3.信息管理 API

3.1 概述

这部分 API 提供了任务或处理单元与不同的处理环境之间的数据交换。这部分 API 函数能够为任务分配和收回消息缓冲区，发送命令消息给以外一个任务并且接收应答。

3.2 osal_msg_allocate ( )

概述：

当一个任务调用这个函数时，将为消息分配缓冲区，函数会将消息加入缓冲区，并调用 osal_msg_send()将消息发送到另一个任务。

原型：

byte *osal_msg_allocate(uint16 len)

参数：

len ： 消息的长度

返回值：

指向消息缓冲区的指针，当分配失败时返回 NULL

3.3 osal_msg_deallocate( )

概述：

用于收回缓冲区

原型：

byte osal_msg_deallocate(byte *msg_ptr)

参数：

Msg_ptr : 指向将要收回的缓冲区的指针

返回值：RETURN VALUE DESCRIPTION

ZSUCCESS 回收成功

INVALID_MSG_POINTER 错误的指针

MSG_BUFFER_NOT_AVAIL 缓冲区在队列中

3.4 osal_msg_send( )

概述：

任务调用这个函数以实现发送指令或数据给另一个任务或处理单元。目标任务的标识必须是一个有效的系统任务，当调用 osal_create_task ( )启动一个任务时，将会分配任务标识。

osal_msg_send()也将在目标任务的事件列表中设置 SYS_EVENT_MSG

原型：

byte osal_msg_send(byte destination_task, byte *msg_ptr)

参数：

destination_task ： 目标任务的标识

msg_ptr ： 指向消息缓冲区的指针

返回值：

ZSUCCESS 消息发送成功

INVALID_MSG_POINTER 无效指针

INVALID_TASK 目标任务无效

3.5 osal_msg_receive( )

概述：

任务调用这个函数来接收消息。消息处理完毕后，发送消息的任务必须调用 osal_msg_deallocate()收回缓冲区。

原型：

byte *osal_msg_receive(byte task_id)

参数：

task_id ： 消息发送者的任务标识

返回值：

指向消息所存放的缓冲区指针，如果没有收到消息将返回 NULL。

4.任务同步 API

4.1 概述

这个 API 使能一个任务等待一个事件的发生和返回控制而不是一直等待。在这个 API 中的函数可以用来为任务设置事件，立刻通知任务有事件被设置。

4.2 osal_set_event( )

概述：

函数用来设置一个任务的事件标志

原型：

byte osal_set_event(byte task_id, UINT16 event_flag)

参数：

task_id ： 任务标识

event_flag ： 2 个字节，每个位特指一个事件。只有一个系统事件，其他事件在接收任务中定义。

返回值：

ZSUCCESS 成功设置

INVALID_TASK 无效任务

５.定时器管理ＡＰＩ

５.１概述

这个ＡＰＩ允许内部任务（Z-Stack）以及应用层任务使用定时器。函数提供了启动和停止定时器的功能，定时器最小增量为 1MS。

5.2 osal_start_timer( )

概述：

启动定时器函数。当定时器到点时，the given event bit will be set。事件将在任务中设置，要指明具体任务，调用 osal_start_timerEx()

原型:

byte osal_start_timer(UINT16 event_id, UINT16 timeout_value);

参数：

event_id : 用户定义的 event bit. 当定时器到点时，事件将通知任务。

timeout_value ：定时值（ms）

返回值：

ZSUCCESS Timer 成功开启

NO_TIMER_AVAILABLE 无法开启

5.3 osal_start_timerEx( )

概述：

功能与 osal_start_timer()相近，这个函数允许调用者为另一个任务启动定时器

原型：

byte osal_start_timerEx(byte taskID, UINT16 event_id, UINT16 timeout_value);

参数：

略

返回值：

ZSUCCESS Timer 成功开启

NO_TIMER_AVAILABLE 无法开启

5.4 osal_stop_timer( )

概述：

停止正在运行的定时器，停止外部事件调用 osal_stop_timerEx()

原型：

byte osal_stop_timer(UINT16 event_id);

参数：

event_id ：将要结束的目标事件（该事件是启动定时器的事件）

返回值：

ZSUCCESS Timer 成功停止

INVALID_EVENT_ID 无效事件

5.5 osal_stop_timerEx( )

