

MiCO Documentation Working Group (MDWG)

Jenny Liu

Track Number: RM0016CN

MXCHIP Co., Ltd

Category: Reference Manual

2016.4.26

Version: 1.2

Open

MiCO SDK 基础 Demo 功能说明

摘要 (Abstract)

本文主要介绍 MiCO SDK (V3.0.0 版本) 的 Demos 文件夹中提供了哪些单一功能，这些功能是如何实现的？功能实现调用的 API 函数有哪些？API 函数具备什么功能？请详细阅读本文档。本文档会对 Demos 文件夹中例程使用进行说明，旨在为用户在学习使用 Demo 时提供指导。

适合读者 (Suitable Readers)

本文适用于具备一定 C 语言基础和了解嵌入式实时操作系统的 MiCO 开发者（已下载 MiCO SDK 开发包，下载地址：<http://mico.io/> 页面的 WiKi 的下载中心）。

获取更多帮助 (More Help)

登录上海庆科官方网站：<http://mxchip.com/>，获取公司最新产品信息。

登录 MiCO 开发者论坛：<http://mico.io/>，获取更多 MiCO 最新开发资料。

登录 FogCloud 开发者中心：<http://easylink.io/>，获取更多 FogCloud 云开发文档。

微信“扫一扫”关注：“MiCO 总动员”公众号，获取 MiCO 团队小伙伴最新活动信息。



版权声明 (Copyright Notice)

Copyright (c) 2015 MDWG Trust and the persons identified as the document authors. All rights reserved.

版本记录

日期	修改人	版本	更新内容
2015-9-16	Jenny Liu	V1.0	初始版本
2016-4-28	Jenny Liu	V1.1	修订版本，增加支持 v2.4.1，包括加密算法和文件系统
2016-9-28	Jenny Liu	V1.2	支持 MiCO SDK V3.0.0 版本

目录

MiCO SDK 基础 Demo 功能说明	1
版本记录	1
1. 概述	9
2. 第一个 MiCO 应用程序	11
2.1 HELLO WORLD 功能说明	11
3. RTOS 实时操作系统	12
3.1 OS_THREAD	12
3.1.1 Thread 线程概述	12
3.1.2 os_thread.c 功能说明	12
3.2 OS_TIMER	14
3.2.1 Timer 定时器概述	14
3.2.2 os_timer.c 功能说明	14
3.3 OS_MUTEX	15
3.3.1 Mutex 互斥锁概述	15
3.3.2 os_mutex.c 功能说明	15
3.4 OS_SEM	18
3.4.1 Sem 信号量概述	18
3.4.2 os_sem.c 功能说明	19
3.5 OS_QUEUE	21
3.5.1 Queue 消息队列概述	21
3.5.2 os_queue.c 功能说明	21
4. 系统电源管理	24
4.1 MiCO 电源管理概述	24
4.2 POWER_MEASURE.C 功能说明	24
5. Wi-Fi 功能	32
5.1 WIFI_SCAN	32
5.1.1 wifi_scan.c 功能说明	32
5.2 WIFI_SOFTAP	33
5.2.1 SoftAP 概述	33
5.2.2 wifi_softap 功能说明	33
5.3 WIFI_STATION_CORE_API	34
5.3.1 Station 模式概述	34
5.3.2 wifi_station_core_api.c 功能说明	34
5.4 WIFI_STATION_SYSTEM_API.C	36
5.4.1 wifi_station_system_api 功能说明	36
6. 网络通信	38
6.1 DNS	38
6.1.1 dns 概述	38
6.1.2 dns.c 功能描述	38

6.2	MDNS	40
6.2.1	mDNS 概述.....	40
6.2.2	mDNS.c 功能描述.....	40
6.3	GAGENT 连接	43
6.4	SNTP_CLINET 网络时间获取.....	43
6.4.1	Sntp 概述.....	43
6.4.2	sntp_client.c 功能描述.....	44
6.5	UDP_UNICAST	44
6.5.1	UDP 单播概述.....	44
6.5.2	udp_unicast.c 功能说明.....	45
6.6	UDP_BROADCAST.C.....	48
6.6.1	UDP 广播概述.....	48
6.6.2	udp_broadcast 功能说明.....	48
6.7	TCP_CLIENT	51
6.7.1	TCP 客户端概述.....	51
6.7.2	tcp_client.c 功能说明	52
6.8	TCP_SEVER	56
6.8.1	tcp_sever.c 功能说明	56
7.	HTTP 通信	61
7.1	HTTP_CLIENT	61
7.1.1	HTTP 概述.....	61
7.1.2	http_client.c 功能说明	62
7.2	HTTP_SEVER.C	70
7.2.1	http_sever.c 功能说明	70
8.	JSON 解析	77
8.1	JSON 概述	77
8.1.1	什么是 JSON	77
8.1.2	JSON 语言规则	77
8.1.3	认识 JSON	77
8.2	JSON_OP.C 功能说明	78
8.2.1	JSON 对象的组装与解析.....	78
9.	WiFi 串口透传.....	81
9.1	概述	81
9.2	基本思路	81
9.3	DEMO 运行过程.....	82
9.3.1	Easylink 配网	82
9.3.2	设备参数设置	83
9.3.3	数据透传	83
10.	校验算法	85
10.1	CRC 校验	85
10.1.1	概述	85

10.1.2	crc8_mico_check_test.c 功能	85
10.1.3	crc16_mico_check_test.c 功能	85
10.1.4	CRC API 接口定义	85
10.1.5	CRC 使用注意事项	86
10.2	MD5.....	86
10.2.1	md5_test.c 源码	87
10.2.2	md5_test.c 运行结果	89
10.2.3	md5_test.c API 接口定义	89
10.3	SHA	90
10.3.1	sha1_test.c 源码	90
10.3.2	sha1_test.c 运行结果	93
10.3.3	sha1_test.c API 接口定义	93
11.	加密算法	94
11.1	加密算法概述	94
11.2	AES 加密	94
11.2.1	aes_cbc_test.c : cbc 模式的 AES 加密	94
11.2.2	aes_ecb_test.c : ecb 模式的 AES 加密	95
11.2.3	aes_with_pkcs5_padding_test.c: cbc 模式的 PKCS#5 填充 AES 加密	95
11.3	ARC4 加密	95
11.3.1	arc4_test.c 的功能	95
11.4	DES 加密	95
11.4.1	des_test.c: 64 位密钥的 DES 加密	95
11.4.2	13.5.1 des3_test.c: 192 位密钥的 3DES 加密	96
11.5	HMAC 加密	96
11.5.1	hmac_md5_test.c: HAMC MD5 加密	96
11.5.2	hmac_sha1_test.c: HAMC SHA1 加密	96
11.5.3	hmac_sha256_test.c: HAMC SHA256 加密	96
11.5.4	hmac_sha384_test.c: HAMC SHA384 加密	97
11.5.5	hmac_sha512_test.c: HAMC SHA512 加密	97
11.6	RABBIT 加密	97
11.7	RSA 加密	97
12.	文件系统 fatfs.....	98
12.1	文件系统概述	98
12.2	FATFS.C 功能说明	98
12.3	FATFS.C 文档参考	101
13.	Hardware 驱动.....	102
13.1	MiCOKIT 单一外设驱动	102
13.1.1	ext_dc_motor.c: 直流电机	102
13.1.2	ext_rgb_led.c: RGB 灯	102
13.1.3	ext_temp_hum_sensor.c: 温湿度传感器	104
13.1.4	ext_oled.c: OLDE 显示屏	105

13.1.5 ext_light_sensor.c: 光强传感器.....	106
13.1.6 ext_infrared_reflective.c: 外设之红外反射传感器	107
13.1.7 ext_motion_sensor.c: 外设之运动传感器.....	108
13.1.8 ext_environmental_sensor.c: 环境传感器.....	109
13.1.9 ext_ambient_light_sensor.c:近距离环境光三合一传感器.....	110
13.2 STMEMS 开发板外设控制	111
13.2.1 micokit_Stmems_demo.c 功能说明.....	111
13.2.2 micokit_Stmems_demo API 说明	111
14. MiCO 低功耗蓝牙功能示例	114
14.1 BLE_HELLO_SENSOR: 低功耗蓝牙服务端应用	114
14.2 BLE_SCAN : 低功耗蓝牙客户端的设备扫描	115
14.3 BLE_HELLO_CENTER: 低功耗蓝牙客户端应用	115
14.4 BT_RFCOMM_SERVER : 虚拟蓝牙串口服务器:	117
14.5 BLE_ADVERTISEMENTS: 低功耗蓝牙客户端广播	118

图目录

图 1.1 Demos 文件夹例程文件	9
图 1.2 mico_config.h 文件路径	10
图 1.3 修改头文件路径	10
图 2.1 Hello World 串口 log.....	11
图 3.1 多线程串口 log	13
图 3.2 定时器串口 log	14
图 3.3 未加互斥锁串口 log	18
图 3.4 加入互斥锁串口 log	18
图 3.5 信号量串口 log	19
图 3.6 消息队列串口 log	21
图 4.1 低功耗串口 log	24
图 5.1 wifi 扫描串口 log.....	32
图 5.2 SoftAP 串口 log	33
图 5.3 Station 模式（指定 wifi）串口 log.....	34
图 5.4 Station 模式（Easylink 配网）串口 log.....	36
图 6.1 dns 域名解析串口 log	38
图 6.2 mDNS 串口 log	40
图 6.3 sntp_client.c 串口 log 信息	44

图 6.4 UDP 传输机制	44
图 6.5 UDP 单播串口 log	45
图 6.6 UDP 单播测试	45
图 6.7 UDP 广播 log	48
图 6.8 UDP 广播测试	49
图 6.9 Socket 机制	51
图 6.10 TCP Client 串口 log	52
图 6.11 TCP Client 测试	53
图 6.12 TCP Server 串口 log	56
图 6.13 tcp server 测试界面	57
图 7.1 HTTP 客户端 GET 方法	62
图 7.2 HTTP 客户端 POST 方法	62
图 7.3 HTTP Client 串口 log	62
图 7.4 http server 串口 log	70
图 7.5 PC 端配网页面	70
图 7.6 网页配网设置成功	70
图 7.7 模块 Station 模式成功，Station up	71
图 8.1 JSON 格式解析串口 log	78
图 9.1 wifi 串口透传示意图	81
图 9.2 启动 Easylink 配网	82
图 9.3 设备参数设置界面	83
图 9.4 wifi 串口透传 log	83
图 9.5 透传 APP 使用过程演示	84
图 10.1 CRC API 定义	86
图 10.2 CRC8 校验参数设置	86
图 10.3 CRC16 校验参数配置	86
图 10.4 MD5 串口 log	89
图 10.5 MD5 API 定义	90
图 10.6 SHA 算法文件目录	90
图 10.7 SHA 相关头文件路径	92
图 10.8 SHA1 串口 log	93
图 10.9 SHA1 API 定义	93

图 11.1 aes cbc 加密串口 log.....	94
图 11.2 aes ecb 加密串口 log.....	95
图 11.3 aes with pkcs5 加密串口 log.....	95
图 11.4 arc4 加密串口 log.....	95
图 11.5 DES 加密串口 log.....	96
图 11.6 DES3 加密串口 log.....	96
图 11.7 hmac_md5 加密串口 log.....	96
图 11.8 hmac_sha1 加密串口 log	96
图 11.9 hmac_sha256 加密串口 log	97
图 11.10 hmac_sha384 加密串口 log.....	97
图 11.11 hmac_sha512 加密串口 log.....	97
图 11.12 Rabbit 加密串口 log.....	97
图 11.13 RSA 加密串口 log.....	97
图 12.1 文件系统添加文件	98
图 12.2 添加工程包含路径	98
图 12.3 文件系统串口 log	99
图 13.1 温湿度检测 log	104
图 13.2 光强检测 log	106
图 13.3 红外反射传感器 log	107
图 14.1 设备名称修改位置	114
图 14.2 ble_hello_sensor 串口 log 信息	114
图 14.3 设备广播数据信息	115
图 14.4 ble_scan 运行 log 信息	115
图 14.5 ble_hello_center 中代码修改.....	116
图 14.6 ble_hello_center 串口 log 信息.....	116
图 14.7 ble_hello_sensor 串口 log 信息	116
图 14.8 PC 端串口确认	117
图 14.9 ble_rfcomm_server 串口 log 信息	118
图 14.10 ble_advertisements 运行 log 信息	119

表目录

表 1.1 Demos 文件夹例程列表	9
表 3.1 RTOS Demo 列表.....	12

Mxchip
reprint prohibited

1. 概述

demos 文件夹中包含了 MiCO 系统各单一功能例程文件，路径如图 1.1：

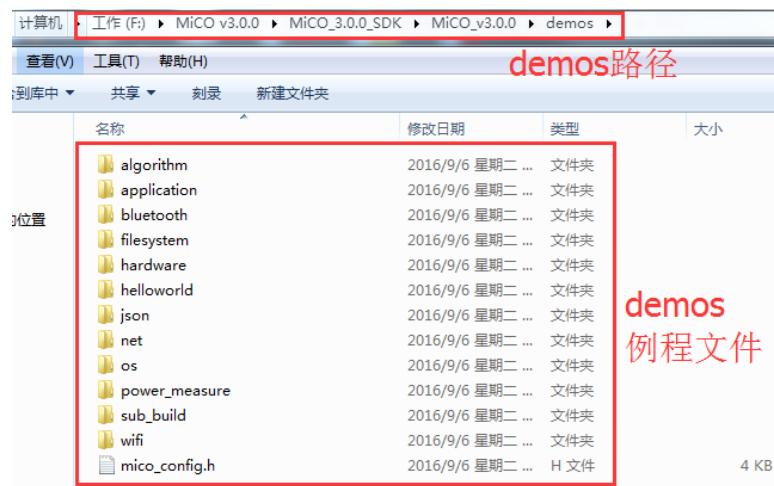


图 1.1 Demos 文件夹例程文件

Demo 由以下几部分组成，如表 1.1：

表 1.1 Demos 文件夹例程列表

序号	文件夹名称	功能描述
1	application	综合应用示例，包括 wifi_uart 串口透传功能，需和手机 APP--Easylink 配合使用
2	algorithm	校验算法包括 CRC8、CRC16、MD5、SHA1, SHA256, SHA384, SHA512。 安全算法包括 AES、ARC4、DES、HMAC、Rabbit、RSA。
3	bluetooth	MiCO 系统 BLE 蓝牙功能示例，包括：低功耗蓝牙服务端应用，低功耗蓝牙设置 Wi-Fi 网络参数，低功耗蓝牙客户端设备扫描，低功耗蓝牙客户端应用，虚拟蓝牙串口服务器，低功耗蓝牙客户端广播示例。
4	Filesystem	MiCO 支持的 FATFS 文件系统例程（目前仅适支持 EMW3165）
5	hardware	MiCOKit 开发板外设单一功能实现（包括电机、红外、环境等传感器）
6	helloworld	第一个 MiCO 应用程序，启动 MiCO 系统，LED 闪烁，串口 log 打印输出 log
7	json	MiCO 支持的 json 格式数据解析方法
8	net	包括：DNS 解析，mDNS 本地设备发现，SNTP 时间获取， TCP 客户端，TCP 服务器端通信服务示例； UDP 广播，UDP 单播通信服务示例， HTTP 客户端 和 HTTP 服务器端应用示例； GAgent 机智云服务接入示例。
9	os	多线程，定时器，互斥锁，信号量，消息队列的基本例程
10	power measure	MiCO 设备开启低功耗模式
11	sub_build	配合 MiCoder 使用，实现在 GCC 环境下完成编译下载功能的部件，禁止删除。

12

wifi

Wi-Fi 功能示例，包括：设置 Station 模式（Easylink 配网）和 SoftAP 模式，Wi-Fi 本地搜索

注意：每个 Demo 的系统配置文件 mico_config.h 分别放在.c 文件所在文件夹，工程路径需重新配置。

以 hello world.c 为例，其 mico_config.h 头文件路径如图 1.2：



图 1.2 mico_config.h 文件路径

头文件路径修改方法 如下：

- (1) IAR 中，右键单击工程 COM.MXCHIP.BASIC，单击“option”；
- (2) option 界面中，按图 1.3 中的 1,2,3 步骤，修改头文件路径。

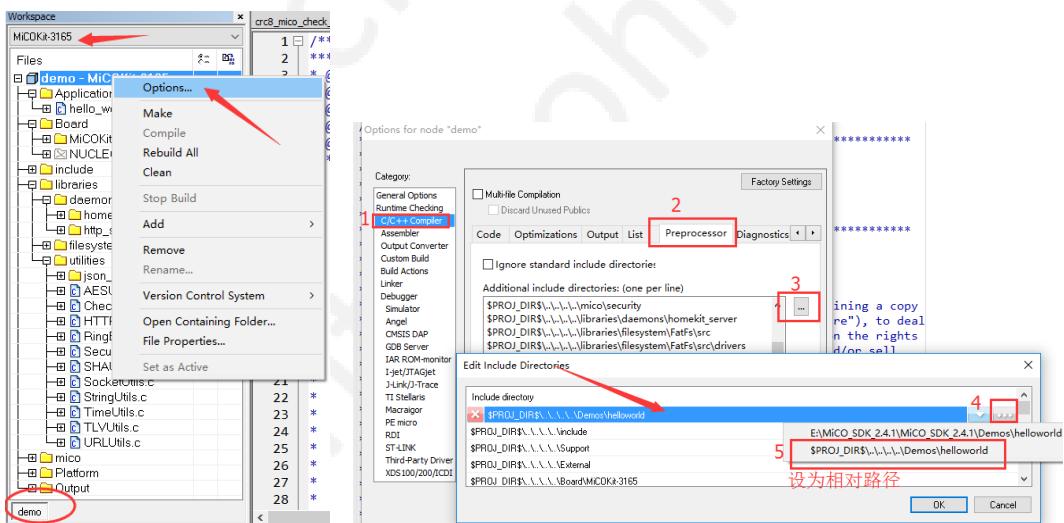


图 1.3 修改头文件路径

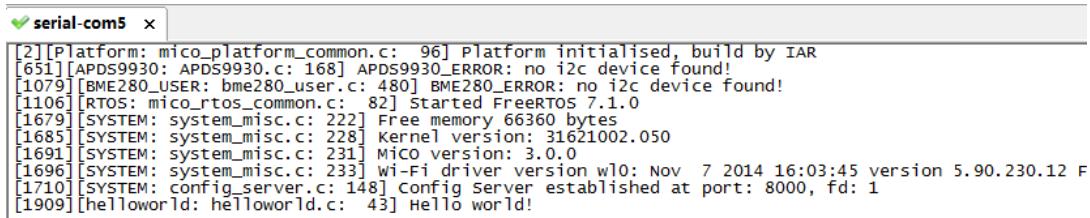
用户注意：

- (1) 如果.c 文件所在文件夹没有配置头文件，请使用 Demos 文件夹中的头文件 mico_config.h，如上图 1 中的 mico_config.h 文件。
 - (2) 下面章节针对上表 13 大组成部分进行讲解，介绍 MiCO 系统如何调用 API 函数，实现某单一功能。
 - (3) IAR 环境中加载和下载固件的具体方法请参考“MiCO 用户手册”中的“固件下载与调试方法”，本文不再详细描述。
 - (4) MiCO 系统串口 log 说明：
- [1034] [SYSTEM: system_misc.c: 234]: MiCO libairy version: 31620002.026, 依次为：
[系统时间] [文件夹名称: 所在.c 文件: 行数]: log 内容

2. 第一个 MiCO 应用程序

2.1 Hello world 功能说明

Demo 运行后，串口 log 输出如图 2.1：



```
[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1079][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1106][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1679][SYSTEM: system_misc.c: 222] Free memory 66360 bytes
[1685][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1691][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1696][SYSTEM: system_misc.c: 233] Wi-Fi driver version w10: Nov 7 2014 16:03:45 version 5.90.230.12 F
[1710][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1909][helloworld: helloworld.c: 43] Hello world!
```

图 2.1 Hello World 串口 log

本例实现：

- (1) 串口 log 输出字符串："Hello world Demo!"。
- (2) 通过当前线程休眠 1s，实现系统 LED 灯间隔 1s 闪烁，LED 闪烁现象请参见：
http://v.youku.com/v_show/id_XMTI1ODUwMDMwMA==.html。其中，log 前 5 行分别输出：
 - (1) 工程编译环境为 IAR。
 - (2) BME280 初始化找不到 I2C 设备。(BME280 暂时未焊，不影响其它使用)
 - (3) 系统可自由支配的存储空间大小 64440 bytes。
 - (4) MiCO 库的版本号。
 - (5) Wi-Fi 驱动的版本号。程序代码如下：

```
#include "mico.h"

#define os_helloworld_log(format, ...) custom_log("helloworld", format, ##__VA_ARGS__)

int application_start( void )
{
    /* 根据 mico_config.h 配置，启动 MiCO 系统功能*/
    mico_system_init( mico_system_context_init( 0 ) );

    /* 调试串口 log 输出 */
    os_helloworld_log( "Hello world Demo!" );

    while(1)
    {
        MicoGpioOutputTrigger( MICO_SYS_LED ); /* 触发 LED 状态翻转 /
        mico_thread_sleep(1); /* 当前线程休眠 1 秒钟*/
    }
}
```

3. RTOS 实时操作系统

本节主要介绍：如何通过调用 MiCO API 来实现 RTOS 实时操作系统基本功能，包括：多线程，定时器，互斥锁，信号量，消息队列。功能说明如表 3.1：

表 3.1 RTOS Demo 列表

序号	文件名	机制	功能描述
1	os_thread.c	多线程	创建两个线程，分别完成各自功能。
2	os_timer.c	定时器	开启 1s 定时器，并实现累加计数
3	os_mutex.c	互斥锁	创建 4 个线程，实现多窗口不重复卖票功能，临界代码段加入互斥锁，保证共
4	os_sem.c	信号量	创建两个线程，一个获取信号量，一个释放信号量，谁拿到信号量谁执行，拿
5	os_queue.c	消息队列	创建两个线程，一个写入数据，一个读出数据，实现线程间通信

3.1 os_thread

3.1.1 Thread 线程概述

通过 MiCO RTOS 线程控制 API 可以在系统中：定义，创建，控制和销毁线程。

线程优先级分为 10 级，从 0-9，级数越低，优先级越高。高优先级的线程可以抢占低优先级线程，如果高优先级线程不能挂起，会导致低优先级的线程无法得到时间去运行。相同优先级的各个线程通过时间片轮转的方式分时运行，使得这些线程看起来是同时运行的。

在 MiCO 系统初始化时，会创建一个以函数 int application_start(void)为主执行体的线程，该线程的优先级是 7 (MICO_APPLICATION_PRIORITY)。

一个线程可以处于以下几种状态：

RUNNING, 运行：线程正在运行中，在同一个时间，MiCO RTOS 中之可能有一个线程处于运行状态。

Ready, 就绪：线程已经就绪并且等待运行。一旦当前的运行线程被终止，或者挂起，所有就绪的线程中优先级最高的线程将会变成运行状态。

Suspend, 挂起：线程正在等待事件（一段时间，信号量，互斥锁，消息队列）发生后，转换成就绪状态。

Terminate, 终止：线程处于非活动状态，在 MiCO 系统的 IDLE 线程中，所有的非活动状态的线程所拥有的私有资源将会被自动销毁。

共享资源（如通过 malloc 创建的内存区块）需要通过人工进行销毁。

3.1.2 os_thread.c 功能说明

串口 log 输出如图 3.1：

```
[0][OS: os_thread.c: 40] This is thread 1
[4][OS: os_thread.c: 48] This is thread 2
[2004][OS: os_thread.c: 40] This is thread 1
[4008][OS: os_thread.c: 40] This is thread 1
[4012][OS: os_thread.c: 48] This is thread 2
[6012][OS: os_thread.c: 40] This is thread 1
[8016][OS: os_thread.c: 40] This is thread 1
[8021][OS: os_thread.c: 48] This is thread 2
[10020][OS: os_thread.c: 40] This is thread 1
[12024][OS: os_thread.c: 40] This is thread 1
[12032][OS: os_thread.c: 48] This is thread 2
[14028][OS: os_thread.c: 40] This is thread 1
[16032][OS: os_thread.c: 40] This is thread 1
[16042][OS: os_thread.c: 48] This is thread 2
[18036][OS: os_thread.c: 40] This is thread 1
[20040][OS: os_thread.c: 40] This is thread 1
[20052][OS: os_thread.c: 48] This is thread 2
[22044][OS: os_thread.c: 40] This is thread 1
[24048][OS: os_thread.c: 40] This is thread 1
[24063][OS: os_thread.c: 48] This is thread 2
```

图 3.1 多线程串口 log

本 Demo 实现：

- (1) 创建两个线程，thread1 和 thread2
- (2) thread1 完成：串口 log 输出“This is thread 1”，然后线程挂起 2s
- (3) thread2 完成：串口 log 输出“This is thread 2”，然后线程挂起 4s，并且删除线程，释放资源。

程序代码如下：

```
int application_start( void )
{
    OSStatus err = kNoErr;
    mico_thread_t t_handler = NULL;

    /* 创建一个新线程 */
    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY, "Thread 1", thread_1, 500, NULL );
    require_noerr_string( err, exit, "ERROR: Unable to start the thread 1." ); /* 出错处理 */

    while(1){

        /* 创建一个新线程，该线程可删除自身并释放资源 */
        err = mico_rtos_create_thread( &t_handler, MICO_APPLICATION_PRIORITY, "Thread 2", thread_2, 500, NULL );
        require_noerr_string( err, exit, "ERROR: Unable to start the thread 2." );
        mico_rtos_thread_join( &t_handler ); /* 当前线程休眠至其它线程终止 */

    }

exit:
    if( err != kNoErr )

        os_thread_log( "Thread exit with err: %d", err );

    /* 删除当前线程 */
    mico_rtos_delete_thread(NULL);

    return err;
}
```

3.2 os_timer

3.2.1 Timer 定时器概述

通过 MiCO RTOS 定时器控制 API，可以在系统中实现：定时器的初始化，启动，重加载计数值，停止，销毁和查询定时器运行状态的功能。

3.2.2 os_timer.c 功能说明

串口 log 输出如图 3.2：

```
[0][OS: os_timer.c: 50] time is coming,value = 0
[1003][OS: os_timer.c: 50] time is coming,value = 1
[2003][OS: os_timer.c: 50] time is coming,value = 2
[3003][OS: os_timer.c: 50] time is coming,value = 3
[4003][OS: os_timer.c: 50] time is coming,value = 4
[5003][OS: os_timer.c: 50] time is coming,value = 5
[6003][OS: os_timer.c: 50] time is coming,value = 6
[7003][OS: os_timer.c: 50] time is coming,value = 7
[8003][OS: os_timer.c: 50] time is coming,value = 8
[9003][OS: os_timer.c: 50] time is coming,value = 9
[10003][OS: os_timer.c: 50] time is coming,value = 9
```

图 3.2 定时器串口 log

本 Demo 实现：

- (1) 启动一个 1000ms 的系统定时器，同时累加计数
- (2) 计数到 9 时，销毁定时器

程序代码如下：

```
#include "mico.h"

#define os_timer_log(M, ...) custom_log("OS", M, ##_VA_ARGS__)

mico_timer_t timer_handle;

/* 用户自定义函数声明 */

void destroy_timer( void );
void alarm( void* arg );

/* 用户自定义函数 destroy_timer() 定义，销毁定时器 */

void destroy_timer( void )
{
    mico_stop_timer( &timer_handle ); /* 停止一个运行中的定时器 */
    mico_deinit_timer( &timer_handle ); /* 删除一个由 mico_init_timer() 初始化的 RTOS 定时器 */
}

/* 计时时间到，调用 destroy_timer() 销毁定时器 */

void alarm( void* arg )
{
    int* count = (int*)arg;
    os_timer_log("time is coming,value = %d", (*count)++ );
```

```
if( *count == 10 )  
    destroy_timer(); /* 销毁定时器 */  
}  
  
int application_start( void )  
{  
    OSStatus err = kNoErr;  
    os_timer_log("timer demo");  
    int arg = 0;  
    /* 初始化一个定时器，定时时间 1000ms */  
    err = mico_init_timer(&timer_handle, 1000, alarm, &arg);  
    require_noerr(err, exit);  
    /* 启动一个定时器 */  
    err = mico_start_timer(&timer_handle);  
    require_noerr(err, exit);  
    /* 当前线程无限期休眠 */  
    mico_thread_sleep( MICO_NEVER_TIMEOUT );  
  
exit:  
    if( err != kNoErr )  
        os_timer_log( "Thread exit with err: %d", err );  
    mico_rtos_delete_thread( NULL );  
    return err;  
}
```

3.3 os_mutex

3.3.1 Mutex 互斥锁概述

通过 MiCO RTOS 互斥锁控制 API 可以在系统中实现：互斥锁的初始化，获取，释放，和销毁功能。

互斥锁是用来保证多线程互斥的，比如，一个线程占用了某一个资源，那么别的线程就无法访问，直到这个线程离开，其他线程才开始可以利用这个资源。用来保证一段时间内只有一个线程在访问同一资源。

3.3.2 os_mutex.c 功能说明

本 Demo 实现：

- (1) 模拟多窗口卖票功能，开启 4 个线程，即 4 个窗口，实现 99-0 数字的递减输出，即出票过程。
- (2) 4 个线程执行同一回调函数，在临界代码段处加互斥锁，实现数字资源某一时刻的独有，避免重复卖票。

