Linux下I2C驱动架构全面分析

Y沉浮 嵌入式之旅 2021-07-02 00:00

I2C 概述

I2C是philips提出的外设总线。

I2C只有两条线,一条串行数据线:SDA,一条是时钟线SCL ，使用SCL，SDA这两根信号线就实现了设备之间的数据交互，它方便了工程师的布线。

因此，I2C总线被非常广泛地应用在EEPROM，实时钟，小型LCD等设备与CPU的接口中。

Linux下的驱动思路

在linux系统下编写I2C驱动，目前主要 有两种方法， 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下I2C驱动体系结构来完成。

第一种方法思路比较直接，不需要花很多时间去了解linux中复杂的I2C子系统的操作方法。缺点是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(I2C控制器)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以移植性差。对内核的资源无法直接使用，因为内核提供的所有I2C设备器以及设备驱动都是基于I2C子系统的格式。第一种方法的优点就是第二种方法的缺点，第一种方法的缺点就是第二种方法的优点。

I2C架构概述

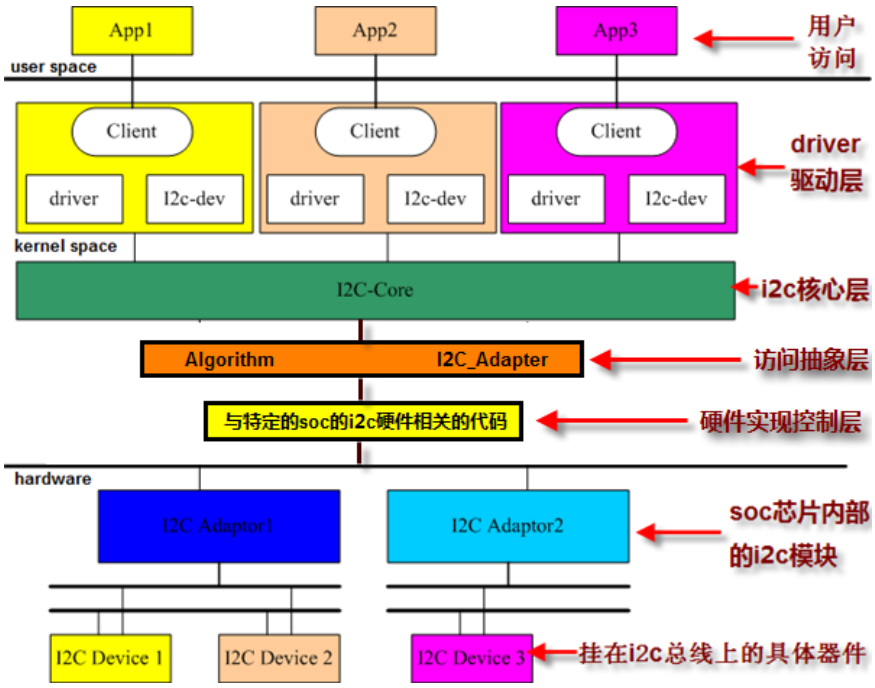
Linux的I2C体系结构分为3个组成部分：

I2C核心：I2C核心提供了I2C总线驱动和设备驱动的注册，注销方法，I2C通信方法(”algorithm”)上层的，与具体适配器无关的代码以及探测设备，检测设备地址的上层代码等。

I2C总线驱动：I2C总线驱动是对I2C硬件体系结构中适配器端的实现，适配器可由CPU控制，甚至可以直接集成在CPU内部。

I2C设备驱动：I2C设备驱动(也称为客户驱动)是对I2C硬件体系结构中设备端的实现，设备一般挂接在受CPU控制的I2C适配器上，通过I2C适配器与CPU交换数据。

Linux驱动中i2c驱动架构



上图完整的描述了linux i2c驱动架构，虽然I2C硬件体系结构比较简单，但是i2c体系结构在linux中的实现却相当复杂。

那么我们如何编写特定i2c接口器件的驱动程序？就是说上述架构中的那些部分需要我们完成，而哪些是linux内核已经完善的或者是芯片提供商已经提供的？

架构层次分类

第一层：提供i2c adapter的硬件驱动，探测、初始化i2c adapter（如申请i2c的io地址和中断号），驱动soc控制的i2c adapter在硬件上产生信号（start、stop、ack）以及处理i2c中断。覆盖图中的硬件实现层

第二层：提供i2c adapter的algorithm，用具体适配器的xxx_xferf()函数来填充i2c_algorithm的master_xfer函数指针，并把赋值后的i2c_algorithm再赋值给i2c_adapter的algo指针。覆盖图中的访问抽象层、i2c核心层

