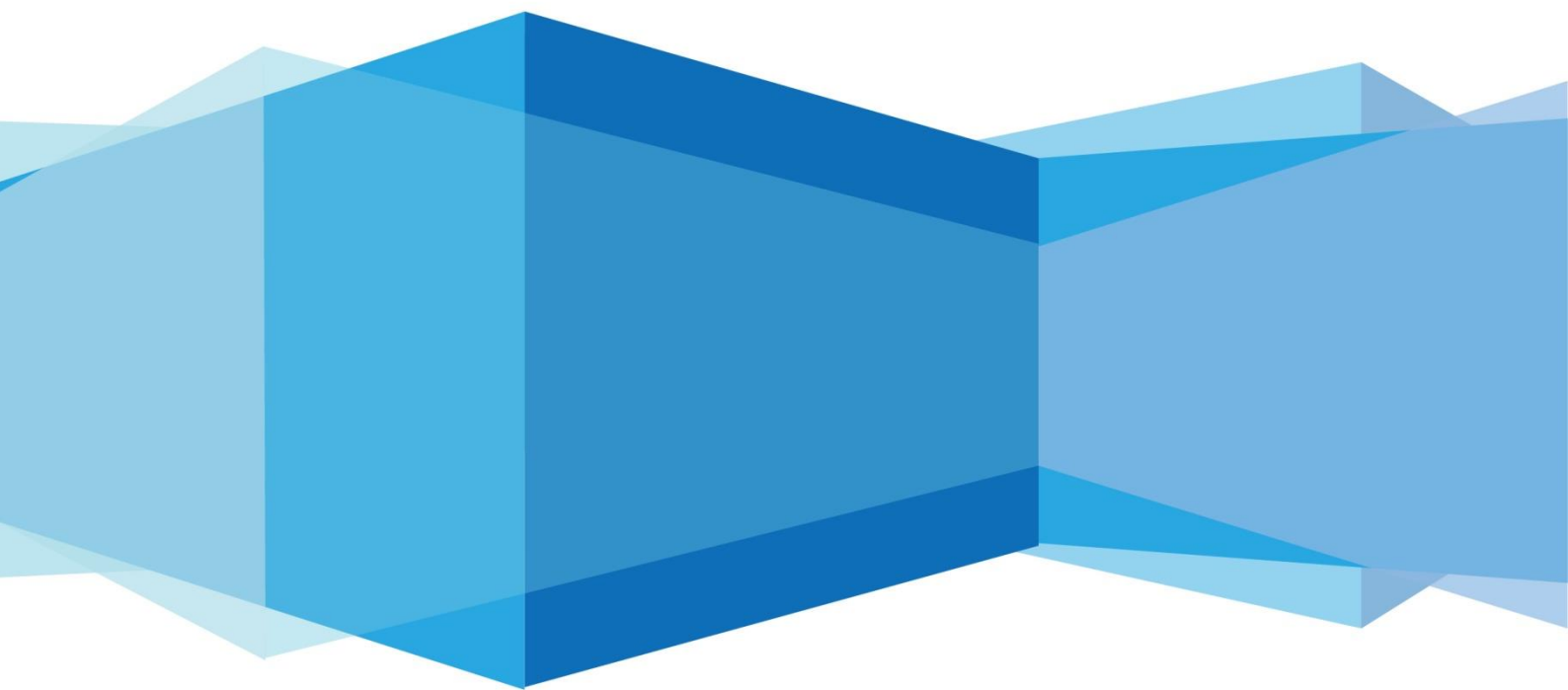


Lierda NB-IoT 模组 OpenCPU

DemoCode 说明文档

版本: Rev1.8

日期: 2019-04-03



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。

Lierda Technology Group Co., Ltd.

文件修订历史

版本	修订日期	修订日志
1.0	2018-09-07	第一次发布版本
1.1	2018-09-09	文档格式修改
1.2	2018-09-29	文档格式修改
1.3	2018-10-10	新增串口接收，GPIO 中断，队列收发，NNMI 数据接收
1.4	2018-11-16	新增 kv 储存，ADC/DAC，传感器库
1.5	2018-11-26	新增 B300SP5 固件打包方式
1.6	2019-01-02	新增 DNS 解析、UDP/TCP 下行数据接收、软件模拟 SPI 接口函数
1.7	2019-01-10	修改 openCPU 固件烧写方法
1.8	2019-04-03	增加事件状态说明接口；新增硬件 SPI 接口函数；新增 LWM2M 数据发送接口；更改获取当前 Vbat 电压原始数值接口函数；新增 FOTA 规避相关 API；新增固件生成批处理文件，方便客户合成固件包；新增常见编译出错解决办法。

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）

Lierda Science & Technology Group Co.,Ltd

Address: 杭州市余杭区文一西路 1326 号利尔达物联网科技园

利尔达科技集团: <https://www.lierda.com/>

Tel: 86-571-88800000

物联网开发者社区 <http://bbs.lierda.com/>

Email: nbiot_support@lierda.com
--

Lierda Technology Group Co., Ltd..

目录

法律声明.....	2
文件修订历史	3
适用模块型号	4
目录	6
1 概览	9
1.1 Demo 程序流程说明	10
2 DEMO 程序编写说明	13
2.1 添加应用文件 (xx.c/xx.h)	13
2.1.1 application_core 文件下添加用户文件.....	14
2.1.2 lib 文件夹里创建用户所需的文件.....	15
2.2 创建任务	16
2.3 配置任务相关信息.....	16
2.4 lierda_App_task 任务.....	17
2.4.1 LED 功能函数说明 (GPIO 输出)	18
2.4.2 按键函数测试 (GPIO 读)	20
2.4.3 AT 指令测试函数.....	22
2.4.4 I2C 调用说明	22
2.4.5 队列测试.....	24
2.4.6 KV 测试.....	25
2.4.7 ADC 测试	26
2.4.8 DAC 测试.....	27
2.4.9 SPI 测试	27
2.4.10 DNS 解析接口函数.....	31
2.5 lierda_NNMI_task 任务.....	32
2.5.1 NNMI 下行数据测试.....	33
2.6 lierda_USRT_task 任务.....	34
2.6.1 串口数据收发说明	35
2.7 传感器库	36
2.7.1 HDC1000 温湿度传感器	37
2.7.2 OPT3001 光照传感器.....	37

2.7.3	LIS3DH 三轴传感器	37
2.7.4	华大北斗 GPS	37
2.8	lierda_UDP_TCP_task 任务	38
2.8.1	UDP 下行数据接收测试	39
2.8.2	TCP 下行数据接收测试	40
2.9	事件状态通知	41
2.9.1	相关接口函数介绍	41
2.10	FOTA 相关 API.....	42
2.10.1	FOTA 状态查询 API.....	42
2.10.2	判断 FOTA 过程中能否做业务	43
2.11	LWM2M 数据发送接口.....	44
3	工程编译	45
3.1	编译相关配置	45
3.2	编译步骤	45
3.3	常见编译出错解决办法.....	46
3.3.1	编译后报找不到 xx.a 库	46
3.3.2	编译后报.py 导入出错.....	46
3.3.3	环境变量未添加导致编译出错.....	46
3.3.4	编译后报编码问题	46
4	生成 Package 固件包	47
4.1	openCPU_B300SP2 版本.....	47
4.1.1	UpdatePackage 软件安装.....	47
4.1.2	Package 包生成	47
4.2	openCPU_B300SP5&B500SP1 版本.....	50
4.2.1	UpdatePackage 软件安装.....	50
4.2.2	Package 包生成	50
4.3	openCPU Package 批处理文件	52
4.3.1	UpdatePackage 软件安装.....	52
4.3.2	Package 包生成	53
5	固件烧录	54
5.1	UEUpdaterUI 软件安装	54
5.2	固件烧录	54

6	程序调试.....	55
6.1	通过 AT 指令串口打印 Log.....	55
6.2	通过 UE Log 串口打印 Log.....	56
7	相关文档及术语缩写.....	57

Lierda Technology Group Co., Ltd..

1 概览

本文旨在将 Lierda NB86-G 模组 Open CPU SDK 程序中包含的 Demo 程序进行说明，用于向用户展示基于 Lierda API 如何在模组上进行二次开发简单应用程序。

主要描述 Lierda NB86-G 模组 Open CPU 交付件中的：src 文件夹下 application_core 文件夹，如图 1-1-1、build_scons 文件夹下 application.bin 文件如图 1-1-2、Demo 示例的开发及调试过程，方便客户使用 NB86-G 模组 Open CPU API 设计、调试自己的应用程序。

src 文件夹下 application_core 文件夹：存放 main.c 以及应用程序的.c 文件和.h 文件。

application.bin 文件：应用核镜像文件，客户编译后生成的应用核镜像文件，用于后面生成 Package 包，详见 4.2 小节。

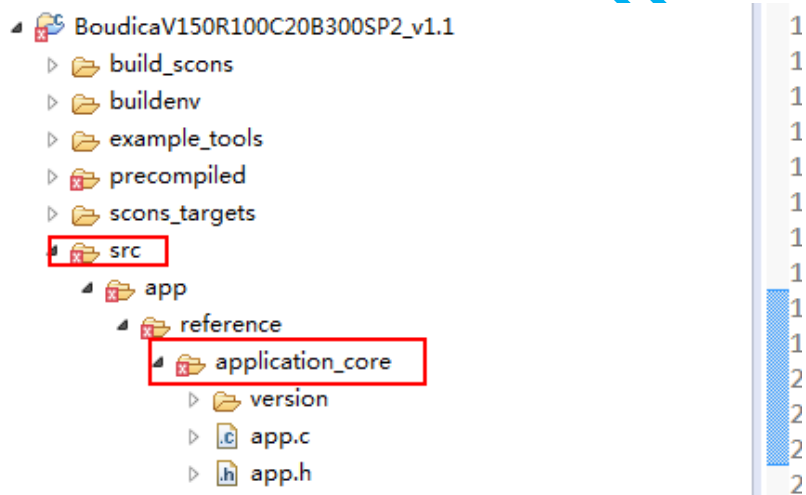


图 1-1-1 应用程序存放位置图

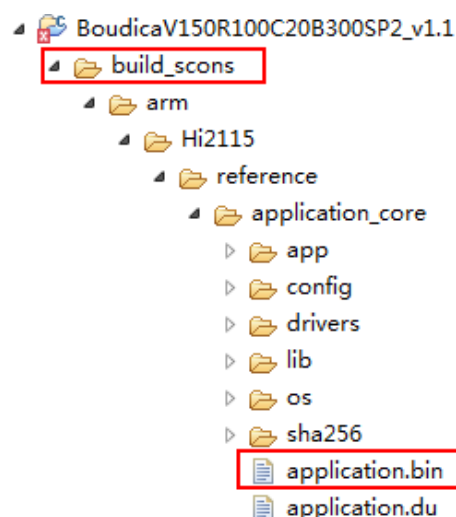


图 1-1-2 应用核镜像文件存放位置图

1.1 Demo 程序流程说明

Demo 程序主要以测试 AT 指令、GPIO(加中断)、I2C、串口、NNMI 下行数据处理以及队列消息发送与接收为目的，Demo 程序创建了三个任务和一个队列，lierda_App_task 任务用于跑测试函数（GPIO、I2C、SPI、AT 指令、串口、两个 log 打印函数、KV 存储、ADC、DAC，如图 1-1-3）；lierda_NNMI_task 任务用于处理 NNMI 下行数据，如图 1-1-4。lierda_USRT_task 任务用于处理串口接收数据，如图 1-1-5。lierda_UDP_TCP_task 任务用于处理 UDP/TCP 下行数据，如图 1-1-6 应用程序文件位置如图 1-1-7。程序采用两种方式获取传感器数据：传感器库的方式调用和自己写驱动调用。程序默认传感器库的方式调用(define USE_LIERDA_SENSOR 1 //定义是否使用传感器库)

```

27 //*****
28 * @函数名    app任务线程
29 * @参数    param : 空参数, 无效
30 * @返回值    无
31 //*****
32 void lierda_App_task(void *param)
33 {
34     UNUSED(param);
35     uint8 temp=0;
36     osDelay(500); //等待模组初始化完成
37     lierdaLog("DBG_INFO:Demo测试程序"); //通过AT指令串口打印Log测试
38     Lierda_boardInit(); //传感器&外设初始化
39     Lierda_ATDemotest(); //AT指令测试
40     Lierda_Led_App(); //GPIO测试函数 (LED闪烁)
41     Lierda_KVTest(); //KV测试
42     lierdaLog("DBG_INFO:等待按键中断来临, 请按下KEY_2"); //通过AT指令串口打印Log测试
43     for (;;)
44     {
45         if (osMessageQueueGet(mess_QueueId, &temp, NULL, 0xffffffff) == osOK) //队列接收按键中断发送的消息
46         {
47             if (temp == 5) //按键中断来临, 开始做测试任务
48             {
49                 Lierda_I2CSensorTest(); //I2C传感器测试
50                 AT_sendData(); //LWM2M发送数据测试
51                 lierdaUARTSend(&UARTHandle, (uint8 *) "这是一个串口测试\r\n", strlen((char *) "这是一个串口测试\r\n")); //串口发送测试
52                 Lierda_ADC_Test(); //ADC测试
53                 Lierda_DAC_Test(); //DAC测试
54                 lierdaLogview("DBG_INFO:%d", 111); //通过UE Log串口打印Log测试
55             }
56         }
57         osDelay(1); //用于任务切换
58     }
59 }
60

```

图 1-1- 3 lierda_App_task 任务

```

//*****
* @函数名    NNMI任务线程, 用于处理NNMI下行数据
* @参数    param : 空参数, 无效
* @返回值    无
//*****
void lierda_NNMI_task(void *param)
{
    uint8 i;
    UNUSED(param);
    osDelay(500); //加延时等待模组初始化完成
    lierdaNNMIDataInit(); //NNMI下行数据接收初始化
    for (;;)
    {
        lierdaNNMIDataReceived(nnmi_buff, &nnmi_buff_len, 0xFFFFFFFF); //NNMI下行数据接收
        if (nnmi_buff_len > 0)
        {
            lierdaLog("下行数据 :");
            for(i=0; i<nnmi_buff_len; i++)
            {
                lierdaLog("%x", nnmi_buff[i]);
            }
            memset(nnmi_buff, 0, nnmi_buff_len);
            nnmi_buff_len = 0;
        }
        osDelay(20);
    }
}

```

图 1-1- 4 lierda_NNMI_task 任务

```

16 /*****
17 * @函数名  串口接收任务线程, 用于处理串口数据接收
18 * @参数 param : 空参数, 无效
19 * @返回值 无
20 *****/
21 void lierda_USRT_task(void *param)
22 {
23     UNUSED(param);
24     osDelay(500); //加延时等待模组初始化完成
25     usrt_init();
26     for (;;)
27     {
28         lierdaUARTReceive(&lierdaUARTHandle, usrt_buff, &usrt_buff_len, 0xFFFFFFFF); //串口数据接收
29         if (usrt_buff_len > 0)
30         {
31             lierdaLog("串口数据 :%s", usrt_buff);
32             memset(usrt_buff, 0, usrt_buff_len);
33             usrt_buff_len = 0;
34         }
35         osDelay(10);
36     }
37 }

```

图 1-1- 5 lierda_USRT_task 任务

```

/*****
* @函数名  UDP/TCPData下行数据处理任务线程
* @参数 param : 空参数, 无效
* @返回值 无
*****/
void lierda_UDP_TCP_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/TCP下行数据接收测试");
    for (;;)
    {
        lierdaSocketAcquireSemaphore();
        udpcmd_buff[9]=socket_num;
        memcpy(udpcmd_buff + 10, "512", 4);
        lierdaLog("DBG_INFO:cmd:%s", udpcmd_buff);
        udpData_addr = lierdaATCall(udpcmd_buff, 3000); //读取数据
        lierdaLog("DBG_INFO:result:%s", udpData_addr); //打印UDP/TCP下行的数据
        osDelay(1);
    }
}

