华为认证 HarmonyOS 系列教程

HCIA-HarmonyOS Device Developer 实验手册

学员用书

版本：1.0

华为技术有限公司

商标声明


和其他华为商标均为华为技术有限公司的商标。

本文档提及的其他所有商标或注册商标，由各自的所有人拥有。

注意

您购买的产品、服务或特性等应受华为公司商业合同和条款的约束，本文档中描述的全部或部分产品、服务或特性可能不在您的购买或使用范围之内。除非合同另有约定，华为公司对本文档内容不做任何明示或暗示的声明或保证。

由于产品版本升级或其他原因，本文档内容会不定期进行更新。除非另有约定，本文档仅作为使用指导，本文档中的所有陈述、信息和建议不构成任何明示或暗示的担保。

华为技术有限公司

地址： 深圳市龙岗区坂田华为总部办公楼 邮编：518129

网址： http://e.huawei.com

华为认证体系介绍

华为认证是华为公司基于“平台+生态”战略，围绕“云-管-端”协同的新ICT技术架构，打造的覆盖ICT（Information and Communications Technology 信息通信技术）全技术领域的认证体系，包含ICT技术架构与应用认证、云服务与平台认证两类认证。

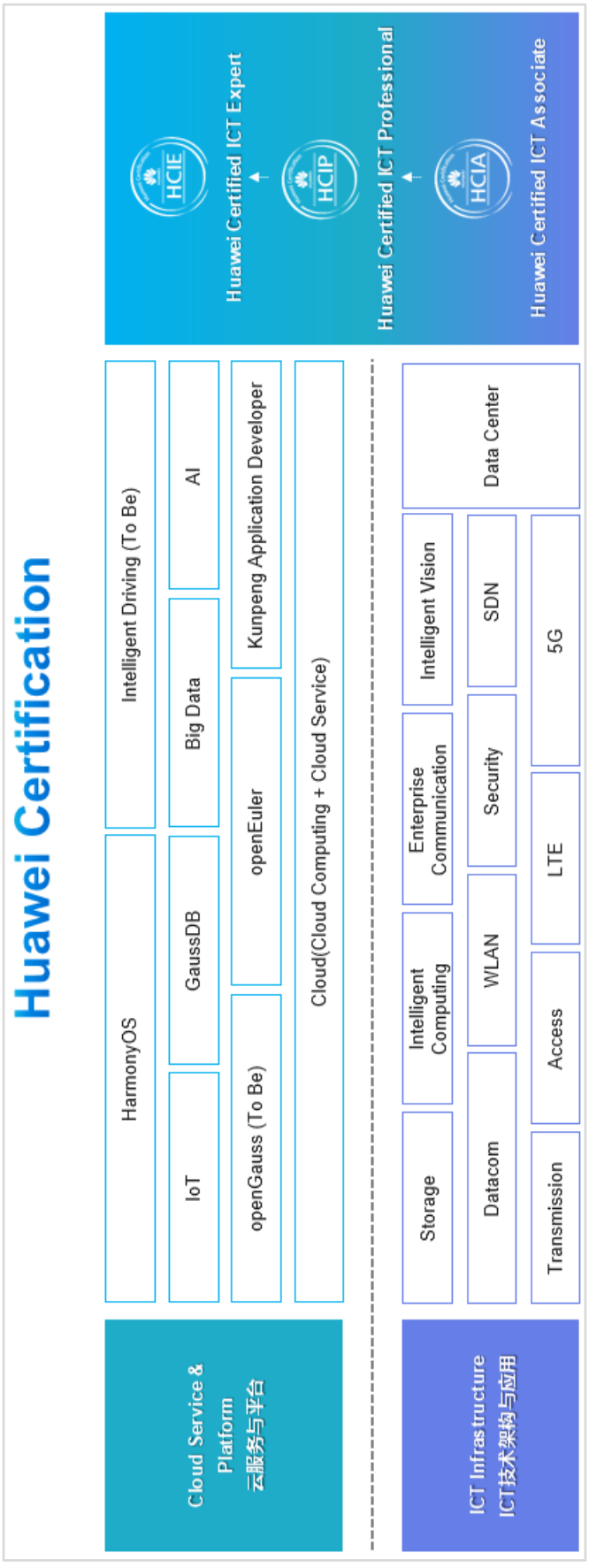
根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

华为认证覆盖ICT全领域，符合ICT融合的技术趋势，致力于提供领先的人才培养体系和认证标准，培养数字化时代新型ICT人才，构建良性ICT人才生态。

HCIA-HarmonyOS Device Developer V1.0定位于培养基于HarmonyOS设备开发场景具备专业知识和技能水平的工程师。

通过HCIA-HarmonyOS Device Developer V1.0认证，您将掌握HarmonyOS基本概念及原理、HarmonyOS技术架构、HarmonyOS设备开发流程、内核与驱动知识，具备HarmonyOS设备子系统开发、移植的能力，能够胜任HarmonyOS设备开发工程师岗位。

华为认证协助您打开行业之窗，开启改变之门，屹立在HarmonyOS行业的潮头浪尖！


Huawei Certification
Huawei Certified ICT Expert
Huawei Certified ICT Professional
Huawei Certified ICT Associate
HCIE
HCIP
HCIA
Cloud Service & Platform
云服务与平台
ICT Infrastructure
ICT技术架构与应用
HarmonyOS
Intelligent Driving (To Be)
IoT
GaussDB
Big Data
AI
openGauss (To Be)
openEuler
Kunpeng Application Developer
Cloud(Cloud Computing + Cloud Service)
Storage
Intelligent Computing
Enterprise Communication
Intelligent Vision
Data Center
Datacom
WLAN
Security
SDN
Transmission
Access
LTE
5G

前 言

简介

本书为 HCIA-HarmonyOS Device Developer 认证培训教程，适用于准备参加 HCIA-HarmonyOS Device Developer 考试的学员或者希望了解 HarmonyOS 基础知识、HarmonyOS 技术架构、HarmonyOS 设备开发流程、HarmonyOS 内核基础开发、HarmonyOS 驱动开发、HarmonyOS 子系统开发、HarmonyOS 移植开发，具备 HarmonyOS 设备功能开发、调试与烧写的能力等相关 HarmonyOS 技术的读者。

内容描述

本实验指导书共包含 5 个实验，分别为 HarmonyOS 设备开发内核基础实验、驱动基础实验、子系统基础实验、移植实验等，最后通过一个综合实验来将知识点串联起来。

- 实验一为 HarmonyOS 设备开发内核基础实验，通过一些演练场景，如生产者消费者、打印机的使用、信息传递、定时投喂等场景，理解 HarmonyOS 设备开发内核基础中的信号量、互斥锁、消息队列、事件管理、软件定时器等内核基础开发知识。
- 实验二为 HarmonyOS 设备开发驱动基础实验，通过一些演练场景，如路灯控制、呼吸灯的使用、光照感应、环境监测等场景，理解 HarmonyOS 设备开发驱动基础中的 GPIO、I2C、UART、PWM、ADC 等驱动基础开发知识。
- 实验三为 HarmonyOS 设备开发内核子系统实验，通过一些演练场景，如 WiFi 账号密码管理、文件操作、WiFi 操作、网络通信等场景，理解 HarmonyOS 设备开发内核基础中的 KV 存储、文件操作、WiFi 操作等内核子系统开发知识。
- 实验四为 HarmonyOS 设备开发移植实验，通过一些演练场景，如三方库的移植、编译和使用，理解 HarmonyOS 设备开发移植中的 gn 脚本文件，代码结构等移植开发知识。
- 实验五为综合实验，通过三个的案例实验，包含内核基础、驱动开发、子系统开发等知识点，帮助读者将技术点串起来，并掌握 HarmonyOS 设备开发流程和关键技术点。

读者知识背景

本课程为华为认证基础课程，为了更好地掌握本书内容，阅读本书的读者应首先具备以下基本条件：

- 具有基本的代码编程能力，同时熟悉 C 语言，了解基本设备开发知识。

实验环境说明

设备介绍

为了满足 HCIA-HarmonyOS Device Developer 实验需要，建议每套实验环境采用以下配置：

设备名称、型号与版本的对应关系如下：

设备名称	设备型号	软件版本
PC机	Windows系统	Windows 10 64位
Hi3861开发板	HiSpark	HarmonyOS 1.1.0

准备实验环境

检查设备

实验开始之前请每组学员检查自己的实验设备是否齐全，实验清单如下。

设备名称	数量	备注
笔记本或台式机	每组 1 台	台式机要有无线网卡

每组检查自己的设备列表如下：

- 笔记本或台式机 1 台。

参考资料及工具

文档中所列出的命令以及参考文档，请根据实际环境中的不同产品版本使用对应的命令以及文档。

参考文档：

1. 《 HarmonyOS 官方文档 》，获取地址：
https://developer.harmonyos.com/cn/documentation

2. 《 HarmonyOS Device 官方文档 》，获取地址：

https://device.harmonyos.com/cn/home/

软件工具：

编号	工具名称	版本
1	Visual Studio Code	V1.55.2
2	HiBurn	V2.2

3	DevEco Device Tool	V2.1
4	Oracle VM VirtualBox	V6.1.22

目录

1 内核开发入门

1.1 课程介绍

本文将为大家介绍 HarmonyOS 的 LiteOS-M 内核中的线程、网络、信号量、互斥锁、消息队列、软件定时器、事件管理等基础功能，包括一些基础概念、实现步骤和使用场景等，供想要深入了解 HarmonyOS 操作系统的初学者学习参考。

1.2 教学目标

- 能够掌握使用多线程编程；
- 能够掌握信号量、互斥锁、消息队列、事件管理等内核基础单元编程；
- 能够掌握使用网络编程。

1.3 案例背景

说明：本文所涉及的案例仅为样例，实际操作中请以真实设备环境为准，具体配置步骤请参考对应的产品文档。

某公司需要开发一款设备，但是目前设备开发系统多样化，接口没有统一，导致代码移植相对复杂，几乎是推倒重来，那么统一的内核，标准的接口变得尤为重要，为了学习内核基础编程，课程采用案例化的方式讲述以下知识点（演练场景中体会）：

- 多线程编程；
- 信号量；
- 互斥锁；
- 消息队列；
- 软件定时器；
- 事件管理。

1.4 演练任务

1.4.1 演练场景 1：生产者消费者

背景

生产者和消费者模式在生活中随处可见，描述的是协调与协作的关系。比如 A 正在准备食物（生产者），而 B 正在食用（消费者），使用一个共用的桌子用于放置盘子和取走盘子，生产者 A 准备食物，如果桌子上的盘子已经满了就需要等待，反之对于消费者 B，如果桌子上的盘子空了就需要等待。这里桌子就是一个共享的对象。

思考

上述例子中，用到了内核基础中的什么机制?

任务一　创建 HarmonyOS 的第一个程序

在开始生产者、消费者场景演练之前，我们必须要清楚：如何创建一个 HarmonyOS 设备程序?

本任务将演示如何编写简单业务，输出“Hello World”，初步了解 HarmonyOS 如何运行在开发板上。

打开 DevEco Device Tool 工程，该过程在《HCIA-HarmonyOS Device Developer 环境搭建指南》中已经描述，请参考其中章节，这里不再赘述。

步骤 1　新建目录

开发者编写业务时，务必先在./applications/sample/wifi-iot/app 路径下新建一个目录（或一套目录结构），用于存放业务源码文件。

例如：在 app 下新增业务 my_first_app，其中 hello_world.c 为业务代码，BUILD.gn 为编译脚本，具体规划目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├── my_first_app
                │   ├── hello_world.c
                │   └── BUILD.gn
```

```
└── BUILD.gn
```

步骤 2 编写业务代码

新建./applications/sample/wifi-iot/app/my_first_app 下的 hello_world.c 文件，在 hello_world.c 中新建业务入口函数 HelloWorld，并实现业务逻辑。在代码最下方，使用 HarmonyOS 启动恢复模块接口 APP_FEATURE_INIT()启动业务，APP_FEATURE_INIT 定义在 ohos_init.h 文件中。

```
#include <stdio.h>
#include "ohos_init.h"
#include "ohos_types.h"

void HelloWorld(void)
{
    /*作为第一个 HarmonyOS 程序的开始，打印 HelloWorld，开始学习 HarmonyOS 设备开发*/
    printf("[DEMO] Hello world.\n");
}
APP_FEATURE_INIT(HelloWorld);
```

步骤 3 编写 BUILD.gn 文件，用于将业务构建成静态库

新建./applications/sample/wifi-iot/app/my_first_app 下的 BUILD.gn 文件，并完成如下配置。

BUILD.gn 文件由三部分内容（目标、源文件、头文件路径）构成，需由开发者完成填写。

```
static_library("myapp") {
    sources = [
        "hello_world.c"
    ]
    include_dirs = [
        "//utils/native/lite/include"
    ]
}
```

static_library 中指定业务模块的编译结果，为静态库文件 libmyapp.a，开发者根据实际情况完成填写。

sources 中指定静态库.a 所依赖的.c 文件及其路径，若路径中包含“//”则表示绝对路径（此处为代码根路径），若不包含“//”则表示相对路径。

include_dirs 中指定 source 所需要依赖的.h 文件路径。

步骤 4 编写模块 BUILD.gn 文件，指定需参与构建的特性模块

配置./applications/sample/wifi-iot/app/BUILD.gn 文件，在 features 字段中增加索引，使目标模块参与编译。features 字段指定业务模块的路径和目标，以 my_first_app 举例，features 字段配置如下。

```
import("//build/lite/config/component/lite_component.gni")
```

```
lite_component("app") {
    features = [
        "my_first_app:myapp",
    ]
}
```

my_first_app 是相对路径指向./applications/sample/wifiiot/app/my_first_app/BUILD.gn。

myapp 是目标，指向./applications/sample/wifi-iot/app/my_first_app/BUILD.gn 中的 static_library("myapp")。

步骤 5 编译源码

在源码路径下执行 hb set，选择路径为当前目录。

product 为 wifiiot_hispark_pegasus@hisilicon

```
hb build -b release -f
```

-b release 表示不编译 XTS 测试，编译过程会更快，程序启动后不会运行 XTS 相关内容。

-f 表示全量编译，当更改 BUILD.gn 时，需要执行-f，将所有代码重新编译。

执行编译命令后，如下所示表示编译成功。

```
 [OHOS INFO] [11/20] ACTION //test/xts/acts/utils_lite/utilsfile_hal:ActsUtilsFileTest(//build/lite/toolchain:riscv32-unknown-elf)
[OHOS INFO] [12/15] ACTION //build/lite:gen_rootfs(//build/lite/toolchain:riscv32-unknown-elf)
[OHOS INFO] [13/14] ACTION //device/hisilicon/hispark_pegasus/sdk_liteos:run_WiFiiot_scons(//build/lite/toolchain:riscv32-unknown-elf)
[OHOS INFO] wifiiot_hispark_pegasus build success
```

步骤 6 烧写固件

过程不再赘述，请参考《HCIA-HarmonyOS Device Developer 环境搭建指南》。

```
Processing hi3861 (platform: hisilicon; board: hi3861; framework: ohos-sources)
-------------------------------------------------------------------------------------------------------------------------
Verbose mode can be enabled via `-v, --verbose` option
Configuring upload protocol...
AVAILABLE: burn-serial
CURRENT: upload_protocol = burn-serial
Uploading with BurnSerial
upload args: ['hiburn.exe', '-com:3', '-bin:Z:\\L01\\out\\hispark_pegasus\\wifiiot_hispark_pegasus\\Hi3861_wifiiot_app_allinone.bin', '-signalbaud:921600']
Connecting, please reset device...


