浙江大学
ZHEJIANG UNIVERSITY

嵌入式系统

第10课 嵌入式驱动程序设计（1）

王总辉
zhwang@zju.edu.cn

http://course.zju.edu.cn

提纲

- Linux设备驱动程序简介
- Linux驱动相关内核机制
- Linux字符设备及驱动设计

Linux设备驱动程序简介

- 设备驱动功能
- 设备分类
- 设备文件和设备号
- 代码分布

设备驱动功能

- 设备驱动程序是内核的一部分
  - 对设备的初始化和释放
  - 把数据从内核传到硬件/从硬件读数据到内核
  - 读取应用程序传送给设备文件的数据和回送应用程序请求的数据。这需要在用户空间，内核空间，总线以及外设之间传输数据
  - 检测和处理设备出现的错误

设备分类

- Linux支持三类硬件设备
  - 字符设备：无需缓冲直接读写
  - 块设备
    - 通过buffer或cache进行读写
    - 支持随机访问
  - 网络设备：通过BSD套接口访问

设备文件和设备号

- 设备文件
  - Linux抽象了对硬件设备的访问，可作为普通文件一样访问，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
- 每个设备文件都对应有两个设备号
  - 主设备号，标识设备的种类，使用的驱动程序
  - 次设备号，标识使用同一设备驱动程序的不同硬件设备

代码分布

- Drivers
  - block
  - char
  - cdrom
  - pci
  - scsi
  - net
  - sound

Linux驱动相关内核机制

- 设备模型概述
- sysfs
- 设备驱动模型
- 关键数据结构
- 同步机制
- 工作队列
- 异步IO

设备模型概述

- 设备模型提供独立的机制表示设备及其在系统中的拓扑结构
  - 代码重复最小
  - 提供如引用计数这样的统一管理机制
  - 例举系统中所有设备，观察其状态，查看其连接总线
  - 用树的形式将全部设备结构完整、有效地展现，包括所有总线和内部连接
  - 将设备和对应驱动联系起来
  - 将设备按类型分类
  - 从树的叶子向根的方向依次遍历，确保以正确顺序关闭各个设备的电源

sysfs（1）

- sysfs是设备拓扑结构的文件系统表现
  - block：每个子目录分别对应系统中的一个块设备，每个目录又都包含该块设备的所有分区
  - bus：内核设备按总线类型分层放置的目录结构， devices中的所有设备都是连接于某种总线之下可以找到每一个具体设备的符号链接
  - class：包含以高层功能逻辑组织起来的系统设备视图
  - dev：维护一个按字符设备和块设备的主次号码（major/minor）链接到真实的设备（devices下）的符号链接

sysfs（2）

- devices：系统设备拓扑结构视图，直接映射出内核中设备结构体的组织层次
- firmware：包含一些如ACPI，EDD，EFI等底层子系统的特殊树
- fs：存放的已挂载点，但目前只有fuse，gfs2等少数文件系统支持sysfs接口，传统的虚拟文件系统（VFS）层次控制参数仍然在sysctl（/proc/sys/fs）接口
- kernel：新式的slab分配器等几项较新的设计在使用它，其它内核可调整参数仍然位于sysctl（/proc/sys/kernel）接口中
- module：系统中所有模块的信息，不管这些模块是以内联（inlined）方式编译到内核映像文件（vmlinuz）还是编译到外部模块（ko文件）

设备模型基本元素

- 设备类结构classes
- 总线结构bus
- 设备结构devices
- 驱动结构drivers

Linux统一设备模型的基本结构

类型	说明	对应内核数据结构	对应/sys项
总线类型（Bus Types）	系统中用于连接设备的总线	struct bus_type	/sys/bus/*/
设备（Devices）	内核识别的所有设备，依照连接它们的总线进行组织	struct device	/sys/devices/*/*/../
设备类别（Device Classes）	系统中设备的类型（声卡，网卡，显卡，输入设备等），同一类中包含的设备可能连接不同的总线	struct class	/sys/class/*/
设备驱动（Device Drivers）	在一个系统中安装多个相同设备，只需要一份驱动程序的支持	struct device_driver	/sys/bus/pic/drivers/*/

