

# 【富芮坤物联网开发板评测】富芮坤 BLE5.0 开发板-应用实例：智能全彩 LED 鱼缸灯



andery88

2020-2-6 20:24:44 只看该作者 倒序浏览

阅读：6876 回复：14

面包板社区的各位伙伴，大家春节好！

在这个本该喜庆的日子里，大家不免被新型冠状病毒的阴影笼罩着，为奋斗在一线的医护人员点赞！

各位伙伴也都在家为社会做着贡献！

我也和大家伙一样在家待着，实在闲来无趣，我就一直盯着我们家的鱼缸看，回想起去年元旦新购的 10 余条金鱼，由于刚开始养金鱼，没有什么经验，现在的鱼缸就只剩下两条较大个的金鱼了，来上图。



而沉在鱼缸底部的全彩 LED 灯由于红外遥控器丢失（遥控器太小）一直无法打开，没了灯光的陪衬，夜晚的鱼缸暗淡了许多。

就在这时，突然想到春节前刚刚收到的来自面包板社区的【富芮坤物联网开发板 BLE5.0】评测活动而获得的低功耗蓝牙 V5（BLE5.0）开发板，想利用此开发板来作为控制板，对原有的采用红外线+解码器的全彩 LED 灯控制方案改为【富芮坤 BLE5.0 开发板】+全彩 LED 驱动板的控制方案，此应用场景取名为：【智能全彩 LED 鱼缸灯】。

说干就干！

俗话说：知己知彼，百战不殆！

首先，先弄清楚【富芮坤物联网 BLE5.0 开发板】上都有什么硬件，这决定着我用那些端口去驱动全彩 LED 灯。

首先，查看【富芮坤开发板】文件夹里的【开发板 使用手册】/【BLE5.0 开发板】/【FR8016HA 开发板使用手册 V1.1.pdf】。

开发板使用手册具体文件下载地址可参考：

<https://pan.baidu.com/share/init?surl=91JY-x2G7rWIVebdVoafww>，提取码：zdov。

经过学习开发板使用手册，获得【富芮坤物联网 BLE5.0 开发板】的硬件与软件相关信息如下：

（由于个人能力有限，下面也是根据笔者自己的理解对此芯片进行评估，有不对的地方还请伙伴们批准指正，共同学习！）

【芯片型号】：FR8016H

该芯片为 SOC 芯片（system on chip），芯片内含有 ARM 主控芯片、电源管理芯片、音频解码芯片、低功耗蓝牙 5.0 通信芯片等。

在软件上，该芯片的蓝牙协议栈建立在一个微型嵌入式操作系统上，也许是 uCOSII，笔者在之前的小型嵌入式系统中曾在基于 STM32F103 芯片上移植成功过 uCOSII，只记得 uCOSII 可同时处理多个并行任务，但对于 ROM 存储容量受限的普通 MCU 芯片来说，uCOSII 程序本身占用的存储空间比较大，这就造成移植后真正留给应用程序的空间会很小，但反观此款 SOC 芯片，掉电不丢失存储部件由两部分构成：ROM 和 Flash，其中，ROM 的大小为：128kB，主要用于存储启动代码、BLE controller 部分协议栈等，这部分存储容量软件开发者是无法使用的，而开发者可以使用的为 Flash，主要用于存储用户程序、用户数据等，其大小为：512kB 或者 1MB 之多，由于存储空间大，所以对开发者来说不用担心芯片程序容量不够的情况发生。

程序在运行时使用的内存即 RAM，用于存储程序运行时的各种变量、堆栈等，也是在运行

OS 时容易饱和的地方，此 SOC 芯片的 RAM 容量为 48kB，较常规的 20kB 的容量大了一倍多，由于笔者此次开发的程序比较简单，还无法遇到 RAM 的瓶颈，不过笔者从此开发板自带的名为【ble\_simple\_peripheral】例子来看，其具有音频（.wav）解码的功能，众所周知，音频解码的过程是需要较大 RAM 的，其能够顺利完成音频解码说明其内部 RAM 足够常规程序使用。

SDK 具体文件下载地址可参考：<https://gitee.com/freqchip/FR801xH-SDK>

说完了存储空间，再说一下主频，此 SOC 支持的最大主频为：48MHz，其它可设置的主频为：6MHz、12MHz、24MHz，说实话此 SOC 主频与常规 MCU 的 72MHz 主频相比并不高，但是如果从此 SOC 的定位来看就很容易理解了：BLE（Bluetooth Low Energy，低功耗蓝牙），内含电源管理芯片，且协议栈中具有单独电源管理的驱动函数，所有的这一切都是为了节能，直接对标智能移动物联网终端领域，例如：血压计、温湿度传感器、门禁开关、红外线传感器、儿童智能积木玩具、烟雾传感器等，而这些领域也都是 BLE 的优势领域，况且 48MHz 的主频已经能够满足绝大部分的需求场合。

废话不多说，直接上我整理的 **FR8016H 芯片外围管脚及功能表**：

管脚功能	管脚名称	芯片管脚序号	可扩展功能
I/O 管脚-PA	PA0	30	PWM0,LCD_RST
	PA1	29	PWM1,PASD(音频功放使能),GINT,待用
	PA2	28	PWM2,U0RXD,(串口)
	PA3	27	PWM3,U0TXD,(串口)
	PA4	1	PWM4,SPI_CLK,LCD_CLK,(LCD)
	PA5	2	PWM5,SPI_CS,LCD_CS,(LCD)
	PA6	32	SCL,SPI_DO,LCD_SDA,(LCD)
	PA7	31	SDA,SPI_DI,LCD_D/C,(LCD)
I/O 管脚-PC	PC5	26	KEY1,待用
	PC6	25	SCL,待用,(温湿度、气压)
	PC7	24	SDA,待用,(温湿度、气压)
I/O 管脚-PD	PD4	23	ADC0,U1RXD,PWMA,待用
	PD5	22	ADC1,U1TXD,PWMB,待用
	PD6	21	ADC2,KEY2,QDO,PWMC,SHTA,待用,(H/T)
	PD7	20	ADC3,QAQ,PWMD,SHTR,待用,(H/T)
晶振管脚	XO	7	/
	XI	8	/
RF 蓝牙天线管脚	RF	5	/
GND 管脚	GND	6	/
	PGND	33	/
LED2 电源指示管脚	LED2	3	LCD_BLK(复位时为低电平，复位后为高电平)
MIC 管脚	MIC_BIAS	9	/
	MIC_IN	10	待用(麦克风输入)
	VMID	11	/
扬声器管脚	AOP	12	/
	AON	13	/
复位管脚	RSTP	14	KEY1
5V 充电管脚	VCHG	15	/(接 5V 充电接口)
3V3 电池管脚	VBAT	16	/(接 3V3 电池)
	BSW	17	/
	BFB	18	/
	LDO_OUT	19	BLEO 3V3
NC 管脚	NC	4	/



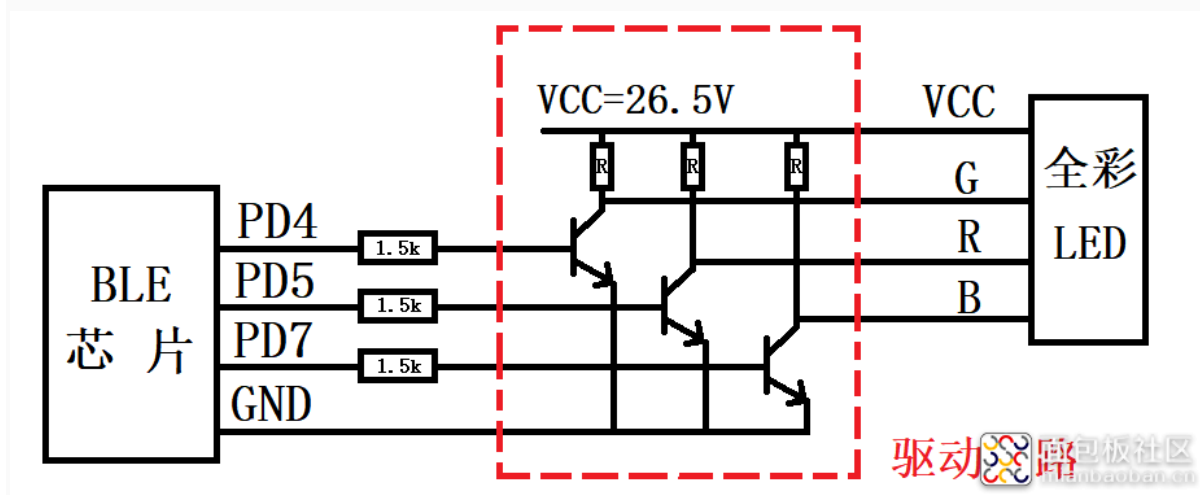
上表是我刚开时由于对此 SOC 硬件结构不太理解整理的，后来发现每个管脚的硬件功能是可配置的，例如上表中的 PD4、PD5、PD6、PD7 可以配置成 PWM 功能、IIC 功能、SPI 功能、UART 功能、PDM 功能等，硬件管脚与端口功能之间在硬件上是通过一个矩阵表进

