1. 概述

OSAL (Operating System Abstraction Layer)，翻译为“操作系统抽象层”。

在基于 ZigBee 协议的应用开发中，应用程序框架中包含了最多 240 个应用程序对象。如果我们把一个应用程序对象看做为一个任务的话，那么应用程序框架将包含一个支持多任务的资源分配机制。于是 OSAL 便有了存在的必要性，它正是 Z-Stack 为了实现这样一个机制而存在的。

OSAL 就是以实现多任务为核心的系统资源管理机制。所以 OSAL 与标准的操作系统还是有很大的区别的。简单而言，OSAL 实现了类似操作系统的某些功能，但并不能称之为真正意义上的操作系统。

2．OSAL 的 API 接口

函数名称	功能描述
void osal_nv_init()	初始化 FLASH 存储器
uint8 osal_init_system()	初始化操作系统
void osal_mem_init()	初始化内存分配系统
void osalTimerInit()	初始化定时器
void osalInitTasks()	初始化系统任务
void osal_start_system()	进入操作系统
void osal_run_system()	运行操作系统
void osalTimeUpdate()	操作系统时间更新
void Hal_ProcessPoll()	硬件层检查

2.1 消息管理功能

（1）uint8 * osal_msg_allocate(uint16 len)：申请一个指定长度的消息缓存区，

该函数调用 void *osal_mem_alloc(uint16 size)函数实现，从堆中申请存储空间。

（2）uint8 osal_msg_deallocate(uint8 *msg_ptr)：接收到消息的任务处理完成后释放消息的缓存空间。

（3）uint8 osal_msg_send(uint8 destination_task, uint8 *msg_ptr)：发送消息到指定任务，将消息放入队列，并把任务的相应事件标志置位。

（4）uint8 *osal_msg_receive(uint8 task_id)：接收发送到某个消息的任务，在任务处理完消息后，必修释放消息的存储空间。该函数查找消息队列，如果消息队列中有多个发送给该任务的消息，保持事件标志位。

（5）osal_event_hdr_t *osal_msg_find(uint8 task_id, uint8 event)：寻找发送给具有某个事件的任务的消息。

2.2 任务同步功能

（1）uint8 osal_set_event(uint8 task_id, uint16 event_flag)：设置某个任务的某个事件标志。event_flag 为 16 位，只有一个系统事件 SYS_EVENT_MSG，其余的事件都是用户定义的事件。

2.3 时间管理功能

时间管理的 API 既可以被 Z-stack 协议栈中的任务使用，也可以被应用级任务使用。粒度为 1ms。

（1）uint8 osal_start_timerEx(uint8 taskID, uint16 event_id, uint16 timeout_value)：为某个任务设置一个定时器，taskID 为任务 ID，event_id 为用户指定的事件标志位，timeout_value 为超时时间，以 ms 为单位。

（2）uint8 osal_start_reload_timer(uint8 taskID, uint16 event_id, uint16 timeout_value)：设置定时器，与上一个函数不同的是该函数设置的定时器超时后被重新装载。

（3）uint8 osal_stop_timerEx(uint8 task_id, uint16 event_id)：停止一个已经开始的定时器。

（4）uint32 osal_GetSystemClock(void)：获取系统时间，返回值以 ms 为单位。

2.4 中断管理功能

（1）uint8 osal_int_enable(uint8 interrupt_id) ：使能某个中断

（2）uint8 osal_int_disable(uint8 interrupt_id) ：禁止某个中断

2.5 任务管理功能

```
const pTaskEventHandlerFn tasksArr[] =
{
  macEventLoop,
  nwk_event_loop,
  Hal_ProcessEvent,
  MT_ProcessEvent,
  APS_event_loop,
  APSF_ProcessEvent,
  ZDApp_event_loop,
  ZDNwkMgr_event_loop,
  GenericApp_ProcessEvent
};
 const uint8 tasksCnt = sizeof( tasksArr ) / sizeof( tasksArr[0] );
```

数组 tasksArr 定义了各个任务的事件回调函数，如果有用户自己定义的任务，必须将其事件回调函数加入到该数组中。

```
void osalInitTasks( void )
{
  uint8 taskID = 0;
```

```
  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt); //分配内存
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt)); //清零
  macTaskInit( taskID++ );
  nwk_init( taskID++ );
  Hal_Init( taskID++ );
  MT_TaskInit( taskID++ );
  APS_Init( taskID++ );
  APSF_Init( taskID++ );
  ZDApp_Init( taskID++ );
  ZDNwkMgr_Init( taskID++ );
  GenericApp_Init( taskID );
}
```

