山东科技大学

SHANDONG UNIVERSITY OF SCIENCE AND TECHNOLOGY

计算机工程综合实践报告

题目：智能风扇项目及平时实验实践报告

学年学期：	2022-2023-2
学　　院：	计算机科学与工程学院
专业班级：	计算机科学与技术 2020-1 班
姓　　名：	何超
学　　号：	202001021107
指导教师：	冷钦街，东野长磊，韩进，刁秀丽，赵晓燕
完成日期：	2023 年 7 月 9 日

目录

一、综合性项目报告

组长：何超（本人）、小组成员：孙文斌、杨昊明、李超楠

我负责的有：设计智能风扇的全部功能、编写智能风扇项目代码、录制演示视频、对全部功能进行测试以及代码修复、设计答辩 PPT 并制作程序流程图

组员孙文斌：协助编写代码、协助测试、搭建华为云云控平台的规则与指令、负责答辩演示环节、提供 WIFI 连接和 UDP 指令解析的思路

组员杨昊明：协助设计 PPT，提供 UDP 思路

组员李超楠：协助设计 PPT、提供模式切换思路

1. 设计任务与要求

检测出温湿度

开启和关闭风扇

支持多个档位的风扇

开启和关闭风扇附近的 LED 灯

通过华为云平台观察温湿度值，并控制风扇和 LED 灯

通过按键调整自动和手动模式

自动模式下将温湿度高低和风扇启停进行联动

通过按键调整联动所触发的阈值

通过液晶屏显示合理的内容给予用户正向和负向的反馈，指导用户操作

通过串口诊断常见的软件错误，如网络不通等

可以使用主控板或智能风扇 LED 灯协助用户判断系统状态等

手动模式下解除温湿度与风扇的联动

手动模式下可以通过按键控制风扇，

手动模式下可以通过局域网 socket 控制风扇

手动模式可以通过华为云控制风扇

2. 系统分析

经过对老师给出的实验要求的仔细研究，我们认为可以基于实验 22 的 MOTOR 项目以基础进行开发，同时我们打算在原项目的基础上结合 OLED 显示屏优化原理，WIFI 项目代码、PWM 项目代码、人体感应项目代码以及 UDP

原理等基础之上开发新的系统

我们计划设计共享数据池，其内部包含各个模块工作产生的有效信息，从而实现系统联动，协同调用和数据共享，然后编写合理逻辑与通用数据池的控制函数，从而开发一套完整的智能风扇项目。

3. 设计方案

对于检测温湿度的需求，我们直接将温湿度传感器的 ADC 接入到共享数据中，计划是为温湿度和按键等各类传感器单独创建一个进程，然后在进程中高速刷新，保证数据及时准确。

开启和关闭风扇直接应用 GPIO 的电平调整即可，风扇挡位的调整我们可以应用 PWM 占空比来控制，同理，LED 灯也是通过调整 GPIO 电平来实现

华为云平台和 UDP 局域网指令方面我们是打算在连接 WIFI 后，新建进程，在进程中高速刷新，查看云端是否有符合对应指令出现，然后对接收到的命令进行解析，利用云端命令，修改本地数据池中的各个参数，从而调控各个部件的工作状态

按键调整模式需求方面，我们计划了 3 中工作状态，分别是手动模式、贴近模式（红外探头检测人体时触发）、自动模式（根据温湿度自动触发）、不管当前处于这三种的哪种模式我们都可以随时使用华为云来进行操控。我们计划使用一个变量表示当前的系统状态，按键进程经过防抖操作后，直接对这个状态变量进行修改，接下来当主进程的下一次工作扫描时若发现模式变量值改变则进入不同的工作模式，实现模式切换。

自动模式联动方面，我们计划设计一个电机进程，电机进程不断高速扫描上述温湿度模块存储于全局变量池中的最新温湿度数据，若超过设定的阈值，且上述的模式变量处于自动模式，那么则启动震动模块，低于阈值时则停止。

调整阈值方面，我们计划使用 UDP 或华为云的单独进程来接受云端指令，然后对全局变量池中的温湿度阈值进行修改，这样当下次自动模式时即可根据最新的阈值触发电机。

液晶屏显示方面，我们设计单独的进程来不断刷新全局变量池中最新的数据，并打印到显示屏上，计划分为几个部分：第一个是顶部状态栏，我们

打算连接到 NTP 时间服务器并在左上角显示当前日期，中部是红外感应器状态，当靠近时显示 CLOS，远离时显示 AWAY。右上角显示 WIFI 的状态，断开连接时显示----，连接成功后显示 WIFI 名称前四位。第二个是数据显示区，分为四个部分，分别是转速，模式，温度，湿度。

针对全局刷新速度慢的情况下，我们决定使用局部刷新。也就是只在初始化时刷新固定不变的屏幕部分，不使用清屏命令，而直接打印新的应更新数值。对于数据长度不一致而导致的旧字母残留问题我们使用固定长度的空格占位即可解决。

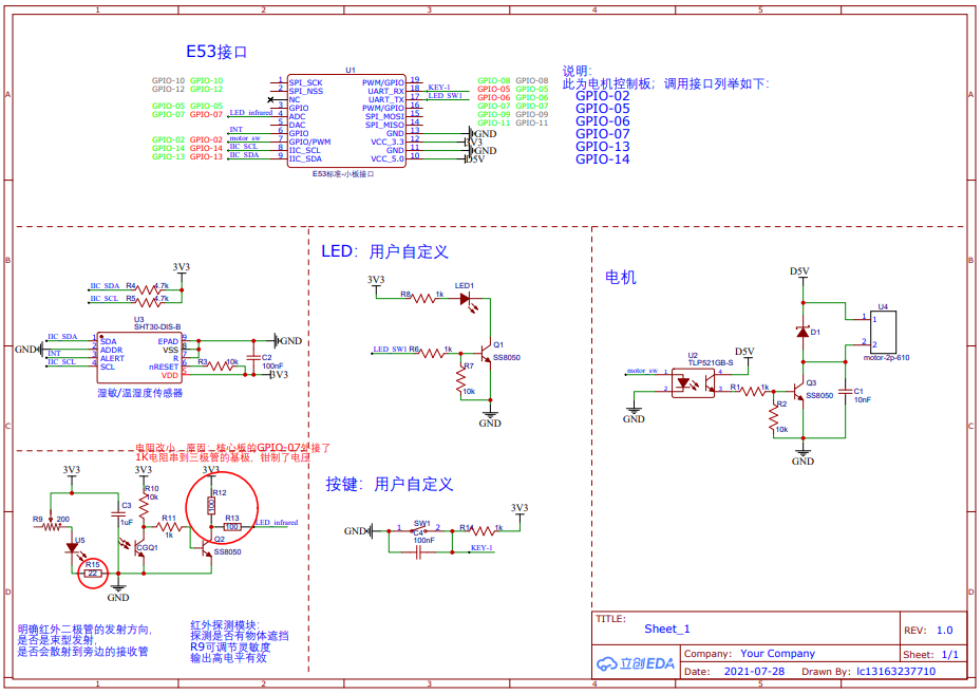
串口输出部分主要是方便于编写代码的调试过程，我们在代码每个关键位置使用 printf 或者调试信息打印来确认程序的状态。或者是收到云端下发指令时打印收到的包的信息。以确定收到网络信息的丢包情况。

至于判断系统状态我们已计划和操作提示一同设计在 OLED 屏幕中，LED 灯我们设计用来指示电机启停状态，或通过华为云、UDP 单独操作

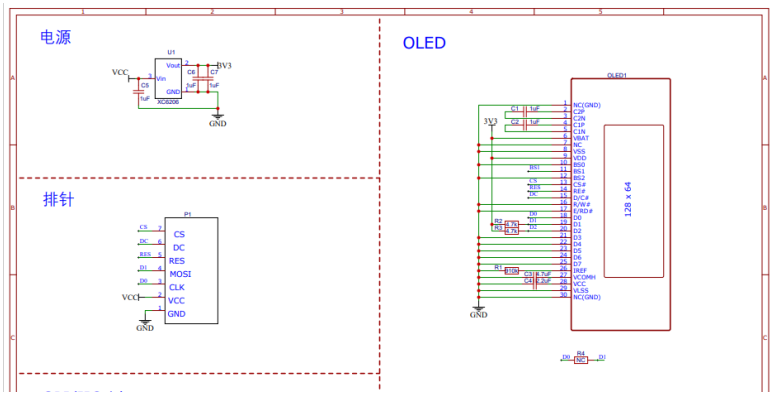
控制风扇方面，我们计划在手动模式下每按一下按键增加百分之 20 在转速。为了进行更精细的操控，减少用户点按次数，我们还计划利用红外传感器来调整风扇转速。用户在传感器前轻勾即可调整转速，省去了点按的繁琐。

4. 硬件电路图

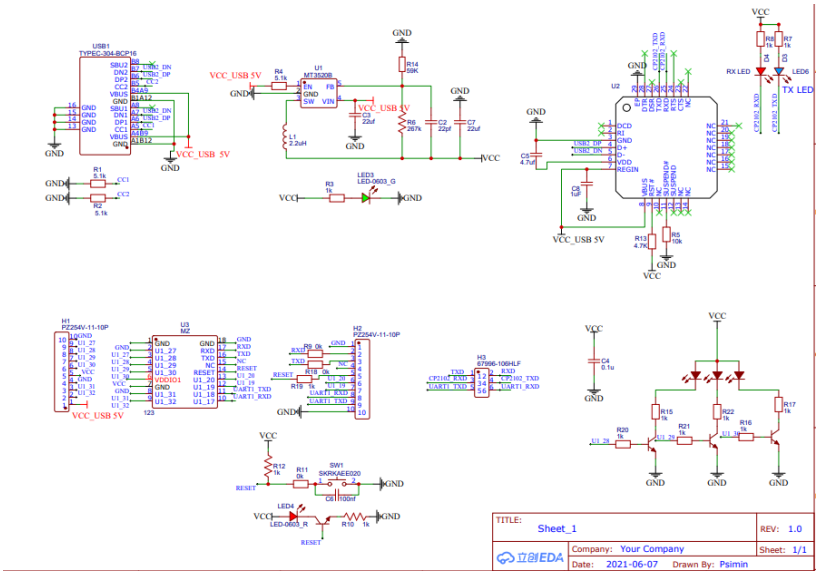
智能风扇模块



OLED 显示屏模块



主控板模块



底板：

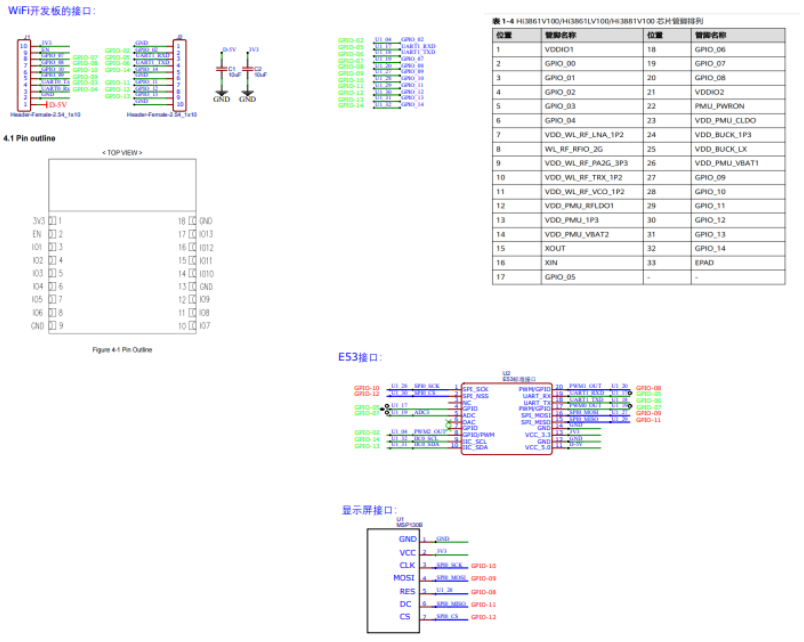


表 1-4 Hi3861V100/Hi3861LV100/Hi3881V100 芯片管脚排列

位置	管脚名称	位置	管脚名称
1	VDDIO1	18	GPIO_06
2	GPIO_00	19	GPIO_07
3	GPIO_01	20	GPIO_08
4	GPIO_02	21	VDDIO2
5	GPIO_03	22	PMU_PWRON
6	GPIO_04	23	VDD_PMU_CLDO
7	VDD_WL_RF_LNA_1P2	24	VDD_BUCK_1P3
8	WL_RF_RFIO_2G	25	VDD_BUCK_LX
9	VDD_WL_RF_PA2G_3P3	26	VDD_PMU_VBAT1
10	VDD_WL_RF_TRX_1P2	27	GPIO_09
11	VDD_WL_RF_VCO_1P2	28	GPIO_10
12	VDD_PMU_RFLDO1	29	GPIO_11
13	VDD_PMU_1P3	30	GPIO_12
14	VDD_PMU_VBAT2	31	GPIO_13
15	XOUT	32	GPIO_14
16	XIN	33	EPAD
17	GPIO_05	-	-