Ready to load at 0x10A000
CCStartBurn
total size:0x3C00
loady succ
Entry loader
```

```
=============================================

erase flash 0x0 0x200000
Ready for download
CC
CStartBurn
total size:0x11E030

Execution Successful
=============================================

erase flash 0x1FA000 0x6000
Ready for download
CCCtotal size:0x6000

Execution Successful
================================================= [SUCCESS] Took 39.96 seconds
=============================================
```

步骤 7 验证

```
The VersionID is [****/****/****/****/OHOS/****/****/5/OpenHarmony 1.0/debug]
The buildType is [debug]
The buildUser is [jenkins]
The buildHost is [linux]
The buildTime is [1618186527650]
The BuildRootHash is []
******To Obtain Product Params End   ******
[DEMO] Hello world.
```

问题研讨

任务一中，HarmonyOS 源码中是如何运行我们指定的 app?

任务二 创建生产者消费者线程

步骤 1 新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 producer_consumer，将框架代码中 producer_consumer 中的文件拷贝到工程里，如下图所示。

```
.
```

```
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──producer_consumer
                │   ├──producer_consumer.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2 编写 producer_consumer 目录下的 BUILD.gn

自行填入源文件“producer_consumer.c”；

```
static_library("producer_consumer") {
    sources = [
        "producer_consumer.c"
    ]
    include_dirs = [
        "//utils/native/lite/include"
    ]
}
```

步骤 3 修改 app 目录下的 BUILD.gn

自行修改 features 字段中的 producer_consumer:producer_consumer。第一个 producer_consumer 指得是目录，第二个 producer_consumer 是指的是上面的静态库名称 producer_consumer；

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "producer_consumer:producer_consumer",
    ]
}
```

步骤 4 编写生产者消费者代码

关键代码参考如下;

```
/*创建生产者线程，取线程名字为 producer1，学员自行补充*/
osThreadId_t ptid1 = newThread("producer1", producer_thread, NULL);
/*创建生产者线程，取线程名字为 consumer1，学员自行补充*/
osThreadId_t ctid1 = newThread("consumer1", consumer_thread, NULL);
/*运行 500ms*/
osDelay(50);
/*终止生产者线程，学员自行补充*/
osThreadTerminate(ptid1);
/*终止消费者线程，学员自行补充*/
osThreadTerminate(ctid1);
```

步骤 5 编译烧录程序

查看运行结果，执行结果如下。

```
[consumer_thread]consumer1 consumes a product.
[producer_thread]producer1 produces a product.
[consumer_thread]consumer1 consumes a product.
[consumer_thread]consumer1 consumes a product.
[producer_thread]producer1 produces a product.
[consumer_thread]consumer1 consumes a product.
[producer_thread]producer1 produces a product.
[consumer_thread]consumer1 consumes a product.
[producer_thread]producer1 produces a product.
```

问题研讨

线程执行的顺序是否符合预期，如何保证按照预期的顺序来执行线程?

任务三 生产者消费者的同步

在任务二的代码的基础上进行扩展，使用信号量来保证生产者消费者线程之间的同步。

将框架代码中该任务目录下 producer_consumer 中的文件拷贝到工程里。

定义两个信号量，第一个信号量（empty_id）对可用（空）缓冲区进行倒计数，即生产者线程可以通过从这个缓冲区获取可用缓冲区时隙，第二个信号量（filled_id）对使用过的（填充的）缓冲区进行计数，即消费者线程可以通过从这个缓冲区获取可用数据来等待用于生产者消费者的同步，关键代码（关键代码并非完整代码）如下：

```
    /*线程在给定序列中获取和释放两个信号量的正确行为是至关重要的*/
void producer_thread(void *arg) {
    (void)arg;
    while(1) {
        /*如果没有令牌可用，则获取信号量标记或超时，学员自行补充*/
        osSemaphoreAcquire(empty_id, osWaitForever);
        product_number++;
        /*生产者线程，打印信息，证明生产者线程正在被执行*/
        printf("[producer_thread]%s produces a product, now product number: %d.\r\n", osThreadGetNam
e(osThreadGetId()), product_number);
        osDelay(4);
        /*释放一个信号量令牌直到最初的最大数量，学员自行补充*/
        osSemaphoreRelease(filled_id);
    }
}
```

```
void consumer_thread(void *arg) {
    (void)arg;
    while(1){
        /*如果没有令牌可用，则获取信号量标记或超时，学员自行补充*/
        osSemaphoreAcquire(filled_id, osWaitForever);
        product_number--;
        /*消费者线程，打印信息，证明消费者线程正在被执行*/
        printf("[consumer_thread]%s consumes a product, now product number: %d.\r\n", osThreadGetName(osThreadGetId()), product_number);
        osDelay(3);
        /*释放一个信号量令牌直到最初的最大数量，学员自行补充*/
        osSemaphoreRelease(empty_id);
    }
}
```

可以看到，创建了 3 个生产者线程，2 个消费者线程，运行结果如下：

```
[producer_thread]producer1 produces a product, now product number: 1.
[producer_thread]producer2 produces a product, now product number: 2.
[producer_thread]producer3 produces a product, now product number: 3.
[producer_thread]producer1 produces a product, now product number: 4.
[producer_thread]producer2 produces a product, now product number: 5.
[producer_thread]producer3 produces a product, now product number: 6.
[consumer_thread]consumer1 consumes a product, now product number: 5.
[consumer_thread]consumer2 consumes a product, now product number: 4.
[producer_thread]producer1 produces a product, now product number: 5.
```

分析结果可以看到，生产者确保先生成出产品，消费者才能去消费，而且由于生产者多，能保证我们的产能过剩，消费者需要的时候都能得到满足。

问题研讨

更改消费者，生产者线程数量，会有什么效果？比如 4 个消费者，1 个生产者？

1.4.2 演练场景 2：打印机的使用

背景

在日常生活工作中，打印机是常用设备，当正在使用打印机打印资料的同时（还没有打印完），其他人刚好也在此刻使用打印机打印资料，如果不做任何处理的话，打印出来的资料可能是错乱的。

思考

HarmonyOS 的 LiteOS-M 内核基础中，哪个功能保证这个资源（打印机）不被同时使用?

任务一　创建两个打印任务

在我们的实验中，我们用打印信息来模拟打印机，比如线程 1 连续打印 5 次 Hello，线程 2 连续打印 5 次 World，看看会出现什么情况。

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 printer，将框架代码中 printer 中的文件拷贝到工程里，如下图所示；

```
.
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──printer
                │   ├──printer.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 producer_consumer 目录下的 BUILD.gn

自行填入源文件“printer.c”；

```
static_library("printer") {
    sources = [
        "printer.c"
    ]
    include_dirs = [
        "//utils/native/lite/include"
    ]
}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 printer: printer。第一个 printer 指得是目录，第二个 printer 是指的是上面的静态库名称 printer；

```
import("//build/lite/config/component/lite_component.gni")
```

```
lite_component("app") {
      features = [
           "printer:printer",
      ]
}
```

步骤 4 编写模拟打印机代码

关键代码参考如下；

```
void rtosv2_printer_main(void *arg) {
    (void)arg;
    /*创建打印机 1 线程，取线程名字为 printer1，学员自行补充*/
    osThreadId_t tid1 = newThread("printer1", printer1_thread, NULL);
    /*创建打印机 2 线程，取线程名字为 printer2，学员自行补充*/
    osThreadId_t tid2 = newThread("printer2", printer2_thread, NULL);
    /*延时运行 500ms，供线程运行*/
    osDelay(50);
    /*终止打印机线程，创建了就要进行销毁，学员自行补充*/
    osThreadTerminate(tid1);
    osThreadTerminate(tid2);
}
```

步骤 5 编译烧录程序，

查看运行结果，执行结果如下。

```
Hello
World
Hello
World
Hello
World
Hello
World
Hello
World
```

问题研讨

1. 执行结果是否符合设计（预期），我们需要的是连续打印 5 次 Hello，5 次 World，线程 1 还没有用完这个“打印机”，线程 2 就想使用“打印机，这样用“打印机”是不是乱套了，那如何保证线程 1 用完后，才能让线程 2 再用，线程 2 使用完后，再让线程 1 使用?

2. 为什么在打印后面增加延时 osDelay()?

任务二　给打印机加把互斥锁

我们对任务一的代码进行优化，使用互斥锁，来保证“打印机”这个资源能够被有序的使用。

我们定义一个互斥锁，用于两个任务对资源操作的互斥，关键代码如下：

```
/*打印机 1 执行线程*/
void printer1_thread(void *arg) {
    osMutexId_t *mid = (osMutexId_t *)arg;
    while(1) {
        /*获取的互斥锁，学员自行补充互斥锁实现*/
        if (osMutexAcquire(*mid, 100) == osOK) {
            /*打印信息 Hello,延时 2 秒*/
                {...}
            /*释放互斥锁，学员自行补充*/
            osMutexRelease(*mid);
            osDelay(200);
        }
    }
}
/*打印机 2 执行线程*/
void printer2_thread(void *arg) {
    osMutexId_t *mid = (osMutexId_t *)arg;
    while(1){
        /*获取的互斥锁，学员自行补充互斥锁实现*/
        if (osMutexAcquire(*mid, 100) == osOK) {
            {...}
            /*释放互斥锁，学员自行补充*/
            osMutexRelease(*mid);
            osDelay(200);
        }
    }
}

void rtosv2_printer_main(void *arg) {
    (void)arg;
    osMutexAttr_t attr = {0};
    /*创建一个互斥锁 mid，学员自行补充*/
    osMutexId_t mid = osMutexNew(&attr);
    if (mid == NULL) {
        printf("osMutexNew, create mutex failed.\r\n");
    } else {
        printf("osMutexNew, create mutex success.\r\n");
    }
    /*创建打印机 1 线程，取线程名字为 printer1，并且传输互斥锁信息 mid，学员自行补充*/
```

```
    osThreadId_t tid1 = newThread("printer1", printer1_thread, &mid);
    /*创建打印机 2 线程，取线程名字为 printer2，并且传输互斥锁信息 mid，学员自行补充*/
    osThreadId_t tid2 = newThread("printer2", printer2_thread, &mid);
    /*延时运行 1000ms，供线程运行*/
    osDelay(100);
    /*终止打印机线程，创建了就要进行销毁，学员自行补充*/
    osThreadTerminate(tid1);
    osThreadTerminate(tid2);
    osMutexDelete(mid);
}
```

运行结果如下：

```
Hello
Hello
Hello
Hello
Hello
World
World
World
World
World
```

分析结果可以看到，线程 1 连续打印了 5 次 Hello，线程 2 连续打印了 5 次 World，符合预期，线程在执行打印（使用“打印机”）期间，锁定住了资源，独占打印机。

问题研讨

互斥锁和信号量有什么关系？

1.4.3 演练场景 3：消息传递

背景

日常生活中，我们经常需要沟通交流，交换消息，那么在 HarmonyOS 系统的任务（线程）间，是如何进行沟通交流的，是否有方法和桥梁来传递消息，从而达到信息交流的目的。

思考

线程间的数据通信是通过什么机制？

任务一　两个任务之间的消息传递

在我们的实验中，我们创建两个线程，比如发送者每次将自己 count 的值与线程 ID 发送，并将 count 加 1，接受者从消息队列中获取一条信息，然后将其打印输出。

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 message，将框架代码中 message 中的文件拷贝到工程里，如下图所示。

```
.
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──message
                │    ├──message.c
                │    └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 message 目录下的 BUILD.gn

自行填入源文件“message.c”；

```
static_library("message") {
    sources = [
        "message.c"
    ]
    include_dirs = [
        "//utils/native/lite/include"
    ]
}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 message: message。第一个 message 指得是目录，第二个 message 是指的是上面的静态库名称 message；

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "message:message",
```

```
    ]
}
```

步骤 4 编写消息队列机制实现

关键代码参考如下：

```
/*消息队列发送，学员自行补充*/
    osMessageQueuePut(qid, (const void *)&sentry, 0, osWaitForever);
    /*消息队列获取,学员自行补充*/
    osMessageQueueGet(qid, (void *)&rentry, NULL, osWaitForever);
   /*创建一个消息队列,学员自行补充*/
   qid = osMessageQueueNew(QUEUE_SIZE, sizeof(message_entry), NULL);
```

步骤 5 编译烧录程序

查看运行结果，执行结果如下：

```
[Message Test] recv get 1 from send by message queue.
[Message Test] send send 2 to message queue.
[Message Test] recv get 2 from send by message queue.
[Message Test] osMessageQueueGetCapacity, capacity: 3.
[Message Test] osMessageQueueGetMsgSize, size: 8.
[Message Test] osMessageQueueGetCount, count: 0.
[Message Test] osMessageQueueGetSpace, space: 3.
[Message Test] send send 3 to message queue.
[Message Test] recv get 3 from send by message queue.
[Message Test] send send 4 to message queue.
[Message Test] recv get 4 from send by message queue.
[Message Test] send send 5 to message queue.
[Message Test] recv get 5 from send by message queue.
[Message Test] send send 6 to message queue.
[Message Test] recv get 6 from send by message queue.
[Message Test] send send 7 to message queue.
[Message Test] recv get 7 from send by message queue.
[Message Test] send send 8 to message queue.
[Message Test] recv get 8 from send by message queue.
[Message Test] send send 9 to message queue.
[Message Test] recv get 9 from send by message queue.
[Message Test] send send 10 to message queue.
[Message Test] recv get 10 from send by message queue.
```

问题研讨

使用消息队列有哪些好处?

