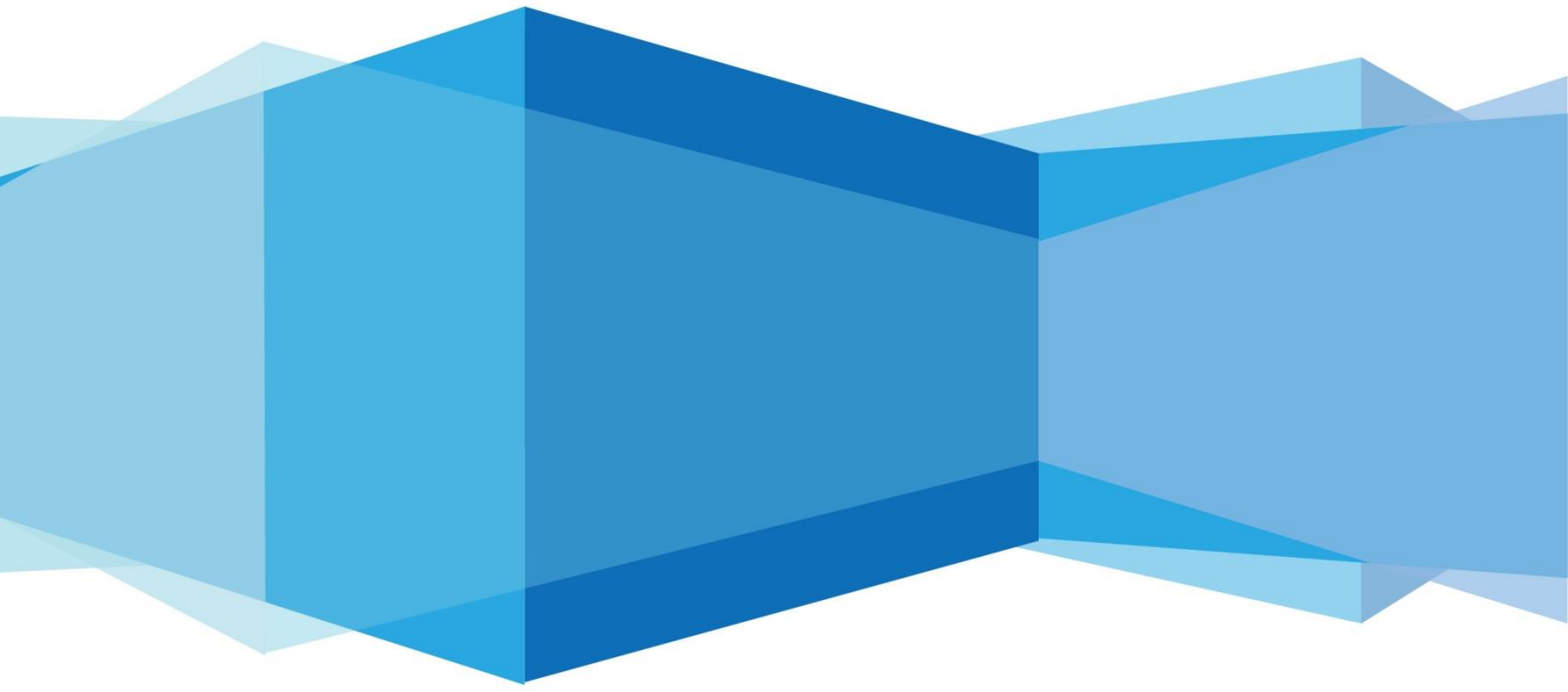
lierda 利尔达科技集团 | 证券代码 832149

Lierda NB-IoT 模组

OpenCPU API 使用文档

版本：Rev2.5

日期：2019-04-12



利尔达科技集团股份有限公司
LIERDA SCIENCE & TECHNOLOGY GROUP CO.,LTD

法律声明

若接收浙江利尔达物联网技术有限公司（以下称为“利尔达”）的此份文档，即表示您已经同意以下条款。若不同意以下条款，请停止使用本文档。

本文档版权所有浙江利尔达物联网技术有限公司，保留任何未在本文档中明示授予的权利。文档中涉及利尔达的专有信息。未经利尔达事先书面许可，任何单位和个人不得复制、传递、分发、使用和泄漏该文档以及该文档包含的任何图片、表格、数据及其他信息。

本产品符合有关环境保护和人身安全方面的设计要求，产品的存放、使用和弃置应遵照产品手册、相关合同或者相关法律、法规的要求进行。

本公司保留在不预先通知的情况下，对此手册中描述的产品进行修改和改进的权利；同时保留随时修订或收回本手册的权利。

文件修订历史

版本	修订日期	修订日志
1.0	2018-08-30	第一次发布版本
1.1	2018-09-03	增加 UART 接口函数
1.2	2018-09-06	整合 LiteOS API
1.3	2018-09-06	添加 Logview 接口函数
1.4	2018-09-07	修改文档格式
1.5	2018-09-07	修改文档格式
1.6	2018-09-10	修改文档格式
1.7	2018-09-11	修改 I2C 写寄存器函数，入参“uint8 data”改为“uint8 *data”，可写入多个字节。
1.8	2018-09-12	修改 I2C 读函数，入参增加“uint8 readOnly”，针对某些传感器读的时候不需要发送寄存器地址。
1.9	2018-09-12	修改 I2C 接口函数；修改 UART 接口函数
2.0	2018-09-26	增加 NOTE
2.1	2018-09-30	1. 新增“UART 初始化函数”，支持自定义一路串口 2. 新增 GPIO 中断回调函数 3. 新增下行数据接收函数 4. 新增 LiteOS 消息队列接口
2.2	2018-11-16	1. 新增 ADC 接口函数 2. 新增 DAC 接口函数 3. 新增 KV 相关函数 4. 新增华大北斗 GPS 接口函数 5. 新增 HDC1000 温湿度传感器接口函数 6. 新增 OPT3001 光照传感器接口函数

		7. 新增 LIS3DH 三轴传感器接口函数
2.3	2018-11-29	1. 更改 KV 函数参数和返回值 2. 调整 DACAPI 函数顺序 3. 更改传感器库 lierda_HDC1000_UpdateInfo 参数为有符号整型 4. 更改传感器库 lierda_LIS3DH_UpdateInfo 参数为有符号整型
2.4	2018-12-25	1. 新增 DNS 解析函数 2. 新增 UDP,TCP 下行数据接收信号量函数 3. 新增软件模拟 SPI 接口函数
2.5	2019-04-12	1.新增事件状态查询接口 2.新增硬件 SPI 接口函数 3.新增 LWM2M 数据发送接口 4.更改获取当前 Vbat 电压原始数值接口函数 5.新增 FOTA 状态查询接口 6.新增 FOTA 环境下能否做业务接口 7.新增 I2C 通信接口，使用更灵活

适用模块型号

序号	模块型号	模块简介
1	NB86-G	全频段，支持频段 1/2/3/5/8 等，20×16×2.2（mm）

目录

1 介绍

本文旨在帮助基于使用 Lierda NB86-G 模组进行 OpenCPU 开发的用户，让其能快速使用模组本身的各种硬件资源（I2C、GPIO、UART）和 LiteOS 操作系统（创建、删除、挂起、恢复线程。创建、删除、启动、停止软件定时器）。

考虑到用户的集成难度，Lierda Open CPU SDK 目前提供一种基础的指令调用方式，如：

```
uint8_t * lierdaATCall(uint8_t * p_paramater_string)
```

在函数参数中输入 AT 指令字符串，函数会直接执行 AT 指令，并将执行的返回结果以字符串形式返回。

目前能够提供的 API 接口主要是：

- AT 指令调用
- 软件定时
- Log 打印
- 操作系统接口
- 外设驱动（GPIO，UART，I2C，SPI）

图 1-1 OpenCPU SDK 功能分布

后续会继续完善升级 SDK，提供封装后的 AT 指令调用接口，不再需要用户将数据及指令封装成字符串形式的 AT 指令，外设驱动也会继续丰富，提供海思芯片能支持的芯片外设驱动。

2 硬件驱动接口

2.1 I2C 接口

2.1.1 I2C 初始化

功能分类	内容	描述
I2C 初始化	lierdaI2CInit	
函数原型	HAL_StatusTypeDef lierdaI2CInit(I2C_HandleTypeDef *hi2c)	
参数说明	I2C_HandleTypeDef *hi2c	初始化配置句柄
返回值	成功返回 HAL_OK 失败返回 HAL_ERROR	