程序代码如下：

```
#include "mico.h"

#define os_mutex_log(M, ...) custom_log("OS", M, ##__VA_ARGS__)

// 本例提供两种输出效果对比：一个是无互斥锁，一个是有互斥锁。

// 取消此句注释，打开互斥锁功能；否则，关闭互斥锁功能

//#define USE_MUTEX


#ifndef USE_MUTEX

#define MUTEX_LOCK() mico_rtos_lock_mutex(&mutex)
#define MUTEX_UNLOCK() mico_rtos_unlock_mutex(&mutex)

#else

#define MUTEX_LOCK()
#define MUTEX_UNLOCK()

#endif


int g_tickets = 100; //总票数

mico_mutex_t mutex = NULL;

char *p_name1 = "t1";
char *p_name2 = "t2";
char *p_name3 = "t3";
char *p_name4 = "t4";


void run( void *arg )
{
    char *name = (char *)arg;
    char debug[20];

    while(1)
        /* 临界代码段 */
    {
        MUTEX_LOCK();      /* 先上锁 */
        if( g_tickets <= 0 ){ /* 若出现异常 */
            MUTEX_UNLOCK(); /* 解锁 */
            goto exit;      /* 退出 用户程序 */
        }
        g_tickets--;       /* 票号递减 */
    }
}
```

```
sprintf( debug, "Thread %s, %d\r\n", name, g_tickets );

MicoUartSend(STDIO_UART, debug, strlen(debug) ); /* 向串口输出 出票情况 */

MUTEX_UNLOCK(); /* 解锁 */

}

exit:

os_mutex_log( "thread: %s exit now", name );
mico_rtos_delete_thread(NULL);
}

int application_start( void )
{
    OSStatus err = kNoErr;

    /* 创建一个互斥锁 */
    err = mico_rtos_init_mutex( &mutex );
    require_noerr(err, exit);

    /* 创建 4 个线程, 分别买票 */
    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY+1, "t1", run, 0x800, p_name1 );
    require_noerr(err, exit);

    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY+1, "t2", run, 0x800, p_name2 );
    require_noerr(err, exit);

    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY+1, "t3", run, 0x800, p_name3 );
    require_noerr(err, exit);

    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY+1, "t4", run, 0x800, p_name4 );
    require_noerr(err, exit);

exit:
    if( err != kNoErr )
        os_mutex_log( "Thread exit with err: %d", err );

    mico_rtos_delete_thread(NULL);
    return err;
}
```

- 1. 无互斥锁时, 运行结果如图 3.3: (分 3 个截图显示)

The screenshot shows two separate terminal windows labeled "serial-com11". Both windows display a series of thread IDs (t1 through t4) and their corresponding ticket numbers. The ticket numbers are listed in a seemingly random order across both windows, indicating that no mutex locks were used to ensure sequential execution.

```

serial-com11 x
Thread t1, 99
Thread t1, 95
Thread t1, 94
Thread t1, 93
Thread t1, 92
Thread t1, 91
Thread t1, 90
Thread t1, 89
Thread t1, 88
Thread t1, 87
Thread t1, 86
Thread t1, 85
Thread t1, 84
Thread t1, 83
Thread t3, 97
Thread t3, 81
Thread t3, 80
Thread t3, 79
Thread t3, 78
Thread t3, 77
Thread t3, 76
Thread t3, 75
Thread t3, 74
Thread t3, 73
Thread t3, 72
Thread t3, 71
Thread t3, 70
Thread t3, 69
Thread t3, 68
Thread t3, 67
Thread t3, 66
Thread t2, 98
Thread t2, 65
Thread t2, 63
Thread t2, 62
Thread t2, 61
Thread t2, 60
Thread t2, 59
Thread t2, 58
Thread t2, 57
Thread t2, 56
Thread t2, 55

serial-com11 x
Thread t2, 54
Thread t2, 52
Thread t2, 50
Thread t2, 48
Thread t4, 98
Thread t4, 47
Thread t4, 46
Thread t4, 44
Thread t4, 43
Thread t4, 42
Thread t4, 41
Thread t4, 35
Thread t4, 33
Thread t4, 32
Thread t1, 82
Thread t1, 30
Thread t1, 29
Thread t1, 28
Thread t1, 27
Thread t1, 26
Thread t1, 25
Thread t1, 24
Thread t1, 23
Thread t1, 21
Thread t1, 20
Thread t1, 19
Thread t1, 18
Thread t1, 17
Thread t1, 16
Thread t1, 15
Thread t3, 65
Thread t3, 13
Thread t3, 12
Thread t3, 11
Thread t3, 10
Thread t3, 9
Thread t3, 8
Thread t3, 7
Thread t3, 6
Thread t3, 5
Thread t3, 4
Thread t3, 3
Thread t3, 2
Thread t3, 1
Thread t3, 0
[131][OS: os_mutex.c: 75] thread: t3 exit now
[137][OS: os_mutex.c: 75] thread: t4 exit now
[142][OS: os_mutex.c: 75] thread: t1 exit now
[148][OS: os_mutex.c: 75] thread: t2 exit now

```

图 3.3 未加互斥锁串口 log

结果显示：未加入互斥锁时，0-99 的票号输出是杂乱无章的，未严格逐次减一。

- 2. 有互斥锁时，运行结果如图 3.4:

The screenshot shows two separate terminal windows labeled "serial-com11". Both windows display a series of thread IDs (t1 through t4) and their corresponding ticket numbers. The ticket numbers are listed sequentially from 1 to 99 in both windows, indicating that mutex locks were used to ensure that threads only decrease their ticket numbers one at a time.

```

serial-com11 x
Thread t1, 99
Thread t1, 98
Thread t1, 97
Thread t1, 96
Thread t1, 95
Thread t1, 94
Thread t1, 93
Thread t1, 92
Thread t1, 91
Thread t1, 90
Thread t1, 89
Thread t1, 88
Thread t1, 87
Thread t1, 86
Thread t1, 85
Thread t1, 84
Thread t1, 83
Thread t1, 82
Thread t1, 81
Thread t1, 80
Thread t1, 79
Thread t1, 78
Thread t1, 77
Thread t1, 76
Thread t1, 75
Thread t1, 74
Thread t1, 73
Thread t1, 72
Thread t1, 71
Thread t1, 70
Thread t1, 69
Thread t1, 68
Thread t1, 67
Thread t1, 66
Thread t1, 65
Thread t1, 64
Thread t1, 63
Thread t1, 62
Thread t1, 61
Thread t1, 60
Thread t1, 59
Thread t1, 58
Thread t4, 57
Thread t4, 56
Thread t4, 55
Thread t4, 54
Thread t4, 53
Thread t4, 52
Thread t4, 51
Thread t4, 50
Thread t4, 49
Thread t4, 48
Thread t4, 47
Thread t4, 46
Thread t4, 45
Thread t2, 44
Thread t2, 43
Thread t2, 42
Thread t2, 41
Thread t2, 40
Thread t2, 39
Thread t2, 38
Thread t2, 37
Thread t2, 36
Thread t2, 35
Thread t2, 34
Thread t2, 33
Thread t2, 32
Thread t2, 31
Thread t2, 30
Thread t2, 29
Thread t2, 28
Thread t2, 27
Thread t2, 26
Thread t2, 25
Thread t2, 24
Thread t2, 23
Thread t2, 22
Thread t2, 21
Thread t2, 20
Thread t2, 19
Thread t2, 18
Thread t2, 17
Thread t2, 16
Thread t3, 15
Thread t3, 14
Thread t3, 13
Thread t3, 12
Thread t3, 11
Thread t3, 10
Thread t3, 9
Thread t3, 8
Thread t3, 7
Thread t3, 6
Thread t3, 5
Thread t3, 4
Thread t3, 3
Thread t3, 2
Thread t3, 1
Thread t3, 0
[136][OS: os_mutex.c: 75] thread: t3 exit now
[144][OS: os_mutex.c: 75] thread: t2 exit now
[145][OS: os_mutex.c: 75] thread: t4 exit now
[149][OS: os_mutex.c: 75] thread: t1 exit now

```

图 3.4 加入互斥锁串口 log

结果显示：加入互斥锁后，0-99 的票号输出是按顺序的，严格逐次递减一。

MiCO Mutex API 内容，请参考：“MiCO API 参考手册”。

3.4 os_sem

3.4.1 Sem 信号量概述

通过 MiCO RTOS 信号量控制 API 可以在系统中实现：信号量的初始化，设置，获取和销毁功能。

信号量(Semaphore)，有时被称为信号灯，是在多线程环境下使用的一种设施，是可以用来保证两个或多个关键代码段不被并发调用。在进入一个关键代码段之前，线程必须获取一个信号量；一旦该关键代码段完成了，那么该线程必须释放信号量。其它想进入该关键代码段的线程必须等待直到第一个线程释放信号

量。为了完成这个过程，需要创建一个信号量 VI，然后将 Acquire Semaphore VI 以及 Release Semaphore VI 分别放置在每个关键代码段的首末端，确认这些信号量 VI 引用的是初始创建的信号量。

3.4.2 os_sem.c 功能说明

本 Demo 运行输出 log 如图 3.5：

```
[0][OS: os_sem.c: 54] test binary semaphore
[4][OS: os_sem.c: 43] release semaphore
[8][OS: os_sem.c: 64] get semaphore
[3008][OS: os_sem.c: 43] release semaphore
[3012][OS: os_sem.c: 64] get semaphore
[6012][OS: os_sem.c: 43] release semaphore
[6016][OS: os_sem.c: 64] get semaphore
[9016][OS: os_sem.c: 43] release semaphore
[9020][OS: os_sem.c: 64] get semaphore
[12020][OS: os_sem.c: 43] release semaphore
[12024][OS: os_sem.c: 64] get semaphore
[15024][OS: os_sem.c: 43] release semaphore
[15028][OS: os_sem.c: 64] get semaphore
[18028][OS: os_sem.c: 43] release semaphore
[18032][OS: os_sem.c: 64] get semaphore
```

图 3.5 信号量串口 log

本例实现功能：

- (1) 主线程和创建的一个新线程分别通过信号量来获取执行权
- (2) 新线程设置信号量，主线程获取信号量，拿到执行权。

```
#include "mico.h"

#define os_sem_log(M, ...) custom_log("OS", M, ##__VA_ARGS__)

static mico_semaphore_t os_sem = NULL;

void release_thread(void *arg)
{
    UNUSED_PARAMETER(arg);

    while(1){
        os_sem_log( "release semaphore" );
        mico_rtos_set_semaphore( &os_sem );
        mico_thread_sleep( 3 );
    }
    mico_rtos_delete_thread(NULL);
}

int application_start( void )
{
    OSStatus err = kNoErr;
    os_sem_log( "test binary semaphore" );

    err = mico_rtos_init_semaphore( &os_sem, 1 ); /* 0/1 二进制信号量或 0/N 信号量 */
}
```

```
require_noerr( err, exit );

/* 创建一个信号量 */

err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY, "release sem", release_thread, 500, NULL );
require_noerr( err, exit );

while(1){

    mico_rtos_get_semaphore( &os_sem, MICO_WAIT_FOREVER ); /* 等待直到获取信号量 */
    os_sem_log( "get semaphore" );
}

exit:

if( err != kNoErr )

    os_sem_log( "Thread exit with err: %d", err );

if( os_sem != NULL )

    mico_rtos_deinit_semaphore( &os_sem ); /* 销毁信号量 */

mico_rtos_delete_thread( NULL );           /* 删除当前线程 */

return err;

}
```

3.5 os_queue

3.5.1 Queue 消息队列概述

通过 MiCO RTOS 消息队列控制 API 可以在系统中实现：消息队列的初始化，数据写入，数据读出，销毁，及查询队列状态（空或满）的功能。

消息被发送到队列中，“消息队列”是在消息的传输过程中保存消息的容器。消息队列管理器在将消息从它的源中继到它的目标时充当中间人。消息队列主要作用是提供路由并保证消息的传递；如果发送消息时接收者不可用，消息队列会保留消息，直到可以成功地传递它。

3.5.2 os_queue.c 功能说明

本 Demo 运行结果如图 3.6：

```

[0][OS: os_queue.c: 78] send data to queue
[4][OS: os_queue.c: 55] Received data from queue:value = 1
[1004][OS: os_queue.c: 78] send data to queue
[1008][OS: os_queue.c: 55] Received data from queue:value = 2
[2008][OS: os_queue.c: 78] send data to queue
[2012][OS: os_queue.c: 55] Received data from queue:value = 3
[3012][OS: os_queue.c: 78] send data to queue
[3016][OS: os_queue.c: 55] Received data from queue:value = 4
[4016][OS: os_queue.c: 78] send data to queue
[4020][OS: os_queue.c: 55] Received data from queue:value = 5
[5020][OS: os_queue.c: 78] send data to queue
[5024][OS: os_queue.c: 55] Received data from queue:value = 6
[6024][OS: os_queue.c: 78] send data to queue
[6028][OS: os_queue.c: 55] Received data from queue:value = 7
[7028][OS: os_queue.c: 78] send data to queue
[7032][OS: os_queue.c: 55] Received data from queue:value = 8
[8032][OS: os_queue.c: 78] send data to queue
[8036][OS: os_queue.c: 55] Received data from queue:value = 9
[9036][OS: os_queue.c: 78] send data to queue
[9040][OS: os_queue.c: 55] Received data from queue:value = 10

```

图 3.6 消息队列串口 log

本例实现功能：

- (1) 初始化消息队列；同时创建两个线程，一个向消息队列中写入数据，一个从消息队列中读出数据。
- (2) 其中一个线程写入数据时，做增 1 处理。

本 Demo 程序如下：

```

#include "mico.h"

#define os_queue_log(M, ...) custom_log("OS", M, ##__VA_ARGS__)

static mico_queue_t os_queue = NULL;

typedef struct _msg
{
    int value;
} msg_t;

/* 接收线程 定义*/
void receiver_thread(void *arg)
{
    UNUSED_PARAMETER( arg );
}

```

```
OSStatus err;
msg_t received = { 0 };

while(1)
{
    /* 将对象从消息队列中读出 */
    err = mico_rtos_pop_from_queue( &os_queue, &received, MICO_WAIT_FOREVER );
    require_noerr( err, exit );
    os_queue_log( "Received data from queue:value = %d", received.value );
}

exit:
if( err != kNoErr )
{
    os_queue_log( "Receiver exit with err: %d", err );
    mico_rtos_delete_thread( NULL );
}

/* 发送线程 定义*/
void sender_thread(void *arg)
{
    UNUSED_PARAMETER( arg );

    OSStatus err = kNoErr;
    msg_t my_message = { 0 };

    while(1)
    {
        my_message.value++;

        /* 将一个对象写入消息队列 */
        mico_rtos_push_to_queue( &os_queue, &my_message, MICO_WAIT_FOREVER );
        require_noerr( err, exit );

        os_queue_log( "send data to queue" );
        mico_thread_sleep( 1 );
    }

exit:
if( err != kNoErr )
```

```
    os_queue_log( "Sender exit with err: %d", err );
    mico_rtos_delete_thread( NULL );
}

int application_start( void )
{
    OSStatus err = kNoErr;
    /* 初始化 FIFO 队列 */
    err = mico_rtos_init_queue( &os_queue, "queue", sizeof(msg_t), 3 );
    require_noerr( err, exit );
    /* 创建队列接收线程 */
    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY, "receiver", receiver_thread, 500, NULL );
    require_noerr( err, exit );
    /* 创建队列发送线程 */
    err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY, "sender", sender_thread, 500, NULL );
    require_noerr( err, exit );

exit:
    if( err != kNoErr )
        os_queue_log( "Thread exit with err: %d", err );
    mico_rtos_delete_thread( NULL );

    return err;
}
```

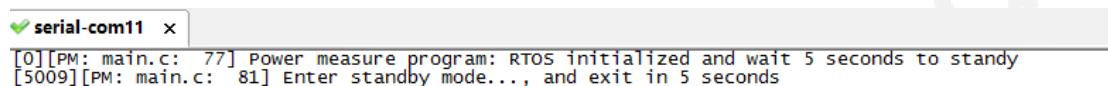
4. 系统电源管理

4.1 MiCO 电源管理概述

MiCO 系统为用户提供多种电源工作模式管理，根据配置参数定义，选择进入不同的模式，如 STANDBY_MODE-待机模式，RTOS_INITIALIZED-实时操作系统初始化模式等，详见程序代码参数定义。

4.2 power_measure.c 功能说明

本例实现 MiCO 设备低功耗模式开启（待机模式），运行 log 如图 4.1：



```
[0][PM: main.c: 77] Power measure program: RTOS initialized and wait 5 seconds to standby
[5009][PM: main.c: 81] Enter standby mode..., and exit in 5 seconds
```

图 4.1 低功耗串口 log

运行后，

(1) 系统首先根据 POWER_MEASURE_PROGRAM 参数，进入相应的入口函数：

```
int application_start( void )
```

(2) 如本例，进入 STANDBY_MODE 待机模式

本例程序代码如下：

```
#include "mico.h"

#define power_log(M, ...) custom_log("PM", M, ##_VA_ARGS__)

/* 定义电源管理模式参数 宏定义 */

#define POWER_MEASURE_PROGRAM STANDBY_MODE

#define MCU_POWERSAVE_ENABLED 0

#define IEEE_POWERSAVE_ENABLED 1

/* 以下，电源管理模式参数值 宏定义 */

#define RTOS_INITIALIZED 1
#define RTOS_FULL_CPU_THREAD 2
#define RTOS_FLASH_READ 3
#define RTOS_FLASH_ERASE 4
#define RTOS_FLASH_WRITE 5
#define RTOS_WLAN_INITIALIZED 6
#define RTOS_WLAN_SOFT_AP 7
#define RTOS_WLAN_EASYLINK 8
#define RTOS_WLAN_CONNECT 9
#define RTOS_WLAN_UDP_SEND 10
#define STANDBY_MODE 11
```

```
/* wifi 状态改变通知*/
void micoNotify_WifiStatusHandler(WiFiEvent event, const int inContext)
{
    switch (event) {
        case NOTIFY_STATION_UP:
            power_log("Station up");
            MicoRfLed(true);
            break;
        case NOTIFY_STATION_DOWN:
            power_log("Station down");
            MicoRfLed(false);
            break;
        default:
            break;
    }
    return;
}

//电源工作模式 设置为 STANDBY_MODE 时，函数入口
#if POWER_MEASURE_PROGRAM == STANDBY_MODE
int application_start( void )
{
#if MCU_POWERSAVE_ENABLED
    MicoMcuPowerSaveConfig(true);
#endif
    power_log( "Power measure program: RTOS initialized and wait 5 seconds to standby" );
    mico_thread_sleep( 5 ); //Wait a period to avoid enter standby mode when boot
    power_log( "Enter standby mode..., and exit in 5 seconds" );
}

#endif EMW1088
//MicoInit( );
//micoWlanPowerOff( );
//wlan_deepsleeps_on( );
#endif

#if IEEE_POWERSAVE_ENABLED
    micoWlanEnablePowerSave();
#endif
```

```
MicoSystemStandBy( 5 );

power_log( "Enter standby mode error!" );

mico_rtos_delete_thread( NULL );
return 0;
}

#endif

//电源工作模式 设置为 RTOS_INITIALIZED 时，函数入口

#if POWER_MEASURE_PROGRAM == RTOS_INITIALIZED
int application_start( void )
{
#if MCU_POWERSAVE_ENABLED
    MicoMcuPowerSaveConfig(true);
#endif

    power_log( "Power measure program: RTOS initialized and no application is running" );

    mico_rtos_delete_thread( NULL );
    return 0;
}
#endif

//电源工作模式 设置为 RTOS_FULL_CPU_THREAD 时，函数入口

#if POWER_MEASURE_PROGRAM == RTOS_FULL_CPU_THREAD
int application_start( void )
{
#if MCU_POWERSAVE_ENABLED
    MicoMcuPowerSaveConfig(true);
#endif

    power_log( "Power measure program: RTOS initialized and application is running full speed" );

    while(1);
    return 0;
}
```

```
}

#endif

//电源工作模式 设置为 RTOS_FLASH_ERASE 时，函数入口

#if POWER_MEASURE_PROGRAM == RTOS_FLASH_ERASE

int application_start( void )

{

#if MCU_POWERSAVE_ENABLED

    MicoMcuPowerSaveConfig(true);

#endif

    power_log( "Power measure program: RTOS initialized and erase flash" );

    MicoFlashInitialize( MICO_FLASH_FOR_UPDATE );

    MicoFlashErase( MICO_FLASH_FOR_UPDATE, UPDATE_START_ADDRESS, UPDATE_END_ADDRESS );

    MicoFlashFinalize( MICO_FLASH_FOR_UPDATE );

    mico_rtos_delete_thread( NULL );

    return 0;

}

#endif

//电源工作模式 设置为 RTOS_WLAN_INITIALIZED 时，函数入口

#if POWER_MEASURE_PROGRAM == RTOS_WLAN_INITIALIZED

int application_start( void )

{

#if MCU_POWERSAVE_ENABLED

    MicoMcuPowerSaveConfig(true);

#endif

    power_log( "Power measure program: RTOS and wlan initialized and wlan is initialized" );

    MicoInit( );

    #if IEEE_POWERSAVE_ENABLED

        micoWlanEnablePowerSave();

    #endif

    mico_rtos_delete_thread( NULL );

    return 0;

}
```

```
#endif

//电源工作模式 设置为 RTOS_WLAN_SOFT_AP 时, 函数入口

#if POWER_MEASURE_PROGRAM == RTOS_WLAN_SOFT_AP

int application_start( void )

{

    network_InitTypeDef_st wNetConfig;

#if MCU_POWERSAVE_ENABLED

    MicoMcuPowerSaveConfig(true);

#endif

    power_log( "Power measure program: RTOS and wlan initialized and setup soft ap" );

    MicoInit( );




    memset(&wNetConfig, 0, sizeof(network_InitTypeDef_st));

    wNetConfig.wifi_mode = Soft_AP;

    sprintf(wNetConfig.wifi_ssid, 32, "EasyLink_PM" );

    strcpy((char*)wNetConfig.wifi_key, "");

    strcpy((char*)wNetConfig.local_ip_addr, "10.10.10.1");

    strcpy((char*)wNetConfig.net_mask, "255.255.255.0");

    strcpy((char*)wNetConfig.gateway_ip_addr, "10.10.10.1");

    wNetConfig.dhcpMode = DHCP_Server;

    micoWlanStart(&wNetConfig);


    mico_rtos_delete_thread( NULL );

    return 0;

}

#endif

//电源工作模式 设置为 RTOS_WLAN_EASYLINK 时, 函数入口

#if POWER_MEASURE_PROGRAM == RTOS_WLAN_EASYLINK

int application_start( void )

{

    network_InitTypeDef_st wNetConfig;

#if MCU_POWERSAVE_ENABLED

    MicoMcuPowerSaveConfig(true);

#endif

    power_log( "Power measure program: RTOS and wlan initialized and start easylink" );

    MicoInit( );


}
```

```
micoWlanStartEasyLinkPlus( MICO_NEVER_TIMEOUT );\n\nmico_rtos_delete_thread( NULL );\n\nreturn 0;\n}\n#endif\n//电源工作模式 设置为 RTOS_WLAN_CONNECT 时，函数入口\n#if POWER_MEASURE_PROGRAM == RTOS_WLAN_CONNECT\nint application_start( void )\n{\n    MICOAddNotification( mico_notify_WIFI_STATUS_CHANGED, (void *)micoNotify_WifiStatusHandler );\n\n    network_InitTypeDef_adv_st wNetConfig;\n\n#if MCU_POWERSAVE_ENABLED\n    MicoMcuPowerSaveConfig(true);\n#endif\n\n    power_log( "Power measure program: RTOS and wlan initialized and connect wlan, wait station up to\nmeasure" );\n\n    MicoInit( );\n\n#if IEEE_POWERSAVE_ENABLED\n    micoWlanEnablePowerSave();\n#endif\n\n    memset(&wNetConfig, 0x0, sizeof(network_InitTypeDef_adv_st));\n\n    strncpy((char*)wNetConfig.ap_info.ssid, "William Xu", 32);\n    wNetConfig.ap_info.security = SECURITY_TYPE_AUTO;\n    strncpy((char*)wNetConfig.key, "mx099555", 64);\n    wNetConfig.key_len = 8;\n    wNetConfig.dhcpMode = true;\n    wNetConfig.wifi_retry_interval = 100;\n    micoWlanStartAdv(&wNetConfig);\n    power_log("connect to %s.....", wNetConfig.ap_info.ssid);\n\n    mico_rtos_delete_thread( NULL );\n\n    return 0;\n}
```

```
}

#endif

//电源工作模式 设置为 RTOS_WLAN_UDP_SEND 时，函数入口

#if POWER_MEASURE_PROGRAM == RTOS_WLAN_UDP_SEND

int application_start( void )

{

    network_InitTypeDef_adv_st wNetConfig;

    int udp_fd = -1;

    struct sockaddr_t addr;

    socklen_t addrLen;

    uint8_t *buf = NULL;

    #if MCU_POWERSAVE_ENABLED

        MicoMcuPowerSaveConfig(true);

    #endif

    power_log( "Power measure program: RTOS and wlan initialized and connect wlan, wait station up to
measure" );

    MicoInit( );

    #if IEEE_POWERSAVE_ENABLED

        micoWlanEnablePowerSave();

    #endif

    memset(&wNetConfig, 0x0, sizeof(network_InitTypeDef_adv_st));

    strncpy((char*)wNetConfig.ap_info.ssid, "William Xu", 32);

    wNetConfig.ap_info.security = SECURITY_TYPE_AUTO;

    strncpy((char*)wNetConfig.key, "mx099555", 64);

    wNetConfig.key_len = 8;

    wNetConfig.dhcpMode = true;

    wNetConfig.wifi_retry_interval = 100;

    micoWlanStartAdv(&wNetConfig);

    power_log("connect to %s.....", wNetConfig.ap_info.ssid);

    buf = malloc(1024);

    udp_fd = socket(AF_INET, SOCK_DGRAM, IPPROTO_UDP);;

    addr.s_port = 2000;
```

```
addr.s_ip = INADDR_ANY;  
bind(udp_fd, &addr, sizeof(addr));  
  
addr.s_port = 2001;  
addr.s_ip = inet_addr( "192.168.2.1" );  
  
connect( udp_fd, &addr, sizeof(addr) );  
  
while(1) {  
    send( udp_fd, buf, 1024, 0 );  
    mico_thread_msleep( 10 );  
}  
  
mico_rtos_delete_thread( NULL );  
return 0;  
}  
#endif
```

5. Wi-Fi 功能

5.1 wifi_scan

本例为 MiCO 的 Wi-Fi 扫描 demo，扫描完成后，产生系统通知，执行回调函数。

5.1.1 wifi_sacn.c 功能说明

本例输出 log 如下：

```

[2][P]latform: mico_platform_common.c: 96] Platform initialised, build by IAR
[631][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1039][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1066][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1640][SYSTEM: system_misc.c: 222] Free memory 66360 bytes
[1646][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1652][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1657][SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Nov 7 2014 16:03:45 version 5.90.230.12 FWI, ma
[1670][WIFI: wifi_scan.c: 56] start scan mode, please wait...
[3214][WIFI: wifi_scan.c: 39] got 29 AP
[3218][WIFI: wifi_scan.c: 43] ap0: name = Xiaomi.Router | strength=100
[3225][WIFI: wifi_scan.c: 43] ap1: name = AP008 | strength=100
[3230][WIFI: wifi_scan.c: 43] ap2: name = _MICO_ALINK_HA_ | strength=100
[3237][WIFI: wifi_scan.c: 43] ap3: name = SWYANG | strength=100
[3243][WIFI: wifi_scan.c: 43] ap4: name = mxchip-offices | strength=100
[3250][WIFI: wifi_scan.c: 43] ap5: name = mxchip-rd | strength=100
[3256][WIFI: wifi_scan.c: 43] ap6: name = william xu | strength=100
[3263][WIFI: wifi_scan.c: 43] ap7: name = MX_00 | strength=100
[3268][WIFI: wifi_scan.c: 43] ap8: name = wifi | strength=100
[3274][WIFI: wifi_scan.c: 43] ap9: name = mx_maojian | strength=100
[3281][WIFI: wifi_scan.c: 43] ap10: name = mxchip-offices | strength=95
[3287][WIFI: wifi_scan.c: 43] ap11: name = mxchip-guest | strength=95
[3294][WIFI: wifi_scan.c: 43] ap12: name = mxchip-offices | strength=87
[3301][WIFI: wifi_scan.c: 43] ap13: name = mxchip-rd | strength=87
[3307][WIFI: wifi_scan.c: 43] ap14: name = SAD | strength=85
[3313][WIFI: wifi_scan.c: 43] ap15: name = mxchip-offices | strength=85

```

图 5.1 wifi 扫描串口 log

本例完成：

(1) 扫描可用的 Wi-Fi，结束后，产生系统通知，执行回调函数，打印输出 wifi 列表。实例代码如下：

```

int application_start( void )
{
    /* 根据 mico_config.h 配置，启动 MiCO 系统功能*/
    mico_system_init( mico_system_context_init( 0 ) );

    /* 当 wlan 扫描完成，注册一个用户功能通知 */
    mico_system_notify_register( mico_notify_WIFI_SCAN_COMPLETED, (void *)micoNotify_ApListCallback, NULL );

    wifi_scan_log("start scan mode, please wait...");

    micoWlanStartScan( ); /* 基于 MiCO 系统启动 2.4GHz 频段的 wlan 扫描功能 */

    mico_rtos_delete_thread( NULL );

    return kNoErr;
}

```

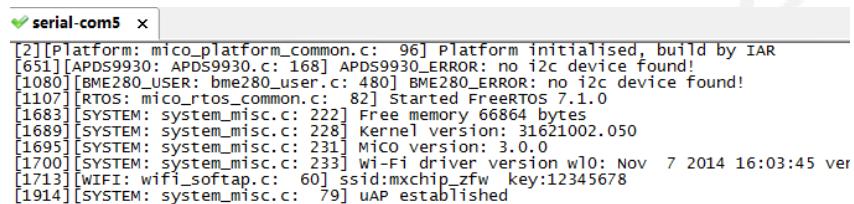
5.2 wifi_softap

5.2.1 SoftAP 概述

Soft AP 模式是一种通过无线网卡，使用专用软件在 PC 上实现 AP 功能的技术，它可以取代无线网络中的 AP (Access Point, 无线接入点)，从而会降低无线组网的成本。也就是可以把载体（这里是 MiCO 设备）作为无线接入点，让电脑、手机或者其他上网设备的无线网连接到载体上，然后通过载体的网络 (GPRS 或者 3G) 上网。

