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。

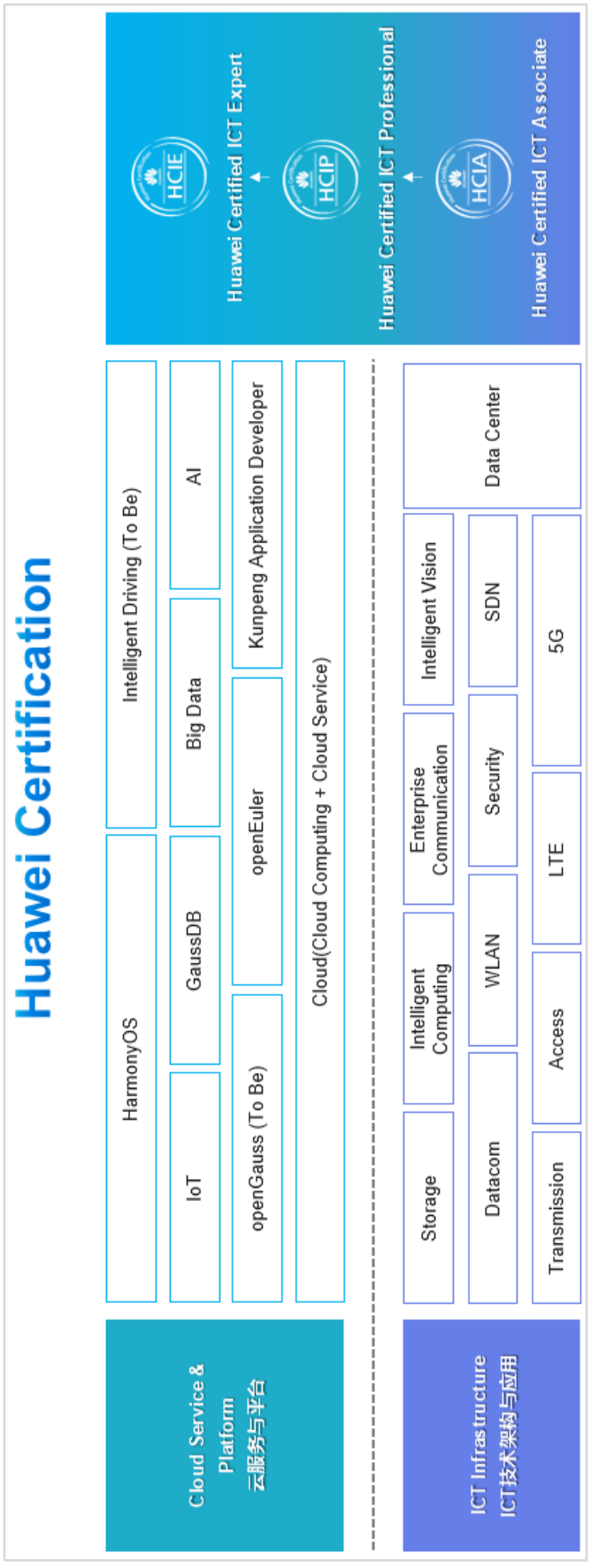
根据ICT从业者的学习和进阶需求，华为认证分为工程师级别、高级工程师级别和专家级别三个认证等级。

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

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，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、屏幕显示等场景，理解 HarmonyOS 设备开发驱动基础中的 GPIO、I2C、UART、PWM、ADC、SPI 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，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，通过智能小屋、护花使者、WiFi 智能时钟三个案例，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	BearPi-HM Nano	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?

【参考答案】

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　确定目录结构

开发者编写业务时，务必先在./applications/sample/BearPi-HM_Nano/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
        └── BearPi-HM_Nano
            └── app
                ├── my_first_app
                │   ├── hello_world.c
                │   └── BUILD.gn
```

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

步骤 2 编写业务代码

新建./applications/sample/BearPi-HM_Nano/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。并在代码最下方，使用 HarmonyOS 应用层功能的初始化启动接口 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)
{
    //延时 2 秒便于观察现象
    osDelay(200);

    /*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 编写用于将业务构建成静态库的 BUILD.gn 文件

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

如步骤 1 所述，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 写模块 BUILD.gn 文件，指定需参与构建的特性模块

配置./applications/sample/BearPi-HM_Nano/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/BearPi-HM_Nano/app/my_first_app/BUILD.gn。

myapp 是目标，指向./applications/sample/BearPi-HM_Nano/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，输入 . 选择路径为当前目录。

product 选择 bearpi_hm_nano@bearpi；

在源码路径下执行 hb build -f ；

运行效果如下：

```
 [OHOS INFO] bearpi_hm_nano build success
```

步骤 6 烧写固件

过程不再赘述，请参考《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\\bearpi_hm_nano\\bearpi_hm_nano\\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

按一下开发板 RESET 按键复位开发板，使用串口工具查看开发板日志，输出如下。

```
[DEMO] Hello world.
```

问题研讨

在任务一中，HarmonyOS 源码是如何运行我们指定的 App?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 producer_consumer，将框架代码 producer_consumer 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──producer_consumer
                │   ├──producer_consumer.c
```

```
│   └── BUILD.gn
└── BUILD.gn
```

步骤 2 编写 producer_consumer 目录下的 BUILD.gn

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 500 毫秒*/
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

按一下开发板 RESET 按键复位开发板，使用串口工具查看开发板日志，输出如下。

```
[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

1、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?

2、编译命令是什么?

【参考答案】

任务三　生产者消费者的同步

我们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。

步骤 1　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);
    }
}
```

步骤 2 查看运行结果

可以看到，我们创建了 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.
```

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。

【参考答案】

1.4.2 演练场景 2：打印机的使用

背景

在日常生活工作中，打印机是我们经常使用的设备，当正在使用打印机打印资料时（还没有打印完），别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 内核基础中哪个功能保证这个资源（打印机）不被同时使用?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 printer，将框架代码 printer 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──printer
                │   ├──printer.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 producer_consumer 目录下的 BUILD.gn

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 500 毫秒，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

按一下开发板 RESET 按键复位开发板，使用串口工具查看开发板日志，输出如下：

```
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 就急着也想打印，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(200)?

【参考答案】

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 20 毫秒*/
                {...}
            /*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 1000 毫秒，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);
}
```

编译烧录程序，按一下开发板 RESET 按键复位开发板，使用串口工具查看开发板日志，输出如下：

```
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?

【参考答案】

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?

【参考答案】

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/BearPi-HM_Nano/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，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──message
                │   ├──message.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 message 目录下的 BUILD.gn

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?

【参考答案】

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/BearPi-HM_Nano/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，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──timer2feed
                │   ├──timer2feed.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 timer 目录下的 BUILD.gn

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

按一下开发板 RESET 按键复位开发板，使用串口工具查看开发板日志，输出如下：

```
[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或者说优缺点?
2. 如何修改定时间隔?

【参考答案】

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);
```

编译烧录程序，按一下开发板 RESET 按键复位开发板，使用串口工具查看开发板日志，输出如下：

```
[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，不是更简单吗？

【参考答案】

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、ADC、SPI、UART 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?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 basic_led_blink，将框架代码中 basic_led_blink 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──basic_led_blink
                │   ├── led_example.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 basic_led_blink 目录下的 BUILD.gn

填入源文件“led_example.c”；

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

    include_dirs = [
        "//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 basic_led_blink:led_example。basic_led_blink 指得是目录，led_example 是指的是上面的静态库名称 led_example；

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

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

步骤 4　关键代码编写

参考如下：