1.4.4 演练场景 4：定时投食

背景

家庭生活中，我们经常遇到这样的困难，人在工作或者需要离开家一段时间，亦或是疫情隔离在外，家里的萌宠没人投食，又不放心寄放或无处寄放的情况下，我们需要设计一个定时投食器来解决我们的实际问题。

任务一　创建一个软件定时器

在我们的实验中，我们需要创建一个线程，线程中创建一个软件定时器用于计时。

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 timer2feed，将框架代码中 timer2feed 中的文件拷贝到工程里，如下图所示；

```
.
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──timer2feed
                │   ├──timer2feed.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 timer 目录下的 BUILD.gn

自行填入源文件“timer2feed.c”；

```
static_library("timer2feed") {
    sources = [
        "timer2feed.c"
    ]
    include_dirs = [
        "//utils/native/lite/include"
    ]
}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 timer2feed: timer2feed。第一个 timer2feed 指得是目录，第二个 timer2feed 是指的是上面的静态库名称 timer2feed。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
```

```
    features = [
        "timer2feed:timer2feed",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
/*创建一个软件定时器，学员自行补充*/
osTimerId_t periodic_tid = osTimerNew(cb_timeout_periodic, osTimerPeriodic, NULL, NULL);
/*开始计时 100 个时钟周期，学员自行补充*/
osStatus_t status = osTimerStart(periodic_tid, 100);
/*停止软件定时器，学员自行补充*/
status = osTimerStop(periodic_tid);
printf("[Timer Test] stop periodic timer, status :%d.\r\n", status);
/*删除软件定时器，学员自行补充*/
status = osTimerDelete(periodic_tid);
printf("[Timer Test] kill periodic timer, status :%d.\r\n", status);
```

步骤 5 编译烧录程序

查看运行结果，执行结果如下：

```
[Timer Test] times:0.
It's time to do somethings.
[Timer Test] times:1.
It's time to do somethings.
[Timer Test] times:2.
It's time to do somethings.
[Timer Test] times:3.
It's time to do somethings.
[Timer Test] times:4.
It's time to do somethings.
[Timer Test] times:5.
It's time to do somethings.
[Timer Test] times:6.
It's time to do somethings.
[Timer Test] times:7.
It's time to do somethings.
[Timer Test] times:8.
It's time to do somethings.
[Timer Test] times:9.
It's time to do somethings.
```

问题研讨

1. 硬件定时器和软件定时器区别以及各自的优缺点?
2. 如何修改定时间隔?

任务二　通过事件管理机制来控制投食

我们对任务一的代码进行扩展，将框架代码该任务中 timer2feed 中的文件拷贝到工程里。使用事件管理，来通知喂食的任务（线程）来执行喂食操作。

新增关键代码如下：

```
/*创建一个事件,学员自行补充*/
evt_id = osEventFlagsNew(NULL);

/*设置事件触发，学员自行补充*/
osEventFlagsSet(evt_id, FLAGS_MSK1);
/*等待事件触发，学员自行补充*/
flags = osEventFlagsWait(evt_id, FLAGS_MSK1, osFlagsWaitAny, osWaitForever);
```

运行结果如下：

```
[Timer Test] times:0.
It's time to do somethings.
send Event(feed food) to feed thread .
[Event Test] receive Event(0x1) success.
feed food to dog.
[Timer Test] times:1.
It's time to do somethings.
send Event(feed food) to feed thread .
[Event Test] receive Event(0x1) success.
feed food to dog.
[Timer Test] times:2.
It's time to do somethings.
send Event(feed food) to feed thread .
[Event Test] receive Event(0x1) success.
feed food to dog.
```

分析结果可以看到，软件定时器 timer 定时周期到以后通知喂食任务，喂食任务收到事件通知，执行喂食操作。

问题研讨

为什么需要设计的这么复杂，直接在定时器周期到了以后直接执行喂食程序，岂不是更简单吗？

1.5 案例总结

我的案例总结：

_____

_____

_____

2 驱动基础

2.1 课程介绍

本章节介绍 HarmonyOS 的 IoT 专有硬件服务子系统中的驱动开发。

2.2 教学目标

- 能够掌握 GPIO 驱动开发和使用；
- 能够掌握 I2C 驱动开发和使用；
- 能够掌握 PWM 和 ADC 等硬件驱动开发和使用。

2.3 案例背景

说明：本文所涉及的案例仅为样例，实际操作中请以真实设备环境为准，具体配置步骤请参考对应的产品文档。

驱动开发是设备开发必不可少的环节，是操作系统屏蔽硬件的保护伞，使得应用开发者不在需要了解硬件的具体实现。

2.4 演练任务

2.4.1 演练场景 1：路灯控制

背景

生活中经常会遇到需要通过软件控制我们的路灯开关，其他需要用到 GPIO 的操作。

思考

1、GPIO 是什么意思?

2、HarmonyOS 的 GPIO 有哪些接口?

任务一　灯光闪烁

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 led_control，将框架代码中 led_control 中的文件拷贝到工程里；

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├──led_control
                │   ├──led_control.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 led_control 目录下的 BUILD.gn

自行填入源文件“led_control.c”；

```
static_library("led_control") {
  sources = [
    "led_control.c",
  ]

  include_dirs = [
    "//utils/native/lite/include",
    "//kernel/liteos_m/kal",
    "//base/iot_hardware/peripheral/interfaces/kits",
    ]
}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 led_control:led_control。第一个 led_control 指得是目录，第二个 led_control 是指的是上面的静态库名称 led_control。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "led_control:led_control",
    ]
}
```

步骤 4　关键代码编写

参考如下：

```
/*LED 的 GPIO 初始化，学员自行补充*/
```

```
IoTGpioInit(LED_TASK_GPIO);
/*GPIO 设置为输出，学员自行补充*/
IoTGpioSetDir(LED_TASK_GPIO,IOT_GPIO_DIR_OUT);

        case LED_ON:
        //GPIO 控制灯开，学员自行补充
            IoTGpioSetOutputVal(LED_TASK_GPIO,0);
            osDelay(LED_INTERVAL_TIME);
            break;
        case LED_OFF:
        //GPIO 控制灯关，学员自行补充
            IoTGpioSetOutputVal(LED_TASK_GPIO,1);
            osDelay(LED_INTERVAL_TIME);
            break;
        case LED_SPARK:
        //GPIO 控制灯闪，学员自行补充
            IoTGpioSetOutputVal(LED_TASK_GPIO,0);
            osDelay(LED_INTERVAL_TIME);
            IoTGpioSetOutputVal(LED_TASK_GPIO,1);
            osDelay(LED_INTERVAL_TIME);
            break;
```

步骤 5 编译烧录程序

查看运行结果，参考《HCIA-HarmonyOS Device Developer 实验环境搭建指南》或《内核开发入门》运行效果查看开发板。

烧写成功后可以看到，开发板上的 LED 一闪一闪的，说明我们使用 GPIO 控制 LED 成功。

问题研讨

用 GPIO 控制灯光闪烁，是把 GPIO 当做输出，那么 GPIO 可以当输入吗?

任务二 按键控制灯光状态

在任务一的代码基础上，编写按键相关代码，关键代码参考如下：

```
/*按键 GPIO 初始化，学员自行补充*/
IoTGpioInit(IOT_GPIO_KEY);
hi_io_set_func(IOT_GPIO_KEY, 0);
/*按键 GPIO 初始化为输入，学员自行补充*/
IoTGpioSetDir(IOT_GPIO_KEY, IOT_GPIO_DIR_IN);
```

```
hi_io_set_pull(IOT_GPIO_KEY, 1);
/*注册按键执行函数，学员自行补充*/
IoTGpioRegisterIsrFunc(IOT_GPIO_KEY, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_LOW,
    OnButtonPressed, NULL);
```

编译烧录程序，查看运行结果，运行效果查看开发板。

通过 USER 按键控制 LED 灯状态切换，灭、亮、闪。

问题研讨

GPIO 还有哪些用途?

2.4.2 演练场景 2：呼吸灯

背景

通过上面的演练场景，我们学会了通过 GPIO 控制灯光状态，那问题来了，如何控制亮暗程度？当然我们可以机械的装一个滑动电阻，通过旋钮去控制，这样需要修改硬件电路，操作起来会比较麻烦。假如灯被安装在电线杆上面，每次都需要爬上去调节吗？显然我们有更好的方式：PWM。

思考

PWM 是什么，还用在哪些场合?

任务一　制作一个呼吸灯

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 breathing_light，将框架代码中 breathing_light 中的文件拷贝到工程里；

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├──breathing_light
```

```
│   ├──pwm_led_demo.c
│   └── BUILD.gn
└── BUILD.gn
```

步骤 2 编写 breathing_light 目录下的 BUILD.gn

自行填入源文件“pwm_led_demo.c”；

```
static_library("pwm_led_demo") {
    sources = [
        "pwm_led_demo.c",
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//device/hisilicon/hispark_pegasus/sdk_liteos/include",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 breathing_light:pwm_led_demo。第一个 breathing_light 指得是目录，第二个 pwm_led_demo 是指的是上面的静态库名称 pwm_led_demo。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "breathing_light:pwm_led_demo",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
  //初始化一个 PWM 模块，学员自行补充
  IoTPwmInit(1);
      //PWM 开始，i<99，学员自行补充
      IoTPwmStart(1, i, PWM_FREQ_DIVITION);
      usleep(250000);
      //PWM 停止，学员自行补充
      IoTPwmStop(1);
```

步骤 5 编译烧录程序，查看运行结果

编译过程中报错：undefined reference to hi_pwm_init 等几个 hi_pwm_开头的函数 原因：因为默认情况下，hi3861_sdk 中，PWM 的 CONFIG 选项没有打开 解决： 修改 device/hisilicon/hispark_pegasus/sdk_liteos/build/config/usr_config.mk 文件中的

CONFIG_PWM_SUPPORT 行： # CONFIG_PWM_SUPPORT is not set 修改为 CONFIG_PWM_SUPPORT=y。

烧写成功后可以看到，插上交通灯板，我们使用 PWM 控制 LED 成功，可以看到红色 LED 由暗到亮，达到呼吸的效果。

问题研讨

HarmonyOS 的 PWM 有哪些接口?

2.4.3 演练场景 3：光照感应

背景

在某个项目中，需要根据光照的强度来执行不同的业务，或者做一些策略来弥补光照的不足和过剩，在我们的监控摄像头中就会用到红外补光，还有针对于环境光强做一些图像算法，达到夜视的效果。

思考

1、光照强度使用了什么传感器?

2、传感器的模拟信号如何转换成数字信号?

3、ADC 有哪些 API 接口?

