

AliOS Things操作系统开发

阿里云 XXXX

课程目标

学习完本课程后，你将能够：

1. 了解AliOS Things操作系统的内核编程
2. 了解AliOS Things常用组件的使用

课程目录

1. AliOS Things源代码编译

1.1 使用命令行

1.2 使用VSCODE

2. AliOS Things源代码

3. 智能硬件

AliOS Things操作系统的特点

- 内核: rhino
- 公开源代码 (大部分)
- 可移植性 (Portable)
 - 大部分源码用C编写
 - 微处理器硬件相关的那部分用汇编语言写的, 尽可能少
 - 可以在很多微处理器上运行
- 可固化 (ROMable)
- 可裁剪 (Scalable)
 - 代码以组件形式, 通过文件中的#define进行裁剪
 - ESP32内核功能裁剪: AliOS-Things-rel_3.1.0\platform\board\board_legacy\esp32devkitc\k_config.h(覆盖默认配置)
 - 内核的默认配置: AliOS-Things-rel_3.1.0\core\rhino\include\k_default_config.h
- 调度方式
 - 基于优先级的调度
 - 基于优先级的时间片调度 (默认)
- 系统服务
 - 信号量、互斥信号量 (处理优先级反转问题)、消息队列、事件、内存管理、软件定时器等

嵌入式系统代码结构（不带操作系统）

- 带中断的轮转结构

- 中断程序处理硬件特别紧急的需求，设置标志
- 主循环轮询这些标志，根据这些需求进行后续的处理
- 前后台系统：前台—中断，后台—循环

```
BOOL fDeviceA=FALSE;
BOOL fDeviceB=FALSE;
...
BOOL fDeviceZ=FALSE;
void interrupt vHandleDeviceA(void)
{
    fDeviceA=TRUE;
}
void interrupt vHandleDeviceB(void)
{
    fDeviceB=TRUE;
}
...
void interrupt vHandleDeviceB(void)
{
    fDeviceZ=TRUE;
}
```

```
void main()
{
    while (TRUE){
        if (fDeviceA){
            fDeviceA=FALSE;
        }
        if (fDeviceB){
            fDeviceB=FALSE;
        }
        ...
        if (fDeviceZ) {
            fDeviceZ=FALSE;
        }
    }
}
```

嵌入式系统代码结构（带操作系统）

- 代码由一系列任务代码组成
- 将系统功能分解成一系列的任务
- 任务轮流运行互相配合，实现系统功能
- 任务就是一个进程，解决特定的问题，具有特定的数据结构
- 任务由操作系统内核，根据任务优先级及时间片轮流调度运行
- 任务分类
 - 系统任务—系统自带，提供某些服务，如空闲任务
 - 用户任务—解决用户的实际问题实际应用问题

嵌入式系统代码结构（带操作系统）

- Alios程序结构
 - 用户负责创建任务
 - 不直接调用任务代码，任务由操作系统调度运行

```
void MyTask1(void *pdata)
{
    for(;;)
    {
        .....
    }
}

void MyTask2(void *pdata)
{
    for(;;)
    {
        .....
    }
}
```

```
void MyTask3(void *pdata)
{
    for(;;)
    {
        .....
    }
}
```

```
int application_start(int argc, char *argv[]){
    .....
    aos_task_new ("MyTask1", MyTask1, .....);
    aos_task_new ("MyTask2", MyTask2, .....);
    aos_task_new ("MyTask3", MyTask3, .....);
    while(1)
    {
        .....
    }
    .....
}
```

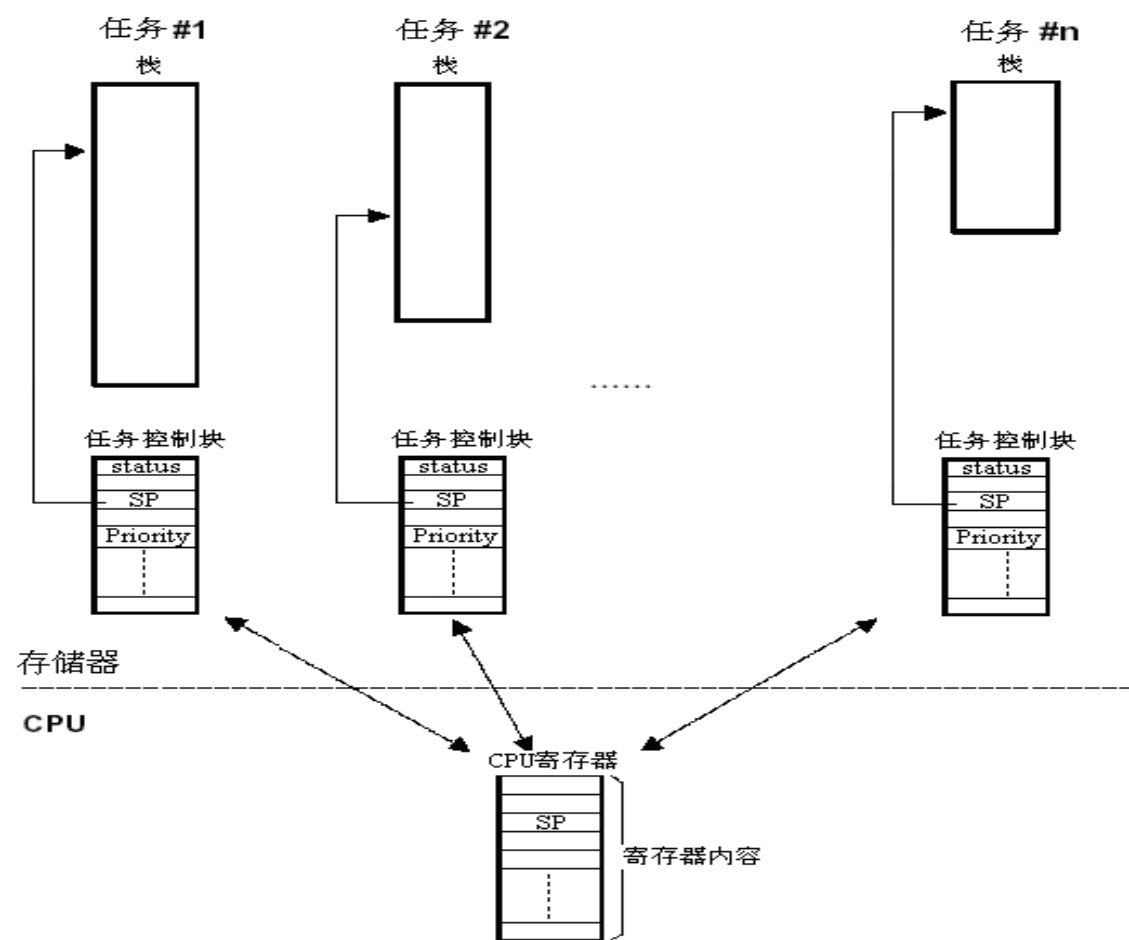
任务的基本概念

- 任务代码的结构
 - 形式：C语言函数
- 参数
 - 指向void的指针，可以指向任何内容
- 代码结构
 - 无限循环，永不返回
 - 包括放弃CPU使用权的代码
- 代码特点
 - 各任务互相独立，运行时独占处理器
 - 任务局部变量放在栈中，为任务私有数据
 - 任务间交换私有数据通过操作系统的服务
 - 全局变量需要互斥访问

```
void MyTask(void *pdata)
{
    for(;;)
    {
        用户代码;

        .....
        放弃CPU使用权;
    }
}
```


任务的基本概念



AliOS中的任务

- 两种调度策略

- 基于优先级的抢占式调度—任务间优先级不能相同
 - 运行系统中优先级最高的、就绪的任务
- 基于优先级的抢占+基于时间片的轮转调度（默认方式） —任务间优先级可以相同
 - 运行系统中优先级最高的、就绪的任务
 - 假如这样的任务有多个，互相间根据时间片进行轮转调度，时间片用完放到当前优先级，调度队列的最后
 - 默认时间片为50 ticks (RHINO_CONFIG_TIME_SLICE_DEFAULT)，系统时钟频率为100 (RHINO_CONFIG_TICKS_PER_SECOND)

- 优先级分配—可配置修改

- 默认为0~62，值越大表示优先级越低，
- 其中空闲任务的优先级是61，无任务可运行时，执行空闲任务，内有钩子函数，用户可以编写自己的代码
- 用户任务优先级范围1-60，0、61、62保留 建议不要超过32的优先级

- 任务堆栈

- 根据任务局部变量的多少，分配堆栈的大小
- 堆栈溢出是造成系统崩溃的主要原因
- 系统具有栈溢出检测功能 (RHINO_CONFIG_TASK_STACK_OVF_CHECK)，开启后一旦溢出产生异常

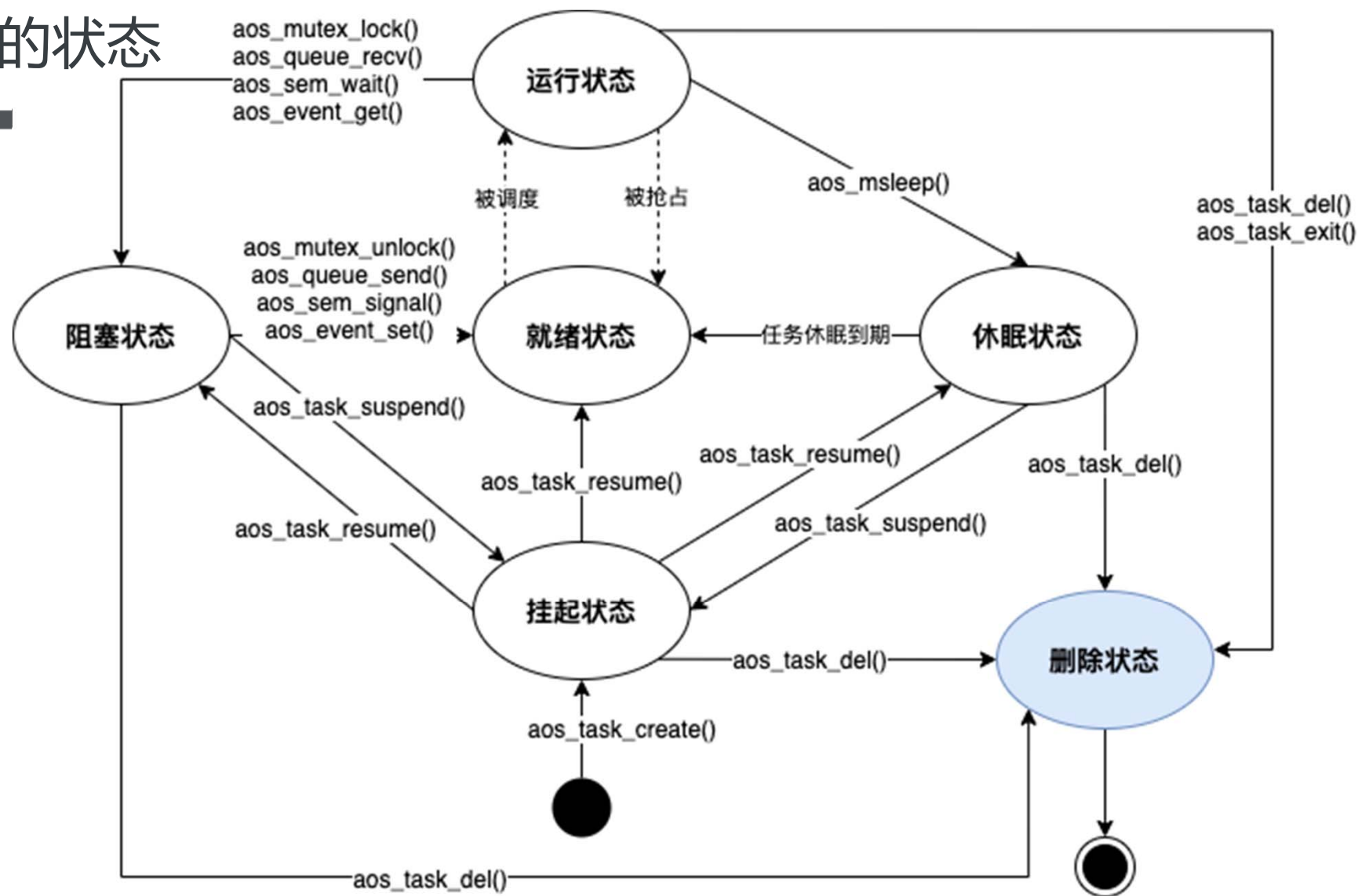
AliOS中的任务

- 任务的状态

- 任务状态分为就绪状态、挂起状态、休眠状态、阻塞状态、运行状态和删除状态
- 某些状态可以互相组合
- 任务必定处理以上状态中

状态符号	描述
RDY	任务已在就绪队列或已被调度运行，处于就绪状态或运行状态
PEND	任务因等待资源或事件的发生而处于阻塞状态
SUS	任务因被其他或自身调用挂起函数aos_task_suspend()后所处的挂起状态
SLP	任务处于休眠状态
PEND_SUS	任务在阻塞状态下，被其他任务挂起，处于阻塞挂起状态
SLP_SUS	任务在休眠状态下，被其他任务挂起，处于休眠挂起状态
DELETED	任务处于删除状态