5. 软件设计

项目创建 5 个进程，分别负责不同的工作，协同实现最终功能：

1.温湿度传感器进程

2.风扇 LED 按键红外进程

3.OLED 屏幕打印进程

4.WIFI、日期、云指令进程

5.UDP 服务&&解析进程

全局变量池变量介绍：

Infrared_info

该变量在传感器进程中高速刷新，红外探头的值被简化为 0，1，代表探头前是否有物体遮挡。我们利用该值进行了：A、在 CLOSER 贴近模式中，当用户靠近探头时，利用对 Infrared 的检索启动电机。B、在 ADJUST 手动模式中，使用红外探头实现手指悬浮调值功能。C、同时在 OLED 屏幕进程中，手指靠近时屏幕右上角显示 CLOS，否则显示 AWAY。

Temperature/Humidity

传感器进程共享的检测信息。所有进程在需要温湿度查询时只需要读取即可

is_connected

WIFI 连接状态，关联于屏幕 WIFI 状态，指令接受，日期刷新等

Temperature/Humidity--Threshold

自动模式（AUTO，根据温湿度触发）的温湿度触发阈值，可以根据华为云和 UDP 终端更改该阈值。所以程序中写定的阈值只是初始状态。

auto_fan_strength

自动模式的风扇执行强度，可以根据华为云和 UDP 终端更改。

RPM

当前状态下的风扇强度，在各个模式下均需要被调用，当其接受到华为云或 UDP 终端的修改命令后，转速即刻改变，同时 MODE 强制归为 ADJUST（手动）。

locker_lan_led

LED 锁，当其为-1 时不加锁，所有进程都可操作 LED 灯，否则 LED 被强制点亮（1）或熄灭（0）,受控于 UDP 和华为云

全局变量池变量介绍：

按键、LED、红外、电机模块：

LED 控制锁

LED 控制锁主要服务于云控制，当锁禁止 LED 后，马达的启动将与 LED 断开关联。锁关闭时则同步执行

风扇电机

当为手动模式时，电机关联于手动调节数值，通过按键与红外协同控制强度，当为距离模式时，仅通过红外与默认风扇强度启动电机。为温湿度传感模式时，根据传感器与阈值来执行电机操作。无论在哪个模式下，都可以通过华为云与 UDP 来控制

按键防抖

通过 SLEEP(1)函数防止一次点击被识别多次

红外操控

在 ADJUST 模式下，按键上升调整数值过于繁琐，而长按识别误触率高，所以我们使用 ADJUST 红外通过手指触发上升操作，只需要用手指在红外探头前轻勾即可调整强度。

按键，红外综合逻辑判断

读取当前数据池 MODE 以及点击次数、红外探头数值后执行跳转动作，分别是 CLOSER->AUTO->ADJUST(0~100)

OLED 屏幕部分刷新流程介绍：

局部刷新

我们在实验初期发现屏幕的刷新率上限很低，即使 SLEEP 函数被禁用仍然无效。且每次刷新的速度都很慢，影响操作。所以我们将内容分为两部分，固定不刷新部分和更新部分，然后不使用 CLEAR 命令，直接覆盖，实现刷新。

基础框架

屏幕只打印一次的四个部分，横线竖线，以及数值标题。这一部分不需要刷新，所以只在初始化 INIT 阶段打印一次。

局部固定部分

```
OLED_Clear(0);
GUI_DrawLine(0, 10, WIDTH-1, 10,1);
GUI_DrawLine(WIDTH/2-1,11,WIDTH/2-1,HEIGHT-1,1);
GUI_DrawLine(0,10+(HEIGHT-10)/2-1,WIDTH-1,10+(HEIGHT-10)/2-1,1);
GUI_ShowString(0,39,"FAN_MODE",8,1);
GUI_ShowString(0,13,"STH",8,1);
GUI_ShowString(WIDTH/2+1,13,"TEMP",8,1);
GUI_ShowString(WIDTH/2+1,39,"HUMI",8,1);
while(1)
{
```

变化部分

变化数值部分包括：日期，WIFI 状态，红外状态，温度，湿度，风扇转速，模式。

局部刷新部分

```
void TEST_Menu2(char* mystr1,char* mystr2,char* mystr3,char* mystr4,char* my
{
#if 1
    extern int year;
    extern int month;
    extern int day;
    extern float Temperature;
    extern float Humidity;
    extern int RPM;
    extern int mode;
    extern hi_gpio_value Infrared_info;
    extern int is_connected;
    extern char out_name[50];
#endif
    sprintf(mydate,"%04d-%02d-%02d",year,month,day);
    GUI_ShowString(0,1,mydate,8,18);
    sprintf(mystr3,"%s",mode==0?"CLOS":mode==1?"AUTO":"ADJU");
    GUI_ShowString(0+8,HEIGHT-16,mystr3,16,1);
```

温湿度监控进程介绍：

数据共享

温湿度监控主要算法在 sht3x_i2c 文件中来实现，其他程序使用 EXTERN 来共享读取这些数据

```
float Temperature = 0;
float Humidity = 0;
int RPM = 0;
hi_gpio_value Infrared_info = 0;
int judge_temp;
int norm_temp = 512;
int norm_hp = 512;
```

温湿度防抖

温湿度数据收到各种干扰的情况下会发生异变。为了消除这些错误数据，我们设计了防抖算法。如下所示。

```
Humidity = *humidity;
if(norm_temp == 512){
    if(Temperature != 0){
        norm_temp = Temperature;
    }
}
else{
    judge_temp = norm_temp - Temperature;
    if(judge_temp>=20||judge_temp<=-20){
        Temperature = norm_temp;
    }
    else{
        norm_temp = Temperature;
    }
}
```

```
if(norm_hp == 512){
    if(Humidity != 0){
        norm_hp = Humidity;
    }
}
else{
    if(Humidity == 0){
        Humidity = norm_hp;
    }
    else{
        norm_hp = Humidity;
```

WIFI 和日期同步进程介绍：

日期更新

访问国家授时中心 IP 获取时间戳，转换后赋给全局变量 YEAR,MONTH,DAY，以供屏幕下次刷新时显示最新日期。

网络命令解析

重写函数 DEAL_CMD_MSG，通过 JSON 解包获取各个字符串，IF 判断执行对应命令的对应值修改或者通过 cJSON_GetNumber

Value 解析发来的 INT 值，直接进行赋值

```
temp_threshold = (int)cJSON_GetNumberValue(obj_para);
if(temp_threshold > 100 || temp_threshold <-100){
    printf("temp_threshold illegal %s\r\n", cJSON_GetStringValue(obj_para));
    temp_threshold = 30;
}
printf("New Temperature_Threshold %d\r\n", temp_threshold);
cmdret = 0;
```

```
memset(recvBuf, 0, sizeof(recvBuf));
client_addr.sin_port = htons(123);
client_addr.sin_family = AF_INET;
client_addr.sin_addr.s_addr = inet_addr("210.72.145.44");
ret = sendto(sock_fd, sendBuf, strlen(sendBuf), 0, (struct sockaddr *)&client_addr, sizeof(client_addr));
if (ret < 0){
    printf("Time server send failed!\r\n");
}
memset(recvBuf, 0, sizeof(recvBuf));
ret = recvfrom(sock_fd, recvBuf, sizeof(recvBuf), 0, (struct sockaddr*)&client_addr,(socklen_t*)&size_client_addr);
```

```
if(0 == strcmp(cJSON_GetStringValue(obj_cmdname),"Fan_Status_Control")){
    printf("fun1\r\n");
    obj_para = cJSON_GetObjectItem(obj_paras,"Fan_Status");
    mode = 2;
    if(NULL == obj_para){
        printf("OBJ_PARA_NULL\r\n");
        return;
    }
    else if(0 == strcmp(cJSON_GetStringValue(obj_para),"LEVEL01")){
        RPM = 1;
        IoTPwmStart(motor_pwm_controller, 1, freq);
        printf("LEVEL01\r\n");
    }
    else if(0 == strcmp(cJSON_GetStringValue(obj_para),"LEVEL02")){
        RPM = 20;
        IoTPwmStart(motor_pwm_controller, 10, freq);
        printf("LEVEL02\r\n");
    }
```

UDP 指令扫描介绍：

连接华为云

配置华为云 IP 与端口后，循环检查是否收到了 UDP 数据包，如下所示，当检测到 UDP 数据包后进行解析，同时类似于华为云云控解析函数类似的进行命令执行，如下所示。

```c
if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
    perror("socket is error.\r\n");
}
bzero(&server_sock, sizeof(server_sock));
server_sock.sin_family = AF_INET;
server_sock.sin_addr.s_addr = htonl(INADDR_ANY);
server_sock.sin_port = htons(8888);
if (bind(sock_fd, (struct sockaddr *)&server_sock, sizeof(struct sockaddr)) == -1)
{
    perror("bind is error.\r\n");
}
int ret;
char recvBuf[512] = {0};
char sendBuf[512] = "\x1b\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0";
struct sockaddr_in client_addr;
int size_client_addr= sizeof(struct sockaddr_in);
while (1)
{
    printf("Waiting to receive data...\r\n");
    memset(recvBuf, 0, sizeof(recvBuf));
    ret = recvfrom(sock_fd, recvBuf, sizeof(recvBuf), 0, (struct sockaddr*)&client_addr,(socklen_t*)&size_client_addr);
    if(ret < 0)
    {
        printf("UDP server receive failed!\r\n");
```

```c
int length = strlen(message);
int whichdata = message[0] - '0';
int value;
if(message[3]>='0'&&message[3]<='9'){
    value = (message[2]-'0')*10+(message[3]-'0');
}
else{
    value = message[2]-'0';
}
printf("%d %d udp \r\n",whichdata, value);
if(whichdata == 1){
    RPM = value;
    mode = 2;
    if(value > 0){
        IoTPwmStart(motor_pwm_controller, value, freq);
        printf("MADAR START\r\n");
        return "MADAR START\r\n";
    }
    else{
        IoTPwmStop(motor_pwm_controller);
        printf("MADAR STOP\r\n");
        return "MADAR STOP\r\n";
    }
}
```

程序总流程介绍：

基础变量定义：

```c
#include <stdio.h>
#include <unistd.h>
#include "motor_module.h"
#include "hi_timer.h"
#include "hi_gpio.h"
#include "hi_pwm.h"
#include "dbg.h"
#include "hi_spi.h"
#include "ohos_init.h"
#include "cmsis_os2.h"
#include "iot_pwm.h"
```

```
#include "wifi_device.h"
#include "wifi_hotspot.h"
#include "wifi_error_code.h"
#include "lwip/netifapi.h"
#define WIFI_SSID   "woyaofangjia"
#define WIFI_PASSWORD "20020617"
#define CLIENT_ID "64a74dbd5c51f150f4d3e2c8_123456789_0_0_2023070702"
#define USERNAME "64a74dbd5c51f150f4d3e2c8_123456789"
#define PASSWORD "2824511944a1554958760d8cfcdb20505074a2de54a644ed017b601a47c87320"
#define MSGQUEUE_OBJECTS 16 // number of Message Queue Objects
#define motorio HI_IO_NAME_GPIO_2
#define pwm_motor_bridge HI_IO_FUNC_GPIO_2_PWM2_OUT
#define motor_pwm_controller HI_PWM_PORT_PWM2
#define PWM_CLK_FREQ 160000000
```

```
#define freq 15000  //min
#define FAN_LED HI_IO_NAME_GPIO_6
#define FAN_LED_GPIO HI_IO_FUNC_GPIO_6_GPIO
extern hi_gpio_value Infrared_info;
extern float Temperature; //环境温度，状态
extern float Humidity; //环境湿度，状态
char out_name[50] = WIFI_SSID;
int year=8888;
int month=8;
int day=8;
int is_connected = 0;
```

华为云、UDP 控制变量部分：

```
//--------------------------------------huawei_data-------------------------
extern int RPM;
int mode = 0;
int auto_fan_strength = 50; //触发时的风扇强度
int temp_threshold = 30; //温度触发阈值
int humi_threshold = 80; //湿度触发阈值
//--------------------------------------------------------------------------------
```

