OneMO | 中国移动 China Mobile

MN316
OpenCPU 操作系统开发指导手册

NB-IoT 系列

版本：V1.0.0

日期：2020 年 12 月

中移物联网有限公司
iot.10086.cn

服务与支持

如果您有任何关于模组产品及产品手册的评论、疑问、想法，或者任何无法从本手册中找到答案的疑问，请通过以下方式联系我们。

中移物联网有限公司

OneMO 官网：onemo10086.com

邮箱：SmartModule@cmiot.chinamobile.com

客户服务热线：400-110-0866

微信公众号：CMOneMO

文档声明

注意

本手册描述的产品及其附件特性和功能，取决于当地网络设计或网络性能，同时也取决于用户预先安装的各种软件。由于当地网络运营商、ISP，或当地网络设置等原因，可能也会造成本手册中描述的全部或部分产品及其附件特性和功能未包含在您的购买或使用范围之内。

责任限制

除非合同另有约定，中移物联网有限公司对本文档内容不做任何明示或暗示的声明或保证，并且不对特定目的适销性及适用性或者任何间接的、特殊的或连带的损失承担任何责任。

在适用法律允许的范围内，在任何情况下，中移物联网有限公司均不对用户因使用本手册内容和本手册中描述的产品而引起的任何特殊的、间接的、附带的或后果性的损坏、利润损失、数据丢失、声誉和预期的节省而负责。

因使用本手册中所述的产品而引起的中移物联网有限公司对用户的最大赔偿（除在涉及人身伤害的情况中根据适用法律规定的损害赔偿外），不应超过用户为购买此产品而支付的金额。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。公司保留随时修改本手册中任何信息的权利，无需进行提前通知且不承担任何责任。

商标声明

中国移动 China Mobile 为中国移动注册商标。

本手册和本手册描述的产品中出现的其他商标、产品名称、服务名称和公司名称，均为其各自所有者的财产。

进出口法规

出口、转口或进口本手册中描述的产品（包括但不限于产品软件和技术数据），用户应遵守相关进出口法律和法规。

隐私保护

关于我们如何保护用户的个人信息等隐私情况，请查看相关隐私政策。

操作系统更新声明

操作系统仅支持官方升级；如用户自己刷非官方系统，导致安全风险和损失由用户负责。

固件包完整性风险声明

固件仅支持官方升级；如用户自己刷非官方固件，导致安全风险和损失由用户负责。

版权所有©中移物联网有限公司。保留一切权利。

关于文档

修订记录

版本	发布日期	作者	描述
V1.0.0	2020/12/23	张雄威/任亚洲	初版

目录

1 操作系统

OpenCPU 已对原生操作系统进行封装，开发时请使用封装后的接口。

1.1 线程

系统线程资源少于 20 个，使用的详细示例可参考 SDK 代码。

1.1.1 创建线程

创建线程	
函数原型	
osThreadId_t osThreadNew (osThreadFunc_t func, void *argument, const osThreadAttr_t *attr)	
备注	
<osThreadAttr_t->priority>用户线程优先级范围 osPriorityBelowNormal- osPriorityBelowNormal5	
线程栈的长度等于<osThreadAttr_t->stack_size> * 4	
示例	

osThreadAttr_t attr = {
.name = "OpenCPU-DEMO",
.priority = osPriorityNormal5,
.stack_size = 1024,// size = 1024*4
};
 	
g_opencpu_tsk_handle =
osThreadNew(cm_task_loop, NULL, (const osThreadAttr_t *)&attr);
	//创建线程。

1.1.2 获取线程状态

获取线程状态	
函数原型	
osThreadState_t osThreadGetState (osThreadId_t thread_id)	
备注	
返回状态参见 osThreadState_t 枚举说明	
示例	
int stat = osThreadGetState(g_tsk_handle);	//获取线程状态。

1.1.3 退出当前线程

退出当前线程
函数原型
void osThreadExit (void)
备注
此接口用于退出当前线程，置于线程末尾。
示例

```
void demo_task(void* arg)
{
        do_something();                                  //线程运行。
        osThreadExit();                                  //退出当前线程。
}
int cm_main()
{
       osThreadAttr_t attr = {                           //创建线程。
             .name = "OpenCPU-DEMO",
             .priority = osPriorityNormal5,
             .stack_size = 1024,//size = 1024*4
       };

        g_opencpu_tsk_handle =
          osThreadNew(demo_task, NULL, (const
          osThreadAttr_t *)&attr);
}
```