行映射的，此表可在此 SOC 配套的 SDK 的【driver\_iomux.h】中找到，程序开发者可根据应用需求自由支配。

上表中凡带有颜色的均表示此管脚已在此开发板有用途，为了给此应用后续的升级（例如：增加鱼缸周围环境的温湿度监测、大气压监测通过蓝牙回传至手机并存储等功能）留有余地，经过分析，采用 PD4、PD5、PD7（后来深入了解发现，温湿度传感器并未使用此管脚作为 SHTR，所以这里可以放心使用，而且还不用担心后续应用升级时的温湿度读取）作为外部控制功能引脚，用于控制全彩 LED 鱼缸灯，以实现智能全彩 LED 鱼缸灯的设计。

这里首先将本应用的原理框图画一下：

（原理图是我用画图自己手绘的，仅用说明原理，驱动电路仍沿用原来全彩 LED 鱼缸灯的，不专业，不喜勿喷！）



上图中间红色虚框处即为全彩 LED 驱动电路。

在这里说明一下，全彩 LED 的类型有许多种，我所知道的大致有两大类，一类是接口有三个引脚：GND、VCC、Din(Dout 作为输出引脚，暂且不计算在内)，控制时仅需要 1 个 I/O 引脚即可，另一类是接口有四个引脚：共阴类则是 GND、G、R、B，共阳类则是：



VCC、G、R、B，经过反复测试发现我鱼缸里原有全彩 LED 灯的类型属于：共阳 GRB 类，如上图所示有四个引脚：VCC、G、R、B 来驱动。

全彩 LED 灯带的每个灯泡里分别含有红灯、绿灯、蓝灯各一个，每个子灯分别由 R、G、B 三个引脚分别控制亮度，又有不同亮度的红、绿、蓝灯同时点亮而形成五颜六色的颜色。

为了能够控制每个子灯的亮度，这里采用 PWM 波输出的方式，通过实时调节 PWM 波占空比的大小来控制 BLE 芯片 I/O 管脚输出电压有效值的大小，进而可以近似连续地控制全彩 LED 灯子灯的亮度，G、R、B 三路同时控制，进而实现 LED 的全彩点亮。

接下来我将用于配置 G (PD4)、R (PD5)、B (PD7) 各管脚的程序代码粘贴如下：

```
// 使能PWM
system_set_port_mux(GPIO_PORT_D, GPIO_BIT_4, PORTD4_FUNC_PWM4); //Green control
system_set_port_mux(GPIO_PORT_D, GPIO_BIT_5, PORTD5_FUNC_PWM5); //Red control
system_set_port_mux(GPIO_PORT_D, GPIO_BIT_7, PORTD7_FUNC_PWM1); //Blue control
pwm_init(PWM_CHANNEL_4, 10000, 50); //PWM4:10kHz、50%占空比
pwm_init(PWM_CHANNEL_5, 10000, 50); //PWM5:10kHz、50%占空比
pwm_init(PWM_CHANNEL_1, 10000, 50); //PWM1:10kHz、50%占空比
```



上述 PWM 使能及初始化程序是写在了【proj\_main.c】的

【user\_entry\_before\_ble\_init(void)】函数里，建议伙伴们把硬件赋能、初始化等程序都放在这里哦！

说到这里，我在编写此 PWM 初始化代码时死活找不到 pwm\_init() 函数，后来发现 SDK 文件下的 components 里包含所有的库函数，包括：driver\_pwm.h，后来在 proj\_main.c 及其他程序文件中添加 driver\_pwm.h 头文件，并且在 uV5 里将 driver\_pwm.c 添加到 driver 组里后，pwm 相关函数就可以随意使用了。

对了，在调用常规 PWM 驱动函数的时候，我还发现一类 pmu\_pwm 的函数，按照官方的说法，这叫【低功耗 PWM 接口】，后来发现不仅仅是 PWM 接口，任何其他通信接口均对应一个低功耗函数，这也印证了前面所说的该 SOC 是面向低功耗领域的移动终端应用领域的智能控制芯片。不过我的智能全彩 LED 鱼缸灯由于需要的电量比较大，需要通过市电经变压器实时供电，所以，这里我也就没必要采用低功耗函数了，但是这里给伙伴们提个建议，如果您将该 SOC 运行到移动终端等需要低功耗的领域的应用，建议每个端口均采用的对应的低功耗接口，只有这样，系统中定义低功耗模式才会真正起作用，由于时间原因，我也没有对其中的低功耗函数做过多研究，在这里抛砖引玉，希望看到伙伴们在低功耗方面的更好的作品。

PWM 使能和初始化完毕之后，我又写了一个全彩 LED 灯上电后红、绿、蓝三颜色灯自动依次点亮后熄灭的程序，目的是在上电时监测全彩 LED 的每个子灯是否完好，这部门程序的代码我写在了【ble\_simple\_peripheral.c】的【simple\_peripheral\_init(void)】函数里，建议伙伴们把应用级别的初始化程序都放在这里哦！当然，这里我还是偷懒了，其实应该把 ble\_simple\_peripheral.c 给替换掉，替换成例如

【Intelligent\_Fullcolor\_Fishbowl\_LED】，说到这里的意思是这里的 ble\_simple\_peripheral.c 文件属于应用层文件，当然应用程序文件名也应该与应用相匹配吧！所以这里建议伙伴们把应用程序都放在 ble\_simple\_peripheral.c 或者伙伴们自己新建的应用程序文件里。



上述全彩 LED 灯自检程序如下：

```
//全彩LED 5050 GRB依次点亮绿、红、蓝各3s后依次熄灭
pwm_start(PWM_CHANNEL_4); //点亮绿灯
co_delay_100us(10000); //延时1s
pwm_stop(PWM_CHANNEL_4); //熄灭绿灯
pwm_start(PWM_CHANNEL_5); //点亮红灯
co_delay_100us(10000); //延时1s
pwm_stop(PWM_CHANNEL_5); //熄灭红灯
pwm_start(PWM_CHANNEL_1); //点亮蓝灯
co_delay_100us(10000); //延时1s
pwm_stop(PWM_CHANNEL_1); //熄灭蓝灯
```



哈哈，刚刚发现上述程序中的备注写错了，不应是 3s，应该是 1s，刚开始写程序时的确是 3s，后来发现程序在初始化的过程中时间太长，要等待 9s 以上，如此，也将降低智能全彩 LED 鱼缸灯的用户体验，所以后来就改为了 1s，绿、红、蓝各点亮 1s 后熄灭，整个初始化过程 3s 多一点，不仅完成了自检，而且用户等待时间也不会太长。

全彩 LED 自检完后，该干什么了呢？

当然是去响应接收来自手机端的蓝牙指令了！

响应接收来自手机端的蓝牙指令程序我放在了【simple\_gatt\_service.c】的

【sp\_gatt\_write\_cb(...)】函数里了，当蓝牙模块接收到来自手机端的控制指令后，此 SOC 软件系统会自动调用该函数，接收到来自手机端的控制指令就存储在指针 write\_buf 所指向的 RAM 里，程序会根据蓝牙 UUID 的不同，自动将其 copy 到对应的变量里，在 BLE 协议栈里，有服务、特征的概念，每个服务都有一个 UUID，每个特征也有一个 UUID，一个服务下可以有多个特征，我这里选择了具有【写功能】（对手机端（central 端）而言）的特征 3 的值：sp\_char3\_value，BLE 接收到的控制指令均存放在 sp\_char3\_value 数组里。

对于服务和特征等的概念伙伴们可能不太理解，这里贴一张笔者整理的

【simple\_gatt\_service】服务里的【Attribute（属性）表】的其中一部分：

（这里先声明一下，我是个 BLE 新手，因为此活动才临时学习的 BLE 协议栈的，理解有不对的地方还请伙伴们批评指正！）

Attribute（属性）表			
SP_IDX_SERVICE	服务声明		
	UUID	2800	主要服务声明
	权限	只读	
	UUID 长度	2 字节	
	属性值	sp_svc_uuid	0xffff0
SP_IDX_CHAR1_DECLARATION	特征 1 声明		
	UUID	2803	特征声明
	权限	只读	
	UUID 长度	0	
	属性值	NULL	
SP_IDX_CHAR1_VALUE	特征 1 值		
	UUID	FFF1	特征 1
	权限	读和写	
	UUID 长度	SP_CHAR1_VALUE_LEN	10 字节
	属性值	NULL	
SP_IDX_CHAR1_USER_DESCRIPTION	特征 1 描述		
	UUID	2901	特征用户描述
	权限	只读	
	UUID 长度	SP_CHAR1_DESC_LEN	17 字节
	属性值	sp_char1_desc	



