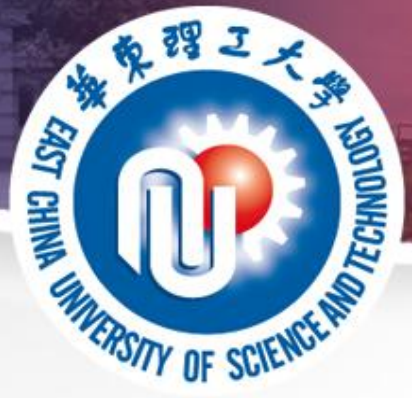


第十章 字符设备和驱动程序设计

华东理工大学计算机系

罗 飞

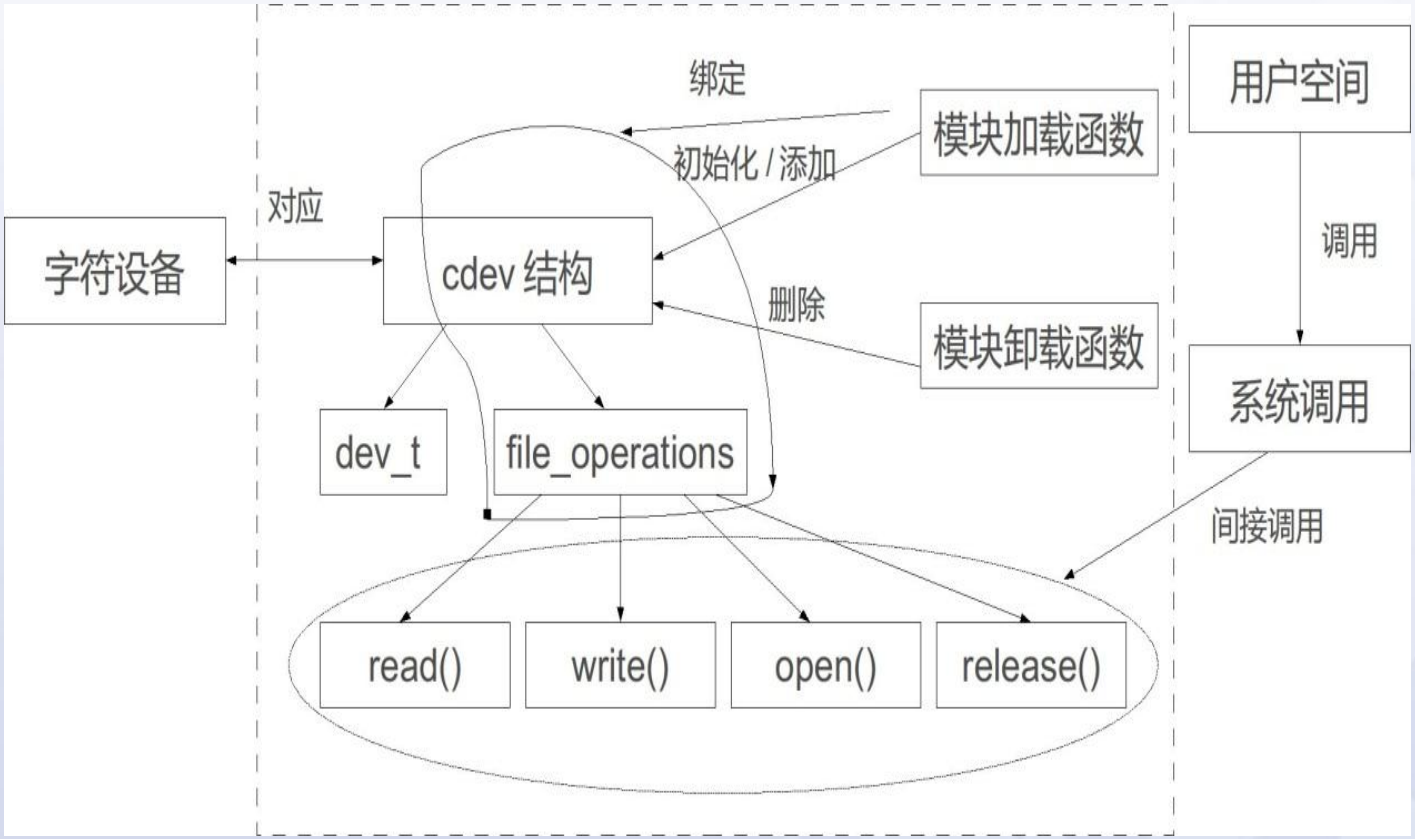
设备文件（回顾）

- Linux将所有的设备都当作文件来处理包括设备文件
- 它们可以使用和操作文件相同的、标准的系统调用接口来完成打开、关闭、读写和I/O控制操作

Content

1、字符设备驱动框架

- 驱动程序任务?



字符设备驱动程序

- Linux设备之一

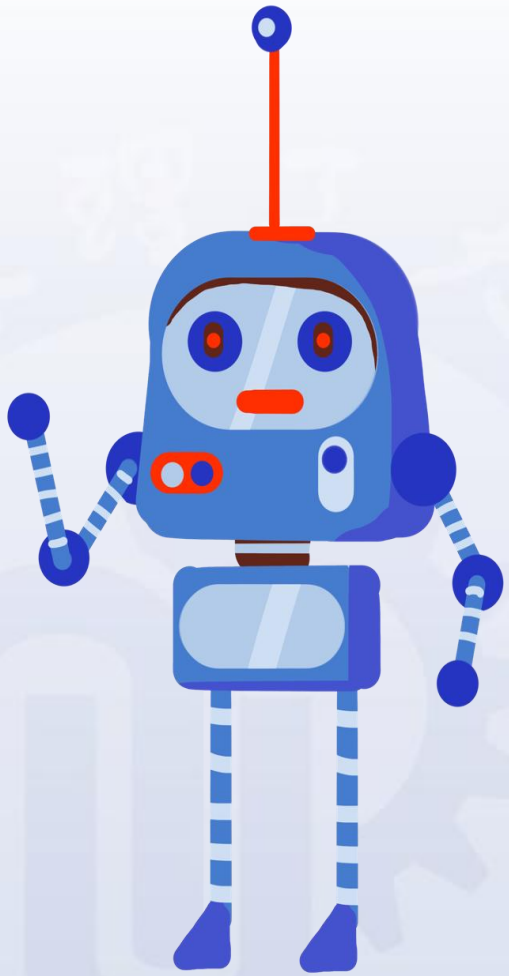
Linux设备种类?

- 字符设备功能非常强大
  - 几乎可以描述不涉及挂载文件系统的所有硬件设备

- 实现方式分为两种
  - 一种是直接编译进内核
  - 另一种是以模块方式加载，然后在需要使用驱动时加载

关键数据结构

- struct_cdev结构
  - 包含着字符设备所需的全部信息
- dev_t
  - 设备号
- file_operations



2、字符设备驱动开发

- 设备文件→设备节点
  - 位于/dev目录
- 查看设备文件
  - # ls –l
    - 第一列若是“c”表明字符设备，“b”则是块设备

P220

#mknod /dev/lp0 c 6 0

设备号

主设备号?
次设备号?

- 在Linux内核中，使用dev_t类型来表示设备号，这个类型在<linux/types.h>头文件中定义。

```
typedef  __u32          __kernel_dev_t;
typedef  __kernel_dev_t       dev_t;
```

- dev_t是一个32位的无符号数
  - 高12位用来表示主设备号，低20位用来表示次设备号
- 在2.6内核中，可以容纳大量的设备
  - 先前的内核版本最多只能使用255个主设备号和255个次设备号

设备号操作函数

- 内核主要提供了三个操作dev_t类型的函数
  - MAJOR(dev)、MINOR(dev)和MKDEV(ma, mi)。
  - 其中MAJOR(dev)用于获取主设备号。
  - MINOR(dev)则用于获取次设备号。
  - MKDEV(ma, mi) 根据主设备号ma和次设备号mi构造dev_t设备号。