示例	#include "lierdaI2C.h"
	#define I2C_SCL PIN_12
	#define I2C_SDA PIN_15
	I2C_HandleTypeDef sensorI2CHandle;
	
	sensorI2CHandle.i2c_bus = I2C_BUS1;//I2C 总线 ID
	sensorI2CHandle.pin_scl = I2C_SCL;//时钟引脚
	sensorI2CHandle.pin_sda = I2C_SDA;//数据引脚
	sensorI2CHandle.i2c_address_type = HAL_I2C_ADDRESS_TYPE_7_BIT;//地址类型
	sensorI2CHandle.i2c_half_time = 256;//速率，256 为 100k,64 为 400k
	sensorI2CHandle.i2c_mode= HAL_I2C_BUS_MODE_MASTER;//主从模式	

	uint8 ret = lierdaI2CInit (&sensorI2CHandle);	

2.1.2 I2C 读寄存器

功能分类	内容	描述
I2C 读操作	lierdaI2CReadreg	
函数原型	Uint8 lierdaI2CReadreg(I2C_HandleTypeDef *hi2c uint8 i2c_slave_addr, uint16 i2c_rx_reg_addr, uint8 *data , uint8 len uint8 readOnly)	
参数说明	*hi2c:初始化配置句柄 i2c_slave_addr : 从机硬件地址 i2c_rx_reg_addr: 需要读取的从机寄存器地址 data: 读出的数据缓存地址 len: 需要读取的长度 readOnly:是否只读取数值，不发送传感器地址，1 表示是，0 表示否	readOnly 参数：对于某些传感器如 HDC1000，读取传感器数值时可以不指定寄存器地址，此时 readOnly 为 1，此时 i2c_rx_reg_addr 建议设置为 0；如要指定传感器上的寄存器地址则为 0
返回值	0 :成功 1 :失败	
示例	#include "lierdaI2C.h" uint8 RecData; lierdaI2Creadreg (&sensorI2Chandle，0x4c, 0x01, &RecData, 1，0)	

2.1.3 I2C 写寄存器

功能分类	内容	描述
I2C 写操作	lierdaI2CWritereg	
函数原型	char lierdaI2CWritereg(I2C_HandleTypeDef *hi2c uint8 i2c_slave_addr, uint16 i2c_tx_reg_addr, uint8 *data , uint8 len)	
参数说明	*hi2c:初始化配置句柄 i2c_slave_addr : 从机硬件地址 i2c_tx_reg_addr: 需要写的从机寄存器地址 data: 写入的值 len: 写入的长度	
返回值	0 :成功 1 :失败	
示例	#include "lierdaI2C.h" uint8 WirteData[1]={ 0x01}; lierdaI2CWritereg (&sensorI2Chandle，0x4c, 0x11, WirteData, 1);	

2.1.4 I2C 取消初始化

功能分类	内容	描述
I2C 取消初始化	lierdaI2CDeinit	用于解除用户已分配的 I2C 功能
函数原型	Uint8 lierdaI2Cdeinit (I2C_HandleTypeDef *hi2c)	
参数说明	*hi2c:初始化配置句柄	

返回值	0 :成功 1 :失败	
示例	#include "lierdaI2C.h" lierdaI2CDeinit(&sensorI2CHandle);	

2.1.5 I2C 读接口

功能分类	内容	描述
I2C 读操作	lierdaI2CReadreg	
函数原型	I2C_RET lierdaI2CRead(I2C_HandleTypeDef *hi2c, uint8 i2c_slave_addr, uint8 *data, uint8 data_len);	
参数说明	*hi2c:初始化配置句柄 i2c_slave_addr : 从机硬件地址 data: 读出的数据缓存地址 data_len: 需要读取的长度	
返回值	0 :成功 1 :失败	
示例	#include "lierdaI2C.h" lierdaI2CRead(&sensorI2CHandle, gssI2C_ADDRESS, &response,1);	

2.1.6 I2C 写接口

功能分类	内容	描述
I2C 写操作	lierdaI2CWrite	
函数原型	I2C_RET lierdaI2CWrite(I2C_HandleTypeDef *hi2c, uint8 i2c_slave_addr, uint8 *data, uint8 data_len)	
参数说明	*hi2c:初始化配置句柄 i2c_slave_addr : 从机硬件地址 data: 写入的值 data_len: 写入的长度	该接口的传输方式为：写入数值到某个寄存器时，需要在数据前加上寄存器地址，譬如，往 0x0F 寄存器写入数值 0x11，则传入的“data”应为“0F11”，“data_len”长度包括寄存器地址
返回值	0 :成功 1 :失败	

示例	#include "lierdaI2C.h"
	Uint8 sendDataBuffer[3];
	sendDataBuffer[0]=0xf;
	sendDataBuffer[1]=0x11;
	lierdaI2CWrite(&sensorI2CHandle,
	gssI2C_ADDRESS,
	sendDataBuffer,2);	

2.2 GPIO 接口函数

2.2.1 GPIO 初始化

功能分类	内容	描述
GPIO 初始化	lierdaGPIOInit	初始化所有的 GPIO，默认状态为浮空
函数原型	void lierdaGPIOInit(void)	
参数说明	无	
返回值	无	
示例	#include "lierdaGPIO.h" lierdaGPIOInit();	

2.2.2 GPIO 取消初始化

功能分类	内容	描述
GPIO 取消初始化	lierdaGPIODeinit	
函数原型	void lierdaGPIODeinit (void)	
参数说明	无	
返回值	无	
示例	#include "lierdaGPIO.h" lierdaGPIODeinit ();	

2.2.3 GPIO 声明函数

功能分类	内容	描述
GPIO 声明	lierdaGPIOClaim	为具体的 IO 口分配方向，目前不支持芯片内对 IO 上拉

函数原型	Bool lierdaGPIOClaim(PINpin,GPIO_DIRECTION dir)	
参数说明	pin: IO 引脚号。（枚举类型值，具体见源代码） dir: GPIO_DIRECTION_INPUT:输入 GPIO_DIRECTION_OUTPUT:输出（枚举类型值，具体见源代码）	
返回值	true： 成功 false: 失败	
示例	#include "lierdaGPIO.h" bool Vlaue; Vlaue=lierdaGPIOClaim(PIN_25, GPIO_DIRECTION_OUTPUT);	

2.2.4 GPIO 电平拉高函数

功能分类	内容	描述
GPIO 拉高电平	lierdaGPIOSet	
函数原型	void lierdaGPIOSet(PIN pin)	
参数说明	pin: IO 引脚号。（枚举类型值，具体见源代码）	
返回值	无	
示例	#include "lierdaGPIO.h" lierdaGPIOSet(PIN_25);	控制前必须完成如下流程： lierdaGPIOInit lierdaGPIOClaim

2.2.5 GPIO 电平拉低函数

功能分类	内容	描述
GPIO 拉低电平	lierdaGPIOClear	
函数原型	void lierdaGPIOClear (PIN pin)	
参数说明	pin: IO 引脚号。（枚举类型值，具体见源代码）	
返回值	无	
示例	#include "lierdaGPIO.h" lierdaGPIOClear (PIN_25);	控制前必须完成如下流程： lierdaGPIOInit lierdaGPIOClaim

2.2.6 GPIO 释放函数

功能分类	内容	描述
GPIO 释放	lierdaGPIORelease	对应 lierdaGPIOClaim，释放已经分配的 GPIO 功能
函数原型	void lierdaGPIORelease (PIN pin)	
参数说明	pin: IO 引脚号。（枚举类型值，具体见源代码）	
返回值	无	
示例	#include "lierdaGPIO.h" lierdaGPIORelease (PIN_25);	控制前必须完成如下流程： lierdaGPIOInit lierdaGPIOClaim

2.2.7 GPIO 翻转函数

功能分类	内容	描述
GPIO 翻转	lierdaGPIOToggle	
函数原型	void lierdaGPIOToggle (PIN pin)	

参数说明	pin: IO 引脚号。（枚举类型值，具体见源代码）	
返回值	无	
示例	#include "lierdaGPIO.h" lierdaGPIOToggle (PIN_25);	控制前必须完成如下流程： lierdaGPIOInit lierdaGPIOClaim

2.2.8 GPIO 状态读取函数

功能分类	内容	描述
GPIO 状态读取	lierdaGPIORead	
函数原型	void lierdaGPIORead (PIN pin)	
参数说明	pin: IO 引脚号。（枚举类型值，具体见源代码）	
返回值	1:高电平 0:低电平	
示例	#include "lierdaGPIO.h" bool Value; value = lierdaGPIORead (PIN_25);	控制前必须完成如下流程： lierdaGPIOInit lierdaGPIOClaim

2.2.9 GPIO 中断回调函数

功能分类	内容	描述
GPIO 状态读取	lierdaGPIORegisterCallback	
函数原型	void lierdaGPIORegisterCallback(PIN pin, GPIO_INTERRUPT trigger, GPIO_CALLBACK callback)	
参数说明	pin: IO 引脚号。（枚举类型值，具体见源代	typedef enum {