上图中，红颜色的部分代表的就是服务：SP\_IDX\_SERVICE（16 位 UUID=0xffff0），紧接着绿色、蓝色和黄色这三个部分共同组成了 UUID=0xffff0 的服务下的一个特征，而一个特征又有三部分组成：绿色部分的特征声明（SP\_IDX\_CHAR1\_DECLARATION）、蓝色部分的特征值（SP\_IDX\_CHAR1\_VALUE）以及黄色部分的特征描述

（SP\_IDX\_CHAR1\_USER\_DESCRIPTION），其中的特征值就是 GATT 服务所提供的通信数据

内容，此次【智能全彩 LED 鱼缸灯】应用所使用的是 UUID=0xfff0 服务下的 UUID=0xfff3 的特征，其值对应于【SP\_IDX\_CHAR3\_VALUE】所对应的数组变量【sp\_char3\_value】，也就是上面所说的特征 3 的值。

我刚开始直接读 BLE 协议栈理论的时候更难理解，这里我通过解析【属性表】并以表格的形式向伙伴们展示 BLE 协议栈中服务与特征之间的关系，希望对伙伴们加快理解 BLE 协议有所帮助。

好了，关于 GATT 服务就先说到这里，下面我将解析来自手机端发来的控制指令的程序贴出来，供伙伴们参考：

```
//对sp_char3_value数组中的数据进行解析[0]:0xa5,[1]:开关,[2]:Red(0xff),[4]:Green(0xff),[6]:Blue(0xff),[9]:0x5a
if(sp_char3_value[0] == 0xa5 && sp_char3_value[9] == 0x5a)//包首尾正确
{
    if(sp_char3_value[1] == 1)//灯打开
    {
        if(PWM_Start_Flag == false)
        {
            pwm_start(PWM_CHANNEL_4);//点亮绿灯
            pwm_start(PWM_CHANNEL_5);//点亮红灯
            pwm_start(PWM_CHANNEL_1);//点亮蓝灯
            PWM_Start_Flag = true;
        }
        if(sp_char3_value[2] > 99) sp_char3_value[2] = 99;//红色
        if(sp_char3_value[4] > 99) sp_char3_value[4] = 99;//绿色
        if(sp_char3_value[6] > 99) sp_char3_value[6] = 99;//蓝色
        uint32_t Green_Value = sp_char3_value[4];
        uint32_t Red_Value = sp_char3_value[2];
        uint32_t Blue_Value = sp_char3_value[6];
        pwm_update(PWM_CHANNEL_4, 10000, Green_Value);//更新绿灯亮度
        pwm_update(PWM_CHANNEL_5, 10000, Red_Value);//更新红灯亮度
        pwm_update(PWM_CHANNEL_1, 10000, Blue_Value);//更新蓝灯亮度
    }
    else
    {
        if(PWM_Start_Flag == true)
        {
            pwm_stop(PWM_CHANNEL_4);//熄灭绿灯
            pwm_stop(PWM_CHANNEL_5);//熄灭红灯
            pwm_stop(PWM_CHANNEL_1);//熄灭蓝灯
            PWM_Start_Flag = false;
        }
    }
}
```



至此，所有需要我在 peripheral 端（此 SOC 端）编写的程序编写完毕，伙伴们一定惊讶此应用程序如此之短。

是啊，我还特地数了一下，此应用程序需要开发者编写的程序有效行仅 36 行，就这 36 行就实现了一个比较复杂的蓝牙通信控制终端。

当然，其实如果从头至尾程序都要开发者自己编写这样功能的程序，估计至少要成百上千行代码吧，之所以我用了极少的代码编写出了一个功能相对完备的 BLE peripheral 端应用程序，是建立在富芮坤完备的 SDK 基础之上，官方提供了完备的库函数供开发者调用，完全不用将大把时间花在具体驱动细节上，这样的开发模式我喜欢！

对了，看到这里，伙伴们也许会说我忽悠的吧，没看到实物啊！

接下来，我逐一秀一下我的作品细节：

作品整体图：



来，给控制盒一个特写：



哈哈，这是我翻箱倒柜找到的一个还算比较合适的旧手机包装盒做的，我是业余的，伙伴们凑合着看吧！不喜勿喷！



来，打开盒子看看都有什么：



嗨！我本来想让此图片横放，但设置了半天也没弄过来，就这样凑合着看吧！

图中最核心的就是富芮坤提供的 BLE5.0 开发板了，开发板的上边是一个 AC220V/50Hz 转



DC26.5V 的电源模块，还有一个全彩 LED 驱动模块，当初只顾安装了，忘了给它拍照了 o(╯▽╰)o。具体原理呢就请伙伴们参考我上边手绘的原理图吧！

对了，BLE5.0 的开发板也是由全彩 LED 驱动模块里的一个 DC5V 电源供通过电池接口供电的，此时，【BT 蓝牙芯片电源供电】跳帽应该选择【BAT】（外部电池供电），当然了我这里不是电池，而是 DC5V 电源模块 o(\*￣▽￣\*)o。

说了半天 BLE 的 peripheral 端，那么 central 端我是如何实现的呢？

手机下载“蓝牙调试器”，APP 的 Logo 长这个样子：



这是一个免费软件，除了打开 APP 时有一次广告推送外，其它都十分好用，真心感谢此 APP 的开发者，可以让蓝牙（BLE 版）初学者不写一行代码就可以控制蓝牙模块，赞！在这里也推荐伙伴们调试用哦！

此软件进入后先设置拟用蓝牙模块数传的 UUID，我设置的如下图所示：

# 未连接设备

Tx: 0B/s

Rx: 0B/s

Err: 0B/s

支持所有蓝牙透传模块，如遇问题请修改BLE的UUID配置



Simple Per

10:80:F0:D0:AD:BD



## 选择BLE透传参数

透传服务 UUID

0000fff0-0000-1000-8000-00805f9b34fb

透传TX特征的UUID (模块->手机 方向)

0000fff2-0000-1000-8000-00805f9b34fb

透传RX特征的UUID (手机->模块 方向)

0000fff3-0000-1000-8000-00805f9b34fb

由于不同公司推出的BLE蓝牙串口模块实现透明传输的策略有所差异，因此如果出现通信失败的情况时，请耐心等待查看蓝牙模块对应的说明书，调整通信的参数。

确定



设备连接



对话模式



专业调试



按钮控制



设置



伙伴们，看到了吗？这里第一 UUID 就是服务 UUID=0xffff0，这里说一下，UUID 是一个 128bits 长的二进制数，全球唯一，但对于蓝牙协议数据来说，其余位数的数字均一致，变的就这 16 位（也不绝对哦，如果用户自己闭环使用，UUID 的 128bits 数值任君使用，但通用性就荡然无存了，这也就失去了 UUID 的意义）。0xffff0 服务就是上面所说到的 SP\_IDX\_SERVICE 服务（即：simple\_gatt\_service 服务）。UUID=0xffff3 的特征存储有此应用要使用到的控制命令。UUID=0xffff2 是用于手机端接收 peripheral 端发回的命令的，目前暂未使用，主要是由于笔者能力有限、时间有限，还没搞明白如何使 peripheral 端主动发送命令（Notification 方式或者是 Indication 方式，前者无需 central 端应答，而后者需要），希望后续的伙伴们加油能给我以启发。

