腾讯物联网终端操作系统

开发指南

产品文档

腾讯云

【联系我们】

我们致力于为您提供个性化的售前购买咨询服务，及相应的技术售后服务，任何问题请联系 4009100100。

开发指南

最近更新时间：2020-11-03 17:21:19

概述

TencentOS tiny 是面向物联网（IOT）领域的操作系统，由一个实现精简的实时操作系统（RTOS）内核，以及丰富的物联网组件组成。

基础内核组件

- **系统管理**：系统管理模块，主要提供了内核的初始化、内核运行启动，中断进入/退出流程托管、系统调度锁定及解锁等功能。
- **任务管理**：提供了任务的创建、删除、睡眠、取消睡眠、挂起、恢复、优先级修改、主动放弃 CPU 等功能。
- **任务间通信**：提供互斥量、信号量、队列、事件等常用任务间通信机制。
- **内存管理**：提供了基于堆的动态内存管理，以及静态内存块管理机制。
- **时间管理**：提供了获取/设置系统时钟滴答数、系统时钟滴答数与墙上时钟时间转换、基于墙上时钟时间的任务睡眠等机制。
- **软件定时器**：提供了软件定时器的创建、删除、启动、停止等机制。
- **时间片轮转机制**：TencentOS tiny 内核在提供可抢占式调度内核基础上，还提供了按时间片轮转的 robin 机制。
- **内核基础组件**：提供了消息队列、字符流先入先出队列等机制。
- **功耗管理**：提供了 CPU 低功耗运行模式设置、低功耗设备注册、板级唤醒闹钟设置等机制。

基础内核

系统管理

概述

系统管理模块提供了几个接口，用以初始化/启动 TencentOS tiny 内核、锁定/解锁系统调度等。

API 接口说明

接口名称	说明
k_err_t tos_knl_init(void);	初始化内核
k_err_t tos_knl_start(void);	启动运行内核，开始第一个任务调度

接口名称	说明
int tos_knl_is_running(void);	判断内核是否已启动运行
tos_knl_irq_enter();	此函数应该在中断调用函数的最前端被调用
tos_knl_irq_leave();	此函数应该在中断调用函数的尾端被调用
k_err_t tos_knl_sched_lock(void);	锁定系统调度，此函数被调用并返回 K_ERR_NONE 时，系统调度会被锁定，系统调度器不再进行任务的切换
k_err_t tos_knl_sched_unlock(void);	解锁系统调度，允许任务切换

任务管理

概述

TencentOS tiny 内核是单地址空间的可抢占式实时内核，TencentOS tiny 内核不提供进程模型，任务对应线程的概念，是最小的调度运行体，也是最小的资源持有单位。

任务的本质是一个拥有独立栈空间的可调度运行实体，用户可以在任务的入口函数中编写自己的业务逻辑；多个任务之间可以通过系统提供的任务间通信机制进行同步或者信息传递等操作；每个任务都有优先级，高优先级任务可以抢占低优先级任务的运行。

API 接口说明

创建任务的系统 API 接口为 tos_task_create，接口原型如下：

```
k_err_t tos_task_create(k_task_t *task,
char *name,
k_task_entry_t entry,
void *arg,
k_prio_t prio,
k_stack_t *stk_base,
size_t stk_size,
k_timeslice_t timeslice);
```

详细的 API 参数说明如下：

接口名称	说明

接口名称	说明
task	一个 k_task_t 类型的指针
name	指向任务名字符串的指针
entry	任务体运行的函数入口
arg	传递给任务入口函数的参数
prio	任务优先级
stk_base	任务在运行时使用的栈空间的起始地址
stk_size	任务的栈空间大小
timeslice	时间片轮转机制下当前任务的时间片大小

编程实例

1. 在 tos_config.h 中，配置最大任务优先级 TOS_CFG_TASK_PRIO_MAX：
   #define TOS_CFG_TASK_PRIO_MAX 10u
2. 配置每秒钟的系统滴答数 TOS_CFG_CPU_TICK_PER_SECOND：
   #define TOS_CFG_CPU_TICK_PER_SECOND 1000u
3. 编写 main.c 示例代码：

```
#include "tos.h" // 添加TencentOS tiny内核接口头文件
#include "mcu_init.h" // 包含mcu初始化头文件，里面有board_init等板级启动代码函数原型声明

#define STK_SIZE_TASK_PRIO4 512 // 优先级为4的任务栈大小为512
#define STK_SIZE_TASK_PRIO5 1024 // 优先级为5的任务栈大小为1024

k_stack_t stack_task_prio4[STK_SIZE_TASK_PRIO4]; // 优先级为4的任务栈空间
k_stack_t stack_task_prio5[STK_SIZE_TASK_PRIO5]; // 优先级为5的任务栈空间

k_task_t task_prio4; // 优先级为4的任务体
k_task_t task_prio5; // 优先级为5的任务体

extern void entry_task_prio4(void *arg); // 优先级为4的任务体入口函数
extern void entry_task_prio5(void *arg); // 优先级为5的任务体入口函数

uint32_t arg_task_prio4_array[3] = { // 优先级为4的任务体入口函数入参
```

```
1, 2, 3,
};

char *arg_task_prio5_string = "arg for task_prio5"; // 优先级为5的任务体入口函数入参

static void dump_uint32_array(uint32_t *array, size_t len)
{
size_t i = 0;

for (i = 0; i < len; ++i) {
printf("%d\t", array[i]);
}
printf("\n\n");

}

void entry_task_prio4(void *arg)
{
uint32_t *array_from_main = (uint32_t *)arg; // 捕获调用者传入的参数

printf("array from main:\n");
dump_uint32_array(array_from_main, 3); // dump传入的参数（数组）

while (K_TRUE) {
printf("task_prio4 body\n"); // 任务运行体，不断打印这条信息
tos_task_delay(1000); // 睡眠1000个系统时钟滴答（以下记作systick），因为TOS_CFG_CPU_TICK_PER_SECOND为1000，也就是一秒钟会有1000个systick，因此睡眠1000个systick就是睡眠了1秒。
}
}

void entry_task_prio5(void *arg)
{
int i = 0;
char *string_from_main = (char *)arg;
printf("string from main:\n");
printf("%s\n\n", string_from_main); // 打印出调用者传入的字符串参数

while (K_TRUE) {
```

```
if (i == 2) {
printf("i = %d\n", i); // i为2时，挂起task_prio4，task_prio4停止运行
tos_task_suspend(&task_prio4);
} else if (i == 4) {
printf("i = %d\n", i); // i为4时，恢复task_prio4的运行
tos_task_resume(&task_prio4);
} else if (i == 6) {
printf("i = %d\n", i); // i为6时，删除task_prio4，task_prio4不再运行
tos_task_destroy(&task_prio4);
}
printf("task_prio5 body\n");
tos_task_delay(1000);
++i;
}
}

int main(void)
{
board_init(); // 执行板级初始化代码，初始化串口等外设。
tos_knl_init(); // 初始化TencentOS tiny内核
// 创建一个优先级为4的任务
(void)tos_task_create(&task_prio4, "task_prio4", entry_task_prio4,
(void *)(&arg_task_prio4_array[0]), 4,
stack_task_prio4, STK_SIZE_TASK_PRIO4, 0);
// 创建一个优先级为5的任务
(void)tos_task_create(&task_prio5, "task_prio5", entry_task_prio5,
(void *)arg_task_prio5_string, 5,
stack_task_prio5, STK_SIZE_TASK_PRIO5, 0);
// 开始内核调度
tos_knl_start();
}
```