```
    //初始化 GPIO_2，学员自行补充
    IoTGpioInit(LED_GPIO);
```

```
//设置 GPIO_2 为输出模式，学员自行补充
IoTGpioSetDir(LED_GPIO, IOT_GPIO_DIR_OUT);

while (1)
{
    switch (g_LedState) {
        case LED_ON:
            //设置 GPIO_2 输出高电平点亮 LED 灯，学员自行补充
            IoTGpioSetOutputVal(LED_GPIO, 1);
            osDelay(100);
            break;
        case LED_OFF:
            //设置 GPIO_2 输出低电平熄灭 LED 灯，学员自行补充
            IoTGpioSetOutputVal(LED_GPIO, 0);
            osDelay(100);
            break;
        case LED_SPARK:
            //设置 GPIO_2 交替输出高低电平，使 LED 闪烁，学员自行补充
            IoTGpioSetOutputVal(LED_GPIO, 0);
            osDelay(100);
            IoTGpioSetOutputVal(LED_GPIO, 1);
            osDelay(100);
            break;
        default:
            osDelay(100);
            break;
    }
}
```

步骤 5 编译烧录程序

按一下开发板的 RESET 按键复位开发板，查看运行结果，参考《 环境搭建部分 》或《 内核开发入门 》编译烧录程序。

烧写成功后可以看到，开发板上的蓝色 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?

【参考答案】

任务二　按键控制灯光状态

我们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 F1 GPIO 初始化，学员自行补充
      IoTGpioInit(Button_F1_GPIO);
      //按键 F1 GPIO 设置为输入方向，学员自行补充
      IoTGpioSetDir(Button_F1_GPIO, IOT_GPIO_DIR_IN);
      //按键 F1 GPIO 设置为上拉模式，学员自行补充
      IoTGpioSetPull(Button_F1_GPIO, IOT_GPIO_PULL_UP);
      //注册按键 F1  的中断回调函数
      IoTGpioRegisterIsrFunc(Button_F1_GPIO, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_LOW,
F1_Pressed, NULL);

      //按键 F2 GPIO 初始化，学员自行补充
      IoTGpioInit(Button_F2_GPIO);
      //按键 F2 GPIO 设置为输入方向，学员自行补充
      IoTGpioSetDir(Button_F2_GPIO, IOT_GPIO_DIR_IN);
      //按键 F2 GPIO 设置为上拉模式，学员自行补充
      IoTGpioSetPull(Button_F2_GPIO, IOT_GPIO_PULL_UP);
      //注册按键 F2  的中断回调函数
      IoTGpioRegisterIsrFunc(Button_F2_GPIO, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_LOW,
F2_Pressed, NULL);
```

编译烧录程序，按一下开发板的 RESET 按键复位开发板，查看运行结果。

按下按键 F1 可以打开灯，按键 F2 关闭灯。

问题研讨

GPIO 还有哪些用途?

【参考答案】

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，每次都需要爬上去去调节吗？显然我们有更好的方式：PWM。

思考

PWM 是什么，它还用在哪些场合?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 basic_pwm_led，将框架代码中 basic_pwm_led 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──basic_pwm_led
                │   ├──pwm_example.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 basic_pwm_led 目录下的 BUILD.gn

填入源文件“pwm_led_demo.c";

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

    include_dirs = [
        "//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 basic_pwm_led:pwm_example。basic_pwm_led 指得是目录，pwm_example 是指的是上面的静态库名称 pwm_example。

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

lite_component("app") {
    features = [
```

```
        "basic_pwm_led:pwm_example",
    ]
}
```

步骤 4 关键代码编写

参考如下，详细代码可参考完整代码：

```
    //初始化 PWM2 端口，学员自行补充
    IoTPwmInit(IOT_PWM_PORT_PWM2);

            //输出不同占空比的 PWM 波，学员自行补充
            IoTPwmStart(IOT_PWM_PORT_PWM2, i, 4000);
            usleep(1000);
```

步骤 5 编译烧录程序

按一下开发板的 RESET 按键复位开发板，查看运行结果：

编译过程中如果报错：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/bearpi/bearpi-hm_nano/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，开发板上的蓝色 LED 由暗到亮，达到“呼吸”的效果。

问题研讨

HarmonyOS 的 PWM 有哪些接口?

【参考答案】

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?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 basic_adc，将框架代码中 basic_adc 中的文件拷贝到工程里，如下图所示。

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──basic_adc
                │   ├──adc_example.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 basic_adc 目录下的 BUILD.gn

填入源文件“adc_example.c”；

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

    include_dirs = [
        "//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 basic_adc:adc_example，asic_adc 指的是目录，adc_example 是指的是上面的静态库名称 adc_example；

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

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

步骤 4　关键代码编写

参考如下：

```
//读取 ADC 数据，学员自行补充
ret = IoTAdcRead(6, &data, IOT_ADC_EQU_MODEL_8, IOT_ADC_CUR_BAIS_DEFAULT, 0xff);
if (ret != IOT_SUCCESS)
{
    printf("ADC Read Fail\n");
}
return (float)data * 1.8 * 4 / 4096.0;
```

步骤 5 编译烧录程序

插上基础扩展板，按一下开发板的 RESET 按键复位开发板，使用串口工具查看运行结果。程序开始运行，改变基础板光敏电阻周围环境的光，会发现串口打印的 ADC 电压会发生变化。串口打印结果如下：

```
=======================================
**************ADC_example*************
=======================================
vlt:0.200V
=======================================
**************ADC_example*************
=======================================
vlt:0.212V
=======================================
**************ADC_example*************
=======================================
vlt:1.255V
```

问题研讨

计算电压值的公式中 data * 1.8 * 4 / 4096.0。为什么是 4096?

【参考答案】

2.4.4 演练场景 4：气象监测

背景

现在的气象信息中，大多数是提供某个区域的平均温度、气压、湿度等数据，无法准确地反映出当前所处环境的一些气象数据。那么使用传感器就能够准确地测量当前环境的一些温度、气压等数据，并可以通过气压信息准确地得知当前的海拔高度。

思考

开发板中的气压传感器是如何把数据传输给主芯片?

HarmonyOS I2C 的 API 有哪些?

【参考答案】

任务一　获取温度、气压、海拔

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/BearPi-HM_Nano/app 路径下新建一个目录 basic_i2c，将框架代码中 basic_i2c 中的文件拷贝到工程里，如下图所示。BMP280 气压传感器支持 I2C 接口，这边不具体编写，直接拿来使用其驱动程序。

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──basic_i2c
                │   ├──bmp280.c
                │   ├──bmp280.h
                │   ├──i2c_example.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 basic_i2c 目录下的 BUILD.gn

填入源文件 ”p280.c”，”c_example.c”；

```
static_library("i2c_example") {
    sources = [
        "bmp280.c",
        "i2c_example.c"
    ]

    include_dirs = [
        "//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 basic_i2c:i2c_example，asic_i2c 指得是目录，i2c_example 是指的是上面的静态库名称 i2c_example；

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

lite_component("app") {
    features = [
        "basic_i2c:i2c_example",
    ]
}
```

步骤 4 关键代码编写

关键代码只提供关键技术点，实现请阅读完整代码：

```
//读取温度、气压、海拔数据，学员自行补充
while (!bmp280_read_float(&bmp280, &temperature, &pressure, &asl)) {
    printf("Temperature/pressure reading failed\n");
}
printf("Pressure: %.2f hPa, Temperature: %.2f C, Altiude: %.2f m\r\n",pressure, temperature,asl);
```

步骤 5 编译烧录程序

插上基础扩展板，按一下开发板的 RESET 按键复位开发板，使用串口工具查看运行结果。

编译过程中若报错：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/bearpi/bearpi-hm_nano/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。

```
=======================================
**********I2C_example**********
=======================================
Pressure: 1009.30 hPa, Temperature: 27.29 C, Altitude: 6.07 m
=======================================
**********I2C_example**********
=======================================
Pressure: 1009.29 hPa, Temperature: 27.31 C, Altitude: 6.17 m
=======================================
**********I2C_example**********
```

问题研讨

是否可以接多路支持 I2C 总线的硬件?

【参考答案】

2.4.5 演练场景 5：屏幕显示

背景

在现在许多智能产品中，都需要使用屏幕进行人机交互，例如显示设备工作状态、网络信息、电池电量、传感器信息、日期等信息。在小型的智能设备中通常会使用比如水墨屏、单色 OLED 屏、TFT-LCD 屏等屏幕，其中 TFT-LCD 屏具备显示彩色图像的能力，所以能够大大提升用户的交互体验。

思考

1、板中的主芯片是如何控制 LCD 屏幕显示数据的?

2、rmonyOS SPI 的 API 有哪些?

【参考答案】

任务一　显示温度、气压、海拔

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/BearPi-HM_Nano/app 路径下新建一个目录 basic_spi_lcd，将框架代码中 basic_spi_lcd 中的文件拷贝到工程里，如下图所示。屏幕的显示控制芯片为 st7789，使用 SPI 协议与主芯片交互数据，驱动程序这边不具体编写，直接拿来使用其驱动程序。

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──basic_spi_lcd
                │   ├──bmp280.c
                │   ├──bmp280.h
                │   ├──font.h
                │   ├──lcd_st7789.c
                │   ├──lcd_st7789.h
                │   ├──i2c_example.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 basic_spi_lcd 目录下的 BUILD.gn

填入源文件" bmp280.c", "lcd_st7789.c"," spi_lcd_example.c"。

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

