Doctors of Intelligence &Technology(DOIT)

W800 开发套件鸿蒙系统教程

（文档版本：V1.0 软件版本：V 1.0 硬件版本：V 1.0）

2022-04-21

目录

一、 w800开发套件及鸿蒙系统介绍

源码下载：链接：https://pan.baidu.com/s/1q2asbei_QsZs867eSIVxfw 提取码：z53u

1.1. 开发板硬件资料分配

w800 的 IO 口和外接设备的连接关系，其中外接设备包括：显示屏、按键、RGB 彩灯、温度传感器、光敏传感器、USB 串口。通过板上的拨码开关可以控制板上不可插拔的设备，板上不可插拔的设备是指：彩灯、温度传感器、光敏传感器和 USB 口。

1.2 供电

开发板可以通过 USB 接口。板上集成有AMS1117-3.3V 稳压芯片，把输入的5V 电压稳压到3.3V， 用于给板上的设备和模块都是供电。

1.3 USB 接口

USB 接口有三种功能：

- 供电：只要插上USB 线到USB 接口即可。
- 下载：USB 下载会有单独的一节说明。
- 串口调试：串口调试在实验中有说明。

注：当使用 USB 接口进行调试或者程序下载的时候，拨码 开关 5 和 6 必须打开。

1.4. 按键

开发板一个复位键用于复位 W800 系统。点击下载软件复位后会进入下载模式。

1.5. RGB 彩灯

W800 的底板上集成了一个共阳极的彩灯，彩灯的阴极通过拨码开关后连接到w800 的IO 口(PB2、 PB7 和PB11)，控制IO 口为低电平时，对应的灯会亮起来。要使用彩灯，必须打开拨码开关的234。

1.6. DS18B20 温度传感器

W800 的底板上集成有一个 DS18B20 温度传感器，通过拨码开关后连接到W800 的PB3 口，W800通过PB3 可以读出当前的温度。要使用DS18B20 传感器，须打开拨码开关的 1。

1.7. 显示屏接口

开发板支持两种显示屏：1.44 寸TFT 彩屏、0.96 寸OLED 屏。 这 2 种显示屏都是4 线 SPI 接口。注：底板上显示屏接口边上都有丝印，插显示屏的时候，一定要注意显示屏方向要正确，电源标志要 对应上，否则会烧坏显示屏。

1.8. 光敏传感器

W800通过光敏传感器可以采集环境光线的强弱，W800的PA1通过拨码开关后连接到光敏传感器 上，W800 使用ADC 功能可以读取当前光线的强弱。要使用光敏传感器功能，必须打开拨码开关的7。

1.9. w800 鸿蒙系统介绍

HarmonyOS 是一款“面向未来”、面向全场景（移动办公、运动健康、社交通信、媒体娱乐等）的分布式操作系统。在传统的单设备系统能力的基础上，HarmonyOS 提出了基于同一套系统能力、适配多种终端形态的分布式理念，能够支持多种终端设备。

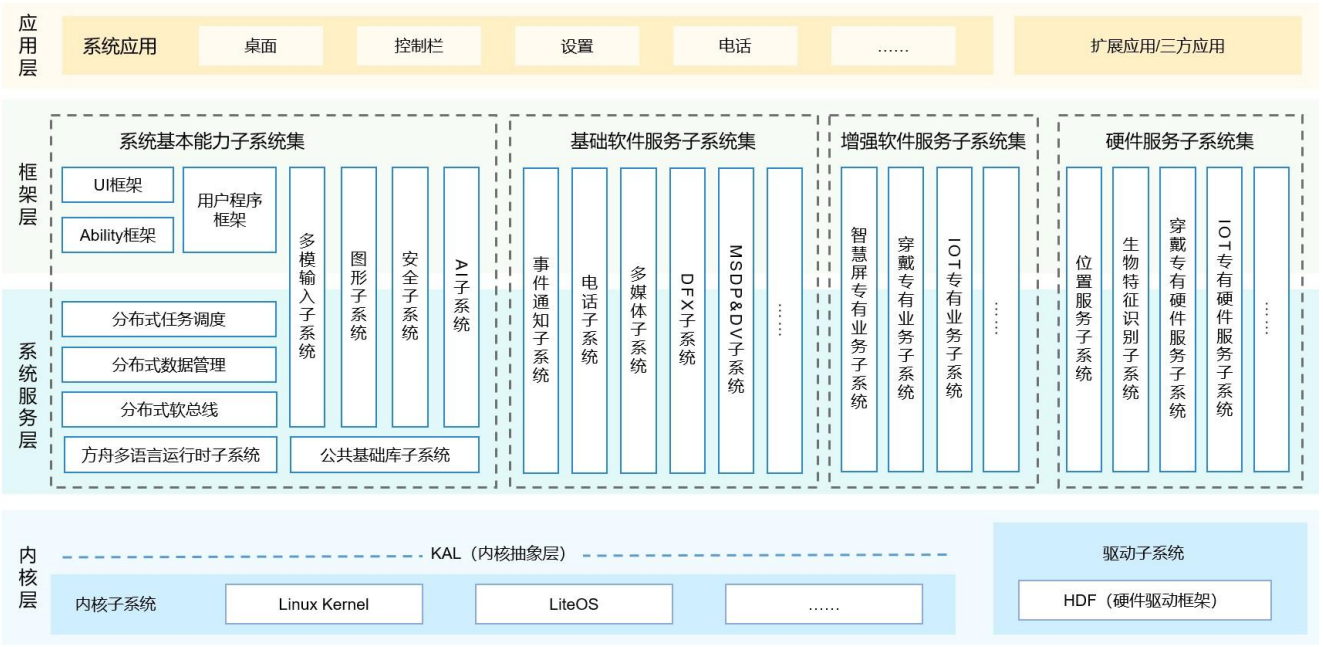
对消费者而言，HarmonyOS 能够将生活场景中的各类终端进行能力整合，可以实现不同的终端设备之间的快速连接、能力互助、资源共享，匹配合适的设备、提供流畅的全场景体验。

对应用开发者而言，HarmonyOS 采用了多种分布式技术，使得应用程序的开发实现与不同终端设备的形态差异无关。这能够让开发者聚焦上层业务逻辑，更加便捷、高效地开发应用。

对设备开发者而言，HarmonyOS 采用了组件化的设计方案，可以根据设备的资源能力和业务特征进行灵活裁剪，满足不同形态的终端设备对于操作系统的要求。

HarmonyOS 提供了支持多种开发语言的 API，供开发者进行应用开发。支持的开发语言包括 Java、XML（Extensible Markup Language）、C/C++ 、 JS（JavaScript）、CSS（Cascading Style Sheets）和 HML（HarmonyOS Markup Language）。

HarmonyOS 整体遵从分层设计，从下向上依次为：内核层、系统服务层、框架层和应用层。系统功能按照“系统 > 子系统 > 功能/模块”逐级展开，在多设备部署场景下，支持根据实际需求裁剪某些非必要的子系统或功能/模块。HarmonyOS 技术架构如下所示。



HarmonyOS 官网：https://device.harmonyos.com/cn/home 技术特性：硬件互助，资源共享一次开发，多端部署统一 OS，弹性部署

HarmonyOS 提供了用户程序框架、Ability 框架以及 UI 框架，支持应用开发过程中多终端的业务逻辑和界面逻辑进行复用，能够实现应用的一次开发、多端部署，提升了跨设备应用的开发效率。

HarmonyOS 通过组件化和小型化等设计方法，支持多种终端设备按需弹性部署，能够适配不同类别的硬件资源和功能需求。支撑通过编译链关系去自动生成组件化的依赖关系，形成组件树依赖图，支撑产品系统的便捷开发，降低硬件设备的开发门槛。

在搭载 HarmonyOS 的分布式终端上，可以保证“正确的人，通过正确的设备，正确地使用数据”。

通过“分布式多端协同身份认证”来保证“正确的人”。 通过“在分布式终端上构筑可信运行环境”来保证“正确的设备”。 通过“分布式数据在跨终端流动的过程中，对数据进行分类分级管理”来保证“正确

地使用数据”。 在官网获取源码后，源码

目录说明：

目录名	描述
applications	应用程序样例
base	基础软件服务子系统集&硬件服务子系统集
build	组件化编译、构建和配置脚本
docs	说明文档
domains	增强软件服务子系统集
drivers	驱动子系统
foundation	系统基础能力子系统集
kernel	内核子系统
prebuilts	编译器及工具链子系统
test	测试子系统
third_party	开源第三方组件
utils	常用的工具集
vendor	厂商提供的软件
build.py	编译脚本文件