```

图 1-1- 6 lierda_UDP_TCP_task 任务

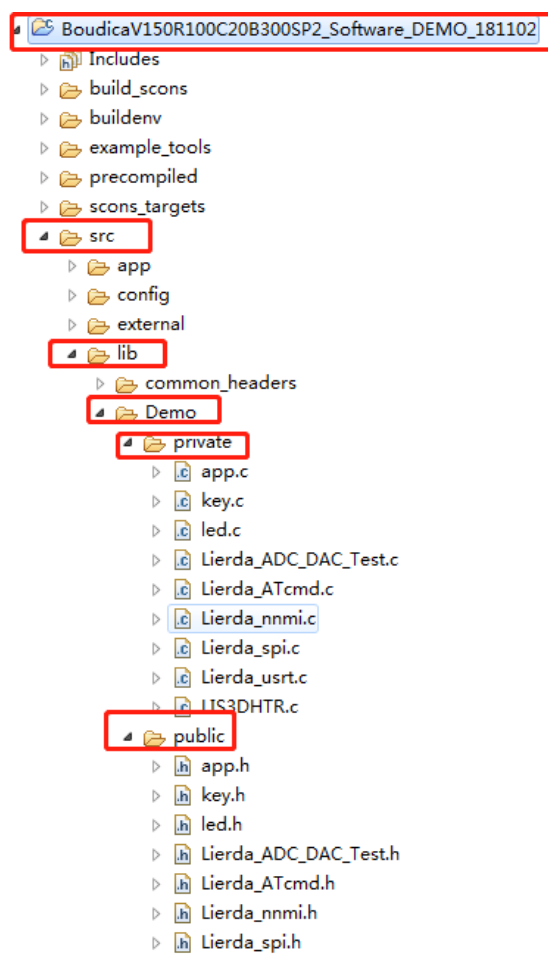


图 1-1-7 应用程序文件位置

2 DEMO程序编写说明

2.1 添加应用文件（xx.c/xx.h）

添加 xx.c 文件和 xx.h 文件可以在 application_core 文件下添加如图 2-1-1。也可以在 src 文件夹下 lib 文件夹里创建用户所需的文件夹，如图 2-1-2。

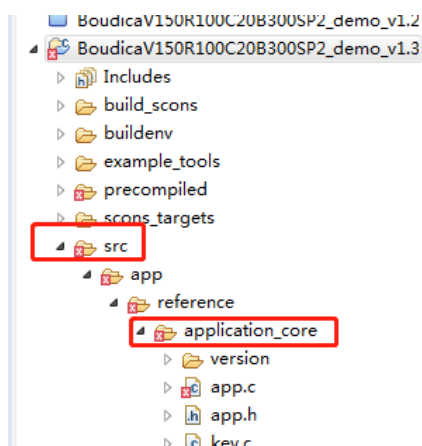


图 2-1- 1 application_core 文件下添加用户文件

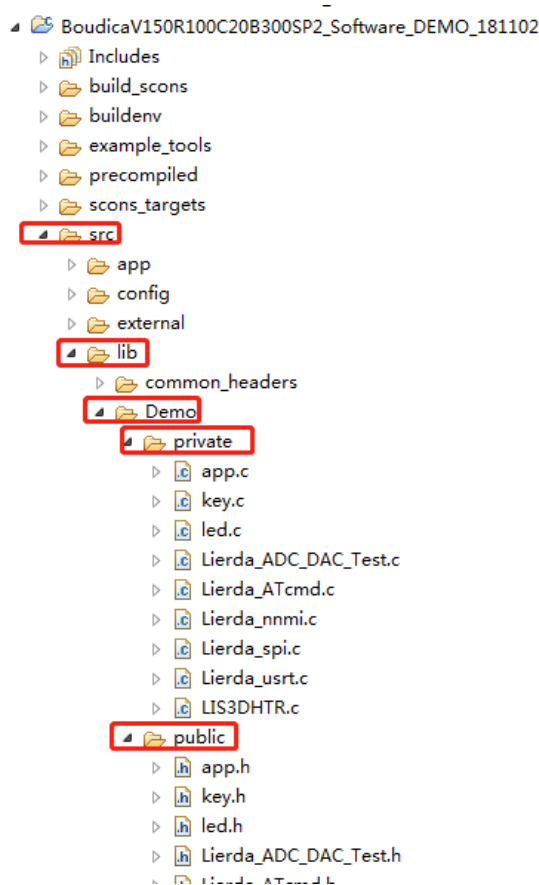


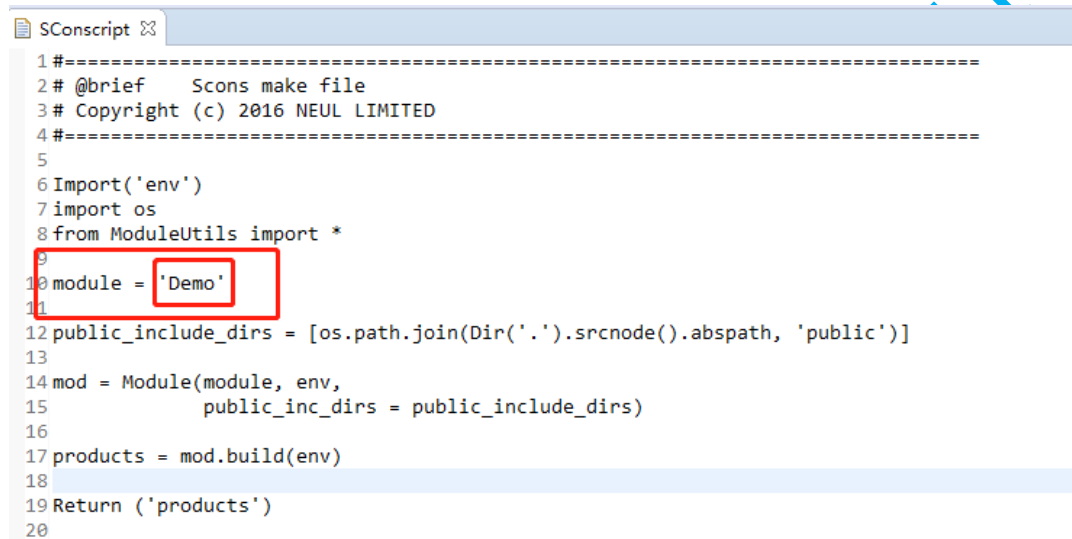
图 2-1- 2 lib 文件夹下添加用户文件

2.1.1 application_core 文件下添加用户文件

用户在 application_core 文件下添加用户文件不需要做编译脚本的改动，只需将所用的 xx.c 和 xx.h 文件添加在 application_core 文件下即可。

2.1.2 lib 文件夹里创建用户所需的文件

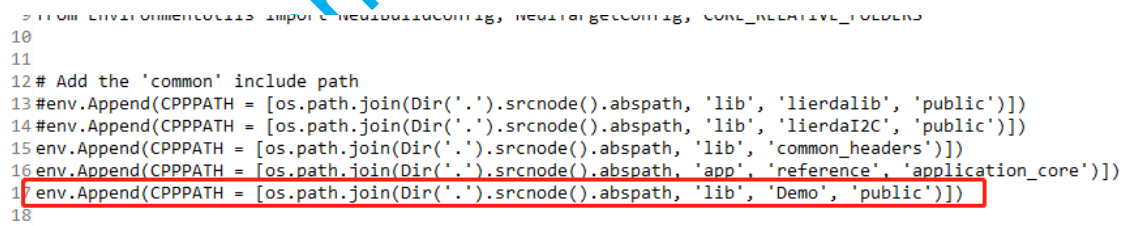
- 1、在 lib 文件夹下创建文件夹，用户创建的文件夹里必须按一定格式创建子文件：一个 private 文件夹存放 xx.c，一个 public 文件夹存放 xx.h 文件，还要创建一个 SConscript 脚本文件（可从其他文件里复制过来更改相应配置即可）。
- 2、更改用户创建文件夹下 SConscript 脚本配置，只需更改一个地方即可如图 2-1-3. module = 'xxxx'，xxxx 为客户创建的文件名。



```
1 #=====
2 # @brief   Scons make file
3 # Copyright (c) 2016 NEUL LIMITED
4 #=====
5
6 Import('env')
7 import os
8 from ModuleUtils import *
9
10 module = 'Demo'
11
12 public_include_dirs = [os.path.join(Dir('.').srcnode().abspath, 'public')]
13
14 mod = Module(module, env,
15               public_inc_dirs = public_include_dirs)
16
17 products = mod.build(env)
18
19 Return ('products')
20
```

图 2-1-3 用户文件里 SConscript 脚本配置

- 3、更改 src 文件下 SConscript 脚本配置，需要更改三个地方如图 2-1-4 和 2-1-5.



```
10 from EnvironmentUtils import NEUL_I2C_CONFIG, NEUL_I2C_CONFIG, CORE_NEUTRAL_FOLDERS
11
12 # Add the 'common' include path
13 #env.Append(CPPPATH = [os.path.join(Dir('.').srcnode().abspath, 'lib', 'lierdalib', 'public')])
14 #env.Append(CPPPATH = [os.path.join(Dir('.').srcnode().abspath, 'lib', 'lierdaI2C', 'public')])
15 env.Append(CPPPATH = [os.path.join(Dir('.').srcnode().abspath, 'lib', 'common_headers')])
16 env.Append(CPPPATH = [os.path.join(Dir('.').srcnode().abspath, 'app', 'reference', 'application_core')])
17 env.Append(CPPPATH = [os.path.join(Dir('.').srcnode().abspath, 'lib', 'Demo', 'public')])
18
```

图 2-1-4 src 文件里 SConscript 脚本配置

```

79 platform = [
80     #os.path.join('lib', 'arm_core', NeulTargetConfig.get_build_chip_family(env)),
81     #os.path.join('lib', 'arm_core', NeulTargetConfig.get_build_chip(env)),
82     #os.path.join('os', NeulTargetConfig.get_build_os(env)),
83     #os.path.join('os', 'cmsis'),
84     #os.path.join('drivers'),
85     #os.path.join('lib', 'irmalloc'),
86     #os.path.join('lib', 'nlibc'),
87     #os.path.join('lib', 'preserve'),
88     #os.path.join('lib', 'lierdolib'),
89     #os.path.join('lib', 'lierdaI2C'),
90     os.path.join('lib', 'Demo'),
91 ]
92
93 env.Append(CPPDEFINES = [ 'USE_FREERTOS' ]) #for consistency with legacy code
94 env.Append(CPPDEFINES = [ 'USE_CMSIS_OS' ])
95 else:
96 platform = [
97     #os.path.join('lib', 'arm_core', NeulTargetConfig.get_build_chip_family(env)),
98     #os.path.join('lib', 'arm_core', NeulTargetConfig.get_build_chip(env)),
99     #os.path.join('drivers'),
100    #os.path.join('lib', 'irmalloc'),
101    #os.path.join('lib', 'nlibc'),
102    #os.path.join('lib', 'preserve'),
103    #os.path.join('lib', 'lierdolib'),
104    #os.path.join('lib', 'lierdaI2C'),
105    os.path.join('lib', 'Demo'),
106 ]

```

图 2-1- 5 src 文件里 SConscript 脚本配置

NOTE:建议客户在 application_core 文件下创建文件，2.1.2 所述的方法涉及脚本配置，若脚本配置错误会导致工程编译出错。

2.2 创建任务

在 main.c 文件的 lierda_app_main() 里创建用户所用的任务如图 2-2-1.

```

37 project_data project_userdata = {0},
38
39 void lierda_app_main(void);
40 void hw_init(void);
41
42 /*****
43  * @函数名 在此函数里面创建用户线程
44  * @参数 param : 空参数
45  * @返回值 无
46  * @重要修改记录 180830, qhq 创建函数
47  *****/
48 void lierda_app_main(void)
49 {
50     lierda_App_main();
51 }
52
53 void hw_init(void)
54 {
55     hal_cpu_init();
56     hal_reboot_init();
57

```

图 2-2- 1 创建任务函数

2.3 配置任务相关信息

1、创建任务句柄结构体

NOTE: 优先级用 10-12.

```
osThreadAttr_t App_task_attr = {"lierda_App_task"/*任务名称*/, 0, NULL, 0, NULL, (128)
/*任务堆栈大小*/, 10/*任务优先级*/, 0, 0};
```

- 2、创建测试任务（lierda_App_task）、NNMI 下行数据接收任务（lierda_NNMI_task）、串口接收数据处理任务（lierda_USRT_task）、UDP/TCP 下行数据接收任务（lierda_UDP_TCP_task）如图 2-3-1。



```
/* *****
 * @函数名 创建用户线程
 * @参数 param : 空参数
 * @返回值 无
 * ***** */
void lierda_App_main(void)
{
    App_task_handle = osThreadNew(lierda_App_task, NULL, &App_task_attr); //创建测试任务
    if (App_task_handle == NULL)
    {
        lierdaLog("lierda_App_task任务创建失败");
    }
    NNMI_task_handle = osThreadNew(lierda_NNMI_task, NULL, &NNMI_task_attr); //创建NNMI下行数据接收任务
    if (NNMI_task_handle == NULL)
    {
        lierdaLog("lierda_NNMI_task任务创建失败");
    }
    UDP_TCP_task_handle = osThreadNew(lierda_UDP_TCP_task, NULL, &UDP_TCP_task_attr); //创建UDP/TCP下行数据接收任务
    if (UDP_TCP_task_handle == NULL)
    {
        lierdaLog("lierda_UDP_TCP_task任务创建失败");
    }
    #if USE_LIERDA_SENSOR //如果定义使用传感器库
    ;
    #else //不使用传感器库
    UART_task_handle = osThreadNew(lierda_UART_task, NULL, &UART_task_attr); //创建串口数据接收任务
    if (UART_task_handle == NULL)
    {
        lierdaLog("lierda_UART_task任务创建失败");
    }
    #endif
    mess_QueueId = osMessageQueueNew(2, 1, NULL); //创建队列
    if (mess_QueueId == NULL)
    {
        lierdaLog("mess_Queue创建失败");
    }
}
```

图 2-3-1 创建任务

2.4 lierda_App_task 任务

lierda_App_task 任务用于 GPIO、I2C、AT 指令、串口、两个 log 打印函数的测试。

NOTE: 进入任务函数后加一个延时（100ms-1s 秒），等待模组上电后初始化完成。否则可能造成程序不正常运行如图 2-4-1。

```

30 ⑥/*****
31  * @函数名   app任务线程
32  * @参数   param : 空参数, 无效
33  * @返回值 无
34  *****/
35 void lierda_App_task(void *param)
36 {
37     UNUSED(param);
38     uint8 temp;
39     uint16 temp1, temp2, temp3;
40     osDelay(500); //等待模组初始化完成
41     lierdaLog("DBG_INFO:这是一个测试程序"); //通过AT指令串口打印Log测试
42     usrt_init(); //串口初始化
43     lierdaGPIOInit(); //GPIO初始化
44     Lierda_Led_Init(LED_0, GPIO_DIRECTION_OUTPUT); //LED IO声明
45     Lierda_KEY_Init(KEY2, GPIO_DIRECTION_INPUT); //按键 IO声明
46     I2c_Init(); //I2C初始化
47     LIS3DH_init(); //LIS3DH三轴传感器初始化
48     lierdaATDemotest(); //AT指令测试

```

图 2-4- 1 任务函数

2.4.1 LED 功能函数说明（GPIO 输出）

1、GPIO 的初始化和声明，GPIO 初始化函数：void lierdaGPIOInit(void)如图 2-4-2； GPIO 声明函数：bool lierdaGPIOClaim(PIN pin,GPIO_DIRECTION dir)如图 2-4-3。

```

30 ⑥/*****
31  * @函数名   app任务线程
32  * @参数   param : 空参数, 无效
33  * @返回值 无
34  *****/
35 void lierda_App_task(void *param)
36 {
37     UNUSED(param);
38     uint8 temp;
39     uint16 temp1, temp2, temp3;
40     osDelay(500); //等待模组初始化完成
41     lierdaLog("DBG_INFO:这是一个测试程序"); //通过AT指令串口打印Log测试
42     usrt_init(); //串口初始化
43     lierdaGPIOInit(); //GPIO初始化
44     Lierda_Led_Init(LED_0, GPIO_DIRECTION_OUTPUT); //LED IO声明
45     Lierda_KEY_Init(KEY2, GPIO_DIRECTION_INPUT); //按键 IO声明
46     I2c_Init(); //I2C初始化
47     LIS3DH_init(); //LIS3DH三轴传感器初始化
48     lierdaATDemotest(); //AT指令测试
49     lierdaLog("DBG_INFO:等待按键中断来临, 请按下KEY_2"); //通过AT指令串口打印Log测试
50     for (;;)
51     {