任务的状态



内核函数说明

- 内核函数分为两个系列
- krhino_xx系列
 - rhino原生的API函数
 - rhino升级后可能改变
 - 功能丰富，参数多，使用复杂
- aos_xx系列—**推荐使用**
 - alios封装后的内核操作函数
 - 内部调用krhino_xx系列的函数实现
 - 对函数的调用进行了部分简化，功能受一定限制
 - 相对稳定，受内核升级影响较小

任务常用函数

函数名	描述
aos_task_new()	任务创建函数
aos_task_new_ext()	任务创建函数
aos_task_exit()	任务退出函数
aos_task_delete()	任务删除函数
aos_task_resume()	任务恢复函数
aos_task_suspend()	任务挂起函数
aos_task_yield()	任务让出CPU函数
aos_task_self()	获取当前任务的句柄
aos_task_name_get()	获取任务名称

任务常用函数

• 任务创建函数

◆ aos_task_new()

```
aos_status_t aos_task_new (const char *    name,  
                           void(*) (void *) fn,  
                           void *         arg,  
                           size_t         stack_size  
                           )
```

Create a task. Deprecated, not Recommended.

参数

[in] **name** task name.
[in] **fn** function to run.
[in] **arg** argument of the function.
[in] **stacksize** stack-size in bytes.

返回

0: success, otherwise: fail.

优先级是AOS_DEFAULT_APP_PRI 32

◆ aos_task_new_ext()

```
aos_status_t aos_task_new_ext (aos_task_t *    task,  
                               const char *    name,  
                               void(*) (void *) fn,  
                               void *         arg,  
                               size_t         stack_size,  
                               int32_t        prio 定义优先级  
                               )
```

Create a task. Deprecated, not Recommended.

参数

[in] **task** handle.
[in] **name** task name.
[in] **fn** task function.
[in] **arg** argument of the function..
[in] **stack_size** stack-size in bytes.
[in] **prio** priority value, the max is RHINO_CONFIG_USER_PRI_MAX(default 60).

返回

0: success.

任务常用函数

• 任务的删除

◆ aos_task_exit()

```
void aos_task_exit ( int32_t code )
```

任务退出，该接口功能是任务删除自身，且IDLE任务不允许删除。

使用约束

该接口不能在中断上下文中调用

错误处理

如果挂起任务为IDLE，则直接返回

参数

[in] `code` 未使用.

返回

无

◆ aos_task_delete()

```
aos_status_t aos_task_delete ( aos_task_t * task )
```

删除任务，该接口删除一个任务并回收任务资源，不允许删除IDLE任务。

使用约束

该接口不能在中断上下文中调用

错误处理

如果任务句柄为NULL，则返回错误码-EINVAL

如果删除的任务为IDLE，则返回错误码-EPERM

参数

[in] `task` 任务对象句柄.

返回

状态码

返回值

0 恢复任务成功

-EINVAL 输入非法参数导致失败

-EPERM 尝试删除IDLE任务导致失败

-1 其他原因导致的失败

任务常用函数

• 任务的挂起与恢复

◆ aos_task_suspend()

```
aos_status_t aos_task_suspend ( aos_task_t * task )
```

挂起任务，该接口将已创建的任务挂起，暂时不执行，挂起的对象既可以是任务自身也可以是其他任务，但不允许挂起IDLE任务。

使用约束

该接口不能在中断上下文中调用

错误处理

如果任务句柄为NULL，则返回错误码-EINVAL
如果挂起任务为IDLE，则返回错误码-EPERM

参数

[in] **task** 任务对象句柄。

返回

状态码

返回值

0 挂起任务成功
-EINVAL 输入非法参数导致失败
-EPERM 尝试挂起IDLE任务导致失败
-1 其他原因导致的失败

◆ aos_task_resume()

```
aos_status_t aos_task_resume ( aos_task_t * task )
```

恢复任务，该接口将挂起任务恢复，取消暂时不执行状态。

使用约束

该接口不能在中断上下文中调用

错误处理

如果任务句柄为NULL，则返回错误码-EINVAL

参数

[in] **task** 任务对象句柄。

返回

状态码

返回值

0 恢复任务成功
-EINVAL 输入非法参数导致失败
-1 其他原因导致的失败

任务常用函数

- 让出CPU资源

- 仅在时间片调度时有效
- 将当前运行任务放至同优先级就绪任务队列的最后
- 任务仍为就绪，并不挂起任务

◆ aos_task_yield()

```
aos_status_t aos_task_yield (void )
```

当前任务让出CPU资源，该接口将当前任务唤出，放入就绪队列对尾，暂时放弃CPU的使用权。

使用约束

该接口不能在中断上下文中调用

错误处理

如果任务句柄为NULL，则返回错误码-EINVAL

参数

[in] 无。

返回

状态码

返回值

0 恢复任务成功
-EINVAL 输入非法参数导致失败
-1 其他原因导致的失败

任务常用函数

获取当前任务句柄

◆ aos_task_self()

```
aos_task_t aos_task_self ( void )
```

获取当前任务的任务对象句柄。

使用约束

无。

错误处理

无。

参数

[in] 无。

返回

任务对象句柄

获取任务名

◆ aos_task_name_get()

```
aos_status_t aos_task_name_get ( aos_task_t * task,  
                                char *      buf,  
                                size_t      buf_size  
                                )
```

获取任务名称，该接口将指定任务的任务名称拷贝到用户缓冲区。

使用约束

该接口不能在中断上下文中调用

错误处理

如果任务句柄为NULL，则返回错误码-EINVAL

如果用户缓冲区地址参数为NULL，则返回错误码-EINVAL

如果用户缓冲区大小为0，则返回错误码-EINVAL

参数

[in] task 任务对象句柄

[out] buf 输出任务名的用户缓冲区地址

[in] buf_size 输出任务名的用户缓冲区大小

返回

状态码

返回值

0 恢复任务成功

-EINVAL 输入非法参数导致失败

-1 其他原因导致的失败

例1

```
#include <stdio.h>
#include <aos/kernel.h>
void func1(void * arg)
{
    int tmp=0;
    while(1)
    {
        printf("task1 print %d\r\n",tmp);
        tmp++;
        aos_msleep(1000);
    }
}
void func2(void * arg)
{
    int tmp=0;
    while(1)
    {
        printf("task2 print %d\r\n",tmp);
        tmp++;
        aos_msleep(1000);
    }
}
```

```
int application_start(int argc, char *argv[])
{
    int tmp=0;
    printf("nano entry here!\r\n");    创建两个任务，优先级32
        aos_task_new("task1", func1,0, 8192);
        aos_task_new("task2", func2,0, 8192);
    while(1) {
        printf("maintask print %d\r\n",tmp);
        tmp++;
        aos_msleep(1000);    死循环
    };
}
```

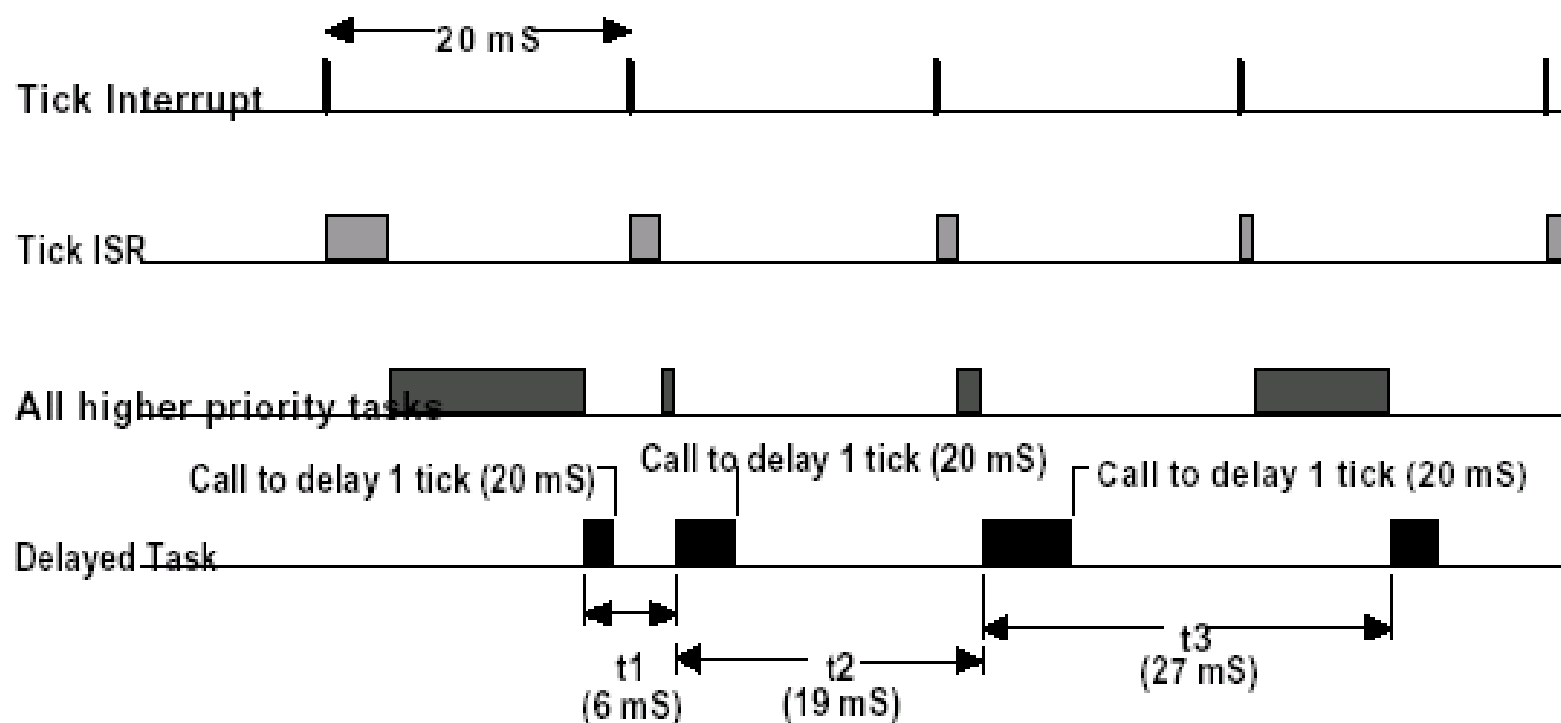
时钟节拍与软件定时器

- 时钟节拍 (Clock Tick)

- 操作系统通过硬件定时器，产生周期性的中断，是所有操作系统必须的
- 作用：（1）依靠该中断来调度任务（2）计算延迟时间
- 时钟节拍的频率：根据操作系统不同而不同，AliOS默认周期为10ms，频率通过RHINO_CONFIG_TICKS_PER_SECOND定义
 - 时钟节拍率越快，任务切换越及时，实时性更好
 - 时钟节拍率越快，延时分辨率越高
 - 时钟节拍率越快，系统的额外开销就越大
- 使用最小时钟节拍进行延时，为了确保延时时间，设置的延时数要+1
 - 例如时钟节拍每20ms发生一次，现在要至少延迟20ms，假设Delay(x)是延迟时钟节拍数函数，则要设置为Delay(2)

时钟节拍与软件定时器

- 操作系统中软件延时的波动原理



时钟节拍与软件定时器

- 软件定时器

- 定时器在嵌入式开发中广泛应用
- 系统利用时钟节拍和高优先级任务，提供了软件定时器服务
- **软件定时器功能：**指定延时特定时间后，一次或周期调用某个函数（回调函数）。延时时间以时钟节拍为单位，回调函数由用户设定
- 实现原理：
 - 创建了一个高优先级的定时器任务（RHINO_CONFIG_TIMER_TASK_PRI 默认为5，堆栈大小为RHINO_CONFIG_TIMER_TASK_STACK_SIZE默认为768字节）
 - 定时器任务从消息队列中读取消息并处理该命令（消息数RHINO_CONFIG_TIMER_MSG_NUM 默认为20）
 - 将各个软件定时器挂到g_timer_head链表上，利用系统tick管理定时器，时间到时调用函数

软件定时器常用函数

函数名	描述
aos_timer_new()	定时器创建函数
aos_timer_new_ext()	定时器创建函数
aos_timer_free()	定时器删除函数
aos_timer_start()	定时器启动函数
aos_timer_stop()	定时器停止函数
aos_timer_change()	定时器初始时长和周期间隔参数变更函数

软件定时器常用函数

• 定时器的创建

◆ aos_timer_new()

```
aos_status_t aos_timer_new ( aos_timer_t *      timer,
                             void(*) (void *, void *) fn,
                             void *             arg,
                             uint32_t           ms,
                             bool               repeat
                             )
```

This function will create a timer and run auto. Deprecated, not Recommended.

