网名“鱼树”的学员聂龙浩，

学习“韦东山 Linux 视频第 2 期”时所写的笔记很详细，供大家参考。

也许有错漏，请自行分辨。

目录

I2C 驱动程序框架：

内核中 I2C 的处理已经做好了，我们只需要做设备驱动程序相关的内容。
总线处理好了 I2C 协议，即总线知道如何收发数据，而不知道数据的含义，我们要做的只是设备相关层的代码。
I2C 协议中，先发出 7bit“设备地址”，然后是 1 位“写”或“读”的标志位。然后接着是每发出 8 位数据有一个 ACK 位。

一般 I2C 驱动分为两层：

总线层：

知道设备如何读写。芯片厂家会帮我们做好。操作寄存器。`drivers\i2c\busses`

设备层驱动层：

知道数据的含义。`drivers\i2c\chips`

一，分析：\drivers\i2c\busses\I2c-s3c2410.c

1，找到 probe 函数：

```
int __init i2c_adap_s3c_init(void)
platform_driver_register(&s3c2410_i2c_driver);
static struct platform_driver s3c2440_i2c_driver = {
    .probe      = s3c24xx_i2c_probe,
    .remove     = s3c24xx_i2c_remove,
    .resume     = s3c24xx_i2c_resume,
    .driver     = {
        .owner  = THIS_MODULE,
        .name   = "s3c2440-i2c",
    },
};
```

注册一个平台设备，当内核中有同名“s3c2440-i2c”的平台设备时，“.probe = s3c24xx_i2c_probe,”就被调用。

2，分析 probe 函数：

```
int s3c24xx_i2c_probe(struct platform_device *pdev)
-->i2c->clk = clk_get(&pdev->dev, "i2c"); 使能I2C时钟。
-->I2C适配器结构“i2c_adapter”：
i2c->adap.algo_data = i2c;
i2c->adap.dev.parent = &pdev->dev;
-->ret = s3c24xx_i2c_init(i2c); 硬件相关初始化。
-->request_irq(res->start, s3c24xx_i2c_irq, IRQF_DISABLED,  pdev->name, i2c); 注册中断
-->i2c_add_adapter(&i2c->adap); 注册I2C适配器
   -->i2c_register_adapter(adapter);
```

“总线设备驱动”模型：

一，I2C 总线驱动程序：

插槽 （分析：linux-2.6.22.6\drivers\i2c\busses\i2c-s3c2410.c）

1，分配结构：i2c_adapter：

```
int s3c24xx_i2c_probe(struct platform_device *pdev)
-->struct s3c24xx_i2c *i2c = &s3c24xx_i2c;
```

```
struct i2c_adapter {
    struct module *owner;
    unsigned int id;
    unsigned int class;
    const struct i2c_algorithm *algo; /* the algorithm to access the bus */
    void *algo_data;
```

i2c_adapter结构中有算法结构

2，设置结构 i2c_adapter：

核心是设置“i2c_algorithm 算法结构”。

a,如何收发起始信号、数据、响应等.

b,i2c_adapter 结构中有 i2c_algorithm 算法结构。

```
struct s3c24xx_i2c *i2c = &s3c24xx_i2c;
-->.algo = &s3c24xx_i2c_algorithm,  其中的核心就是算法
```

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

static struct s3c24xx_i2c s3c24xx_i2c = {
    .lock       = __SPIN_LOCK_UNLOCKED(s3c24xx_i2c.lock),
    .wait       = __WAIT_QUEUE_HEAD_INITIALIZER(s3c24xx_i2c.wait),
    .tx_setup   = 50,
    .adap       = {
        .name           = "s3c2410-i2c",
        .owner          = THIS_MODULE,
        .algo           = &s3c24xx_i2c_algorithm,
        .retries        = 2,
        .class          = I2C_CLASS_HWMON,
    },
};
```

算法

①，算法结构中“.master_xfer”是核心。

```
int s3c24xx_i2c_xfer(struct i2c_adapter *adap,struct i2c_msg *msgs, int num)
-->s3c24xx_i2c_doxfer(i2c, msgs, num); “doxfer”执行传输。
```

②，执行传输：

```
int s3c24xx_i2c_doxfer(struct s3c24xx_i2c *i2c, struct i2c_msg *msgs, int num)
-->s3c24xx_i2c_enable_irq(i2c);  使能中断
-->s3c24xx_i2c_message_start(i2c, msgs); 起动传输，会产生各种中断。
-->wait_event_timeout(i2c->wait, i2c->msg_num == 0, HZ * 5); 等待事件完成。
```

③，起动传输：设置寄存器

寄存器：