```

图 2-4- 2 GPIO 初始化

```

9- /*****
0 * @函数名    GPIO声明
1 * @参数 led_pin:初始化的GPIO引脚
2 * @参数      mode: 引脚的工作模式: 输出/输入
3 * @返回值 无
4 *****/
5- void Lierda_led_Init(PIN led_pin,GPIO_DIRECTION mode)
6 {
7     lierdaGPIOClaim(led_pin,mode); //GPIO声明
8 }

```

图 2-4- 3 GPIO 声明

参数说明: led_pin 要使用的 GPIO mode: GPIO 的工作方式 (输出/输入)

2、GPIO 的高电平输出实现 LED 灯的灭如图 2-4-4。

```

8- /*****
9 * @函数名    LED灭
0 * @参数 led_pin:GPIO引脚
1 * @返回值 无
2 *****/
3- void Lierda_LED_OFF(PIN led_pin)
4 {
5     lierdaGPIOSet(led_pin); //输出高电平
6 }

```

图 2-4- 4 GPIO 的高电平输出

3、GPIO 的低电平输出实现 LED 灯的亮如图 2-4-5。

```

9- /*****
0 * @函数名    LED亮
1 * @参数 led_pin:GPIO引脚
2 * @返回值 无
3 *****/
4- void Lierda_LED_ON(PIN led_pin)
5 {
6     lierdaGPIOClear(led_pin); //输出低电平
7 }

```

图 2-4- 5 GPIO 的低电平输出

4、GPIO 电平翻转如图 2-4-6。

```

7- /*****
8 * @函数名    LED电平翻转
9 * @参数 led_pin:GPIO引脚
0 * @返回值 无
1 *****/
2- void Lierda_LED_Toggle(PIN led_pin)
3 {
4     lierdaGIOToggle(led_pin); //翻转引脚的电平
5 }

```

图 2-4- 6 GPIO 电平翻转

5、LED 应用程序, 高低电平实现 LED 灯的闪烁如图 2-4-7。

```

16- /*****
17-  * @函数名    LED测试函数
18-  * @参数    无
19-  * @返回值    无
20-  *****/
21- void Lierda_Led_App(void)
22- {
23-     Lierda_LED_ON(LED_0); //打开LED灯
24-     osDelay(500);
25-     Lierda_LED_OFF(LED_0); //关闭LED灯
26-     osDelay(500);
27-     Lierda_LED_ON(LED_0); //打开LED灯
28-     osDelay(500);
29-     Lierda_LED_OFF(LED_0); //关闭LED灯
30- }

```

图 2-4- 7 LED 应用程序

2.4.2 按键函数测试（GPIO 读）

- 1、使用时要对 GPIO 的进行初始化和声明， GPIO 初始化见 2.3.2 小节的 LED 灯的 GPIO 初始化，GPIO 声明如图 2-4-8。

NOTE:使用 GPIO 中断时需进行中断初始化，如图 2-4-8。

```

21- /*****
22-  * @函数名    GPIO声明和中断初始化
23-  * @参数    key_pin:初始化的GPIO引脚
24-  * @参数    mode: 引脚的工作模式：输出/输入
25-  * @返回值    无
26-  *****/
27- void Lierda_KEY_Init(PIN key_pin, GPIO_DIRECTION mode)
28- {
29-     lierdaGPIOClaim(key_pin, mode); //GPIO声明
30-     lierdaGPIORegisterCallback(key_pin, GPIO_INTERRUPT_LOW, sos_key_callback); //按键中断初始化
31- }

```

图 2-4- 8 GIPO 声明、GPIO 中断初始化

- 2、KEY 功能函数，高电平返回 1，低电平返回 0 如图 2-4-9。

```

/*****
 * @函数名    GPIO读取函数
 * @参数    key_pin:读取引脚
 * @返回值    1: 高电平    0: 低电平
 *****/
uint8_t Lierda_KEY_Read(PIN key_pin)
{
    if (lierdaGPIORead(key_pin))
    {
        osDelay(10); //消抖
        if (lierdaGPIORead(key_pin)) //GPIO读取函数
            return 1;
        else
            return 0;
    }
    else
        return 0;
}

```

图 2-4- 9 KEY 功能函数

3、GPIO 中断回调函数，使用 GPIO 中断首先要对中断进行初始化（见 2.3.3 的小节 1），中断回调函数为用户自己编写，如图 2-4-10。

```

/*****
 * @函数名    按键中断回调函数
 * @参数    PIN 中断引脚
 * @返回值    无
 *****/
void sos_key_callback(PIN pin)
{
    if (0==lierdaGPIORead(pin))
    {
        osDelay(10); //消抖
        if (lierdaGPIORead(pin)==0) //GPIO读取函数
        {
            osMessageQueuePut(mess_QueueId, &KEY_flag, 0,osNowait);//队列发送消息
        }
    }
}

```

图 2-4- 10 GPIO 中断回调函数

4、测试结果，当按键中断来临时，队列会发送消息给 lierda_App_task 任务，lierda_App_task 任务收到队列的消息，开始做测试任务如图 2-4-11。

```

[11:04:43.438]收←◆
DBG_INFO:这是一个测试程序

[11:04:51.122]收←◆
DBG_INFO:等待按键中断来临, 请按下KEY_2

[11:04:51.682]收←◆
AT+MLWEVTIND=0

AT+MLWEVTIND=3

[11:04:54.373]收←◆
DBG_INFO:按键中断来临, 开始做测试

[11:04:55.910]收←◆
X:256,Y:0,Z:16128

```

图 2-4-11 按键中断测试结果

2.4.3 AT 指令测试函数

NOTE: 如图 2-4-12, 红框里要写入的 AT 指令不用加 “\r\n”, “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?” 读出来。

```

3 //*****
3 * @函数名   AT指令测试函数
3 * @参数     无
3 * @返回值   无
3 //*****
3 void lierdaATDemotest(void)
3 {
3     at_cgpadddr_ret = lierdaATCall("AT+NCDP?", 3000); //查询NCDP
3     lierdaLog("DBG_INFO:NCDP?:%s", at_cgpadddr_ret);
3     if (strstr(at_cgpadddr_ret, "+NCDP:180.101.147.115,5683") == NULL)
3     {
3         lierdaATDemoCall("AT+CFUN=0", "OK", 3000, 5); //关闭射频
3         lierdaATDemoCall("AT+NCDP=180.101.147.115,5683", "OK", 3000, 5); //配置NCDP
3         lierdaATDemoCall("AT+NRB", "OK", 7000, 5); //复位保存
3     }
3     lierdaATDemoCall("AT+CGATT=1", "OK", 3000, 5); //附着网络
3     lierdaATDemoCall("AT+CGATT?", "+CGATT:1", 1000, 20); //查询是否附着上网络
3     lierdaATDemoCall("AT+CGPADDR", "+CGPADDR:0", 2000, 10); //查询核心网分配地址
3     lierdaATDemoCall("AT+NNMI=1", "OK", 2000, 5); //打开CoAP数据下行通知
3     lierdaATDemoCall("AT+CPSMS=1", "OK", 2000, 5); //打开PSM模式
3     lierdaATDemoCall("AT+NPSMR=1", "OK", 2000, 5); //打开PSM模式通知
3     lierdaATDemoCall("AT+LPVER=LIERDA-OPENCPU_V2.3", "OK", 2000, 5); //客户设置版本或其他信息, 客户可根据自己的需求更改(不支持小写字母)
3     at_cgpadddr_ret = lierdaATCall("AT+LPVER?", 3000); //查询设置的厂商
3     lierdaLog("DBG_INFO:+LPVER?:%s", at_cgpadddr_ret);
3 }
3 //*****

```

图 2-4-12 AT 指令测试函数

2.4.4 I2C 调用说明

1、I2C 初始化, 传入 I2C 的 SCL 和 SDA 引脚如图 2-4-13。

```

39 / *****
40 * @函数名 I2C初始化函数
41 * @参数 param : 空参数
42 * @返回值 无
43 *****/
44 void I2c_Init(void) {
45     lierdaI2CInit(I2C_SCL, I2C_SDA); //初始化I2C
46 }

```

图 2-4- 13 I2C 初始化

2、LISD3DH 初始化, 进行三轴传感器初始化前要保证 I2C 初始化成功, I2C 初始化后, 如果能成功读到传感器的名称, 说明 I2C 可以正常通信如图 2-4-14。

```

39 }
40 *****
41 * @函数名 三轴传感器初始化函数
42 * @参数 param : 空参数
43 * @返回值 1: 成功 0 : 失败
44 *****/
45 uint8 LIS3DH_init(void)
46 {
47     uint8 ucResponse;
48     uint8 ucState, ucDataTemp = 0x05;
49     lierdaI2CReadreg(&sensorI2CHandle, gssI2C_ADDRESS, LIS3DH_WHO_AM_I, &ucResponse, 1, 0); //读取设备名称
50     while ((ucResponse != 0x33) && ((ucDataTemp-- > 0)))
51     { //读取设备名称异常
52         lierdaI2CReadreg(&sensorI2CHandle, gssI2C_ADDRESS, LIS3DH_WHO_AM_I, &ucResponse, 1, 0);
53         if (ucDataTemp <= 1)
54         { //读取名称失败
55             return 0;
56         }
57     }
58     ucResponse = LIS3DH_SetODR(LIS3DH_ODR_100Hz);
59     if (ucResponse == 1)
60     {
61         osDelay(1);
62         ucResponse = LIS3DH_SetMode(LIS3DH_LOW_POWER);
63         if (ucResponse == 1)
64         {
65             osDelay(1);
66             //set Fullscale
67             ucResponse = LIS3DH_SetFullScale(LIS3DH_FULLSCALE_2);
68         }
69     }
70 }

```

图 2-4- 14 LISD3DH 初始化

3、LISD3DH 数据读取, 首先读取中断位, 当有数据触发中断时, 进行数据的读取如图 2-4-15。

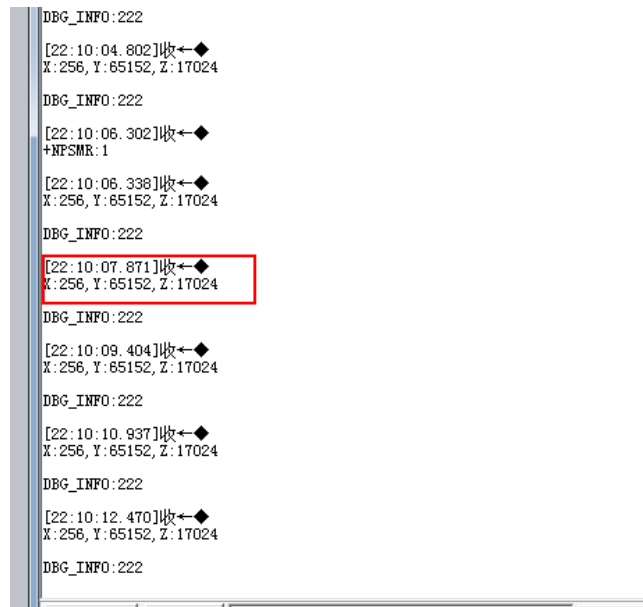
```

39 }
40 *****
41 * @函数名 三轴传感器数据获取函数
42 * @参数 *LIS3DH_X : X轴数据地址
43 * @参数 *LIS3DH_Y : Y轴数据地址
44 * @参数 *LIS3DH_Z : Z轴数据地址
45 * @返回值 无
46 *****/
47 void LIS3DHUpdateInfo(int16 *LIS3DH_X, int16 *LIS3DH_Y, int16 *LIS3DH_Z)
48 {
49     uint8 ucResponse, position;
50     ucResponse = LIS3DH_Get6DPosition(&position); //中断位检测
51     if ((ucResponse == 1) && (position))
52     {
53         LIS3DH_GetAccAxesRaw(&gssAccData); //数据读取
54         *LIS3DH_X = gssAccData.AXIS_X;
55         *LIS3DH_Y = gssAccData.AXIS_Y;
56         *LIS3DH_Z = gssAccData.AXIS_Z;
57     }
58 }
59 *****

```

图 2-4- 15 LISD3DH 数据读取

4、LISD3DH 数据 AT 串口输出信息如图 2-4-16。



```
DBG_INFO:222
[22:10:04.802]收<◆
X:256,Y:65152,Z:17024
DBG_INFO:222
[22:10:06.302]收<◆
+NPSMR:1
[22:10:06.338]收<◆
X:256,Y:65152,Z:17024
DBG_INFO:222
[22:10:07.871]收<◆
X:256,Y:65152,Z:17024
DBG_INFO:222
[22:10:09.404]收<◆
X:256,Y:65152,Z:17024
DBG_INFO:222
[22:10:10.937]收<◆
X:256,Y:65152,Z:17024
DBG_INFO:222
[22:10:12.470]收<◆
X:256,Y:65152,Z:17024
DBG_INFO:222
```

图 2-4- 16 LISD3DH 数据串口输出

5、LISD3DH 数据逻辑分析仪波形如图 2-4-17。

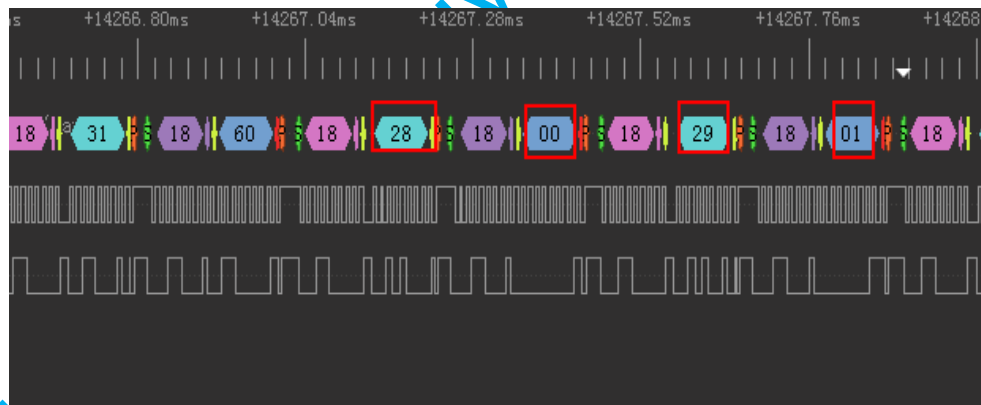


图 2-4- 17 LISD3DH 数据逻辑分析仪波形

分析：读 0x28 寄存器（X 轴低位寄存器）的值为 0x00,读 0x29 寄存器（x 轴高位寄存器）的值为 0x01，由此算出 x 轴的值为 256，和图 2-4-16 所对应。

2.4.5 队列测试

1、队列创建，如图 2-4-18。


```

114  * @返回值 无
115  *****/
116 void lierda_App_main(void)
117 {
118     App_task_handle = osThreadNew(lierda_App_task, NULL, &App_task_attr); //创建任务
119     if (App_task_handle == NULL)
120     {
121         lierdaLog("App_task_handle任务创建失败");
122         return;
123     }
124     NNMI_task_handle = osThreadNew(lierda_NNMI_task, NULL, &NNMI_task_attr); //创建任务
125     if (NNMI_task_handle == NULL)
126     {
127         lierdaLog("NNMI_task_handle任务创建失败");
128         return;
129     }
130     mess_QueueId = osMessageQueueNew(2, 1, NULL); //创建队列
131     if (mess_QueueId == NULL)
132     {
133         lierdaLog("mess_Queue创建失败");
134         return;
135     }
136 }
137 *****/

```

图 2-4- 18 创建队列

2、队列的发送和接收，发送调用 `osStatus_t osMessageQueuePut (osMessageQueueId_t mq_id, const void *msg_ptr, uint8_t msg_prio, uint32_t timeout)` 函数，接收调用 `osStatus_t osMessageQueueGet (osMessageQueueId_t mq_id, void *msg_ptr, uint8_t *msg_prio, uint32_t timeout)` 函数。队列的发送和接收测试见 2.4.2 的 GPIO 中断测试。

