Huawei IoT Certification Training

HCIA-IoT

Lab Guide

ISSUE: 3.0

HUAWEI TECHNOLOGIES CO., LTD

Trademarks and Permissions

 and other Huawei trademarks are trademarks of Huawei Technologies Co., Ltd.

All other trademarks and trade names mentioned in this document are the property of their respective holders.

Notice

The purchased products, services and features are stipulated by the contract made between Huawei and the customer. All or part of the products, services and features described in this document may not be within the purchase scope or the usage scope. Unless otherwise specified in the contract, all statements, information, and recommendations in this document are provided "AS IS" without warranties, guarantees or representations of any kind, either express or implied.

The information in this document is subject to change without notice. Every effort has been made in the preparation of this document to ensure accuracy of the contents, but all statements, information, and recommendations in this document do not constitute a warranty of any kind, express or implied.

Huawei Technologies Co., Ltd.

Address:	Huawei Industrial Base Bantian, Longgang Shenzhen 518129 People's Republic of China
Website:	https://e.huawei.com/en/

Huawei Certification System

Huawei Certification is an integral part of the company's Platform + Ecosystem strategy. It supports the development of ICT infrastructure that features Cloud-Pipe-Device synergy. Our certification is always evolving to reflect the latest trends in ICT development.

Huawei Certification consists of three categories: ICT Infrastructure Certification, Basic Software & Hardware Certification, and Cloud Platform & Services Certification, making it the most extensive technical certification program in the industry.

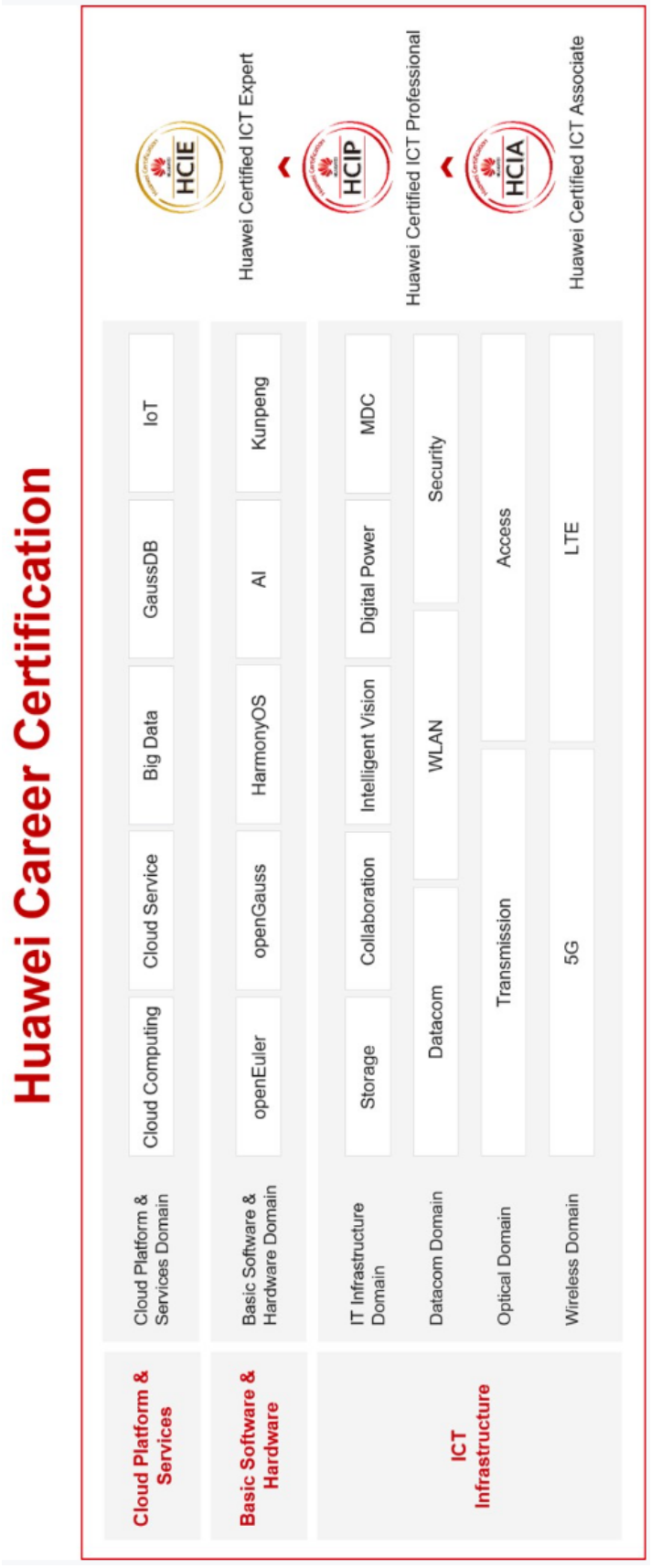
Huawei offers three levels of certification: Huawei Certified ICT Associate (HCIA), Huawei Certified ICT Professional (HCIP), and Huawei Certified ICT Expert (HCIE).

Our programs cover all ICT fields and follow the industry's trend of ICT convergence. With our leading talent development system and certification standards, we are committed to fostering new digital ICT talent and building a sound ICT talent ecosystem.

HCIA-IoT aims to cultivate IoT engineers who have professional knowledge and are capable of development using Huawei IoT products.

Engineers certified by HCIA-IoT master IoT basics, solutions, and development trends; data collection technologies and microcontroller unit (MCU) basics; IoT communications technologies, protocols, and platform basics; and device-cloud connection development based on the Huawei Cloud IoT platform.

Huawei certification helps you unlock opportunities to advance your career and take one more step towards the top of the IoT industry.


Huawei Career Certification
HCIE
Huawei Certified ICT Expert
HCIP
Huawei Certified ICT Professional
HCIA
Huawei Certified ICT Associate
Cloud Platform & Services
Basic Software & Hardware
ICT Infrastructure
Cloud Platform & Services Domain
Basic Software & Hardware Domain
IT Infrastructure Domain
Datacom Domain
Optical Domain
Wireless Domain
Cloud Computing
Cloud Service
Big Data
GaussDB
IoT
openEuler
openGauss
HarmonyOS
AI
Kunpeng
Storage
Collaboration
Intelligent Vision
Digital Power
MDC
Datacom
WLAN
Security
Transmission
Access
5G
LTE

About This Document

Overview

This document is intended for candidates who are preparing for HCIA-IoT V3.0 exam or anyone who wants to understand Huawei's full-stack IoT solution.

Description

This lab guide includes four exercises: MCU peripheral exercise, connection to Wi-Fi, device-cloud connection using the Huawei Cloud IoT platform, and a comprehensive exercise that combines operations in the first three exercises.

- MCU Peripheral Exercise cover six scenarios to help you understand how MCU peripherals work and how to develop embedded software.
- Connection to Wi-Fi consists of two exercises that describe how to connect to the Wi-Fi network using AT commands and code. It helps you understand how Wi-Fi works and how to use AT commands to control a Wi-Fi module and enable automatic connection to Wi-Fi.
- Device-Cloud Connection Using the Huawei Cloud IoT Platform helps you understand the operations and development process on the IoT Device Access (IoTDA) platform. It describes how to connect an MQTT device simulator to the platform, making you prepared to develop physical devices.
- Comprehensive Exercise covers four scenarios of smart vehicle device-cloud connection. You will learn about how to report device data and deliver commands. This exercise combines operations in the first three exercises, allowing you to implement multiple functions in one exercise.

Background Knowledge Required

This lab guide is with HCIA-IoT V3.0. To better understand this guide, you should be able to:

- Understand and develop code in C.
- Understand the basic process of embedded development.

Experiment Environment Overview

Device Introduction

The following table lists devices recommended for HCIA-IoT experiments and the mappings between the device name, model, and software version.

Table 1-1 Lab environment configurations

Device Name	Model	Software Version
PC	Windows	Windows 10 (64-bit)
HiSilicon HiSpark development kit	HiSpark Pegasus T1 IoT development kit	/

Experiment Environment Preparation

Checking Whether All Devices Are Available

Before starting the experiment, check whether all required devices are ready. The following table lists the required devices.

Table 1-2 Lab device list

Device	Quantity	Remarks
Laptop or desktop computer	One for each person	A desktop computer requires a network adapter.
HiSilicon HiSpark development kit	One kit for each person	HiSpark Pegasus T1 IoT development kit

Contents

1 MCU Peripheral Exercise

1.1 Overview

1.1.1 About This Exercise

This exercise uses various peripherals on the HiSpark development board to help you understand how MCU peripherals work and master how to control MCU peripherals.

1.1.2 Objectives

Upon completion of this exercise, you will be able to:

- Understand how MCU peripherals work;
- Control MCU peripherals.

1.1.3 Background

After learning *Data Collection Technologies* and *MCU Basics*, you may only have a basic understanding of MCU peripherals. To further understand how to control them with programs, hands-on practices are required.

1.2 Scenarios and Answers

1.2.1 Scenario 1: Turning On an LED

1.2.1.1 Background

During program design, developers usually check whether each program compilation process works properly by printing "Hello World!" on the terminal. In embedded development, a developer wants to check whether the compilation and burning work properly on the HiSpark development board by turning on a light emitting diode (LED).

In this exercise, the developer wants to:

- Turn on the red LED (LED3) on the development board and make it blink at an interval of 1 second.

1.2.1.2 Quiz

Is the general-purpose input/output (GPIO) pin in input or output direction when the LED is turned on?

[Answer]

Output direction

1.2.1.3 Procedure

Step 1 Set up a lab environment.

Set up a lab environment and complete compilation, burning, and testing. For details, see the *HCIA-IoT V3.0 Lab Environment Setup Guide*.

Step 2 Configure hardware.

Set the **MOTOR_EN** toggle switch on the back of the development board from ON to OFF (to the number side).

Step 3 Open the **lab1_1.c** file.

