EAIDK-610 公开教程

C 版

2020/08/06

OPEN AI LAB

版本 V0.1.4

目录(catalog)

1 前言

1.1 目的

本文档主要介绍 EAIDK-610 产品基本功能，硬件特性，软件特点及软件调试操作方法，旨在帮助开发人员更快、更好熟悉 EAIDK-610 硬件平台（EAIDK-610）和使用 EAIDK-610 开发套件。

1.2 术语

- **EAIDK**：Embedded AI Development Kit。嵌入式人工智能开发套件。
- **AID**：AID 是 OPEN AI LAB 开发的一个面向嵌入式平台前端智能，跨 SoC 的 AI 核心软件平台。
- **GPIO**：通用型之输入输出。
- **IIC**：Inter-Integrated Circuit（集成电路总线）
- **SPI**：Serial Peripheral Interface（串行外设接口）
- **bme280**：博世气压，温湿度三合一传感器

2 硬件介绍

2.1 硬件总览

EAIDK-610 采用 8 层板设计，沉金工艺。正面如图 2-1 所示，背面如图 2- 2 所示。

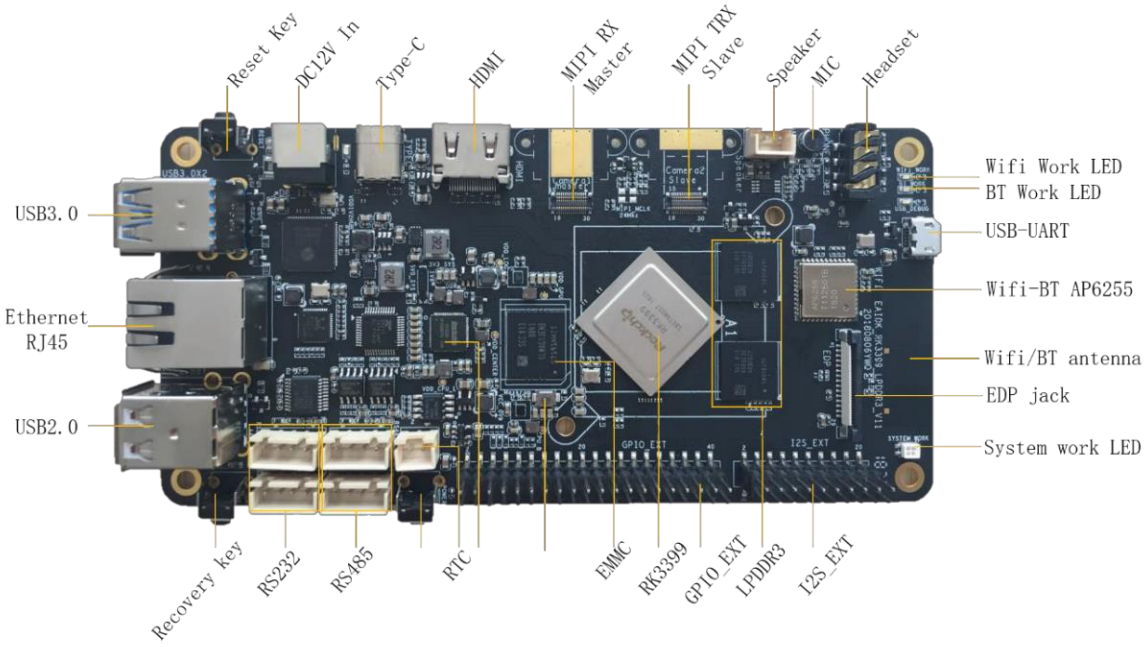

图 2-1 Top Layer 接口图

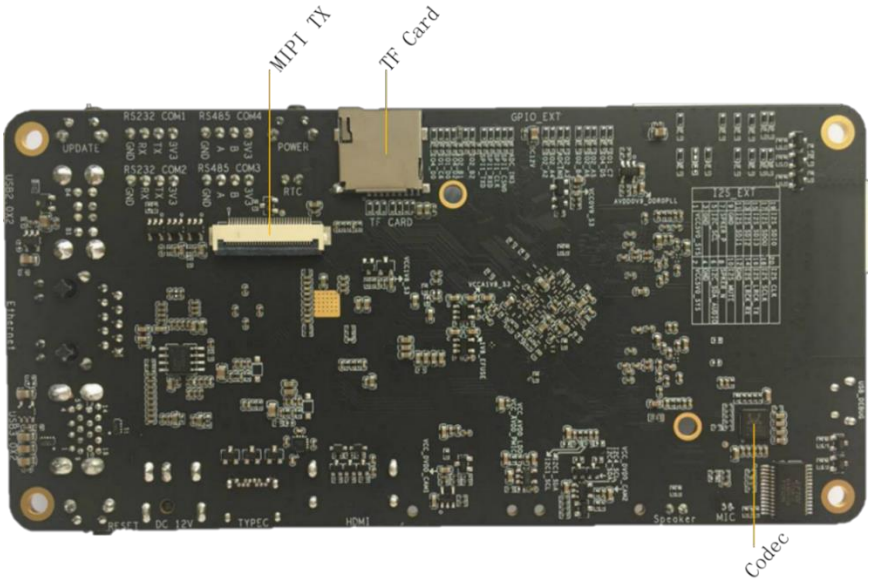

图 2-2 Bottom Layer 接口图

2.2 调试接口

开发板提供调试串口供开发调试使用。调试串口连接主控的 UART2 接口，通过板上集成 FT232RL UART 转 USB 接口转换芯片，外接 Micro USB 座子。用户只需要一个普通 Micro USB 线即可。

说明：EAIDK-610 的调试串口的波特率为 1500000。

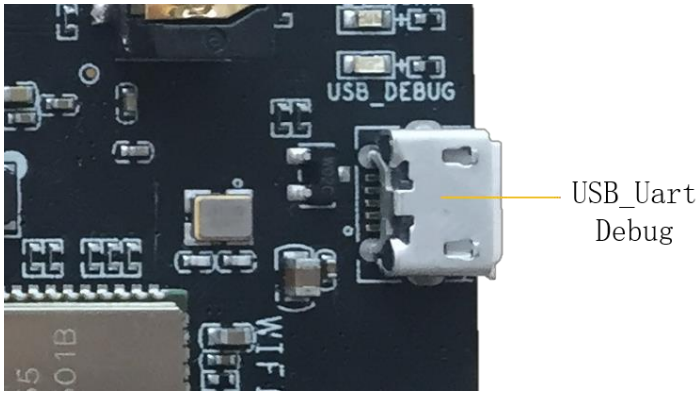

图 2-3 USB UART Debug 接口示意图

2.3 电源模块

EAIDK-610 开发板的电源模块采用 PMIC RK808 为核心芯片，配合外围的 Buck、LDO 组成。

图 2-4 DC 输入接口示意图

2.4 存储模块

2.4.1 内存

EAIDK-610 开发板采用两颗 32bit 2GB LPDDR3 颗粒，构成 64bit 4GB DDR。

图 2-5 LPDDR3 位置示意图

2.4.2 EMMC

EAIDK-610 开发板采用 EMMC 作为系统盘，默认容量 16GB。

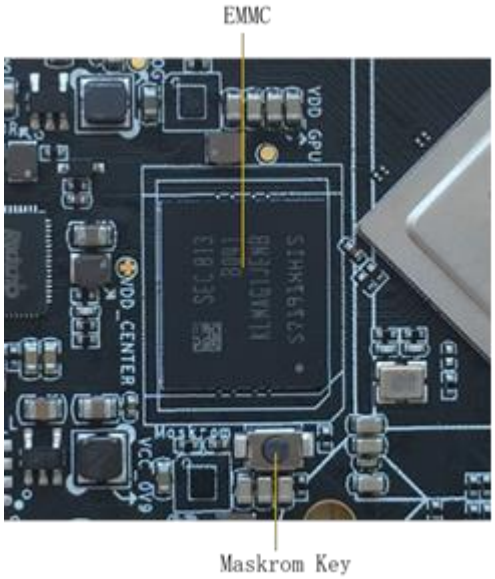

图 2-6 EMMC 位置示意图

2.4.3 TF 卡

EAIDK-610 开发板带有 TF Card 卡座，连接 RK3399 SDMMC0。数据总线宽带为 4bit，支持热插拔。

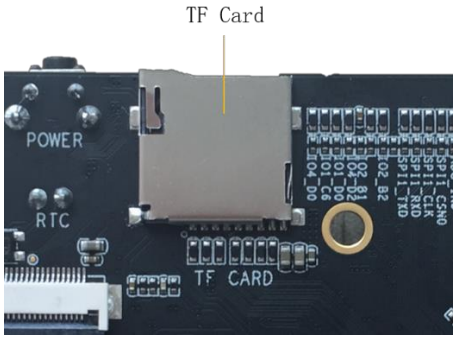

图 2-7 TFcard 位置示意图

2.5 显示模块

2.5.1 MIPI 显示

EAIDK-610 开发板标配显示屏为 5.5 寸 720P MIPI 显示屏，支持 5 点触摸。

图 2-8 MIPI-TX 连接座位置示意图

MIPI 管脚定义如下表所示：

表 2-1 MIPI_TX 管脚定义表

Pin	Name	Pin	Name
1	GND	**16**	GND
2	MIPI_TX0_D0N	**17**	LCD_BL_PWM
3	MIPI_TX0_D0P	**18**	NC
4	GND	**19**	NC
5	MIPI_TX0_D1N	**20**	LCD_RST_H
6	MIPI_TX0_D1P	**21**	GND
7	GND	**22**	LCD_EN_H
8	MIPI_TX0_CLKN	**23**	I2C_SCL_TP
9	MIPI_TX0_CLKP	**24**	I2C_SDA_TP
10	GND	**25**	TOUCH_INT_L
11	MIPI_TX0_D2P	**26**	TOUCH_RST_L
12	MIPI_TX0_D2N	**27**	GND
13	GND	**28**	VCC5V0_SYS
14	MIPI_TX0_D3N	**29**	VCC5V0_SYS
15	MIPI_TX0_D3P	**30**	VCC5V0_SYS