1.2 队列

系统队列的发送与接收使用指针传递方式来减少内存拷贝操作，即队列发送端将消息地址发出，接收端读取该地址，并对指针指向的内容进行处理。使用时需注意内存的申请与释放。队列操作成功返回 0，失败则返回其他值。

系统队列资源少于 20 个。

1.2.1 创建队列

创建队列		
函数原型		
osMessageQueueId_t osMessageQueueNew (uint32_t msg_count, uint32_t msg_size, const osMessageQueueAttr_t *attr)		
参数描述		
<msg_count>		
队列长度；即队列最多可存储的消息个数。		
<msg_size>		
消息长度；队列配置为指针传递，消息长度应等于地址长度，使用时传入 sizeof(void*)即可。		
<attr>		
传入 NULL		
返回值		
队列 ID		
示例		
g_test_queue = sizeof(void*), NULL);	osMessageQueueNew(10,	//创建队列长度为 10，消息长度为 4 的队列。

1.2.2 发送数据

发送数据		
函数原型		
osStatus_t osMessageQueuePut (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio, uint32_t timeout)		
参数描述		
<msg_ptr>		
待传消息地址		
<msg_prio>		
使用时传入 0		
<timeout>		
阻塞时长，一般设置为 osNoWait 或 osWaitForever。		
示例		

Msg* msg = cm_malloc(sizeof(Msg));
msg.type = 1;
osMessageQueuePut(g_test_queue, &msg, 0, osNoWait);		

1.2.3 接收数据

接收数据
函数原型
osStatus_t osMessageQueueGet (osMessageQueueId_t mq_id, void *msg_ptr, uint8_t *msg_prio, uint32_t timeout)
备注
与队列发送类似，接收时仅接受消息的地址；使用完成后需注意消息的内存释放。
示例

Msg* msg = NULL;
if(osMessageQueueGet(g_test_queue, &msg, NULL, //永久阻塞，直至队列不为空。
osWaitForever) == osOK)
{
do_something();
}
cm_free(msg);

1.2.4 删除队列

删除队列
函数原型
osStatus_t osMessageQueueDelete (osMessageQueueId_t mq_id)
示例
osMessageQueueDelete(g_test_queue); g_test_queue = NULL;

1.3 互斥锁

互斥锁用于多线程间数据同步，保证数据操作的原子性；加锁与释放锁成对出现。请勿在任何中断处理函数中使用互斥锁。

系统互斥锁资源少于 30 个。

1.3.1 创建互斥锁

创建互斥锁
函数原型
osMutexId_t osMutexNew (const osMutexAttr_t *attr)
备注
使用时<attr>传入 NULL
返回互斥锁 ID
示例
osMutexId_t g_test_mutex = NULL; g_test_mutex = osMutexNew(NULL);

1.3.2 持有锁

持有锁
函数原型
osStatus_t osMutexAcquire (osMutexId_t mutex_id, uint32_t timeout)
备注
使用时采用阻塞模式，即<timeout>设置为 osWaitForever。
示例

if(g_test_mutex != NULL)
{
　　osMutexAcquire(g_test_mutex, osWaitForever);　　//持有锁。
}
do_something();　　//同步数据。
osMutexRelease(g_test_mutex);

1.3.3 释放锁

释放锁	
函数原型	
osStatus_t osMutexRelease (osMutexId_t mutex_id)	
备注	
与加锁操作必须成对出现。	
示例	

if(g_test_mutex != NULL)
{
　　osMutexAcquire(g_test_mutex, osWaitForever);
}
do_something();
osMutexRelease(g_test_mutex);
	//持有锁。
	