概述：

结束外部事件的定时器

原型：

byte osal_stop_timerEx(byte task_id, UINT16 event_id);

参数：

略

返回值

ZSUCCESS Timer 成功停止

INVALID_EVENT_ID 无效事件

5.6 osal_GetSystemClock( )

概述：

读取系统时间

原型：

uint32 osal_GetSystemClock(void);

参数：

无

返回值：

系统时间（ms）

6.中断管理 API

6.1 概述：

这个 API 实现任务与外部中断的接口，函数允许任务关联每一个具体的中断程序，可以开关中断。在中断服务程序内，其他任务可以设置事件。

6.2 osal_int_enable( )

概述：

函数用于使能中断。

原型：

byte osal_int_enable(byte interrupt_id)

参数：

interrupt_id ： 目标中断

返回值：

ZSUCCESS Interrupt 成功使能

INVALID_INTERRUPT_ID 无效中断

6.3 osal_int_disable( )

概述：

关闭中断

原型：

byte osal_int_disable(byte interrupt_id)

参数：

略

返回值：

ZSUCCESS Interrupt 成功关闭

INVALID_INTERRUPT_ID 无效中断

7.任务管理 API

7.1 概述

这个 API 用于在 OSAL 中增加和管理任务。每一个任务由任务初始化函数和时间处理函数组成。

OSAL calls osalInitTasks() [application supplied] to initialize the tasks and OSAL uses a task table (const pTaskEventHandlerFn tasksArr[]) to call the event processor for each task (also application supplied).

建立 task table 的例子：

```
const pTaskEventHandlerFn tasksArr[] =

{

macEventLoop,                    //我的理解是已经建立好的事件队列

nwk_event_loop,

Hal_ProcessEvent,
```

```
MT_ProcessEvent,

APS_event_loop,

ZDApp_event_loop,

};

//任务数

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

建立任务的例子：

```
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++ );

MT_TaskInit( taskID++ );

APS_Init( taskID++ );

ZDApp_Init( taskID++ );

}
```

7.2 osal_init_system()

概述：

该函数初始化 OSAL 系统。必须用在任何一个使用 OSAL 的地方。

原型：

byte osal_init_system(void)

参数:

无

返回值：

ZSUCCESS 成功

7.3 osal_start_system()

概述：

这个函数是系统任务的主循环函数，在循环里面将遍历所有的任务事件，为触发事件的任务调用任务事件处理函数。当事件处理完之后，将返回主循环。如果没有事件，函数将把处理器转到睡眠模式。

原型：

void osal_start_system(void)

参数：

无

返回值：

无

7.4 osal_self()

概述：不在支持，不推荐

7.5 osalTaskAdd ()

概述：不在支持，不推荐　可以参考 7.1 任务初始化。

8.内存管理 API

8.1 概述

简单的内存分配系统，支持动态分配内存。

8.2 osal_mem_alloc()

概述：

如果成功的话，该函数将分配内存并返回指针。

原型：