2.5.2 eDP 显示

EAIDK-610 开发板可选配件为 7.85 寸 2K eDP 显示屏，支持 5 点触摸。

图 2-9 EDP 连接示意图

eDP 管脚定义如下表所示：

表 2-2 EDP 管脚定义表

Pin	Name	Pin	Name
1	GND	16	GND
2	EDP_TX0N	17	LCD_BL_PWM
3	EDP_TX0P	18	GND
4	GND	19	VCC3V3_S0
5	EDP_TX1N	20	LCD_RST_H
6	EDP_TX1P	21	NC
7	GND	22	LCD_EN_H
8	EDP_AUXN	23	I2C_SCL_TP
9	EDP_AUXP	24	I2C_SDA_TP
10	GND	25	TOUCH_INT_L
11	EDP_TX2N	26	TOUCH_RST_L
12	EDP_TX2P	27	GND
13	GND	28	VCC5V0_SYS
14	EDP_TX3N	29	VCC5V0_SYS
15	EDP_TX3P	30	VCC5V0_SYS

2.5.3 HDMI 显示

EAIDK-610 开发板支持 HDMI 显示，采用 A 型接口，可以同其他显示接口组成双屏 显示：双屏同显和双屏异显。

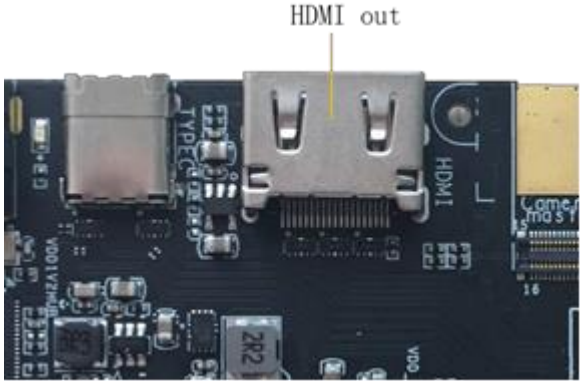

图 2-10 HDMI 位置示意图

2.6 MIPI 相机接口

EAIDK-610 开发板拥有 2 路 MIPI Camera 接口，可外接 2 个 OV9750 摄像头组成双 MIPI Camera 同步显示和前后摄像模式；也可外接 1 路 IMX258 实现 4K 高清摄像。

开发板上 2 路 MIPI 接口采用兼容设计。用户只需要设计简单的电源转换电路即可匹配其他 Camera 模组。

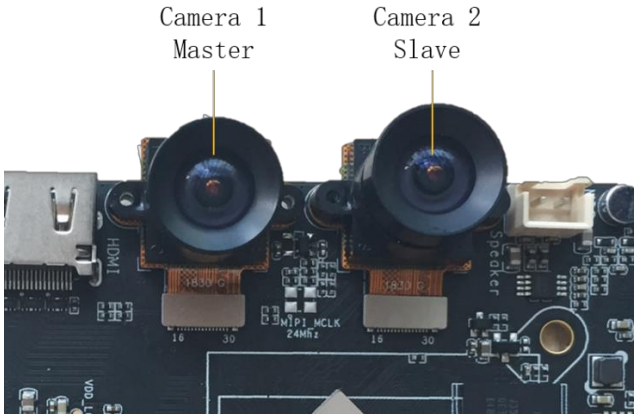

图 2-11 Camera 位置示意图

表 2-3 MIPI Rx0 管脚定义表

Pin	Name	Pin	Name
1	GND	**16**	GND
2	MIPI_RX0_D0P	**17**	VCC_AVDD
3	MIPI_RX0_D0N	**18**	VCC_AVDD
4	GND	**19**	GND
5	MIPI_RX0_D2P	**20**	I2C_SCL_1V8_CAM1
6	MIPI_RX0_D2N	**21**	I2C_SDA_1V8_CAM1
7	GND	**22**	VCC_DVDD_CAM11
8	MIPI_RX0_D3P	**23**	GND
9	MIPI_RX0_D3N	**24**	VCC1V8_DVP
10	GND	**25**	GND
11	MIPI_MCLK_CAM1	**26**	MIPI_RX0_D1N
12	MIPI_RST_CAM1J	**27**	MIPI_RX0_D1P
13	GND	**28**	GND
14	MIPI_PDN_CAM1J	**29**	MIPI_RX0_CLKP
15	FSIN/VSYNC1	**30**	MIPI_RX0_CLKN

2.7 音频模块

EAIDK-610 开发板集成 Realtek ALC5651 Codec 芯片，内置 Charge Pump，板载 MIC。支持立体声耳机无电容耦合输出和耳麦输入；支持麦克风差分输入，Speaker 输出。

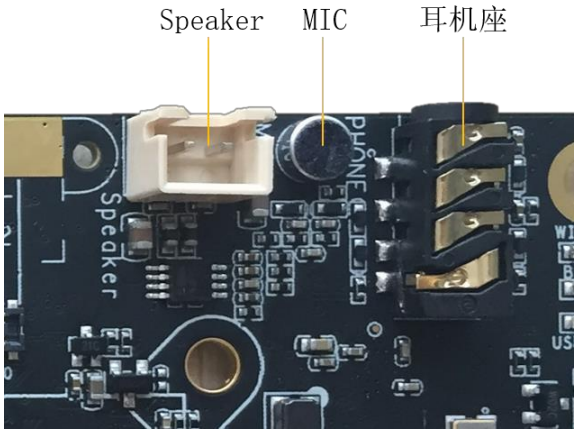

图 2-12 音频接口位置示意图

2.8 USB 模块

2.8.1 USB Host

EAIDK-610 开发板集成 2 路 USB2.0 Host 和 2 路 USB3.0 Host。外接 USB 鼠标、键盘 和 U 盘等多种的人机交互方式。

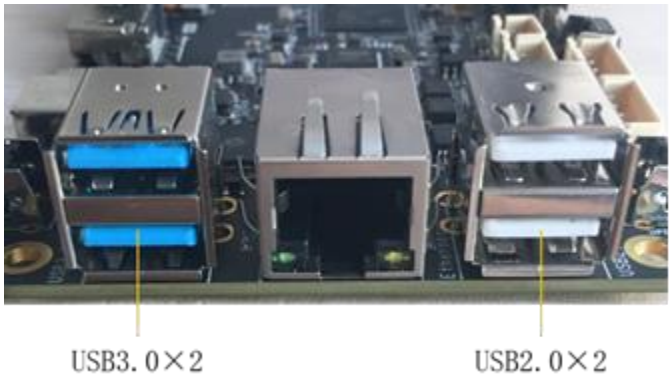

图 2-13 USB 位置示意图

2.8.2 Type-C

EAIDK-610 集成 Type-C 接口，支持 USB OTG 功能。可作为 android 的 adb device；当外接 USB 鼠标、键盘和 U 盘等多种的人机交互方式时，自动切换到 Host 模式。

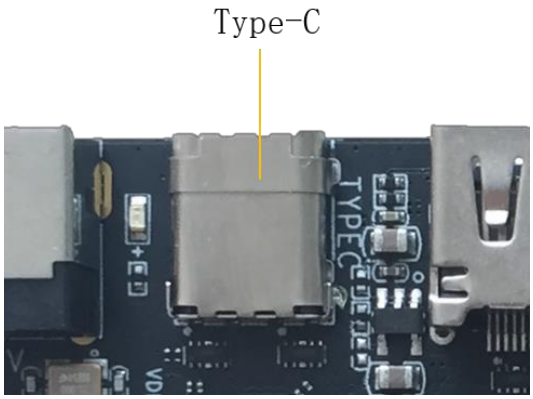

图 2-14 Type-C 位置示意图

2.9 网络通讯

2.9.1 以太网

EAIDK-610 开发板支持 RJ45 接口，可提供千兆以太网连接功能，选用 PHY 为 RTL8211E-VB-CG，其特性如下：

- 兼容 IEEE802.3 标准，支持全双工和半双工操作，支持交叉检测和自适应
- 支持 10/100/1000M 数据速率。
- 接口采用具有指示灯和隔离变压器的 RJ45 接口。

图 2-15 RJ45 位置示意图

2.9.2 WIFI/BT

开发板上 WIFI+BT 模组采用台湾正基的 AP6255，其特性如下：

- 支持 WIFI 2.4G 和 5G，802.11 ac，采用 4bits SDIO 通讯。
- 支持 BT4.1 功能，采用 UART 通讯。

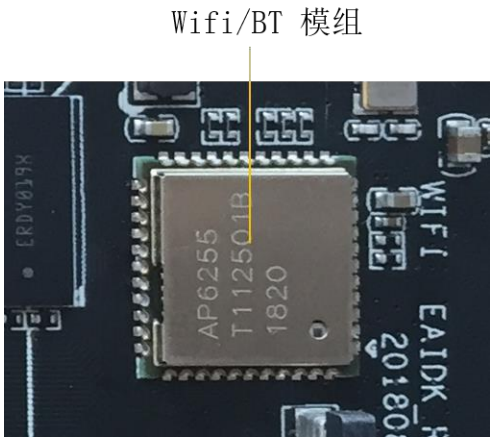

图 2-16 WIFI/BT 模组示意图

2.10 低速 IO 接口

EAIDK-610 集成 40 Pins IO 扩展接口和 1 路 8 通道 I2S 接口。其配套低速 IO 配件，支 持 I2C/SPI/ADC/GPIO 教学实验和 6 路麦克风阵列+ADC 回采。

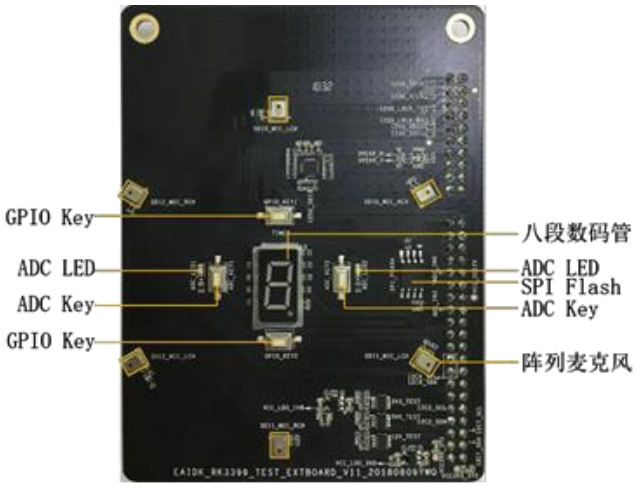

图 2-17 低速 IO 板示意图

低速接口 GPIO_EXT 的管脚定义如表 3-4，I2S 接口 I2S_EXT 的管脚定义如表 3-5

表 2-4 GPIO_EXT 管脚定义表

Pin	Name	Pin	Name
1	VCC3V3_SYS	2	VCC5V0_SYS
3	GPIO2_A7/I2C7_SDA	4	VCC5V0_SYS
5	GPIO2_B0/I2C7_SCL	6	GND
7	GPIO4_D0	8	GPIO2_A0/I2C2_SDA
9	GND	10	GPIO2_A1/I2C2_SCL
11	GPIO1_C6	12	
13	GPIO1_D0	14	GND
15	GPIO4_D2	16	GPIO2_B1/I2C6_SDA
17	VCC3V3_SYS	18	GPIO2_B2/I2C6_SCL
19	SPI1_TXD	20	GND
21	SPI1_RXD	22	
23	SPI1_CLK	24	SPI1_CSn0
25	GND	26	ADC_IN3
27	VCC_DC12V	28	VCC_DC12V
29	GPIO2_A2	30	GND
31	GPIO2_A4	32	ADC_IN0
33	GPIO2_A3	34	GND
35	GPIO2_B4	36	GPIO2_A6
37	GPIO4_D5	38	GPIO2_A5
39	GND	40	GPIO1_C7