NOTE: 队列的发送和接收都会引起任务的阻塞，用户在使用过程中合理安排等待时间。

2.4.6 KV 测试

KV：保存数据到 flash。客户在使用的过程中注意写入的 kv 映射 ID（范围在 0 到 26880）。

测试代码如图 2-19.测试结果如图 2-4-20.

```

174  * @返回值 无
175  *****/
176 uint8 kv_buff[56]={0};
177 uint16 kvdata_len;
178 static void Lierda_KVTest(void)
179 {
180     if (lierdaKVSet(kv_KEY, (const uint8 *) "kvtest lierda", strlen("kvtest lierda")) != NEUL_RET_OK) //设定KV值
181         lierdaLog("kv设置失败");
182     else
183     {
184         if (NEUL_RET_OK == lierdaKVGet(kv_KEY, sizeof(kv_buff), &kvdata_len, kv_buff)) //读取KV值
185             lierdaLog("DBG_INFO:kv_buff: %s", kv_buff);
186         else
187             lierdaLog("DBG_INFO:kv读取失败");
188     }
189 }
190

```

图 2-4- 19 kv 测试代码

```

[17:23:39.409]收←◆
REBOOT_CAUSE_SECURITY_RESET_PIN
Lierda
OK

[17:23:39.962]收←◆
DBG_INFO:Demo测试程序

[17:23:50.394]收←◆
AT+MLWEVTIND=3

[17:23:52.091]收←◆
DBG_INFO:kv_buff:  lierda kv test

DBG_INFO:等待按键中断来临, 请按下KEY_2

```

图 2-4- 20 kv 测试结果

2.4.7 ADC 测试

- 1、读取 Vbat 引脚原始电压采样值: LIERDA_ADC_RET lierdaADCGetRaw(uint32 *voltage);
- 2、读取 Vbat 引脚和环境温度（需要矫正，读出来有偏差）: lierdaAIOTempVolt(int16 *temp,uint32 *voltage);
- 3、读取 AIO 引脚的电压采样值: lierdaReadAIOPin(uint32 *voltage, uint8 aio_pin_number);

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

测试代码如图 2-4-21.测试结果如图 2-4-22.

```

/*****
* @函数名    ADC测试 (1: 读取Vbat引脚电压 2: 读取Vbat电压和温度信息 3: 读取相应的AIO引脚电压)
* @参数      无
* @返回值    无
*****/
void Lierda_ADC_Test(void)
{
    uint32 Vbat=0,Vbat1=0,AIO_V=0;int16 temp=0;

    lierdaADCGetRaw(&Vbat);//ADC采样Vbat电压(原始ADC采样值)
    lierdaLog("DBG_INFO:Vbat=%d", Vbat); //通过AT指令串口打印ADC采样Vbat电压(原始ADC采样值)

    if (AIO_FUNC_RET_OK == lierdaAIOTempVolt(&temp, &Vbat1))//ADC采样温度和电压
        lierdaLog("DBG_INFO:Vbat=%d,temp_t=%d", Vbat1, temp); //通过AT指令串口打印ADC采样Vbat电压(mv)和温度(℃)
    else
        lierdaLog("DBG_INFO:电源电压和温度读取失败");
    if(AIO_FUNC_RET_OK==lierdaReadAIOPin(&AIO_V, LIERDA_AIO0))//ADC采样AIO0电压
        lierdaLog("DBG_INFO:AIO_V=%d", AIO_V); //通过AT指令串口打印ADC采样AIO0电压(mv)
    else
        lierdaLog("DBG_INFO:AIO_V读取失败");
}

```

图 2-4- 21 ADC 测试代码

```
DBG_INFO:Vbat=776
DBG_INFO:Vbat=3248,temp_t=34
DBG_INFO:AIO_V=3272
DBG_INFO:AIO电压输出成功
[17:43:06.356]收←◆
+NPSMR:1
```

图 2-4- 22 ADC 测试结果

2.4.8 DAC 测试

NOTE:使用前需使用 `lierdaDACInit(void)`函数进行初始化和使用 `lierdaDACConnect(uint32 aio)`函数进行声明要连接的 AIO 口。

测试代码如 2-4-23。测试结果可用万用表测量相应的 AIO 引脚电压。

```
3 /*****
3 * @函数名    DAC测试(输出电压到相应AIO引脚)
1 * @参数      无
2 * @返回值    无
3 *****/
4 void Lierda_DAC_Test(void)
5 {
5     if(LIERDA_DAC_OK==lierdaDACWriteRaw(1023))//输出电压到AIO引脚
7         lierdaLog("DBG_INFO:AIO电压输出成功");
3     else
3         lierdaLog("DBG_INFO:AIO电压输出失败");
3 }
1
```

图 2-4- 23 DAC 测试代码

2.4.9 SPI 测试

OpenCPU 方案提供两套 SPI 接口供客户选择：硬件 SPI 和软件模拟 SPI，具体操作使用方式如下：

2.4.9.1 软件 SPI 测试

概述：这里操作 flash 芯片来做 SPI 通信测试，flash 芯片型号：MX25L12835FM2I。

该芯片的时钟模式为 0 或者 3.如图 2-4-24.使用软件 SPI 测试前需打开软件 SPI 测试开关：

```
#define USESOFTSPI 1 //定义是否使用软件模拟SPI口    0:硬件SPI    1: 软件SPI
```

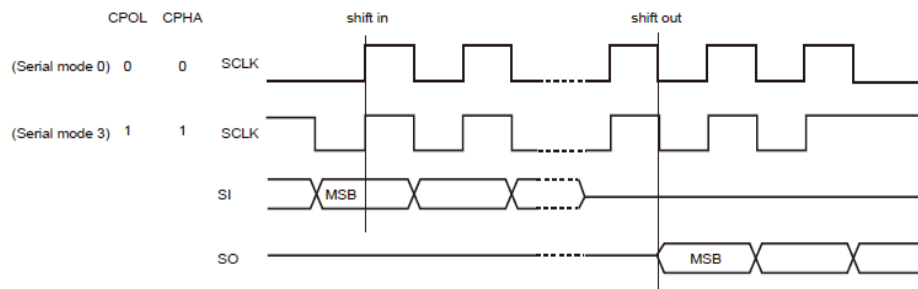


图 2-4- 24 flash 芯片时钟时序图

1、SPI 初始化

NOTE: 定义 SPI 的 CS、SCK、MISO、MOSI 引脚应处于同一电源域(电源域的定义参照《NB-IoT 模块硬件应用手册 NB86 型》手册)。

SPI 初始化如图 2-4-25.

```
SPI_InitTypeDef SPI_MX25L12835F;

/*****
函数名称：MX25L12835F_spiInit
功 能：SPI初始化
参 数：无
返回值：无
*****/
void MX25L12835F_spiInit(void)
{
    #if USESOFTSPI
        SPI_MX25L12835F.DataSize=8;
        SPI_MX25L12835F.Mode=0;
        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) == LierdaSPI_RET_OK)
            lierdaLog("SPI Init OK");
        else
            lierdaLog("SPI Init ERROR");
    #else

```

图 2-4- 25 SPI 初始化代码

2、SPI 写一个字节测试：测试代码如图 2-4-26.

```

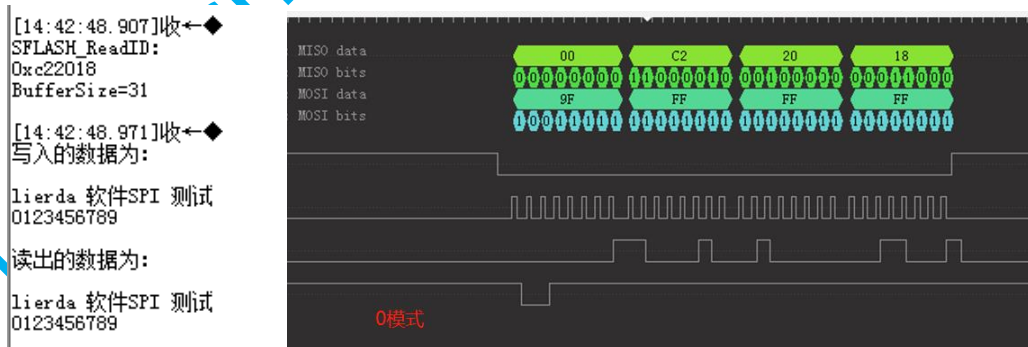
0
7- /*****
8 函数名称：SFLASH_ReadID
9 功 能：读取芯片ID SFLASH的ID
0 参 数：无
1 返回值：ID --- 32位ID号
2 *****/
3 uint32 SFLASH_ReadID(void)
4 {
5     uint32 Temp = 0;
6     uint8 idbuff[4]={0};uint8 cmd_JedecDeviceID=W25X_JedecDeviceID;
7     #if USESOFTSPI
8         SPI_CS_ENABLE(); //使能器件
9
10        lierdaSPIWriteByte(&SPI_MX25L12835F,cmd_JedecDeviceID); //发送一个字节数据《设备ID》指令
11
12        idbuff[0]= lierdaSPIReadByte(&SPI_MX25L12835F); //读一个字节数据
13
14        idbuff[1]= lierdaSPIReadByte(&SPI_MX25L12835F); //读一个字节数据
15
16        idbuff[2]= lierdaSPIReadByte(&SPI_MX25L12835F); //读一个字节数据
17        SPI_CS_DISABLE(); //失能器件
18    #else
19        lierdaSPIRecvData(0,&cmd_JedecDeviceID,1,idbuff, 3,NULL,true); //false true
20    #endif
21    Temp = (idbuff[0] << 16) | (idbuff[1] << 8) | idbuff[2];
22    return Temp;
23 }

```

图 2-4- 26 SPI 写一个字节测试代码

- 3、SPI 读一个字节，测试代码如 2-4-26，直接调用读一个字节函数即可，这里只贴出来读取 FLASH 的设备 ID 代码，具体的读写数据函数见 Demo 中 SPI_MX25L12835F.c 和 SPI_MX25L12835F.h。

- 4、测试结果：



左边为串口调试助手通过 AT 串口打印的测试结果，右边图为逻辑分析仪抓的读取 FLASH 设备 ID 波形（时钟模式：0 模式）。

2.4.9.2 硬件 SPI

概述：这里操作 flash 芯片来做 SPI 通信测试，flash 芯片型号：MX25L12835FM2I。

该芯片的时钟模式为 0 或者 3。如图 2-4-24。使用软件 SPI 测试前需打开软件 SPI 测试开关：

```
#define USESOFTSPI 0 //定义是否使用软件模拟SPI口 0:硬件SPI 1: 软件SPI
```

1、SPI 初始化

NOTE: 定义 SPI 的 CS、SCK、MISO、MOSI 引脚应处于同一电源域(电源域的定义参照《NB-IoT 模块硬件应用手册 NB86 型》手册)。

SPI 初始化如图 2-4-27。

```
#else
    SPI_CONFIGURATION lierdaSPIconfig;
    SPI_PIN lierdaSPIpin;
    lierdaSPIconfig.data_size = 8;
    lierdaSPIconfig.clk_mode = SPI_CLK_MODE3;
    lierdaSPIconfig.clk_div = 0x02;
    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");
#endif
}
```

图 2-4-27 硬件 SPI 初始化

2、SPI 写数据测试：测试代码如图 2-4-28。

```
249
250 /* 禁用串行FLASH: CS 高电平 */
251 SPI_CS_DISABLE();
252 #else
253 if(NumByteToWrite > SPI_FLASH_PerWritePageSize)
254 {
255     NumByteToWrite = SPI_FLASH_PerWritePageSize;
256 }
257 lierdaSPISendData(0, writaddr, 4, pBuffer, NumByteToWrite, NULL);
258 #endif
259 /* 等待写入完毕*/
260 SPI_FLASH_WaitForWriteEnd();
```

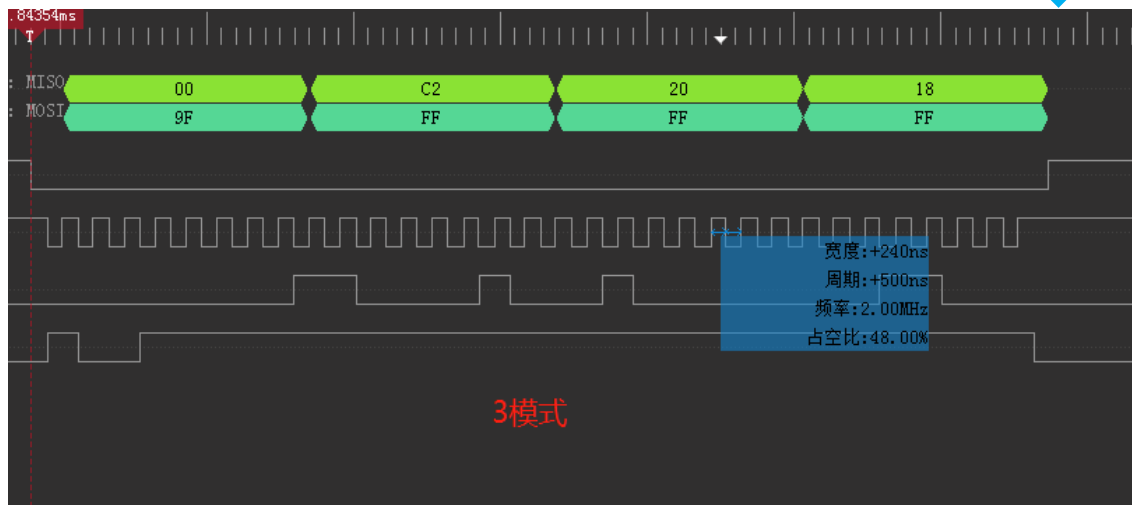
图 2-4-28 硬件 SPI 写数据

3、SPI 读数据测试：测试代码如图 2-4-29。

```
82 SPI_CS_DISABLE();  
83 #else  
84 lierdaSPIRecvData(0,READADDE,4,pBuffer, NumByteToRead,NULL,true);  
85 #endif  
86 }  
87  
88  
89
```

图 2-4-29 硬件 SPI 读数据测试

4、测试结果



```
[14:52:59.011]收←◆  
SFLASH_ReadID:  
0xc22018  
BufferSize=19  
写入的数据为:  
lierda 软件SPI 测试  
读出的数据为:  
lierda 软件SPI 测试  
[14:52:59.181]收←◆  
光照:140 (Lux)  
Temper:24, Humidity:44%  
v$zb·0 v$zb·255 7$zb·15272
```

2.4.10 DNS 解析接口函数

1、接口函数：void lierdaDNSResolve(char *uHostname,char *uIPAddr);

参数：uHostname：需要解析的域名，例如"www.baidu.com"

ulPAddr:解析的结果，以 IP 形式输出，例如"111.13.100.91"

NOTE: DNS 解析超时间为 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直到 DNS 解析完成，最大阻塞时间 200s。

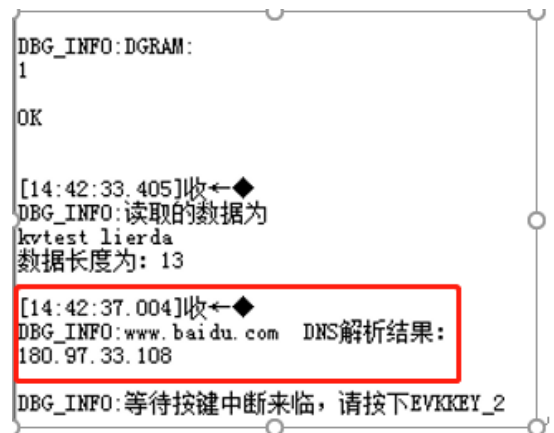
2、测试代码如图 2-4-30。

```
lierdaATDemotest(); //AT指令测试
Lierda_Led_App(); //GPIO测试函数（LED闪烁）
Lierda_KVTest(); //KV测试
lierdaDNSResolve("www.baidu.com",DNS_buff); //DNS测试
lierdaLog("DBG_INFO:www.baidu.com DNS解析结果: \r\n%s",DNS_buff);
lierdaLog("DBG_INFO:等待按键中断来临, 请按下EVKKEY_2"); //通过AT指令串口打印Log测试
for (;;)

```

图 2-4- 30 DNS 解析测试代码

2、测试结果如图 2-4-31。



The screenshot shows a serial terminal window with the following text:

```
DBG_INFO:DGRAM:
1
OK

[14:42:33.405]收←◆
DBG_INFO:读取的数据为
kvtest lierda
数据长度为: 13

[14:42:37.004]收←◆
DBG_INFO:www.baidu.com DNS解析结果:
180.97.33.108

DBG_INFO:等待按键中断来临, 请按下EVKKEY_2
```