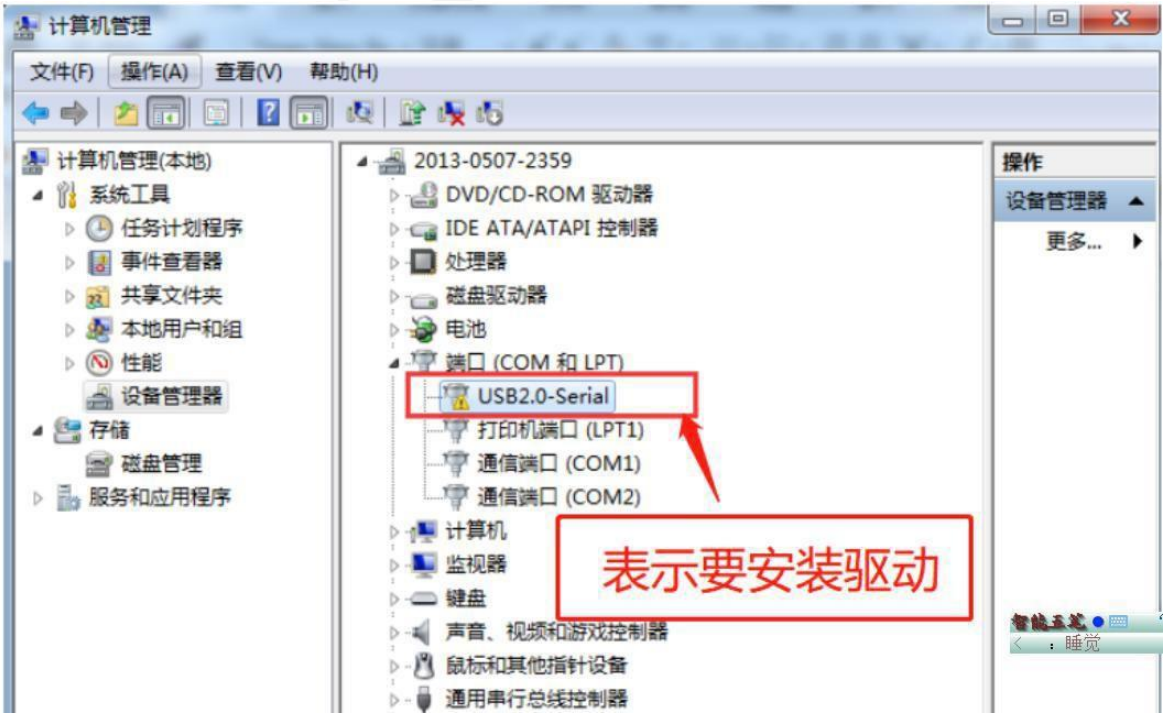
二. 开发环境搭建

这一章的内容是驱动的安装和软件开发环境的搭建。

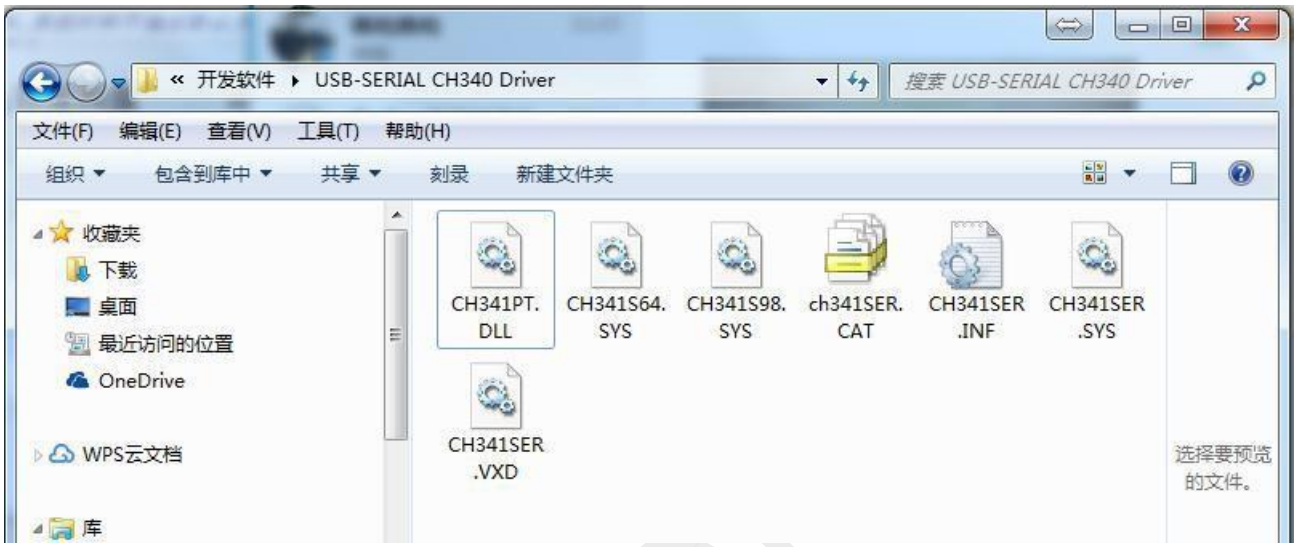
2.1. 串口驱动的安装

我们的开发板使用了 CH340G 的 USB 转串口芯片。通过 USB 线把开发板接到电脑上：接着查看电脑的设备管理器，如下图表示驱动已经正确安装，可以跳过这一节：

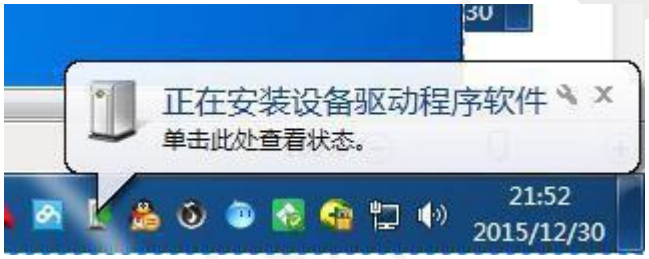
如下图，表示要安装驱动：



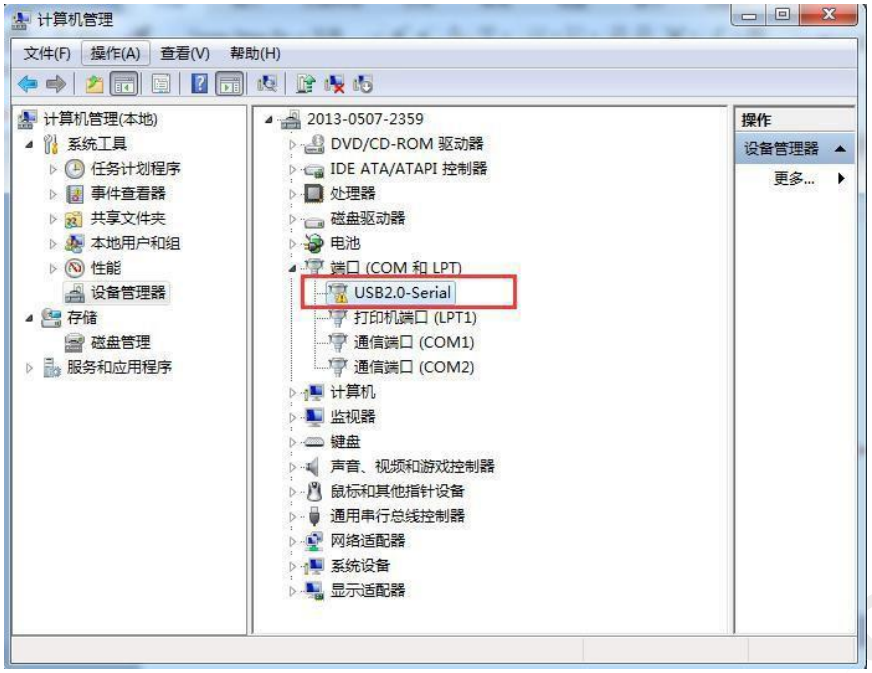
安装驱动步骤一:CH340 串口驱动位置：.\开发软件\USB-SERIAL CH340 Driver.rar，解压后如
下：



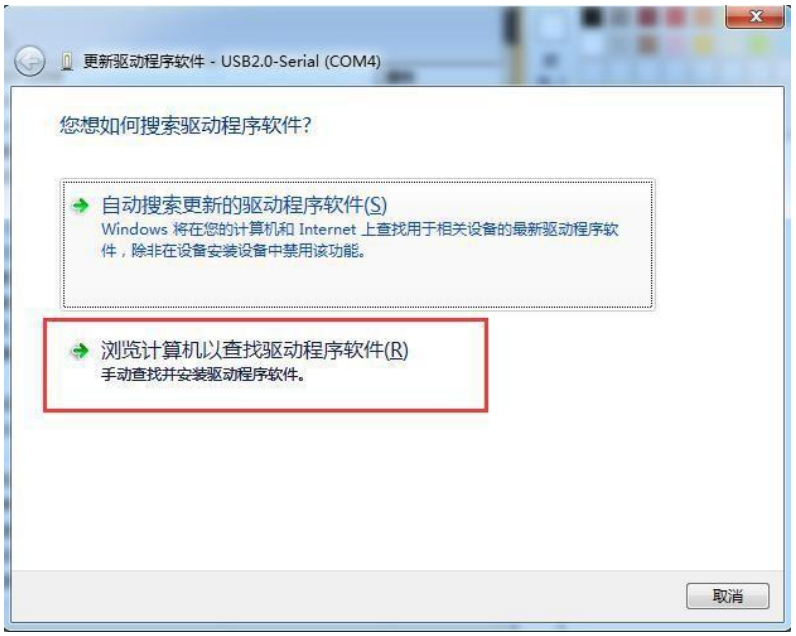
安装驱动步骤二:把开发板通过USB线接到电脑上（要打开开发板电源），提示安装驱动如下：



