STM32-WIFI-W8782开发板OSAPI手册

Revision History

Version	Date	Author	Description
V1.0	2015-03-5	lck	文档新建

目录

一、 引言

二、 任务API

原型	备注
int thread_myself **()**	返回任务自身的ID号
int **thread_create**(void (*task)(void *p_arg), void*p_arg, unsigned int prio, unsigned int *pbos, unsigned int stk_size, unsigned char *name)	创建一个线程
int **thread_exit** **(int thread_id)**	退出线程

1. int thread_myself **()**

返回任务自身的ID号。

③ **形参**

③ **返回**
ID 号一般等于优先级

2. int thread_create (void (*task)(void *p_arg),
void *p_arg,
unsigned int prio,
unsigned int *pbos,
unsigned int stk_size，
unsigned char *name)

创建一个线程。

③ **形参**

task

线程函数

p_arg

线程函数执行时的参数

prio

线程的优先级，优先级在app_cfg.h里面分配及定义

pbos

线程堆栈指针，若指定为0，则由系统分配堆栈

stk_size,

线程堆栈大小, 堆栈大小的设置必须很小心,建议打开monitor观察堆栈占用量, 设置时需留一定余量

name

线程名字

③ **返回**

>=0

创建线程成功，返回线程id号，调用退出线程时将使用此id。

<0

创建线程失败。

3. int thread_exit (int thread_id)

退出一个线程

③ **形参**

thread_id

创建线程时返回的线程id

③ **返回**

0

退出线程成功

<0

退出线程出错

三、 中断API

原型	备注
int in_interrupt (void)	查询是否在中断里面

1. int in_interrupt (void)

查询当前是否在中断里面

③ **形参**

无

③ **返回**

>0

当前正在中断函数里

0

当前不在中断函数里

四、 等待事件wait

原型	备注
wait_event_t init_event ()	初始化等待事件
int wait_event (wait_event_t wq, int condition)	等待事件发生
int wait_event_timeout (wait_event_t wq, unsigned int timeout)	等待事件发生或超时
int wake_up (wait_event_t wq)	唤醒事件
void del_event(wait_event_t wq)	删除事件
void clear_wait_event(wait_event_t wq)	清除事件

1. wait_event_t init_event()

初始化一个等待事件队列，暂时只支持一个任务等待事件。不建议多个任务等待，否则响应未知情况。

③ **形参**

wq

等待队列头

2. int wait_event(wait_event_t wq)

将任务加入等待事件队列，阻塞当前任务，等待事件发生(无限阻塞)。

③ **形参**

wq

等待队列头

③ **返回**

0

成功

<0

失败

3. int wait_event_timeout(wait_event_t wq, unsigned int timeout)

将任务加入等待事件队列，阻塞当前任务，等待事件发生或者超时返回。超时值分辨率为10ms，即最小超时值为10ms并以10ms递增。

③ **形参**

wq

等待队列头

timeout

超时值，最小超时值为10ms并以10ms递增。100即100ms超时。

③ **返回**

0

事件发生正常返回

1

超时返回

<0

出错

4. void wake_up(wait_event_t wq)

唤醒一个等待事件队列。

③ **形参**

wq

等待队列头

5. void del_event(wait_event_t wq)

删除一个已经初始化的事件

③ 形参

wq

等待队列头

6. void clear_wait_event (wait_event_t wq)

如果已经有事件发生,将其清0

五、 信号量sem

原型	备注
void sem_init (sem_t *sem, int pshard, int value)	初始化信号量
int sem_wait (sem_t *sem, unsigned int timeout)	阻塞任务直到信号量值大于0
int sem_post (sem_t *sem)	信号量值加1
void sem_destory (sem_t *sem)	释放信号量

1. void sem_init (sem_t *sem, int pshard, int value)

初始化一个信号量，设置信号量的初值。

③ 形参

sem

初始化信号量结构体

pshard

兼容linux共享参数，无实际意义，赋0

value

信号量初始值

2. int sem_wait (sem_t *sem, unsigned int timeout)

信号的P操作，将sem信号量的值减1，阻塞当前任务直到信号量的值大于0。若减1之后当前信号量值大于0，则任务不阻塞，若小于0，则阻塞当前任务。

③ 形参

sem

初始化信号量结构体

timeout

超时值，赋0表示永远阻塞直到信号量激活。大于0的值则是ms值，表示超时多少ms。

③ **返回**

>0 超时

0 成功返回

<0 出错

3. int sem_post (sem_t *sem)

信号的V操作，将sem信号量的值加1，当有任务阻塞在这个信号量上时，调用这个函数会使其中一个任务不再阻塞。

③ **形参**

sem

初始化信号量结构体

4. void sem_destory (sem_t *sem)