任务一 获取光照强度

步骤 1 新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 adc，将框架代码中 adc 中的文件拷贝到工程里；

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├──adc
```

```
│   ├──adc_demo.c
│   └── BUILD.gn
└── BUILD.gn
```

步骤 2 编写 adc 目录下的 BUILD.gn

自行填入源文件“adc_demo.c”；

```
static_library("adc_demo") {
    sources = [
        "adc_demo.c",
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//device/hisilicon/hispark_pegasus/sdk_liteos/include",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 adc:adc_demo。第一个 adc 指得是目录，第二个 adc_demo 是指的是上面的静态库名称 adc_demo。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "adc:adc_demo",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    //读取 ADC 信息，学员自行补充
    if (hi_adc_read(LIGHT_SENSOR_CHAN_NAME, &data, HI_ADC_EQU_MODEL_4, HI_ADC_CUR_BAIS_
DEFAULT, 0)
        == IOT_SUCCESS) {
        printf("ADC_VALUE = %d\n", (unsigned int)data);
        osDelay(100);
    }
```

步骤 5 编译烧录

烧录文件后，按下 reset 按键，程序开始运行，改变炫彩灯板光敏电阻周围环境的光，会发现串口打印的 ADCvalue 会发生变化；

有光时，串口输出的 ADC 的值为 120 左右；无光时，串口输出的 ADC 的值为 1800 左右。

由 ADC 值计算对应引脚电压的公式为 Value=voltage/4/1.8×4096。

串口打印结果如下：

```
ADC_VALUE = 125
ADC_VALUE = 125
ADC_VALUE = 124
ADC_VALUE = 1827
ADC_VALUE = 1830
ADC_VALUE = 125
ADC_VALUE = 126
```

问题研讨

为什么上面的 ADC 值 Value=voltage/4/1.8×4096。为什么是 4096?

2.4.4 演练场景 4：环境监测

背景

现代农业中，已经非常科学化了，对环境指标不能仅仅靠人去感知，传感器可以很好的把一些农作物生长指标进行量化，比如什么温度，湿度条件最适合生长。我们可以通过监测这些环境因素来得到当前农作物的生长环境。

思考

1、开发板中的温湿度传感器是如何把数据传输给主芯片?

2、HarmonyOSI2C 的 API 有哪些?

任务一　获取温湿度

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 environment，将框架代码中 environment 中的文件拷贝到工程里；

```
.
```

```
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├──environment
                │   ├──aht20.c
                │   ├──aht20.h
                │   ├──envrionment_demo.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2 编写 environment 目录下的 BUILD.gn

自行填入源文件“envrionment_demo.c”，“aht20.c”；

```
static_library("sensing_demo") {
    sources = [
        "envrionment_demo.c", "aht20.c"
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//base/iot_hardware/peripheral/interfaces/kits",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 environment:sensing_demo。第一个 environment 指得是目录，第二个 sensing_demo 是指的是上面的静态库名称 sensing_demo。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "environment:sensing_demo",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    /*开始测量，学员自行补充*/
    retval = AHT20_StartMeasure();
    if (retval != IOT_SUCCESS) {
        printf("trigger measure failed!\r\n");
    }
    /*获取温湿度，学员自行补充*/
    retval = AHT20_GetMeasureResult(&temperature, &humidity);
    if (retval != IOT_SUCCESS) {
```

```
        printf("get humidity data failed!\r\n");
    }
```

步骤 5 编译烧录程序

插入环境监测板，查看运行结果。

编译过程中报错：undefined reference to hi_i2c_write 等几个 hi_i2c_开头的函数

原因：因为默认情况下，hi3861_sdk 中，I2C 的 CONFIG 选项没有打开

解决：修改 device/hisilicon/hispark_pegasus/sdk_liteos/build/config/usr_config.mk 文件中的 CONFIG_I2C_SUPPORT 行：

CONFIG_I2C_SUPPORT is not set 修改为 CONFIG_I2C_SUPPORT=y

```
temp = 35.595703, humi= 27.499008
temp = 35.763741, humi= 27.563763
temp = 35.881042, humi= 27.241039
temp = 35.928154, humi= 27.080250
temp = 36.034775, humi= 27.312088
temp = 36.164284, humi= 26.858902
temp = 36.310387, humi= 27.117348
```

问题研讨

是否可以接多路支持 I2C 总线的硬件?

任务二 显示温湿度

在任务一的代码基础上，编写 OLED 相关代码，将该任务框架代码中的 oled 的驱动程序拷贝到工程目录下。

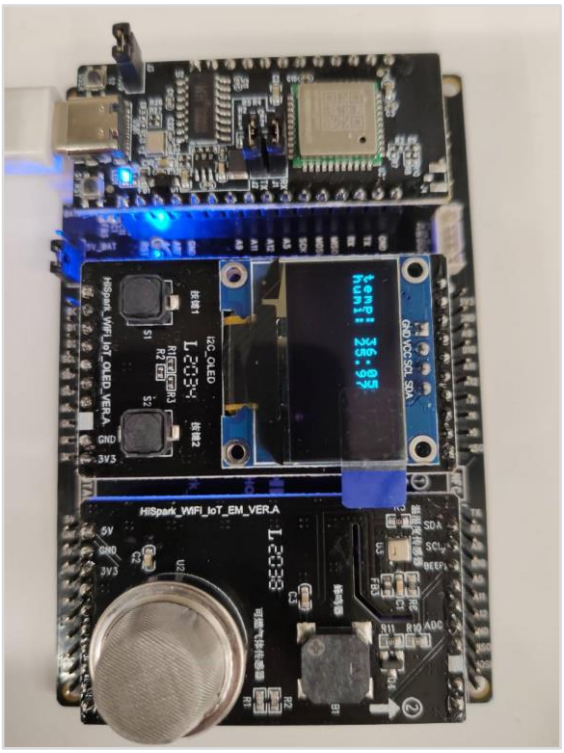
environment 目录下的 BUILD.gn 增加 oled_ssd1306.c 源文件

```
static_library("sensing_demo") {
    sources = [
        "envrionment_demo.c", "oled_ssd1306.c", "aht20.c"
    ]
```

关键代码参考如下：

```
    /*Oled 显示温度，学员自行补充*/
    snprintf(line, sizeof(line), "temp: %.2f", temperature);
    OledShowString(0, 1, line, 1);
    /*Oled 显示湿度，学员自行补充*/
    snprintf(line, sizeof(line), "humi: %.2f", humidity);
    OledShowString(0, 2, line, 1);
```

编译烧录程序，查看运行结果，运行效果查看开发板。



问题研讨

驱动和应用分层有什么好处?

2.5 案例总结

我的案例总结:

_____

_____

_____

3 子系统开发入门

3.1 课程介绍

下面将对我们内核子系统中的 KV 存储、文件操作、WiFi 操作和网络编程分别进行编程实践。

3.2 教学目标

- 能够掌握使用 KV 存储；
- 能够掌握使用文件操作；
- 能够掌握 WiFi 模块编程；
- 能够熟悉网络编程。

3.3 案例背景

我们内核子系统中的 KV 存储、文件操作、WiFi 操作和网络编程分别进行编程实践具体需要完成以下步骤：

- 基于 KV 存储的 WiFi 账号密码管理；
- 文件操作；
- WiFi 操作；
- 网络编程。

3.4 演练任务

3.4.1 演练场景 1：WiFi 账号密码管理

背景

生活中经常会遇到，WiFi 名字记住了，但是密码忘记了，很多时候手机会把 WiFi 账号对应的密码记住。

思考

1、KV 操作是什么?

2、HarmonyOS 子系统中有哪些 KV 操作的接口?

任务一　使用 KVstore 来管理 WiFi

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 kvstore，将框架代码中 kvstore 中的文件拷贝到工程里；

```
.
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──kvstore
                │   ├──kvstore.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 kvstore 目录下的 BUILD.gn

自行填入源文件“kvstore.c”；

```
static_library("kvstore") {
    sources = [
        "kvstore.c",
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//utils/native/lite/include",
    ]
}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 kvstore:kvstore。第一个 kvstore 指得是目录，第二个 kvstore 是指的是上面的静态库名称 kvstore。

```
import("//build/lite/config/component/lite_component.gni")
```

```
lite_component("app") {
    features = [
        "kvstore:kvstore",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
//设置关键值 wifi1_name_key 和值 wifi1_passwd_value，学员自行补充
ret = UtilsSetValue(wifi1_name_key, wifi1_passwd_value);
printf("[wifi manage] set key = %s, value = %s\n", wifi1_name_key, wifi1_passwd_value);
//设置关键值 wifi2_name_key 和值 wifi2_passwd_value，学员自行补充
ret = UtilsSetValue(wifi2_name_key, wifi2_passwd_value);
printf("[wifi manage] set key = %s, value = %s\n", wifi2_name_key, wifi2_passwd_value);

char temp[128] = {0};
//根据关键字（wifi1 账号），获取值（wifi1 密码），学员自行补充
ret = UtilsGetValue(wifi1_name_key, temp, 128);
printf("[wifi manage] get %s passwd = %s\n", wifi1_name_key, temp);
//根据关键字（wifi2 账号），获取值（wifi2 密码），学员自行补充
ret = UtilsGetValue(wifi2_name_key, temp, 128);
printf("[wifi manage] get %s passwd = %s\n", wifi2_name_key, temp);

//删除关键字 wifi1_name_key，学员自行补充
ret = UtilsDeleteValue(wifi1_name_key);
//删除关键字 wifi2_name_key，学员自行补充
ret = UtilsDeleteValue(wifi2_name_key);
```

步骤 5 编译烧录程序

查看运行结果，运行效果查看开发板。

```
[WiFi manage] set key = huawei_WiFi_ap1, value = 12345678
[WiFi manage] set key = huawei_WiFi_ap2, value = abcdefgh
[WiFi manage] get huawei_WiFi_ap1 passwd = 12345678
[WiFi manage] get huawei_WiFi_ap2 passwd = abcdefgh
```

问题研讨

KV 存储有哪些优点?

3.4.2 演练场景 2：文件操作

背景

文件操作是操作系统中常用的，文件对数据进行规范化的管理，也可以屏蔽硬件侧的操作。

思考

HarmonyOS 系统中文件操作有哪些接口?

3.4.2.1 任务一　文件测试

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 fileoperation，将框架代码中 fileoperation 中的文件拷贝到工程里；

```
.
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──fileoperation
                │   ├──fileoperation.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 kvstore 目录下的 BUILD.gn

自行填入源文件" fileoperation.c"。

```
static_library("fileoperation") {
    sources = [
        "fileoperation.c",
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//utils/native/lite/include",
    ]
}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 fileoperation:fileoperation。第一个 fileoperation 指得是目录，第二个 fileoperation 是指的是上面的静态库名称 fileoperation。

```
import("//build/lite/config/component/lite_component.gni")
```

```
lite_component("app") {
    features = [
        "fileoperation:fileoperation",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    //打开名为 testfile 的文件，并且可读可写，没有自行创建，学员自行补充
    int fd = UtilsFileOpen(fileName, O_RDWR_FS | O_CREAT_FS | O_TRUNC_FS, 0);
    printf("[FileOperation] file handle = %d\n", fd);
    //对文件进行写入操作，学员自行补充
    int ret = UtilsFileWrite(fd, def, strlen(def));
    printf("[FileOperation] write ret = %d\n", ret);

    //文件定位从第五个字节开始
    ret = UtilsFileSeek(fd, 5, SEEK_SET_FS);
    printf("[FileOperation] lseek ret = %d\n", ret);

    char buf[64] = {0};
    //读取文件，学员自行补充
    int readLen = UtilsFileRead(fd, buf, 64);
    //关闭文件，学员自行补充
    ret = UtilsFileClose(fd);
    printf("[FileOperation] read len = %d : buf = %s\n", readLen, buf);
    // stat
    int fileLen = 0;
    ret = UtilsFileStat(fileName, &fileLen);
    printf("[FileOperation] file size = %d\n", fileLen);
    //删除文件,学员自行补充
    ret = UtilsFileDelete(fileName);
    printf("[FileOperation] delete ret = %d\n", ret);
```

步骤 5 编译烧录程序

查看运行结果，运行效果查看开发板。

```
[FileOperation] write ret = 31
[FileOperation] lseek ret = 5
[FileOperation] read len = 26 : buf = _file_operation implement.
[FileOperation] file size = 31
[FileOperation] delete ret = 0
```

问题研讨

为什么关键代码中，写入到文件的内容是 "utils_file_operation implement."，最后读取出来的是“_file_operation implement.”?

3.4.3 演练场景 3：WiFi 操作

背景

使用 3861 开发板连接路由器（手机热点），或者将 3861 开发板当做热点供其他设备连接，从而进行数据通信。

思考

WiFi 模块有哪些模式?

WiFi 模块有哪些 API 来支持我们开发?

详细参考 HarmonyOS 官方 API 参考：

https://device.harmonyos.com/cn/docs/develop/apiref/WiFiservice-0000001055195054

任务一　WiFi 连接模式

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 WiFi_connect，将框架代码中 WiFi_connect 中的文件拷贝到工程里。

```
.
└──── applications
    └── sample
        └── wifi-iot
            └── app
                ├──WiFi_connect
                │   ├──WiFi_connect_demo.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 WiFi_connect 目录下的 BUILD.gn

自行填入源文件“WiFi_connect_demo.c”；

```
static_library("WiFi_demo") {
    sources = [
```

```
        "WiFi_connect_demo.c",
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//foundation/communication/WiFi_lite/interfaces/WiFiservice",
        "//vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include/",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 WiFi_connect:WiFi_demo。第一个 WiFi_connect 指得是目录，第二个 WiFi_demo 是指的是上面的静态库名称 WiFi_demo。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "WiFi_connect:WiFi_demo",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    // setup your AP params，配置 wifi 账号密码，学员自行补充
    strcpy(apConfig.ssid, "Huawei");
    strcpy(apConfig.preSharedKey, "12345678");
    apConfig.securityType = WIFI_SEC_TYPE_PSK;

        //使能 wifi 功能，学员自行补充
        errCode = EnableWifi();
        printf("EnableWifi: %d\r\n", errCode);
        osDelay(10);
        //增加设备配置，学员自行补充
        errCode = AddDeviceConfig(&apConfig, &netId);
        printf("AddDeviceConfig: %d\r\n", errCode);
        //连接网络，学员自行补充
        g_connected = 0;
        errCode = ConnectTo(netId);
        printf("ConnectTo(%d): %d\r\n", netId, errCode);
```

步骤 5 编译烧录程序

查看运行结果，运行效果查看开发板。

```
RegisterWiFiEvent: 0
```

```
EnableWiFi: 0
AddDeviceConfig: 0
ConnectTo(0): 0
+NOTICE:SCANFINISH
+NOTICE:CONNECTED
OnWiFiConnectionChanged 58, state = 1, info =
bssid: 42:01:67:72:39:79, rssi: 0, connState: 0, reason: 0, ssid: Huawei
g_connected: 1
netifapi_dhcp_start: 0
server :
        server_id : 192.168.43.1
        mask : 255.255.255.0, 1
        gw : 192.168.43.1
        T0 : 3599
        T1 : 1799
        T2 : 3149
clients <1> :
        mac_idx mac             addr            state   lease   tries   rto
        0       b4c9b9af66ec    192.168.43.55   10      0       1       2
netifapi_netif_common: 0
after 59 seconds, I'll disconnect WiFi!
after 58 seconds, I'll disconnect WiFi!
```

问题研讨

代码中哪些关键信息决定开发板连接的是哪个 WiFi 热点?

任务二 WiFi 热点模式

步骤 1 新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 WiFi_hotspot，将框架代码中 WiFi_hotspot 中的文件拷贝到工程里；

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├──WiFi_hotspot
                │   ├──WiFi_hotspot_demo.c
                │   └── BUILD.gn
```

```
                    └── BUILD.gn
```

