实 验 报 告

实验课程：《电子工艺实训》

实验项目：鸿蒙小车

系　　别：计算机科学与技术系

年　　级：2023 级

小组编号：48

组　　长：马成龙/25120232201399

成　　员：马成龙/25120232201399

实验日期：2025 年 6 月 23 日-- 7 月 11 日

目录

一、 OpenHarmony 理论 3

1.1 简述 OpenHarmony 系统架构及与其它主流操作系统的对比。 3

1.1.1 OpenHarmony 系统架构 3

1.1.2 与其它主流操作系统的对比 3

1.2 简述 OpenHarmony 与 Harmony OS 的异同点。 4

1.2.1 相同点 4

1.2.2 不同点 4

1.3 简述华为云与其它主流云的异同点。 5

1.3.1 相同点 5

1.3.2 不同点 5

二、 OpenHarmony 实验 5

2.1 GPIO 实验 6

a) 实验 2-1 点亮 LED 6

b) 实验 2-2 按键控制 LED 7

2.2 ADC 实验 10

a) 实验 3 使用按键控制 LED 亮灭 10

2.3 OLED 实验 15

a) 实验 4 OLED 屏幕显示学号姓名 15

2.4 PWM 和 WiFi 实验 18

a) 实验 5-1 PWM 实现 LED 呼吸灯 18

b) 实验 5-2 开发板连接 WiFi 热点并在 OLED 显示 21

2.5 MQTT 实验 25

a) 实验 6 开发板和电脑通过 MQTT 代理互发消息 25

三、 智能小车基础部分 27

3.1 小车的测试 27

3.2 系统主线程 28

3.3 小车的循迹逻辑 30

3.4 细节功能 40

3.5 实验心得 42

四、 智能小车创新部分 42

4.1 远程控制 43

4.2 单独避障模式 45

4.3 心得体会 46

五、 课程改进建议 48

5.1 课程内容方面 48

5.2 项目与创新拓展方面 48

5.3 常见问题与调试指南 49

一、 OpenHarmony 理论

1.1 简述 OpenHarmony 系统架构及与其它主流操作系统的对比。

1.1.1 OpenHarmony 系统架构

OpenHarmony 采用面向全场景、全连接、多设备的分布式理念设计，其架构具有高度灵活性和可伸缩性。总体上是分层、可裁剪的架构，从下至上分为四层：

a) 内核层： 这是系统的基础，采用多内核设计。它提供一个内核抽象层（KAL, Kernel Abstraction Layer）来屏蔽底层内核实现差异。根据设备资源大小，可搭载不同的 OS 内核：

- LiteOS： 华为自研的轻量级实时操作系统，适用于轻量与小型设备（如智能手环、传感器等）。
- Linux 内核： 适用于具有较丰富资源的标准系统设备（如智能手机、平板电脑等）。

b) 系统服务层： 这是 OpenHarmony 的核心能力层，包含了分布式架构的关键模块和基础的系统服务。它由一系列子系统集构成，例如：

- 分布式技术： 包括分布式软总线、分布式数据管理、分布式任务调度等，是实现设备间无缝协同的基石。
- HDF 驱动框架： （Hardware Driver Foundation）提供统一的硬件驱动接入和管理框架。
- 基础服务： 提供如公共通信、多媒体、安全等基础能力。

c) 框架层： 为应用程序提供开发所需的各种 API 和框架，支持多种应用形态，如元服务（原子化服务）、传统应用等。

d) 应用层： 包含使用框架层 API 开发的各类系统应用和第三方应用。

1.1.2 与其它主流操作系统的对比

对比维度	OpenHarmony	Android (Google)	iOS (Apple)
核心设计哲学	分布式架构，为多设备协同而生，一次开发、多端部署。	单设备优化，围绕智能手机等单一设备构建的生态系统。	垂直整合的封闭生态，为苹果自身硬件打造的极致单设备体验。

对比维度	OpenHarmony	Android (Google)	iOS (Apple)
内核架构	多内核、可伸缩 (LiteOS 微内核, Linux 宏内核)。	宏内核 (高度定制的 Linux Kernel)。	混合内核 (XNU)，闭源。
分布式能力	系统级、原生支持，通过分布式软总线等技术实现设备间资源共享与任务协同。	应用层实现，原生缺乏系统级分布式能力，需依赖第三方应用或协议。	生态内协同，通过 Handoff 等功能实现苹果设备间的协同，但仅限苹果生态。
驱动框架	HDF 驱动框架，目标是平台与内核解耦，实现“一次开发，多系统部署”。	HAL (硬件抽象层)，作为 Android 框架与底层驱动的桥梁，与 Linux 内核强耦合。	私有驱动框架，完全闭源，为特定硬件服务。
开源策略	完全开源，由开放原子开源基金会（OAT）孵化和运营。	部分开源 (AOSP)，核心服务 (GMS) 闭源，由 Google 主导。	完全闭源，由 Apple 完全掌控。

1.2 简述 OpenHarmony 与 Harmony OS 的异同点。

1.2.1 相同点

- 技术同源： HarmonyOS（鸿蒙操作系统）是以 OpenHarmony 为技术基础和底座进行开发的。两者共享核心的分布式架构、多内核等底层技术。

1.2.2 不同点

对比维度	OpenHarmony	HarmonyOS
定义与定位	“毛坯房”，是一个由开放原子开源基金会运营和治理的开源操作系统项目，是众多下游发行版的基础。	“精装修”，是华为基于 OpenHarmony 推出的面向消费者的商业发行版。
治理模式	开放治理，由开放原子开源基金会（OAT）主导，多家企业共同参与贡献。	商业主导，由华为公司主导其发展方向、版本发布和商业策略。
包含组件	纯粹的开源代码，不包含任何商业闭源组件。	包含商业闭源组件，在 OpenHarmony 基础上，增加了华为移动服务（HMS Core）、华为方舟编译器、专有的 UI 控件和 AI 增强能力等。
设备支持	提供基础能力，理论上可被任何厂商用于开发各种智能设备。	主要用于华为自家的终端设备，如手机、平板、手表、智慧屏等。
演进方向	HarmonyOS NEXT（纯鸿蒙）进一步凸显了差异，该版本完全移除了对 Android 应用的兼容层，构建了完全自主的内核和应用生态，与 OpenHarmony 的关系更趋向于“上游与下游”的关系。	

1.3 简述华为云与其它主流云的异同点。

1.3.1 相同点

- 核心服务趋同： 作为主流云服务商，华为云与其他主流云（如 AWS, Azure, Google Cloud, 阿里云, 腾讯云）在核心产品线上具有很高的相似性。它们都提供全面的 IaaS（计算、存储、网络）、PaaS（数据库、中间件、大数据平台）和 SaaS（企业应用）服务。
- 技术架构相似： 均采用虚拟化、分布式、软件定义等主流云计算技术，提供弹性伸缩、按需付费的服务模式。
- 重视 AI 与大数据： 所有主流云厂商都将 AI/ML、大数据和物联网（IoT）作为战略发展重点，提供丰富的平台工具和解决方案。

1.3.2 不同点

对比维度	华为云	其它主流云 (AWS, Azure, Google, 阿里云等)
独特优势	“云-管-端”软硬件协同：拥有从底层芯片（鲲鹏、昇腾）、服务器硬件，到操作系统（OpenHarmony/EulerOS），再到上层云平台的全栈技术能力。	各有侧重：AWS 以全面的服务和市场领导地位著称；Azure 凭借与 Windows 生态和企业软件的深度集成占据优势；Google Cloud 在云原生、大数据和 AI 领域技术领先；阿里云在国内电商和金融场景根基深厚。
核心竞争力来源	深厚的 ICT 基础设施和电信背景：依托其全球领先的 5G、光网络等电信设备能力，在边缘计算（MEC）、云网融合、物联网等领域有天然优势。	主要源于其在软件、互联网服务或电商领域的长期积累和创新。
主要市场与客户	在政企市场拥有极强的客户基础和行业理解，尤其在政府、金融、运营商、能源、交通等领域。	市场更为多样化。例如，AWS 和 Google Cloud 在互联网和初创企业中广受欢迎，Azure 在大型企业市场表现强劲，阿里云在中国互联网和电商领域占主导地位。
市场表现	中国市场领导者之一，全球市场追赶者。根据 Canalys 2024 年第四季度数据，华为云在中国大陆市场份额位居第二，在全球市场则处于追赶位置。	全球市场三巨头 (AWS, Azure, Google Cloud) 和 中国市场领导者 (阿里云, 腾讯云)。

二、 OpenHarmony 实验

2.1 GPIO 实验

a) 实验 2-1 点亮 LED

实验代码：

原 led_demo.c 略，这里仅对代码修改部分进行展示：

将 IoTGpioSetDir()改为 IoTGpioSetOutputVal()。前者是用来设置 GPIO 引脚的工作方向，当方向为 1（输出）/0（输入）时，确实可以达到控制 LED 灯亮/暗的效果，但并不是直接设置引脚的输出值，这种使用方式在其他地方可能会出现问题。而后者是在将引脚设置为输出模式后，对输出值进行设置，更符合这里的使用情况。完整代码片段如下：

```
while (1)
{
    //输出低电平
    IoTGpioSetOutputVal(LED_TEST_GPIO, 0);
    usleep(300000);
    //输出高电平
    IoTGpioSetOutputVal(LED_TEST_GPIO, 1);
    usleep(300000);
}
```