表 2-5 I2S_EXT 管脚定义表

Pin	Name	Pin	Name
1	VCC5V0_SYS	**11**	I2S0_SDI1
2	VCC5V0_SYS	**12**	GND
3	GND	**13**	I2S0_SDI2
4	GND	**14**	I2S0_LRCK_RX
5	I2C_SCL_AUDIO	**15**	I2S0_SDI3
6	I2C_SDA_AUDIO	**16**	I2S0_LRCK_TX
7	SPKER_P	**17**	I2S0_SDO0
8	SPKER_N	**18**	I2S0_SCLK

9	GND	**19**	I2S0_SDI0
10	I2S_MUTE	**20**	I2S_CLK

2.11 UART 接口

2.11.1 RS232

EAIDK-610 集成 2 路 RS232 接口，支持双工通讯，支持软件标准 UART 编程。

2.11.2 RS485

EAIDK-610 集成 2 路 RS485 接口，支持半双工通讯，支持软件标准 UART 编程。

3 连接外部设备

3.1 登录

1. 连接电源，启动 EAIDK
2. 连接鼠标键盘，输入用户名密码 openailab/oal20200230（如果需要输入用户名密码）,登录 EAIDK

   连接线和输入界面如下图所示：

图 3-1 连接线正面照片

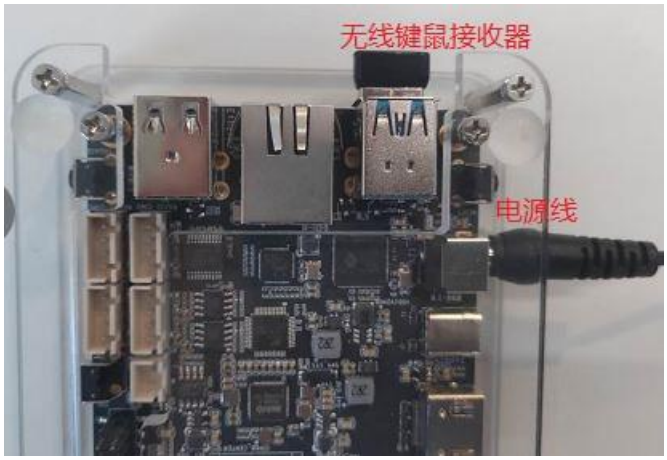

图 3-2 连接线背面照片

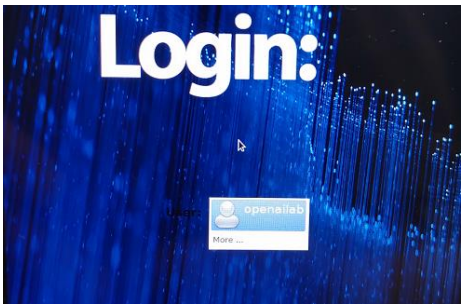

图 3-3 登录界面

3.2 网络配置

3.2.1 连接有线网络(以 IPv4 为例)

1. 使用网线连接 EAIDK-610 与交换机。
2. 右键点击屏幕右下角，网络连接图标，选择 Edit Connections。

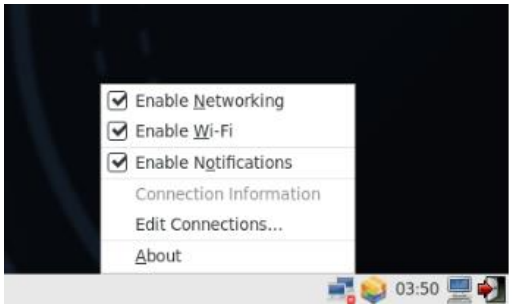

图 3-4 编辑网络设置

3. 双击 Wired connection 1,选择 IPv4 Settings。

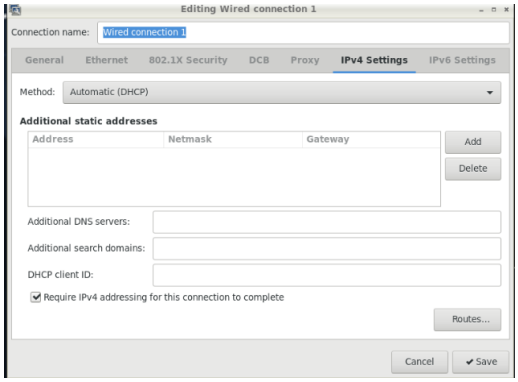

图 3-5 有线设置

4. 如果使用 DHCP，删除已经配置好的静态 ip，method 选择 Automatic(DHCP),如果需要手动设置 IP，则 Method 选择 Manual,并点击 Add 按钮，输入要设置的 IP，掩码和网关，并点击 Save 按钮。

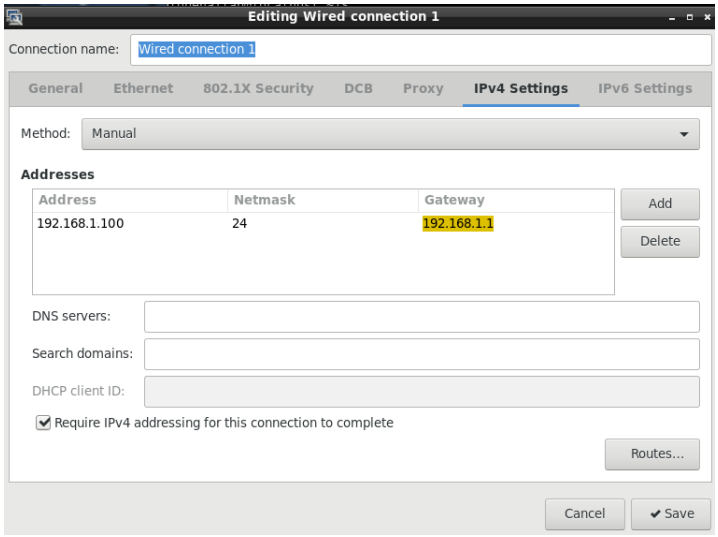

图 3-6 有线网络设置

3.2.2 连接 WIFI

1. 左键点击右下角网络连接图标

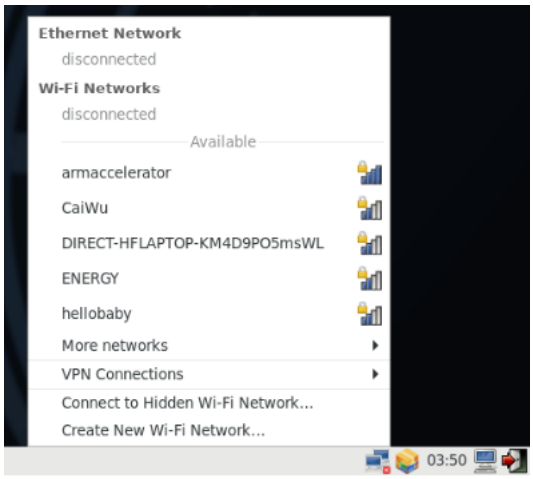

图 3-7 无线网络

2. 点击需要连接的 wifi，输入密码，点击 Connect 按钮

图 3-8 输入无线密码

4 软件及开发

4.1 系统登陆

EAIDK-610 预装 Fedora 28 及轻量级桌面系统 LXDE。缺省登录账号为 openailab，密码为 oal20200230

4.2 软件下载

EAIDK-610 的固件和源码都可以从 EAIDK 的官方 FTP 获取，其地址为 ftp://ftp.eaidk.net

4.3 环境设置

4.3.1 添加源（默认已经配置）

1. 安装 RK3399 硬件相关的系统库 RPM 包的源：

```
sudo yum -y localinstall --nogpgcheck  http://www.eaidk.net/rockchip/rockchip-repo-1.0-1.fc28.aarch64.rpm
```

2. 安装 Openailab 的系统和应用 RPM 包的源：

```
sudo yum -y localinstall --nogpgcheck http://www.eaidk.net/openailab/openailab-repo-1.0-1.fc28.aarch64.rpm
```

3. 安装第三方非开源的 RPM 包的源（可选）：

```
sudo yum localinstall --nogpgcheck http://download1.rpmfusion.org/free/fedora/rpmfusion-free-release-28.noarch.rpm

sudo yum localinstall --nogpgcheck http://download1.rpmfusion.org/nonfree/fedora/rpmfusion-nonfree-release-28.noarch.rpm
```

4.3.2 安装 RPM 包（默认已经安装）

1. 安装编译工具：
   sudo dnf -y install gcc gcc-c++ cmake automake
2. 安装 openssl(开发板上编译内核需要)：
   sudo dnf -y install openssl-devel

4.4 固件烧写

4.4.1 Windows 主机烧写

首次烧写前需要安装 Windows PC 端 USB 驱动：

双击 Tools\Windows\DriverAssitant_v4.5\ DriverInstall.exe 打开安装程序，点击“驱动安装”按提示安装驱动即可，安装界面如下所示：

图 4-1 安装界面图

驱动安装完成后，固件烧写步骤如下：

1. Type-C 线连接主机端的 USB 接口和 EAIDK-610 开发板的 Type-C 接口。
2. 双击 Tools\Windows\EAIDK_FlashTool\ EAIDK_FlashTool.exe 打开程序。
3. 长按 EAIDK-610 开发板上 update 按键后重启机器，直到系统进入 Loader 模式，FlashTool 显示如下所示：

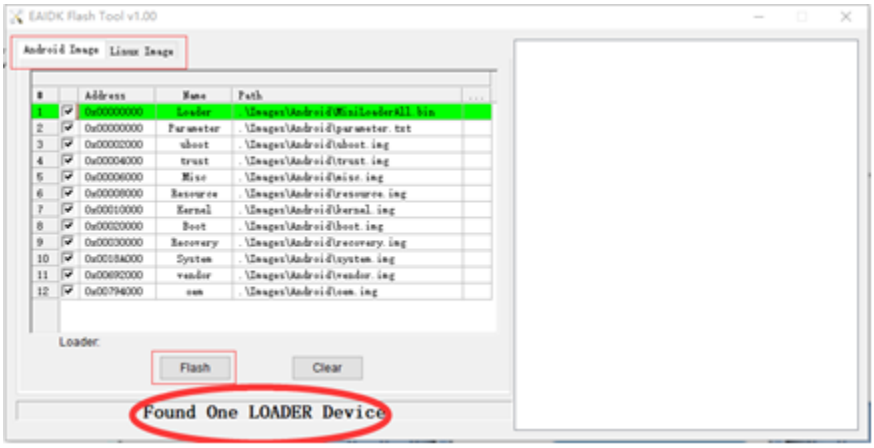

图 4-2 FlashTool 显示界面图

注意：如果 uboot 启动异常而无法进入 Loader 模式，需要长按 Maskrom 按键让系统进入 Maskrom 模式;FlashTool 提示：Found one Maskrom Device!