```
S3C2410_IICCON  ：I2C控制寄存器
S3C2410_IICSTAT ：I2C状态寄存器
S3C2410_IICADD  ：
S3C2410_IICDS   ：I2C DS寄存器
S3C2440_IICLC   ：
void s3c24xx_i2c_message_start(struct s3c24xx_i2c *i2c, struct i2c_msg *msg)
-->iiccon = readl(i2c->regs + S3C2410_IICCON);
-->writel(stat, i2c->regs + S3C2410_IICSTAT);
-->writeb(addr, i2c->regs + S3C2410_IICDS);
```

3，注册“i2c_adapter”：i2c_add_adapter()

```
int s3c24xx_i2c_probe(struct platform_device *pdev)
-->i2c_add_adapter(&i2c->adap);
    -->i2c_register_adapter(adapter);

int i2c_register_adapter(struct i2c_adapter *adap)
-->device_register(&adap->dev);
```


二，（分析：linux-2.6.22.6\drivers\i2c\chips\eeprom.c）

```
i2c_add_driver
i2c_register_driver
driver->driver.bus = &i2c_bus_type;
driver_register(&driver->driver);


list_for_each_entry(adapter, &adapters, list) {
driver->attach_adapter(adapter);
i2c_probe(adapter, &addr_data, eeprom_detect);
i2c_probe_address // 发出S信号,发出设备地址(来自addr_data)
i2c_smbus_xfer
i2c_smbus_xfer_emulated
i2c_transfer
adap->algo->master_xfer // s3c24xx_i2c_xfer
int __init eeprom_init(void)
-->i2c_add_driver(&eeprom_driver);
   -->i2c_register_driver(THIS_MODULE, driver);
```

```
static struct i2c_driver eeprom_driver = {
    .driver = {
        .name   = "eeprom",
    },
    .id      = I2C_DRIVERID_EEPROM,
    .attach_adapter = eeprom_attach_adapter,
    .detach_client  = eeprom_detach_client,
};
```

裸板程序中发现设备：

先发一个“STATT”信号，再发出 7bit 的设备地址，若设备存在，则在第 9 个时钟里，此存在的设备（从机）会把“SDA”信号线拉低作为 ACK 回应，这样主机就知道有相对应地址的设备存在。

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
-->driver->driver.bus = &i2c_bus_type;  总线是“i2c_bus_type”。
-->driver_register(&driver->driver);  注册一个“i2c_driver”结构体。
-->list_for_each_entry(adapter, &adapters, list) {driver->attach_adapter(adapter);}
```

对"adapters"链表里的每一个成员，调用"i2c_driver"结构里面的 attach_adapter()。

```
struct i2c_driver {
    int id;
    unsigned int class;

    int (*attach_adapter)(struct i2c_adapter *);
    int (*detach_adapter)(struct i2c_adapter *);

    int (*detach_client)(struct i2c_client *);

    int (*probe)(struct i2c_client *);
    int (*remove)(struct i2c_client *);

    void (*shutdown)(struct i2c_client *);
    int (*suspend)(struct i2c_client *, pm_message_t mesg);
    int (*resume)(struct i2c_client *);

    int (*command)(struct i2c_client *client,unsigned int cmd, void *arg);

    struct device_driver driver;
    struct list_head list;
} ? end i2c_driver ? ;
```

此实例代码中的"i2c_driver"结构体是“eeprom_driver”，它其中的“attach_adapter()”如下：

```
static struct i2c_driver eeprom_driver = {
    .driver = {
        .name   = "eeprom",
    },
    .id      = I2C_DRIVERID_EEPROM,
    .attach_adapter = eeprom_attach_adapter,
    .detach_client  = eeprom_detach_client,
};
```

```
int eeprom_attach_adapter(struct i2c_adapter *adapter)
-->i2c_probe(adapter, &addr_data, eeprom_detect);  其中参2就是设备地址信息。
  -->i2c_probe_address(adapter,forces[kind][i + 1],kind, found_proc);  发出start信号，发出设备地址。
    -->i2c_smbus_xfer(adapter, addr, 0, 0, 0, I2C_SMBUS_QUICK, NULL); i2c传输
      -->adapter->algo->smbus_xfer(adapter,addr,flags,read_write,command,size,data);
```

调用“i2c_adapter” 结构里面的成员“i2c_algorithm”结构中的“算法函数”。要是没有这个“smbus_xfer（）”，就使用：此代码中是用下面的代码。