第三层：实现i2c设备驱动中的i2c_driver接口，用具体的i2c device设备的attach_adapter()、detach_adapter()方法赋值给i2c_driver的成员函数指针。实现设备device与总线（或者叫adapter）的挂接。覆盖图中的driver驱动层

第四层：实现i2c设备所对应的具体device的驱动，i2c_driver只是实现设备与总线的挂接，而挂接在总线上的设备则是千差万别的，所以要实现具体设备device的write()、read()、ioctl()等方法，赋值给file_operations，然后注册字符设备（多数是字符设备）。覆盖图中的driver驱动层。

第一层和第二层又叫i2c总线驱动(bus)，第三第四属于i2c设备驱动(device driver)。

在linux驱动架构中，几乎不需要驱动开发人员再添加bus，因为linux内核几乎集成所有总线bus，如usb、pci、i2c等等。并且总线bus中的(与特定硬件相关的代码)已由芯片提供商编写完成，例如三星的s3c-2440平台i2c总线bus为/drivers/i2c/buses/i2c-s3c2410.c

第三第四层与特定device相干的就需要驱动工程师来实现了。

Linux下I2C体系文件构架

在Linux内核源代码中的driver目录下包含一个i2c目录。i2c-core.c这个文件实现了I2C核心的功能以及/proc/bus/i2c*接口。

i2c-dev.c实现了I2C适配器设备文件的功能，每一个I2C适配器都被分配一个设备。通过适配器访设备时的主设备号都为89，次设备号为0-255。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。

busses文件夹这个文件中包含了一些I2C总线的驱动，如针对S3C2410，S3C2440，S3C6410等处理器的I2C控制器驱动为i2c-s3c2410.c。algos文件夹实现了一些I2C总线适配器的algorithm。

重要的结构体

i2c_driver

```
struct i2c_driver {
unsigned int class;
int (*attach_adapter)(struct i2c_adapter *);//依附i2c_adapter函数指针
int (*detach_adapter)(struct i2c_adapter *);//脱离i2c_adapter函数指针
int (*probe)(struct i2c_client *, const struct i2c_device_id *);
int (*remove)(struct i2c_client *);
void (*shutdown)(struct i2c_client *);
int (*suspend)(struct i2c_client *, pm_message_t mesg);
int (*resume)(struct i2c_client *);
void (*alert)(struct i2c_client *, unsigned int data);
int (*command)(struct i2c_client *client, unsigned int cmd, void*arg);//命令
struct device_driver driver;
const struct i2c_device_id *id_table;//该驱动所支持的设备ID表
int (*detect)(struct i2c_client *, struct i2c_board_info *);
```

```
const unsigned short *address_list;
struct list_head clients;
};
```

i2c_client

```
struct i2c_client {
 unsigned short flags; //标志
 unsigned short addr; //低7位为芯片地址
 char name[I2C_NAME_SIZE];//设备名称
 struct i2c_adapter *adapter;//依附的i2c_adapter
 struct i2c_driver *driver;//依附的i2c_driver
 struct device dev;//设备结构体
 int irq;//设备所使用的结构体
 struct list_head detected;//链表头
 };
```

i2c_adapter

```
struct i2c_adapter {
 struct module *owner;//所属模块
 unsigned int id;//algorithm的类型，定义于i2c-id.h,
 unsigned int class;
 const struct i2c_algorithm *algo; //总线通信方法结构体指针
 void *algo_data;//algorithm数据
 struct rt_mutex bus_lock;//控制并发访问的自旋锁
 int timeout;
 int retries;//重试次数
 struct device dev; //适配器设备
 int nr;
 char name[48];//适配器名称
 struct completion dev_released;//用于同步
 struct list_head userspace_clients;//client链表头
};
```

i2c_algorithm

```
struct i2c_algorithm {
    int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs, int nu
    int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr,unsigned short flag
    i2c_smbus_data *data);//smbus传输函数指针
    u32 (*functionality) (struct i2c_adapter *);//返回适配器支持的功能
};
```