5.2.2 wifi_softap 功能说明

本例 log 输出如下：



```
[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1080][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1107][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1683][SYSTEM: system_misc.c: 222] Free memory 66864 bytes
[1689][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1695][SYSTEM: system_misc.c: 231] Mico version: 3.0.0
[1700][SYSTEM: system_misc.c: 233] Wi-Fi driver version w10: Nov 7 2014 16:03:45 ver
[1713][WIFI: wifi_softap.c: 60] ssid:mxchip_zfw key:12345678
[1914][SYSTEM: system_misc.c: 79] uAP established
```

图 5.2 SoftAP 串口 log

本 Demo 实现：

- (1) 通过设置 MiCO 无线网络配置参数 wNetConfig，开启 softap 工作模式

本例程序代码如下：

```
#include "mico.h"

#define wifi_softap_log(M, ...) custom_log("WIFI", M, ##__VA_ARGS__)

//此处为新建 AP 的 ssid 和 key，用户可自定义

static char *ap_ssid = "mxchip_zfw";
static char *ap_key = "12345678";

int application_start( void )
{
    OSStatus err = kNoErr;

    network_InitTypeDef_st wNetConfig;

    err = mico_system_init( mico_system_context_init( 0 ) );
    require_noerr( err, exit );

    memset(&wNetConfig, 0x0, sizeof(network_InitTypeDef_st));

    strcpy((char*)wNetConfig.wifi_ssid, ap_ssid);
```

```

strcpy((char*)wNetConfig.wifi_key, ap_key);

wNetConfig.wifi_mode = Soft_AP;
wNetConfig.dhcpMode = DHCP_Server;
wNetConfig.wifi_retry_interval = 100;
strcpy((char*)wNetConfig.local_ip_addr, "192.168.0.1");
strcpy((char*)wNetConfig.net_mask, "255.255.255.0");
strcpy((char*)wNetConfig.dnsServer_ip_addr, "192.168.0.1");

wifi_softap_log("ssid:%s key:%s", wNetConfig.wifi_ssid, wNetConfig.wifi_key);

Connect or establish a Wi-Fi network in normal mode (station or soft ap mode).

/* 连接或建立一个 Wi-Fi 网络，工作正常模式下，如 station 或 soft ap 模式*/
micoWlanStart(&wNetConfig);

exit:
mico_rtos_delete_thread(NULL);
return err;
}

```

5.3 wifi_station_core_api

5.3.1 Station 模式概述

Station (工作站): 表示连接到无线网络中的设备，这些设备通过 AP，可以和内部其它设备或者无线网络外部通信。(AP (Access point 的简称，即访问点，接入点): 是一个无线网络中的特殊节点，通过这个节点，无线网络中的其它类型节点可以和无线网络外部以及内部进行通信。)

5.3.2 wifi_station_core_api.c 功能说明

本例运行 log 输出如下:

```

[0][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[629][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1037][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1064][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1640][WIFI: wifi_station_core_api.c: 81] connecting to Xiaomi.Router...
[6538][WIFI: wifi_station_core_api.c: 46] Station up

```

图 5.3 Station 模式（指定 wifi）串口 log

本例实现:

- (1) 根据 mico_config.h 的配置，进行 MiCO 功能初始化
- (2) 根据定义的 MiCO 系统无线网络配置参数，包括 wifi 的 ssid 账号和 key 密码写入设备，接入 wifi

网络，开启 Station 模式

本 Demo 程序代码如下：

```
#include "mico.h"

#define wifi_station_log(M, ...) custom_log("WIFI", M, ##_VA_ARGS__)

/* 一次连接失败时的，用户通知功能回调函数 */

static void micoNotify_ConnectFailedHandler(OSStatus err, void* inContext)
{
    wifi_station_log("join Wlan failed Err: %d", err);
}

/* wifi 状态改变时的，用户通知功能回调函数 */

static void micoNotify_WifiStatusHandler(WiFiEvent event, void* inContext)
{
    switch (event)
    {
        case NOTIFY_STATION_UP: //成功连接
            wifi_station_log("Station up");
            break;

        case NOTIFY_STATION_DOWN: //连接失败
            wifi_station_log("Station down");
            break;

        default:
            break;
    }
}

int application_start( void )
{
    OSStatus err = kNoErr;
    network_InitTypeDef_adv_st wNetConfigAdv={0};

    MicoInit( );

    /* 当 wlan 连接状态改变时，注册一个用户通知功能 micoNotify_WifiStatusHandler () */
    err = mico_system_notify_register(mico_notify_WIFI_STATUS_CHANGED, (void *)micoNotify_WifiStatusHandler,
    NULL );
    require_noerr( err, exit );

    /* 当 wlan 连接尝试一次失败时，注册一个用户通知功能 micoNotify_ConnectFailedHandler, () */
    err = mico_system_notify_register( mico_notify_WIFI_CONNECT_FAILED, (void *)micoNotify_ConnectFailedHandler,
```

```

NULL );

require_noerr( err, exit );

/* 初始化 wlan 参数 */

// 用户需修改 ssid, key 和 keylen 为自己的密码

strcpy((char*)wNetConfigAdv.ap_info.ssid, "Xiaomi.Router"); /* wlan ssid 的字符串 */

strcpy((char*)wNetConfigAdv.key, "stm32f215"); /* wlan key 的字符串或 WEP 模式下的十六进制数据 */

wNetConfigAdv.key_len = strlen("stm32f215"); /* wlan key 的长度 */

wNetConfigAdv.ap_info.security = SECURITY_TYPE_AUTO; /* wlan 安全模式 */

wNetConfigAdv.ap_info.channel = 0; /* 自动选择通道 */

wNetConfigAdv.dhcpMode = DHCP_Client; /* 从 DHCP 服务器获取 IP 地址 */

wNetConfigAdv.wifi_retry_interval = 100; /* 尝试一次失败后的重试时间间隔 */

/* 此刻开始连接 */

wifi_station_log("connecting to %s...", wNetConfigAdv.ap_info.ssid);

/* 根据设置，连接 Wi-Fi 网络（仅 station 模式） */

micoWlanStartAdv(&wNetConfigAdv);

exit:

mico_rtos_delete_thread(NULL);

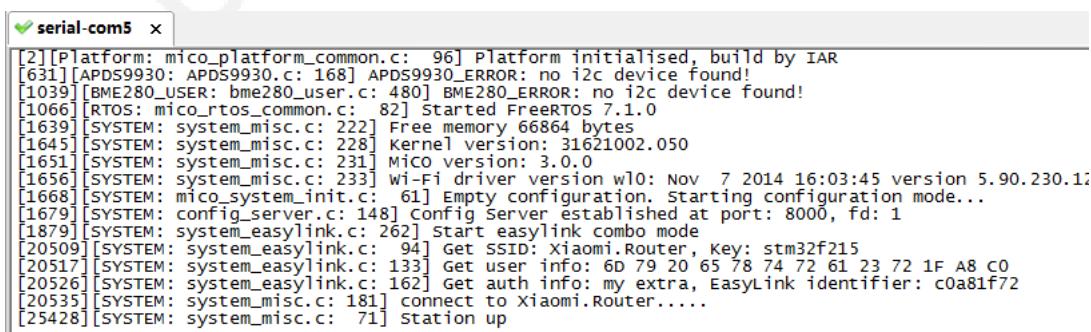
return err;
}

```

5.4 wifi_station_system_api.c

5.4.1 wifi_station_system_api 功能说明

本例实现 MiCO 设备 Station 模式入网，运行 log 如下：



```

[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[631][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1039][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1066][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1639][SYSTEM: system_misc.c: 222] Free memory 66864 bytes
[1645][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1651][SYSTEM: system_misc.c: 231] Mico version: 3.0.0
[1656][SYSTEM: system_misc.c: 233] Wi-Fi driver version w10: Nov 7 2014 16:03:45 version 5.90.230.1
[1668][SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode...
[1679][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1879][SYSTEM: system_easylink.c: 262] Start easylink combo mode
[20509][SYSTEM: system_easylink.c: 94] Get SSID: Xiaomi.Router, Key: stm32f215
[20517][SYSTEM: system_easylink.c: 133] Get user info: 60 79 20 65 78 74 72 61 23 72 1F A8 C0
[20526][SYSTEM: system_easylink.c: 162] Get auth info: my extra, EasyLink identifier: c0a81f72
[20535][SYSTEM: system_misc.c: 181] connect to Xiaomi.Router.....
[25428][SYSTEM: system_misc.c: 71] Station up

```

图 5.4 Station 模式 (Easylink 配网) 串口 log

运行后：

- (1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)

(2) 设备进入无线网络配置，首先检测 flash 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网。

(3) 配网成功后，输出 Station up，如上图 log。

- 本例是一个 MiCO 系统实现 Easylink 或 Airkiss 配网的单一例程，开发者只需：

基于 MiCO SDK，在用户入口函数：int application_start(void)中只需加入如下代码，即可使工程中具备 Easylink 或 Airkiss 配网功能。程序代码如下：

```
#include "mico.h"

int application_start( void )
{
    OSStatus err = kNoErr;
    /*
        根据 mico_config.h 配置，启动 MiCO 系统功能
        定义 MICO_WLAN_CONNECTION_ENABLE 以使能 wlan 连接功能
        选择 wlan 配置模式：MICO_CONFIG_MODE，定义 Easylink 设置
        该示例需重新配置的 mico_config.h 文件路径，目录：demos/wifi/station，该头文件已配置好。
    */
    err = mico_system_init( mico_system_context_init( 0 ) );

    mico_rtos_delete_thread(NULL);
    return err;
}
```

6. 网络通信

6.1 dns

6.1.1 dns 概述

DNS 是域名系统 (Domain Name System) 的缩写，该系统用于命名组织到域层次结构中的计算机和网络服务。在 Internet 上域名与 IP 地址之间是一一对应的，域名虽然便于人们记忆，但机器之间只能互相认识 IP 地址，它们之间的转换工作称为域名解析，域名解析需要由专门的域名解析服务器来完成，DNS 就是进行域名解析的服务器。DNS 命名用于 Internet 等 TCP/IP 网络中，通过用户友好的名称查找计算机和服务。当用户在应用程序中输入 DNS 名称时，DNS 服务可以将此名称解析为与之相关的其他信息，如 IP 地址。因为，你在上网时输入的网址，是通过域名解析系统解析找到相对应的 IP 地址，这样才能上网。其实，域名的最终指向是 IP。

本例将展现 MiCO 中完成 DNS 域名解析功能，需要调用哪些 API 接口函数。

6.1.2 dns.c 功能描述

本例实现对"www.baidu.com"的域名解析，运行 log 如下：

```

[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[652][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1081][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1108][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1685][SYSTEM: system_misc.c: 222] Free memory 66736 bytes
[1691][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1697][SYSTEM: system_misc.c: 231] Micro version: 3.0.0
[1702][SYSTEM: system_misc.c: 233] Wi-Fi driver version v10: Nov 7 2014 16:03:45 version 5.90.230.12 FWI, mac D0:BA
[1713][SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode.
[1724][SYSTEM: config_server.c: 148] Config server established at port: 8000, fd: 1
[1924][SYSTEM: system_easylink.c: 262] Start easylink combo mode
[6367][SYSTEM: system_easylink.c: 94] Get SSID: Xiaomi.Router, key: STM32F215
[6375][SYSTEM: system_easylink.c: 133] Get user info: 6D 79 20 65 78 74 72 61 23 72 1F A8 C0
[6383][SYSTEM: system_easylink.c: 162] Get auth info: my extra, EasyLink identifier: c0a81f72
[6393][SYSTEM: system_misc.c: 181] connect to Xiaomi.Router....
[14046][SYSTEM: system_misc.c: 71] Station up
[14143][DNS: dns.c: 74] wifi connected successful
[14202][DNS: dns.c: 84] www.baidu.com ip address:115.239.211.112 解析域名

```

图 6.1 dns 域名解析串口 log

(1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)

(2) 设备进入无线网络配置，首先检测 flash 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网。

(3) 配网成功后，输出 Station up，如上图 log。

(4) 然后解析 www.baidu.com 的 IP 地址

```

#include "mico.h"

#define dns_log(M, ...) custom_log("DNS", M, ##__VA_ARGS__)

static mico_semaphore_t wait_sem = NULL;

static char *domain = "www.baidu.com";

static void micoNotify_WifiStatusHandler(WiFiEvent status, void* const inContext)
{

```

```
switch (status) {
    case NOTIFY_STATION_UP:
        mico_rtos_set_semaphore(&wait_sem);
        break;
    case NOTIFY_STATION_DOWN:
        break;
}
}

int application_start( void )
{
    OSStatus err = kNoErr;
    char ipstr[16];
    /* 初始化信号量 */
    mico_rtos_init_semaphore( &wait_sem,1 );
    /*为 MiCO 通知注册用户功能： Wi-Fi 状态改变*/
    err = mico_system_notify_register( mico_notify_WIFI_STATUS_CHANGED, (void *)micoNotify_WifiStatusHandler,
NULL );
    require_noerr( err, exit );
    /* 根据 mico_config.h 设置，启动 MiCO 系统功能*/
    err = mico_system_init( mico_system_context_init( 0 ) );
    require_noerr( err, exit );

    /* 等待至 wifi 连接成功*/
    mico_rtos_get_semaphore( &wait_sem, MICO_WAIT_FOREVER );
    dns_log( "wifi connected successful" );

    /* 解析 DNS 地址 */
    err = gethostbyname( domain, (uint8_t *)ipstr, 16 );
    require_noerr( err, exit );
    dns_log( "%s ip address is %s",domain, ipstr );

exit:
    if( wait_sem != NULL)
        mico_rtos_deinit_semaphore( &wait_sem ); /* 删除信号量 */
    mico_rtos_delete_thread( NULL );
    return err;
}
```

6.2 mDNS

6.2.1 mDNS 概述

mDNS 主要实现：在没有传统 DNS 服务器的情况下，使局域网内的主机实现相互发现和通信。如：苹果的 Bonjour 就是一个基于 mDNS 的产品。

6.2.2 mDNS.c 功能描述

本例实现开启 Bonjour 服务，设备自动发现。运行 log 如下：

```

[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1080][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1107][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1680][SYSTEM: system_misc.c: 222] Free memory 66864 bytes
[1686][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1692][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1697][SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Nov 7 2014 16:03:45 version 5.90.230.12 FWI, 1
[1709][SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode...
[1720][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1920][mDNS: mdns.c: 102] TXT RECORD=MAC=d0bae4078ff0.Firmware Rev=MICO_BASIC_1_0.Hardware Rev=MK3165_1.MIC
[1939][SYSTEM: system_easylink.c: 262] Start easylink combo mode
[17262][SYSTEM: system_easylink.c: 94] Get SSID: Xiaomi.Router, Key: stm32f215
[17270][SYSTEM: system_easylink.c: 133] Get user info: 6D 79 20 65 78 74 72 61 23 72 1F A8 C0
[17279][SYSTEM: system_easylink.c: 162] Get auth info: my extra, EasyLink identifier: c0a81f72
[17289][SYSTEM: system_misc.c: 181] connect to Xiaomi.Router....
[25194][SYSTEM: system_misc.c: 71] Station up

```

图 6.2 mDNS 串口 log

运行后，

- (1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)
- (2) 设备进入无线网络配置，首先检测 flash 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网。
- (3) 配网成功后，输出 Station up，如上图 log。
- (4) 刷新设备列表，会发现当前 Mac 地址的 MiCOKit-3165 在线，并可查看设备信息

```

#include "mico.h"
#include "platform_config.h"
#include "StringUtils.h"

#define mdns_log(M, ...) custom_log("mDNS", M, ##__VA_ARGS__)

/* mdns 服务中组播包携带的数据 */
#define PROTOCOL          "com.mxchip.basic"
#define BONJOURNAME       "MiCOKit"
#define BONJOUR_SERVICE   "_easylink._tcp.local."
#define LOCAL_PORT        8080

/* 定义 bonjour 服务器 */
void myBonjour_server( WiFi_Interface interface )

```

```
{  
    char *temp_txt= (char*)malloc(500);  
    char *temp_txt2;  
    IPStatusTypeDef para;  
    mdns_init_t init;  
    //读取网络接口的当前 IP 状态  
    micowlanGetIPStatus(&para, interface);  
    //mDNS 记录数据初始化  
    init.service_name = BONJOUR_SERVICE; /*"_easylink._tcp.local."*/  
  
    /*format*/  
    /*  name#xxxxxx.local. */  
    sprintf( temp_txt, 100, "%s%c%c%c%c%c.local.", BONJOURNAME,  
            para.mac[6],  para.mac[7], \  
            para.mac[8],  para.mac[9], \  
            para.mac[10], para.mac[11] );  
    init.host_name = (char*)__strup(temp_txt);  
  
    /*  name#xxxxxx. */  
    sprintf( temp_txt, 100, "%s%c%c%c%c%c", BONJOURNAME,  
            para.mac[6],  para.mac[7], \  
            para.mac[8],  para.mac[9], \  
            para.mac[10], para.mac[11] );  
    init.instance_name = (char*)__strup(temp_txt);  
  
    init.service_port = LOCAL_PORT;  
  
    /*可以修改此处测试得到不同的数据*/  
    temp_txt2 = __strup_trans_dot(para.mac);  
    sprintf(temp_txt, "MAC=%s.", temp_txt2);  
    free(temp_txt2);  
  
    temp_txt2 = __strup_trans_dot(FIRMWARE_REVISION);  
    sprintf(temp_txt, "%sFirmware Rev=%s.", temp_txt, temp_txt2);  
    free(temp_txt2);  
  
    temp_txt2 = __strup_trans_dot(HARDWARE_REVISION);
```

```
sprintf(temp_txt, "%sHardware Rev=%s.", temp_txt, temp_txt2);
free(temp_txt2);

temp_txt2 = __strdup_trans_dot(MicoGetVer());
sprintf(temp_txt, "%sMICO OS Rev=%s.", temp_txt, temp_txt2);
free(temp_txt2);

temp_txt2 = __strdup_trans_dot(MODEL);
sprintf(temp_txt, "%sModel=%s.", temp_txt, temp_txt2);
free(temp_txt2);

temp_txt2 = __strdup_trans_dot(PROTOCOL);
sprintf(temp_txt, "%sProtocol=%s.", temp_txt, temp_txt2);
free(temp_txt2);

temp_txt2 = __strdup_trans_dot(MANUFACTURER);
sprintf(temp_txt, "%sManufacturer=%s.", temp_txt, temp_txt2);
free(temp_txt2);

/*printf*/
mdns_log("TXT RECORD=%s",temp_txt);

/*
TXT RECORD=MAC=c89346918152.
Firmware Rev=MICO_BASE_1_0.
Hardware Rev=MK3288_1.
MICO OS Rev=10880002/.032.
Model=MiCOKit-3288.
Protocol=com/.mxchip/.basic.
Manufacturer=MXCHIP Inc/..*/
/* txt_record 中写入数据*/
init.txt_record = (char *)__strdup(temp_txt);

mdns_add_record(init, interface, 1500 );
/*释放内存*/
free(init.host_name);
free(init.instance_name);
```

```
free(init.txt_record);

if(temp_txt)
    free(temp_txt);
}

int application_start( void )
{
    OSStatus err = kNoErr;

    /* 根据 mico_config.h 启动 MiCO 系统功能*/
    err = mico_system_init( mico_system_context_init( 0 ) );
    require_noerr( err, exit );

    /* 注册 mDNS 记录，启用 Bonjour 服务 */
    my_bonjour_server( Station );

exit:
    if( err != kNoErr )
        mdns_log("Thread exit with err: %d", err);
    mico_rtos_delete_thread( NULL );
    return err;
}
```

6.3 gagent 连接

本 Demo 主要实现：将 MiCO 设备连接到 GAgent 云服务器的功能。开发中，说明暂不提供。

6.4 sntp_clinet 网络时间获取

6.4.1 Sntp 概述

在一些特定的场景中，经常需要整个网络中的计算机保持时间同步。科学家发明了一种叫做 NTP 的网络时间协议。网络时间协议是一种在网络计算机上同步计算机时间的的协议，它具有高度的精确性（能精确到几十毫秒），但是算法非常复杂。实际上，在很多应用场景中，并不需要这么高的精确度，通常只要达到秒级的精确度就足够了。于是，科学家在 NTP 的基础上推出了 SNTP（简单网络时间协议，*Simple Network Time Protocol*）。SNTP 大大简化了 NTP 协议，同时也能保证时间达到一定的精确度。在实际应用中，SNTP 协议主要被用来同步因特网上计算机的时间。

SNTP 协议采用客户端/服务器的工作方式，可以采用单播（点对点）或者广播（一点对多点）模式操作。SNTP 服务器通过接收 GPS 信号或自带的原子钟作为系统的时间基准。单播模式下，SNTP 客户端能够通过定期访问 SNTP 服务器获得准确的时间信息，用于调整客户端自身所在系统的时间，达到同步时间的目的。广播模式下，SNTP 服务器周期性地发送消息给指定的 IP [广播地址](#)或者 IP [多播地址](#)。SNTP 客户端通过监

听这些地址来获得时间信息。本 Demo 采用的是单播模式，即 SNTP 客户端定期访问 SNTP 服务器获得准确时间信息。

6.4.2 sntp_client.c 功能描述

本例实运行 log 如图 6.3：

```

10][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
639][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
1048][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
1075][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
1655][SYSTEM: system_misc.c: 222] Free memory 66864 bytes
1661][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
1662][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
1664][SYSTEM: system_misc.c: 233] WiFi driver version w10: Nov 7 2014 16:03:45 version 5.90.230.12 FWI, mac D0:B4
1684][SYSTEM: system_system_init.c: 61] Empty configuration. Starting configuration mode...
1594][SYSTEM: config_server.c: 148] Config server established at port: 8000, fd: 1
1894][SNTP: sntp.c: 252] Getting NTP time...
1898][SNTP: sntp.c: 270] Resolving SNTP server address...
1904][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:49:45.044032
1911][SYSTEM: system_easylk_combo.c: 262] Start easylk combo mode
11911][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:49:55.051032
15458][SYSTEM: system_easylk.c: 94] Get SSID: Xiaomi_Router, Key: stm32f215
15466][SYSTEM: system_easylk.c: 133] Get user info: 6d 79 20 65 78 74 72 61 23 72 1F A8 C0
15475][SYSTEM: system_easylk.c: 162] Get auth info: my_extra, EasyLink identifier: c0a81f72
15484][SYSTEM: system_misc.c: 181] connect to Xiaomi_Router...
21918][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:50:05.058032
22484][SNTP: sntp.c: 274] SNTP server address can not be resolved
29720][SYSTEM: system_misc.c: 71] Station up
31925][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:50:15.065032
41932][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:50:25.006496
51939][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:50:35.013496
61904][SNTP: sntp.c: 252] Getting NTP time...
61908][SNTP: sntp.c: 270] Resolving SNTP server address...
61946][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:50:45.020496
64653][SNTP: sntp.c: 279] SNTP server address: pool.ntp.org, host ip: 202.118.1.81
67661][MICO:sntp:c:int sntp_get_time(struct in_addr const *, struct <unnamed> *): 214] **ASSERT**
67670][SNTP: sntp.c: 235] Exit: NTP Client exit with err = -6722
71933][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:50:55.027496
72676][SNTP: sntp.c: 291] failed, trying again...
72680][SNTP: sntp.c: 270] Resolving SNTP server address...
72689][SNTP: sntp.c: 279] SNTP server address: pool.ntp.org, host ip: 110.75.186.249
72710][SNTP: sntp.c: 232] Time Synchronized, Fri Sep 23 08:50:59 2016
72717][SNTP: sntp.c: 285] success
72720][SNTP DEMO: sntp_client.c: 72] sntp_time_synced: 2016-09-23T08:50:59.003784
72728][SNTP: sntp.c: 314] Current time is: 2016-09-23T08:50:59.011784
81960][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:51:08.047176
91967][SNTP DEMO: sntp_client.c: 114] Current time: 2016-09-23T08:51:18.054176

```

等待用户
Easylk 配网

配网成功

间隔 10s 从
SNTP 服务器
获取一次当
前网络时间

图 6.3 sntp_client.c 串口 log 信息

6.5 udp_unicast

6.5.1 UDP 单播概述

UDP 是一种无连接的传输层协议，它主要用于不要求分组顺序到达的传输中，分组传输顺序的检查与排序由应用层完成，提供面向事务的简单不可靠信息传送服务。UDP 协议基本是 IP 协议与上层协议接口。UDP Socket 编程模型：

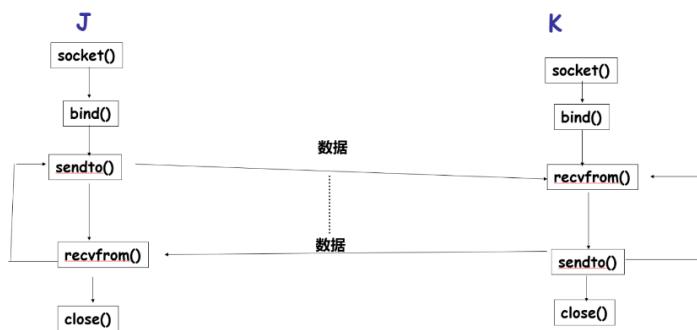


图 6.4 UDP 传输机制

- 什么是 UDP 单播？

在客户端与服务器之间，需要建立一个单独的数据通道，从一台服务器送出的每个数据包只能传送给一

个客户机，这种传送方式称为单播。

6.5.2 udp_unicast.c 功能说明

本例实现设备的 udp 单播服务，即与指定 IP 地址和端口号的连接进行数据通信。运行 log 如下：

```
[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1079][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1106][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1680][SYSTEM: system_misc.c: 222] Free memory 66840 bytes
[1687][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1692][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1698][SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Nov 7 2014 16:03:45 version 5.90.230.12 Fv
[1709][SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode...
[1720][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1920][SYSTEM: system_easylink.c: 262] Start easylink combo mode ← 启动Easylink配网
[1926][UDP: udp_unicast.c: 80] Open local UDP port 20000 ← 本地端口号20000
[29598][SYSTEM: system_easylink.c: 94] Get SSID: Xiaomi.Router, Key: 5tm32F215
[29606][SYSTEM: system_easylink.c: 133] Get user info: 6D 79 20 65 78 74 72 61 23 72 1F A8 C0
[29615][SYSTEM: system_easylink.c: 162] Get auth info: my.extra, EasyLink identifier: c0a81f72
[29624][SYSTEM: system_misc.c: 181] connect to Xiaomi.Router....
[44815][UDP: udp_unicast.c: 45] WLAN connected, Local ip address: 192.168.31.202 ← 本地IP地址
[44822][SYSTEM: system_misc.c: 71] station up ← 配网成功
[52745][UDP: udp_unicast.c: 97] udp recv from 192.168.31.133:8526, len:32 ← PC端发来的udp单播数据信息
```

图 6.5 UDP 单播串口 log

运行后，

(1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)

(2) 设备进入无线网络配置，首先检测 flas

h 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网。

(3) 配网成功后，输出 Station up，如上图 log。

(4) 配网完成后，设备启动 UDP 单播服务，等待接收数据。

(5) 在 [TCP UDP 测试工具](#) 中创建客户端连接，其中，IP 地址和端口号信息，见图 6.5。

(6) TCP UDP 测试工具发数据给 MiCO 设备，设备收到数据后，又返回给 TCP UDP 测试工具，并且在接收区显示了设备返回数据。具体如图 6.6。

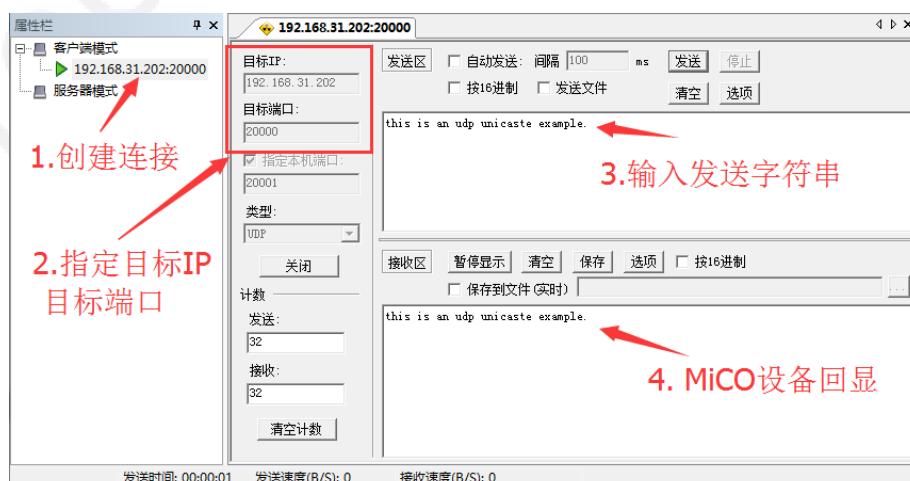


图 6.6 UDP 单播测试

本例代码如下：

```
#include "mico.h"

#define udp_unicast_log(M, ...) custom_log("UDP", M, ##__VA_ARGS__)

//定义本地 UDP 端口号
#define LOCAL_UDP_PORT 20000

/* 定义 wifi 状态改变的用户通知功能回调函数*/
void micoNotify_WifiStatusHandler(WiFiEvent event, void* const inContext)

{

    IPStatusTypedef para;

    switch (event) {

        case NOTIFY_STATION_UP:

            micoWlanGetIPStatus(&para, Station);

            udp_unicast_log( "Wlan connected, Local ip address: %s", para.ip );

            break;

        case NOTIFY_STATION_DOWN:

            break;

    }

}

/*创建 udp socket*/
void udp_unicast_thread(void *arg)

{

    UNUSED_PARAMETER(arg);

    OSStatus err;

    struct sockaddr_t addr;

    fd_set readfds;

    socklen_t addrLen = sizeof(addr);

    int udp_fd = -1 , len;

    char ip_address[16];

    uint8_t *buf = NULL;

    buf = malloc(1024);

    require_action(buf, exit, err = kNoMemoryErr);

    /*建立一个 udp 端口用于接收发送至该端口的数据*/

    udp_fd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );

    require_action( IsValidSocket( udp_fd ), exit, err = kNoResourcesErr );
```

```
addr.s_ip = INADDR_ANY; /*local ip address*/
addr.s_port = LOCAL_UDP_PORT; /*20000*/
err = bind( udp_fd, &addr, sizeof(addr) );
require_noerr( err, exit );

udp_unicast_log("Open local UDP port %d", LOCAL_UDP_PORT);

while(1)
{
    FD_ZERO(&readfds);
    FD_SET(udp_fd, &readfds);

    require_action( select(udp_fd + 1, &readfds, NULL, NULL, NULL) >= 0, exit, err = kConnectionErr );
/*从 udp 读取数据，并返回相同数据*/
    if (FD_ISSET( udp_fd, &readfds ))
    {
        len = recvfrom(udp_fd, buf, 1024, 0, &addr, &addrLen);
        require_action( len >= 0, exit, err = kConnectionErr );

        inet_ntoa( ip_address, addr.s_ip );
        udp_unicast_log( "udp recv from %s:%d, len:%d", ip_address,addr.s_port, len );
        sendto( udp_fd, buf, len, 0, &addr, sizeof(struct sockaddr_t) );
    }
}

exit:
if( err != kNoErr )
    udp_unicast_log("UDP thread exit with err: %d", err);
if( buf != NULL ) free(buf);
mico_rtos_delete_thread(NULL);
}

int application_start( void )
{
    OSStatus err = kNoErr;
/*注册一个 WiFi 状态改变时的用户通知功能函数*/
    err = mico_system_notify_register( mico_notify_WIFI_STATUS_CHANGED, (void *)micoNotify_WifiStatusHandler,
```

```

NULL );

require_noerr( err, exit );

/* 根据 mico_config.h 设置，启动 MiCO 系统功能 */

err = mico_system_init( mico_system_context_init( 0 ) );

require_noerr( err, exit );

/* 创建 udp 单播线程 */

err = mico_rtos_create_thread(NULL, MICO_APPLICATION_PRIORITY, "udp_unicast", udp_unicast_thread, 0x800,
NULL );

require_noerr_string( err, exit, "ERROR: Unable to start the UDP thread." );

exit:

if( err != kNoErr )

    udp_unicast_log("Thread exit with err: %d", err);

mico_rtos_delete_thread( NULL );

return err;

}

```

6.6 udp_broadcast.c

6.6.1 UDP 广播概述

主机之间“一对所有”的通讯模式，网络对其中每一台主机发出的信号都进行无条件复制并转发，所有主机都可以接收到所有信息。这种传送方式称为广播。

6.6.2 udp_broadcast 功能说明

本例实现 UDP 组播服务，MiCO 设备可同时与不同 IP 地址主机的同一端口号进行通讯。运行 log 如下：

```

[2] [Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651] [APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1080] [BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1107] [RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1671] [SYSTEM: system_misc.c: 222] Free memory 66864 bytes
[1677] [SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1683] [SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1688] [SYSTEM: system_misc.c: 233] Wi-Fi driver version w10: Nov 7 2014 16:03:45 version
[1699] [SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode.
[1710] [SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1910] [SYSTEM: system_easylink.c: 262] start easylink combo mode
[1926] [UDP: udp_broadcast.c: 61] Start UDP broadcast mode, local port: 20000, remote por
[1925] [UDP: udp_broadcast.c: 65] broadcast now! ← 开始 UDP 广播
[3920] [UDP: udp_broadcast.c: 65] broadcast now!
[4788] [SYSTEM: system_easylink.c: 94] Get SSID: Xiaomi.Router, Key: stm32f215
[4796] [SYSTEM: system_easylink.c: 133] Get user info: 6D 79 20 65 78 74 72 61 23 72 1F A8
[4804] [SYSTEM: system_easylink.c: 162] Get auth info: my extra, EasyLink identifier: coaf
[4814] [SYSTEM: system_misc.c: 181] connect to Xiaomi.Router.....
[5933] [UDP: udp_broadcast.c: 65] broadcast now!
[7937] [UDP: udp_broadcast.c: 65] broadcast now!
[9941] [UDP: udp_broadcast.c: 65] broadcast now!
[10696] [SYSTEM: system_misc.c: 71] Station up ← 配网成功
[11945] [UDP: udp_broadcast.c: 65] broadcast now!
[13949] [UDP: udp_broadcast.c: 65] broadcast now!