-->i2c_smbus_xfer_emulated(adapter,addr,flags,read_write,command,size,data);

-->i2c_transfer(adapter, msg, num);

-->adap->algo->master_xfer(adap,msgs,num);

调用“i2c_adapter” 结构里面的成员“i2c_algorithm”结构中的“算法函数-smbus_xfer()”。

对于“i2c_s3c2410.c” 中，这个“smbus_xfer()”函数即是：

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

USB 总线会自动识别新接入的 USB 设备。但 I2C 总线不能。需要：

1，发出 START 信号

2，发出设备地址。

才能知道是否有此设备存在。

从“i2c_adapter”结构的“adapter”链表取出“i2c 总线”中的一个一个驱动程序（称为“适配器”），使用里面的“.smbus_xfer”函数发“start”信号、发设备地址（在 I2C 设备驱动的 i2c_driver 结构中，成员“.id”就是表示支持哪些 I2C 设备）、

怎么写 I2C 设备驱动程序？

1. 分配一个 i2c_driver 结构体。
2. 设置：
 i2c_driver 结构体中有一个“attach_adapter”和“detach_client”函数。
 attach_adapter // 它直接调用 i2c_probe(adap, 设备地址, 发现这个设备后要调用的函数);
 detach_client（一个设备） // 卸载这个驱动后,如果之前发现能够支持的设备,则调用它来清理

3. 注册：i2c_add_driver

代码：

1，分配 i2c_driver 结构：

```
/*1,分配一个 i2c_driver 结构体*/
/*2，设置 i2c_driver 结构体 */
static struct i2c_driver at24cxx_driver = { /*直接定义*/
.driver = {
.name = "at24cxx",
},
/*.id = , 可能没有用 */
.attach_adapter = at24cxx_attach, /*添加*/
.detach_client  = at24cxx_detach, /*卸载*/
};
```

2，有了结构后，就在入口函数中注册此结构：

```
i2c_del_driver(at24cxx_driver);
```

3，开始写“i2c_driver at24cxx_driver”中的成员函数：

```
.attach_adapter = at24cxx_attach, /*添加*/
/*4，写".attach_adapter"函数*/
static int at24cxx_attach(struct i2c_adapter *adater)
{
return i2c_probe(adapter, &addr_data, at24cxx_detect);
}
```

4，实现“i2c_probe（）”函数：

```
/*5,定义i2c_probe中的&addr_data*/
static unsigned short ignore[] = { I2C_CLIENT_END };
static unsigned short normal_addr[] = { 0x50, I2C_CLIENT_END}; /* 此时设备地址是1010000即0x50 */


static struct i2c_client_address_data addr_data = {/* 结构中包含三项: */
.normal_i2c = normal_addr, /* 要发出地址信号才能确定是否存在此设备 */
```

```
.probe = ignore, /* ignore 是省略的意思.具体作用得跟踪i2c_probe函数 */
.ignore = ignore,
/* .forces 即forces等于某个地址，就强制认为此设备存在。 */
};
```

①，跟踪 i2c_probe()看看 "ignore"的意思：

```
int i2c_probe(struct i2c_adapter *adapter,struct i2c_client_address_data *address_data,
        int (*found_proc) (struct i2c_adapter *, int, int))
-->if (address_data->forces) { }
Forces 是强制认为此设备存在。当 forces等于某地址时，就强制认为此设备存在。
```

②，实现“i2c_probe()”中通过地址找到设备后的处理函数：

```
i2c_probe(adapter, &addr_data, at24cxx_detect);其中的“at24cxx_detect（）”。
/*6，写i2c_probe中的处理函数:int (*found_proc) (struct i2c_adapter *, int, int)*/
static int at24cxx_detect (struct i2c_adapter *adapter, int address, int kind)
{
    printk("at24cxx_detect\n");
    return 0;
}
```

5，开始写“i2c_driver at24cxx_driver”中的成员函数：

```
.detach_client   = at24cxx_detach, /*卸载*/
```

开始第一次编译测试：

```
root@ian:/work/nfs_root/romfs/19th_i2c_drv# make
make -C /work/system/linux-2.6.22.6 M=`pwd` modules
make[1]: Entering directory `/work/system/linux-2.6.22.6'
  CC [M]  /work/nfs_root/romfs/19th_i2c_drv/at24cxx.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /work/nfs_root/romfs/19th_i2c_drv/at24cxx.mod.o
  LD [M]  /work/nfs_root/romfs/19th_i2c_drv/at24cxx.ko
make[1]: Leaving directory `/work/system/linux-2.6.22.6'
root@ian:/work/nfs_root/romfs/19th_i2c_drv#
```

```
# insmod at24cxx.ko
i2c-adapter i2c-0: Invalid probe address 0xa0
#
```