实验现象：

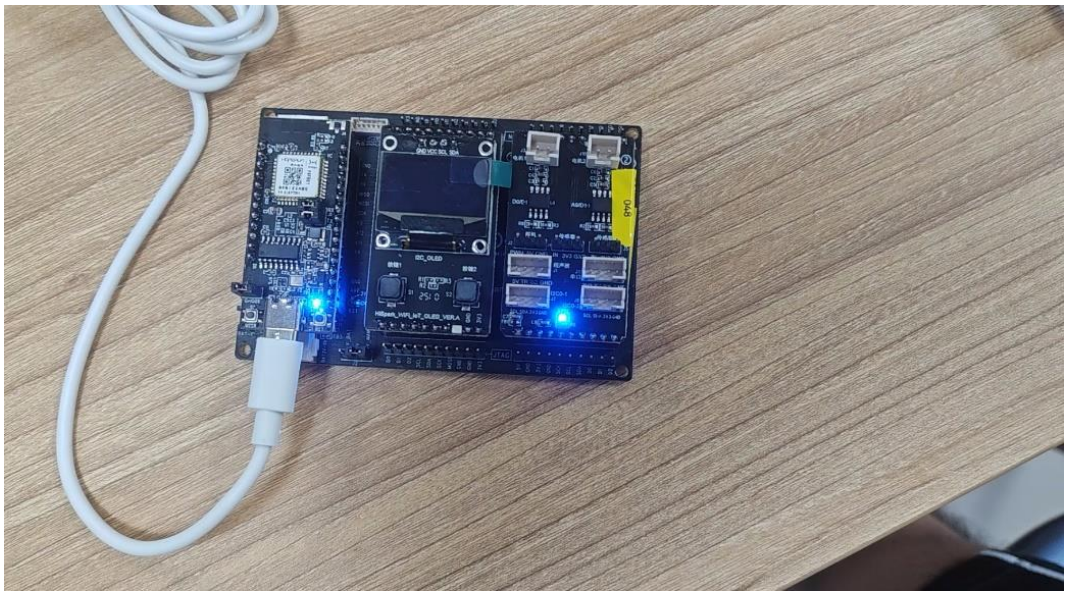

图1：实验 2-1 点亮 LED

具体实验视频请见打包目录下“实验记录”文件夹中的视频“实验 2-1 点亮 LED”。

b) 实验 2-2 按键控制 LED

实验代码：

led_set_demo.c：

```c
#include <unistd.h>
#include "stdio.h"
#include "ohos_init.h"
#include "cmsis_os2.h"
#include "iot_gpio.h"
#include "hi_io.h"

// LED 接 GPIO9，按键 S2 接 GPIO5
#define LED_GPIO 9
#define KEY_GPIO 5

static void LedTask(void *arg)
{
    (void)arg;

    // ---- 初始化 LED 引脚 ----
    IoTGpioInit(LED_GPIO);
    // 设置复用为 GPIO9 模式
    hi_io_set_func(HI_IO_NAME_GPIO_9, HI_IO_FUNC_GPIO_9_GPIO);
    IoTGpioSetDir(LED_GPIO, IOT_GPIO_DIR_OUT);

    // ---- 初始化 按键 引脚 ----
    IoTGpioInit(KEY_GPIO);
    // 设置复用为 GPIO5 模式
    hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO);
    // 打开内部上拉，保证按下前为高电平
```

```
    hi_io_set_pull(HI_IO_NAME_GPIO_5, HI_IO_PULL_UP);
    // 启用输入，CPU 才能读取到电平变化
    hi_io_set_input_enable(HI_IO_NAME_GPIO_5, HI_TRUE);
    // 使用斯密特触发以做简单去抖
    hi_io_set_schmitt(HI_IO_NAME_GPIO_5, HI_TRUE);
    IoTGpioSetDir(KEY_GPIO, IOT_GPIO_DIR_IN);

    while (1)
    {
        IotGpioValue keyVal;
        // 读取按键状态
        IoTGpioGetInputVal(KEY_GPIO, &keyVal);

        if (keyVal == IOT_GPIO_VALUE0) {
            // 按下时，点亮 LED，打印 1
            IoTGpioSetOutputVal(LED_GPIO, IOT_GPIO_VALUE0);
            printf("HI_GPIO_VALUE_1\n");
        } else {
            // 松开时，熄灭 LED，打印 0
            IoTGpioSetOutputVal(LED_GPIO, IOT_GPIO_VALUE1);
            printf("HI_GPIO_VALUE_0\n");
        }
        // 延迟 50ms，防抖 + 减小打印频率
        osDelay(50);
    }
}

void led_set_demo(void)
{
    osThreadAttr_t attr;
    attr.name = "LedTask";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
```

```
61      attr.stack_mem = NULL;
62      attr.stack_size = 1024;
63      attr.priority = 26;
64      if (osThreadNew((osThreadFunc_t)LedTask, NULL, &attr) == NULL) {
65          printf("[LedExample] Falied to create LedTask!\n");
66      }
67  }
68
69  SYS_RUN(led_set_demo);
```

需注意打开 GPIO5（按键 S2）的内部上拉，才能使按键松开时为高电平，做到“松开时灯灭，按下时灯亮”。

实验现象：

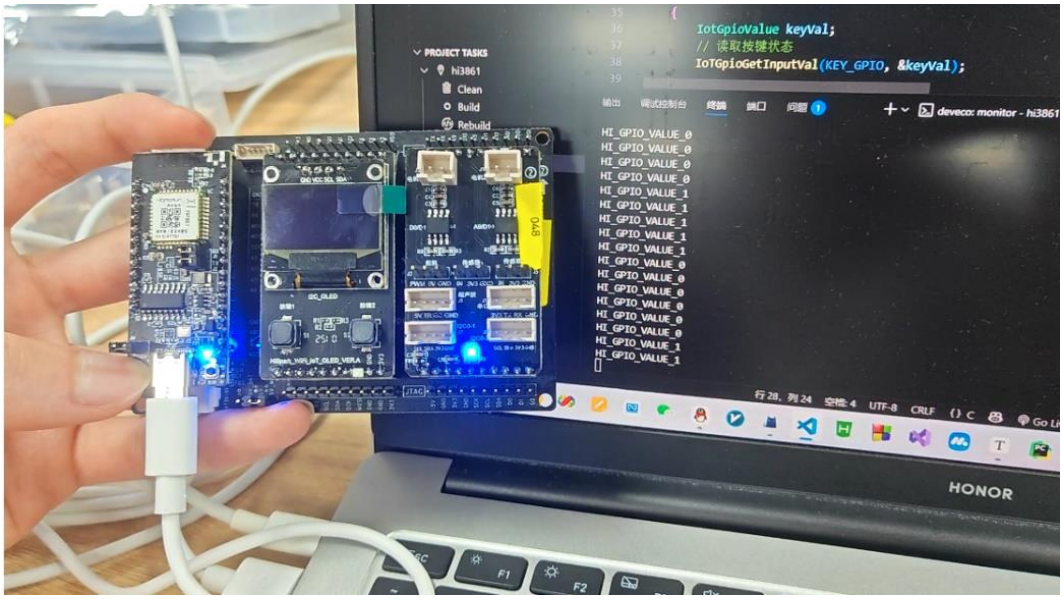

图2：实验 2-2 按键控制 LED

具体实验视频请见“实验记录”文件夹。

实验分析：

在实验 2-1 中，程序烧录到开发板后，LedTask 线程会持续运行。该线程以 0.6 秒（0.3 秒亮 + 0.3 秒灭）为周期，不断改变 GPIO9 的电平，从而实现 LED 灯的闪烁效果 。

在实验 2-2 中，程序烧录后，当按下 S2 按键时，GPIO5 被拉低，程序检测到低电平后，将 GPIO9 置为低电平点亮 LED，并打印信息；当松开按键，GPIO5 因内部上拉而恢复高电平，程序检测到后将 GPIO9 置为高电平熄灭 LED。

2.2 ADC 实验

a) 实验 3 使用按键控制 LED 亮灭

实验代码：

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>

#include <hi_types_base.h>
#include <hi_io.h>
#include <hi_gpio.h>
#include <hi_adc.h>
#include <hi_stdlib.h>
#include <hi_task.h>

#include "ohos_init.h"
#include "cmsis_os2.h"

#define LED_GPIO_PIN       9                /* 板载 LED 连接在 GPIO9 */
#define ADC_CHANNEL        HI_ADC_CHANNEL_2  /* 多键接在 ADC2 上 */
#define ADC_SAMPLE_COUNT   64               /* 采样 64 次做滤波 */
#define VLT_MIN_INIT       100              /* 初始最小电压阈值 */

#define KEY_NONE           0
#define KEY_S1             1  /* 打开 LED */
#define KEY_S2             2  /* 关闭 LED */
#define KEY_S3             3  /* 切换 LED */

static hi_u16  g_adc_buf[ADC_SAMPLE_COUNT];
```

```
26  static int     key_status = KEY_NONE;
27  static char    key_flg    = 0;
28  static char    led_state  = 0;   /* 0=关, 1=开 */
29
30  static int get_key_event(void)
31  {
32      int e = key_status;
33      key_status = KEY_NONE;
34      return e;
35  }
36
37  /* 把 ADC 原始码转为电压并判定是哪键 */
38  static void convert_to_voltage(hi_u32 len)
39  {
40      float vlt_max = 0.0f, vlt_min = VLT_MIN_INIT, vlt;
41      for (hi_u32 i = 0; i < len; i++) {
42          float voltage = (float)g_adc_buf[i] * 1.8f * 4.0f / 4096.0f;
43          if (voltage > vlt_max) vlt_max = voltage;
44          if (voltage < vlt_min) vlt_min = voltage;
45      }
46      vlt = (vlt_max + vlt_min) / 2.0f;
47
48      if ((vlt > 0.4f && vlt < 0.6f) && !key_flg) {
49          key_flg    = 1;
50          key_status = KEY_S1;
51      }
52      else if ((vlt > 0.8f && vlt < 1.1f) && !key_flg) {
53          key_flg    = 1;
54          key_status = KEY_S2;
55      }
56      else if ((vlt > 0.01f && vlt < 0.3f) && !key_flg) {
57          key_flg    = 1;
58          key_status = KEY_S3;
59      }
```

```
60      else if (vlt > 3.0f) {
61          /* 松开时复位，允许下一次检测 */
62          key_flg = 0;
63      }
64  }
65
66  /* 连续读 ADC_SAMPLE_COUNT 次 ADC，并更新 key_status */
67  static void sample_adc_key(void)
68  {
69      hi_u16 raw;
70      memset_s(g_adc_buf, sizeof(g_adc_buf), 0, sizeof(g_adc_buf));
71
72      for (int i = 0; i < ADC_SAMPLE_COUNT; i++) {
73          if (hi_adc_read(ADC_CHANNEL, &raw,
74                          HI_ADC_EQU_MODEL_1,
75                          HI_ADC_CUR_BAIS_DEFAULT, 0) != HI_ERR_SUCCESS) {
76              printf("ADC read failed\n");
77              return;
78          }
79          g_adc_buf[i] = raw;
80      }
81      convert_to_voltage(ADC_SAMPLE_COUNT);
82  }
83
84  /* 主任务：检测按键，控制 LED 并在串口输出状态 */
85  static void LedTask(void *arg)
86  {
87      (void)arg;
88
89      /* 1. 初始化 LED GPIO */
90      hi_gpio_init();
91      hi_io_set_func(HI_IO_NAME_GPIO_9, HI_IO_FUNC_GPIO_9_GPIO);
92      hi_gpio_set_dir(HI_GPIO_IDX_9, HI_GPIO_DIR_OUT);
93      /* 默认为关灯（GPIO 输出高电平）*/
```

```
94      hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE1);
95      led_state = 0;
96
97      hi_io_set_func(HI_IO_NAME_GPIO_5, HI_IO_FUNC_GPIO_5_GPIO);
98
99      while (1) {
100         sample_adc_key();
101         int evt = get_key_event();
102         if (evt == KEY_S1) {
103             if (!led_state) {
104                 hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE0);
105                 led_state = 1;
106                 printf("LED ON\n");
107             }
108         } else if (evt == KEY_S2) {
109             if (led_state) {
110                 hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE1);
111                 led_state = 0;
112                 printf("LED OFF\n");
113             }
114         } else if (evt == KEY_S3) {
115             /* 切换状态 */
116             if (led_state) {
117                 hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE1);
118                 led_state = 0;
119                 printf("LED OFF\n");
120             } else {
121                 hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE0);
122                 led_state = 1;
123                 printf("LED ON\n");
124             }
125         }
126         // 50ms 延时
127         osDelay(50);
```