Open Visual Studio Code and open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_1/lab1_1.c** in the source code directory.

Step 4 Add related header file names.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
#include <hi_io.h>// Sets the I/O function.
#include <hi_gpio.h>// Sets the GPIO direction.
#include <hi_time.h>// Sets the LED blinking latency.
```

Step 5 Add the GPIO pin configuration.

Add the following configuration code to the part marked with **Step 2: GPIO Pin Configuration** in the source code:

```
[Answer]
// GPIO initialization of LED3
hi_gpio_init();
// GPIO Set the pin reuse relationship of GPIO9 to GPIO
hi_io_set_func(HI_IO_NAME_GPIO_9, HI_IO_FUNC_GPIO_9_GPIO);
// GPIO direction set to output
hi_gpio_set_dir(HI_GPIO_IDX_9, HI_GPIO_DIR_OUT);
```

[API Definition]

hi_u32 hi_gpio_init(hi_void);

- Function: Initializes the GPIO module.
- Return value: **0** indicates a success. Other values indicate a failure.

hi_u32 hi_io_set_func(hi_io_name id, hi_u8 *val);

- Function: Configures multiplexing for an I/O pin.
- **id**: hardware pin name.
- **val**: I/O pin multiplexing.
- Return value: **0** indicates a success. Other values indicate a failure.

hi_u32 hi_gpio_set_dir(hi_gpio_idx id, hi_gpio_dir *dir);

- Function: Sets the GPIO pin direction.
- **id**: GPIO index (GPIO pin number).
- **dir**: GPIO direction.
- Return value: **0** indicates a success. Other values indicate a failure.

Step 6 Add the LED blinking code.

Add the following service code to the part marked with **Step 3: Add Service Code** in the source code:

```
[Answer]
hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE1);
hi_udelay(DELAY_1S);
hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE0);
hi_udelay(DELAY_1S);
```

[Note]

To make the LED blink, the LED must be on and off repeatedly.

[API Definition]

hi_u32 hi_gpio_set_ouput_val(hi_gpio_idx id, hi_gpio_value val);

- Function: Sets the output level of a GPIO pin.
- **id**: GPIO index (GPIO pin number).
- **val**: output value of the pin level. **0** indicates low level. **1** indicates high level.
- Return value: **0** indicates a success. Other values indicate a failure.

hi_void hi_udelay(hi_u32 us);

- Function: Sets the microsecond-level latency.
- **us**: latency time, in microseconds.

Step 7 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_1/BUILD.gn** file and add the following content:

```
static_library("lab1_1") {
  sources = [
    "lab1_1.c",
  ]

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

Step 8 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab1_1:lab1_1",
    ]
}
```

Step 9 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, press the reset button on the development board. After the development board is reset, the red LED blinks at an interval of 1 second.

1.2.2 Scenario 2: Enabling the Marquee Effect

1.2.2.1 Background

After achieving LED blinking, the developer also wants to achieve the marquee effect with LEDs. Tri-color LEDs on the left and right sides of the HiSpark development board can be used to implement the marquee effect. Tri-color LEDs display red, green, and blue lights. When these lights are on at the same time, they display a white light. In this exercise, the developer wants to achieve:

- The green, blue, red, and white lights turn on in sequence.
- Each light is on for 1 second.

Inter integrated circuit (I2C) is used to control tri-color LEDs on the development board. You need to learn about how to control the LEDs using I2C before conducting this exercise.

The circuit diagram of the HiSpark development board shows that the pins of the tri-color LEDs connect to the PCA9555 chip, through which the LEDs connect to the main chip. The LEDs communicate with the main chip using I2C. Before implementing the marquee effect, you need to enable the I2C function of the main chip by referring to the I/O port function mapping table, and write commands to the main chip based on the PCA9555 address to control the peripherals connected to PCA9555.

For details, see the PCA9555A documentation.

1.2.2.2 Quiz

1. What are the I2C pins and LED control pins of the PCA9555 chip?

[Answer]

I2C SDA: GPIO13; I2C SCL: GPIO14; LED control: GPIO11.

Q2: What is the hardware address of the PCA9555 chip?

[Answer]

According to the code or HiSpark hardware circuit diagram, the hardware address of the PCA9555 chip is 0100101b.

1.2.2.3 Procedure

Step 1 Configure hardware.

Set the **MOTOR_EN** toggle switch on the back of the development board from ON to OFF (to the number side).

Step 2 Open the **lab1_2.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_2/lab1_2.c** in the source code directory.

Step 3 Add related header files.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
[Answer]
#include "hi_time.h"
#include "pca9555.h"
```

Step 4 Initialize the I2C function and LED pins.

Add the following code to the part marked with **Step 2: Initialization of the I2C function and tri-color LED pin** in the source code.

```
[Answer]
PCA9555Init();
```

[API Definition]

void PCA9555Init(void);

- Function: Initializes the PCA9555 chip, that is, initialize the GPIO11, GPIO13, and GPIO14 pins connected to the main chip.

Step 5 Configure the I/O pin direction.

Add the following code to the part marked with **Step 3: PCA9555 IO direction configuration** in the source code:

```
[Answer]
SetPCA9555GpioValue(PCA9555_PART1_IODIR, PCA9555_OUTPUT);
```

[API Definition]

uint32_t SetPCA9555GpioValue(uint8_t addr, uint8_t buffer);

- Function: Sets the chip pin parameters. It is an I2C write command.
- **addr**: transmission address, that is, each register of the chip.
- **buffer**: transmission parameter value, that is, the value set by a register.
- Return value: **0** indicates a success. Other values indicate a failure.

Step 6 Add the service code.

Add the following code to the part marked with **Step 4: Add Service Code** in the source code based on the code comment:

```
[Answer]
/*
* Set green light: IO1_ 3 Output high level to turn on the left green light
*/
SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, GREEN_LED);
/*
* Delay function milliseconds (set high level duration)
*/
hi_udelay(DELAY_1S);
/*
* Set blue light: IO1_ 4 Output high to turn on the left car blue light
*/
SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, BLUE_LED);
/*
* Delay function milliseconds (set high level duration)
*/
hi_udelay(DELAY_1S);
/*
* Set red light: IO1_ 3 Output high level to turn on the left vehicle red light
*/
SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, RED_LED);
/*
* Delay function milliseconds (set high level duration)
*/
hi_udelay(DELAY_1S);
/*
* Set IO1_ 3 IO1_ 4 IO1_ 5 output high level, left vehicle white light
*/
SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, WHITE_LED);
/*
* Delay function milliseconds (set high level duration)
*/
hi_udelay(DELAY_1S);
```

Step 7 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_2/BUILD.gn** file and add the following content:

```
static_library("lab1_2") {
  sources = [
    "lab1_2.c",
    "pca9555.c",
  ]

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

```
    ]
}
```

Step 8 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab1_2:lab1_2",
    ]
}
```

Step 9 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, press the reset button on the development board. After the development board is reset, the tri-color LED on the left of the development board blinks green, blue, red, and white in sequence at an interval of 1 second.

1.2.3 Scenario 3: Controlling GPIO Interrupt Using Keys

1.2.3.1 Background

In the preceding two exercises, the developers have learned how to configure GPIO pins and control their directions and how to enable I2C communications. Next, the developer wants to output data using GPIO pins, that is, to implement functions using keys on the development board.

In this exercise, the developer wants to achieve:

- When the three keys are pressed, data is printed on the serial port terminal.

1.2.3.2 Quiz

How do you configure the keys in this exercise?

[Answer]

Configure the I2C function and GPIO11 because the three keys also connect to the PCA9555 chip.

1.2.3.3 Procedure

Step 1 Configure hardware.

Set the **MOTOR_EN** toggle switch on the back of the development board from ON to OFF (to the number side).

Step 2 Open the **lab1_3.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_3/lab1_3.c** in the source code directory.

Step 3 Add related header files.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
[Answer]
#include "pca9555.h"
#include "hi_gpio.h"
#include "hi_io.h"
```

Step 4 Configure the GPIO interrupt function.

Add the following code to the part marked with **Step 2: GPIO interrupt function configuration** in the source code:

```
[Answer]
hi_gpio_register_isr_function(HI_IO_NAME_GPIO_11, HI_INT_TYPE_EDGE,
HI_GPIO_EDGE_FALL_LEVEL_LOW, OnFuncKeyPressed, NULL);
```

[API Definition]

hi_u32 hi_gpio_register_isr_function(hi_gpio_idx id, hi_gpio_int_type int_type, hi_gpio_int_polarity int_polarity, gpio_isr_callback func, hi_void *arg);

- Function: Enables the interrupt function of a GPIO pin.
- **id**: GPIO index (GPIO pin number).
- **int_type**: interrupt type.
- **int_polarity**: interrupt polarity.
- **func**: interrupt callback function.
- **arg**: input parameter of the interrupt callback function.
- Return value: **0** indicates a success. Other values indicate a failure.

Step 5 Configure the key direction.

Add the following code to the part marked with **Step 3: Key pin direction configuration** in the source code:

```
[Answer]
SetPCA9555GpioValue(PCA9555_PART0_IODIR, 0x1c);
```

Step 6 Add the service code.

Add the service code to the part marked with **Step 4: Complete the code according to the prompts in the comments.** in the source code based on comments.

```
[Answer]
//When the difference is zero, it indicates that there is an error and the error content is printed.
if (diff == 0) {
    printf("diff = 0! state:%0X, %0X\r\n", ext_io_state, ext_io_state_d);
```

```
}
//When the difference is 04, it indicates that key S3 is pressed and the content is printed.
if ((diff & 0x04) && ((ext_io_state & 0x04) == 0)) {
    printf("Key S3 is pressed\r\n");
}
//When the difference is 08, it indicates that key S2 is pressed and the content is printed.
else if ((diff & 0x08) && ((ext_io_state & 0x08) == 0)) {
    printf("Key S2 is pressed\r\n");
}
//When the difference is 10, it indicates that key S1 is pressed and the content is printed.
else if ((diff & 0x10) && ((ext_io_state & 0x10) == 0)) {
    printf("Key S1 is pressed\r\n");
}
```

Step 7 Complete code in the main function.

Add the following code to the main function in the part marked with **Step 5: Completing the code according to the comments** in the source code based on the comments.