	码） trigger：触发方式（例如：高电平触发。具体类型见“GPIO_INTERRUPT”枚举类型） callback：回调函数指针	/*上升沿触发中断*/ GPIO_INTERRUPT_RISING_EDGE, /*下降沿触发中断*/ GPIO_INTERRUPT_FALLING_EDGE, /*上升沿或下降沿触发中断*/ GPIO_INTERRUPT_ANY_EDGE, /*低电平触发中断*/ GPIO_INTERRUPT_LOW, /*高电平触发中断*/ GPIO_INTERRUPT_HIGH } GPIO_INTERRUPT;
返回值	无	

示例	#include "lierdaGPIO.h"
	
	gpio_claim(PIN_12,GPIO_DIRECTION_INPUT);
	lierdaGPIORegisterCallback(PIN_12, GPIO_INTERRUPT_FALLING_EDGE,
	sos_key_callback);
	
	void sos_key_callback(PIN pin)
	{
	if(0 == lierdaGPIORead (pin))
	{	

	key_fall_flag = 1;
	}
	}	

2.3 UART 函数

2.3.1 UART 初始化

功能分类	内容	描述
UART 初始化	lierdaUARTInit	
函数原型	Void lierdaUARTInit(UART_HandleTypeDef *huart)	
参数说明	*huart：初始化配置的句柄	
返回值	无	

示例	#include "lierdaUART.h"
	#include "string.h"
	UART_HandleTypeDef lierdaUARTHandle;
	lierdaUARTHandle.baudrate = 9600;
	lierdaUARTHandle.data_bits = UART_DATA_BITS_8;
	lierdaUARTHandle.parity = UART_PARITY_NONE;
	lierdaUARTHandle.stopbits = UART_STOP_BITS_1;
	lierdaUARTHandle.rx_pin = PIN_12;
	lierdaUARTHandle.tx_pin = PIN_14;
	lierdaUARTInit(&lierdaUARTHandle);
		备注：
		Baudrate：波特率（ 4800,9600,57600,115200,230400,460800）
		Parity：校验（支持奇偶校验）
		Stopbits:停止位
		rx_pin：接收引脚
		tx_pin：发送引脚

2.3.2 UART 发送

功能分类	内容	描述

UART 发送	lierdaUARTSend	
函数原型	void lierdaUARTSend(UART_HandleTypeDef *huart,const uint8* buffer, uint32 length)	
参数说明	*huart：初始化配置的句柄 buffer: 需要发送的数据起始地址。 length: 需要发送的数据长度	
返回值	无	
示例	#include "lierdaUART.h" #include "string.h" UART_HandleTypeDef uarthandle; uint8 SendBuff[]= "OpenMcu" lierdaUARTSend(&uarthandle,SendBuff, strlen(sendbuff));	

2.3.3 UART 接收

功能分类	内容	描述
UART 接收	lierdaUARTReceive	
函数原型	uint16lierdaUARTReceive(UART_HandleTypeDef *huart, uint8 *UserDataPtr, uint16 * UserDataLen， uint32_t WiatTimeOut);	
参数说明	*huart：初始化配置的句柄 UserDataPtr: 用户接收缓存区首地址，内部会将接收到的数据拷贝到这个地址。 UserDataLen: 接收到的数据长度。 WaitTimeOut: 等待接收超时时间(单位：	

	tick。默认一个 tick=1ms) 永久等待填 0xFFFFFFFF.	
返回值	0 :失败 1:成功	
示例	#include "lierdaUART.h" uint8 RecvBuff[256]; uint16 RecvLen; lierdaUARTReceive(&uarthandle,RecvBuff, &RecvLen, 3000);	函数体内部采用队列通信，所以用户应用程序调用会导致当前任务进入阻塞。正确接收数据后会唤醒。用户设计应用程序框架时应考虑此情况。

2.4 ADC 函数

NOTE：使用 ADC 相关 API 之前须使用 lierdaAIOCalibrateADC 校准函数校准。

2.4.1 ADC 初始化

功能分类	内容	描述
ADC 初始化	lierdaADCInit	
函数原型	LIERDA_ADC_RET lierdaADCInit(void)	
参数说明	无	
返回值	若成功，返回 LIERDA_ADC_OK 若失败，返回 LIERDA_ADC_FAILED 或 LIERDAS_ERROR_BAD_PARAMS	
示例	#include "lierdaADC.h" Uint8 ret = lierdaADCInit();	

2.4.2 ADC 取消初始化

功能分类	**内容**	**描述**
ADC 取消初始化	lierdaADCDeinit	
函数原型	LIERDA_ADC_RET lierdaADCDeinit (void)	
参数说明	无	
返回值	若成功，返回 LIERDA_ADC_OK 若失败，返回 LIERDA_ADC_FAILED 或 LIERDAS_ERROR_BAD_PARAMS	
示例	#include "lierdaADC.h" Uint8 ret = lierdaADCDeinit ();	

2.4.3 校准电压

功能分类	内容	描述
校准电压	lierdaAIOCalibrateADC	使用 ADC 相关 API 之前必须用此函数校准
函数原型	AIO_FUNC_RET lierdaAIOCalibrateADC(void)	
参数说明	无	
返回值	若成功，返回 AIO_FUNC_RET_OK 若失败，返回 AIO_FUNC_RET_ERROR	
示例	#include "lierdaADC.h" Uint8 ret = lierdaAIOCalibrateADC ();	

2.4.4 获取当前 Vbat 电压和温度

功能分类	内容	描述
获取当前 Vbat 电压和温度	lierdaAIOTempVolt	
函数原型	AIO_FUNC_RET lierdaAIOTempVolt(int16 temp,uint32 voltage)	
参数说明	int16 temp：温度的数值 ℃； uint32 voltage：电压的数值 mV；	读取的温度需要调用温度校准 AT 指令”AT+NCALTEMPSENSOR=xx”进行校准，xx 为实际温度（单位：℃）。
返回值	若成功，返回 AIO_FUNC_RET_OK 若失败，返回 AIO_FUNC_RET_ERROR	
示例	#include "lierdaADC.h" Int16 temp; Uint32 voltage; Uint8 ret = lierdaAIOTempVolt(temp, voltage);	

2.4.5 获取当前 Vbat 电压原始数值

功能分类	内容	描述
获取电压原始数值	lierdaADCGetRaw	用于读取模组 Vbat 电压
函数原型	LIERDA_ADC_RET lierdaADCGetRaw(uint32 *voltage)	
参数说明	uint32 *voltage：读取到的数值	
返回值	:若成功：返回 AIO_FUNC_RET_OK	

	若失败： 返回 AIO_FUNC_RET_ERROR	

示例	#include "lierdaADC.h"
	
	uint32 Vbat=0;
	lierdaADCGetRaw(&Vbat);//ADC 采样 Vbat 电压(原始 ADC 采样值)	

2.4.6 获取当前 PIO 的电压

功能分类	内容	描述
获取当前 PIO 的电压	lierdaReadAIOPin	
函数原型	AIO_FUNC_RET lierdaReadAIOPin(uint32 *voltage, uint8 aio_pin_number)	电源引脚必须接在 AIO1 或者 AIO0 上
参数说明	uint32 *voltage 获取到的电压值 uint8 aio_pin_number 所要获取电压的引脚	voltage 单位： mV aio_pin_number： 0：AIO0 1:AIO1
返回值	若成功，返回 LIERDA_ADC_OK 若失败，返回 LIERDA_ADC_FAILED 或 LIERDAS_ERROR_BAD_PARAMS	
示例	#include "lierdaADC.h" Uint32 voltage; Uint8 ret; ret = lierdaReadAIOPin (&voltage,0);	

2.5 DAC 函数

NOTE：使用 DAC 输出电压时先使“lierdaDACConnect”函数连接 AIO 引脚。

2.5.1 DAC 初始化

功能分类	内容	描述
DAC 初始化	lierdaDACInit	
函数原型	LIERDA_DAC_RET lierdaDACInit (void)	
参数说明	无	
返回值	若成功，返回 LIERDA_DAC_OK 若失败，返回 LIERDA_ DAC _FAILED 或 LIERDAS_DAC_ERROR_BAD_PARAMS	
示例	#include "lierdaDAC.h" Uint8 ret = lierdaDACInit ();	

2.5.2 DAC 取消初始化

功能分类	内容	描述
DAC 取消初始化	lierdaDACDeinit	
函数原型	LIERDA_DAC_RET lierdaDACInit (void)	
参数说明	无	
返回值	若成功，返回 LIERDA_DAC_OK 若失败，返回 LIERDA_ DAC _FAILED 或 LIERDAS_DAC_ERROR_BAD_PARAMS	
示例	#include "lierdaDAC.h" Uint8 ret = lierdaDACDeinit ();	