以上函数在 osal_init_system() 函数中被调用。

2.6 电源管理功能

（1）void osal_pwrmgr_init(void)：初始化电源管理模块的变量，被 osal_init_system()调用。

（2）void osal_pwrmgr_powerconserve(void)：使系统进入节电模式，已经在系统的主循环中被调用，不能调用。

（3）void osal_pwrmgr_device(uint8 pwrmgr_device)：全局设备电源的开关。

（4）uint8 osal_pwrmgr_task_state(uint8 task_id, uint8 state)：控制任务节电状态。

2.7 非易失性存储器管理功能

非易失性存储器的操作很耗时，并且将暂时关闭中断，因此最好在收发器关闭的时候调用这些函数。不要频繁的写 NV mem，因为这非常耗时耗电。

（1）uint8 osal_nv_item_init(uint16 id, uint16 len, void *buf)：初始化一个 NV item

（2）uint8 osal_nv_read(uint16 id, uint16 offset, uint16 len, void *buf)：读取 NV

（3）uint8 osal_nv_write(uint16 id, uint16 offset, uint16 len, void *buf)：写入 NV

（4）osal_offsetof(type, member)：计算一个结构体中某个成员的偏移。

2.8 内存管理功能

（1）void *osal_ mem_alloc(uint16 size)：在堆上分配指定大小的缓冲区。

（2）void osal mem_free(void *ptr)：释放使用 osal_mem_alloc()分配的缓冲区。

以上两个函数要成对使用，防止产生内存泄露。

3．Zigbee 协议栈 OSAL 分析

3.1 OSAL 运行机理

OSAL 是事件驱动的操作系统(Event-driven OS) ，负责调度各个任务的运行，如果有事件发生了，则会调用相应的事件处理函数进行处理。

ZigBee 协议栈的实时性要求并不高，因此在设计任务调度程序时，OSAL 只采用了轮询任务调度队列的方法来进行任务调度管理。

那么，事件和任务的事件处理函数是如何联系起来的呢？ZigBee 中采用的方法是：建立一个事件表，保存各个任务的对应的事件，建立另一个函数表，保存各个任务事件处理函数的地址，然后将这两张表建立某种对应关系，当某一事件发生时则查找函数表找到对应的事件处理函数即可。

ZigBee 协议栈中，有三个变量至关重要。

（1）tasksCnt----该变量保存了任务的总个数。

（2）tasksEvents----这是一个指针，指向了事件表的首地址。这个数组中的

某个元素不为 0，即代表此任务有事件需要相应，事件类型取决于这个元素的值。

（3）tasksArr----这是一个数组，该数组的每一项都是一个函数指针，指向了事件处理函数。该数组的声明为：const pTaskEventHandlerFn tasksArr[]。其中 pTaskEventHandlerFn 的定义为：typedef unsigned short (*pTaskEventHandlerFn)(unsigned char task_id, unsigned short event)。这是定义了一个函数指针。因此，tasksArr 数组的每一项都是一个函数指针，指向了事件处理函数。

总结一下 OSAL 的工作原理：通过 tasksEvents 指针访问事件表的每一项，如果有事件发生，则查找函数表找到相应事件处理函数进行处理，处理完后，继续访问事件表，查看是否有事件发生，无限循环。通常用关中断的方式保护共享资源。

从这种意义上说， OSAL 是一种基于事件驱动的轮询式操作系统。事件驱动是指发生事件后采取相应的事件处理方法，轮询指的是不断地查看是否有事件发生。

不过接下来就有了更加深入的问题了，事件是如何被捕获的？直观一些来说就是，tasksEvents 这个数组里的元素是什么时候被设定为非零数，来表示有事件需要处理的？

在 OSAL 的死循环中，各个事件只是在某些特定的情况下发生，所以这里就引入了心跳的概念，也就是 OS 的时钟节奏。在 Zstack OSAL 中这个节奏定义为 1ms, 由 8 bits HW_TIMER4 来控制。每当 1ms 心跳来临时，Timer4 的中断标志置位，这样在 OSAL 的死循环中检测到这个标志置位后，就通过 osalTimerUpdate（）函数去轮询 Timer 事件链表，没有检测到这个标志位则继续死循环。其中，Timer 事件链表通常在各任务事件的初始化时建立。

Timer 事件链表是下面这样一个结构，next 指向下一个 Timer 事件，timeout 值表明本 Timer 事件还需要 timeout 个心跳才需要被处理，因为此处心跳是 1ms，所以也就是说还需要 timeout 个 ms 才处理。所谓的处理也就是检测 timeout 是否小于 1ms,如果小于 1ms, 则通过 osal_set_event（）将 tasksEvents 相应位置置为 event_flag。如果大于 1ms,说明该 Timer 事件还不到处理的时候，则 Timeout = Timeout-1,然后继续耐心等待下一次心跳。Timer 事件链表结构体定义如下：

```
typedef struct
{
  void *next;
  UINT16 timeout;
  UINT16 event_flag;
  byte task_id;
} osalTimerRec_t;
```