4. 将 Android 固件拷贝到 Tools\Windows\EAIDK_FlashTool\Images\Android 目录，或将 Linux 固件拷贝到 Tools\Windows\EAIDK_FlashTool\Images\Linux 目录。
5. 点击 TAB“Android Image”或“Linux Image”，选择要烧写的系统固件。
6. 在选择好的系统固件页面，点击按钮“Flash”，开始烧写固件。

4.4.2 Linux 主机烧写

1. Type-C 线连接主机端的 USB 接口和 EAIDK-610 开发板的 Type-C 接口。

说明：EAIDK-610 的 fedora28 系统也可以作为开发主机给其他开发板烧写固件。

2. 长按 EAIDK-610 开发板上 update 按键后重启机器，进入 Loader 模式。
3. 进入 Tools/Linux 目录：

```
cd Tools/Linux
```

4. 将 Android 固件拷贝到 Tools/Linux/Images/Android 目录。
5. 将 Linux 固件拷贝到 Tools/Linux/Images/Linux 目录。
6. 执行命令烧写固件：

```
./flash.sh
```

1) 烧写 Android 固件：

./flash.sh -a: 烧写所有 Android 固件。

./flash.sh -a boot: 烧写 Android 的 Miniloader.img、trust.img 和 uboot.img。

./flash.sh -a kernel: 烧写 Android 的 kernel.img 和 resource.img。

./flash.sh -a system: 烧写 Android 的 system.img。

./flash.sh -a android: 烧写 Android 的 boot.img、system.img、misc.img、recovery.img、vendor.img 和 oem.img。

2) 烧写 Linux 固件：

./flash.sh -l：烧写所有 Linux 固件

./flash.sh -l boot: 烧写 Linux 的 Miniloader.img、trust.img 和 uboot.img。

./flash.sh -l kernel: 烧写 Linux 的 kernel.img 和 resource.img。

./flash.sh -l rootfs：烧写 fedora 根文件系统。

5 GPIO 编程

5.1 GPIO 分布图

EAIDK-610 管脚分布如下图，Pin1 和 Pin2 已经用红圈标出，其他管脚以此类推。

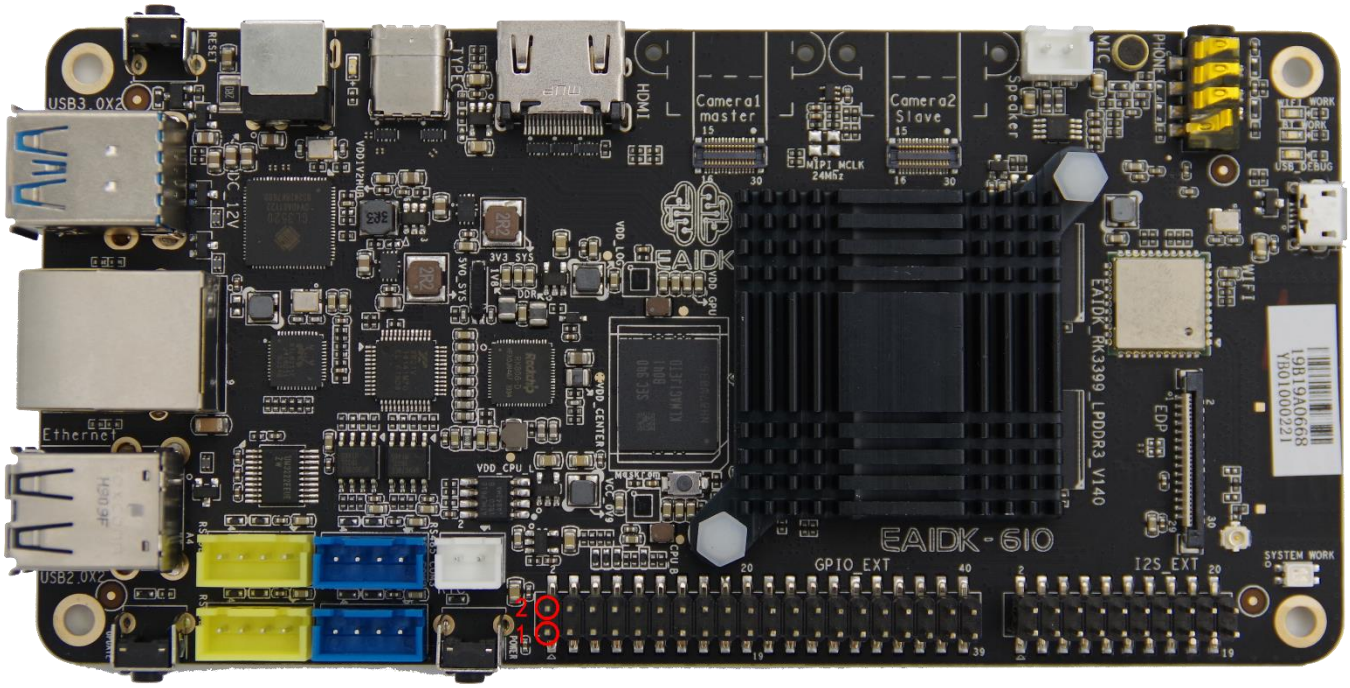

图 5-1 管脚分布图

各个管脚定义如下表

表 5-1 管脚定义

Pin	Name	Pin	Name
1	VCC3V3_SYS	2	VCC5V0_SYS
3	GPIO2_A7/I2C7_SDA	4	VCC5V0_SYS
5	GPIO2_B0/I2C7_SCL	6	GND
7	GPIO4_D0	8	GPIO2_A0/I2C2_SDA
9	GND	10	GPIO2_A1/I2C2_SCL
11	GPIO1_C6	12	
13	GPIO1_D0	14	GND
15	GPIO4_D2	16	GPIO2_B1/I2C6_SDA
17	VCC3V3_SYS	18	GPIO2_B2/I2C6_SCL
19	SPI1_TXD	20	GND

21	SPI1_RXD	22	
23	SPI1_CLK	24	SPI1_CSn0
25	GND	26	ADC_IN3
27	VCC_DC12V	28	VCC_DC12V
29	GPIO2_A2	30	GND
31	GPIO2_A4	32	ADC_IN0
33	GPIO2_A3	34	GND
35	GPIO2_B4	36	GPIO2_A6
37	GPIO4_D5	38	GPIO2_A5
39	GND	40	GPIO1_C7

从上表可以得出。PIN3, PIN5, PIN7, PIN8, PIN10, PIN11, PIN13, PIN15, PIN16, PIN18, PIN29, PIN31, PIN33, PIN35, PIN36, PIN37, PIN38, PIN40 为 GPIO。

GPIO 设置成输出模式时，高电平为 3.3V，低电平为 0V。

GPIO 设置成输入模式时，高于 1.8V 为高电平，低于 1.8V 为低电平。

5.2 控制 gpio

从图 2.2-2 可以看出，GPIO 的 Name 格式都是 GPIOX_YZ 形式，其中 X，Z 是一个数字，Y 是 A-D 的大写字母，引脚号计算公式为

$$X * 32 + F(Y) * 8 + Z \qquad 式1$$

其中当 Y=A,B,C,D 时, $F(Y)$分别为 0,1,2,3。

以 GPIO2_A7 为例。通过式 1，计算出 GPIO2_A7 引脚号为$2 * 32 + 0 * 8 + 7 = 71$。

1. 切换 root 权限。

```
[openailab@localhost ~]$ sudo su
```

2. 进入 /sys/class/gpio/ 目录 。

```
[root@localhost openailab]# cd /sys/class/gpio/
```

