linux 下的 IIC 子系统

1、IIC 总线知识介绍

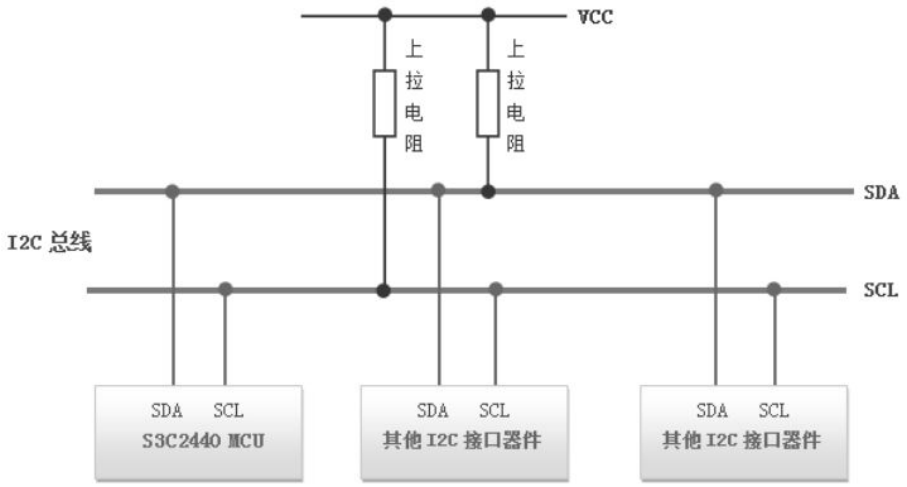
2、linux 下的 IIC 子系统

一、IIC 总线知识介绍

1.1、I2C 总线知识

IIC 设备是一种通过 IIC 总线连接的设备，由于其简单性，被广泛引用于电子系统中。在现代电子系统中，有很多的 IIC 设备需要进行相互之间通信，IIC 总线是由 PHILIPS 公司开发的两线式串行总线，用于连接微处理器和外部 IIC 设备。IIC 设备产生于 20 世纪 80 年代，最初专用与音频和视频设备，现在在各种电子设备中都广泛应用

1.2、IIC 总线物理拓扑



I2C 总线在物理连接上非常简单，分别由 SDA(串行数据线)和 SCL(串行时钟线)及上拉电阻组成。通信原理是通过对 SCL 和 SDA 线高低电平时序的控制，来产生 I2C 总线协议所需要的信号进行数据的传递。在总线空闲状态时，这两根线一般被上面所接的上拉电阻拉高，保持着高电平。

1.3、I2C 总线特征

I2C 总线上的每一个设备都可以作为主设备或者从设备，而且每一个设备都会对应一个唯一的地址(可以从 I2C 器件的数据手册得知)，主从设备之间就通过这个地址来确定与哪个器件进行通信，在通常的应用中，我们把 CPU 带 I2C 总线接口的模块作为

主设备，把挂接在总线上的其他设备都作为从设备。

与其他总线相比，IIC 总线有很多重要的特点。在选择一种设备来完成特定功能时，这些特点是选择 IIC 设备的重要依据。

主要特点：

1，每一个连接到总线的设备都可以通过唯一的设备地址单独访问

2，串行的 8 位双向数据传输，位速率在标准模式下可达到 100kb/s;快速模式下可以达到 400kb/s;高速模式下可以达到 3.4Mb/s

3，总线长度最长 7.6m 左右

4，片上滤波器可以增加抗干扰能力，保证数据的完成传输

5，连接到一条 IIC 总线上的设备数量只受到最大电容 400pF 的限制

6，它是一个多主机系统，在一条总线上可以同时有多个主机存在，通过冲突检测方式和延时等待防止数据不被破坏。同一时间只能有一个主机占用总线

1.4、I2C 总线协议

空闲状态： SCL 和 SDA 都保持着高电平。

起始条件： 总线在空闲状态 时， SCL 和 SDA 都保持着高电平，当 SCL 为高电平而 SDA 由高到低的跳变，表示产生一个起始条件。在起始条件产生后，总线处于忙状态，由本次数据传输的主从设备独占，其他 I2C 器件无法访问总线。