安装驱动步骤三:打开电脑的设备管理器，查看串口的驱动是否已自动安装，如下图是未安装的。



安装驱动步骤四:右键更新驱动，如下图：



安装驱动步骤五:选择第一步解压好的目录：

安装驱动步骤六:选择确定后，有可能会出现以下的提示，选择“始终安装此驱动程序软件”。

安装驱动步骤七：安装完成后，设备管理器，如下图：

串口驱动程序安装好之后，关于串口程序下载和调试，后面会陆续讲解。

2.2 w800 Linux 开发环境搭建

推荐使用 Ubuntu 18.04。从官网 https://occ.t-head.cn 平头哥芯片开

放社区->技术部落->资源下载->工具，根据自己本地系统环境选择下载适用的“800 Series Toolchain”。

工具链名称	适用系统
-i386-minilibc-.tar	32 位 linux 系统
-x86_64-minilibc-.tar	64 位 linux 系统
-mingw-minilibc-.tar	windows 系统

下载后将编译工具链解压到的某个路径下（如 opt 目录），设置编译工具链路径至环境变量，举例如下：

export PATH=$PATH:/opt/ csky-elfabiv2-tools/bin 上述设置完成，编译工具链即可用，可以进行接下来的编译工作。 用户也可将工具链的路径配置写至.profile 等配置文件中达到自动配置的目的,用户还可以在 sdk 中直接指定工具链路径。

2.3 w800 鸿蒙系统环境搭建

安装 gn、ninja wget

https://repo.huaweicloud.com/harmonyos/compiler/gn/1523/linux/gn.1523.tar

wget https://repo.huaweicloud.com/harmonyos/compiler/ninja/1.9.0/linux/ninja.1.9.0.tar

tar -xvf gn.1523.tar -C ~/

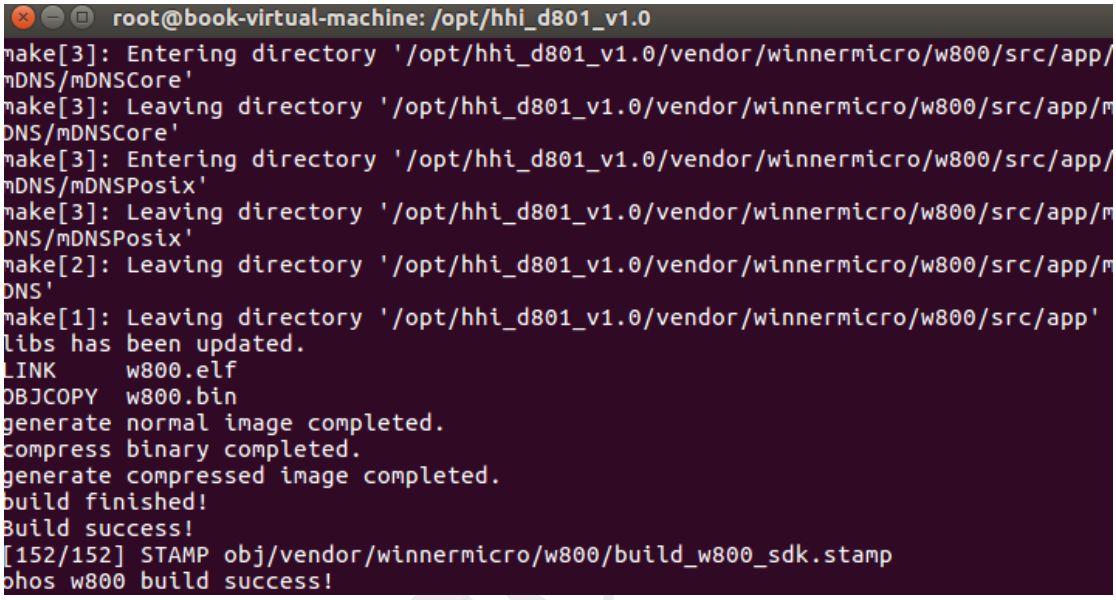
tar -xvf ninja.1.9.0.tar -C ~/ sudo vim ~/.bashrc

export PATH=~/gn:$PATH

export PATH=~/ninja:$PATH

source ~/.bashrc

进入代码 hhi_d801_v1.0 根目录，编译命令： python build.py w800



完成编译链接后，w800 固件会生成，固件位于 out/w800/ 目录下，生成文件有：

w800.fls：串口烧录

w800_ota.img：OTA 升级

w800.map：map 文件

接下来介绍如何把固件烧录到 w800 开发板，以 Windows 环境的烧录为 例：打开软件目录 ThingsTurn_Serial_Tool_V1.8.0.0，双击 ThingsTurn_Serial_Tool.exe。