端口初始化及创建任务（由于任务有 5 个且内容相同，不重复展示）：

```
int ret;
motor_gpio_io_init();
SHT3X_init();
// motor_pwm_init();
IoTGpioInit(motorio);
(void)hi_io_set_func(motorio,pwm_motor_bridge);
(void)hi_gpio_set_dir(motorio,HI_GPIO_DIR_OUT);
IoTPwmInit(HI_PWM_PORT_PWM2);
(void)hi_io_set_func(FAN_LED,FAN_LED_GPIO);
(void)hi_gpio_set_dir(FAN_LED,HI_GPIO_DIR_OUT);
hi_spi_deinit(HI_SPI_ID_0);
screen_spi_master_init(0);

ret = hi_task_create(&g_MonitorTask, // task标识 //
    &MonitorTaskAttr,
    MonitorOledTask, // task处理函数 //
    NULL); // task处理函数参数 //

if (ret < 0) {
    printf("Create monitor oled task failed [%d]\r\n", ret);
    return;
}

ret = hi_task_create(&g_MonitorTask, // task标识 //
```

关键函数代码：

解析华为云数据（部分）

```
static void deal_cmd_msg(cmd_t *cmd)
{
    // {"paras":{"LED_Satus":"ON"},"service_id":"SmartFan_Service","command_name":"LED_Status_Control"}
    cJSON *obj_root;
    cJSON *obj_cmdname;
    cJSON *obj_paras;
    cJSON *obj_para;
    int cmdret = 1;
    oc_mqtt_profile_cmdresp_t cmdresp;
    obj_root = cJSON_Parse(cmd->payload);
    if(NULL == obj_root){
        printf("OBJ_ROOT_NULL\r\n");
        return;
    }
    obj_cmdname = cJSON_GetObjectItem(obj_root,"command_name");
    printf("got %s \r\n", cJSON_GetStringValue(obj_cmdname));
    obj_paras = cJSON_GetObjectItem(obj_root,"paras");
    if(NULL == obj_cmdname){
        cmdresp.paras = 1;
        cmdresp.request_id = cmd->request_id;
        cmdresp.ret_code = cmdret;
        cmdresp.ret_name = "success";
        (void)oc_mqtt_profile_cmdresp("64a74dbd5c51f150f4d3e2c8_123456789",&cmdresp);
        printf("cmdname_null\r\n");
        return;
    }
```

执行云命令：

```
else if(0 == strcmp(cJSON_GetStringValue(obj_cmdname),"LED_Status_Control")){
    printf("fun2\r\n");
    obj_para = cJSON_GetObjectItem(obj_paras,"LED_Satus");
    if(NULL == obj_para){
        printf("OBJ_PARA_NULL\r\n");
        return;
    }
    else if(0 == strcmp(cJSON_GetStringValue(obj_para),"ON")){
        locker_lan_led = -1;
        control_fan_led(1);
        locker_lan_led = 1;
        printf("LED ON\r\n");
    }
```

扫描网络命令：

```
static int network_main_entry( void )
{
    WifiConnect(WIFI_SSID,WIFI_PASSWORD);
    device_info_init(CLIENT_ID,USERNAME,PASSWORD);
    oc_mqtt_init();
    oc_set_cmd_rsp_cb(oc_cmd_rsp_cb);

    while(1){
        if(is_connected){
            update_date();
        }
        app_msg = NULL;
        (void)osMessageQueueGet(mid_MsgQueue,(void **)&app_msg,NULL, 0U);
        if(NULL != app_msg){
            switch(app_msg->msg_type){
                case en_msg_cmd:
                    deal_cmd_msg(&app_msg->msg.cmd);
```

模式变量识别执行函数：

```
hi_void exec_fan(int mode){
    if(mode == 0){
        if(Infrared_info){ //距离
            RPM = auto_fan_strength;
            IoTPwmStart(motor_pwm_controller, auto_fan_strength, freq);
            control_fan_led(1);
        }
        else{
            RPM = 0;
            IoTPwmStop(motor_pwm_controller);
            control_fan_led(0);
        }
    }
    else if(mode == 1){
        if(Temperature > temp_threshold || Humidity > humi_threshold){//温湿度
            RPM = auto_fan_strength;
            IoTPwmStart(motor_pwm_controller, auto_fan_strength, freq);
            control_fan_led(1);
        }
        else{
            RPM = 0;
            IoTPwmStop(motor_pwm_controller);
            control_fan_led(0);
        }

    }
```

UDP 解析函数：

```
char* exec_udp(char* message){
    //1 马达
    //2 LED
    //3 temp thr
    //4 humi thr
    //5 auto strg
    //6 mode
    int length = strlen(message);
    int whichdata = message[0] - '0';
    int value;
    if(message[3]>='0'&&message[3]<='9'){
        value = (message[2]-'0')*10+(message[3]-'0');
    }
    else{
        value = message[2]-'0';
    }
    printf("%d %d udp \r\n",whichdata, value);
    if(whichdata == 1){
        RPM = value;
        mode = 2;
        if(value > 0){
            IoTPwmStart(motor_pwm_controller, value, freq);
            printf("MADAR START\r\n");
            return "MADAR START\r\n";
        }
        else{
            IoTPwmStop(motor_pwm_controller);
            printf("MADAR STOP\r\n");
            return "MADAR STOP\r\n";
        }
    }
    else if(whichdata == 2){
```

马达启停函数及红外操控

```c
hi_void manage_motor(hi_void){
    exec_fan(mode);
    if(mode == 2){
        if(Infrared_info&&RPM!=0){
            rpmi++;
            if(rpmi % 10000 == 0){
                RPM = RPM + 1;
                rpmi ++;
            }
            if(RPM >= 99){
                RPM = 0;
            }
        }
    }
    if(get_info(HI_GPIO_IDX_5,HI_GPIO_VALUE1) == 0){
        if(mode == 0){
            mode = 1;
            RPM = 0;
        }
        else if(mode == 1){
            mode = 2;
            RPM = 0;
        }
        else{
            if(RPM == 0){
                RPM = 1;
            }
            else if(RPM >= 80){
                mode = 0;
            }
            else{
                RPM = RPM + 20;
            }
```

日期刷新：

```c
void update_date(char* time){
    int ay=2023,am=7,ad=9;
    time_t PTime = 0;
    struct tm* timeP;
    PTime += (8*60*60);
    parse_timet(&ay,&am,&ad,time,timeP);
    year = ay;
    month = am;
    day = ad;
}
```

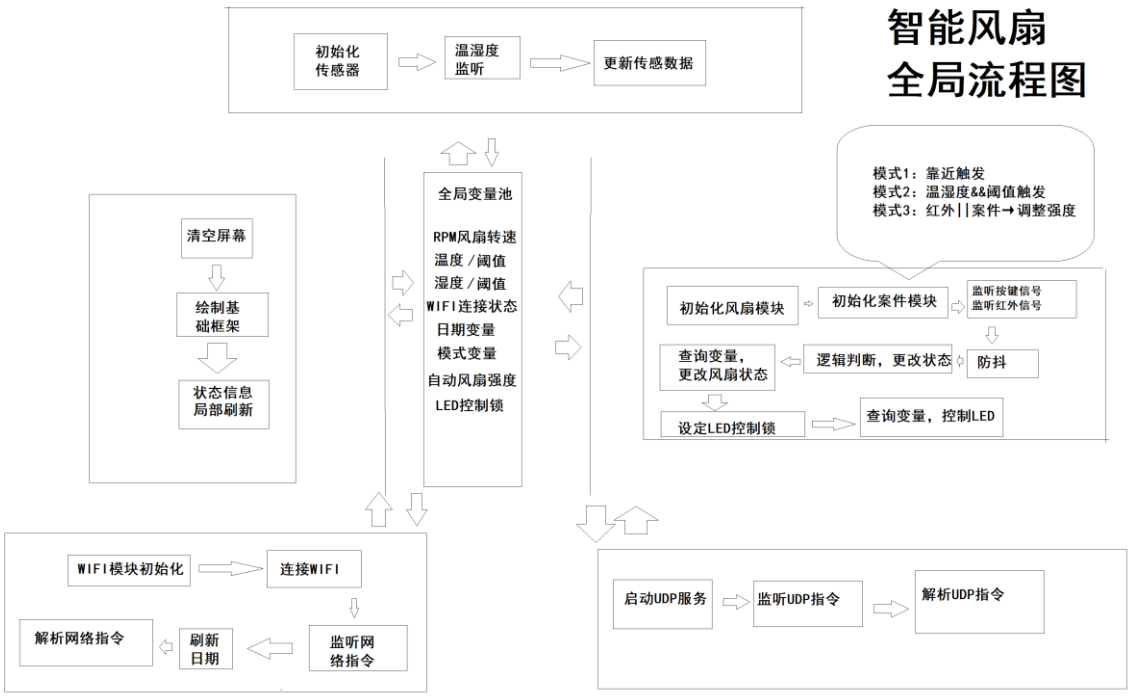
UDP 监听：

```c
//在sock_fd 进行监听
int sock_fd;
//服务端地址信息
struct sockaddr_in server_sock;

if ((sock_fd = socket(AF_INET, SOCK_DGRAM, 0)) == -1)
{
    perror("socket is error.\r\n");
}
bzero(&server_sock, sizeof(server_sock));
server_sock.sin_family = AF_INET;
server_sock.sin_addr.s_addr = htonl(INADDR_ANY);
server_sock.sin_port = htons(8888);
if (bind(sock_fd, (struct sockaddr *)&server_sock, sizeof(struct sockaddr)) == -1
{
    perror("bind is error.\r\n");
}
int ret;
char recvBuf[512] = {0};
char sendBuf[512] = "\x1b\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0\0
struct sockaddr_in client_addr;
int size_client_addr= sizeof(struct sockaddr_in);
while (1)
{
    printf("Waiting to receive data...\r\n");
    memset(recvBuf, 0, sizeof(recvBuf));
    ret = recvfrom(sock_fd, recvBuf, sizeof(recvBuf), 0, (struct sockaddr*)&client
```

6. 程序流程图

根据上述设计的详细细节，我们的程序流程图主要分为前面所述的五大模块，以及全局变量池。



7. 程序清单

include	2023/7/10 9:41	文件夹	
BUILD.gn	2023/7/8 13:45	GN 文件	2 KB
dbg.c	2023/7/8 13:45	C Source File	1 KB
E53_IA1.c	2023/7/8 13:45	C Source File	10 KB
gui.c	2023/7/8 13:45	C Source File	22 KB
led_screen_main.c	2023/7/8 13:45	C Source File	1 KB
motor_module.c	2023/7/8 13:45	C Source File	24 KB
oc_mqtt.c	2023/7/8 13:45	C Source File	14 KB
oc_mqtt_profile_package.c	2023/7/8 13:45	C Source File	15 KB
oled.c	2023/7/8 13:45	C Source File	13 KB
sht3x_i2c.c	2023/7/8 13:45	C Source File	9 KB
spi_screen.c	2023/7/8 13:45	C Source File	3 KB
test.c	2023/7/8 13:45	C Source File	9 KB
wifi_connect.c	2023/7/8 13:45	C Source File	8 KB

主要的重要程序有：

E53_IA1

配置 UDP 连接的相关配置

LED_SRCEEN_MAIN

配置 LED 屏幕的总体布局框架与静态部分，引入全局数据并进行格式化与初步处理。

MOTOR_MODULE

项目最重要的主要文件，包含了内部除屏幕模块外几乎全部进程以及全局变量池。

SHT3X_I2C

传感器处理模块，内部添加了数据处理，防抖，以及共享操作

TEST

OLED 屏幕的变化部分的布局与数据格式化安排

WIFI_CONNECT

配置 WIFI 的有关信息

8. 系统调试过程

遇到问题 1：LED 屏幕刷新率过低，操作不连贯

调试解决办法：最后制定了部分刷新方法如上文所述，只刷新变化数据部分，固定基础部分。残留的旧字符使用空格占位清除。

遇到问题 2：发送指令后本地收到指令，但不解析

调试方法：将解析步骤提前，使系统一旦检测到发来的命令立刻执行解析动作并执行命令。

遇到问题 3：按键容易误识别

调试方法：使用按键防抖，利用 sleep 函数，在一次点击触发后保持休眠一秒再第二次检测，防止误识别

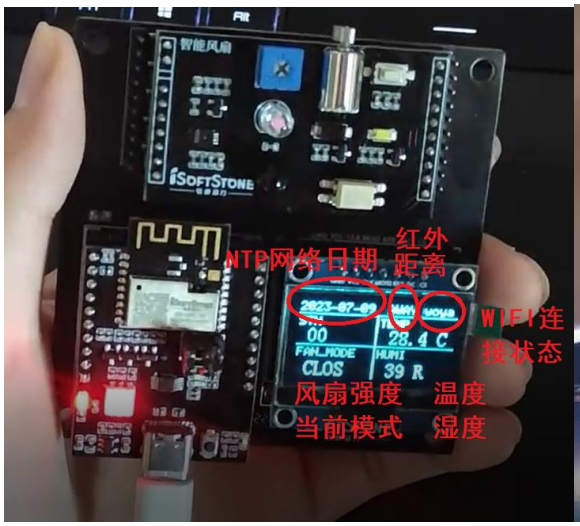
遇到问题 4：温湿度由于接触不良等原因出现骤变

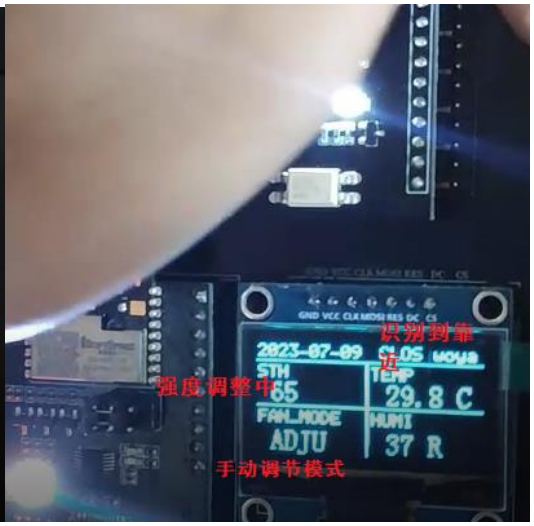
调试方法：编写温湿度数据防抖，在同步数据到变量池之前先判断数据是否合理，如果合理则记录下来，如果明显不合理（即前后差距过大）则将最近一次的正确记录作为该时刻的温湿度。