- 在<linux/kdev_t.h>头文件中给出了这些宏的定义

```
#define MINORBITS    20
#define MINORMASK ((1U << MINORBITS) - 1)
#define MAJOR(dev)  ((unsigned int) ((dev) >>
MINORBITS))
#define MINOR(dev)  ((unsigned int) ((dev) &
MINORMASK))
#define MKDEV(ma,mi)      (((ma) << MINORBITS) | (mi))
```

注册和注销设备号

- 向内核请求分配一个或多个设备号。
- 内核专门提供了字符设备号管理的函数接口

```
int register_chrdev_region(dev_t first, unsigned int count, const char
*name);
int alloc_chrdev_region(dev_t *dev, unsigned int firstminor, unsigned int
count, const char *name);
void unregister_chrdev_region(dev_t first, unsigned int count);
```

- register_chrdev_region函数和alloc_chrdev_region函数用于分配设备号
  - 后者是以动态的方式分配的
- unregister_chrdev_region函数则用于释放设备号

关键数据结构

- 基本的驱动程序操作都会涉及
- 内核提供的三个关键数据结构
- 它们都在<linux/fs.h>头文件中定义
  - struct file_operations
  - struct file
  - struct inode

file_operations

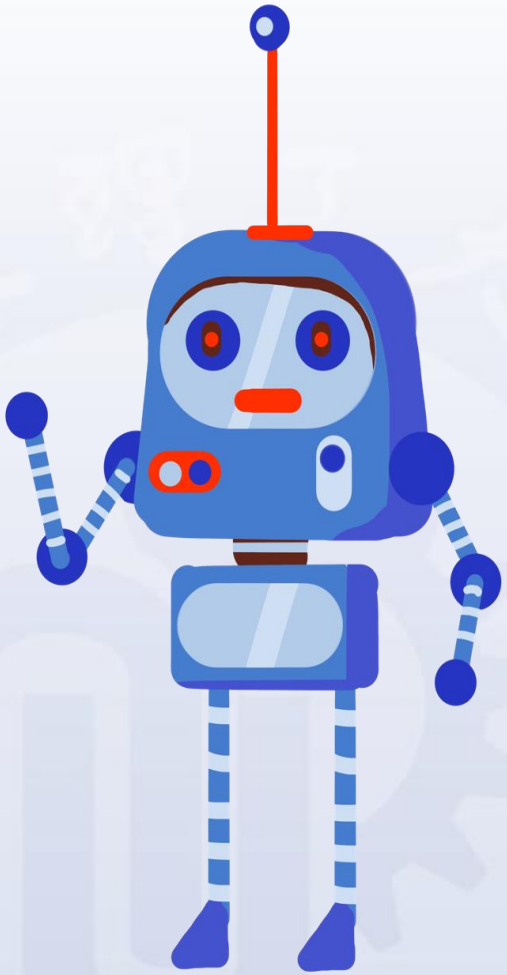
- file_operations结构体描述了一个文件操作所需要的所有函数
  - 以函数指针的形式给出的
- 每个打开的文件，在内核里都用file结构体表示
  - 结构体中一个成员为f_op，指向一个file_operations结构体
  - 通过这种形式将一个文件同它自身的操作函数关联起来，这些函数实际上是系统调用的底层实现

file_operations

- 编写驱动程序的主要工作就是实现这些函数中的一部分
- 字符设备一般只要实现open、release、read、write、mmap、ioctl
- 随着内核版本的不断改进，file_operations结构体的规模也越来越大

file_operations-P222-223

```
struct file_operations {
    //指向拥有该结构的模块的指针，一般初始化为THIS_MODULE
    struct module *owner;
    //用来改变文件中的当前读/写位置
    loff_t (*llseek) (struct file *, loff_t, int);
    //用来从设备中读取数据
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    //用来向设备写入数据
    ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
    //初始化一个异步读取操作
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    //初始化一个异步写入操作
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *, unsigned long, loff_t);
    //用来读取目录，对于设备文件，该成员应当为NULL
     int (*readdir) (struct file *, void *, filldir_t);
    //轮询函数，查询对一个或多个文件描述符的读或写是否会阻塞
     unsigned int (*poll) (struct file *, struct poll_table_struct *);
    //用来执行设备I/O操作命令
     int (*ioctl) (struct inode *, struct file *, unsigned int, unsigned long);
    //不使用BKL文件系统，将使用此函数代替ioctl
```