	//同步数据。
	//释放锁。

1.3.4 删除互斥锁

删除互斥锁	
函数原型	
osStatus_t osMutexDelete (osMutexId_t mutex_id)	
备注	
删除时必须保证锁已被正常释放，即没有任何线程持有该锁。	
示例	

if(mutex != NULL)
{
　　osMutexDelete(mutex);
　　mutex = NULL;
}
	
	//删除锁。

1.4 信号量

Semaphore 包含二值信号量和整型信号量，与 Mutex 类似，用于数据同步等场景。

系统信号量资源少于 60 个。

1.4.1 创建信号量

创建信号量
函数原型
osSemaphoreId_t osSemaphoreNew (uint32_t max_count, uint32_t initial_count, const osSemaphoreAttr_t *attr)
备注
<max_count>为 1 时，表示创建二值信号量，即仅存在 0/1 两种状态。
<initial_count>表示信号量的初始状态
<attr>传入 NULL
示例
osSemaphoreId_t g_test_semap = NULL; g_test_seamp = osSemaphoreNew(1, 0, NULL);　　//创建二值信号量，初始值为 0。

1.4.2 等待信号

信号量获取
函数原型
osStatus_t osSemaphoreAcquire (osSemaphoreId_t semaphore_id, uint32_t timeout)
备注
使用时采用阻塞模式，即<timeout>设置为 osWaitForever。
示例

if(g_test_semap != NULL)
{
　　osSemaphoreAcquire(g_test_seamp, osWaitForever);　　//永久阻塞，直至其他线程有发送信号操作。
}
do_something();　　//同步数据。

1.4.3 发出信号

发出信号	
函数原型	
osStatus_t osSemaphoreRelease (osSemaphoreId_t semaphore_id)	
备注	
发出信号后，等待信号的线程才能退出阻塞状态执行后续操作，保证操作的原子性。	
示例	
do_something();	//同步数据。
if(g_test_semap != NULL)	
{	
osSemaphoreRelease(g_test_seamp);	//发出信号。
}	

1.4.4 删除信号量

删除信号量	
函数原型	
osStatus_t osSemaphoreDelete (osSemaphoreId_t semaphore_id)	
备注	
删除时不能有任何线程处于发送或等待信号的操作中。	
示例	
if(g_test_seamp != NULL)	
{	
osSemaphoreDelete(g_test_seamp);	//删除信号量。
g_test_seamp = NULL;	
}	

1.5 内存申请与释放

用户在进行 OpenCPU 开发时，禁止使用 C 库中的 malloc 与 free 函数及其他涉及到内存申请释放的函数，须使用 SDK 提供的 cm_malloc 与 cm_free 函数。

1.5.1 申请内存

申请内存
函数原型
void* cm_malloc (size_t size)

1.5.2 释放内存

释放内存
函数原型
void cm_free (void* buf)

1.6 软件定时器

用户在进行 OpenCPU 开发时，可使用软件定时器触发周期性运行的业务。系统软件定时器资源少于 16 个。

1.6.1 创建定时器

创建定时器
函数原型
osTimerId_t osTimerNew (osTimerFunc_t func, osTimerType_t type, void *argument, const osTimerAttr_t *attr)
备注
<type>设置为 osTimerPeriodic，定时器会周期性运行；设置为 osTimerOnce，仅运行一次。请勿在定时器中断函数中执行耗时操作。
<argument>及<attr>传入 NULL
示例

if(g_test_timer == NULL)
{
g_test_timer = osTimerNew(test_timer_callback, osTimerPeriodic, //设置周期性定时器。
NULL, NULL);
}

1.6.2 启动定时器

启动定时器
函数原型
osStatus_t osTimerStart (osTimerId_t timer_id, uint32_t ticks)
备注
<ticks>单位为毫秒
示例
osTimerStart(g_test_timer, 5000); //启动定时器，定时时间 5 秒

1.6.3 停止定时器

停止定时器
函数原型
osStatus_t osTimerStop (osTimerId_t timer_id)
示例
osTimerStop(g_test_timer);

1.6.4 删除定时器

删除定时器
函数原型
osStatus_t osTimerDelete (osTimerId_t timer_id)
示例
osTimerDelete(g_test_timer);

2 附录

2.1 参考文档

序号	文档名称	备注
[1]	MN316_OpenCPU MANUAL	API 接口手册
[2]	MN316_OpenCPU 资源综述	资源介绍