“菜鸟玩转嵌入式”视频培训讲座
— Linux驱动开发基础班

主办：上海申嵌信息科技有限公司
承办：嵌入式家园
协办：上海嵌入式家园-开发板商城
广州友善之臂计算机科技有限公司
主讲：贺光辉（嵌入式系统工程师）

嵌入式家园 www.embedclub.com

上海嵌入式家园-开发板商城 http://embedclub.taobao.com/

第五章

中断与时钟

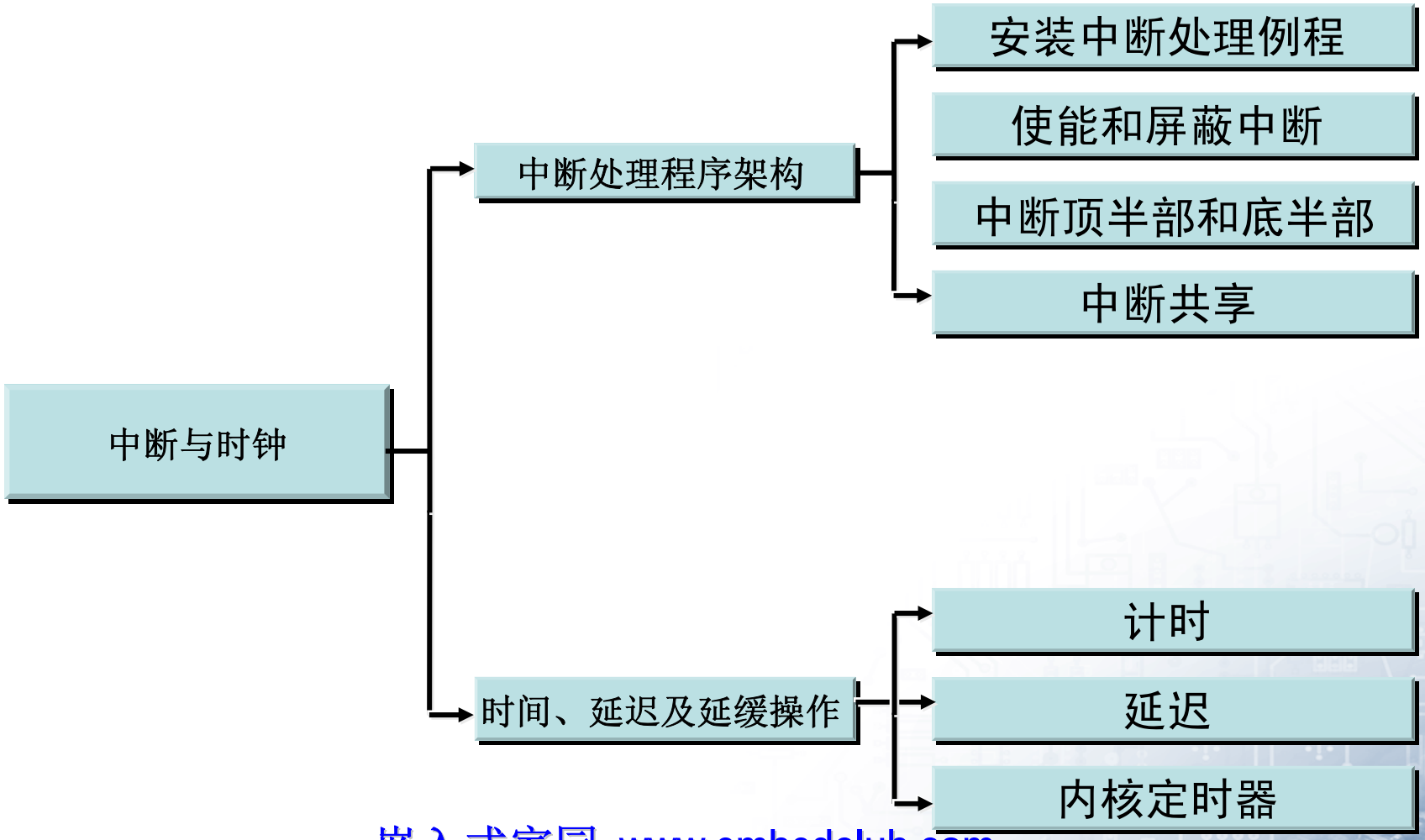
预习检查

- 简要说明中断处理流程
- 列举常用中断源

本章目标

- 掌握设备驱动中中断处理例程的实现方法
- 掌握内核中实现计时、延迟操作的函数

本章结构



5-1 中断处理程序架构

- 什么是中断？
  - 中断是指CPU在执行程序的过程中，出现了某些突发事件
  - CPU必须暂停执行当前程序，转去处理突发事件
  - 处理完毕后CPU又返回原程序被中断的位置并继续执行。
- 中断分类
  - 按中断来源分类：
    - 内部中断
    - 外部中断
  - 按中断是否可屏蔽分类：
    - 可屏蔽中断
    - 不可屏蔽中断(NMI)
  - 按中断入口跳转方法的不同分类
    - 向量中断
    - 非向量中断

5-1-1 安装中断处理例程

- 申请和释放IRQ

要申请的中断号

要安装的中断处理函数指针

```
int request_irq(unsigned int irq,
                irqreturn_t (*handler)(int，void *，struct pt_regs *)，
                unsigned long flags，
                const char *dev_name，