file_operations

```
 //在64位系统上，使用32位的ioctl调用将使用此函数代替
long (*compat_ioctl) (struct file *, unsigned int, unsigned long);
//用来将设备内存映射到进程的地址空间
int (*mmap) (struct file *, struct vm_area_struct *);
//用来打开设备
int (*open) (struct inode *, struct file *);
//执行并等待设备的任何未完成的操作
int (*flush) (struct file *, fl_owner_t id);
//用来关闭设备
int (*release) (struct inode *, struct file *);
//用来刷新待处理的数据
int (*fsync) (struct file *, struct dentry *, int datasync);
//fsync的异步版本
int (*aio_fsync) (struct kiocb *, int datasync);
//通知设备FASYNC标志的改变
```

file_operations

- 主要的函数
  - llseek()函数用于改变文件中的读写位置，并将新位置返回。
  - open()函数负责打开设备和初始化I/O。
  - release()函数负责释放设备占用的内存并关闭设备。
  - read()函数用来从设备中读取数据。
  - write()函数用来向设备上写入数据。
  - ioctl()函数实现对设备的控制。
  - mmap()函数将设备内存映射到进程的地址空间。若
  - aio_read()和aio_write()函数分别实现对设备进程异步的读写操作。

file

- Linux中的所有设备都是文件，在内核中使用file结构体来表示一个打开的文件

- file结构体中的重要成员
  - fmode_t f_mode
  - loff_t f_pos
  - unsigned int f_flags
  - const struct file_operations *f_op
  - void *private_data

inode

- inode是一个内核文件系统索引节点对象
  - 包含了内核在操作文件或目录时所需要的全部信息

- 在内核中inode结构体用来表示文件
  - 与表示打开文件的file结构体的区别：同个文件可能会有多个打开文件，因此一个inode结构体可能会对应着多个file结构体

inode

- 对于字符设备驱动来说，需要关心的是如何从inode结构体中获取设备号
- 与此相关的两个成员
  - dev_t i_rdev
    - 对于设备文件而言，此成员包含实际的设备号
  - struct cdev *i_cdev
    - 字符设备在内核中是用cdev结构来表示的。此成员是指向cdev结构的指针

inode

- 内核开发者提供了两个函数来从inode对象中获取设备号

```
static inline unsigned iminor(const struct inode
*inode)
{
    return MINOR(inode->i_rdev);
}

static inline unsigned imajor(const struct inode
*inode)
{
    return MAJOR(inode->i_rdev);
}
```

字符设备注册和注销

- 字符设备初始化
  - void cdev_init(struct cdev *, const struct file_operations *);
- 字符设备注册
  - struct cdev *cdev_alloc(void);
- 字符设备添加
  - void cdev_add(struct cdev *, dev_t, unsigned);
- 字符设备删除
  - void cdev_del(struct cdev *);

字符设备分配与初始化方法

- 方法一：

```
struct cdev *my_cdev = cdev_alloc();
my_cdev->owner = THIS_MODULE;
my_cdev->ops = &fops;
```

- 方法二：

```
struct cdev my_cdev;
cdev_init(&my_cdev, &fops);
my_cdev.owner = THIS_MODULE;
```

3、GPIO概述

- **GPIO接口**
  - 利用工业标准I22C、SMBus或SPI接口简化了I/O接口的扩展
- **功能**
  - 当微控制器或芯片组没有足够的I/O端口，或当系统需要使用远程串行通信或控制时，GPIO接口能够提供额外的控制和监视功能

GPIO驱动概述

- 在ARM里，所有I/O都是通用的，称为GPIO（General Purpose Input/Output，通用输入输出）
- 每个GPIO端口一般包含8个引脚，例如PA端口为PA0～PA7，可以控制I/O接口作为输入或者输出
- 许多设备或电路通常只需要一位，即表示开/关两状态就够了，例如LED灯的亮和灭
- GPIO接口一般至少会有两个寄存器，即控制寄存器和数据寄存器