```

图 6.7 UDP 广播 log

运行后：

(1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink

(Easylink 是 MiCO 设备的标准配网协议)

(2) 设备进入无线网络配置, 首先检测 flash 中有无配网参数即 wifi 账号和密码, 若有接入上次配网 wifi。若无, 进入 Easylink 快速配网程序, 等待用户手机 APP Easylink 配网。

(3) 配网成功后, 输出 Station up, 如上图 log。

(4) 配网完成后, 设备启动 UDP 组播服务, 不断的向外部发送特定字符串

(5) 在 PC 端 (必须和 MiCO 设备接入同一 Wi-Fi 网络), 利用 [TCP UDP 测试工具](#), 创建 UDP 连接, 指定目标端口和本机口号, 如图 6.8。

(6) 连接成功创建后, 即可接收设备发来的 UDP 广播数据, 如下图接收区中显示字符串情况。



图 6.8 UDP 广播测试

本例代码如下:

```
#include "MICO.h"

#define udp_broadcast_log(M, ...) custom_log("UDP", M, ##__VA_ARGS__)

#define LOCAL_UDP_PORT 20000
#define REMOTE_UDP_PORT 20001

char* data = "UDP broadcast data";

/*创建 udp socket*/
void udp_broadcast_thread(void *arg)
{
    UNUSED_PARAMETER(arg);

    OSStatus err;
    struct sockaddr_t addr;
    int udp_fd = -1;

    /*创建 UDP 端口, 接收从此端口发出的数据*/
}
```

```
udp_fd = socket( AF_INET, SOCK_DGRAM, IPPROTO_UDP );

require_action( IsValidSocket( udp_fd ), exit, err = kNoResourcesErr );


addr.s_ip = INADDR_ANY;

addr.s_port = LOCAL_UDP_PORT;

err = bind(udp_fd, &addr, sizeof(addr));

require_noerr( err, exit );

udp_broadcast_log("Start UDP broadcast mode, local port: %d, remote port: %d", LOCAL_UDP_PORT,
REMOTE_UDP_PORT);

while(1)

{

    udp_broadcast_log( "broadcast now!" );


    addr.s_ip = INADDR_BROADCAST;

    addr.s_port = REMOTE_UDP_PORT;

    /*接收端口号应设置为: 20000*/

    sendto( udp_fd, data, strlen(data), 0, &addr, sizeof(addr) );


    mico_thread_sleep(2);

}

exit:

if( err != kNoErr )

    udp_broadcast_log("UDP thread exit with err: %d", err);

mico_rtos_delete_thread(NULL);

}

int application_start( void )

{

OSStatus err = kNoErr;

/* Start MiCO system functions according to mico_config.h */

err = mico_system_init( mico_system_context_init( 0 ) );

require_noerr( err, exit );

err = mico_rtos_create_thread(NULL, MICO_APPLICATION_PRIORITY, "udp_broadcast", udp_broadcast_thread, 0x800,
```

```
NULL );  
  
require_noerr_string( err, exit, "ERROR: Unable to start the UDP thread." );  
  
exit:  
  
    if( err != kNoErr )  
        udp_broadcast_log("Thread exit with err: %d", err);  
  
    mico_rtos_delete_thread( NULL );  
  
    return err;  
}
```

主机之间“一对一组”的通讯模式，也就是加入了同一个组的主机可以接受到此组内的所有数据，网络中的交换机和路由器只向有需求者复制并转发其所需数据。这种传送方式称为组播。

6.7 tcp_client

6.7.1 TCP 客户端概述

- 什么是 TCP？

TCP 是面向连接的通信协议，通过三次握手建立连接，提供一种可靠的字节流服务，采用重传肯定机制来实现传输的可靠性。在因特网协议族（Internet protocol suite）中，TCP 层是位于 IP 层之上，应用层之下的中间层。

- TCP Socket 编程模型

Socket(套接字)是一种通讯机制，它包含一整套的调用接口和数据结构的定义，它给应用进程提供了使用如 TCP/UDP 等网络协议进行网络通讯的手段。

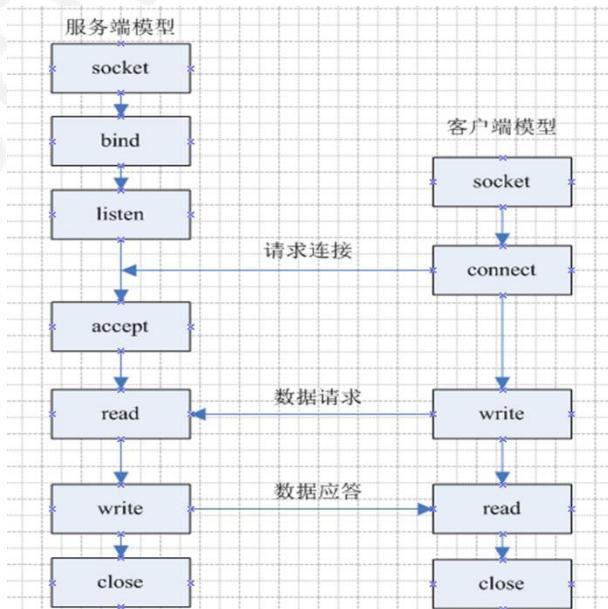


图 6.9 Socket 机制

- TCP 客户端与服务器端有什么区别？

TCP 客户端主动发起连接，服务器在连接前监听。本例 `tcp_client.c` 即为客户端主动发起连接的例子。

6.7.2 `tcp_client.c` 功能说明

本例实现将 MiCO 设备作为 TCP 客户端，连接到指定的 TCP 服务器，并将服务器发来数据回传给 TCP server。运行 Demo 前，需先修改 `tcp_client.c` 文件中代码部分：

```
static char tcp_remote_ip[16] = "192.168.31.133"; /*remote ip address*/
static int tcp_remote_port = 6000; /*remote port*/
```

将远程 IP 地址和远程端口号改为用户测试 PC 的 IP 地址和端口号。运行 log 如下：

图 6.10 TCP Client 串口 log

运行后：

- (1) 设备根据 `mico_config.h` 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)；
- (2) 设备进入无线网络配置，首先检测 flash 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网；
- (3) 配网成功后，输出 Station up，如上图 log；
- (4) 配网完成后，创建一个 TCP 客户端，其要连接的服务器 IP：192.168.31.133，端口号：6000；
- (5) 在 PC 端（必须和 MiCO 设备接入同一 Wi-Fi 网络），利用 [TCP UDP 测试工具](#)，创建 TCP 服务器并启动，等待自动生成连接，如下图；
- (6) 连接建立后，设备端等待接收服务器发送数据；
- (7) 在 TCP UDP 测试工具的发送区输入数据，并发送给设备，可在接收区看到设备回传的数据。



图 6.11 TCP Client 测试

本例程序代码如下：

```
#include "MICO.h"
#include "SocketUtils.h"

#define tcp_client_log(M, ...) custom_log("TCP", M, ##__VA_ARGS__)

static char tcp_remote_ip[16] = "192.168.31.133"; /*remote ip address*/
static int tcp_remote_port = 6000; /*remote port*/

static mico_semaphore_t wait_sem = NULL;

static void micoNotify_WifiStatusHandler( WiFiEvent status, void* const inContext )
{
    switch ( status ) {
        case NOTIFY_STATION_UP:
            mico_rtos_set_semaphore( &wait_sem );
            break;
        case NOTIFY_STATION_DOWN:
            break;
    }
}

/*当客户端连接到网络成功，创建连接*/
void tcp_client_thread( void *arg )
{
    UNUSED_PARAMETER(arg);
}
```

```
OSStatus err;

struct sockaddr_t addr;
struct timeval_t t;
fd_set readfds;
int tcp_fd = -1 , len;
char *buf = NULL;

buf = (char*)malloc( 1024 );
require_action( buf, exit, err = kNoMemoryErr );

tcp_fd = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
require_action(IsValidSocket( tcp_fd ), exit, err = kNoResourcesErr );

addr.s_ip = inet_addr( tcp_remote_ip );
addr.s_port = tcp_remote_port;

tcp_client_log( "Connecting to server: ip=%s port=%d!", tcp_remote_ip,tcp_remote_port );
err = connect( tcp_fd, &addr, sizeof( addr ) );
require_noerr( err, exit );
tcp_client_log( "Connect success!" );

t.tv_sec = 2;
t.tv_usec = 0;

while(1)
{
    FD_ZERO( &readfds );
    FD_SET( tcp_fd, &readfds );

    require_action( select( tcp_fd + 1, &readfds, NULL, NULL, &t ) >= 0, exit, err = kConnectionErr );
    /*接收网络数据，并回传*/
    if( FD_ISSET( tcp_fd, &readfds ) )
    {
        len = recv( tcp_fd, buf, 1024, 0 );
        require_action( len >= 0, exit, err = kConnectionErr );

        if( len == 0){

```

```
tcp_client_log( "TCP Client is disconnected, fd: %d", tcp_fd );
    goto exit;
}

tcp_client_log("Client fd: %d, recv data %d", tcp_fd, len);
len = send( tcp_fd, buf, len, 0 );
tcp_client_log("Client fd: %d, send data %d", tcp_fd, len);
}

exit:
if( err != kNoErr ) tcp_client_log( "TCP client thread exit with err: %d", err );
if( buf != NULL ) free( buf );
SocketClose( &tcp_fd );
mico_rtos_delete_thread( NULL );
}

int application_start( void )
{
OSStatus err = kNoErr;

mico_rtos_init_semaphore( &wait_sem,1 );
/* 注册一个 MiCO 通知的用户功能: wifi 状态改变 */
err = mico_system_notify_register( mico_notify_WIFI_STATUS_CHANGED, (void *)micoNotify_WifiStatusHandler,
NULL );
require_noerr( err, exit );
/* 根据 mico_config.h 设置, 启动 MiCO 系统功能 */
err = mico_system_init( mico_system_context_init( 0 ) );
require_noerr( err, exit );
/* 等待 WiFi 连接成功*/
mico_rtos_get_semaphore( &wait_sem, MICO_WAIT_FOREVER );
tcp_client_log( "wifi connected successful" );
/* 启动 TCP 客户端线程*/
err = mico_rtos_create_thread(NULL, MICO_APPLICATION_PRIORITY, "TCP_client", tcp_client_thread, 0x800,
NULL );
require_noerr_string( err, exit, "ERROR: Unable to start the tcp client thread." );
```

```

exit:

if( wait_sem != NULL)

    mico_rtos_deinit_semaphore( &wait_sem );

mico_rtos_delete_thread( NULL );

return err;

}

```

6.8 tcp_sever

6.8.1 tcp_sever.c 功能说明

本例实现：将 MiCO 设备做为 TCP server，PC 端作为 TCP client 连接 MiCO 设备，并向 TPC server 发送字符，TCP server 将接收到的字符回传给 TCP client。

运行 log 如下：

```

[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1079][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1106][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1683][SYSTEM: system_misc.c: 222] Free memory: 66840 bytes
[1690][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1695][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1701][SYSTEM: system_misc.c: 233] Wi-Fi driver version w10: Nov 7 2014 16:03:45 version 5.90.230.12
[1712][SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode...
[1723][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1923][SYSTEM: system_easylink.c: 262] Start easylink combo mode Easylink配网
[6782][SYSTEM: system_easylink.c: 94] Get SSID: Xiaomi.Router, Key: stm32f219
[6789][SYSTEM: system_easylink.c: 133] Get user info: 6D 79 20 65 78 74 72 61 23 72 1F A8 C0
[6798][SYSTEM: system_easylink.c: 162] Get auth info: my extra, EasyLink identifier: c0a81f72
[6808][SYSTEM: system_misc.c: 181] connect to Xiaomi.Router.....
设备服务器建立
[11758][TCP: tcp_server.c: 46] Server established at ip: 192.168.31.202 port: 20000
[11766][SYSTEM: system_misc.c: 71] Station up 配网成功
[36139][TCP: tcp_server.c: 138] TCP Client 192.168.31.133:8526 connected, fd: 3 socket创建成功
[53621][TCP: tcp_server.c: 88] fd: 3, recv data 30 from client
[53627][TCP: tcp_server.c: 90] fd: 3, send data 30 to client 接收客户端数据并回显

```

图 6.12 TCP Server 串口 log

运行后：

- (1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)；
- (2) 设备进入无线网络配置，首先检测 flash 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网；
- (3) 同时，设备创建一个 TCP 服务器，IP：192.168.31.202，端口号：20000；
- (4) 配网成功后，输出 Station up，如上图 log；
- (5) 在 PC 端（必须和 MiCO 设备接入同一 Wi-Fi 网络），利用 [TCP UDP 测试工具](#)，创建 TCP 客户端并启动，等待自动生成连接，如图 6.13；
- (6) 连接建立后，设备端等待接收服务器发送数据；
- (7) 在 TCP UDP 测试工具的发送区输入数，并发送给设备，可在接收区看到设备回传的数据。

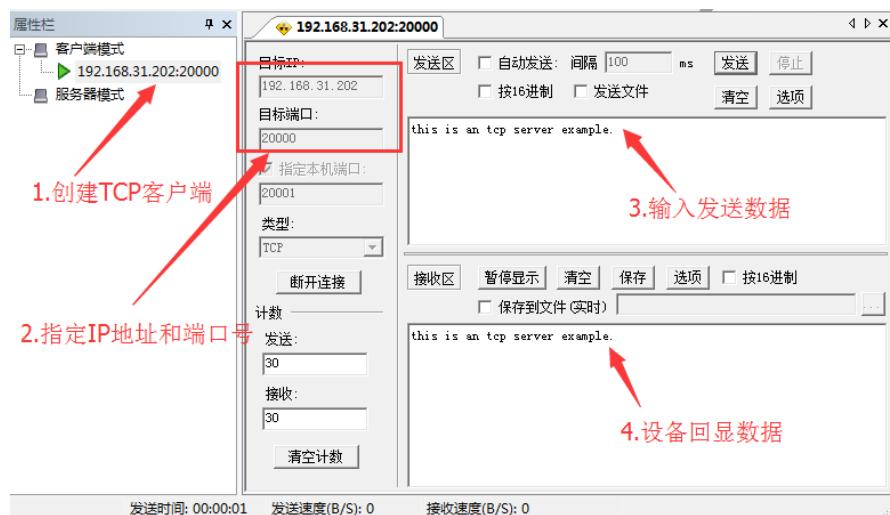


图 6.13 tcp server 测试界面

本例程序代码如下：

```
#include "mico.h"
#include "SocketUtils.h"

#define tcp_server_log(M, ...) custom_log("TCP", M, ##__VA_ARGS__)

#define SERVER_PORT 20000 /*set up a tcp server,port at 20000*/
/*WiFi 状态改变通知*/
void micoNotify_WifiStatusHandler(WiFiEvent event, void* const inContext)
{
    IPStatusTypeDef para;
    switch (event) {
    case NOTIFY_STATION_UP:
        micoWlanGetIPStatus(&para, Station);
        tcp_server_log("Server established at ip: %s port: %d", para.ip, SERVER_PORT);
        break;
    case NOTIFY_STATION_DOWN:
        break;
    }
}

void tcp_client_thread(void *arg)
{
    OSStatus err = kNoErr;
```

```
int fd = (int)arg;
int len = 0;
fd_set readfds;
char *buf = NULL;
struct timeval_t t;

buf=(char*)malloc(1024);
require_action(buf, exit, err = kNoMemoryErr);

t.tv_sec = 5;
t.tv_usec = 0;

while(1)
{
    FD_ZERO(&readfds);
    FD_SET(fd, &readfds);

    require_action( select(1, &readfds, NULL, NULL, &t) >= 0, exit, err = kConnectionErr );

    if( FD_ISSET( fd, &readfds ) ) /*one client has data*/
    {
        len = recv( fd, buf, 1024, 0 );
        require_action( len >= 0, exit, err = kConnectionErr );

        if( len == 0){
            tcp_server_log( "TCP Client is disconnected, fd: %d", fd );
            goto exit;
        }

        tcp_server_log("fd: %d, recv data %d from client", fd, len);
        len = send( fd, buf, len, 0 );
        tcp_server_log("fd: %d, send data %d to client", fd, len);
    }
}

exit:
if( err != kNoErr ) tcp_server_log( "TCP client thread exit with err: %d", err );
if( buf != NULL ) free( buf );
```

```
SocketClose( &fd );

mico_rtos_delete_thread( NULL );

}

/* TCP 服务器监听线程*/
void tcp_server_thread( void *arg )
{
    OSStatus err = kNoErr;

    struct sockaddr_t server_addr,client_addr;
    socklen_t sockaddr_t_size = sizeof( client_addr );
    char client_ip_str[16];
    int tcp_listen_fd = -1, client_fd = -1;
    fd_set readfds;

    tcp_listen_fd = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
    require_action(IsValidSocket( tcp_listen_fd ), exit, err = kNoResourcesErr );

    server_addr.s_ip = INADDR_ANY; /* Accept connection request on all network interface */
    server_addr.s_port = SERVER_PORT; /* Server listen on port: 20000 */
    err = bind( tcp_listen_fd, &server_addr, sizeof( server_addr ) );
    require_noerr( err, exit );

    err = listen( tcp_listen_fd, 0 );
    require_noerr( err, exit );

    while(1)
    {
        FD_ZERO( &readfds );
        FD_SET( tcp_listen_fd, &readfds );

        require( select(1, &readfds, NULL, NULL, NULL) >= 0, exit );

        if(FD_ISSET(tcp_listen_fd, &readfds)){
            client_fd = accept( tcp_listen_fd, &client_addr, &sockaddr_t_size );
            if( IsValidSocket( client_fd ) ) {
                inet_ntoa( client_ip_str, client_addr.s_ip );
                tcp_server_log( "TCP Client %s:%d connected, fd: %d", client_ip_str, client_addr.s_port, client_fd );
            }
        }
    }
}
```

```
    if ( kNoErr != mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY, "TCP Clients",
tcp_client_thread, 0x800, (void *)client_fd ) )
{
    SocketClose( &client_fd );
}
}

}

exit:
if( err != kNoErr ) tcp_server_log( "Server listerner thread exit with err: %d", err );
SocketClose( &tcp_listen_fd );
mico_rtos_delete_thread(NULL );
}

int application_start( void )
{
    OSStatus err = kNoErr;
/*注册一个 MiCO 通知用户功能: WiFi 状态改变*/
err = mico_system_notify_register( mico_notify_WIFI_STATUS_CHANGED, (void *)micoNotify_WifiStatusHandler,
NULL );
require_noerr( err, exit );

/* 根据 mico_config.h 设置, 启动 MiCO 系统功能 */
err = mico_system_init( mico_system_context_init( 0 ) );
require_noerr( err, exit );

/* 启动 TCP 服务器监听线程*/
err = mico_rtos_create_thread( NULL, MICO_APPLICATION_PRIORITY, "TCP_server", tcp_server_thread, 0x800,
NULL );
require_noerr_string( err, exit, "ERROR: Unable to start the tcp server thread." );

exit:
mico_rtos_delete_thread( NULL );
return err;
}
```

7. HTTP 通信

7.1 http_client

7.1.1 HTTP 概述

● 认识 HTTP

现代网络数据交互一般是客户端访问一个网络地址，服务器收到网络申请后返回数据给客户端。数据交互已经形成了规范，这就是著名的 HTTP 协议（超文本传送协议）。超文本传送协议定义了客户端怎样向万维网服务器请求万维网文档，以及服务器怎样把文档传送给浏览器。HTTP 是一个客户端和服务器端请求和应答的标准 TCP。

通常，由 HTTP 客户端发起一个请求，建立一个到服务器指定端口（默认是 80 端口）的 TCP 连接。HTTP 服务器则在那个端口监听客户端发送过来的请求。一旦收到请求，服务器（向客户端）发回一个状态行（比如“HTTP/1.1 200 OK”）和响应的消息，消息的消息体可能是请求的文件、错误消息、或者其它一些信息。HTTP 使用 TCP，而不是 UDP 的原因在于（打开）一个网页必须传送很多数据，而 TCP 协议提供传输控制，按顺序组织数据和错误纠正。

HTTP 协议的主要特点可概括如下：

1. 简单快速：客户向服务器请求服务时，只需传送请求方法和路径。常用方法有 POST 和 GET。每种方法规定了客户与服务器联系的类型不同。由于 HTTP 协议简单，使得 HTTP 服务器的程序规模小，因而通信速度很快。
2. 灵活：HTTP 允许传输任意类型的数据对象。正在传输的类型由 Content-Type 加以标记。
3. 无状态：HTTP 协议是无状态协议。无状态是指协议对于事务处理没有记忆能力。缺少状态意味着如果后续处理需要前面的信息，则它必须重传，这样可能导致每次连接传送的数据量增大。

● HTTP 请求之 POST 和 GET 方法

HTTP GET 和 POST 的区别：

4. HTTP 主要有 POST 和 GET 两种命令模式；上行从客户端到服务端 POST 上传大容量数据，下行从服务端传送数据到客户端，简单的一般都用 GET。
5. POST 是被设计用来向上上传数据的，而 GET 是被设计用来从服务器获取数据的，GET 也能够向服务器传送较少的数据，而 GET 之所以也能传送数据，只是用来设计告诉服务器，你到底需要什么样的数据。
6. POST 与 GET 在 HTTP 中传送的方式不同，GET 的参数是在 HTTP 的头部传送的，而 POST 的数据则是在 HTTP 请求的包体中传送；
7. POST 传输数据时，不需要在 URL 中显示出来，而 GET 方法要在 URL 中显示；所以就传输的安全性来讲，POST 方式比 GET 方式更加安全。
8. GET 方法由于受到 URL 长度的限制，只能传递大约 1024 字节；POST 传输的数据量可以达到 2M。

如图所示 POST 和 GET 的差别。图片表示的意思是：根据火车车次代码获取火车信息。

```
GET /WebServices/TrainTimeWebService.asmx/getStationAndTimeByTrainCode?TrainCode=string&UserID=string HTTP/1.1
Host: webservice.webxml.com.cn
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length

<?xml version="1.0" encoding="utf-8"?>
<ArrayOfString xmlns="http://WebXml.com.cn/">
  <string>string</string>
  <string>string</string>
</ArrayOfString>
```

图 7.1 HTTP 客户端 GET 方法

```
POST /WebServices/TrainTimeWebService.asmx/getStationAndTimeByTrainCode HTTP/1.1
Host: webservice.webxml.com.cn
Content-Type: application/x-www-form-urlencoded
Content-Length: length
```

```
TrainCode=string&UserID=string
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Length: length
```

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfString xmlns="http://WebXml.com.cn/">
  <string>string</string>
  <string>string</string>
</ArrayOfString>
```

图 7.2 HTTP 客户端 POST 方法

7.1.2 http_client.c 功能说明

本例实现 MiCO 设备作为客户端访问某网站信息的功能，运行 log 如下：

```
# [2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[650][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1079][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1106][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1683][SYSTEM: system_misc.c: 222] Free memory 60040 bytes
[1689][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1695][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1700][SYSTEM: system_misc.c: 233] Wi-Fi driven version w10: Nov 7 2014 16:03:45 version 5.90.230.12 FWI, mac D0:BA:E4:07:8F:F0
[1712][SYSTEM: mico_system_init.c: 84] Available configuration. starting wi-Fi connection...
[1720][SYSTEM: system_misc.c: 209] Connect to Xiaomi_Router....
[1732][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1997][SYSTEM: system_misc.c: 71] Station up
[2001][HTTP: http_client.c: 94] WiFi connected successful
[2420][HTTP: http_client.c: 123] HTTP server address: www.baidu.com, host ip: 115.239.210.27
[2459][HTTPUtils: HTTPUtils.c:1256] Method:
[2464][HTTPUtils: HTTPUtils.c:1258] URL:
[2468][HTTPUtils: HTTPUtils.c:1260] Protocol: HTTP/1.1
[2473][HTTPUtils: HTTPUtils.c:1261] Status code: 200
[2478][HTTPUtils: HTTPUtils.c:1262] ChannelID: 0
[2482][HTTPUtils: HTTPUtils.c:1263] Content length: 14613
[2487][HTTPUtils: HTTPUtils.c:1264] Persistent: NO
[3566][HTTP: http_client.c: 154] Content Data: <!DOCTYPE html><!--STATUS OK-->
<html>
<head>
  <meta http-equiv="content-type" content="text/html; charset=utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=Edge">
  <link rel="dns-prefetch" href="http://s1.bdstatic.com"/>
  <link rel="dns-prefetch" href="http://t1.baidu.com"/>
  <link rel="dns-prefetch" href="http://t2.baidu.com"/>
  <link rel="dns-prefetch" href="http://t3.baidu.com"/>
  <link rel="dns-prefetch" href="http://t10.baidu.com"/>
  <link rel="dns-prefetch" href="http://t11.baidu.com"/>
  <link rel="dns-prefetch" href="http://t12.baidu.com"/>
  <link rel="dns-prefetch" href="http://bi.bdstatic.com"/>
  <title>百度一下，你就知道</title>
  <link href="http://s1.bdstatic.com/r/www/cache/static/home/css/index.css" rel="stylesheet" type="text/css" />
  <!--[if lt IE 8]><style index="index" >#content{height:480px\9}#{top:260px\9}</style>![endif]-->
  <!--[if IE 8]><style index="index" >#u1 a.mnav, #u1 a.mnav:visited{font-family:simsun}</style>![endif]-->
  <script>var hashMatch = document.location.href.match(/#(.+?)(\?.+?)?/);if (hashMatch && hashMatch[0] && hashMatch[1]) {document.documentElement.className = hashMatch[1];}function h(obj){obj.style.behavior='url(#default#homepage)';var a = obj.setHomePage('http://www.baidu.com/');}</script>
```

获取指定IP地址的数据信息

图 7.3 HTTP Client 串口 log

运行过程：

- (1) 设备根据 mico_config.h 的配置，启动 MiCO 系统。其中，使能无线连接功能，选择模式为 Easylink (Easylink 是 MiCO 设备的标准配网协议)；

(2) 设备进入无线网络配置，首先检测 flash 中有无配网参数即 wifi 账号和密码，若有接入上次配网 wifi。若无，进入 Easylink 快速配网程序，等待用户手机 APP Easylink 配网；

(3) 配网成功后，从服务器端（设备指定 IP 地址）向客户端（MiCO 设备）传送数据。

本例程序代码如下：