关键数据结构——kobject

```
struct kobject{
        const char *name;/*短名字*/
        struct kobject *parent;/*表示对象的层次关系*/
        struct sysfs_dirent  *sd;/*表示sysfs中的一个目录项*/
        struct kref kref;/*提供一个统一的计数系统*/
        struct list_head entry;
        struct kset *kset;
        struct kobj_type *ktype;
};
```

关键数据结构——kref

- 定义

```
struct kref{
        atomic_t refcount;
}
```

- 初始化

```
void kref_init(struct kref *kref)
```

- 增加计数

```
struct kobject* kobject_get(struct kobject *kobj)
```

- 减少计数

```
void kobject_put(struct kobject *kobj)
```

关键数据结构——ktype

```
struct kobj_type{
        void (*release)(struct kobject *kobj);  /*析构函数*/
        struct sysfs_ops *sysfs_ops;
                /*包含对属性进行操作的读写函数的结构体*/
        struct attribute **default_attrs;
                /*定义了kobject相关的默认属性*/
}
```

关键数据结构——kset

```
struct kset{
        struct list_head list;
                /*在该kset下的所有kobject对象*/
        spinlock_t list_lock;
                /*在kobject上进行迭代时用到的锁*/
        struct kobject kobj;
        /*该指针指向的kobject对象代表了该集合的基类*/
        struct kset_uevent_ops *uevent_ops; /*指向一个用于处理集合中kobject对象的热插拔结构操作的结构体*/
}
```

同步机制

- 同步锁
- 信号量
- 读写信号量
- 原子操作
- 完成事件（completion）

同步锁 — 自旋锁Spinlock（1）

- 自旋锁被别的执行单元保持，调用者就一直循环，看是否该自旋锁的保持者已经释放了锁。

- 自旋锁和互斥锁的区别是，自旋锁不会引起调用者睡眠，自旋锁使用者一般保持锁事件非常短，所以选择自旋而不是睡眠，效率会高于互斥锁。

同步锁 — 自旋锁Spinlock（2）

- **自旋锁的类型** spinlock_t
- **初始化**

  spinlock_t my_spinlock = SPIN_LOCK_UNLOCKED;

  void spin_lock_init(spinlock_t *lock);
- **获得锁** void spin_lock(spinlock_t *lock);
- **释放锁** void spin_unlock(spinlock_t *lock);
- **非阻塞版本**

  int spin_trylock(spinlock_t *lock);

  int spin_trylock_bh(spinlock_t *lock);

同步锁 — 读写锁（1）

- rwlock是内核提供的一个自旋锁的读者/写者形式。
- 读者/写者形式的锁允许任意数目的读者同时进入临界区，但是写者必须是排他的。
- 读写锁类型是rwlock_t，在<linux/spinlock.h>

同步锁 — 读写锁（2）

- 初始化

```
rwlock_t my_rwlock=RW_LOCK_UNLOCKED;/*静态*/
rwlock_init(&my_rwlock); /* 动态 */
```

- 读者锁的获取和释放

```
void read_lock(rwlock_t *lock);
void read_unlock(rwlock_t *lock);
```

- 写者锁的获取和释放

```
void write_lock(rwlock_t *lock);
int write_trylock(rwlock_t *lock);
void write_unlock_bh(rwlock_t *lock);
```

同步锁 — RCU锁（1）

- RCU（Read-Copy Update）锁机制是Linux2.6内核中新的锁机制
- 高性能的锁机制RCU克服了获得锁的开销和访问内存速度挂钩的问题
- RCU是改进的读写锁
  - 读者锁基本上没有同步开销，不需要锁，不使用原子指令，死锁问题也不用考虑
  - 写者锁同步开销相对较大，因为需要延迟数据结构的释放，复制被修改的数据结构，也必须用某种锁机制同步并行的其它写者的修改操作

同步锁 — RCU锁（2）

- 加锁

  rcu_read_lock()

- 释放锁

  rcu_read_unlock()

- 同步RCU锁

  synchronize_rcu()

