Linux I2C 驱动

作者：zhang-ke-wei
原文链接：https://ld246.com/article/1592110281046
来源网站：链滴

一、I2C 总线驱动

platform 是虚拟出来的一条总线，目的是为了实现总线、设备、驱动框架。而对于 I2C 而言，不需虚拟出一条总线，直接使用 I2C总线即可。 I2C 总线驱动重点是 I2C 适配器(也就是 SOC 的 I2C 接口制器)驱动，这里要用到两个重要的数据结构： i2c_adapter 和 i2c_algorithm， Linux 内核将 SOC 的 I2C 适配器(控制器)抽象成 i2c_adapter， i2c_adapter 结构体定义在 include/linux/i2c.h 文件中，构体内容如下：

```
struct i2c_adapter {
    struct module *owner;
    unsigned int class;     /* classes to allow probing for */
    const struct i2c_algorithm *algo;   /* 总线访问算法 */
    void *algo_data;

    /* data fields that are valid for all devices */
    struct rt_mutex bus_lock;

    int timeout;        /* in jiffies */
    int retries;
    struct device dev;      /* the adapter device */

    int nr;
    char name[48];
    struct completion dev_released;

    struct mutex userspace_clients_lock;
    struct list_head userspace_clients;

    struct i2c_bus_recovery_info *bus_recovery_info;
    const struct i2c_adapter_quirks *quirks;
};
```

i2c_algorithm 类型的指针变量 algo，对于一个 I2C 适配器，要对外提供读写 API 函数，设备驱动序可以使用这些 API 函数来完成读写操作。 i2c_algorithm 就是 I2C 适配器与 IIC 设备进行通信的方。

i2c_algorithm 结构体定义在 include/linux/i2c.h 文件中，内容如下(删除条件编译)：

```
struct i2c_algorithm {
    ......
    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);

    /* To determine what the adapter supports */
    u32 (*functionality) (struct i2c_adapter *);
    ......
};
```

master_xfer就是 I2C 适配器的传输函数，可以通过此函数来完成与 IIC 设备之间的通信。

smbus_xfer 就是 SMBUS 总线的传输函数。

综上所述， I2C 总线驱动，或者说 I2C 适配器驱动的主要工作就是初始化 i2c_adapter 结构体变量然后设置 i2c_algorithm 中的 master_xfer 函数。完成以后通过 i2c_add_numbered_adapter或 i2c_dd_adapter 这两个函数向系统注册设置好的 i2c_adapter，这两个函数的原型如下：

```
int i2c_add_adapter(struct i2c_adapter *adapter);
int i2c_add_numbered_adapter(struct i2c_adapter *adap);
```

这两个函数的区别在于 i2c_add_adapter 使用动态的总线号，而 i2c_add_numbered_adapter使用态总线号。函数参数和返回值含义如下：

adapter 或 adap：要添加到 Linux 内核中的 i2c_adapter，也就是 I2C 适配器。

返回值： 0，成功；负值，失败。

如果要删除 I2C 适配器的话使用 i2c_del_adapter 函数即可，函数原型如下：

```
void i2c_del_adapter(struct i2c_adapter * adap)
```

函数参数和返回值含义如下：

adap：要删除的 I2C 适配器。

返回值： 无。

一般 SOC 的 I2C 总线驱动都是由半导体厂商编写，不需要用户去编写。因此 I2C 总线驱动对我们这些 SOC 使用者来说是被屏蔽掉的，我们只要专注于 I2C 设备驱动即可。

二、I2C 设备驱动

I2C 设备驱动重点关注两个数据结构： i2c_client 和 i2c_driver，根据总线、设备和驱动模型，I2C 线上一小节已经讲了。还剩下设备和驱动， i2c_client 就是描述设备信息的， i2c_driver 描述驱动内，类似于 platform_driver。

1、i2c_client 结构体

i2c_client 结构体定义在 include/linux/i2c.h 文件中，内容如下：

```
struct i2c_client {
    unsigned short flags;  /* 标志 */
    unsigned short addr;  /* 芯片地址，  7 位，存在低 7 位*/
    ......
    char name[I2C_NAME_SIZE];  /* 名字 */
    struct i2c_adapter *adapter;  /* 对应的 I2C 适配器 */
    struct device dev;  /* 设备结构体 */
    int irq;  /* 中断 */
    struct list_head detected;
    ......
};
```