参数

[in] **timer** pointer to the timer.
[in] **fn** callback of the timer.
[in] **arg** the argument of the callback.
[in] **ms** ms of the normal timer trigger.
[in] **repeat** repeat or not when the timer is created.

注解

fn first arg: timer->hdl, not aos_timer_t *timer; second arg: user param arg.

返回

0: success.

◆ aos_timer_new_ext()

```
aos_status_t aos_timer_new_ext ( aos_timer_t *      timer,
                                  void(*) (void *, void *) fn,
                                  void *             arg,
                                  uint32_t           ms,
                                  bool               repeat,
                                  bool               autorun
                                  )
```

This function will create a timer. Deprecated, not Recommended.

参数

[in] **timer** pointer to the timer.
[in] **fn** callback of the timer.
[in] **arg** the argument of the callback.
[in] **ms** ms of the normal timer trigger.
[in] **repeat** repeat or not when the timer is created.
[in] **autorun** auto run.

注解

fn first arg: timer->hdl, not aos_timer_t *timer; second arg: user param arg.

返回

0: success.

软件定时器常用函数

- 定时器的启动与停止

◆ aos_timer_start()

```
aos_status_t aos_timer_start ( aos_timer_t * timer )
```

This function will start a timer.

参数

[in] **timer** pointer to the timer.

返回

0: success.

◆ aos_timer_stop()

```
aos_status_t aos_timer_stop ( aos_timer_t * timer )
```

This function will stop a timer.

参数

[in] **timer** pointer to the timer.

返回

0: success.

软件定时器常用函数

定时器时长的修改（只能修改周期定时器）

◆ aos_timer_change()

```
aos_status_t aos_timer_change ( aos_timer_t * timer,
                                uint32_t      ms
                                )
```

This function will change attributes of a timer.

参数

[in] **timer** pointer to the timer.
[in] **ms** ms of the timer trigger.

注解

change the timer attributes should follow the sequence as timer_stop->timer_change->timer_start

返回

0: success.

定时器删除

◆ aos_timer_free()

```
void aos_timer_free ( aos_timer_t * timer )
```

This function will delete a timer.

参数

[in] **timer** pointer to a timer.

例2

```
#include <stdio.h>
#include <aos/kernel.h>
aos_timer_t mytimer1;
void func(void * timer,void * arg)
{
    printf("Hello\r\n");
}
int application_start(int argc, char *argv[])
{
    int count = 0;
    printf("nano entry here!\r\n");
    aos_timer_new(&mytimer1, func, 0, 2000, 1);
    aos_timer_start(&mytimer1);
    while(1) {
        printf("hello world! count %d \r\n", count++);
        aos_msleep(1000);
    };
}
```

idle任务钩子的使用

- 当AliOS无高优先级任务可以运行时，会运行idle任务
- idle优先级默认为 $62-1=61$
- 用户可以定义idle hook函数，在系统空闲时运行指定的程序
 - 系统维护，内存清理，系统状态上报等
 - 休眠、节能
- **idle任务不能挂起，不能删除**
- idle hook的实现
 - AliOS-Things-rel_3.1.0\platform\mcu\esp32-aos\hook_impl.c
- idle hook的应用配置
 - `#define OS_RHINO_TICK_HOOK_NUM 10`
 - `#define RHINO_CONFIG_USER_HOOK 1`
- idle hook的api函数 **要使用必须设置为1**
 - `#include <k_api.h>`
 - `void krhino_idle_add_hook(void * fun, int cpuid)`: 增加hook函数
 - `void krhino_idle_del_hook(void * fun, int cpuid)`: 删除hook函数
 - fun原型: `void fun(void)`
 - cpuid: 对应的cpu号，一般为0

例3

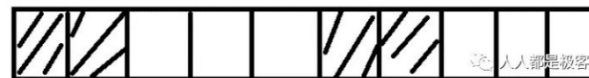
```
#include <stdio.h>
#include <k_api.h>
#include <aos/kernel.h>
void hook1(void)
{
    printf("idle hook1\r\n");
}
int application_start(int argc, char *argv[])
{
    int count = 0;
    printf("nano entry here!\r\n");
    krhino_idle_add_hook(hook1,0);
    while(1) {
        printf("task run now!\r\n");
        aos_msleep(500);
    };
}
```

内存管理

- 操作系统对内存提供了管理服务

- 内存块的高效分配
- 长时间运行，内存碎片的处理

- 例如：需要分配4个连续单位的内存，如右图



- AliOS提供了2种内存管理策略

- Buddy算法—提高内存的利用率，速度略慢，适用大内存块分配
- BLK算法—速度快，但有浪费，适用小内存块分配

内存管理

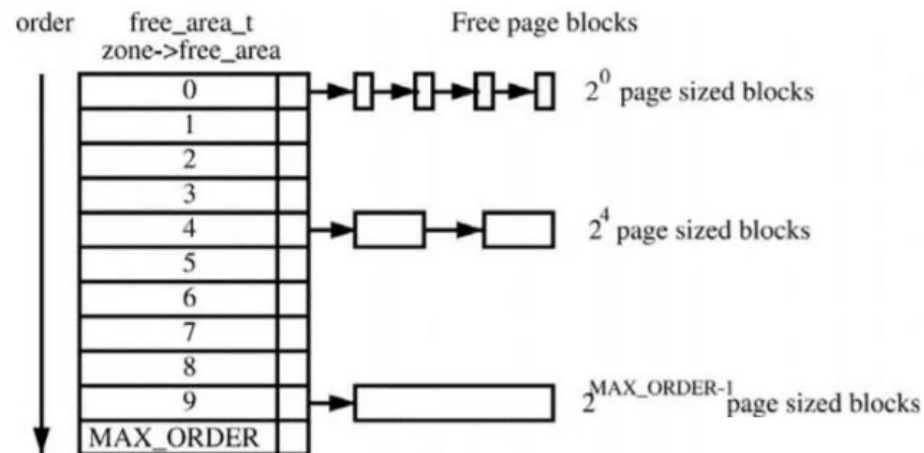
- Buddy算法原理

- 内存块buddy概念的定义

- 1) 两个块大小相同;
- 2) 两个块地址连续;
- 3) 两个块必须是同一个大块中分离出来的;

- 内存分配算法

- 以特定大小的内存块作为内存分配的基本单元 (AliOS 3.1中为32字节)
- 不同大小的空闲内存块通过链表形成结构, 链表的编号与内存块的大小关系为 $2^{id} \times 32$
 - 需要分配32字节的内存块从id=0的链表中取, 分配 $4 \times 32 = 128$ 字节从id=2的链表中取
- 当id号的链表为空时, 则取id+1链表中的内存块, 将其先分成两个大小相同的内存块, 1个进行分配, 一个链入id链表中
 - 例如, 分配4单位 (128字节) 的内存块, 此时2号链表为空, 要取3号, 3号也为空, 取4号, 4号的内存块大小为 $2^4 = 16$ 单位
 - 先将其1分为2, 为2个8单位内存块, 1个链入3号链表, 另一个继续拆分成2个4单位内存块, 1个分配, 1个链入2号链表
- 释放内存块作逆向的合并操作
 - 先从对应大小的id链表中找buddy块, 找到就合并后, 再在id+1链表中寻找合并, 假如无法合并就直接链入



内存管理

• BLK算法原理

- 同样采用链表的方式对大小相同的内存块进行管理
- 分配最相近大小的内存块
- 内存块分配与释放时，不拆分，不合并

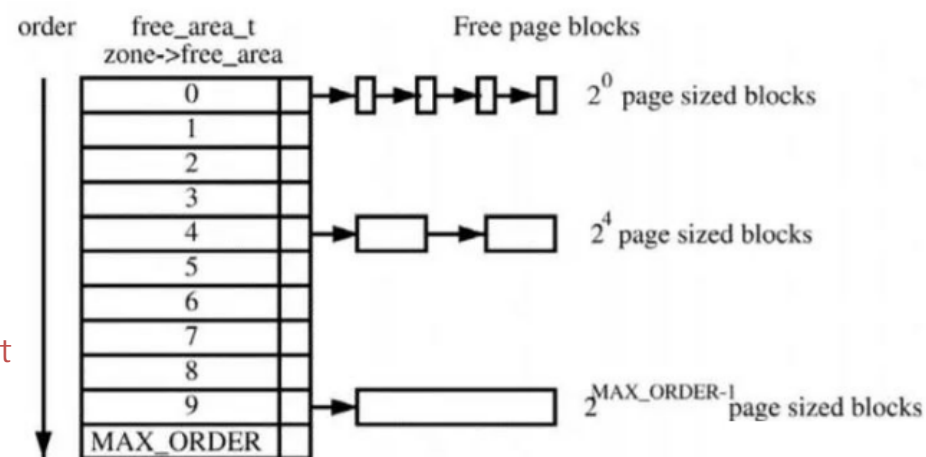
适用于小于32bit

• 内存管理的默认配置

- 打开关闭内存模块功能: RHINO_CONFIG_MM_TLF: 1
- 打开关闭buddy内存模块的维测功能: RHINO_CONFIG_MM_DEBUG: 1
- 最大\最小支持申请的内存大小bit位, 实际大小换算为字节时=1ul << RHINO_CONFIG_MM_MAXMSIZEBIT
 - RHINO_CONFIG_MM_MINISIZEBIT 6
 - RHINO_CONFIG_MM_MAXMSIZEBIT 20
- 打开关闭固定长度小内存块快速申请: RHINO_CONFIG_MM_BLK: 1
- 固定长度小内存块总空间大小 (byte) : RHINO_CONFIG_MM_TLF_BLK_SIZE: 8192
- 设定从blk小内存申请的阈值 (byte) : RHINO_CONFIG_MM_BLK_SIZE: 32

避免内存出现碎片

适用于大于32bit



内存管理常用函数

函数名	描述
aos_malloc()	从系统heap分配内存给用户
aos_zalloc()	从系统heap分配内存给用户，并且将分配的内存初始化为0
aos_calloc()	从系统heap分配内存给用户，并且将分配的内存初始化为0
aos_realloc()	重新调整之前调用 aos_malloc() (aos_calloc() 、 aos_zalloc())所分配的内存块的大小
aos_free()	内存释放函数

内存管理常用函数

◆ aos_malloc()

```
void* aos_malloc (size_t  size)
```

Alloc memory.

参数

[in] **size** size of the mem to malloc.

返回

NULL: error.

◆ aos_realloc()

```
void* aos_realloc (void *  mem,  
                  size_t  size  
                  )
```

Realloc memory.

参数

[in] **mem** current memory address point.

[in] **size** new size of the mem to realloc.

返回

NULL: error.

◆ aos_zalloc()

```
void* aos_zalloc (size_t  size)
```

Alloc memory and clear to zero.

参数

[in] **size** size of the mem to malloc.

返回

NULL: error.

◆ aos_free()

```
void aos_free (void *  mem)
```

Free memory.

参数

[in] **ptr** address point of the mem.

返回

none.

◆ aos_calloc()

```
void* aos_calloc (size_t  nitems,  
                 size_t  size  
                 )
```

Alloc memory and clear to zero.

参数

[in] **nitems** number of items to malloc.

[in] **size** size of one item to malloc.

返回

NULL: error.