                void *dev_id);
void free_irq(unsigned int irq，void *dev_id);
```

用于共享的中断数据线。它是用来唯一的标识设备。

flags:

0：普通外部中断

SA_INTERRUPT： 快速中断

SA_SHIRQ： 共享中断

5-1-1 安装中断处理例程

申嵌

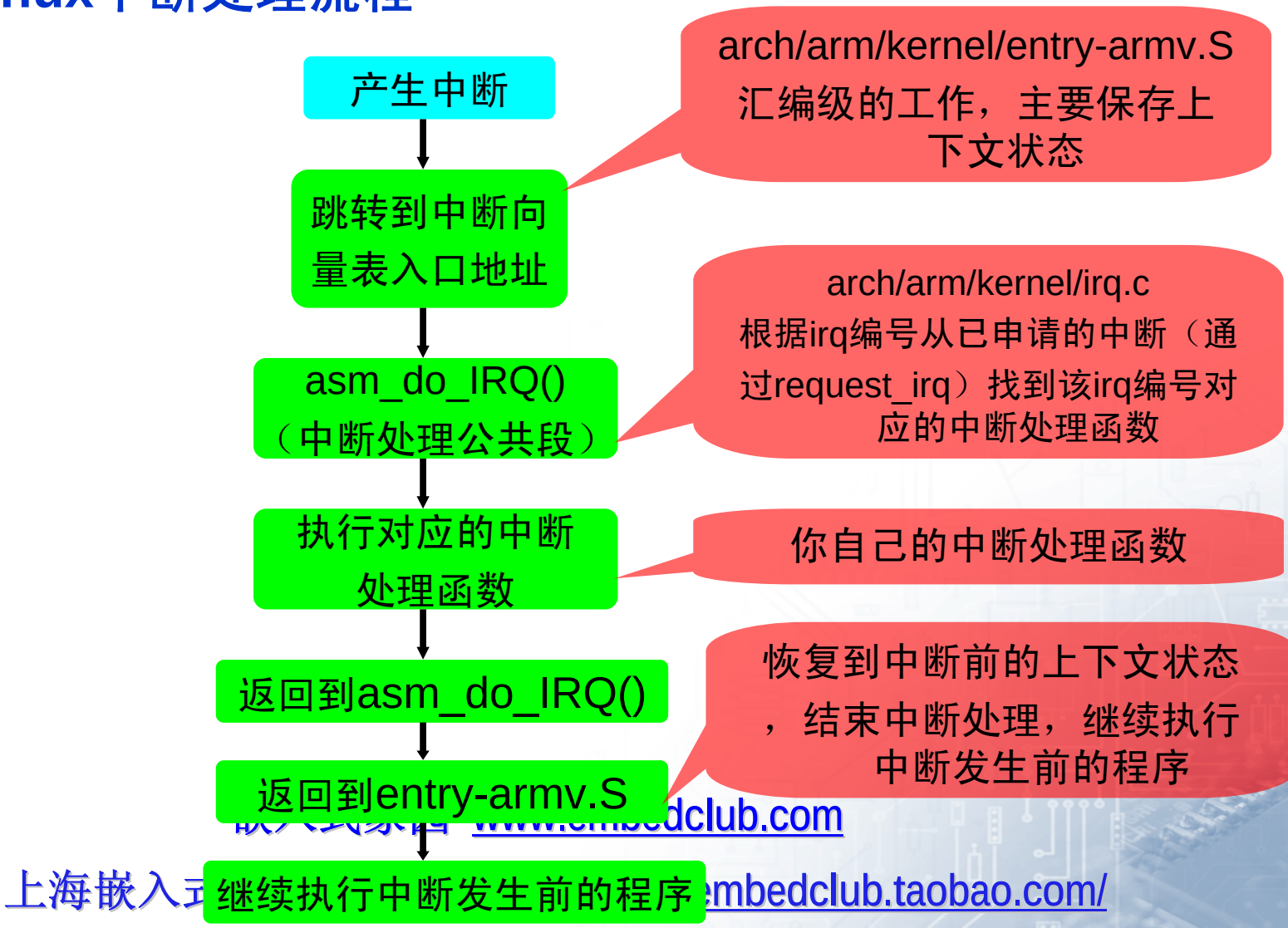
- 申请和释放IRQ实例

```
/*中断处理函数*/
 irqreturn_t
xxx_interrupt(int irq,
              void *dev_id,
              struct pt_regs *regs)
{
      …
      /*中断处理的具体内容*/
      …
}
```

```
/*设备驱动模块加载函数*/
 int __init xxx_init(void)
 {
   …
   /*申请中断*/
   result = request_irq(xxx_irq,
                        xxx_interrupt,
                        SA_INTERRUPT,
                        “xxx”,
                        NULL);
    …
}
/*设备驱动模块卸载函数*/
void __exit xxx_exit(void)
 {
          …
          /*释放中断*/
          free_irq(xxx_irq,NULL);
          …
}
```