```
128         }
129     }
130
131     void adc_led(void)
132     {
133         osThreadAttr_t attr = {
134             .name       = "ADC_LED_Task",
135             .stack_size = 2048,
136             .priority   = 26,
137         };
138         if (osThreadNew(LedTask, NULL, &attr) == NULL) {
139             printf("Failed to create ADC_LED_TASK\n");
140         }
141     }
142     SYS_RUN(adc_led);
```

实验现象：

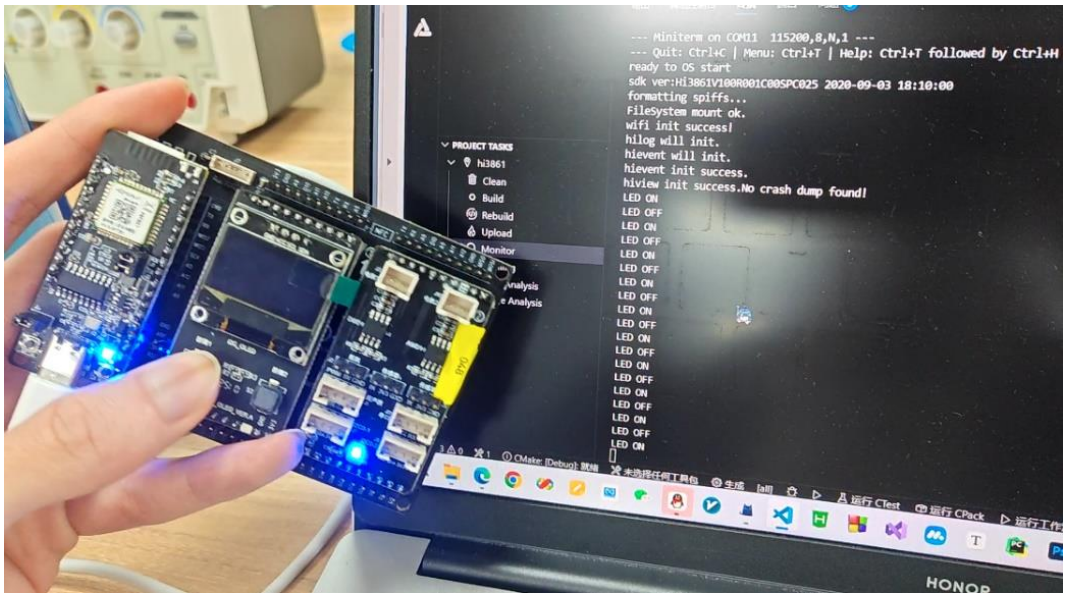

图3：实验 3 使用按键控制 LED 亮灭

具体实验视频请见“实验记录”文件夹。

实验分析：

本次实验的目是利用 Hi3861V100 芯片的 ADC (Analog-to-Digital Converter) 功能，通过 一个 GPIO 引脚 检测三个不同按键的按下事件，并根据不同的按键来控制板载 LED 灯的亮、灭或状态切换。

实验原理是三个按键（OLED 板上的 S1、S2 和核心板上的 S3）并非简单地将引脚接地，而是通过一个电阻分压网络连接到同一个 ADC 引脚（GPIO05/ADC2） 。当没有按键按下时，ADC 引脚通过上拉电阻 R2（4.7KΩ）被拉到较高的电压 。当按下不同的按键时，会将不同阻值的电阻接入电路，从而改变分压比，使 ADC 引脚上产生一个独特的、稳定的电压值。通过读取 ADC 转换后的数字值，就可以反推出当前 ADC 引脚上的电压，进而根据电压所在的范围判断出是哪个按键被按下了 。

在代码主循环中，while(1)循环以大约 50ms 的周期运行 (osDelay(50))。每次循环都调用 sample_adc_key()来采集和分析 ADC 电压。通过 get_key_event()获取被识别的按键事件。if - else if 结构根据获取的按键事件执行相应的操作：

KEY_S1: 调用 hi_gpio_set_ouput_val 将 GPIO9 置为低电平，点亮 LED，并打印 "LED ON" 。

KEY_S2: 将 GPIO9 置为高电平，熄灭 LED，并打印"LED OFF" 。

KEY_S3: 检查当前 LED 状态 led_state，并执行相反的操作（亮则灭，灭则亮），实现切换功能，并打印对应状态 。

本次实验实现了在单个引脚上对多个按键输入的识别，通过“采样-滤波-判断-执行”的流程，将不同的按键输入映射为对 LED 的“开、关、切换”控制。

2.3 OLED 实验

a) 实验 4 OLED 屏幕显示学号姓名

实验代码：

中文汉字的显示，仿照示例代码中的 TestDrawChinese1()函数，并且实用工具生成自己名字的字模，在函数 Ssd1306TestTask()中进行调用即可。学号的显示，可以使用 ssd1306 库中的函数 ssd1306_SetCursor()将指针移到第二行，使用 ssd1306_DrawString()函数绘制学号。

部分代码如下：

```
void TestDrawChinese3(void)
{
    // 1) 定义你的字宽/字高（以像素为单位）
    const uint32_t W = 16/* 字宽，比如 16 */;
    const uint32_t H = 16/* 字高，比如 16 */;

    // 2) 计算每个字符占用的字节数 = W * H / 8，填入字模
    uint8_t myFonts[][32] = {
        {
            /*-- ID:0,字符:"马",ASCII 编码:C2ED,对应字:宽 x 高=16x16,
画布:宽 W=16 高 H=16,共 32 字节*/
            0x00,0x20,0x3F,0xF0,0x00,0x20,0x08,0x20,0x08,0x20,0x08,
0x20,0x08,0x20,0x08,0x24,
            0x0F,0xFE,0x00,0x04,0x00,0x24,0xFF,0xF4,0x00,0x04,0x00,
0x04,0x00,0x28,0x00,0x10
        },
        {
            /*-- ID:1,字符:"成",ASCII 编码:B3C9,对应字:宽 x 高=16x16,
画布:宽 W=16 高 H=16,共 32 字节*/
            0x00,0x80,0x00,0xA0,0x00,0x90,0x3F,0xFC,0x20,0x80,0x20,
0x80,0x20,0x84,0x3E,0x44,
            0x22,0x48,0x22,0x48,0x22,0x30,0x2A,0x20,0x24,0x62,0x40,
0x92,0x81,0x0A,0x00,0x06
        },
        {
            /*-- ID:2,字符:"龙",ASCII 编码:C1FA,对应字:宽 x 高=16x16,
画布:宽 W=16 高 H=16,共 32 字节*/
            0x02,0x00,0x02,0x40,0x02,0x20,0x02,0x04,0xFF,0xFE,0x02,
0x80,0x02,0x88,0x04,0x88,
            0x04,0x90,0x04,0xA0,0x08,0xC0,0x08,0x82,0x11,0x82,0x16,
0x82,0x20,0x7E,0x40,0x00
        },


    };

    // 3) 清屏（先把缓冲区全填黑）
    ssd1306_Fill(Black);

    // 4) 循环绘制每个字模到缓冲区
```

```
    //    x 偏移 = i * W; y 固定为 0
    //    stride 参数传 W，即每行 W 位（内部会自动按 8 位分组）
    for (size_t i = 0; i < sizeof(myFonts)/sizeof(myFonts[0]); i++) {
        ssd1306_DrawRegion(
            /* x */ i * W,
            /* y */ 0,
            /* w */ W,
            /* h */ H,
            /* data */ myFonts[i],
            /* size */ sizeof(myFonts[0]),
            /* stride */ W
        );
    }

    // 到第二行
    ssd1306_SetCursor(0, 20);
    ssd1306_DrawString("25120232201399", Font_7x10, White);

    // 5) 一次性把缓冲区刷新到屏幕
    ssd1306_UpdateScreen();
}
```

实验现象：

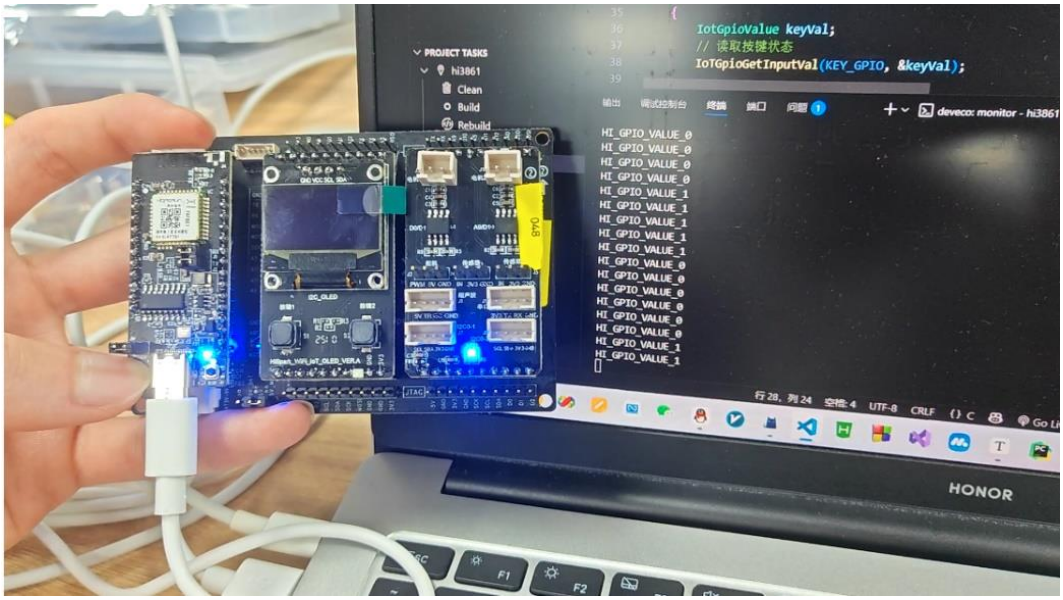

图4：实验 4 OLED 屏幕显示学号姓名

具体实验图片请见“实验记录”文件夹。

实验分析：

驱动库的工作模式：

第一步：在内存缓冲区中绘制内容：

代码首先调用 TestDrawChinese3() 函数。

ssd1306_Fill(Black)：调用库函数，将整个内存缓冲区填充为黑色，实现清屏效果 。

ssd1306_DrawRegion(...)：此函数用于绘制位图。代码通过它将 myFonts 数组中预先定义好的汉字点阵数据绘制到缓冲区的指定位置。

ssd1306_DrawString(...)：接着调用库函数，将学号绘制到缓冲区的另一行 。

至此，所有操作都只在内存中进行，物理屏幕尚未改变。

第二步：将缓冲区内容刷新到屏幕

ssd1306_UpdateScreen()：这是关键的一步。调用此函数后，驱动库会将整个内存缓冲区的数据通过 I2C 总线一次性发送给 SSD1306 芯片，芯片再根据这些数据点亮或熄灭对应的物理像素点，最终完成屏幕的刷新 。

程序通过“配置硬件 -> 初始化驱动 -> 绘制到内存 -> 更新到屏幕”的流程，在 OLED 屏幕上显示了姓名和学号。

2.4 PWM 和 WiFi 实验

a) 实验 5-1 PWM 实现 LED 呼吸灯

实验代码：

```
#include <stdio.h>
#include "ohos_init.h"      // 系统启动宏
#include "cmsis_os2.h"      // RTOS2 接口
#include "iot_gpio.h"       // HAL: GPIO 接口
#include "hi_io.h"          // SDK: IO 复用、上拉
#include "iot_pwm.h"        // HAL: PWM 接口
#include "hi_pwm.h"         // SDK: PWM 接口

```