```
void *osal_mem_alloc( uint16 size );
```

参数：

size – 想分配的内存空间的字节数

返回值：

返回一个 void 指针指向新分配的缓冲区。如果没有足够的空间将返回 NULL。

8.3 osal_mem_free()

概述：

释放内存空间

原型：

void osal_mem_free(void *ptr);

参数：

Ptr ：指向将要释放的空间，改空间必须是分配过的。

返回值：

无

9.电源管理 API

9.1 概述

这个部分阐述了 OSAL 的电源管理系统。系统提为应用或者任务提供了通知 OSAL 的方式，包括何时可以安全关闭，接收设备和其他设备，以及何时将处理器处于睡眠模式。

有两类控制电源管理的函数。第一个，osal_pwrmgr_device()，设置设备级别模式（节点或不节点）。其次是 osal_pwrmgr_task_state(PWRMGR_HOLD)，每个任务可以通过调用它函数 hold off the power manager，如果一个任务“Holds” the power manager 后就需要调用 osal_pwrmgr_task_state(PWRMGR_CONSERVE) 允许电源管理来保存电源管理模式。

默认当任务建立时，每个任务的电源管理状态被设置成 PWRMGR_CONSERVE。如果任务不想实行断电保护（没有变化），不需要调用 osal_pwrmgr_task_state(). 电源管理将在进入电源保护状态之前察看设备模式和所有任务共有的电源状态。

9.2 osal_pwrmgr_device()

概述：

函数在上电或电源需求变更时调用（例如电源支持协调器）。这一函数设置了大体的设备电源管理的开/关状态。该函数应当从中央控制实体（如 ZDO)

被调用。

原型：

void osal_pwrmgr_state(byte pwrmgr_device);

参数：

pwrmgr_device ：更改或设置节电模式

PWRMGR_ALWAYS_ON 无节电

PWRMGR_BATTERY 开节电

返回值：

无

9.3 osal_pwrmgr_task_state()

概述：

任务调用这个函数决定是否让 OSAL 保存电源状态。默认当任务创建时，它自己的电源状态设置成保存，如果任务总是想保存电源状态，就不需要调用这个函数。

原型：

byte osal_pwrmgr_task_state(byte task_id, byte state);

参数：

State – 变更的电源状态

PWRMGR_CONSERVE 打开节电，初始化默认

PWRMGR_HOLD 关闭节电

返回值：

ZSUCCESS
成功

INVALID_TASK 无效任务 ID

10.非易失性（NV）内存管理

10.1 概述

这部分阐述了 OSAL 的非易失性内存管理系统。系统为 应用 提供了一种永久储存信息的方式。协议栈也使用它来保存一些协议必要的项目。非易失性函数用来读写用户定义的项目包括任意的数据类型例如结构体和数组。用户可以读写这个项目，或者项目里的单一元素通过设置适当的长度。

每部分 NV 有一个 ID，有些被占用了，以下是分块表。

0x0000 Reserved

0x0001 – 0x0020 OSAL

0x0021 – 0x0040 NWK

0x0041 – 0x0060 APS

0x0061 – 0x0080 Security

0x0081 – 0x00A0 ZDO

0x00A1 – 0x0200 Reserved

0x0201 – 0x0FFF Application

0x1000 -0xFFFF Reserved

在使用 API 时有一些重要的注意点：

1. 。。。例如，最好的写 NV 的时间是关闭接收之后

2. 尽量减少 NV 的写频率，耗费时间和电源，对于 flash 有擦写次数的限制。

3. 如果一个或多个 NV 项目的结构发生变化，特别是当从一个升级版本的 z-stack 到另一个，有必要擦除和重新初始化 NV 内存。否则，在 NV 项目上的读和写操作的改变将失败或产生错误的结果。

10.2 osal_nv_item_init()

概述：

初始化一个 NV 项目。这个函数检查一个 NV 项目的存在与否。如果不存在，它将被建立和初始化随着数据一起传给函数，这个函数必须在调用 osal_nv_read() or osal_nv_write()之前被调用

原型：

byte osal_nv_item_init(uint16 id, uint16 len, void *buf);

参数：

Id ：用户定义的项目 ID

Len ：项目的大小

Buf ：指向项目初始化的数据。如果没有初始化的数据，设置为 NULL

返回值：

ZSUCCESS 成功

NV_ITEM_UNINIT 成功但是项目之前不存在

NV_OPER_FAILED 操作失败

10.3 osal_nv_read()

概述：

从 NV 中读数据，可以读取整个项目，或是项目中有索引指定的元素，数据复制到*buf

原型：

byte osal_nv_read(uint16 id, uint16 offset, uint16 len, void *buf);

参数：

Id ：用户定义的项目 ID

Offset ： Memory offset into item in bytes.

Len ：项目长度

Buf ：数据保存缓冲区指针

返回值：

ZSUCCESS Success

NV_ITEM_UNINIT Item is not initialized

NV_OPER_FAILED Operation failed

10.4 osal_nv_write()

概述：

写数据到 NV

原型：

byte osal_nv_write(uint16 id, uint16 offset, uint16 len, void *buf);

参数：

Id : 用户定义的 ID

Offset ： Memory offset into item in bytes.

Len ： 项目长度

Buf ： 写数据

返回值：

ZSUCCESS
成功

NV_ITEM_UNINIT	项目没有初始化
NV_OPER_FAILED	操作失败

10.5 osal_offsetof()

概述：

这个宏计算出一个单元内结构的内存偏移量。他对 NV API 函数涌来计算偏移量参数很有用。

原型：

osal_offsetof(type, member)

参数：

Type ： 结构类型

Member ： 结构成员