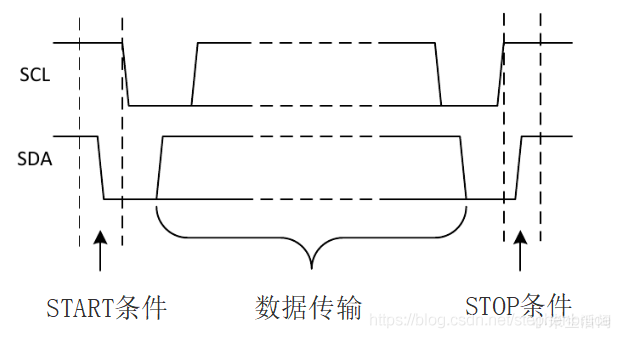
# 硬件说明

## 时序说明

I2C总线由时钟线SCL和数据线SDA组成，总线上可以连接多个主设备以及从设备，主设备触发从设备的数据传输，同一时间只能有一个主控。

主设备想要与从设备通信时，主设备通过发送START和STOP来启动和停止数据传输。当SCL为高电平时，SDA线由高变低的变化定义为START，当SCL为高电平时，SDA线由低变高的变化定义为STOP，如下图所示：



主设备发送START后，还需发送器件地址、寄存器地址、数据等信息，可以向从设备写入数据也可以从从设备读取数据。SDA线上的数据必须在时钟高电平期间保持稳定，也就是说SDA线上的数据变化必须在SCL的低电平期间完成。

每个数据字节或地址字节传输完成后，在第九个SCL的上升沿到来前，接收方需发送1个ACK应答位，即将SDA拉低，表示字节数据传输有效，才能继续传输。如果接收方将SDA拉高，即发送了NACK，则表示接受方无法解析数据或者无法再继续接收数据等。

对从设备的读或写通过读写位进行区分，读写位为地址字节的最低位，0表示写入，1表示读取。

## 错误说明

### 仲裁错误

前面已提到，i2c总线允许接入多个主设备，如果多个主设备想同时占用总线，如果某个主设备发送高电平，另一主设备发送低电平，因为总线的“线与”结果为低电平，所以发送高电平的主器件会在总线仲裁中失败。

在实际应用过程中，出现过一个主设备的情况下在i2c通信过程中控制器出现仲裁错误的状况，最终分析原因为SCL被旁边的高频信号线干扰，导致从设备将SCL被干扰出现的电平变化认定为正确的时钟，导致提前发送了应答位，而应答位为低电平，如果此时主设备发送的数据为高电平，那么就是出现总裁错误。此后主设备因为SDA线被拉低不能再继续发送数据，而从设备在等待SCL的上升沿来释放SDA，导致总线锁死，此时必须复位从设备才能修复。

# 驱动框架

应用程序

i2c client

i2c-dev

i2c设备驱动

i2c核心

i2c适配器

i2c控制器

i2c设备

i2c设备

i2c设备

用户空间

内核空间

硬件

i2c\_bus\_type

i2c驱动框架如图，用户空间可以直接通过i2c-dev.c提供的通用接口访问设备，也可以通过i2c设备驱动访问设备，设备的访问都是通过i2c适配器提供的通信方法，每个i2c适配器就对应着硬件上的i2c控制器，每个i2c控制器可以外挂多个i2c设备，直接通过设备驱动访问设备时，需要设备与驱动进行匹配，一个设备驱动可以支持多个同类型的i2c设备。

现在直接看框架肯定是不知所云，但是了解了框架后可以更快的去理解下面所讲的内容，等理解了下面的内容后，再回头来看框架就豁然开朗了。

# i2c核心

i2c核心i2c-core.c提供了i2c适配器、i2c设备驱动的注册和注销方法以及i2c通信方法等，完成i2c框架中硬件与软件的分层。

## i2c传输/发送/接收

int i2c\_master\_send(const struct i2c\_client \*client, const char \*buf, int count)

主机发送一条写消息。

int i2c\_master\_recv(const struct i2c\_client \*client, char \*buf, int count)

主机发送一条收消息。

int i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num)

执行一条或一组i2c消息，返回负数为错误码，其他为消息执行数目。

i2c\_master\_send/i2c\_master\_recv内部只是创建了i2c消息，然后还是调用了i2c\_transfer，下面我们分析一下其实现：

int i2c\_transfer(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs, int num)