3. 生成一个 gpio71 的目录。

```
[root@localhost gpio]# echo 71 > export
```

4. 进入 gpio71 目录。

```
[root@localhost gpio]# cd gpio71/
```

gpio71 目录下主要有以下文件

1)direction： 设置输出还是输入模式

设置输入模式。

```
[root@localhost gpio71]# echo in > direction
```

设置输出模式。

```
[root@localhost gpio71]# echo out > direction
```

2)value： 输出时，控制高低电平；输入时，获取高低电平

设置高电平。

```
[root@localhost gpio71]# echo 1 > value
```

设置低电平。

```
[root@localhost gpio71]# echo 0 > value
```

3)edge： 控制中断触发模式，引脚被配置为中断后可以使用 poll() 函数监听引脚

非中断引脚：

```
[root@localhost gpio71]#  echo “none” > edge
```

上升沿触发：

```
[root@localhost gpio71]# echo “rising” > edge
```

下降沿触发：

```
[root@localhost gpio71]# echo “falling” > edge
```

边沿触发：

```
[root@localhost gpio71]# echo “both” > edge
```

5.如果想要卸载 gpio71，需要回到`/sys/class/gpio/` 目录并执行 `echo 71 > unexport`

5.3 点亮 LED 实例

LED 实物如图所示。

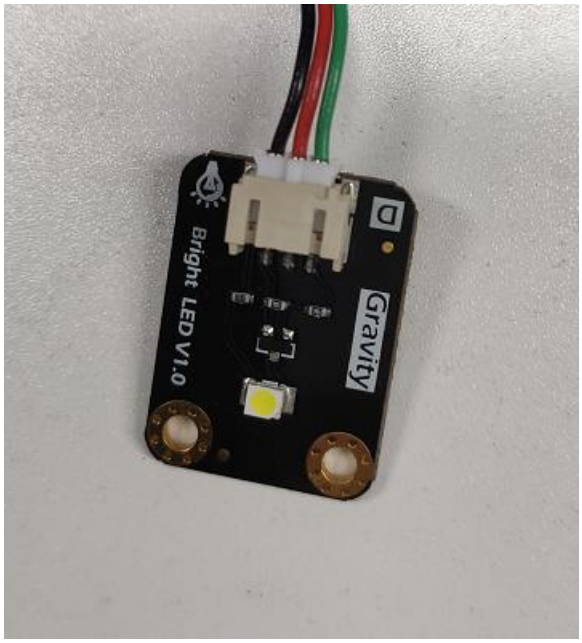

图 5-2 LED 实物

将 LED 的黑线接地，红线接 VCC，绿线接 GPIO1_C6。可以通过控制 GPIO1_C6 电位来观察 LED 是否发光。电路连线如下图。

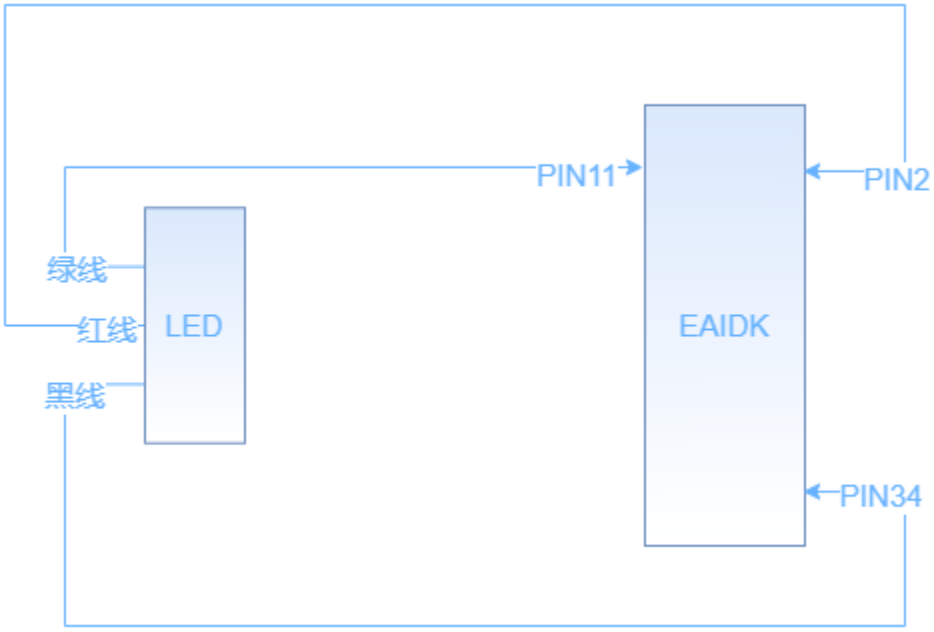

图 5-3 LED 实例电路图

根据 EAIDK 管脚分布图，PIN2 为 5V 直流电压，PIN11 为 GPIO1_C6,PIN34 为地。

把 PIN11 设置成输出模式。当 PIN11 为高电位时，LED 被点亮；当 PIN11 为低电位时，LED 被熄灭。

5.4 捕获按钮按下实例

按钮实物如图所示。

图 5-4 按钮实物

将按钮的黑线接地，红线接 VCC，绿线接 GPIO1_C6。可以通过观察 GPIO1_C6 电位来检测按钮是否被按下。电路连线如下图。

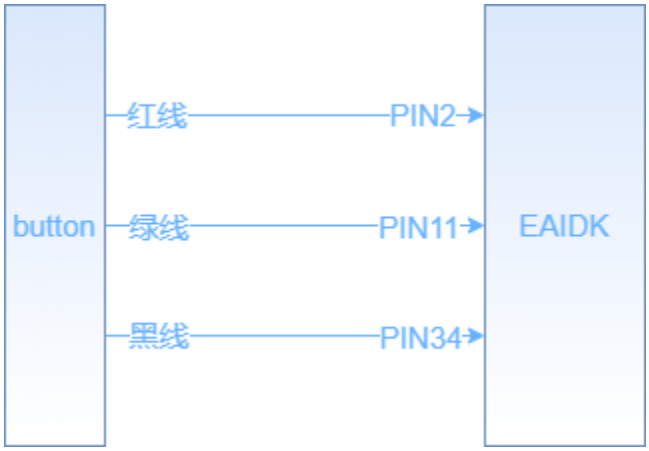

图 5-5 按钮实例电路图

根据 EAIDK 管脚分布图，PIN2 为 5V 直流电压，PIN11 为 GPIO1_C6,PIN34 为地。

把 PIN11 设置成输入模式。当按钮没有被按下，PIN11 为低电位；当按钮被按下，PIN11 为高电位。

5.5 点亮数码管实例

四位串行数码管实物图如下。

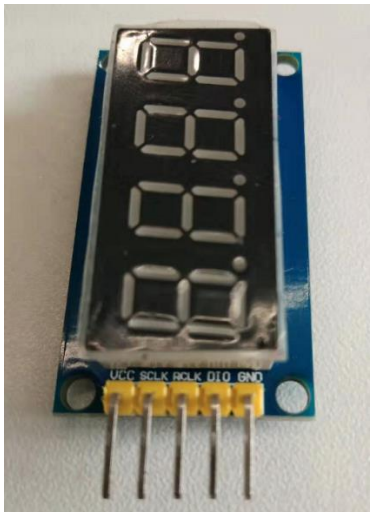

图 5-6 数码管实物

从上图可知，数码管有 5 个管脚，各管脚作用如下

表 5-2 四位串行数码管管脚定义

管脚	作用
VCC	接 5v 直流电，与 GND 一起为数码管供电
SCLK	位输入确认时钟，每次向 DIO 输入一个 bit 后要给 SCLK 一个低电平脉冲
RCLK	字节位置输入确认时钟，每次向 DIO 输入一个字节和位置后要给 RCLK 一个低电平脉冲
DIO	数据输入
GND	接地，与 VCC 一起为数码管供电

DIO 输入的数据有两种作用。一种是在数码管显示数字，一种是指定哪一个数码管显示。

1.显示数字

DIO 需要输入一个 8bit 的字节来表示需要显示的数字。一个 8 段数码管引脚定义如下。

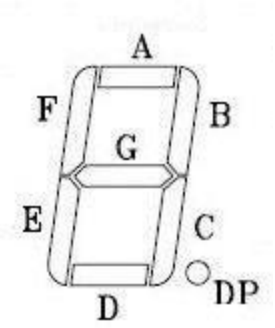

图 5-7 八段数码管引脚定义

输入的 8bit 字节第一位至第八位分别控制 DP,G,F,E,D,C,B,A。低电平点亮，高电平熄灭。比如，当 DIO 输入 0xF9 时，字节第 6 位和第 7 位是低电平，其余位为高电平。第 6 位和第 7 位分别对应数码管引脚 C 和 B，C 和 B 点亮，显示数字'1'。

2.指定数码管

DIO 需要输入一个 8bit 的字节来指定显示数字的数码管。四位数码管总共有四个数码管，输入 0x08 指定第一个数码管，输入 0x04 指定第二个数码管, 输入 0x02 指定第三个数码管，输入 0x01 指定第四个数码管。

电路连线如下图。

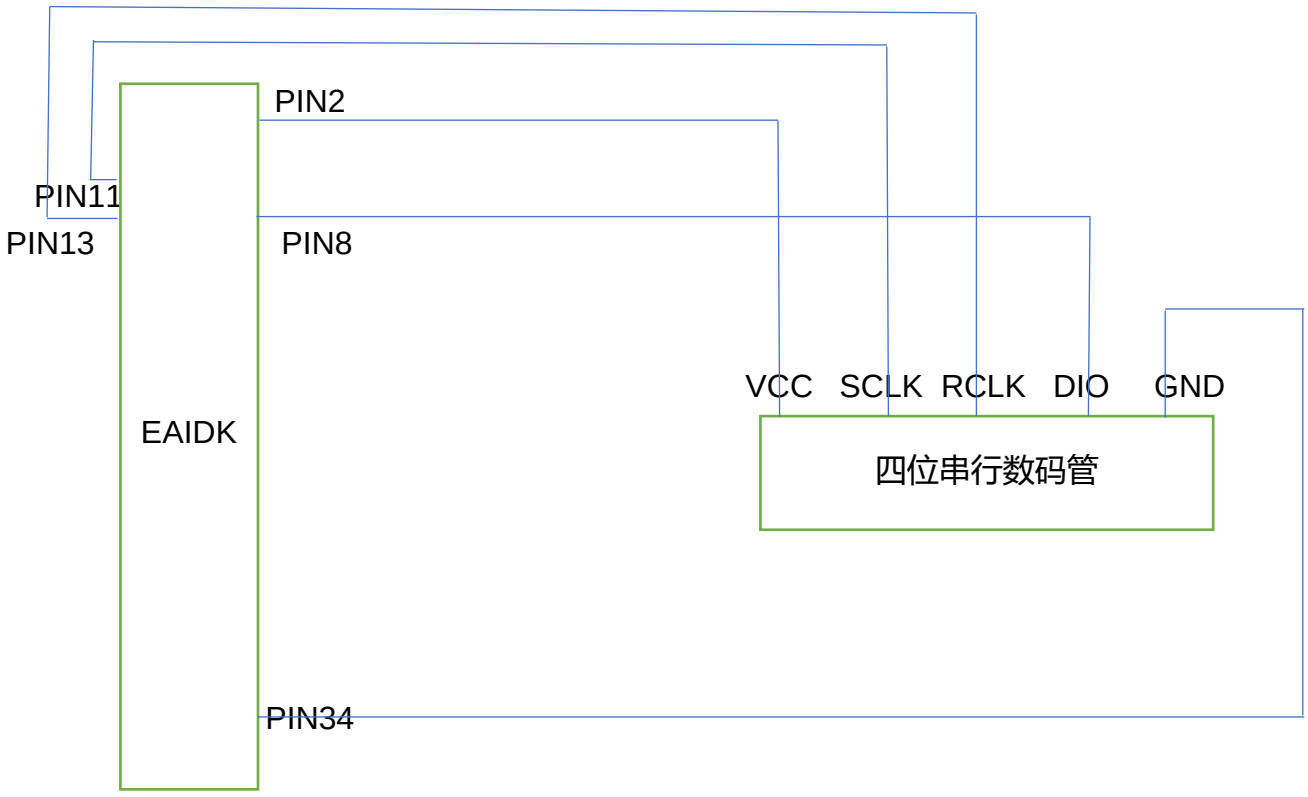

图 5-8 数码管实例电路图

根据 EAIDK 管脚分布图，PIN2 为 5V 高电平，PIN34 为地，PIN11,PIN13,PIN8 分别为 GPIO1_C6，GPIO1_D0，GPIO2_A0。