此函数由RCU写端调用，会阻塞写者，直到所有读者完成读端临界区，写者才能进行后续操作。

同步锁 — seqlock（1）

- seqlock是2.6内核包含的一对新机制，能够快速地、无锁地存取一个共享资源
- seqlock实现原理是依赖一个序列计数器
  - 当写者写入数据的时候，会得到一把锁，并且把序列值增加1。当读者读取数据之前和之后，这个序列号都会被读取，如果两次读取的序列号相同，则说明写没有发生
  - 如果表明发生过写事件，则放弃已经进行的操作，重新循环一次，一直到成功。

同步锁 — seqlock（2）

- 读者锁

```
unsigned int seq;
do {
        seq = read_seqbegin(&the_lock);
        /* Do what you need to do */
} while read_seqretry(&the_lock, seq);
```

- 写者锁由自旋锁实现

```
void write_seqlock(seqlock_t *lock);
void write_sequnlock(seqlock_t *lock);
```

信号量（1）

- Linux内核的信号量在概念和原理上与用户态的IPC机制信号量是一样的，是一种睡眠锁
  - 当一个任务试图获得已被占用的信号量时，会进入一个等待队列，然后睡眠
  - 当持有该信号量的进程释放信号量后，位于等待队列的一个任务就会被唤醒，这个任务获得信号量
- 信号量不会禁止内核抢占
- 信号量适用于锁会被长期持有的情况，而自旋锁比较适合被短期持有
- 信号量允许有多个持有者，而自旋锁在任何时候只能有一个持有者

信号量（2）

- 宏声明一个信号量

  DECLARE_MUTEX(name)//信号量name并初始化它的值为0

  DECLARE_MUTEX_LOCKED(name)　　//互斥锁name，但把它的初始值设置为0，即锁在创建时就处在已锁状态

- 声明和初始化互斥锁

  void sema_init(struct semaphore *sem, int val);　　//初始化设置信号量的初值，它设置信号量sem的值为val

  void init_MUTEX(struct semaphore *sem);　//初始化一个互斥锁，即它把信号量sem的值设置为1

  void init_MUTEX_LOCKED(struct semaphore *sem);　　//初始化一个互斥锁，但它把信号量sem的值设置为0，即一开始就处在已锁状态

信号量（3）

- 获得/释放信号量

  void down(struct semaphore *sem);　　//获得信号量sem，它会导致睡眠，因此不能在中断上下文（包括IRQ上下文和softirq上下文）使用该函数。该函数将把sem的值减1，如果信号量sem的值非负，就直接返回，否则调用者将被挂起，直到别的任务释放该信号量才能继续运行。

  int down_interruptible(struct semaphore * sem);　　//down不会被信号（signal）打断，但down_interruptible能被信号打断，因此该函数有返回值来区分是正常返回还是被信号中断，如果返回0，表示获得信号量正常返回，如果被信号打断，返回-EINTR

  int down_trylock(struct semaphore *sem);　　//试着获得信号量sem，如果能够立刻获得，它就获得该信号量并返回0，否则，表示不能获得信号量sem，返回值为非0值。

  void up(struct semaphore *sem);　//释放信号量sem，即把sem的值加1，如果sem的值为非正数，表明有任务等待该信号量，因此唤醒这些等待者。

信号量（4）

- 在内核源码树的kernel/printk.c中，使用宏DECLARE_MUTEX声明了一个互斥锁console_sem，它用于保护console驱动列表console_drivers以及同步对整个console驱动系统的访问

- 其中定义了函数acquire_console_sem来获得互斥锁 console_sem，定义了release_console_sem来释放互斥锁console_sem，定义了函数try_acquire_console_sem来尽力得到互斥锁console_sem。这三个函数实际上是分别对函数down，up和 down_trylock的简单包装

- 需要访问console_drivers驱动列表时就需要使用acquire_console_sem来保护console_drivers列表，当访问完该列表后，就调用release_console_sem释放信号量console_sem