```c
#define LED_GPIO_PIN        HI_IO_NAME_GPIO_9
#define PWM_PORT            HI_PWM_PORT_PWM0
#define PWM_FREQ            4000    // 频率 4kHz（>=2442Hz）
#define MAX_DUTY_PERCENT    99      // HAL 占空比最大 99%
#define DUTY_STEP 1                 // 每次改变 1%

static void PwmBreathingTask(void *arg)
{
    (void)arg;
    // 1. 初始化 GPIO
    IoTGpioInit(LED_GPIO_PIN);
    hi_io_set_func(LED_GPIO_PIN, HI_IO_FUNC_GPIO_9_PWM0_OUT);    // 复用为PWM0
    IoTGpioSetDir(LED_GPIO_PIN, IOT_GPIO_DIR_OUT);

    // 2. 初始化 PWM 模块
    IoTPwmInit(PWM_PORT);

    // 3. 呼吸灯循环
    while (1) {
        for (int d = 1; d <= 30; d+= DUTY_STEP) {
            IoTPwmStart(PWM_PORT, d, PWM_FREQ);
            osDelay(5);
        }
        for (int d = 31; d <= MAX_DUTY_PERCENT; d+= (DUTY_STEP+3)) {
            IoTPwmStart(PWM_PORT, d, PWM_FREQ);
            osDelay(5);
        }
        for (int d = MAX_DUTY_PERCENT; d >= 31; d-= (DUTY_STEP+3)) {
            IoTPwmStart(PWM_PORT, d, PWM_FREQ);
            osDelay(5);
        }
        for (int d = 30; d >= 1; d-= DUTY_STEP) {
            IoTPwmStart(PWM_PORT, d, PWM_FREQ);
            osDelay(5);
```

```
        }
    }
}

// 入口：创建线程
static void PwmBreathingEntry(void)
{
    osThreadAttr_t attr = {
        .name       = "PwmBreathing",
        .stack_size = 2048,
        .priority   = osPriorityNormal
    };
    if (osThreadNew(PwmBreathingTask, NULL, &attr) == NULL) {
        printf("Failed to create PwmBreathingTask!\n");
    }
}
SYS_RUN(PwmBreathingEntry);
```

实验现象：

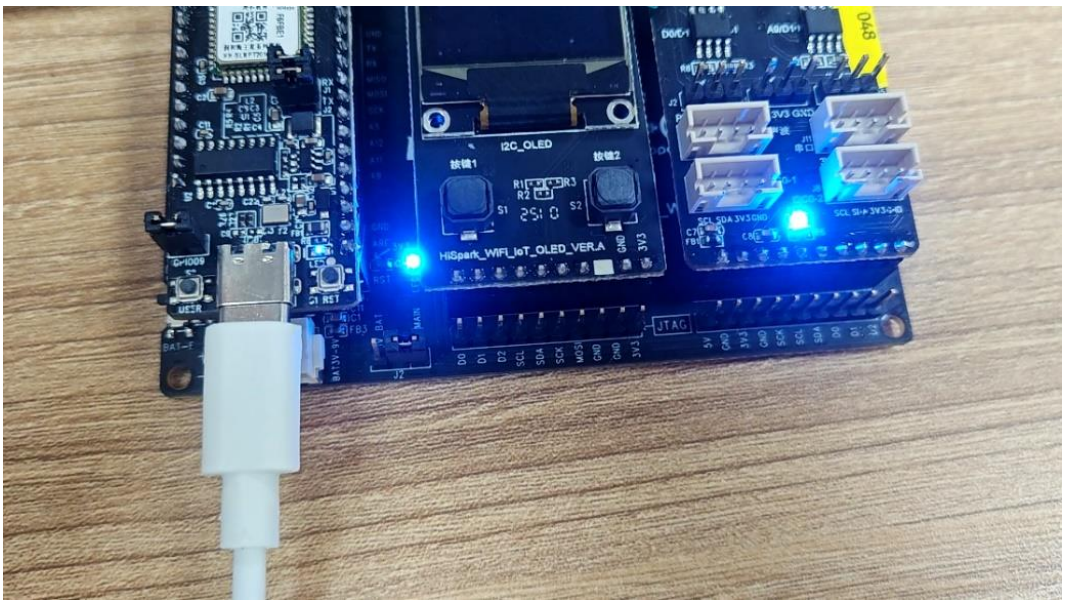

图5：实验 5-1 PWM 实现 LED 呼吸灯

具体实验视频请见“实验记录”文件夹。

实验分析：

PWM 是一种通过高速开关数字信号来控制模拟电路的技术。通过改变一个周期内高电平时间占总时间的比例（即占空比），可以控制输出到负载（如 LED）的平均功率。占空比越高，在一个周期内高电平时间越长，输出的平均电压越高。占空比越低，高电平时间越短，输出的平均电压越低。只要开关频率足够高（人眼无法分辨闪烁），改变占空比就能平滑地控制 LED 的亮度。

代码中，初始化流程 (PwmBreathingTask 函数):

GPIO 初始化: IoTGpioInit()初始化 GPIO 引脚。

引脚功能复用: hi_io_set_func(LED_GPIO_PIN, HI_IO_FUNC_GPIO_9_PWM0_OUT) 是关键一步。它将 GPIO9 的功能从普通 IO 切换为 PWM0 的输出引脚，这样 PWM 硬件模块才能通过这个引脚输出波形。

PWM 模块初始化: IoTPwmInit(PWM_PORT) 初始化 PWM0 硬件模块。

呼吸效果逻辑 (while(1)循环):

程序通过四个连续的 for 循环来动态调整占空比 d，从而实现呼吸效果。

IoTPwmStart(PWM_PORT, d, PWM_FREQ): 这是核心 API 调用，它启动 PWM 端口，并设置其占空比为 d，频率为 PWM_FREQ。

变暗过程: 前两个 for 循环将占空比 d 从 1%逐步增加到 99%。根据硬件分析，占空比升高，LED 会逐渐变暗。

变亮过程: 后两个 for 循环将占空比 d 从 99%逐步减小到 1%。占空比降低，LED 会逐渐变亮。

同时，代码在占空比的不同阶段设计了不同的步进值，使得呼吸的节奏快慢结合，效果更自然。

b) 实验 5-2 开发板连接 WiFi 热点并在 OLED 显示

实验代码：

由于代码整体较长，这里仅对两个重要的函数进行展示：

Wi-Fi 连接发起 (hi_wifi_start_sta 函数)：

```
1  int hi_wifi_start_sta(void)
2  {
```

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

    const unsigned char wifi_vap_res_num = APP_INIT_VAP_NUM;
    const unsigned char wifi_user_res_num = APP_INIT_USR_NUM;
    unsigned int  num = WIFI_SCAN_AP_LIMIT;

    // 1) 初始化 Wi-Fi
    ret = hi_wifi_init(wifi_vap_res_num, wifi_user_res_num);

    // 2) 启动 STA 模式
    ret = hi_wifi_sta_start(ifname, &len);

    // 3) 注册事件回调
    ret = hi_wifi_register_event_callback(wifi_wpa_event_cb);

    // 4) 查找 netif
    g_lwip_netif = netifapi_netif_find(ifname);

    // 5) 扫描
    ret = hi_wifi_sta_scan();

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

    hi_wifi_ap_info *pst_results = malloc(sizeof(hi_wifi_ap_info) *
WIFI_SCAN_AP_LIMIT);
    ret = hi_wifi_sta_scan_results(pst_results, &num);
    for (unsigned int loop = 0; (loop < num) && (loop < WIFI_SCAN_AP_LIMIT); loop++) {
        printf("SSID: %s\n", pst_results[loop].ssid);
    }
    free(pst_results);

    /* if received scan results, select one SSID to connect */
    ret = hi_wifi_start_connect();
```

```
37
38      return 0;
39  }
```

hi_wifi_init(): 初始化 Wi-Fi 底层模块。

hi_wifi_sta_start(): 启动 Wi-Fi 的 STA 模式 。

hi_wifi_register_event_callback(): 注册事件回调函数。这是异步编程的关键，它告诉 Wi-Fi 服务，当有连接、断开等事件发生时，应该去调用 wifi_wpa_event_cb 函数。

hi_wifi_sta_scan(): 发起主动扫描，寻找周边的 WiFi 热点 。

hi_wifi_start_connect(): 在扫描之后，调用此函数发起连接。函数内部会填充一个 hi_wifi_assoc_request 结构体，包含要连接的目标 SSID、密码和加密方式 (HI_WIFI_SECURITY_WPA2PSK)，然后发起连接请求 。

Wi-Fi 事件处理 (wifi_wpa_event_cb 回调函数)：

```
1   void wifi_wpa_event_cb(const hi_wifi_event *hisi_event)
2   {
3       if (hisi_event == NULL)
4           return;
5
6       switch (hisi_event->event) {
7           case HI_WIFI_EVT_SCAN_DONE:
8               printf("WiFi: Scan results available\n");
9               break;
10          case HI_WIFI_EVT_CONNECTED:
11              printf("WiFi: Connected\n");
12              netifapi_dhcp_start(g_lwip_netif);
13              // OLED 上显示已连接及 SSID
14              Oled_Init();
15              Oled_ShowString(0, 0, "WiFi: Connected");
16              Oled_ShowString(0, 16, g_connected_ssid);
17              break;
18          case HI_WIFI_EVT_DISCONNECTED:
19              printf("WiFi: Disconnected\n");
```

```
            netifapi_dhcp_stop(g_lwip_netif);
            hi_sta_reset_addr(g_lwip_netif);
            // OLED 上显示断开
            Oled_ShowString(0, 0, "WiFi: Disconnected");
            break;
        case HI_WIFI_EVT_WPS_TIMEOUT:
            printf("WiFi: wps is timeout\n");
            break;
        default:
            break;
    }
}
```

此函数在 Wi-Fi 状态发生变化时被系统自动调用。

case HI_WIFI_EVT_CONNECTED: 当开发板成功连接到热点时，此分支被触发。代码会打印日志，并调用 Oled_ShowString 在 OLED 屏幕上显示"WiFi: Connected"和连接上的 SSID 名称。

case HI_WIFI_EVT_DISCONNECTED: 当连接断开时，此分支被触发。代码同样会更新 OLED 屏幕显示"WiFi: Disconnected"。

实验现象：

图6：实验 5-2 开发板连接 WiFi 热点并在 OLED 显示

具体实验图片请见“实验记录”文件夹。

实验分析：

本次实验让我们体验到了 Hi3861 开发板在 OpenHarmony 系统下的两种核心驱动能力。实验 5-1 通过配置引脚复用和调用 PWM API，实现了对 LED 亮度的控制。实验 5-2 通过调用 Wi-Fi API，以 STA 模式成功实现了扫描、认证和关联的完整网络连接流程，并通过事件回调机制和 OLED 显示提供人机交互。两个实验都展现了 OpenHarmony 驱动开发的分层思想和 API 易用性。

2.5 MQTT 实验

a）实验 6 开发板和电脑通过 MQTT 代理互发消息

实验代码：

本次实验完整代码已经给出，只需要修改代码中的几个值为自己的即可。部分代码如下：

```
    /* copy SSID to assoc_req */
    rc = memcpy_s(assoc_req.ssid, HI_WIFI_MAX_SSID_LEN + 1, "Ma", 8);
    /* 热点密码 */
    memcpy(assoc_req.key, "md205525", 11);

    NetworkConnect(&n, "192.168.114.144", 1883);
    data.username.cstring = "admin";
    data.password.cstring = "md205525";
    //订阅消息，并设置回调函数
    MQTTSubscribe(&mq_client, "name", 0, mqtt_callback);
while(1)
    {
        MQTTMessage message;

        message.qos = QOS1;
        message.retained = 0;
        const char student_id[] = "25120232201399";
        message.payload = (void *)student_id;
        message.payloadlen = strlen(student_id);

        //发送消息
        if (MQTTPublish(&mq_client, "ID", &message) < 0)
        {
            printf("MQTTPublish faild !\r\n");
        }
        usleep(1000000);
    }
```