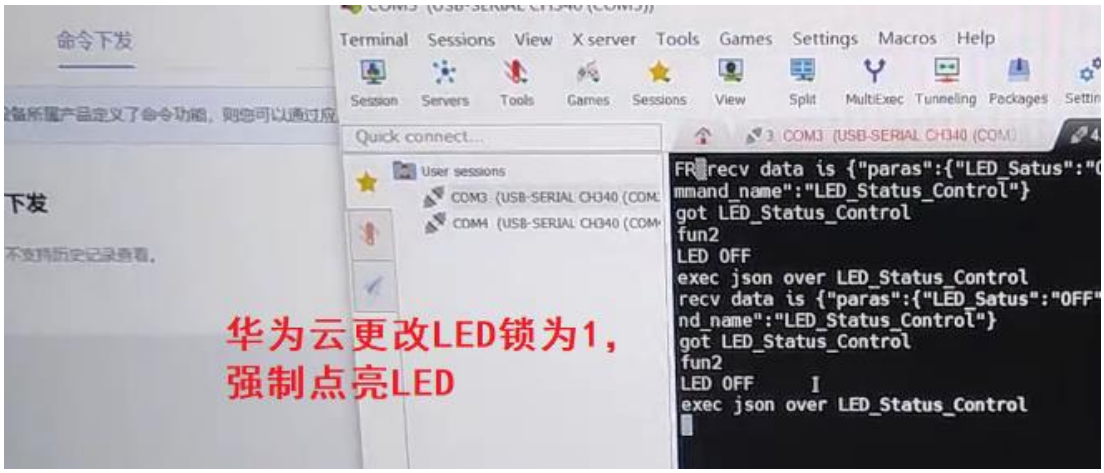
遇到问题 5：WIFI 无法连接，DHCP 不启动

调试方法：发现 WIFI_SSID 变量数组开小了，扩大申请长度即可。

运行情况展示：（详见演示视频）







9. 设计总结

我们在本次课程设计中实现了众多功能，包括：

1.温湿度监测

2.关联模式/手动模式/红外模式

3.UDP/华为云指令操控：即刻转速，默认转速，模式，阈值，LED

4.红外操作/局部刷新/WIFI 状态状态显示/NTP 时间同步

10. 心得体会

本项目代码量很大，平常实验都是按单个模块来进行实现的，但这个实验考察我把零散的模块联动起来，通过全局变量池进行数据共享和分析，同时更应该注重整体的使用体验。比如用户屏幕界面的布局设计美观，频繁点按按键的简化，屏幕刷新的显示效果是不是流畅，屏幕刷新的时机与按键按下是否配合等。

在做这个项目时，我经常思考用现有的开发板还能开发出什么功能。比如红外操控数字和红外感应其实并不在原有的要求中，但当我看到红外探头被闲置时，我认为应该将这个资源利用起来，由此写了红外部分的程序。手指靠近调整强度的用法我认为比按键要更加有趣，只需要在探头前轻轻勾下手指就能调整参数。同时，贴近触发电机旋转也让我想起一种小时候戴的帽子，上面是太阳能板，当有阳光的时候人感到热，这时恰好能启动电机吹风就很适宜。而在这个项目中我设计人们拿起风扇靠近时可以自动启动，放下不用的时候自动停止节省电量，我认为我实现的贴近方法也是类似思想。

此外，在默认的 motor 程序中左上角显示的是一个固定的日期，开始时我以为是已经实现好的日期联网同步，仔细看完源代码后才发现只是字符串而已。本着改进的想法，我向其中加入了联网时间同步，实现了真正意义的日期显示。恰好也可以利用上 WIFI 模块的资源

同时，我也尽可能想起了单个设备的控制，因为我既想单独控制 LED，又希望它自动一点，能跟随电机启停亮灭，所以我给 LED 设置了控制锁，当你想手动点亮熄灭的时候，其他操作不影响这个过程，当他的锁被解除后，它也可以跟随电机来启动停止。

总的来说，这个项目多多少少还存在一些 bug 需要修复完善，但利用这些硬件能做的功能我认为我完成的还是比较全面的，编写这个项目花费了我很多精力，也让我收获了许多

11. 参考资料

1. 启航开发板实验代码 README 文档
2. OPENHARMONY 开发者文档

二、平时实验项目实验报告

1. 实验一、事件

1.1 待完成功能

使用启航开发板来完成操作系统中事件的实现。事件是一种任务间通信的机制，用于实现任务间的同步。一个任务可以等待多个事件的发生，任意一个事件发生时唤醒任务进行事件处理。几个事件都发生后才唤醒任务进行事件处理。多任务环境下，任务之间需要同步操作，一个等待即是一个同步。事件可以提供一对多、多对多的同步操作。其中一对多同步模型是一个任务等待多个事件的触发。而多对多同步模型则是多个任务等待多个事件的触发。任务可以通过创建事件控制块来实现对事件的触发和等待操作。

事件接口不与任务相关联，事件相互独立，内部实现为一个 32 位的无符号整型变量，用于标识该任务发生的事件类型，其中每一位表示一种事件类型：0 表示该事件类型未发生。而 1 表示该事件类型已经发生。事件仅用于任务间的同步，不提供数据传输功能。多次向任务发送同一事件类型等效于只发送一次。同时，多个任务可以对同一事件进行读写操作。本实验要求支持事件读写超时机制。另外软件定时器提供的是 2 类定时器机制：包括单次触发定时器：在启动后只会触发一次定时器事件，然后定时器自动删除。还有周期触发定时器：会周期性地触发定时器事件，直到用户手动停止定时器，否则将永远持续执行。

1.2 实验原理图

1.3 实验代码

```c
#include <stdio.h>
#include "hi_task.h"
#include "hi_event.h"
#include "ohos_init.h"
#define TEST_EVENT 1

hi_u32 event1,taskid;

const hi_task_attr Monitor1 = {
    .task_name = "task_test", // 任务名称
    .task_prio = 20,          // 优先级
    .stack_size = 4096        // 大小
};

void *example_event(void){
    hi_u32 ret;
    hi_u32 uwEvent;
    printf("example_event wait event 0x%x \n", TEST_EVENT);
    /*超时等待方式读事件,超时时间为200ms 若200ms 后未读取到指定事件，读事件超时，任务直接唤醒*/
    ret = hi_event_wait(event1, TEST_EVENT,&uwEvent,200,HI_EVENT_WAITMODE_AND);
    if (ret == HI_ERR_SUCCESS) {
        printf("example_event read event :0x%x\n", uwEvent);
    } else {
        printf("example_event read event fail!\n");
    }
    printf("example_task_entry_event event clear success.\n");
    return HI_NULL;
}

  void event_demo(void){ /*创建任务*/
      /*创建任务*/
      hi_u32 ret;
      hi_event_create(&event1);
      ret = hi_task_create(&taskid, &Monitor1, example_event, 0);
      if (ret!=HI_ERR_SUCCESS) {
          printf("create task failed .\n");
          return HI_ERR_FAILURE;
      }
      /*写用例任务等待的事件类型*/
      printf("example_task_entry_event write event .\n");
      ret = hi_event_send(event1, TEST_EVENT);
      if (ret != HI_ERR_SUCCESS)
      {
          printf("event write failed\n");
          return HI_ERR_FAILURE;
      }
      printf("example_task_entry_event event write success .\n");

  }

  APP_FEATURE_INIT(event_demo);

```

在上述程序的设计中，首先包含任务和事件相关的函数和数据类型的所需头文件，然后我们设定两个全局变量 event1 和 taskid 用于事件和任务的标识。接下来配置任务的属性，定义 Monitor1 的 hi_task_attr 结构体，其中需要设置了任务名称为 "task_test"，优先级 20，堆栈大小 4096。

然后定义函数 example_event 用于处理事件的等待和读取操作。首先在 example_event 函数中打印一条消息表示等待事件的开始。调用 hi_event_wait 函数等待事件的发生。函数参数包括事件标识、待等待的事件类型、用于保存读取到的事件、超时时间和等待模式标记量等。hi_event_wait 函数返回有效值后即可判断等待是否成功。如果返回值 HI_ERR_SUCCESS 表示成功读取到事件，即可打印读取到的事件值，否则就代表失败。回到主函数，最后打印消息表示事件处理完成并返回 HI_NULL。事件的创建和发送主要是在 event_demo 函数中实现，在 event_demo 中首先调用 hi_event_create 函数创建事件 event1，调用 hi_task_create 函数创建任务，将任务标识保存在 taskid 变量中，参数包括保存任务标，任务属性结构体，任务入口函数和任务入口函数参数。接下来通过返回值 HI_ERR_SUCCESS 判断任务的创建是否成功，并打印消息表示写入事件开始，调用 hi_event_send 函数将事件 TEST_EVENT 发送到 event1，根据 hi_event_send 函数的返回值判断事件的发送结果 。

1.4 编译与运行结果

在 build.gn 文件中选定 EVENT 项目，清除旧版本代码，执行编译命令：

```
# "01_KP_OS_HelloWorld:helloworld_example",
# "02_KP_OS_Task:task_test",
# "03_KP_OS_Mutex:mutex_test",
# "04_KP_OS_Timer:timer_test",
  "05_KP_OS_Event:event_test",
# "06_KP_OS_Semp:semp_test",
# "07_KP_OS_Message:msg_test",
# "08_KP_GPIO_led:gpioled_example",
# "09_KP_PWM_led:pwmled_example",
# "10 KP KEY led:keyled example",
```

```
问题    输出    调试控制台    终端

[OHOS INFO] startup                    5       1.2%          5       1.2%    1.00
[OHOS INFO] third_party                2       0.5%          2       0.5%    1.00
[OHOS INFO] updater                    4       1.0%          4       1.0%    1.00
[OHOS INFO] utils                      4       1.0%          4       1.0%    1.00
[OHOS INFO]
[OHOS INFO] c overall build overlap rate: 1.00
[OHOS INFO]
[OHOS INFO]
[OHOS INFO] qihang build success
[OHOS INFO] cost time: 0:00:06
root@54911a7ea290:/home/openharmony/OpenHarmony#
```

运行结果如下：

```
ready to OS start
sdk ver:Hi3861V100R001C00SPC025 2020-09-03 18:10:00
formatting spiffs...
FileSystem mount ok.
wifi init success!
hilog will init.

hievent will init.

hievent init success.

hiview init success.
example_event wait event 0x1
example_task_entry_event write event .
example_task_entry_event event write success .
example_event read event :0x1
example_task_entry_event event clear success.
No crash dump found!
```

1.5 遇到的问题

在编写本次代码的过程中，我遇到了读事件和写事件顺序混乱的问题，为了查明原因，我仔细研究了读写进程中的等待和写入的优先级顺序以及执行的详细流程，并仔细分析读和写直接的制约关系