5-1-1 安装中断处理例程

- Linux中断处理流程



5.1.2 使能和屏蔽中断

- 禁用/使能单个中断

等待目前的中断处理完成，禁用该IRQ

```
void disable_irq(int irq);
void disable_irq_nosync(int irq);
void enable_irq(int irq);
```

禁用并立即返回

- 禁用/使能所有的中断

把中断状态保存到flags中，然后禁用。

```
void local_irq_disable(void);
void local_irq_enable(void);

void local_irq_save(unsigned long flags);
void local_irq_restore(unsigned long flags);
```

5-1-3 中断顶半部和底半部

- 为什么将中断处理程序分成顶半部和底半部?
  - 需求：
    - 中断处理要求尽快结束，而不能使中断阻塞的时间过长
    - 而有些处理例程确实要完成耗时的任务
  - 解决方案
    - 顶半部，让中断阻塞尽可能的短
    - 底半部，完成耗时的任务

5-1-3 中断顶半部和底半部

- **顶半部：**
  - 是实际响应中断的例程
  - 用request_irq注册的中断例程
  - 它在很短的时间内完成

- **底半部：**
  - 被顶半部调度，并在稍后更安全的时间内执行的例程。

5-1-3 中断顶半部和底半部

- 实现底半部的机制
  - tasklet(任务队列)
  - Work queue(工作队列)

- 二者的区别
  - tasklet
    - 速度快，优先选择
    - 原子操作，运行于中断上下文
  - 工作队列
    - 高延迟
    - 允许休眠，运行于进程上下文

- **tasklet的使用**
  - 声明tasklet

底半部执行的函数

DECLARE_TASKLET(name, function, data)；
name:tasklet的名字
funciton:执行tasklet时调用的函数
data:一个用来传递给tasklet函数的unsigned long类型的值

例如：
DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);