运行效果

```
array from main:
1 2 3

task_prio4 body
```

```
string from main:
arg for task_prio5

task_prio5 body
task_prio4 body
task_prio5 body
task_prio4 body
i = 2
task_prio5 body
task_prio5 body
i = 4
task_prio4 body
task_prio5 body
task_prio4 body
task_prio5 body
task_prio4 body
i = 6
task_prio5 body
task_prio5 body
task_prio5 body
task_prio5 body
task_prio5 body
task_prio5 body
task_prio5 body
```

任务间通信

互斥量

概述

互斥量又称互斥锁，一般用于共享资源的互斥排他性访问保护。

互斥量在任意时刻处于且仅会处于解锁或锁定状态，当一个任务获取到一把锁后（互斥量锁定），其他任务再尝试获得这把锁时会失败或进入阻塞状态，当该任务释放持有的锁时（互斥量解锁），会唤醒一个正阻塞等待此互斥量的任务，被唤醒的任务将会获取这把锁。

在多任务运行环境中，有些共享资源不具有多线程可重入性，对于这类不希望被多任务同时访问的资源（临界资源），可以采用互斥量来进行保护。

编程实例

1. 在 tos_config.h 中，配置互斥量组件开关 TOS_CFG_MUTEX_EN：

   #define TOS_CFG_MUTEX_EN 1u

2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_WRITER 512
#define STK_SIZE_TASK_READER 512

k_stack_t stack_task_writer[STK_SIZE_TASK_WRITER];
k_stack_t stack_task_reader[STK_SIZE_TASK_READER];

k_task_t task_writer;
k_task_t task_reader;

extern void entry_task_writer(void *arg);
extern void entry_task_reader(void *arg);

k_mutex_t critical_resource_locker;

// 一片临界区内存
static uint32_t critical_resource[3];

static void write_critical_resource(int salt)
{
size_t i = 0;
// 此函数每次向共享内存中按递增顺序写入三个无符号整数
printf("writting critical resource:\n");
for (i = 0; i < 3; ++i) {
printf("%d\t", salt + i);
critical_resource[i] = salt + i;
}
printf("\n");
}

void entry_task_writer(void *arg)
```

```
{
size_t salt = 0;
k_err_t err;

while (K_TRUE) {
// 在向临界区写入数据之前，先尝试获取临界区保护锁
err = tos_mutex_pend(&critical_resource_locker);
if (err == K_ERR_NONE) {
// 成功获取锁之后，向临界区写入数据
write_critical_resource(salt);
// 写完数据后，释放互斥锁
tos_mutex_post(&critical_resource_locker);
}
tos_task_delay(1000);
++salt;
}
}

static void read_critical_resource(void)
{
size_t i = 0;

// 从临界区读取数据
printf("reading critical resource:\n");
for (i = 0; i < 3; ++i) {
printf("%d\t", critical_resource[i]);
}
printf("\n");
}

void entry_task_reader(void *arg)
{
k_err_t err;

while (K_TRUE) {
// 读取临界区数据之前，先尝试获取临界区保护锁
err = tos_mutex_pend(&critical_resource_locker);
if (err == K_ERR_NONE) {
```

```
// 成功获取锁之后，从临界区读取数据
read_critical_resource();
// 读取数据完毕后，释放互斥锁
tos_mutex_post(&critical_resource_locker);
}
tos_task_delay(1000);
}
}

int main(void)
{
board_init();
tos_knl_init();
// 创建临界区保护互斥锁
tos_mutex_create(&critical_resource_locker);
(void)tos_task_create(&task_writer, "writer", entry_task_writer, NULL,
4, stack_task_writer, STK_SIZE_TASK_WRITER, 0);
(void)tos_task_create(&task_reader, "reader", entry_task_reader, NULL,
4, stack_task_reader, STK_SIZE_TASK_READER, 0);
tos_knl_start();
}
```

运行效果

```
writting critical resource:
0 1 2
reading critical resource:
0 1 2
writting critical resource:
1 2 3
reading critical resource:
1 2 3
writting critical resource:
2 3 4
reading critical resource:
2 3 4
writting critical resource:
```

```
3 4 5
reading critical resource:
3 4 5
writting critical resource:
4 5 6
reading critical resource:
4 5 6
writting critical resource:
5 6 7
reading critical resource:
5 6 7
writting critical resource:
6 7 8
reading critical resource:
6 7 8
writting critical resource:
7 8 9
reading critical resource:
7 8 9
```

信号量

概述

信号量是一种实现任务间同步的机制，一般用于多个任务间有限资源竞争访问。

一个信号量中持有一个整形数值，用以表示可用资源的数量。

- 当一个信号量的可用资源数量大于0时，任务尝试获取该信号量成功，信号量的可用资源数减一。
- 当一个信号量的可用资源数等于0时，任务尝试获取该信号量失败或进入阻塞状态。
- 信号量的这一模式，当可用资源数为1时，可将其用于资源的互斥访问，或者解决生产者 – 消费者问题中的资源生产 – 消费问题。

编程实例

1. 在tos_config.h中，配置信号量组件开关 TOS_CFG_SEM_EN：
   ```
   #define TOS_CFG_SEM_EN 1u
   ```
2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_PRODUCER 512
#define STK_SIZE_TASK_CONSUMER 512

k_stack_t stack_task_producer[STK_SIZE_TASK_PRODUCER];
k_stack_t stack_task_consumer[STK_SIZE_TASK_CONSUMER];

k_task_t task_producer;
k_task_t task_consumer;

extern void entry_task_producer(void *arg);
extern void entry_task_consumer(void *arg);

k_mutex_t buffer_locker;
k_sem_t full;
k_sem_t empty;

#define RESOURCE_COUNT_MAX 3

struct resource_st {
int cursor;
uint32_t buffer[RESOURCE_COUNT_MAX];
} resource = { 0, {0} };

static void produce_item(int salt)
{
printf("produce item:\n");

printf("%d", salt);
resource.buffer[resource.cursor++] = salt;
printf("\n");
}

void entry_task_producer(void *arg)
{
```

```
size_t salt = 0;
k_err_t err;

while (K_TRUE) {
err = tos_sem_pend(&empty, TOS_TIME_FOREVER);
if (err != K_ERR_NONE) {
continue;
}
err = tos_mutex_pend(&buffer_locker);
if (err != K_ERR_NONE) {
continue;
}

produce_item(salt);

tos_mutex_post(&buffer_locker);
tos_sem_post(&full);
tos_task_delay(1000);
++salt;
}
}

static void consume_item(void)
{
printf("cosume item:\n");
printf("%d\t", resource.buffer[--resource.cursor]);
printf("\n");
}

void entry_task_consumer(void *arg)
{
k_err_t err;

while (K_TRUE) {
err = tos_sem_pend(&full, TOS_TIME_FOREVER);
if (err != K_ERR_NONE) {
continue;
}
```

```
tos_mutex_pend(&buffer_locker);
if (err != K_ERR_NONE) {
continue;
}

consume_item();

tos_mutex_post(&buffer_locker);
tos_sem_post(&empty);
tos_task_delay(2000);
}
}

int main(void)
{
board_init();
tos_knl_init();
tos_mutex_create(&buffer_locker);
tos_sem_create(&full, 0);
tos_sem_create(&empty, RESOURCE_COUNT_MAX);
(void)tos_task_create(&task_producer, "producer", entry_task_producer, NULL,
4, stack_task_producer, STK_SIZE_TASK_PRODUCER, 0);
(void)tos_task_create(&task_consumer, "consumer", entry_task_consumer, NULL,
4, stack_task_consumer, STK_SIZE_TASK_CONSUMER, 0);
tos_knl_start();
}
```