例4

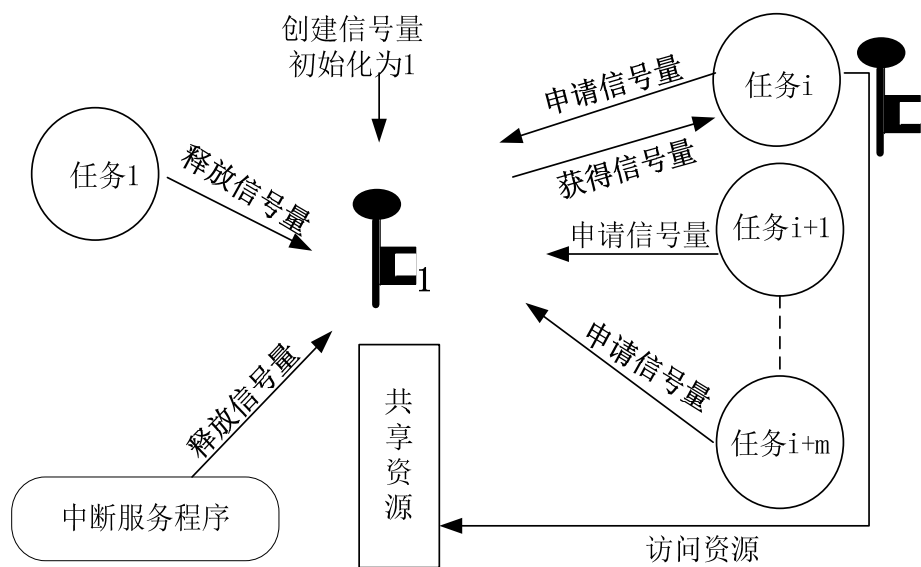
```
#include <stdio.h>
#include <aos/kernel.h>
#include <stdint.h>
#define STRBUFSIZE 1024      分配内存区域1024
int application_start(int argc, char *argv[])
{
    char * str=NULL;
    int count=0;
    printf("nano entry here!\r\n");
    while(1) {
        str=(char*)aos_malloc(STRBUFSIZE);      分配存储区域
        if(str==NULL)
        {
            printf("aos_malloc fail!\r\n");
            continue;
        }
        sprintf(str, "aos_malloc running %d\r\n",count);      打印字符串到缓存里
        printf("%s",str);      输出字符串
        count++;
        aos_free(str);
        aos_msleep(1000);
    };
}
```

信号量

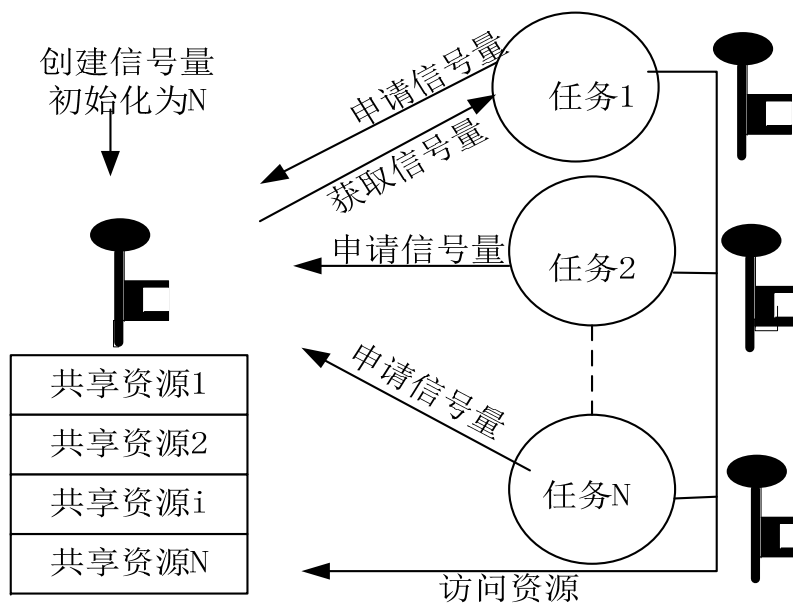
- 60年代中期Edgser Dijkstra发明的一种程序中的约定机制
- 信号量PV操作
 - P (S) : (获取信号量操作)
 - ①若 $S > 0$, 将信号量S的值减1, 即 $S = S - 1$;
 - ②若 $S = 0$, 则该进程进入等待状态, 排入等待队列
 - V (S) : (释放信号量操作)
 - ①若S上有任务等待, 则唤醒任务
 - ②若S上无任务等待, 将信号量S的值加1, 即 $S = S + 1$
- 信号量的作用
 - 实现共享资源的管理, 实现互斥访问
 - 实现任务间的同步
- 与全局变量相比的优点
 - 当信号量不可得时, 任务会放弃cpu, 进入等待状态
 - 当信号量可得时, 任务会被自动唤醒, 不需要去轮询

信号量--共享资源的管理

管理单个资源

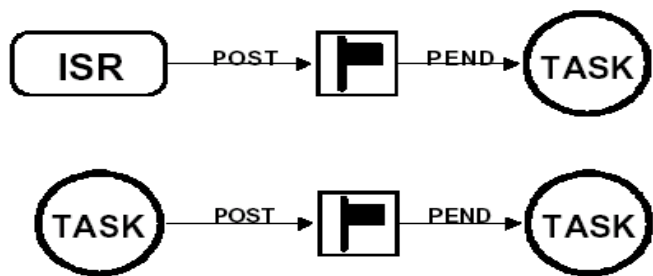


管理多个资源



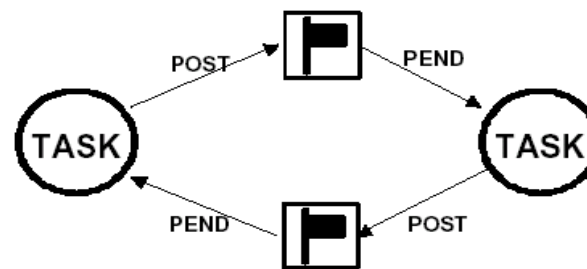
信号量--任务间的同步

控制任务的先后次序



单向同步

A->B



两个任务间双向同步

A->B->A->B->

信号量常用函数

函数名	描述
<code>aos_sem_new()</code>	信号量创建函数，需指定计数值
<code>aos_sem_free()</code>	信号量删除函数
<code>aos_sem_wait()</code>	信号量获取函数，可以指定超时时间
<code>aos_sem_signal()</code>	信号量释放函数，只唤醒阻塞在此信号量上的最高优先级任务
<code>aos_sem_signal_all()</code>	信号量释放函数，唤醒阻塞在此信号量上的所有任务
<code>aos_sem_is_valid()</code>	判断信号量句柄是否合法函数

信号量常用函数

- 创建与删除

◆ aos_sem_new()

```
aos_status_t aos_sem_new ( aos_sem_t * sem,
                           uint32_t   count )
```

大于零可以获取

Alloc a semaphore. Deprecated, not Recommended.

参数

[out] **sem** pointer of semaphore object, semaphore object must be allocated, hdl pointer in aos_sem_t will refer a kernel obj internally.
[in] **count** initial semaphore counter.

返回

0:success.

◆ aos_sem_free()

```
void aos_sem_free ( aos_sem_t * sem )
```

Destroy a semaphore.

参数

[in] **sem** pointer of semaphore object, mem refered by hdl pointer in aos_sem_t will be freed internally.

信号量常用函数

• 获取与释放

◆ aos_sem_wait()

```
aos_status_t aos_sem_wait ( aos_sem_t * sem,
                             uint32_t timeout
                           )
```

信号量时间
信号量句柄

Acquire a semaphore.

参数

[in] **sem** semaphore object, it contains kernel obj pointer which aos_sem_new allocated.
[in] **timeout** waiting until timeout in milliseconds.

返回

0: success.

◆ aos_sem_signal_all()

```
void aos_sem_signal_all ( aos_sem_t * sem )
```

Release all semaphore.

参数

[in] **sem** semaphore object, it contains kernel obj pointer which aos_sem_new allocated.

◆ aos_sem_signal()

```
void aos_sem_signal ( aos_sem_t * sem )
```

扫描优先级高的先释放访问共享资源

Release a semaphore.

参数

[in] **sem** semaphore object, it contains kernel obj pointer which aos_sem_new allocated.

注意:

中断中不能获取信号量, 否则直接获取失败

timeout=0: 获取不到信号量会立即报失败

timeout=AOS_WAIT_FOREVER: 永远等待, 直到获取信号量

信号量常用函数

- 信号量有效性验证

◆ `aos_sem_is_valid()`

```
bool aos_sem_is_valid ( aos_sem_t * sem )
```

This function will check if semaphore is valid. Deprecated, not Recommended.

参数

[in] **sem** pointer to the sem.

返回

false: invalid, true: valid.

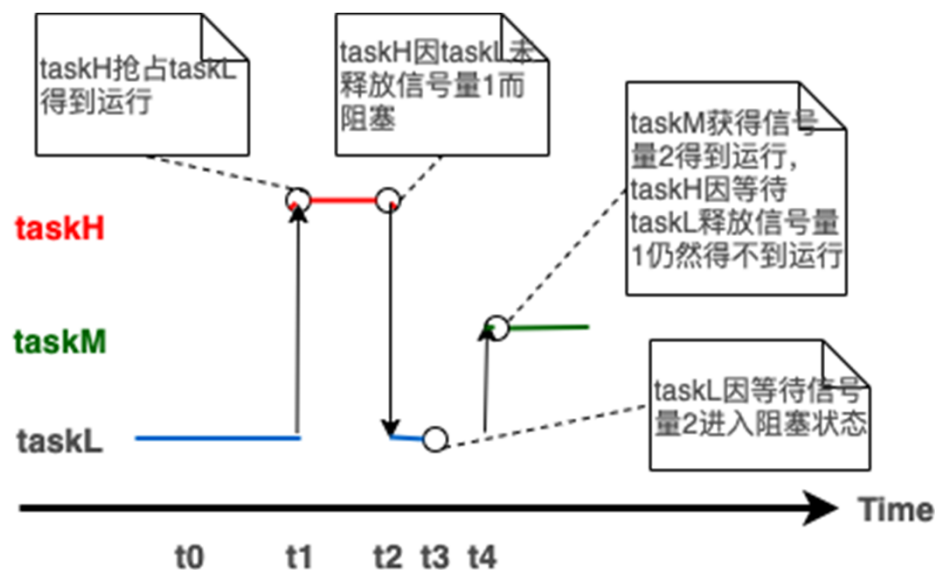
例5

```
#include <stdio.h>
#include <aos/kernel.h>
#include <stdint.h>
aos_task_t htask1;
aos_task_t htask2;
aos_sem_t semcount;
void func1(void * arg)
{
    while(1)
    {
        aos_sem_wait(&semcount, AOS_WAIT_FOREVER);
        printf("task1 running!\r\n");
    }
}
void func2(void * arg)
{
    while(1)
    {
        printf("task2 running!\r\n");
        aos_sem_signal(&semcount);
        aos_msleep(1000);
    }
}
```

```
int application_start(int argc, char *argv[])
{
    printf("nano entry here!\r\n");
    aos_sem_new(&semcount, 0);
    aos_task_new_ext(&htask1, "task1", func1, 0, 8192, 33);
    aos_task_new_ext(&htask2, "task2", func2, 0, 8192, 34);
    while(1) {
        aos_msleep(1000);
    };
}
```

互斥信号量

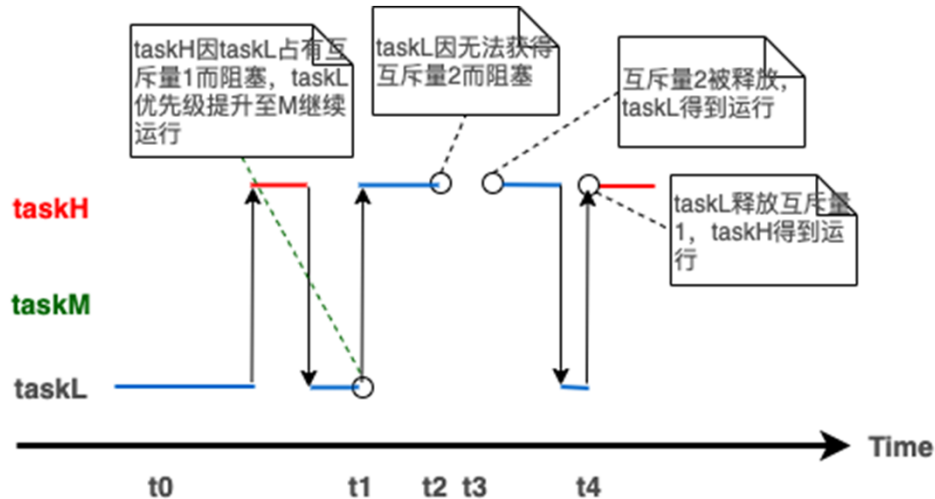
- 在任务使用信号量同步过程中，容易引起优先级反转
- 优先级反转
 - 当高优先级任务访问共享资源时，该资源已被低优先级任务占有
 - 低优先级任务在访问共享资源时又被其它中优先级的任务抢占
 - 造成高优先级任务被许多中优先级任务阻塞
 - 高优先级任务在中优先级任务之后运行