  - 调度tasklet

```
void tasklet_schedule(struct tasklet_struct  *t)
```

- tasklet使用模板

```
/*定义tasklet和底半部函数并关联*/
 void xxx_do_tasklet(unsigned long);//底半部函数声明
 DECLARE_TASKLET(xxx_tasklet, xxx_do_tasklet, 0);

/*中断处理底半部*/
 void xxx_do_tasklet(unsigned long)
{
        …
 }
 /*中断处理顶半部*/
 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        …
        tasklet_schedule(&xxx_tasklet);
        …
 }
```

- 创建新的工作队列

```
struct workqueue_struct  *
create_workqueue(const char *name);

struct workqueue_struct  *
create_singlethread_workqueue(const char *name);
```

一个工作队列可对应多个“内核线程”

一个工作队列对应单个线程

5-1-3 中断顶半部和底半部

- 向工作队列提交任务，首先填充一个work_struct 结构

DECLARE_WORK(name，void(*function)(void *)，void *data)；

INIT_WORK(struct work_struct *work，void(*function)(void *)，void *data)；

不会初始化用来将work_struct结构链接到工作队列的指针，一般适用于任务已经提交，只是修改了任务时使用PREPARE_WORK。

PREPARE_WORK(struct work_struct *work，void(*function)(void *)，void *data)

5-1-3 中断顶半部和底半部

- 提交任务

```
int queue_work(struct workqueue_struct *queue,
               struct work_struct *work);

int queue_delayed_work(struct workqueue_struct *queue,
                       struct work_struct *work,
                       unsigned long delay);
```

实际的工作至少会在经过指定的jiffies(由delay指定)之后才会被执行

- 取消某个队列的入口项

```
int cancel_delayed_work(struct work_struct *work);
```

!= 0：内核会确保不会初始化给定入口项的执行

==0：则说明该入口项已经在其他处理器上运行

因此在cancel_delayed_work返回后可能仍在运行，怎么办？

5-1-3 中断顶半部和底半部

- 之后，需要强制刷新工作队列

```
void flush_workqueue(struct workqueue_struct *queue);
```

- 销毁工作队列

```
void destroy_workqueue(struct workqueue_struct *queue);
```

5-1-3 中断顶半部和底半部

- 共享工作队列
  - 在许多情况下，驱动不需要有自己的工作队列，只是偶尔地向工作队列添加任务
  - 使用内核提供的共享的默认工作队列
  - 不应该长期独占该队列，即不能长时间休眠
  - 我们的任务可能需要更长的时间延迟才能获得处理器时间

5-1-3 中断顶半部和底半部

- 使用共享队列
  - 初始化

```
INIT_WORK(struct work_struct *work，void(*function)(void *)，void *data)；
```

  - 调度

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

  - 取消共享工作队列的一个入口项（即一个工作任务work）

```
int cancel_delayed_work(struct work_struct *work);
```

  - 刷新共享工作队列

```
void flush_scheduled_work(void)；
```

- Work queue的使用模板

```
/*定义工作队列和关联函数*/
 struct work_struct xxx_wq;
 void xxx_do_work(unsigned long);
 /*中断处理底半部*/
 void xxx_do_work(unsigned long)
 {
        …
 }
 /*中断处理顶半部*/
 irqreturn_t xxx_interrupt(int irq, void *dev_id, struct pt_regs *regs)
{
        …
        schedule_work(&xxx_wq);
        …
 }
```

5-1-3 中断顶半部和底半部

```
/*设备驱动模块加载函数*/
int __init xxx_init(void)
{
        …
        /*申请中断*/
        result = request_irq(xxx_irq, xxx_interrupt,
                SA_INTERRUPT, “xxx”,NULL);
        …
        /*初始化工作队列*/
        INIT_WORK(&xxx_wq, (void (*)(void *)) xxx_do_work, NULL);
        …
}
```

5-1-4 中断共享

- Linux中断共享：多个设备使用同一个中断线号，同一个中断设备线号的所有处理程序链接成一个链表。
- 共享中断的多个设备在申请中断时都应使用SA_SHIRQ标志
- 设备结构指针可以作为request_irq（…, void *dev_id）的最后一个参数dev_id 传入， dev_id这个参数必须是唯一的，用来标识一个唯一的设备。
- 在中断到来时，对应链表的所有共享该中断的中断处理程序都会被执行，它们会检查dev_id参数信息，并根据硬件中断寄存器中的信息判断是否是本设备的中断，如果不是，应迅速返回，如果是，则处理完成，如果链表中没有一个是，则说明出现错误。