运行效果

```
produce iterm:
0
cosume iterm:
0
produce iterm:
1
produce iterm:
2
```

```
cosume iterm:
2
produce iterm:
3
produce iterm:
4
cosume iterm:
4
produce iterm:
5
cosume iterm:
5
produce iterm:
6
cosume iterm:
6
produce iterm:
7
cosume iterm:
7
produce iterm:
8
cosume iterm:
8
produce iterm:
9
cosume iterm:
9
produce iterm:
10
```

事件

概述

- 事件提供了一种任务间实现同步和信息传递的机制。一个事件中包含了一个旗标，这个旗标的每一位表示一个“事件”。
- 一个任务可以等待一个或者多个“事件”的发生，其他任务在一定的业务条件下可以通过写入特定“事件”唤醒等待此“事件”的任务，实现一种类似信号的编程范式。

编程实例

1. 在 tos_config.h 中，配置事件组件开关 TOS_CFG_EVENT_EN：
   #define TOS_CFG_EVENT_EN 1u
2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_LISTENER 512
#define STK_SIZE_TASK_TRIGGER 512

k_stack_t stack_task_listener1[STK_SIZE_TASK_LISTENER];
k_stack_t stack_task_listener2[STK_SIZE_TASK_LISTENER];
k_stack_t stack_task_trigger[STK_SIZE_TASK_TRIGGER];

k_task_t task_listener1;
k_task_t task_listener2;
k_task_t task_trigger;

extern void entry_task_listener1(void *arg);
extern void entry_task_listener2(void *arg);
extern void entry_task_trigger(void *arg);

const k_event_flag_t event_eeny = (k_event_flag_t)(1 << 0);
const k_event_flag_t event_meeny = (k_event_flag_t)(1 << 1);
const k_event_flag_t event_miny = (k_event_flag_t)(1 << 2);
const k_event_flag_t event_moe = (k_event_flag_t)(1 << 3);

k_event_t event;

void entry_task_listener1(void *arg)
{
k_event_flag_t flag_match;
k_err_t err;

while (K_TRUE) {
// 此任务监听四个事件，因为使用了TOS_OPT_EVENT_PEND_ALL选项，因此必须是四个事件同时到达此任
```

```
务才会被唤醒
err = tos_event_pend(&event, event_eeny | event_meeny | event_miny | event_moe,
&flag_match, TOS_TIME_FOREVER, TOS_OPT_EVENT_PEND_ALL | TOS_OPT_EVENT_PEND_CLR);
if (err == K_ERR_NONE) {
printf("entry_task_listener1:\n");
printf("eeny, meeny, miny, moe, they all come\n");
}
}
}

void entry_task_listener2(void *arg)
{
k_event_flag_t flag_match;
k_err_t err;

while (K_TRUE) {
// 此任务监听四个事件，因为使用了TOS_OPT_EVENT_PEND_ANY选项，因此四个事件任意一个到达（包括四个事件同时到达）任务就会被唤醒
err = tos_event_pend(&event, event_eeny | event_meeny | event_miny | event_moe,
&flag_match, TOS_TIME_FOREVER, TOS_OPT_EVENT_PEND_ANY | TOS_OPT_EVENT_PEND_CLR);
if (err == K_ERR_NONE) {
printf("entry_task_listener2:\n");
// 有事件到达，判断具体是哪个事件
if (flag_match == event_eeny) {
printf("eeny comes\n");
}
if (flag_match == event_meeny) {
printf("meeny comes\n");
}
if (flag_match == event_miny) {
printf("miny comes\n");
}
if (flag_match == event_moe) {
printf("moe comes\n");
}
if (flag_match == (event_eeny | event_meeny | event_miny | event_moe)) {
printf("all come\n");
}
```

```
}
}
}

void entry_task_trigger(void *arg)
{
int i = 1;

while (K_TRUE) {
if (i == 2) {
printf("entry_task_trigger:\n");
printf("eeny will come\n");
// 发送eeny事件，task_listener2会被唤醒
tos_event_post(&event, event_eeny);
}
if (i == 3) {
printf("entry_task_trigger:\n");
printf("meeny will come\n");
// 发送eeny事件，task_listener2会被唤醒
tos_event_post(&event, event_meeny);
}
if (i == 4) {
printf("entry_task_trigger:\n");
printf("miny will come\n");
// 发送eeny事件，task_listener2会被唤醒
tos_event_post(&event, event_miny);
}
if (i == 5) {
printf("entry_task_trigger:\n");
printf("moe will come\n");
// 发送eeny事件，task_listener2会被唤醒
tos_event_post(&event, event_moe);
}
if (i == 6) {
printf("entry_task_trigger:\n");
printf("all will come\n");
// 同时发送四个事件，因为task_listener1的优先级高于task_listener2，因此这里task_listener1会被唤醒
tos_event_post(&event, event_eeny | event_meeny | event_miny | event_moe);
```

```
}
tos_task_delay(1000);
++i;
}
}

int main(void)
{
board_init();
tos_knl_init();
tos_event_create(&event, (k_event_flag_t)0u);
// 这里task_listener1的优先级比task_listener2高，因此在task_trigger发送所有事件时，task_listener1会被唤醒
// 读者可以尝试将task_listener1优先级修改为5（比task_listener2低），此设置下，在task_trigger发送所有事件时，task_listener2将会被唤醒。
(void)tos_task_create(&task_listener1, "listener1", entry_task_listener1, NULL,
3, stack_task_listener1, STK_SIZE_TASK_LISTENER, 0);
(void)tos_task_create(&task_listener2, "listener2", entry_task_listener2, NULL,
4, stack_task_listener2, STK_SIZE_TASK_LISTENER, 0);
(void)tos_task_create(&task_trigger, "trigger", entry_task_trigger, NULL,
4, stack_task_trigger, STK_SIZE_TASK_TRIGGER, 0);
tos_knl_start();
}
```

运行效果

```
entry_task_trigger:
eeny will come
entry_task_listener2:
eeny comes
entry_task_trigger:
meeny will come
entry_task_listener2:
meeny comes
entry_task_trigger:
miny will come
entry_task_listener2:
```