实验现象：

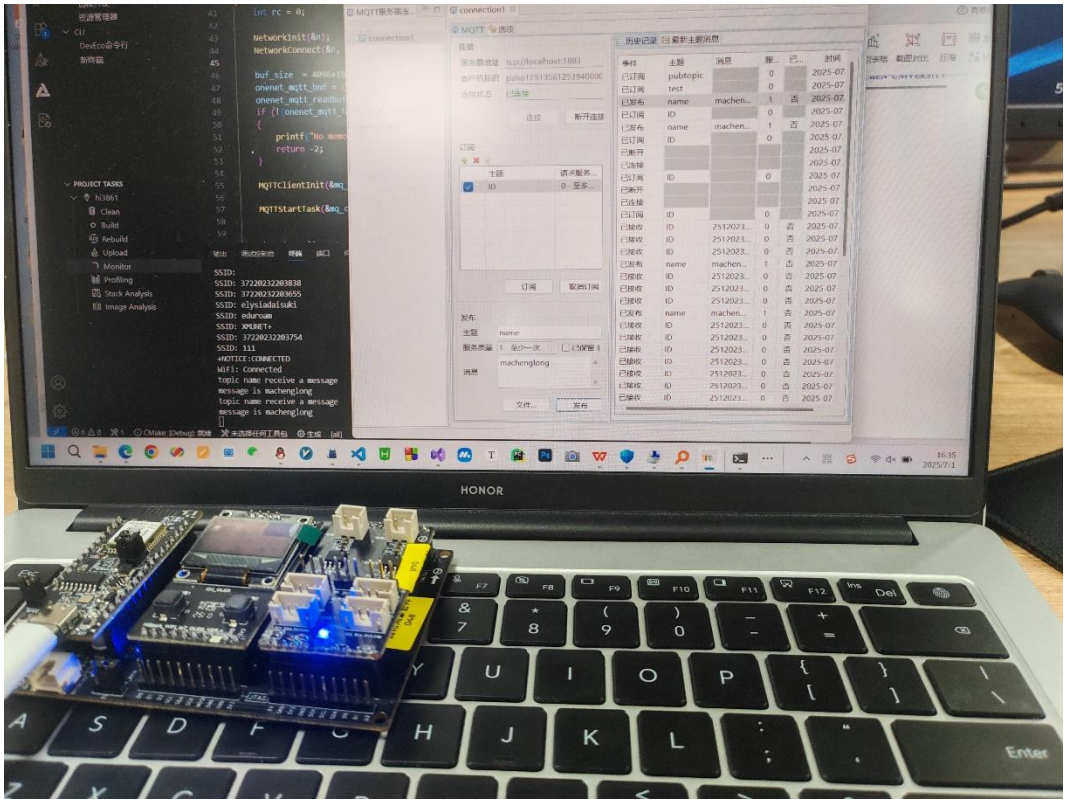

图7：实验 6 开发板和电脑通过 MQTT 代理互发消息

具体实验图片请见“实验记录”文件夹。

实验分析：

本实验的代码逻辑分为两个主要步骤：首先通过 mqtt_entry.c 建立网络连接，然后调用 mqtt_test.c 中的函数执行 MQTT 通信。实验将网络编程与应用层协议结合起来，使我们学习了物联网设备进行数据上报和指令接收的核心流程。通过调用 Paho-MQTT 库的 API，代码实践了 MQTT 协议中建立连接（Connect）、订阅（Subscribe）、发布（Publish）以及接收消息的完整生命周期。

三、 智能小车基础部分

小车的组装过程略。

3.1 小车的测试

各组件功能均正常，但是存在以下几个小问题：

红外传感器灵敏度非常高：发出红外后，即使只接收到微弱的红外信号，指示灯也会亮。这导致，小车传感器处于黑线上方时，红外传感器也不会指示灯也不会灭，而是持续输出 VALUE1 高电平，使得小车循迹功能无法正常实现。解决方法在探索几天后得出，期间尝试过：更换自购的传感器（希望灵敏度可以低一点，但新传感器灵敏度都很高），调节传感器与黑线的距离（最开始尝试减少传感器与地面的距离，希望可以让黑线对红外的吸收更彻底。但多次实验后该方案告吹。后来尝试增加传感器与地面的距离，以此降低传感器的灵敏度，此方案最终被采纳）。

小车左右两个**电机**的性能不同：这导致小车两个电机以相同功率（相同占空比）运行时，无法走出直线，而是逐渐往一侧偏移（一侧电机转得更快，一侧转的偏慢）。考虑到不同占空比时电机转速并不是线性相关，无法通过微调左右两侧电机占空比来使得小车走出直线，遂作罢。

3.2 系统主线程

我们组的小车循迹功能由于同时集成了循迹、避障（自动停车）、按键 S2 控制小车速度等等功能，且小车整体包含循迹、避障（绕过障碍物）、远程控制三大模式，所以如果代码的任务没有协调完美，会使得**线程阻塞，按键失灵**。我们最终的方案是：将三大功能以状态机的形式设置在 robot_control.c 中，通过按键 S1 切换状态；循迹功能中的三个模块通过三个 task 同步运行，互相不阻塞，最终成功运行，不再出现阻塞线程、按键失灵等情况。

部分代码如下：

```
1  robot_control.c
2  //按键 S1 切换功能
3  void gpio5_isr_func_mode(void)
4  {
5      printf("gpio5_isr_func_mode start\n");
6      unsigned int tick_interval = 0;
7      unsigned int current_gpio5_tick = 0;
8
9      current_gpio5_tick = hi_get_tick();
```

```
    tick_interval = current_gpio5_tick - g_gpio5_tick;

    if (tick_interval < KEY_INTERRUPT_PROTECT_TIME) {
        return;
    }
    g_gpio5_tick = current_gpio5_tick;

    if (g_car_status == CAR_STOP_STATUS) {
        g_car_status = CAR_TRACE_STATUS;                // 寻迹
        printf("trace\n");
    } else if (g_car_status == CAR_TRACE_STATUS) {
        g_car_status = CAR_OBSTACLE_AVOIDANCE_STATUS;   // 超声波避障
        printf("ultrasonic\n");
    } else if (g_car_status == CAR_OBSTACLE_AVOIDANCE_STATUS)
{
        g_car_status = CAR_REMOTE_CONTROL_STATUS;      // 远程控制
        printf("remote control\n");
    } else if (g_car_status == CAR_REMOTE_CONTROL_STATUS) {
        g_car_status = CAR_STOP_STATUS;                // 停止
        printf("stop\n");
    }
}
```

trace_model.c

```
// 2. 创建超声、按键、巡线任务
        osThreadAttr_t attr;
        attr.priority = 25;

        attr.name = "UltrasonicTask";
        attr.stack_size = 4096;
        if (osThreadNew(UltrasonicTask, NULL, &attr) == NULL)
        {
            printf("Failed to create UltrasonicTask\n");
        }
```

```
43
44          attr.name = "SpeedKeyTask";
45          attr.stack_size = 2048;
46          if (osThreadNew(SpeedKeyTask, NULL, &attr) == NULL)
47          {
48              printf("Failed to create SpeedKeyTask\n");
49          }
50
51          attr.name = "TraceTask";
52          attr.stack_size = 4096;
53          if (osThreadNew(TraceTask, NULL, &attr) == NULL)
54          {
55              printf("Failed to create TraceTask\n");
56          }
```

3.3 小车的循迹逻辑

我们的小车循迹功能，主要使用“状态机”来支持各个逻辑的判断。我们共有：

TRACE_NORMAL：正常行驶；

TRACE_CROSS_FORWARD：两侧传感器均检测到黑线，先向前探测一段路；

TRACE_CROSS_CHECK：接上，探测一段路后进行再次检查；

TRACE_END：到达终点，停止循迹；

TRACE_SHARP_FORWARD：检测到单侧黑线，先向前探测一段路；

TRACE_SHARP_TURN：探测后判断为转弯，进行转弯；

TRACE_SHARP_ADJUST：转弯结束后，对车身进行反向微调，使车身平行黑线；

这七种状态，在不同的情况下进入不同的状态判断；同时，由于状态可能出现误判，所以在所有状态中同时引入修正，即如果在一个状态中检测到了另一个优先度更高的状态，就切换到优先度更高的状态。比如，当单侧传感器检测到黑线时，小车会进入状态TRACE_SHARP_FORWARD（检测到单侧黑线，先向前探测一段路），在此状态中，又出现小车两侧传感器均检测到黑线，那么就可能是这个特殊情况：小车在进入十字交叉口时，由于车身与黑线不平行，导致一侧传感器先检测到黑线，另一侧才检测到黑线。那么，

此时就应该切换到状态 TRACE_CROSS_FORWARD（两侧传感器均检测到黑线，先向前探测一段路），因此，状态 TRACE_CROSS_FORWARD 的优先级比状态 TRACE_SHARP_FORWARD 更高，这种情况下小车会切换到状态 TRACE_CROSS_FORWARD。

主要判断逻辑、状态切换情况如下表所示：

状态	触发条件	动作	下一个状态
TRACE_NORMAL	—	双白：匀速前进; 双黑：认为是交叉或终点; 单黑：识别急转弯方向	双黑→ TRACE_CROSS_FORWARD 左黑→ TRACE_SHARP_FORWARD (左急转) 右黑→ TRACE_SHARP_FORWARD (右急转) 白→保持 TRACE_NORMAL
TRACE_CROSS_FORWARD	来自 TRACE_NORMAL 的“双黑”	前行 CROSS_FORWARD_DELAY_MS，同时监测单黑改为急转	单黑→ TRACE_SHARP_FORWARD 延时完→ TRACE_CROSS_CHECK
TRACE_CROSS_CHECK	来自 CROSS_FORWARD 延时后	双黑→TRACE_END 全白→TRACE_NORMAL 单黑→ TRACE_SHARP_FORWARD	如表所示
TRACE_END	由 CROSS_CHECK 双黑判定终点	停车（spd=0），保持不变	持续 TRACE_END
TRACE_SHARP_FORWARD	来自 NORMAL/CROSS_FORWARD 的“单黑”	前行 SHARP_FORWARD_DELAY_MS，期间也检测双黑→回 CROSS_FORWARD	双黑→ TRACE_CROSS_FORWARD 延时完→ TRACE_SHARP_TURN
TRACE_SHARP_TURN	来自 SHARP_FORWARD 延时后	单侧停轮、另一侧匀速，直到“对侧”也检测到黑（车头对准了弯道出口）	满足退出条件→ TRACE_SHARP_ADJUST

状态	触发条件	动作	下一个状态
TRACE_SHARP_ADJUST	来自 SHARP_TURN 条件满足	微调 SHARP_ADJUST_DELAY_MS：用对侧车轮短转修正姿态；期间如检测到双黑→CROSS_FORWARD 或单黑→重回 SHARP_FORWARD	双黑→TRACE_CROSS_FORWARD 单黑→TRACE_SHARP_FORWARD 延时完→TRACE_NORMAL