释放掉信号量

③ **形参**

sem

初始化信号量结构体

六、 互斥操作mutex

原型	备注
mutex_t mutex_init ()	初始化互斥
int mutex_lock (mutex_t mutex)	获取上锁互斥
int mutex_unlock (mutex_t mutex)	解锁互斥
void mutex_destory (mutex_t mutex)	释放互斥

1. int mutex_init (mutext_t *mutex)

初始化互斥体。
使用形式：

```
mutex_t mutex;
mutex    = mutex_init();
// TASK_A
mutex_lock(mutex);
……..
mutex_unlock(mutex);
// TASK_B
mutex_lock(mutex);
……..
mutex_unlock(mutex);
```

③ **形参**

mutex

初始化互斥体的结构体指针，由使用者定义并传入

③ **返回**

0

初始化成功

<0

初始化失败，可能系统资源已使用完

2. int mutex_lock (mutex_t *mutex)

获取上锁互斥体，若互斥体已经被其他任务获取且未释放，将阻塞当前任务。
与mutex_unlock函数成对使用。

③ **形参**

mutex

已经初始化互斥体的结构体指针

③ **返回**

0

获取上锁互斥体成功

<0

获取上锁互斥体出错

3. int mutex_unlock (mutex_t *mutex)

解锁互斥体，与mutex_lock函数成对使用。

③ **形参**

mutex

已经初始化互斥体的结构体指针

③ **返回**

0

解锁互斥体成功

<0

解锁互斥体出错

4. void mutex_destory (mutex_t *mutex)

释放互斥体。

③ **形参**

mutex

已经初始化互斥体的结构体指针

③ **返回**

0

解锁互斥体成功

<0

释放互斥体出错

七、 消息邮箱mbox

原型	备注
void mbox_new (mbox_t *mbox, void *pmsg)	初始化消息邮箱
void * mbox_get(mbox_t *mbox, unsigned int timeout)	等待消息邮箱中的消息
int mbox_post (mbox_t *mbox, void *pmsg)	向消息邮箱发送一条消息
int mbox_destory (mbox_t *mbox)	释放消息邮箱

1. void mbox_new (mbox_t *mbox, void *pmsg)

初始化一个消息邮箱

③ **形参**

mbox

初始化消息邮箱结构体指针，若返回为NULL，则表示初始化消息邮箱失败。

pmsg

初始化消息，可以为NULL表示初始化邮箱为空。

2. void * mbox_get (mbox_t *mbox, unsigned int timeout)

从消息邮箱中取消息，若消息邮箱中没有消息，将阻塞当前任务，直到消息邮箱中有消息，或者timeout超时。

③ **形参**

mbox

已初始化消息邮箱结构体指针

timeout

等待消息的超时值，ms为单位，最小10ms。0表示不超时，永远阻塞等待。

③ **返回**

>0

返回指向消息的指针

0

出错，返回空指针

3. int mbox_post (mbox_t *mbox, void *pmsg)

向消息邮箱中发送一条消息。

③ **形参**

mbox

已初始化消息邮箱结构体指针

pmsg

发送的消息指针

③ **返回**

>0

返回指向消息的指针

0

出错，返回空指针

4. int mbox_destory (mbox_t *mbox)

释放消息邮箱

③ **形参**

mbox

已初始化消息邮箱结构体指针

③ **返回**

0

释放消息邮箱成功

<0

释放消息邮箱出错

八、 消息队列msg_q

原型	备注
int msgget (msg_q_t *msg_q, int q_size)	初始化消息队列
int msgsnd (msg_q_t *msgid, void *msgbuf)	往消息队列里发送一则消息
int msgrcv (msg_q_t *msgid, void **msgbuf, unsigned int timeout)	从已初始化消息队列中取出一条消息
int msgfree (msg_q_t *msgid)	释放消息队列

③ **使用**

1、初始化

```
struct    user_buf_t{       //消息结构体
      int a;
      int b;
}
msg_q_t    msg_q;       //  消息队列id
if( msgget (&msg_q, 16) < 0)    //初始化16长度的消息队列
      error;     //初始化出错
```

2、往队列里发送一则消息

```
struct user_buf_t    one_time_buf;     //全局变量
if(msg_q != 0)
      msgsnd(&msg_q, (void *)&one_time_buf);
```

3、接收一则消息队列

```
struct user_buf_t    *one_time_buf;
if(msg_q != 0)
      msgsnd(&msg_q, (void **)&one_time_buf, 0);
```

1. int msgget (msg_q_t *msg_q_id, int q_size);

初始化一个消息队列

③ **形参**

msg_q

初始化消息队列结构体指针，若返回为NULL，则表示初始化消息队列失败。

q_size

初始化消息队列的大小

③ **返回**

<0

初始化消息队列失败。

0

初始化消息队列成功。

2. int msgsnd (msg_q_t *msgid, void *msgbuf)