互斥信号量

- 优先级反转的防止：优先级继承算法

- 当低优先级任务占用资源，而高优先级任务要访问此资源时
- 让低优先级任务把优先级提高到与高优先级任务同样的级别（与互斥信号量相关的任务也会被传递）
- 原来的低优先级任务就不会被中间优先级的任务所抢占，从而尽快完成任务，释放所占用资源
- 释放资源后原本提升优先级的任务会被恢复
- 使用互斥信号量mutex可以实现该功能



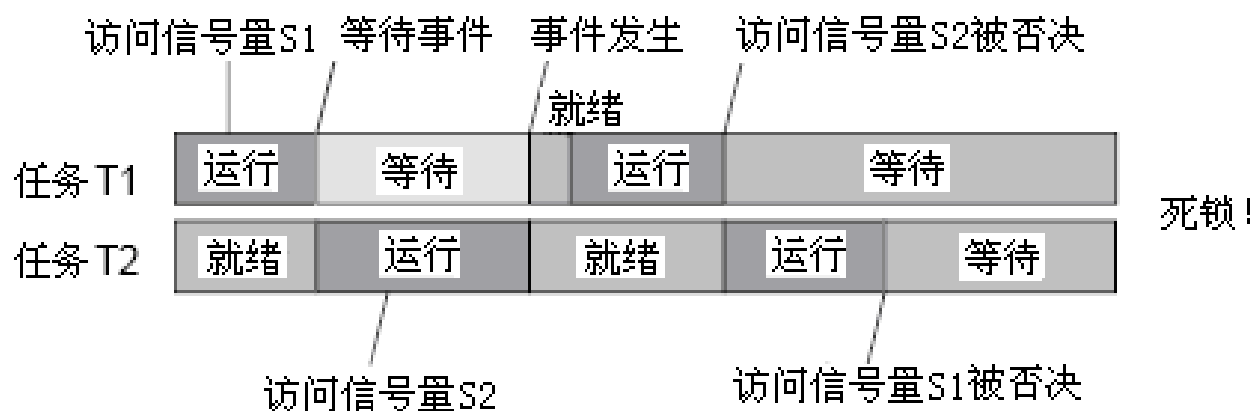
互斥信号量

- 互斥信号量与信号量的比较
- 相同点
 - 可以实现同步与互斥
- 不同点
 - mutex可以实现优先级继承算法，避免优先级反转
 - mutex同一时刻只能被一个任务获取（二值信号量）
 - mutex只能由获取过的任务释放（信号量无此限制）

互斥信号量

• 死锁

- 也称作抱死，指两个任务无限期地互相等待对方控制着的资源而不能执行
- 最简单的防止发生死锁的方法
- 先得到全部需要的资源再做下一步的工作
- 用同样的顺序去申请多个资源
- 释放资源时使用相反的顺序



互斥信号量常用函数—与信号量类似

函数名	描述
aos_mutex_new()	互斥量创建函数
aos_mutex_free()	互斥量删除函数
aos_mutex_lock()	互斥量获取函数
aos_mutex_unlock()	互斥量释放函数
aos_mutex_is_valid()	判断互斥量句柄是否合法函数

互斥信号量常用函数

• 创建与删除

◆ aos_mutex_new()

```
aos_status_t aos_mutex_new (aos_mutex_t * mutex)
```

Alloc a mutex. Deprecated, not Recommended.

参数

[in] **mutex** pointer of mutex object, mutex object must be allocated, hdl pointer in aos_mutex_t will refer a kernel obj internally.

返回

0: success.

◆ aos_mutex_free()

```
void aos_mutex_free (aos_mutex_t * mutex)
```

销毁互斥量，该接口释放互斥量对象的资源，唤醒所有阻塞在该互斥量的任务。

错误处理

如果互斥量对象句柄参数为NULL，或者互斥量对象非法（没有成功创建或者对象类型不是互斥量），则返回-EINVAL。

参数

[in] **mutex** 互斥量对象句柄

返回

无

互斥信号量常用函数

• 获取

◆ aos_mutex_lock()

```
aos_status_t aos_mutex_lock ( aos_mutex_t *  mutex,
                               uint32_t      timeout
                               )
```

锁定互斥量，该接口申请获得一把互斥量锁，常被用于任务之间保护共享资源。 如果该互斥量锁当前没有其他任务持有，则当前任务能够立即获取这把锁并成功返回。 如果该互斥量锁当前被其他任务持有，同时指定AOS_NO_WAIT，则不等待立即返回错误-1。 如果该互斥量锁当前被其他任务持有，同时指定AOS_WAIT_FOREVER，则永远等待直到获得该互斥量锁。 等待时当前任务处于阻塞状态，等待该互斥量锁。

注解

注意内核中允许互斥量嵌套，如果任务再次获得自身持有的互斥量锁，则返回成功。

错误处理

如果互斥量对象句柄参数为NULL， 或者互斥量对象非法（没有成功创建或者对象类型不是互斥量），则返回-EINVAL。

参数

[in] **mutex** 互斥量对象句柄。

[in] **timeout** 超时时间（单位：ms）AOS_WAIT_FOREVER：永远等待。 AOS_NO_WAIT：不等待。

返回

状态码

返回值

0 返回成功，此时当前任务获得这把互斥量锁。

-EINVAL 参数非法

-ETIMEDOUT 等待超时

-1 其他操作

互斥信号量常用函数

• 释放

◆ aos_mutex_unlock()

```
aos_status_t aos_mutex_unlock ( aos_mutex_t * mutex )
```

解锁互斥量，该接口释放自身持有的互斥量锁。如果此时有其他任务阻塞在该互斥量锁上，则从阻塞任务队列中挑选一个优先级最高的任务唤醒，使其继续。

注解

任务只能释放自身持有的互斥量锁，否则返回错误-EPERM。内核中允许互斥量锁嵌套，如果进行过多次的互斥量锁定操作，注意需要进行相同次数的解锁操作，否则其他竞争的任务会一直阻塞。

错误处理

如果互斥量对象句柄参数为NULL，或者互斥量对象非法（没有成功创建或者对象类型不是互斥量），则返回-EINVAL。

参数

[in] **mutex** 互斥量对象句柄。

返回

状态码

返回值

0 返回成功，

-EINVAL 参数非法

-EPERM 权限不够（释放其他任务持有的互斥量锁）

互斥信号量常用函数

- 信号量有效性验证

◆ `aos_mutex_is_valid()`

```
bool aos_mutex_is_valid ( aos_mutex_t * mutex )
```

This function will check if mutex is valid. Deprecated, not Recommended.

参数

[in] **mutex** pointer to the mutex.

返回

false: invalid, true: valid.

事件

- 事件是AliOS内核提供了一种任务间通信方式
- 实现一个任务同时等待多个事件的发生（同时发生或者任一发生）
- 事件组是一个32位的数，每一位都对应一个事件标志
- 事件标志只有两种状态：
 - 1 代表被设置，有事件发生时被设置为1
 - 0 代表被清除，任务获得事件后标志位被清除为0
- 等待的多个事件可以有2种组合方式
 - 任务可以等待事件组中设置的所有事件都发生，即“与”的方式
 - 任务可以等待事件组中设置的任意事件发生，即“或”的方式

事件常用函数（aos没有封装实现）

- 创建
- `kstat_t krhino_event_create(kevent_t *event, const name_t *name, uint32_t flags);`
- 删除
- `kstat_t krhino_event_del(kevent_t *event);`
- 设置
- `kstat_t krhino_event_set(kevent_t *event, uint32_t flags, uint8_t opt);`
 - `opt`: RHINO_AND, RHINO_OR
- 获取
- `kstat_t krhino_event_get(kevent_t *event, uint32_t flags, uint8_t opt, uint32_t *actl_flags, tick_t ticks);`
 - `opt` : RHINO_AND(与)、RHINO_AND_CLEAR（条件成立时清除对应位）、RHINO_OR、RHINO_OR_CLEAR
 - `flags`: 目标比对的位
 - `actl_flags`: 返回参数，实际比较前的标志位
 - `ticks`: 等待的时间，RHINO_WAIT_FOREVER, RHINO_NO_WAIT

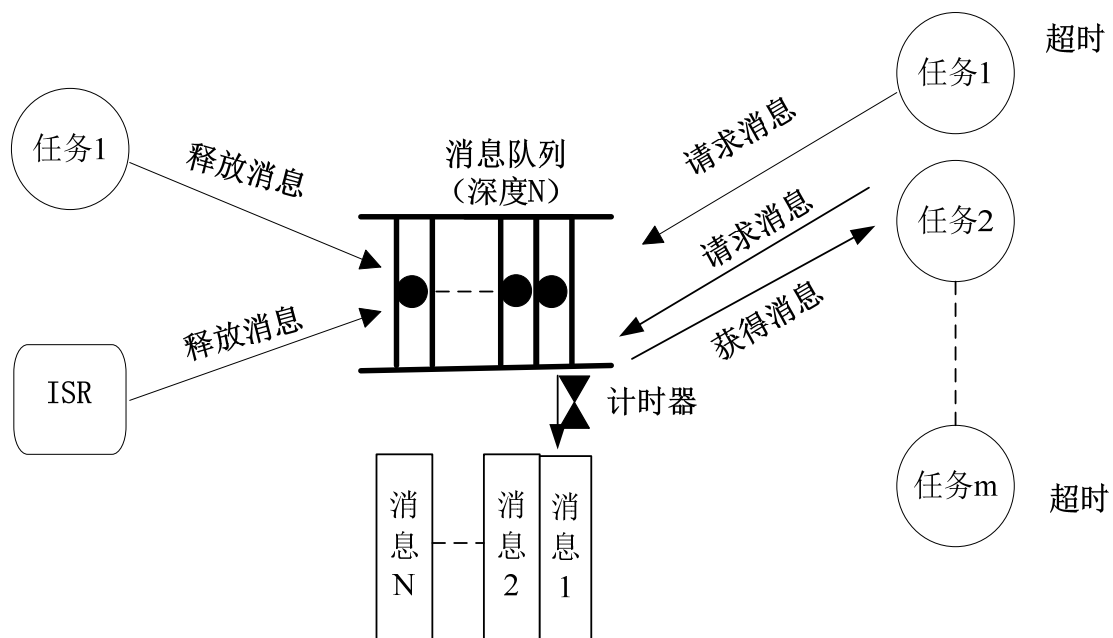
例6

```
#include <stdio.h>
#include <aos/kernel.h>
#include <stdint.h>
#include <k_api.h>
aos_task_t htask1;
aos_task_t htask2;
aos_task_t htask3;
kevent_t event;
void func1(void * arg)
{
    uint32_t flag;
    while(1)
    {
        krhino_event_get(&event,0x03,RHINO_AND_CLEAR,&flag,
            RHINO_WAIT_FOREVER);
        printf("task1 running!\r\n");
    }
}
```

```
void func2(void * arg)
{
    while(1)
    {
        printf("task2 running!\r\n");
        krhino_event_set(&event,0x01,RHINO_OR);
        aos_msleep(1000);
    }
}
void func3(void * arg)
{
    while(1)
    {
        printf("task3 running!\r\n");
        krhino_event_set(&event,0x02,RHINO_OR);
        aos_msleep(2000);
    }
}
int application_start(int argc, char *argv[])
{
    printf("nano entry here!\r\n");
    krhino_event_create(&event,"myevent",0);
    aos_task_new_ext(&htask1,"task1", func1,0, 8192,33);
    aos_task_new_ext(&htask2,"task2", func2,0, 8192,34);
    aos_task_new_ext(&htask3,"task3", func3,0, 8192,35);
    while(1) {
        aos_msleep(1000);
    };
}
```


消息队列

- 消息队列是一种任务间传递数据的机制
- 消息队列使用环形缓冲池（ring buffer）来管理消息的队列缓冲区，并使用类似信号量的机制进行任务间的同步
- 任务通过消息队列可以发送消息，也可以通过它接收消息
- 任务发送的消息会暂存在消息队列中，当接收任务来读时，将暂存的数据传递给接收任务
- 若接收任务在接收数据时，消息队列中无可读数据，任务会阻塞，直到有消息到来解除阻塞而进入就绪状态



消息队列常用函数

函数名	描述
aos_queue_new()	消息队列创建函数
aos_queue_free()	消息队列删除函数
aos_queue_send()	向消息队列发送消息函数
aos_queue_recv()	从消息队列读取消息函数
aos_queue_is_valid()	判断消息队列句柄是否合法
aos_queue_buf_ptr()	获取消息队列消息数据区地址

消息队列常用函数

• 创建与删除

◆ aos_queue_new()

```
aos_status_t aos_queue_new ( aos_queue_t * queue,
                             void *      buf,
                             size_t      size,
                             size_t      max_msg
                             )
```

This function will create a queue. Deprecated, not Recommended.