打开串口，波特率 115200，选择固件文件 w800.fls 后点击下载，然后按一下开发板的复位键开始下载。下载完成后复位板子运行程序。

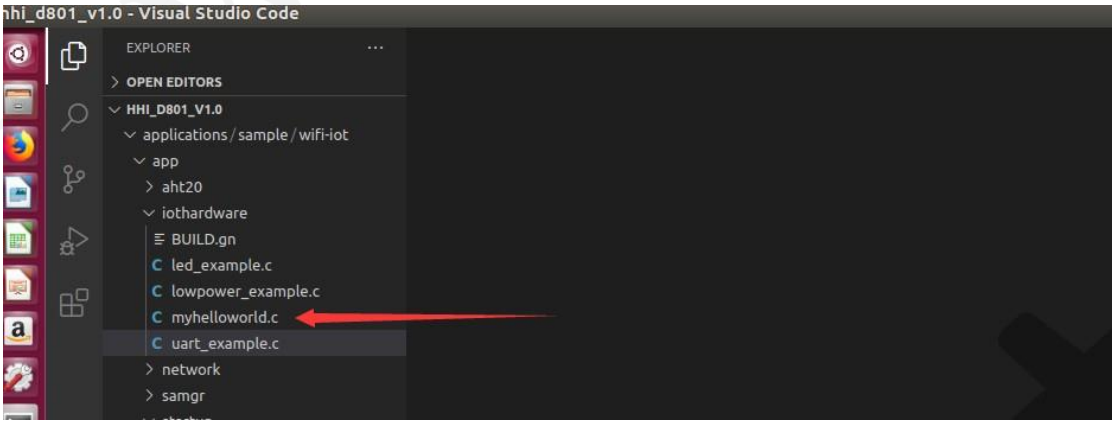
鸿蒙系统启动：

三.、外设基础实验

说明：在这一章实验里，我们以打包好例程开始学习讲解。

3.1 鸿蒙系统第一个驱动程序helloworld

编写第一个程序 helloworld，首先我们在 app 目录新建一个 myhelloworld.c 文件。

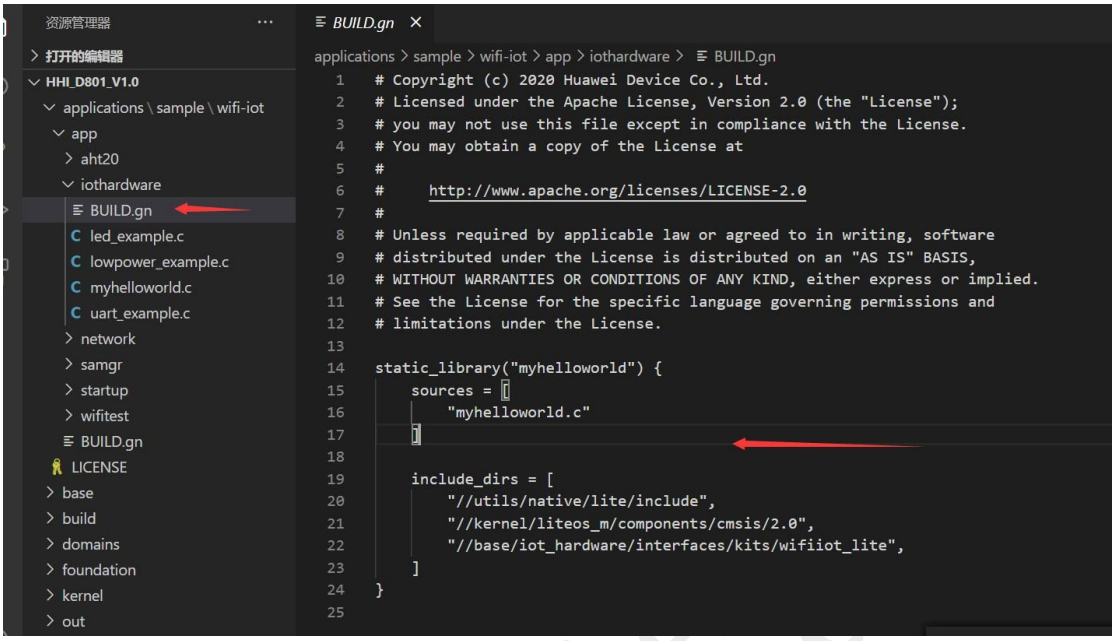


接下来编写 helloworld 任务程序，调用 SYS_RUN 来执行：

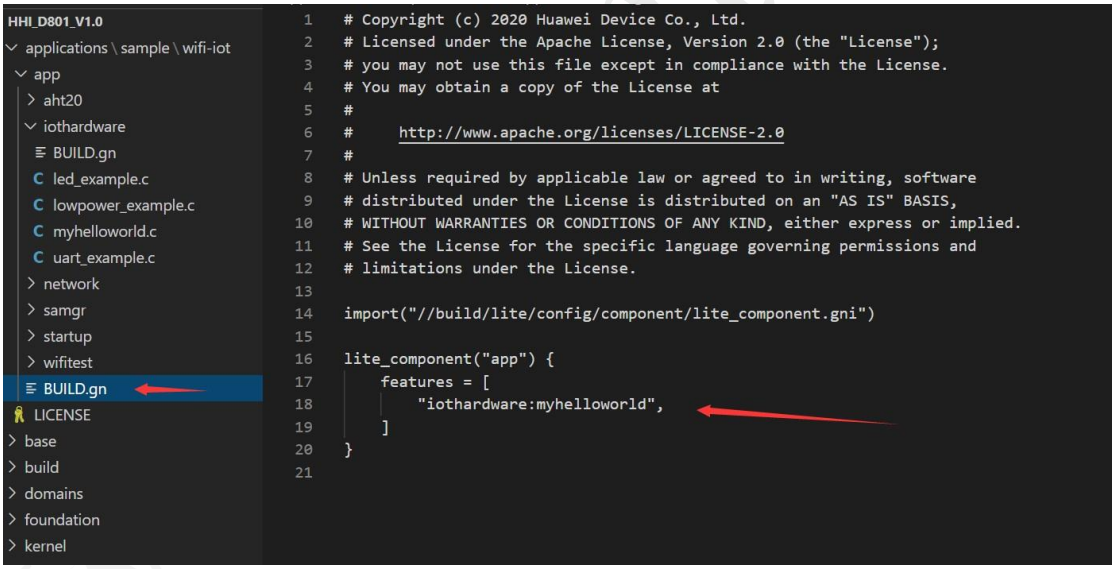
```
#define TASK_STACK_SIZE 512
#define TASK_PRIO 25
```

```
static void *helloworldTask(const char *arg)
{
    (void)arg;
    while (1) {
        printf("hello world!\n");
        osDelay(100);
    }

    return NULL;
}

static void myhelloworld(void)
{
    osThreadAttr_t attr;


    attr.name = "myhelloworldTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = TASK_STACK_SIZE;
    attr.priority = TASK_PRIO;

    if (osThreadNew((osThreadFunc_t)helloworldTask, NULL, &attr) == NUL
L) {
        printf("Falied to create Task!\n");
    }
}
SYS_RUN(myhelloworld); //if test add it
```

在同目录新建一个 BUILD.gn，加入以下内容：



在 app 同目录下的 BUILD.gn 加入我们需要编译运行的程序 helloworld。



编译运行结果:

3.2 w800 鸿蒙系统 GPIO 控制 LED 灯实验

要使用彩灯，必须打开拨码开关的234。本节内容我们实现 RGB 每隔 1 秒进行闪烁。W800 的底板上集成了一个共阳极的彩灯，彩灯的阴极通过拨码 开关后连接到w800 的 IO 口(PB2、 PB7 和 PB11)，控制 IO 口为低电平时，对应的灯会亮起来。这里我们以 PB2 红灯为例子实现控制。

首先创建一个 LED 任务。

```
static void GpioExampleEntry(void)
{
    osThreadAttr_t attr;
    GpioInit();
    GpioSetDir(WIFI_IOT_GPIO_PB_02, WIFI_IOT_GPIO_DIR_OUTPUT); //output
 is 0 PB00 control led
    attr.name = "GpioTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
```

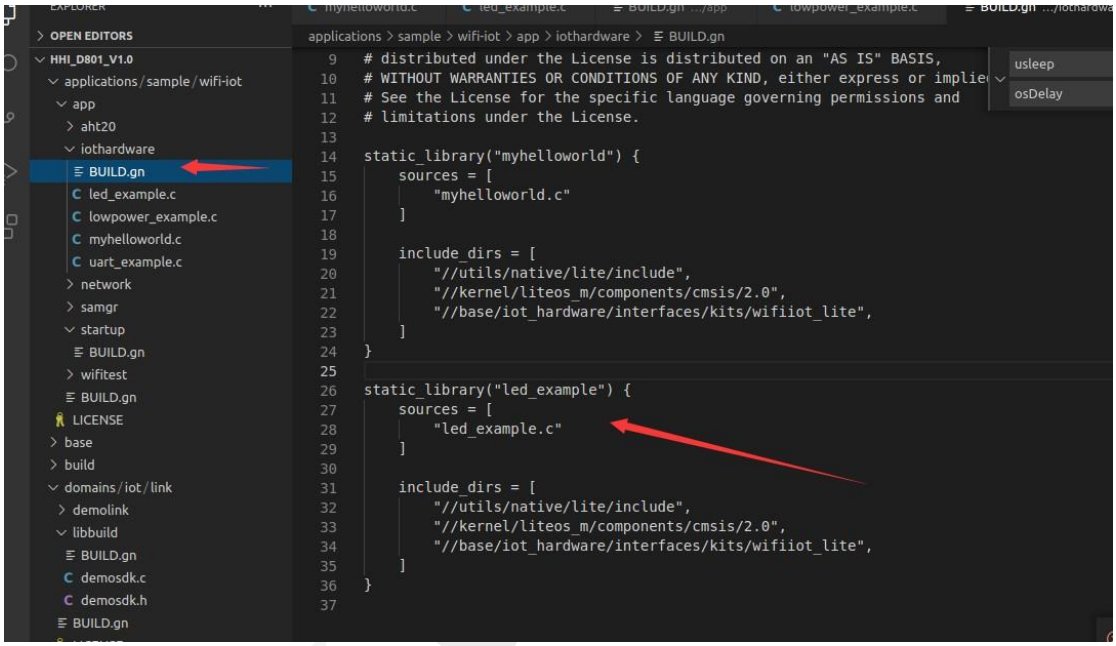
```
    attr.stack_mem  =  NULL;
    attr.stack_size = LED_TASK_STACK_SIZE;
    attr.priority = LED_TASK_PRIO;
     if (osThreadNew((osThreadFunc_t)GpioTask, NULL, &attr) == NULL) {

        printf("[GpioExample] Falied to create GpioTask!\n");
    }
}
```

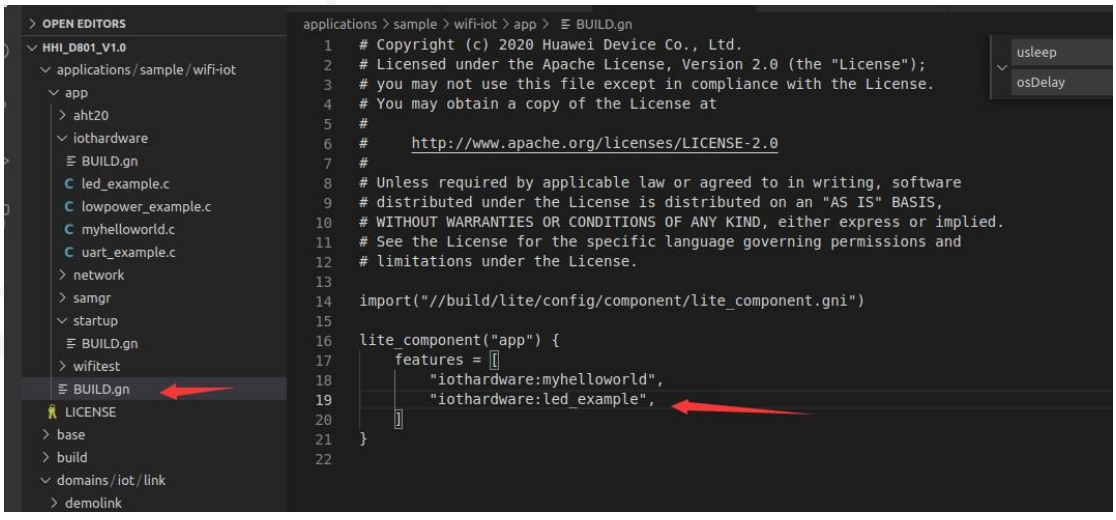
接下来编写任务程序，输出高电平熄灭，输出低电平点亮。

```
enum LedState g_ledState = LED_SPARK;
static void *GpioTask(const char *arg)

{
    (void)arg;
    while (1) {
         switch (g_ledState)
           { case LED_ON:
                printf(" LED_ON! \n");
                GpioSetOutputVal(WIFI_IOT_GPIO_PB_02, WIFI_IOT_GPIO_ATT
R_PULLLOW);
                osDelay(100);
                break;
            case LED_OFF:
                printf(" LED_OFF! \n");
                GpioSetOutputVal(WIFI_IOT_GPIO_PB_02, WIFI_IOT_GPIO_ATT
R_PULLHIGH);
                osDelay(100);
                break;
            case LED_SPARK:
            printf(" LED_SPARK! \n");
                GpioSetOutputVal(WIFI_IOT_GPIO_PB_02, WIFI_IOT_GPIO_ATT
R_PULLLOW);
                osDelay(100);
                GpioSetOutputVal(WIFI_IOT_GPIO_PB_02, WIFI_IOT_GPIO_ATT
R_PULLHIGH);
                osDelay(100);
                break;
            default:
                osDelay(100);
                break;
```

```
        }

    }
    return NULL;

}
```