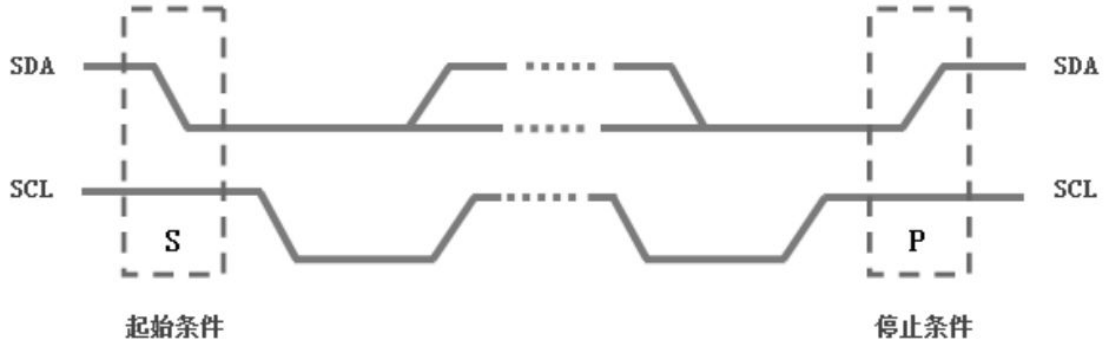
停止条件： 当 SCL 为高而 SDA 由低到高的跳变，表示产生一个停止条件。

应答信号： 从机在接收到每个字节传输完成后的下一个时钟信号， 在 SCL 高电平期间， SDA 为低，则表示一个应答信号。

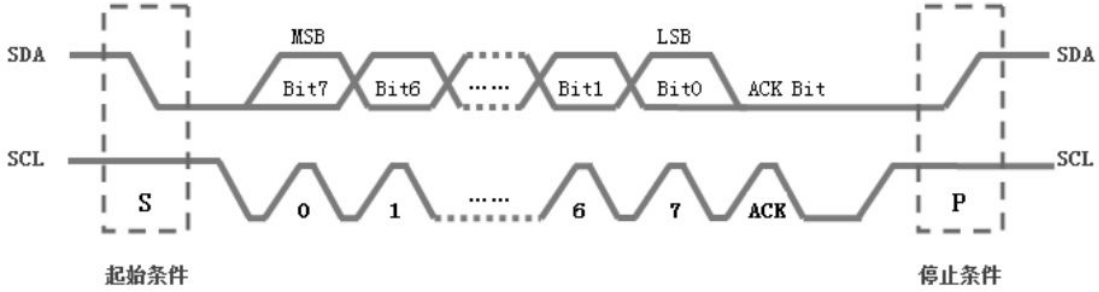
非应答信号： 从机在接收到每个字节传输完成后的下一个时钟信号， 在 SCL 高电平期间， SDA 为高，则表示一个应答信号。

注意：起始和结束信号总是由主设备产生。

基本时序如下图所示：

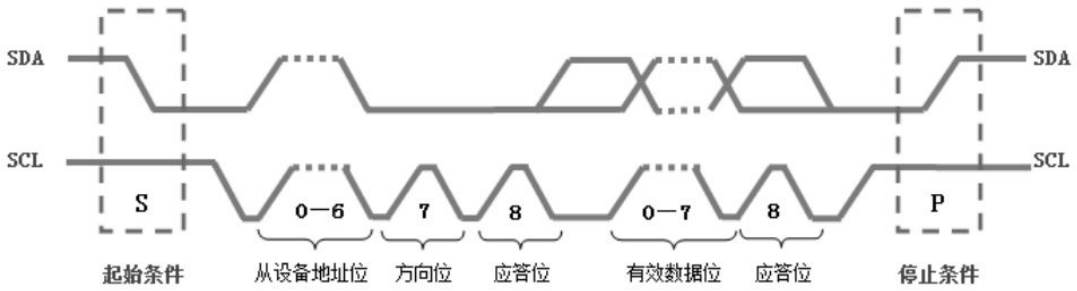


产生起始条件后，开始数据传输，这个阶段主设备在 SCL 线上的每一个脉冲，都会同时在 SDA 线上传输一个数据位(地址数据传输方式和普通数据传输方式相同)，每个字节完成后，跟着一个应答位。 当不想再进行数据传输时，主机产生一个停止信号，总线释放， SCL,SDA 线都回到空闲状态。数据传输时序图如下：