读写信号量（1）

- 读写信号量的访问者被细分为两类，一种是读者，另一种是写者。
  - 读者在拥有读写信号量期间，对该读写信号量保护的共享资源只能进行读访问
  - 如果某个任务同时需要读和写，则被归类为写者，它在对共享资源访问之前须先获得写者身份，写者在不需要写访问的情况下将被降级为读者。
- 可以有任意多个读者同时拥有一个读写信号量。
- 写者具有排他性和独占性

读写信号量（2）

- 声明和初始化

  DELARE_RWSEM(sem)　　//声明一个读写信号量name并对其进行初始化

  void init_rwsem(struct rm_semaphore *sem);　//对读写信号量sem进行初始化

- 读操作

  void down_read(struct rw_semaphore *sem);　　//得到读写信号量sem。该函数会导致调用者睡眠，因此只能在进程上下文使用。

  int down_read_trylock(struct rw_semaphore *sem);　//类似于down_read，只是它不会导致调用者睡眠。它尽力得到读写信号量sem，如果能够立即得到，它就得到该读写信号量，并且返回1，否则表示不能立刻得到该信号量，返回0。

  void up_read(struct rw_semaphore *sem);　　//释放读写信号量sem。它与down_read或down_read_trylock配对使用。如果down_read_trylock返回0，不需要调用up_read来释放读写信号量，因为根本就没有获得信号量。

读写信号量（3）

- 写操作

  void down_write(struct rw_semaphore *sem)； //得到读写信号量sem，它也会导致调用者睡眠，因此只能在进程上下文使用。

  int down_write_trylock(struct rw_semaphore *sem)；//类似于down_write，只是它不会导致调用者睡眠。该函数尽力得到读写信号量，如果能够立刻获得，就获得该读写信号量并且返回1，否则表示无法立刻获得，返回0。

  void up_write(struct rw_semaphore *sem)； //释放信号量sem。它与down_write或down_write_trylock配对使用。如果down_write_trylock返回0，不需要调用up_write，因为返回0表示没有获得该读写信号量。

- 写者降级为读者

  void downgrade_write(struct rw_semaphore *sem)； //把写者降级为读者。写者是排他性的。对于那些当前条件下不需要写访问的写者，降级为读者，使得等待访问的读者能够立刻访问，从而增加了并发性，提高了效率。

读写信号量（4）

- 在Linux中，每一个进程结构体中，类型为struct mm_struct的mmap字段维护了整个进程的内存块列表，该列表被大量地遍历或修改。

- 因此mm_struct结构就有一个字段mmap_sem来对mmap的访问进行保护，mmap_sem就是一个读写信号量，在proc文件系统里有很多进程内存使用情况的接口，通过它们能够查看某一进程的内存使用情况，命令 free、ps和top都是通过proc来得到内存使用信息的，proc接口就使用down_read和up_read来读取进程的mmap信息。

- 当进程动态地分配或释放内存时，需要修改mmap来反映分配或释放后的内存映像，因此动态内存分配或释放操作需要以写者身份获得读写信号量mmap_sem来对mmap进行更新。系统调用brk和munmap就使用了down_write和up_write来保护对mmap的访问。

原子操作（1）

- 原子操作是指该操作在执行完毕前绝不会被任何其他任务或时间打断，是最小的执行单位
- 原子操作和架构有关，需要硬件的支持，它的API和原子类型的定义都在内核源码树include/asm/atomic.h文件中，使用汇编语言实现
- 原子操作主要用在资源计数，很多应用计数（refcnt）就是通过原子操作实现的

原子操作（2）

- 原子操作类型的定义

```
typedef struct {volatile int counter;}atomic_t;
```

- 原子操作API

```
atomic_read(atomic_t *v);
atomic_set(atomic_t *v,int i);
void atomic_add(int I, atomic_t *v);
void atomic_sub(int I,atomic_t *v);
void atomic_inc(atomic_t *v);
void atomic_dec(atomic_t *v);
```