2.5.3 连接到 AIO 引脚

功能分类	内容	描述
连接到 AIO 引脚	lierdaDACConnect	用于连接 AIO 引脚
函数原型	LIERDA_DAC_RET lierdaDACConnect(uint32 aio)	
参数说明	uint32 aio：要连接的 AIO 引脚号（0 或 1）	填 0 表示将以上电压输出函数作用于 AIO0 上，同理 1 表作用于 AIO1 上。
返回值	若成功，返回 LIERDA_DAC_OK 若失败，返回 LIERDA_ DAC _FAILED 或 LIERDAS_DAC_ERROR_BAD_PARAMS	
示例	#include "lierdaDAC.h" Uint8 ret = lierdaDACConnect (0);	

2.5.4 设定输出电压范围

功能分类	内容	描述
设定输出电压范围	lierdaDACSetRange	
函数原型	LIERDA_DAC_RET lierdaDACSetRange(DAC_VOLTAGE_RANGE range)	
参数说明	DAC_VOLTAGE_RANGE range：设定范围数值（0=1200, 1=2000,2=2800, 3=3600）	typedef enum { DAC_VOLTAGE_RANGE_1200, DAC_VOLTAGE_RANGE_2000, DAC_VOLTAGE_RANGE_2800,

		DAC_VOLTAGE_RANGE_3600, DAC_VOLTAGE_RANGE_MAX, } DAC_VOLTAGE_RANGE;
返回值	若成功，返回 LIERDA_DAC_OK 若失败，返回 LIERDA_ DAC _FAILED 或 LIERDAS_DAC_ERROR_BAD_PARAMS	
示例	#include "lierdaDAC.h" Uint8 ret = lierdaDACSetRange (0);	

2.5.5 输出 DAC 值

功能分类	内容	描述
输出 DAC 值	lierdaDACWriteRaw	
函数原型	LIERDA_DAC_RET lierdaDACWriteRaw (uint32 value)	
参数说明	uint32 value：输出的值，范围(0-1023)	输出的电压值范围是由设定的输出电压范围决定。例如： lierdaDACSetRange (0); lierdaDACWriteRaw (1023); 此时输出的电压是 1200.
返回值	若成功，返回 LIERDA_DAC_OK 若失败，返回 LIERDA_ DAC _FAILED 或 LIERDAS_DAC_ERROR_BAD_PARAMS	

示例	#include "lierdaDAC.h"
	lierdaDACInit(); //DAC 初始化
	lierdaDACSetRange(3);//设置 DAC 输出电压范围
	lierdaDACConnect(1);//用于建立 DAC 连接 AIO1 引脚
	lierdaDACWriteRaw(1023)//输出电压到 AIO1 引脚	

2.6 KV 函数

2.6.1 KV 写入

功能分类	内容	描述
KV 写入	lierdaKVSet	用于保存一些数据到 flash
函数原型	LIERDA_RET lierdaKVSet(lierda_kv_key key, const uint8 *kvalue, uint16 kvalue_length)	
参数说明	lierda_kv_key key：写入的 kv 映射 ID（范围在 0 到 26880）	kvalue_length：每一个 key 最大支持存 256 字节

	uint8 *kvalue：要写入的值 kvalue_length：写入的数值长度	
返回值	若成功，返回 LIERDA_RET_OK（0） 若失败，返回 LIERDA_RET_ERROR	
示例	#include "lierdaKVStorage.h" unsigned char iccid_return_num = 0; lierdaKVSet (0, (uint8*)&iccid_return_num, 1);	NOTE:小写字母存入后读出来的是大写字母，建议存字母时转为 16 进制数存储。

2.6.2 KV 读取

功能分类	内容	描述
KV 读取	lierdaKVGet	用于读取保存到 flash 里的数据
函数原型	LIERDA_RET lierdaKVGet(lierda_kv_key key, uint16 kvalue_max_length, uint16 *kvalue_length, uint8 *kvalue)	
参数说明	lierda_kv_key key：读取的 kv 映射 ID（范围在 0 到 26880） uint16 kvalue_max_length：读取数据的最大长度 kvalue_length：读取的数值长度 uint8 *kvalue：读取的数值	
返回值	若成功，返回 LIERDA_RET_OK（0） 若失败，返回 LIERDA_RET_ERROR	

示例	#include "lierdaKVStorage.h"
	uint16 rsp_len = 0;
	unsigned char temp_num = 0;
	lierdaKVGet (0, sizeof(temp_num),
	&rsp_len,
	(uint8*)&temp_num)	

2.7 DNS 解析函数

功能分类	内容	描述
DNS 解析	lierdaDNSResolve	
函数原型	void lierdaDNSResolve(char *uHostname, char *uIPaddr)	
参数说明	uHostname：需要解析的域名，例如 www.baidu.com uIPaddr:解析的结果，以 IP 形式输出， 例如"111.13.100.91"	
返回值	无	

示例	#include "lierda_app_main.h"
	Char ip_name[20] = {0};
	
	lierdaDNSResolve("www..baid.com",ip_name);
	lierdaLog("%s",ip_name);
		注意：
		解析超时时间为 200s，网络情况不同，DNS 解析的时间就会不同；调用此函数会使任务进入阻塞态，最大阻塞时间 200s.

2.8 软件模拟 SPI 接口函数

2.8.1 SPI 初始化

功能分类	内容	描述
SPI 初始化	lierdaSPISoftInit	
函数原型	uint8 lierdaSPISoftInit(SPI_InitTypeDef *SPI_Init)	
参数说明	SPI_InitTypeDef *SPI_Init	初始化配置句柄
返回值	成功： LierdaSPI_RET_OK 失败：LierdaSPI_RET_ERROR	

示例	#include "lierdaSPIsoft.h"
	#include "lierda_app_main.h"
	SPI_MX25L12835F.DataSize=8;
	SPI_MX25L12835F.Mode=3;
	SPI_MX25L12835F.LierdaSPI_CS=SPI_CS;
	SPI_MX25L12835F.LierdaSPI_SCK=SPI_SCK;
	SPI_MX25L12835F.LierdaSPI_MISO=SPI_MISO;
	SPI_MX25L12835F.LierdaSPI_MOSI=SPI_MOSI;
	
	if(lierdaSPISoftInit(&SPI_MX25L12835F)==Lierda SPI_RET_OK)
	lierdaLog("SPI Init OK");
	else
	lierdaLog("SPI Init ERROR");
		typedef struct
		{
		uint32 Mode; // SPI 模式
		uint32 DataSize; //SPI 数据大小 */
		PIN LierdaSPI_CS; //SPI CS
		PIN LierdaSPI_SCK;// SPI SCK
		PIN LierdaSPI_MISO;//SPI MISO
		PIN LierdaSPI_MOSI;//SPI MOSI
		}SPI_InitTypeDef;
		
		NOTE:定义 SPI 的 CS、SCK、MISO、MOSI 引脚应处于同一电源域，电源域的说明请参考《NB-IoT 模块硬件应用手册 NB86 型》

2.8.2 SPI 写寄存器

功能分类	内容	描述
SPI 写寄存器	lierdaSPIWriteByte	
函数原型	uint8 lierdaSPIWriteByte(SPI_InitTypeDef *lierda_spi, uint8 TxData)	
参数说明	SPI_InitTypeDef *SPI_Init ： SPI 句柄地址 uint8 TxData ： 写入的数据	
返回值	成功： LierdaSPI_RET_OK 失败：LierdaSPI_RET_ERROR	
示例	#include "lierdaSPIsoft.h" #include "lierda_app_main.h"	

	uint32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;
	SPI_CS_ENABLE();//使能器件
	lierdaSPIWriteByte(&SPI_MX25L12835F,W25X_JedecDeviceID); //发送《设备 ID》指令
	
	Temp0=lierdaSPIReadByte(&SPI_MX25L12835F);
	Temp1=lierdaSPIReadByte(&SPI_MX25L12835F);
	Temp2=lierdaSPIReadByte(&SPI_MX25L12835F);
	SPI_CS_DISABLE();//失能器件
	Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;
	return Temp;	

2.8.3 SPI 读寄存器

功能分类	内容	描述
SPI 读寄存器	lierdaSPIReadByte	
函数原型	uint8 lierdaSPIReadByte(SPI_InitTypeDef *lierda_spi)	
参数说明	SPI_InitTypeDef *SPI_Init ： SPI 句柄地址	
返回值	读回来的字节数据	

示例	#include "lierdaSPIsoft.h"
	#include "lierda_app_main.h"
	uint32 Temp = 0, Temp0 = 0, Temp1 = 0, Temp2 = 0;
	SPI_CS_ENABLE();//使能器件
	lierdaSPIWriteByte(&SPI_MX25L12835F,W25X_JedecDeviceID); //发送《设备 ID》指令
	
	Temp0=lierdaSPIReadByte(&SPI_MX25L12835F);
	Temp1=lierdaSPIReadByte(&SPI_MX25L12835F);
	Temp2=lierdaSPIReadByte(&SPI_MX25L12835F);
	SPI_CS_DISABLE();//失能器件
	Temp = (Temp0 << 16) | (Temp1 << 8) | Temp2;
	return Temp;	