各结构体的作用与它们之间的关系

i2c_adapter与i2c_algorithm

i2c_adapter对应与物理上的一个适配器，而i2c_algorithm对应一套通信方法，一个i2c适配器需要i2c_algorithm中提供的（i2c_algorithm中的又是更下层与硬件相关的代码提供）通信函数来控制适配器上产生特定的访问周期。缺少i2c_algorithm的i2c_adapter什么也做不了，因此i2c_adapter中包含其使用i2c_algorithm的指针。

i2c_algorithm中的关键函数master_xfer()用于产生i2c访问周期需要的start stop ack信号，以i2c_msg（即i2c消息）为单位发送和接收通信数据。

i2c_msg也非常关键，调用驱动中的发送接收函数需要填充该结构体

```
struct i2c_msg {
    __u16 addr; /* slave address            */
    __u16 flags;
    __u16 len;      /* msg length                */
    __u8 *buf;      /* pointer to msg data          */
};
```

i2c_driver和i2c_client

i2c_driver对应一套驱动方法，其主要函数是attach_adapter()和detach_client()。i2c_client对应真实的i2c物理设备device，每个i2c设备都需要一个i2c_client来描述i2c_driver与i2c_client的关系是一对多。一个i2c_driver上可以支持多个同等类型的i2c_client。

i2c_adapter和i2c_client

i2c_adapter和i2c_client的关系与i2c硬件体系中适配器和设备的关系一致，即i2c_client依附于i2c_adapter,由于一个适配器上可以连接多个i2c设备，所以i2c_adapter中包含依附于它的i2c_client的链表。

从i2c驱动架构图中可以看出，linux内核对i2c架构抽象了一个叫核心层core的中间件，它分离了设备驱动device driver和硬件控制的实现细节（如操作i2c的寄存器），core层不但为上面的设备驱动提供封装后的内核注册函数，而且还为小面的硬件事件提供注册接口（也就是i2c总线注册接口），可以说core层起到了承上启下的作用。

具体分析

先看一下i2c-core为外部提供的核心函数（选取部分），i2c-core对应的源文件为i2c-core.c，位于内核目录/driver/i2c/i2c-core.c

```
EXPORT_SYMBOL(i2c_add_adapter);
EXPORT_SYMBOL(i2c_del_adapter);
EXPORT_SYMBOL(i2c_del_driver);
EXPORT_SYMBOL(i2c_attach_client);
EXPORT_SYMBOL(i2c_detach_client);

EXPORT_SYMBOL(i2c_transfer);
```

i2c_transfer()函数：i2c_transfer()函数本身并不具备驱动适配器物理硬件完成消息交互的能力，它只是寻找到i2c_adapter对应的i2c_algorithm，并使用i2c_algorithm的master_xfer()函数真正的驱动硬件流程，代码清单如下，不重要的已删除。

```
int i2c_transfer(struct i2c_adapter * adap, struct i2c_msg *msgs, int num)
{
    int ret;
    if (adap->algo->master_xfer) {//如果master_xfer函数存在，则调用，否则返回错
        ret = adap->algo->master_xfer(adap,msgs,num);//这个函数在硬件相关的代码
        return ret;
    } else {
        return -ENOSYS;
    }
}
```

当一个具体的client被侦测到并被关联的时候，设备和sysfs文件将被注册。

相反的，在client被取消关联的时候，sysfs文件和设备也被注销，驱动开发人员在开发i2c设备驱动时，需要调用下列函数。程序清单如下：

```
int i2c_attach_client(struct i2c_client *client)
{
    ...
    device_register(&client->dev);
    device_create_file(&client->dev, &dev_attr_client_name);
    ...
    return 0;
}


[cpp] view plaincopy
int i2c_detach_client(struct i2c_client *client)
{
    ...
    device_remove_file(&client->dev, &dev_attr_client_name);
    device_unregister(&client->dev);
    ...
    return res;
}

i2c_add_adapter()函数和i2c_del_adapter()在i2c-davinci.c中有调用，稍后分析


int i2c_add_adapter(struct i2c_adapter *adap)
{
    ...
    device_register(&adap->dev);
    device_create_file(&adap->dev, &dev_attr_name);
    ...
    /* inform drivers of new adapters */
    list_for_each(item,&drivers) {
        driver = list_entry(item, struct i2c_driver, list);
        if (driver->attach_adapter)
            /* We ignore the return code; if it fails, too bad */
            driver->attach_adapter(adap);
    }
    ...
```

```
}



int i2c_del_adapter(struct i2c_adapter *adap)
{
    ...
    list_for_each(item,&drivers) {
        driver = list_entry(item, struct i2c_driver, list);
        if (driver->detach_adapter)
            if ((res = driver->detach_adapter(adap))) {
            }
    }
    ...
    list_for_each_safe(item, _n, &adap->clients) {
        client = list_entry(item, struct i2c_client, list);

        if ((res=client->driver->detach_client(client))) {

        }
    }
    ...
    device_remove_file(&adap->dev, &dev_attr_name);
    device_unregister(&adap->dev);

}
```

i2c-davinci.c是实现与硬件相关功能的代码集合，这部分是与平台相关的，也叫做i2c总线驱动，这部分代码是这样添加到系统中的：

```
static struct platform_driver davinci_i2c_driver = {
    .probe      = davinci_i2c_probe,
    .remove     = davinci_i2c_remove,
    .driver     = {
        .name   = "i2c_davinci",
        .owner  = THIS_MODULE,
    },
};