图 2-4- 31 DNS 解析测试结果

2.5 lierda_NNMI_task 任务

lierda_NNMI_task 任务用于 NNMI 下行数据接收测试。

NOTE: 进入任务函数后加一个延时（100ms-1s 秒），等待模组上电后初始化完成。否则可能造成程序异常复位如图 2-5-1。


```

76 /*****
77 * @函数名    NNMI任务线程，用于处理NNMI下行数据
78 * @参数 param : 空参数，无效
79 * @返回值 无
80 *****/
81 void lierda_NNMI_task(void *param)
82 {
83     uint8 i;
84     UNUSED(param);
85     osDelay(500); //加延时等待模组初始化完成
86     lierdaNNMIDataInit(); //NNMI下行数据接收初始化
87     for (;;)
88     {
89         lierdaNNMIDataReceived(test_buff, &nnmi_buff_len, 0xFFFFFFFF); //NNMI下行数据接收
90         if (nnmi_buff_len > 0)
91         {
92             lierdaLog("下行数据 :");
93             for(i=0;i<nnmi_buff_len;i++)
94             {
95                 lierdaLog("%x", test_buff[i]);
96             }
97         }
98     }
99 }

```

图 2-5-1 lierda_NNMI_task 任务

2.5.1 NNMI 下行数据测试

NOTE:使用前需使用 AT 指令" AT+NNMI=1" 设置数据下行通知。

1、NNMI 初始化，使用 NNMI 下行数据回调函数时首先要进行初始化操作如图 2-5-2。

```

1 /*****
2 * @函数名    NNMI任务线程，用于处理NNMI下行数据
3 * @参数 param : 空参数，无效
4 * @返回值 无
5 *****/
6 void lierda_NNMI_task(void *param)
7 {
8     uint8 i;
9     UNUSED(param);
10    osDelay(500); //加延时等待模组初始化完成
11    lierdaNNMIDataInit(); //NNMI下行数据接收初始化
12    for (;;)
13    {
14        lierdaNNMIDataReceived(test_buff, &nnmi_buff_len, 0xFFFFFFFF); //NNMI下行数据接收
15        if (nnmi_buff_len > 0)
16        {
17            lierdaLog("下行数据 :");
18            for(i=0;i<nnmi_buff_len;i++)
19            {
20                lierdaLog("%x", test_buff[i]);
21            }
22        }
23    }
24 }

```

图 2-5-2 NNMI 初始化

2、NNMI 接收测试，接收调用：void lierdaNNMIDataReceived(uint8* nnmi_buff,uint16* nnmi_buff_len,uint32 timeout)函数，调用接收函数会引起任务的阻塞，所以用户要对等待时间做合理的安排，测试代码如 2-5-3，程序收到下行数据后会通过 AT 串口打印收到的数据，测试结果如 2-5-4。

```
80 *****/
81 void lierda_NNMI_task(void *param)
82 {
83     uint8 i;
84     UNUSED(param);
85     osDelay(500); //加延时等待模组初始化完成
86     lierdaNNMIDataInit(); //NNMI下行数据接收初始化
87     for (;;)
88     {
89         lierdaNNMIDataReceived(test_buff, &nnmi_buff_len, 0xFFFFFFFF); //NNMI下行数据接收
90         if (nnmi_buff_len > 0)
91         {
92             lierdaLog("下行数据 :");
93             for(i=0;i<nnmi_buff_len;i++)
94             {
95                 lierdaLog("%x", test_buff[i]);
96             }
97         }
98     }
99 }
```

图 2-5- 3 NNMI 下行接收函数



图 2-5- 4 NNMI 下行数据测试结果

2.6 lierda_USRT_task 任务

lierda_USRT_task 任务用于串口数据的接收测试。

NOTE: 进入任务函数后加一个延时（100ms-1s 秒），等待模组上电后初始化完成。否则可能造成程序不正常运行如图 2-6-1。

```
5 *****/
6 * @函数名 串口接收任务线程，用于处理串口数据接收
7 * @参数 param : 空参数，无效
8 * @返回值 无
9 *****/
10 void lierda_USRT_task(void *param)
11 {
12     UNUSED(param);
13     osDelay(500); //加延时等待模组初始化完成
14     for (;;)
15     {
16         lierdaUARTReceive(&lierdaUARTHandle, test_buff, &usrt_buff_len, 0xFFFFFFFF); //串口数据接收
17         if (usrt_buff_len > 0)
18         {
19             lierdaLog("串口数据 :%s", test_buff);
20             memset(test_buff, 0, usrt_buff_len);
21             usrt_buff_len = 0;
22         }
23         osDelay(20);
24     }
25 }
```

图 2-6- 1 lierda_USRT_task 任务

2.6.1 串口数据收发说明

- 1、串口的初始化，这里映射了第三路串口，使用时首先要进行初始化操作如图 2-6-2。

```
34 }
35 /**
36  * @函数名  串口初始化函数
37  * @参数    无
38  * @返回值  无
39  */
40 void usrt_init(void)
41 {
42     lierdaUARTHandle.baudrate = 9600;
43     lierdaUARTHandle.data_bits = UART_DATA_BITS_8;
44     lierdaUARTHandle.parity = UART_PARITY_NONE;
45     lierdaUARTHandle.stopbits = UART_STOP_BITS_1;
46     lierdaUARTHandle.rx_pin = PIN_24;
47     lierdaUARTHandle.tx_pin = PIN_26;
48     lierdaUARTInit(&lierdaUARTHandle);
49 }
50 }
```

图 2-6-2 串口初始化

- 2、串口发送,调用 void lierdaUARTSend(const uint8 *buffer, uint32 length), 测试代码如 2-6-3,

测试结果如 2-6-4

```
159     GPS_flag=0;
160     lierda_GPS_task_disable();//结束GPS线程
161     lierdaLog("结束GPS线程");
162 }
163 GPS_cont--;
164 #else //不使用传感器库
165     lierdaUARTSend(&UARTHandle, (uint8 *) "这是一个串口测试\r\n", strlen((char *) "这是一个串口测试\r\n")); //串口发送测试
166     LIS3DHUpdateInfo(&LIS3DH_X, &LIS3DH_Y, &LIS3DH_Z); //三轴数据获取
167 #endif
168     lierdaLog("X轴:%d,Y轴:%d,Z轴:%d", LIS3DH_X, LIS3DH_Y, LIS3DH_Z); //三轴数据打印
169 }
170 }
171 /**
```

图 2-6-3 串口发送测试代码

```
11:16:04.228]收←◆这是一个串口测试
11:16:25.805]收←◆这是一个串口测试
```

图 2-6-4 串口发送测试结果

- 3、串口接收,调用 uint16 lierdaUARTReceive(uint8 *UserDataPtr, uint16 *UserDataLen), 调用串口接收函数会引起任务的阻塞,所以用户要对等待时间做合理的安排,串口函数接收如图 2-6-5,收到串口数据后通过 AT 串口打印出来,测试结果如图 2-6-6。

```

void lierda_USRT_task(void *param)
{
    UNUSED(param);
    osDelay(500); //加延时等待模组初始化完成
    for (;;)
    {
        lierdaUARTReceive(&lierdaUARTHandle, test_buff, &usrt_buff_len, 0xFFFFFFFF); //串口数据接收
        if (usrt_buff_len > 0)
        {
            lierdaLog("串口数据 :%s", test_buff);
            memset(test_buff, 0, usrt_buff_len);
            usrt_buff_len = 0;
        }
        osDelay(20);
    }
}

```

图 2-6-5 串口接收函数

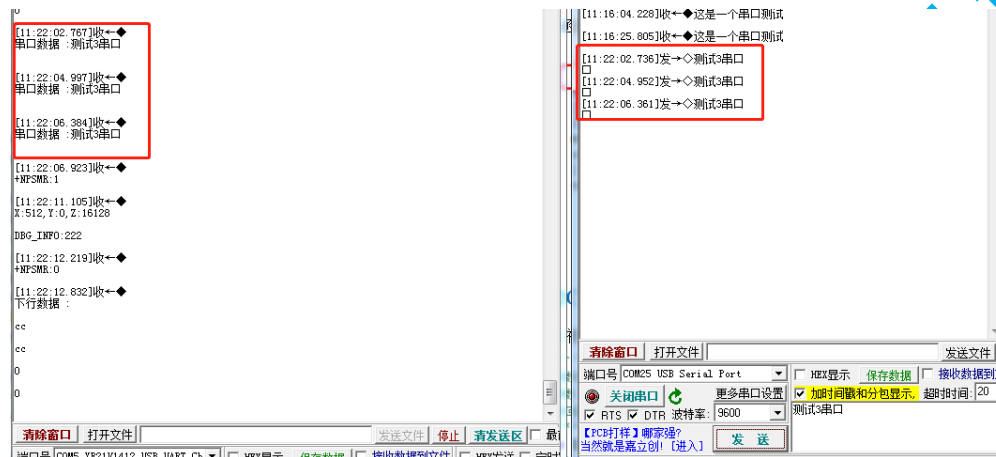


图 2-6-6 串口接收测试结果

2.7 传感器库

Demo 可以在 app.h 里打开或者关闭使用传感器库，1：使用传感器库 0：不使用传感器库。如图 2-7-1。

```

app.c  app.h  Lierda_usrt.c  lierda_gps.h
20 #include "Lierda_usrt.h"
21 #include "Lierda_ATcmd.h"
22 #include "Lierda_spi.h"
23 #include "Lierda_ADC_DAC_Test.h"
24 #include "lierda_HDC1000.h"
25 #include "lierda_OPT3001DN.h"
26 #include "lierda_gps.h"
27
28 #define USE_LIERDA_SENSOR 1 //定义是否使用传感器库
29 #define KV_KEY 38401 //KV键值
30

```

图 2-7-1 使能传感器库

NOTE:使用 I2C 传感器库前需使用 lierda_sensor_I2C_init(uint8 SCL_pin, uint8 SDA_pin)函数进行 I2C 初始化。不可用 lierdaI2CInit(I2C_HandleTypeDef *hi2c)此函数进行初始化。

2.7.1 HDC1000 温湿度传感器

1、初始化

调用 `lierda_HDC1000_Init(void)` 函数进行初始化。

2、数据获取

调用 `lierda_HDC1000_UpdateInfo(int16 *Temper,int16 *Humidity)` 函数进行数据获取。

2.7.2 OPT3001 光照传感器

1、初始化

调用 `lierda_OPT3001_Init(void)` 函数进行初始化。

2、数据获取

调用 `lierda_OPT3001_UpdateInfo(uint32 *Lux)` 函数进行数据获取。

2.7.3 LIS3DH 三轴传感器

1、初始化

调用 `lierda_LIS3DH_Init(void)` 函数进行初始化。

2、数据获取

调用 `lierda_LIS3DH_UpdateInfo(int16 *LIS3DH_X,int16 *LIS3DH_Y,int16 *LIS3DH_Z)` 函数进行数据获取。

2.7.4 华大北斗 GPS

NOTE: 华大北斗 GPS 库的 `lierda_user_GPS_uart_init()` 初始化的是第三路串口，此时第三路串口被 GPS 所占用，客户需要权衡使用。

1、GPS 串口初始化

调用 `lierda_user_GPS_uart_init(uint32 uBaudRate, uint8 uRX_pin, uint8 uTX_pin)` 进行第三路串口初始化。

2、GPS 线程使能

调用 `lierda_GPS_task_enable(void)` 进行使能 GPS 线程。此函数实际上是建立了一个任务用来处理 GPS 数据。

3、获取 GPS 数据

调用 `lierda_GPS_data_get(GPSRMCMStruct *p_gps_user)` 函数进行获取 GPS 数据，客户需

建立一个 GPSRMCSStruct 结构体。

4、终止 GPS 线程

调用 `lierda_GPS_task_disable(void)` 函数进行停止 GPS 数据解析，此函数实际是调用任务删除 API 删除 GPS 线程。

传感器库测试代码如图 2-7-2，测试结果如图 2-7-3。

```
/* *****  
 * @函数名    I2C传感器&华大GPS测试  
 * @参数      无  
 * @返回值    无  
 ***** */  
static void Lierda_I2CSensorTest(void)  
{  
    uint16 LIS3DH_X=0,LIS3DH_Y=0,LIS3DH_Z=0;  
    #if USE_LIERDA_SENSOR  
    uint16 Temper=0, Humidity=0;  
    uint32 Lux=0;  
    #endif  
    #if USE_LIERDA_SENSOR //如果定义使用传感器库  
    lierda_HDC1000_UpdateInfo(&Temper,&Humidity); //温湿度数据获取  
    lierda_LIS3DH_UpdateInfo(&LIS3DH_X,&LIS3DH_Y,&LIS3DH_Z); //三轴数据获取  
    lierda_OPT3001_UpdateInfo(&Lux); //光照数据获取  
    lierda_GPS_data_get(&HD_gpsStruct); //GPS数据获取  
    lierdaLog("光照: %d (Lux)", Lux/100); //光照数据打印  
    lierdaLog("Temper: %d, Humidity: %d%%", Temper/100, Humidity); //温湿度数据打印  
    if ((HD_gpsStruct.Valid_status == 'A') && (GPS_flag==1))  
        lierdaLog("EW: %c, NS: %c, Latitude: %s, Longitude: %s", HD_gpsStruct.EW_indicator, HD_gpsStruct.NS_indicator, HD_gpsStruct.Latitude, HD_gpsStruct.Longitude); //GPS数据打印  
    else  
    {  
        if (GPS_flag==1)  
            lierdaLog("GPS未定位成功");  
    }  
    if ((GPS_cont==0) && (GPS_flag==1))  
    {  
        GPS_flag=0;  
        lierda_GPS_task_disable(); //结束GPS线程  
        lierdaLog("结束GPS线程");  
    }  
    GPS_cont--;  
#else //不使用传感器库  
    LIS3DH_UpdateInfo(&LIS3DH_X, &LIS3DH_Y, &LIS3DH_Z); //三轴数据获取  
#endif  
    lierdaLog("X轴: %d, Y轴: %d, Z轴: %d", LIS3DH_X, LIS3DH_Y, LIS3DH_Z); //三轴数据打印  
}
```

图 2-7-2 传感器库测试代码

```
[16:38:07.202]收←◆  
光照: 68 (Lux)  
  
Temper: 26, Humidity: 56%  
  
EW: E, NS: N, Latitude: 3016.62375, Longitude: 11959.25124  
X轴: 1024, Y轴: 65280, Z轴: 15872  
  
DBG_INFO: Vbat=774  
  
DBG_INFO: Vbat=3316, temp_t=33  
  
DBG_INFO: AIO_V=270  
  
DBG_INFO: AIO电压输出成功  
  
[16:38:08.707]收←◆  
+NFSMR: 0  
|
```

图 2-7-3 传感器库测试结果

2.8 lierda_UDP_TCP_task 任务

概述：Demo 程序定义了一个宏（`#define UDP_TEST 0`）用于打开测试 UDP 下行数据和 TCP

数据下行。定义 `#define UDP_TEST 1` 时用于 UDP 下行数据测试；定义 `#define UDP_TEST 0` 时用于 TCP 下行数据测试。默认打开 UDP 测试。

2.8.1 UDP 下行数据接收测试

- 1) 测试平台: <http://nbiot.iot-ism.com>
- 2) 测试步骤:
 - 1、启动 UDP 测试的宏: `#define UDP_TEST 1`
 - 2、创建 UDP socket
 - 3、发送数据到 UDP 平台（注意使用移动卡）
 - 4、平台下发数据至设备。
- 3) 测试代码如图 2-8-1.

```
33 #if UDP_TEST
34   at_cgpadr_ret = lierdaATCall("AT+NSOCR=DGRAM,17,6555,1", 3000); //创建UDP socket
35   lierdaLog("DBG_INFO:UDP socket:%s",at_cgpadr_ret);
36   socket_num=at_cgpadr_ret[2];
37   if(strstr(at_cgpadr_ret,"OK"))
38   {
39       memcpy(tcp_UDPCmd_buff, UDP_DATASEND, 9);
40       tcp_UDPCmd_buff[9] = socket_num;
41       memcpy(tcp_UDPCmd_buff + 10, ",54.222.172.6,7500,18,0F3836353335323033303436383132371151", 58);
42       at_cgpadr_ret = lierdaATCall(tcp_UDPCmd_buff, 3000); //UDP 发送数据
43       lierdaLog("DBG_INFO:cmd:%s result:%s", tcp_UDPCmd_buff, at_cgpadr_ret);
44   }
45 #else
46   at_cgpadr_ret = lierdaATCall("AT+NSOCR=STREAM,6,31001,1", 3000); //创建TCP socket
47   lierdaLog("DBG_INFO:TCP socket:%s",at_cgpadr_ret);
48   socket_num=at_cgpadr_ret[2];
49   if(strstr(at_cgpadr_ret,"OK"))
50   {
51       memcpy(tcp_UDPCmd_buff, TCP_CONNECT, 9);
52       tcp_UDPCmd_buff[9]=socket_num;
53       memcpy(tcp_UDPCmd_buff + 10, ",54.222.172.6,31001", 19);
54       at_cgpadr_ret = lierdaATCall(tcp_UDPCmd_buff, 3000); //连接TCP服务器
55       lierdaLog("DBG_INFO:cmd:%s result:%s",tcp_UDPCmd_buff,at_cgpadr_ret);
56       if (strstr(at_cgpadr_ret, "OK"))
57       {
58           memcpy(tcp_UDPCmd_buff, TCP_DATASEND, 9);
59           tcp_UDPCmd_buff[9]=socket_num;
60           memcpy(tcp_UDPCmd_buff + 10, ",5,1212121212", 13);
61           at_cgpadr_ret = lierdaATCall(tcp_UDPCmd_buff,3000); //发送数据
62           lierdaLog("DBG_INFO:cmd:%s result:%s",tcp_UDPCmd_buff,at_cgpadr_ret);
63       }
64   }
65 #endif
66 }
```

图 2-8-1 TCP/UDP 测试代码

- 4) 测试结果如图 2-8-2.