下面上图——控制主界面：

Tx: 0B/s Rx: 0B/s Err: 0B/s



开关:0

红色:0



绿色:3



蓝色:0

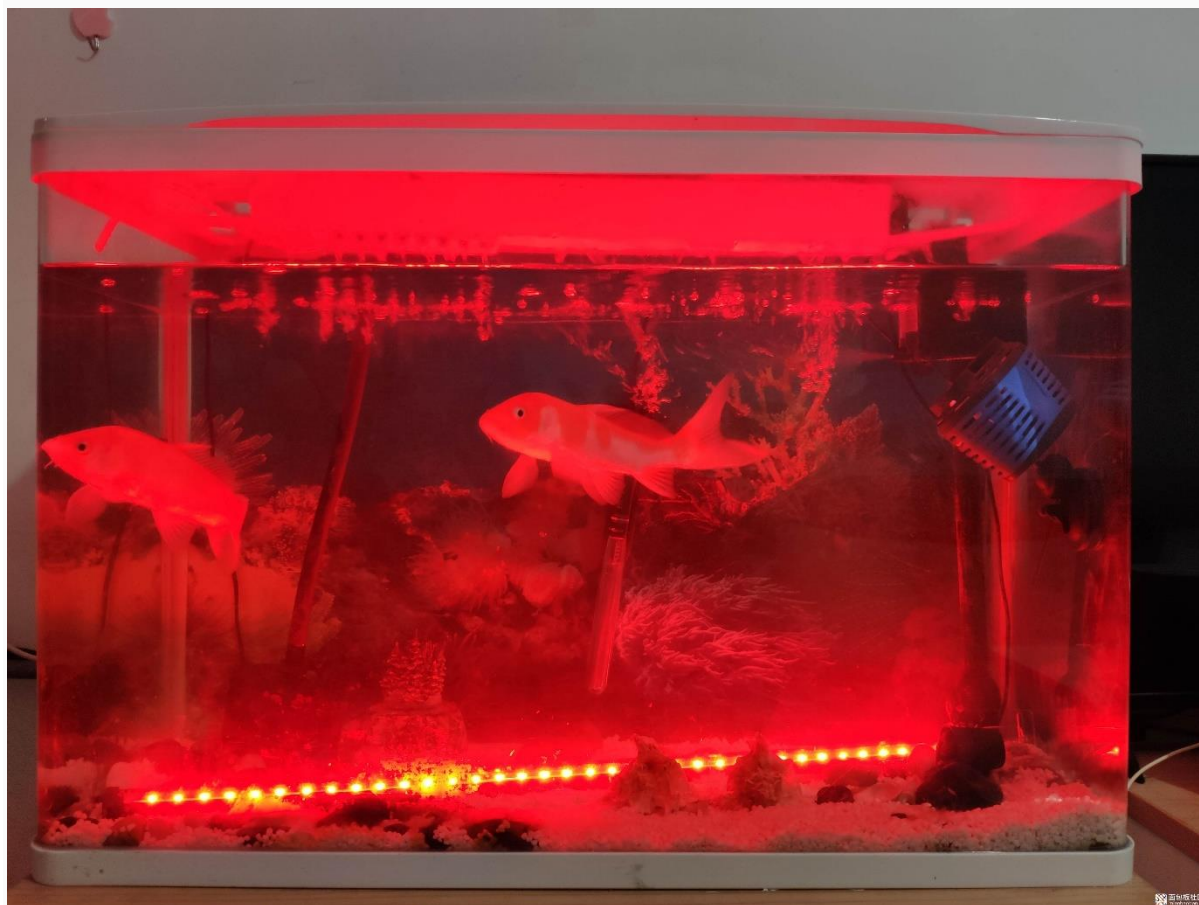


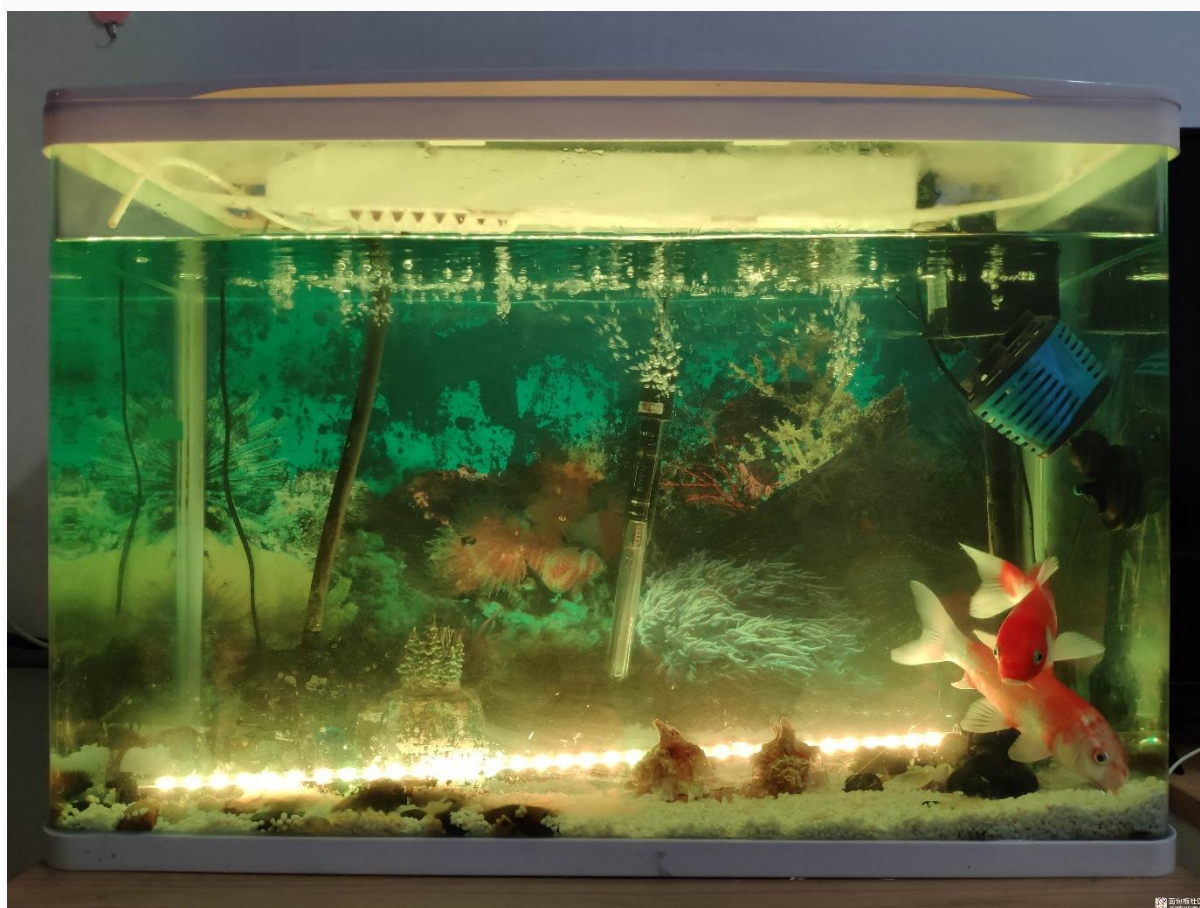
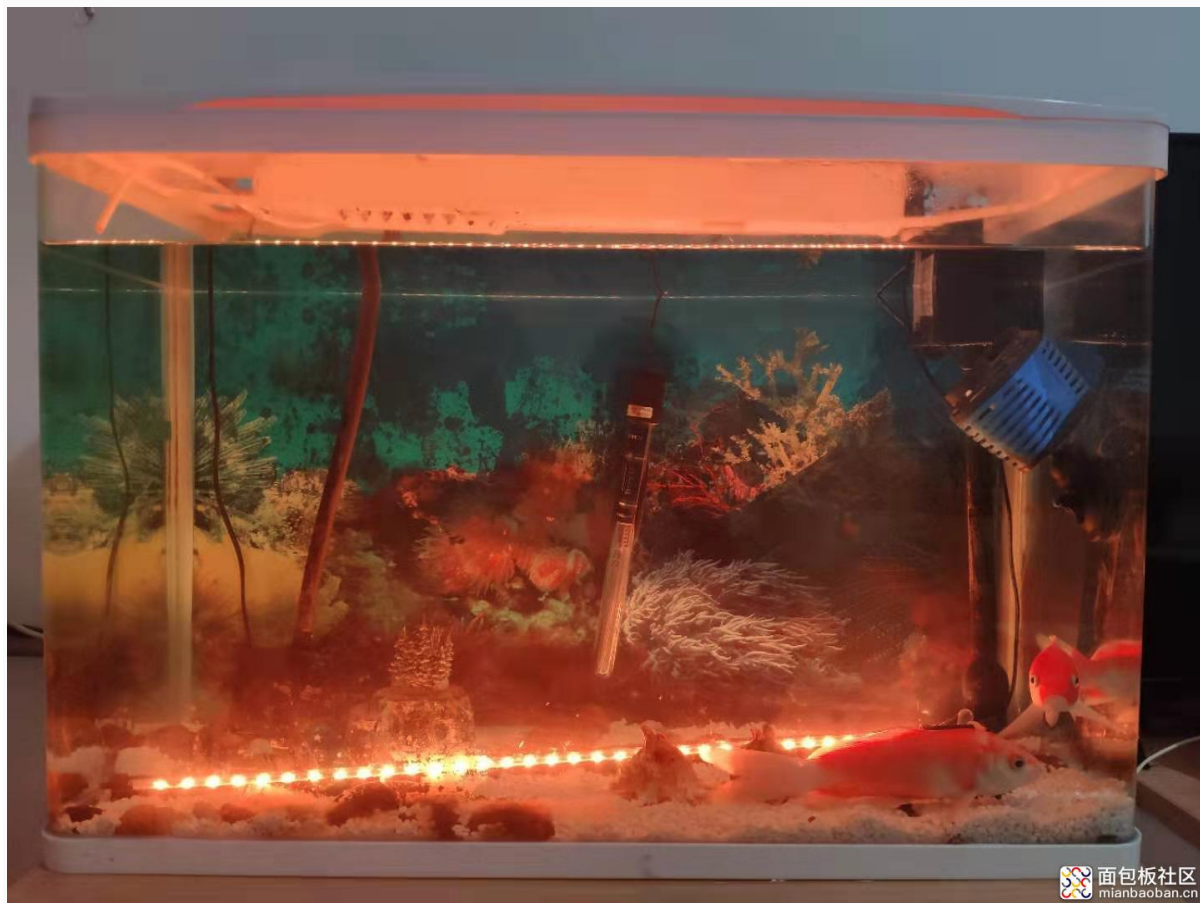