```
miny comes
entry_task_trigger:
moe will come
entry_task_listener2:
moe comes
entry_task_trigger:
all will come
entry_task_listener1:
eeny, meeny, miny, moe, they all come
```

队列

概述

队列提供了一种任务间实现同步和数据传递的机制。事件只能用于任务间传递某类“事件”是否发生的信号，无法传递更为复杂的数据，队列弥补了事件的这一不足，可以在任务间传递不定长度的消息。

编程实例

1. 在 tos_config.h 中，配置队列组件开关 TOS_CFG_QUEUE_EN：
   #define TOS_CFG_QUEUE_EN 1u
2. 在 tos_config.h 中，配置消息队列组件开关 TOS_CFG_MSG_EN：
   #define TOS_CFG_MSG_EN 1u
3. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_RECEIVER 512
#define STK_SIZE_TASK_SENDER 512

#define PRIO_TASK_RECEIVER_HIGHER_PRIO 4
#define PRIO_TASK_RECEIVER_LOWER_PRIO (PRIO_TASK_RECEIVER_HIGHER_PRIO + 1)

k_stack_t stack_task_receiver_higher_prio[STK_SIZE_TASK_RECEIVER];
k_stack_t stack_task_receiver_lower_prio[STK_SIZE_TASK_RECEIVER];
k_stack_t stack_task_sender[STK_SIZE_TASK_SENDER];

k_task_t task_receiver_higher_prio;
k_task_t task_receiver_lower_prio;
```

```
k_task_t task_sender;

k_queue_t queue;

extern void entry_task_receiver_higher_prio(void *arg);
extern void entry_task_receiver_lower_prio(void *arg);
extern void entry_task_sender(void *arg);

void entry_task_receiver_higher_prio(void *arg)
{
k_err_t err;
void *msg_received;
size_t msg_size;

while (K_TRUE) {
err = tos_queue_pend(&queue, &msg_received, &msg_size, TOS_TIME_FOREVER);
if (err == K_ERR_NONE) {
printf("entry_task_receiver_higher_prio:\n");
printf("message body: %s\n", (char *)msg_received);
printf("message size: %d\n", msg_size);
}
}
}

void entry_task_receiver_lower_prio(void *arg)
{
k_err_t err;
void *msg_received;
size_t msg_size;

while (K_TRUE) {
err = tos_queue_pend(&queue, &msg_received, &msg_size, TOS_TIME_FOREVER);
if (err == K_ERR_NONE) {
printf("entry_task_receiver_lower_prio:\n");
printf("message body: %s\n", (char *)msg_received);
printf("message size: %d\n", msg_size);
}
}
```

```
}

void entry_task_sender(void *arg)
{
int i = 1;
char *msg_to_one_receiver = "message for one receiver[with highest priority]";
char *msg_to_all_receiver = "message for all receivers";

// 此任务不断通过队列queue发送消息
while (K_TRUE) {
if (i == 2) {
printf("entry_task_sender:\n");
printf("send a message to one receiver, and shoud be the highest priority one\n");
// 发送消息并唤醒一个等待任务，唤醒的应该是等待任务中优先级最高的
tos_queue_post(&queue, msg_to_one_receiver, strlen(msg_to_one_receiver));
}
if (i == 3) {
printf("entry_task_sender:\n");
printf("send a message to all recevier\n");
// 发送消息并唤醒所有正在等待的任务
tos_queue_post_all(&queue, msg_to_all_receiver, strlen(msg_to_all_receiver));
}
if (i == 4) {
printf("entry_task_sender:\n");
printf("send a message to one receiver, and shoud be the highest priority one\n");
// 发送消息并唤醒一个等待任务，唤醒的应该是等待任务中优先级最高的
tos_queue_post(&queue, msg_to_one_receiver, strlen(msg_to_one_receiver));
}
if (i == 5) {
printf("entry_task_sender:\n");
printf("send a message to all recevier\n");
// 发送消息并唤醒所有正在等待的任务
tos_queue_post_all(&queue, msg_to_all_receiver, strlen(msg_to_all_receiver));
}
tos_task_delay(1000);
++i;
}
}
```

```
int main(void)
{
board_init();
tos_knl_init();
tos_queue_create(&queue);
// task_receiver_higher_prio任务的优先级较高
(void)tos_task_create(&task_receiver_higher_prio, "receiver_higher_prio",
entry_task_receiver_higher_prio, NULL, PRIO_TASK_RECEIVER_HIGHER_PRIO,
stack_task_receiver_higher_prio, STK_SIZE_TASK_RECEIVER, 0);
// task_receiver_lower_prio任务的优先级较低
(void)tos_task_create(&task_receiver_lower_prio, "receiver_lower_prio",
entry_task_receiver_lower_prio, NULL, PRIO_TASK_RECEIVER_LOWER_PRIO,
stack_task_receiver_lower_prio, STK_SIZE_TASK_RECEIVER, 0);
(void)tos_task_create(&task_sender, "sender", entry_task_sender, NULL,
4, stack_task_sender, STK_SIZE_TASK_SENDER, 0);
tos_knl_start();
}
```

运行效果

```
entry_task_trigger:
send a message to one receiver, and shoud be the highest priority one
entry_task_receiver_higher_prio:
message body: message for one receiver[with highest priority]
message size: 47
entry_task_trigger:
send a message to all recevier
entry_task_receiver_higher_prio:
message body: message for all receivers
message size: 25
entry_task_receiver_lower_prio:
message body: message for all receivers
message size: 25
entry_task_trigger:
send a message to one receiver, and shoud be the highest priority one
entry_task_receiver_higher_prio:
```

```
message body: message for one receiver[with highest priority]
message size: 47
entry_task_trigger:
send a message to all recevier
entry_task_receiver_higher_prio:
message body: message for all receivers
message size: 25
entry_task_receiver_lower_prio:
message body: message for all receivers
message size: 25
```

内存管理

动态内存

概述

动态内存管理模块，提供了一套动态管理系统内存的机制，支持用户动态的申请、释放不定长内存块。

编程实例

1. 在 tos_config.h 中，配置动态内存组件开关 TOS_CFG_MMHEAP_EN：
   #define TOS_CFG_MMHEAP_EN 1u
2. 在 tos_config.h 中，配置动态内存池大小：
   #define TOS_CFG_MMHEAP_POOL_SIZE 0x2000
3. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

extern void entry_task_demo(void *arg);

void entry_task_demo(void *arg)
{
```

```
void *p = K_NULL, *p_aligned = K_NULL;
int i = 0;

while (K_TRUE) {
if (i == 1) {
p = tos_mmheap_alloc(0x30);
if (p) {
printf("alloc: %x\n", (cpu_addr_t)p);
}
} else if (i == 2) {
if (p) {
printf("free: %x\n", p);
tos_mmheap_free(p);
}
} else if (i == 3) {
p = tos_mmheap_alloc(0x30);
if (p) {
printf("alloc: %x\n", (cpu_addr_t)p);
}
} else if (i == 4) {
p_aligned = tos_mmheap_aligned_alloc(0x50, 16);
if (p_aligned) {
printf("aligned alloc: %x\n", (cpu_addr_t)p_aligned);
if ((cpu_addr_t)p_aligned % 16 == 0) {
printf("%x is 16 aligned\n", (cpu_addr_t)p_aligned);
} else {
printf("should not happen\n");
}
}
} else if (i == 5) {
p = tos_mmheap_realloc(p, 0x40);
if (p) {
printf("realloc: %x\n", (cpu_addr_t)p);
}
} else if (i == 6) {
if (p) {
tos_mmheap_free(p);
}
```

```
if (p_aligned) {
tos_mmheap_free(p_aligned);
}
}

tos_task_delay(1000);
++i;
}
}

int main(void)
{
board_init();
tos_knl_init();
(void)tos_task_create(&task_demo, "receiver_higher_prio", entry_task_demo, NULL,
4, stack_task_demo, STK_SIZE_TASK_DEMO, 0);
tos_knl_start();
}
```

运行效果

```
alloc: 20000c8c
free: 20000c8c
alloc: 20000c8c
aligned alloc: 20000cc0
20000cc0 is 16 aligned
realloc: 20000d14
```

静态内存

概述

静态内存管理模块，提供了一套管理静态内存块的机制，支持用户申请、释放定长的内存块。

API 接口说明

创建静态内存池接口：