5-1-4 中断共享

- 共享中断使用模板

```
/*中断处理顶半部*/
irqreturn_t
xxx_interrupt(int irq, void *dev_id,
              struct pt_regs *regs)
{
    …
    int status = read_int_status();
    if(!is_myint(dev_int, status))
    {
        return IRQ_NONE;
    }
    …
    return IRQ_HANDLED;
}
```

获知中断源

判断是否是本设备中断

通知内核该中断不需要处理

通知内核处理该中断

```
/*设备驱动模块加载函数*/
int xxx_init(void)
{
        …
    /*申请共享中断*/
    result = request_irq(sh_irq,   xxx_interrupt,
                SA_SHIRQ,
                “xxx”,
                xxx_dev);
                …
}
/*设备驱动模块卸载函数*/
void __exit xxx_exit(void)
{
                …
                /*释放中断*/
                free_irq(xxx_irq,xxx_dev);
                …
}
```

阶段总结

- 介绍了Linux下中断的申请和释放
- 介绍了Linux中断底半部的主要处理方法:tasklet和工作队列
- 介绍了Linux下中断共享的实现方法

5-2 时间、延迟及延缓操作

- 计时
- 延迟
- 内核定时器

5-2-1 计时

- **内核时钟**
  - 内核通过定时器（timer）中断来跟踪时间流
  - 硬件定时器以周期性的间隔产生时钟中断，这个间隔（即频率）由内核根据HZ来确定，HZ是一个与体系结构无关的常数。
  - 这个时间间隔通常取1ms到10ms。
- **jiffies计算器**
  - 每次当定时器中断发生时，内核内部通过一个64位的变量jiffies_64做加一计数。
  - 驱动程序开发者通常访问的是jiffies变量，它是jiffies_64的低32位。
  - jiffies是unsigned long型的变量，该变量被声明为volatile，这样可避免编译器对访问该变量的语句的优化 。
  - jiffies记录了自最近一次Linux启动后到当前的时间间隔（即时钟中断发生的次数）。驱动程序常利用jiffies来计算不同事件间的时间间隔。
- **HZ**
  - HZ 即表示每秒钟产生的定时中断次数
  - HZ就代表1秒，HZ/2就代表0.5秒

- 使用jiffies计数器

```
#include <linux/jiffies.h>。
unsigned long current_j，stamp_1，stamp_half，stamp_n；
current_j = jiffies；          /*读取当前值*/
stamp_1 = current_j + HZ；     /*未来的1秒*/
stamp_half = current_j + HZ/2；  /*半秒*/
stamp_n = current_j + n*HZ/1000；/*n毫秒*/
```

- 为了防止因jiffies溢出导致问题，最好使用宏比较

判断a代表的时间是否在b时间之后

```
#include <linux/jiffies.h>
int time_after(unsigned long a，unsigned long b)；
int time_before(unsigned long a，unsigned long b)；
int time_after_eq(unsigned long a，unsigned long b)；
int time_before_eq(unsigned long a，unsigned long b)；
```

5-2-1 计时

1)用户空间的**timeval**：

```
struct timeval {
   time_t    tv_sec;   /* 秒 */
  suseconds_t   tv_usec; /* 毫秒 */
};
```

2)用户空间的**timespec**：

```
struct timespec {
   time_t     tv_sec;   /* 秒 */
   long   tv_nsec;    /* 纳秒 */
};
```

- **内核空间jiffies值和用户空间timeval、timespec之间转换**

```
#include<linux/time.h>
unsigned long timespec_to_jiffies(struct timespec *value);
void jiffies_to_timespec(unsigned long jiffie， struct timespec *value);
unsigned long timeval_to_jiffies(struct timeval *value);
void jiffies_to_timeval(unsigned long jiffies，struct timeval *value);
```

- 获取当前时间

```
#include <linux/time.h>
void do_gettimeofday(struct timeval *tv);

struct timespec current_kernel_time(void);
```