步骤 2 编写 WiFi_hotspot 目录下的 BUILD.gn

自行填入源文件 “WiFi_hotspot_demo.c”；

```
static_library("WiFi_demo") {
    sources = [
        "WiFi_hotspot_demo.c",
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//foundation/communication/WiFi_lite/interfaces/WiFiservice",
        "//vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include/",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 WiFi_hotspot:WiFi_demo。第一个 WiFi_hotspot 指得是目录，第二个 WiFi_demo 是指的是上面的静态库名称 WiFi_demo。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "WiFi_hotspot:WiFi_demo",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    // 准备 AP 的配置参数,热点名称密码，学员自行补充
    strcpy(config.ssid, "HiSpark-AP");
    strcpy(config.preSharedKey, "12345678");
    config.securityType = WIFI_SEC_TYPE_PSK;
    config.band = HOTSPOT_BAND_TYPE_2G;
    config.channelNum = 7;
```

步骤 5 编译烧录程序

查看运行结果，运行效果查看开发板。

```
starting AP ...
RegisterWiFiEvent: 0
SetHotspotConfig: 0
OnHotspotStateChanged: 1.
EnableHotspot: 0
```

```
g_hotspotStarted = 1.
netifapi_netif_set_addr: 0
netifapi_dhcp_start: 0
StartHotspot: 0
After 590 seconds Ap will turn off!
After 580 seconds Ap will turn off!
+NOTICE:STA CONNECTED
 PrintStationInfo: mac=50:E0:85:DA:3C:43, reason=0.
+OnHotspotStaJoin: active stations = 1.
```

可以通过串口工具发送 AT+PING=192.168.xxx.xxx(如电脑连接到该热点后的 IP，本电脑连接到 HiSpark-AP 热点后，IP 地址为 192.168.1.3) 去 ping 连接到该热点的设备的 IP 地址。

进入 windows 命令行模式，输入 ipconfig。

```
ipconfig
无线局域网适配器 WLAN:

   连接特定的 DNS 后缀 . . . . . . . :
   IPv4 地址 . . . . . . . . . . . . : 192.168.1.3
   子网掩码 . . . . . . . . . . . . : 255.255.255.0
   默认网关. . . . . . . . . . . . . : 192.168.1.1
```

问题研讨

代码中哪些关键配置决定开发板的 WiFi 热点信息?

3.4.4 演练场景 4：通过 WiFi 进行网络通信

背景

前面讲的都是单机，那如果需要使用手机或者电脑，甚至第三方硬件设备来控制我们的开发板，我们的板子配置 WiFi 芯片，轻而易举的就能实现，通过网络来控制设备和数据传输等跨设备的场景。

思考

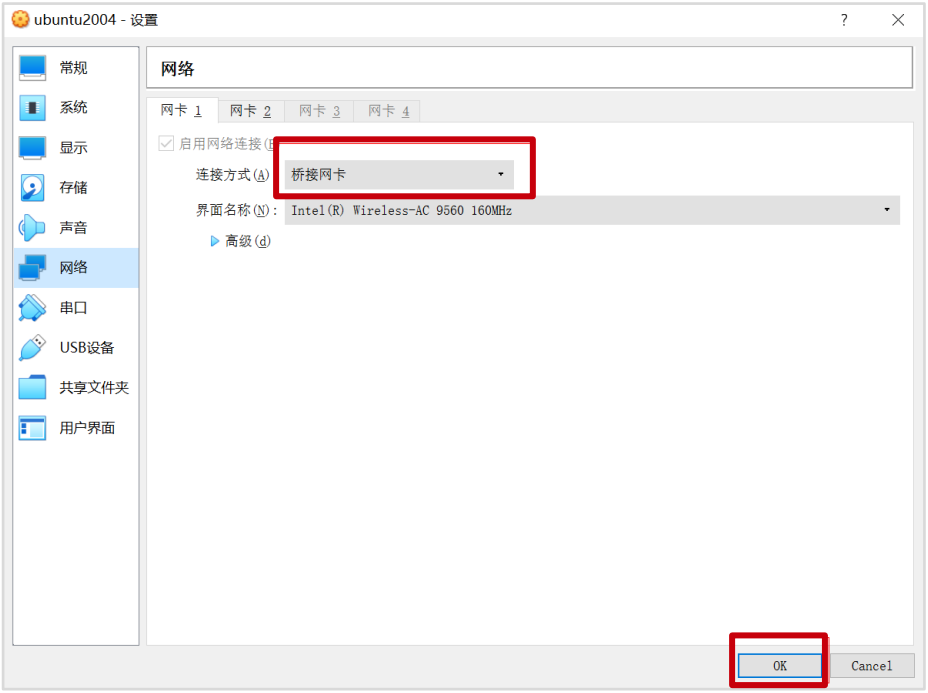
WiFi 有几种模式?

任务一 配置网络环境

准备一个无线路由器，或者直接拿手机当热点。

热点名字 Huawei，密码 12345678。

PC 连到这个热点，虚拟机网络设置如下：



启动虚拟机，查看 IP 地址，记住这个主机 IP:192.168.43.190。

```
harmonyos@harmonyos-VirtualBox:~/桌面$ ifconfig
enp0s3: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.43.190  netmask 255.255.255.0  broadcast 192.168.43.255
        inet6 fe80::5360:28ba:726f:9050  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:5e:43:ee  txqueuelen 1000  (以太网)
        RX packets 421  bytes 403228 (403.2 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 417  bytes 47251 (47.2 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0

enp0s8: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 192.168.56.101  netmask 255.255.255.0  broadcast 192.168.56.255
        inet6 fe80::a00:27ff:fea0:13a1  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:a0:13:a1  txqueuelen 1000  (以太网)
        RX packets 73  bytes 12152 (12.1 KB)
        RX errors 0  dropped 0  overruns 0  frame 0
        TX packets 188  bytes 31105 (31.1 KB)
        TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

问题研讨

为什么要记住这个 IP 地址?

任务二　搭建 TCP 服务端

使用 Ubuntu 虚拟机，当做 TCP 服务端。安装 netcat 工具，netcat 是一个非常强大的网络实用工具，可以用它来调试 TCP/UDP 应用程序；

通过命令：sudo apt-get install netcat 安装在 ubuntu 虚拟机上。

```
harmonyos@harmonyos-VirtualBox:~/share/code-1.1.0$ sudo apt-get install netcat
[sudo] harmonyos 的密码：
正在读取软件包列表... 完成
正在分析软件包的依赖关系树
正在读取状态信息... 完成
netcat 已经是最新版 (1.206-1ubuntu1)。
下列软件包是自动安装的并且现在不需要了：
  libffi6 libhunspell-1.6-0 libisl19 libncursesw5 libpython3.6 libpython3.6-minimal libpython3.6-stdlib
  libreadline7 libtinfo5 xdg-dbus-proxy
使用'sudo apt autoremove'来卸载它(它们)。
升级了 0 个软件包，新安装了 0 个软件包，要卸载 0 个软件包，有 1 个软件包未被升级。
```

开始之前，先简单介绍一下 netcat 的几种用法：

TCP 服务端模式：netcat -l 5678 ，会启动一个 TCP 服务器，监听 5678 端口，可以换成其他端口；

TCP 客户端模式：netcat localhost 5678，localhost 是目标主机参数，可以换成其他想要连接的主机（主机名、IP 地址、域名都可以），5678 是端口；

如果在同一台机器的两个终端中分别执行上述两条命令，它们两者之间就会建立连接一条 TCP 连接，此时在其中一个终端上输入字符，敲回车就会发送到另一个终端中；

UDP 服务端模式：netcat -u -l 6789，只需要加一个-u 参数，就可以启动一个 UDP 服务端；

UDP 客户端模式：netcat -u localhost 6789。

类似的，在同一台机器的两个终端中分别执行上述两条命令，他们两者之间也可以收发消息，只不过是 UDP 报文；

问题研讨

在案例中，ubuntu 上的 netcat 工具充当的是 TCP 服务端还是客户端?

任务三　搭建 TCP 客户端

步骤 1　新建目录

复制框架代码 network 文件夹到 app 下。

使用开发板作为 TCP 客户端，修改 applications/sample/wifi-iot/app 下的 BUILD.gn。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "network:net_demo",
    ]
}
```

在 Hi3861 开发板上运行测试程序之前，需要提前将 PC 连接到热点，然后重新虚拟机，查看 IP，最后根据无线路由、Linux 系统 IP 修改 net_params.h 文件的相关代码：

PARAM_HOTSPOT_SSID 修改为热点名称；

PARAM_HOTSPOT_PSK 修改为热点密码；

PARAM_SERVER_ADDR 修改为 Linux 主机 IP 地址。

```
#ifndef PARAM_HOTSPOT_SSID
#define PARAM_HOTSPOT_SSID "Huawei"    // your AP SSID
#endif

#ifndef PARAM_HOTSPOT_PSK
#define PARAM_HOTSPOT_PSK  "12345678"  // your AP PSK
#endif

#ifndef PARAM_HOTSPOT_TYPE
#define PARAM_HOTSPOT_TYPE WIFI_SEC_TYPE_PSK // defined in WiFi_device_config.h
#endif

#ifndef PARAM_SERVER_ADDR
#define PARAM_SERVER_ADDR "192.168.43.190"  //"192.168.1.100" // your Linux server IP address
#endif

#ifndef PARAM_SERVER_PORT
#define PARAM_SERVER_PORT 5678
#endif
```

步骤 2 编译烧录，测试运行结果：

1、在 Linux 终端中使用 netcat 启动一个 TCP 服务端：netcat -l 5678；

2、连接开发板串口，复位开发板，板上程序启动后，首先会连接 WiFi 热点，然后会尝试连接到 Linux 上用 netcat 启动的 TCP 服务端；

3、在 Linux 终端中应该会出现开发板上 TCP 客户端通过发来的 Hello，输入 World 并回车，World 将会发送到开发板上，同时开发板的串口会有相关打印。

Linux 虚拟机：

```
harmonyos@harmonyos-VirtualBox:~/桌面$ netcat -l 5678
HelloWorld
```

开发板：

```
RegisterWiFiEvent: 0
EnableWiFi: 0
AddDeviceConfig: 0
ConnectTo(0): 0
+NOTICE:SCANFINISH
+NOTICE:CONNECTED
OnWiFiConnectionChanged 54, state = 1, info =
bssid: 42:01:67:72:39:79, rssi: 0, connState: 0, reason: 0, ssid: Huawei
g_connected: 1
netifapi_set_hostname: 0
netifapi_dhcp_start: 0
server :
        server_id : 192.168.43.1
        mask : 255.255.255.0, 1
        gw : 192.168.43.1
        T0 : 3599
        T1 : 1799
        T2 : 3149
clients <1> :
        mac_idx mac             addr            state   lease   tries   rto
        0       b4c9b9af66ec    192.168.43.55   10      0       1       4
netifapi_netif_common: 0
After 9 seconds, I will start TcpClientTest test!
After 8 seconds, I will start TcpClientTest test!
After 7 seconds, I will start TcpClientTest test!
After 6 seconds, I will start TcpClientTest test!
After 5 seconds, I will start TcpClientTest test!
After 4 seconds, I will start TcpClientTest test!
After 3 seconds, I will start TcpClientTest test!
After 2 seconds, I will start TcpClientTest test!
After 1 seconds, I will start TcpClientTest test!
After 0 seconds, I will start TcpClientTest test!
TcpClientTest start
I will connect to 192.168.43.190:5678
connect to server 192.168.43.190 success!
send request{Hello} 6 to server done!
recv response{World
} 6 from server done!
do_cleanup...
TcpClientTest done!
disconnect to AP ...
netifapi_dhcp_stop: 0
+NOTICE:DISCONNECTED
OnWiFiConnectionChanged 54, state = 0, info =
bssid: 42:01:67:72:39:79, rssi: 0, connState: 0, reason: 3, ssid:
Disconnect: 0
```

```
UnRegisterWiFiEvent: 0
RemoveDevice: 0
DisableWiFi: 0
disconnect to AP done!
```

问题研讨

在案例中，开发板充当的是 TCP 服务端还是客户端?

3.5 案例总结

我的案例总结:

_____

_____

_____

4 移植实验

4.1 课程介绍

HarmonyOS 作为开源的操作系统，最主要的是开放，那么如何将公司的业务代码融合进 HarmonyOS 系统，或者将一些开源库添加到 HarmonyOS 内核中。

4.2 教学目标

- 能够掌握使用 gn 编译脚本将第三方库编译进 HarmonyOS 程序。

4.3 案例背景

将一些公司原有的业务代码移植到 HarmonyOS 源码中，经过 gn 编译脚本将库编译进 HarmonyOS 的应用程序。

具体需要完成以下步骤：

- 第三方库的移植、编译和使用。

4.4 演练任务

4.4.1 演练场景 1：三方库的移植、编译和使用

背景

公司需要开发一款新产品，或者新增一个新功能，一般来说有两种方式：

1、自己新增一个库，自己实现功能模块（函数）；

2、直接使用第三方的开源库。

思考

为什么需要移植三方库?