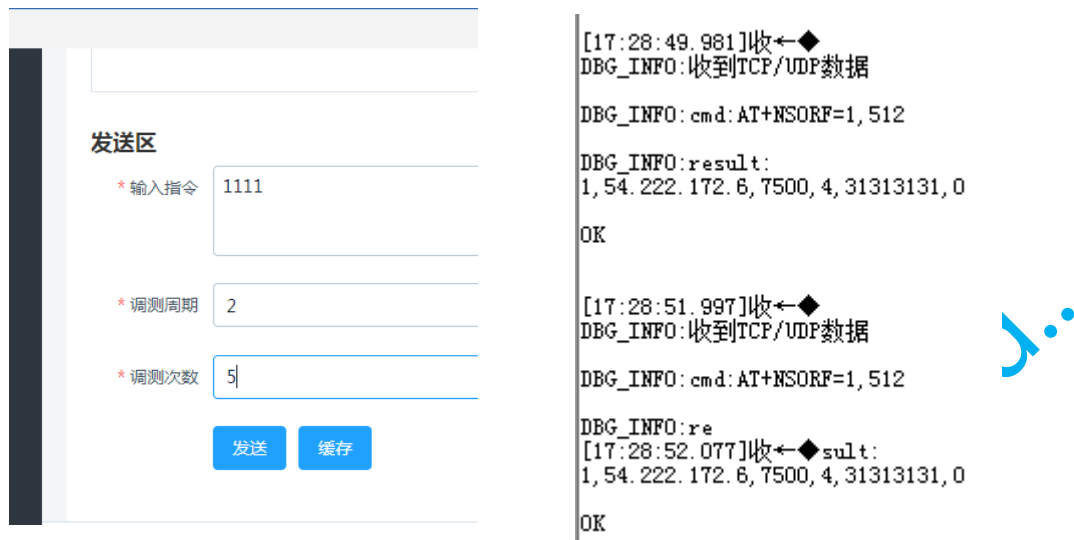


图 2-8- 2 UDP 测试结果

2.8.2 TCP 下行数据接收测试

1) 测试步骤

- 1、启动 TCP 测试的宏：`#define UDP_TEST 0`
- 2、创建 TCP socket
- 3、连接 TCP 服务器
- 3、发送数据到 TCP 服务器
- 4、平台下发数据至设备（步骤 3 给 TCP 服务器发什么，服务器就会给设备回什么）。

2) 测试代码如图 2-8-1.

3) 测试结果如图 2-8-3.


```

DBG_INFO:TCP socket:
1
OK

AT+MLWEVTIND=0

DBG_INFO:cmd:AT+NSOCO=1,54.222.172.6,31001 result:
OK
AT+MLWEVTIND=3

DBG_INFO:cmd:AT+NSOSD=1,5,1212121212,31001 result:
1,5
OK

[18:38:22.984]收←◆
DBG_INFO:读取的数据为
kvtest lierda
数据长度为: 13

[18:38:23.607]收←◆
DBG_INFO:收到TCP/UDP数据

DBG_INFO:cmd:AT+NSORF=1,512

DBG_INFO:result:
1,54.222.172.6,31001,5,1212121212,0
OK

```

图 2-8- 3 TCP 测试结果

2.9 事件状态通知

2.9.1 相关接口函数介绍

1、事件状态更新接口函数

```
void lierda_module_status_read(void)
```

此函数用于事件状态的更新，获取某一事件前需调用此函数进行更新事件状态。

2、事件状态结构体

```
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;
```

3、实例说明

查询是否附着上网络：示例代码如下：

```
do
{
    if(count!=20)
        osDelay(1000);
    lierda_module_status_read(); //读取是否附着网络
    lierdaLog("%s", module_status.charCGATT);
    count--;
}

while((strstr(module_status.charCGATT, "+CGATT:1")==NULL)&&(count>
0));
```

测试结果如图 2-9-1.

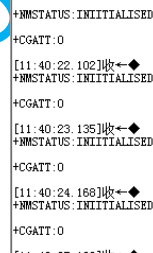


图 2-9-1 附着网络事件状态测试

2.10 FOTA 相关 API

终端在进行 FOTA 升级时不能做发数据业务，此时客户需要做规避，故提供 FOTA 状态查询 API 和 FOTA 环境下是否能进行发数据动作。

2.10.1 FOTA 状态查询 API

1、函数原型

```
lierda_fotaSta lierda_FotaStatus(void);
```

* @参数 param：空参数

```

* @返回值 FOTA 状态，枚举变量

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;

```

2、测试代码

```

switch(lierda_FotaStatus())//获取FOTA状态
{
    case FOTA_NON://此时不处于FOTA状态
        lierdaLog("DBG_INFO:无FOTA事件，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)
        break;
    case FOTA_UPDATE_OVER://FOTA
        lierdaLog("DBG_INFO: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)
        break;
    default:lierdaLog("DBG_INFO:FOTA ING.....");break;
}

```

2.10.2 判断 FOTA 过程中能否做业务

此 API 为了方便开发者在有 FOTA 需求时，规避 FOTA 中不能发数据的 API 接口，建议在发数据前调用此 API 判断是否可以发业务。

1、函数原型

lierdaFota lierda_FotaEnableData(void);

参数 param：空参数

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

2、测试代码

```
lierda_module_status_read();//更新此时事件状态
if (strstr(module_status.charNMSTATUS,"MO_DATA_ENABLED")!=NULL)//判断
LWM2M是否注册成功
{
    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");
    }
}
else
    lierdaLog("DBG_INFO:LWM2M 服务器未注册");
```

2.11 LWM2M 数据发送接口

1、函数接口：CLOUD_RET lierdaSendMsgToPlatform(uint8 *data, uint16 data_len, MSG_MODE mode, uint8 seq_num,uint8 isHEX);

此函数用于向 IoT 平台发送 CON 或 NON 数据。详细见《Lierda NB-IoT 模组 OpenCPU API 使用文档》

2、测试代码

```
static void Lwm2m_sendData(void)
{
    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);
```

技术支持：nbiot_support@lierda.com

44 / 57

```
//发送数据
```

```
}
```

3 工程编译

3.1 编译相关配置

单机要编译的工程，快捷键 Alt+Enter 进入配置界面（或者鼠标右击要编译的工程 -> properties ->进入配置界面），点击 SCons，选项框填写：target=ref-2115 如图 3-1。

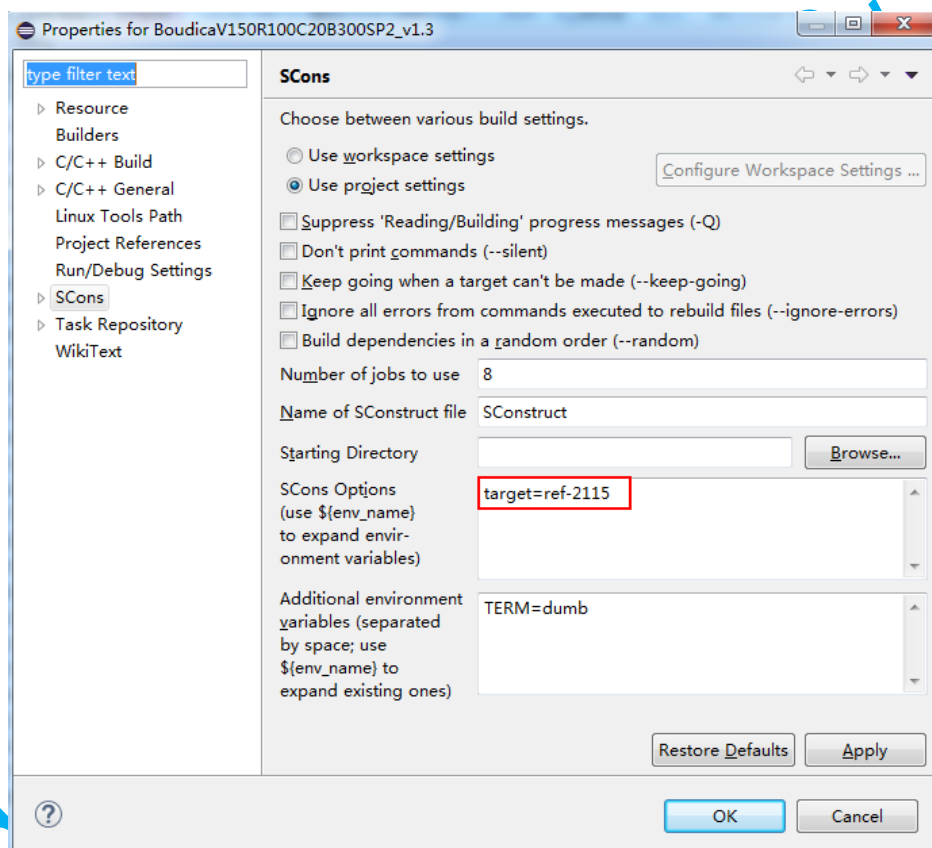


图 3-1 编译配置界面

3.2 编译步骤

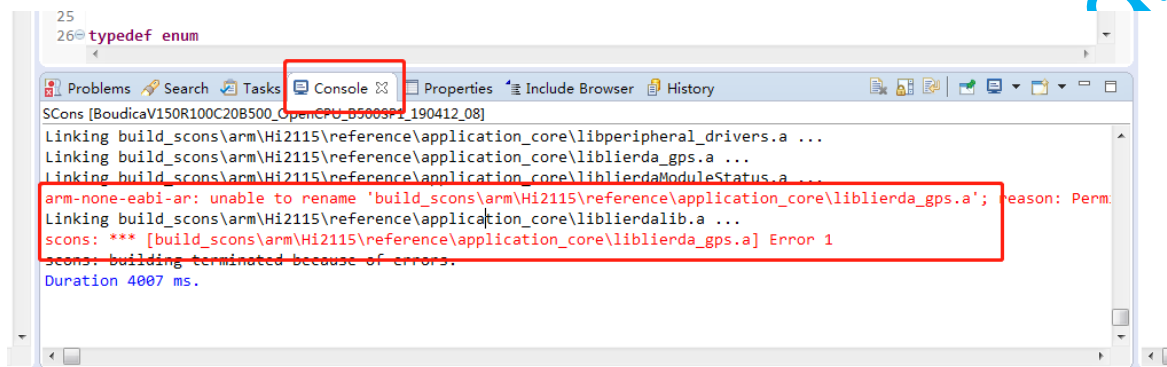
选中工程鼠标右击要编译的工程 --> Build Project

NOTE: 编译之前要对写好的工程进行保存，否则编译生成的 application.bin 文件没被更新。

3.3 常见编译出错解决办法

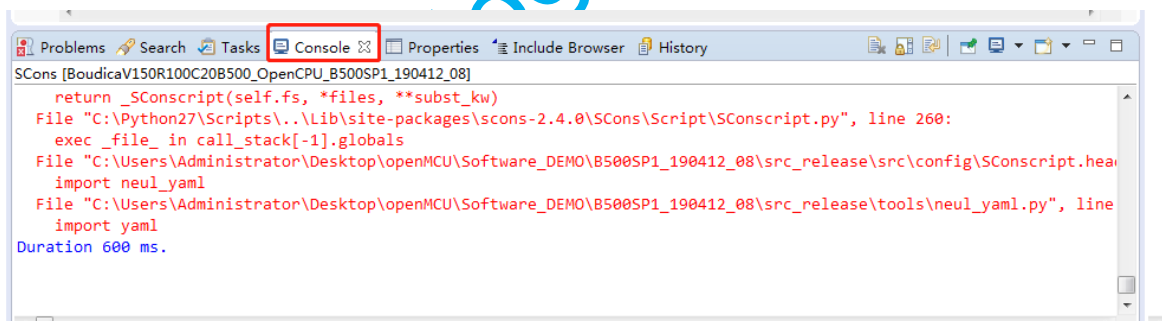
3.3.1 编译后报找不到 **xx.a** 库

若编译出错，如下图类似错误，请先 Clean Project 工程一下，然后在进行编译。若还是报错建议重复此操作多试几次。Clean Project 工程方法：选中工程鼠标右击要 Clean 的工程 --> Clean Project.



3.3.2 编译后报.py 导入出错

pyyaml 库导入失败，如下图，解决办法：尝试重新安装 pyyaml 解决问题，安装方法见《OpenCPU 开发环境搭建指南》

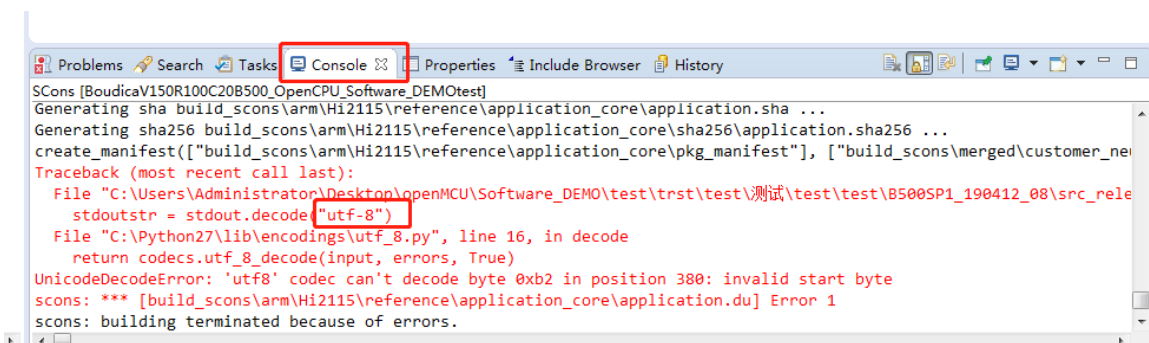


3.3.3 环境变量未添加导致编译出错

若出现编译报找不到 gcc 编译器或者 python 找不到，请检查环境变量是否添加，确保环境变量添加成功。

3.3.4 编译后报编码问题

编译后报编码出错如下图，解决办法：工程路径中坚决不能出现中文路径，存放工程的路径不能太深，建议 3 层以内。

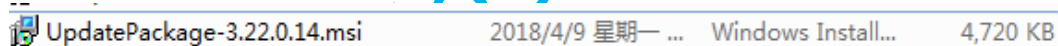


4 生成Package固件包

由生成 Package 固件包的命令比较复杂，需要复制粘贴各种文件路径，故提供生成 Package 的批处理文件，以方便客户生成 Package 固件包，具体的操作方法详见 4.3 小节。以前生成 Package 的命令行方式保留，客户可根据 SDK 版本参考相应的小节进行 Package 生成。