编译选项添加如下内容：





程序下载到开发板，可以看到红灯周期性的闪烁。

3.3 w800 鸿蒙系统低功耗休眠模式

在某些时候我们设计的产品可能不具备持久供电的环境，那通常会采用锂电池、干电池一类的轻便型的非持久性电源。当遇到这种情况时，产品的续航能力可能就会成用户评估产品的一个重要指标，加大电池容量当然是最为直接的方案，但是这也意味着提高产品的生产成本。那增加产品续航能力的另一个方案就是原自产品自身——降低不必要的能源消耗。

首先建立低功耗的任务：

```
static void LowpowerExampleEntry(void)
{
    osThreadAttr_t attr;

    LpcInit();

    attr.name = "LowpowerTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
```

```
    attr.stack_mem = NULL;
    attr.stack_size = LP_TASK_STACK_SIZE;
    attr.priority = LP_TASK_PRIO;

    if (osThreadNew((osThreadFunc_t)LowpowerTask, NULL, &attr) == NULL)
 {
        printf("[LowpowerExample] Falied to create LowpowerTask!\n");
    }
}
static void *LowpowerTask(const char *arg)
{
    (void)arg;
    static int i = 0;

    printf("enter LowpowerTask\n");

    while (1) {
        if (i == 0)
            { i++;
            LpcSetType(LIGHT_SLEEP);
        }
        else
            LpcSetType(DEEP_SLEEP);
        osDelay(500);
    }
}
```

配置编译选后编译下载程序，程序进入低功耗后停止串口打印，停止：

3.4w800 鸿蒙系统串口收发控制实验

本例我们实现 w800 鸿蒙系统串口收发控制实验。 建立一个串口收发的任务：

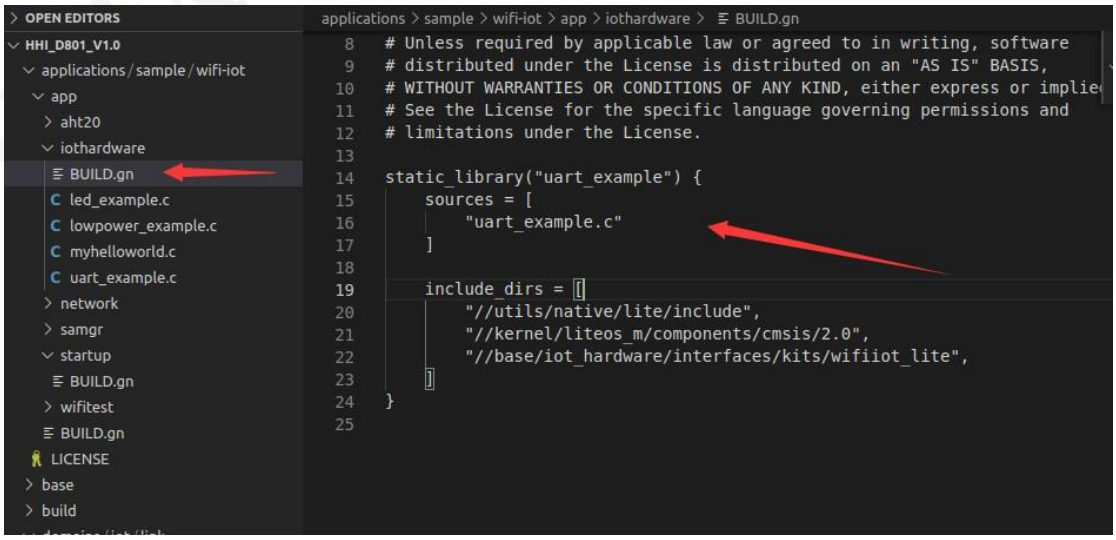
```
static void UartExampleEntry(void)
{
    osThreadAttr_t attr;

    attr.name = "UartTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = UART_TASK_STACK_SIZE;
    attr.priority = UART_TASK_PRIO;
    printf("[UartExample] create UartTask!\n");
    if (osThreadNew((osThreadFunc_t)UartTask, NULL, &attr) == NULL)
        { printf("[UartExample] Falied to create UartTask!\n");
    }
}
```

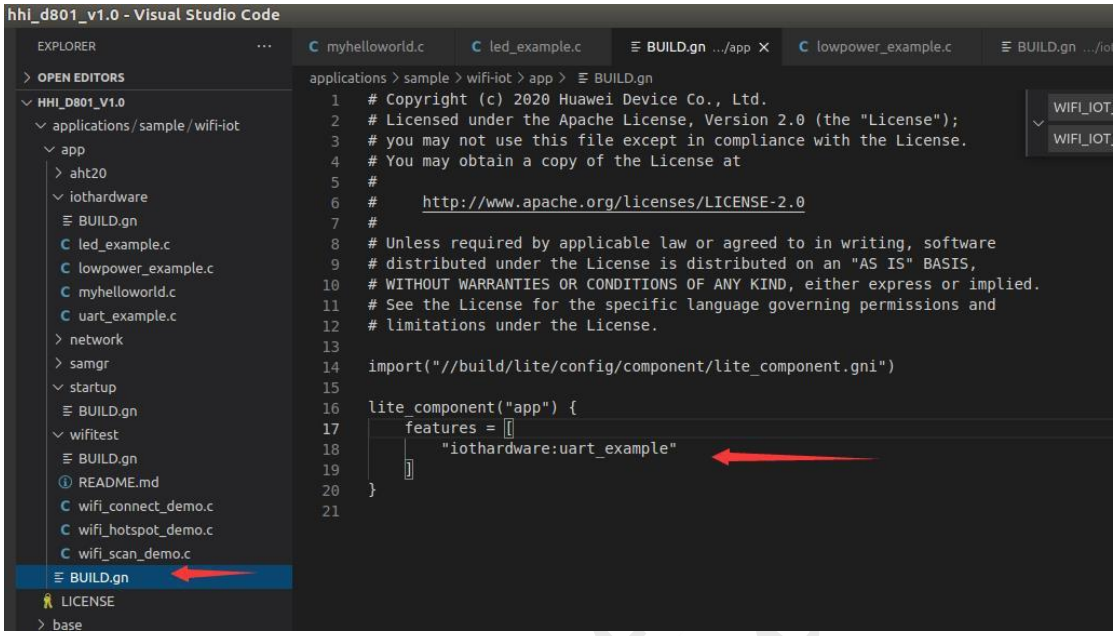
我们配置为串口 0，串口 0 接收到数据并原样返回数据：

```
static void *UartTask(const char *arg)
{
    (void)arg;
    unsigned int ret;
    int len;
    unsigned char uartWriteBuff[] = "hello uart!";
    WifiIotUartAttribute uartAttr;
    uartAttr.baudRate = INIT_BAUD_RATE,
    uartAttr.dataBits = WIFI_IOT_UART_DATA_BIT_8;
    uartAttr.stopBits = WIFI_IOT_UART_STOP_BIT_1;
    uartAttr.parity = WIFI_IOT_UART_PARITY_NONE;
    uartAttr.pad ='M';

    WifiIotUartExtraAttr extraAttr;
    (void)memset_s(&extraAttr, sizeof(WifiIotUartExtraAttr), 0,
sizeof( WifiIotUartExtraAttr));
    extraAttr.txFifoLine = WIFI_IOT_FIFO_LINE_ONE_EIGHT;
```

```
    extraAttr.rxFifoLine = WIFI_IOT_FIFO_LINE_SEVEN_EIGHTS;
    extraAttr.flowFifoLine = WIFI_IOT_FIFO_LINE_ONE_EIGHT;
    extraAttr.txBlock = WIFI_IOT_UART_BLOCK_STATE_BLOCK;
    extraAttr.rxBlock = WIFI_IOT_UART_BLOCK_STATE_BLOCK;
    extraAttr.txBufSize = TEST_BUF_SIZE;
    extraAttr.rxBufSize  =  TEST_BUF_SIZE;
    extraAttr.txUseDma = WIFI_IOT_UART_USE_DMA;
    extraAttr.rxUseDma = WIFI_IOT_UART_USE_DMA;

    ret = UartDeinit(WIFI_IOT_UART_IDX_0);

    ret = UartInit(WIFI_IOT_UART_IDX_0, &uartAttr, &extraAttr);

    len = UartWrite(WIFI_IOT_UART_IDX_0, uartWriteBuff, sizeof(uartWrit
eBuff));
    len = UartWrite(WIFI_IOT_UART_IDX_0, uartWriteBuff, sizeof(uartWrit
eBuff));
    while (1) {
        len = UartRead(WIFI_IOT_UART_IDX_0, uartReadBuff, sizeof(uartRe
adBuff));
        if(len > 0)
            UartWrite(WIFI_IOT_UART_IDX_0, uartReadBuff, len);
    }

    return NULL;
}
```

配 BUILD.gn：





编译下载到开发板，串口随便输数据并发送，数据会原样返回：

3.5w800 鸿蒙系统定时器实验

这节我们使用鸿蒙系统定时 1 秒时间，首先建立一个定时器任务：

```
static void os_timer(void)
{
    osThreadAttr_t attr;
    attr.name = "os_timerTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = TASK_STACK_SIZE;
    attr.priority = TASK_PRIO;
    if (osThreadNew((osThreadFunc_t)os_timerTask, NULL, &attr) == NULL)
 {

        printf("Falied to create Task!\n");
    }
}
```