3.2 OSAL 消息队列

事件是驱动任务去执行某些操作的条件，当系统中产生了一个事件，OSAL 将这个事件传递给相应的任务后，任务才能执行一个相应的操作（调用事件处理函数去处理）。通常某些事件发生时，又伴随着一些附加信息的产生，例如：从天线接收数据后，会产生 AF_INCOMING_MSG_CMD 事件，但是任务的事件处理函数在处理这个事件的时候，还需要得到所收到的数据。因此，这就需要将事件和数据封装成一个消息，将消息发送到消息队列，然后在事件处理函数中就可以使用 osal_msg_receive，从消息队列中得到该消息。如下代码可以从消息队列中得到一个消息：MSGpkt = (afIncomingMSGPacket_t *)osal_msg _receive (GenericApp_TaskID)。

OSAL 维护了一个消息队列，每一个消息都会被放到这个消息队列中去，当任务接收到事件后，可以从消息队列中获取属于自己的消息，然后调用消息处理函数进行相应的处理即可。通常在不同任务间通讯时使用消息机制。

3.4 OSAL 添加任务

tasksArr[]数组里存放了所有任务的事件处理函数的地址；osallnitTasks 是 OSAL 的任务初始化函数，所有任务的初始化工作都在这里面完成，并且自动给每个任务分配一个 ID。因此，要添加新任务，只需要编写两个函数：新任务的初始化函数和新任务的事件处理函数。

将新任务的初始化函数添加在 osallnitTasks 函数的最后，如下代码所示。

```
const pTaskEventHandlerFn tasksArr[] = {
  macEventLoop,
  nwk_event_loop,
  Hal_ProcessEvent,
#if defined( MT_TASK )
  MT_ProcessEvent,
#endif
  APS_event_loop,
#if defined ( ZIGBEE_FRAGMENTATION )
  APSF_ProcessEvent,
#endif
  ZDApp_event_loop,
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
  ZDNwkMgr_event_loop,
#endif
  GenericApp_ProcessEvent
};
```

将事件处理函数的地址加入 tasksArr[]数组，如下所示。

```
void osalInitTasks( void )
{
  uint8 taskID = 0;

  tasksEvents = (uint16 *)osal_mem_alloc( sizeof( uint16 ) * tasksCnt);
  osal_memset( tasksEvents, 0, (sizeof( uint16 ) * tasksCnt));

  macTaskInit( taskID++ );
  nwk_init( taskID++ );
  Hal_Init( taskID++ );
#if defined( MT_TASK )
  MT_TaskInit( taskID++ );
#endif
  APS_Init( taskID++ );
#if defined ( ZIGBEE_FRAGMENTATION )
  APSF_Init( taskID++ );
#endif
  ZDApp_Init( taskID++ );
#if defined ( ZIGBEE_FREQ_AGILITY ) || defined ( ZIGBEE_PANID_CONFLICT )
  ZDNwkMgr_Init( taskID++ );
#endif
  GenericApp_Init( taskID );
}
```

需要注意两点：

（1）tasksArr[] 数组里各事件处理函数的排列顺序要与 osallnitTasks 函数中调用各任务初始化函数的顺序保持一致，只有这样才能够保证每个任务的事件处理函数能够接收到正确的任务 ID （在 osallnitTasks 函数中分配）。

（2）为了保存 osallnitTasks 函数所分配的任务 ID，需要给每一个任务定义一个全局变量。如在 Coordinator.c 中定义了一个全局变量 GenericApp_TaskID，并且在 GenericApp_lnit 函数中进行了赋值。

3.5 原语通信

原语只是一个理论层面上的术语，描述了服务层次的关系，以及两个通信的N用户和它们相连的N层（子层）对待协议实体之间的关系。

一个原语的操作往往需要逐层调用下层函数并根据下层返回的结果来进行进一步的操作。在这种情况下，一个原主的操作从发起到完成需要很长时间。因此，如果让程序一直等待下层返回的结果再进一步处理，会使微处理器大部分时间处于循环等待之中，无法及时处理其它请求。

因此，与请求、响应原语操作相对应的函数，一旦调用了下层相关函数后，就立即返回。下层处理函数在操作结束后，将结果以消息的形式发送到上层并产生一个系统事件，调度程序发现这个事件后就会调用相应的事件处理函数对它进行处理。（调用就返回，而不管函数有没有处理完成。当函数处理完成后将结果以消息的形式发送到上层产生一个系统事件）。