至于此 APP 如何使用我在这里就不详述了，否则就跑题了，感兴趣的伙伴可以自己看其帮助，里面有很详细的指导步骤，而且也不复杂。

最后，贴个我体验此应用：**【智能全彩 LED 鱼缸灯】**的视频吧！

最后的最后，奉上笔者鱼缸七彩图（无美颜，不喜勿喷 0(n\_n)0 哈哈~）：













好了，集齐 7 张彩图就可以... 分享啦！

对了，还没结束，末了，我还要分享一下我在制作此应用中发现的此开发板设计不合理的地方：

此开发板的蓝牙天线旁边设计了一个安装孔，手里有开发板的伙伴们估计也都看到了，起初我是用四个螺丝通过四个金属杆固定在外壳上的，但在安装后发现蓝牙信号变得十分微弱，有时手机与蓝牙模块距离很近也连接不上，后来试着将靠近蓝牙天线的螺母拧掉，信号明显好了些，但感觉仍没有单板是信号强，搜索时仍不容易搜索到，后来索性把该定位孔下的金属杆也撤了，再试发现蓝牙信号钢钢的，连接无阻，测试了 20 米的距离仍能正常通信，不错！不错！

后续的伙伴们一定要注意哦！此定位孔不要使用，及时要用也要用绝缘材料滴！

笔者是一个标准化工作者，文章末尾以标准的结束符结尾吧！

**祝大家身体健康！万事如意！**

# 【富芮坤物联网开发板评测】富芮坤 BLE5.0 开发板-应用实例：智能环境监测仪



andery88

2020-2-25 10:27:24 只看该作者 倒序浏览

阅读：3693 回复：8

本帖最后由 andery88 于 2020-2-25 17:30 编辑

各位伙伴们，距离我上次【智能全彩 LED 鱼缸灯】

的帖子已经有半月有余，受新型冠状病毒的影响，笔者虽已开工，但是单位为了降低办公的密集度，所以仍有一半时间是在家办公，为了充实宅在家的时间，笔者又基于富芮坤 BLE5.0 开发板开发了一个新的应用：【智能环境监测仪】。

首先，祝愿大家：身体健康！

好了，言归正传，上回说到：【智能全彩 LED 鱼缸灯】的应用其实是展示了富芮坤 SOC 芯片 FR8016H 核心功能：低功耗蓝牙的数据传输之 peripheral 端接收功能，即富芮坤 SOC 芯片作为蓝牙的外围端，由手机作为中央端，然后，由手机端软件单方向地向富芮坤 SOC 芯片端发送全彩 LED 鱼缸灯控制指令（数据：手机-->SOC），SOC 端通过解析手机端发来的信息来控制全彩 LED 鱼缸灯的开、关及各颜色的度。

可是，如何实现富芮坤 SOC 芯片端向手机端回传信息呢（数据：SOC-->手机），想着想着突然冒出个想法：即通过富芮坤 BLE5.0 开发板上原有的温湿度传感器获取环境的温度和湿度信息，通过 MQ-135 空气质量传感器获取空气质量信息，通过光传感器获取环境光照强度信息，这些环境信息组合在一起，然后通过低功耗蓝牙功能回传至手机端，并在手机端显示，达到环境数据的获取和监



测功能，这也是本应用名称【智能环境监测仪】的初衷。

说干就干！

不过通过什么样的方式，才能使 BLE 外围端设备向中心端设备回传数据呢？

对此，分析了 BLE 协议中 GATT 服务操作的 5 种类型：

操作类型	功能说明
Write/写	允许 GATT 客户端写入一个值到 GATT 服务器的一个特性中，有应用层上的确认和回应。
Write without Response/没有回应的写	允许 GATT 客户端写入一个值到 GATT 服务器的一个特性中，没有应用层上的确认和回应。
Read/读	一个 GATT 客户端可以读取在 GATT 服务器中特性的值。
Notify/通知	允许 GATT 服务器在其某个特性改变的时候对 GATT 客户端进行提醒，无应用层上的确认
Indicate/指示	允许 GATT 服务器在其某个特性改变的时候对 GATT 客户端进行提醒，有应用层上的确认

当然，上述 GATT 服务的 5 种操作类型对应着 GATT 服务的 5 种权限，在富芮坤 SOC 的 BLE5.0 协议栈里定义了这 5 种权限：

权限类型	富芮坤 SOC 的 BLE5.0 协议中的定义字符
Writable/可写	GATT_PROP_WRITE、GATT_PROP_AUTHEN_WRITE
Readable/可读	GATT_PROP_READ、GATT_PROP_AUTHEN_READ
Can send Notification/可发通知	GATT_PROP_NOTI
Can send Indication/可发指示	GATT_PROP_INDI

据此，笔者梳理了富芮坤 SOC 的 dev1.0\ble\_simple\_peripheral\simple\_gatt\_service.c 里定义的 GATT 服务配置表 ( simple\_profile\_att\_table ) 里所有特征值的权限：

特征值序号	对应特征的 UUID	权限
SP_IDX_CHAR1_VALUE	0xfff1	GATT_PROP_READ、GATT_PROP_WRITE/可读、可写
SP_IDX_CHAR2_VALUE	0xfff2	GATT_PROP_READ/可读
SP_IDX_CHAR3_VALUE	0xfff3	GATT_PROP_WRITE/可写
SP_IDX_CHAR4_VALUE	0xfff4	GATT_PROP_WRITE、GATT_PROP_NOTI/可写、可发通知
SP_IDX_CHAR5_VALUE	0xfff5	GATT_PROP_AUTHEN_READ、GATT_PROP_AUTHEN_WRITE/可读、可写

从上述表中可以清晰地看出在 0xfff0 服务下的各特征值的权限。

笔者上一篇【智能全彩 LED 鱼缸灯】的帖子中，控制全彩 LED 鱼缸灯的指令就是发送至 0xfff0 服务的 0xfff3 特征的特征值里，因其具有【可写】权限，所以，可以通过手机向该特征值写入控制指令数据，然后，在 sp\_gatt\_write\_cb ( ) 回调函数里对手机发来的指令数据进行解析，从而达到利用手机控制全彩 LED 鱼缸灯的目的。

有点扯远了，回到本帖的应用【智能环境监测仪】，其需要由 SOC 端向手机端不停地回传环境数据而不需手机端给其发任何指令，手机端则只需不断地接收并处理数据即可。

那么，上表中的 0xfff4 特征就是一个很好的选择，因其具有【Notify】功能，即 GATT 服务(SOC 端)可向 GATT 客户端 ( 手机端 ) 发送数据而不需要确认或回应。


伙伴们一定会说：说了半天也看不明白，还不如直接上代码来的直接！

好勒！上代码！

首先，笔者在富芮坤 SOC 的 dev1.0\ble\_simple\_peripheral\simple\_gatt\_service.c 文件的 sp\_gatt\_write\_cb ( ) 函数里的 if(uuid == GATT\_CLIENT\_CHAR\_CFG\_UUID)判断里增加如下代

码：

```
if (uuid == GATT_CLIENT_CHAR_CFG_UUID)
{
    co_printf("Notification status changed\r\n");
    if (att_idx == SP_IDX_CHAR4_CFG)
    {
        sp_char4_ccc[0] = write_buf[0];
        sp_char4_ccc[1] = write_buf[1];
        co_printf("Char4 ccc: 0x%x 0x%x \r\n", sp_char4_ccc[0], sp_char4_ccc[1]);
        if(sp_char4_ccc[0] == 1 && sp_char4_ccc[1] == 0)
        {
            sp_svc_ntf_flag = true;
            //在接收到对端ntf使能的消息之后，通过调用gatt_notification()函数实现BLE peripheral端向手机端发送数据
            ntf_att.att_idx = SP_IDX_CHAR4_VALUE;//0xffff0服务的第4个att:0xffff4的特征值序号
            ntf_att.conidx = conn_idx;//链接号
            ntf_att.svc_id = sp_svc_id;//此GATT服务号
            ntf_att.data_len = 6;//Notification数据的长度，根据发送的数据实时调整
            uint8_t ntf_data[] = "NTF OK";//虚拟数据，用户可根据需要替换成需要Notification的数据
            ntf_att.p_data = ntf_data;//Notification数据的指针
            gatt_notification(ntf_att);//peripheral端设备向手机端执行一次notification操作；
            co_printf("Notification Data: %x\r\n", ntf_data);
        }
        else
        {
            sp_svc_ntf_flag = false;
        }
    }
}
```