```

```
/* I2C may be needed to bring up other drivers */
static int __init davinci_i2c_init_driver(void)
{
    return platform_driver_register(&davinci_i2c_driver);
}
subsys_initcall(davinci_i2c_init_driver);

static void __exit davinci_i2c_exit_driver(void)
{
    platform_driver_unregister(&davinci_i2c_driver);
}
module_exit(davinci_i2c_exit_driver);
```

并且，i2c适配器控制硬件发送接收数据的函数在这里赋值给i2c-algorithm，i2c_davinci_xfer稍加修改就可以在裸机中控制i2c适配器。

```
static struct i2c_algorithm i2c_davinci_algo = {
    .master_xfer    = i2c_davinci_xfer,
    .functionality  = i2c_davinci_func,
};
```

然后在davinci_i2c_probe函数中，将i2c_davinci_algo添加到添加到algorithm系统中。

```
adap->algo = &i2c_davinci_algo;
```

适配器驱动程序分析

在linux系统中，适配器驱动位于linux目录下的\drivers\i2c\busses下，不同的处理器的适配器驱动程序设计有差异，但是总体思路不变。

在适配器的驱动中，实现两个结构体非常关键，也是整个适配器驱动的灵魂。

下面以某个适配器的驱动程序为例进行说明：

```
static struct platform_driver tcc_i2c_driver = {
 .probe    = tcc_i2c_probe,
 .remove   = tcc_i2c_remove,
```

```
    .suspend  = tcc_i2c_suspend_late,
    .resume   = tcc_i2c_resume_early,
    .driver   = {
    .owner  = THIS_MODULE,
    .name   = "tcc-i2c",
    },
};
```

以上说明这个驱动是基于平台总线的，这样实现的目的是与CPU紧紧联系起来。

```
static const struct i2c_algorithm tcc_i2c_algorithm = {
    .master_xfer = tcc_i2c_xfer,
    .functionality = tcc_i2c_func,
};
```

这个结构体也是非常的关键，这个结构体里面的函数tcc_i2c_xfer是适配器算法的实现，这个函数实现了适配器与I2C CORE的连接。

tcc_i2c_func是指该适配器所支持的功能。tcc_i2c_xfer这个函数实质是实现I2C数据的发送与接收的处理过程。不同的处理器实现的方法不同，主要表现在寄存器的设置与中断的处理方法上。