在任务中启动定时器，注册定时回调函数，并设置时间为 1 秒：

```
static void *os_timerTask(const char *arg)
{

   (void)arg;
    osTimerId_t id;
    unsigned int value=0xffff;
    id = osTimerNew((osTimerFunc_t)TimerFun, osTimerPeriodic, &value, N
ULL);
    osTimerStart(id, 100);
    while (1) {
        osDelay(1000);
    }
    return NULL;
}
```

定时回调函数：

```
tatic void TimerFun(void const *argument)
{
    (void)argument;
    printf("TimerFunTask!\n");
    return;
```

```
}
```

编译并运行程序，每隔 1 秒进行打印：

3.6 w800 鸿蒙系统 PWM LED以一定频率闪烁

W800 芯片支持 5 路 PWM 接口，有 10 个引脚可被配置为 PWM 功能，具体如下：

PWM	W800 引脚组1	W800 引脚组2
PWM1	WM_IO_PB_00	WM_IO_PB_19
PWM2	WM_IO_PB_01	WM_IO_PB_20
PWM3	WM_IO_PB_02	WM_IO_PA_00
PWM4	WM_IO_PB_03	WM_IO_PA_01
PWM5	WM_IO_PA_07	WM_IO_PA_04

代码里默认把 W800 芯片的 10 个可配置为 PWM 功能的引脚分成两组：W800 引脚组 1 和 W800 引脚组 2。

1. 鸿蒙系统 PWM API 头文件 base/iot_hardware/interfaces/kits/wifiiot_lite/wifiiot_pwm.h 2. 鸿蒙系统 PWM API 使用说明

1) unsigned int PwmInit(WifiIotPwmPort port);初始化一个 PWM 设备。

说明：参数可设置为 WIFI_IOT_PWM_PORT_PWM0 ~ WIFI_IOT_PWM_PORT_PWM4 其中之一，对应 W800 引脚组 1 的相关引脚。如果想使用 W800 引脚组 2 的相关引脚，可以在代码里扩展枚举变量 WifiIotPwmPort。

unsigned int PwmDeinit(WifiIotPwmPort port); 释放一个 PWM 设

备。

unsigned int PwmStart(WifiIotPwmPort port, unsigned short duty, unsigned short freq); 按照入参配置启动对应的 PWM 功能。

建立一个 PWM 任务：

```
static void pwm(void)
{
    osThreadAttr_t attr;
    attr.name = "pwmTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = TASK_STACK_SIZE;
    attr.priority = TASK_PRIO;
    if (osThreadNew((osThreadFunc_t)pwmTask, NULL, &attr) == NULL)
        { printf("Falied to create Task!\n");
    }

}
```

任务中初始化 PWM，设置频率 20hz 并启动：

```
#define TEST_PWM_PORT 1200
#define TEST_PWM_DUTY 20

static void *pwmTask(const char *arg)
{
    (void)arg;
```

```
    PwmDeinit(WIFI_IOT_PWM_PORT_PWM2);
    PwmInit(WIFI_IOT_PWM_PORT_PWM2);
    PwmStart(WIFI_IOT_PWM_PORT_PWM2, TEST_PWM_PORT, TEST_PWM_DUTY);
    while (1) {
        printf("pwmTask!\n");
        osDelay(1000);
    }
    return NULL;
}
```

编译并下载到开发板，开发板红灯以 20hz 频率闪烁。

四、wifi 通信实验

4.1 w800 鸿蒙系统 wifi AP 模式

AP 是（Wireless）AccessPoint 的缩写，即（无线）访问接入点。简单来讲就像是无线路由器一样， 设备打开后进入 AP 模式，在手机的网络列表里面，可以搜索到类似 AP 的名字（SSID）。我们的手机和笔记本电脑均可连接到创建的 AP热点。

```
* WiFi 的工作模式
  * AP 模式：热点模式，提供无线接入服务，允许其它无线设备接入，提供数据访问，一般的无线路由/网桥工作在该模式。
  * STA 模式：类似于无线终端，本身并不接受其他设备的接入，它可以连接到 AP，一般无线网卡即工作在该模式。
* Harmony OS 的 WiFi 相关 API 头文件位于 `foundation\communication\interfaces\kits\wifi_lite\wifiservice` 目录，该目录下有 9 个文件；
  * `wifi_device.h` 中定义的是 STA 模式的主要接口，例如扫描其他热点、添加热点配置（热点名称、密码等）、连接其他热点；
  * `wifi_hotspot.h` 中定义的是 AP 模式的主要接口，例如设置热点信息（热点名称、密码等）、查询连接的设备列表；
  * `wifi_hotsport_config.h` 中定义了设置和获取当前工作在 2.4G 或者 5G 频段的接口`SetBand`和`GetBand`；
  * 另外 6 个文件中定义了上述接口相关的类型，例如扫描结果、热点配置、热点连接状态等；
```

创建 AP 的任务：

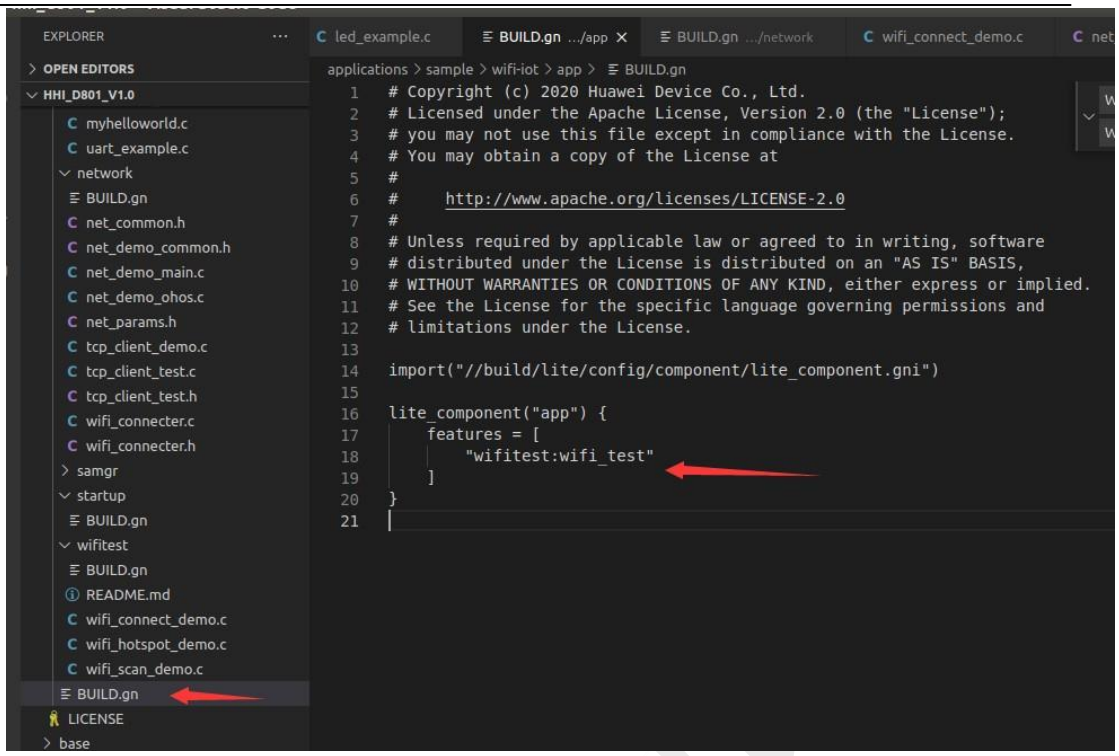
```
static void WifiHotspotDemo(void)
{
    osThreadAttr_t attr;

    attr.name = "WifiHotspotTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 10240;
    attr.priority = osPriorityNormal;

    if (osThreadNew(WifiHotspotTask, NULL, &attr) == NULL)
        { LOGI("[WifiHotspotDemo] Falied to create
        WifiHotspotTask!\n");
    }
```

任务程序中注册联网事件，设置 AP 热点名称，密码：

```
static void WifiHotspotTask(void *arg)
{
    (void)arg;
    WifiErrorCode errCode;
    HotspotConfig config = {0};
```

```
    // 准备 AP 的配置参数
    strcpy(config.ssid, "HiSpark-AP");
    strcpy(config.preSharedKey, "12345678");
    config.securityType = WIFI_SEC_TYPE_PSK;
    config.band = HOTSPOT_BAND_TYPE_2G;
    config.channelNum = 7;

    osDelay(10);

    while (1) {
        LOGI("starting AP ...");
        LOGI("IsHotspotActive(): %d", IsHotspotActive());
        errCode = StartHotspot(&config);
        LOGI("StartHotspot: %d", errCode);
        LOGI("IsHotspotActive(): %d", IsHotspotActive());

        HotspotConfig result = {0};
        errCode = GetHotspotConfig(&result);
        LOGI("GetHotspotConfig: %d, %d", errCode, memcmp(&config, &resu
lt, sizeof(result)));

        const int step = 5;
        for (int timeout = 60; timeout >= 0; timeout -= step)
            { LOGI("after %d seconds, I will dsiable AP.",
            timeout); osDelay(step * 100);
        }

        LOGI("stop AP ...");
        StopHotspot();
        LOGI("stop AP done!");

        osDelay(100);
    }
}
```