1.5、I2C 寻址方式

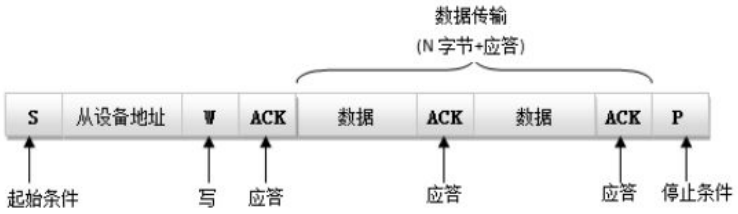
I2C 总线上的每一个 I2C 设备都对应一个唯一的地址，主从设备之间的数据传输是建立在地址的基础上，主设备在传输有效数据之前要先指定从设备的地址，地址指定的过程和上面数据传输的过程一样， 只不过大多数从设备的地址是 7 位的（有的设备地址是 10 位的， 发送地址要使用两个字节， 这里仅以 7 位地址为例子），然后协议规定再给地址添加一个最低位用来表示接下来数据传输的方向， 0 表示主设备向从设备写数据， 1 表示主设备向从设备读数据。如图所示：



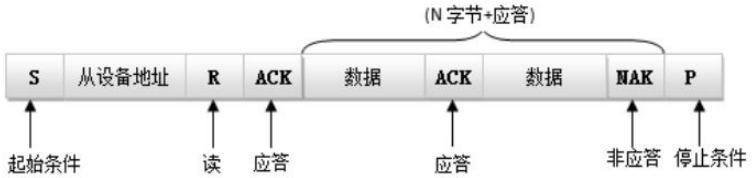
1.6、I2C 总线操作

对 I2C 总线的操作实际就是主从设备之间的读写操作。大致可分为以下三种操作情况：

主设备往从设备中写数据。数据传输格式如下：



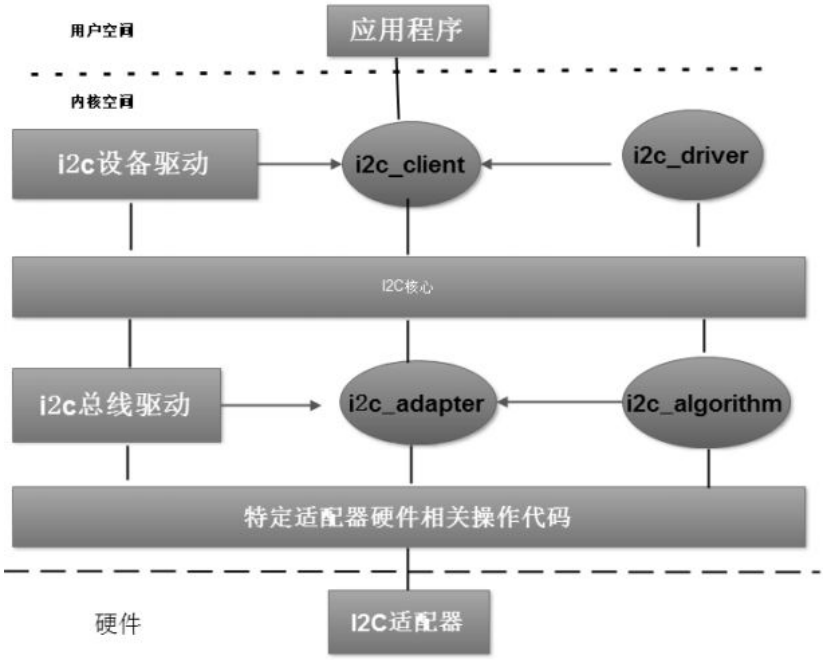
主设备从从设备中读数据。数据传输格式如下：



主设备往从设备中读写数据，数据传输格式如下：

二、linux 下的 IIC 子系统

在 Linux 下要使用 I2C 总线并没有像裸机的那样简单，为了体现 Linux 中的模块架构， Linux 把 I2C 总线的使用进行了结构化。这种结构分三部分组成，他们分别是：I2C 核心层、 I2C 总线层和 I2C 设备层。结构图如下：



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 总线层和设备层的注册、注销方法、I2C 通信方法、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.这部分是与平台无关的.与其对应的是 Linux 内核源代码中的 drivers 目录下的 i2c-core.c.

I2C 总线层:I2C 总线层是对 I2C 硬件体系结构中适配器端的实现.I2C 总线驱动主要包含了 I2C 适配器数据结构 i2c_adapter,I2C 适配器的数据结构 i2c_algorithm 和控制 I2C 适配器产生通信信号的函数,经由 I2C 总线驱动的代码,我们可以控制 I2C 适配器以主控方式产生开始位,停止位,读写周期,以及以从设备方式被读写,产生 ACK 等.不同的 CPU 平台对应着不同的 I2C 总线驱动.