2.9 硬件 SPI 接口函数

2.9.1 SPI 初始化

功能分类	内容	描述
SPI 初始化	lierdaSPIInit(SPI_CONFIGURATION spi_config, SPI_PIN spi_pin)	
函数原型	SPI_RET lierdaSPIInit(	

	SPI_CONFIGURATION spi_config, SPI_PIN spi_pin)	
参数说明	spi_config:SPI 相关配置 spi_pin：SPI 引脚配置	
返回值	成功： SPI_RET_OK 失败：SPI_RET_ERROR	

示例	#include "lierdaSPI.h"
	#include "lierda_app_main.h"
	SPI_CONFIGURATION lierdaSPIconfig;
	SPI_PIN lierdaSPIpin;
	lierdaSPIconfig.data_size = 8;
	lierdaSPIconfig.clk_mode = SPI_CLK_MODE3;
	lierdaSPIconfig.clk_div = 0x08;
	lierdaSPIpin.interface = SPI_INTERFACE_SINGLE_UNIDIR;
	lierdaSPIpin.clk_pin = SPI_SCK;
	lierdaSPIpin.csb_pin = SPI_CS;
	lierdaSPIpin.miso_pin = SPI_MISO;
	lierdaSPIpin.mosi_pin = SPI_MOSI;
	if (lierdaSPIInit(lierdaSPIconfig, lierdaSPIpin) == SPI_RET_OK)
	lierdaLog("SPI Init OK");
	else
	lierdaLog("SPI Init ERROR");
		data_size：数据位
		clk_div：数据分频，0x02 为 2M Hz 速率
		clk_mode：SPI 通信时钟模式（0,1,2,3）
		interface：选择 SPI_INTERFACE_SINGLE_UNIDIR 方式即可
		clk_pin：SPI 时钟引脚
		csb_pin：SPI 片选引脚
		miso_pin：SPI MISO 引脚
		mosi_pin：SPI MOSI 引脚
		
		NOTE：定义 SPI 的 CS、SCK、MISO、MOSI 引脚应处于同一电源域，电源域的说明请参考《NB-IoT 模块硬件应用手册 NB86 型》

2.9.2 SPI 发送数据

功能分类	内容	描述
SPI 写寄存器	lierdaSPISendData	
函数原型	SPI_RET lierdaSPISendData(SPI_BUS bus, uint8* cmd_buff, uint16 cmd_len, uint8* data_buff, uint16 data_len, SPI_CALLBACK callback)	
参数说明	SPI_BUS bus：声明的 SPI 总线 uint8* cmd_buff：命令 buffer，对于某些不需要的设备使用时填 NULL，对应的"cmd_len"也填 0 uint16 cmd_len：命令长度 uint8* data_buff：写入的数据 uint16 data_len：写入的数据长度	

	SPI_CALLBACK callback：回调函数	
返回值	成功：SPI_RET_OK 失败：SPI_RET_ERROR	

示例	#include "lierdaSPI.h"
	#include "lierda_app_main.h"
	uint8 writaddr[5]={0};
	uint8 cmd_PageProgram=W25X_PageProgram;
	writaddr[0]=cmd_PageProgram;
	writaddr[1]=(WriteAddr & 0xFF0000)>>16;
	writaddr[2]=(WriteAddr & 0xFF00)>>8;
	writaddr[3]=WriteAddr & 0xFF;
	lierdaSPISendData(0, writaddr,4, pBuffer,
	NumByteToWrite, NULL);	

2.9.3 SPI 读数据

功能分类	内容	描述
SPI 读寄存器	lierdaSPIRecvData	
函数原型	SPI_RET lierdaSPIRecvData(SPI_BUS bus, uint8* cmd_buff, uint16 cmd_len, uint8* data_buff, uint16 data_len, SPI_CALLBACK callback, bool ignore_rx_while_tx)	
参数说明	SPI_BUS bus：声明的 SPI 总线 uint8* cmd_buff：命令 buffer，对于某些不需要的设备使用时填 NULL，对应的"cmd_len"也填 0 uint16 cmd_len：命令长度 uint8* data_buff：写入的数据 uint16 data_len：写入的数据长度 SPI_CALLBACK callback：回调函数 bool ignore_rx_while_tx：当发送时，是否忽略接收。一般填 TRUE	
返回值	成功：SPI_RET_OK 失败：SPI_RET_ERROR	

示例	#include "lierdaSPI.h"
	#include "lierda_app_main.h"
	uint32 Temp = 0;
	uint8 idbuff[4]={0};
	uint8 cmd_JedecDeviceID=W25X_JedecDeviceID;
	lierdaSPIRecvData(0,&cmd_JedecDeviceID,1,idbuff,	

	3,NULL,true);//false true
	Temp = (idbuff[0] << 16) | (idbuff[1] << 8) |
	idbuff[2];
	return Temp;	

3 LiteOS操作系统API接口

3.1 Task 接口

3.1.1 创建 Task

功能分类	内容	描述
创建 Task	osThreadNew	
函数原型	osThreadId_t osThreadNew (osThreadFunc_t func, void *argument, const osThreadAttr_t *attr)	
参数说明	func: 任务入口函数。 argument: 保留 填 NULL 即可。 attr: 任务相关配置参数	func 函数入口参数必须遵循 LiteOS 的定义规则，否则存在运行风险
返回值	返回任务的句柄（所创建线程的标识符）。后续对任务的其它操作都需要用到句柄。	建议用户创建任务后判断返回句柄是否为 NULL 创建失败，根据返回值执行相应的异常处理

示例	#include “cmsis_os2.h”
	void TestTask_main(void *argument);
	//任务属性结构体，
	osThreadAttr_t Test_task_attr;
	//声明一个任务句柄
	osThreadId_t TestTaskHandle;
		name：任务别名
		priority：数值越大优先级越高，目前只支持 10~14，建议 10~12，用户创建过高优先级的任务将导致 SDK 整体运行异常
		stack_size：单位为 4Bytes，即

	//结构体属性赋值
	Test_task_attr.name = "TestTask";
	Test_task_attr.attr_bits = 0;
	Test_task_attr.cb_name = NUll
	Test_task_attr.cb_size = 0;
	Test_task_attr.stack_mem = NULL;
	Test_task_attr.stack_size= 128
	Test_task_attr.Priority= 11;
	Test_task_attr.tz_module = 0;
	Test_task_attr.reserved = 0;
	//创建任务
	TestTaskHandle=osThreadNew(
	TestTask_main,
	NULL,
	&Test_task_attr);
	void TestTask_main(void *argument)
	{
	/* 用户自己的代码 */
	}
		128 实际分配的 stack_size 为 512 bytes
		
		NOTE: 用户自行创建的任务执行前必须等待 500ms 以上，确保 SDK 整体初始化完成

3.1.2 删除 Task

功能分类	内容	描述
删除 Task	osThreadTerminate	
函数原型	osStatus_t osThreadTerminate (osThreadId_t thread_id)	
参数说明	thread_id: 任务句柄	任务内部删除自身仍然需要任务句柄，不能填 NULL。
返回值	typedef enum {	

	osOK = 0, //删除成功
	osError = -1, //未知类型错误
	osErrorTimeout = -2, //删除超时
	osErrorResource = -3, //资源不可用
	osErrorParameter= -4, //参数错误
	osErrorNoMemory = -5, //内存不够
	osErrorISR = -6, //中断中请使用特殊删除函数
	} osStatus_t;	

示例	#include “cmsis_os2.h” osStatus=osThreadTerminate(TestTaskHandle); 	

3.1.3 Task 延时

功能分类	内容	描述
Task 延时	osDelay	如当前仅有一个用户创建的任务，且该任务执行 osDelay，则 Application 核进入低功耗状态。 模组整机的电流应同时考虑 Protocol 核的状态，如当前模组与网络侧处于 PSM 状态，则整个模组功耗为 PSM 功耗。
函数原型	osStatus_t osDelay (uint32_t ticks)	
参数说明	ticks: 延时时间（单位：默认 1 tick = 1 ms）	
返回值	参见 3.1.2 节。	
示例	#include “cmsis_os2.h”	

	osStatus= osDelay (5000);	

3.1.4 Task 挂起

功能分类	内容	描述
Task 挂起	osThreadSuspend	如一个任务被挂起，在恢复其运行状态前，该任务不会被操作系统内核调度，任务内的所有业务均不执行
函数原型	osStatus_t osThreadSuspend (osThreadId_t thread_id)	
参数说明	thread_id: 任务句柄。	
返回值	参见 3.1.2 节。	
示例	#include “cmsis_os2.h” osStatus=osThreadSuspend (TestTaskHandle);	