{

if (adap->algo->master\_xfer) {

ret = \_\_i2c\_transfer(adap, msgs, num);

for (ret = 0, try = 0; try <= adap->retries; try++) {

ret = adap->algo->master\_xfer(adap, msgs, num);

}

}

}

可以看到i2c\_transfer并不能真正完成i2c消息的交互，其只是完成i2c适配器中实现的i2c传输方法的调用，具体传输过程还需i2c适配器完成，保证了与硬件的分离。

# i2c控制器驱动

内核使用i2c\_adapter结构来描述一个i2c适配器，而i2c适配器就对应着i2c控制器：

struct i2c\_adapter {

struct module \*owner; //所属模块

unsigned int class;

const struct i2c\_algorithm \*algo; //总线通信方法

void \*algo\_data; //algo的数据

const struct i2c\_lock\_operations \*lock\_ops;

struct rt\_mutex bus\_lock; //用于控制并发访问

struct rt\_mutex mux\_lock;

int timeout; //通信超时时间，如不设置内核默认设置为1s

int retries; //重试次数

struct device dev; //适配器设备

int nr;//适配器编号，决定了/dev/下i2c-x的编号

char name[48];//适配器编号

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;

};

其中的algo为总线通信方法，完成i2c的收发操作：

struct i2c\_algorithm {

//用于与i2c设备通信

int (\*master\_xfer)(struct i2c\_adapter \*adap, struct i2c\_msg \*msgs,int num);

//用于smbus的通信

int (\*smbus\_xfer) (struct i2c\_adapter \*adap, u16 addr,

unsigned short flags, char read\_write,

u8 command, int size, union i2c\_smbus\_data \*data);

u32 (\*functionality) (struct i2c\_adapter \*);

#if IS\_ENABLED(CONFIG\_I2C\_SLAVE)

int (\*reg\_slave)(struct i2c\_client \*client);

int (\*unreg\_slave)(struct i2c\_client \*client);

#endif

};

i2c控制器驱动的实现就是定义并初始化i2c\_adapter结构，对于i2c\_algorithm中的函数，至少要实现master\_xfer。

初始化完成后就可以向内核中添加：

int i2c\_add\_adapter(struct i2c\_adapter \*adapter)

int i2c\_add\_numbered\_adapter(struct i2c\_adapter \*adap)

两者的区别是前者动态申请适配器编号，后者指定适配器编号。

对应的注销函数为：

void i2c\_del\_adapter(struct i2c\_adapter \* adap)

# i2c设备驱动

## 描述

内核使用 i2c\_client来描述i2c设备信息：

struct i2c\_client {

unsigned short flags; //标志

unsigned short addr; //设备地址，存储在低7位

char name[I2C\_NAME\_SIZE]; //设备名称

struct i2c\_adapter \*adapter; //设备使用的i2c适配器

struct device dev; //设备

int irq; struct list\_head detected;

#if IS\_ENABLED(CONFIG\_I2C\_SLAVE)

i2c\_slave\_cb\_t slave\_cb;

#endif

};

i2c\_driver结构则描述了一个i2c设备驱动：

struct i2c\_driver {

unsigned int class;

int (\*attach\_adapter)(struct i2c\_adapter \*) \_\_deprecated;

int (\*probe)(struct i2c\_client \*, const struct i2c\_device\_id \*);//设备与驱动匹配成功后执行

int (\*remove)(struct i2c\_client \*);

void (\*shutdown)(struct i2c\_client \*);

void (\*alert)(struct i2c\_client \*, enum i2c\_alert\_protocol protocol,unsigned int data);

int (\*command)(struct i2c\_client \*client, unsigned int cmd, void \*arg);

struct device\_driver driver;

const struct i2c\_device\_id \*id\_table;

int (\*detect)(struct i2c\_client \*, struct i2c\_board\_info \*);

const unsigned short \*address\_list;

struct list\_head clients;

};

定义并初始化i2c\_driver后，可向内核进行注册：

int i2c\_register\_driver(struct module \*owner, struct i2c\_driver \*driver)

owner一般为THIS\_MODULE。

当然也可以直接使用i2c\_add\_driver(driver)，其只是对i2c\_register\_driver()的封装。

相应的注销函数为：

void i2c\_del\_driver(struct i2c\_driver \*driver)

内核中使用更普遍的是module\_i2c\_driver(\_\_i2c\_driver)可以直接完成i2c驱动的注册和注销，这是一个宏定义：

#define module\_i2c\_driver(\_\_i2c\_driver) \

module\_driver(\_\_i2c\_driver, i2c\_add\_driver, \

i2c\_del\_driver)