同时，我们在循迹模块中加入声明超声波测距的函数，并在循迹主函数中调用以达到“循迹过程中，如果前方有障碍物，就自动停止，障碍物移除后，恢复前进”的效果。主要代码如下：

```
1  // 声明超声波测距函数
2  extern float GetDistance(void);
3
4  // 循迹模式下的七个状态
5  typedef enum
6  {
7      TRACE_NORMAL,        // 正常巡线
8      TRACE_CROSS_FORWARD, // 探测到左右同时黑，前行短距用于区分交叉口/终点
9      TRACE_CROSS_CHECK,   // 前行后复查：判定交叉口还是终点
10     TRACE_END,           // 终点：停车
11     TRACE_SHARP_FORWARD, // 探测到单边黑，前行短距进入急转弯
12     TRACE_SHARP_TURN,    // 急转弯阶段：持续转向
13     TRACE_SHARP_ADJUST   // 急转弯结束后微调
14 } TraceState;
15
16 static TraceState trace_state = TRACE_NORMAL;
17 static int state_counter = 0;        // 毫秒级计数
18 static bool sharp_left_turn = false; // 记录急转弯方向：true=左侧先黑，做左急转
19
20 // 延时参数
21 #define CROSS_FORWARD_DELAY_MS 12 // 遇横线，停止时间（双黑，停止）
22 #define SHARP_FORWARD_DELAY_MS 12 // 单侧黑线，继续前行时间
```

```
#define SHARP_ADJUST_DELAY_MS 30  // 转弯后，调整时间

// 普通线程里做巡线+遇障停车逻辑
static void TraceTask(void *arg)
{
    (void)arg;
    for (;;)
    {
        // 等待进 trace 模式
        while (g_car_status != CAR_TRACE_STATUS)
        {
            osDelay(5);
        }

        // 模式进入瞬间重置状态机
        trace_state = TRACE_NORMAL;
        state_counter = 0;
        sharp_left_turn = false;

        // 进入 trace 模式后循环
        while (g_car_status == CAR_TRACE_STATUS)
        {
            // ——— (A) 超声波遇障停车 ———
            if (g_distance < DISTANCE_BETWEEN_CAR_AND_OBSTACLE)
            {
                set_car_speed(0, 0);
                osDelay(5);
                continue; // 跳过红外
            }

            // 1) 读取左右传感器电平
            IotGpioValue io_left, io_right;
            IoTGpioGetInputVal(GPIO11, &io_left);
            IoTGpioGetInputVal(GPIO12, &io_right);
```

```
57
58              // IOT_GPIO_VALUE0 表示检测到黑色
59              bool left_black = (io_left == IOT_GPIO_VALUE0);
60              bool right_black = (io_right == IOT_GPIO_VALUE0);
61
62              // 2) 状态机
63              int spd_l = default_speed;
64              int spd_r = default_speed;
65
66              switch (trace_state)
67              {
68              case TRACE_NORMAL:
69                  if (left_black && right_black)
70                  {
71                      // 同时检测到黑：可能交叉口或终点
72                      trace_state = TRACE_CROSS_FORWARD;
73                      state_counter = 0;
74                      // 前行短距：维持前进
75                      spd_l = spd_r = default_speed;
76                  }
77                  else if (left_black && !right_black)
78                  {
79                      // 仅左检测到黑：急转弯开始
80                      sharp_left_turn = true;
81                      trace_state = TRACE_SHARP_FORWARD;
82                      state_counter = 0;
83                      spd_l = spd_r = default_speed;
84                  }
85                  else if (right_black && !left_black)
86                  {
87                      // 仅右检测到黑：急转弯开始
88                      sharp_left_turn = false;
89                      trace_state = TRACE_SHARP_FORWARD;
90                      state_counter = 0;
```

```
                spd_l = spd_r = default_speed;
            }
            else
            {
                // 白线：正常前进
                spd_l = spd_r = default_speed;
            }
            break;

        case TRACE_CROSS_FORWARD:
            // 前行一段时间，再做复查

            if (left_black && !right_black)
            {
                // 仅左检测到黑：急转弯开始
                sharp_left_turn = true;
                trace_state = TRACE_SHARP_FORWARD;
                state_counter = 0;
                spd_l = spd_r = default_speed;
                break;
            }
            else if (right_black && !left_black)
            {
                // 仅右检测到黑：急转弯开始
                sharp_left_turn = false;
                trace_state = TRACE_SHARP_FORWARD;
                state_counter = 0;
                spd_l = spd_r = default_speed;
                break;
            }

            if (state_counter < CROSS_FORWARD_DELAY_MS)
            {
                spd_l = spd_r = default_speed;
```

```
125                     state_counter++;
126                 }
127                 else
128                 {
129                     trace_state = TRACE_CROSS_CHECK;
130                     state_counter = 0;
131                 }
132                 break;
133 
134             case TRACE_CROSS_CHECK:
135                 if (left_black && right_black)
136                 {
137                     // 复查依然是宽线：到达终点
138                     trace_state = TRACE_END;
139                     spd_l = spd_r = 0;
140                 }
141                 else if (!left_black && !right_black)
142                 {
143                     // 复查全白：经过交叉口，继续正常巡线
144                     trace_state = TRACE_NORMAL;
145                     spd_l = spd_r = default_speed;
146                 }
147                 else
148                 {
149                     // 复查单边：实际是一个弯道，进入急转弯逻辑
150                     sharp_left_turn = left_black;
151                     trace_state = TRACE_SHARP_FORWARD;
152                     state_counter = 0;
153                     spd_l = spd_r = default_speed;
154                 }
155                 break;
156 
157             case TRACE_END:
158                 // 终点一直停车
```

```
159                 spd_l = spd_r = 0;
160                 break;
161
162             case TRACE_SHARP_FORWARD:
163                 // 一侧黑线，前行短距，进入急转弯
164                 if (left_black && right_black)
165                 {
166                     // 同时检测到黑：可能交叉口或终点
167                     trace_state = TRACE_CROSS_FORWARD;
168                     // 前行短距：维持前进
169                     spd_l = spd_r = default_speed;
170                 }
171                 // 急转弯前行短距，保证车头完全驶入弯道
172                 else if (state_counter < SHARP_FORWARD_DELAY_MS)
173                 {
174                     spd_l = spd_r = default_speed;
175                     state_counter++;
176                 }
177                 else
178                 {
179                     trace_state = TRACE_SHARP_TURN;
180                     state_counter = 0;
181                 }
182                 break;
183
184             case TRACE_SHARP_TURN:
185                 // 持续转向
186                 if (sharp_left_turn)
187                 {
188                     spd_l = 0;
189                     spd_r = default_speed;
190                 }
191                 else
192                 {
```

```
                spd_l = default_speed;
                spd_r = 0;
            }
            // 只有急转方向的另一侧检测到黑线，才结束转弯
            if ((sharp_left_turn && right_black) ||
(!sharp_left_turn && left_black))
            {
                trace_state = TRACE_SHARP_ADJUST;
                state_counter = 0;
            }
            break;

        case TRACE_SHARP_ADJUST:
            // 急转弯后微调

            if (left_black && right_black)
            {
                // 同时检测到黑：可能交叉口或终点
                trace_state = TRACE_CROSS_FORWARD;
                state_counter = 0;
                // 前行短距：维持前进
                spd_l = spd_r = default_speed;
            }
            else if (state_counter < SHARP_ADJUST_DELAY_MS)
            {
                // 微调：相反轮短时转动以校正姿态 {
                if (sharp_left_turn)
                {
                    if (left_black && !right_black)
                    {
                        // 仅左检测到黑：急转弯开始
                        sharp_left_turn = true;
                        trace_state = TRACE_SHARP_FORWARD;
                        state_counter = 0;
                        spd_l = spd_r = default_speed;
```

```
                    break;
                }

                spd_l = default_speed;
                spd_r = 0;
            }
            else
            {
                if (right_black && !left_black)
                {
                    // 仅右检测到黑：急转弯开始
                    sharp_left_turn = false;
                    trace_state = TRACE_SHARP_FORWARD;
                    state_counter = 0;
                    spd_l = spd_r = default_speed;
                    break;
                }

                spd_l = 0;
                spd_r = default_speed;
            }
            state_counter++;
        }
        else
        {
            // 调整完成，返回正常巡线
            trace_state = TRACE_NORMAL;
        }
        break;
    }

    // 3) 更新全局速度，由主循环调用 set_car_speed
    g_car_speed_left = spd_l;
    g_car_speed_right = spd_r;
```

```
261
262             set_car_speed(g_car_speed_left, g_car_speed_right);
263             osDelay(1); // 延时，避免过快循环
264         }
265         // 走到这里说明模式退出，循环回去等待下一次进入
266     }
267 }
268
269 // 普通线程里做测距 —— 避免在回调/中断上下文里调用 GetDistance()
270 static void UltrasonicTask(void *arg)
271 {
272     (void)arg;
273     while (1)
274     {
275         // 这是在任务上下文里调用，不会触发 LOS_SemPend 警告
276         g_distance = GetDistance(); // 来自
robot_hcsr04.c :contentReference[oaicite:0]{index=0}
277         osDelay(50);
278     }
279 }
```

3.4 细节功能

除了以上的逻辑实现，我们的小车还实现了一些细节上的功能：

可以使用按键 S2 控制小车的速度，在 60~100 切换；OLED 上显示小车的运动状态（速度，以及运动方向：forward，left，right，backward 或者 stop）

