1. **定时器**

1.内核时间相关转换函数

unsigned long usecs\_to\_jiffies(const unsigned int u)

功能： 把微秒转换成时钟节拍

参数： u 时间微秒

返回： 对应的时钟节拍数量

unsigned long msecs\_to\_jiffies(const unsigned int m)

功能： 把毫秒转换成时钟节拍

参数： m 时间毫秒

返回： 对应的时钟节拍数量

2.初始化定时器

1）静态初始化（=定义定时器+动态初始化+设置）

DEFINE\_TIMER(\_name, \_function, \_expires, \_data)

1

功能： 定义一个名字为\_name 的 struct timer\_list 结构的变量， 并且初始化它的 function, expires, data 成员。

2）动态初始化

init\_timer (struct timer\_list \*timer);

功能： 只是对 struct timer\_list 结构成员进行一些基础初始化操作， function, expires, data 成员还需要用户自己填充。

3）设置定时器

setup\_timer(struct timer\_list \*timer, void \*function, unsigned long data);

功能： 设置定时器中的 function, data 和一些基础成员， expires 并没有初始化， 需要用户自己进行初始化

参数：

timer： struct timer\_list 结构变量地址

function：超时函数的指针

expires：时间节拍数，可以通过上面的时间相关转换获得

data ：超时函数的参数

2.注册定时器到内核

void add\_timer(struct timer\_list \*timer)

功能： 向内核注册一个定时器， 注册后会马上开始计时。

参数： 是一个 struct timer\_list 结构变量地址， 并且要求这个结构变量是已经初始化好必须成员

3.修改定时器时间，重新注册

int mod\_timer(struct timer\_list \*timer, unsigned long expires);

1

功能： 修改定时器定时时间值， 并且重新注册， 不管这个定时的超时函数是否执行过。 执行完成后会马上启动定时。 功能有点像 add\_timer。

参数：

timer 要修改的定时器结构变量地址， 要求这个结构变量的 function, data 已经初始化好的。

expires： 有来填充结构中的 expires 成员， 也就是未来的时间点。

4.从内核注销定时器

int del\_timer(struct timer\_list \* timer);

1

功能： 从内核定时链表上删除指定的定时器， 删除后就不会再执行绑定的函数

参数： 是一个 struct timer\_list 结构变量地址

例子：

#include <linux/module.h>

#include <linux/init.h>

//1）添加头文件

#include <linux/timer.h>

//3)定义一个timer\_list结构变量

struct timer\_list timerlist;

//2)实现一个超时函数

void timerlist\_func(unsigned long data)

{

printk("%s is call! data:%d\r\n",\_\_FUNCTION\_\_,(int)data);

///再次修改本定时器超时时间 为当前时间 后面2秒

//mod\_timer(&timerlist, jiffies + HZ\*2); //循环使用定时器

}

static int \_\_init timerlist\_init(void)

{

//4)静态初始化

//DEFINE\_TIMER(timelist , time\_function, 0, 123);

//4)动态初始化，对timer\_list结构变量进行初始

init\_timer(&timerlist);

setup\_timer(&timerlist, timerlist\_func, (unsigned long)123);

timerlist.expires = jiffies + HZ\*2;//如果不启用定时器，就不用赋值

//5）注册定时器，启动定时

add\_timer(&timerlist); //启动定时器

printk("%s is call!\r\n",\_\_FUNCTION\_\_);

return 0;

}

static void \_\_exit timerlist\_exit(void) //Module exit function specified by module\_exit()

{

//6）注销定时器

del\_timer(&timerlist);

}

module\_init(timerlist\_init);

module\_exit(timerlist\_exit);

MODULE\_LICENSE("GPL");

1. **二、tasklet机制**

1.tasklet定义以及初始化函数

1）静态初始化

DECLARE\_TASKLET(name, func, data)

#define DECLARE\_TASKLET(name, func, data) \

struct tasklet\_struct name = { NULL, 0, ATOMIC\_INIT(0), func, data }

作用：定义一个名字为 name 的 tasklet\_struct 结构变量，并且初始化这个结构。 所定义的这个 tasklet 是可以被调度，默认是处于被使能状态。

2）静态初始化

DECLARE\_TASKLET\_DISABLED(name, func, data)

#define DECLARE\_TASKLET\_DISABLED(name, func, data) \

struct tasklet\_struct name = { NULL, 0, ATOMIC\_INIT(1), func, data }

作用：定义一个名字为 name 的 tasklet\_struct 结构变量，并且初始化这个结构。所定义的这个 tasklet 是不能被调度，默认是处于被禁能状态。 要调度这个 tasklet 需要先使能。

3）动态初始化

void tasklet\_init(struct tasklet\_struct \*t, void (\*func)(unsigned long), unsigned long data)