通过上述代码实现对 BLE 通知格式的初始化，当然这里的 BLE 通知数据结构 ntf\_att 和通知使能标志位 sp\_svc\_ntf\_flag 要定义为全局变量，以便应用层函数的调用。


然后，在应用层文件【ble\_simple\_peripheral.c】里的 simple\_peripheral\_init()函数里增加如下代码：

```
//电池电压读取初始化
memset((void*)&cfg_vbat, 0, sizeof(cfg_vbat));//为ADC配置结构体分配内存
cfg_vbat.src = ADC_TRANS_SOURCE_VBAT;//设置ADC源为电池电压
cfg_vbat.ref_sel = ADC_REFERENCE_INTERNAL;//设置ADC参考电压为内部参考电压
cfg_vbat.int_ref_cfg = ADC_INTERNAL_REF_1_2;//设置ADC内部参考

//空气质量QM-135初始化
memset((void*)&cfg_QAO, 0, sizeof(cfg_QAO));
cfg_QAO.src = ADC_TRANS_SOURCE_PAD;//设置ADC源为SOC的pin管脚
cfg_QAO.ref_sel = ADC_REFERENCE_AVDD;//设置ADC参考电压为VDD
//adc_channel = ascii_strn2val((const char *)&data[0], 16, 2);
cfg_QAO.channels = adc_channel_QAO;//低4位的每一位代表一个通道:ADC3
cfg_QAO.route.pad_to_sample = 1;

//光照强度LUX初始化
memset((void*)&cfg_LUX, 0, sizeof(cfg_LUX));
cfg_LUX.src = ADC_TRANS_SOURCE_PAD;//设置ADC源为SOC的pin管脚
cfg_LUX.ref_sel = ADC_REFERENCE_AVDD;//设置ADC参考电压为VDD
cfg_LUX.channels = adc_channel_LUX;//低4位的每一位代表一个通道:ADC2
cfg_LUX.route.pad_to_sample = 1;

//OS Timer
//os_timer_init(&timer_refresh,timer_refresh_fun,NULL);//创建一个周期性1s定时的系统定时器
//os_timer_start(&timer_refresh,1000,1);
os_timer_init(&ble_ntf_timer_refresh,ble_ntf_timer_refresh_fun,NULL);//创建一个周期性定时的系统定时器
os_timer_start(&ble_ntf_timer_refresh,1000,1);//启动定时器（用于BLE Notification）每1秒执行一次ble_ntf_timer_refresh函数
```



上图主要功能是定义了一个周期为 1s 的时钟，以便将富芮坤 SOC 获取的环境数据每隔 1s 向手机端发送一次，在其上几行代码是分别为【电池电压】、【空气质量】、【光照强度】而做的初始化工作。

这里需要重点说明一下，这里的 adc\_channel\_DAQ 赋值为 8，对应的二进制为 1000（ADC3，ADC2，ADC1，ADC0），所以，这里选择的的就是 ADC3 通道，而我正是使用富芮坤 SOC 芯片

FR8016H 的 PD7 引脚与 MQ-135 空气质量传感器的 AOUT 引脚相连，同时，根据要求，4 个 ADC 通道中每次仅能选择一个，所以 ADC 通道的赋值与 ADC 通道直接的关系如下表：

ADC 通道赋值 十进制（二进制）	ADC 通道选择	对应 SOC 引脚
1（0001）	ADC0	PD4
2（0010）	ADC1	PD5
4（0100）	ADC2	PD6
8（1000）	ADC3	PD7

然后，在应用层文件【ble\_simple\_peripheral.c】里定义一个定时时钟回调函数：void

ble\_ntf\_timer\_refresh\_fun(void \*arg)，函数里的代码如下：

```
void ble_ntf_timer_refresh_fun(void *arg)
{
    //Notification消息变量
    uint8_t ble_ntf_buff[30]; //定义一个足够大的数组存放Notification数据包
    uint8_t ble_ntf_length = 0;

    //SHT30数据读取
    int32_t temperature, humidity;
    float temperature_sh = 0, hmdity_sh = 0;
    int8_t ret=0;
    ret = sht3x_measure_blocking_read(&temperature, &humidity); //Read temperature humidity

    //读取电池电压
    uint16_t result_vbat; //结果(份数, 中间变量)
    uint16_t vbat_vol; //结果(mv)
    float vbat_vol_v;

    adc_init(&cfg_vbat); //ADC初始化
    adc_enable(NULL, NULL, 0); //使能ADC
    adc_get_result(ADC_TRANS_SOURCE_VBAT, 0, &result_vbat); //读取数据
    vbat_vol = (result_vbat * 4800) / 1024;
    vbat_vol_v = (float) vbat_vol/1000.0;
    co_printf("Vbat_mV Value: %d\r\n", vbat_vol);

    //空气质量数据读取
    uint16_t result_QAO;
    float QAO_ppm;
    uint16_t QAO_ppm1000; //1000倍的ppm
    uint16_t Volt_QAO_mv; //QAO实际输出电压(mV)
    float Volt_QAO_V; //QAO实际输出电压(V)

    adc_init(&cfg_QAO);
    adc_enable(NULL, NULL, 0);
    adc_get_result(ADC_TRANS_SOURCE_PAD, 0x01, &result_QAO);
    co_printf("QAO Value: %d\r\n", result_QAO);
    Volt_QAO_mv = (result_QAO * 3300) / 1024; //把ADC值转化为电压数值
    co_printf("QAO Volt_Value(mV): %d\r\n", Volt_QAO_mv);
    Volt_QAO_V = (float) Volt_QAO_mv / 1000.0;
    QAO_ppm = (float) pow(11.5428 * 35.904 * Volt_QAO_V / (25.5-5.1* Volt_QAO_V), 0.6549); //ppm计算公式
    QAO_ppm1000 = (uint16_t) QAO_ppm * 1000;
    co_printf("QAO_ppm*1000 Value: %d\r\n", QAO_ppm1000);

    //光照强度数据读取
    uint16_t result_LUX;
    uint8_t LUX_Value;

    adc_init(&cfg_LUX);
    adc_enable(NULL, NULL, 0);
    adc_get_result(ADC_TRANS_SOURCE_PAD, 0x01, &result_LUX);
    co_printf("LUX Value: %d\r\n", result_LUX);
    LUX_Value = 100 - result_LUX * 100 / 400;
    co_printf("LUX Value(%): %d%%\r\n", LUX_Value);
```



```

if (ret == STATUS_OK)
{
    co_printf("BLE Notification: temperature = %d,humidity = %d\r\n",temperature,humidity);
    temperature_sh = temperature/1000.0;
    hmidity_sh = humidity/1000.0;
}
else
{
    temperature_sh = 0;//温湿度读取错误，温度赋值为0
    hmidity_sh = 0;//温湿度读取错误，湿度赋值为0
}

if(sp_svc_ntf_flag)//如果Notification使能
{
    //在接收到对端ntf使能的消息之后，通过调用gatt_notification()函数实现BLE peripheral端向手机端发送数据
    /*****开始Notification数据打包（温度、湿度、空气质量和电池电压）*****/
    startValuePack(ble_ntf_buff);//开始打包，将发送缓冲数组指针传入
    //将要回传的值放入，注意顺序为Bool->Byte->Short->Int->Float
    putByte(LUX_Value);//光照强度
    putFloat(temperature_sh);//温度
    putFloat(hmidity_sh);//湿度
    putFloat(QAO_ppm);//空气质量
    putFloat(vbat_vol_v);//电池电压
    ble_ntf_length = endValuePack();//结束打包，并返回包的长度
    /*****Notification数据打包结束*****/
    ntf_att.data_len = ble_ntf_length;//Notification数据的长度，根据发送的数据实时调整
    ntf_att.p_data = ble_ntf_buff;//Notification数据的指针
    gatt_notification(ntf_att);//peripheral端设备向手机端执行一次notification操作；
    co_printf("Notification Data OK!\r\n");
}
}

```



好了，至此，所有码砖工作已经完毕。

看了上述应用程序的代码，伙伴们也许会注意到，在最后将环境数据打包时用到了几个函数：startValuePack()、putByte()、putFloat()、endValuePack()，这些函数是为了将环境数据打包而用的，手机端软件配套使用一款名为【蓝牙调试器】的免费

APP，具体长什么样子可参考我上一个帖子【智能全彩 LED 鱼缸灯】，这里就不赘述了，经过简单的设置后就可以在手机端接收到来自富芮坤 SOC 端【通知】的环境数据了。