把 PIN11,PIN13,PIN8 都设置成输出模式。通过改变 PIN11,PIN13,PIN8 的电平来实现数码管的显示。

5.6 C 语言程序说明

5.6.1 文件说明

GPIO 外设应用包括如下源文件：

gpio_common.cpp：gpio 公共接口定义

gpio_common.h：gpio 公共接口声明

gpio_case.cpp：gpio 案例接口定义

gpio_case.h：gpio 案例接口声明

main.cpp：主程序

5.6.2 枚举说明

```
typedef enum //gpio 的 IO 模式
{
    INVALID_DIRECTION, //无效值
    INPUT, //输入模式
    OUTPUT, //输出模式
} DIRECTION;

typedef  enum //gpio 的中断触发模式
{
    INVALID_ EDGE, //无效值
    NONE, //无中断模式
    RISING, //上升沿触发
    FALLING, //下降沿触发
    BOTH, //边沿触发
} EDGE;
```

5.6.3 接口说明

接口名称	int gpio_init (int gpio,DIRECTION direction,EDGE edge,char* value,int value_len)	
接口说明	gpio管脚初始化	
参数	int gpio	gpio管脚号
	DIRECTION direction	IO模式。INPUT表示输入，OUTPUT表示输出，INVALID_DIRECTION表示不设置
	EDGE edge	中断触发模式。NONE表示无中断模式，RISING表示上升沿触发，FALLING表示下降沿触发，BOTH表示边沿触发，INVALID_ EDGE表示不设置
	char* value	gpio管脚对应的控制电平的文件，即/sys/class/gpio/gpio%d/value。作输出用
	int value_len	gpio管脚对应的控制电平的文件长度
返回值	0：成功	-1：失败

接口名称	int set_voltage(FILE * fd,int value)	
接口说明	设置gpio管脚电平值	
参数	int fd	gpio管脚对应的控制电平的文件描述符
	int value	电平值。0代表低电平，1代表高电平
返回值	0：成功	-1：失败

接口名称	int get_voltage(int fd)	
接口说明	获取gpio管脚电平值	
参数	int fd	gpio管脚对应的控制电平的文件句柄
返回值	0：低电平；1：高电平；	-1：失败

接口名称	void gpio_release(int gpio)
接口说明	gpio管脚资源释放

参数	int gpio	gpio管脚号
返回值	无	

接口名称	int led_demo()	
接口说明	点亮LED案例演示	
参数	无	
返回值	0：成功	-1：失败

接口名称	int button_demo()	
接口说明	捕获按钮按下案例演示	
参数	无	
返回值	0：成功	-1：失败

接口名称	int digtron_demo ()	
接口说明	点亮数码管案例演示	
参数	无	
返回值	0：成功	-1：失败

5.6.4 运行程序

1、 获取 gpio 源码;
2、 解压文件并进入/home/openailab/case/gpio 目录，使用./compile.sh 编译本案例
3、 进入/home/openailab/case/gpio/build 目录
4、 按照图 5-3 连接好电路。运行 sudo ./gpio -c 0，可演示点亮 LED 实例。可以观察到发光二极管亮灭交替;
5、 按照图 5-4 连接好电路。运行 sudo ./gpio -c 1，可演示捕获按钮按下实例。当按下按钮是，可以屏幕输出信息;
6、 按照图 5-8 连接好电路。运行 sudo ./gpio -c 2，可演示点亮数码管实例。可以观察到数码管显示 1234。

注意，在选择 GPIO PIN 时，不要选择具有多重功能的 PIN，例如 PIN3 和 PIN5。如果此引脚复用容易导致 IIC 案例运行失败。如真出现此类情况，我们 unexport 该引脚后重启设备，即可恢复正常。同时，在运行 GPIO 案例时可能遇到 device or resource busy 的报错。出现这个的原因是因为我们多次运行了 GPIO 案例，导致同一引脚重复注册，该打印并不影响设备正常功能。若想解决此类打印，我们可以先进入到/sys/class/gpio 目录下 unexport 对应引脚，再运行程序，即可消除报错。

6 同步串行口编程

6.1 同步串行口分布图

EAIDK-610 管脚分布实物图如下图,Pin1 和 Pin2 已经用红圈标出，其他管脚以此类推。

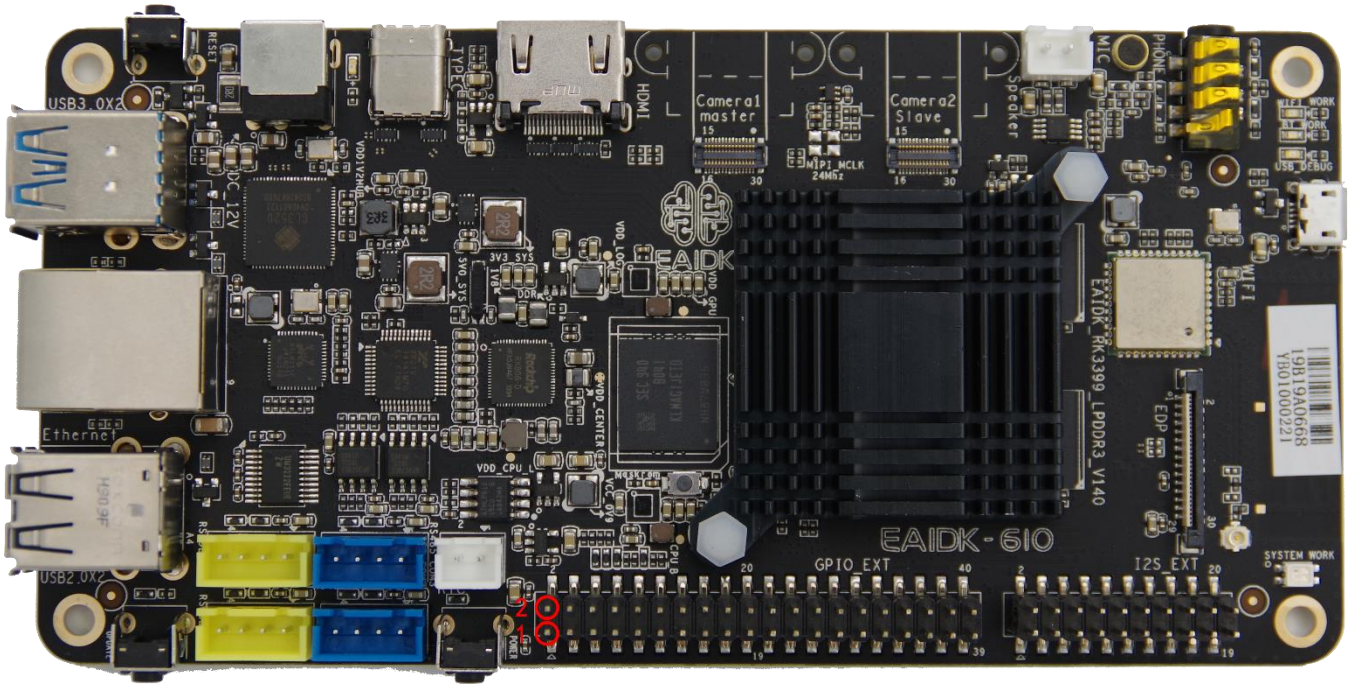

图 6-1 管脚分布图

各个管脚定义如下表

表 6-1 管脚定义

Pin	Name	Pin	Name
1	VCC3V3_SYS	2	VCC5V0_SYS
3	GPIO2_A7/I2C7_SDA	4	VCC5V0_SYS
5	GPIO2_B0/I2C7_SCL	6	GND
7	GPIO4_D0	8	GPIO2_A0/I2C2_SDA
9	GND	10	GPIO2_A1/I2C2_SCL
11	GPIO1_C6	12	
13	GPIO1_D0	14	GND
15	GPIO4_D2	16	GPIO2_B1/I2C6_SDA
17	VCC3V3_SYS	18	GPIO2_B2/I2C6_SCL
19	SPI1_TXD	20	GND

21	SPI1_RXD	22	
23	SPI1_CLK	24	SPI1_CSn0
25	GND	26	ADC_IN3
27	VCC_DC12V	28	VCC_DC12V
29	GPIO2_A2	30	GND
31	GPIO2_A4	32	ADC_IN0
33	GPIO2_A3	34	GND
35	GPIO2_B4	36	GPIO2_A6
37	GPIO4_D5	38	GPIO2_A5
39	GND	40	GPIO1_C7

从上表可以得出。EAIDK-610 支持的 IIC 有三组，分别是｛PIN3（I2C7_SDA）,PIN5（I2C7_SCL）｝，｛PIN8（I2C2_SDA）,PIN10（I2C2_SCL）｝，｛PIN16（I2C6_SDA）,PIN18（I2C6_SCL）｝。支持的 SPI 管脚分别是 PIN19（SPI_TXD）,PIN21（SPI_RXD）,PIN23（SPI_CLK）,PIN24（SPI_CSn0）。

6.2 通过 bme280 获取实时温度实例

bme280 是一种温度湿度气压三合一的传感器，可以提供 IIC 和 spi 两种接口模式。bme280 实物如图所示。

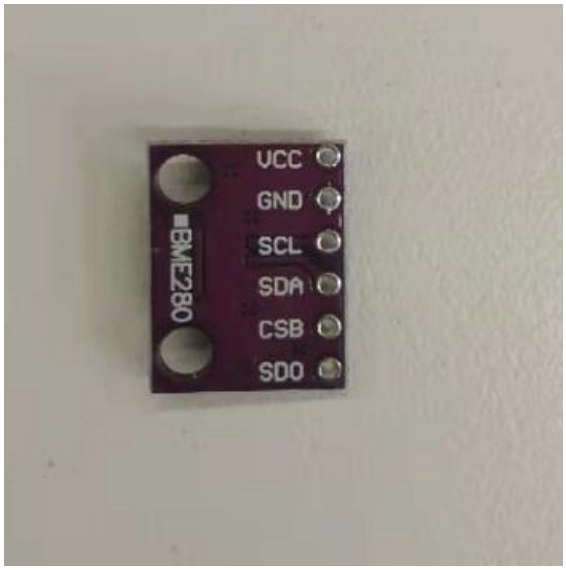

图 6-2 bme280 实物

6.2.1 通过 IIC 获取温度值

将 bme280 的 VCC 接高电平，GND 接低电平，SCL 接 EAIDK-610 的 PIN5（I2C7_SCL），SDA 接 EAIDK-610 的 PIN3（I2C7_SDA）。可以实时监测当前温度。电路连线如下图。