```
        "bmp280.c",
        "lcd_st7789.c",
        "spi_lcd_example.c"
    ]

    include_dirs = [
        "//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 basic_spi_lcd:spi_example。basic_spi_lcd 指得是目录，spi_example 是指的是上面的静态库名称 spi_example。

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

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

步骤 4 关键代码编写

关键代码只提供关键技术点，实现请阅读完整代码：

```
    //初始化 LCD 屏幕，学员自行补充
    LCD_Init();

            //显示温度、气压、海拔数据，学员自行补充
            POINT_COLOR = BROWN;
            LCD_ShowString(10,30,240,32,24,"Welcome to BearPi!");
            sprintf(pressure_s,"Pressure:%.1f hPa          ",pressure);
            POINT_COLOR = RED;
            LCD_ShowString(5,80,240,32,24,pressure_s);
            sprintf(temperature_s,"Temperature:%.1f C          ",temperature);
            POINT_COLOR = MAGENTA;
            LCD_ShowString(5,130,240,32,24,temperature_s);
            sprintf(asl_s,"Altitude: %.1f m          ",asl);
            POINT_COLOR = BLUE;
            LCD_ShowString(5,180,240,32,24,asl_s);
```

步骤 5 编译烧录程序

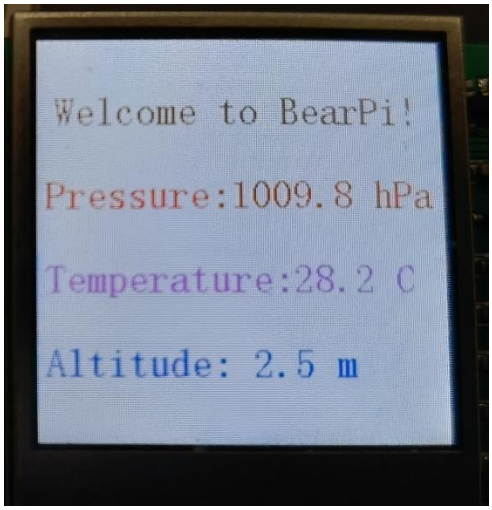
插上基础扩展板，按一下开发板的 RESET 按键复位开发板，可在屏幕上查看运行结果。

编译过程中若报错：undefined reference to hi_spi_write 等几个 hi_spi_开头的函数；

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

解决：修改 device/bearpi/bearpi-hm_nano/sdk_liteos/build/config/usr_config.mk 文件中的 CONFIG_SPI_SUPPORT 行：

CONFIG_SPI_SUPPORT is not set 修改为 CONFIG_SPI_SUPPORT=y



问题研讨

屏幕显示的字体大小是否可以改变?

【参考答案】

2.4.6 演练场景 6：数据通信

背景

数据的传输可以通过有线或无线进行传输，有线传输通常有串口 UART、485、CAN、USB 等形式。其中串口 UART 通常运用于两个 MCU 之间进行数据传输，一些较为复杂的传感器通常会有单独 MCU 进行数据采集处理，然后通过串口 UART 发送给其他 MCU 做更多的业务处理。

思考

1、MCU A 和 MCU B 之间的 RXD 和 TXD 怎么连接的?

2、HarmonyOS SPI 的 API 有哪些?

【参考答案】

任务一　串口数据读写

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/BearPi-HM_Nano/app 路径下新建一个目录 basic_uart，将框架代码中 basic_uart 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──basic_uart
                │   ├──uart_example.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 basic_uart 目录下的 BUILD.gn

填入源文件“uart_example.c”；

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

    include_dirs = [
        "//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 basic_uart:uart_example。basic_uart 指得是目录，uart_example 是指的是上面的静态库名称 uart_example。

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

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

步骤 4　关键代码编写

参考如下：

```
    //初始化串口 1，学员自行补充
    IotUartAttribute uart_attr = {
        //baud_rate: 9600
```

```
        .baudRate = 9600,

        //data_bits: 8bits
        .dataBits = 8,
        .stopBits = 1,
        .parity = 0,
    };
    ret = IoTUartInit(IOT_UART_IDX_1, &uart_attr);


    //通过串口 1 发送数据，学员自行补充
    IoTUartWrite(IOT_UART_IDX_1, (unsigned char *)data, strlen(data));

    //通过串口 1 接收数据，并打印出来，学员自行补充
    IoTUartRead(IOT_UART_IDX_1, uart_buff_ptr, UART_BUFF_SIZE);
    printf("Uart1 read data:%s", uart_buff_ptr);
```

步骤 5 编译烧录程序

插入基础扩展板，将扩展板上的串口调试开关拨到 ON 模式，复位开发板，使用串口工具查看运行结果。

实际上将基础扩展板的开关拨到 ON 模式，是将串口 1 的 TXD 与 RXD 接在一起，实现自发自收的测试。

```
=======================================
**********UART_example**********
=======================================
Uart1 read data:Hello, BearPi!
=======================================
**********UART_example**********
=======================================
Uart1 read data:Hello, BearPi!
```

问题研讨

串口的波特率是否可以改变?

【参考答案】

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 名字你记住了，但是密码忘记了，很多时候手机会帮你把 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?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 kv_store，将框架代码中 kv_store 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──kv_store
                │   ├──kv_store.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 kv_store 目录下的 BUILD.gn

填入源文件 “kv_store.c”；

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

    include_dirs = [
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//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 kv_store:kv_store_example。kv_store 指得是目录，kv_store_example 是指的是上面的静态库名称 kv_store_example。

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

lite_component("app") {
```

```
    features = [
        "kv_store:kv_store_example",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    //根据关键字（ssid），获取值 wifi1 账号名，学员自行补充
    if ((UtilsGetValue(ssid_key, buf, 512)) < 0) {
        printf("get value failed\n");

    }
    printf("---> get ssid_key's value is %s <---\n", buf);

    //根据关键字（pwd），获取值 wifi1 密码，学员自行补充
    if ((UtilsGetValue(pwd_key, buf, 512)) < 0) {
        printf("get value failed\n");

    }
    printf("---> get pwd_key's value is %s <---\n", buf);

    //设置关键字（ssid）的值，学员自行补充
    UtilsSetValue(ssid_key, ssid_value);

    //设置关键字（pwd）的值，学员自行补充
    UtilsSetValue(pwd_key, pwd_value);
```

步骤 5 编译烧录程序

按一下开发板的 RESET 按键复位开发板，使用串口工具查看运行结果。

烧录程序后第一次复位因为还未写入 KV 值所以会读取失败，后面几次复位后都可以直接读出 KV 值并打印出来。

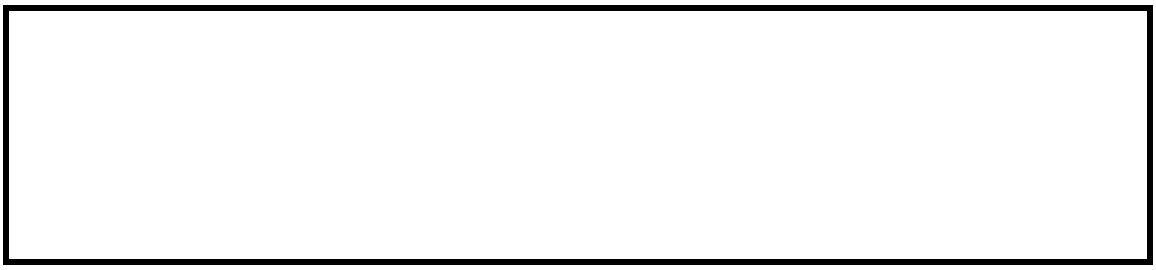
```
get value failed
---> get ssid_key's value is   <---
get value failed
---> get pwd_key's value is   <---