GPIO结构

- GPIO接口一般至少会有两个寄存器
  - 控制寄存器和数据寄存器
  - 数据寄存器的各位都直接引到芯片外部
    - 针对该寄存器的每一位的功能，则可以通过控制寄存器中相应的位来设置
- GPIO接口的优点
  - 低功耗、小封装、低成本、较好的灵活性

4、串行总线概述

- 通信
  - 串行或并行模式，而串行模式应用更广泛
- 大多数微控制器都提供SPI 和I^2C接口
  - 用于发送、接收数据
  - 微处理器通过几条总线控制周边的设备
- 串行相比于并行的主要优点
  - 要求的线数较少

SPI总线

- 同步外设接口（Serial Peripheral Interface，SPI）
  - 由摩托罗拉公司推出
  - 一种高速的、全双工、同步的串行总线

- 它主要应用在EEPROM、Flash、实时时钟、AD转换器以及数字信号处理器和数字信号解码器之间。

SPI总线两种工作模式

- 主模式
- 从模式
  - 都支持3Mbit/s的速率
  - 并且还具有传输完成标志和写冲突保护标志

I²C总线

- 内部集成电路（Internal Integrated Circuit），通常也被称为I22C或者IIC
  - 主要用于连接微控制器和外围设备
  - 由Philips公司开发的二线式串行总线标准

- 最主要的特点
  - 简单性和高效性

SMBus总线

- 系统管理总线（System Management Bus，SMBus）
  - 最初由Intel提出
  - 应用于移动PC和桌面PC系统中的低速通讯
- SMBus总线同I^2C总线一样也是一种二线式串行总线，它使用一条数据线和一条时钟线进行通信
- 特点
  - 数据传输率较慢，只有大约100kbit/s
  - 结构简单、造价低

5、I²C总线驱动开发

- **I²C驱动架构三个组成部分**
- 由I²C核心
  - I²C总线驱动
  - I²C设备驱动

I²C核心

- 主要提供了以下几个功能
  - I²C总线驱动和设备驱动的注册及注销函数
  - I²C algorithm的上层代码实现
  - 探测设备、检测设备地址的上层代码实现

I²C总线驱动

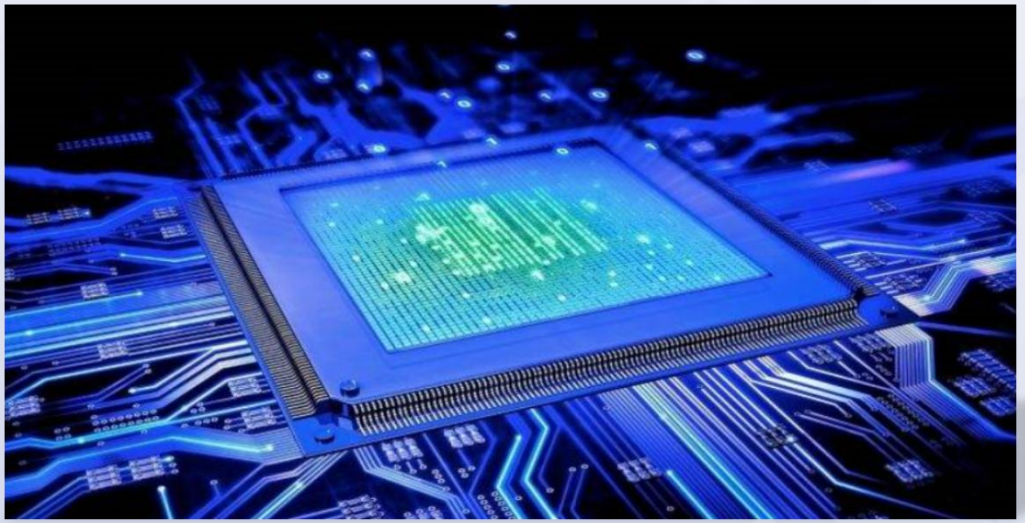
- 是对I²C硬件架构中适配器的具体实现
- 负责实现I²C适配器数据结构（i2c_adapter）、I²C适配器的algorithm数据结构（i2c_algorithm）以及控制适配器产生通信信号的函数