往消息队列里发送一则消息，应用必须保证消息指针不会被销毁，即谨慎使用局部变量，因为局部变量从函数返回时即被回收。

③ **形参**

msgid

已初始化消息队列的msgget返回的msg_q_id，即表征要发往的消息队列。

msgbuf

发往消息队列的消息指针，应用必须保证消息指针不会被销毁，即谨慎使用局部变量，因为局部变量从函数返回时即被回收。

③ **返回**

>0

消息队列已满，发送失败

0

消息正确发往消息队列

<0

消息发送失败

3. int msgrcv (msg_q_t *msgid, void **msgbuf, unsigned int timeout)

从已初始化消息队列中取出一条消息，若消息队列为空则阻塞当前任务直到消息队列返回一条消息。

③ **形参**

msgid

已初始化消息队列的msgget返回的msg_q_id，即表征要接收消息的消息队列。

msgbuf

接收队列中消息的buf

timeout

超时值，赋0表示永远阻塞直到队列有消息。大于0的值则是ms值，表示超时多少ms。

③ **返回**

>0

超时返回

0

成功返回，队列返回消息到msgbuf

<0

出错返回

4. int msgfree(msg_q_t *msgid)

不再使用消息对列，释放消息队列返回给系统。

③ **形参**

msgid

已初始化消息队列的msgget返回的msg_q_id

③ **返回**

0

成功释放返回

<0

出错返回

九、 定时器timer

原型	备注
timer_t * timer_setup (int time_val, int type, timer_callback_func callback, void *callback_arg)	初始化设置一个定时器
int timer_pending (timer_t *tmr)	查询定时器状态
int mod_timer (timer_t *tmr, unsigned int expires)	修改定时器超时值，并激活
int add_timer (timer_t *tmr)	激活定时器
int del_timer (timer_t *tmr)	停止定时器
int timer_free (timer_t *tmr)	释放删除定时器
void sleep (int ms)	睡眠函数

系统提供分辨率稍低的软定时器，分辨率为100ms。

定时器回调函数原型

typedef void (*timer_callback_func)(void *ptmr, void *parg);

定时器函数里面不支持阻塞/睡眠等信号操作,但是支持唤醒/释放信号的操作

1. timer_t * timer_setup (int time_val,
int type,
timer_callback_func callback,
void *callback_arg);

初始化设置一个定时器，仅仅初始化，未激活定时器。

③ **形参**

time_val

定时器超时值，以100ms递增，小于100为100ms。

type

定时器类型，1 为周期执行定时器，0 为执行一次定时器。

callback

定时器回调函数。

callback_arg

传递给回调函数的参数。

③ **返回**

0

设置定时器出错，可能系统定时器已经用完。

>0

定时器初始化成功，返回定时器结构指针给后续激活定时器使用。

2. int timer_pending(timer_t *tmr)

查询定时器状态。

③ **形参**

tmr

已初始化成功的定时器指针。

③ **返回**

0

定时器未激活运行。

>0

定时器正在运行。

<0

查询出错。

3. int mod_timer (timer_t *tmr, unsigned int expires)

修改定时器超时值，并激活。也可以用于修改已经激活的定时器的超时值。

③ **形参**

tmr

已初始化成功的定时器指针。

expires

超时值，分辨率为100ms，即以100ms递增。

③ **返回**

0

定时器超时值修改成功并激活运行定时器。

<0

修改失败。

4. int add_timer(timer_t *tmr);

激活运行已初始化的定时器。

③ **形参**

tmr

已初始化成功的定时器指针。

③ **返回**

0

成功激活运行定时器。

<0

激活定时器出错。

5. int del_timer(timer_t *tmr);

停止定时器。

③ **形参**

tmr

已初始化成功的定时器指针。

③ **返回**

0

成功停止运行定时器。

<0

停止定时器出错。

6. int timer_free(timer_t *tmr);

释放删除定时器。

③ **形参**

tmr

已初始化成功的定时器指针。

③ **返回**

0

成功释放定时器。

<0

释放定时器出错。

7. void sleep (int ms)

睡眠函数，最小睡眠10ms，并以10ms递增。

③ **形参**

ms

睡眠ms数，100即为睡眠100ms，最小值为10。

十、 原子操作atomic

原型	备注
unsigned int lcal_irq_save (void)	储存中断状态
void lcal_irq_restore (unsigned int cpu_sr)	还原中断状态

使用方法：

```
unsigned int cpu_sr;
cpu_sr = local_irq_save;
//do the thing
local_irq_restore(cpu_sr);
```

1. unsigned int local_irq_save (void);

储存cpu中断状态

③ **返回**

返回cpu中断状态，应用必须保存起来，然后给local_irq_restore使用。

2. void local_irq_restore (unsigned int cpu_sr);

还原cpu中断状态

③ **形参**

cpu_sr

local_irq_save返回的cpu中断状态值

十一、