当然，这里为了方便伙伴们使用，特将这些打包函数所需要的驱动文件整理出来，供伙伴们使用，如下为打包驱动（valuepack.h 和 valuepack.c）下载地址：

<https://pan.baidu.com/s/10aM9Z341zQIFkZpsIYJvEw>

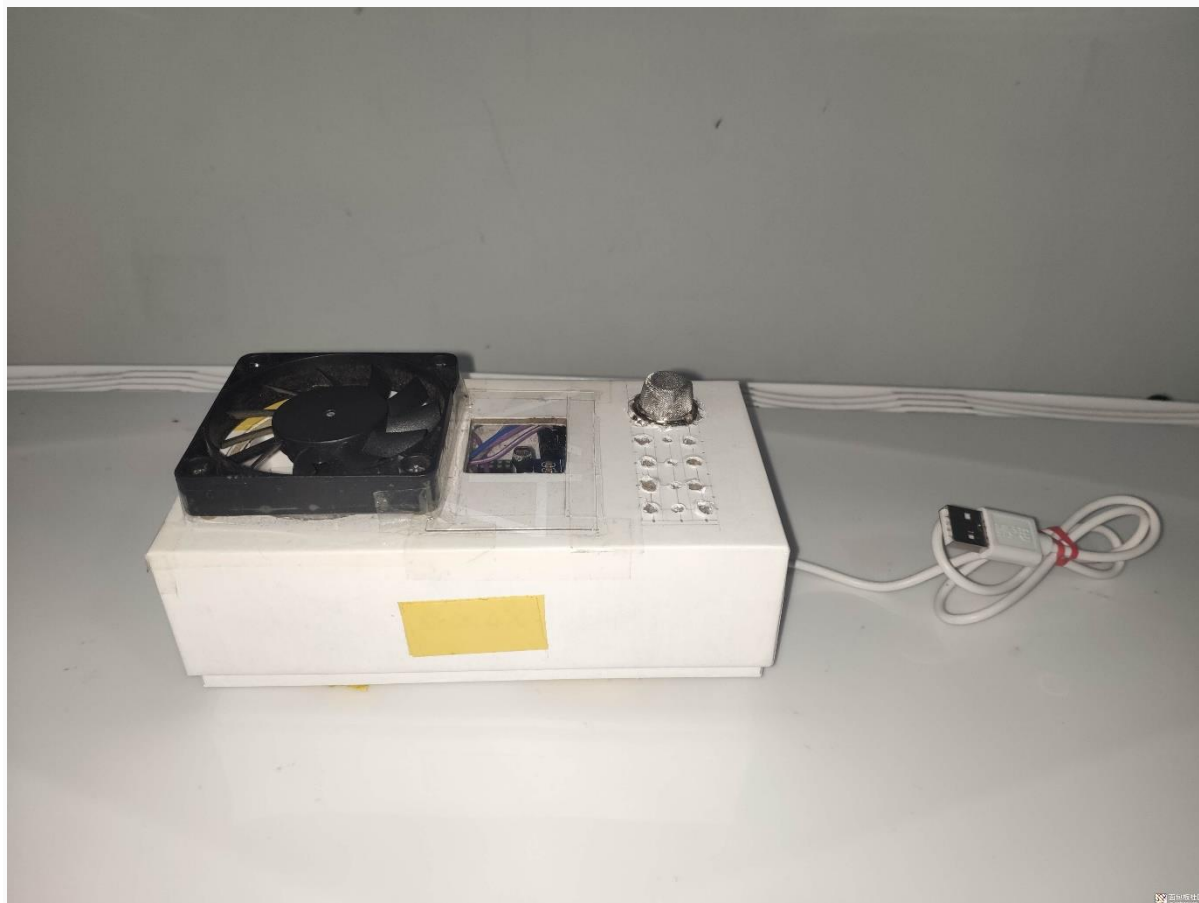
提取码：**v35c**

到这里，有伙伴问了，说了半天也没见到实物，硬件是啥样子啊！

好勒！马上上图：



智能环境监测仪硬件实物图：



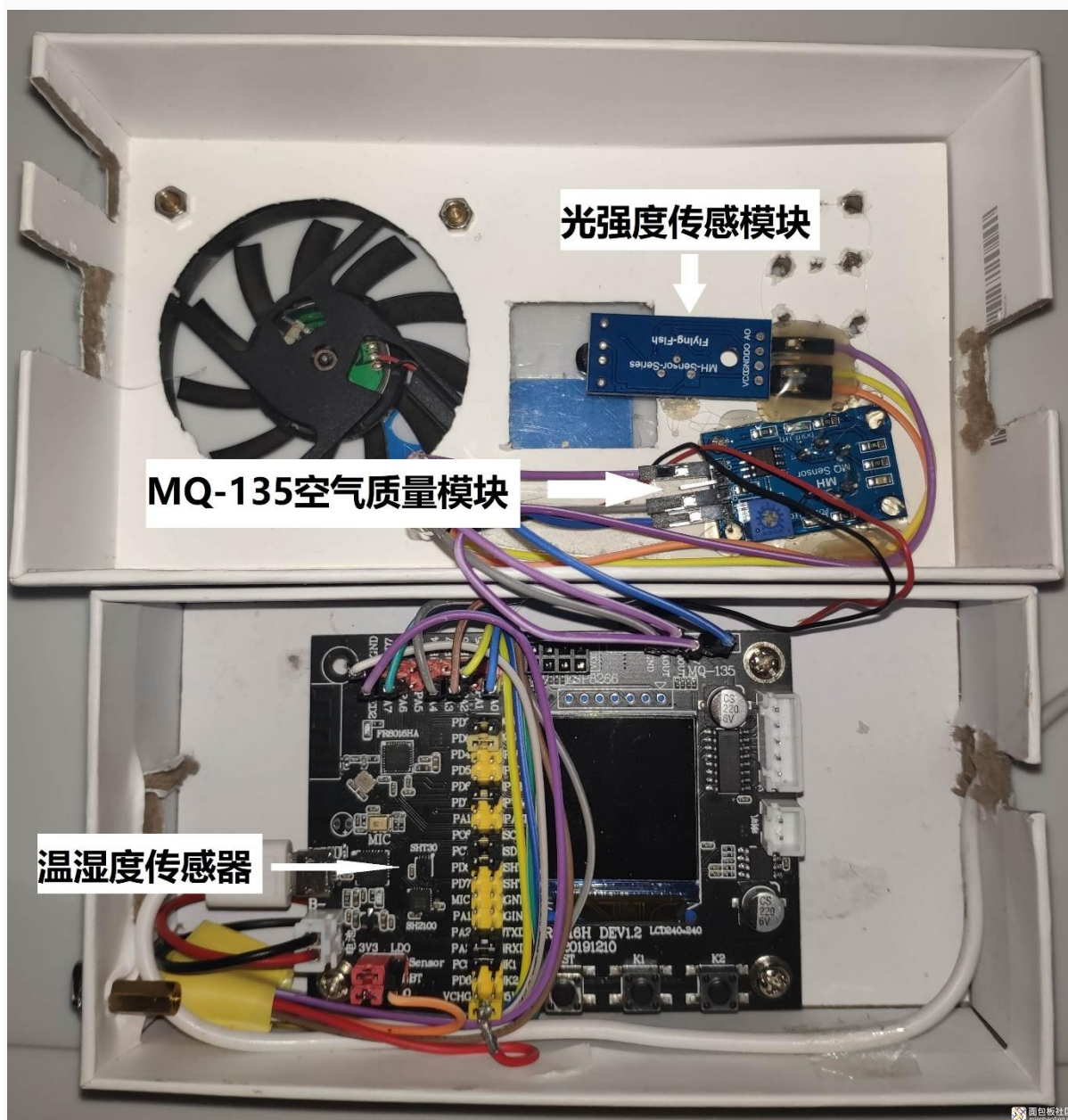
笔者怎么看怎么像一只小老鼠，在此也祝伙伴们鼠年大吉！身体健康！也愿新冠疫情早早结束！