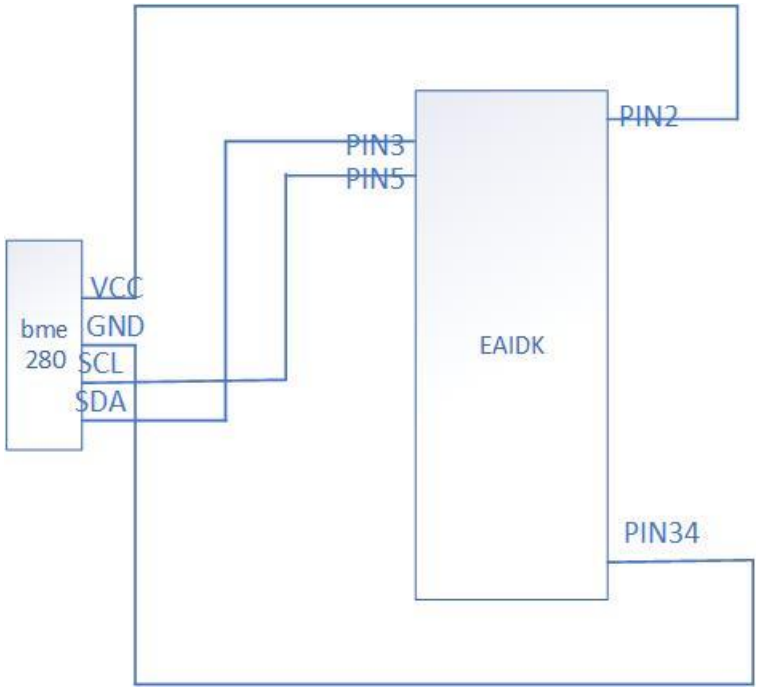

图 6-3 bme28 iic 实例电路图

根据 EAIDK-610 管脚分布图，PIN5 为 I2C7_SCL，PIN3 为 I2C7_SDA。,PIN2 为 5V 高电平，PIN34 为地。

通过 I2C 接口，可以实时读取 bme280 监测到的温度值。

6.2.2 通过 spi 获取温度值

bme280 的 SPI 接口复用了 IIC 接口，复用关系如下：

表 6-2 bme280 接口复用表

IIC 接口	SPI 接口
SCL	SCK
SDA	SDI

将 bme280 的 VCC 接高电平，GND 接低电平，SCK（SCL）接 EAIDK-610 的 PIN23（SPI1_CLK），SDI（SDA）接 EAIDK-610 的 PIN19（SPI1_TXD），CSB 接 EAIDK-610 的 PIN24（SPI1_CSn0），SDO 接 EAIDK-610 的 PIN21（SPI1_RXD）。可以实时监测当前温度。电路连线如下图。

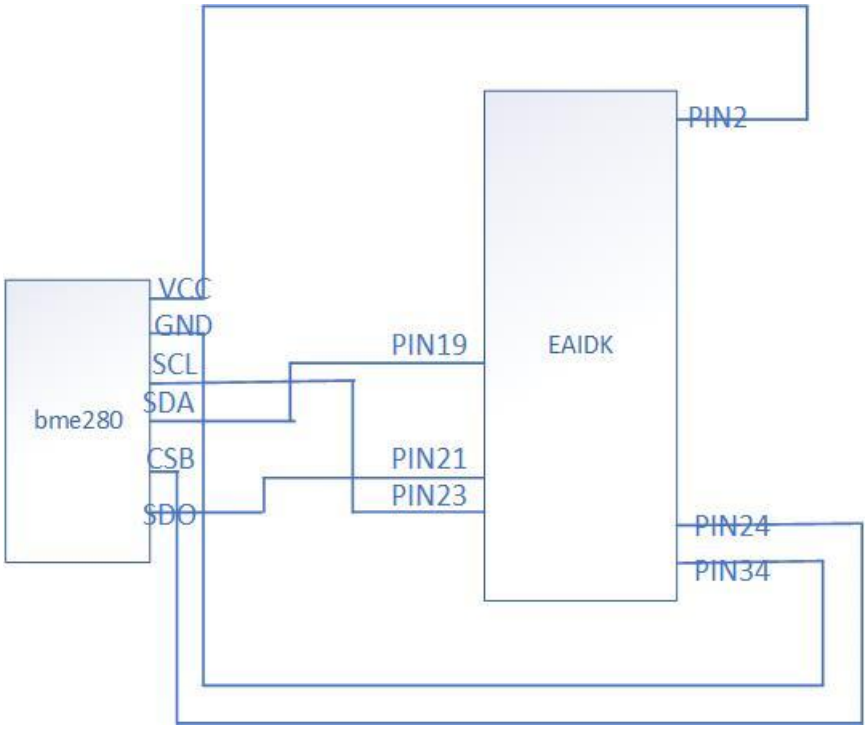

图 6-4 bme280 spi 实例电路图

6.3 程序说明

6.3.1 IIC 程序说明

6.3.1.1 文件说明

IIC 外设应用包括如下源文件：

iic_common.cpp：iic 公共接口定义

iic_common.h：iic 公共接口声明

main.cpp：主程序

CMakeLists.txt:：控制程序编译的文件，会根据此文件生成 Makefile 以及编译。

compile.sh：为编译本案例的脚本

6.3.1.2 接口说明

接口名称	int i2c_open(char* dev)	
接口说明	打开IIC设备文件	
参数	char* dev	iic设备路径
返回值	成功：文件描述符	失败：-1

接口名称	void i2c_write_bytes (int fd, unsigned char addr,unsigned char* data, unsigned short len)	
接口说明	写数据	
参数	int fd	数据写入的文件描述符
	unsigned char addr	写入的寄存器地址
	unsigned char* data	写入的数据内容
	unsigned short len	写入的数据大小
返回值	无	

接口名称	int i2c_read_bytes(int fd, unsigned char addr, unsigned char* buf, unsigned short len)	
接口说明	读数据	
参数	int fd	数据读取的文件描述符
	unsigned char addr	读取寄存器的地址
	unsigned char* buf	读取数据的缓存区
	unsigned short len	最大读取大小
返回值	成功：0	失败：-1

6.3.1.3 运行程序

1、 获取 iic 源码;

2、 进入案例目录/cases/iic, 通过./compile.sh 编译案例

3、 按照图 6-4 连接好电路。进入 build 目录，运行 sudo ./iic，可以实时观察到温度值。

6.3.2 spi 程序说明

6.3.2.1 文件说明

SPI 外设应用包括如下源文件：

spi_common.cpp：spi 公共接口定义

spi_common.h：spi 公共接口声明

main.cpp：主程序

CMakeLists.txt:：控制程序编译的文件，会根据此文件生成 Makefile 以及编译。

compile.sh：为编译本案例的脚本

6.3.2.2 接口说明

接口名称	`void write_addr(int fd,unsigned short addr,unsigned char ch)`	
接口说明	写数据	
参数	`int fd`	数据写入的文件描述符
	`unsigned char addr`	写入的寄存器地址
	`unsigned char ch`	写入的数据内容
返回值	无	

接口名称	`void read_addr(int fd,unsigned short addr,unsigned char* buf, int size)`	
接口说明	读数据	
参数	`int fd`	数据读取的文件描述符

	unsigned short addr	读取寄存器的地址
	char* buf	读取数据的缓存区
	int size	最大读取大小
返回值	成功：0	失败：-1

6.3.2.3 运行程序

1、 获取 spi 源码;