参数

[in] **queue** pointer to the queue(the space is provided by user).
[in] **buf** buf of the queue(provided by user).
[in] **size** the bytes of the buf.
[in] **max_msg** the max size of the msg.

返回

0: success.

◆ aos_queue_free()

```
void aos_queue_free ( aos_queue_t * queue )
```

This function will delete a queue.

参数

[in] **queue** pointer to the queue.

消息队列常用函数

• 发送消息与接收消息

◆ aos_queue_send()

```
aos_status_t aos_queue_send ( aos_queue_t * queue,
                              void *      msg,
                              size_t      size
                              )
```

This function will send a msg to the front of a queue.

参数

[in] **queue** pointer to the queue.
[in] **msg** msg to send.
[in] **size** size of the msg.

返回

0: success.

◆ aos_queue_recv()

```
aos_status_t aos_queue_recv ( aos_queue_t * queue,
                              uint32_t      ms,
                              void *      msg,
                              size_t *     size
                              )
```

This function will receive msg from a queue.

参数

[in] **queue** pointer to the queue.
[in] **ms** ms to wait before receive.
[out] **msg** buf to save msg.
[out] **size** size of the msg.

返回

0: success.

消息队列常用函数

- 验证队列句柄的有效性

◆ aos_queue_is_valid()

```
bool aos_queue_is_valid ( aos_queue_t * queue )
```

This function will check if queue is valid. Deprecated, not Recommended.

参数

[in] **queue** pointer to the queue.

返回

false: invalid, true: valid.

- 获取队列缓存区指针

◆ aos_queue_buf_ptr()

```
void* aos_queue_buf_ptr ( aos_queue_t * queue )
```

This function will return buf start ptr if queue is initied. Deprecated, not Recommended.

参数

[in] **queue** pointer to the queue.

返回

NULL: error.

例7

```
#include <stdio.h>
#include <aos/kernel.h>
#include <stdint.h>
#define QUEUESIZE 1024
aos_task_t htask1;
aos_task_t htask2;
aos_queue_t que;
uint8_t queBuf[QUEUESIZE];
void func1(void * arg)
{
    int count=0;
    while(1)
    {
        aos_queue_send(&que, &count, sizeof(int));
        count++;
        aos_msleep(500);
    }
}
```

```
void func2(void * arg)
{
    int count;
    unsigned int size;
    while(1)
    {
        aos_queue_rcv(&que, AOS_WAIT_FOREVER, &count, &size);
        printf("count=%d,size=%d\r\n",count,size);
    }
}
int application_start(int argc, char *argv[])
{
    printf("nano entry here!\r\n");
    aos_queue_new(&que,queBuf,QUEUESIZE,sizeof(int));
    aos_task_new_ext(&htask1,"task1", func1,0, 8192,33);
    aos_task_new_ext(&htask2,"task2", func2,0, 8192,34);
    while(1) {
        aos_msleep(1000);
    };
}
```

常用工具函数

- 时间函数

- `uint64_t aos_now (void)`: 返回系统启动至今的ns值
- `uint64_t aos_now_ms (void)`: 返回系统启动至今的ms值
- `void aos_msleep (uint32_t ms)`: 任务休眠ms

- 随机数

- `void aos_srand (uint32_t seed)`: 设定随机数种子
- `int32_t aos_rand (void)`: 产生随机数
 - 产生从x1到x2的随机数: $k = \text{rand}() \% (x2 - x1 + 1) + x1$;

- 系统相关

- `int32_t aos_get_hz (void)`: 返回系统tick的频率
- `void aos_reboot (void)`: 重启系统
- `const char *aos_version_get(void)`: 获取系统版本号

cli组件

- AliOS的CLI (Command-Line Interface) 组件，实现了简单的命令行交互工具
 - 提供基本的系统交互命令
 - 支持用户自定义命令
 - 组件的源代码
 - 原始版本: AliOS-Things-rel_3.1.0\core\cli, 其中cli.c为基本函数, cli_default_command.c为默认实现的命令
 - 配置参数: AliOS-Things-rel_3.1.0\core\cli\include\cli_conf.h
 - aos封装api: AliOS-Things-rel_3.1.0\core\osal\aos\cli.c
- 使用时在aos make menuconfig中, kernel项下, 打开组件选项即可
 - 初始化时, 根据配置, 系统会自动初始化CLI组件, 并对默认命令进行注册

```
Kernel Configuration
e the menu. <Enter> selects submenus ---> (or empty submenus ---
sing <Y> selects a feature, while <N> excludes a feature. Press
ch. Legend: [*] feature is selected [ ] feature is excluded

--*-- Kernel Core (rhino)
[ ] config micro kernel
--*-- Initialize Function
[ ] Power Management
[ ] C++ Support
[ ] Newlib (C-library) adaptation layer
[*] Command-Line Interface
    Command-Line Configuration --->
[ ] Coredump debug Support
--*-- Key-value Storage
    Key-value Storage Configuration --->
--*-- Virtual File System
    Virtual File System Configuration --->
--*-- AOS API Support
    AOS API Configuration --->
[ ] POSIX API Support
√(+)
```

<Select> <Exit> <Help> <Save> <Load>

cli组件

- 添加用户自定义命令

- 包含头文件 `#include <stdint.h>`, `#include "aos/cli.h"`
- 命令中的输出使用: `aos_cli_printf`函数

- 添加单个命令

```
const struct cli_command cmd = { "使用的命令名", "命令说明", 调用的函数}; //构建命令数据结构
ret = aos_cli_register_command(&cmd); //注册命令
if (ret) {
    /* 错误处理 */
}
```

- 添加多个命令

```
const struct cli_command cmds[] = {
    { "test1", "show test1 info", test1_cmd },
    { "test2", "show test2 info", test2_cmd },
};
ret = aos_cli_register_commands(&cmds, sizeof(cmds) / sizeof(struct cli_command));
if (ret) {
    /* 错误处理 */
}
```

- 函数原型

- `void mycmd(char *buf, int32_t len, int32_t argc, char **argv)`
- `buf, len`: 无用, `argc`: 参数个数, `argv`: 参数数组, 参数个数包含命令, 输入的命令存在`argv[0]`中

Yloop

- Yloop是AliOS实现的异步事件处理框架
- 不同于顺序执行的方式，实现了事件驱动处理程序的机制
 - 设定特定的延时，时间到后执行注册函数（针对timer的调度）
 - 注册一个事件，事件发生时，调用注册函数（针对event的调度）
 - 注册一个文件描述符的操作，当可操作时，执行注册的函数（针对IO的调度）
 - 可以回调一个特定的函数（在Yloop上下文中执行）
 - 条件不满足则挂起当前的任务
- 调用aos_loop_run () 函数的任务是Yloop的上下文
 - 函数的调用对上下文有一定的限制
 - 可以有多个任务实现多个Yloop
- Yloop的优点
 - 可以通过异步处理，实现复杂的操作，而不需要构建多个任务，节约资源
 - 所有的操作都在Yloop的同一个任务中顺序执行，不存在互斥与同步的问题，程序结构简单
- Yloop配置
 - 在Network->http

```
Network Configuration
e the menu. <Enter> selects submenus ---> (or empty
sing <Y> selects a feature, while <N> excludes a feat
ch. Legend: [*] feature is selected [ ] feature is

↑(-)
[ ] srtp
[ ] LoRaWAN Stack
   LoRaWAN Stack Configuration ----
[ ] LoRaWAN Stack v4.4.0
[ ] LoRaWAN Stack v4.4.2
-*- lwip
   AOS TCP/IP: Lightweight TCP/IP Stack(LwIP) --->
[ ] LwM2M
[ ] SAL
[ ] at
-*- netmgr
   NETMGR Configuration ----
   Config Network Interface Types: --->
[ ] RTP
-*- http
[ ] uMesh2
↓(+)
```

<Select> <Exit> <Help> <Save>

Yloop常用的函数

- 说明头文件: AliOS-Things-rel_3.1.0\include\utility\yloop-aos\yloop.h
- void aos_loop_run(void):驱动Yloop循环 启动后不会返回的
- void aos_loop_exit(void): 退出Yloop循环
- int aos_post_delayed_action(int ms, aos_call_t action, void *arg): 添加一个延时操作, 操作仅执行一次
 - 处理函数的原型: void (*aos_call_t)(void *arg);
- int aos_register_event_filter(uint16_t type, aos_event_cb cb, void *priv): 注册一个事件, 当事件发生时, 执行函数
 - type: 事件类型, 系统默认定义的系统事件如EV_WIFI, 用户也可以自行定义EV_USER (0x1000), 同一个事件下可以定义不同的code区分处理
 - 处理函数原型: void (*aos_event_cb)(input_event_t *event, void *private_data), 事件类型与code在event结构
 - priv: 传递给处理函数的参数

Yloop常用的函数

- `int aos_post_event(uint16_t type, uint16_t code, unsigned long value)`: 发送事件
 - type事件类型, code事件码, value附加的值
- `int aos_poll_read_fd(int fd, aos_poll_call_t action, void *param)`: IO处理
 - fd: 文件描述符, param参数
 - 读取函数原型: `void (*aos_poll_call_t)(int fd, void *arg);`
- `int aos_schedule_call(aos_call_t action, void *arg)`: 回调函数
 - 在另一个任务中调用该函数, 可以将aos_call_t action放到yloop上下文任务中调用
 - `void (*aos_call_t)(void *arg)`: 回调函数原型

```
typedef struct {  
    /* The time event is generated, auto filled by aos event system */  
    uint32_t time;  
    /* Event type, value < 0x1000 are used by aos system */  
    uint16_t type;  
    /* Defined according to type */  
    uint16_t code;  
    /* Defined according to type/code */  
    unsigned long value;  
    /* Defined according to type/code */  
    unsigned long extra;  
} input_event_t;
```

Yloop常用的函数

- 注意事项

- Yloop中大多数函数必须在对应的Yloop任务（上下文环境）中调用
- 以下函数可以不受上面的限制
- `aos_post_event`
- `aos_schedule_call`
- `aos_loop_schedule_call`
- `aos_loop_schedule_work`
- `aos_cancel_work`

例8

```
#include <stdio.h>
#include <stdint.h>
#include <aos/kernel.h>
#include "aos/yloop.h"
void func(void * arg)
{
    printf("hello yloop\r\n");
    aos_post_delayed_action(1000,func,0);
}
int application_start(int argc, char *argv[])
{
    int count = 0;
    printf("nano entry here!\r\n");
    aos_post_delayed_action(1000,func,0);
    aos_loop_run();    不会返回的，不会return 0
    return 0;
}
```

cJSON

- JSON -- JavaScript Object Notation

- 轻量级的数据格式
- 采用完全独立于编程语言的文本格式来存储和表示数据
- 语法简洁、层次结构清晰，易于人阅读和编写，同时也易于机器解析和生成

- JSON语法规则

- JSON对象是一个无序的"名称/值"键值对的集合：以"{"开始，以"}"结束，允许嵌套使用
- 每个名称和值成对出现，名称和值之间使用":"分隔
- 键值对之间用","分隔在这些字符前后允许存在无意义的空白符
- 对于键值，可以有如下值：
 - 新的json对象
 - 数组：使用"["和"]"表示
 - 数字：直接表示，可以是整数，也可以是浮点数
 - 字符串：使用引号"表示
 - 特殊的值：false、null、true中的一个(必须是小写)

cJSON

- JSON的例子

```
{  
  "name": "mculover666",  
  "age": 22,  
  "weight": 55.5  
  "address":  
  {  
    "country": "China",  
    "zip-code": 111111  
  },  
  "skill": ["c", "Java", "Python"],  
  "student": false  
}
```


cJSON

• cJSON简介

- cJSON是一个使用C语言编写的JSON数据解析器
- 特点：超轻便，可移植，单文件，使用MIT开源协议。
- Github仓库地址：<https://github.com/DaveGamble/cJSON>
- 被集成到AliOS系统中，只有两个主文件：cJSON.h和cJSON.c
- 使用的时候，只需要将这两个文件复制到工程目录，然后包含头文件cJSON.h即可