原子操作（2）

```
int atomic_sub_and_test(int i, atomic_t *v);
int atomic_dec_and_test(atomic_t *v);
int atomic_inc_and_test(atomic_t *v);

int atomic_add_negative(int i, atomic_t *v);
int atomic_add_return(int i, atomic_t *v);
int atomic_sub_return(int i, atomic_t *v);
int atomic_inc_return(atomic_t * v);
int atomic_dec_return(atomic_t * v);
```

原子操作（3）

- 在TCP/IP协议栈的IP碎片处理中，使用了引用计数，碎片队列结构struct ipq描述了一个IP碎片，字段refcnt就是引用计数器，它的类型为atomic_t，当创建IP碎片时（在函数ip_frag_create中），使用atomic_set函数把它设置为1，当引用该IP碎片时，就使用函数atomic_inc把引用计数加1。

- 当不需要引用该IP碎片时，就使用函数ipq_put来释放该IP碎片，ipq_put使用函数atomic_dec_and_test把引用计数减1并判断引用计数是否为0，如果是就释放IP碎片。函数ipq_kill把IP碎片从ipq队列中删除，并把该删除的IP碎片的引用计数减1（通过使用函数atomic_dec实现）。

完成事件（1）

- 完成事件是一种简单的同步机制，表示“things may proceed”，适用于需要睡眠和唤醒的情景。如果要在任务中实现简单睡眠直到其它进程完成某些处理过程为止，可采用完成事件，不会引起资源竞争

- 如果要使用completion，需要包含<linux/completion.h>，同时创建类型为struct completion的变量

完成事件（2）

- 静态声明和初始化

```
DECLARE_COMPLETION(my_completion);
```

- 动态声明和初始化

```
struct completion my_completion;
init_completion(&my_compleiton);
```

- 等待某个过程的完成

```
void wait_for_completion(struct completion *comp);
```

- 唤醒等待该事件的进程

```
void complete(struct completion *comp);
void complete_all(struct completion *comp);
```

工作队列（1）

- 2.6内核开始引入工作队列，是Linux将工作推后执行的机制
- 工作队列把推后的工作交给内核线程去执行，允许重新调度甚至睡眠。
- 数据结构：

```
struct work_struct {
        atomic_long_t data;
        struct list_head entry;
        work_func_t func;
};
```

工作队列（2）

- 初始化

```
INIT_WORK(_work, _func, _data)
```

- 使用

```
int schedule_work(struct work_struct *work);
int schedule_delayed_work(struct work_struct *work, unsigned long delay);
void flush_scheduled_work(void);
int cancel_delayed_work(struct work_struct *work);
```

- 创建和调度工作队列

```
struct workqueue_struct *create_workqueue(const char *name);
int queue_work(struct workqueue_struct *wq, struct work_struct *work);
...
```

异步I/O

- 异步事件非阻塞I/O，也叫做异步I/O(AIO)
  - 用户程序可以通过向内核发出I/O请求命令，不用等带I/O事件真正发生，可以继续做另外的事情，等I/O操作完成，内核会通过函数回调或者信号机制通知用户进程
  - 很大程度提高了系统吞吐量
- 块设备和网络设备驱动的操作全是异步的，但是对于字符型设备，需要在驱动程序中实现对应的异步函数，才能实现异步操作
- 要使用AIO功能，需要包含头文件aio.h