```
k_err_t tos_mmblk_pool_create(k_mmblk_pool_t *mbp, void *pool_start, size_t blk_num, size_t b
lk_size);
```

详细的 API 参数说明如下：

参数名称	参数说明
mbp	静态内存池句柄
pool_start	静态内存池起始地址
blk_num	内存池将要划分的内存块个数
blk_size	每个内存块的大小

编程实例

1. 在 tos_config.h 中，配置静态内存组件开关 TOS_CFG_MMBLK_EN：
   #define TOS_CFG_MMBLK_EN 1u
2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

#define MMBLK_BLK_NUM 5
#define MMBLK_BLK_SIZE 0x20

k_mmblk_pool_t mmblk_pool;

// 需要管理的静态内存池
uint8_t mmblk_pool_buffer[MMBLK_BLK_NUM * MMBLK_BLK_SIZE];

// 记录从内存池中分配到的地址
void *p[MMBLK_BLK_NUM] = { K_NULL };

extern void entry_task_demo(void *arg);
```

```
void entry_task_demo(void *arg)
{
void *p_dummy = K_NULL;
k_err_t err;
int i = 0;

printf("mmblk_pool has %d blocks, size of each block is 0x%x\n", 5, 0x20);
// 从内存池中获取所有的block
for (; i < MMBLK_BLK_NUM; ++i) {
err = tos_mmblk_alloc(&mmblk_pool, &p[i]);
if (err == K_ERR_NONE) {
printf("%d block alloced: 0x%x\n", i, p[i]);
} else {
printf("should not happen\n");
}
}

// 前文逻辑已经将所有可用block分配完毕，继续分配会返回K_ERR_MMBLK_POOL_EMPTY错误码
err = tos_mmblk_alloc(&mmblk_pool, &p_dummy);
if (err == K_ERR_MMBLK_POOL_EMPTY) {
printf("blocks exhausted, all blocks is alloced\n");
} else {
printf("should not happen\n");
}

// 将前文分配得到的所有block归还到池中
for (i = 0; i < MMBLK_BLK_NUM; ++i) {
err = tos_mmblk_free(&mmblk_pool, p[i]);
if (err != K_ERR_NONE) {
printf("should not happen\n");
}
}
// 前文的归还动作中已经将所有的block归还到池中，继续规范会返回K_ERR_MMBLK_POOL_FULL错误码
err = tos_mmblk_free(&mmblk_pool, p[0]);
if (err == K_ERR_MMBLK_POOL_FULL) {
printf("pool is full\n");
} else {
printf("should not happen\n");
```

```
}
}

int main(void)
{
board_init();
tos_knl_init();
// 创建静态内存池
tos_mmblk_pool_create(&mmblk_pool, mmblk_pool_buffer, MMBLK_BLK_NUM, MMBLK_BLK_SIZ
E);
(void)tos_task_create(&task_demo, "receiver_higher_prio", entry_task_demo, NULL,
4, stack_task_demo, STK_SIZE_TASK_DEMO, 0);
tos_knl_start();
}
```

运行效果

```
mmblk_pool has 5 blocks, size of each block is 0x20
0 block alloced: 0x20000974
1 block alloced: 0x20000994
2 block alloced: 0x200009b4
3 block alloced: 0x200009d4
4 block alloced: 0x200009f4
blocks exhausted, all blocks is alloced
pool is full
```

时间管理

概述

时间管理，提供了一种与时间相关的函数，可以获取/设置系统时钟滴答数（systick）、systick 与毫秒单位之间互相转化、按毫秒、墙上时钟等单位进行任务睡眠的功能。

编程实例

1. 配置每秒钟的系统滴答数 TOS_CFG_CPU_TICK_PER_SECOND：
   #define TOS_CFG_CPU_TICK_PER_SECOND 1000u
2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

extern void entry_task_demo(void *arg);

void entry_task_demo(void *arg)
{
k_time_t ms;
k_tick_t systick, after_systick;

// 因为TOS_CFG_CPU_TICK_PER_SECOND为1000，也就是一秒钟会有1000个systick，因此1000个systick等于1000毫秒。
systick = tos_millisec2tick(2000);
printf("%d millisec equals to %lld ticks\n", 2000, systick);

ms = tos_tick2millisec(1000);
printf("%lld ticks equals to %d millisec\n", (k_tick_t)1000, ms);

systick = tos_systick_get();
printf("before sleep, systick is %lld\n", systick);

tos_msleep(2000);

after_systick = tos_systick_get();
printf("after sleep %d ms, systick is %lld\n", 2000, after_systick);

printf("milliseconds sleep is about: %d\n", tos_ticks2millisec(after_systick - systick));
}

int main(void)
{
```

```
board_init();
tos_knl_init();
(void)tos_task_create(&task_demo, "receiver_higher_prio", entry_task_demo, NULL,
4, stack_task_demo, STK_SIZE_TASK_DEMO, 0);
tos_knl_start();
}
```

运行效果

```
2000 millisec equals to 2000 ticks
1000 ticks equals to 1000 millisec
before sleep, systick is 7
after sleep 2000 ms, systick is 2009
milliseconds sleep is about: 2002
```

软件定时器

概述

软件定时器提供了一套从软件层次实现的定时器机制，相对应的概念是硬件定时器。用户可以创建一系列的软件定时器，并指定软件定时器到期的条件以及执行回调，当软件定时器到期时会执行注册的回调。

API 接口说明

```
#define TOS_CFG_TIMER_AS_PROC 1u
```

```
k_err_t tos_timer_create(k_timer_t *tmr,
k_tick_t delay,
k_tick_t period,
k_timer_callback_t callback,
void *cb_arg,
k_opt_t opt)
```

详细的 API 参数说明如下：

接口名称	接口说明
tmr	软件定时器句柄
delay	该定时器延迟多久后执行

接口名称	接口说明
period	一个定时器的执行周期
callback	定时器到期后的执行回调
cb_arg	执行回调的入参
opt	此 opt 的传入主要是界定 tmr 的属性

编程实例

1. 在 tos_config.h 中，配置软件定时器组件开关 TOS_CFG_TIMER_EN：
   #define TOS_CFG_TIMER_EN 1000u
2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

extern void entry_task_demo(void *arg);

void oneshot_timer_cb(void *arg)
{
printf("this is oneshot timer callback, current systick: %lld\n", tos_systick_get());
}

void periodic_timer_cb(void *arg)
{
printf("this is periodic timer callback, current systick: %lld\n", tos_systick_get());
}

void entry_task_demo(void *arg)
{
k_timer_t oneshot_tmr;
```

```
k_timer_t periodic_tmr;

// 这是一个一次性的timer，且超时时间是3000个tick之后
tos_timer_create(&oneshot_tmr, 3000, 0, oneshot_timer_cb, K_NULL, TOS_OPT_TIMER_ONESHO
T);
// 这是一个周期性的timer，第一次超时时间是2000个tick之后，之后按3000个tick为周期执行回调
tos_timer_create(&periodic_tmr, 2000, 3000, periodic_timer_cb, K_NULL, TOS_OPT_TIMER_PERI
ODIC);

printf("current systick: %lld\n", tos_systick_get());
tos_timer_start(&oneshot_tmr);
tos_timer_start(&periodic_tmr);

while (K_TRUE) {
tos_task_delay(1000);
}
}

int main(void)
{
board_init();
tos_knl_init();
(void)tos_task_create(&task_demo, "receiver_higher_prio", entry_task_demo, NULL,
4, stack_task_demo, STK_SIZE_TASK_DEMO, 0);
tos_knl_start();
}
```

运行效果

```
current systick: 0
this is periodic timer callback, current systick: 2001
this is oneshot timer callback, current systick: 3001
this is periodic timer callback, current systick: 5001
this is periodic timer callback, current systick: 8001
this is periodic timer callback, current systick: 11001
this is periodic timer callback, current systick: 14001
```

时间片轮转机制

概述

TencentOS tiny 操作系统内核是一个抢占式内核，抢占式内核的特点是，如果最高优先级的任务不放弃 CPU（调用tos_task_delay、tos_task_yeild 等主动放权，或者任务间同步通信机制的 pend 接口等），那么 CPU 将会一直被此任务独占。