• cJSON设计思想

- 不是一次将一整段JSON数据解析出来
- 一次只将一条JSON数据解析出来：一次解析一个键值对
- 用结构体 struct cJSON 来表示

• cJSON配置

- Utility->cJSON library

```
Utility
e the menu.  <Enter> selects submenu
sing <Y> selects a feature, while <N
ch.  Legend: [*] feature is selected

[*] cJSON library
-- Mbed TLS 2.16.0
   MbedTLS Configuration --->
[ ] ZLib Support (v1.2.4)
[*] rbtree code

<Select>  <Exit>  <He
```

cJSON

- cJSON结构对数据的表示

- String: 键值对的键名
- type: 用于表示该键值对中值的类型
- valuelstring: 如果键值类型(type)是字符串, 则将该指针指向键值
- valueint: 如果键值类型(type)是整数, 则将该指针指向键值
- valuedouble: 如果键值类型(type)是浮点数, 则将该指针指向键值
- next: 指针: 指向下一个键值对
- prev: 指针指向上一个键值对
- child: 当JSON嵌套时, 值是一个嵌套的JSON数据或者一个数组时, 用child指向它

```
typedef struct cJSON
{
    struct cJSON *next;
    struct cJSON *prev;
    struct cJSON *child;
    int type;
    char *valuelstring;
    int valueint;
    double valuedouble;
    char *string;
} cJSON;
```

cJSON

```
cJSON_AddNullToObject(cJSON * const object, const char * const name);
cJSON_AddTrueToObject(cJSON * const object, const char * const name);
cJSON_AddFalseToObject(cJSON * const object, const char * const name);
cJSON_AddBoolToObject(cJSON * const object, const char * const name, const cJSON_bool boolean);
cJSON_AddNumberToObject(cJSON * const object, const char * const name, const double number);
cJSON_AddStringToObject(cJSON * const object, const char * const name, const char * const string);
cJSON_AddRawToObject(cJSON * const object, const char * const name, const char * const raw);
cJSON_AddObjectToObject(cJSON * const object, const char * const name);
cJSON_AddArrayToObject(cJSON * const object, const char * const name);
```

- cJSON库封装JSON

- 术语

- 指针：指向cJSON结构或者数组结构的指针
- 结点：一个cJSON结构，或者是一个数组结构

- 封装步骤

- 创建头指针
- 创建主结点（代表整个JSON结构）
- 向结点中添加键值对，假如JSON中有嵌套的情况，则创建新结点，将新结点填充完毕后，添加到主结点
- 添加键值时，根据键值不同的类型，调用不同的函数

- cJSON库输出函数

- (char *) cJSON_Print(const cJSON *item);--新JSON结构变为字符串

- 注意事项

- cJSON处理时，采用动态内存分配的方式malloc
- 需要 (void) cJSON_Delete(cJSON *item)释放，调用时会递归释放（有嵌套的cJSON也会一并释放）

cJSON封装与输出例程

```
#include <stdio.h>
#include "cJSON.h"

int main(void)
{
    cJSON* cJSON_test = NULL;
    cJSON* cJSON_address = NULL;
    cJSON* cJSON_skill = NULL;
    char* str = NULL;
    /* 创建一个JSON数据对象(链表头结点) */
    cJSON_test = cJSON_CreateObject();
    /* 添加一条字符串类型的JSON数据(添加一个链表节点) */
    cJSON_AddStringToObject(cJSON_test, "name", "mculover666");
    /* 添加一条整数类型的JSON数据(添加一个链表节点) */
    cJSON_AddNumberToObject(cJSON_test, "age", 22);
    /* 添加一条浮点类型的JSON数据(添加一个链表节点) */
    cJSON_AddNumberToObject(cJSON_test, "weight", 55.5);
```

```
/* 添加一个嵌套的JSON数据 (添加一个链表节点) */
    cJSON_address = cJSON_CreateObject();
    cJSON_AddStringToObject(cJSON_address, "country", "China");
    cJSON_AddNumberToObject(cJSON_address, "zip-code", 111111);
    cJSON_AddItemToObject(cJSON_test, "address", cJSON_address);
    /* 添加一个数组类型的JSON数据(添加一个链表节点) */
    cJSON_skill = cJSON_CreateArray();
    cJSON_AddItemToArray(cJSON_skill, cJSON_CreateString( "C" ));
    cJSON_AddItemToArray(cJSON_skill, cJSON_CreateString( "Java" ));
    cJSON_AddItemToArray(cJSON_skill, cJSON_CreateString( "Python" ));
    cJSON_AddItemToObject(cJSON_test, "skill", cJSON_skill);
    /* 添加一个值为 False 的布尔类型的JSON数据(添加一个链表节点) */
    cJSON_AddFalseToObject(cJSON_test, "student");
    /* 打印JSON对象(整条链表)的所有数据 */
    str = cJSON_Print(cJSON_test);
    printf("%s\n", str);
    return 0;
}
```

cJSON

- cJSON库解析JSON

- 一次一个剥离链表节点(键值对)

- 解析步骤

- 解析整个JSON结构: (cJSON *) cJSON_Parse(const char *value)
 - 获取键值对, 用cJSON结构表示: (cJSON *) cJSON_GetObjectItem(const cJSON * const object, const char * const string), 获得结构中的值就得到键值信息
 - 如果JSON数据的值是数组, 使用下面的两个API提取数据
 - (int) cJSON_GetArraySize(const cJSON *array);
 - (cJSON *) cJSON_GetArrayItem(const cJSON *array, int index);

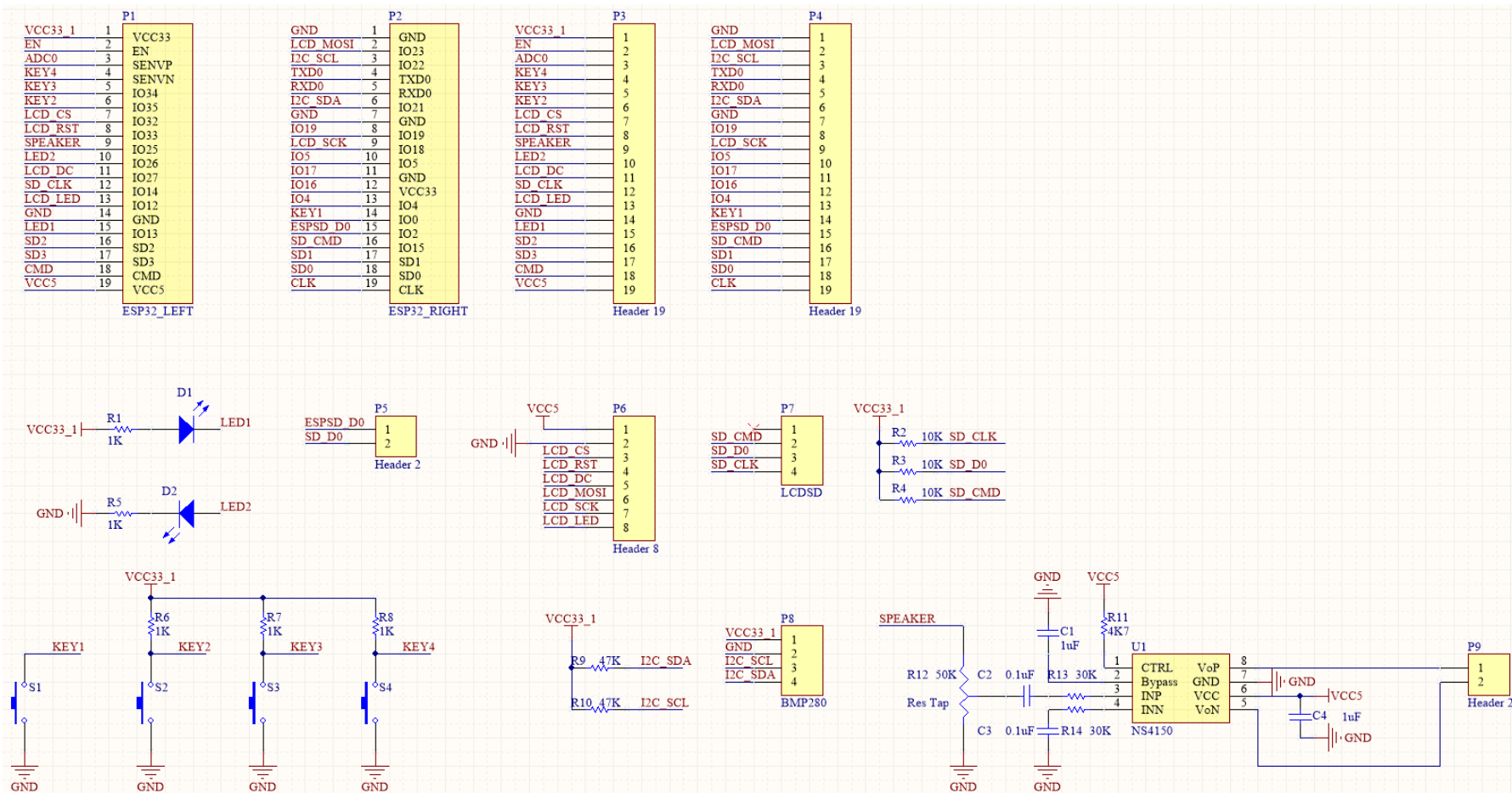
cJSON解析例程

```
#include <stdio.h>
#include "cJSON.h"
char *message =
"{
  \"name\": \"mculover666\",
  \"age\": 22,
  \"weight\": 55.5,
  \"address\":
  {
    \"country\": \"China\",
    \"zip-code\": 111111
  },
  \"skill\": [\"c\", \"Java\", \"Python\"],
  \"student\": false
}";
int main(void)
{
  cJSON* cJSON_test = NULL;
  cJSON* cJSON_name = NULL;
  cJSON* cJSON_age = NULL;
  cJSON* cJSON_weight = NULL;
  cJSON* cJSON_address = NULL;
  cJSON* cJSON_address_country = NULL;
  cJSON* cJSON_address_zipcode = NULL;
  cJSON* cJSON_skill = NULL;
  cJSON* cJSON_student = NULL;
  int skill_array_size = 0, i = 0;
  cJSON* cJSON_skill_item = NULL;
  /* 解析整段JSON数据 */
  cJSON_test = cJSON_Parse(message);
  if(cJSON_test == NULL)
  {
    printf("parse fail.\n");
    return -1;
  }
}
```

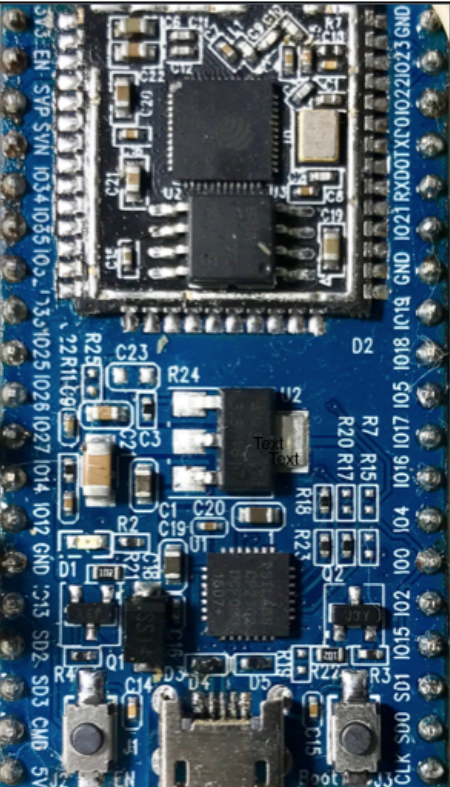
```
/* 依次根据名称提取JSON数据（键值对） */
cJSON_name = cJSON_GetObjectItem(cJSON_test, "name");
cJSON_age = cJSON_GetObjectItem(cJSON_test, "age");
cJSON_weight = cJSON_GetObjectItem(cJSON_test, "weight");
printf("name: %s\n", cJSON_name->valuestring);
printf("age: %d\n", cJSON_age->valueint);
printf("weight: %.1f\n", cJSON_weight->valuedouble);
/* 解析嵌套json数据 */
cJSON_address = cJSON_GetObjectItem(cJSON_test, "address");
cJSON_address_country = cJSON_GetObjectItem(cJSON_address, "country");
cJSON_address_zipcode = cJSON_GetObjectItem(cJSON_address, "zip-code");
printf("address-country: %s\naddress-zipcode: %d\n", cJSON_address_country->valuestring, cJSON_address_zipcode->valueint);
/* 解析数组 */
cJSON_skill = cJSON_GetObjectItem(cJSON_test, "skill");
skill_array_size = cJSON_GetArraySize(cJSON_skill);
printf("skill: [");
for(i = 0; i < skill_array_size; i++)
{
  cJSON_skill_item = cJSON_GetArrayItem(cJSON_skill, i);
  printf("%s, ", cJSON_skill_item->valuestring);
}
printf("\b]\n");

/* 解析布尔型数据 */
cJSON_student = cJSON_GetObjectItem(cJSON_test, "student");
if(cJSON_student->valueint == 0)
{
  printf("student: false\n");
}
else
{
  printf("student: error\n");
}
return 0;
}
```