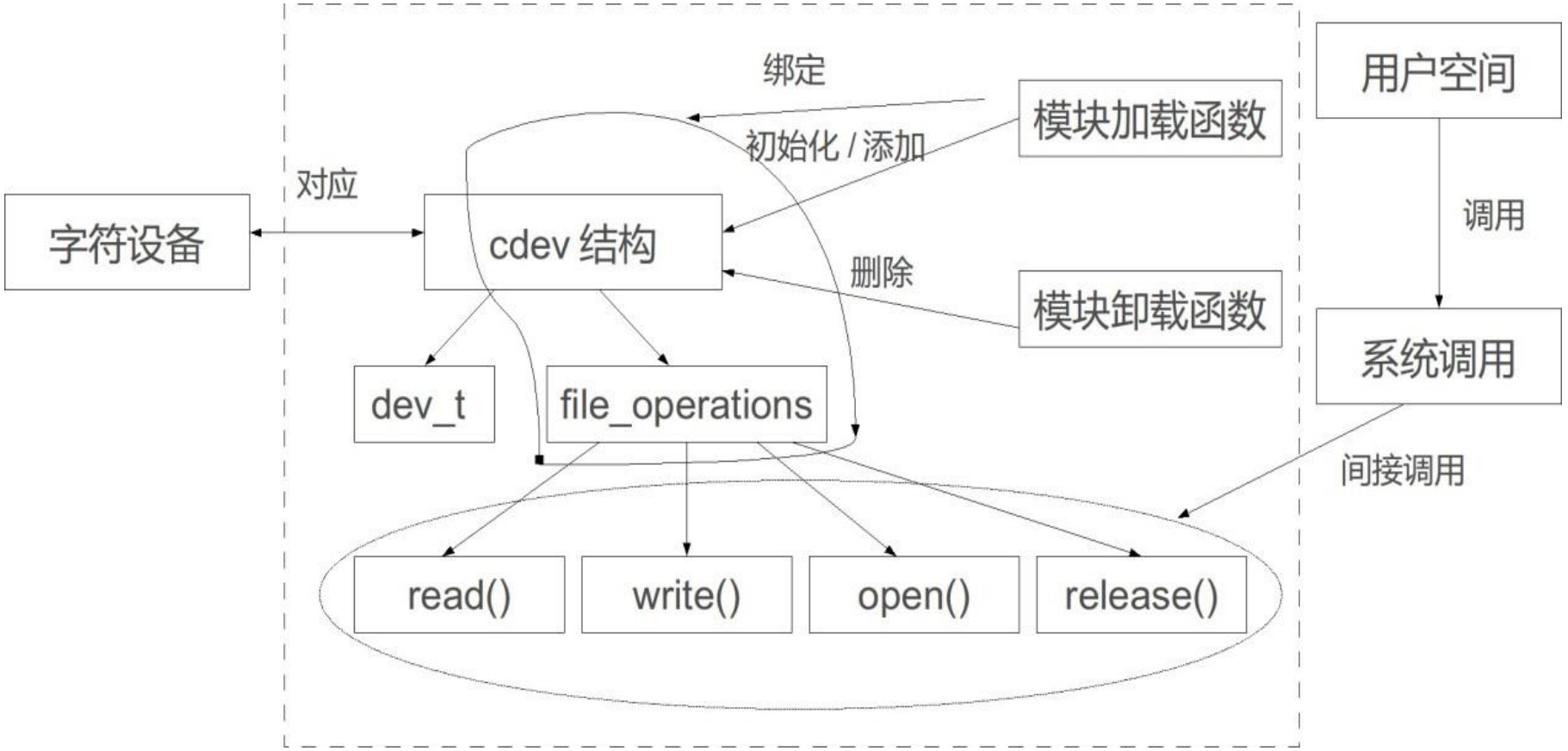
Linux字符设备及驱动设计

- 字符设备驱动框架
- 字符设备驱动开发
- GPIO驱动概述
- 串行总线概述
- I^2C总线驱动开发

字符设备驱动框架（1）

- 字符设备驱动程序是嵌入式Linux最基本、也是最常用的驱动程序
- 字符设备在Linux内核中使用struct cdev结构来表示
- 在struct cdev结构中包含着字符设备需要的全部信息
  - 设备号dev_t
  - 文件操作file_operations

字符设备驱动框架（2）


绑定
初始化 / 添加
对应
删除
字符设备
cdev 结构
模块加载函数
模块卸载函数
用户空间
调用
系统调用
间接调用
dev_t
file_operations
read()
write()
open()
release()

字符设备驱动开发

- 设备类型和设备号
- 关键数据结构
- 字符设备的注册和注销

设备类型和设备号

- 对字符设备的访问是通过文件系统内的设备文件进行的，或者称为设备节点，位于/dev目录
- 设备类型：字符设备/块设备
- 主设备号：主设备号用来标识该设备的种类，也标识了该设备所使用的驱动程序
- 次设备号：次设备号由内核使用，标识使用同一设备驱动程序的不同硬件设备