假设这样一种场景：系统中包含多个同等优先级的任务，且这几个任务体中都没有放弃 CPU 的行为，则会出现的情况是，这几个任务始终只有第一个被得到调度的那个在运行，因为第一个得到调度的任务体中不会主动放弃 CPU，而其他任务优先级上与其相等无法抢占。此种场景下，其他任务会因得不到 CPU 而陷入饥饿状态。

时间片轮转机制提供了按时间片占用调度的策略，可以解决上述场景下的任务饥饿问题。

编程实例

1. 在 tos_config.h 中，配置时间片轮转组件开关 TOS_CFG_ROUND_ROBIN_EN：
   #define TOS_CFG_ROUND_ROBIN_EN 1u
2. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512
#define STK_SIZE_TASK_SAMPLE 512

/*
此代码中创建了两个同等优先级（PRIO_TASK_DEMO）的任务task_demo1、task_demo2，两个任务体中做的操作是不断对各自的计数器（demo1_counter、demo2_counter）做自增操作（没有放弃CPU的操作），时间片分别为timeslice_demo1、timeslice_demo2。
同时创建了一个优先级比task_demo1、task_demo2较高的采样任务task_sample，此任务不间歇的对两个计数器进行采样。在开启时间片轮转的情况下，task_demo1、task_demo2得到运行的时间比例应该是timeslice_demo1与timeslice_demo2的比例，那么demo1_counter和demo2_counter值的比例应该也差不多是timeslice_demo1与timeslice_demo2的比例。
*/

// task_demo1和task_demo2的优先级
#define PRIO_TASK_DEMO 4
// 采样任务的优先级
#define PRIO_TASK_SAMPLE (PRIO_TASK_DEMO - 1)
```

```
// task_demo1的时间片，在tos_task_create时传入
const k_timeslice_t timeslice_demo1 = 10;
// task_demo2的时间片，在tos_task_create时传入
const k_timeslice_t timeslice_demo2 = 20;

k_stack_t stack_task_demo1[STK_SIZE_TASK_DEMO];
k_stack_t stack_task_demo2[STK_SIZE_TASK_DEMO];
k_stack_t stack_task_sample[STK_SIZE_TASK_SAMPLE];

k_task_t task_demo1;
k_task_t task_demo2;
k_task_t task_sample;

extern void entry_task_demo1(void *arg);
extern void entry_task_demo2(void *arg);
extern void entry_task_sample(void *arg);

uint64_t demo1_counter = 0;
uint64_t demo2_counter = 0;

void entry_task_demo1(void *arg)
{
while (K_TRUE) {
++demo1_counter;
}
}

void entry_task_demo2(void *arg)
{
while (K_TRUE) {
++demo2_counter;
}
}

void entry_task_sample(void *arg)
{
while (K_TRUE) {
++demo2_counter;
```

```
printf("demo1_counter: %lld\n", demo1_counter);
printf("demo2_counter: %lld\n", demo2_counter);
printf("demo2_counter / demo1_counter = %f\n",
(double)demo2_counter / demo1_counter);
printf("should almost equals to:\n");
printf("timeslice_demo2 / timeslice_demo1 = %f\n\n", (double)timeslice_demo2 / timeslice_de
mo1);
tos_task_delay(1000);
}
}

int main(void)
{
board_init();
tos_knl_init();
// 配置robin机制参数
tos_robin_config(TOS_ROBIN_STATE_ENABLED, (k_timeslice_t)500u);
(void)tos_task_create(&task_demo1, "demo1", entry_task_demo1, NULL,
PRIO_TASK_DEMO, stack_task_demo1, STK_SIZE_TASK_DEMO,
timeslice_demo1);
(void)tos_task_create(&task_demo2, "demo2", entry_task_demo2, NULL,
PRIO_TASK_DEMO, stack_task_demo2, STK_SIZE_TASK_DEMO,
timeslice_demo2);
(void)tos_task_create(&task_sample, "sample", entry_task_sample, NULL,
PRIO_TASK_SAMPLE, stack_task_sample, STK_SIZE_TASK_SAMPLE,
0);
tos_knl_start();
}
```

运行效果

```
demo1_counter: 0
demo2_counter: 1
demo2_counter / demo1_counter = 0.000000
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 1915369
```

```
demo2_counter: 3720158
demo2_counter / demo1_counter = 1.942267
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 3774985
demo2_counter: 7493508
demo2_counter / demo1_counter = 1.985043
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 5634601
demo2_counter: 11266858
demo2_counter / demo1_counter = 1.999584
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 7546896
demo2_counter: 14987015
demo2_counter / demo1_counter = 1.985852
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 9406512
demo2_counter: 18759340
demo2_counter / demo1_counter = 1.994293
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 11266128
demo2_counter: 22531664
demo2_counter / demo1_counter = 1.999947
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 13177398
demo2_counter: 26251821
demo2_counter / demo1_counter = 1.992185
```

```
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 15037014
demo2_counter: 30023632
demo2_counter / demo1_counter = 1.996649
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 16896630
demo2_counter: 33795443
demo2_counter / demo1_counter = 2.000129
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000