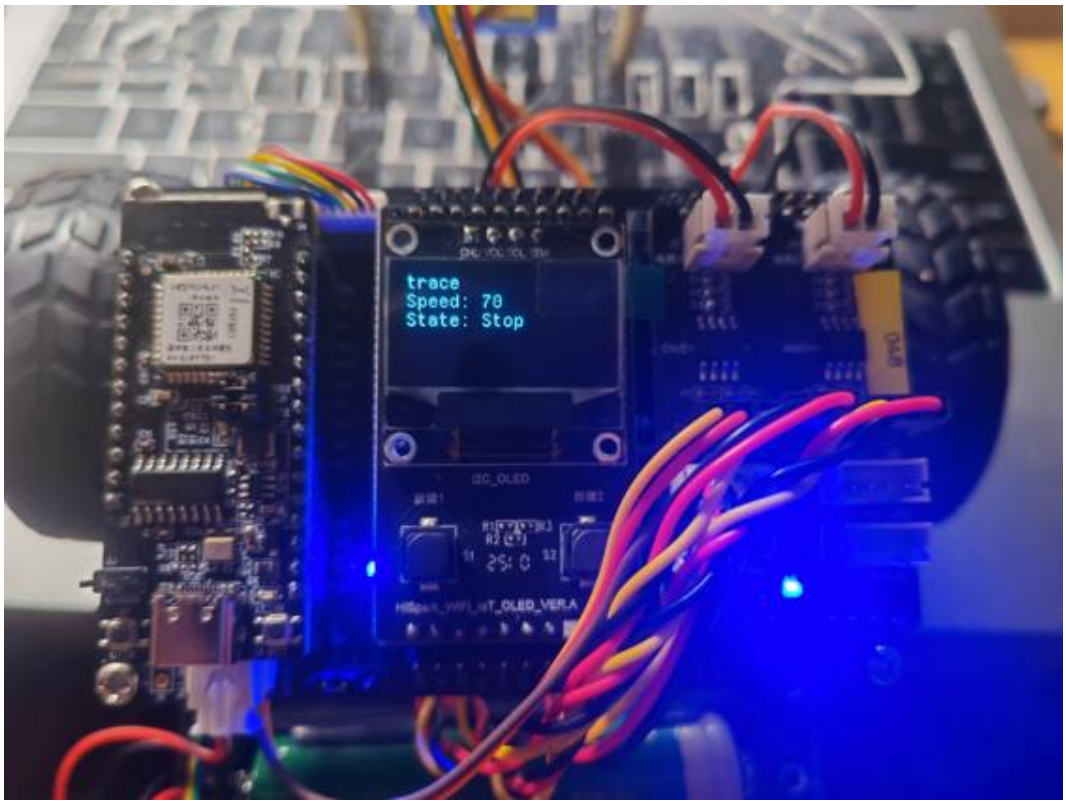

图8：屏幕上显示小车运动状态

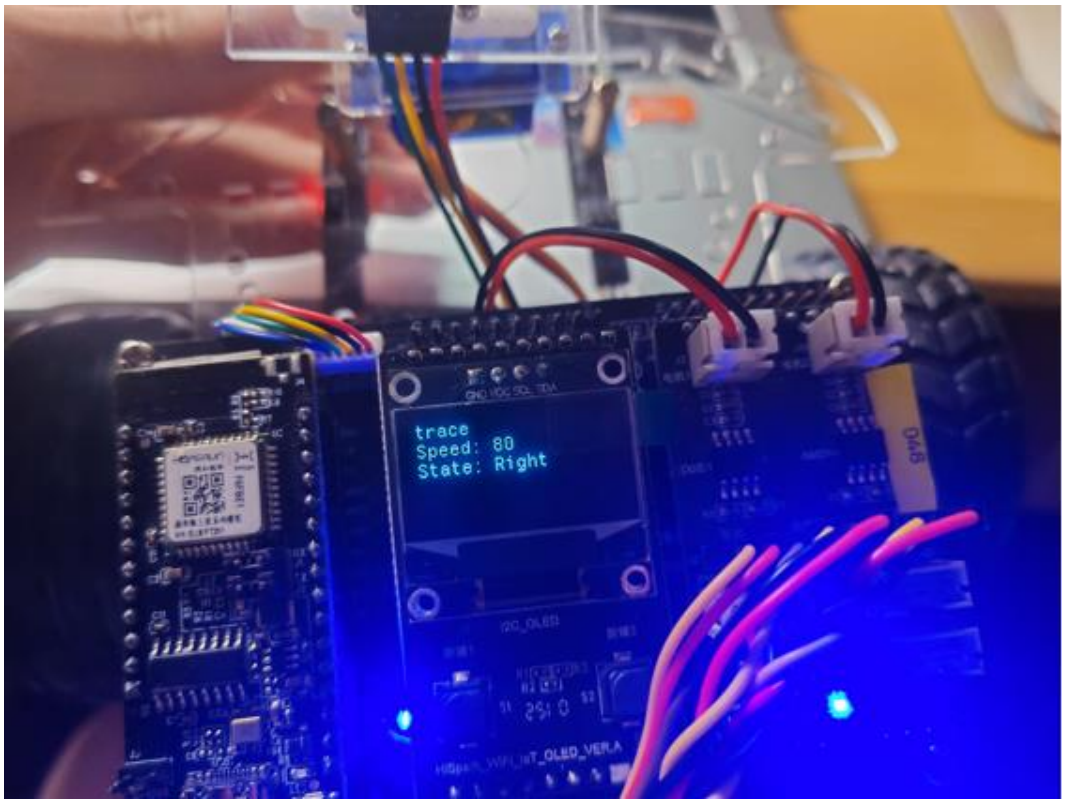

图9：屏幕上显示小车运动状态，以及速度可以切换

3.5 实验心得

通过本次实验，我收获良多，主要列为以下几点：

模块化设计是复杂系统的基石。无论是硬件的组装 还是软件的编码，本项目都体现了清晰的模块化思想。从电机驱动、舵机控制到传感器数据读取，每个功能都被封装在独立的模块或函数中 。这种设计不仅使得分步测试成为可能，也大大降低了主控制逻辑的复杂性，提高了代码的可读性和可维护性。

状态机是嵌入式系统逻辑控制的灵魂。面对循迹、避障、遥控等多种互斥的工作模式，一个清晰、健壮的状态机是保证程序正确运行的关键。本项目通过一个简单的主状态机（g_car_status）和一个复杂的循迹内部状态机（TraceState），很好地解决了模式切换和复杂逻辑判断的难题。

异步与并发编程的实际应用。为了提高系统的响应效率，项目中大量运用了多线程和中断。例如，将耗时的超声波测距放在一个独立的 UltrasonicTask 中持续进行，主控制逻辑只需读取全局变量 g_distance 即可获得最新数据，避免了阻塞。通过 GPIO 中断来响应按键的模式切换，也比轮询方式更加高效和及时。

理论与实践的深度融合。这个项目是前面所有基础实验的升华。从最基础的 GPIO（传感器状态读取）、ADC（按键模式切换），到 PWM（舵机控制与电机调速），再到网络编程（Wi-Fi 连接与 TCP Socket 通信），以及应用层协议（MQTT/TCP 自定义协议），各种知识点在这里汇集，共同构建了一个功能完整的智能系统。这让我深刻体会到，扎实的理论基础是解决复杂工程问题的根本。

调试与优化的重要性。从代码中精细的延时参数（如 CROSS_FORWARD_DELAY_MS）和复杂的循迹判断逻辑可以看出，一个能在真实赛道上稳定运行的小车，绝非一蹴而就。它必然经历了大量的测试、参数调优和算法迭代。这让我认识到，在嵌入式开发中，耐心细致的调试和对物理世界不确定性的妥善处理，与编码本身同等重要。

四、 智能小车创新部分

本次鸿蒙智能小车，我们主要实现了两个创新点：

1、win 端 WiFi 连接 -- 远程控制小车行动状态

2、单独的避障模式，在避障模式中，小车监测到前方 20cm 有障碍物， 会进行以下动作：

后退一定距离，然后转动舵机，使用超声波模块依次探测左方最近障碍物的距离和右侧最近障碍物的距离，然后找出左拐 / 右拐的最优路线，转向后再继续行驶。

4.1 远程控制

我们将开发板连接上 WiFi，使用 python 脚本作为中转，通过 mqtt 向小车发布指令，小车接收到指令后，做出与指令相匹配的动作（前进，后退，左转，右转，强制停止，调节速度）。

```python
self.forward_btn = tk.Button(control_frame, text="前进", **btn_config,
self.forward_btn.grid(row=0, column=1, padx=5, pady=5, sticky=tk.NSEW)
self.forward_btn.bind("<ButtonPress-1>", lambda e: self.send_move_comma
self.forward_btn.bind("<ButtonRelease-1>", lambda e: self.send_command(

self.backward_btn = tk.Button(control_frame, text="后退", **btn_config,
self.backward_btn.grid(row=2, column=1, padx=5, pady=5, sticky=tk.NSEW)
self.backward_btn.bind("<ButtonPress-1>", lambda e: self.send_move_comm
self.backward_btn.bind("<ButtonRelease-1>", lambda e: self.send_command

self.left_btn = tk.Button(control_frame, text="左转", **btn_config, **d
self.left_btn.grid(row=1, column=0, padx=5, pady=5, sticky=tk.NSEW)
self.left_btn.bind("<ButtonPress-1>", lambda e: self.send_move_command(
self.left_btn.bind("<ButtonRelease-1>", lambda e: self.send_command("S"

self.right_btn = tk.Button(control_frame, text="右转", **btn_config, **
self.right_btn.grid(row=1, column=2, padx=5, pady=5, sticky=tk.NSEW)
self.right_btn.bind("<ButtonPress-1>", lambda e: self.send_move_command
self.right_btn.bind("<ButtonRelease-1>", lambda e: self.send_command("S

self.stop_btn = tk.Button(control_frame, text="停止", **btn_config, **s
self.stop_btn.grid(row=1, column=1, padx=5, pady=5, sticky=tk.NSEW)
self.stop_btn.bind("<ButtonPress-1>", lambda e: self.send_command("S"))
```

```c
switch (command) {
    case 'F': // 前进
        printf("car_forward with speed %u\n", speed);
        car_forward(speed);
        break;
    case 'B': // 后退
        printf("car_backward with speed %u\n", speed);
        car_backward(speed);
        break;
    case 'L': // 左转
        printf("car_left with speed %u\n", speed);
        car_left(speed);
        break;
    case 'R': // 右转
        printf("car_right with speed %u\n", speed);
        car_right(speed);
        break;
    case 'S': // 停止
    default:
        printf("car_stop\n");
        car_stop();
        break;
}
```

图10：前端向小车发布指令，小车接收指令并做出相应动作

图11：小车前端页面展示

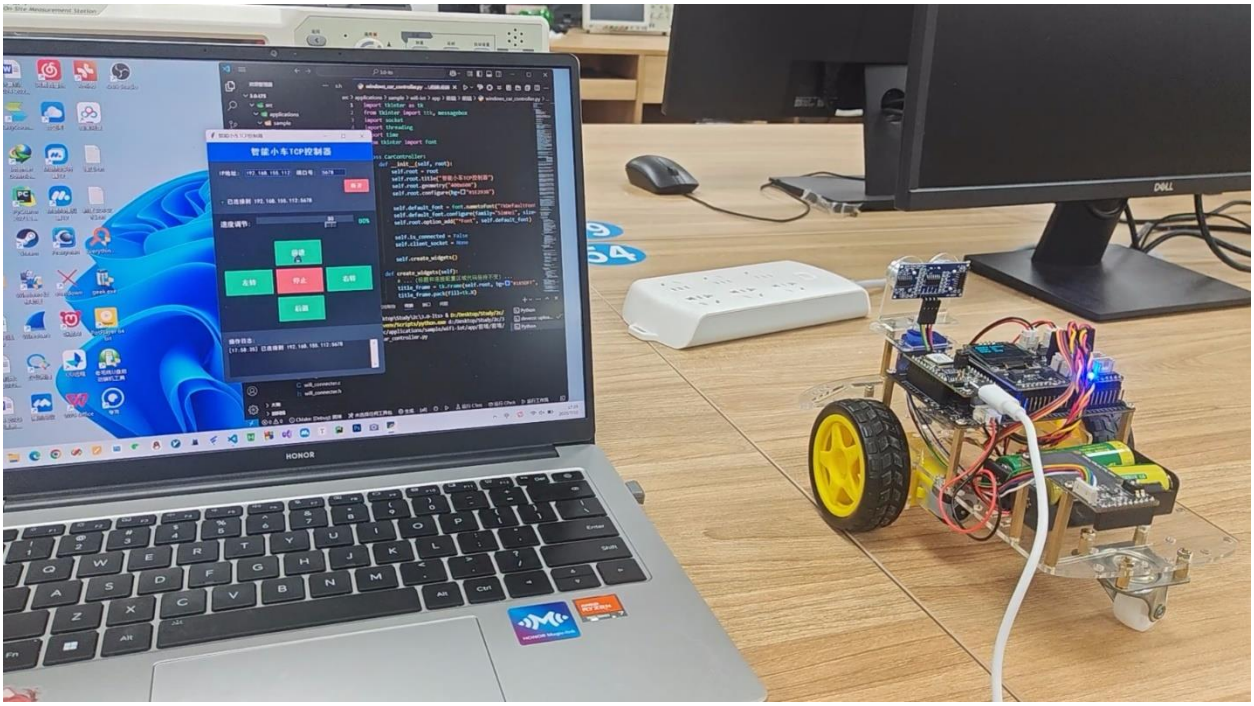

图12：小车远程控制模式展示

具体实验视频请见“实验记录”文件夹“远程控制”视频。

具体代码这里不作赘述，可在“源代码”文件夹下找到 robot_demo 项目查看。远控、WiFi 连接等通过不同的类实现，并在远程控制模式统一调用。

具体工作模式：

网络连接：进入远程控制模式后，remote_control_module 函数首先调用 wifi_connecter.c 中的接口，使小车连接到预设的 Wi-Fi 热点。