```
#include "mico.h"
#include "HTTPUtils.h"
#include "SocketUtils.h"
#include "StringUtils.h"

#define http_client_log(M, ...) custom_log("HTTP", M, ##__VA_ARGS__)

static OSStatus onReceivedData( struct _HTTPHeader_t * httpHeader,
                               uint32_t pos,
                               uint8_t *data,
                               size_t len,
                               void * userContext );
static void onClearData( struct _HTTPHeader_t * inHeader, void * inUserContext );

static mico_semaphore_t wait_sem = NULL;

typedef struct _http_context_t
{
    char *content;
    uint64_t content_length;
} http_context_t;

void simple_http_get( char* host, char* query );
void simple_https_get( char* host, char* query );

#define SIMPLE_GET_REQUEST \
    "GET / HTTP/1.1\r\n" \
    "Host: www.baidu.com\r\n" \
    "Connection: close\r\n" \
    "\r\n"

static void micoNotify_WifiStatusHandler( WiFiEvent status, void* const inContext )
```

```
{  
    UNUSED_PARAMETER( inContext );  
  
    switch ( status )  
    {  
  
        case NOTIFY_STATION_UP:  
            mico_rtos_set_semaphore( &wait_sem );  
            break;  
  
        case NOTIFY_STATION_DOWN:  
        case NOTIFY_AP_UP:  
        case NOTIFY_AP_DOWN:  
            break;  
    }  
}  
  
int application_start( void )  
{  
    OSStatus err = kNoErr;  
  
    mico_rtos_init_semaphore( &wait_sem, 1 );  
  
    /*为 MiCO 通知注册用户函数: WiFi 状态改变 */  
    err = mico_system_notify_register( mico_notify_WIFI_STATUS_CHANGED,  
                                       (void *) micoNotify_WifiStatusHandler, NULL );  
    require_noerr( err, exit );  
  
    /* 根据 mico_config.h 配置信息启动 MiCO 系统功能*/  
    err = mico_system_init( mico_system_context_init( 0 ) );  
    require_noerr( err, exit );  
  
    /* 等待无线连接*/  
    mico_rtos_get_semaphore( &wait_sem, MICO_WAIT_FOREVER );  
    http_client_log( "wifi connected successful" );  
  
    /* 从服务器读取 http 数据*/  
    simple_http_get( "www.baidu.com", SIMPLE_GET_REQUEST );  
    simple_https_get( "www.baidu.com", SIMPLE_GET_REQUEST );
```

```
exit:  
mico_rtos_delete_thread( NULL );  
return err;  
}  
  
void simple_http_get( char* host, char* query )  
{  
    OSStatus err;  
    int client_fd = -1;  
    fd_set readfds;  
    char ipstr[16];  
    struct sockaddr_in addr;  
    HTTPHeader_t *httpHeader = NULL;  
    http_context_t context = { NULL, 0 };  
    struct hostent* hostent_content = NULL;  
    char **pptr = NULL;  
    struct in_addr in_addr;  
  
    hostent_content = gethostbyname( host );  
    require_action_quiet( hostent_content != NULL, exit, err = kNotFoundErr );  
    pptr=hostent_content->h_addr_list;  
    in_addr.s_addr = *(uint32_t *)(*pptr);  
    strcpy( ipstr, inet_ntoa(in_addr));  
    http_client_log("HTTP server address: %s, host ip: %s", host, ipstr);  
  
    /*HTTPHeaderCreateWithCallback set some callback functions */  
    httpHeader = HTTPHeaderCreateWithCallback( 1024, onReceivedData, onClearData, &context );  
    require_action( httpHeader, exit, err = kNoMemoryErr );  
  
    client_fd = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );  
    addr.sin_family = AF_INET;  
    addr.sin_addr = in_addr;  
    addr.sin_port = htons(80);  
    err = connect( client_fd, (struct sockaddr *)&addr, sizeof(addr) );  
    require_noerr_string( err, exit, "connect http server failed" );  
  
    /* 发送 HTTP 请求 */
```

```
send( client_fd, query, strlen( query ), 0 );

FD_ZERO( &readfds );
FD_SET( client_fd, &readfds );

select( client_fd + 1, &readfds, NULL, NULL, NULL );
if ( FD_ISSET( client_fd, &readfds ) )
{
    /*解析头*/
    err = SocketReadHTTPHeader( client_fd, httpHeader );
    switch ( err )
    {
        case kNoErr:
            PrintHTTPHeader( httpHeader );
            err = SocketReadHTTPBody( client_fd, httpHeader );/*get body data*/
            require_noerr( err, exit );
            /*获取数据并打印*/
            http_client_log( "Content Data: %s", context.content );
            break;
        case EWOULDBLOCK:
            case kNoSpaceErr:
            case kConnectionErr:
            default:
                http_client_log("ERROR: HTTP Header parse error: %d", err);
                break;
    }
}

exit:
http_client_log( "Exit: Client exit with err = %d, fd:%d", err, client_fd );
SocketClose( &client_fd );
HTTPHeaderDestory( &httpHeader );
}

void simple_https_get( char* host, char* query )
{
    OSStatus err;
```

```
int client_fd = -1;
int ssl_errno = 0;
mico_ssl_t client_ssl = NULL;
fd_set readfds;
char ipstr[16];
char name[20];
struct sockaddr_in addr;
HTTPHeader_t *httpHeader = NULL;
http_context_t context = { NULL, 0 };
struct hostent* hostent_content = NULL;
char **pptr = NULL;
struct in_addr in_addr;

hostent_content = gethostbyname( host );
require_action_quiet( hostent_content != NULL, exit, err = kNotFoundErr);
pptr = hostent_content->h_addr_list;
strcpy(name,hostent_content->h_name);
http_client_log("name is %s",name);
in_addr.s_addr = *(uint32_t *)(*pptr);
strcpy( ipstr, inet_ntoa(in_addr));
http_client_log("HTTP server address: host:%s, ip: %s", host, ipstr);

/*HTTPHeaderCreateWithCallback 设置回调函数 */
httpHeader = HTTPHeaderCreateWithCallback( 1024, onReceivedData, onClearData, &context );
require_action( httpHeader, exit, err = kNoMemoryErr );

client_fd = socket( AF_INET, SOCK_STREAM, IPPROTO_TCP );
addr.sin_family = AF_INET;
addr.sin_addr = in_addr;
addr.sin_port = htons(443);
err = connect( client_fd, ( struct sockaddr * )&addr, sizeof(addr) );
require_noerr_string( err, exit, "connect http server failed" );

client_ssl = ssl_connect( client_fd, 0, NULL, &ssl_errno );
require_string( client_ssl != NULL, exit, "ERROR: ssl disconnect" );

/* 发送 HTTP 请求 */
```

```
ssl_send( client_ssl, query, strlen( query ) );\n\nFD_ZERO( &readfds );\nFD_SET( client_fd, &readfds );\n\nselect( client_fd + 1, &readfds, NULL, NULL, NULL );\nif ( FD_ISSET( client_fd, &readfds ) )\n{\n    /*解析头*/\n\n    err = SocketReadHTTPSHeader( client_ssl, httpHeader );\n\n    switch ( err )\n    {\n        case kNoErr:\n\n            PrintHTTPHeader( httpHeader );\n\n            err = SocketReadHTTPSBBody( client_ssl, httpHeader );/*get body data*/\n\n            require_noerr( err, exit );\n\n            /*获取数据并打印*/\n\n            http_client_log( "Content Data: %s", context.content );\n\n            break;\n\n        case EWOULDBLOCK:\n\n            case kNoSpaceErr:\n\n            case kConnectionErr:\n\n            default:\n\n                http_client_log("ERROR: HTTP Header parse error: %d", err);\n\n                break;\n\n    }\n}\n\nexit:\n\nhttp_client_log( "Exit: Client exit with err = %d, fd:%d", err, client_fd );\nif ( client_ssl ) ssl_close( client_ssl );\nSocketClose( &client_fd );\nHTTPHeaderDestory( &httpHeader );\n}\n\n/* 一个请求可能会收到多个答复 */\nstatic OSStatus onReceivedData( struct _HTTPHeader_t * inHeader, uint32_t inPos, uint8_t * inData,
```

```
        size_t inLen, void * inUserContext )  
{  
  
    OSStatus err = kNoErr;  
  
    http_context_t *context = inUserContext;  
  
    if ( inHeader->chunkedData == false )  
    { //Extra data with a content length value  
  
        if ( inPos == 0 && context->content == NULL )  
        {  
  
            context->content = calloc( inHeader->contentLength + 1, sizeof(uint8_t) );  
            require_action( context->content, exit, err = kNoMemoryErr );  
            context->content_length = inHeader->contentLength;  
  
        }  
  
        memcpy( context->content + inPos, inData, inLen );  
    } else  
    { //额外的数据使用一块数据协议  
  
        http_client_log("This is a chunked data, %d", inLen);  
  
        if ( inPos == 0 )  
        {  
  
            context->content = calloc( inHeader->contentLength + 1, sizeof(uint8_t) );  
            require_action( context->content, exit, err = kNoMemoryErr );  
            context->content_length = inHeader->contentLength;  
        } else  
        {  
  
            context->content_length += inLen;  
            context->content = realloc( context->content, context->content_length + 1 );  
            require_action( context->content, exit, err = kNoMemoryErr );  
        }  
  
        memcpy( context->content + inPos, inData, inLen );  
    }  
  
    exit:  
  
    return err;  
}  
  
/* 当调用 HTTPHeaderClear 时, 被调用 */  
  
static void onClearData( struct _HTTPHeader_t * inHeader, void * inUserContext )
```

```
{
    UNUSED_PARAMETER( inHeader );

    http_context_t *context = inUserContext;

    if ( context->content )

    {
        free( context->content );

        context->content = NULL;
    }
}
```

7.2 http_sever.c

7.2.1 http_sever.c 功能说明

本例实现：MiCO 设备作为 Softap 入网，启动 HTTP 服务器，PC 端接入该 AP，然后访问指定网址，在网页中为设备进行配网，也可用手机配网。运行 log 如下：

```
[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1080][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1107][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1686][SYSTEM: system_misc.c: 222] Free memory 66360 bytes
[1692][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1698][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1703][SYSTEM: system_misc.c: 233] Wi-Fi driver version w10: Nov 7 2014 16:03:45 version 5.90.230.12
[1715][SYSTEM: mico_system_init.c: 61] Empty configuration. Starting configuration mode...
[1726][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1836][SYSTEM: system_easylink.c: 309] Establish soft ap: EasyLink_078FF0.....
[1839][HTTPServer: http_server.c: 41] http server done!
[1933][HTTPServer: http_server.c: 152] initializing web-services
[1940][SYSTEM: system_misc.c: 79] WAP established → 建立AP, 等待用户网页配网
```

图 7.4 http server 串口 log

1、用电脑或手机连接模块产生的热点：Easylink_078FF0，在浏览器中输入模块的 IP 地址：10.10.10.1，出现如下配置界面：



图 7.5 PC 端配网页面

2、输入要连接的路由器名称和密码，点击下一步，模块收到配置信息后，返回配置成功或失败页面，如果配置成功，返回如下页面



图 7.6 网页配网设置成功

3、模块重新启动，连接所设置的路由器。串口打印：

```
[2][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[651][APDS9930: APDS9930.c: 168] APDS9930_ERROR: no i2c device found!
[1080][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[1107][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1682][SYSTEM: system_misc.c: 222] Free memory 66360 bytes
[1688][SYSTEM: system_misc.c: 228] Kernel version: 31621002.050
[1694][SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1699][SYSTEM: system_misc.c: 233] Wi-Fi driver version wl0: Nov 7 2014 16:03:45 version 5.90.230.12
[1711][SYSTEM: mico_system_init.c: 84] Available configuration. Starting Wi-Fi connection...
[1719][SYSTEM: system_misc.c: 209] Connect to MX_00.....
[1730][SYSTEM: config_server.c: 148] Config Server established at port: 8000, fd: 1
[1929][httpserver: http_server.c: 45] http server Demo!
[1934][apphttpd: app_httpd.c: 152] initializing web-services
[7263][SYSTEM: system_misc.c: 71] Station up
```

配网成功

图 7.7 模块 Station 模式成功，Station up

本例程序代码如下：

```
Http_server.c
#include "MICO.h"
#include "app_httpd.h"

#define http_server_log(format, ...) custom_log("httpserver", format, ##__VA_ARGS__)

int application_start( void )
{
    /* Start MiCO system functions according to mico_config.h*/
    mico_system_init( mico_system_context_init( 0 ) );
    /* Output on debug serial port */
    http_server_log( "http server Demo!" );
    /* start http server thread */
    app_httpd_start();
    mico_rtos_delete_thread( NULL );
    return kNoErr;
}

app_httpd.c
#include <httpd.h>
#include <http_parse.h>
#include <http-strings.h>

#include "MICO.h"
#include "httpd_priv.h"
#include "app_httpd.h"

#define app_httpd_log(M, ...) custom_log("apphttpd", M, ##__VA_ARGS__)
#define HTTPD_HDR_DEFORT(HTTPD_HDR_ADD_SERVER|HTTPD_HDR_ADD_CONN_CLOSE|HTTPD_HDR_ADD_PRAGMA_NO_CACHE)

static bool is_http_init;
static bool is_handlers_registered;
```

```
static int g_app_handlers_no;
struct httpd_wsgi_call g_app_handlers[];

//发送设置页面

static int web_send_wifisetting_page(httpd_request_t *req)
{
    OSStatus err = kNoErr;

    //发送 http 头部
    err = httpd_send_all_header(req, HTTP_RES_200, sizeof(wifisetting), HTTP_CONTENT_HTML_STR);
    require_noerr_action( err, exit, app_httpd_log("ERROR: Unable to send http wifisetting headers.") );

    //发送 http 实部
    err = httpd_send_body(req->sock, wifisetting, sizeof(wifisetting));
    require_noerr_action( err, exit, app_httpd_log("ERROR: Unable to send http wifisetting body.") );
exit:
    return err;
}

//接收配置数据，并且返回结果

static int web_send_result_page(httpd_request_t *req)
{
    OSStatus err = kNoErr;
    bool para_succ = false;
    int buf_size = 512;
    char *buf;
    char value_ssid[maxSsidLen];
    char value_pass[maxKeyLen];
    mico_Context_t* context = NULL;
    context = mico_system_context_get();
    buf = malloc(buf_size);

    //获取配置参数
    err = httpd_get_data(req, buf, buf_size);
    require_noerr( err, Save_Out );
    //解析参数
    err = httpd_get_tag_from_post_data(buf, "SSID", value_ssid, maxSsidLen);
    require_noerr( err, Save_Out );

    if(!strncpy(value_ssid, "\0", 1))
        goto Save_Out;
    //获取路由器的名称
}
```

```
strncpy(context->flashContentInRam.micoSystemConfig.ssid, value_ssid, maxSsidLen);

err = httpd_get_tag_from_post_data(buf, "PASS", value_pass, maxKeyLen);
require_noerr( err, Save_Out );

//获取路由器的密码
strncpy(context->flashContentInRam.micoSystemConfig.key, value_pass, maxKeyLen);
strncpy(context->flashContentInRam.micoSystemConfig.user_key, value_pass, maxKeyLen);

context->flashContentInRam.micoSystemConfig.keyLength=strlen(context->flashContentInRam.micoSystemConfig.key);

context->flashContentInRam.micoSystemConfig.user_keyLength=strlen(context->flashContentInRam.micoSystemConfig.key);

//将通道设置为自动
context->flashContentInRam.micoSystemConfig.channel = 0;
memset(context->flashContentInRam.micoSystemConfig.bssid, 0x0, 6); //将 BSSID 设为 0
context->flashContentInRam.micoSystemConfig.security = SECURITY_TYPE_AUTO; //将加密方式设为自动
context->flashContentInRam.micoSystemConfig.dhcpEnable = true; //开启 DHCP

para_succ = true;

Save_Out:

if(para_succ == true)
{
    err= httpd_send_all_header(req, HTTP_RES_200, sizeof(wifisuccess), HTTP_CONTENT_HTML_STR);
    require_noerr_action( err, exit, app_httpd_log("ERROR: Unable to send http wifisuccess headers."));

    err = httpd_send_body(req->sock, wifisuccess, sizeof(wifisuccess));
    require_noerr_action( err, exit, app_httpd_log("ERROR: Unable to send http wifisuccess body."));

    context->flashContentInRam.micoSystemConfig.configured = allConfigured;
    //更新参数
    mico_system_context_update(context);
    //重新启动模块
    mico_system_power_perform( context, eState_Software_Reset );
}

else
{
```

```
err = httpd_send_all_header(req, HTTP_RES_200, sizeof(wififail), HTTP_CONTENT_HTML_STR);
require_noerr_action( err, exit, app_httpd_log("ERROR: Unable to send http wififail headers.") );

err = httpd_send_body(req->sock, wififail, sizeof(wififail));
require_noerr_action( err, exit, app_httpd_log("ERROR: Unable to send http wififail body.") );
}

exit:
if(buf) free(buf);
return err;
}

//注册网关

struct httpd_wsgi_call g_app_handlers[] = {
{"/", HTTPD_HDR_DEFORT, 0, web_send_wifisetting_page, NULL, NULL, NULL},
{"/result.htm", HTTPD_HDR_DEFORT, 0, NULL, web_send_result_page, NULL, NULL},
{"/setting.htm", HTTPD_HDR_DEFORT, 0, web_send_wifisetting_page, NULL, NULL, NULL},
};

int g_app_handlers_no = sizeof(g_app_handlers)/sizeof(struct httpd_wsgi_call);

static void app_http_register_handlers()
{
    int rc;
    rc = httpd_register_wsgi_handlers(g_app_handlers, g_app_handlers_no);
    if (rc) {
        app_httpd_log("failed to register test web handler");
    }
}

static int _app_httpd_start()
{
    OSStatus err = kNoErr;
    app_httpd_log("initializing web-services");

    /*Initialize HTTPD*/
    if(is_http_init == false) {
        err = httpd_init();
    }
}
```

```
require_noerr_action( err, exit, app_httpd_log("failed to initialize httpd") );
is_http_init = true;
}

/*Start http thread*/
err = httpd_start();
if(err != kNoErr) {
    app_httpd_log("failed to start httpd thread");
    httpd_shutdown();
}
exit:
return err;
}

//开启 http server
int app_httpd_start( void )
{
OSStatus err = kNoErr;

err = _app_httpd_start();
require_noerr( err, exit );

if (is_handlers_registered == false) {
    app_http_register_handlers();
    is_handlers_registered = true;
}

exit:
return err;
}

//停止 http server
int app_httpd_stop()
{
OSStatus err = kNoErr;

/* HTTPD and services */
app_httpd_log("stopping down httpd");
err = httpd_stop();
```

```
require_noerr_action( err, exit, app_httpd_log("failed to halt httpd") );  
  
exit:  
    return err;  
}
```

8. JSON 解析

8.1 JSON 概述

8.1.1 什么是 JSON

JSON(JavaScript Object Notation), JavaScript 对象表示法, 是一种轻量级数据交换格式, 可以用来标记数据, 定义数据类型, 是一种需要用户自行定义自我描述的语言。

1. JSON 的简单使得 Windows、MAC OS、Linux 等不同平台下的应用程序之间可以很轻松地进行信息交互。
2. JSON 设计的宗旨是用来结构化传输和存储数据, JSON 格式的数据非常适合 web 传输。
3. JSON 使用 JavaScript 语法来描述数据对象, 但是 JSON 仍然独立于语言和平台。JSON 解析器和 JSON 库支持许多不同的编程语言。
4. JSON 比 XML 更小、更快、更易解析, 作为一种轻量级的数据交换格式, JSON 正在逐步取代 XML, 成为网络数据的通用格式。

8.1.2 JSON 语言规则

JSON 语法是 JavaScript 对象表示法语法的子集。

1. 数据在名称/值对中
2. 数据由逗号分隔
3. 大括号保存对象, 可嵌套
4. 中括号保存数组, 可嵌套

8.1.3 认识 JSON

```
{  
  "employees": [  
    { "firstName": "Bill" , "lastName": "Gates" },  
    { "firstName": "George" , "lastName": "Bush" },  
    { "firstName": "Thomas" , "lastName": "Carter" }  
  ]  
}
```

此 JSON 格式数据表示的意思是: (请牢记大括号是对象, 中括号是数组)

这是一个 JSON 对象, 这个对象里面表述的是员工的信息, 总共有 3 个员工:

员工 1: 姓 Gates 名 Bill,

员工 2: 姓 Bush 名 George

员工 3: 姓 Carter 名 Thomas

8.2 json_op.c 功能说明

8.2.1 JSON 对象的组装与解析

在 MiCO 实时操作系统中加入了开源的 JSON-C 库，借助 JSON-C 库提供的函数，开发者可以很轻松地组装自己想要的 JSON 格式数据，也可以很轻松地解析 JSON 格式的数据。

本例实现组装 JSON 格式对象，并解析，输出，Demo 运行 log 如下：



```
[1][JSON: json_op.c: 59] { "device_info": { "Hardware": "MiCOKit3288", "RGBSwitch": false, "RGBHues": 0, "RGBSaturation": 100, "RGBBrightness": 100 } }
[15][JSON: json_op.c: 79] rgb_sw=0
[18][JSON: json_op.c: 83] rgb_hue=0
[22][JSON: json_op.c: 87] rgb_sat=100
[25][JSON: json_op.c: 91] rgb_bri=100
[29][JSON: json_op.c: 99] control_rgb led now
```

图 8.1 JSON 格式解析串口 log

程序代码如下：

```
/*程序清单：JSON 对象的组装和解析，对象如下：*/
{"device_info":
  {
    "Hardware": "MiCOKit3288",
    "RGBSwitch": false,
    "RGBHues": 0,
    "RGBSaturation": 100,
    "RGBBrightness": 100
  }
}

/*这个例子有 4 步骤：1 创建 JSON 对象 2 解析 JSON 对象 3 解析完毕释放内存 4 控制 RGB*/
#include "MiCO.h"

#include "json_c/json.h" /*加入 JSON-C 的头文件*/
#include "micokit_ext.h" /*加入外围设备的头文件*/

/*格式化打印宏，打印字符串包括：时间戳、模块名、文件名、行号、内容*/
#define os_json_log(M, ...) custom_log("JSON", M, ##__VA_ARGS__)

void test_jsonc()
{
  /*控制 RGB 灯的信息*/
  bool rgb_sw = false;
  int rgb_hue = 0;
  int rgb_sat = 0;
  int rgb_bri = 0;
```

```
/*1、组装 JSON 对象*/
struct json_object *recv_json_object=NULL;

/*创建一个 recv_json_object 对象，此处是最大的对象*/
recv_json_object=json_object_new_object();

/*创建 device_object 对象，内有 5 个小对象形成 5 个键-值对*/
struct json_object *device_object=NULL;
device_object=json_object_new_object();

/*把字符、布尔、整型等类型转换为对象，并以键值对的方式加到 device_object 对象*/
json_object_object_add(device_object,"Hardware", json_object_new_string("MiCOKit3288"));
json_object_object_add(device_object, "RGBSwitch", json_object_new_boolean(false));
json_object_object_add(device_object, "RGBHues", json_object_new_int(0));
json_object_object_add(device_object, "RGBSaturation", json_object_new_int(100));
json_object_object_add(device_object, "RGBBrightness", json_object_new_int(100));
/*对象嵌套，将 device_object 对象以键值对的方式加到 recv_json_object 对象中*/
json_object_object_add(recv_json_object,"device_info",device_object);/*一个键值对*/
os_json_log("%s",json_object_to_json_string(recv_json_object));

/*2、解析 recv_json_object 对象*/
json_object* parse_json_object=json_object_object_get(recv_json_object,"device_info");
/*遍历解析以 K-V 的方式获取每个数据，需要将对象转为布尔、整型、字符等类型*/
json_object_object_foreach(parse_json_object, key, val) {
    if(!strcmp(key, "RGBSwitch")){
        rgb_sw = json_object_get_boolean(val); /*对象转布尔型*/
        os_json_log("rgb_sw=%d",rgb_sw);
    }
    else if(!strcmp(key, "RGBHues")){
        rgb_hue = json_object_get_int(val); /*对象转整型*/
        os_json_log("rgb_hue=%d",rgb_hue);
    }
    else if(!strcmp(key, "RGBSaturation")){
        rgb_sat = json_object_get_int(val); /*对象转整型*/
        os_json_log("rgb_sat=%d",rgb_sat);
    }
    else if(!strcmp(key, "RGBBrightness")){
        rgb_bri = json_object_get_int(val); /*对象转整型*/
        os_json_log("rgb_bri=%d",rgb_bri);
    }
}
```

```
}

/*3:解析完毕，释放 JSON 对象内存空间*/
json_object_put(recv_json_object);
recv_json_object=NULL;

/*4:操作 RGB 灯*/
os_json_log("control rgb led now");
rgb_led_init();/*RGB 灯初始化*/
hsb2rgb_led_open(rgb_hue, rgb_sat, rgb_bri);/*HSB 色彩模式控制灯*/
}

/*用户应用入口*/
int application_start( void )
{
    test_jsonc(); /*测试 JSON 对象*/
    return 0;
}
```

9. WiFi 串口透传

9.1 概述

本 Demo 数据传输示意如下图:

(说明: 数据传输 in/out 方向定义以 MiCOKit 为准, in: MiCOKit 接收, out: MiCOKit 发送)

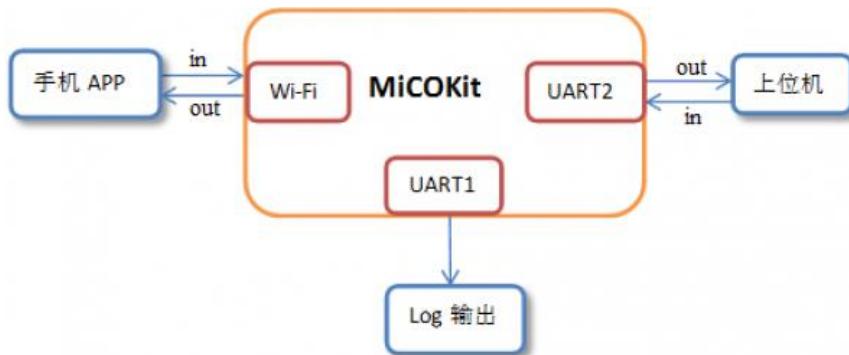


图 9.1 wifi 串口透传示意图

本工程主要实现:

1. 手机 APP 发送数据给 MiCOKit Wi-Fi 模块, 再从 MiCOKit 串口 2 输出给上位机。
2. 上位机发送数据给 MiCOKit 串口 2, 再从 MiCOKit 的 Wi-Fi 模块发送给手机 APP。

以上两个数据传输过程, 发送端和接收端数据格式与内容保持不变, 体现了透明传输特点, 可简称为“透传”。

说明:

1. MiCOKit 串口 2 是用户串口 (请参见 MiCOKit 上层扩展板的 4 脚 UART 单排针接口, 用户可外接 UART), 串口 1 是电源和调试串口(请参见 MiCOKit 下层底板 Mini-USB 口, 用于供电及运行 log 输出)。
2. 假设此时数据收, 发的两端是: 手机 APP(Easylink)和上位机(PC 机端 secureCRT 串口调试工具)。
3. 注意: 如果没有安装程序, 请进入集成开发环境页面下载。
4. 目前 Easylink 仅有 IOS 版支持透传功能, Android 版本只支持配网和参数设置, 还不支持透传功能)。

本文以 串口透传 Demo: wifi_uart.c 在 MiCOKit3165 开发板上运行为例, 详细描述其如何实现本地数据透传功能。

9.2 基本思路

Demo 基本思路如下:

- 1、MiCOKit 首先从 Flash 获取数据: 如果有配网信息, 则直接连接 Wi-Fi; 如果没有配网信息, 那么直接启动 APP Easylink 配网模式。

2、MiCOKit 连接到手机的 FTC server, 将本地配置数据以 json 格式 HTTP POST 给手机, 此时手机获取到设备联网信息。

3、设备端建立了 3 个服务器。

- (1) Config Server : 用来对固件参数读取和修改。
- (2) LocalTcpServer : 和手机 App 进行串口透传。
- (3) RemoteTcpClient : 和 PC 端 TCPsever 进行通信。

9.3 DEMO 运行过程

9.3.1 Easylink 配网

将 Demo 代码下载到 MiCOKit, 程序运行起来后, 打开手机 APP-Easylink, 若没有, 请至: APP 相关下载页面中下载。

安装成功后, APP 打开如下图:



图 9.2 启动 Easylink 配网

- (1) 点击右上角“+”号, 进入配网界面, 如图 9.2 左图。
- (2) 输入手机当前连接的 Wi-Fi 账号和密码, 如图 9.2 中图。点击“Fast Mode”, 开始配网。若 Easylink 配网超时, 设备自动开启 soft_AP 模式, IOS 苹果手机用户可按照 APP 指导界面进行配网。
- (3) 配网完成后, 下拉刷新设备列表, 会出现在线设备, 如图 9.2 右图。

(注意: 若在操作中遇到问题, 可按 MiCOKit 上“Reset”按键, 进行重启; 然后按“Easylink”键进入配网模式, 如此, 可重置配网过程)

9.3.2 设备参数设置

点击上图设备右侧的“i”，进入设置设备参数页面，如图 9.3：

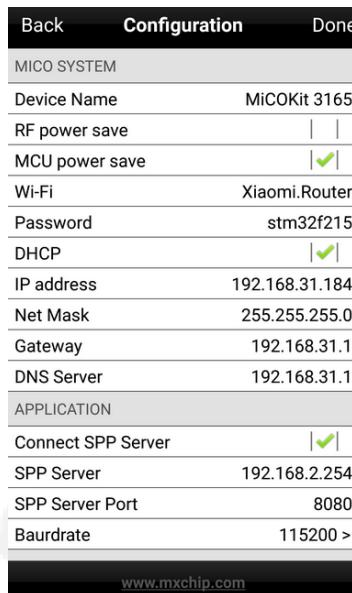


图 9.3 设备参数设置界面

可对上图中各参数项做出修改，修改后单击“Confirm”保存，程序运行 log 输出如下图：