1.6 问题解决方案

在写入操作前限制读取进程的执行，防止程序在没有写入的时候就提前读取，保持等待关系

1.7 总结与心得

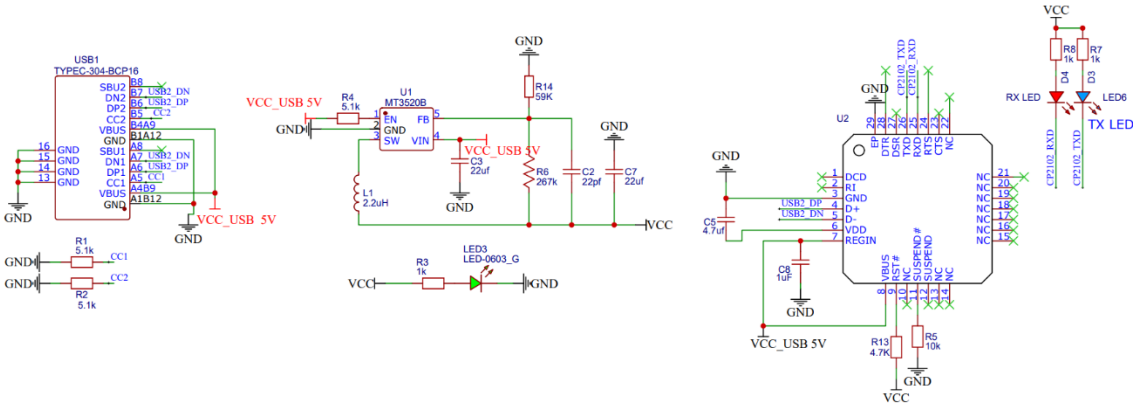
在本实验中我熟悉了在 OPENHARMONY 中编写任务事件的流程，了解了任务和事件的概念，任务是指一个独立的执行单元，可以并发执行，事件是任务之间进行通信和同步的方式之一。同时了解了任务的创建和属性配置，包括创建任务，并通过结构体配置任务名称、优先级和堆栈大小。熟悉了事件的创建和发送，了解事件的创建方式和向事件发送特定的事件类型来执行等待该事件的任务。掌握了事件的等待和读取，了解等待事件的方式。如事件标识、待等待的事件类型、超时时间和等待模式。也理解了任务的优先级和调度机制，任务的优先级决定了任务的执行顺序，需要根据具体需求合理设置任务的优先级以避免任务之间的竞争和冲突。

2. 实验二、呼吸灯

2.1 待完成功能

实现 LED 不断闪烁类似在呼吸的效果。接入的是 GPIO8 号引脚。本实验主要是通过调节 GPIO 的占空比来实现亮度控制。应需要使用相应的 API 完成 GPIO 和 PWM 的初始化设置，使用 API 设置 GPIO 引脚的复用功能和输出方向，根据占空比的递增和递减控制 LED 的亮度，实现呼吸灯效果。即通过 API 调用设置 PWM 的占空比和频率，控制 LED 的亮度和闪烁速度。

2.2 实验原理图



2.3 实验代码

导入头文件：

```
#include <stdio.h>
#include <unistd.h>
#include "ohos_init.h"
#include "cmsis_os2.h"
#include "iot_pwm.h"
#include "iot_errno.h"
#include "iot_gpio.h"
#include "hi_pwm.h"
#include "hi_errno.h"
#include "hi_io.h"
#include "hi_task.h"
#include "hi_gpio.h"
```

定义常量与 GPIO 引脚

```
#define CLK_160M 160000000
#define DUTY_MIN 0
#define DUTY_MAX 100
#define SHORT_MAX 0xFFFF
#define LED_INTERVAL_TIME_US 500
```

```
    #define LED_IO HI_IO_NAME_GPIO_7
    #define LED_IO1 HI_IO_NAME_GPIO_8
    定义 LED 主任务函数
hi_void *LedLightTask(const char *arg){
    初始化 GPIO
    IoTGpioInit(HI_IO_NAME_GPIO_8);
    (void)hi_io_set_func(HI_IO_NAME_GPIO_8, HI_IO_FUNC_GPIO_8_PWM1_OUT);
    IoTGpioSetDir(HI_IO_NAME_GPIO_8, HI_GPIO_DIR_OUT);
    IoTPwmInit(HI_PWM_PORT_PWM1);
    IoTGpioInit(HI_IO_NAME_GPIO_7);
    (void)hi_io_set_func(HI_IO_NAME_GPIO_7, HI_IO_FUNC_GPIO_7_PWM0_OUT);
    IoTGpioSetDir(HI_IO_NAME_GPIO_7, HI_GPIO_DIR_OUT);
    IoTPwmInit(HI_PWM_PORT_PWM0);
    定义占空比，频率，闪烁模式
    unsigned short duty = 0;
    unsigned int freq = 15000;
    static char plus_status = 0;
    int color = 0;
    进入闪烁主循环
    while(1){
    第一种颜色的闪烁周期
        if(color == 0){
            while (1)
            {
    渐亮渐灭过程切换
                if (duty >= 99){
                    plus_status = 0;
                }
                else if (duty <= 0){
                    plus_status = 1;
                }
    渐亮渐灭占空比调整
                if (plus_status){
                    duty++;
                }
                else{
                    duty--;
                    if(duty == 0){
                        color = 1;
                        break;
                    }
                }
    闪烁速度调整，通过 USLEEP 实现
                usleep(LED_INTERVAL_TIME_US);
```

```
        IoTPwmStart(HI_PWM_PORT_PWM1, duty, freq);
    }
}
if(color == 1){
    while (1)
    {
        if (duty >= 99){
            plus_status = 0;
        }
        else if (duty <= 0){
            plus_status = 1;
        }
        if (plus_status){
            duty++;
        }
        else{
            duty--;
            if(duty == 0){
                color = 2;
                break;
            }
        }
        usleep(LED_INTERVAL_TIME_US);
        IoTPwmStart(HI_PWM_PORT_PWM0, duty, freq);
    }
}
if(color == 2){
    while (1){
        if (duty >= 99){
            plus_status = 0;
        }
        else if (duty <= 0){
            plus_status = 1;
        }
        if (plus_status){
            duty++;
        }
        else{
            duty--;
            if(duty == 0){
                color = 0;
                break;
            }
        }
```

```
                usleep(LED_INTERVAL_TIME_US);
                IoTPwmStart(HI_PWM_PORT_PWM0, duty, freq);
                IoTPwmStart(HI_PWM_PORT_PWM1, duty, freq);
            }
        }
    }
    return NULL;
}
```

2.4 编译与运行结果

最终实现了一个周期三种颜色轮播，分别是红色呼吸，蓝色呼吸，紫色呼吸：

2.5 遇到的问题

我们最开始打算设计一个呼吸灯，然后在调用 PWM 模块时遇到了问题，不清楚如何将 PWM 与 GPIO 关联起来，后来参考了 README 的指南，了解了详细步骤如下：

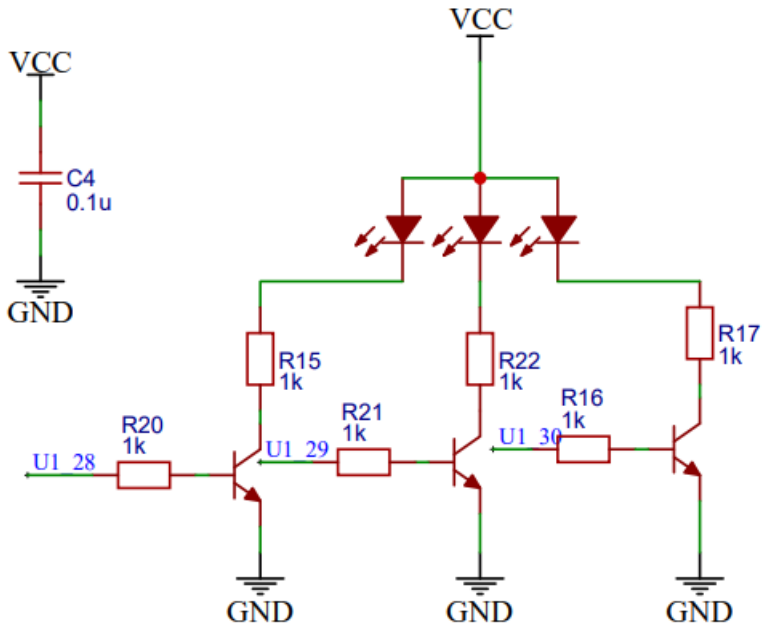
```
IoTGpioInit(HI_IO_NAME_GPIO_8);
(void)hi_io_set_func(HI_IO_NAME_GPIO_8, HI_IO_FUNC_GPIO_8_PWM1_OUT);
IoTGpioSetDir(HI_IO_NAME_GPIO_8, HI_GPIO_DIR_OUT);
IoTPwmInit(HI_PWM_PORT_PWM1);
```

同时，我们在后期打算假如绿色 LED，但不知道绿色 LED 端口是多少，且不

了解绿色 LED 对应的 PWM 是多少。

2.6 问题解决方案

查询电路图如下所示：



了解到三者的端口分别为 2，7，8，且启用 2 号端口时如果连接了智能风扇模块会引发电机转动。

2.7 总结与心得

本实验中我学到了关于使用 PWM 实现 LED 呼吸灯的知识和经验。通过本实验我进一步理解了 PWM 原理和工作方式。PWM 通过调整脉冲的宽度和周期来控制输出信号的占空比从而实现对 LED 的亮度调节。这在嵌入式系统中非常常见且对于控制各种类型的电子设备具有重要作用。编写代码时注意首先是需要正确初始化 GPIO 外设和 PWM 功能。初始化 GPIO 外设时需要获得引脚与实际连接的 LED 灯的对应关系，不能选错，然后设置正确的引脚复用功能和输出方向。此外初始化 PWM 时需要指定所使用的 PWM 端口号。另一个是对呼吸灯效果的理解和控制。呼吸灯是通过不断改变 LED 的亮度实现的，即改变 PWM 的占空比。我们用来一个循环来逐渐调整占空比，从而实现 LED 的呼吸灯效果且达到了平滑的过渡，避免占空比的突变。此外，我们还可以调整频率参数以控制呼吸的速度。

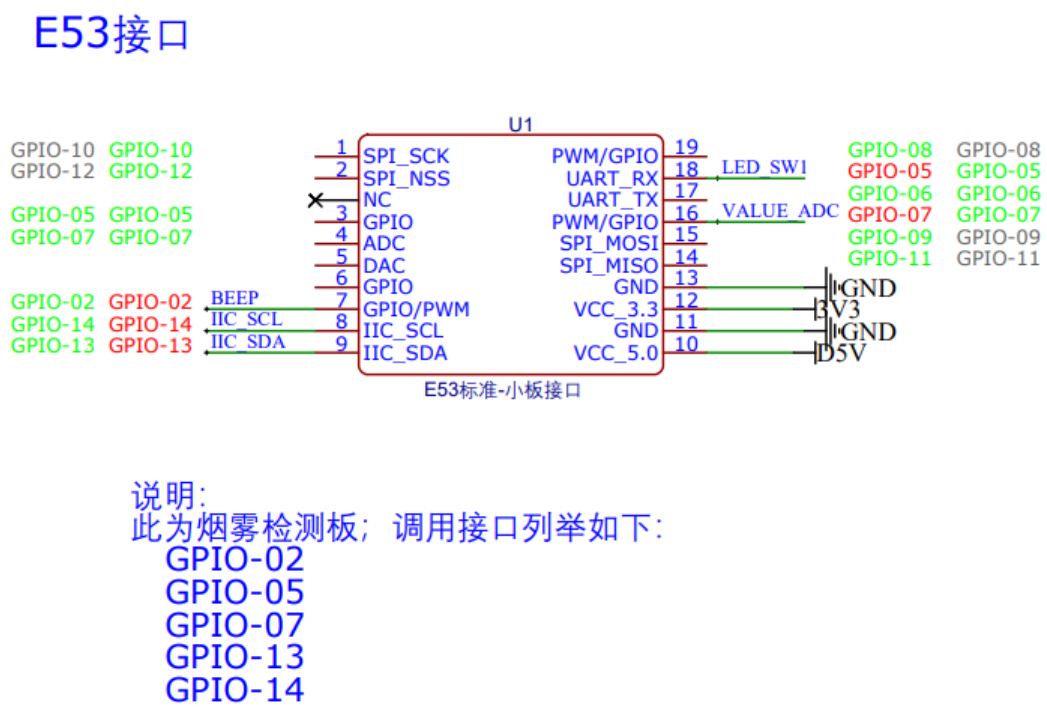
3. 实验三、烟雾探测

3.1 待完成功能

烟雾传感器实验需要控制烟雾传感器实现烟雾检测和报警功能。涉及多个步骤和功能，通过本实验可以搭建一个基本的烟雾检测系统。首先，需要完成蜂鸣器引脚的复用，使其可以通过 PWM 控制。复用蜂鸣器引脚并控制蜂鸣器的报警声音。接下来，初始化 LED 灯的 GPIO 引脚，并设置其功能和方向。LED 灯在烟雾检测中起到指示灯的作用，当检测到烟雾超标时 LED 灯点亮提醒用户。我使用 ADC 采集烟雾传感器引脚的电压值。并将采集到的数据存储在数组中。由于 ADC 采集的数据并不是熟悉的电压值，需要将其转换为可视化的数值。通过一定的计算公式将采集到的数据转换为对应的电压值。这样就可以根据电压值判断烟雾的浓度。检测过程中，当检测到烟雾浓度超过设定的阈值时会触发报警机制。通过控制蜂鸣器引脚的 PWM 输出和 LED 灯的状态实现蜂鸣器的报警声音和 LED 灯的点亮，提醒用户检测到烟雾。