上面这个错误是 I2C 设备地址错误。

```
# insmod at24cxx.ko
at24cxx_detect
#
```

修改 I2C 设备地址后，驱动检测出来了。（I2C 设备地址要是开发板和元理图上真实的地址）。

修改 normal_addr 里的 0x50 为 0x60，这时上面的“at24cxx_detect”并没有打印出来。

```
/*5.定义i2c_probe中的&addr_data*/
static unsigned short ignore[] = { I2C_CLIENT_END };
static unsigned short normal_addr[] = { 0x50, I2C_CLIENT_END}; /* 此时设备地址是1010000即0x50 */
                                       /* 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 struct i2c_client_address_data addr_data = {/* 结构中包含三项： */
    .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 是省略的意思，具体作用得跟踪i2c_probe函数 */
    .ignore = ignore,
    /* .forces 即forces等于某个地址，就强制认为此设备存在。 */
};
```

另一种情况就是，当前 I2C 设备还不能使用，但在程序运行过程中可能以后才能用，好想让它调用“at24cxx_detect”函数（即 i2c_probe()函数的参 3）。这时就在“struct i2c_client_address_data addr_data”地址结构中用“.forces”（ .forces 即 forces 等于某个地址，就强制认为此设备存在。）

二，实现强制使用 I2C 设备地址来驱动设备：

当在“入口函数”中注册了 I2C 总线设备驱动的结构体“i2c_driver”时，结构中有两个处理函数，其一为“attach_adapter”，在 I2C 设备注册成功后，会调用这个函数来处理此设备，此函数最终会调用“i2c_probe()”函数，其中要求了“I2C 设备”的地址。若是强制指定了这个设备地址，通过此地址设备的回应 ACK 后就可以调用“.attach_adapter = at24cxx_attach”函数。则实现如下：

强制使用“.forces”。在 i2c_probe()中的实现过程如下：

```
/* Force entries are done first, and are not affected by ignore
   entries */
if (address_data->forces) {
    unsigned short **forces = address_data->forces;
    int kind;

    for (kind = 0; forces[kind]; kind++) {
        for (i = 0; forces[kind][i] != I2C_CLIENT_END;
             i += 2) {
            if (forces[kind][i] == adap_id
             || forces[kind][i] == ANY_I2C_BUS) {
                dev_dbg(&adapter->dev, "found force "
                    "parameter for adapter %d, "
                    "addr 0x%02x, kind %d\n",
                    adap_id, forces[kind][i + 1],
                    kind);
                err = i2c_probe_address(adapter,
                    forces[kind][i + 1],
                    kind, found_proc);
                if (err)
                    return err;
            }
        }
    }
} ? end if address_data->forces ?
```

```
int i2c_probe(struct i2c_adapter *adapter, struct i2c_client_address_data *address_data,
        int (*found_proc) (struct i2c_adapter *, int, int))
-->if (address_data->forces)
  -->unsigned short **forces = address_data->forces;
“unsigned short **forces”指向数组的指针。
  -->int i2c_probe_address(struct i2c_adapter *adapter, int addr, int kind,
      int (*found_proc) (struct i2c_adapter *, int, int))
      -->if (kind < 0) 若kind小于0时才调用“i2c_smbus_xfer”。
        -->i2c_smbus_xfer(adapter, addr, 0, 0, 0,I2C_SMBUS_QUICK, NULL)
 -->err = found_proc(adapter, addr, kind); 若 kind不小于0则直接找到设备。
```

写代码：

```
static unsigned short force_addr[] = {ANY_I2C_BUS, 0x60, I2C_CLIENT_END};
数组第一个值“ANY_I2C_BUS”（任何的I2C总线）
static unsigned short * forces[] = {force_addr, NULL};
```

最后编译出错：

```
Backtrace:
[<c020bb84>] (i2c_probe+0x0/0x240) from [<bf000038>] (at24cxx_attach+0x18/0x24 [at24cxx])
[<bf000020>] (at24cxx_attach+0x0/0x24 [at24cxx]) from [<c020a304>] (i2c_register_driver+0xcc/0xf8)
[<c020a238>] (i2c_register_driver+0x0/0xf8) from [<bf00007c>] (at24cxx_init+0x18/0x28 [at24cxx])
 r6:bf000680 r5:bf000680 r4:00000000
```

在“i2c_probe”中出错。