4.1 openCPU_B300SP2 版本

4.1.1 UpdatePackage 软件安装



点击安装 UpdatePackage 固件生成软件。

NOTE: 需要安装与工程所对应的 UpdatePackage 版本。

4.1.2 Package 包生成

1、在电脑程序里找到 UpdatePackage.exe，点击进入命令行界面如图 4-1

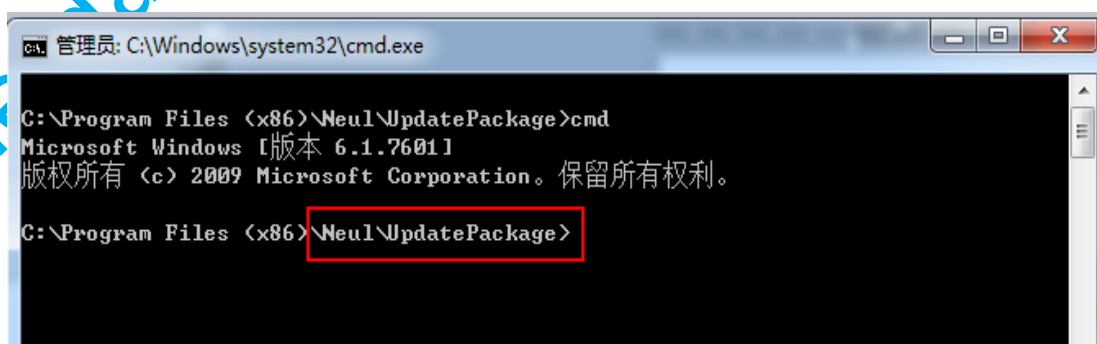


图 4- 1 Package 生成软件

输入生成 Package 包的命令：

技术支持：nbiot_support@lierda.com

```
updatepackage.exe updateApplication --in C:\xx\XXXX.fwpkg --folder C:\zz\application_core  
--out C:\yy\YYYY.fwpkg
```

NOTE:

(1) C:\xx:放 XXXX.fwpkg 文件的路径如图 4-2, XXXX.fwpkg 为 Lierda 发布的原始烧录版本, 这里的 XXXX.fwpkg 版本要和工程文件相匹配;

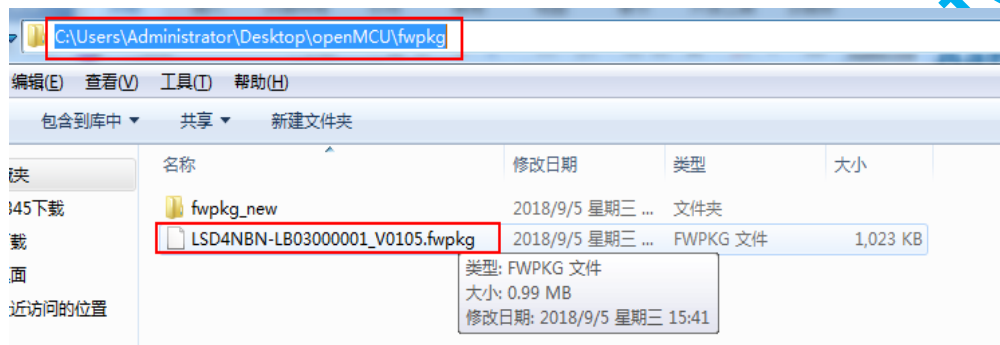


图 4-2 Lierda 发布的原始烧录版本位置

(2) C:\zz\application_core:eclipse 编译工程后生成 application.bin 的文件路径如图 4-3

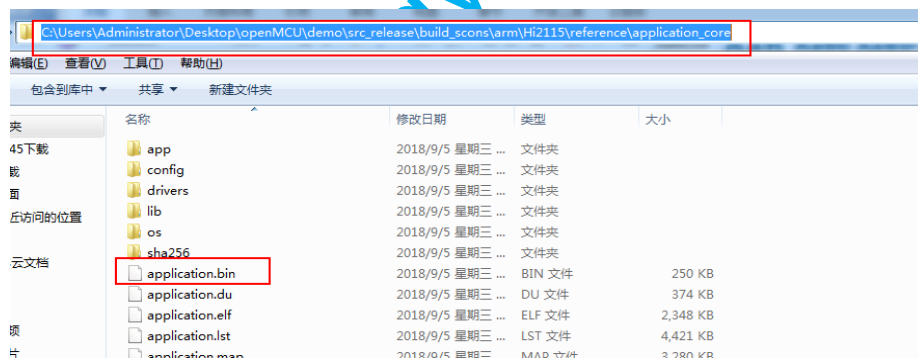


图 4-3 编译工程生成 application.bin 文件位置

(3) C:\yy: 更新后存放新的 Package 包文件路径, YYYY.fwpkg 为更新 application.bin 和 application.sha256 后的烧录版本。如图 4-4

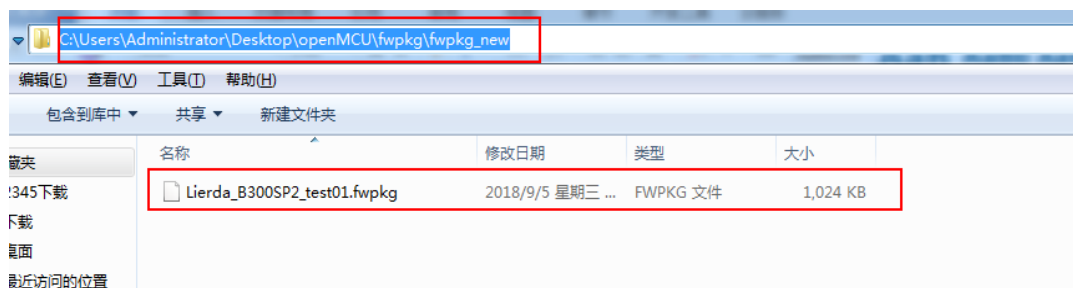


图 4-4 更新后生成固件位置

```

(4) Eg:UpdatePackage.exe                updateApplication                --in
C:\Users\Administrator\Desktop\openMCU\fwpkg\LSD4NBN-LB03000001_V0105.fwpkg  --folder
C:\Users\Administrator\Desktop\openMCU\demo\src_release\build_scons\arm\Hi2115\reference\application_core
                                                                                   --out
C:\Users\Administrator\Desktop\openMCU\fwpkg\fwpkg_new\Lierda_B300SP2_test01.fwpkg

```

如图 4-5

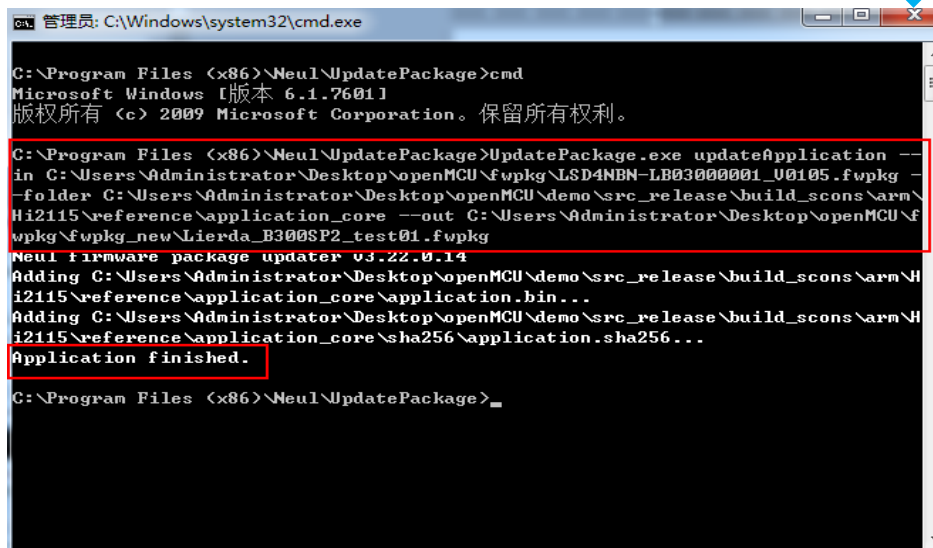


图 4-5 固件包生成

出现 Application finished 则说明新的固件包已经生成。

NOTE:客户可以参照生成固件记事本里格式，更改相应的文件路径和文件名，如图 4-6。

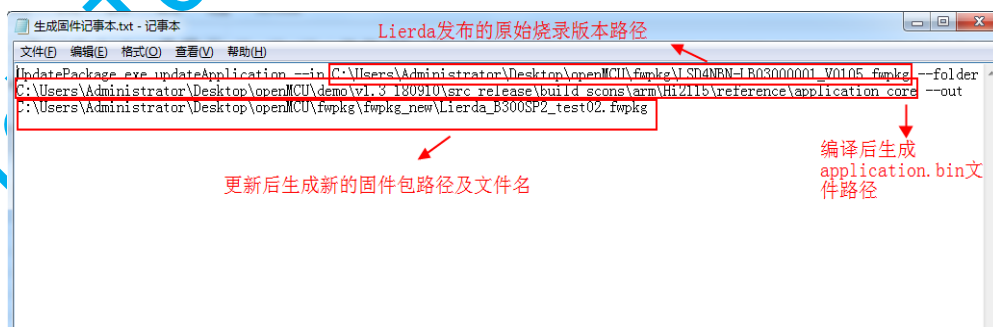
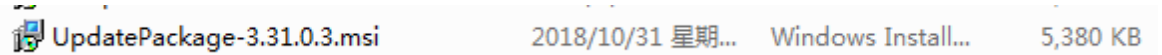


图 4-6 固件生成命名格式

4.2 openCPU_B300SP5&B500SP1 版本

4.2.1 UpdatePackage 软件安装



点击安装 UpdatePackage 固件生成软件。

NOTE: 需要安装与工程所对应的 UpdatePackage 版本。

4.2.2 Package 包生成

1、在电脑程序里找到 UpdatePackage.exe，点击进入命令行界面如图 4-7。

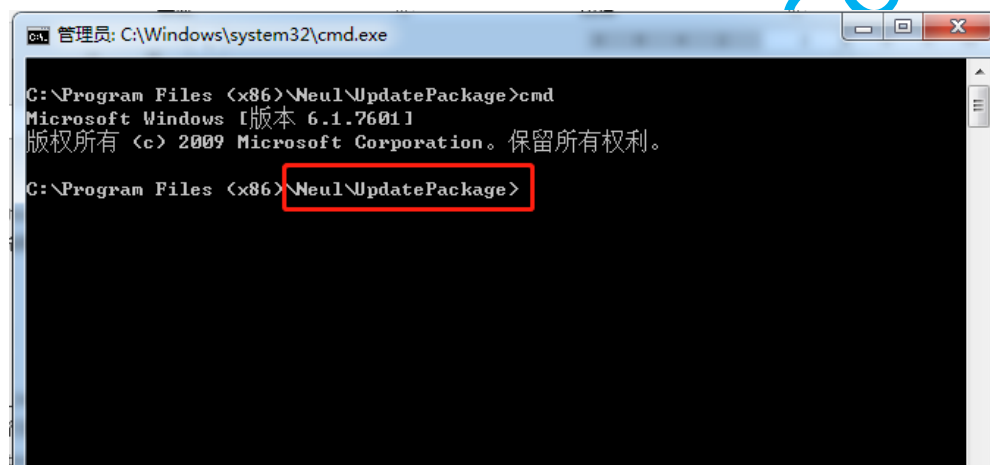


图 4- 7UpdatePackage.exe 界面

输入生成 Package 包的命令：

```
UpdatePackage.exe UpdateApplicationA -f -m C:\xx\manifest.txt --in C:\yy\old.fwpkg -o  
C:\yy\new.fwpkg
```

Note:

(1)、C:\xx: manifest.txt 文件存放位置，如图 4-8 所示。



图 4- 8 manifest.txt 文件存放路径

manifest.txt: 文件用于配置更新 A 核映像，客户在使用过程中不必修改里面的配置，在使用过程中不能删除此文件。

(2)、C:\yy: old.fwpkg 文件的路径如图 4-9。

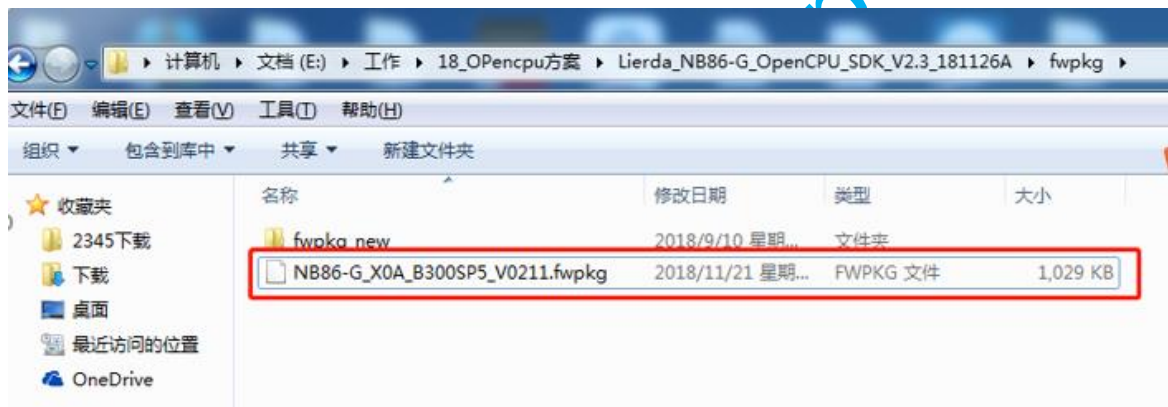


图 4- 9 old.fwpkg 文件存放路径

old.fwpkg 为 Lierda 发布的原始烧录版本，这里的 old.fwpkg 版本要和工程文件相匹配；

(3)、C:\yy: 更新后存放新的 Package 包文件路径，new.fwpkg 为更新 application.bin 和 application.sha256 后的烧录版本。 如图 4-10。

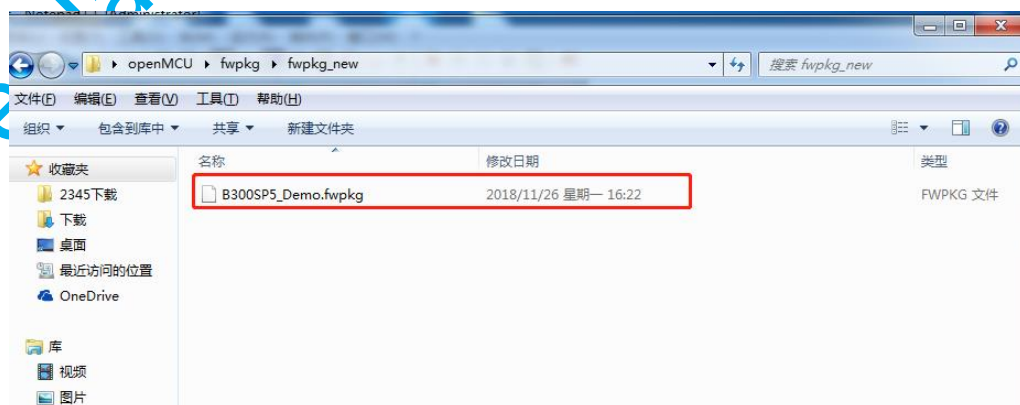


图 4- 10 Package 包文件路径

(5) eg:

```
(6) UpdatePackage.exe UpdateApplicationA -f -m
C:\Users\Administrator\Desktop\openMCU\B300SP5_openCPU\src_release\build_scons\arm\Hi2115
\reference\application_core\manifest.txt --in
C:\Users\Administrator\Desktop\openMCU\fwpkg\NB86-G_X0A_B300SP5_V0211.fwpgk -o
C:\Users\Administrator\Desktop\openMCU\fwpkg\fwpkg_new\B300SP5_Demo.fwpgk
```