配 BUILD.gn:



程序编译下载到开发板运行，可以搜索到板子产生的热点：

4.2 w800 鸿蒙系统 wifi STA模式

这节我们使用 sta 模式连接路由器联网。建立联网任务：

```
static void WifiConnectDemo(void)
{
    osThreadAttr_t attr;

    attr.name = "WifiConnectTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 10240;
    attr.priority = osPriorityNormal;

    if (osThreadNew(WifiConnectTask, NULL, &attr) == NULL)
        { LOGI("[WifiConnectDemo] Falied to create
        WifiConnectTask!");
    }
```

任务中设置我们需要连接的路由器名称，密码：

```
static void WifiConnectTask(void *arg)
{
    (void)arg;
    WifiErrorCode errCode;
    WifiEvent eventListener = {
        .OnWifiConnectionChanged = OnWifiConnectionChanged,
        .OnWifiScanStateChanged = OnWifiScanStateChanged
    };
    WifiDeviceConfig apConfig = {};
    int netId = -1;

    osDelay(100);
    errCode = RegisterWifiEvent(&eventListener);
    LOGI("RegisterWifiEvent: %d", errCode);

    // setup your AP params
    strcpy(apConfig.ssid, "HIHOPE-AP");
    strcpy(apConfig.preSharedKey, "1234567890");
    apConfig.securityType = WIFI_SEC_TYPE_PSK;
```

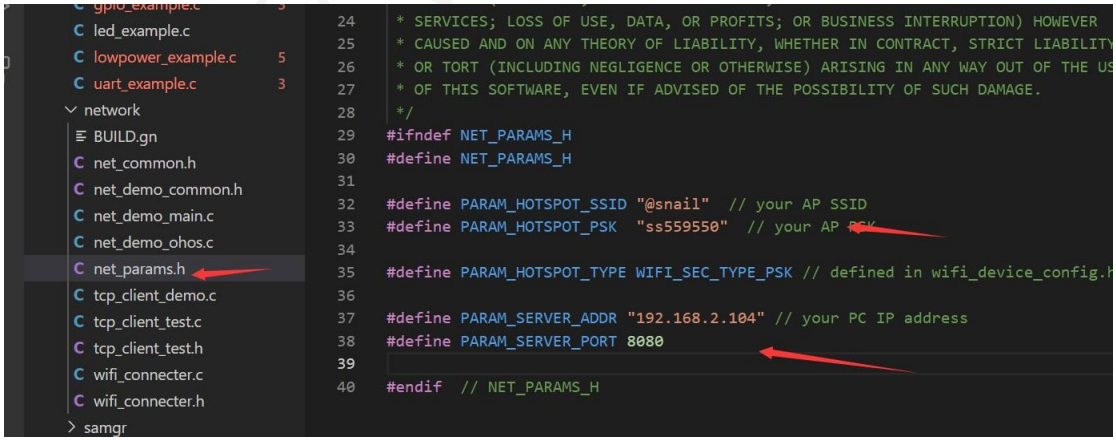
```
    while (1) {
        LOGI("sysTicks: %u, kernelTicks: %u", osKernelGetSysTimerCount(
), osKernelGetTickCount());
        LOGI("IsWifiActive(): %d", IsWifiActive());

        errCode = EnableWifi();
        LOGI("EnableWifi: %d", errCode);
        LOGI("IsWifiActive(): %d", IsWifiActive());
        osDelay(10);

        char buffer[32] = {0};
        unsigned char mac[6] = {0};
        errCode = GetDeviceMacAddress(mac);
        LOGI("GetDeviceMacAddress: %d, %s", errCode, FormatMacAddress(b
uffer, mac));

        g_scanDone = 0;
        errCode = Scan();
        LOGI("Scan: %d", errCode);

        LOGI("waiting for scan done...");
        while (!g_scanDone) {
            osDelay(5);
        }
        PrintScanResult();
        osDelay(100);

        errCode = AddDeviceConfig(&apConfig, &netId);
        LOGI("AddDeviceConfig: %d", errCode);

        g_connected = 0;
        errCode = ConnectTo(netId);
        LOGI("ConnectTo(%d): %d", netId, errCode);

        LOGI("waiting for connect success...");
        while (!g_connected) {
            osDelay(10);
        }
        LOGI("g_connected: %d", g_connected);
        osDelay(100);

        // 联网业务开始
        // 这里是网络业务代码...
```

```
        int i = 0;
        for (struct netif* p = netif_list; p; p = p->next, i++)
            { hexdump(p->name, sizeof(p->name));
            LOGI("[%d]: ip = %s", i, inet_ntoa(p->ip_addr));
            LOGI("[%d]: gw = %s", i, inet_ntoa(p->gw));
            LOGI("[%d]: netmask = %s", i, inet_ntoa(p->netmask));
        }

        // 模拟联网业务
        const int step = 5;
        for (int timeout = 60; timeout >= 0; timeout -= step) {
            LOGI("after %d seconds, I will disconnect with AP.", timeou
t);
            osDelay(step * 100);
        }
        // 联网业务结束

        errCode = Disconnect(); // disconnect with your AP
        LOGI("Disconnect: %d", errCode);

        errCode = RemoveDevice(netId); // remove AP config
        LOGI("RemoveDevice: %d", errCode);

        errCode = DisableWifi();
        LOGI("DisableWifi: %d", errCode);
        osDelay(200);
    }
}
```

编译下载运行程序：

4.3 w800 鸿蒙系统 TCP客户端通信

本节我们使用开发板建立一个 tcp 客户端，再去连接指定地址和端口的 tcp 服务端并进行数据通信。首先配置我们需要连接的服务器 IP，测试时我们写入电脑的 IP 地址，配置我们的路由器信息

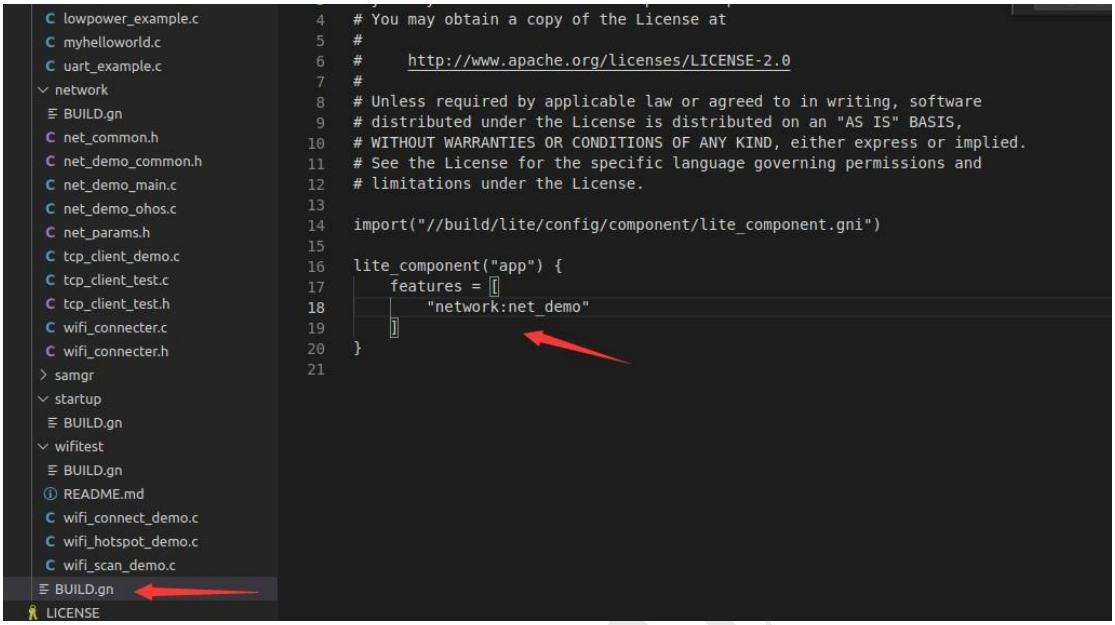


创建一个 TCP 通信的 socket:

```
static char request[] = "Hello";
static char response[128] = "";
```

```
void TcpClientTest(const char* host, unsigned short port)
{
    ssize_t retval = 0;
    int sockfd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket

    struct sockaddr_in serverAddr = {0};
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    if (inet_pton(AF_INET, host, &serverAddr.sin_addr) <= 0)
        { printf("inet_pton failed!\r\n");
        goto do_cleanup;
    }

    if (connect(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAd
dr)) < 0) {
        printf("connect failed!\r\n");
        goto do_cleanup;
    }
    printf("connect to server %s success!\r\n", host);

    retval = send(sockfd, request, sizeof(request), 0);
    if (retval < 0) {
        printf("send request failed!\r\n");
        goto do_cleanup;
    }
    printf("send request{%s} %ld to server done!\r\n", request, retval)
;

    retval = recv(sockfd, &response, sizeof(response), 0);
    if (retval <= 0) {
        printf("send response from server failed or done, %ld!\r\n", re
tval);
        goto do_cleanup;
    }
    response[retval] = '\0';
    printf("recv response{%s} %ld from server done!\r\n", response, ret
val);

do_cleanup:
    printf("do_cleanup...\r\n");
    close(sockfd);
}
```