任务一　最简单的库移植

回顾章节 2 的演练场景 1 中的任务一，其实这是最简单的移植三方库到我们的 HarmonyOS 系统中，我们只需要为之添加一个头文件，就能对外开放其函数实现。

步骤 1　新建目录

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 library_test，把框架代码中需要移植的库放到 lib 下面，本例中 lib 仅仅作为展示，实现了打印 helloworld 的功能。然后我们新建 library_test.c 来调用 lib 中的库函数，来打印 helloworld。目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├── library_test
                │   ├── BUILD.gn
                │   ├── lib
                │   │   ├── BUILD.gn
                │   │   ├── hello_world.c
                │   │   └── hello_world.h
                │   └── library_test.c
```

步骤 2　编写 lib 目录下的 BUILD.gn，用于生成 hello_world 库

```
config("hello_config"){
    include_dirs = [
        "//utils/native/lite/include",
        "."
    ]
}

lite_library("hello_static") {
    target_type = "static_library"
    sources = [
        "hello_world.c"
    ]
    public_configs = [ ":hello_config" ]
}
```

步骤 3　编写 library_test 目录下的 BUILD.gn，用于调用 hello_world 库

```
static_library("library_test") {
    sources = [
        "library_test.c"
    ]
```

```
    deps = [
        "//applications/sample/wifi-iot/app/library_test/lib:hello_static",
    ]
}
```

步骤 4 编写 app 目录下的 BUILD.gn

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "library_test:library_test",
    ]
}
```

步骤 5 关键代码编写

参考如下：

```
//第三方库头文件增加，学员自行补充
#include "hello_world.h"

void library_test(void)
{
   //第三方库中的函数调用，学员自行补充
   hello_world();
}
```

步骤 6 编译烧录程序

查看运行结果，执行结果如下：

```
[HelloWorld library] :Hello world.
```

问题研讨

本任务和第二章演练场景 1 任务一，同样打印 Hello world，实现方式有什么不同?

4.5 案例总结

我的案例总结：

_____

_____

_____

5 综合实验

5.1 课程介绍

本章综合实验讲使用 HarmonyOS 开发板来模拟智慧生活中的一些案例。

5.2 教学目标

- 能够掌握使用 gn 编译脚本；
- 能够使用 WiFi；
- 能够使用 socket 编程。

5.3 案例背景

随着互联网技术的发展，智能家居设备在生活中的应用越来越广泛，本章我们用开发板来模拟一些智慧生活中的案例。

具体需要完成以下步骤：

- 创建工程，使用正确的 gn 编译脚本文件；
- 配置好 WiFi；
- 使用 socket 通信，开发板作为服务端，手机作为客户端。

5.4 演练任务

5.4.1 演练场景 1：小鸿烤箱

背景

在我们的实际生活当中，经常会使用烤箱，本章综合实验讲使用 HarmonyOS 开发板来模拟小鸿烤箱。

思考

小鸿烤箱需要用到哪些知识点?

任务一　创建综合实验工程

步骤 1　为小鸿烤箱综合实验创建一个工程

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 xh_oven，将框架代码中 xh_oven 中的文件拷贝到工程里。目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
            └──xh_oven
            ├── BUILD.gn
            ├── key
            │   ├── key.c
            │   └── key.h
            ├── main.c
            ├── network
            │   ├── net_common.h
            │   ├── net_demo.h
            │   ├── net_params.h
            │   ├── tcp_client_test.c
            │   ├── tcp_server_test.c
            │   ├── udp_client_test.c
            │   ├── udp_server_test.c
            │   ├── WiFi_connecter.c
            │   ├── WiFi_connecter.h
            │   ├── WiFi_hotspot_ap.c
            │   └── WiFi_hotspot_ap.h
            └── oled
                ├── oled_fonts.h
                ├── oled_ssd1306.c
                └── oled_ssd1306.h
```

步骤 2　编写 xh_oven 目录下的 BUILD.gn，用于生成 main_demo 库

```
static_library("main_demo") {
    # uncomment one of following line, to enable one test:
```

```
    sources = [""]

    sources += [""]
    include_dirs = [
        "//utils/native/lite/include",
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//device/hisilicon/hispark_pegasus/sdk_liteos/include",
        "//foundation/communication/WiFi_lite/interfaces/WiFiservice"
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "xh_oven:main_demo",
    ]
}
```

问题研讨

xh_oven 目录下的 BUILD.gn 的编写，需要添加哪些源文件，哪些头文件？请进行补充。

提示：

```
sources = ["main.c"]

sources += ["network/tcp_server_test.c", "network/WiFi_connecter.c", "network/WiFi_hotspot_ap.c",
"oled/oled_ssd1306.c", "key/key.c"]
include_dirs = [
"./key",
"./network",
"./oled",
"//utils/native/lite/include",
```

任务二 补充业务代码

步骤 1 编写 app 目录下的 BUILD.gn

调用 main_demo 库，实现 app；

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
      features = [
           "xh_oven:main_demo",
      ]
}
```

步骤 2 增加关键代码

在 main.c 中有四个任务，分别为网络通信任务，显示任务，定时器任务，蜂鸣器任务。

在 MainDemoTask 函数中创建：

```
//创建四个任务，网络，显示，定时，蜂鸣器，学生自行补充
   osThreadId_t ptid1 = newThread("Network", NetworkTask, NULL);
   osThreadId_t ptid2 = newThread("Display", DisplayTask, NULL);
   osThreadId_t ptid3 = newThread("Timer", TimerTask, NULL);
   osThreadId_t ptid4 = newThread("PWMBeerTask", PWMBeerTask, NULL);
```

步骤 3 填充显示任务

static void DisplayTask(void *arg)，在 while(1)里面填充以下代码：

```
   //显示任务具体业务，学生自行补充
      memset(temp,0,sizeof(temp));
      /*显示蒸烤模式*/
      if(oven_params.mode == 0)
      {
          snprintf(temp, sizeof(temp), "MODE: stream");
      }
      else if(oven_params.mode == 1)
      {
          snprintf(temp, sizeof(temp), "MODE: roast ");
      }
      OledShowString(0, 0, temp, 1);
      /*显示设置时间*/
      memset(temp,0,sizeof(temp));
      snprintf(temp, sizeof(temp), "TIME: %2d", oven_params.time);
      OledShowString(0, 2, temp, 1);

      memset(temp,0,sizeof(temp));
      /*显示火候*/
      if(oven_params.fire == 0)
      {
          snprintf(temp, sizeof(temp), "FIRE: low    ");
      }
      else if(oven_params.fire == 1)
      {
          snprintf(temp, sizeof(temp), "FIRE: middle");
```

```
    }
    else if(oven_params.fire == 2)
    {
        snprintf(temp, sizeof(temp), "FIRE: high  ");
    }
    OledShowString(0, 4, temp, 1);
    /*显示开始，停止*/
    memset(temp,0,sizeof(temp));
    snprintf(temp, sizeof(temp), "START");
    OledShowString(0, 6, temp, 1);

    memset(temp,0,sizeof(temp));
    snprintf(temp, sizeof(temp), "STOP");
    OledShowString(80, 6, temp, 1);
    osDelay(10);
```

在 tcp_server_test.c 中，TcpServerTest 函数中，填入以下代码：

```
    //request 为设备收到 client 手机的数据，进行解析。学员自行补充
    if ( strstr ( ( const char * ) request, ( const char * ) "set mode=stream" ) )
    {
        oven_params.mode = 0;
    }
    else if(strstr ( ( const char * ) request, ( const char * ) "set mode=roast" ))
    {
        oven_params.mode = 1;
    }
    else if(strstr ( ( const char * ) request, ( const char * ) "set fire=low" ))
    {
        oven_params.fire = 0;
    }
        else if(strstr ( ( const char * ) request, ( const char * ) "set fire=middle" ))
    {
        oven_params.fire = 1;
    }
        else if(strstr ( ( const char * ) request, ( const char * ) "set fire=high" ))
    {
        oven_params.fire = 2;
    }
    else if(strstr ( ( const char * ) request, ( const char * ) "set time=" ))
    {
        oven_params.set_time = atoi(&request[9]);
    }
    else if(strstr ( ( const char * ) request, ( const char * ) "set start" ))
    {
        oven_params.start_flag = 1;
        oven_params.time = oven_params.set_time;
    }
```

```
    else if(strstr ( ( const char * ) request, ( const char * ) "set stop" ))
    {
        oven_params.start_flag = 0;
        g_beepState = 0;
    }
```

步骤 4 编译

```
[OHOS INFO] [236/239] ACTION //build/lite:gen_rootfs(//build/lite/toolchain:riscv32-unknown-elf)
[OHOS INFO] [237/239] STAMP obj/build/lite/gen_rootfs.stamp
[OHOS INFO] [238/239] ACTION
//device/hisilicon/hispark_pegasus/sdk_liteos:run_WiFiiot_scons(//build/lite/toolchain:riscv32-unknown-
elf)
[OHOS INFO] [239/239] STAMP
obj/device/hisilicon/hispark_pegasus/sdk_liteos/run_WiFiiot_scons.stamp
[OHOS INFO] wifiiot_hispark_pegasus build success
```

步骤 5 烧录

成功后运行，寻找 WiFi 热点。

```
[HelloWorld library] :Hello world.

EnableWiFi: 0
AddDeviceConfig: 0
ConnectTo(0): 0
4 TaskPool:0xe5398
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4b584c TaskPool:0xe53b8
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4b5aa0 TaskPool:0xe53d8
00 00:00:00 0 164 I 1/SAMGR: Init service 0x4b5214 <time: 0ms> success!
00 00:00:00 0 220 I 1/SAMGR: Init service 0x4b5208 <time: 0ms> success!
00 00:00:00 0 8 I 1/SAMGR: Init service 0x4b584c <time: 10ms> success!
00 00:00:00 0 108 I 1/SAMGR: Init service 0x4b5aa0 <time: 20ms> success!
00 00:00:00 0 108 I 1/SAMGR: Initialized all core system services!
00 00:00:00 0 220 I 1/SAMGR: Bootstrap system and application services(count:0).
00 00:00:00 0 220 I 1/SAMGR: Initialized all system and application services!
00 00:00:00 0 220 I 1/SAMGR: Bootstrap dynamic registered services(count:0).
+NOTICE:SCANFINISH
+NOTICE:NETWORK NOT FIND
OnWiFiConnectionChanged 54, state = 0, info =
bssid: 00:00:00:00:00:00, rssi: 0, connState: 0, reason: 1, ssid:
+NOTICE:SCANFINISH
+NOTICE:NETWORK NOT FIND
OnWiFiConnectionChanged 54, state = 0, info =
bssid: 00:00:00:00:00:00, rssi: 0, connState: 0, reason: 1, ssid:
+NOTICE:SCANFINISH
+NOTICE:NETWORK NOT FIND
OnWiFiConnectionChanged 54, state = 0, info =
```

```
bssid: 00:00:00:00:00:00, rssi: 0, connState: 0, reason: 1, ssid:
```

问题研讨

连接哪个 WiFi 热点，在代码中哪里体现?

任务三 实验效果演示

步骤 1 安装 HarmonyOS 手机应用

1、安装 DevEco Studio 工具

工具下载及安装参考链接：https://developer.harmonyos.com/cn/docs/documentation/doc-guides/installation_process-0000001071425528

2、下载手机应用

小鸿烤箱应用代码在框架代码目录下。

环境检测和智能夜灯案例使用同一个应用，代码下载链接：
https://gitee.com/hihopeorg_group/hcia_harmonyos_application

3、安装手机应用到 HarmonyOS 手机

安装参考链接：https://developer.harmonyos.com/cn/docs/documentation/doc-guides/run_phone_tablat-0000001064774652

安装应用注意事项：

- 在设置->系统和更新->开发人员选项->USB 调试中，打开 HarmonyOS 手机调试模式
- 推荐使用自动化签名方式
- 注意自定义修改应用代码配置文件 entry\src\main\config.json 中的 bundleName，不要使用默认的 bundleName
- AppGallery Connect 应用中配置的应用包名和 config.json 中 bundleName 保持一致

步骤 2 开发板连接 WiFi

准备一个路由器，没有的话，电脑或者手机当热点，网络名称必须为 Huawei，密码为 12345678。

烧录任务二编译生成的二进制文件，并且开启热点，烧录完成后按板子复位键，如下：

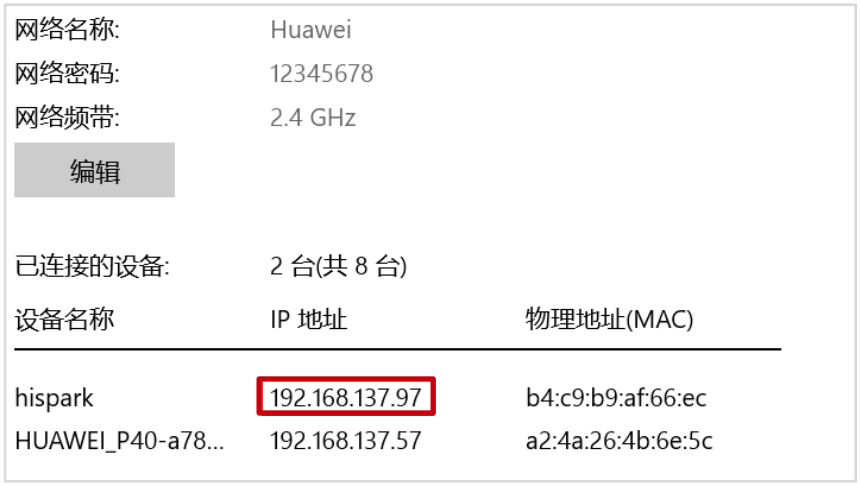
```
+NOTICE:SCANFINISH
+NOTICE:CONNECTED
OnWiFiConnectionChanged 54, state = 1, info =
bssid: 52:E0:85:DA:3C:43, rssi: 0, connState: 0, reason: 0, ssid: Huawei
g_connected: 1
netifapi_set_hostname: 0
netifapi_dhcp_start: 0
server :
        server_id : 192.168.137.1
        mask : 255.255.255.0, 1
        gw : 192.168.137.1
        T0 : 604800
        T1 : 302400
        T2 : 453600
clients <1> :
        mac_idx mac             addr            state   lease   tries   rto
        0       b4c9b9af66ec    192.168.137.97  10      0       1       4
netifapi_netif_common: 0
After 4 seconds, I will start TcpServerTest test!
After 3 seconds, I will start TcpServerTest test!
After 2 seconds, I will start TcpServerTest test!
After 1 seconds, I will start TcpServerTest test!
```

```
After 0 seconds, I will start TcpServerTest test!
TcpServerTest start
I will listen on :5678
bind to port 5678 success!
listen with 1 backlog success!
```