TCP 服务器：连接成功后，小车启动一个 TCP 服务器，并监听指定端口，等待 PC 客户端的连接。

指令解析：服务器接受连接后，会循环接收客户端发来的指令字符串。它能解析单个字符（如'F'代表前进，'S'代表停止）和带速度的指令（如"F80"代表以 80%的速度前进），实现对小车方向和速度的精确控制。

4.2 单独避障模式

在单独的避障模式中，car_mode_control_func 函数被调用。小车会持续前进，直到前方距离小于 20cm。

触发避障后，小车会先停止、短暂后退，然后控制舵机左右转动，分别测量两侧的距离，选择更开阔的方向进行转向，从而绕开障碍物。小车的每一步计算通过贪心算法，得出每一次避障的最优方向，从而达到“走得尽量远”的效果。

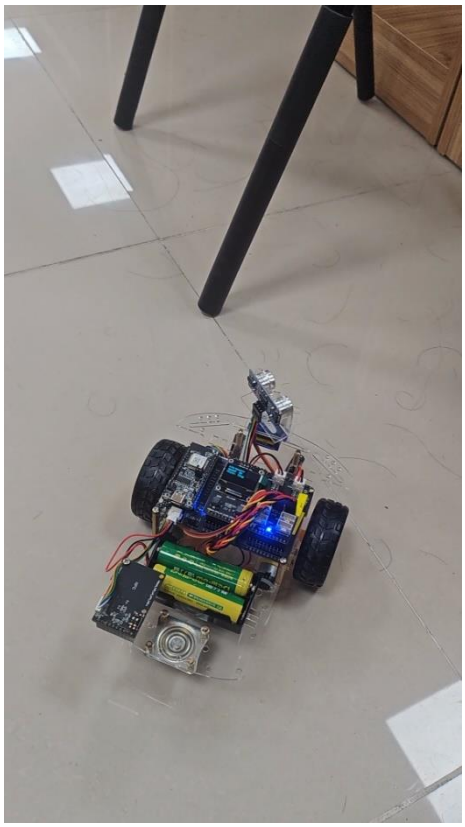

图13：小车避障模式展示

具体实验视频请见“实验记录”文件夹“避障模式”视频。

该避障模式复用了小车在循迹模式“检测障碍物停下”的代码，这里对实验现象一并作展示：

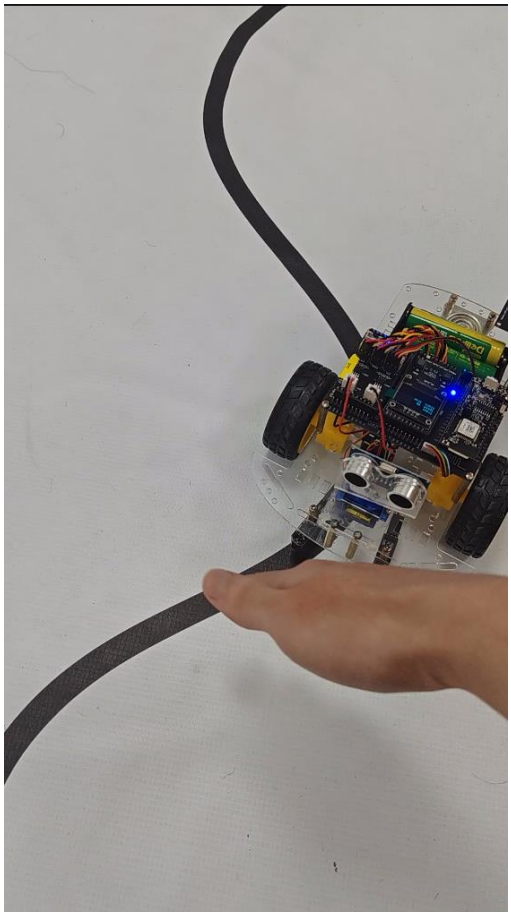

图14：小车循迹模式下的避障功能展示

具体实验视频请见“实验记录”文件夹“循迹-避障”视频。

4.3 心得体会

在完成鸿蒙智能小车的循迹、避障、状态显示等基础功能后，我们并未止步于此，而是着力于实现两个更具挑战性和实用价值的创新点：基于 Wi-Fi 的远程遥控系统和具备决策能力的智能避障模式。完成这两个功能的过程，不仅是对我们技术能力的深度考验，更引发了我们对于物联网（IoT）交互和机器人自主性设计的深刻思考。

一、 从本地到云端：Wi-Fi 远程控制

将远程控制功能集成到小车中，是从“自动化”迈向“智能化与网络化”的关键一步。这次实践让我深刻体会到以下几点：

分层网络架构的力量：我们实现的远程控制功能，本质上是一个迷你的 C/S（客户端/服务器）架构。robot_control.c 中的 remote_control_module 函数作为总调度，首先调用 wifi_connecter.c 模块完成网络接入，然后启动 car_tcp_server_test.c 中的 TCP 服务器。这种**“接入层”与“应用层”分离**的设计思路，使得代码逻辑非常清晰。网络连接的复杂过程被封装，应用层只需关心指令的接收和解析，这正是现代网络软件设计的精髓所在，大大提高了代码的可维护性和扩展性。

协议是设备间沟通的“语言”：在 car_tcp_server_test.c 中，我们定义了一套简单但高效的应用层通信协议，例如用'F'代表前进，'S'代表停止，并能通过"F80"这样的格式实现对速度的精确控制。这个过程让我切身体会到，协议是实现设备互联互通的灵魂。无论是简单的自定义 TCP 协议，还是通过 Python 脚本和 MQTT 进行中转，其核心都是在定义一套双方都能理解的“语言”，从而实现信息的准确传递和命令的有效执行。

异步编程是 IoT 应用的常态：Wi-Fi 连接本身是一个需要时间等待的异步过程。在 wifi_connecter.c 中，通过注册回调函数和循环等待标志位的方式，优雅地处理了这种异步性，避免了主线程的阻塞。这让我认识到，在物联网应用中，设备状态的改变（如网络连接、传感器触发）往往是不可预测的，熟练运用回调、多线程、事件驱动等异步编程模型，是开发高响应、高效率嵌入式系统的必备技能。

二、 从反应到决策：智能避障策略

如果说循迹是让小车“遵守规则”，那么我们设计的独立避障模式，则是赋予了小车**“自主决策”的能力**。这部分的设计带给我最大的启发是：

“感知-思考-行动”（Sense-Think-Act）模型的实践：我们的避障算法完美地诠释了这一经典的人工智能模型。

感知 (Sense)：不仅仅是使用超声波模块被动地“看到”前方的障碍物，我们更进一步，通过控制 SG90 舵机左右转动，主动地扩展了小车的感知范围，获取了更全面的环境信息。

思考 (Think)：这是创新的核心。小车在检测到障碍物后，并非盲目转向，而是通过比较左右两侧的探测距离，执行一次简单的**“贪心算法”决策**——选择当前看起来最开阔、最安全的路径。这虽然是一个简单的决策模型，但它标志着小车从一个纯粹的“反应式”机器，向一个能够进行简单**“规划”**的智能体转变。

行动 (Act)：基于“思考”得出的最优方向，小车精确地执行转向和前进的动作，完成一次完整的、智能的避障流程。

软件设计对机器人行为的决定性作用：robot_control.c 中的 car_where_to_go 函数，是这一智能行为的软件实现核心。它将后退、舵机扫描、比较距离、转向等一系列原子操作，有机地组合成一个高级的避障策略。这让我深刻认识到，机器人的“智能”程度，很大程度上取决于软件算法的设计高度。精巧的算法可以将有限的硬件能力发挥到极致，创造出远超预期的复杂行为。

总结而言，这两个创新点的实现过程，是本次实验中最具价值和挑战性的部分。远程控制让我们跨越了物理的界限，将物联网的交互能力赋予了小车；而智能避障策略则让我们初探了人工智能与机器人技术的门径，体验了赋予机器“思考”能力的乐趣。这次经历不仅锻炼了我们的编程和调试能力，更重要的是，培养了我们进行系统设计、协议定义和算法构思的工程思维。

五、 课程改进建议

5.1 课程内容方面

课程在 Wi-Fi 连接实验后，直接进入了基于 Paho-MQTT 库的应用层协议实验。

建议：在两者之间，可以加入一个更基础的**“TCP Socket 编程”**实验。例如，在开发板连接 Wi-Fi 后，让它作为一个简单的 TCP 客户端，去连接 PC 上的 TCP 服务器，并收发简单的字符串。这能让我们学生更直观地理解 MQTT 所依赖的底层 Socket 通信原理，避免将 Paho-MQTT 库视为一个“黑盒”，从而更深刻地理解网络分层模型。

5.2 项目与创新拓展方面

增加数据持久化元素：

我们大家目前在小车的设置（如速度、运动状态等）是硬编码或临时存储在内存中的，断电即丢失。

建议：可以设置一个拓展任务，要求学生利用 Hi3861 的 Flash 非易失性存储功能，将用户的设置（如调好的车速、Wi-Fi 密码等）保存下来。这能让我们接触到嵌入式设备中一个非常实用的功能——掉电保存。

深化双向遥测与控制：

目前，我们大多数学生在远程控制功能中实现了 PC 向小车发布命令。

建议：可以将其设计为一个更完整的“数字孪生”创新任务。要求小车不仅接收控制指令，还要周期性地通过 MQTT 向特定主题发布自己的完整状态（如：{ "mode": "tracking", "speed": 80, "obstacle_dist": 35.5, "state": "stop" }）。PC 端的控制软件则订阅该主题，实时地、图形化地展示小车的“数字镜像”。这将极大地提升项目的趣味性和技术深度。

5.3 常见问题与调试指南

学生在实践中难免遇到各种问题，我建议为每个实验（尤其是复杂的小车项目）配套一份简短的“FAQ”或“Debugging Guide”。例如，“电机不转怎么办？”（检查接线、L9110S 供电、PWM 引脚复用是否正确）、“Wi-Fi 连接卡住 / 失败？”（检查 SSID/密码、代码）等。这不仅能节省学生排错的时间，更能培养我们系统化的调试思维。