```
[Answer]
/*Key interrupt initialization*/
FuncKeyInit();
/*Obtains the real-time key status*/
GetFunKeyState();
```

Step 8 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_3/BUILD.gn** file and add the following content:

```
static_library("lab1_3") {
  sources = [
    "lab1_3.c",
    "pca9555.c",
  ]

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

Step 9 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab1_3:lab1_3",
```

```
    ]
}
```

Step 10 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, press the reset button on the development board and open the serial port terminal. Press S1, S2, and S3 in sequence. The following information is displayed on the serial port terminal:

```
Key S1 is pressed
Key S2 is pressed
Key S3 is pressed
```

1.2.4 Scenario 4: Making a Breathing Light

1.2.4.1 Background

After learning how to use GPIO, the developer wants to use pulse-width modulation (PWM) to implement some functions, among which buzzers and breathing lights are the most popular ones. The developer plans to use a light to make a breathing light.

In this exercise, the developer wants to:

- Turn LED3 on the development board to a breathing light.

1.2.4.2 Quiz

At which power frequency can an LED be turned on?

[Answer]

Generally, an LED can be turned on at 50 Hz frequency.

1.2.4.3 Procedure

Step 1 Configure hardware.

Set the **MOTOR_EN** toggle switch on the back of the development board from ON to OFF (to the number side).

Step 2 Open the **lab1_4.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_4/lab1_4.c** in the source code directory.

Step 3 Add header files.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
[Answer]
#include "hi_io.h"
#include "hi_gpio.h"
#include "hi_time.h"
#include "hi_pwm.h"
```

Step 4 Add the GPIO pin configuration.

Query the pin number of LED3 on the development board. Add the following configuration code to the part marked with **Step 2: GPIO Pin Configuration** in the source code:

```
[Answer]
hi_io_set_func(HI_IO_NAME_GPIO_9, HI_IO_FUNC_GPIO_9_PWM0_OUT);
hi_gpio_set_dir(HI_GPIO_IDX_9, HI_GPIO_DIR_OUT);
```

Step 5 Add the code for PWM initialization and clock settings.

Add the following configuration code to the part marked with **Step 3: PWM Initialization and Clock Configuration** in the source code:

```
[Answer]
hi_pwm_init(HI_PWM_PORT_PWM0);
hi_pwm_set_clock(PWM_CLK_160M);
```

[API Definition]

hi_u32 hi_pwm_init(hi_pwm_port port);

- Function: Initializes PWM.
- **port**: PWM port number.
- Return value: **0** indicates a success. Other values indicate a failure.
- Note: Before using the PWM function, ensure that PWM has been initialized and GPIO multiplexing has been configured.

hi_u32 hi_pwm_set_clock(hi_pwm_clk_source clk_type);

- Function: Sets the clock type of the PWM module.
- **clk_type**: clock type.
- Return value: **0** indicates a success. Other values indicate a failure.

Step 6 Add the breathing light service code.

Add the following service code to the part marked with **Step 4: Add Service Code** in the source code:

```
[Answer]
for (int i = 1; i <= 8001; i += 10)
{
    hi_pwm_start(HI_PWM_PORT_PWM0, i, 16000);
    hi_udelay(1500);
}
for (int i = 8001; i >= 1; i -=10)
{
    hi_pwm_start(HI_PWM_PORT_PWM0, i, 16000);
    hi_udelay(1500);
}
```

[Note]

The duty cycle of PWM signals changes continuously to achieve the breathing light effect. You can write a loop statement based on the change rule to control the breathing light effect.

[API Definition]

hi_u32 hi_pwm_start(hi_pwm_port port, hi_u16 duty, hi_u16 freq);

- Function: outputs PWM signals based on configured parameters. Duty cycle of PWM signals = **duty**/**freq**. Frequency = clock source frequency/**freq**.
- **port**: PWM port number.
- **duty**: count value of the PWM duty cycle.
- **freq**: frequency division multiple.
- Return value: **0** indicates a success. Other values indicate a failure.

Step 7 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_4/BUILD.gn** file and add the following content:

```
static_library("lab1_4") {
  sources = [
    "lab1_4.c",
  ]

  include_dirs = [
    "//utils/native/lite/include",
    "//device/hisilicon/hispark_pegasus/sdk_liteos/include",
    ]
}
```

Step 8 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab1_4:lab1_4",
    ]
}
```

Step 9 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, press the reset button on the development board. After the development board is reset, LED3 turns from dark to bright and then from bright to dark regularly.

1.2.5 Scenario 5: Enabling Serial Communications

1.2.5.1 Background

Now the developer has learned how to use PWM. Next, the developer plans to use UART serial ports to enable data exchange between the serial port terminal and development board.

In this exercise, the developer wants to:

- Use UART serial ports to make the development board output the user input data to the serial port terminal.

1.2.5.2 Quiz

Which of the following chip pins are used by UART1?

[Answer]

UART_TXD: GPIO 0; UART_RXD: GPIO 1.

1.2.5.3 Procedure

Step 1 Configure hardware.

Connect the C44 jumper cap to TX-RX, J7 to TX-GPIO0, and J8 to RX-GPIO1. (This step is required only in this exercise.)

Step 2 Open the **lab1_5.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_5/lab1_5.c** in the source code directory.

Step 3 Add header files.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
[Answer]
#include "hi_io.h"
#include "hi_gpio.h"
#include "hi_uart.h"
```

Step 4 Configure the GPIO pin.

Add the code for GPIO initialization and pin multiplexing to the part marked with **Step 2: GPIO Pin Configuration** in the source code:

```
[Answer]
//GPIO initialization
hi_gpio_init();
//Set the pin reuse relationship of GPIO0 to UART1_ TX
hi_io_set_func(HI_IO_NAME_GPIO_0, HI_IO_FUNC_GPIO_0_UART1_TXD);
//Set the pin reuse relationship of GPIO1 to UART1_ RX
hi_io_set_func(HI_IO_NAME_GPIO_1, HI_IO_FUNC_GPIO_1_UART1_RXD);
```

Step 5 Configure UART parameters.

Add the following code to the part marked with **Step 3: UART Configuration** in the source code based on the UART parameter values in code comments:

```
[Answer]
/* Initialize UART configuration, baud rate is 9600, data bit is 8, stop bit is 1, parity is NONE */
hi_uart_attribute uart_attr = {
    .baud_rate = 9600,
    .data_bits = 8,
    .stop_bits = 1,
    .parity = 0,
};
```

[API Definition]

hi_u32 hi_uart_init(hi_uart_idx id, const hi_uart_attribute *param, const hi_uart_extra_attr *extra_attr);

- Function: Specifies the UART port.
- **id**: UART port number.
- **param**: basic UART parameters.
- **extra_attr**: UART optimization parameters.
- Return value: **0** indicates a success. **-1** indicates a failure.
- Note: If **extra_attr** is **HI_NULL**, the driver is notified that all optimization parameters use default values. If you want to change UART optimization parameter settings after UART is initialized, call **hi_uart_deinit()** to deinitialize UART, and then call **hi_uart_init()** to change the settings.

```
typedef struct {
    hi_u32 baud_rate;/**< Baud Rate.*/
    hi_u8 data_bits;   /**< Data bit.*/
    hi_u8 stop_bits;   /**< Stop bit.*/
    hi_u8 parity;      /**< Parity check flag.*/
    hi_u8 pad;         /**< reserved pad */
} hi_uart_attribute;
```

Step 6 Add the subfunctions called in the preceding steps to the main function.

Add the following code to the part marked with **Step 4: Add functions based on comments** in the source code based on code comments:

```
[Answer]
//Initialization of UART1
Uart1GpioInit();
//Configurations of UART1 parameters
Uart1Config();
```

Step 7 Write the service code.

Enter the UART service code in the part marked with **Step 5: Add Service Code** in the source code, write the string constant **data** in the main function to UART, read **data** using UART, and print it on the serial port terminal.

```
[Answer]
//Send data through UART1
hi_uart_write(HI_UART_IDX_1, (unsigned char*)data, strlen(data));
//Receive data through UART1
len = hi_uart_read(HI_UART_IDX_1, uartReadBuff, UART_BUFF_SIZE);
if (len > 0) {
    //Print the received data
    printf("Uart Read Data is: [ %d ] %s \r\n", count, uartReadBuff);
}
```

[API Definition]

hi_s32 hi_uart_read(hi_uart_idx id, hi_u8 *data, hi_u32 data_len);

- Function: Reads the data received by UART.
- **id**: UART port number.
- **data**: start address of the data to be read.
- **data_len**: number of bytes to be read.
- Return value: Values greater than or equal to 0 indicate the number of bytes read. **-1** indicates a receiving failure.

hi_s32 hi_uart_write(hi_uart_idx id, const hi_u8 *data, hi_u32 data_len);

- Function: Writes the data to be sent to UART.
- **id**: UART port number.
- **data**: start address of the data to be written.
- **data_len**: number of bytes to be written.
- Return value: Values greater than or equal to 0 indicate the number of bytes sent. **-1** indicates a sending failure.

Step 8 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_5/BUILD.gn** file and add the following content:

```
static_library("lab1_5") {
  sources = [
    "lab1_5.c",
  ]

  include_dirs = [
    "//utils/native/lite/include",
    "//device/soc/hisilicon/hi3861v100/sdk_liteos/include/base",

"device\hisilicon\hispark_pegasus\sdk_liteos\platform\os\Huawei_LiteOS\components\lib\libc\musl\include",
    ]
}
```

Step 9 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab1_5:lab1_5",
    ]
}
```

Step 10 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, open the serial port terminal and press the reset button on the development board. After the development board is reset, the following information is displayed on the serial port terminal:

```
Init Uart1 successfully!
Uart Read Data is: [ 0 ] Hello OpenHarmony!!!