```

[1] [Platform: mico_platform_common.c: 106] Platform initialised, build by IAK
[337] [SYSTEM: system_misc.c: 235] Free memory 62624 bytes
[343] [SYSTEM: system_misc.c: 241] Kernel version: 31620002.042
[349] [SYSTEM: system_misc.c: 244] MiCO version: 2.4.0
[354] [SYSTEM: system_misc.c: 246] Wi-Fi driver version w10: Sep 10 2014 11:28:46 version 5.90.230.10 FWI, mac C8:93:46:54:8D:36
[366] [SYSTEM: mico_system_init.c: 112] Empty configuration. Starting configuration mode...
[377] [CONFIG SERVER: config_server.c: 165] Config Server established at port: 8000, fd: 1
[1078] [SYSTEM: system_easylink.c: 251] Start easylink combo mode
[1084] [TCP SERVER: LocalTcpServer.c: 60] Server established at port: 8080, fd: 2
[3202] [SYSTEM: system_easylink.c: 103] Get SSID: Xiaomi.Router, Key: stm32f215
[3210] [SYSTEM: system_easylink.c: 138] Get user info: 23 AB IF A8 C0
[3216] [SYSTEM: system_easylink.c: 165] Get auth info: , EasyLink identifier: c0a81fab
[3225] [SYSTEM: system_misc.c: 194] connect to Xiaomi.Router.....
[8432] [SYSTEM: system_misc.c: 84] Station up

[8524] [Config Delegate: MiCOConfigDelegate.c: 35] Config Client 192.168.31.171:45790 connected, fd: 3
[214680] [CONFIG SERVER: config_server.c: 185] Config Client 192.168.31.171:45790 connected, fd: 3
[214701] [CONFIG SERVER: config_server.c: 465] Send config object={ "T": "Current Configuration", "N": "MiCOKit-3165(548D36)", "C": [ { "N": "MICO SYSTEM", "C": [ { "N": "Device Name", "C": "MiCOKit 3165", "P": "RW" }, { "N": "RF power save", "C": false, "P": "RW" }, { "N": "MCU power save", "C": false, "P": "RW" }, { "N": "Wi-Fi", "C": "Xiaomi.Router", "P": "RW" }, { "N": "Password", "C": "stm32f215", "P": "RW" }, { "N": "DHCP", "C": true, "P": "RW" }, { "N": "IP address", "C": "192.168.31.184", "P": "RW" }, { "N": "Net Mask", "C": "255.255.255.0", "P": "RW" }, { "N": "Gateway", "C": "192.168.31.1", "P": "RW" }, { "N": "DNS Server", "C": "192.168.31.1", "P": "RW" } ], { "N": "APPLICATION", "C": [ { "N": "Connect SPP Server", "C": true, "P": "RW" }, { "N": "SPP Server", "C": "192.168.2.254", "P": "RW" }, { "N": "SPP Server Port", "C": 8080, "P": "RW" }, { "N": "Baudrate", "C": 115200, "P": "RW" }, { "N": "S", "V": [ 9600, 19200, 38400, 57600, 115200 ] } ] }, "PO": "com.mxchip.spp", "HD": "MK3165_1", "FW": "MICO_SPP_2_6", "RF": "w
[0: Sep 10 2014 11:28:46 version 5.90.230.10 FWI" ]
[214799] [CONFIG SERVER: config_server.c: 473] Current configuration sent
[227504] [CONFIG SERVER: config_server.c: 185] Config Client 192.168.31.171:51255 connected, fd: 4
[227513] [CONFIG SERVER: config_server.c: 237] Free memory 49200 bytes
[227520] [CONFIG SERVER: config_server.c: 478] Recv new configuration, apply
[227527] [CONFIG SERVER: config_server.c: 488] Recv config object={ "MCU power save": true }

[1] [Platform: mico_platform_common.c: 106] Platform initialised, build by IAK
[337] [SYSTEM: system_misc.c: 235] Free memory 62624 bytes
[343] [SYSTEM: system_misc.c: 241] Kernel version: 31620002.042
[349] [SYSTEM: system_misc.c: 244] MiCO version: 2.4.0
[354] [SYSTEM: system_misc.c: 246] Wi-Fi driver version w10: Sep 10 2014 11:28:46 version 5.90.230.10 FWI, mac C8:93:46:54:8D:36
[365] [SYSTEM: mico_system_init.c: 134] Available configuration. Starting Wi-Fi connection...
[874] [SYSTEM: system_misc.c: 222] Connect to Xiaomi.Router.....
[1768] [SYSTEM: system_misc.c: 84] Station up
[1775] [CONFIG SERVER: config_server.c: 165] Config Server established at port: 8000, fd: 1


```

图 9.4 wifi 串口透传 log

9.3.3 数据透传

(1) 配网成功后，单击设备列表中设备图标，进入透传功能界面，示例如图 9.5 左图。

(2) 上位机（串口调试工具）发送与接收如图 9.5 右图，可看出与左图发送对内容对应。



图 9.5 透传 APP 使用过程演示

从以上两图看出透传过程：

- (1) 手机 APP 向上位机发送字符串：“micomicomico”，如串口调试工具中“接收区”显示。
- (2) 上位机向手机 APP 发送字符串：“Hello MiCO”，如串口调试工具“发送区”显示。

10. 校验算法

Micro 为开发者提供了常用校验算法: CRC、MD5、SHA 的 API 接口供开发者使用，并给出使用例程。

10.1 CRC 校验

10.1.1 概述

CRC 即循环冗余校验码 (Cyclic Redundancy Check[1]): 是数据通信领域中最常用的一种查错校验码，其特征是信息字段和校验字段的长度可以任意选定。循环冗余检查 (CRC) 是一种数据传输检错功能，对数据进行多项式计算，并将得到的结果附在帧的后面，接收设备也执行类似的算法，以保证数据传输的正确性和完整性。CRC8 和 CRC16 应用层代码是一样的，这里将 CRC8 和 CRC16 的代码整合到了一起，即把 crc16 mico check test.c 中的应用层代码拷贝到 crc8 mico check test.c 中，详细代码见下面 12.1.1 章节。

10.1.2crc8 mico check test.c 功能

本 Demo 实现对一个数组变量进行 CRC8 计算校验值，log 如下：

```
[0][Security: crc8_mico_check_test.c: 68] CRC8 mico Check Test Start !
[7][Security: crc8_mico_check_test.c: 72] CRC8 mico Check Test Finished! The CRC8result is 128
```

10.1.3crc16 mico check test.c 功能

本 Demo 实现对一个数组变量进行 CRC16 计算校验值，log 如下：

```
[0][Security: crc16_mico_check_test.c: 68] CRC16 mico Check Test Start !
[7][Security: crc16_mico_check_test.c: 72] CRC16 mico Check Test Finished! The CRC16 result is 55564
```

10.1.4 CRC API 接口定义

CRC 的 API 函数在 CheckSumUtils.c 文件中定义，如下图所示：

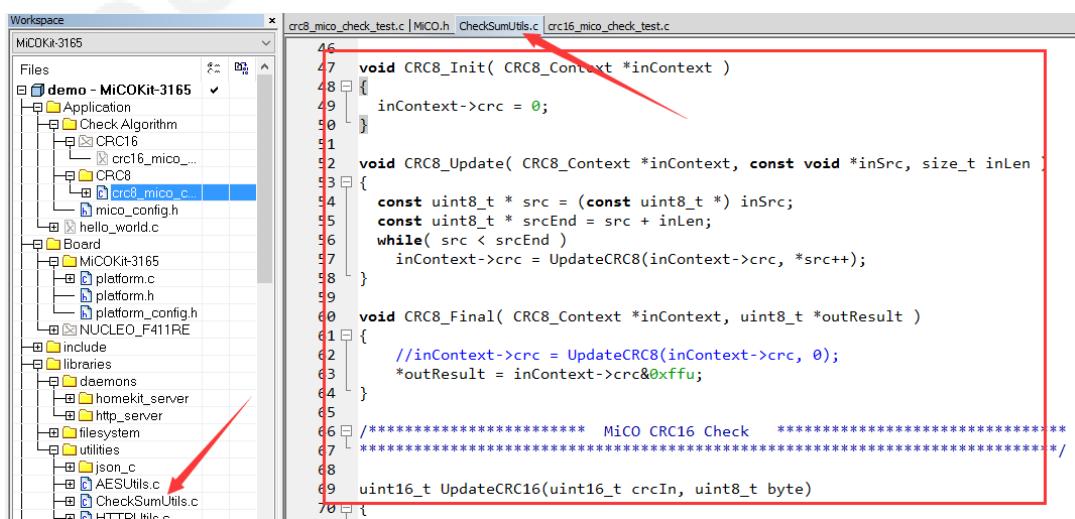


图 10.1 CRC API 定义

10.1.5 CRC 使用注意事项

CRC 校验是有校验前提的，前提配置不同，校验结果也会不一样。

MiCO SDK 中的 CRC8 校验的前提配置多项式为 0x31，数据反转为“低位在前”，初始值为 0（在 CRC8_Init(&crc) 函数中初始化）。推荐开发者使用“格西 CRC 校验计算器”如图 10.2 所示：



图 10.2 CRC8 校验参数设置

MiCO SDK 中的 CRC16 校验的前提配置多项式为 0x1021，数据反转为“高位在前”，初始值为 0（在 CRC16_Init(&crc) 函数中初始化）。算法使用的是移位的方式实现的，不是查表法，如果开发者想要更快的速度，可以自行编写查表法的校验函数，但要注意数据表是要占用 flash 空间的。CRC16 校验的配置如图 10.3 所示：

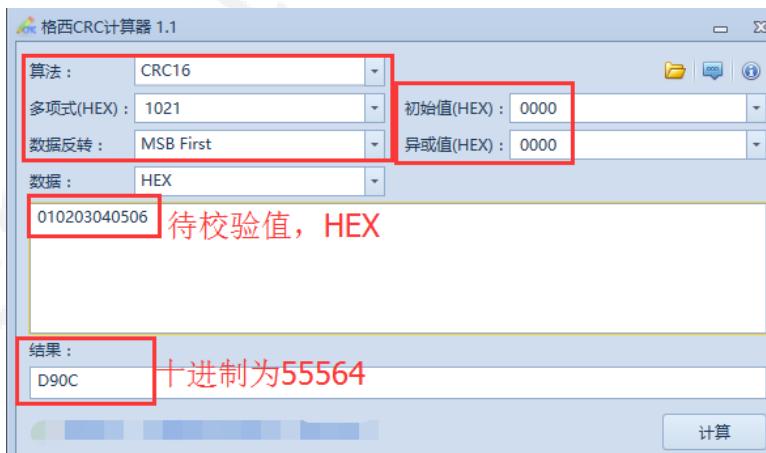


图 10.3 CRC16 校验参数配置

10.2 MD5

MD5 即 Message-Digest Algorithm 5（信息-摘要算法 5），用于确保信息传输完整一致。是计算机广泛使用的杂凑算法之一（又译摘要算法、哈希算法），主流编程语言普遍已有 MD5 实现。将数据（如汉字）运算为另一固定长度值，是杂凑算法的基础原理，MD5 的前身有 MD2、MD3 和 MD4。

MD5 算法具有以下特点：

- 1、压缩性：任意长度的数据，算出的 MD5 值长度都是固定的。
- 2、容易计算：从原数据计算出 MD5 值很容易。
- 3、抗修改性：对原数据进行任何改动，哪怕只修改 1 个字节，所得到的 MD5 值都有很大区别。
- 4、强抗碰撞：已知原数据和其 MD5 值，想找到一个具有相同 MD5 值的数据（即伪造数据）是非常困难的。

MD5 的作用是让大容量信息在用数字签名软件签署私人密钥前被"压缩"成一种保密的格式（就是把一个任意长度的字节串转换成一定长的十六进制数字串）。除了 MD5 以外，其中比较有名的还有 sha-1、RIPEMD 以及 Haval 等。

10.2.1 md5_test.c 源码

```
#include "mico.h"

#define md5_test_log(format, ...) custom_log("Security", format, ##__VA_ARGS__)

typedef struct testVector {
    const char* input;
    const char* output;
    size_t inLen;
    size_t outLen;
} testVector;

int md5_test(void);

int application_start(void)
{
    int ret = 0;

    md5_test_log( "MD5 Test Start ! " );
    printf("\r\n");

    if ( (ret = md5_test()) != 0 )
        md5_test_log("MD5 Test Failed! The Wrong Number is %d",ret);
    else
        md5_test_log("MD5 Test Passed!");

    return 0;
}

/***************** Defination of md5_test() ********************/

int md5_test(void)
{
    md5_context md5;
```

```
uint8_t hash[MD5_DIGEST_SIZE];

testVector a, b, c, d, e;

testVector test_md5[5];

int times = sizeof(test_md5) / sizeof(testVector), i,j;

/* Defination of input string to be used MD5 Enryption */

a.input = "abc";

a.output = "\x90\x01\x50\x98\x3c\xd2\x4f\xb0\xd6\x96\x3f\x7d\x28\xe1\x7f"
"\x72";

a.inLen = strlen(a.input);

a.outLen = MD5_DIGEST_SIZE;

b.input = "message digest";

b.output = "\xf9\x6b\x69\x7d\x7c\xb7\x93\x8d\x52\x5a\x2f\x31\xaa\xf1\x61"
"\xd0";

b.inLen = strlen(b.input);

b.outLen = MD5_DIGEST_SIZE;

c.input = "abcdefghijklmnopqrstuvwxyz";

c.output = "\xc3\xfc\xd3\xd7\x61\x92\xe4\x00\x7d\xfb\x49\x6c\xca\x67\xe1"
"\x3b";

c.inLen = strlen(c.input);

c.outLen = MD5_DIGEST_SIZE;

d.input = "ABCDEFGHIJKLMNPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
d.output = "\xd1\x74\xab\x98\xd2\x77\xd9\xf5\xa5\x61\x1c\x2c\x9f\x41\x9d\x9f";
d.inLen = strlen(d.input);

d.outLen = MD5_DIGEST_SIZE;

e.input = "1234567890123456789012345678901234567890123456789012345678"
"9012345678901234567890";

e.output = "\x57\xed\xf4\xa2\x2b\xe3\xc9\x55\xac\x49\xda\x2e\x21\x07\xb6"
"\x7a";

e.inLen = strlen(e.input);

e.outLen = MD5_DIGEST_SIZE;

test_md5[0] = a;
test_md5[1] = b;
test_md5[2] = c;
test_md5[3] = d;
test_md5[4] = e;

InitMd5(&md5);

for (i = 0; i < times; ++i)
```

```

{
    Md5Update(&md5, (uint8_t*)test_md5[i].input, (uint32_t)test_md5[i].inLen);

    Md5Final(&md5, hash);

    /* Print each hash value of test_md5[i].input */

    printf( "The %d's MD5 Hash Value is: ", i+1);

    for (j = 0; j < MD5_DIGEST_SIZE; ++j)

    {
        printf("-0x%02x", hash[j]);
    }

    printf("\r\n");
    printf("\r\n");

    /* Compare each hash with predefined hash value - test_md5[i].output */

    if (memcmp(hash, test_md5[i].output, MD5_DIGEST_SIZE) != 0)

        return i+1;

    }

    return 0;
}

```

10.2.2 md5_test.c 运行结果

```

serial-com8 - SecureCRT
文件(F) 编辑(E) 查看(V) 选项(O) 传输(T) 脚本(S) 工具(L) 窗口(W) 帮助(H)
输入主机 <Alt+R> | 书签 | 新建 | 打开 | 保存 | 另存为 | 导入 | 导出 | 剪切 | 复制 | 粘贴 | 全选 | 撤销 | 恢复 | 退出 | 
serial-com8 x
0] [Security: md5_test.c: 68] MD5 Test Start !
The 1's MD5 Hash Value is: -0x90-0x1-0x50-0x98-0x3c-0xd2-0x4f-0xb0-0xd6-0x96-0x3f-0x7d-0x28-0xe1-0x7f-0x72
The 2's MD5 Hash Value is: -0xf9-0x6b-0x69-0x7d-0x7c-0xb7-0x93-0x8d-0x52-0x5a-0x2f-0x31-0xaa-0xf1-0x61-0xd0
The 3's MD5 Hash Value is: -0xc3-0xfc-0xd3-0xd7-0x61-0x92-0xe4-0x0-0x7d-0xfb-0x49-0x6c-0xca-0x67-0xe1-0x3b
The 4's MD5 Hash Value is: -0xd1-0x74-0xab-0x98-0xd2-0x77-0xd9-0xf5-0xa5-0x61-0x1c-0x2c-0x9f-0x41-0x9d-0x9f
The 5's MD5 Hash Value is: -0x57-0xed-0xf4-0xa2-0x2b-0xe3-0xc9-0x55-0xac-0x49-0xda-0x2e-0x21-0x7-0xb6-0x7a
[55] [Security: md5_test.c: 74] MD5 Test Passed!

```

图 10.4 MD5 串口 log

10.2.3 md5_test.c API 接口定义

```

InitMd5(&md5);

Md5Update(&md5, (uint8_t*)test_md5[i].input, (uint32_t)test_md5[i].inLen);

Md5Final(&md5, hash);

```

上面这些接口函数在 mico_security.h 中定义，如图 10.5 所示：

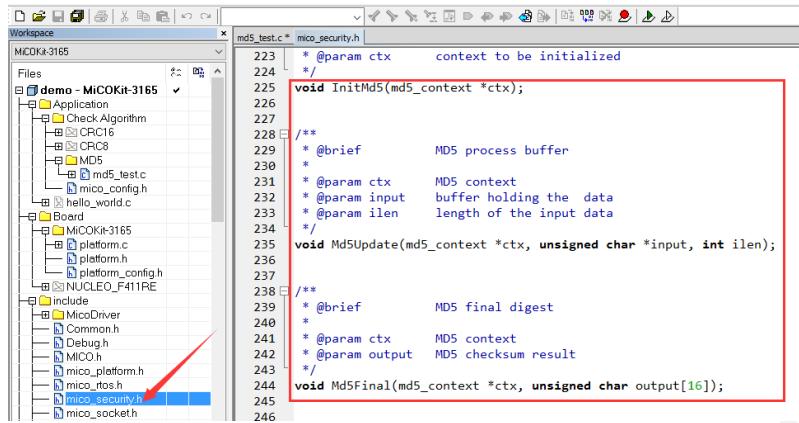


图 10.5 MD5 API 定义

10.3 SHA

SHA (Secure Hash Algorithm, 译作安全散列算法) 是美国国家安全局 (NSA) 设计, 美国国家标准与技术研究院(NIST) 发布的一系列密码散列函数。正式名称为 SHA 的家族第一个成员发布于 1993 年。然而人们给它取了一个非正式的名称 SHA-0 以避免与它的后继者混淆。两年之后, SHA-1, 第一个 SHA 的后继者发布了。另外还有四种变体, 曾经发布以提升输出的范围和变更一些细微设计: SHA-224, SHA-256, SHA-384 和 SHA-512 (这些有时候也被称做 SHA-2)。

MiCO SDK 中整合了 sha1、sha256、sha384 和 sha512 四种算法, 文件目录如图 10.6 所示:

MiCO > Data (E:) > MiCO_SDK_2.4.1 > MiCO_SDK_2.4.1 > Demos > Check Algorithm > SHA			
名称	修改日期	类型	大小
sha1_test.c	2016/4/18 11:33	C Source File	5 KB
sha256_test.c	2016/4/18 11:33	C Source File	5 KB
sha384_test.c	2016/4/18 11:33	C Source File	5 KB
sha512_test.c	2016/4/18 11:33	C Source File	5 KB

图 10.6 SHA 算法文件目录

这里以 SHA1 为例, 简单介绍下代码。

10.3.1 sha1_test.c 源码

```

#include "mico.h"
#include "sha.h"

#define sha1_test_log(format, ...) custom_log("Security", format, ##__VA_ARGS__)

int sha1_test(void);

int application_start(void)
{
    int ret = 0;

    sha1_test_log( "SHA1 Test Start\r\n" );
}

```

```
if ( (ret = sha1_test()) != 0)
    sha1_test_log("SHA1 Test Failed! The Wrong Number is %d ",ret);
else
    sha1_test_log("SHA1 Test Passed!");

return 0;
}

/***************** Defination of sha1_test() ********************/
int sha1_test(void)
{
    uint8_t hash[SHA1HashSize];
    testVector a, b, c, d;
    testVector test_sha[4];
    int times = sizeof(test_sha) / sizeof(struct testVector), i,j;
    USHACContext usercontext;
    /* Defination of input string to be used SHA1 Enryption */
    a.input = "abc";
    a.output = "\xA9\x99\x3E\x36\x47\x06\x81\x6A\xBA\x3E\x25\x71\x78\x50\xC2"
               "\x6C\x9C\xD0\xD8\x9D";
    a.inLen = strlen((char *)a.input);
    a.outLen = SHA1HashSize;
    b.input = "abcdefghijklmnopqrstuvwxyz";
    b.output = "\x84\x98\x3E\x44\x1C\x3B\xD2\x6E\xBA\xAE\x4A\xA1\xF9\x51\x29"
               "\xE5\xE5\x46\x70\xF1";
    b.inLen = strlen((char *)b.input);
    b.outLen = SHA1HashSize;
    c.input = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
              "aaaaaa";
    c.output = "\x00\x98\xBA\x82\x4B\x5C\x16\x42\x7B\xD7\xA1\x12\x2A\x5A\x44"
               "\x2A\x25\xEC\x64\x4D";
    c.inLen = strlen((char *)c.input);
    c.outLen = SHA1HashSize;
    d.input = "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
              "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
              "aaaaaaaaaa";
    d.output = "\xAD\x5B\x3F\xDB\xCB\x52\x67\x78\xC2\x83\x9D\x2F\x15\x1E\xA7"
               "\x53\x99\x5E\x26\xA0";
```

```
d.inLen = strlen((char *)d.input);

d.outLen = SHA1HashSize;

test_sha[0] = a;

test_sha[1] = b;

test_sha[2] = c;

test_sha[3] = d;

for (i = 0; i < times; ++i) {

    USHAReset(&usercontext,SHA1);

    USHAInput(&usercontext,test_sha[i].input,test_sha[i].inLen);

    USHAResult(&usercontext,hash);

    for (j = 0; j < SHA1HashSize; j++)

    {

        printf("-x%x",hash[j]);

    }

    printf("\r\n");

    printf("\r\n");

    if (memcmp(hash, test_sha[i].output, SHA1HashSize) != 0)

        return i+1;

}

return 0;
```

注意：

默认工程中不包含 SHA 相关功能的头文件。添加 SHA 应用时，需添加以下文件路径到工程。见图 10.7。

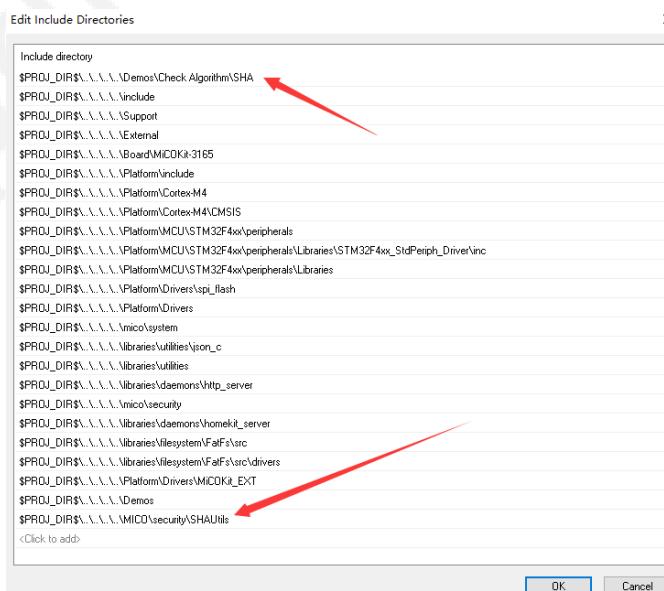


图 10.7 SHA 相关头文件路径

10.3.2 sha1_test.c 运行结果

```
[0][Security: sha1_test.c: 68] SHA1 Test Start
-xa9-x99-x3e-x36-x47-x6-x81-x6a-xba-x3e-x25-x71-x78-x50-xc2-x6c-x9c-xd0-xd8-x9d
-x84-x98-x3e-x44-x1c-x3b-xd2-x6e-xba-xae-x4a-xa1-xf9-x51-x29-xe5-xe5-x46-x70-xf1
-x0-x98-xba-x82-x4b-x5c-x16-x42-x7b-xd7-xa1-x12-x2a-x5a-x44-x2a-x25-xec-x64-x4d
-xad-x5b-x3f-xdb-xcb-x52-x67-x78-xc2-x83-x9d-x2f-x15-x1e-xa7-x53-x99-x5e-x26-xa0
[37] [Security: sha1_test.c: 73] SHA1 Test Passed!
```

图 10.8 SHA1 串口 log

10.3.3 sha1_test.c API 接口定义

SHA 函数接口定义源文件在 MiCO_SDK_2.4.1\MICO\security\SHAUtils 目录下，工程结构如下图所示：

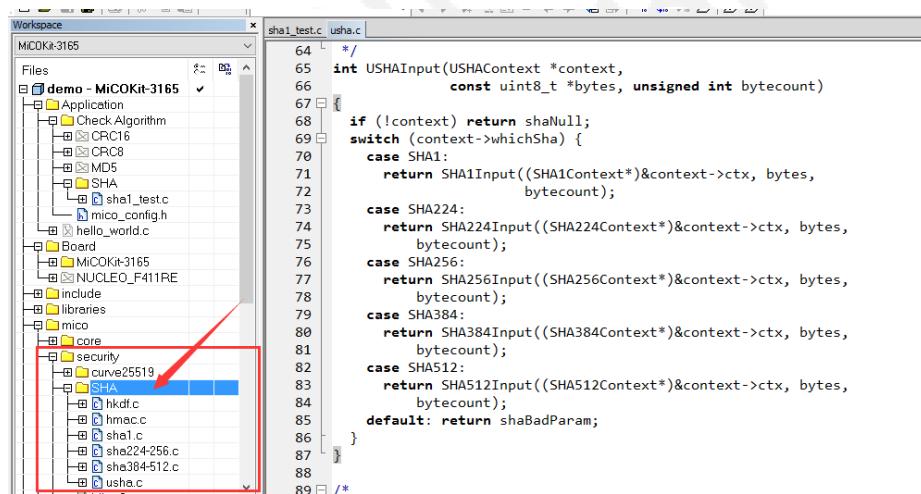


图 10.9 SHA1 API 定义

11. 加密算法

11.1 加密算法概述

加密技术通常分为两大类：“对称式”和“非对称式”。

常见的加密算法有以下几种：

AES (Advanced Encryption Standard): 高级加密标准，对称算法，是下一代的加密算法标准，速度快，安全级别高，在 21 世纪 AES 标准的一个实现是 Rijndael 算法；

RSA: RSA 由 RSA 公司发明，是一个支持变长密钥的公共密钥算法，需要加密的文件块的长度也是可变的，非对称算法；

RC4: RC4 加密算法是大名鼎鼎的 RSA 三人组中的头号人物 Ronald Rivest 在 1987 年设计的密钥长度可变的流加密算法簇。之所以称其为簇，是由于其核心部分的 S-box 长度可为任意，但一般为 256 字节。该算法的速度可以达到 DES 加密的 10 倍左右，且具有很高级别的非线性。RC4 也被叫做 ARC4(Alleged RC4—所谓的 RC4)；

DES (Data Encryption Standard): 对称算法，数据加密标准，速度较快，适用于加密大量数据的场合；

HMAC 是密钥相关的哈希运算消息认证码 (Hash-based Message Authentication Code), HMAC 运算利用哈希算法，以一个密钥和一个消息为输入，生成一个消息摘要作为输出；

Rabbit 流密码是由 Cryptico 公司 (<http://www.cryptico.com>) 设计的，密钥长度 128 位，最大加密消息长度为 2^{64} Bytes，即 16 TB，若消息超过该长度，则需要更换密钥对剩下的消息进行处理。它是目前安全性较高，加/解密速度比较高效的流密码之一，在各种处理器平台上都有不凡的表现。

MiCO SDK 中提供了 AES、ARC4、DES、HMAC、Rabbit 和 RSA 等主流安全加密算法，这些算法以库的形式提供，API 接口函数在 [MiCO_SDK_2.4.1\include\mico_security.h](#) 中声明，有相关应用的开发者，请参考该文件及 DEMO 例程，DMEO 例程在 [MiCO_SDK_2.4.1\Demos\Security Algorithm](#) 目录结构下，SDK 中已经包含了这些加密算法的驱动文件，使用的时候直接把 DEMO 中的应用层代码添加到 DEMO 工程中即可。

13 章节仅介绍每个 Demo 的功能及运行结果。具体的 API 信息请参考：“MiCO API 参考手册”。

11.2 AES 加密

11.2.1aes_cbc_test.c : cbc 模式的 AES 加密

本 Demo 实现对一个数组进行加密，并解密操作，验证其正确性。输出 log 如下：

```
[0][Security: aes_cbc_test.c: 68] AES in CBC mode Test Start
The Cipher Text is: -95-94-92-57-5f-42-81-53-2c-cc-9d-46-77-a2-33-cb
The Plain Text is: -6e-6f-77-20-69-73-20-74-68-65-20-74-69-6d-65-20
[20][Security: aes_cbc_test.c: 74] AES in CBC mode Test Passed!
```

图 11.1 aes cbc 加密串口 log

11.2.2aes_ecb_test.c : ecb 模式的 AES 加密

本 Demo 实现对一个数组进行加密，并解密操作，验证其正确性。输出 log 如下：

