Zstack 中如何实现自己的任务

在 Zstack(TI 的 Zigbee 协议栈)中,对于每个用户自己新建立的任务通常需要两个相关的处理函数,包括:

(1).用于初始化的函数,如:SampleApp_Init(), 这个函数是在 osalInitTasks()这个 osal(Zstack 中自带的小操作系统)中去调用的,其目的就是把一些用户自己写的任务中的一些变量,网络模式,网络终端类型等进行初始化;

(2).用于引起该任务状态变化的事件发生后所需要执行的事件处理函数,如:

SampleApp_ProcessEvent(),这个函数是首先在 const pTaskEventHandlerFn tasksArr[ ] 中进行设置(绑定),然后在 osalInitTasks()中如果发生事件进行调用绑定的事件处理函数.

下面分 3 个部分分析.

1．用户自己设计的任务代码在 Zstack 中的调用过程

(1).main() 执行(在 ZMain.c 中)

main() ---> osal_init_system()

(2). osal_init_system()调用 osalInitTasks(), (在 OSAL.c 中)

osal_init_system() ---> osalInitTasks()

(3). osalInitTasks()调用 SampleApp_Init() , (在 OSAL_SampleApp.c 中)

osalInitTasks() ---> SampleApp_Init()

在 osalInitTasks()中实现了多个任务初始化的设置,其中 macTaskInit(taskID++)到 ZDApp_Init(taskID++)的几行代码表示对于几个系统运行初始化任务的调用,而用户自己实现的 SampleApp_Init()在最后,这里 taskID 随着任务的增加也随之递增.所以用户自己实现的任务的初始化操作应该在 osalInitTasks()中增加.

```
void osalInitTasks( void )

{

uint8 taskID = 0;

//这里很重要, 调用 osal_mem_alloc()为当前 OSAL 中的各任务分配存储空间(实际上是一个任//务数组),并用 tasksEvents 指向该任务数组(任务队列).
```

```
tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);

osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));    //将 taskSEvents 所指向的空间清零

macTaskInit( taskID++ );

nwk_init( taskID++ );

Hal_Init( taskID++ );

#if defined( MT_TASK )

MT_TaskInit( taskID++ );

#endif

APS_Init( taskID++ );

ZDApp_Init( taskID++ );

SampleApp_Init( taskID );    //用户自己需要添加的任务

}
```

2.任务处理调用的重要数据结构

这里要解释一下,在 Zstack 里,对于同一个任务可能有多种事件发生,那么需要执行不同的事件处理,为了方便,对于每个任务的事件处理函数都统一在一个事件处理函数中实现,然后根据任务的 ID 号(task_id)和该任务的具体事件(events)调用某个任务的事件处理函数,进入了该任务的事件处理函数之后,再根据 events 再来判别是该任务的哪一种事件发生,进而执行相应的事件处理.pTaskEventHandlerFn 是一个指向函数(事件处理函数)的指针,这里实现的每一个数组元素各对应于一个任务的事件处理函数,比如 SampleApp_ProcessEvent 对于用户自行实现的事件处理函数 uint16 SampleApp_ProcessEvent(uint8 task_id, uint16 events),所以这里如果我们实现了一个任务,还需要把实现的该任务的事件处理函数在这里添加.

```
const pTaskEventHandlerFn tasksArr[] = {

macEventLoop,

nwk_event_loop,
```

```
Hal_ProcessEvent,

#if defined( MT_TASK )       //一个 MT 任务命令

MT_ProcessEvent,

#endif

APS_event_loop,

ZDApp_event_loop,

SampleApp_ProcessEvent

};
```

注意, tasksEvents 和 tasksArr[]里的顺序是一一对应的, tasksArr[]中的第 i 个事件处理函数对应于 tasksEvents 中的第 i 个任务的事件.

```
//计算出任务的数量

const uint8 tasksCnt = sizeof( tasksArr ) / sizeof( tasksArr[0] );

uint16 *tasksEvents;
```

3. 对于不同事件发生后的任务处理函数的调用

osal_start_system() 很重要，决定了当某个任务的事件发生后调用对应的事件处理函数

```
void osal_start_system(void)

{

#if !defined ( ZBIT )

for(;;) // Forever Loop

#endif

{

      uint8 idx = 0;
```

```
    Hal_ProcessPoll(); // This replaces MT_SerialPoll() and osal_check_timer().
//这里是轮训任务队列,并检查是否有某个任务的事件发生
    do {
      if (tasksEvents[idx]) // Task is highest priority that is ready.
      {
        break;
      }
    } while (++idx < tasksCnt);
    if (idx < tasksCnt)
    {
      uint16 events;
      halIntState_t intState;
      HAL_ENTER_CRITICAL_SECTION(intState);
      events = tasksEvents[idx];     //处理该 idx 的任务事件，是第 idx 个任务的事件发生了
      tasksEvents[idx] = 0; // Clear the Events for this task.
      HAL_EXIT_CRITICAL_SECTION(intState);
      //对应调用第 idx 个任务的事件处理函数,用 events 说明是什么事件
      events = (tasksArr[idx])( idx, events );

      //当没有处理完,把返回的 events 继续放到 tasksEvents[idx]当中
      HAL_ENTER_CRITICAL_SECTION(intState);
    tasksEvents[idx] |= events; // Add back unprocessed events to the current task.
```

```
            HAL_EXIT_CRITICAL_SECTION(intState);
            }
#if defined( POWER_SAVING )
        else // Complete pass through all task events with no activity?
        {
            osal_pwrmgr_powerconserve(); // Put the processor/system into sleep
        }
#endif
        }
}
```