---> get ssid_key's value is Bearpi <---
---> get pwd_key's value is 0987654321 <---
```

问题研讨

KV 存储有哪些优点?

【参考答案】



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 系统中文件操作有接口?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 utils_file，将框架代码中 utils_file 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──utils_file
                │   ├──utils_file.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 utils_file 目录下的 BUILD.gn

填入源文件 “utils_file.c”；

```
static_library("utils_file_example") {
```

```
    sources = [
        "utils_file.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 utils_file:utils_file_example。utils_file 指得是目录，utils_file_example 是指的是上面的静态库名称 utils_file_example。

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

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

步骤 4 关键代码编写

参考如下：

```
    // 1.创建一个文件，若已存在该文件，则删除后重新创建，学员自行补充
    uint32_t fd = UtilsFileOpen(file_name, O_CREAT_FS | O_TRUNC_FS | O_RDWR_FS,NULL);
    if (fd < 0) {
        printf("open file failed\n");
        return;
    }
    // 2.读取该文件内容，学员自行补充
    if (UtilsFileRead(fd, (char *)buf, sizeof(buf)) < 0) {
        printf("read file failed\n");
        return;
    }
    printf("---> first read file content is %s <---\n", buf);

    // 3.向文件中写入内容，学员自行补充
    printf("---> write content to a file <---\n");
    if (UtilsFileWrite(fd, wifi_config,strlen(wifi_config)) < 0) {
        printf("write file failed\n");
        return;
    }
    // 4.关闭该文件，学员自行补充
    if (UtilsFileClose(fd) < 0) {
        printf("close file failed\n");
        return;
```

```
    }
    // 5.打开文件，学员自行补充
    fd = UtilsFileOpen("my_file", O_RDWR_FS,NULL);
    if (fd < 0) {
        printf("open file failed\n");
        return;
    }
    // 6.读取该文件内容，学员自行补充
    if (UtilsFileRead(fd, (char *)buf, sizeof(buf)) < 0) {
        printf("read file failed\n");
        return;
    }
    printf("---> second read file content is %s <---\n", buf);

    // 7.文件定位从第五个字节开始，学员自行补充
    if (UtilsFileSeek(fd, 5, SEEK_SET_FS) < 0) {
        printf("seek file failed\n");
        return;
    }
    // 8.读取该文件内容，学员自行补充
    if (UtilsFileRead(fd, (char *)buf, sizeof(buf)) < 0) {
        printf("read file failed\n");
        return;
    }
    printf("---> third read file content is %s <---\n", buf);

    // 9.关闭该文件，学员自行补充
    if (UtilsFileClose(fd) < 0) {
        printf("close file failed\n");
        return;
    }
    //10.删除文件,学员自行补充
    if(UtilsFileDelete(file_name) < 0){
        printf("delete file failed\n");
        return;
    }
    printf("---> delete file succeed <--- \n");
```

步骤 5 编译烧录程序

按一下开发板的 RESET 按键复位开发板，使用串口工具查看运行结果。

第一次读取文件时因还未写入数据，读出来的数据是空的；写入数据后，第二次就可以读取到数据；第三次读取因为设置从第五个字节开始读，所以前面的 ssid:这五个字节会跳过去。

```
---> first read file content is   <---
---> write content to a file <---
```

```
---> second read file content is ssid:Bearpi,pwd:0987654321 <---
---> third read file content is Bearpi,pwd:098765432154321 <---
```

问题研讨

为什么第三次读出来的数据是“Bearpi,pwd:0987654321”?

【参考答案】

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

1、WiFi 模块有哪些模式?

2、WiFi 模块有哪些 API 来支持我们开发?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 wifi_sta_connect，将框架代码中 wifi_sta_connect 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──wifi_sta_connect
```

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

步骤 2 编写 wifi_sta_connect 目录下的 BUILD.gn

填入源文件“wifi_sta_connect.c”；

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

    include_dirs = [
        "//foundation/communication/wifi_lite/interfaces/wifiservice",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 wifi_sta_connect:wifi_sta_connect_example。wifi_sta_connect 指得是目录，wifi_sta_connect_example 是指的是上面的静态库名称 wifi_sta_connect_example。

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

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

步骤 4 关键代码编写

参考如下：

```
// 配置 wifi 账号密码，学员自行补充
#define SELECT_WIFI_SSID "BearPi"
#define SELECT_WIFI_PASSWORD "0987654321"

    //初始化 WIFI，学员自行补充
    WiFiInit();

    //使能 WIFI，学员自行补充
    if (EnableWifi() != WIFI_SUCCESS)
    {
        printf("EnableWifi failed, error = %d\n", error);
        return -1;
    }
```

```
        //连接配置 WiFi 信息，学员自行补充
        if (AddDeviceConfig(&select_ap_config, &result) == WIFI_SUCCESS)
        {
            //连接指定的 WiFi，学员自行补充
            if (ConnectTo(result) == WIFI_SUCCESS && WaitConnectResult() == 1)
            {
                printf("WiFi connect succeed!\r\n");
                g_lwip_netif = netifapi_netif_find(SELECT_WLAN_PORT);
                break;
            }
        }
```

步骤 5 编译烧录程序

按一下开发板的 RESET 按键复位开发板，使用串口工具查看运行结果。

```
Select:   1 wireless, Waiting...
+NOTICE:CONNECTED
callback function for wifi connect
WaitConnectResult:wait success[1]s
WiFi connect succeed!
begain to dhcp<-- DHCP state:Inprogress -->
<-- DHCP state:Inprogress -->
<-- DHCP state:Inprogress -->
<-- DHCP state:OK -->
server :
    server_id : 192.168.43.1
    mask : 255.255.255.0, 1
    gw : 192.168.43.1
    T0 : 3600
    T1 : 1800
    T2 : 3150
clients <1> :
    mac_idx mac                addr              state    lease    tries    rto
    0       001131007b20       192.168.43.92     10       0        1        4
```

注意热点名称和密码必须和代码中写的一致，否则连不上 WiFi，并且 WiFi 只支持 2.4GHz。

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?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 wifi_ap，将框架代码中 wifi_ap 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──wifi_ap
                │   ├──wifi_ap.c
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 wifi_ap 目录下的 BUILD.gn

填入源文件“wifi_ap.c”；

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

    include_dirs = [
        "//foundation/communication/wifi_lite/interfaces/wifiservice",
    ]

}
```

步骤 3　编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 wifi_ap:wifi_ap_example。wifi_ap 指得是目录，wifi_ap_example 是指的是上面的静态库名称 wifi_ap_example。

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

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

步骤 4　关键代码编写

参考如下：

```
//配置 wifi 热点账号密码 ，学员自行补充
#define AP_SSID "BearPi"
```

```
#define AP_PSK   "0987654321"

    //设置指定的热点配置 ，学员自行补充
    HotspotConfig config = {0};

    strcpy(config.ssid, AP_SSID);
    strcpy(config.preSharedKey, AP_PSK);
    config.securityType = WIFI_SEC_TYPE_PSK;
    config.band = HOTSPOT_BAND_TYPE_2G;
    config.channelNum = 7;

    error = SetHotspotConfig(&config);
    if (error != WIFI_SUCCESS)
    {
        printf("SetHotspotConfig failed, error = %d.\r\n", error);
        return -1;
    }
```

3、编译烧录程序，按一下开发板的 RESET 按键复位开发板，使用串口工具查看运行结果。当有手机接入时会打印 New Sta Join，当手机断开连接时会打印 HotspotStaLeave。

```
RegisterWifiEvent succeed!
SetHotspotConfig succeed!
HotspotStateChanged:state is 1.
wifi hotspot active.
EnableHotspot succeed!
Wifi station is actived!
netifapi_netif_set_addr succeed!
netifapi_dhcps_start succeed!
Waiting to receive data...
+NOTICE:STA CONNECTED
New Sta Join
+NOTICE:STA DISCONNECTED
HotspotStaLeave: macAddress=62:AA:17:A8:3A:9D, reason=0.
```

问题研讨

代码中哪些关键配置决定开发板的 WiFi 热点信息?