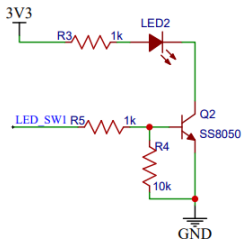
该实验主程序中需要完成引脚初始化、PWM 初始化、GPIO 控制、ADC 采集、数据转换等操作。

3.2 实验原理图

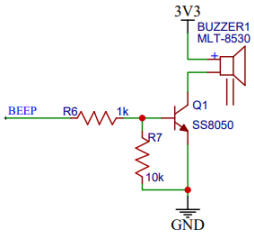


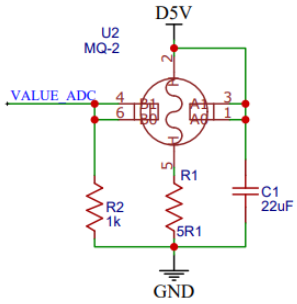
说明:
此为烟雾检测板；调用接口列举如下:
GPIO-02
GPIO-05
GPIO-07
GPIO-13
GPIO-14

LED：用户自定义



蜂鸣器





3.3 实验代码

首先编写电压转换函数：

```
float smoke_convert_to_voltage(hi_u32 data_len)
{
    hi_u32 i;
    float vlt_max = 0;
    hi_u16 vlt;
    for (i = 0; i < data_len; i++) {
        vlt = s_adc_buf[i];
        float voltage=(float)vlt*1.8*4/4096.0;
        vlt_max = (voltage > vlt_max) ? voltage : vlt_max;
 }
    printf("vlt_max:%.3f \n", vlt_max);
    return vlt_max;
}
```

然后编写驱动 ADC 读取电信号的程序 SMOKE_ADC_GATHER

```
float smoke_adc_gather(hi_void)
{
    hi_u32 ret, i;
    hi_u16 data;
    float ans = 0;
    float retans;
    printf("ADC Test Start\n");
    memset_s(s_adc_buf, sizeof(s_adc_buf), 0x0, sizeof(s_adc_buf));
    for (hi_u8 em = 0; em < HI_ADC_EQU_MODEL_BUTT; em++) {
        for (i = 0; i < ADC_TEST_LENGTH; i++) {
            ////采集ADC3电压值
            ret = hi_adc_read(HI_ADC_CHANNEL_3, &data, (hi_adc_equ_model_s
            if (ret != HI_ERR_SUCCESS) {
            printf("ADC Read Fail\n");
                return;
            }
            s_adc_buf[i] = data;
        }
        ans += smoke_convert_to_voltage(ADC_TEST_LENGTH);//将采集的数据转换
    }
    retans = ans / HI_ADC_EQU_MODEL_BUTT;
    printf("ADC Test Average Mode End with %lf\n",retans);
    return retans;
}
```

编写报警时需要的蜂鸣器相关程序：

首先是蜂鸣器的端口，PWM，频率，占空比等定义：

```c
#include <stdio.h>
#include <unistd.h>
#include "ohos_init.h"
#include "cmsis_os2.h"
#include "hi_pwm.h"
#include "hi_errno.h"
#include "hi_io.h"
#include "hi_task.h"
#include "hi_gpio.h"
#define CLK_160M 160000000
#define DUTY_MIN 0
#define DUTY_MAX 100
#define SHORT_MAX 0xFFFF
#define ALARM HI_IO_NAME_GPIO_2
#define PWM_POWER HI_PWM_PORT_PWM2
#define PWM_ALARM_BRIDGE HI_IO_FUNC_GPIO_2_PWM2_OUT
#define DUTY 50
#define PWM_CLK_FREQ 160000000
#define freq 2442
```

然后编写 PWM 初始化函数：

```c
hi_void smoke_pwm_init(hi_void)
{
    int ret = -1;
    ret = hi_pwm_deinit(PWM_POWER);
    if(ret != 0){
        printf("hi_pwm_deinit failed :%#x \r\n",ret);
    }
    ret = hi_pwm_init(PWM_POWER);
    if(ret != 0){
        printf("hi_pwm_init failed :%#x \r\n",ret);
    }
    ret = hi_pwm_set_clock(PWM_CLK_160M);
    if(ret != 0){
        printf("hi_pwm_set_clock failed ret : %#x \r\n",ret);
    }
}
```

编写蜂鸣器初始化函数：

```c
hi_void smoke_alarm_init(hi_void)
{
    IoTGpioInit(ALARM);
    (void)hi_io_set_func(ALARM, PWM_ALARM_BRIDGE);
    IoTGpioSetDir(ALARM, HI_GPIO_DIR_OUT);
    // IoTPwmInit(PWM);
    smoke_pwm_init();
}
```

蜂鸣器启动与终止函数：

```
hi_void smoke_alarm_start(void)
{
    IoTPwmStart(PWM_POWER, DUTY, freq);
}
hi_void smoke_alarm_stop(void)
{
    IoTPwmStop(PWM_POWER);
}
```

LED 指示灯初始化函数：

```
hi_void smoke_gpio_io_init(void)
{
    hi_u32 ret;
    ret = hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR ===== gpio -> hi_io_set_func ret:%d\r\n", ret);
        return;
    }
    printf("----- gpio5 smoke set func success-----\r\n");

    ret = hi_gpio_set_dir(HI_GPIO_IDX_5, HI_GPIO_DIR_OUT);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR ===== gpio -> hi_gpio_set_dir1 ret:%d\r\n", ret);
        return;
    }
    printf("----- gpio set dir success-----\r\n");
}
hi_void smoke_led_init(void)
{
    smoke_gpio_io_init();
}
```

LED 指示灯启动终止控制函数：

```
hi_void smoke_led_ctrl(unsigned int state)
{
    if(state == 1){
        printf("----- smoke alarm led turn on -----\r\n");
        hi_gpio_set_ouput_val(HI_GPIO_IDX_5,HI_GPIO_VALUE1);
    }else{
        printf("----- smoke alarm led turn off -----\r\n");
        hi_gpio_set_ouput_val(HI_GPIO_IDX_5,HI_GPIO_VALUE0);
    }
}
```

主程序任务定义与启动：

```
static unsigned int g_MonitorTask;
static const hi_task_attr MonitorTaskAttr = {
    .task_prio = 20,
    .stack_size = 4096,
    .task_name = "smoke_alarm_task",
};
```

```
hi_void smoke_adc_demo(hi_void)
{
    int ret;

    smoke_led_init();
    smoke_alarm_init();

    ret = hi_task_create(&g_MonitorTask, // task标识 //
        &MonitorTaskAttr,
        MonitorTask_smoke, // task处理函数 //
        NULL); // task处理函数参数 //
}

APP_FEATURE_INIT(smoke_adc_demo);
```

TASK 检测核心流程：

```
void *MonitorTask_smoke(void * para) /* smoke task处理函数
{
    float vlt = 0;
    static int smoke = 0;
    while(1){
        vlt = smoke_adc_gather();
        printf("vlotage is %.3f\r\n", vlt);
        //为了实验安全起见，建议此值不要设置过高
        //可以比正常环境略高即可，即低浓度烟雾也可以触发报警
        if (vlt > 0.35)
        {
            if(smoke == 0)
            {
                printf("start \r\n");
                smoke = 1;
                smoke_led_ctrl(1);
                smoke_alarm_start();
            }
        } else {
            if(smoke == 1)
            {
                printf("stop \r\n");
                smoke = 0;
                smoke_led_ctrl(0);
                smoke_alarm_stop();
            }
        }

        //控制检测频率，目前暂定3秒一次
        sleep(3);
    }
    return NULL;
}
```

3.4 编译与运行结果

经过对阈值合理的调整，我们使模块对空调吹出的空气报警，而对人呼出的气体不报警，实验效果如下：下面是报警情况，同时点亮 LED 并激活蜂鸣器

```
ADC Test Start
vlt_max:0.345
vlt_max:0.345
vlt_max:0.346
vlt_max:0.345
ADC Test Average Mode End with 0.344971
vlotage is 0.345
ADC Test Start
vlt_max:0.373
vlt_max:0.373
vlt_max:0.371
vlt_max:0.369
ADC Test Average Mode End with 0.371338
vlotage is 0.371
start
----- smoke alarm led turn on -----
ADC Test Start
vlt_max:0.380
vlt_max:0.376
vlt_max:0.374
vlt_max:0.373
ADC Test Average Mode End with 0.375732
```

下面是不报警情况

```
ADC Test Start
vlt_max:0.267
vlt_max:0.267
vlt_max:0.267
vlt_max:0.267
ADC Test Average Mode End with 0.267188
vlotage is 0.267
stop
----- smoke alarm led turn off -----
```

3.5 遇到的问题

a. 烟雾传感器的刷新检测频率在最开始时调整的比较低，无法及时的获取最新的环境信息。

b. 处理 ADC 数据时最开始对一组内的 4 个时刻做了平均值处理，导致每个数据组内的数据值都变得一样

c. 达到阈值时小组内数据已有触发警报的数值，但由于取平均值后这个数据被拉低了所以没有触发警报。

3.6 问题解决方案

a. 提高了数据的刷新频率，也就是减小每次 usleep 的时间，这样就可以提高数据的及时性。

b. 直接返回 4 个数值的原始数据，加以简单的防抖操作去除无效数据，即可

返回相对比较准确的数值

c. 我们在上一步中对每组 4 个检测数值取消了平均处理，而是换为了取最值方法，这样即可避免关键数据被削弱的现象

3.7 总结与心得

本次实验我深入了解了烟雾传感器的工作原理和使用方法。烟雾传感器利用气敏材料和 ADC 采集原理，能够检测烟雾浓度并输出相应的信号。实验帮助我理解了烟雾传感器背后的物理原理，并学会了如何使用相关的硬件和软件进行控制和数据处理。在软件开发方面，实验中涉及了蜂鸣器的 PWM 控制、LED 灯的 GPIO 控制和 ADC 采集等功能且要确保正确设置引脚的功能和方向。其次需要仔细研究相应电路图了解各个模块的使用方法。本实验用到了蜂鸣器，在配置蜂鸣器时我们使用了和上个实验类似的 PWM 实现方法，只是需要更换 PWM 绑定通道。

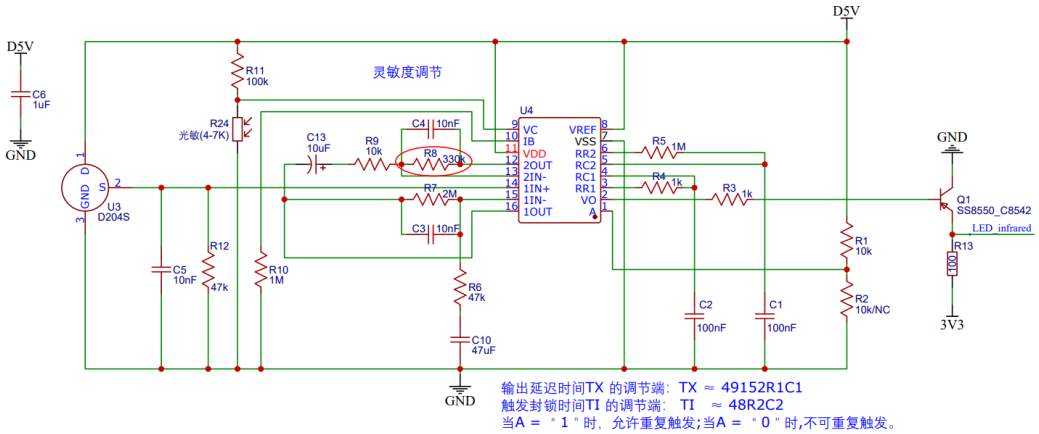
4. 实验四、人体感应

4.1 待完成功能

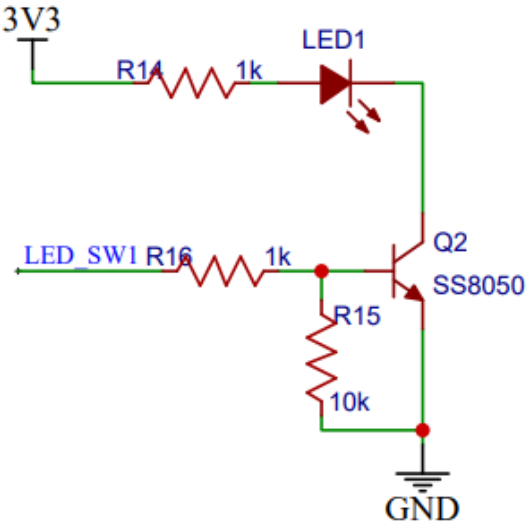
人体体温一般在 37 度，会发出特定波长 10UM 左右的红外线，被动式红外探头靠探测人体发射的 10UM 左右的红外线而进行工作。红外感应源通常采用热释电元件，这种元件在接收到人体红外辐射温度发生变化时就会失去电荷平衡，向外释放电荷，后续电路经检测处理后就能产生报警信号。当产生报警信号之后红外传感器所接的 io 引脚电压会被拉高，我们通过 ADC 采集该引脚电压来判断是否检测到人体红外。