把握上面的两点去分析适配器程序就简单多了。

I2C-core驱动程序分析

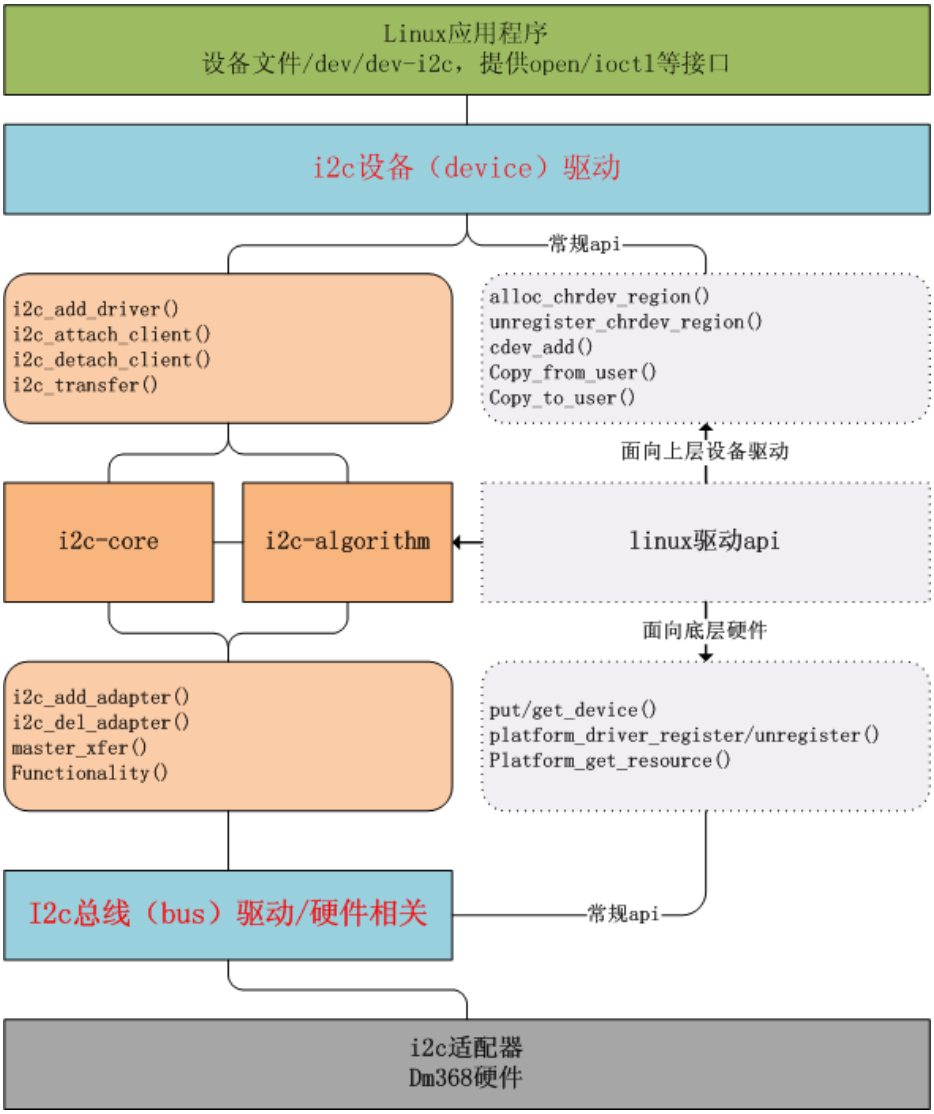
在I2C-core.c这个函数中，把握下面的几个关键函数就可以了。

```
//增加/删除i2c_adapter
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_del_adapter(struct i2c_adapter *adap)

//增加/删除i2c_driver
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
void i2c_del_driver(struct i2c_driver *driver)

//i2c_client依附/脱离
int i2c_attach_client(struct i2c_client *client)
```

```
11
12  //增加/删除i2c_driver
13  int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
14  void i2c_del_driver(struct i2c_driver *driver)
15
16  //i2c_client依附/脱离
17  int i2c_attach_client(struct i2c_client *client)
18  int i2c_detach_client(struct i2c_client *client)
19
20  //I2C传输,发送和接收
21  int i2c_master_send(struct i2c_client *client,const char *buf ,int count)
22  int i2c_master_recv(struct i2c_client *client, char *buf ,int count)
23  int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num)
24  I2c_transfer这个函数实现了core与adapter的联系。
```

代码调用层次图

有时候代码比任何文字描述都来得直接，但是过多的代码展示反而让人觉得枯燥。这个时候，需要一幅图来梳理一下上面的内容：



上面这些代码的展示是告诉我们：linux内核和芯片提供商为我们的的驱动程序提供了 i2c 驱动的框架，以及框架底层与硬件相关的代码的实现。

剩下的就是针对挂载在i2c两线上的i2c设备了device，而编写的即具体设备驱动了，这里的设备就是硬件接口外挂载的设备，而非硬件接口本身（soc硬件接口本身的驱动可以理解为总线驱动）。

编写驱动需要完成的工作

编写具体的I2C驱动时，工程师需要处理的主要工作如下：

1).提供I2C适配器的硬件驱动，探测，初始化I2C适配器(如申请I2C的I/O地址和中断号)，驱动CPU控制的I2C适配器从硬件上产生。
2).提供I2C控制的algorithm, 用具体适配器的xxx_xfer()函数填充i2c_algorithm的master_xfer指针，并把i2c_algorithm指针赋给i2c_adapter的algo指针。
3).实现I2C设备驱动中的i2c_driver接口，用具体yyy的yyy_probe()，yyy_remove()，yyy_suspend(),yyy_resume() 函数指针和i2c_device_id设备ID表赋给i2c_driver的

probe,remove,suspend,resume和id_table指针。

4).实现I2C设备所对应类型的具体驱动，i2c_driver只是实现设备与总线的挂接。

上面的工作中前两个属于I2C总线驱动，后面两个属于I2C设备驱动。

推荐阅读

（点击文字自动跳转）

资源分享 | C/C++ Primer Plus 电子书

VMware 的安装详解

喜欢此内容的人还喜欢

单片微机原理系列——只读存储器

嵌入式之旅