- 使用jiffies延迟
  - 如果对延迟的精度要求不高，最简单的实现方法如下 – 忙等待：

```
unsigned long j = jiffies + delay * HZ;
while (jiffies < j) {
        /*do nothing*/
}
```

5-2-1 延迟

- 长延迟

```
while(time_before(jiffies，end_time)){
        schedule()；
}
```

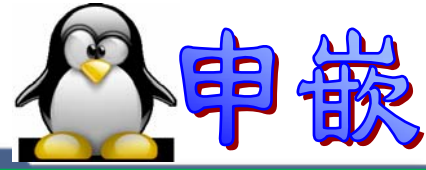
使用jiffies表示的延迟时间

```
#include<linux/sched.h>
signed long schedule_timeout(signed long timeout)；
```

典型应用

```
set_current_state(TASK_INTERRUPTIBLE)；
schedule_timeout(delay)；
```

5-2-1 延迟



- 短延迟（忙等待延时，不发生休眠）

这三个延迟函数均是忙等待函数，因而在延迟过程中无法运行其他任务。不发生休眠！

```
#include <linux/delay.h>
void ndelay(unsigned long nsecs);     /*延时纳秒*/
void udelay(unsigned long usecs);     /*延时微妙*/
void mdelay(unsigned long msecs);         /*延时毫秒*/
```

- 不使用忙等待的延迟方式（将调用进程休眠给定时间）：

不可中断休眠millisecs毫秒

```
#include <linux/delay.h>
void msleep(unsigned int millisecs);  //休眠millisecs毫秒
unsigned long msleep_interruptible(unsigned int millisecs);  //可中断休眠
void ssleep(unsigned int seconds);   //休眠seconds秒
```

5-2-3 内核定时器

- 定时器用于控制某个函数（定时器处理函数）在未来的某个特定时间执行。内核定时器注册的处理函数只执行一次 – 不是循环执行的。

5-2-3 内核定时器

- 内核定时器被组织成双向链表，并使用struct timer_list结构描述。

```
#include <linux/timer.h>
struct timer_list{
        /*…*/
        unsigned long expires；/*超时的jiffies值*/
        void (*function)(unsigned long)；/*超时处理函数*/
        unsigned long data；/*超时处理函数参数*/
};
```

5-2-3 内核定时器操作函数

- 初始化
  - 初始化定时器队列结构

```
void init_timer(struct timer_list *timer);
```

```
struct timer_list TIMER_INITIALIZER(_function, _expires, _data);
```

- 添加定时器
  - 启动定时器，开始倒计时

```
void add_timer(struct timer_list *timer);
```

- 删除定时器
  - 在定时器超时前将它删除。当定时器超时后，系统会自动地将它删除。

```
int del_timer(struct timer_list *timer);
```

5-2-3 内核定时器例程模板

- 内核定时器的使用模板

```
/*定义内核定时器对象*/
 static struct timer_list key_timer;

 /*定时器处理函数*/
 static void key_timer_handle(unsigned long data)
 {
        …
        /*定时器处理函数具体执行代码*/
        …
        /*定时器参数更新，重新启动定时器*/
        key_timer.expires = jiffies + KEY_TIMER_DELAY;
        add_timer(&key_timer);
 }
```

5-2-3 内核定时器例程模板

```
/*设备驱动模块加载函数*/
static int __init xxx_init(void)
{
        …
        /*初始化内核定时器*/
        init_timer(&key_timer);
        key_timer.function = &key_timer_handle;
        key_timer.data = (unsigned long)key_desc;
        key_timer.expires = jiffies + KEY_TIMER_DELAY;
        /*添加内核定时器*/
        add_timer(&key_timer);

        …
 }
static void __exit xxx_exit(void)
{
        …
        /*删除定时器*/
        del_timer(&key_timer);
}
```