本实验用启航主控板和人体传感器模块开发人体检测样例程序。人体传感器模块通过探测人体发出的特定波长的红外线来工作。使用 ADC 采集人体传感器模块引脚的电压，并根据电压值的变化来判断是否检测到人体红外信号。首先需要设置 LED 灯的控制引脚和方向控制 LED 的开关状态。LED 灯接在 GPIO_2 引脚上，将该引脚设置为输出方向。接下使用 ADC 采集引脚电压，了解采集的引脚对应的通道。将程序的功能分为几个步骤来完成。首先是对 LED 引脚的复用和方向的设置。通过调用相应的函数来初始化 GPIO_2 引脚，将其设置为控制 LED 灯的功能和输出方向。在这之后，我们可以根据需要控制 LED 的开关状态。通过传入参数来控制 LED 的状态，当参数为 1 时，将 GPIO_2 引脚输出高电平以点亮 LED；当参数为 0 时，将 GPIO_2 引脚输出低电平以关闭 LED。接下来需要使用 ADC 采集引脚的电压值。创建一个新的文件来完成 ADC 采集的功能。在采集过程中使用一个数组来存储采集到的数据，并进行相应的处理。此外还需要将采集到的数据转换为可视的电压数值从而进一步判断是否检测到人体红外信号。最后将采集到的数据转换为电压值并与阈值进行比较，判断是否超过阈值来决定是否点亮 LED 灯。

4.2 实验原理图



LED：用户自定义



4.3 实验代码

首先编写 ADC 检测函数：

```
hi_void infrared_adc_gather(hi_void)
{
    hi_u32 ret, i;
    hi_u16 data;
    memset_s(g_adc_buf, sizeof(g_adc_buf), 0x0, sizeof(g_adc_buf));
    for (hi_u8 em = 0; em < HI_ADC_EQU_MODEL_BUTT; em++) {
        for (i = 0; i < ADC_TEST_LENGTH; i++) {
            ret = hi_adc_read(HI_ADC_CHANNEL_3, &data, (hi_adc_equ_model_sel)em,
            if (ret != HI_ERR_SUCCESS) {
                printf("ADC Read Fail\n");
                return;
            }
            g_adc_buf[i] = data;
        }
        convert_to_voltage(ADC_TEST_LENGTH);
    }
}
```

然后编写检测函数，在本函数中，我们不停接受 ADC 的信号值，通过与前一时刻的信号对比和是否达到阈值，综合两个条件来返回判断信息并开关 LED 灯：

```
hi_void convert_to_voltage(hi_u32 data_len)
{
    hi_u32 i;
    float vlt_max = 0;
    float vlt_min = VLT_MIN;
    hi_u16 vlt;
    for (i = 0; i < data_len; i++) {
        vlt = g_adc_buf[i];
        float voltage = (float)vlt * 1.8 * 4 / 4096.0;  /* vlt * 1.8 * 4 / 4096.0: Convert code into voltag
        vlt_max = (voltage > vlt_max) ? voltage : vlt_max;
    }
    if(vlt_max > 3.0 &&before_vlt <=3.0){
        printf("---ooooooooooooooooooooooooooo- infrared test led turn on --ooooooooooooooooooo--\r\n");
        printf("previous vlt %lf, now %lf \r\n", before_vlt, vlt_max);
        infrared_led_ctrl(1);
    }else if(vlt_max <= 3.0 &&before_vlt >3.0){
        printf("-xxxxxxxxxxxxxxxxxxxxxxxxxxx---- infrared test led turn off --xxxxxxxxxxxxxxxxxxx--\r\n");
        printf("previous vlt %lf, now %lf \r\n", before_vlt, vlt_max);
        infrared_led_ctrl(0);
    }
    before_vlt = vlt_max;
}
```

设备初始化函数：

```
hi_void infrared_gpio_io_init(void)
{
    hi_u32 ret;
    /*gpio2 人体感应模块led*/
    ret = hi_io_set_func(HI_IO_NAME_GPIO_2, HI_IO_FUNC_GPIO_2_GPIO);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR ===== gpio -> hi_io_set_func ret:%d\r\n", ret);
        return;
    }
    printf("----- gpio2 set body induction led func success-----\r\n");
    //控制led输出，所以将io2方向设置为输出
    ret = hi_gpio_set_dir(HI_GPIO_IDX_2, HI_GPIO_DIR_OUT);
    if (ret != HI_ERR_SUCCESS) {
        printf("===== ERROR ===== gpio -> hi_gpio_set_dir1 ret:%d\r\n", ret);
        return;
    }
    printf("----- gpio set dir success! -----\r\n");
}
```

主函数调用入口：

```
hi_void infrared_adc_demo(hi_void)
{
    //处理任务前先对gpio初始化
    infrared_gpio_io_init();
    //循环采集红外传感器引脚电压
    while(1){
        infrared_adc_gather();
    }
}
APP_FEATURE_INIT(infrared_adc_demo);
```

4.4 编译与运行结果

烧录固件，按下按钮重启，在传感器探头前晃动手指，或者使用手机红外遥控功能向探头发射信号，可见串口显示了检测信息，开关情况，与变化瞬时前后的电压差异。同时观察到 LED 随着手指或手机的遥控而点亮，离开后熄灭。

```
ready to OS start
sdk ver:Hi3861V100R001C00SPC025 2020-09-03 18:10:00
FileSystem mount ok.
wifi init success!
hilog will init.

hievent will init.

hievent init success.

hiview init success.
----- gpio2 set body induction led func success-----
----- gpio set dir success! -----
No crash dump found!
---ooooooooooooooooooooooooooo- infrared test led turn on --ooooooooooooooooooo
--
previous vlt 0.798047, now 3.077930
-xxxxxxxxxxxxxxxxxxxxxxxxx---- infrared test led turn off --xxxxxxxxxxxxxxxxxx
x--
previous vlt 3.076172, now 0.808594
```

4.5 遇到的问题

a. 引脚配置：在初始化 GPIO 引脚时，最开始错误选择了其他的引脚或配置。导致 ADC 采集不到正确的引脚信号。或者是采样频率设置错误或 ADC 模块初始化问题等导致无法正确采集到红外传感器引脚的电压值。

b. 数据转换：数据转换为电压值时，出现了计算错误或类型转换错误。导致无法准确断是否检测到人体红外信号。

4.6 问题解决方案

a. 查阅文档说明书从而获得正确的 ADC 引脚，再结合之前做过的实验的代码内容共同配置正确的 ADC 检测环境

b. 查阅相关资料，重新计算正确的电压与 ADC 信号之间的转化关系，同时划分清识别与否的 ADC 数值。

4.7 总结与心得

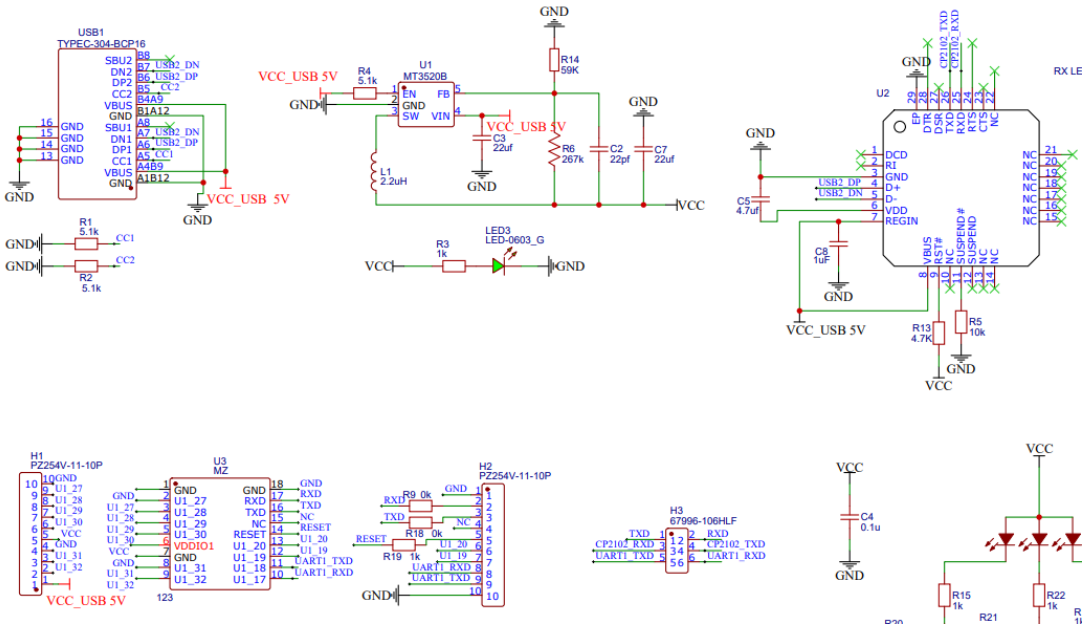
本实验中我们完成了使用启航 KP_IOT 主控板和人体传感器模块开发人体检测样例程序的任务。学习了如何初始化 GPIO 引脚，控制 LED 的开关状态，以及使用 ADC 采集引脚的电压值。通过对采集数据的转换和判断，我们能够检测到人体红外信号并相应地控制 LED 的亮灭。在本次实验中我意识到合理的代码结构和模块化设计非常重要。通过将功能分解为较小的模块可以更容易地理解和调试每个部分。模块化设计也使代码更易于维护和重用。其次在编写代码之前，我们应该详细阅读 README 文档和硬件电路图，仔细了解内部原理。通过这个实验，我深入理解了 GPIO 控制和 ADC 采集的基本原理，并学会了如何将它们应用于实际的物联网开发中。学到了处理传感器数据和控制外部设备的一些基本技巧。

5. 实验五、WIFI 接入

5.1 待完成功能

在开发板上编写一个连接 Wifi 热点的业务程序，实现开发板联网效果

5.2 实验原理图



5.3 实验代码

代码的主要流程如下：

注册 WiFi 事件回调函数。

启用 STA 模式。

添加热点配置信息。

连接到指定的热点。

循环等待 WiFi 连接成功的标志位。

获取 netif 用于 IP 操作。

启动 DHCP，获取 IP 地址。

打印 DHCP 信息。

如果无法获取 IP 地址，则断开 WiFi 连接并重新启动。

取消 WiFi 事件回调函数的注册。

具体代码：

a. 导入头文件，引入 IP 网络接口

```
#include <stdio.h>  // for printf

#include <unistd.h>  // for sleep

#include "ohos_init.h"  // for SYS_RUN
#include "cmsis_os2.h"

#include "hi_wifi_api.h"

#include "lwip/ip_addr.h" // SDK提供的wifi功能接口
#include "lwip/netifapi.h"  //协议栈IP地址操作接口


static struct netif *g_lwip_netif = NULL;  //IP层网络接口，在此接口上部署IP地址
```

b. 启动模块并注册 WIFI 事件

```
int ret;
char ifname[WIFI_IFNAME_MAX_SIZE + 1] = {0};
int len = sizeof(ifname);
unsigned int  num = WIFI_SCAN_AP_LIMIT;

//启动wifi模组，会生成无线网络接口，例如 "wlan 0"
ret = hi_wifi_sta_start(ifname, &len);
if (ret != HISI_OK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return;
}

/* register call back function to receive wifi event, etc scan results event,
 * connected event, disconnected event.
 */
// 注册wifi事件通告处理，用于接收wifi事件通告
ret = hi_wifi_register_event_callback(wifi_wpa_event_cb);
if (ret != HISI_OK) {
    printf("register wifi event callback failed\n");
}
```

c. 获取网络描述符，启用热点扫描

```
g_lwip_netif = netifapi_netif_find(ifname);
if (g_lwip_netif == NULL) {
    printf("%s: get netif failed\n", __FUNCTION__);
    return ;
}

/* start scan, scan results event will be received soon */
//启动wifi热点扫描
ret = hi_wifi_sta_scan();
if (ret != HISI_OK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return ;
}
```