关键数据结构

- file_operations
- file
- inode

file_operations（1）

```
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（2）

```
long (*unlocked_ioctl) (struct file *, unsigned int, unsigned long);
//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
int (*fasync) (int, struct file *, int);
//用来实现文件加锁，通常设备文件不需要实现此函数
int (*lock) (struct file *, int, struct file_lock *);
}
```

file

- file结构体在<linux/fs.h>中定义
  - fmode_t f_mode对文件的读写模式，对应系统调用open的mod_t mode参数
  - loff_t f_pos表示文件当前的读写位置
  - unsigned int f_flags表示文件标志，对应系统调用open的int flags参数
  - const struct file_operations *f_op指向和文件关联的操作
  - void *private_data可以用于保存系统调用之间的信息

inode

- inode是内核文件系统索引节点对象，包含内核在操作文件或目录时需要的全部信息
- 在内核中inode结构体用来表示文件，file是表示打开文件的结构体
- 重要成员
  - 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:指向cdev结构的指针

字符设备的注册和注销

定义

```
struct cdev {
    struct kobject kobj;   /*内嵌的kobject对象*/
    struct module *owner;   /*所属模块*/
    const struct file_operations *ops;  /*文件操作函数*/
    struct list_head list;
    dev_t dev;         /*设备号*/
    unsigned int count;
};
```

使用

```
void cdev_init(struct cdev *, const struct file_operations *);
struct cdev *cdev_alloc(void);
void cdev_add(struct cdev *, dev_t, unsigned);
void cdev_del(struct cdev *);
```

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

串行总线概述

- 串行相比于并行的主要优点是要求的线数较少，通常只需要使用2条、3条或4条数据/时钟总线连续传输数据
  - RS232
  - RS485
  - I^2C
  - SPI
  - SMBUS

I²C总线

- Philips公司开发的二线式串行总线标准，内部集成电路（Internal Integrated Circuit），主要用于连接微控制器和外围设备。
  - 在标准模式下，位速率可以达到100Kbit/s
  - 在快速模式下则是400Kbit/s
  - 在高速模式下可以达到3.4Mbit/s
- I²C总线是由串行数据信号线SDA和串行时钟信号线SCL构成的串行总线，可发送和接收数据。
- 采用该总线连接的设备工作在主/从模式下，主器件既可以作为发送器，也可以作为接收器
- I²C总线最主要的特点是它的简单性和高效性。

SPI总线

- 同步外设接口（Serial Peripheral Interface，SPI）是由摩托罗拉公司推出的一种高速的、全双工、同步的串行总线
- SPI接口在CPU和外围低速器件之间进行同步的串行数据传输，在主器件的移位脉冲下，数据按位传输，并且高位在前、低位在后，是一种全双工通信。数据传输速度总体上来说比I^2C总线要快，速度可以达到几Mbit/s
- SPI 的工作模式有两种：主模式和从模式，无论那种模式，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
- 接口一般使用四条线：串行时钟线SCK、主器件输入/从器件输出数据线MISO、主器件输出/从器件输入数据线MOSI和从器件选择线SS

SMBus总线

- 系统管理总线（System Management Bus，SMBus）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（SMBDATA）和一条时钟线（SMBCLK）进行通信
- SMBus的目标是通过一条廉价但功能强大的总线，来控制主板上的设备和收集设备的信息
- 虽然SMBus的数据传输率较慢，只有大约100kbit/s，却以其结构简单、造价低的特点，受到业界的普遍欢迎
- SMBus总线大部分基于I^2C总线规范，许多I^2C设备也能够在SMBus上正常工作。SMBus与I^2C总线之间在时序特性上存在一些差别

I²C总线驱动开发

- I²C驱动程序概况
- 关键数据结构
- I²C核心
- I²C总线驱动
- I²C设备驱动

I²C总线驱动概况

- Linux下的I²C驱动架构有相当的复杂度，主要由I²C核心、I²C总线驱动以及I²C设备驱动三个部分组成。
  - algos：包含了一些I²C总线适配器的algorithm实现
  - busses：包含了一些I²C总线的驱动，例如AT91的i2c-at91.c
  - chips：包含了一些I²C设备的驱动，例如Dallas公司的DS1682实时钟芯片
  - i2c-boardinfo.c：包含了一些板级信息
  - i2c-core.c/i2c.core.h：实现了I²C核心的功能以及/proc/bus/i2c*接口
  - i2c-dev.c：这是一个通用的驱动，大多数I²C驱动都可以通过调用它操作

关键数据结构

- i2c_adapter
- i2c_algorithm
- i2c_driver
- i2c_cflient

i2c_adapter