```
[0][Security: aes_ecb_test.c: 68] AES Test Start
The Cipher Text is: -f3-ee-d1-bd-b5-d2-a0-3c-6-4b-5a-7e-3d-b1-81-f8
The Plain Text is: -6b-c1-be-e2-2e-40-9f-96-e9-3d-7e-11-73-93-17-2a
[19][Security: aes_ecb_test.c: 74] AES in ECB mode Test Passed!
```

图 11.2 aes ecb 加密串口 log

11.2.3aes_with_pkcs5_padding_test.c: cbc 模式的 PKCS#5 填充 AES 加密

本 Demo 实现对一个字符串进行加密，并解密操作，验证其正确性。输出 log 如下：

```
[0][Security: aes_with_pkcs5_padding_test.c: 38] AES PKCS#5 Padding Demo.
[8][Security: aes_with_pkcs5_padding_test.c: 59] The Cipher Text is:
cb 50 11 b7 d1 b0 9a ec ee 82 e8 df ff b3 55 40
[19][Security: aes_with_pkcs5_padding_test.c: 72] The Plain Text is:
30 31 32 33 34 35 36 37 38
```

图 11.3 aes with pkcs5 加密串口 log

11.3 ARC4 加密

11.3.1arc4_test.c 的功能

本 Demo 实现对 4 个字符串进行加密，并解密操作，验证其正确性。输出 log 如下：

```
[0][Security: arc4_test.c: 68] ARC4 Test Start !
The 0's Cipher Text is: -75-b7-87-80-99-e0-c5-96
The 0's Plain Text is: -1-23-45-67-89-ab-cd-ef
The 1's Cipher Text is: -74-94-c2-e7-10-4b-8-79
The 1's Plain Text is: -0-0-0-0-0-0-0-0
The 2's Cipher Text is: -de-18-89-41-a3-37-5d-3a
The 2's Plain Text is: -0-0-0-0-0-0-0-0
The 3's Cipher Text is: -d6-a1-41-a7-ec-3c-38-df-bd-61
The 3's Plain Text is: -0-0-0-0-0-0-0-0
[43][Security: arc4_test.c: 75] ARC4 Test Passed!
```

图 11.4 arc4 加密串口 log

11.4 DES 加密

MiCO 支持 DES 加密和 3DES 加密两种算法。

11.4.1des_test.c: 64 位密钥的 DES 加密

本 Demo 实现对一个数组进行加密，并解密操作，验证其正确性。输出 log 如下：

```
[0][Security: des_test.c: 69] DES Test Start
The Cipher Text is:-8b-7e-52-b0-1-2b-6e-b8-4f-f-eb-f3-fb-5f-86-73-15-85-b3-22-4b-86-2b-4b
The Plain Text is:-6e-6f-77-20-69-73-20-74-68-65-20-74-69-6d-65-20-66-6f-72-20-61-6e-20
[24][Security: des_test.c: 75] DES Test Passed!
```

图 11.5 DES 加密串口 log

11.4.213.5.1 des3_test.c: 192 位密钥的 3DES 加密

本 Demo 实现对一个数组进行加密，并解密操作，验证其正确性。输出 log 如下：

```
[0][Security: des3_test.c: 69] des3 Test Start
The Cipher Text is:-43-a0-29-7e-d1-84-f8-e-89-64-64-32-12-d5-8-98-18-94-15-74-87-12-7d-b0
The Plain Text is:-4e-6f-77-20-69-73-20-74-68-65-20-74-69-6d-65-20-66-6f-72-20-61-6e-20
[27][Security: des3_test.c: 75] DES3 Test Passed!
```

图 11.6 DES3 加密串口 log

11.5 HMAC 加密

HMAC_MD5, SHA1, SHA256, SHA384, SHA512 的公用一套 API 函数。

该部分 demo 工程需重新选定 mico_config.h 的路径为：MiCO_SDK_2.4.1\Demos\Security Algorithm\HMAC，否则运行出错。

11.5.1 hmac_md5_test.c: HAMC MD5 加密

本 Demo 实现对 3 个字符串进行加密，并验证其正确性。输出 log 如下：

```
[0][Security: hmac_md5_test.c: 64] HMAC MD5 Test Start
The 1's hmac_md5 Hash Value is: -0x92-0x94-0x72-0x7a-0x36-0x38-0xbb-0x1c-0x13-0xf4-0x8e-0xf8-0x15-0x8b-0xfc-0x9d
The 2's hmac_md5 Hash Value is: -0x75-0xc-0x78-0x3e-0x6a-0xb0-0xb5-0x3-0xea-0xa8-0x6e-0x31-0xa-0x5d-0xb7-0x38
The 3's hmac_md5 Hash Value is: -0x56-0xbe-0x34-0x52-0x1d-0x14-0x4c-0x88-0xdb-0xb8-0xc7-0x33-0xf0-0xe8-0xb3-0xf6
[38][Security: hmac_md5_test.c: 70] HMAC MD5 Test Passed!
```

图 11.7 hmac_md5 加密串口 log

11.5.2 hmac_sha1_test.c: HAMC SHA1 加密

本 Demo 实现对 3 个字符串进行加密，并验证其正确性。输出 log 如下：

```
[0][Security: hmac_sha1_test.c: 64] HMAC SHA1 Test Start
-0xb6-0x17-0x31-0x86-0x55-0x5-0x72-0x64-0xe2-0x8b-0xc0-0xb6-0xfb-0x37-0x8c-0x8e-0xf1-0x46-0xbe-0x0
-0xef-0xfc-0xdf-0x6a-0xe5-0xeb-0x2f-0xa2-0xd2-0x74-0x16-0xd5-0xf1-0x84-0xdf-0x9c-0x25-0x9a-0x7e-0x79
-0x12-0x5d-0x73-0x42-0xb9-0xac-0x11-0xcd-0x91-0xa3-0xf4-0x8a-0xa1-0x7b-0x4f-0x63-0xf1-0x75-0xd3
[36][Security: hmac_sha1_test.c: 70] HMAC SHA1 Test Passed!
```

图 11.8 hmac_sha1 加密串口 log

11.5.3 hmac_sha256_test.c: HAMC SHA256 加密

本 Demo 实现对 3 个字符串进行加密，并验证其正确性。输出 log 如下：

```
[0] [Security: hmac_sha256_test.c: 64] HMAC SHA256 Test Start
-0xb0-0x34-0x4c-0x61-0xd8-0xdb-0x38-0x53-0x5c-0xa8-0xaf-0xce-0xaf-0xb-0xf1-0x2b-0x88-0x1d-0xe2-0x0-0xc9-0x83-0x3d-0xa7-0x26-0xe9-0x37-0x6c-0x2e-0x32-0xcf-0xf7
-0x5b-0xdc-0xcl-0x46-0xbf-0x60-0x75-0x4e-0x6a-0x4-0x24-0x26-0x8-0x95-0x75-0x7-0x5a-0x0-0x3f-0x8-0x9d-0x27-0x39-0x83-0x9d-0xec-0x58-0xb9-0x64-0xec-0x38-0x43
-0x77-0x3e-0xa9-0x1e-0x36-0x80-0xe-0x46-0x85-0x4d-0xb8-0xeb-0xd0-0x91-0x81-0xa7-0x29-0x59-0x9-0x8b-0x3e-0x8f-0xcl-0x22-0xd9-0x63-0x55-0x14-0xce-0x45-0x65-0xfe
[52] [Security: hmac_sha256_test.c: 70] HMAC SHA256 Test Passed!
```

图 11.9 hmac_sha256 加密串口 log

11.5.4 hmac_sha384_test.c: HAMC SHA384 加密

本 Demo 实现对 3 个字符串进行加密，并验证其正确性。输出 log 如下：

图 11.10 hmac_sha384 加密串口 log

11.5.5 hmac_sha512_test.c: HAMC SHA512 加密

本 Demo 实现对 3 个字符串进行加密，并验证其正确性。输出 log 如下：

图 11.11 hmac_sha512 加密串口 log

11.6 Rabbit 加密

本 Demo 实现对 1 个字符串进行加密，并验证其正确性。输出 log 如下：

```
[0][Security: rabbit_test.c: 69] Rabbit Test Start  
The 1's Cipher Text is: -ed-b7-5-67-37-5d-cd-7c  
The 1's Plain Text is: -0-0-0-0-0-0-0-0  
The 2's Cipher Text is: -6d-7d-1-22-92-cc-dc-e0-e2  
The 2's Plain Text is: -0-0-0-0-0-0-0-0
```

图 11.12 Rabbit 加密串口 log

11.7 RSA 加密

本 Demo 实现对 1 个字符串进行加密，并验证其正确性。输出 log 如下：

```
[0][Security: rsa_test.c: 130] RSA Test Start  
[1607][Security: rsa_test.c: 136] RSA Test Passed!
```

图 11.13 RSA 加密串口 log

12. 文件系统 fatfs

12.1 文件系统概述

文件系统是操作系统用于明确磁盘或分区上的文件的方法和数据结构；即在磁盘上组织文件的方法。也指用于存储文件的磁盘或分区，或文件系统种类。操作系统中负责管理和存储文件信息的软件机构称为文件管理系统，简称文件系统。文件系统由三部分组成：与文件管理有关软件、被管理文件以及实施文件管理所需数据结构。从系统角度来看，文件系统是对文件存储器空间进行组织和分配，负责文件存储并对存入的文件进行保护和检索的系统。具体地说，它负责为用户建立文件，存入、读出、修改、转储文件，控制文件的存取，当用户不再使用时撤销文件等。

12.2 fatfs.c 功能说明

MICO SDK 中使用文件系统，工程中需添加特定.c 文件。文件在 SDK 目录下 libraries 文件中。

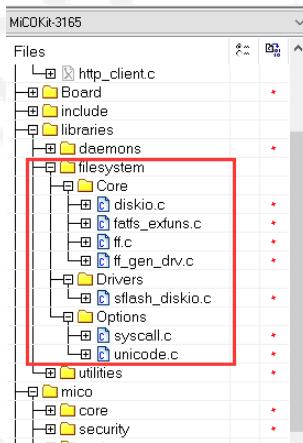


图 12.1 文件系统添加文件

然后，需要在工程中 include 如下路径：

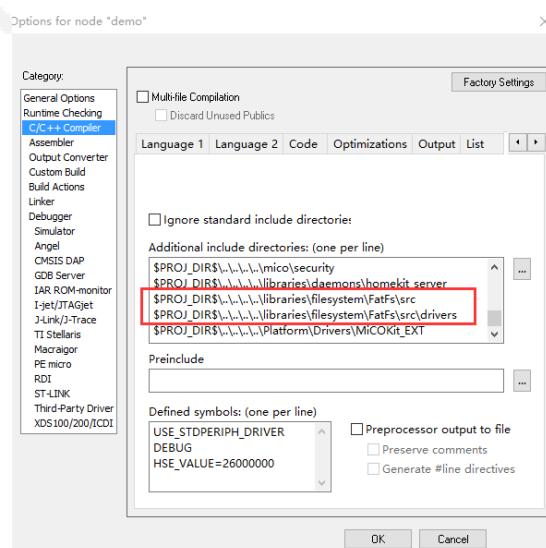


图 12.2 添加工程包含路径

运行后，log 输出如下图。

本 Demo 实现：

- (1) 创建文件：MiCO.txt，数据：hello MiCO!，将文件写入 flash
- (2) 读出文件：MiCO.tex 的数据，并将数据 log 输出打印。

```
[6][FATFS: fatfs.c: 64] filesystem total space is 972KB, free space is 971KB
[124][FATFS: fatfs.c: 75] fatfs write file, name:MiCO.txt, data:hello MiCO!
[290][FATFS: fatfs.c: 89] fatfs read file, name:MiCO.txt, data:hello MiCO!
```

图 12.3 文件系统串口 log

Demo 程序代码如下：

```
#include "mico.h"
#include "ff_gen_drv.h"
#include "sflash_diskio.h"

#define os_fatfs_log(M, ...) custom_log("FATFS", M, ##__VA_ARGS__)

FATFS SFLASHFatFs;           /* File system object for flash disk logical drive */
FIL SFLASHFile;             /* File object */
char SFLASHPath[4];          /* SFLASH logical drive path */

void test_fatfs()
{
    HRESULT err;
    filesystem_info fatfs_info;
    uint32_t byteswritten, bytesread;
    char filename[] = "MiCO.txt";
    uint8_t wtext[] = "hello MiCO!";
    uint8_t rtext[100];
    FATFS_LinkDriver(&SFLASHDISK_Driver, SFLASHPath);

    err = f_mount(&SFLASHFatFs, (TCHAR const*)SFLASHPath, 0);
    require_noerr(err, exit);

    do{
        fatfs_info = fatfs_get_info((uint8_t*)SFLASHPath);
        if( fatfs_info.total_space == 0 ){
            break;
        }
        /* Write file */
        err = f_open(&SFLASHFile, filename, FA_WRITE | FA_CREATE_ALWAYS);
        require_noerr(err, exit);
        err = f_write(&SFLASHFile, wtext, sizeof(wtext));
        require_noerr(err, exit);
        byteswritten += err;
        /* Read file */
        err = f_open(&SFLASHFile, filename, FA_READ);
        require_noerr(err, exit);
        err = f_read(&SFLASHFile, rtext, sizeof(rtext));
        require_noerr(err, exit);
        bytesread += err;
        /* Print log */
        log_printf("File size: %d\n", byteswritten);
        log_printf("Read data: %s\n", rtext);
    }while(1);
}

exit:
    /* Clean up */
    /* ... */
}
```

```
os_fatfs_log("filesystem free space is %d, need format", fatfs_info.total_space);
os_fatfs_log("start format filesystem");
err = f_mkfs((TCHAR const*)SFLASHPath, 0, _MAX_SS);
}

}while( fatfs_info.total_space == 0 );

os_fatfs_log("filesystem total space is %dKB, free space is %dKB", fatfs_info.total_space,
fatfs_info.free_space);

err = f_open(&SFLASHFile, filename, FA_CREATE_ALWAYS|FA_WRITE);
require_noerr(err, exit);

err = f_write(&SFLASHFile, wtext, sizeof(wtext), (void *)&byteswritten);
if( (byteswritten == 0) || (err != FR_OK) ){
    os_fatfs_log("fatfs write error %d", err);
    goto exit;
}

os_fatfs_log("fatfs write file, name:%s, data:%s", filename, wtext);

err = f_close(&SFLASHFile);
require_noerr(err, exit);

err = f_open(&SFLASHFile, filename, FA_READ);
require_noerr(err, exit);

err = f_read(&SFLASHFile, rtext, sizeof(rtext), (void *)&bytesread);
if( (bytesread == 0) || (err != FR_OK) ){
    os_fatfs_log("fatfs read error %d", err);
    goto exit;
}

os_fatfs_log("fatfs read file, name:%s, data:%s", filename, rtext);

err = f_close(&SFLASHFile) ; /* close files*/
require_noerr(err, exit);
```

```
exit:  
    FATFS_UnLinkDriver(SFLASHPath);  
}  
  
int application_start( void )  
{  
    test_fatfs();  
    return 0;  
}
```

12.3 fatfs.c 文档参考

文件系统 APIs 详细信息, 请参考 MiCO SDK: \libraries\filesystem\FatFs\doc 中 00index_e.html 文件。

更多 FatFs 信息请参考: http://elm-chan.org/fsw/ff/00index_e.html。

13. Hardware 驱动

13.1 MiCOKit 单一外设驱动

13.1.1 ext_dc_motor.c: 直流电机

MiCO 为用户提供 MiCOKit 外设之直流电机控制 API，实现电机启动与停止功能。

本例通过调用 MiCO dc motorAPI 实现对 MiCOKit 电机的间歇性开关控制。本例程序代码如下：

```
#include "mico.h"
#include "micokit_ext.h"

#define ext_dc_motor_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)

int application_start( void )
{
    OSStatus err = kNoErr;

    /* 初始化 直流电机 */
    err = dc_motor_init();

    require_noerr_action( err, exit, ext_dc_motor_log("ERROR: Unable to Init DC motor") );

    while(1)
    {
        dc_motor_set(1); /* 直流电机 运转*/
        mico_thread_msleep(1500);
        dc_motor_set(0); /* 直流电机 停止*/
        mico_thread_msleep(4500);
    }

exit:
    return err;
}
```

13.1.2 ext_rgb_led.c: RGB 灯

MiCO 为用户提供 MiCOKit 外设 RGB 灯控制 API。实现 RGB 灯颜色，亮度，饱和度和开关控制功能。

本例通过调用 MiCO rgb led 控制 API，实现对 MiCOKit RGB 灯间歇性闪烁控制。例程代码如下：

```
#include "mico.h"
#include "micokit_ext.h"

#define ext_rgb_led_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)
```

```
#define COLOR_MODE RGB_MODE

#define RGB_MODE 0
#define HSB_MODE 1

#if (COLOR_MODE == RGB_MODE)
int application_start( void )
{
    ext_rgb_led_log("rgb led control demo(RGB_MODE)");
    /*初始化 RGB LED(P9813)*/
    rgb_led_init();
    while(1)
    {
        /*RGB LED 亮红灯,#FF0000*/
        rgb_led_open(255, 0, 0);
        mico_thread_sleep(1);
        /*RGB LED 亮绿灯 #00FF00*/
        rgb_led_open(0, 255, 0);
        mico_thread_sleep(1);
        /*RGB LED 亮蓝灯,#0000FF*/
        rgb_led_open(0, 0, 255);
        mico_thread_sleep(1);
    }
}
#else
int application_start( void )
{
    ext_rgb_led_log("rgb led control demo(HSB_MODE)");
    /*init RGB LED(P9813)*/
    rgb_led_init();
    while(1)
    {
        /* RGB 灯输出红色*/
        hsb2rgb_led_open(0, 100, 100);
        mico_thread_sleep(1);
        /* RGB 灯输出绿色*/
        hsb2rgb_led_open(120, 100, 100);
    }
}
```

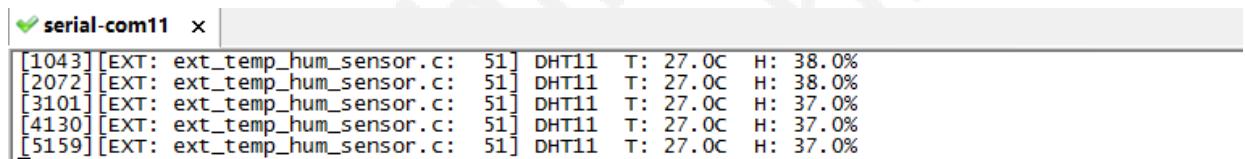
```
mico_thread_sleep(1);  
/* RGB 灯输出蓝色*/  
hsb2rgb_led_open(240, 100, 100);  
mico_thread_sleep(1);  
}  
}  
#endif
```

13.1.3 ext_temp_hum_sensor.c: 温湿度传感器

MiCO 为用户提供 MiCOKit 外设之温湿度传感器值读取 API，实现环境温湿度的检测输出功能。

本例通过调用 MiCOKit 读取温湿度传感器值 API，实现温度和湿度实时检测输出功能。

本例运行串口 log 输出如图 13.1：



```
[1043][EXT: ext_temp_hum_sensor.c: 51] DHT11 T: 27.0C H: 38.0%  
[2072][EXT: ext_temp_hum_sensor.c: 51] DHT11 T: 27.0C H: 38.0%  
[3101][EXT: ext_temp_hum_sensor.c: 51] DHT11 T: 27.0C H: 37.0%  
[4130][EXT: ext_temp_hum_sensor.c: 51] DHT11 T: 27.0C H: 37.0%  
[5159][EXT: ext_temp_hum_sensor.c: 51] DHT11 T: 27.0C H: 37.0%
```

图 13.1 温湿度检测 log

本例程序代码如下：

```
#include "mico.h"  
  
#include "micokit_ext.h"  
  
#include "temp_hum_sensor\DH1T1\DH1T1.h" /*温湿度传感器相关头文件*/  
  
#define ext_temp_hum_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)  
  
int application_start( void )  
{  
    OSStatus err = kNoErr;  
  
    uint8_t dht11_temp_data = 0;  
    uint8_t dht11_hum_data = 0;  
    err = DH1T1_Init();  
    require_noerr_action( err, exit, ext_temp_hum_log("ERROR: Unable to Init DH1T1") );  
    while(1)  
    {  
        /* DH1T1 数据读取时间间隔必须大于等于 1 秒*/  
        mico_thread_sleep(1);  
        err = DH1T1_Read_Data(&dht11_temp_data, &dht11_hum_data);  
    }  
}
```

```
require_noerr_action( err, exit, ext_temp_hum_log("ERROR: Can't Read Data") );
ext_temp_hum_log("DHT11 T: %3.1fC H: %3.1f%%", (float)dht11_temp_data, (float)dht11_hum_data);
}

exit:
return err;
}
```

13.1.4 ext_oled.c: OLDE 显示屏

MiCO 为用户提供 MiCOKit 外设之 OLDE 屏写入 API，实现用户 OLED 显示输出功能。

本例通过调用 MiCOKit OLED 的 API 函数，实现显示屏特定内容写入与显示功。

本例运行输出 log 如下，显示屏内容如下：

本例程序代码如下：

```
#include "mico.h"
#include "micokit_ext.h"

#define ext_oled_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)

int application_start( void )
{
    char time[16]={0};

    ext_oled_log("OLED control demo!");

/*初始化 光发射二极管*/
    OLED_Init();

/*在第一列起始位置显示字符串*/
    OLED_ShowString(0, 0, "MXCHIP Inc.");

/*在第二列起始位置显示字符串*/
    OLED_ShowString(0, 2, "MiCO run time:");

    while(1)
    {
        memset(time, 0, sizeof(time));

/*从 MiCO 系统一启动，就获取系统运行时间，单位 s*/
        sprintf(time, "%d ms", mico_get_time());

/*在第三列起始位置显示时间*/
        OLED_ShowString(0, 4, (uint8_t *)time);

        mico_thread_sleep(1);
    }

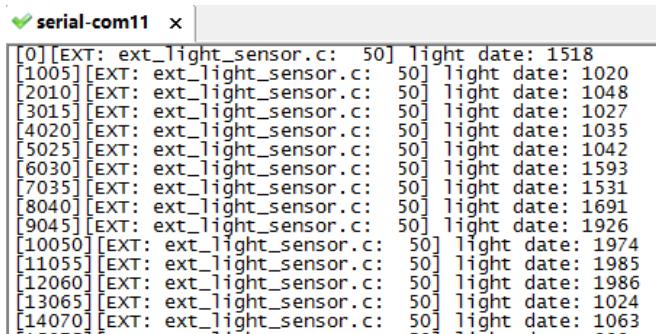
//OLED_Clear(); //清空显示屏
    return 1;
}
```

{}

13.1.5 ext_light_sensor.c: 光强传感器

MiCO 为用户提供 MiCOKit 光强传感器值读取 API，实现环境亮度的检测功能。

本例通过调用 MiCOKit 光强读取 API 函数，实现对环境亮度的检测。运行 log 如图 13.2：



```
[0][EXT: ext_light_sensor.c: 50] light date: 1518
[1005][EXT: ext_light_sensor.c: 50] light date: 1020
[2010][EXT: ext_light_sensor.c: 50] light date: 1048
[3015][EXT: ext_light_sensor.c: 50] light date: 1027
[4020][EXT: ext_light_sensor.c: 50] light date: 1035
[5025][EXT: ext_light_sensor.c: 50] light date: 1042
[6030][EXT: ext_light_sensor.c: 50] light date: 1593
[7035][EXT: ext_light_sensor.c: 50] light date: 1531
[8040][EXT: ext_light_sensor.c: 50] light date: 1691
[9045][EXT: ext_light_sensor.c: 50] light date: 1926
[10050][EXT: ext_light_sensor.c: 50] light date: 1974
[11055][EXT: ext_light_sensor.c: 50] light date: 1985
[12060][EXT: ext_light_sensor.c: 50] light date: 1986
[13065][EXT: ext_light_sensor.c: 50] light date: 1024
[14070][EXT: ext_light_sensor.c: 50] light date: 1063
```

图 13.2 光强检测 log

本例程序代码如下：

```
#include "mico.h"
#include "micokit_ext.h"

#define ext_light_sensor_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)

int application_start( void )
{
    OSStatus err = kNoErr;
    uint16_t light_sensor_data = 0;

    /*初始化光强传感器*/
    err = light_sensor_init();
    require_noerr_action( err, exit, ext_light_sensor_log("ERROR: Unable to Init light sensor") );

    while(1)
    {
        err = light_sensor_read(&light_sensor_data);/*读取光强值*/
        require_noerr_action( err, exit, ext_light_sensor_log("ERROR: Can't light sensor read data") );
        ext_light_sensor_log("light date: %d", light_sensor_data);
        mico_thread_sleep(1);
    }
exit:
    return err;
```

{}

13.1.6 ext_infrared_reflective.c：外设之红外反射传感器

MiCO 为用户提供 MiCOKit 红外反射传感器值读取 API，实现外置遮挡物对红外信号作用的检测功能。

本例通过调用 MiCOKit 红外信号读取 API 函数，实现对红外信号强弱的检测输出功能。

运行 log 如图 13.3：

```
[0] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[1007] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[2013] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[3019] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[4025] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4078  
[5031] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4079  
[6037] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[7043] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4076  
[8049] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[9055] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4077  
[10061] [EXT: ext_infrared_reflective.c: 50] infrared reflective date: 4078
```

图 13.3 红外反射传感器 log

本例程序代码如下：

```
#include "mico.h"  
  
#include "micokit_ext.h"  
  
  
#define ext_infrared_refective_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)  
  
  
int application_start( void )  
{  
    OSStatus err = kNoErr;  
  
    uint16_t infrared_reflective_data = 0;  
  
  
    /*初始化红外反射传感器*/  
    err = infrared_reflective_init();  
    require_noerr_action( err, exit, ext_infrared_refective_log("ERROR: Unable to Init infrared refective") );  
  
    while(1)  
    {  
        err = infrared_reflective_read(&infrared_reflective_data);/*读取红外反射信号值*/  
        require_noerr_action( err, exit, ext_infrared_refective_log("ERROR: Can't infrared refectiver read data") );  
        ext_infrared_refective_log("infrared reflective date: %d", infrared_reflective_data);  
        mico_thread_sleep(1);  
    }  
exit:  
}
```

```
    return err;  
}
```

13.1.7 ext_motion_sensor.c：外设之运动传感器

MiCO 为用户提供 MiCOKit 九轴运动传感器值读取 API，实现运动参数检测功能。

本例通过调用 MiCOKit 运动传感器读取 API 函数，实现对 MiCOKit 运动状态的检测。

本例程序代码如下：

```
#include "mico.h"  
  
#include "micokit_ext.h"  
  
  
#define ext_motion_sensor_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)  
  
  
int application_start( void )  
{  
    OSStatus err = kNoErr;  
    motion_data_t *motion_data;  
  
    motion_data = (motion_data_t *)malloc(sizeof(motion_data_t));  
    memset(motion_data, 0x0, sizeof(motion_data_t));  
    /*运动传感器初始化*/  
    err = motion_sensor_init();  
    require_noerr( err, exit );  
  
    while(1)  
    {  
        mico_thread_sleep(1);  
        err = motion_sensor_readout(motion_data);/*读取运动参数*/  
        require_noerr( err, exit );  
        /*输出各项运动参数项*/  
        ext_motion_sensor_log("accel x %d, y %d z %d", motion_data->accel_data.accel_datax,  
                               motion_data->accel_data.accel_datay,  
                               motion_data->accel_data.accel_dataz);  
  
        ext_motion_sensor_log("gyro x %d, y %d z %d", motion_data->gyro_data.gyro_datax,  
                               motion_data->gyro_data.gyro_datay,  
                               motion_data->gyro_data.gyro_dataz);  
    }  
exit:  
}
```

```
ext_motion_sensor_log("mag    x %d, y %d z %d", motion_data->mag_data.mag_datax,
                      motion_data->mag_data.mag_datay,
                      motion_data->mag_data.mag_dataz);
}

exit:
return err;
}
```

本例中运动传感器未在 micokit 上焊接，

13.1.8 ext_environmental_sensor.c: 环境传感器

MiCO 为用户提供 MiCOKit 环境传感器读取 API，实现运动参数的检测功能。

本例通过调用 MiCOKit 运动传感器读取 API 函数，实现对 MiCOKit 环境参数的检测。

本例程序代码如下：

```
#include "mico.h"
#include "micokit_ext.h"
#include "temp_hum_sensor\BME280\bme280_user.h"

#define ext_environmental_sensor_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)

int application_start( void )
{
    OSStatus err = kNoErr;
    int32_t bme280_temp = 0;
    uint32_t bme280_hum = 0;
    uint32_t bme280_press = 0;
    /*初始化环境传感器*/
    err = bme280_sensor_init();
    require_noerr_action( err, exit, ext_environmental_sensor_log("ERROR: Unable to Init BME280") );

    while(1)
    {
        /* BME280 Read Data Delay must >= 1s*/
        /*环境传感器 BME280 数据读取时间间隔必须大于等于 1s*/
        mico_thread_sleep(1);
    }
}
```

```
/*环境传感器值读取函数*/  
  
err = bme280_data_readout(&bme280_temp, &bme280_press, &bme280_hum);  
  
require_noerr_action( err, exit, ext_environmental_sensor_log("ERROR: Can't Read Data") );  
  
ext_environmental_sensor_log("BME280 T: %3.1fC H: %3.1f% P: %5.2fkPa",  
    (float)bme280_temp/100, (float)bme280_hum/1024, (float)bme280_press/1000);  
}  
  
exit:  
  
return err;  
}
```

13.1.9 ext_ambient_light_sensor.c:近距离环境光三合一传感器

MiCO 为用户提供 AVAGO 近距离环境光三合一传感器 APDS_9930 的检测值读取 API。

本例通过调用 APDS_9930 读取 API，实现对 MiCO 设备环境光强度及接近距离的检测。

本例程序代码如下：

```
#include "mico.h"  
  
#include "micokit_ext.h"  
  