3.1.5 Task 恢复

功能分类	内容	描述
Task 恢复	osThreadResume	
函数原型	osStatus_t osThreadResume (osThreadId_t thread_id)	
参数说明	thread_id: 任务句柄。	
返回值	参见 3.1.2 节。	
示例	#include “cmsis_os2.h”	

	osStatus= osThreadResume (TestTaskHandle);	

3.2 SoftTime 接口

3.2.1 创建 SoftTime

功能分类	内容	描述
创建 softtime	osTimerNew	创建后必须执行 osTimerStart，否则该定时器不会实际运行，不会触发超时回调
函数原型	osTimerId_t osTimerNew (osTimerFunc_t func, osTimerType_t type, void *argument, constosTimerAttr_t *attr)	
参数说明	func: 定时器回调函数。定时超时后会被调用。 type: 定时器类型 运行一次或者周期运行 typedef enum { osTimerOnce = 0, //< One-shot timer. osTimerPeriodic = 1 //< Repeating timer. } osTimerType_t; argument: 作为定时器回调函数的的参数 attr: 保留未使用 填 NULL 即可。	func 函数入口参数必须遵循 LiteOS 的定义规则，否则存在运行风险
返回值	定时器句柄 ID	
示例	#include “cmsis_os2.h”	

	osTimerId_t xTimers=NULL;
	xTime = osTimerNew(
	vTimerCallback,
	osTimerPeriodic,
	(void*)1,
	NULL);
	if(xTimers == NULL)
	{
	/* 没有创建成功，用户可以在这里加入创建失败的处理机制 */
	}
	void vTimerCallback(void *argument)
	{ }	

3.2.2 删除 SoftTime

功能分类	内容	描述
删除 softtime	osTimerDelete	
函数原型	osStatus_t osTimerDelete(osTimerId_t timer_id)	
参数说明	timer_id :定时器 ID 句柄	
返回值	参见 3.1.2 章节	
示例	#include “cmsis_os2.h” osStatus = osTimerDelete(xTimers);	

3.2.3 启动 SoftTime

功能分类	内容	描述
启动 softtime	osTimerStart	
函数原型	osStatus_t osTimerStart (osTimerId_t timer_id, uint32_t ticks)	
参数说明	timer_id :定时器 ID 句柄 ticks: 定时器超时时间（单位：tick 默认 1 tick = 1 ms）	
返回值	参见 3.1.2 章节	
示例	#include “cmsis_os2.h” osStatus = osTimerStart (xTimers, 10);	

3.2.4 停止 SoftTime

功能分类	内容	描述
停止 softtime	osTimerStop	
函数原型	osStatus_t osTimerStop (osTimerId_t timer_id)	
参数说明	timer_id :定时器 ID 句柄	
返回值	参见 3.1.2 章节	
示例	#include “cmsis_os2.h” osStatus = osTimerStop (xTimers);	

3.2.5 获取 SoftTime 运行状态

功能分类	内容	描述

获取 softtime 状态	osTimerIsRunning	
函数原型	uint32_t osTimerIsRunning(osTimerId_t timer_id)	
参数说明	timer_id :定时器 ID 句柄	

返回值	{
	OS_SWTMR_STATUS_UNUSED,
	//定时器未使用
	OS_SWTMR_STATUS_CREATED,
	//定时器刚创建
	OS_SWTMR_STATUS_TICKING,
	//定时器正在运行
	};	

示例	#include “cmsis_os2.h” uint32_t Vlaue; Vlaue = osTimerIsRunning (xTimers);	

3.3 队列接口

3.3.1 创建队列

功能分类	内容	描述
创建队列	osMessageQueueNew	
函数原型	osMessageQueueId_t osMessageQueueNew(uint32_t msg_count, uint32_t msg_size, const osMessageQueueAttr_t *attr)	
参数说明	msg_count :队列深度	

	msg_size：每条消息的长度 *attr：消息队列属性，默认为 NULL	
返回值	创建成功返回队列句柄； 创建失败返回 NULL	

示例	#include “cmsis_os2.h”
	static osMessageQueueId_t app_incoming_queue = NULL;
	app_incoming_queue = osMessageQueueNew(APP_MAIN_QUEUE_LEN, APP_MAIN_QUEUE_ITEM_SIZE, NULL);	

3.3.2 发送消息

功能分类	内容	描述
发送队列消息	osMessageQueuePut	
函数原型	osStatus_t osMessageQueuePut (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio, uint32_t timeout)	
参数说明	mq_id :创建好的队列句柄 *msg_ptr：指向放入队列的消息的缓冲区的指针 msg_prio：消息优先级 timeout：超时时间	
返回值	返回 osStatus_t typedef enum { osOK = 0, osError = -1,	

	osErrorTimeout = -2, osErrorResource =-3, osErrorParameter = -4, osErrorNoMemory = -5, osErrorISR = -6, osStatusReserved = 0x7FFFFFFF } osStatus_t;	

示例	#include “cmsis_os2.h”
	static osMessageQueueId_t app_incoming_queue = NULL;
	app_incoming_queue = osMessageQueueNew(APP_MAIN_QUEUE_LEN, APP_MAIN_QUEUE_ITEM_SIZE, NULL);
	
	(void)osMessageQueuePut(app_incoming_queue, (void*)&APP_MAIN_QUEUE_TEMP, 0, osNoWait);	

3.3.3 消息接收

功能分类	内容	描述
获取队列消息	osMessageQueueGet	
函数原型	osStatus_t osMessageQueueGet (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio, uint32_t timeout)	
参数说明	mq_id :创建好的队列句柄 *msg_ptr：指向放入队列的消息的缓冲区的指针	

	msg_prio：消息优先级 timeout：超时时间	
返回值	返回 osStatus_t typedef enum { osOK = 0, osError = -1, osErrorTimeout = -2, osErrorResource =-3, osErrorParameter = -4, osErrorNoMemory = -5, osErrorISR = -6, osStatusReserved = 0x7FFFFFFF } osStatus_t;	

示例	#include “cmsis_os2.h”
	
	Uint32 temp;
	static osMessageQueueId_t app_incoming_queue = NULL;
	app_incoming_queue = osMessageQueueNew(APP_MAIN_QUEUE_LEN, APP_MAIN_QUEUE_ITEM_SIZE, NULL);
	
	(void)osMessageQueuePut(app_incoming_queue, (void*)&APP_MAIN_QUEUE_TEMP, 0, osNoWait);
	
	if(osMessageQueueGet(app_incoming_queue,	

	(void*)&temp, NULL, osWaitForever) == osOK)
	{
	//TODO
	}	

4 Log、AT指令、事件状态接口

4.1 Log 调试接口

功能分类	内容	描述
Log 调试	lierdaLog	该函数打印的 Log 默认从 AT command 口打印
函数原型	int lierdaLog(const char *pcFormat, ...);	同 printf 用法,打印数据的最大长度：512 字节。
参数说明	timer_id :定时器 ID 句柄	
返回值	无	

示例	#include “lierda_app_main.h”
	
	uint8 tmp = 10;
	char strbuf[] = "OpenMcu"
	lierdaLog("debug %d %s\r\n", tmp, strbuf);
		NOTE:不能在中断回调函数里用此函数打 log

4.2 LogView 调试接口

功能分类	内容	描述
Logview 调试	lierdaLogview	该函数打印的 Log 可通过 UE Monitor 过滤 Application 关键字

		查看
函数原型	lierdaLogview();	同 printf 用法
参数说明	timer_id :定时器 ID 句柄	
返回值	无	

示例	#include “app_at_log.h”
	uint8 sock = 10;
	lierdaLogview("Socket %d had connected", sock_num);
		NOTE:不能在中断回调函数里用此函数打 log

4.3 下行数据处理函数

4.3.1 初始化

功能分类	内容	描述
下行数据接收初始化	lierdaNNMIDataInit	创建用于推送 NNMI 下行数据的队列
函数原型	void lierdaNNMIDataInit(void)	
参数说明	无	
返回值	无.	

示例	#include “lierdaNNMIData.h”
	lierdaNNMIDataInit();	

4.3.2 接收函数

功能分类	内容	描述
下行数据接收	lierdaNNMIDataReceived	1、使用前需调用初始化函数 lierdaNNMIDataInit()进行初始化；2、需配置 AT+NNMI=1;
函数原型	void lierdaNNMIDataReceived(uint8* nnmi_buff,	

	uint16* nnmi_buff_len,uint32 timeout)	
参数说明	nnmi_buff：接收到的数据 nnmi_buff_len：接收到的数据长度 timeout: 等待接收超时时间(单位：tick。默认一个 tick=1ms)永久等待填 0xFFFFFFFF.	
返回值	无.	