```
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;  /*I²C 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_algorithm

```
struct i2c_algorithm {
        //I²C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 *);
};

struct i2c_msg {
        __u16 addr;          /*从设备地址*/
        __u16 flags;     /*标志位*/
        __u16 len;                  /*消息长度*/
        __u8 *buf;                  /*消息内容*/
};
```

i2c_client

```
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

```
struct i2c_driver {
        int id;         /*唯一的驱动id*/
        unsigned int class;
        int (*attach_adapter)(struct i2c_adapter *);    /*适配器添加函数（旧式）*/
        int (*detach_adapter)(struct i2c_adapter *);    /*适配器删除函数（旧式）*/
        int (*detach_client)(struct i2c_client *) __deprecated;  /*设备删除函数（旧式）*/
        int (*probe)(struct i2c_client *, const struct i2c_device_id *);
                /*设备添加函数（新式）*/
        int (*remove)(struct i2c_client *);          /*设备删除函数（新式）*/
        void (*shutdown)(struct i2c_client *);       /*设备关闭函数*/
        int (*suspend)(struct i2c_client *, pm_message_t mesg);  /*设备挂起函数*/
        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*/
        const struct i2c_device_id *id_table;        /*此驱动支持的I²C设备列表*/
        int (*detect)(struct i2c_client *, int kind, struct i2c_board_info *);  /*检测函数*/
        const struct i2c_client_address_data *address_data;
        struct list_head clients;                    /*链表头*/
};
```

I²C核心

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。

```
int i2c_add_adapter(struct i2c_adapter *adapter);
int i2c_del_adapter(struct i2c_adapter *adapter);
int i2c_register_driver(struct module *owner, struct i2c_driver *driver);
void i2c_del_driver(struct i2c_driver *driver);
int  i2c_attach_client(struct i2c_client *);
int  i2c_detach_client(struct i2c_client *);
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

- I²C总线驱动的任务，是为系统中各个I²C总线增加相应的读写方法
- I²C总线驱动模块的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，然后通过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
- 当I²C总线驱动模块被卸载时，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
- 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。主要是实现其中的master_xfer() 函数和functionality() 函数

I²C设备驱动（1）

- 与I²C总线驱动对应的是I²C设备驱动，I²C只有总线驱动是不够的，必须有设备才能正常工作
- 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。 drivers/i2c/chips目录下已经包含了部分设备的设备驱动代码，负责相应从设备的注册
- i2c-dev.c文件中提供了一个通用的I²C设备驱动程序，实现了字符设备的文件操作接口，对设备的具体访问是通过I²C适配器来实现的。构造一个针对I²C核心层接口的数据结构，即i2c_driver结构体，通过接口函数向 I²C核心注册一个I²C设备驱动。同时构造一个对用户层接口的数据结构，并通过接口函数向内核注册一个主设备号为89的字符设备

I²C设备驱动（2）

- 该文件提供了用户层对I²C设备的访问，包括 open、release、read、write、ioctl等常规文件操作，应用程序可以通过open函数打开 I²C的设备文件，通过ioctl函数设定要访问从设备的地址，然后就可以通过read和write函数完成对I²C设备的读写操作。
- i2c-dev.c中提供的i2cdev_read() 和 i2cdev_write() 函数分别实现了用户空间的read和write操作，这两个函数又分别会调用I²C核心的i2c_master_recv() 和 i2c_master_send() 函数来构造一条I²C消息，并且最终调用i2c_algorithm提供的函数接口来完成消息传输

□谢　谢！