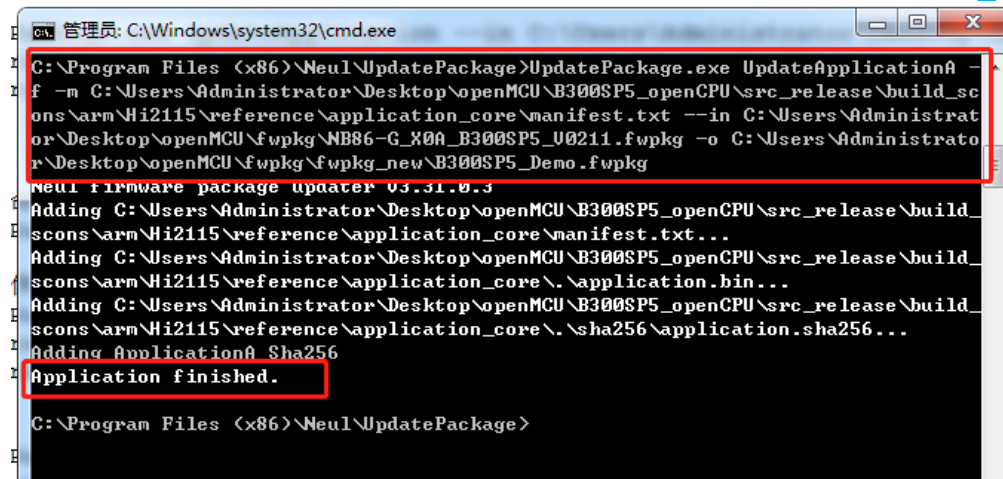


图 4-11 生成固件包界面

出现 Application finished 则说明新的固件包已经生成如图 4-11。

NOTE:客户可以参照生成固件记事本里格式，更改相应的文件路径和文件名，如图 4-12。

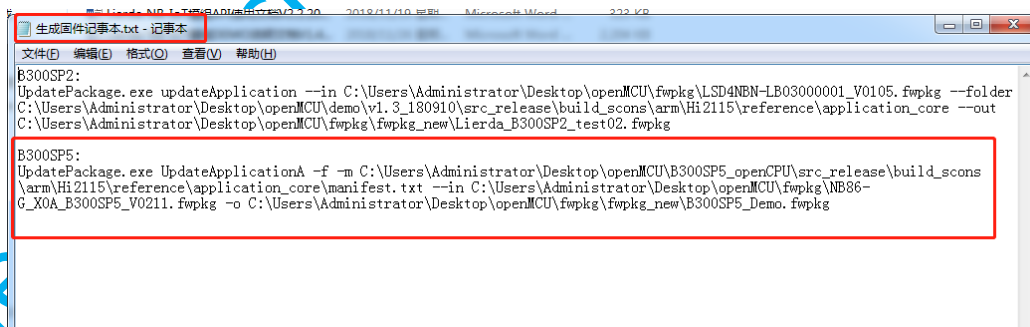


图 4-12 生成固件记事本样例

4.3 openCPU Package 批处理文件

4.3.1 UpdatePackage 软件安装

点击安装 UpdatePackage 固件生成软件。

 UpdatePackage-3.31.0.3.msi	2018/10/31 星期...	Windows Install...	5,380 KB
--	------------------	--------------------	----------

NOTE: 需要安装与工程所对应的 UpdatePackage 版本

4.3.2 Package 包生成

NOTE: 由于批处理文件中涉及相对路径获取，客户在拿到 OpenCPU SDK 交付件后，不要更改文件目录格式，否则可能导致生成固件出错。

- 1、打开 OpenCPU 交付的资料，进入 fwpkg 文件，找到 make_fwpgk.bat 文件如图 4-13

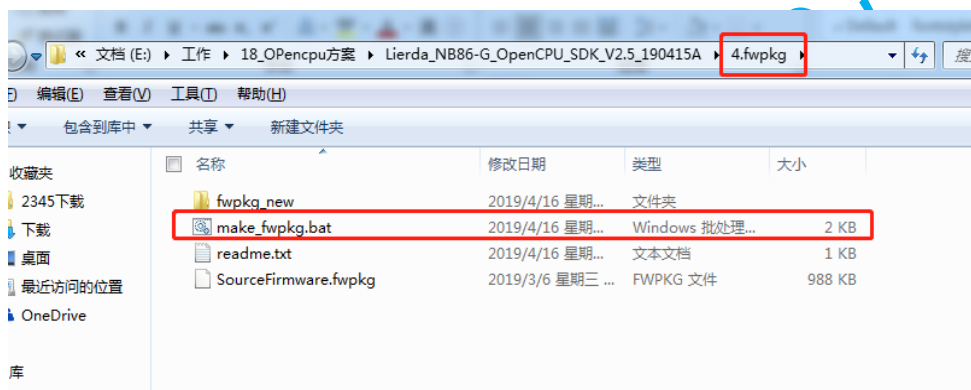


图 4-13 make_fwpgk.bat 文件

- 2、点击运行该批处理文件，选择要合成的工程对应的序号后按回车即可开始生成固件包。如图 4-14。

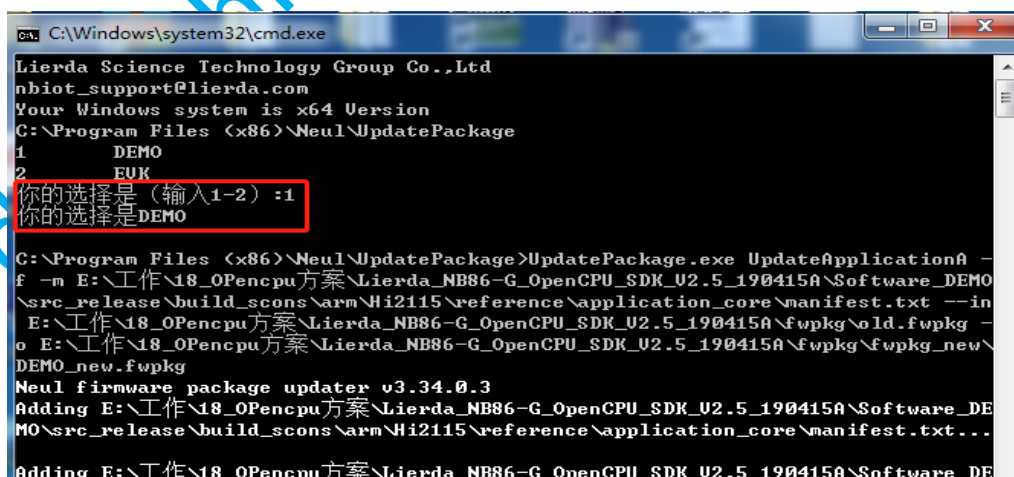


图 4-14 固件包生成

- 3、make_fwpgk.bat 批处理文件运行完成后，在 fwpkg_new 文件夹下就会生成相应的固件包如图 4-15

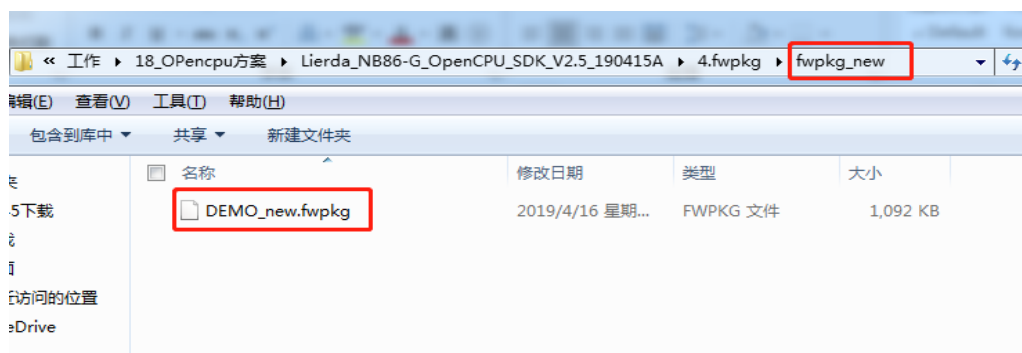


图 4-15 新固件目录

5 固件烧录

5.1 UEUpdaterUI 软件安装



点击安装 UEUpdaterUI 软件。

NOTE: 需要安装与工程所对应的 UEUpdaterUI 版本。

5.2 固件烧录

NOTE: 烧写 openCPU 固件和烧写 Lierda 发布的标准固件方法不太一样，具体方法如下：

- 1、烧写与模组型号和开发所用的 SDK 相对应的 Lierda 标准固件，UEUpdaterUI 软件中勾选 write KV 如图 5-1.

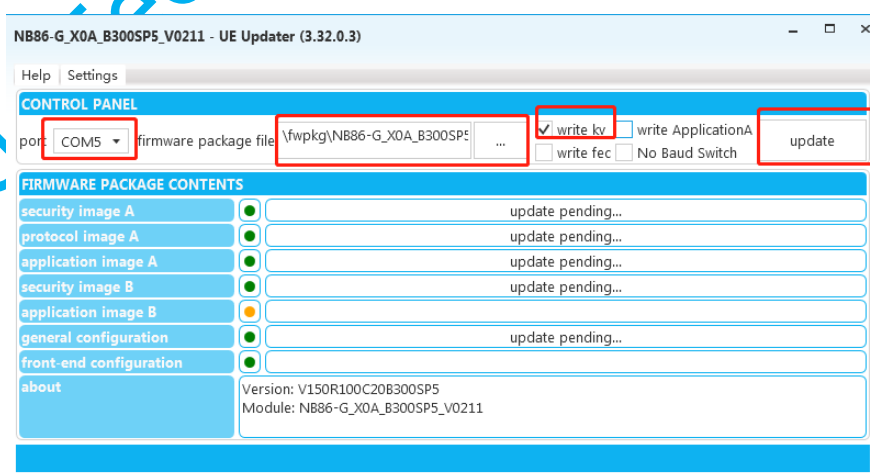


图 5-1 烧写标准固件

- 2、烧写 openCPU 生成的固件，UEUpdaterUI 软件中只用勾选 write ApplicationA 如图 5-2.

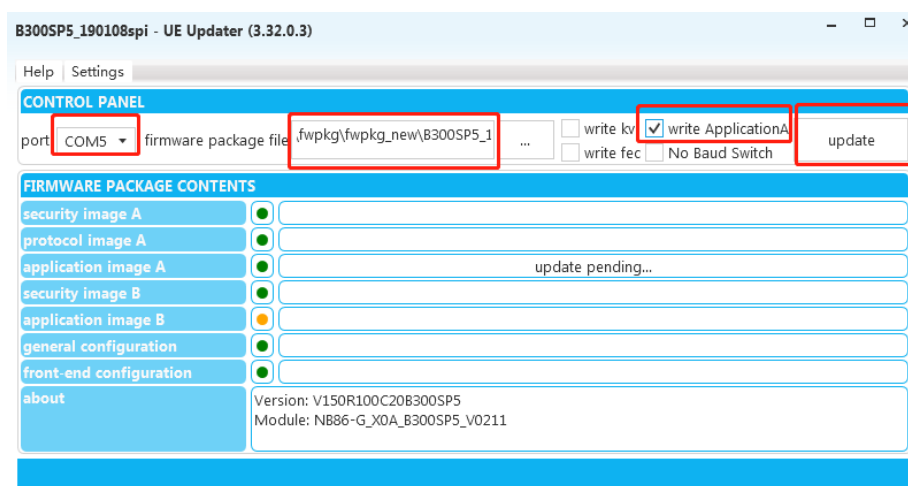


图 5-2 烧写 openCPU 固件

例如：模组型号：NB86-G(X0A) 开发所用 SDK：B300SP5

- 1、首先烧写 Lierda 发布的标准 NB86-G(X0A)固件：NB86-G_X0A_B300SP5_V0312.fwpkg（只用烧写一次即可）
- 2、烧写 openCPU 开发生成的固件包。

NOTE:

- 烧写生成的 openCPU 固件前必须烧写一次与模组和开发所用 SDK 相对应的标准固件。
- 烧写标准固件必须勾选 write KV。
- 烧写 openCPU 固件只用勾选 write ApplicationA。

6 程序调试

由于目前无法提供在线仿真调试，建议客户在设计及调试应用时，使用 AT 串口或芯片 Log 串口打印应用程序日志，用于诊断程序运行状态。

NOTE: 此 LOG 打印函数不能在中断回调中使用。

6.1 通过 AT 指令串口打印 Log

AT 指令串口打印 Log 函数：lierdaLog(const char *pcFormat, ...);用法同 printf();

头文件 #include “lierda_app_main.h”

测试代码如图 6-1.

```
Lierda_Led_App(); //GPIO测试函数 (LED闪烁)
LIS3DHUpdateInfo(&temp1, &temp2, &temp3); //三轴数据获取
lierdaLog("X:%d,Y:%d,Z:%d", temp1, temp2, temp3); //三轴数据打印
lierdaATDemoCall("AT+NMGS=10,A9A1A2A3A4A5A6A7A8A9", "OK", 3000, 5); //CoAP发送数据
lierdaUARTSend(&lierdaUARTHandle, (uint8 *) "这是一个串口测试\r\n", strlen((char *) "这是一个串口测试\r\n")); //串口发送测试
lierdaLogview("DBG_INFO:%d", 111); //通过UE Log串口打印Log测试
lierdaLog("DBG_INFO:%d", 222); //通过AT指令串口打印Log测试
osDelay(20000);
```

图 6-1 AT 指令串口打印 Log

测试结果如图 6-2.

```
[10:24:51.021]收←◆
+NPSMR:1

[10:25:29.304]收←◆
X:256,Y:65280,Z:16640

DBG_INFO:222

[10:25:30.666]收←◆
+NPSMR:0

[10:25:52.381]收←◆
+NPSMR:1

[10:26:30.856]收←◆
X:256,Y:65280,Z:14080
DBG_INFO:222

[10:26:32.265]收←◆
+NPSMR:0

[10:26:53.901]收←◆
+NPSMR:1
```

图 6-2 测试结果

6.2 通过 UE Log 串口打印 Log

AT 指令串口打印 Log 函数: lierdaLogview (const char *pcFormat, ...);用法同 printf();

头文件: #include "app_at_log.h"

测试代码如图 6-3.

```
Lierda_Led_App(); //GPIO测试函数 (LED闪烁)
LIS3DHUpdateInfo(&temp1, &temp2, &temp3); //三轴数据获取
lierdaLog("X:%d,Y:%d,Z:%d", temp1, temp2, temp3); //三轴数据打印
lierdaATDemoCall("AT+NMGS=10,A9A1A2A3A4A5A6A7A8A9", "OK", 3000, 5); //CoAP发送数据
lierdaUARTSend(&lierdaUARTHandle, (uint8 *) "这是一个串口测试\r\n", strlen((char *) "这是一个串口测试\r\n")); //串口发送测试
lierdaLogview("DBG_INFO:%d", 111); //通过UE Log串口打印Log测试
lierdaLog("DBG_INFO:%d", 222); //通过AT指令串口打印Log测试
osDelay(20000);
```

图 6-3 测试代码

测试结果如图 6-4.

3814	+02:00.975...	APPLICATION_REPORT	"OK"	[170,170,170,170,0,0,2,0] "OK"
3822	+02:00.982...	APPLICATION_REPORT	"DBG_IN..."	[170,170,170,170,0,0,12,0] "DBG_INFO:111"
3823	+02:00.982...	APPLICATION_REPORT	"DBG_IN..."	[170,170,170,170,0,0,12,0] "DBG_INFO:222"
4146	+02:02.286...	APPLICATION_REPORT	"NPSM..."	[170,170,170,170,0,0,8,0] "NPSMR:0"
7084	+02:23.846...	APPLICATION_REPORT	"NPSM..."	[170,170,170,170,0,0,8,0] "NPSMR:1"
7114	+03:02.537...	APPLICATION_REPORT	"X:512, Y:..."	[170,170,170,170,0,0,17,0] "X:512, Y:0, Z:16384"
7115	+03:02.538...	APPLICATION_REPORT	"NMG..."	[170,170,170,170,0,0,5,0] "NMG"
7116	+03:02.540...	APPLICATION_REPORT	"send mi..."	[170,170,170,170,0,0,27,0] "send mid 16223 tk 1c d6 * 2"
7117	+03:02.541...	APPLICATION_REPORT	"udp_sen..."	[170,170,170,170,0,0,16,0] "udp_send ret=25!"
7118	+03:02.541...	APPLICATION_REPORT	"OK"	[170,170,170,170,0,0,2,0] "OK"
7126	+03:02.548...	APPLICATION_REPORT	"DBG_IN..."	[170,170,170,170,0,0,12,0] "DBG_INFO:111"
7127	+03:02.548...	APPLICATION_REPORT	"DBG_IN..."	[170,170,170,170,0,0,12,0] "DBG_INFO:222"
10417	+04:04.103...	APPLICATION_REPORT	"X:512, Y:..."	[170,170,170,170,0,0,17,0] "X:512, Y:0, Z:16384"

图 6-4 测试结果

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。

表格 1 相关文档

序号	文档名称	注释
[1]	Lierda NB-IoT模组OpenCPU API 使用文档	
[2]	Lierda NB-IoT模组V150 OpenCPU版本更新说明	
[3]	Lierda NB-IoT模组V150 OpenCPU开发环境搭建指南	
[4]	Lierda NB86-EVK操作使用手册	
[5]	NB-IoT模块硬件应用手册 NB86型	