【参考答案】

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。

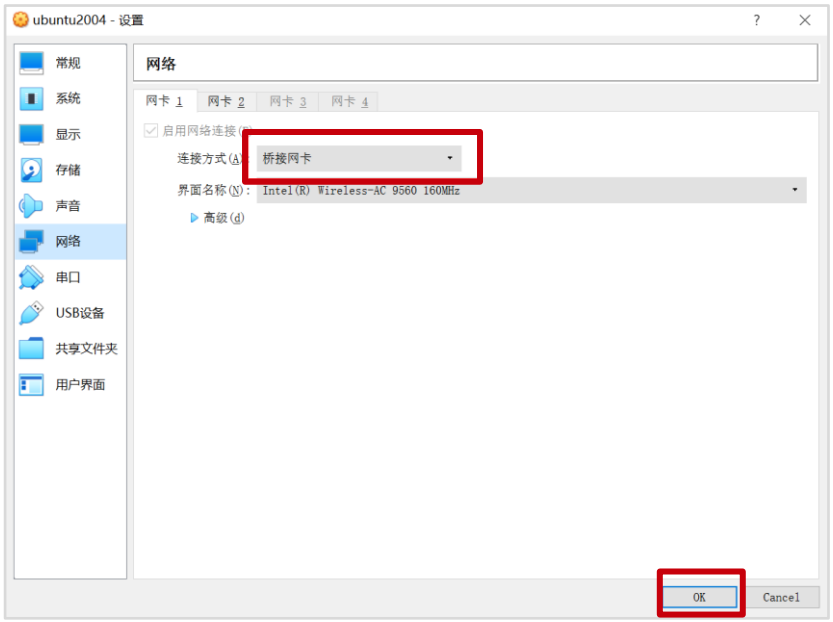
思考

WiFi 有几种模式?

【参考答案】

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。例如设置热点名字 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?

【参考答案】

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， 没错，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?

【参考答案】

任务三　搭建 TCP 客户端

使用开发板作为 TCP 客户端，

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/BearPi-HM_Nano/app 路径下新建一个目录 tcp_client，将框架代码中 tcp_client 中的文件拷贝到工程里，如下所示：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                ├──tcp_client
                │   ├──tcp_client_demo.c
                │   ├──wifi_connect.c
                │   ├──wifi_connect.h
                │   └── BUILD.gn
                └── BUILD.gn
```

步骤 2　编写 tcp_client 目录下的 BUILD.gn

填入源文件 “tcp_client_demo.c","wifi_connect.c”；

```
static_library("tcp_client_example") {
    sources = [
        "tcp_client_demo.c",
        "wifi_connect.c"
    ]

    include_dirs = [
        "//foundation/communication/wifi_lite/interfaces/wifiservice",
        "include"
```

```
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

自行修改 features 字段中的 tcp_client:tcp_client_example。tcp_client 指得是目录，tcp_client_example 是指的是上面的静态库名称 tcp_client_example。

```
import("//build/lite/config/component/lite_component.gni")
lite_component("app") {
    features = [
        "tcp_client:tcp_client_example",
    ]
}
```

步骤 4 关键代码编写

参考如下：

```
    //创建 socket，学员自行补充
    if((sock_fd = socket(AF_INET, SOCK_STREAM, 0)) == -1)
    {
        printf("create socket failed!\r\n");
        exit(1);
    }
    //初始化预连接的服务端地址，学员自行补充
    send_addr.sin_family = AF_INET;
    send_addr.sin_port = htons(CONFIG_SERVER_PORT);
    send_addr.sin_addr.s_addr = inet_addr(CONFIG_SERVER_ADDR);
    addr_length = sizeof(send_addr);
```

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 tcp_client_demo.c 文件的相关代码：

CONFIG_WIFI_SSID 修改为自己的热点名称；

CONFIG_WIFI_PWD 修改为自己的热点密码；

CONFIG_SERVER_ADDR 修改为自己的 Linux 主机 IP 地址。

```
// 配置 wifi 账号密码，学员自行补充
#define CONFIG_WIFI_SSID "Huawei"
#define CONFIG_WIFI_PWD "12345678"
// 配置服务器 IP 和端口 ，学员自行补充
#define CONFIG_SERVER_ADDR "192.168.43.190"
#define CONFIG_SERVER_PORT 5678
```

步骤 5 编译烧写，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! I'm BearPi-HM_Nano TCP Client!，输入 Hello! I'm TCP Server! 并回车，消息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
Hello! I'm BearPi-HM_Nano TCP Client!
```

开发板：

```
Select:   1 wireless, Waiting...
+NOTICE:CONNECTED
callback function for wifi connect
WaitConnectResult:wait success[1]s
WiFi connect succeed!
began to dhcp<-- DHCP state:Inprogress -->
<-- DHCP state:Inprogress -->
<-- DHCP state:Inprogress -->
<-- DHCP state:OK -->
server :
    server_id : 192.168.43.1
    mask : 255.255.255.0, 1
    gw : 192.168.43.1
    T0 : 3600
    T1 : 1800
    T2 : 3150
clients <1> :
    mac_idx mac                 addr              state    lease    tries    rto
    0       001131007b20        192.168.43.92     10       0        1        4
recv:Hello! I'm TCP Server!
```

问题研讨

在案例中，开发板充当的是 TCP 服务端还是客户端?

【参考答案】

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?

【参考答案】

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/BearPi-HM_Nano/app 路径下新建一个目录 library_test，把我们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
        └── BearPi-HM_Nano
            └── 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/BearPi-HM_Nano/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
   osDelay(200);
   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？

【参考答案】

4.5 案例总结

我的案例总结：

_____

_____

_____

5 综合实验

5.1 课程介绍

本章综合实验将使用 HarmonyOS 开发板来模拟实现家庭中的一些智能设备。

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

随着智能设备地快速发展，许多家庭中家用电器都实现了智能化，能感知环境的变化及实现远程控制。本章综合实验将使用 HarmonyOS 开发板来模拟实现家庭中的一些智能设备。

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：智能小屋

背景

当你准备入睡时，发现房间的灯和风扇还没关闭，但是这时你已经有了睡意，不想起来关闭这些设备了，此时若能对这些设备进行远程控制，是不是就很方便了，本节实验将使用 HarmonyOS 开发板来模拟一个智能小屋，能实现对灯和风扇的远程控制。

思考

智能小屋需要用到哪些知识点?

【参考答案】

任务一　创建综合实验工程

步骤 1　为智能小屋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/BearPi-HM_Nano/app 路径下新建一个目录 smart_home，将框架代码中 smart_home 中的文件拷贝到工程里，目录结构如下：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                └──────smart_home
                    ├── nfc
                    ├── profile_package
                    ├── sensor
                    ├── wifi
                    ├── key
                    ├── BUILD.gn
                    ├── app_report_message.c
                    ├── app_deal_command.c
                    ├── app_main.c
                    └── config_property.c
```

步骤 2　编写 smart_home 目录下的 BUILD.gn，用于生成 smart_home 静态库

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

    ]
    include_dirs = [
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//foundation/communication/wifi_lite/interfaces/wifiservice",
        "//device/bearpi/bearpi_hm_nano/sdk_liteos/include",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

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

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

问题研讨

smart_home 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?

【参考答案】

```
    sources = [
        "app_main.c",
        "app_deal_command.c",
        "app_report_message.c",
        "config_property.c",
        "wifi/wifi_connect.c",
        "sensor/E53_IA1.c",
        "nfc/NT3H.c",
        "nfc/nfc.c",
        "nfc/ndef/rtd/nfcForum.c",
        "nfc/ndef/rtd/rtdText.c",
        "nfc/ndef/rtd/rtdUri.c",
        "nfc/ndef/ndef.c",
        "profile_package/profile_package.c"
    ]
    include_dirs = [
        "//third_party/cJSON",
        "nfc/ndef",
        "nfc/ndef/rtd/",
        "nfc",
        "sensor",
        "wifi",
        "profile_package",
    ]
```

任务二 补充业务代码

为智能小屋综合实验添加业务代码。

步骤 1 增加关键代码

在 app_main.c 中实现外设初始化、创建 TCP 服务器、命令处理任务以及数据上报任务。

在 main_task 函数中添加以下代码：

```
    //外设初始化,学员自行补充
    borad_init();

    // 创建 TCP 服务器,学员自行补充
    if(creat_tcp_server() == 0)
    {

        // 创建命令处理任务,学员自行补充
        attr.name = "deal_command_task";
        attr.priority = 24;
        if (osThreadNew((osThreadFunc_t)deal_command_task, NULL, &attr) == NULL)
        {
            printf("Falied to create deal_command_task!\n");
        }
        //创建数据上报任务,学员自行补充
        attr.name = "report_message_task";
        attr.priority = 24;
        if (osThreadNew((osThreadFunc_t)report_message_task, NULL, &attr) == NULL)
        {
            printf("Falied to create report_message_task!\n");
        }
    }
```