I²C设备驱动

- 对I²C硬件架构中设备的具体实现
- I²C设备驱动负责实现i2c_driver和i2c_client两个数据结构

关键数据结构

- i2c_adapter是对硬件上的适配器的抽象
  - 相当于整个I²C驱动的控制器
  - 作用是产生总线时序，例如开始位、停止位、读写周期等，用以读写I²C从设备。



```
struct i2c_adapter {
        struct module *owner;   /*所属模块*/
        unsigned int id;
        unsigned int class;     /*用来允许探测的类*/
        const struct i2c_algorithm *algo;  /*I22C algorithm结构体指针*/
        void *algo_data;                   /*algorithm所需数据*/

    /*client注册和注销时调用*/
        int (*client_register)(struct i2c_client *) __deprecated;
        int (*client_unregister)(struct i2c_client *) __deprecated;
        int timeout;                                    /*超时限制*/
        int retries;          /*重试次数*/
        struct device dev;                              /*适配器设备*/
        int nr;
        struct list_head clients;          /* client链表头*/
        char name[48];       /*适配器名称*/
        struct completion dev_released;
};
```

- i2c_alogorithm是提供控制适配器产生总线时序的函数

```
struct i2c_algorithm {
   //I22C传输函数指针
   int (*master_xfer)(struct i2c_adapter *adap, struct i2c_msg *msgs,
int num);
  //SMBus传输函数指针
   int (*smbus_xfer) (struct i2c_adapter *adap, u16 addr, unsigned
short flags, char read_write, u8 command, int size, union
i2c_smbus_data *data);
   //确定适配器所支持的功能
   u32 (*functionality) (struct i2c_adapter *);
};
```

- 与适配器对应的是从设备，表示它的数据结构为i2c_client
- 每个I²C设备都需要一个i2c_client来描述

```
struct i2c_client {
        unsigned short flags;   /*标志*/
        unsigned short addr;  /*芯片地址，注意：7位地址存储在低7
位*/
        char name[I2C_NAME_SIZE];    /*设备名字*/
         struct i2c_adapter *adapter;   /*依附的i2c_adapter指针*/
         struct i2c_driver *driver;         /*依附的i2c_driver指针*/
         struct device dev;                 /*设备结构体*/
         int irq;
         struct list_head list;             /*链表头*/
         struct list_head detected;
         struct completion released;     /*用于同步*/
};
```

- i2c_driver并不是任何真实物理设备的对应，它只是一套驱动函数
- 在I²C驱动架构中的设备驱动部分，i2c_driver是辅助类型的数据结构。

```
struct i2c_driver {
    int id;             /*唯一的驱动id*/
    unsigned int class;
    /*适配器添加函数（旧式）*/
    int (*attach_adapter)(struct i2c_adapter *);
    /*适配器删除函数（旧式）*/
    int (*detach_adapter)(struct i2c_adapter *);
    /*设备删除函数（旧式）*/
    int (*detach_client)(struct i2c_client *) __deprecated;
    /*设备添加函数（新式）*/
    int (*probe)(struct i2c_client *, const struct i2c_device_id *);
```

```
 /*设备删除函数（新式）*/
    int (*remove)(struct i2c_client *);
    void (*shutdown)(struct i2c_client *);          /*设备关闭函数*/
    /*设备挂起函数*/
    int (*suspend)(struct i2c_client *, pm_message_t mesg);
    int (*resume)(struct i2c_client *);             /*设备恢复函数*/
    int (*command)(struct i2c_client *client, unsigned int cmd, void
*arg); /*类似ioctl*/
    struct device_driver driver;                    /*设备驱动结构体*/
    /*此驱动支持的I22C设备列表*/
    const struct i2c_device_id *id_table;
    /*检测函数*/
    int (*detect)(struct i2c_client *, int kind, struct i2c_board_info *);
     const struct i2c_client_address_data *address_data;
     struct list_head clients;                      /*链表头*/
};
```

I²C核心

- I²C Core用于维护Linux的I²C核心部分
- 主要提供了一套接口函数，允许一个I²C adapter、I²C driver和I²C client在初始化时在I²C Core中进行注册，以及在退出时进行注销。