I2C 设备层:I2C 设备层是对 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.I2C 设备层主要包含了数据结构 i2c_driver 和 i2c_client,我们需要根据具体设备实现其中的成员函数.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.

由此看来,在 Linux 下驱动 I2C 总线不像单片机中那样简单的操作几个寄存器了,而是把 I2C 总线结构化、抽象化了,符合通用性和 Linux 设备模型。

三、IIC 子系统下重要的数据结构

1、struct i2c_driver 对应驱动方法，当总线上注册了对应的 i2c 从设备时，如果可以匹配成功，则调用 probe 函数初始化设备，注册字符设备，提供接口给应用程序。

```
struct    i2c_driver
{
    unsigned int class;                                    //驱动的类型
    int (*attach_adapter)(struct i2c_adapter *); //当检测到适配器时调用的函数
    int (*detach_adapter)(struct i2c_adapter*);     //卸载适配器时调用的函数

    int (*probe)(struct i2c_client *,const struct    i2c_device_id *);//设备探测函数
    int (*remove)(struct i2c_client *);              //设备的移除函数

    void (*shutdown)(struct i2c_client *);                    //关闭 IIC 设备
    int (*suspend)(struct    i2c_client *,pm_messge_t mesg); //挂起 IIC 设备
    int (*resume)(struct    i2c_client *);                    //恢复 IIC 设备

    void (*alert)(struct i2c_client *, unsigned int data);

    int (*command)(struct i2c_client *client,unsigned int cmd,void *arg);//使用命令使设备完成特殊的功能。类似ioctl（）函数

    struct device_driver driver;                        //设备驱动结构体
    const struct    i2c_device_id *id_table;            //设备 ID 表

    int (*detect)(struct i2c_client *,int    kind,struct    i2c_board_info *); //自动探测设备的回调函数
    const unsigned short *address_list;
    struct    list_head       clients;                 //指向驱动支持的设备
};
```

2、struct i2c_client

一个 struct i2c_client 结构体代表 i2c 总线上一个从设备

```
struct      i2c_client
{
    unsigned short flags;              //标志位
    unsigned short    addr;            //设备的地址，低 7 位为芯片地址
    char name[I2C_NAME_SIZE]; //设备的名称，最大为 20 个字节
    struct i2c_adapter *adapter; //依附的适配器 i2c_adapter，适配器指明所属的总线
    struct     i2c_driver *driver;     //指向设备对应的驱动程序
    struct device    dev;              //设备结构体
    int irq;                           //设备申请的中断号
    struct list_head detected;         //已经被发现的设备链表
};
```

设备结构体 i2c_client 中 addr 的低 8 位表示设备地址。设备地址由读写位、器件类型和自定义地址组成，如下图：

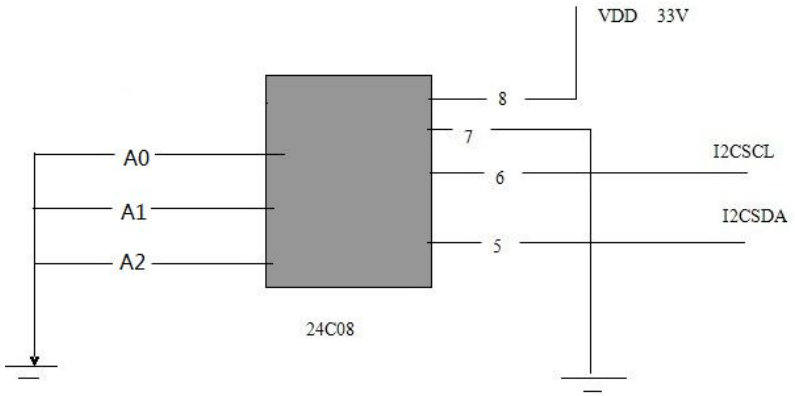
R/W(1 位)	器件类型（4 位）	自定义地址(3 位)

（1）第 7 位是 R/W 位，0 表示写，1 表示读，所以 I2C 设备通常有两个地址，即读地址和写地址。

（2）类型器件由中间 4 位组成，这是由半导体公司生产的时候就已经固化了。

（3）自定义类型由低 3 位组成。由用户自己设置，通常的做法如 EEPROM 这些器件是由外部 I 芯片的 3 个引脚所组合电平决定的（A0,A1,A2）。A0,A1,A2 就是自定义的地址码。自定义的地址码只能表示 8 个地址，所以同一 IIC 总线上同一型号的芯片最多只能挂载 8 个。