在 app_report_message.c 的 deal_report_msg 函数中添加数据上报代码：

```
    //配置要发送的数据,学员自行补充
    service.event_time = NULL;
    service.service_id = "Smart_Home";
    service.service_property = &lux;
    service.nxt = NULL;

    lux.key = "Lux";
    lux.value = &report->lux;
    lux.type = PROFILE_VALUE_INT;
    lux.nxt = &temperature;

    temperature.key = "Temperature";
    temperature.value = &report->temp;
    temperature.type = PROFILE_VALUE_INT;
    temperature.nxt = &humidity;

    humidity.key = "Humidity";
    humidity.value = &report->hum;
    humidity.type = PROFILE_VALUE_INT;
    humidity.nxt = &motor;

    motor.key = "Motor";
```

```
    motor.value = motor_status?"ON":"OFF";
    motor.type = PROFILE_VALUE_STRING;
    motor.nxt = &light;

    light.key = "Light";
    light.value = light_status?"ON":"OFF";
    light.type = PROFILE_VALUE_STRING;
    light.nxt = NULL;

    //打包数据,学员自行补充
    msg = profile_package_propertyreport(&service);

    //发送数据,学员自行补充
    if(send(new_fd, msg, strlen(msg), 0) < 0)
    {
        printf("send error\r\n");
        close(new_fd);
    }
```

在 app_deal_command.c 的 deal_command_task 函数中添加命令处理代码：

```
            //接收数据,并根据命令对相应的设备执行控制命令，学员自行补充
            if (recv(new_fd, recvbuf, sizeof(recvbuf), 0) < 0)
            {
                printf("recv error\r\n");
                close(new_fd);
                is_accepted = 0;
                break;
            }
            else
            {
                if (NULL != strstr(recvbuf, "Motor"))
                {
                    //开启风扇
                    if (NULL != strstr(recvbuf, "ON"))
                    {
                        Motor_StatusSet(ON);
                        motor_status = 1;
                        printf("Motor On!\r\n");
                    }
                    //关闭风扇
                    else if (NULL != strstr(recvbuf, "OFF"))
                    {
                        Motor_StatusSet(OFF);
                        motor_status = 0;
                        printf("Motor Off!\r\n");
                    }
                }
```

```
            else if (NULL != strstr(recvbuf, "Light"))
            {
                //开启灯
                if (NULL != strstr(recvbuf, "ON"))
                {
                    Light_StatusSet(ON);
                    light_status = 1;
                    printf("Light On!\r\n");
                }
                //关闭灯
                else if (NULL != strstr(recvbuf, "OFF"))
                {
                    Light_StatusSet(OFF);
                    light_status = 0;
                    printf("Light Off!\r\n");
                }
            }
        }
```

步骤 2 编译烧录

运行程序，开发板会去连接指点的 WiFi 热点，提示当前打印要连接的 WiFi 不存在。

```
<--System Init-->
<--Wifi Init-->
register wifi event succeed!
callback function for wifi scan:0, 0
+NOTICE:SCANFINISH
callback function for wifi scan:1, 16
WaitSacnResult:wait success[1]s
********************
no:001, ssid:Huawei-Employee                    , rssi:   -58
no:002, ssid:Huawei-Employee                    , rssi:   -68
********************
ERROR: No wifi as expected
```

问题研讨

连接哪个 WiFi 热点，在代码中哪里体现?

【参考答案】

修改 app_main.c 中的 WiFi 账号密码，可以改成开发板连接的热点。

```
// 配置 wifi 账号密码，学员自行补充
#define CONFIG_WIFI_SSID "Huawei"
#define CONFIG_WIFI_PWD "12345678"
```

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

应用代码下载链接：https://gitee.com/bearpi/HCIA-HarmonyOS-FA

智能小屋案例使用 smart_home 工程，护花使者使用 smart_watering 工程。

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
- 修改 entry\src\main\js\default\pages.index\index.js 中的 bundleName，保持和 config.json 中一致
- 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，热点名称和密码设置为与 app_main.c 中对应的名称和密码。

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：

```
Select:   1 wireless, Waiting...
+NOTICE:CONNECTED
WaitConnectResult:wait success[1]s
WiFi connect succeed!
begain to dhcp
<-- DHCP state:Inprogress -->
<-- DHCP state:OK -->
server :
    server_id : 192.168.43.1
    mask : 255.255.255.0, 1
    gw : 192.168.43.1
    T0 : 3600
    T1 : 1800
    T2 : 3150
```

```
clients <1> :
    mac_idx mac                addr              state    lease    tries    rto
    0       080002b5769a       192.168.43.202    10       0        1        4
start accept
```

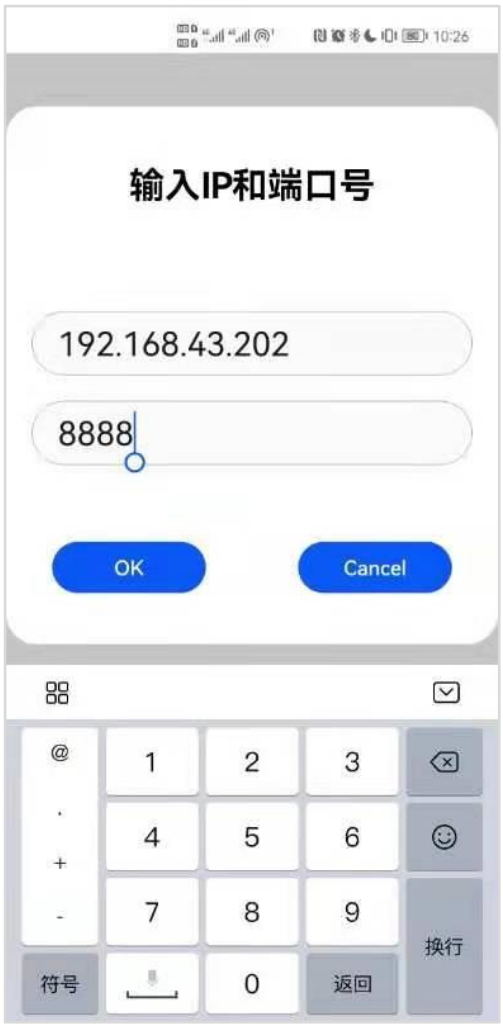
开发板连接 WiFi 成功。

步骤 3 手机和开发板互通

手机连接相同 WiFi

在上面日志中查看开发板 IP 为 192.168.43.202。

打开手机端的智能小屋 APP，点击右上角的加号按钮，输入开发板的 IP 和开放的端口，点击 OK 后会自动连接开发板。



开发板的日志中会打印：

```
start accept
SENSOR:lux:430.00 temp:27.36 hum:78.99
SENSOR:lux:430.00 temp:27.36 hum:78.64
SENSOR:lux:436.67 temp:27.39 hum:78.48
```

```
SENSOR:lux:437.50 temp:27.40 hum:78.41
SENSOR:lux:439.17 temp:27.41 hum:78.24
SENSOR:lux:440.00 temp:27.43 hum:78.13
SENSOR:lux:437.50 temp:27.48 hum:78.16
SENSOR:lux:438.33 temp:27.47 hum:78.04
accept succeed
```

至此，手机可以控制开发板打开或关闭风扇和电灯了。

问题研讨

手机端连接板子，输入的端口号与代码中哪个地方对应?

【参考答案】

与 app_main.c 中的 CONFIG_SERVER_PORT 定义的参数对应。

```
// 配置 TCP 服务端开放端口，学员自行补充
#define CONFIG_SERVER_PORT 5678
```

5.4.2 演练场景 2：护花使者

背景

植物需要合适的水分才能健康生长，水分过多或者过少都会影响植物的生长。如果能实现监控土壤的湿度并手机控制浇水，这样就可以实时呵护家中植物的健康成长。本节实验将使用 HarmonyOS 开发板来模拟一个护花使者，能实现监测土壤湿度和控制抽水电机浇水。

思考

智能小屋需要用到哪些知识点?