再来一张正面的：

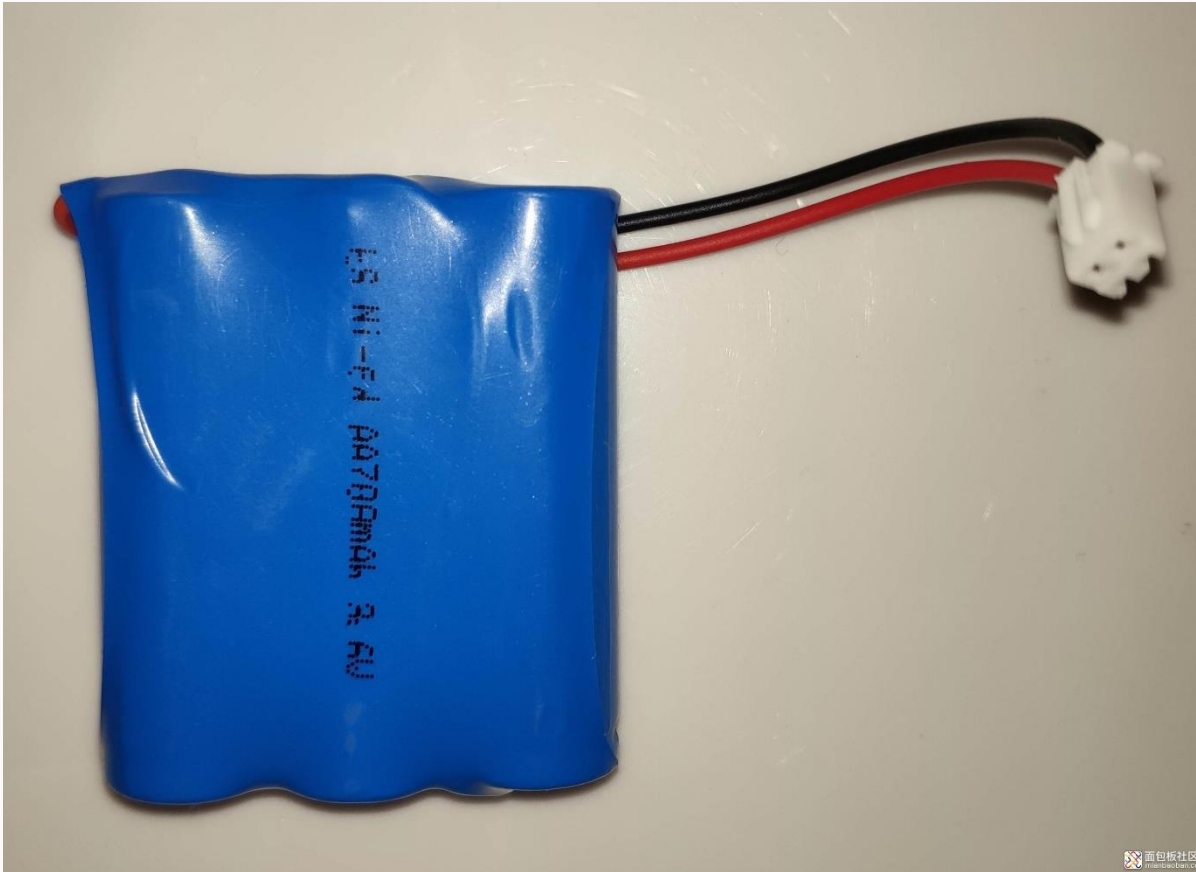




再来一张内部照片：



在上图富芮坤 BLE5.0 开发板下还有一块如下图所示的 3.3V 的可充电电池模块：

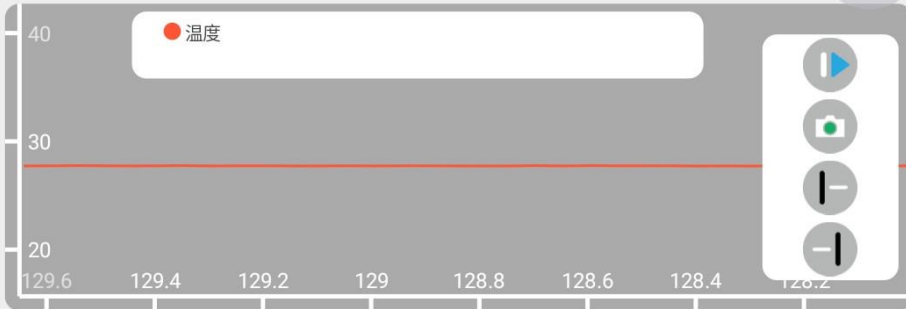


因为 MQ-135 空气质量传感器和光照强度传感器均需 5V 电压才能工作，所以，当外部 5V USB 供电断开时，这两个传感器就无法正常工作了，但温湿度传感器和 SOC 芯片均能够正常工作，且能够正常的回传温湿度和电池电压的数据。

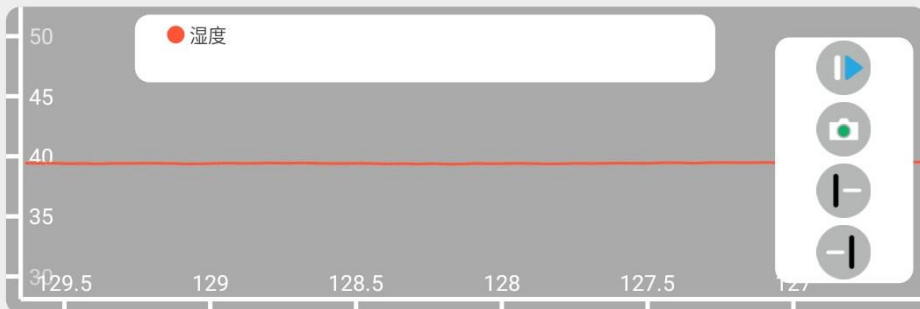
接着放一张手机端接收环境数据的界面：

Tx: 0B/s Rx: 0B/s Err: 0B/s

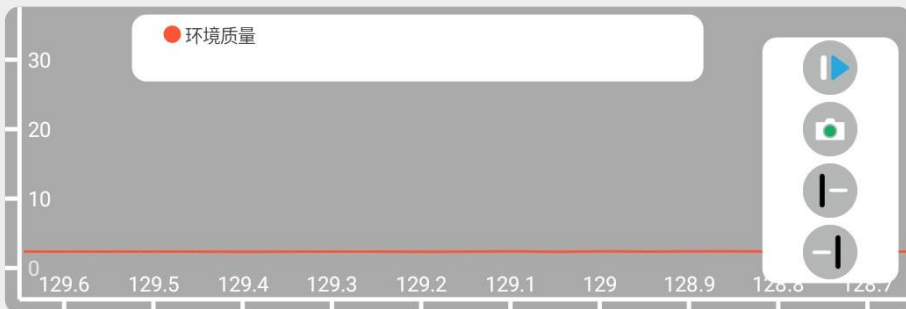
温度 : 27.73



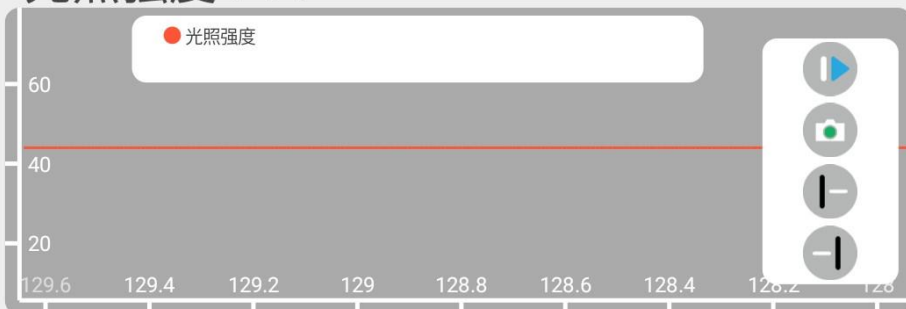
湿度 : 39.433



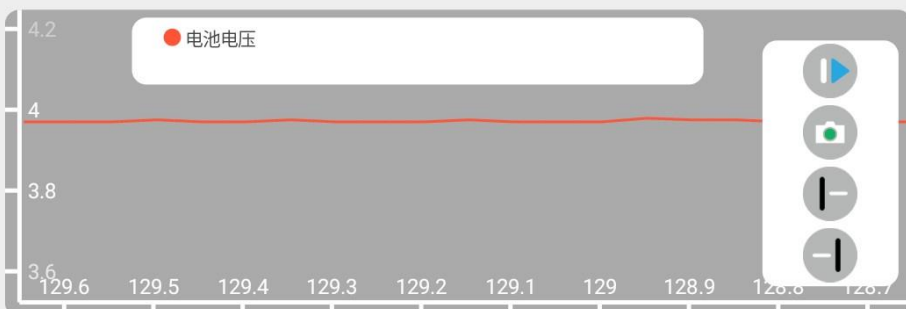
环境质量 : 2.365



光照强度 : 44



电池电压 : 3.97



手机端界面由 5 个文本框和 5 个曲线图组成，分别用于显示富芮坤 SOC 端的温度、湿度、空气质量、光照强度和电池电压数据。

最后，贴个我体验此应用【智能环境监测仪】的视频吧！

```
<iframe height=498 width=510 src='http://player.youku.com/embed/XNDU2MTM0NzIwNA==' frameborder=0  
'allowfullscreen'></iframe>
```

祝大家身体健康！万事如意！

---

软件编程, 智能手机, 智能家居, 智能硬件, 无线环境监测