demo1_counter: 18807900
demo2_counter: 37515600
demo2_counter / demo1_counter = 1.994672
should almost equals to:
timeslice_demo2 / timeslice_demo1 = 2.000000
```

内核基础组件

消息队列

概述

消息队列提供了一种同步的传递/收取消息的机制，与队列（tos_queue）不同的是，tos_queue 基于消息队列封装了一层异步的机制，实际上 tos_queue 的底层消息管理采用的就是消息队列。

编程实例

1. 在 tos_config.h 中，配置消息队列组件开关 TOS_CFG_MSG_EN：
   ```
   #define TOS_CFG_MSG_EN 1u
   ```
2. 在 tos_config.h 中，配置消息队列池大小 TOS_CFG_MSG_POOL_SIZE：
   ```
   #define TOS_CFG_MSG_POOL_SIZE 3u
   ```

> 说明：
> 消息队列池中可以承载的最大消息数量

3. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

#define PRIO_TASK_DEMO 4

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

k_msg_queue_t msg_queue;

struct msg_st {
char *msg;
size_t size;
} msgs[TOS_CFG_MSG_POOL_SIZE] = {
{ "msg 0", 6 },
{ "msg 1", 6 },
{ "msg 2", 6 },
};

struct msg_st dummy_msg = { "dummy msg", 10 };

extern void entry_task_demo(void *arg);

void fifo_opt(void) {
k_err_t err;
int i = 0;
char *msg_received = K_NULL;
size_t msg_size = 0;

for (; i < TOS_CFG_MSG_POOL_SIZE; ++i) {
printf("msg put: %s\n", msgs[i].msg);
err = tos_msg_queue_put(&msg_queue, (void *)msgs[i].msg, msgs[i].size, TOS_OPT_MSG_PUT_
```

```
FIFO);
if (err != K_ERR_NONE) {
printf("should never happen\n");
}
}

err = tos_msg_queue_put(&msg_queue, (void *)dummy_msg.msg, dummy_msg.size, TOS_OPT_
MSG_PUT_FIFO);
if (err == K_ERR_MSG_QUEUE_FULL) {
printf("msg queue is full\n");
} else {
printf("should never happen\n");
}

for (i = 0; i < TOS_CFG_MSG_POOL_SIZE; ++i) {
err = tos_msg_queue_get(&msg_queue, (void **)&msg_received, &msg_size);
if (err == K_ERR_NONE) {
printf("msg received: %s\n", msg_received);
printf("msg size: %d\n", msg_size);
} else {
printf("should never happen\n");
}
}
err = tos_msg_queue_get(&msg_queue, (void **)&msg_received, &msg_size);
if (err == K_ERR_MSG_QUEUE_EMPTY) {
printf("msg queue is empty\n");
} else {
printf("should never happen\n");
}
}

void lifo_opt(void) {
k_err_t err;
int i = 0;
char *msg_received = K_NULL;
size_t msg_size = 0;

for (; i < TOS_CFG_MSG_POOL_SIZE; ++i) {
```

```
printf("msg put: %s\n", msgs[i].msg);
err = tos_msg_queue_put(&msg_queue, (void *)msgs[i].msg, msgs[i].size, TOS_OPT_MSG_PUT_
LIFO);
if (err != K_ERR_NONE) {
printf("should never happen\n");
}
}
err = tos_msg_queue_put(&msg_queue, (void *)dummy_msg.msg, dummy_msg.size, TOS_OPT_
MSG_PUT_LIFO);
if (err == K_ERR_MSG_QUEUE_FULL) {
printf("msg queue is full\n");
} else {
printf("should never happen\n");
}

for (i = 0; i < TOS_CFG_MSG_POOL_SIZE; ++i) {
err = tos_msg_queue_get(&msg_queue, (void **)&msg_received, &msg_size);
if (err == K_ERR_NONE) {
printf("msg received: %s\n", msg_received);
printf("msg size: %d\n", msg_size);
} else {
printf("should never happen\n");
}
}
err = tos_msg_queue_get(&msg_queue, (void **)&msg_received, &msg_size);
if (err == K_ERR_MSG_QUEUE_EMPTY) {
printf("msg queue is empty\n");
} else {
printf("should never happen\n");
}
}

void entry_task_demo(void *arg)
{
tos_msg_queue_create(&msg_queue);

printf("max msg in pool: %d\n", TOS_CFG_MSG_POOL_SIZE);
printf("msg queue using TOS_OPT_MSG_PUT_FIFO\n");
```

```
fifo_opt();

printf("msg queue using TOS_OPT_MSG_PUT_LIFO\n");
lifo_opt();
}

int main(void)
{
board_init();
tos_knl_init();
(void)tos_task_create(&task_demo, "demo1", entry_task_demo, NULL,
PRIO_TASK_DEMO, stack_task_demo, STK_SIZE_TASK_DEMO,
0);
tos_knl_start();
}
```

运行效果

```
max msg in pool: 3
msg queue using TOS_OPT_MSG_PUT_FIFO
msg put: msg 0
msg put: msg 1
msg put: msg 2
msg queue is full
msg received: msg 0
msg size: 6
msg received: msg 1
msg size: 6
msg received: msg 2
msg size: 6
msg queue is empty
msg queue using TOS_OPT_MSG_PUT_LIFO
msg put: msg 0
msg put: msg 1
msg put: msg 2
msg queue is full
msg received: msg 2
```

```
msg size: 6
msg received: msg 1
msg size: 6
msg received: msg 0
msg size: 6
msg queue is empty
```

字符流先入先出队列

概述

字符流先入先出队列，提供的是一个面向字符操作的环形队列实现，提供了基本的字符流入队出队操作。

编程实例

编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

#define PRIO_TASK_DEMO 4

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

#define FIFO_BUFFER_SIZE 5
uint8_t fifo_buffer[FIFO_BUFFER_SIZE];

k_fifo_t fifo;

extern void entry_task_demo(void *arg);

void char_push(void)
{
k_err_t err;
int i = 0;
uint8_t data;
```

```
// 往fifo中压入FIFO_BUFFER_SIZE个字符，分别是a、b、c、d、e
for (i = 0; i < FIFO_BUFFER_SIZE; ++i) {
printf("char pushed: %c\n", 'a' + i);
err = tos_fifo_push(&fifo, 'a' + i);
if (err != K_ERR_NONE) {
printf("should never happen\n");
}
}

// fifo最多包含FIFO_BUFFER_SIZE个字符，上文逻辑中已经压入了最大的字符量，此时继续压入字符会返回K_ERR_FIFO_FULL（fifo已满）
err = tos_fifo_push(&fifo, 'z');
if (err == K_ERR_FIFO_FULL) {
printf("fifo is full: %s\n", tos_fifo_is_full(&fifo) ? "TRUE" : "FALSE");
} else {
printf("should never happen\n");
}

// 从fifo中把所有的字符都弹出来
for (i = 0; i < FIFO_BUFFER_SIZE; ++i) {
err = tos_fifo_pop(&fifo, &data);
if (err == K_ERR_NONE) {
printf("%d pop: %c\n", i, data);
} else {
printf("should never happen\n");
}
}
// 此时继续弹出字符，会返回K_ERR_FIFO_EMPTY（fifo已空）
err = tos_fifo_pop(&fifo, &data);
if (err == K_ERR_FIFO_EMPTY) {
printf("fifo is empty: %s\n", tos_fifo_is_empty(&fifo) ? "TRUE" : "FALSE");
} else {
printf("should never happen\n");
}
}

void stream_push(void)
```

```
{
int count = 0, i = 0;
uint8_t stream[FIFO_BUFFER_SIZE] = { 'a', 'b', 'c', 'd', 'e' };
uint8_t stream_dummy[1] = { 'z' };
uint8_t stream_pop[FIFO_BUFFER_SIZE];

// 压入字符流，字符流的长度是5，不超过fifo的最大长度FIFO_BUFFER_SIZE，会压入成功，并返回压入字符流的长度5
count = tos_fifo_push_stream(&fifo, &stream[0], FIFO_BUFFER_SIZE);
if (count != FIFO_BUFFER_SIZE) {
printf("should never happen\n");
}

// 继续压入字符流（即使是长度为1的字符流），因fifo已满无法继续压入，返回长度0（压入失败）
count = tos_fifo_push_stream(&fifo, &stream_dummy[0], 1);
if (count == 0) {
printf("fifo is full: %s\n", tos_fifo_is_full(&fifo) ? "TRUE" : "FALSE");
} else {
printf("should never happen\n");
}

// 将前文中压入的字符流全部弹出，返回前文压入的字符流长度5（弹出的字符流长度）
count = tos_fifo_pop_stream(&fifo, &stream_pop[0], FIFO_BUFFER_SIZE);
if (count == FIFO_BUFFER_SIZE) {
printf("stream popped:\n");
for (i = 0; i < FIFO_BUFFER_SIZE; ++i) {
printf("%c", stream_pop[i]);
}
printf("\n");
} else {
printf("should never happen\n");
}

// 继续弹出，因fifo已空，返回0
count = tos_fifo_pop_stream(&fifo, &stream_pop[0], 1);
if (count == 0) {
printf("fifo is empty: %s\n", tos_fifo_is_empty(&fifo) ? "TRUE" : "FALSE");
} else {
```

```
printf("should never happen\n");
}
}

void entry_task_demo(void *arg)
{
// 创建了一个最多包含FIFO_BUFFER_SIZE个字符的fifo
tos_fifo_create(&fifo, &fifo_buffer[0], FIFO_BUFFER_SIZE);

printf("fifo, dealing with char\n");
char_push();

printf("fifo, dealing with stream\n");
stream_push();
}

int main(void)
{
board_init();
tos_knl_init();
(void)tos_task_create(&task_demo, "demo1", entry_task_demo, NULL,
PRIO_TASK_DEMO, stack_task_demo, STK_SIZE_TASK_DEMO,
0);
tos_knl_start();
}
```

运行效果