- 这些接口函数都是与具体硬件平台无关的函数，它们在drivers/i2c-core.c文件中定义，在linux/ic.h文件中声明。

I²C核心接口函数

- i2c_adapter的注册和注销
  - int i2c_add_adapter(struct i2c_adapter *adapter);
  - int i2c_del_adapter(struct i2c_adapter *adapter)
- i2c_driver的注册和注销
  - int i2c_register_driver(struct module *owner, struct i2c_driver *driver);
  - void i2c_del_driver(struct i2c_driver *driver);
  - static inline int i2c_add_driver(struct i2c_driver *driver);

I²C核心接口函数

- i2c_client的注册和注销
  - int i2c_attach_client(struct i2c_client *);
  - int i2c_detach_client(struct i2c_client *);

- 传输、发送和接收
  - int i2c_transfer(struct i2c_adapter *adap, struct i2c_msg *msgs, int num);
  - int i2c_master_send(struct i2c_client *client, const char *buf, int count);
  - int i2c_master_recv(struct i2c_client *client, char *buf, int count);

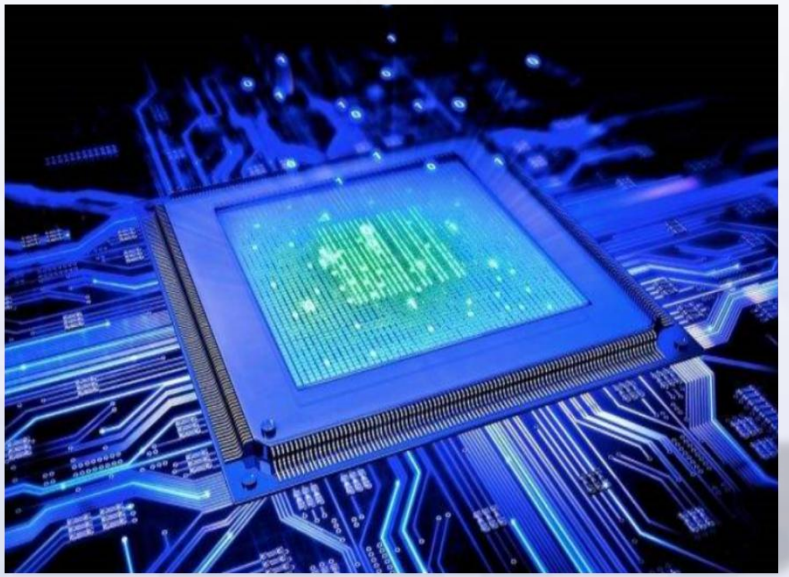
I²C总线驱动

- **任务**
  - 为系统中各个I²C总线增加相应的读写方法。
  - 一个总线驱动用于支持一条特定的I²C总线的读写。

- **组成**
  - 一个总线驱动通常需要两个模块，分别用一个i2c_adapter结构体和i2c_algorithm结构体来描述。

I²C总线驱动具体实现

- 加载函数
- 卸载函数
- 通信方法



加载函数

- 负责初始化I²C适配器所要使用的硬件资源

- 例如
  - 申请I/O地址、中断号等

  - 通过i2c_add_adapter() 函数注册i2c_adapter结构体，此结构体的成员函数指针已经被相应的具体实现函数初始化。

卸载函数

- 需要释放I²C适配器所占用的硬件资源
- 然后通过i2c_del_adapter() 函数注销i2c_adapter结构体

通信方法

- 针对特定的I2C适配器，还需要实现适合其硬件特性的通信方法
  - 即实现i2c_algorithm结构体

5、I²C设备驱动

- I²C设备驱动与与I²C总线驱动对应
- 组成
  - I²C设备驱动也分成两个模块，它们分别是i2c_driver和i2c_client结构体。

- I²C设备驱动要使用这两个结构体，并且负责填充其中的成员函数。

I²C设备驱动实现

- Linux内核中提供了一个通用的方法来实现I²C设备驱动
- 这个通用的I²C设备驱动由i2c-dev.c文件实现

THANKS!