理解一下module\_driver()的实现，module\_i2c\_driver(xxx\_driver)会展开为：

static int \_\_int xxx\_driver\_init(void)  
 {  
 return i2c\_register\_driver(&xxx\_driver);  
 }  
 module\_init(xxx\_driver\_init);  
 static void \_\_exit xxx\_driver\_exit(void)  
 {  
 return i2c\_del\_driver(&xxx\_driver);  
 }  
 module\_exit(xxx\_driver\_exit);

## 设备与驱动的匹配

i2c device(i2c\_client)和i2c driver的匹配与platform框架的匹配过程相似，区别是改用i2c\_bus：

struct bus\_type i2c\_bus\_type = {

.name = "i2c",

.match = i2c\_device\_match,

.probe = i2c\_device\_probe,

.remove = i2c\_device\_remove,

.shutdown = i2c\_device\_shutdown,

};

同样，无论是i2c设备去匹配i2c驱动还是i2c驱动去匹配i2c设备，都会执行 i2c\_device\_match：

static int i2c\_device\_match(struct device \*dev, struct device\_driver \*drv)

{

struct i2c\_client \*client = i2c\_verify\_client(dev);

struct i2c\_driver \*driver;

if (of\_driver\_match\_device(dev, drv))

return 1;

if (acpi\_driver\_match\_device(dev, drv))

return 1;

driver = to\_i2c\_driver(drv);

if (driver->id\_table)

return i2c\_match\_id(driver->id\_table, client) != NULL;

return 0;

}

可以看到i2c的匹配方法有三种：

1. 基于设备树匹配，就是对比device和driver的“compatible”，过程参考platform的基于设备树匹配。
2. 基于ACPI的匹配。
3. 如果设置了driver的id\_table，则会遍历id\_table，如果设备的名称与id\_table中的任意一个匹配则成功。

在platform设备添加或驱动注册时执行platform bus的匹配，那么i2c设备与驱动的匹配在哪执行呢。

driver去匹配device：

i2c\_register\_driver

driver->driver.bus = &i2c\_bus\_type;

driver\_register(&driver->driver);

device去匹配driver：

i2c\_add\_adapter

i2c\_register\_adapter

of\_i2c\_register\_devices

for\_each\_available\_child\_of\_node(bus, node) {

client = of\_i2c\_register\_device(adap, node);

i2c\_new\_device

client->dev.bus = &i2c\_bus\_type;

device\_register(&client->dev);

}

以上可以看到i2c device的展开是与platform device的展开是有区别的，i2c device的展开是在添加i2c适配器时遍历其子节点然后添加i2c设备。

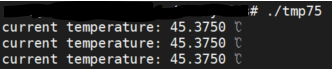
# 示例说明

我们以tmp75温度芯片为例，来完成对其温度的读取，内核中已经实现tmp75的驱动lm75.c，但是我们自己实现一个，tmp75可以设置分辨率、可以报警等，我们不去关注具体的功能，我们只是更改其分辨率然后读取温度，完成对其的读和写操作以演示i2c设备驱动的实现。当然也可以不实现驱动，直接通过i2c\_dev.c提供的接口与应用层通信。

## i2c-dev使用

虽然i2c-dev.c提供了read、write接口，但是他只能完成单条的i2c读或写操作，而且在写之前还需通过ioctl的I2C\_SLAVE命令设置从设备地址，不太通用也不太好用，所以使用ioctl的I2C\_RDWR命令完成i2c读写操作。

test/tmp75.c使用ioctl完成了i2c的读和写操作，利用写操作设置了tmp75的分辨率，利用读操作读取了温度，运行结果如下：



## 设备驱动使用

设备树节点如下：

&i2c2 {

pinctrl-names = "default";

pinctrl-0 = <&i2c2\_pins>;

status = "okay";

clock-frequency = <100000>;

tmp75@4f {

compatible = "tmp75";

reg = <0x4f>;

status = "okay";

};

};

测试程序test\_tmp75\_driver.c同样也是完成对tmp75分辨率的设置与温度的读取，与使用i2c-dev不同的是，与设备的读写操作过程在内核中实现，其实也就是调用了在i2c核心中已经提供的传输函数，这样应用程序就不用再关心芯片的寄存器或地址等硬件信息了，直接调用接口即可。运行结果如下：