5-2-3 内核定时器

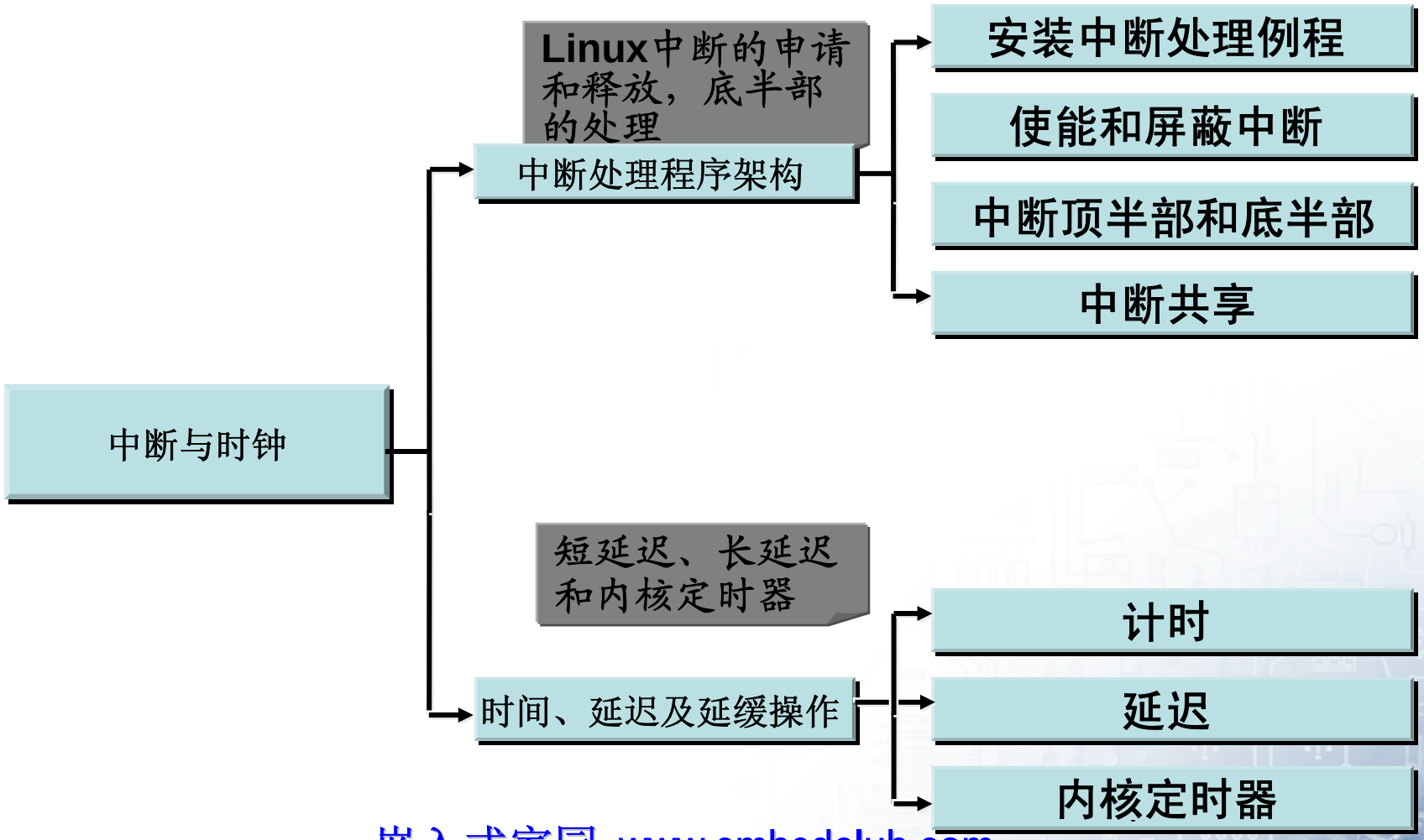
- 内核定时器与tasklet比较
  - 相同点
    - 在中断期间运行
    - 始终会在调度他们的同一cpu上运行。
    - 软件中断上下文，原子模式运行。
  - 不同点
    - 不能要求TASKLET在给定的时间执行

软件中断是打开硬件中断的同时执行某些异步任务的一种内核机制

阶段总结

- 内核的计时和延迟方法
- 内核定时器

本章总结



- 任务一、按键扫描驱动程序（通过外部中断）
- 任务二、按键扫描驱动程序（通过内核定时器）

六按键接口电路

6个按键中断引脚分布于CPU的不同部分，在此分别截图