```
fifo, dealing with char
char pushed: a
char pushed: b
char pushed: c
char pushed: d
char pushed: e
fifo is full: TRUE
0 pop: a
1 pop: b
```

```
2 pop: c
3 pop: d
4 pop: e
fifo is empty: TRUE
fifo, dealing with stream
fifo is full: TRUE
stream popped:
abcde
fifo is empty: TRUE
```

功耗管理

低功耗

概述

TencentOS tiny 提供了多级低功耗管理框架。初级低功耗的方案是，当系统处于“空闲”状态，也即进入 idle 任务时，系统调用处理器（目前支持的架构是 arm v7m）低功耗接口进入短暂的睡眠模式。

编程实例

对于初级低功耗模式，无需用户编写任何代码，直接通过在 tos_config.h 打开 TOS_CFG_PMR_MGR_EN 开关即可： #define TOS_CFG_PWR_MGR_EN 1u

tickless

概述

TencentOS tiny 的 tickless 机制提供了一套非周期性时钟的方案，在系统无需 systick 驱动调度的情况下，停掉 systick。

初级功耗管理方案下，因为还有系统 systick 的存在，因此系统进入 idle 任务后，并不会在睡眠模式下停留太久。如需进入到更极致的低功耗状态，需要暂停 systick。

arm 架构提供三级低功耗模式，sleep、stop、standby 模式，三种模式运行功耗逐次降低，standby 模式最低。TencentOS tiny 的内核提供了简洁清晰的接口来管理各级模式。

API 接口说明

```
void tos_tickless_wkup_alarm_install(k_cpu_lpwr_mode_t mode, k_tickless_wkup_alarm_t *wkup_alarm);
```

此接口用以安装各低功耗模式下的唤醒闹钟。当内核进入 tickless 模式下后，systick 以及停止了，因此需要其他计时器来将 CPU 从低功耗模式下唤醒。

根据 arm v7m 的芯片规格，三种模式下的唤醒源分别为：

- sleep：CPU 进入 sleep 模式后，可以由 systick、硬件 timer、RTC 时钟唤醒（wakeup/alarm 中断）。
- stop：CPU 进入 stop 模式后，可以由 RTC 时钟（wakeup/alarm 中断）唤醒。
- standby： CPU 进入 standby 模式后，只可由 RTC 时钟的 alarm 中断唤醒（还可以通过外部管脚唤醒，但这不属于TencentOS tiny 内核机制设计的范畴）。

k_tickless_wkup_alarm_t 定义如下：

```
typedef struct k_tickless_wakeup_alarm_st {
int (*init)(void);
int (*setup)(k_time_t millisecond);
int (*dismiss)(void);
k_time_t (*max_delay)(void); /* in millisecond */
} k_tickless_wkup_alarm_t;
```

一个唤醒闹钟有四个成员方法：

接口名称	接口说明
init	闹钟初始化函数
setup	闹钟设定函数，入参为闹钟到期时间（单位毫秒）。此闹钟在设定完毕后的 millisecond 毫秒时来中断
dismiss	闹钟解除函数，执行完后闹钟中断不会再来
max_delay	此闹钟最长的到期时间（单位为毫秒）

```
k_err_t tos_tickless_wkup_alarm_init(k_cpu_lpwr_mode_t mode);
```

此函数用来初始化特定模式下的唤醒闹钟（实际上调用的是 tos_tickless_wkup_alarm_install 接口中安装的 k_tickless_wkup_alarm_t 的 init 方法）。

```
k_err_t tos_pm_cpu_lpwr_mode_set(k_cpu_lpwr_mode_t cpu_lpwr_mode);
```

设置内核在 tickless 模式下进入的 CPU 低功耗模式。

编程实例

1. 在 tos_config.h 中，配置低功耗组件开关 TOS_CFG_PWR_MGR_EN：
   #define TOS_CFG_PWR_MGR_EN 1u
2. 在 tos_config.h 中，配置 tickless 组件开关 TOS_CFG_TICKLESS_EN：
   #define TOS_CFG_TICKLESS_EN 1u
3. 编写 main.c 示例代码：

```
#include "tos.h"
#include "mcu_init.h"

#define STK_SIZE_TASK_DEMO 512

#define PRIO_TASK_DEMO 4

k_stack_t stack_task_demo[STK_SIZE_TASK_DEMO];

k_task_t task_demo;

extern void entry_task_demo(void *arg);

void timer_callback(void *arg)
{
printf("timer callback: %lld\n", tos_systick_get());
}

void entry_task_demo(void *arg)
{
k_timer_t tmr;

// 创建一个软件定时器，每6000个tick触发一次
tos_timer_create(&tmr, 0u, 6000u, timer_callback, K_NULL, TOS_OPT_TIMER_PERIODIC);
tos_timer_start(&tmr);

// 此任务体内每3000个tick运行一次
while (K_TRUE) {
printf("entry task demo: %lld\n", tos_systick_get());
tos_task_delay(3000);
```

```
}
}

int main(void)
{
board_init();
tos_knl_init();
(void)tos_task_create(&task_demo, "demo1", entry_task_demo, NULL,
PRIO_TASK_DEMO, stack_task_demo, STK_SIZE_TASK_DEMO,
0);
tos_knl_start();
}
```

4. 实现 tos_bsp_tickless_setup 回调（参考 board\TOS_tiny_EVK_STM32L431CBT6\BSP\Src\tickless\bsp_pwr_mgr.c、board\TOS_tiny_EVK_STM32L431CBT6\BSP\Src\tickless\bsp_tickless_alarm.c）：

```
#include "tos.h"
#include "tickless/bsp_pm_device.h"
#include "tickless/bsp_tickless_alarm.h"

int tos_bsp_tickless_setup(void)
{
#if TOS_CFG_TICKLESS_EN > 0u
// sleep模式下的唤醒源，基本定时器
tos_tickless_wkup_alarm_install(TOS_LOW_POWER_MODE_SLEEP, &tickless_wkup_alarm_tim);
// 初始化唤醒源闹钟
tos_tickless_wkup_alarm_init(TOS_LOW_POWER_MODE_SLEEP);
// 设置tickless状态时进入sleep模式
tos_pm_cpu_lpwr_mode_set(TOS_LOW_POWER_MODE_SLEEP);
#endif
}
```

5. 为了观察在 tickless 时是否无 systick 中断，请在 tos_sys.c 的 idle 任务体内加一句调试代码：

```
__STATIC__ void knl_idle_entry(void *arg)
{
arg = arg; // make compiler happy

while (K_TRUE) {
// 这里在idle任务体内加上一句打印，如果systick正常开启，在没有用户任务运行时，此调试信息会不断打印；如果是tickless状态，此调试信息应该只会第一次进入idle任务时，或在用户任务等待到期，或用户的软件定时器到期时，才打印一次。
printf("idle entry: %lld\n", tos_systick_get());
#if TOS_CFG_PWR_MGR_EN > 0u
pm_power_manager();
#endif
}
}
```

运行效果

```
entry task demo: 0
idle entry: 2
entry task demo: 3002
idle entry: 3002
timer callback: 6000
idle entry: 6000
entry task demo: 6002
idle entry: 6002
entry task demo: 9002
idle entry: 9002
timer callback: 12000
idle entry: 12000
entry task demo: 12002
idle entry: 12002
entry task demo: 15002
idle entry: 15002
timer callback: 18000
idle entry: 18000
entry task demo: 18002
idle entry: 18002
```