作用：初始化一个 tasklet\_struct 结构变量，初始化的结构默认是处于激活状态，可以被调度。

参数：

t：tasklet核心数据结构，已经申明 好（只需定义），如下：

struct tasklet\_struct {

struct tasklet\_struct \*next; //链表中的下一个tasklet

unsigned long state; //tasklet的状态

atomic\_t count; //引用计数器

void(\*func) (unsigned long data); //tasklet处理函数 --参数就下面的data

unsigned long data; //给tasklet处理函数的参数

}；

\*\*注意：\*\*是一个引用定时器，只有其值为0的时候，tasklet才回被激活；否则被禁止，不能被执行。但是，实际上并没有什么用，因为基本都在激活状态，只是在调度的时候才跑到指针函数那里。

func：是一个指针函数，指向tasklet处理函数，这个处理函数的唯一参数为data

data：func指针函数的参数。

2.tasklet 机制使能、禁能函数

1）使能tasklet

void tasklet\_disable(struct tasklet\_struct \*t)

功能：激活tasklet ，可以调度；实质是就是让count加1。

2）失能tasklet

void tasklet\_enable (struct tasklet\_struct \*t)

功能：禁止tasklet ，不可以调度；实质是就是让count减1。

注意：如果对一个 tasklet\_struct 结构变量调用了两次 tasklet\_disable 函数，则 count 会增加 1两次，这时你再调用一次tasklet\_enable ，那实际 count 值还是 1，还是处于非激活状态。

\*\*3.tasklet 调度函数 \*\*

tasklet\_schedule (struct tasklet\_struct \*t)

功能：tasklet\_struct 结构中绑定了一个函数，当用户需要执行这个函数时候，需要自己调用 tasklet\_schedule 函数去通知内核帮我们调度所绑定的函数，但是具体什么时候执行就有内核来处理。（要是绑定函数还没有执行，无论调度多少次，效果等同一次。）

4.tasklet取消调度函数

void tasklet\_kill(struct tasklet\_struct \*t)

函数功能：该函数确保指定的tasklet不会被再次调度运行。如果tasklet正被调度执行，该函数会等待其退出。一般来说，设备要被关闭或者模块要被移除时，我们通常调用这个函数。(注意：不能在tasklet的处理函数中调用这个函数，否则会系统异常)

例子：

#include<linux/kernel.h>

#include<linux/module.h>

#include<linux/interrupt.h>

#include <linux/delay.h>

#define mode 1 //0：静态初始化 1：动态初始化

static void my\_tasklet\_function(unsigned long data);

char my\_tasklet\_data[]="my\_tasklet\_function was called.";

#if mode

//1.1）动态定义结构体变量，用于动态初始化

struct tasklet\_struct my\_tasklet;

#else

//1）静态定义并初始化一个tasklet 内核微线程

DECLARE\_TASKLET(my\_tasklet,my\_tasklet\_function,(unsigned long)my\_tasklet\_data);

#endif

//2）定义tasklet工作函数

static void my\_tasklet\_function(unsigned long data)

{

printk("%s\r\n",(char \*)data);

udelay(2000); //延时2ms，而不是休眠

}

static int \_\_init tasklet\_module\_init(void)

{

printk("mondule insmod\r\n");

#if mode

//1.2）动态初始化

tasklet\_init(&my\_tasklet, my\_tasklet\_function,(unsigned long) my\_tasklet\_data);

#endif

//3）调度tasklet

tasklet\_schedule(&my\_tasklet);

return 0;

}

static void \_\_exit tasklet\_cleanup(void)

{

printk("mondule remove\r\n");

//4）销毁tasklet

tasklet\_kill(&my\_tasklet);

}

module\_init(tasklet\_module\_init);

module\_exit(tasklet\_cleanup);

MODULE\_LICENSE("GPL");

1. **三、工作队列**

1.共享工作队列调度

1）静态初始化

void work\_func(struct work\_struct \* dat)

{

printk(“%p:”,dat);

}

DECLARE\_WORK(mywork,work\_func)

2）动态初始化

struct work\_struct work;

void work\_func(struct work\_struct \* dat)

{

printk(“%p:”,dat);

}

INIT\_WORK(&work, work\_func);

功能：定义一个工作结构变量，初始化工作结构。

参数：

work：类型如下：

struct work\_struct {

atomic\_long\_t data;

struct list\_head entry;

work\_func\_t func; /\* 工作函数指针 \*/

#ifdef CONFIG\_LOCKDEP

struct lockdep\_map lockdep\_map;

#endif

};

3）调度

int schedule\_work(struct work\_struct \*work)

1

功能：调度共享工作队列，将工作结构体变量添加入系统的共享工作队列中，添加入队列的工作完成后会自动从队列中删除。

例子：

#include <linux/module.h>

#include <linux/init.h>

#include <linux/delay.h>

//1）添加头文件

#include <linux/workqueue.h>

//2)定义一个工作函数

void mywork\_func(struct work\_struct \*work)

{

printk("befor:%s is call! work:%p\r\n",\_\_FUNCTION\_\_,work);

ssleep(1); //内核休眠

printk("after:%s is call! work:%p\r\n",\_\_FUNCTION\_\_,work);

}

//3)定义一个 work\_struct 结构变量,并且进行初始化

DECLARE\_WORK(mywork, mywork\_func); //定义并且初始化了

static int \_\_init myworkqueue\_init(void)