示例	#include “lierdaNNMIData.h”
	lierdaNNMIDataInit();
	static void lierda_nnmi_task(void *unused)
	{
	UNUSED(unused);
	
	uint8 nnmi_test[100];
	uint16 nnmi_test_len;
	
	lierdaNNMIDataInit();
	for(;;)
	{
	
	lierdaNNMIDataReceived(nnmi_test,
	&nnmi_test_len,3000);
	
	lierdaLog("nnmi_test :%s",nnmi_test);
	osDelay(500);
	(void) osThreadYield();
	}
	}
		NOTE：
		执行该接收函数时当前任务会阻塞直到超时触发

4.4 UDP/TCP 下行数据提示信号量

4.4.1 获取信号量

功能分类	内容	描述
UDP/TCP 下行数据提示信号量	lierdaSocketAcquireSemaphore	当 UDP 或 TCP 收到下行数据时，该信号量被激活
函数原型	void lierdaSocketAcquireSemaphore(void)	
参数说明	无	
返回值	无	

示例	#include “lierdaNNMIData.h”
	void lierda_UDP_task(void *param)
	{
	char *udpData_addr=NULL;
	UNUSED(param);
	osDelay(500); //加延时等待模组初始化完成
	lierdaLog("DBG_INFO:UDP 下行数据接收测试");
	for (;;)
	{
	lierdaSocketAcquireSemaphore();
	udpData_addr=lierdaATCall("AT+NSORF=1,512", 3000); //读取 UDP/TCP 下行的数据
	lierdaLog("DBG_INFO:result:%s", udpData_addr);//打印 UDP/TCP 下行的数据
	osDelay(1);
	}
		如果收到下行数据，激活此信号量，此时用 AT+NSORF 指令读取下行的 UDP/TCP 数据即可。
		NOTE：
		执行该接收函数时当前任务会阻塞直到有下行 UDP/TCP 消息。

	}	

4.5 AT 指令调用接口

功能分类	内容	描述
AT 指令调试	lierdaATCall	
函数原型	char * lierdaATCall (char *at_cmd_buf_param，uint16 timeout)	
参数说明	at_cmd_buf_param :需要模组执行的完整 AT 指令，不用加\r\n Timeout:AT 指令执行超时时间。	
返回值	以字符串形式返回 AT 指令的执行结果.	
示例	#include “lierda_app_main.h” char RecBuff[100]; RecBuff = lierdaATCall("AT+CGPADDR",5000); lierdaLog("Recv Vlaue = %s", RecBuff);	

4.6 事件通知与状态参数

功能分类	内容	描述
事件状态更新	lierda_module_status_read	
函数原型	void lierda_module_status_read(void)	
参数说明	无	
返回值	无	
示例	#include "lierda_module_status.h"	事件通知和状态机的参数放在

	lierda_module_status_read();
	lierdaLog("STATUS: %s\r\n %s\r\n %s\r\n",
	module_status.charCGATT,
	module_status.charCSCON,
	module_status.charCEREG);
		结构体 lierdaModStatus 里面，用户调用函数 lierda_module_status_read(); 更新最新的参数，然后可以把需要的参数打印出来。
		typedef struct
		{
		char charCGATT[AT_MAX_STRING_LEN];//模组附着网络状态
		char charCSCON[AT_MAX_STRING_LEN];//模组连接基站状态，设置 AT+CSCON=1 才能生效
		char charCEREG[AT_MAX_STRING_LEN];//模组连接核心网状态，设置 AT+CEREG=1 才能生效
		char charCPSMS[AT_MAX_STRING_LEN];//PSM 模式指示
		char charNPSMR[AT_MAX_STRING_LEN];//模组是否进入 PSM 状态，设置 AT+NPSMR=1 之后才能生效
		char

		charNMSTATUS[AT_PLUS_MAX_STRING_LEN];// LWM2M 注册状态
		char charFOTASTATUS[AT_PLUS_MAX_STRING_LEN];//FOTA 状态
		char charSockNum[AT_PLUS_MAX_STRING_LEN];//TCP 创建的 Socket 状态，建立连接 TCP 服务器之后才能生效
		}lierdaModStatus;
		
		extern lierdaModStatus module_status;

4.7 FOTA 状态查询

功能分类	内容	描述
FOTA 状态获取	lierda_FotaStatus	用于查询此时 FOTA 状态
函数原型	lierda_fotaSta lierda_FotaStatus(void);	
参数说明	无参数	
返回值	以枚举变量的形式返回 FOTA 的状态 typedef enum { FOTA_NON=0,//无 FOTA 状态 FOTA_DOWNLOADING,//FOTA 下载中 FOTA_DOWNLOAD_FAILED,//FOTA	

	下载失败 FOTA_DOWNLOADED,//FOTA 下载结束 FOTA_UPDATING,//FOTA 加载中 FOTA_UPDATE_SUCCESS,//FOTA 升级成功 FOTA_UPDATE_FAILED,//FOTA 升级失败 FOTA_UPDATE_OVER,//FOTA 升级结束 }lierda_fotaSta;	

示例	#include “lierda_module_status.h”
	
	`if(lierda_FotaStatus()==FOTA_NON)`
	`{`
	`lierdaLog("DBG_INFO:无FOTA事件发生");`
	`}`
	`else if(lierda_FotaStatus()==FOTA_UPDATE_OVER)`
	`{`
	`lierdaLog("DBG_INFO:FOTA事件结束");`
	`}`	

4.8 FOTA 环境下判断能否发数据

功能分类	内容	描述
数据业务环境判断	lierda_FotaEnableData	此函数用于 FOTA 环境下检测能否发数据，客户有 FOTA 需

		求的建议发数据前做一下判断
函数原型	lierdaFota lierda_FotaEnableData(void);	
参数说明	无参数	
返回值	LierdaFota_DataEnable：可以做发数据做业务 LierdaFota_DataDisable：不可以发数据做业务	

示例	#include “lierda_module_status.h”
	
	`if(lierda_FotaEnableData()==LierdaFota_DataEnable)`
	`{`
	`lierdaLog("DBG_INFO:无FOTA事件或FOTA事件结束，可以发数据");`
	`lierdaSendMsgToPlatform((uint8*) "30313233343536373839", 10,MSG_NON_NORAI, 0, 0); //发送数据(ASCII)`
	`lierdaSendMsgToPlatform((uint8*) "1234567890", 10, MSG_NON_NORAI, 0,1); //发送数据(原数据)`
	`}`
	`else`
	`{`
	`lierdaLog("DBG_INFO:正在FOTA中，不可以发数据");`
	`}`	

4.9 向 IoT 平台发送 CON 或 NON 数据

功能分类	内容	描述
向 IOT 平台发送数据	lierdaSendMsgToPlatform	
函数原型	CLOUD_RETlierdaSendMsgToPlatform(	

	uint8 *data, uint16 data_len, MSG_MODE mode, uint8 seq_num, uint8 isHEX)	
参数说明	uint8 *data：需要传送的数据 uint16 data_len：数据长度 MSG_MODE mode：发送数据模式 uint8 seq_num：0~255，根据 AT 指令手册一般填 0 uint8 isHEX：0：不转为十六进制数据 （发送数据时，发送的是 data 是 16 进制字符串） 1：转为十六进制数据（发送数据时，发送的是 data 是原始的数据结构，不是 16 进制字符串，接口内部会转换为 16 进制字符串进行发送。）	MSG_MODE mode： MSG_NON_NORAI (0x0000): 发送 NON 消息 ； MSG_NON_WITHRAI (0x0001): 发送 NON 消息，且携带 RELEASE 释放辅助指示； MSG_NON_WITH_SENDBACKRAI (0x0010): 发送 NON 消息，且携带 RELEASE_AFTER_REPLY 释放辅助指示 ； MSG_CON_NORAI (0x0100): 发送 CON 消息 ； MSG_CON_WITHRAI (0x0101): 发送 CON 消息，且携带 RELEASE_AFTER_REPLY 释放辅助指示
返回值	若成功：返回 CLOUD_RET_OK 若失败：返回相应错误原因 CLOUD_RET	

示例	#include "lierdaCloudConnected.h"
	
	`lierda_module_status_read();`
	`//读取此时LWM2M状态`	

	if(strstr(module_status.charNMSTATUS,"MO_DATA_ENABLED"))
	
	lierdaSendMsgToPlatform((uint8 *)"123456789",9,*MSG_NON_NORAI*,0,1); //发送数据	

5 传感器接口

5.1 华大北斗 GPS 接口

NOTE：GPS 串口初始化实际上是初始化第三路串口，若客户使用 GPS 串口则第三路串口将被占用。

功能分类	内容	描述
GPS 串口初始化	lierda_user_GPS_uart_init	先执行串口初始化函数，后执行创建并启动 GPS 任务函数
函数原型	void lierda_user_GPS_uart_init(uint32 uBaudRate, uint8 uRX_pin, uint8 uTX_pin)	
参数说明	uBaudRate 串口波特率 uRX_pin 串口接收管脚 uTX_pin 串口发送管脚	
返回值	无	
示例	#include "lierda_gps.h" lierda_user_GPS_uart_init(9600, PIN_26, PIN_24); // 初始化 GPS 串口	

