom_data.msgs[0]).buf[0]=0x10;//e2prom数据地址
80.     (e2prom_data.msgs[1]).len=1;//读出的数据
81.     (e2prom_data.msgs[1]).addr=0x50;// e2prom 设备地址
82.     (e2prom_data.msgs[1]).flags=I2C_M_RD;//read
83.     (e2prom_data.msgs[1]).buf=(unsigned char*)malloc(1);//存放返回值的地址。
84.     (e2prom_data.msgs[1]).buf[0]=0;//初始化读缓冲
85.
86.         ret=ioctl(fd,I2C_RDWR,(unsigned long)&e2prom_data);
87.     if(ret<0)
88.     {
89.         perror("ioctl error2");
90.     }
91.     printf("buff[0]=%x/n",(e2prom_data.msgs[1]).buf[0]);
92.         /***打印读出的值，没错的话，就应该是前面写的0x58了***/
93.     close(fd);
94.     return 0;
95. }
```

二、内核中编写i2c设备驱动

内核编写i2c设备驱动支持两种方式：Adapter方式（LEGACY）和Probe方式（new style）。

1.legacy方式

此方法驱动需要自己创建i2c_client，并且要知道芯片的地址，在内核目录documentation/i2c/upgrading-clients中有一个例程。

i2c_driver构建

```
static struct i2c_driver at24c02_driver = {
    .driver = {
        .name    = "at24c02",
    },
    .id         = I2C_DRIVERID_EEPROM,
    .attach_adapter     = at24c02_attach_adapter,
    .detach_client  = at24c02_detach_client,
};
```

I2C设备驱动的加载与卸载函数模板

```
static int __init at24c02_init(void)
{
    return i2c_add_driver(&at24c02_driver);
}
```

i2c_add_driver的执行回引发i2c_driver结构体at24c02_attach_adapter的执行，若内核中注册了i2c适配器，就顺序调用这些适配器来连接i2c设备，

at24c02_attach_adapter调用i2c核心i2c_probe探测设备

```
static int at24c02_attach_adapter(struct i2c_adapter *adapter)
{
    return i2c_probe(adapter, &addr_data, at24c02_detect);
}
static unsigned short normal_i2c[] = { 0x50, 0x51, 0x52, 0x53, 0x54,
                        0x55, 0x56, 0x57, I2C_CLIENT_END };
/* Insmod parameters */
I2C_CLIENT_INSMOD_1(at24c02);
```

其中第二个参数add_data是i2c_client_address_data类型的变量，由normal_i2c[]通过宏I2C_CLIENT_INSMOD_1构建而成，具体方法参考i2c.h文件。

normal_i2c是i2c芯片的地址，如果地址与芯片对应不上，无法探测到设备，当探测到目标设备后，调用at24c02_detect，把探测到得地址address作为参数传入。

```
#define AT24C02_MAJOR 250
static int at24c02_major = AT24C02_MAJOR;
struct at24c02_data {
    struct cdev cdev;
    struct i2c_client client;
    struct mutex update_lock;
    u8 data[EEPROM_SIZE];           /* Register values */
};
static int at24c02_detect(struct i2c_adapter *adapter, int address, int kind)
{
    struct i2c_client *new_client;
    struct at24c02_data *data;
    int err = 0,result;
    dev_t at24c02_dev=MKDEV(at24c02_major,0);

    if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_READ_BYTE_DATA
                        | I2C_FUNC_SMBUS_BYTE)) //判断适配器功能
        goto exit;

    if (!(data = kzalloc(sizeof(struct at24c02_data), GFP_KERNEL))) {
        err = -ENOMEM;
        goto exit;
    }
    new_client = &data->client;
    memset(data->data, 0xff, EEPROM_SIZE);
    i2c_set_clientdata(new_client, data);
    new_client->addr = address;
    new_client->adapter = adapter;
    new_client->driver = &at24c02_driver;
    new_client->flags = 0;
```

```
    /* Fill in the remaining client fields */
    strlcpy(new_client->name, "at24c02", I2C_NAME_SIZE);
    mutex_init(&data->update_lock);