【参考答案】

任务一　创建综合实验工程

步骤 1　为智能小屋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/BearPi-HM_Nano/app 路径下新建一个目录 smart_watering，将框架代码中 smart_watering 中的文件拷贝到工程里，目录结构如下：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                └──────smart_watering
                    ├── nfc
                    ├── profile_package
                    ├── sensor
                    ├── wifi
                    ├── key
                    ├── BUILD.gn
                    ├── app_report_message.c
                    ├── app_deal_command.c
                    ├── app_main.c
                    └── config_property.c
```

步骤 2　编写 smart_watering 目录下的 BUILD.gn，用于生成 smart_watering 静态库

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

```
    ]
    include_dirs = [
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//foundation/communication/wifi_lite/interfaces/wifiservice",
        "//device/bearpi/bearpi_hm_nano/sdk_liteos/include",
    ]
}
```

步骤 3 编写 app 目录下的 BUILD.gn

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

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

问题研讨

smart_home 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?

【参考答案】

```
    sources = [
        "app_main.c",
        "app_deal_command.c",
        "app_report_message.c",
        "config_property.c",
        "wifi/wifi_connect.c",
        "sensor/flower.c",
        "nfc/NT3H.c",
        "nfc/nfc.c",
        "nfc/ndef/rtd/nfcForum.c",
        "nfc/ndef/rtd/rtdText.c",
        "nfc/ndef/rtd/rtdUri.c",
        "nfc/ndef/ndef.c",
        "profile_package/profile_package.c"
    ]
    include_dirs = [
        "//third_party/cJSON",
        "nfc/ndef",
        "nfc/ndef/rtd/",
        "nfc",
        "sensor",
        "wifi",
        "profile_package",
    ]
```

任务二 补充业务代码

为护花使者综合实验添加业务代码。

步骤 1 增加关键代码

在 app_main.c 中实现外设初始化、创建 TCP 服务器、命令处理任务以及数据上报任务。

在 main_task 函数中添加以下代码：

```
//外设初始化,学员自行补充
borad_init();

// 创建 TCP 服务器,学员自行补充
if(creat_tcp_server() == 0)
{

    // 创建命令处理任务,学员自行补充
    attr.name = "deal_command_task";
    attr.priority = 24;
    if (osThreadNew((osThreadFunc_t)deal_command_task, NULL, &attr) == NULL)
    {
        printf("Falied to create deal_command_task!\n");
    }
    //创建数据上报任务,学员自行补充
    attr.name = "report_message_task";
    attr.priority = 24;
    if (osThreadNew((osThreadFunc_t)report_message_task, NULL, &attr) == NULL)
    {
        printf("Falied to create report_message_task!\n");
    }
}
```

在 app_report_message.c 的 deal_report_msg 函数中添加数据上报代码：

```
//配置要发送的数据,学员自行补充
service.event_time = NULL;
service.service_id = "AutoWater";
service.service_property = &smo;
service.nxt = NULL;

smo.key = "Soil_Moisture";
smo.value = &report->smo;
smo.type = PROFILE_VALUE_INT;
smo.nxt = &temperature;

temperature.key = "Temperature";
temperature.value = &report->temp;
temperature.type = PROFILE_VALUE_INT;
temperature.nxt = &humidity;
```

```
    humidity.key = "Humidity";
    humidity.value = &report->hum;
    humidity.type = PROFILE_VALUE_INT;
    humidity.nxt = &motor;

    motor.key = "Motor";
    motor.value = motor_status?"ON":"OFF";
    motor.type = PROFILE_VALUE_STRING;
    motor.nxt = NULL;

    //打包数据,学员自行补充
    msg = profile_package_propertyreport(&service);
    //发送数据,学员自行补充
    if(send(new_fd, msg, strlen(msg), 0) < 0)
    {
        printf("send error\r\n");
        close(new_fd);
    }
```

在 app_deal_command.c 的 deal_command_task 函数中添加命令处理代码：

```
            //接收数据,并根据命令对相应的设备执行控制命令，学员自行补充
            if (recv(new_fd, recvbuf, sizeof(recvbuf), 0) < 0)
            {
                printf("recv error\r\n");
                close(new_fd);
                is_accepted = 0;
                break;
            }
            else
            {
                if (NULL != strstr(recvbuf, "Motor"))
                {
                    //开启抽水电机
                    if (NULL != strstr(recvbuf, "ON"))
                    {
                        Motor_StatusSet(ON);
                        motor_status = 1;
                        printf("Motor On!\r\n");
                    }
                    //关闭抽水电机
                    else if (NULL != strstr(recvbuf, "OFF"))
                    {
                        Motor_StatusSet(OFF);
                        motor_status = 0;
                        printf("Motor Off!\r\n");
                    }
                }
```

```
        }
```

步骤 2 编译烧录

运行程序，开发板会去连接指点的 WiFi 热点，提示当前打印要连接的 WiFi 不存在。

```
<--System Init-->
<--Wifi Init-->
register wifi event succeed!
callback function for wifi scan:0, 0
+NOTICE:SCANFINISH
callback function for wifi scan:1, 16
WaitSacnResult:wait success[1]s
********************
no:001, ssid:Huawei-Employee                    , rssi:   -58
no:002, ssid:Huawei-Employee                    , rssi:   -68
********************
ERROR: No wifi as expected
```

问题研讨

连接哪个 WiFi 热点，在代码中哪里体现?

【参考答案】

修改 app_main.c 中的 WiFi 账号密码，可以改成开发板连接的热点。

```
// 配置 wifi 账号密码，学员自行补充
#define CONFIG_WIFI_SSID "Huawei"
#define CONFIG_WIFI_PWD "12345678"
```

任务三 实验效果演示

准备一个路由器，没有的话，电脑或者手机当热点，热点名称和密码设置为与 app_main.c 中对应的名称和密码。

步骤 1 烧录编译生成的二进制文件

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：

```
Select:   1 wireless, Waiting...
+NOTICE:CONNECTED
WaitConnectResult:wait success[1]s
WiFi connect succeed!
began to dhcp
<-- DHCP state:Inprogress -->
<-- DHCP state:OK -->
server :
    server_id : 192.168.43.1
    mask : 255.255.255.0, 1
    gw : 192.168.43.1
    T0 : 3600
    T1 : 1800
```

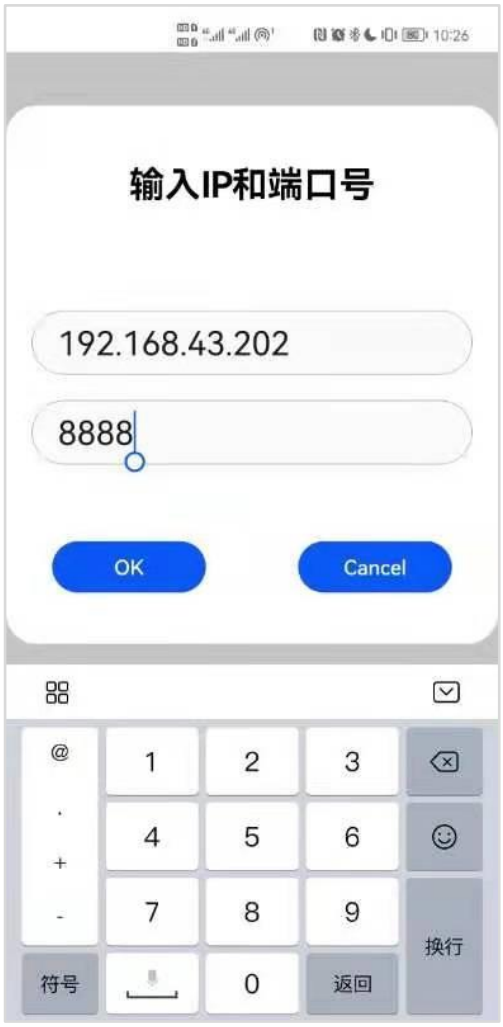
```
    T2 : 3150
clients <1> :
    mac_idx mac                addr              state    lease    tries    rto
    0       080002b5769a       192.168.43.202    10       0        1        4
start accept
```

开发板连接 WiFi 成功。

步骤 2 手机连接相同 WiFi

在 window 热点中查看开发板 IP 为 192.168.43.202。

打开手机端的护花使者 APP，点击右上角的加号按钮，输入开发板的 IP 和开放的端口，点击 OK 后会自动连接开发板。



开发板的日志中会打印：