5.1.1 创建开启 GPS 任务

功能分类	内容	描述
创建并开启 GPS 任务	lierda_GPS_task_enable	先执行串口初始化函数，后执行创建并启动 GPS 任务函数
函数原型	void lierda_GPS_task_enable(void)	
参数说明	无	
返回值	无	
示例	#include "lierda_gps.h" lierda_GPS_task_enable(); // 创建开启 GPS 任务	如果线程创建失败 ，会打印 lierdaLog("USRTTaskHandle,fail");

5.1.2 终止 GPS 任务

功能分类	内容	描述
终止 GPS 任务	lierda_GPS_task_disable	终止 GPS 任务运行后，将不会再获取到 GPS 数据
函数原型	void lierda_GPS_task_disable(void);	
参数说明	无	
返回值	无	
示例	#include "lierda_gps.h" lierda_GPS_task_disable();	终止 GPS 任务执行

5.1.3 获取 GPS 数据

功能分类	内容	描述
获取 GPS 数据	lierda_GPS_data_get	
函数原型	void lierda_GPS_data_get(GPSRMCStruct *p_gps_user);	

参数说明	GPSRMCStruct *p_gps_user	
返回值	无	

示例	#include "lierda_gps.h"
	
	typedef struct
	{
	char UTC_Time[20];
	//Current time hhmmss.sss
	char Valid_status;
	//V = Data Invalid / Receiver Warning, A=Data Valid
	char Latitude[20];
	//User datum latitude degrees, minutes, decimal minutes format
	char NS_indicator;
	//N/S Indicator Hemisphere N=north or S=south
	char Longitude[20];
	//User datum latitude degrees, minutes, decimal minutes format
	char EW_indicator;
	//E/W indicator 'E'= East, or 'W' = West
	char Spd[20];
	//Speed Speed Over Ground
	char cog[20];
	//COG Course Over Ground
	char Date[7];
		用户自定义一个 GPS 数据结构体，获取结构体指针赋值，GPS 定位成功后，GPS 数据每一秒更新一次。

	//Current Date in Day, Month Year
	format ddmmyy
	}GPSUSERStruct;
	
	GPSUSERStruct GPSUser;
	
	lierda_GPS_data_get((GPSRMCStruct*)
	&GPSUser); // 用户获取 GPS 数据	

5.2 HDC1000 接口

5.2.1 传感器 I2C 初始化

NOTE：使用 I2C 接口的传感器库之前，需使用此函数进行 I2c 总线初始化，初始化一次即可。此时不得调用 lierdaI2CInit 初始化函数进行初始化。

功能分类	内容	描述
传感器 I2C 初始化	lierda_sensor_I2C_init	先执行传感器 I2C 初始化函数，后执行各个传感器初始化
函数原型	void lierda_sensor_I2C_init(uint8 SCL_pin, uint8 SDA_pin);	
参数说明	SCL_pin I2C SCL 时钟管脚 SDA_pin I2C SDA 数据管脚	
返回值	无	

示例	#include "lierda_HDC1000.h"
	
	lierda_sensor_I2C_init(PIN_14, PIN_15);
	// 初始化传感器 I2C 接口	

5.2.2 HDC1000 初始化

功能分类	内容	描述
HDC1000 初始化	lierda_HDC1000_Init	在执行 HDC1000 初始化函数之前，必须先执行传感器 I2C 初始化函数
函数原型	uint8 lierda_HDC1000_Init(void);	
参数说明	无	
返回值	1， 成功 0，失败	
示例	#include "lierda_HDC1000.h" lierda_HDC1000_Init(); // 初始化 HDC1000	

5.2.3 获取 HDC1000 温湿度数据

功能分类	内容	描述
获取 HDC1000 温湿度数据	lierda_HDC1000_UpdateInfo	
函数原型	void lierda_HDC1000_UpdateInfo(int16 *Temper,int16 *Humidity);	参数类型：有符号整型
参数说明	int16 *Temper 温度 int16 *Humidity 湿度	
返回值	无	
示例	#include "lierda_HDC1000.h" int16 temperature = 0;	

	int16 humidity = 0;
	
	lierda_HDC1000_UpdateInfo (&temperature, &humidity); // 获取 HDC1000 的温度，湿度数据	

5.3 OPT3001 接口

5.3.1 OPT3001 初始化

功能分类	内容	描述
OPT3001 初始化	lierda_OPT3001_Init	在执行 OPT3001 初始化函数之前，必须先执行传感器 I2C 初始化函数
函数原型	uint8 lierda_OPT3001_Init(void);	
参数说明	无	
返回值	1，成功 0，失败	
示例	#include " lierda_OPT3001DN.h" lierda_OPT3001_Init (); //初始化 OPT3001	

5.3.2 获取 OPT3001 环境光数据

功能分类	内容	描述
获取 OPT3001 环境光数据	lierda_OPT3001_UpdataInfo	
函数原型	void lierda_OPT3001_UpdataInfo(uint32 *Lux);	

参数说明	uint32 *lux 环境光照	
返回值	无	

示例	#include " lierda_OPT3001DN.h"
	
	uint32 lux = 0;
	
	lierda_OPT3001_UpdataInfo (&lux); // 获取 OPT3001 环境光数据	

5.4 LIS3DH 接口

5.4.1 LIS3DH 初始化

功能分类	内容	描述
LIS3DH 初始化	lierda_LIS3DH_Init	在执行 LIS3DH 初始化函数之前，必须先执行传感器 I2C 初始化函数
函数原型	uint8 lierda_LIS3DH_Init(void);	
参数说明	无	
返回值	1，成功 0，失败	
示例	#include " lierda_LIS3DHTR.h" lierda_OPT3001DN_Init (); // 初始化 LIS3DH	

5.4.2 获取 LIS3DH 三轴数据

功能分类	内容	描述

获取 LIS3DH 三轴数据	lierda_LIS3DH_UpdateInfo	
函数原型	void lierda_LIS3DH_UpdateInfo(int16 *LIS3DH_X,int16 *LIS3DH_Y,int16 *LIS3DH_Z);	参数类型：有符号整型
参数说明	int16 *LIS3DH_X int16 *LIS3DH_Y int16 *LIS3DH_Z	
返回值	无	

示例	#include " lierda_LIS3DHTR.h"
	
	int16 lis3dh_x = 0;
	int16 lis3dh_y = 0;
	int16 lie3dh_z = 0;
	
	lierda_LIS3DH_UpdateInfo (&lis3dh_x, &lis3dh_y, &lis3dh_z); // 获取 LIS3DH 三轴数据	

6 NOTE

功能分类	内容
队列信号量	创建的队列、信号量等需要返回句柄的资源时，一定要做错误处理或者 log 打印（辅助调试找 bug）
软件定时器	软件定时器回调函数中不能调用会导致定时器任务阻塞的 API（比如队列发送、接收 API 的超时参数不能为非 0，不能调用 osDelay）、函数内不宜做太多

	事情，建议通过 IPC 通知其他任务处理。
操作系统配置	在 los_config.h 中配置任务模块。配置 LOSCFG_BASE_CORE_TSK_CONFIG 系统支持最大任务数，队列、信号量、定时器等资源最大数量都在这里配置。用户产品过于复杂需要资源更多时一定注意这几项配置。
用户任务配置	除去 Idle 任务以外，系统可配置的任务资源个数是指：整个系统的任务资源总个数，而非用户能使用的任务资源个数。例如：系统软件定时器多占用一个任务资源数，那么系统可配置的任务资源就会减少一个。
	应用任务的优先级不能太高，建议 10 或者 11. 在任务开始处应加延时，保证海思任务能正常初始化完成
Log 打印	推荐使用 lierdaLogview 输出应用 Log，可通过 UE Monitor 连接模组 DBG_TX 过滤 Application 关键字查看，此 log 打印不能在中断回调中使用，否则会引起模组异常复位
外设	使用 I2C 传感器库时，对 I2C 初始化需使用 lierda_sensor_I2C_init 函数进行初始化，调用一次即可。
	使用华大北斗 GPS 传感器库时，调用 GPS 串口初始化 lierda_user_GPS_uart_init 函数实际上是初始化第三路串口，此时用户不得再次创建第三路串口
	使用 SPI 接口时注意定义的引脚应处于同一电源域
AT 指令	“AT+LPVER=XXXX”AT 指令用于客户配置自己的版本或其他信息。配置好可由 AT 指令”AT+LPVER?”读出所配置的值。配置的字符串长度最大为 30 个字节。

7 相关文档及术语缩写

以下相关文档提供了文档的名称，版本请以最新发布的为准。

序号	文档名称	注释
[1]	Lierda NB-IoT模组V150 OpenCPU版本更新说明	

[2]	Lierda NB-IoT模组DEMO说明文档	
[3]	Lierda NB-IoT模组V150 OpenCPU开发环境搭建指南	