{

printk("%s is call!\r\n",\_\_FUNCTION\_\_);

printk("&mywork:%p\r\n",&mywork); //输出 mywork变量地址

//3)如果动态初始化，定义好全局变量后，在这里初始化，可以参考下面的例子

//4)开始调度

schedule\_work(&mywork);

return 0;

}

static void \_\_exit myworkqueue\_exit(void)

{

}

module\_init(myworkqueue\_init);

module\_exit(myworkqueue\_exit);

MODULE\_LICENSE("GPL");

1. **2.自定义工作队列**

1）创建工作队列

create\_workqueue(name)

1

作用： 创建一个名字为 name 的工作队列

参数：

name：工作队列的名字，是一个字符串。

返回值：成功：返回工作队列指针，即struct workqueue\_struct 的结构体（需自己定义）；失败：NULL。

2）往工作队列加入工作节点

int queue\_work(struct workqueue\_struct \*wq, struct work\_struct \*work)

1

功能：往 wq 工作队列中加入工作节点，并调度；

参数：

wq：自定义的工作队列结构变量指针

work：要添加的工作节点

3）销毁工作队列

void destroy\_workqueue(struct workqueue\_struct \*wq)

1

功能： 销毁一个自定义工作队列

参数：

wq：要销毁的工作队列指针

例子：

#include <linux/module.h>

#include <linux/init.h>

#include <linux/delay.h>

//0）添加头文件

#include <linux/workqueue.h>

struct mydata {

int x;

int y;

struct work\_struct work;

int z;

};

struct mydata g\_var;

//1）定义一个工作队列指针

struct workqueue\_struct \* mywq;

//2）定义一个工作函数

void mywork\_func(struct work\_struct \*mywork)

{

struct mydata \*p = (struct mydata\*)container\_of(mywork, struct mydata, work);

printk("%s is call! work:%p\r\n", \_\_FUNCTION\_\_, mywork);

//这个使用全局变量来访问x,y,z

printk("g\_var.x:%d,g\_var.y:%d,g\_var.z:%d,\r\n", g\_var.x, g\_var.y, g\_var.z);

//这个使用函数的形式参数来访问x,y,z

printk("p->x:%d,p->y:%d,p->z:%d,\r\n", p->x, p->y, p->z);

}

static int \_\_init myworkqueue\_init(void)

{

printk("%s is call!\r\n", \_\_FUNCTION\_\_);

printk("&g\_var.work:%p\r\n", &g\_var.work); //输出 mywork变量地址

g\_var.x = 123;

g\_var.y = 456;

g\_var.z = 789;

//3） 创建工作队列

mywq = create\_workqueue("mywq");

if(mywq == NULL) {

printk("create\_workqueue error\r\n");

return -ENOMEM;

}

printk("create\_workqueue ok\r\n");

//4）使用动态初始化工作队列

INIT\_WORK(&g\_var.work, mywork\_func);

//5）开始调度

//schedule\_work(&g\_var.work); //调度共享工作队列

queue\_work(mywq, &g\_var.work); //调度自定义工作队列

return 0;

}

static void \_\_exit myworkqueue\_exit(void)

{

//6） 销毁工作队列

destroy\_workqueue(mywq);

}

module\_init(myworkqueue\_init);

module\_exit(myworkqueue\_exit);

MODULE\_LICENSE("GPL");

1. **3.延时工作队列**

注意：

相同点：和上面的内核工作队列一样，可以实现中断的底部代码功能。

不同点：和内核工作队列惟一区别就是延后的时间可控制。

1）延时调度函数

int schedule\_delayed\_work(struct delayed\_work \*dwork，unsigned long delay)

功能：延时调度工作函数。登记延时工作，一旦登记完成，CPU在适当的时间去执行延时工作的延后处理函数。

参数：

dwork：要调度的延时工作结构指针。

delay：延后调度的时间，单位是时钟节拍。和内核定时器定时时间单位相同，但是不是到期时间，而是定时时间。

例子：

#include <linux/module.h>

#include <linux/init.h>

#include <linux/delay.h>

//0.添加头文件

#include <linux/workqueue.h>

//1)实现一个工作函数

void mywork\_func(struct work\_struct \*mywork)

{

printk("%s is call! work:%p\r\n", \_\_FUNCTION\_\_, mywork);

}

//2)定义一个延时工作

//struct delayed\_work mydelayed\_work; //动态定义

DECLARE\_DELAYED\_WORK(mydelayed\_work, mywork\_func); //静态定义

static int \_\_init myworkqueue\_init(void)

{

printk("%s is call!\r\n", \_\_FUNCTION\_\_);

//2）动态初始化延时工作

// INIT\_DELAYED\_WORK(&mydelayed\_work, mywork\_func);

//3）一安装模块就开始调度,1秒后开始调度工作函数

schedule\_delayed\_work(&mydelayed\_work, msecs\_to\_jiffies(1000)); //调度延时共享工作队列

return 0;

}

static void \_\_exit myworkqueue\_exit(void)

{

}

module\_init(myworkqueue\_init);

module\_exit(myworkqueue\_exit);

MODULE\_LICENSE("GPL");