Uart Read Data is: [ 1 ] Hello OpenHarmony!!!

Uart Read Data is: [ 2 ] Hello OpenHarmony!!!

Uart Read Data is: [ 3 ] Hello OpenHarmony!!!

Uart Read Data is: [ 4 ] Hello OpenHarmony!!!

Uart Read Data is: [ 5 ] Hello OpenHarmony!!!

Uart Read Data is: [ 6 ] Hello OpenHarmony!!!
```

1.2.6 Scenario 6: Comprehensive GPIO Exercise

1.2.6.1 Background

After completing the preceding exercises, the developer has learned how to control input/output using GPIO, control LEDs using PWM, and implement serial communications using UART and I2C. This comprehensive exercise combines the preceding operations.

In scenarios 2 and 3, I2C is used to control the LED and keys. In this comprehensive exercise, the developer wants to achieve:

- The three colors of the tri-color LED are controlled by three keys.
- The LED emits light of a color when a key is pressed and turns off when the key is pressed again. When the three keys are pressed at the same time to turn on the LED, it displays a white light.
- S1 controls the green light. S2 controls the blue light. S3 controls the red light.

1.2.6.2 Quiz

In the preceding exercise scenarios, I2C write commands are executed by writing values directly to the register to turn on the LED. How do you ensure that the status of the previous LED is not affected when we turn on LEDs one by one?

[Answer]

Before writing data to the register, read values of the register, and then modify a bit in the register based on the LED to be turned on or off.

1.2.6.3 Procedure

Step 1 Configure hardware.

Restore the hardware configuration in Scenario 5. Set the **MOTOR_EN** toggle switch on the back of the development board from ON to OFF (to the number side).

Step 2 Open the **lab1_6.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_6/lab1_6.c** in the source code directory.

Step 3 Add header files.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
[Answer]
#include "hi_time.h"
#include "hi_io.h"
#include "hi_gpio.h"
#include "pca9555.h"
```

Step 4 Complete configuration code.

In the part marked with **Step 2: Complete the configuration according to the comments.** in the source code, configure the tri-color LED, keys, and interrupt settings based on the code comments.

```
[Answer]
//Initialize the PCA9555
PCA9555Init();
//Pin direction configuration of the tri-color lamp
SetPCA9555GpioValue(PCA9555_PART1_IODIR, PCA9555_OUTPUT);
//Key interrupt configuration
hi_gpio_register_isr_function(HI_IO_NAME_GPIO_11, HI_INT_TYPE_EDGE,
HI_GPIO_EDGE_FALL_LEVEL_LOW, OnFuncKeyPressed, NULL);
//Key Direction Configurations
SetPCA9555GpioValue(PCA9555_PART0_IODIR, 0x1C);
```

Step 5 Obtains values in the register.

Complete the part marked with **Step 3: Obtaining values from LED registers and key registers** in the source code to write values in the current register to **ext_io_state** and **led_io_state**.

```
[Answer]
status = PCA9555I2CReadByte(&ext_io_state);
PCA9555I2CLEDReadByte(&led_io_state);
```

Step 6 Determine the key based on the difference.

Add code to the part marked with **Step 4: Calculate the difference between the key register before and after the interrupt** in the source code. Determine which key is pressed based on the key register parameter values obtained before and after the interrupt.

```
[Answer]
diff = ext_io_state ^ ext_io_state_d;
```

Step 7 Complete the code for controlling the red light.

Complete the part marked with **Step 5: Write code to control red LED using Key 3** in the source code to use Key 3 (S3) to control the red light.

```
[Answer]
//When key 3 is pressed
if ((diff & 0x04) && ((ext_io_state & 0x04) == 0))
{
    //print the information that key 3 is pressed
    printf("Key 3 is pressed\r\n");
    //Turns off if the RED LED is on and vice versa
    if ((led_io_state & 0X20) == 0x20)
    {
        led_curr_state = led_io_state ^ 0x20;
        SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, led_curr_state);
    }
    else{
        led_curr_state = led_io_state | 0x20;
        SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, led_curr_state);
    }
}
```

Step 8 Complete the code for controlling the blue light.

Complete the part marked with **Step 6: Write code to control blue LED using Key 2** in the source code to use Key 2 (S2) to control the blue light.

```
[Answer]
//When Key 2 is pressed
else if ((diff & 0x08) && ((ext_io_state & 0x08) == 0))
{
    //print the information that key 2 is pressed
    printf("Key 2 is pressed\r\n");
    //Turns off if the BLUE LED is on and vice versa
```

```
    if ((led_io_state & 0x10) == 0x10)
    {
        led_curr_state = led_io_state ^ 0x10;
        SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, led_curr_state);
    }
    else
    {
        led_curr_state = led_io_state | 0x10;
        SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, led_curr_state);
    }
}
```

Step 9 Complete the code for controlling the green light.

Complete the part marked with **Step 7: Write code to control green LED using Key 1** in the source code to use Key 1 (S1) to control the green light.

```
[Answer]
else if ((diff & 0x10) && ((ext_io_state & 0x10) == 0))
{
    //print the information that key 1 is pressed
    printf("Key 1 is pressed\r\n");
    //Turns off if the GREEN LED is on and vice versa
    if ((led_io_state & 0x08) == 0x08)
    {
        led_curr_state = led_io_state ^ 0x08;
        SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, led_curr_state);
    }
    else
    {
        led_curr_state = led_io_state | 0x08;
        SetPCA9555GpioValue(PCA9555_PART1_OUTPUT, led_curr_state);
    }
}
```

Step 10 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab1_6/BUILD.gn** file and add the following content:

```
static_library("lab1_6") {
  sources = [
    "lab1_6.c",
    "pca9555.c",
  ]

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

Step 11 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab1_6:lab1_6",
    ]
}
```

Step 12 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, press the reset button on the development board. After the development board is reset, S1, S2, and S3 can be used to control the green, blue, and red light respectively.

2 Connection to Wi-Fi

2.1 Introduction

In this exercise, you will connect a Wi-Fi module to the network using AT commands and by coding. It helps you master how to use AT commands and develop a Wi-Fi module.

2.2 Objectives

Upon completion of this exercise, you will be able to:

- Master how to develop a Wi-Fi module.
- Be familiar with the function of each command.
- Use AT commands to control the Wi-Fi module network connection.
- Use code to connect to the Wi-Fi network.

2.3 Background

After learning the theories in *IoT Communications Technologies*, *IoT Communications Protocols*, and *AT Commands for IoT Communication Modules*, you should have understood how Wi-Fi works, types of AT commands, and meanings of common commands. This exercise allows you to use AT commands to perform actual development and connect a Wi-Fi module to the network.

2.4 Scenarios and Answers

2.4.1 Scenario 1: Connecting to the Wi-Fi Network Using AT Commands

2.4.1.1 Background

Wi-Fi has always been a common communication mode in IoT scenarios. In smart home scenarios, Wi-Fi is used the most. It is important to understand how a Wi-Fi module is connected to the network. After learning theoretical knowledge, the developer has learned how to use a Wi-Fi module to connect to the Huawei Cloud platform. However, the AT commands of the Hi3861 chip are special. The developer can only understand specific usage rules through the exercise.

In this exercise, the developer wants to:

- Connect a Wi-Fi module to the Wi-Fi hotspot using AT commands.
- Ping the Huawei Cloud official website using AT commands.

2.4.1.2 Quiz

How do you obtain the IP address of the Huawei Cloud official website?

[Answer]

Ping the domain name of the Huawei Cloud official website in the CLI. The returned result contains its IP address.

2.4.1.3 Procedure

Step 1 Open the serial port terminal on Visual Studio Code.

Power on the development board and open the serial port terminal on Visual Studio Code.

Step 2 Modify the input mode of the serial port terminal.

On the serial port terminal, select **Line ending** and set it to **CRLF** for inputting AT commands.

Step 3 Reset the board.

Open the serial port terminal and enter the following content in the sending area:

```
AT+RST
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+RST\r\n" ----
AT+RST
OK
ready to OS start
sdk ver:Hi3861V100R001C00SPC025 2020-09-03 18:10:00
FileSystem mount ok.
wifi init success!
hilog will init.


hiview init success.
```

Step 4 Start the station (STA).

Enter the following content in the sending area to start the Wi-Fi STA:

```
AT+STARTSTA
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+STARTSTA\r\n" ----
AT+STARTSTA
OK
```

Step 5 Initiate STA scanning.

Enable the personal hotspot, set the hotspot name and password, and enter the following content in the sending area:

```
AT+SCAN
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+SCAN\r\n" ----
AT+SCAN
OK

+NOTICE:SCANFINISH
```

Step 6 View the scan result.

Enter the following content in the sending area to view the scan result:

```
AT+SCANRESULT
```

Click **Send Message** on the right of the sending area. Check whether the name of the enabled personal hotspot is displayed in the receiving area. If not, repeat Steps 5 and 6.

Step 7 Initiate a connection to an access point (AP).

Enter the following content in the sending area to initiate a Wi-Fi connection:

```
AT+CONN=<ssid>,<bssid>,<auth_type>,[<passwd>]
Example:
AT+CONN="Huawei",,2,"12345678"
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+CONN=\"Huawei\",,2,\"12345678\"\r\n" ----
AT+CONN="Huawei",,2,"12345678"
OK

+NOTICE:SCANFINISH
+NOTICE:CONNECTED
```

[Command Definition]

AT+CONN=<ssid>,<bssid>,<auth_type>,[<passwd>]

- Response: OK or ERROR

- **ssid**: service set identifier, that is, the router name. Use double quotation marks to enclose the value.
- **bssid**: basic service set identifier, which is usually the MAC address of the router.
- **auth_type**: authentication mode. **0**: **OPEN**; **1**: **WEP**; **2**: **WPA2_SPK**; **3**: **WPA_WPA2_PSK**.
- **passwd**: password. Use double quotation marks to enclose the value. If the authentication mode of the peer network is **WEP** and the password is in ASCII format, Use two pairs of double quotation marks to enclose the values.
- Notes:

  ssid and **bssid** cannot be both empty.