    /* Tell the I2C layer a new client has arrived */
    if ((err = i2c_attach_client(new_client)))
        goto exit_kfree;

    if(at24c02_major)
    {
        result=register_chrdev_region(at24c02_dev,1,"at24c02");
    }
    else
    {
        result=alloc_chrdev_region(&at24c02_dev,0,1,"at24c02");
        at24c02_major=MAJOR(at24c02_dev);
    }
    if(result<0)
    {
        printk(KERN_NOTICE "Unable to get region %d/n",result);
        err=result;
        goto exit_detach;
    }
    at24c02_setup_cdev(data,0);    //注册字符设备at24c02_fops
    return 0;

exit_detach:
    i2c_detach_client(new_client);
exit_kfree:
    kfree(data);
exit:
    return err;
}
static void at24c02_setup_cdev(struct at24c02_data *dev,int index)
{
    int err,at24c02_dev=MKDEV(at24c02_major,index);
    cdev_init(&dev->cdev,&at24c02_fops);
    dev_cdev.owner=THIS_MODULE;
    err=cdev_add(&dev->cdev,at24c02_dev,1);
    if(err)
        printk(KERN_NOTICE "error %d adding at24c02 %d/n",err,index);
}
```

i2c设备驱动卸载函数进行i2c_del_driver调用后，顺序调用内核中注册的适配器断开注册过的i2c设备，此过程是调用at24c02_detach_client实现的。

```
static void __exit at24c02_exit(void)
{
    i2c_del_driver(&at24c02_driver);
}
static int at24c02_detach_client(struct i2c_client *client)
{
    int err;
    struct at24c02_data *data;

    data=i2c_get_client(client);
    cdev_del(&(data->cdev));
    unregister_chrdev_region(MKDEV(at24c02_major, 0), 1);
    err = i2c_detach_client(client);
    if (err)
        return err;
    kfree(data);

    return 0;
}
```

字符驱动的实现

```
struct file_operations at24c02_fops = {
        .owner = THIS_MODULE,
        .read= at24c02 _read,
        .write= at24c02 _write,
        .open= at24c02 _open,
        .release = at24c02_release,
    };
```

字符设备驱动不在详述，主要说明如何调用适配器完成数据传输。首先构造消息，通过i2c_transfer来传递，i2c_transfer找到对应适配器algorithm通信方法master_xfer最终完成i2c消息处理。下面是一个关于设备驱动的读函数，其他类似。

```
static int at24c02_open(struct inode *inode, struct file *file)
{
    struct at24c02_data *data;
    data=container_of(inode->i_cdev,struct at24c02_data,cdev);
    file->private_data = data;
    return 0;
}
static int at24c02_read(struct file *filp, char __user *buff,size_t count, loff_t *offp)
{
    int ret;
    struct i2c_msg msg[2];
    char addr = 0;
```

```
    struct at24c02_data *data=file->private_data;

    if (count > 1024)
    {
        return -EINVAL;
    }

    msg[0].addr  = data->client->addr;
    msg[0].flags = 0;       /* write */
    msg[0].len   = 1;       /* 1个地址 */
    msg[0].buf   = &addr;
    msg[1].addr  = data->client->addr;
    msg[1].flags = I2C_M_RD;       /* read */
    msg[1].len   = count;          /* 要读的数据个数 */
    msg[1].buf   = data->data;

    ret = i2c_transfer(at24c02_client->adapter, msg, 2);
    if (ret == 2)
    {
        copy_to_user(buff, data->data, count);
        return count;
    }
    else
        return -EIO;
}
```

目前适配器主要支持的传输方法有两种：master_xfer和smbus_xfer，从i2c_algorithm结构体可看出。一般来说，若适配器支持master_xfer那么它可以模拟支持smbus，但只实现smbus_xfer，则不支持master_xfer传输。当然，上面的驱动也可以采用smbus方式完成传输。采用LEGACY方式在2.6.32.2内核下编译不过去，主要是i2c核心中有些函数不存在，所以，设备驱动发展方向是new style方式。

2.new style方式

构建i2c_driver

```
static struct i2c_driver at24c02_driver = {
    .driver = {
        .name = "at24c02",
        .owner = THIS_MODULE,
    },
    .probe = at24c02_probe,
    .remove = __devexit_p(at24c02_remove),
    .id_table = at24c02_id,
};
```

加载与注销

```
static int __init at24c02_init(void)
{
    return i2c_add_driver(&at24c02_driver);
}
module_init(at24c02_init);
```

i2c_add_driver会将驱动注册到总线上，探测到i2c设备就会调用at24c02_probe，探测主要是用i2c_match_id函数比较client的名字和id_table中名字，如果相等，则探测到i2c设备，本驱动中id_table如下：

```
static const struct i2c_device_id at24c02_id[] = {
    { "at24c02", 0 },
    { }
};
MODULE_DEVICE_TABLE(i2c, at24c02_id);
```

MODULE_DEVICE_TABLE宏是用来生成i2c_device_id。在legacy方式中i2c_client是自己创建的，而此处的i2c_client如何得到？实际上是在i2c_register_board_info函数注册i2c_board_info过程中构建的，所以arch/arm/mach-s3c2440/mach-smdk2440.c中添加注册信息。