2、i2c_driver 结构体

一个设备对应一个 i2c_client，每检测到一个 I2C 设备就会给这个 I2C 设备分配一个i2c_client

i2c_driver 类似 platform_driver，是编写 I2C 设备驱动重点要处理的内容， i2c_driver 结构体定义在 nclude/linux/i2c.h 文件中，内容如下：

```
struct i2c_driver {
    unsigned int class;

    /* Notifies the driver that a new bus has appeared. You should
    * avoid using this, it will be removed in a near future.
    */
    int (*attach_adapter)(struct i2c_adapter *) __deprecated;

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

    /* driver model interfaces that don't relate to enumeration */
    void (*shutdown)(struct i2c_client *);

    /* Alert callback, for example for the SMBus alert protocol.
     * The format and meaning of the data value depends on the
     * protocol.For the SMBus alert protocol, there is a single bit
     * of data passed as the alert response's low bit ("event flag"). */
    void (*alert)(struct i2c_client *, unsigned int data);

    /* a ioctl like command that can be used to perform specific
     * functions with the device.
     */
    int (*command)(struct i2c_client *client, unsigned int cmd, void *arg);

    struct device_driver driver;
    const struct i2c_device_id *id_table;

    /* Device detection callback for automatic device creation */
    int (*detect)(struct i2c_client *, struct i2c_board_info *);
    const unsigned short *address_list;
    struct list_head clients;
```

```
};
```

当 I2C 设备和驱动匹配成功以后 probe 函数就会执行，和 platform 驱动一样。

device_driver 驱动结构体，如果使用设备树的话，需要设置 device_driver 的of_match_table 成员量，也就是驱动的兼容(compatible)属性。

id_table 是未使用设备树的设备匹配 ID 表。

对于 I2C 设备驱动编写人来说，重点工作就是构建 i2c_driver，构建完成以后需要向Linux 内核注册个 i2c_driver。 i2c_driver 注册函数为 int i2c_register_driver，此函数原型如下：

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
```

函数参数和返回值含义如下：

owner： 一般为 THIS_MODULE。

driver：要注册的 i2c_driver。

返回值： 0，成功；负值，失败。

另外 i2c_add_driver 也常常用于注册 i2c_driver， i2c_add_driver 是一个宏，定义如下：

```
#define i2c_add_driver(driver) i2c_register_driver(THIS_MODULE, driver)
```

i2c_add_driver 就是对 i2c_register_driver 做了一个简单的封装，只有一个参数，就是要注册的 i2c_river。

注销 I2C 设备驱动的时候需要将前面注册的 i2c_driver 从 Linux 内核中注销掉，需要用到i2c_del_drier 函数，此函数原型如下：

```
void i2c_del_driver(struct i2c_driver *driver)
```

函数参数和返回值含义如下：

driver：要注销的 i2c_driver。

返回值： 无。

3、示例代码

```
static int xxx_probe(struct i2c_client *client,const struct i2c_device_id *id)
{
    /* 函数具体程序 */
    return 0;
}

/* i2c 驱动的 remove 函数 */
static int xxx_remove(struct i2c_client *client)
{
    /* 函数具体程序 */
    return 0;
}

/* 传统匹配方式 ID 列表 */
static const struct i2c_device_id xxx_id[] = {
    {"xxx", 0},
    {}
```

```
};

/* 设备树匹配列表 */
static const struct of_device_id xxx_of_match[] = {
    { .compatible = "xxx" },
    { /* Sentinel */ }
};

/* i2c 驱动结构体 */
static struct i2c_driver xxx_driver = {
    .probe = xxx_probe,
    .remove = xxx_remove,
    .driver = {
    .owner = THIS_MODULE,
    .name = "xxx",
        .of_match_table = xxx_of_match,
    },
    .id_table = xxx_id,
};

/* 驱动入口函数 */
static int __init xxx_init(void)
{
    int ret = 0;

    ret = i2c_add_driver(&xxx_driver);
    return ret;
}

/* 驱动出口函数 */
static void __exit xxx_exit(void)
{
    i2c_del_driver(&xxx_driver);
}

module_init(xxx_init);
module_exit(xxx_exit);
```

三、I2C 设备和驱动匹配过程

I2C 设备和驱动的匹配过程是由 I2C 核心来完成的， drivers/i2c/i2c-core.c 就是 I2C 的核心部分， I2C 核心提供了一些与具体硬件无关的 API 函数

1、 i2c_adapter 注册/注销函数