  
#define ext_ambient_light_sensor_log(M, ...) custom_log("EXT", M, ##__VA_ARGS__)  
  
  
int application_start( void )  
{  
  
    OSStatus err = kNoErr;  
  
    uint16_t apds9930_Prox = 0;  
  
    uint16_t apds9930_Lux = 0;  
  
    /*环境传感器初始化*/  
  
    err = apds9930_sensor_init();  
  
    require_noerr_action( err, exit, ext_ambient_light_sensor_log("ERROR: Unable to Init APDS9930") );  
  
    while(1)  
    {  
  
        mico_thread_sleep(1);  
  
        /*读取传感器数据*/  
  
        err = apds9930_data_readout(&apds9930_Prox, &apds9930_Lux);  
  
        require_noerr_action( err, exit, ext_ambient_light_sensor_log("ERROR: Can't Read Data") );  
  
        ext_ambient_light_sensor_log("APDS9930 Prox: %.1fmm Lux: %d",  
            (float)(10239-apds9930_Prox)/100, apds9930_Lux);  
    }  
}
```

```

exit:
    return err;
}

```

13.2 Stmems 开发板外设控制

13.2.1 micokit_Stmems_demo.c 功能说明

本 Demo 是一个仅运行在 MiCOKit-MEMS（Nucleo）开发板上的示例程序，完成：开发板上各传感器数据在 OLED 屏上实时显示功能。运行 log，输出如下图。

实现功能：间隔约 1 秒钟，刷新传感器值，功能包含：

- (1) 温湿度，log 为：Temp: 24.3°C Humi: 58.8%
- (2) 微型压力传感器，log 为：Pressure: 1056.88m
- (3) 紫外线指数，log 为：0.0uw
- (4) 9 轴运动位置传感器，log 为：X=69 Y=-230 Z=1299 MAGX=3184 MAGY=1623 MAGZ=-2265
- (5) 光照强度传感器，log 为：Light ADC:390

```

[7824][MiCOKit_STm[21862][MiCOKit_STmems: micokit_STmems_demo.c: 78] ///////////////////////////////
[21904][MiCOKit_STmems: micokit_STmems_demo.c: 83] Temp:24.3C Humi:58.8%
[21937][MiCOKit_STmems: micokit_STmems_demo.c: 89] Pressure: 1056.88m
[21958][MiCOKit_STmems: micokit_STmems_demo.c: 95] 0.0uw
[22009][MiCOKit_STmems: micokit_STmems_demo.c: 101] X=69 Y=-230 Z=1299
[22018][MiCOKit_STmems: micokit_STmems_demo.c: 105] MAGX=3184 MAGY=1623 MAGZ=-2265
[22026][MiCOKit_STmems: micokit_STmems_demo.c: 108] Light ADC: 390
[23032][MiCOKit_STmems: micokit_STmems_demo.c: 78] ///////////////////////////////
[23074][MiCOKit_STmems: micokit_STmems_demo.c: 83] Temp:23.9C Humi:58.9%
[23107][MiCOKit_STmems: micokit_STmems_demo.c: 89] Pressure: 1056.93m
[23128][MiCOKit_STmems: micokit_STmems_demo.c: 95] 0.0uw
[23179][MiCOKit_STmems: micokit_STmems_demo.c: 101] X=3 Y=-136 Z=1293
[23188][MiCOKit_STmems: micokit_STmems_demo.c: 105] MAGX=3201 MAGY=1542 MAGZ=-2203
[23196][MiCOKit_STmems: micokit_STmems_demo.c: 108] Light ADC: 362
[24202][MiCOKit_STmems: micokit_STmems_demo.c: 78] ///////////////////////////////
[24244][MiCOKit_STmems: micokit_STmems_demo.c: 83] Temp:23.9C Humi:58.5%
[24277][MiCOKit_STmems: micokit_STmems_demo.c: 89] Pressure: 1057.13m
[24298][MiCOKit_STmems: micokit_STmems_demo.c: 95] 0.0uw
[24349][MiCOKit_STmems: micokit_STmems_demo.c: 101] X=6 Y=-122 Z=1301
[24358][MiCOKit_STmems: micokit_STmems_demo.c: 105] MAGX=3201 MAGY=1542 MAGZ=-2203
[24366][MiCOKit_STmems: micokit_STmems_demo.c: 108] Light ADC: 375
[25372][MiCOKit_STmems: micokit_STmems_demo.c: 78] ///////////////////////////////

```

13.2.2 micokit_Stmems_demo API 说明

- **1. micokit_STmems_init()**
功能：初始化 MiCOKit-MEMS 外设传感器。无返回值
- **2. OLED_Clear()**
功能：清空 OLED 屏内容
- **3. void OLED_ShowString(u8 x,u8 y,u8 *chr)**

功能：在 OLED 指定 X,Y 轴坐标处显示字符串 chr。无返回值

参数：

x: X 轴坐标参数

y: Y 轴坐标参数

chr: 要显示的字符串内容

- **4. void hsb2rgb_led_open(float hues, float saturation, float brightness)**

功能: 指定 RGB 灯色调, 饱和度, 亮度, 并开启。无返回值

参数:

hues: 指向色调的指针

saturation: 指向饱和度的指针

brightness: 指向亮度的指针

- **5. OSStatus hts221_Read_Data(float *temperature, float *humidity)**

功能: 读取 hts221 温湿度值。无错返回 0, 出错返回-1.

参数:

temperature: 指向温度值的指针

humidity: 指向湿度值的指针

- **6. OSStatus lps25hb_Read_Data(float *temperature, float *pressure)**

功能: 读取微型压力传感器 lps25hb 的压力值。无错返回 0, 出错返回-1.

参数:

temperature: 指向温度值的指针

pressure: 指向压力值的指针

- **7. OSStatus uvis25_Read_Data(float *uv_index)**

功能: 读取紫外线指数传感器 uvis25 的指数值。无错返回 0, 出错返回-1.

参数:

uv_index: 指向紫外线指数的指针

- **8. OSStatus lsm9ds1_acc_read_data(int16_t *ACC_X, int16_t *ACC_Y, int16_t *ACC_Z)**

功能: 读取 9 轴运动位置传感器 lsm9ds1 的 X,Y,Z 三轴位置的值。无错返回 0, 出错返回-1.

参数:

ACC_X: 指向 X 轴位置值的指针

ACC_Y: 指向 Y 轴位置值的指针

ACC_Z: 指向 Z 轴位置值的指针

- **9. OSStatus lsm9ds1_mag_read_data(int16_t *MAG_X, int16_t *MAG_Y, int16_t *MAG_Z)**

功能：读取 9 轴运动位置传感器 lsm9ds1 的 X,Y,Z 三轴运动加速度的值。无错返回 0，出错返回-1.

参数：

MAG_X：指向 X 轴运动加速度值的指针

ACC_Y：指向 Y 轴位置值的指针

ACC_X：指向 Z 轴位置值的指针

- **10. int light_sensor_read(uint16_t *data)**

功能：读取光照轻度传感器的光强 AD 值。无错返回 0，出错返回-1.

参数：

data：指向传感器光强 AD 值的指针

14. MiCO 低功耗蓝牙功能示例

蓝牙示例程序代码位于“demo/bluetooth/”文件夹下，在 IAR 中使用时，应首先将相应的示例代码加入软件工程中，然后将示例代码的文件夹路径增加到软件工程的搜索路径下，并选择正确的蓝牙应用框架。最后，编译成功后，下载到模块中，即可执行相应功能。详细的蓝牙应用框架使用请查看：《MiCO 蓝牙协议栈使用指南》。

在 MiCoder IDE 开发环境中使用时，用户无需关心蓝牙应用框架使用详细，只需正确书写 make 命令即可。MiCoder IDE 使用详细请参考：<http://mico.io/wiki/download> 页面中 wiki 中的文档内容：《RM1044CN_MiCoder_ToolChain_and_IDE.pdf》。

针对独立的服务端示例程序，在手机端使用相应的客户端程序进行测试，以下将以 iOS 平台上的 APP：LightBlue 为例进行说明。

14.1 ble_hello_sensor：低功耗蓝牙服务端应用

该示例演示了一个低功耗蓝牙服务端程序的实现，需要调用 bt_smart_controller 框架。示例需要配合手机蓝牙 APP 客户端：BlueDeng（Android）或者 LightBlue（Apple Store），进行使用。

本示例中，为了和 ble_hello_center 示例配合使用，两个文件中的蓝牙服务器设备名称必须一致，用户可自行修改，修改位置：文件 ble_hello_sensor.c 中第 299 行，如图 14.1。这里的设备名称为：MiCOKit Ble_Hello_Sensor。

```
298 /* Initialise MiCO Bluetooth Framework */
299 err = mico_bt_init( MICO_BT_HCI_MODE, "MiCOKit Ble_Hello_Sensor", 0, 1 ); //Client
300 require_noerr_string( err, exit, "Error initialising MiCO Bluetooth Framework" );
```

图 14.1 设备名称修改位置

蓝牙服务器设备启动后，在串口终端上显示如下信息：

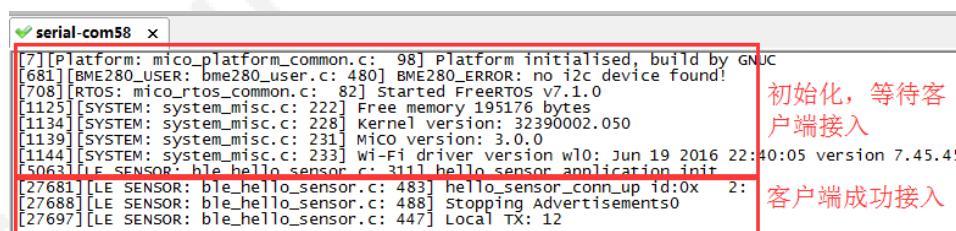


图 14.2 ble_hello_sensor 串口 log 信息

可以看到，蓝牙服务器设备会自动开启广播，使客户端设备可以扫描到。这里客户端使用手机 APP：Light Blue 进行测试。具体如图 14.3。

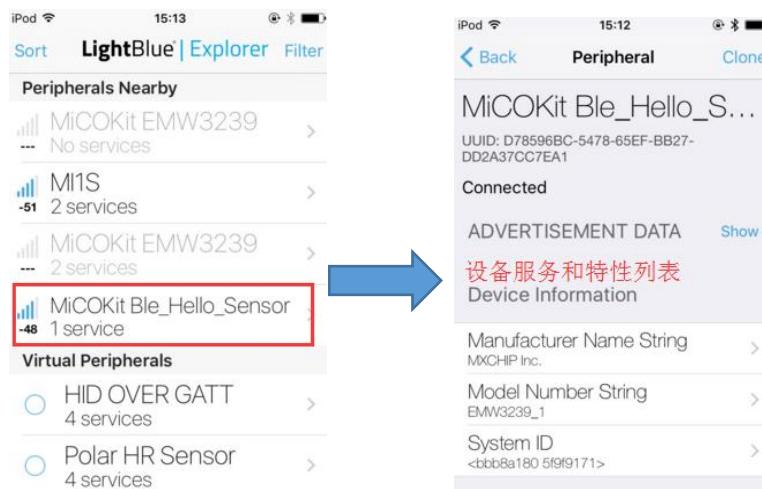


图 14.3 设备广播数据信息

14.2 ble_scan：低功耗蓝牙客户端的设备扫描

该示例构建了一个客户端程序，扫描环境中的低功耗蓝牙广播信号。支持被动扫描和主动扫描。该示例程序需要调用 bt_smartbridge 框架。

使用方法是：在串口终端上输入命令：blescan 命令来启动扫描，其中 blescan passive 表示被动扫描，blescan active 是主动扫描。以下是一个被动扫描结果的示例：其中 Advertisement data 是广播报文中的数据，按照（类型）：数据的形式罗列在下方。示例程序 ble_scan 运行 log 如图 14.4：

```

[5][Platform: mico_platform_common.c: 96] Platform initialised, build by IAR
[701][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[728] [RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1145] [SYSTEM: system_misc.c: 228] Free memory 193128 bytes
[1147] [SYSTEM: system_misc.c: 228] Kernel version: 32390002.050
[1147] [SYSTEM: system_misc.c: 231] Micro Version: 3.0.0
[1152] [SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Jun 19, 2016 22:40:05 version 7.45.45.17 (r64,
[5067] [LE SCAN: ble_scan.c: 102] Initialize success, input "blescan" to start...
blescan

start ble passive scan
# Scan complete
[4F:D3:D6:C6:E1] address type: 1 rssi: -88 name:
=>Advertisement data:
=>(_):06 13
=>(ff):4C 00 0C 0E 00 AA D5 1D E8 84 B3 0A 1A FF FE 33 DA 12 00

[6D:DE:65:C2:33:35] address type: 1 rssi: -99 name:
=>Advertisement data:
=>(_):1A 07
=>(ff):4C 00 10 02 0A 40 00

[AC:BC:32:83:AA:CE] address type: 0 rssi: -93 name:
=>Advertisement data:
=>(_):06 07
=>(ff):4C 00 10 02 0B 00 00

[56:CF:C2:5E:12:7D] address type: 1 rssi: -91 name:
=>Advertisement data:
=>(_):06 07
=>(ff):4C 00 10 02 0B 00 00

```

图 14.4 ble_scan 运行 log 信息

14.3 ble_hello_center：低功耗蓝牙客户端应用

该示例程序需要调用 bt_smartbridge 框架。该示例程序完成的功能如下：

1. 扫描环境中的低功耗蓝牙设备，寻找符合要求的 ble_hello_sensor 设备来进行连接
2. 对连接的设备进行绑定操作，并对连接进行加密
3. 对连接的设备进行服务和特性发现，并将结果保存在缓存中。下一次连接可以跳过该步骤，提高

效率

4. 向 ble_hello_sensor 中的写入数据，控制 MiCOKit 上 LED 灯的颜色
5. 在连接成功后，设置 ble_hello_sensor 的通知发送配置，并从 ble_hello_sensor 读取通知消息。

注意：该示例与 ble_hello_sensor 服务端示例程序配合，替代手机，实现了一个蓝牙客户端的功能。

首先需保证 ble_hello_sensor 和 ble_hello_center 的服务器设备名称一致，在 ble_hello_center 中需修改的代码位置在文件 ble_hello_center 中第 181 行，如图 14.5：

```

177 */ ****
178 * Variable Definitions
179 ****
180
181 const char desired_peer_device[] = "MiCOKit Ble_Hello_Sensor"; /* Name of
182 const mico_bt_uuid_t hello_service_uuid = { .len = LEN_UUID_128, .uu.uuid128 =
183 const mico_bt_uuid_t hello_color_uuid = { .len = LEN_UUID_128, .uu.uuid128 =
184 static mico_bt_smarthbridge_socket + smarthbridge_socket MAY CONCURRENT CONNECT
185

```

图 14.5 ble_hello_center 中代码修改

使用方法：

1. 在使用这个示例时，应首先运行一个 ble_hello_sensor 设备。
2. ble_hello_center 运行后，会自动完成扫描，连接，配对，加密，属性发现等操作，并最终可以看到 ble_hello_sensor 设备上的 LED 灯每一秒改变一次颜色。
3. 在 ble_hello_sensor 设备的串口终端上，输入 notify 可以向 ble_hello_center 发送消息，并显示在 ble_hello_center 的串口终端上。

```

[6] [Platform: mico_platform_common.c: 98] Platform initialised, build by GNUC
[680] [BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[707] [RTOS: mico_rtos_common.c: 82] Started FreeRTOS V7.1.0
[1145] [SYSTEM: system_misc.c: 222] Free memory 195216 bytes
[1154] [SYSTEM: system_misc.c: 228] Kernel version: 32390002.050
[1159] [SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1164] [SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Jun 19 2016 22:40:05 version 7.45.45.17 (r64, mac c
[1176] [SYSTEM: mico_system_init.c: 84] Available configuration. Starting Wi-Fi connection...
[1184] [SYSTEM: system_misc.c: 209] Connect to MX_00....
[1191] [SYSTEM: config_server.c: 148] Config Server established at port: 8000 fd: 1
[1390] [MICO:ble_wlan_coexist:cttcp_client_thread: 75] **ASSERT**
[3396] [MICO:ble_wlan_coexist:cttcp_client_thread: 75] **ASSERT**
[4372] [SYSTEM: system_misc.c: 71] Station up
[5299] [Hello Peripheral: ble_hello_peripheral.c: 156] Hello Peripheral started.
[5306] [BT Demo: ble_hello_center.c: 223] Initialize success
[5312] [BT Demo: ble_hello_center.c: 242] Scanning for MiCOKit Ble_Hello_Sensor...
[6716] [BT Demo: ble_hello_center.c: 345] Opening GATT connection to [C8:93:46:00:27:05] (addr type=Public)...
[6726] [BT Demo: ble_hello_center.c: 366] Bond info found. Encrypt use bond info.
[8463] [BT Demo: ble_hello_center.c: 374] Smartbridge connection established.


```

图 14.6 ble_hello_center 串口 log 信息

4. 与此同时，ble_hello_sensor 上的串口 log 信息如图 14.7 所示。

```

[7] [Platform: mico_platform_common.c: 98] Platform initialised, build by GNUC
[681] [BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[708] [RTOS: mico_rtos_common.c: 82] Started FreeRTOS V7.1.0
[1125] [SYSTEM: system_misc.c: 222] Free memory 195176 bytes
[1134] [SYSTEM: system_misc.c: 228] Kernel version: 32390002.050
[1139] [SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1144] [SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Jun 19 2016 22:40:05 version 7.45.45.17 ...
[5063] [LE SENSOR: ble_hello_sensor.c: 311] hello_sensor_application_init
[19014] [LE SENSOR: ble_hello_sensor.c: 483] hello_sensor_conn_up id:0x 2: 初始
[19022] [LE SENSOR: ble_hello_sensor.c: 488] Stopping Advertisements
[19030] [LE SENSOR: ble_hello_sensor.c: 447] Local TX: 12
[19697] [LE SENSOR: ble_hello_sensor.c: 598] hello_sensor_write_handler: led_color_idx: 7 与客户端连接成功
[19715] [LE SENSOR: ble_hello_sensor.c: 465] RSSI: -51
[20720] [LE SENSOR: ble_hello_sensor.c: 598] hello_sensor_write_handler: led_color_idx: 8
[20738] [LE SENSOR: ble_hello_sensor.c: 465] RSSI: -59
[21695] [LE SENSOR: ble_hello_sensor.c: 598] hello_sensor_write_handler: led_color_idx: 9
[221713] [LE SENSOR: ble_hello_sensor.c: 465] RSSI: -59
[22719] [LE SENSOR: ble_hello_sensor.c: 598] hello_sensor_write_handler: led_color_idx: 10
[22737] [LE SENSOR: ble_hello_sensor.c: 465] RSSI: -53
[23694] [LE SENSOR: ble_hello_sensor.c: 598] hello_sensor_write_handler: led_color_idx: 11

```

图 14.7 ble_hello_sensor 串口 log 信息

14.4 bt_rfcomm_server : 虚拟蓝牙串口服务器:

bt_rfcomm_server 是 EMW3239 串口仿真协议的服务端使用示例。RFCOMM 功能简单概括就是将经典蓝牙（非 BLE）仿真为一个串口，方便与其他串口类 APP 兼容。详细情况，请参考 RFCOMM_SPEC.pdf, <<https://developer.bluetooth.org/TechnologyOverview/Pages/RFCOMM.aspx>>。

示例中需要的库是 Lib_BTE_RFCOMM.Cortex-M4.IAR.release.a, 而不是 Lib_BTE.Cortex-M4.IAR.release.a。该示例完成的功能如下：

1. 使用 libraries/bluetooth 中的 API, 而不是 libraries/daemons/bt_smart 中的 API。
2. 初始化底层协议栈，并注册消息回调函数。
3. 处理各种协议栈消息，比如配对、绑定等。
4. 初始化 RFCOMM 协议栈，并注册其连接管理以及数据管理的回调函数。
5. 处理 RFCOMM 的连接管理以及数据管理类消息。
6. RFCOMM 接收 client 发送的数据并回写到 client 设备。

该示例程序功能实现操作如下（可选择用 PC 或手机作蓝牙客户端）：

- PC 作蓝牙客户端：

1. MiCO 蓝牙设备上电后，等待连接配对中，log: Waiting for RFCOMM connection (scn = 1)...;
2. PC 开启蓝牙功能，发现 3239 蓝牙设备，并与之配；
3. 此时，PC 端“设备端口”中将增加出现 2 个 COM 串口，为了确认哪个串口可传送数据，需打开 PC 端设备管理界面，进入相关设置中，更多蓝牙选项，打开 Bluetooth 设置界面，如图 14.8。

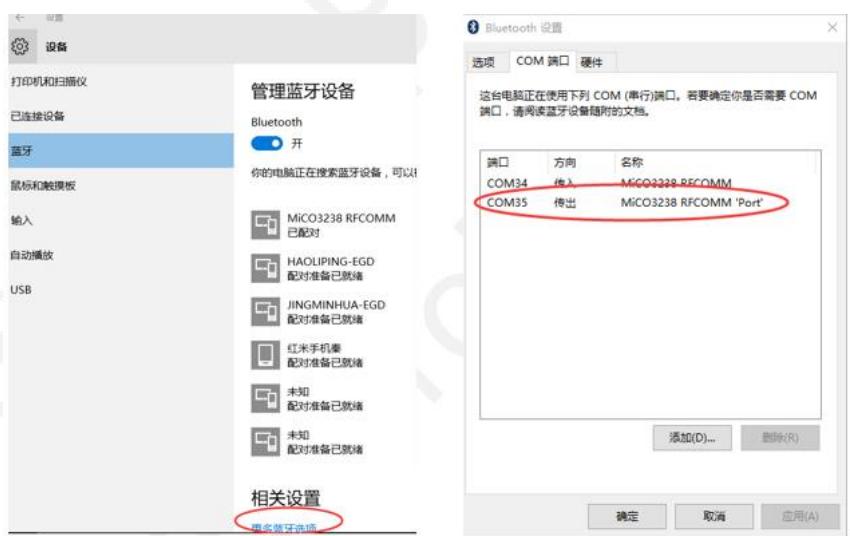


图 14.8 PC 端串口确认

其中，COM35 端口，代表本 PC 作为蓝牙 Client 端，数据传输方向为：传出，即：可以通过 PC 端串口调试工具软件，向 MiCO 蓝牙服务器端发送数据。

4. 打开串口调试工具软件，连接上述 2 个串口中方向为传出的 COM 端口，波特率：115200bps。
5. 此时，可通过串口工具向 MiCO 蓝牙设备发送数据，设备成功接收数据后，返回回传信息如下：

发送：PC Client to MiCO ble server

接收：echo from server: PC Client to MiCO ble server

- 手机作蓝牙客户端时

(目前手机测试仅支持 Android 系统, 请至应用商店 获取 “蓝牙串口” APP):

1. MiCO 蓝牙设备上电启动后, 等待连接配对中, log: Waiting for RFCOMM connection (scn = 1);
2. 手机开启蓝牙功能, 发现 MiCO 蓝牙设备, 并与之配对;
3. 配对成功后, 打开蓝牙串口 APP, 点击“连接”, 可连接到已配对的 MiCO 蓝牙设备上;
4. 此时, 通过该 APP 向 MiCO 蓝牙设备发送消息数据, 设备成功接收数据后返回回传信息如下:

蓝牙串口 APP 发送内容: mobile Client to MiCO ble server

蓝牙串口 APP 接收内容: echo from server: mobile Client to MiCO ble server

ble_rfcomm_server 运行后串口 log 信息:

```

[0] [APPLOG: bt_rfcomm_main.c: 105] BT RFCOMM SERVER DEMO
[1] [Platform: mico_platform_common.c: 98] Platform initialised, build by GNUC
[706] [BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[733] [RTOS: mico_rtos_common.c: 82] Started FreeRTOS v7.1.0
[1187] [SYSTEM: system_misc.c: 222] Free memory: 196576 bytes
[1196] [SYSTEM: system_misc.c: 228] Kernel version: 32390002.050
[1201] [SYSTEM: system_misc.c: 231] MiCO version: 3.0.0
[1206] [SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Jun 19 2016 22:40:05 version 7.45.45.17 (r64, mac C8:93:46:00:27
[5107] [APPLOG: bt_rfcomm_main.c: 154] Bluetooth enabled (success)

=====Peer Device:[74 23 44 48 67 2F ]=====
BT ER/EDR Link key:
BR/EDR KEY type :4
BR/EDR KEY :[BE 00 D6 A6 55 4B C4 F0 06 E6 79 62 77 C3 F6 DC ]
[5136] [APPLOG: bt_rfcomm_main.c: 135] Local Bluetooth Address: [c8:93:46:00:27:0f]
[5144] [APPLOG: bt_rfcomm_server.c: 128] Waiting for RFCOMM connection (scn = 1)...
[155078] [APPLOG: bt_rfcomm_main.c: 197] The Peer device IO_CAP is 0x1
[155313] [APPLOG: bt_rfcomm_main.c: 204] User confirmation request: numeric = 47734, Just works
[155413] [APPLOG: bt_rfcomm_main.c: 212] Bluetooth pairing complete, bonding_status = 0, pairing_complete_info_status = 0x0
=====Peer Device:[74 23 44 48 67 2F ]=====
BT ER/EDR Link key:
BR/EDR KEY type :4
BR/EDR KEY :[BE 00 D6 A6 55 4B C4 F0 06 E6 79 62 77 C3 F6 DC ]
[155536] [APPLOG: bt_rfcomm_main.c: 212] Bluetooth pairing complete, bonding_status = 0, pairing_complete_info_status = 0x0
=====Peer Device:[74 23 44 48 67 2F ]=====
BT ER/EDR Link key:
BR/EDR KEY type :4
BR/EDR KEY :[09 B8 B5 DB 85 28 03 83 F9 FC 1B 9C F1 A0 41 0D ]
[339488] [APPLOG: bt_rfcomm_server.c: 136] RFCOMM Connection established.
[339496] [APPLOG: bt_rfcomm_server.c: 147] RFCOMM Event: 0xc18
[370478] [APPLOG: bt_rfcomm_server.c: 156] RFCOMM RX (len = 13)
[370484] [APPLOG: bt_rfcomm_server.c: 177] 0000:
[370488] [APPLOG: bt_rfcomm_server.c: 183] 68

```

启动并等待蓝牙客户端连接

蓝牙客户端成功连接

蓝牙客户端发数据给设备

图 14.9 ble_rfcomm_server 串口 log 信息

注意：

MiCO 蓝牙设备成功配对后, 再次重启时会自动与上次配对的蓝牙客户端连接成功, 无须重复配对。

14.5 ble_advertisements: 低功耗蓝牙客户端广播

该实例程序展示了如何使用 bt_smartbridge 框架操作蓝牙设备进行不同模式的广播。蓝牙设备的广播模式主要有四种：

1. 通用广播：进行通过广播的设备能够被扫描设备扫描到，或者在接收到连接请求时作为从设备进入一个连接。
2. 定向广播：定向广播是一种为了尽可能快的建立连接而向指定设备广播自己的一种广播模式。定向广播时必须指定对端设备的地址。

3. 不可连接广播：不想被连接的设备使用不可连接广播事件。这种广播的典型应用包括设备只想广播数据，而不想被扫描或者连接。

4. 可发现广播：这种设备不能用于发起连接，但是允许其他设备扫描该设备。

本示例程序要求用户输入相应的指令触发相应的广播模式，为了简化程序，具体的广播数据不能通过指令配置，这里只是作为一种使用方式展示出来。在实际的开发中，具体的广播数据还是要根据需要填充的。本示例的广播数据一律使用默认设置，用户可以在程序代码中自行修改。

示例程序启动之后，会默认进入通用广播模式。用户可以使用指定的指令设置为不同的广播模式。用户指令如下：

1) 开始广播指令

advert start <mode> [duty]

其中， mode 必填， duty 可选填。

mode 就是广播的模式，包括：

1 – 通用广播模式, 2 – 可发现广播, 3 – 定向广播, 4 – 不可连接广播

duty 是指定广播的间隔大小，示例中采用两个默认的设置：

1 – 短间隔高负载, 2 – 长间隔低负载。

当模式为 3（定向广播）时，不必指定 duty 值，内部默认使用 high duty 方式。用户可以在代码中更改 duty 的具体间隔值，但是必须符合蓝牙协议规范的取值范围。

2) 停止广播指令

advert stop

```
[5][P]atform: mico_platform_common.c: 96] Platform initialised, build by IAR
[679][BME280_USER: bme280_user.c: 480] BME280_ERROR: no i2c device found!
[706][RTOS: mico_rtos_common.c: 82] Started FreeRTOS 7.1.0
[1113][SYSTEM: system_misc.c: 222] Free memory 193128 bytes
[1120][SYSTEM: system_misc.c: 228] Kernel version: 32390002.050
[1125][SYSTEM: system_misc.c: 231] MICO version: 3.0.0
[1131][SYSTEM: system_misc.c: 233] Wi-Fi driver version wlo: Jun 19 2016 22:40:05 version 7.45.45.17 (r64, ma
[5058][LE ADVERT: ble_advertisements.c: 433] Advertisements Started successfully! Arguments: 开始广播
[5062][LE ADVERT: ble_advertisements.c: 436]
[5071][LE ADVERT: ble_advertisements.c: 444]
[5075][LE ADVERT: ble_advertisements.c: 445]
[5084][LE ADVERT: ble_advertisements.c: 447]
[5091][LE ADVERT: ble_advertisements.c: 449]
[5100][LE ADVERT: ble_advertisements.c: 451]
[5107][LE ADVERT: ble_advertisements.c: 453]
[5116][LE ADVERT: ble_advertisements.c: 455]
[5123][LE ADVERT: ble_advertisements.c: 456]
[33206][LE ADVERT: ble_advertisements.c: 473] Advertisements Stopped successfully! 广播结束
[33214][LE ADVERT: ble_advertisements.c: 478]
[33218][LE ADVERT: ble_advertisements.c: 274] Connection established! id: 0x2
[33225][LE ADVERT: ble_advertisements.c: 275] 建立连接
```

图 14.10 ble_advertisements 运行 log 信息