  If **ssid** and **bssid** are both specified and **ssid** does not match **bssid**, the connection fails.

  If the value of **ssid** or **passwd** contains special characters, escape them with backslashes (\). (For example, if the value of **ssid** is **ab,c**, escape it into **ab\,c**.) If the value contains a double quotation mark ("), escape it with a backslash (\). For example, if the value of **ssid** is **ab"c**, escape it into **ab\"c** or use **ab"c**.)

  If **auth_type** is set to **OPEN**, **passwd** and the comma before **passwd** are not required.

Step 8 Obtain the IP address of the DHCP client.

Enter the following command in the sending area to obtain the IP address:

```
AT+DHCP=wlan0,1
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+DHCP=wlan0,1\r\n" ----
AT+DHCP=wlan0,1
OK
```

Step 9 Check the interface configuration.

Enter the following command in the sending area to check the interface configuration:

```
AT+IFCFG
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+IFCFG\r\n" ----
AT+IFCFG
+IFCFG:wlan0,ip=192.168.137.40,netmask=255.255.255.0,gateway=192.168.137.1,ip6=FE80::1AEF:3AFF:FE57:7BB7,HWaddr=18:ef:3a:57:7b:b7,MTU=1500,LinkStatus=1,RunStatus=1
+IFCFG:lo,ip=127.0.0.1,netmask=255.0.0.0,gateway=127.0.0.1,ip6=::1,HWaddr=00,MTU=16436,LinkStatus=1,RunStatus=1
OK
```

Step 10 Ping the host IP address to test the connection.

After the network connection is successful, you can ping an IP address to test the network connection. In this example, the IP address of the Huawei Cloud home page is used. (You can enter **ping** + *web page URL* in the CLI to obtain the IP address.)

Enter the following command in the sending area to test the connection:

```
AT+PING=203.193.226.107
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+PING=203.193.226.107\r\n" ----
AT+PING=203.193.226.107
+PING:

[0]Reply from 203.193.226.107:time=4ms TTL=127
[1]Reply from 203.193.226.107:time=1ms TTL=127
[2]Reply from 203.193.226.107:time=25ms TTL=127
[3]Reply from 203.193.226.107:time=27ms TTL=127
4 packets transmitted, 4 received, 0 loss, rtt min/avg/max = 1/14/27 ms

OK
```

According to the returned information, the four data packets are received. The network connection is successful.

2.4.2 Scenario 2: Connecting to the Wi-Fi Network Using Code

2.4.2.1 Background

The developer has learned how to use common AT commands in the preceding exercise. However, it is not feasible to enter AT commands one by one on the serial port terminal. The developer wants to implement a simpler, automatic network connection.

In this exercise, the developer wants to achieve:

- Automatic network connection by burning the network connection program to the development board.

2.4.2.2 Quiz

Which working mode is used when a Wi-Fi module is used to connect to a hotspot?

[Answer]

The STA mode is used.

2.4.2.3 Procedure

Step 1 Open the **lab2.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab2/lab2.c** in the source code directory.

Step 2 Add header files.

Add the following header file to the part marked with **Step 1: Add Header File** in the source code:

```
[Answer]
#include "wifi_connecter.h"
```

Step 3 Enter the Wi-Fi username and password.

Scenario 1 has introduced detailed steps for Wi-Fi connection, which are not repeated here. Enter your Wi-Fi username and password in the part marked with **Step 2: Fill the Wi-Fi ssid and password** in the source code.

```
[Answer]
strcpy(apConfig.ssid, "Huawei"); // Set Wi-Fi ssid
strcpy(apConfig.preSharedKey, "12345678"); // Set Wi-Fi password
```

[API Definition]

char * strcpy(char * dest,const char *src);

- Function: Copies the string pointed by **src** to **dest**.
- **dest**: target array used to store the copied content.
- **src**: string to be copied.
- Return value: pointer to the target string **dest**.

Step 4 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab2/BUILD.gn** file and add the following content:

```
static_library("lab2") {
    sources = [
        "lab2.c",
        "wifi_connecter.c",
    ]

    include_dirs = [
        "./",
        "//utils/native/lite/include",
        "//kernel/liteos_m/kal/cmsis",
        "//base/iot_hardware/peripheral/interfaces/kits",
        "//foundation/communication/wifi_lite/interfaces/wifiservice",
        "//vendor/hisi/hi3861/hi3861/third_party/lwip_sack/include",
    ]
}
```

Step 5 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

```
lite_component("app") {
    features = [
        "lab2:lab2",
    ]
}
```

Step 6 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, open the serial port terminal and press the reset button on the development board. After the development board is reset, the following information (example) is displayed on the serial port terminal:

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

RegisterWifiEvent: 0

EnableWifi: 0
AddDeviceConfig: 0
ConnectTo(1): 0
+NOTICE:SCANFINISH
+NOTICE:CONNECTED
OnWifiConnectionChanged 42, state = 1, info =
bssid: 62:F2:62:E6:69:41, rssi: 0, connState: 0, reason: 0, ssid: Huawei
g_connected: 1
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       18ef3a577bb7       192.168.137.232 10       0        1        4
netifapi_netif_common: 0
hiview init success.
```

If the preceding information is displayed on the serial port terminal, the Wi-Fi network connection is successful.

Enter the **AT+PING** command in the sending area of the serial port terminal to test the connection to the Huawei Cloud official website by referring to Scenario 1.

Enter the following command in the sending area to test the connection:

```
AT+PING=203.193.226.107
```

Click **Send Message** on the right of the sending area. The following information is displayed in the receiving area:

```
---- Sent message: "AT+PING=203.193.226.107\r\n" ----
AT+PING=203.193.226.107
+PING:

[0]Reply from 203.193.226.107:time=250ms TTL=51
[1]Reply from 203.193.226.107:time=235ms TTL=51
[2]Reply from 203.193.226.107:time=232ms TTL=51
[3]Reply from 203.193.226.107:time=230ms TTL=51
4 packets transmitted, 4 received, 0 loss, rtt min/avg/max = 230/236/250 ms

OK
```

According to the returned information, the four data packets are received. The network connection is successful.

3 Device-Cloud Connection Using the Huawei Cloud IoT Platform

3.1 Introduction

In this exercise, you will develop product models and codecs on IoTDA to learn about how to use them and how to perform development on the platform.

3.2 Objectives

Upon completion of this exercise, you will be able to:

- Develop product models.
- Develop codecs.
- Use simulators.
- Understand the development process on IoTDA.

3.3 Background

You have learned about product models and codecs of IoTDA in *IoT Platform Overview* and *IoT Device-Cloud Connection Development*.

Internet of Vehicles (IoV) is a common use case of IoT. It enables drivers to remotely control vehicles using mobile phones, for example, opening or closing doors and turning on air conditioners. During driving, ultrasonic sensors measure the distance between the vehicle and roadside facilities or other vehicles to generate warnings. In this exercise, the developer will perform IoV-related development and simulation.

CoAP and MQTT are commonly used for development on the Huawei Cloud IoT platform. This exercise will use the two protocols for development and provide results through simulators.

3.4 Scenarios and Answers

3.4.1 Scenario 1: Developing a CoAP Product Model

3.4.1.1 Background

The following tables list details about the smart vehicle CoAP product model.

Table 3-1 Product information

Parameter	Value
Product Name	smart_vehicle_coap
Protocol	LwM2M over CoAP
Data Type	Binary
Manufacturer	Custom
Industry	None
Device Type	Custom

Table 3-2 Service information

Service ID	Service Type (same as the service ID by default)	Description
SmartVehicle	SmartVehicle	/

Table 3-3 SmartVehicle property information

Capability	Property Name	Data Type	Access Permissions	Value Range
Properties	length	int	**Read** and **Write**	0–255
	distance	decimal	**Read** and **Write**	0–65535

Table 3-4 SmartVehicle command information

Command Name	Parameters	Parameter Name	Data Type	Length/ Value Range	Enumerated Values
SMART_VEHICLE_DOOR	Command Parameter	door	string	3	ON,OFF
	Response	doorStatus	int	0–1	/

Command Name	Parameters	Parameter Name	Data Type	Length/ Value Range	Enumerated Values
	Parameter				

Table 3-5 Message list

Message Name	Message Type	MessageId
SmartVehicle	Data reporting	00
SMART_VEHICLE_DOOR	Command delivery	01
	Command response	02

Table 3-6 SmartVehicle message

Code Stream Offset	0–1	1–2	3–*X*
Field Name	messageId	length	distance
Data Type	int8u	int8u	varstring
Length	1	1	Value of **length**

Table 3-7 Command fields of the SMART_VEHICLE_DOOR message

Code Stream Offset	0–1	1–3	3–6
Field Name	messageId	mid	door
Data Type	int8u	int16u	String
Length	1	2	3

Table 3-8 Command response fields of the SMART_VEHICLE_DOOR message

Code Stream Offset	0–1	1–3	3–4	4–5
Field Name	messageId	mid	errcode	doorStatus
Data Type	int8u	int16u	int8u	int8u
Length	1	2	1	1

3.4.1.2 Quiz

Why is the length of the **SMART_VEHICLE_DOOR** command parameter 3?

[Answer]

Enumerated values of the parameter are **ON** and **OFF** and the maximum value length is used, so the length is **3** (number of characters of **OFF**).

3.4.1.3 Task 1: Developing a Product Model

Step 1 Log in to Huawei Cloud IoTDA.

Log in to the IoTDA console and select a region by referring to the lab environment setup guide. (This document uses the AP-Singapore region as an example.)

Step 2 Create a product.

Choose **Products** in the navigation pane and click **Create Product** in the upper right corner. Set the parameters by referring to the product information in 3.4.1.1 and click **OK**.

Step 3 Add a service.

On the **Products** page, click the created product and choose **Model Definition** > **Customize Model**.

In the displayed dialog box, set **Service ID**, **Service Type**, and **Description** based on the service information in 3.4.1.1, and click **OK**.

Step 4 Add properties.

Click the service added in step 3. On the displayed page, click **Add Property**. In the displayed dialog box, set parameters based on the SmartVehicle property information in 3.4.1.1 and click **OK**.

Step 5 Add a command.

Click **Add Command**. In the displayed dialog box, set parameters based on the SmartVehicle command information in 3.4.1.1 and click **OK**.

Product model development is complete.

3.4.1.4 Task 2: Developing a Codec

Step 1 Develop a codec on the GUI.

On the product details page, click the **Codec Development** tab and **Develop Codec** to go to the GUI-based codec development page.

Step 2 Add a data reporting message.

On the codec development page, click **Add Message**, enter **SmartVehicle** in **Message Name**, and select **Data reporting** for **Message Type**.

Click **Add Field** and add each field based on the SmartVehicle message information in 3.4.1.1.

Click **OK**.

Note: The length of **distance** reported by the device is not fixed. You can set its value type to a variable-length string and determine its length based on the value of **length**.

Step 3 Add a command delivery message.

Click **Add Message**, enter **SMART_VEHICLE_DOOR** in **Message Name**, select **Command delivery** for **Message Type**, and select **Add Response Field**.

- Add command delivery fields.

Click **Add Field** and add each field based on the command fields of the **SMART_VEHICLE_DOOR** message in 3.4.1.1.

- Add command response fields.

Click **Add Response Field** and add each field based on the command response fields of the **SMART_VEHICLE_DOOR** message in 3.4.1.1.

Step 4 Drag properties to match the fields.

On the codec development page, click **Properties** in the **Product Model** area on the right, and drag the two properties to the left one by one to match the fields in the message.

In the **Product Model** area on the right, click **Commands**, and drag the command fields and response fields to the left one by one to match the fields in the message.

Click **Save** in the upper right corner.

Click **Deploy** to deploy the codec.

Codec development is complete.

3.4.1.5 Task 3: Verifying Functions and Codecs

Step 1 Create a virtual device.

On the product details page, click the **Online Debugging** tab and **Add Test Device**.

In the displayed dialog box, select **Virtual device** and click **OK**.

Step 2 Simulate data reporting.

Click **Debug** to access the debugging page.

Click **Device Simulator** on the right of the page, enter the hexadecimal code stream **000436352E33**, and click **Send**.

If the application simulator receives the data, the following JSON data will be displayed in the message area:

```
{serviceId: SmartVehicle, data: {"length":4,"distance":"65.3"}}
```

Step 3 Simulate the process of delivering the command for enabling the LED.

Click **Application Simulator** on the right of the page, set **door** to **ON**, and click **Send**.

If the command is sent successfully, **0100014F4E** (example) will be displayed in the message receiving area of the device simulator and the message similar to the following will be displayed in the message area of the application simulator:

{ "service_id": "SmartVehicle", "command_name": "SMART_VEHICLE_DOOR", "paras": { "door": "ON" }, "send_strategy": "immediately", "expire_time": 0 }

Click **Device Simulator** on the right of the page, enter the hexadecimal code stream **0200010001**, and click **Send**. In the upper part of the message tracing page, click **IoT Platform** (with red message notification in the upper right corner). Then, you can find the following information:

[The IoT platform receives a command response from the device]
Receiving command response from the device.commandId:728b3b98-d2c4-478c-afff-c83e32dae2demid:6resultCode:SUCCESSFULresultDetail:{"doorStatus":1}, device_id: 640801d2df787d56d5d8342c_1678258421514, request_id: a47c3407-7a57-4f6a-89c6-1c37643ee1b7

The product model and codec are developed, and the platform can receive the hexadecimal code stream from the device properly.

3.4.2 Scenario 2: Developing an MQTT Product Model

3.4.2.1 Background

The following tables list details about the smart vehicle MQTT product model.

Table 3-9 Product information

Parameter	Value
Product Name	smart_vehicle_mqtt
Protocol	MQTT
Data Type	JSON
Manufacturer	Custom
Industry	None
Device Type	Custom

Table 3-10 Service information

Service ID	Service Type (same as the service ID by default)	Description
SmartVehicle	SmartVehicle	/

Table 3-11 SmartVehicle property

Capability	Property Name	Data Type	Access Permissions	Length
Properties	distance	decimal	**Read** and **Write**	0–65535

Table 3-12 SmartVehicle command

Command Name	Parameters	Parameter Name	Data Type	Length/ Value Range	Enumerated Values
SMART_VEHICLE_DOOR	Command Parameter	door	string	3	ON,OFF
	Response Parameter	doorStatus	int	0–1	/

3.4.2.2 Quiz

Is a codec required when the MQTT product uses JSON data for property reporting? And why?

[Answer]

No. The platform understands JSON data, so a codec is not required.

3.4.2.3 Task 1: Developing a Product Model

Step 1 Log in to Huawei Cloud IoTDA.

Log in to the IoTDA console and select a region by referring to the lab environment setup guide. (This document uses the AP-Singapore region as an example.)

Step 2 Create a product.

Choose **Products** in the navigation pane and click **Create Product** in the upper right corner. Set the parameters by referring to the product information in 3.4.2.1 and click **OK**.

Step 3 Add a service.

On the **Products** page, click the created product and choose **Model Definition** > **Customize Model**.

In the displayed dialog box, set **Service ID**, **Service Type**, and **Description** based on the service information in 3.4.2.1, and click **OK**.

Step 4 Add properties.

Click the service added in step 3. On the displayed page, click **Add Property**. In the displayed dialog box, set parameters based on the SmartVehicle property information in 3.4.2.1 and click **OK**.

Step 5 Add a command.

Click **Add Command**. In the displayed dialog box, set parameters based on the SmartVehicle command information in 3.4.2.1 and click **OK**.

Product model development is complete.

3.4.2.4 Task 2: Verifying Functions

Step 1 Create a virtual device.

On the product details page, click the **Online Debugging** tab and **Add Test Device**.

In the displayed dialog box, select **Virtual device** and click **OK**.

Step 2 Simulate data reporting.

Click **Debug** to access the debugging page.

Click **Device Simulator** on the right of the page, enter a value in **distance**, for example, **65.3**. Click **Send**.

If the application simulator receives the data, the following JSON data will be displayed in the message area:

```
{serviceId: SmartVehicle, data: {"distance":"65.3"}}
```

Step 3 Simulate the process of delivering the command for enabling the buzzer.

Click **Application Simulator** on the right of the page, set **door** to **ON**, and click **Send**.

If the command is sent successfully, the following message (example) will be displayed in the message receiving area of the device simulator:

```
{ "paras": { "door": "ON" }, "service_id": "SmartVehicle", "command_name":
"SMART_VEHICLE_DOOR" }
```

The following JSON data (example) will be displayed in the message area of the application simulator:

```
{ "service_id": "SmartVehicle", "command_name": "SMART_VEHICLE_DOOR", "paras": { "door":
"ON" } }
```

The JSON device simulator cannot simulate command response reporting, so command response debugging is not performed. You can use a physical device to debug this function later.

The product model and codec are developed, and the MQTT product model is tested.

4 Comprehensive Exercise

4.1 Introduction

After completing the preceding exercise, you should have a better understanding of basic operations on the Huawei Cloud IoT platform, and usage of the device simulator, CoAP, and MQTT.

In this exercise, the HiSpark development board is connected to the Huawei Cloud IoT platform. The development board uploads data to the platform, and the platform delivers commands to the development board, which also responds to the commands accordingly. This allows you to connect a physical device to the platform and have a better understanding of device-side and platform-side code development.

4.2 Objectives

Upon completion of this exercise, you will be able to:

- Master the ultrasonic sensor development process.
- Be familiar with the MQTT development mode.
- Understand the assembly mode of JSON data.
- Master the development process of data reporting and command delivery.
- Master how to test the connection between a physical device and platform.

4.3 Background

With the continuous upgrade of information and communications technologies, vehicle intelligence has been greatly improved, enabling autonomous driving and remote control.

In the reversing scenario, a smart vehicle measures the distance between the vehicle and surrounding obstacles. When the vehicle is too close to an obstacle, it generates an alarm. In addition, if the temperature inside the smart vehicle is too high or too low, you can use your mobile phone to turn on the air conditioner in advance to reduce or increase the temperature inside the vehicle and to open or close doors remotely.

This exercise uses a smart vehicle as an example and allows you to enable the device-cloud connection and simulate vehicle data reporting and control command delivery.

In this exercise, the developer connects an ultrasonic sensor to the HiSpark development board and connects the development board to the Huawei Cloud IoT platform to upload data and deliver commands.

In this exercise, the developer wants to:

- Enable the ultrasonic sensor to measure distance.
- Enable the Wi-Fi module to report the distance data.
- Register a physical device on IoTDA and deliver commands to the device.
- Develop the command response function on the device side.

4.4 Scenarios and Answers

4.4.1 Scenario 1: Developing an Ultrasonic Sensor

4.4.1.1 Background

The first step for device-platform connection is to collect device data. In this exercise, the developer uses the ultrasonic sensor in the development kit and converts the data collected by the sensor into distance values that can be used.

In this exercise, the developer wants to:

- Enable the ultrasonic sensor.
- Collect distance data.

[HC-SR04]

- Product features

The HC-SR04 ultrasonic distance measurement module provides non-contact measurement of distance (2–400 cm) with the precision of 3 mm. The module includes an ultrasonic transmitter, a receiver, and a control circuit.

- Working principles

The TRIG pin of the I/O port is used to trigger distance measurement and provide a high-level signal of at least 10 us.

The module automatically sends eight 40 kHz square waves and automatically checks whether signals are returned.

When signals are returned, a high-level signal is output through the ECHO pin of the I/O port. The duration of the high-level signal is the duration from the time when the ultrasonic wave is sent to the time when the ultrasonic wave is returned.

Test distance = High-level signal duration x Sound speed (340 m/s)/2.

4.4.1.2 Quiz

Which GPIO pins on the development board correspond to TRIG and ECHO on the ultrasonic sensor module?

[Answer]

TRIG corresponds to GPIO7. ECHO corresponds to GPIO8.

4.4.1.3 Procedure

Step 1 Configure hardware.

Set the **MOTOR_EN** toggle switch from ON to OFF. Connect J19 on the ultrasonic module to J1 on the development board, and connect GPIO7 to TRIG using the J6 jumper cap.

Step 2 Open the **lab3.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab3/lab3.c** in the source code directory.

Step 3 Add header files.

Add the following header files to the part marked with **Step 1: Add Header Files** in the source code:

```
[Answer]
#include "hi_gpio.h"
#include "hi_time.h"
#include "hi_io.h"
```

Step 4 Configure pins of the ultrasonic sensor.

After adding the header files, configure GPIO multiplexing and pin direction of TRIG and ECHO. Add the following configuration code to the part marked with **Step 2: Ultrasonic Module Configuration** in the source code:

```
[Answer]
/*
* Set ultrasonic echo as input mode
* Set GPIO8 function (set as GPIO function)
*/
hi_io_set_func(HI_IO_NAME_GPIO_8, HI_IO_FUNC_GPIO_8_GPIO);
//Set GPIO8 as the input direction
hi_gpio_set_dir(HI_GPIO_IDX_8, HI_GPIO_DIR_IN);
/*
* Set GPIO7 function (set as GPIO function)
*/
hi_io_set_func(HI_IO_NAME_GPIO_7, HI_IO_FUNC_GPIO_7_GPIO);
/*
* Set GPIO7 as the output direction
*/
hi_gpio_set_dir(HI_GPIO_IDX_7, HI_GPIO_DIR_OUT);
```

Step 5 Control the ultrasonic sensor to send pulses.

According to ultrasonic sensor working principles, the sensor sends a square wave first, and the square wave duration is at least 10 us. Add the following code to the part marked with **Step 3: Controls the TRIG port to send pulse** in the source code:

```
[Answer]
hi_gpio_set_ouput_val(HI_GPIO_IDX_7, HI_GPIO_VALUE1);
hi_udelay(DELAY_US10);
hi_gpio_set_ouput_val(HI_GPIO_IDX_7, HI_GPIO_VALUE0);
```

Step 6 Calculate the duration of the high-level signal.

After the pulse is sent, the duration of the ECHO high-level signal is the duration from the time when the pulse is sent to the time when the pulse is received. Add the time calculation code to the part marked with **Step 4: Calculate the high level duration of the echo port** in the source code.

```
[Answer]
/*Judge whether the input level of GPIO8 is high and the flag is 0*/
if (value == HI_GPIO_VALUE1 && flag == 0) {
    /*get SysTime*/
    start_time = hi_get_us();
    flag = 1;
}
/*Judge whether the input level of GPIO8 is low and the flag is 1*/
if (value == HI_GPIO_VALUE0 && flag == 1) {
    /*Get high level duration*/
    time = hi_get_us() - start_time;
    break;
}
```

[API Definition]

hi_u64 hi_get_us(hi_void);

- Function: Obtains the system time (unit: us).
- Return value: system time.

Step 7 Calculate the distance based on the time.

The distance can be calculated based on the obtained time and the calculation formula in 4.4.1.1. Add the following code to the part marked with **Step 5: Calculate distance based on time** in the source code.

```
[Answer]
distance = time * 0.034 / 2;
```

Step 8 Print the result to the serial port.

Add the following code to the part marked with **Step 6: Print the distance measurement result to the serial port** in the source code to print the measurement result to the serial port.

```
[Answer]
printf("distance is %0.2f cm\r\n", distance);
```

Step 9 Open the **BUILD.gn** file and copy the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab3/BUILD.gn** file and add the following content:

```
static_library("lab3") {
  sources = [
    "lab3.c",
```

```
    "cjson_init.c",
    "iot_main.c",
    "iot_profile.c",
    "wifi_connecter.c",
    "iot_hmac.c",
  ]

  include_dirs = [
    "./",
    "//utils/native/lite/include",
    "//kernel/liteos_m/kal/cmsis",
    "//base/iot_hardware/peripheral/interfaces/kits",
    "//device/soc/hisilicon/hi3861v100/sdk_liteos/third_party/lwip_sack/include/lwip",
    "//third_party/cJSON",
    "//device/hisilicon/hispark_pegasus/sdk_liteos/third_party/mbedtls/include/mbedtls",
    "//foundation/communication/wifi_lite/interfaces/wifiservice",
    "//device/hisilicon/hispark_pegasus/sdk_liteos/third_party/paho.mqtt.c/include/mqtt",
    "//device/hisilicon/hispark_pegasus/sdk_liteos/third_party/libcoap/include/coap2",
    ]
}
```

[Remarks]

The source files except **lab3.c** added to the GN file are used for function code implementation when you add other functions in the future. All source files are added here. When other functions are implemented in the future, you do not need to modify the GN file.

Step 10 Open the **BUILD.gn** file and modify the code.

Open the **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/BUILD.gn** file and modify the content as follows:

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

lite_component("app") {
    features = [
        "lab3:lab3",
    ]
}
```

Step 11 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, open the serial port terminal and press the reset button on the development board. After the development board is reset, the following information is displayed on the serial port terminal:

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

```
distance is 24.11 cm
distance is 24.11 cm
distance is 24.11 cm
distance is 24.11 cm
distance is 24.12 cm
distance is 24.11 cm
distance is 24.11 cm
```

The ultrasonic sensor has been enabled and can complete distance measurement. The exercise is successful.

4.4.2 Scenario 2: Registering a Device on the Platform

4.4.2.1 Background

After the ultrasonic sensor is developed, the sensor needs to report the collected data to the platform. Before data reporting, the MQTT device needs to be registered on the platform. After registration, the device can connect to the platform based on its client ID, device ID, and password.

In this exercise, the developer wants to:

- Register the device on IoTDA.
- Obtain the MQTT server IP address.

4.4.2.2 Quiz

How many statuses does a device have on the Huawei Cloud IoT platform?

[Answer]

A device has 5 statuses:

- **ONLINE**: The device is online.
- **OFFLINE**: The device is offline.
- **ABNORMAL**: The device is abnormal.
- **INACTIVE**: The device is not activated.
- **FROZEN**: The device is frozen.

4.4.2.3 Procedure

Step 1 Log in to Huawei Cloud IoTDA.

Log in to the IoTDA console and select a region by referring to the lab environment setup guide. (This document uses the AP-Singapore region as an example.)

Step 2 Register a device.

In the navigation pane on the left, choose **Devices** > **All Devices**.

Click **Individual Register** in the upper right corner. In the displayed dialog box, register a device based on the information provided in the following table.

Table 4-1 Device registration information

Parameter	Value
Resource Space	/
Product	smart_vehicle_mqtt
Node ID	Custom
Device Name	Hi3861
Device ID	Custom (for example, **Hispark**)
Authentication Type	Secret
Secret	Custom (for example, **12345678**)

Step 3 View access information.

In the navigation pane on the left, choose **Overview**. In the middle of the page, click **Access Details**.

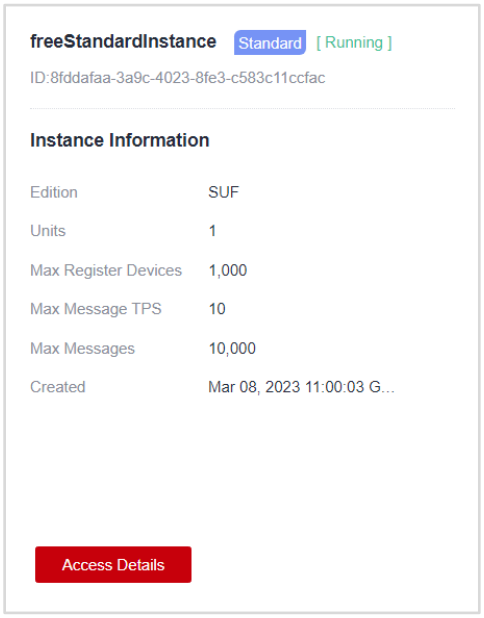


Figure 4-1 Viewing access information

Step 4 Obtain the IP address.

Copy the MQTT access address because MQTT is used for data transmission in this exercise.

Press **win+R** to open the **Run** dialog box, enter **cmd** to open the CLI, and ping the MQTT access address.

Obtain the returned IP address, for example, **119.8.186.222**.