AT24C08 的自定义地址码如图：A0,A1,A2 接低电平，所以自定义地址码为 0；



3、struct i2c_adapter

IIC 总线适配器就是一个 IIC 总线控制器，在物理上连接若干个 IIC 设备。IIC 总线适配器本质上是一个物理设备，其主要功能是完成 IIC 总线控制器相关的数据通信，定义如下：

```
struct i2c_adapter
{
    struct module *owner;                   //模块者
    unsigned    int    class;               //允许探测的驱动类型如传感器，  eeprom  等
    const struct i2c_algorithm *algo;       //指向适配器的驱动程序
```

```
    void *algo_data;                    //指向适配器的私有数据，根据不同的情况使用方法不同

    struct  rt_mutex  bus_lock;         //对总线进行操作时，将获得总线锁

    int timeout;                        //超时 in jiffies
    int retries;                        //重试次数
    struct device dev;                  //指向 适配器的设备结构体

    int  nr ;  //适配器编号也是 bus 编号，第几条 i2c 总线
    char name[48];                      //适配器名称
    struct completion  dev_released;    //用于同步的完成量

    struct mutex userspace_clients_lock;    //链表操作的互斥锁
    struct list_head userspace_clients;     //连接总线上的设备的链表
};
```

4、struct i2c_algorithm

每一个适配器对应一个驱动程序，该驱动程序描述了适配器与设备之间的通信方法:

```
struct  i2c_algorithm
{
    /*传输函数指针，指向实现 IIC 总线通信协议的函数，用来确定适配器支持那些传输类型，提供的是 i2c_transfer 实现部分*/
    int  (*master_xfer)(struct  i2c_adapter *adap,  struct  i2c_msg *msg, int num);

    /*smbus 方式传输函数指针，指向实现 SMBus 总线通信协议的函数。SMBus 和 IIC 之间可以通过软件方式兼容，所以这里提供了一个函数，但是一般都赋值为 NULL*/
    int  (*smbus_xfer)(struct  i2c_adapter *adap, u16  addr, unsigned  short flags, char read_write, u8 command, int size, union  i2c_smbus_data  *data);

    /*返回适配器支持的功能,用户自己定义*/
    u32  (*functionality)(struct  i2c_adapter *);
};
```

5、struct i2c_msg

```
、struct i2c_msg
{
    __u16 addr;                         // 从机地址，这个字段说明一个适配器在获得总线控制权后，可以与多个 IIC 设备进行交互。
    __u16 flags;                        // 消息类型标志
    #define I2C_M_TEN          0x0010   // 十位地址标志
    #define I2C_M_RD           0x0001   // 接收数据标志
    #define I2C_M_NOSTART   0x4000 // FUNC_PROTOCOL_MANLING 协议的相关标志
    #define I2C_M_REV_DIR_ADDR   0x2000 //FUNC_PROTOCOL_MANLING 协议的相关标志
    #define I2C_M_IGNORE_NAK 0x1000//FUNC_PROTOCOL_MANLIN 协议的相关标志
    #define I2C_M_NO_RD_ACK 0x0800 //FUNC_PROTOCOL_MANLING 协议的相关标志
```

```
#define I2C_M_RECV_LEN     0x0400 //第一次接收的字节长度
__u16 len;                                  // 数据长度
__u8 *buf;                                  //指向消息数据的缓冲区
};
```

6、struct i2c_board_info

该结构体表示一个 i2c 设备模板，系统初始化期间，通过函数 i2c_register_board_info 将设备信息添加到全局链表中，在适配器注册时，从模板生成 i2c 设备（ struct i2c_client），定义如下：

```
struct i2c_board_info {
        char        type[I2C_NAME_SIZE];//用于初始化  i2c_client.name
        unsigned short flags;                   //用于初始化  i2c_client.flags
        unsigned short addr;                    //用于初始化  i2c_client.addr
        void        *platform_data;             //用于初始化  i2c_client.dev.platform_data
        struct dev_archdata *archdata;      //用于初始化  i2c_client.dev.archdata
        struct device_node *of_node;
        int         irq;                        //用于初始化  i2c_client.irq
};
```