d. 存放扫描结果，持续遍历

```
sleep(5);   /* sleep 5s, waiting for scan result. */

//用来存储扫描结果的内存块
hi_wifi_ap_info *pst_results = malloc(sizeof(hi_wifi_ap_info) * WIFI_SCAN_AP_LIMIT);
if (pst_results == NULL) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return ;
}

//存放扫描结果
ret = hi_wifi_sta_scan_results(pst_results, &num);
if (ret != HISI_OK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    free(pst_results);
    return ;
}

//遍历wifi热点扫描结果，并输出到串口显示
for (unsigned int loop = 0; (loop < num) && (loop < WIFI_SCAN_AP_LIMIT); loop++) {
    printf("SSID: %s\n", pst_results[loop].ssid);
}
```

e. 执行连接操作

```
ret = hi_wifi_start_connect();
if (ret != 0) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return ;
}


return;
```

f. 连接函数详情

```
assoc_req.auth = HI_WIFI_SECURITY_WPA2PSK;

/* 热点密码,请修改成上述热点名称对应的密码 */
memcpy(assoc_req.key, "20020617", 8);


//调用SDK进行wifi连接过程
ret = hi_wifi_sta_connect(&assoc_req);
if (ret != HISI_OK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return -1;
}
printf("%s %d \r\n", __FILE__, __LINE__);
return 0;
```

g. 回调函数详情

```
void wifi_wpa_event_cb(const hi_wifi_event *hisi_event)
{
    if (hisi_event == NULL)
        return;

    switch (hisi_event->event) {
        //热点扫描结束
        case HI_WIFI_EVT_SCAN_DONE:
            printf("WiFi: Scan results available\n");
            break;
        //wifi已连接，开始dhcp动态获取IP地址
        case HI_WIFI_EVT_CONNECTED:
```

```
            printf("WiFi: Connected\n");
            netifapi_dhcp_start(g_lwip_netif);
            break;
        //wifi断开连接，停止dhcp过程，并清空已获取到的IP地址等信息
        case HI_WIFI_EVT_DISCONNECTED:
            printf("WiFi: Disconnected\n");
            netifapi_dhcp_stop(g_lwip_netif);
            hi_sta_reset_addr(g_lwip_netif);
            break;
        //wifi连接超时
        case HI_WIFI_EVT_WPS_TIMEOUT:
            printf("WiFi: wps is timeout\n");
```

h. 清空数据函数：

```
void hi_sta_reset_addr(struct netif *pst_lwip_netif)
{
    ip4_addr_t st_gw;
    ip4_addr_t st_ipaddr;
    ip4_addr_t st_netmask;
    printf("%s %d \r\n", __FILE__, __LINE__);
    if (pst_lwip_netif == NULL) {
        printf("hisi_reset_addr::Null param of netdev\r\n");
        return;
    }

    IP4_ADDR(&st_gw, 0, 0, 0, 0);
    IP4_ADDR(&st_ipaddr, 0, 0, 0, 0);
    IP4_ADDR(&st_netmask, 0, 0, 0, 0);

    netifapi_netif_set_addr(pst_lwip_netif, &st_ipaddr, &st_netmask, &st_gw);
}
```

5.4 编译与运行结果

编译顺利通过：

```
问题    输出    调试控制台    终端

[OHOS INFO] startup                  5       1.2%          5       1.2%    1.00
[OHOS INFO] third_party              2       0.5%          2       0.5%    1.00
[OHOS INFO] updater                  4       0.9%          4       0.9%    1.00
[OHOS INFO] utils                    4       0.9%          4       0.9%    1.00
[OHOS INFO]
[OHOS INFO] c overall build overlap rate: 1.00
[OHOS INFO]
[OHOS INFO]
[OHOS INFO] qihang build success
[OHOS INFO] cost time: 0:00:05
root@54911a7ea290:/home/openharmony/OpenHarmony#
```

运行结果：预定的 WIFI 名称是 woyaofangjia，可见连接成功

```
<--System Init-->
<--Wifi Init-->
register wifi event succeed!
callback function for wifi scan:0, 0
+NOTICE:SCANFINISH
callback function for wifi scan:1, 10
WaitSacnResult:wait success[1]s
********************
no:001, ssid:woyaofangjia                  , rssi:  -28
no:002, ssid:Hot topics for handsome guys, rssi:  -68
no:003, ssid:                              , rssi:  -70
no:004, ssid:123                           , rssi:  -71
no:005, ssid:GiWiFi-SDUST                  , rssi:  -64
no:006, ssid:GiWiFi-SDUST                  , rssi:  -64
no:007, ssid:GiWiFi-SDUST                  , rssi:  -69
no:008, ssid:GiWiFi-SDUST                  , rssi:  -79
no:009, ssid:GiWiFi-SDUST                  , rssi:  -83
no:010, ssid:GiWiFi-SDUST                  , rssi:  -88
********************
```

DHCP 服务也成功启动

```
Select:  1 wireless, Waiting...
+NOTICE:CONNECTED
WaitConnectResult:wait success[1]s
WiFi connect succeed!
begain to dhcp
<-- DHCP state:Inprogress -->
<-- DHCP state:OK -->
server :
        server_id : 192.168.128.229
        mask : 255.255.255.0, 1
        gw : 192.168.128.229
        T0 : 3599
        T1 : 1799
        T2 : 3149
clients <1> :
        mac_idx mac             addr            state   lease   tries   rto
        0       f0b0408acfb2    192.168.128.235 10      0       1       4
```

5.5 遇到的问题

在设定 WIFI 信息时，没有为 SSID 开辟足够大的内存空间，如下图所示：

```
rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, "woyaofangjia", 12); /* 9:ssid length */
if (rc != EOK) {
    printf("%s %d \r\n", __FILE__, __LINE__);
    return -1;
}
```

SSID 的长度是 12 位，所以后面的数字也应该填写 12，但最开始我填入了 9，所以导致了内存溢出，WIFI 连接成功但不能自动获取 IP 地址，也就是 DHCP 服务不能成功启动

5.6 问题解决方案

将 SSID 的长度容量设为字符串串长即可正常启用 DHCP 服务，并获取 IP。

5.7 总结与心得

本次实验我们学习了如何在开发板上编写连接 WiFi 的程序，实现开发板的联网功能。通过调用一系列的 WiFi API 和对程序的设计，我们能够在开发板上配置和连接到指定的 WiFi 热点，并获取 IP 地址以进行网络通信。实验过程中，我了解了 WiFi API 及其相关函数。README 中还提供了一系列 API 函数的说明，包括注册 WiFi 事件、启用和禁用 STA 模式、添加热点配置信息、连接和断开热点等。其次，要熟悉 WIFI 的连接流程，包括注册 WiFi 事件、启用 STA 模式、添加热点配置、连接热点、获取 IP 地址等。实验过程中，我们遇到了各种问题，如无法连接到热点、获取不到 IP 地址等。经过对终端信息的打印和代码排查，最终确定了是内存溢出错误。此外在 WIFI 连接过程中也应合理使用延时函数和循环等待来等待 WiFi 连接成功的标志位和 IP 地址的获取。网络通信是嵌入式开发必不可少的一部分，通过本次实验我对 WiFi 联网的流程和 API 有了更深入的了解。

6. 额外实验、蜂鸣器播放音乐

6.1 实验总体概览

本次额外实验是老师提出的建议，在课后我完成了蜂鸣器播放音乐实验
原理是利用 PWM 可调频与调整占空比的特性，可以将蜂鸣器作为扬声器，并播放音乐。
本实验播放的一首节选的歌曲：

6.2 实验步骤

首先通过 MP3 转 wav，再编写解码程序，获得 MIDI 代码，WAV 解码程序如下：

```
typedef struct WAV_data {
    char Subchunk2ID[4];
    uint32_t Subchunk2Size;
    Data_block_t block[900000];
} Data_t;

typedef struct WAV_fotmat {
   RIFF_t riff;
   FMT_t fmt;
   Data_t data;
} Wav;

int main()
{
    FILE *fp = NULL;
    Wav wav;
    RIFF_t riff;
    FMT_t fmt;
    Data_t data;
    fp = fopen("test.wav", "rb");
    if (!fp) {
        printf("can't open audio file\n");
        exit(1);
    }
    fread(&wav, 1, sizeof(wav), fp);
    riff = wav.riff;
    fmt = wav.fmt;
```

运行该程序，将 WAV 文件解码为 TXT 的 INT 型数值格式，然后将其作为数组保存起来。
DEFINE 各个音节作为简化的 MIDI 声调，

```
#define DOL 1908
#define REL 1701
#define MIL 1515
#define FAL 1449
#define SOL 1275
#define LAL 1136
#define SIL 1012
//中音
#define DOM 956
#define REM 852
#define MIM 759
#define FAM 716
#define SOM 638
#define LAM 568
#define SIM 506
//高音
#define DOH 478
#define REH 426
#define MIH 379
#define FAH 358
#define SOH 319
#define LAH 284
#define SIH 254
//停止
#define STOP 65535
```

PWM 初始化：

```
hi_void kunkun_pwm_init(hi_void)
{
    int ret = -1;
    ret = hi_pwm_deinit(PWM_POWER);
    if(ret != 0){
        printf("hi_pwm_deinit failed :%#x \r\n",ret);
    }
    ret = hi_pwm_init(PWM_POWER);
    if(ret != 0){
        printf("hi_pwm_init failed :%#x \r\n",ret);
    }
    ret = hi_pwm_set_clock(PWM_CLK_160M);
    if(ret != 0){
        printf("hi_pwm_set_clock failed ret : %#x \r\n",ret);
    }
}
```

GPIO 初始化与任务创建：

```
hi_void kunkun_alarm_init(hi_void)
{
    IoTGpioInit(ALARM);
    (void)hi_io_set_func(ALARM, PWM_ALARM_BRIDGE);
    IoTGpioSetDir(ALARM, HI_GPIO_DIR_OUT);
    kunkun_pwm_init();
}

static unsigned int g_MonitorTask;
static const hi_task_attr MonitorTaskAttr = {
    .task_prio = 20,
    .stack_size = 4096,
    .task_name = "CAIXUKUN",
};
```

在任务中使用 for 循环遍历简化的 MIDI 乐谱，使用蜂鸣器发出声音

```
void *MonitorTask_kunkun(void * para) /* kunkun task处理函数 */
{
    printf("stg1\n");
    printf("stg2\n");
    int music[]=
    {
        LAM,1,LAM,1,LAM,1,LAM,DOH,REH,MIH,
        LAM,1,LAM,1,LAM,SOM,SOM,LAM,
        LAM,1,LAM,1,LAM,1,LAM,DOH,REH,MIH,
        LAM,1,LAM,1,LAM,FAH,FAH,MIH,
        LAM,1,LAM,1,LAM,1,LAM,DOH,REH,MIH,
        LAM,1,LAM,1,LAM,1,LAM,SOM,LAM,
        LAM,1,LAM,1,LAM,1,LAM,DOH,REH,MIH,
        LAM,SOM,SOM,LAM,1,LAM,
        LAM,1,LAM,SOM,SOM,SOM,LAM,
        SOM,MIM,MIM,SOM,MIM,
        LAM,1,LAM,SOM,LAM,SIM,
        DOH,SIM,LAM,SIM,LAM,SOM,0,0
    };
    for(int i=0;music[i]>0;i=i+1){
        music[i] = (int)(2000000/music[i]);
    }
    for(int i=0;music[i]>0;i=i+1){
        IoTPwmStart(PWM_POWER,40,music[i]);
        usleep(200*1000);
        IoTPwmStop(PWM_POWER);
    }
    printf("stg3\n");
    return NULL;
}
```

主程序及入口：

```
hi_void kunkun_adc_demo(hi_void)
{
    kunkun_alarm_init();
    (void)hi_task_create(&g_MonitorTask, // task标识 //
        &MonitorTaskAttr,
        MonitorTask_kunkun, // task处理函数 //
        NULL); // task处理函数参数 //
}

APP_FEATURE_INIT(kunkun_adc_demo);
```

6.3 实验结果

在初步测试成功后，我延长了歌曲乐谱，并找到了冷老师展示播放成果。本实验的全部代码被新建为工程 26_CAIXUKUN 文件夹。

6.4 心得体会

在本次附加实验中，我进一步了解了 PWM 的原理，用途，以及调整配置方法，本实验可参照前面实验的蜂鸣器模块进行发声，同时参考 PWM_LED 呼吸灯实验做可控 PWM 调频。除了这方面以外，我认为如何用 PWM 把乐谱转换为信号量也是很重要的一点。歌曲应包括音调，音量和保持时间，其中音调是 PWM 中的频率，音量是占空比，而保持时间是每个音调播放完后的 SLEEP 休眠时间。除了这两个方面，我认为将音乐音频文件转化为 MIDI 格式的这种频谱也是很有难度。本实验我专门在环境之外编写了一个 WAV 解析程序，从而将其转化为可读的文本格式。做到这三点之后，使用蜂鸣器播放简单的 MIDI 音乐我认为是很容易的了