```
start accept
SENSOR:smo:7.00 temp:27.36 hum:78.99
SENSOR:smo:7.00 temp:27.36 hum:78.64
SENSOR:smo:7.67 temp:27.39 hum:78.48
SENSOR:smo:7.50 temp:27.40 hum:78.41
SENSOR:smo:7.17 temp:27.41 hum:78.24
```

```
SENSOR:smo:7.00 temp:27.43 hum:78.13
SENSOR:smo:7.33 temp:27.47 hum:78.18
accept succeed
```

至此，手机可以控制开发板打开抽水电机了

问题研讨

手机端连接板子，输入的端口号与代码中哪个地方对应?

【参考答案】

与 app_main.c 中的 CONFIG_SERVER_PORT 定义的参数对应。

```
// 配置 TCP 服务端开放端口，学员自行补充
#define CONFIG_SERVER_PORT 5678
```

5.4.3 演练场景 3：WiFi 智能时钟

背景

传统的时钟功能单一，多数只能显示时间和日期，且运行久了后时间会出现偏差。本节实验将使用 HarmonyOS 开发板来开发一款 WiFi 智能时钟，具备网络授时和天气信息显示。

思考

WiFi 智能时钟需要用到哪些知识点?

【参考答案】

获取天气信息;

WiFi 操作;

Socket 编程。

任务一　创建综合实验工程

步骤 1　为 WiFi 智能时钟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/BearPi-HM_Nano/app 路径下新建一个目录 smart_clock，将框架代码中 smart_clock 中的文件拷贝到工程里，如下图所示，目录结构如下：

```
.
└── applications
    └── sample
        └── BearPi-HM_Nano
            └── app
                └──────smart_clock
                    ├── font
                    ├── lcd
                    ├── sensor
                    ├── wifi
                    ├── BUILD.gn
                    ├── app_background.c
                    ├── app_main.c
                    ├── app_sensor.c
                    ├── app_time.c
                    └── app_weather.c
```

步骤 2　编写 smart_clock 目录下的 BUILD.gn，用于生成 smart_clock 静态库

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

    ]
    include_dirs = [
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//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 = [
        "smart_clock:smart_clock",
    ]
}
```

问题研讨

smart_home 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?

【参考答案】

```
    sources = [
        "app_main.c",
        "app_deal_command.c",
        "app_report_message.c",
        "config_property.c",
        "wifi/wifi_connect.c",
        "sensor/flower.c",
        "nfc/NT3H.c",
        "nfc/nfc.c",
        "nfc/ndef/rtd/nfcForum.c",
        "nfc/ndef/rtd/rtdText.c",
        "nfc/ndef/rtd/rtdUri.c",
        "nfc/ndef/ndef.c",
        "profile_package/profile_package.c"
    ]
    include_dirs = [
        "sensor",
        "wifi",
        "lcd",
        "lcd/GUI",
        "lcd/GUI/lvgl",
        "lcd/GUI/lvgl_driver",
        "//third_party/cJSON",
    ]
```

任务二 补充业务代码

为 WiFi 智能时钟综合实验添加业务代码。

步骤 1 增加关键代码

在 app_main.c 中实现连接 Wifi,创建背景显示任务、传感器数据采集并显示任务、时间数据获取并显示任务、天气数据获取并显示任务。

在 APP_TaskCreate 函数中添加以下代码：

```
    // 配置 wifi 账号密码，学员自行补充
    #define CONFIG_WIFI_SSID "Huawei"
    #define CONFIG_WIFI_PWD "12345678"
```

```
//创建背景显示任务,学员自行补充
 if (osThreadNew((osThreadFunc_t)app_background_task, NULL, &attr) == NULL)
 {
      printf("Falied to create app_background_task!\n");
 }

 //创建时间数据获取并显示任务,学员自行补充
 attr.name = "app_time_task";
 attr.stack_size = 1024*4;
 if (osThreadNew((osThreadFunc_t)app_time_task, NULL, &attr) == NULL)
 {
      printf("Falied to create time task!\n");
 }

 //创建天气数据获取并显示任务,学员自行补充
 attr.stack_size = 1024*10;
 attr.name = "app_weather_task";
 if (osThreadNew((osThreadFunc_t)app_weather_task, NULL, &attr) == NULL)
 {
      printf("Falied to create time task!\n");
 }

 //创建传感器数据采集并显示任务,学员自行补充
 attr.name = "app_sensor_task";
 if (osThreadNew((osThreadFunc_t)app_sensor_task, NULL, &attr) == NULL)
 {
      printf("Falied to create time task!\n");
 }
```

步骤 2 在 app_background.c 的中添加显示动画代码

```
//显示动画,学员自行补充
for(int i=0 ;i <27;i++){
LCD_Show_Image(86, 117, 62, 62, gImage_taikongren[i]);
osDelay(8);
}
```

步骤 3 在 app_sensor.c 的中添加显示气压海拔数据代码

```
//显示气压海拔，学员自行补充
sprintf(pressure_show,"%d hPa", (int)pressure);
lv_label_set_static_text(label_pressure,pressure_show);
sprintf(asl_show,"%d m", (int)asl);
lv_label_set_static_text(label_asl,asl_show);
```

步骤 4 在 app_time.c 的中添加显示时间代码

```
//显示时间,学员自行补充
sprintf(hi3861softrtc_time,"%u:%02d:%02d",
hi3861softrtc.hour,hi3861softrtc.minute,hi3861softrtc.second);
lv_label_set_static_text(time_label,hi3861softrtc_time);
lv_label_set_static_text(data_label,hi3861softrtc_data);
lv_label_set_static_text(week_label,week_text);
osDelay(100);
hi3861softrtc.second +=1;
```

在 app_weather.c 的中添加显示动画代码：

```
//显示天气,学员自行补充
gui_display_wea(info.wea->valuestring);
gui_display_cur_temp(info.cur_temp->valuestring);
gui_display_high_temp(info.high_temp->valuestring);
gui_display_low_temp(info.low_temp->valuestring);
gui_display_air_level(info.air_level->valuestring);
gui_display_wea_img(info.wea_img->valuestring);
gui_display_humidity(info.humidity->valuestring);
gui_display_air_pm25(info.air_pm25 ->valuestring);
gui_display_city(info.city->valuestring);
gui_display_win(info.win->valuestring,info.win_speed->valuestring);
```

步骤 5 编译烧录

运行程序，开发板会去连接指定的 WiFi 热点，提示当前打印要连接的 WiFi 不存在。

```
<--System Init-->
<--Wifi Init-->
register wifi event succeed!
callback function for wifi scan:0, 0
+NOTICE:SCANFINISH
callback function for wifi scan:1, 16
WaitSacnResult:wait success[1]s
********************
no:001, ssid:                                   , rssi:   -79
no:002, ssid:                                   , rssi:   -82
no:003, ssid:                                   , rssi:   -87
********************
ERROR: No wifi as expected
```

问题研讨

连接哪个 WiFi 热点，在代码中哪里体现?

【参考答案】

修改 app_main.c 中的 WiFi 账号密码，可以改成开发板连接的热点。

```
// 配置 wifi 账号密码，学员自行补充
#define CONFIG_WIFI_SSID "Huawei"
#define CONFIG_WIFI_PWD "12345678"
```

任务三 实验效果演示

准备一个路由器，没有的话，电脑或者手机当热点，热点名称和密码设置为与 app_main.c 中对应的名称和密码。

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，屏幕上会显示地理位置、天气信息、时间日期、气压海拔等信息。

至此，我们已经完成了 WiFi 智能时钟的开发。

问题研讨

如何获取到天气信息?

【参考答案】

在 app_weather.c 中通过天气 API 接口获取天气信息，appid 和 appsecret 参数可 www.tianqiapi.com 网站上注册账号获取。

```
#define   GET_REQUEST_PACKAGE       \
          "GET http://www.tianqiapi.com/api/?version=v6&appid=%s&appsecret=%s\r\n\r\n"
#define   appid      "XXXXXX"
#define   appsecret      "XXXXXX"
```

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