开发板硬件原理图



GPIO编程

					3.3V		GND																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																										</
--	--	--	--	--	------	---	-----	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	--	----

GPIO API

- GPIO相关文件

- AliOS-Things-rel_3.1.0\include\aos\hal\gpio.h—头文件
- AliOS-Things-rel_3.1.0\core\mk\syscall\usyscall\hal_gpio_usyscall.c—系统中间层
- AliOS-Things-rel_3.1.0\platform\mcu\esp32\hal\gpio.c—板级实现
- AliOS-Things-rel_3.1.0\platform\mcu\esp32\bsp\include\driver\include\driver\gpio.h—gpio号对应关系

- 常用API函数

- int32_t hal_gpio_init(gpio_dev_t *gpio);--初始化
- int32_t hal_gpio_output_high(gpio_dev_t *gpio);--输出高电平
- int32_t hal_gpio_output_low(gpio_dev_t *gpio);--输出低电平
- int32_t hal_gpio_output_toggle(gpio_dev_t *gpio);--电平反转
- int32_t hal_gpio_input_get(gpio_dev_t *gpio, uint32_t *value);--读取电平
- int32_t hal_gpio_enable_irq(gpio_dev_t *gpio, gpio_irq_trigger_t trigger, gpio_irq_handler_t handler, void *arg);--启用中断功能
- int32_t hal_gpio_disable_irq(gpio_dev_t *gpio);--关闭中断
- int32_t hal_gpio_clear_irq(gpio_dev_t *gpio);--清中断
- int32_t hal_gpio_finalize(gpio_dev_t *gpio);--恢复IO口默认设置

- 包含头文件

- #include "aos/hal/gpio.h"
- #include <aos/kernel.h>

结构体

```
typedef struct {  
    uint8_t  port;  /** AliOS中的GPIO号*/  
    gpio_config_t config; /**IO口模式 */  
    void      *priv; /**没有使用*/  
} gpio_dev_t;
```

例9

```
#include <stdio.h>
#include <aos/kernel.h>
#include "aos/hal/gpio.h"

int application_start(int argc, char *argv[])
{
    printf("nano entry here!\r\n");
    gpio_dev_t Led1;
    gpio_dev_t Led2;
    Led1.port=13;
    Led1.config=OUTPUT_PUSH_PULL;
    Led2.port=26;
    Led2.config=OUTPUT_PUSH_PULL;
    hal_gpio_init(&Led1);
    hal_gpio_init(&Led2);
    hal_gpio_output_high(&Led1);
    hal_gpio_output_low(&Led2);
    while(1) {
        hal_gpio_output_toggle(&Led1);
        hal_gpio_output_toggle(&Led2);
        aos_msleep(1000);
    };
}
```

ADC API

- ADC相关文件

- AliOS-Things-rel_3.1.0\include\aos\hal\adc.h—头文件
- AliOS-Things-rel_3.1.0\core\mk\syscall\usyscall\hal_adc_usyscall.c—系统中间层
- AliOS-Things-rel_3.1.0\platform\mcu\esp32\hal\adc.c—板级实现

- 常用API函数

- int32_t hal_adc_init adc_dev_t *adc);--初始化
- int32_t hal_adc_value_get(adc_dev_t *adc, uint32_t *output, uint32_t timeout);--获取ADC的结果
 - timeout为等待的最长毫秒值，在esp32实现中，无作用
- int32_t hal_adc_finalize(adc_dev_t *adc);--恢复ADC默认设置，为空函数

- 包含头文件

- #include "aos/hal/adc.h"
- #include <aos/kernel.h>

```
#define HAL_WAIT_FOREVER 0xFFFFFFFFU
```

```
typedef struct {  
    uint32_t sampling_cycle;  
    /**< sampling period in number of ADC clock cycles */  
} adc_config_t;
```

//支持的ADC通道

```
static adc_obj_t obj[]={  
    {GPIO_NUM_36, ADC1_CHANNEL_0},  
    {GPIO_NUM_37, ADC1_CHANNEL_1},  
    {GPIO_NUM_38, ADC1_CHANNEL_2},  
    {GPIO_NUM_39, ADC1_CHANNEL_3},  
    {GPIO_NUM_32, ADC1_CHANNEL_4},  
    {GPIO_NUM_33, ADC1_CHANNEL_5},  
    {GPIO_NUM_34, ADC1_CHANNEL_6},  
    {GPIO_NUM_35, ADC1_CHANNEL_7},};
```

```
typedef struct {  
    uint8_t      port;  /**< adc port */  
    adc_config_t config; /**< adc config */  
    void         *priv; /**< priv data */  
} adc_dev_t;
```

DAC API

- DAC相关文件

- AliOS-Things-rel_3.1.0\include\aos\hal\dac.h—头文件
- AliOS-Things-rel_3.1.0\core\mk\syscall\usyscall\hal_dac_usyscall.c—系统中间层
- AliOS-Things-rel_3.1.0\platform\mcu\esp32\hal\dac.c—板级实现

- 常用API函数

- `int32_t hal_dac_init(dac_dev_t *dac);`--初始化, 无用, **dac可以为0, 下同**
- `int32_t hal_dac_start(dac_dev_t *dac, uint32_t channel);`--启动DAC模块
 - **DAC_CHANNEL_1=1—GPIO25, DAC_CHANNEL_2=2—GPIO26**
- `int32_t hal_dac_stop(dac_dev_t *dac, uint32_t channel);`--关闭DAC模块
- `int32_t hal_dac_set_value(dac_dev_t *dac, uint32_t channel, uint32_t data);`--设置输出电压值
- `int32_t hal_dac_get_value(dac_dev_t *dac, uint32_t channel);`--获取输出电压值
 - 输出时会把值存储在静态变量中, 读取时返回静态变量的值
- `int32_t hal_dac_finalize(dac_dev_t *dac);`--恢复DAC默认设置, 无用, 空函数
- 包含头文件
- `#include "aos/hal/dac.h"`
- `#include <aos/kernel.h>`

PWM API

```
typedef struct {  
    uint8_t    port; /*IO口 */  
    pwm_config_t config; /*配置结构*/  
    void      *priv; /*N.A*/  
} pwm_dev_t;
```

```
typedef struct {  
    float  duty_cycle; /*占空比0~1*/  
    uint32_t freq; /*pwm频率*/  
} pwm_config_t;
```

- PWM相关文件

- AliOS-Things-rel_3.1.0\include-aos\hal\pwm.h—头文件
- AliOS-Things-rel_3.1.0\platform\mcu\esp32\hal\pwm.c—板级实现

- 常用API函数

- int32_t hal_pwm_init(pwm_dev_t *pwm);--初始化pwm
- int32_t hal_pwm_start(pwm_dev_t *pwm);--启动pwm
- int32_t hal_pwm_stop(pwm_dev_t *pwm);--关闭pwm
- int32_t hal_pwm_para_chg(pwm_dev_t *pwm, pwm_config_t para);--修改pwm参数, 没有实现
- int32_t hal_pwm_finalize(pwm_dev_t *pwm);--恢复pwm默认设置
- 包含头文件
- #include "aos/hal/pwm.h"
- #include <aos/kernel.h>

- 使用方法

- pwm功能最多同时使用4组
- 调用次序
 - hal_pwm_init-> hal_pwm_start-> hal_pwm_stop->hal_pwm_init-> hal_pwm_start-> hal_pwm_stop.....

UART API

- UART相关文件

- AliOS-Things-rel_3.1.0\include\aos\hal\uart.h—头文件
- AliOS-Things-rel_3.1.0\core\mk\syscall\usyscall\hal_uart_usyscall.c—系统中间层
- AliOS-Things-rel_3.1.0\platform\mcu\esp32\hal\uart.c—板级实现

- 常用API函数

- int32_t hal_uart_init(aos_uart_dev_t *uart);--初始化uart
- int32_t hal_uart_send(aos_uart_dev_t *uart, const void *data, uint32_t size, uint32_t timeout);—发送
 - uart: 结构, data: 发送的数据缓存, size: 发送数据字节数, timeout: 最大的等待时间, 实现中无用, 返回0成功。
- int32_t hal_uart_recv_ll(aos_uart_dev_t *uart, void *data, uint32_t expect_size, uint32_t *recv_size, uint32_t timeout);--接收
 - uart: 结构, data: 接收的数据缓存, expect_size : 希望接收数据字节数, recv_size: 实际接收的字节数, timeout: 最大的等待时间tick值, HAL_WAIT_FOREVER为永远等待, 返回0成功。
- int32_t hal_uart_finalize(aos_uart_dev_t *uart); --恢复uart默认设置, 空函数

- 包含头文件

- #include "aos/hal/uart.h"
- #include <aos/kernel.h>

UART配置结构

```
typedef struct {  
    uint8_t      port;//端口  
    uart_config_t config;//配置结构  
    void        *priv;//NA  
} uart_dev_t;
```

```
typedef struct {  
    uint32_t      baud_rate;    /**< Uart baud rate */  
    hal_uart_data_width_t data_width; /**< Uart data width */  
    hal_uart_parity_t parity;    /**< Uart parity check mode */  
    hal_uart_stop_bits_t stop_bits; /**< Uart stop bit mode */  
    hal_uart_flow_control_t flow_control; /**< Uart flow control mode */  
    hal_uart_mode_t mode;        /**< Uart send/receive mode */  
} uart_config_t;
```

```
typedef enum {  
    DATA_WIDTH_5BIT,  
    DATA_WIDTH_6BIT,  
    DATA_WIDTH_7BIT,  
    DATA_WIDTH_8BIT,  
    DATA_WIDTH_9BIT  
} hal_uart_data_width_t;
```

```
typedef enum {  
    STOP_BITS_1,  
    STOP_BITS_2  
} hal_uart_stop_bits_t;
```

```
typedef enum {  
    FLOW_CONTROL_DISABLED,  
    FLOW_CONTROL_CTS,  
    FLOW_CONTROL_RTS,  
    FLOW_CONTROL_CTS_RTS  
} hal_uart_flow_control_t;
```

```
typedef enum {  
    NO_PARITY,  
    ODD_PARITY,  
    EVEN_PARITY  
} hal_uart_parity_t;
```

```
typedef enum {  
    MODE_TX,  
    MODE_RX,  
    MODE_TX_RX  
} hal_uart_mode_t;
```

例10

```
#include <stdio.h>
#include <string.h>
#include <aos/kernel.h>
#include "aos/hal/uart.h"
```

```
uart_dev_t uart0;
int application_start(int argc, char *argv[])
{
    int count = 0;
    uint32_t ret;
    char SendBuf[]="hello uart\r\n";
    printf("nano entry here!\r\n");
```

```
//uart0 setting
memset(&uart0, 0, sizeof(uart0));
uart0.port = 0;
uart0.config.baud_rate = 115200;
uart0.config.data_width = DATA_WIDTH_8BIT;
uart0.config.flow_control = FLOW_CONTROL_DISABLED;
uart0.config.mode = MODE_TX_RX;
uart0.config.parity = NO_PARITY;
uart0.config.stop_bits = STOP_BITS_1;
ret = hal_uart_init(&uart0);
if (ret != 0) {
    return -1;
}
while(1) {
    printf("hello world! count %d \r\n", count++);
    hal_uart_send(&uart0,SendBuf,sizeof(SendBuf),HAL_WAIT_FOREVER);
    aos_msleep(1000);
};
}
```


[-] 阿里云