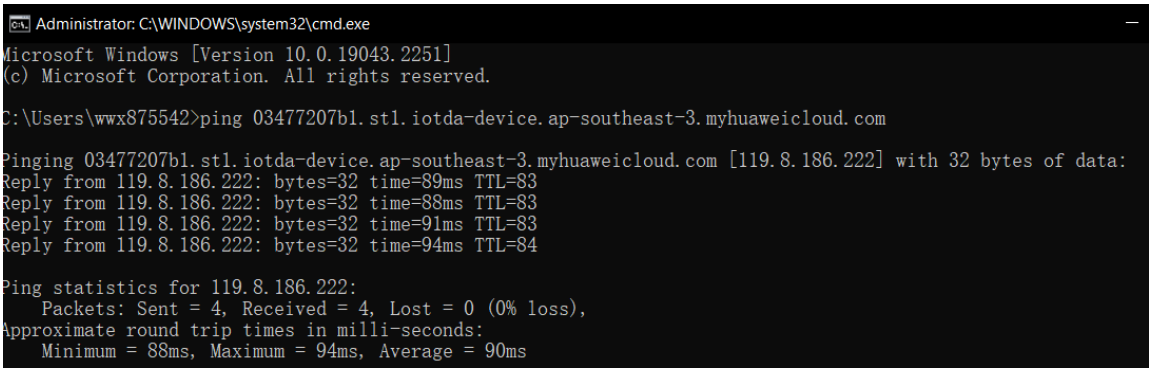


Figure 4-2 Obtaining the IP address

4.4.3 Scenario 3: Reporting Distance Data

4.4.3.1 Background

After the device is registered on the Huawei Cloud IoT platform, the developer needs to upload the data collected by the sensor to the platform.

In this exercise, the developer wants to:

- Connect to the platform over a Wi-Fi network.
- Assemble data to the JSON format and send it to the platform over MQTT.

4.4.3.2 Quiz

How do you upload multiple properties and their values to the platform over MQTT?

[Answer]

When multiple properties need to be reported, the data needs to be assembled into the JSON format, which then is parsed by the platform.

4.4.3.3 Procedure

Step 1 Open the **lab3.c** file.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab3/lab3.c** in the source code directory.

Step 2 Add header files.

Add the following header files to the part marked with **Step 7: Add Header Files** in the source code:

```
[Answer]
#include "wifi_connecter.h"
#include "cjson_init.h"
#include "cJSON.h"
#include "iot_main.h"
#include "iot_profile.h"
```

Step 3 Set the Wi-Fi username and password.

Uncomment the WifiConnect() function. Enter your Wi-Fi username and password in the part marked with **Step 8: Set Wi-Fi ssid & password** in the source code.

```
[Answer]
strcpy(apConfig.ssid, "Huawei"); // Set Wi-Fi ssid
strcpy(apConfig.preSharedKey, "12345678"); // Set Wi-Fi password
```

Step 4 Obtain distance data and convert the data format.

The distance data generated by the sensor is floating-point data. When the data is uploaded to the platform, it needs to be converted into strings.

The content in the data reporting function is the service code for reporting distance data to the platform. Part of the code has been finished. You need to pass the distance data obtained in Scenario 1 to the data reporting function and change the data format to a string. Uncomment IoTPublish() function. Add the following code to the part marked with **Step 9: Obtain distance data and transfer it to string** in the source code:

```
[Answer]
distance = GetDistance();
float_to_string(distance, distance_buff);
```

[API Definition]

hi_u8 *float_to_string(hi_double d, hi_u8 *str);

- Function: Converts a floating-point data to a string.
- **d**: decimal of the double type to be converted.
- **str**: string after conversion.
- Return value: string after conversion.

Step 5 Assemble JSON data.

After the distance data is collected, assemble the data to the JSON format. Uploaded data contains not only the distance property value but also the service name (such as code). In this step, you only need to enter parameter values based on the property structure.

Add the following code to the part marked with **Step 10: Complete each parameter of property** in the source code:

```
[Answer]
property.type = EN_IOT_DATATYPE_STRING;
property.key = "distance";
property.value = (char *)distance_buff;
property.nxt = NULL;
```

Step 6 Upload the assembled data to the platform.

Add the following code to the part marked with **Step 11: Send the assembled JSON data** in the source code to send the data to the platform.

```
[Answer]
```

```
IoTProfilePropertyReport(CONFIG_DEVICE_ID, &service);
```

[API Definition]

int IoTProfilePropertyReport(char *deviceID,IoTProfileService_t *payload);

- Function: Reports properties to the platform.
- **deviceID**: ID of the device registered with the platform.
- **payload**: MQTT payload in JSON format to be uploaded to the platform.
- Return value: **0** indicates a success. Other values indicate a failure.

Step 7 Uncomment the code in the main function.

Uncomment the code in the main function **Lab3()** in the part marked with **Step 12: Uncomment the functions below** in the source code.

Step 8 Comment out the distance measurement function and uncomment the data upload function.

In scenario 1, you need to obtain the distance data directly from the main function. In scenario 2, the data has been obtained from the **IoTPublish()** function, so you need to comment out the **GetDistance()** function and uncomment the **IoTPublish()** function.

Comment out and uncomment the corresponding code in the part marked with **Step 13: Annotate GetDistance function and uncomment the IoTPublish function** in the source code.

Step 9 Configure IoTDA connection parameters.

After the preceding steps are complete, the service code for reporting device data to the IoT platform is developed. Then, configure platform connection parameters in the **iot_main.h** file in the same folder.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab3/iot_main.h** in the source code directory.

Complete the connection information (example) in the part marked with **Step 14: Modify the platform connection information** in the source code.

```
[Answer]
#define CN_IOT_SERVER      "119.8.186.222:1883"
#define CONFIG_DEVICE_ID   "hispark"
#define CONFIG_DEVICE_PWD "12345678"
```

Step 10 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, enable the personal hotspot, open the serial port terminal, and press the reset button on the development board.

If information similar to the following is displayed on the serial port terminal, the device is connected to the platform:

```
IOTSERVER:119.8.186.222:1883
Posting connect semaphore for client hispark_0_0_1970000100 rc 0Connect success
```

```
Subscribe success
```

After the connection is successful, click **Devices** in the navigation pane on the IoTDA console, locate the physical device created in Scenario 2, and click **Detail** on the right to view device details.

If data uploaded by the device is displayed in the **Latest Data Reported** area, data reporting is successful.

4.4.4 Scenario 4: Enabling Platform Command Delivery

4.4.4.1 Background

Now, the developer has completed data reporting and processing. Then, the platform command needs to be delivered to the device, which then responds to the command.

In the smart vehicle scenario, a vehicle can be remotely controlled. For example, a vehicle door can be opened remotely. The developer uses an LED to indicate the door status to simulate the above scenario.

In this exercise, the developer wants to:

- Use online debugging of IoTDA to deliver the commands for opening and closing the door.
- Use a red LED (LED3) to simulate the door. If the LED is on, it indicates the door is open. If the LED is off, it indicates the door is closed.

4.4.4.2 Quiz

Are the topic names used for device data reporting and platform command delivery the same during MQTT development on the Huawei Cloud IoT platform?

[Answer]

No. The topic name used for device data reporting is **$oc/devices/{device_id}/sys/properties/report**, and the topic name used for command delivery is **$oc/devices/{device_id}/sys/commands/request_id={request_id}**.

4.4.4.3 Procedure

Step 1 Open the **lab3.c** file.

In the preceding steps, the preparation for delivering the command on the platform is complete. In the following steps, you need to edit the code for device command response.

Open **hi3861_hdu_iot_application-master/src/applications/sample/wifi-iot/app/lab3/lab3.c** in the source code directory.

Step 2 Configure LED pins.

LED3 is used to simulate the door opening and closing effect, so you need to configure LED3 pins before coding for device command response.

Add the LED3 GPIO multiplexing and pin direction configuration code in the part marked with **Step 15: LED Configuration** in the source code.

```
[Answer]
//GPIO Set the pin reuse relationship of GPIO9 to GPIO
```

```
hi_io_set_func(HI_IO_NAME_GPIO_9, HI_IO_FUNC_GPIO_9_GPIO);
//GPIO direction set to output
hi_gpio_set_dir(HI_GPIO_IDX_9, HI_GPIO_DIR_OUT);
```

Step 3 Complete the service code.

After JSON data is parsed, the variable **params** indicates the parameter value of the delivered command. You need to determine whether the parameter is **ON** or **OFF** and write the code for turning on or off the LED based on the parameter value.

Complete the service code in the part marked with **Step 16: Complete Command Response Control Service Code** in the source code.

```
[Answer]
if (strcmp(value, "ON") == 0)
{
    hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE1);
}
else
{
    hi_gpio_set_ouput_val(HI_GPIO_IDX_9, HI_GPIO_VALUE0);
}
```

Step 4 Uncomment the callback function.

In the main program of the **Lab3.c** file, a callback function is used to receive commands and respond to the commands. You need to uncomment the callback function.

Uncomment the callback function in the part marked with **Step 17: Uncomment the Command response function** in the source code.

Step 5 Compile and burn the program and view the result.

After writing the code, click **Build** on the DevEco Device Tool page to recompile the code.

After compilation, click **Upload** to burn code. Then, enable the personal hotspot, open the serial port terminal, and press the reset button on the development board.

Access the IoTDA console. In the navigation pane, choose **O&M** > **Online Debug**.

Click **Select Device** in the upper right corner and select the physical device created in Scenario 2. On the debugging page, set **door** to **ON** and click **Send**.

The delivered command information is displayed in the application simulator area, and the red light on the development board turns on.

```
Message Body: { "service_id": "SmartVehicle", "command_name": "SMART_VEHICLE_DOOR", "paras": { "door": "ON" } }
```

The platform command delivery is verified. The exercise is complete.

5 Appendix: Acronyms and Abbreviations

Table 5-1 Acronyms and abbreviations

Acronym/Abbreviation	Full Name
AT	Attention
CRLF	Carriage-Return Line-Feed
CSS	Cascading Style Sheets
GPIO	General-purpose Input/Output
HTML	HyperText Mark-up Language
I2C	Inter-integrated Circuit
IoT	Internet of Things
JS	JavaScript
JSON	JavaScript Object Notation
LED	Light-emitting Diode
MQTT	Message Queuing Telemetry Transport
NAT	Network Address Translation
PWM	Pulse Width Modulation
RPC	Remote Procedure Call
RX	Receiver
SCL	Serial Clock
SDA	Serial Data
STA	Station
TX	Transmitter
UART	Universal Asynchronous Receiver/Transmitter