```
int i2c_add_adapter(struct i2c_adapter *adapter)
int i2c_add_numbered_adapter(struct i2c_adapter *adap)
void i2c_del_adapter(struct i2c_adapter * adap)
```

2、 i2c_driver 注册/注销函数

```
int i2c_register_driver(struct module *owner, struct i2c_driver *driver)
int i2c_add_driver (struct i2c_driver *driver)
void i2c_del_driver(struct i2c_driver *driver)
```

设备和驱动的匹配过程也是由 I2C 总线完成的， I2C 总线的数据结构为 i2c_bus_type，定义在 driver/i2c/i2c-core.c 文件， i2c_bus_type 内容如下：

```
struct bus_type i2c_bus_type = {
    .name = "i2c",
    .match = i2c_device_match,
    .probe = i2c_device_probe,
    .remove = i2c_device_remove,
    .shutdown = i2c_device_shutdown,
};
```

match 是I2C总线的设备和驱动匹配函数，在这里就是 i2c_device_match 这个函数，此函数内容如：

```
static int i2c_device_match(struct device *dev, structdevice_driver *drv)
{
    struct i2c_client *client = i2c_verify_client(dev);
    struct i2c_driver *driver;

    if (!client)
        return 0;

    /* Attempt an OF style match */ /* 用于完成设备树设备和驱动匹配 compatible 属性和 of_device_id 中的 compatible 属性是否相等，如果相当的话就表示 I2C设备和驱动匹配 */
    if (of_driver_match_device(dev, drv))
        return 1;

    /* Then ACPI style match */
    if (acpi_driver_match_device(dev, drv))  // ACPI形式的匹配。
        return 1;

    driver = to_i2c_driver(drv);
    /* match on an id table if there is one */ /* 无设备树的 I2C 设备和驱动匹配过程，比较 I2C设名字和 i2c_device_id 的 name 字段是否相等，相等的话就说明 I2C 设备和驱动匹配*/
    if (driver->id_table)
        return i2c_match_id(driver->id_table, client) != NULL;

    return 0;
}
```

该匹配过程与platform总线的匹配方式类似

四、I2C 设备驱动编写流程

I2C 适配器驱动 SOC 厂商已经编写好了，我们需要做的就是编写具体的设备驱动。

1、I2C 设备信息描述

（1）未使用设备树

在未使用设备树的时候需要使用i2c_board_info 结构体来描述一个具体的 I2C 设备。i2c_board_info 结构体如下：

```
struct i2c_board_info {
    char type[I2C_NAME_SIZE];     /* I2C 设备名字 */
    unsigned short flags;     /* 标志 */
    unsigned short addr;     /* I2C 器件地址 */
    void *platform_data;
    struct dev_archdata *archdata;
    struct device_node *of_node;
    struct fwnode_handle *fwnode;
    int irq;
};
```

type和addr这两个成员变量是必须要设置的，一个是I2C设备的名字，一个是 I2C 设备的器件地址。

（2）使用设备树

使用设备树的时候 I2C 设备信息通过创建相应的节点就行了，参考Documentation/devicetree/bindngs/i2c中的相关文档，根据文档中的要求对设备资源进行描述，实际上使用设备树的方式内核会读设备树种的信息将对应的设备节点转换为设备结构体。

2、I2C 设备数据收发处理流程

I2C 设备驱动首先要做的就是初始化 i2c_driver 并向 Linux 内核注册。当设备和驱动匹配以后 i2c_drier 里面的 probe 函数就会执行，一般需要在 probe 函数里面初始化 I2C 设备，要初始化 I2C 设备必须能够对 I2C 设备寄存器进行读写操作，这里要用到 i2c_transfer 函数，i2c_transfer 函数最终会用 I2C 适配器中 i2c_algorithm 里面的 master_xfer 函数，对于 I.MX6U（恩智浦半导体公司的系列片）而言就是i2c_imx_xfer 这个函数。 i2c_transfer 函数原型如下：

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

函数参数和返回值含义如下：

adap： 所使用的 I2C 适配器， i2c_client 会保存其对应的 i2c_adapter。

msgs： I2C 要发送的一个或多个消息。

num： 消息数量，也就是 msgs 的数量。

返回值： 负值，失败，其他非负值，发送的 msgs 数量。

重点来看一下 msgs 这个参数，这是一个 i2c_msg 类型的指针参数， I2C 进行数据收发说白了就是息的传递， Linux 内核使用 i2c_msg 结构体来描述一个消息。 i2c_msg 结构体定义在 include/uapi/linux/i2c.h 文件中，结构体内容如下：