开发板连接 WiFi 成功。

步骤 3 手机和开发板互通

手机连接相同 WiFi，在 window 热点中查看开发板 IP 为 192.168.137.97。



打开手机端的小鸿烤箱 APP，打开设置，填入服务端 IP 和端口号，点击确认。

手机连接开发板，点击连接，状态由连接失败改为连接成功。

至此，手机可以控制开发板进行设置蒸烤操作。

小鸿烤箱配套手机应用使用说明：

1、设置开发板 IP 地址和端口号：将开发板与手机连接到同一局域网后，点击右上角设置按钮，在弹出的对话框填写开发板的局域网 IP 和端口号；

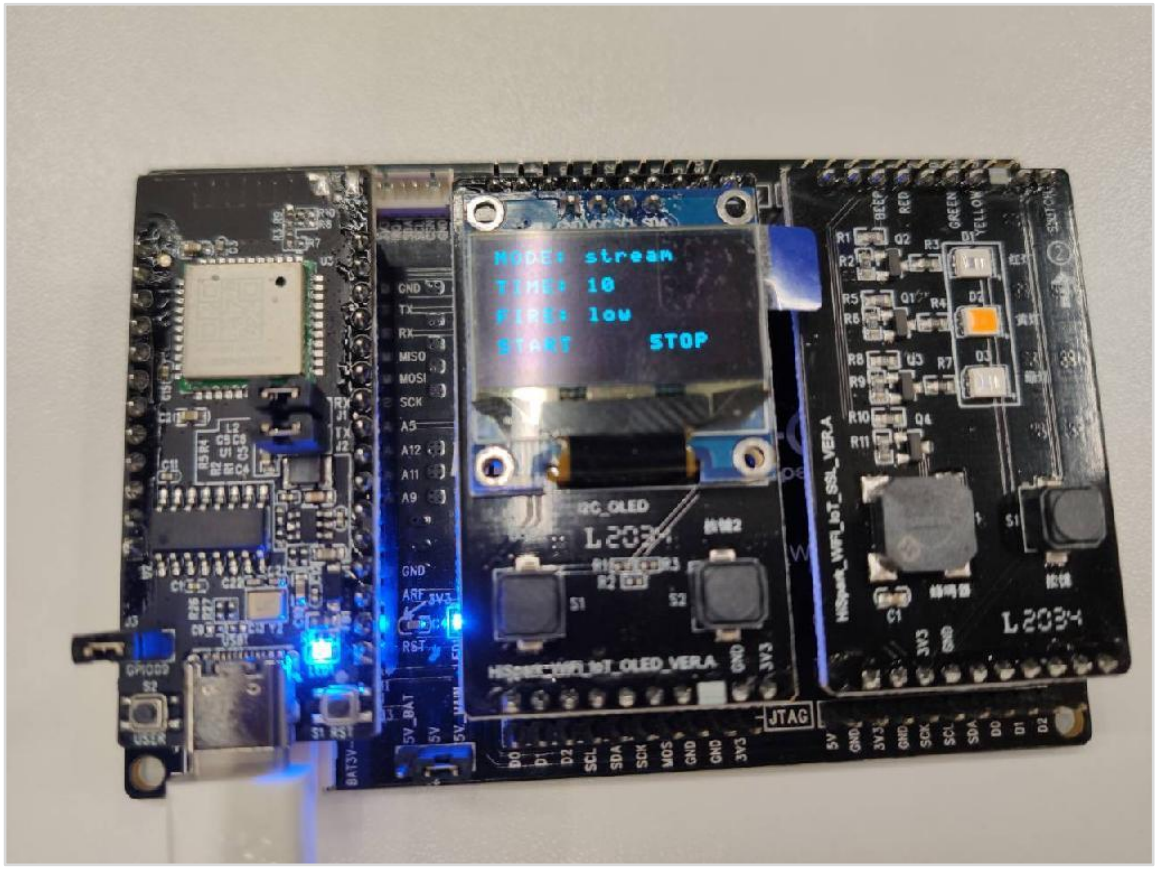
2、连接开发板：设置好开发板 IP 地址和端口号后，在首页点击“连接”按钮即可发起连接，连接成功后连接状态会变为“已连接”；

3、发送操作指令：在操作列表中点击操作项，即可向开发板发送操作指令；

4、添加操作指令：点击右上角添加按钮，在弹出的对话框中填写“操作名称”和“操作指令代码”，并确认，即可向操作列表中添加一条新的操作指令；

5、删除操作指令：在操作列表中，找到需要删除的操作项，点击该项右侧的删除按钮，即可删除该项；

6、显示开发板信息：开发板返回的信息将会被展示在屏幕上方“收到信息”栏目中。



问题研讨

为什么手机端需要设置服务端的 IP 地址?

5.4.2 演练场景 2：环境检测

5.4.2.1 背景

环境检测使用的场景很多，比如：大家对水果蔬菜的新鲜度要求越来越高，可以使用温湿度计监测水果蔬菜的存储环境，保证食物的新鲜度；可燃气体监测可以防止因室内可燃气体泄露导致的一系列问题。

5.4.2.2 思考

1、AHT20 驱动库的实现?

2、如何获取可燃气体传感器的数值?

3、如何驱动 OLED 屏?

4、使用什么控制蜂鸣器发声?

【参考答案】

1、参考任务一

2、ADC

3、参考任务三

4、PWM 信号

5.4.2.3 任务一　创建综合实验工程

步骤 1　为环境检测综合实验创建一个工程。

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 environment，将框架代码中 environment 中的文件拷贝到工程里。目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                └── environment
                    ├ aht20.c
                    ├ aht20.h
                    ├ aht20_test.c
                    ├ beep.c
                    ├ BUILD.gn
                    ├ config_params.h
                    ├ environment.h
                    ├ environment_demo.c
                    ├ mq2_test.c
                    ├ oled_fonts.h
                    ├ oled_ssd1306.c
                    ├ oled_ssd1306.h
                    └ oled_test.c
```

步骤 2　编写 app 目录下的 BUILD.gn

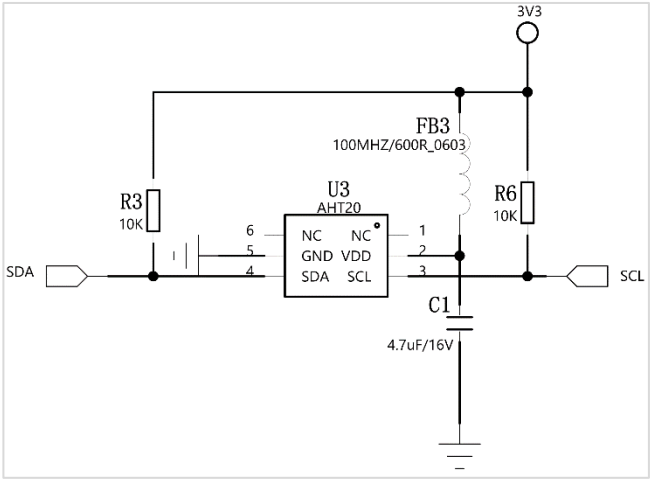
```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "environment:sensing_demo",
    ]
}
```

5.4.2.4 任务二 温湿度测量

步骤 1 读写函数封装

温湿度传感器原理图：



主控芯片 Pegasus 的硬件 I2C 有两个：I2C0、I2C1。环境监测板的 AHT20 温湿度传感器使用的 I2C0，引脚关系如下：

（1）GPIO13: I2C0_SDA。

（2）GPIO14: I2C0_SCL。

在 aht20.c 的函数 AHT20_Read()和 AHT20_Write()中使用 OpenHarmony 提供的 I2C 接口封装读写函数，代码参考 aht20.c 中的实现。

步骤 2 打开 I2C 开关

HarmonyOS 的 LiteOS-M 内核默认将 I2C 功能关闭，需要手动将 device\hisilicon\hispark_pegasus\sdk_liteos\build\config\usr_config.mk 文件中

CONFIG_I2C_SUPPORT is not set 改为 CONFIG_I2C_SUPPORT=y。

步骤 3 温湿度传感器测量功能实现

传感器常用命令：

命令	字节	功能说明
初始化（校准）命令	0xBE	初始化传感器并进行校准
触发测量命令	0xAC	触发采集后，传感器在采集时需要75ms完成采集
软复位命令	0xBA	用于在无须关闭和再次打开电源的情况下，重新启动传感器系统
获取状态命令	0x71	该命令的回复有如下两种情况： 1、在初始化后触发测量之前，STATUS 只回复 1B

		状态值； 2、在触发测量之后，STATUS 回复6B： 1B 状态值 + 2B 湿度 + 4b湿度 + 4b温度 + 2B 温度

传感器读取过程：

1.上电后要等待 40ms，读取温湿度值之前， 首先要看状态字的校准使能位 Bit[3]是否为 1(通过发送 0x71 可以获取一个字节的状态字)，如果不为 1，要发送 0xBE 命令(初始化)，此命令参数有两个字节， 第一个字节为 0x08，第二个字节为 0x00；

2.直接发送 0xAC 命令(触发测量)，此命令参数有两个字节，第一个字节为 0x33，第二个字节为 0x00；

3.等待 75ms 待测量完成，忙状态 Bit[7]为 0，然后可以读取六个字节(发 0X71 即可以读取)；

4.计算温湿度值。

代码参考 aht20.c 中的实现。

步骤 4 获取传感器的值

在 aht20_test.c 中创建一个任务，完成使用 AHT20 数字温湿度传感器测量室内温湿度的功能，将获取的值打印到串口调试接口。

在 EnvironmentTask()函数中添加代码：

```
//开始测量
retval = AHT20_StartMeasure();
if (retval != IOT_SUCCESS) {
    printf("trigger measure failed!\r\n");
}
//接收测量结果
retval = AHT20_GetMeasureResult(&g_temperature, &g_humidity);
if (retval != IOT_SUCCESS) {
    printf("get humidity data failed!\r\n");
}
```

步骤 5 运行程序

开发板上更换环境检测扩展板，更改 environment 目录下的 BUILD.gn 的编译文件为 aht20_test.c 和 aht20.c，编译烧录成功后，在串口查看运行结果。

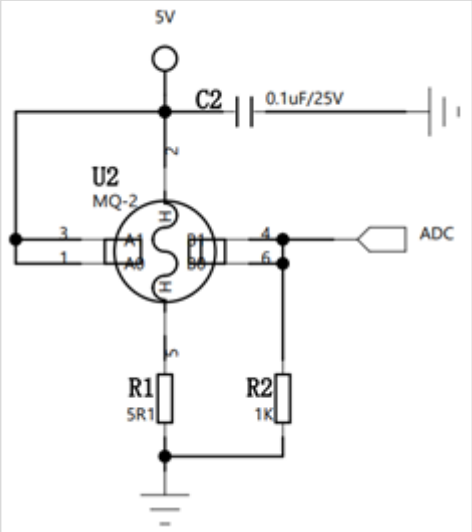
```
g_temperature is 28.34
g_humidity is 68.78
```

问题研讨

设置一个标准值，测量结果超出标准值则输出一条日志，告知超出标准范围 。

5.4.2.5 任务三 可燃气体监测

使用 MQ-2 可燃气体传感器进行环境中可燃气体的监测，MQ-2 气体传感器对丙烷、烟雾的灵敏度高，对天然气和其它可燃气体的检测也很理想。这种传感器可检测多种可燃性气体，是一款适合多种应用的低成本传感器。开发套件使用 ADC5（GPIO11 复用）通道采集 MQ-2 传感器电压。



步骤 1 读取 ADC 值

OpenHarmony 1.1.0 目前没有提供 ADC 接口，可以直接使用 Hi3861 提供的接口 hi_adc_read()。

代码参考 mq2_test.c 中的 EnvironmentTask。

```
hi_adc_read(GAS_SENSOR_CHAN_NAME, &data, HI_ADC_EQU_MODEL_4, HI_ADC_CUR_BAIS_DEFAULT, 0)
```

步骤 2 将获取的值转为烟雾浓度 ppm

根据电路图，根据电压计算阻值比，计算某种可燃气体的浓度。

代码参考 mq2_test.c 中的 EnvironmentTask。

```
//甲烷浓度计算
gasSensorResistance = 5 / Vx - 1;
g_gasValuetemp = 1 / (1 + gasSensorResistance) * (-2593) + 1078;
```

步骤 3 运行程序

修改 BUILD.gn 中的编译文件为 mq2_test.c，编译烧录，在串口查看运行结果。

```
g_gasValuetemp is 945.817749
```

问题研讨

设置一个标准值，测量结果超出标准值则输出一条日志，告知超出标准范围。

5.4.2.6 任务四 OLED 显示屏控制

开发套件搭载的是 0.96 寸的 OLED 显示屏，该显示屏有以下特点：0.96 寸 OLED 模块采用 SSD1306 驱动芯片，分辨率为 128 像素×64 像素。通信接口为 I2C，地址为 0x78，模块内带有稳压芯片，支持 3.3～5 V 电压供电。

步骤 1 OLED 初始化

开发套件中 ssd1306 显示屏驱动芯片通过 I2C 和主控芯片进行通信。

I2C 接口的两个引脚为 SCL 和 SDA，一个用于控制信号，另一个用于控制数据。

查找速查手册，可以了解引脚 GPIO13、GPIO14，分别对应 I2C0_SDA、I2C0_SCL。

在初始化之前需要先封装 IoTI2cWrite()接口，实现命令写入 WriteCmd()接口函数

OLED 初始化步骤：

1）将 GPIO13、GPIO14 引脚复用为 I2C0 功能；

2）调用 IoTI2cInit 接口初始化 Hi3861 I2C0，并设置 I2C0 的传输速率为 400 kbit/s；

3）将 SSD1306 驱动芯片初始化。主要通过 WriteCmd 函数依次向 SSD1306 驱动芯片发送 initCmds 命令。

代码参考 oled_ssd1306.c。

步骤 2 在 OLED 屏上绘制字符串

OLED 显示屏的显示原理

OLED 显示屏本身是没有显存的，它的显存依赖于 SSD1306 驱动芯片提供。SSD1306 驱动芯片内部有一个被称为 GDDRAM（Graphic Display Data RAM，图像显示数据内存，即通常简称的“显存”）的 SRAM，大小是 128×64 位，被分为 8 个页（Page），用于单色 128×64 点阵显示。当往 SSD1306 驱动芯片的 RAM 中写入数据时，相应的画面就会显示在 OLED 显示屏上。

在 oled_test.c 中创建一个任务，在 oled 屏上写出 Hello HarmonyOS!