```
static struct i2c_board_info i2c_devices[] __initdata = {
    { I2C_BOARD_INFO("at24c02", 0x50), },
};
static void __init smdk2440_machine_init(void)
{
    i2c_register_board_info(0,i2c_devices,ARRAY_SIZE(i2c_devices));
    s3c24xx_fb_set_platdata(&smdk2440_fb_info);
    s3c_i2c0_set_platdata(NULL);

    platform_add_devices(smdk2440_devices, ARRAY_SIZE(smdk2440_devices));
    smdk_machine_init();
}
```

如果没有注册i2c信息，就探测不到i2c设备。探测到at24c02设备后就会调用at24c02_probe函数。

```
static int __devinit at24c08b_probe(struct i2c_client *client,const struct i2c_device_id *id)
{
    struct at24c02_data *data;
    int err = 0,result;
    dev_t at24c02_dev=MKDEV(at24c02_major,0);

    if (!i2c_check_functionality(adapter, I2C_FUNC_SMBUS_READ_BYTE_DATA
                                 | I2C_FUNC_SMBUS_BYTE)) //判断适配器功能
        goto exit;

    if (!(data = kzalloc(sizeof(struct at24c02_data), GFP_KERNEL))) {
        err = -ENOMEM;
        goto exit;
    }

    memset(data, 0, sizeof(struct at24c02_data));
    data->client=client;
```

```
    i2c_set_clientdata(client, data);

    if(at24c02_major)
    {
        result=register_chrdev_region(at24c02_dev,1,"at24c02");
    }
    else
    {
        result=alloc_chrdev_region(&at24c02_dev,0,1,"at24c02");
        at24c02_major=MAJOR(at24c02_dev);
    }
    if(result<0)
    {
        printk(KERN_NOTICE "Unable to get region %d/n",result);
        err=result;
        goto exit_kfree;
    }
    at24c02_setup_cdev(data,0);    //注册字符设备at24c02_fops
    return 0;

exit_kfree:
    kfree(data);
exit:
    return err;
}
```

probe函数主要注册了字符设备，通过data->client=client获得的相关信息。关于at24c02_fops结构体的完善，由于与前面一种方法类似，这里就不详述。最后就是驱动的注销。

```
static void __exit at24c02_exit(void)
{
    i2c_del_driver(&at24c02_driver);
}
module_exit(at24c02_exit);
static int __devexit at24c02_remove(struct i2c_client *client)
{
    int err;
    struct at24c02_data *data;
    data=i2c_get_client(client);
    cdev_del(&(data->cdev));
    unregister_chrdev_region(MKDEV(at24c02_major, 0), 1);
    kfree(data);
    return 0;
}
```


上一篇 Linux内核I2C子系统驱动(二)

我的同类文章

Linux设备驱动开发(9)

Linux内核I2C子系统驱动(二)	2011-08-02 阅读 2574	Linux内核I2C子系统驱动(一)	2011-08-02 阅读 2531
S3C2440驱动篇—RTC驱动分析	2011-08-01 阅读 957	S3C2440驱动篇—看门狗驱动分析	2011-07-24 阅读 916
S3C2440驱动篇—Linux平台设备驱动	2011-07-21 阅读 2063	内核访问外设I/O资源方式	2011-07-21 阅读 950
S3C2440驱动篇—触摸屏驱动分析	2011-07-19 阅读 1976	S3C2440驱动篇—ADC驱动分析	2011-07-19 阅读 1381
国嵌memdev驱动试验	2010-12-31 阅读 2812		

猜你在找

- 从零写Bootloader及移植uboot、linux内核、文件系统和驱动
- 话说linux内核-uboot和系统移植第14部分
- linux嵌入式开发+驱动开发
- “攒课”课题3：安卓编译与开发、Linux内核及驱动
- 嵌入式Linux项目实战：三个大项目(数码相框、摄像头驱动和电源管理),
- Linux内核I2C子系统驱动三
- Linux内核I2C子系统驱动
- linux内核I2C驱动子系统分析一
- Linux内核I2C子系统驱动一
- Linux内核I2C子系统驱动一

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场

核心技术类目

全部主题 Hadoop AWS 移动游戏 Java Android iOS Swift 智能硬件 Docker OpenStack VPN Spark ERP IE10
Eclipse CRM JavaScript 数据库 Ubuntu NFC WAP jQuery BI HTML5 Spring Apache .NET API HTML SDK IIS
Fedora XML LBS Unity Splashtop UML components Windows Mobile Rails QEMU KDE Cassandra CloudStack FTC

coremail OPhone CouchBase 云计算 iOS6 Rackspace Web App SpringSide Maemo Compuware 大数据 aptech Perl
Tornado Ruby Hibernate ThinkPHP HBase Pure Solr Angular Cloud Foundry Redis Scala Django Bootstrap