```
struct i2c_msg {
    __u16 addr;  /* 从机地址 */
    __u16 flags; /* 标志 */
    #define I2C_M_TEN 0x0010
    #define I2C_M_RD 0x0001
    #define I2C_M_STOP 0x8000
    #define I2C_M_NOSTART 0x4000
    #define I2C_M_REV_DIR_ADDR 0x2000
    #define I2C_M_IGNORE_NAK 0x1000
    #define I2C_M_NO_RD_ACK 0x0800
    #define I2C_M_RECV_LEN 0x0400
    __u16 len; /* 消息(本 msg)长度 */
```

```
    __u8 *buf; /* 消息数据 */
}
```

使用i2c_transfer 函数发送数据之前要先构建好 i2c_msg，使用 i2c_transfer 进行 I2C 数据收发的示代码如下：

```
/* 设备结构体 */
struct xxx_dev {
    ......
    void *private_data; /* 私有数据，一般会设置为 i2c_client */
};

/*
 * @description : 读取 I2C 设备多个寄存器数据
 * @param – dev : I2C 设备
 * @param – reg : 要读取的寄存器首地址
 * @param – val : 读取到的数据
 * @param – len : 要读取的数据长度
 * @return : 操作结果14
 */
static int xxx_read_regs(struct xxx_dev *dev, u8 reg, void *val,int len)
{
    int ret;
    struct i2c_msg msg[2];
    struct i2c_client *client = (struct i2c_client *)dev->private_data;

    /* msg[0]，第一条写消息，发送要读取的寄存器首地址 */
    msg[0].addr = client->addr;     /* I2C 器件地址 */
    msg[0].flags = 0;     /* 标记为发送数据 */
    msg[0].buf = &reg     /* 读取的首地址 */
    msg[0].len = 1;     /* reg 长度 */

    /* msg[1]，第二条读消息，读取寄存器数据 */
    msg[1].addr = client->addr;     /* I2C 器件地址 */
    msg[1].flags = I2C_M_RD;     /* 标记为读取数据 */
    msg[1].buf = val;     /* 读取数据缓冲区 */
    msg[1].len = len;     /* 要读取的数据长度 */

    ret = i2c_transfer(client->adapter, msg, 2);
    if(ret == 2) {
        ret = 0;
    } else {
        ret = -EREMOTEIO;
    }
    return ret;
}

/*
 * @description : 向 I2C 设备多个寄存器写入数据
 * @param – dev : 要写入的设备结构体
 * @param – reg : 要写入的寄存器首地址
 * @param – val : 要写入的数据缓冲区
 * @param – len : 要写入的数据长度
 * @return : 操作结果
```

```
 */
static s32 xxx_write_regs(struct xxx_dev *dev, u8 reg, u8 *buf,u8 len)
{
    u8 b[256];
    struct i2c_msg msg;
    struct i2c_client *client = (struct i2c_client *)dev->private_data;

    b[0] = reg; /* 寄存器首地址 */
    memcpy(&b[1],buf,len); /* 将要发送的数据拷贝到数组 b 里面 */

    msg.addr = client->addr; /* I2C 器件地址 */
    msg.flags = 0; /* 标记为写数据 */

    msg.buf = b; /* 要发送的数据缓冲区 */
    msg.len = len + 1; /* 要发送的数据长度 */

    return i2c_transfer(client->adapter, &msg, 1);
}
```

还有两个API函数分别用于I2C数据的收发操作，这两个函数最终都会调用i2c_transfer。首先来看一下2C 数据发送函数 i2c_master_send，函数原型如下：

```
int i2c_master_send(const struct i2c_client *client,const char *buf,int count)
```

函数参数和返回值含义如下：

client： I2C 设备对应的 i2c_client。

buf：要发送的数据。

count： 要发送的数据字节数，要小于 64KB，以为 i2c_msg 的 len 成员变量是一个 u16(无符号 16 位)类型的数据。

返回值： 负值，失败，其他非负值，发送的字节数

I2C 数据接收函数为 i2c_master_recv，函数原型如下：

```
int i2c_master_recv(const struct i2c_client *client, char *buf, int count)
```

函数参数和返回值含义如下：

client： I2C 设备对应的 i2c_client。

buf：要接收的数据。

count： 要接收的数据字节数，要小于 64KB，以为 i2c_msg 的 len 成员变量是一个 u16(无符号 16 位)类型的数据。

返回值： 负值，失败，其他非负值，发送的字节数。