```
static void OledTask(void *arg)
{
    (void)arg;
    //OLED 初始化及显示
    OledInit();

    OledFillScreen(0x00);
    OledShowString(0, 0, "Hello HarmonyOS!", 2);
    sleep(30);
}
```

步骤 3 运行程序

修改 BUILD.gn 中的编译文件为 oled_test.c 和 oled_ssd1306.c，编译烧录，在 OLED 屏幕上可以看到 Hello HarmonyOS!字样显示。

5.4.2.7 任务五 蜂鸣器报警

在环境监测过程中，当温、湿度或者烟雾浓度超过标准值时，可以使用蜂鸣器进行报警，那么如何让蜂鸣器发声呢?

步骤 1 通过输出 PWM 方波控制蜂鸣器发声

在 beep.c 中创建一个任务，通过输出 PWM 方波控制蜂鸣器发声

输出 PWM 方波步骤：

1、BEEP 与主板的 GPIO9 相连（PWM0），将 GPIO9 设置为 PWM0 功能；

2、PWM 初始化；

3、设置占空比与频率控制输出方波。

```
static void PwmBuzTask(void *arg)
{
    (void) arg;
    //输出 PWM 方波控制蜂鸣器发声
    IoTGpioInit(BEEP_PIN_NAME);
    hi_io_set_func(BEEP_PIN_NAME, BEEP_PIN_FUNCTION);
    IoTPwmInit(WIFI_IOT_PWM_PORT_PWM0);

    IoTPwmStart(WIFI_IOT_PWM_PORT_PWM0, 50, 4000);
    osDelay(100);
    IoTPwmStop(WIFI_IOT_PWM_PORT_PWM0);
}
```

步骤 2 打开 PWM 开关

HarmonyOS 的 LiteOS-M 内核默认将 PWM 功能关闭，需要手动设置 device\hisilicon\hispark_pegasus\sdk_liteos\build\config\usr_config.mk 文件中

CONFIG_PWM_SUPPORT is not set 改为 CONFIG_PWM_SUPPORT=y。

步骤 3 运行程序

修改 BUILD.gn 中的编译文件为 beep.c，编译烧录，可以听见蜂鸣器发声。

问题研讨

完成上述的所有任务之后，考虑如何一起完成下面这些任务：

1、使用环境监测板监测环境的温湿度以及可燃气体；

2、将测得数据在 OLED 屏上显示；

3、设置温湿度以及可燃气体浓度数值上下限，在超出范围时使用蜂鸣器进行报警。

5.4.2.8 任务六 WiFi 通信

步骤 1 添加网络通信代码

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 udpserver_env，将框架代码中 udpserver_env 中的文件拷贝到工程里。目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├── environment
                └── udpserver_env
                    ├── BUILD.gn
                    ├── cjson.c
                    ├── demo_entry_cmsis.c
                    ├── libudpserverstart.a
                    ├── net_common.h
                    ├── net_demo.h
                    ├── net_params.h
                    ├── udp_server_test.c
                    ├── wifi_connecter.c
                    ├── wifi_connecter.h
                    ├── wifi_starter.c
                    └── wifi_starter.h
```

编写 app 目录下的 BUILD.gn。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "environment:sensing_demo",
        "udpserver_env:net_demo",
    ]
}
```

步骤 2 业务逻辑

1、板端作为服务端；

2、板端作为热点，APP 连接该热点，将新的热点名称与密码下发；

3、板端接收到信息后，解析该信息，并将数据保存下来并重启设备；

4、板端获取保存下来的热点名称与密码，连接该热点，重新与 APP 建立连接；

5、接收 APP 下发的数据或者向 APP 发送数据；

6、板端接收到 APP 下发的消息、解析后，按照 APP 下发的命令进行对板端的控制；

7、结合前面的任务，实现通过手机端 APP 设置温湿度以及烟雾浓度的监控范围，板端使用传感器进行测量，并将结果显示在 OLED 屏上，如何超出设置的范围，则报警；

8、APP 端也可以远程查看数据，在进入环境监测界面 1 秒后自动更新数据，之后每隔 1 分钟更新一次；除此之外也可以按下更新按钮，同样可以查看当前环境下温湿度以及可燃气体浓度。

步骤 3 运行程序

1、修改 environment 下 BUILD.gn 中的编译文件为 environment_demo.c、oled_ssd1306.c 和 aht20.c，重新编译烧录。

2、复位开发板上的程序后，手机连接上开发板的热点“HarmonyOS-AP”,然后打开安装好的 APP，配置新的热点名称与密码，点击“配网”按键，将热点的名称与密码下发到板端，然后关闭 APP。

3、板端接收到 APP 下发的热点与密码之后会重启设备，并连接上配置好的热点，手机同样连接上配置的热点，再次打开 APP。板端与手机 APP 建立通信。

4、APP 切换到环境监测界面，在 1 秒之后会自动更新温湿度以及可燃气体浓度的值，之后会每隔 1 分钟更新一次数据，还可以点击 Update 按钮进行数据更新。

5、环境监测界面可以修改需要监测的数值范围（最大值和最小值要同时设置），在超出设置的范围时，蜂鸣器会响，进行报警。

5.4.3 演练场景 3：智能夜灯

背景

节能减排一直是全社会关注的一大热点问题，随之而来的智能环保产品也是层出不穷；本次实验案例可以应用到各种场景，比如说可以安装到家里，楼道里、作为小区路灯也是可以的。

在有人经过的无光环境下，三色灯会自动点亮，可以通过手机控制灯的亮度以及时间。

思考

1、可以用什么功能控制灯的亮度?

2、如何采集光敏电阻与人体红外传感器的值?

【参考答案】

1、PWM

2、ADC

任务一 创建综合实验工程

步骤 1 为智能夜灯综合实验创建一个工程

为了区分不同演练场景的案例，重新在./applications/sample/wifi-iot/app 路径下新建一个目录 night_light，将框架代码中 night_light 中的文件拷贝到工程里。目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                └──night_light
                  ├ ssd1306
                      ├ BUILD.gn
                      ├ ssd1306.c
                      ├ ssd1306.h
                      ├ ssd1306_conf.h
                      ├ ssd1306_fonts.c
                      ├ ssd1306_fonts.h
                  ├ BUILD.gn
                  ├ config_params.h
                  ├ night_light_demo.c
                  ├ night_light_demo.h
                  └ ssd1306test.c
```

步骤 2 编写 app 目录下的 BUILD.gn

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "night_light:night_light",
        "night_light/ssd1306:oled_ssd1306",
    ]
}
```

任务二 LED 亮度调节

步骤 3 传感器的控制与数据采集

1、将三色灯管脚设置为 PWM；

2、点亮 LED 灯；

3、使用 ADC 采集光敏电阻的值；

4、使用 ADC 采集人体红外传感器的值；

5、OLED 显示屏的控制；

参考 night_light 文件夹下的代码。

步骤 4 运行程序

开发板上更换炫彩灯扩展板，编译烧录，查看运行结果，可以实现无光有人的情况下自动亮灯。

任务三 WiFi 通信

步骤 1 添加网络通信代码

重新在./applications/sample/wifi-iot/app 路径下新建一个目录 udpserver_light，将框架代码中 udpserver_light 中的文件拷贝到工程里。目录结构如下：

```
.
└── applications
    └── sample
        └── wifi-iot
            └── app
                ├── night_light
                └── udpserver_light
                    ├── BUILD.gn
                    ├── cjson.c
                    ├── demo_entry_cmsis.c
                    ├── libudpserverstart.a
                    ├── net_common.h
                    ├── net_demo.h
                    ├── net_params.h
                    ├── udp_server_test.c
                    ├── wifi_connecter.c
                    ├── wifi_connecter.h
                    ├── wifi_starter.c
                    └── wifi_starter.h
```

编写 app 目录下的 BUILD.gn。

```
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
    features = [
        "night_light:night_light",
        "night_light/ssd1306:oled_ssd1306",
        "udpserver_light:net_demo",
    ]
}
```

步骤 2 业务逻辑

1、板端作为服务端；

2、板端作为热点，APP 连接该热点，将新的热点名称与密码下发；

3、板端接收到信息后，解析该信息，并将数据保存下来并重启设备；

4、板端获取保存下来的热点名称与密码，连接该热点，重新与 APP 建立连接；

5、接收 APP 下发的数据或者向 APP 发送数据；

6、板端接收到 APP 下发的消息、解析后，按照 APP 下发的命令进行对板端的控制；

7、结合任务二，实现通过手机端 APP 设置控制输出 PWM 方波的占空比控制 LED 的亮度，设置延时时间，控制灯的亮的时长；并将设置接口显示在 OLED 屏上。

步骤 3 运行程序

1、重新编译烧录；

2、复位开发板上的程序后，手机连接上开发板的热点“HarmonyOS-AP”,然后打开安装好的 APP，配置新的热点名称与密码，点击配网按键，将热点的名称与密码下发到板端，然后关闭 APP；

3、板端接收到 APP 下发的热点与密码之后会重启设备，并连接上配置好的热点，手机同样连接上配置的热点，再次打开 APP。板端与手机 APP 建立通信；

4、APP 切换到智能夜灯界面，可以设置 LED 的亮度以及实践，通过这两个参数控制 LED。实现在无光有人的情况下自动亮灯。

问题研讨

如何保存 app 侧下发到板端的数据呢?

5.5 案例总结

我的案例总结：

_____

_____

_____

6 附录：术语及缩略语

缩略语或术语	全称	描述
HPM	HarmonyOS Package Manager	HarmonyOS包管理工具
CMSIS	Cortex Microcontroller Software Interface Standard	微控制器软件接口标准
POSIX	Portable Operating System Interface	可移植操作系统接口
WLAN	Wireless Local Area Network	无线局域网
GPIO	General-purpose input/output	通用型之输入输出
I2C	Inter – Integrated Circuit	集成电路间总线
SPI	Serial Peripheral Interface	串行外设接口
AD	Analog to Digital Convert	模拟-数字信号转换
DA	Digital to Analog Convert	数字-模拟信号转换
IoT	Internet of Things	物联网
DFX	Design for X	面向产品生命周期各环节的设计
ROTS	Real Time Operating System	实时操作系统
API	Application Programming Interface	应用程序编程接口
MCU	Microcontroller Unit	微控制单元
MMU	Memory Management Unit	内存管理单元
UCOS	u control operation system	微型嵌入式实时系统
FAT	File Allocation Table	文件配置表
YAFFS2	Yet Another Flash File System	嵌入式文件系统

RAMFS	Ram File System	基于内存的文件系统
JFFS2	Journalling Flash File System Version2	闪存日志型文件系统第2版
NFS	Network Files System	网络文件系统
PROC	process	操作系统的/proc目录,即proc文件系统
IPC	Inter-Process Communication	进程间通信
VFS	virtual File System	虚拟文件系统
CPU	central processing unit	中央处理器
SCHED_RR	Schedule Round-Robin	时间片轮询调度
SCHED_FIFO	Schedule First Input First Output	先进先出调度
UDP	User Datagram Protocol	用户数据报协议
TCP	Transmission Control Protocol	传输控制协议
DIR	Directory	根目录区
DBR	Dos Boot Record	操作系统引导记录区
MBR	Master Boot Record	主引导分区
HDF	Hardware Driver Foundation	硬件驱动框架
LED	Light Emitting Diode	发光二极管
SD	Secure Digital	安全数字卡
HCS	HDF Configuration Source	HDF驱动框架的配置描述源码
HC-GEN	HDF Configuration Generator	HCS配置转换工具
UART	Universal Asynchronous Receiver/Transmitter	通用异步收发传输器
SDA	SerialData	串行数据线
SCL	SerialClock	串行时钟线
ACK	Acknowledge character	确认字符

SCLK	System clock	系统时钟信号
MISO	SPI Bus Master Input/Slave Output	SPI 总线主输入/从输出
MOSI	SPI Bus Master Output/Slave Input	SPI 总线主输出/从输入
SDIO	Secure Digital Input and Output	安全数字输入输出接口
GPS	Global Positioning System	全球定位系统
RTC	real-time clock	实时时钟
ADC	Analog to Digital Converter	模数转换器
PWM	Pulse Width Modulation	脉冲宽度调制
OTA	Over the Air	远程升级
Gn	Generate ninja	ninja生成器
FA	Feature Ability	代表有界面的Ability，用于与用户进行交互
IP	Internet Protocol	网际互连协议
SAMGR	/samgr	HarmonyOS系统服务框架子系统
OEM	Original Equipment Manufacturer	原始设备制造商
DMA	Direct Memory Access	直接存储器访问
AR	Augmented Reality	增强现实
VR	Virtual Reality	虚拟现实技术
FPS	Frames Per Second	帧速率
AI	Artificial Intelligence	人工智能
UDID	Unique Device Identifier	设备的唯一设备识别符
DFR	Design for Reliability	可靠性
DFT	Design for Testability	可测试性
XTS	/xts	HarmonyOS生态认证测试套件的集合

acts	application compatibility test suite	应用兼容性测试套件
AP	Access Point	无线接入点
BIOS	Basic Input Output System	基本输入输出系统
JTAG	Joint Test Action Group	联合测试工作组
GCC	GNU Compiler Collection	GNU编译器套件
GNU	GNU's Not Unix!	GNU是一个操作系统，其内容软件完全以通用公共许可证的方式发布