配 BUILD.gn：



电脑建立一个 TCP 服务器：

编译程序并下载到开发板，通信成功发 hello 到服务器：

4.4 w800 鸿蒙系统 wifi扫描热点

这节我们使用开发板扫描周围的热点并列出来，建立扫描任务

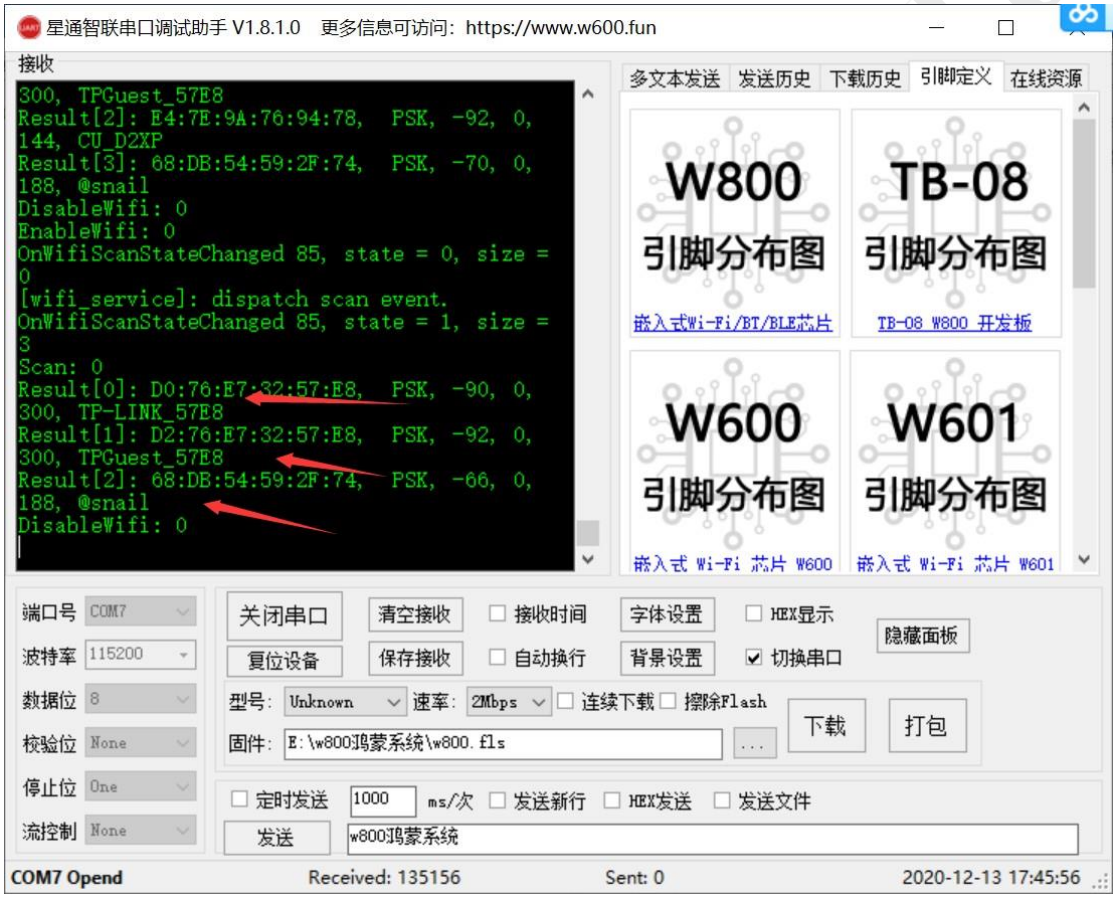
```
static void WifiScanDemo(void)
{
    osThreadAttr_t attr;

    attr.name = "WifiScanTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 10240;
    attr.priority = osPriorityNormal;

    if (osThreadNew(WifiScanTask, NULL, &attr) == NULL)
        { printf("[WifiScanDemo] Falied to create
        WifiScanTask!\n");
    }
}
```

任务程序中执行扫描操作：

```
static void WifiScanTask(void *arg)
{
    (void)arg;
    WifiErrorCode errCode;
    WifiEvent eventListener = {
        .OnWifiConnectionChanged = OnWifiConnectionChanged,
        .OnWifiScanStateChanged = OnWifiScanStateChanged
    };

    osDelay(100);
    errCode = RegisterWifiEvent(&eventListener);
    printf("RegisterWifiEvent: %d\r\n", errCode);

    while (1) {
        errCode = EnableWifi();
        printf("EnableWifi: %d\r\n", errCode);
        osDelay(200);

        g_scanDone = 0;
        errCode = Scan();
        printf("Scan: %d\r\n", errCode);

        // wait for scan done!
        while (!g_scanDone) {
```

```
            osDelay(5);
        }
        PrintScanResult();
        osDelay(500);

        errCode = DisableWifi();
        printf("DisableWifi: %d\r\n", errCode);
        osDelay(300);
    }
}
```

编译下载运行程序：



五、w800 鸿蒙系统连接 dohome 平台云

Doit 智能云（wechat.doit.am）是由深圳四博智联科技有限公司开发的可直接用于生产环境的物联网云平台。Doit 智能云可对单个设备或是一组设备进行远程控制、接收上传数据并实时展示、实现定时任务（精确到秒）等，特有的事件统计功能可以对每台设备的开机时间和时长进行统计和分析。

针对日益增长的微信控制智能设备需求，Doit 智能云实现了：

1、 针对每台设备生成唯一的二维码，该二维码可被微信和手机 APP 同时扫描绑定。若设备数量在 10 万以下，可直接免费使用 Doit 智能云实现微信控制，省去微信 API 复杂开发流程。

2、 针对每一类产品，生成产品标示二维码，通过微信或者手机 APP 实现该类产品的批量推送和控制。

3、 在设备端提供最全面的配置上网方式案例，包括微信的 Airkiss、ESP - Touch（针对 ESP8266）、Easylink（针对 EMW3165）、Soft AP、网页配置等，确保只要有路由器，就能使设备配置上网成功。

4、 控制方式多种多样，手机 app 控制、微信控制、直连 Soft AP 控制、局域网控制等。

5、 支持 TCP、Websocket 等多种接入方式。在协议设计上，采用纯文本协议，支持推送、上传、管道等多种通讯功能，保证数据传输的便利性、实时性和安全性。

基于强大的 Doit 智能云平台，开发一个智能插座或智能小车类功能，包含手机端、微信端、设备端的程序仅需要半个工作日的时间！在开发过程中，Doit 智能云提供设备虚拟功能，可实现并行开发，加速产品的开发进程。基于 Doit 智能云，我们已经开发了智能插座、智能小车等产品，后续会有新的产品模板添加，利用这些产品模板，厂商不需要写任何代码，就能直接拿去生产相应的产品到市场销售。

首先我们建立一个任务，让板子连接上路由器：

```
static void NetDemoEntry(void)
{
    osThreadAttr_t attr;

    attr.name = "NetDemoTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 10240;
    attr.priority = osPriorityNormal;

    if (osThreadNew(NetDemoTask, NULL, &attr) == NULL) {
        printf("[NetDemoEntry] Falied to create NetDemoTask!\n");
    }
}
```

配置路由器信息并连接：

```
#define PARAM_HOTSPOT_SSID "@snail"  // your AP SSID
```

```
#define PARAM_HOTSPOT_PSK  "ss559550"  // your AP PSK

    // 准备 AP 的配置参数
    strcpy(config.ssid, PARAM_HOTSPOT_SSID);
    strcpy(config.preSharedKey, PARAM_HOTSPOT_PSK);
    config.securityType = PARAM_HOTSPOT_TYPE;

    osDelay(10);

    int netId = ConnectToHotspot(&config);

    int timeout = 10;
    while (timeout--) {
        printf("After %d seconds, I will start %s test!\r\n", timeout, GetN
etDemoName());
        osDelay(100);
    }
```

等待 10 秒后开始连接 dohome 的服务器：

```
void TcpClientTest(const char* host, unsigned short port)
{
    ssize_t retval = 0;
    int sockfd = socket(AF_INET, SOCK_STREAM, 0); // TCP socket

    struct sockaddr_in serverAddr = {0};
    serverAddr.sin_family = AF_INET;
    serverAddr.sin_port = htons(port);
    if (inet_pton(AF_INET, host, &serverAddr.sin_addr) <= 0) {
        printf("inet_pton failed!\r\n");
        goto do_cleanup;
    }

    if (connect(sockfd, (struct sockaddr *)&serverAddr, sizeof(serverAddr))
 < 0) {
        printf("connect failed!\r\n");
        goto do_cleanup;
    }
    printf("connect to server %s success!\r\n", host);

    retval = send(sockfd, request, sizeof(request), 0);
    if (retval < 0) {
        printf("send request failed!\r\n");
        goto do_cleanup;
    }
    printf("send request{%s} %ld to server done!\r\n", request, retval);

    retval = recv(sockfd, &response, sizeof(response), 0);
    if (retval <= 0) {
```

```
        printf("send response from server failed or done, %ld!\r\n", retval
);
        goto do_cleanup;
    }
    response[retval] = '\0';
    printf("recv response{%s} %ld from server done!\r\n", response, retval);

do_cleanup:
    printf("do_cleanup...\r\n");
    close(sockfd);
}
```

程序编译下载到开发板运行：