2、 进入案例目录/cases/spi, 通过./compile.sh 编译案例

3、 按照图 6-5 连接好电路。进入 build 目录，运行 sudo ./spi，可实时检测 bme280 的温度值。

7 异步串行口编程

7.1 异步串行口分布图

EAIDK-610 管脚分布实物图如下图,Pin1 和 Pin2 已经用红圈标出，其他管脚以此类推。

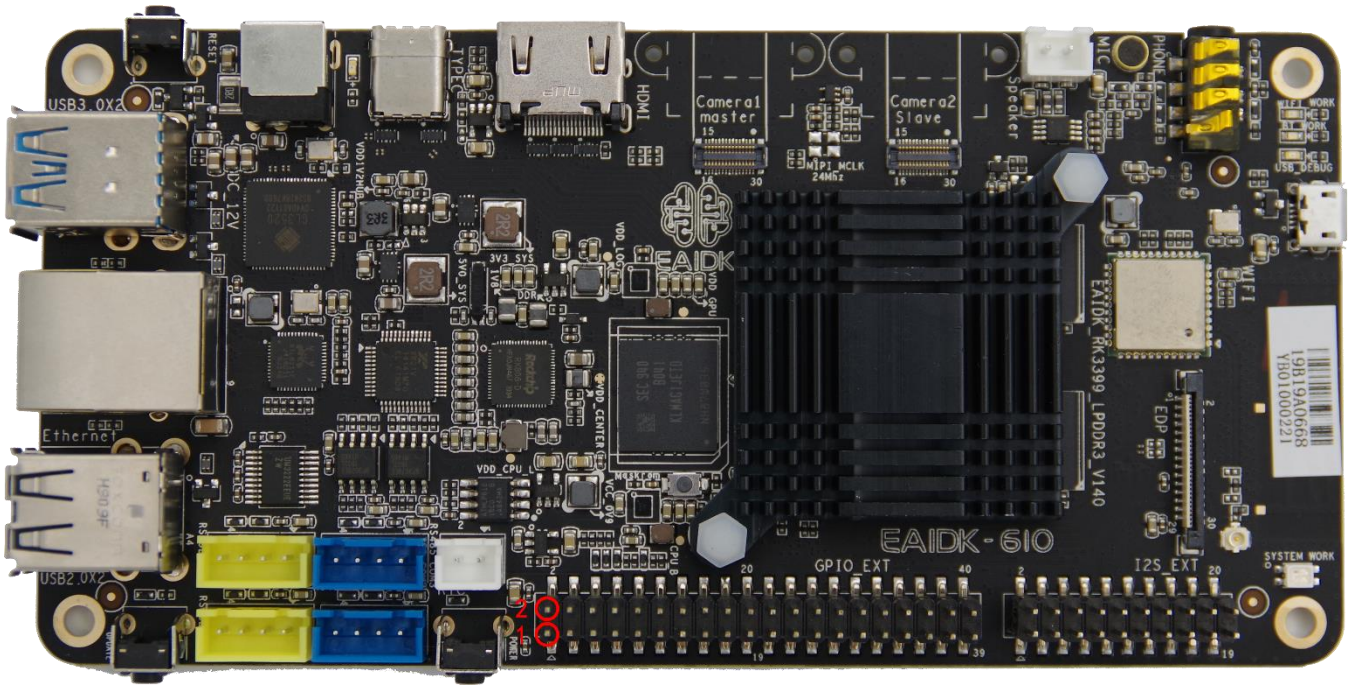

图 7-1 管脚分布图

各个管脚定义如下表

表 7-1 管脚定义

Pin	Name	Pin	Name
1	VCC3V3_SYS	2	VCC5V0_SYS
3	GPIO2_A7/I2C7_SDA	4	VCC5V0_SYS
5	GPIO2_B0/I2C7_SCL	6	GND
7	GPIO4_D0	8	GPIO2_A0/I2C2_SDA
9	GND	10	GPIO2_A1/I2C2_SCL
11	GPIO1_C6	12	
13	GPIO1_D0	14	GND
15	GPIO4_D2	16	GPIO2_B1/I2C6_SDA
17	VCC3V3_SYS	18	GPIO2_B2/I2C6_SCL
19	SPI1_TXD	20	GND

21	SPI1_RXD	22	
23	SPI1_CLK	24	SPI1_CSn0
25	GND	26	ADC_IN3
27	VCC_DC12V	28	VCC_DC12V
29	GPIO2_A2	30	GND
31	GPIO2_A4	32	ADC_IN0
33	GPIO2_A3	34	GND
35	GPIO2_B4	36	GPIO2_A6
37	GPIO4_D5	38	GPIO2_A5
39	GND	40	GPIO1_C7

从上表可以得出。EAIDK-610 支持的 TTL 管脚分别是 PIN8（UART1_TX）,PIN10（UART1_RX）。另外，由于板卡没有 RS485 的接口，故需要用 usb 转 RS485 接口来实现 RS485 通信功能。

7.2 3 通过 TTL 获取 TGS2600 实时烟雾浓度实例

TGS2600 是一种烟雾浓度传感器，可以提供 TTL 接口。由于 EAIDK-610 板卡没有 TTL 接口，故用 USB 转 TTL 设备来实现获取 TGS2600 的烟雾浓度。如下图所示。

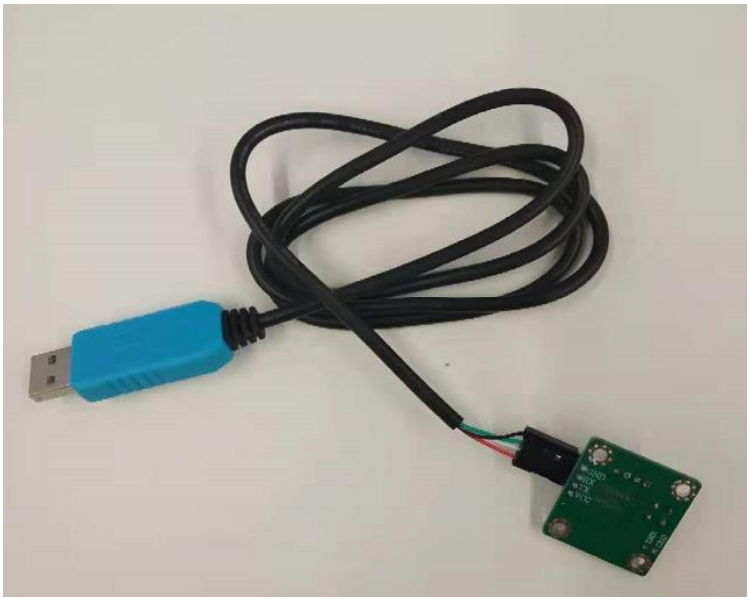

图 7-2 USB 转 TTL TGS2600 实物图

注意接线顺序，黑线接 GND，绿线接 RX，白线接 TX，红线接 VCC。

图 7-3 接线图

将 USB 口接入 EAIDK-610，可以在 EAIDK-610 上看到生成了/dev/ttyUSB0 这个设备（如果/dev/ttyUSB0 被其他 USB 设备占用了，就生成/dev/ttyUSB1，以此类推。具体情况以实际为准）。接下来通过运行程序，可以实现实时检测 TGS2600 的烟雾浓度。

7.3 通过 RS485 获取 SHT20 温度实例

SHT20 是一种温度传感器，可以提供 RS485 接口。我们用 USB 转 RS485 设备来实现获取 SHT20 的温度值。如下图所示。

图 7-4 USB 转 RS485 SHT20 实物图

注意接线顺序，红线接+5v，黑线接 GND，白线接 B，黄线接 A。

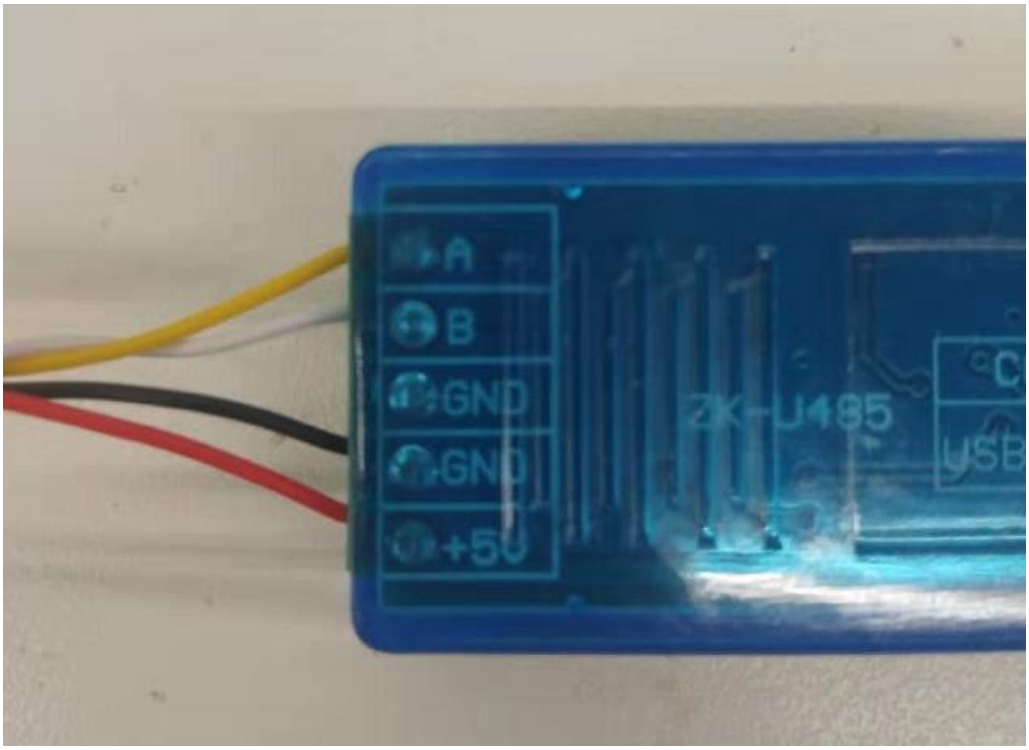

图 7-5 接线图

将 USB 口接入 EAIDK-610，可以在 EAIDK-610 上看到生成了/dev/ttyUSB0 这个设备（如果/dev/ttyUSB0 被其他 USB 设备占用了，就生成/dev/ttyUSB1，以此类推。具体情况以实际为准）。接下来通过运行程序，可以实现实时检测 SHT20 的温度值。

7.4 程序说明

7.4.1 TTL 程序说明

7.4.1.1 文件说明

TTL 外设应用包括如下源文件：

ttl_common.cpp：spi 公共接口定义

ttl_common.h：spi 公共接口声明

main.cpp：主程序

CMakeLists.txt:：控制程序编译的文件，会根据此文件生成 Makefile 以及编译。

compile.sh：为编译本案例的脚本

7.4.1.2 接口说明

接口名称	int open_uart(const char* device_name)	
接口说明	打开RS485设备	
参数	const char* device_name	打开的RS485设备名
返回值	成功：0	失败：-1

接口名称	int set_uart_attr(int fd, int nSpeed, int nBits, char nEvent, int nStop)	
接口说明	设置RS485属性	
参数	int fd	被设置的设备文件描述符
	int nSpeed	波特率
	int nBits	数据位
	char nEvent	奇偶校验位
	int nStop	停止位
返回值	成功：0	失败：-1

接口名称	int read_data(int fd,char* buf,int size)	
接口说明	读数据	
参数	int fd	数据读取的文件描述符
	char* buf	保存读取数据的buf
	int size	最大读取大小
返回值	实际读取数据的大小	

接口名称	int write_data(int fd,char* buf,int size)	
接口说明	写数据	
参数	int fd	数据写入的文件描述符
	char* buf	写入的数据buf

	int size	写入的数据大小
返回值	实际写入数据的大小	

7.4.1.3 运行程序

1、 获取 ttl 源码;

2、 进入案例目录/cases/ttl/case/embedded/ttl, 通过./compile.sh 编译案例

3、 按照图 7-5 连接好电路，进入 build 目录，运行 sudo ./ttl，可实时检测 TGS2600 的烟雾浓度值。

7.4.2 USB-RS485 程序说明

7.4.2.1 文件说明

USB-RS485 外设应用包括如下源文件：

rs485_common.cpp：spi 公共接口定义

rs485_common.h：spi 公共接口声明

main.cpp：主程序

CMakeLists.txt:：控制程序编译的文件，会根据此文件生成 Makefile 以及编译。

compile.sh：为编译本案例的脚本

7.4.2.2 接口说明

接口名称	int open_uart(const char* device_name)	
接口说明	打开RS485设备	
参数	const char* device_name	打开的RS485设备名
返回值	成功：0	失败：-1

接口名称	int set_uart_attr(int fd, int nSpeed, int nBits, char nEvent, int nStop)
接口说明	设置RS485属性

参数	int fd	被设置的设备文件描述符
	int nSpeed	波特率
	int nBits	数据位
	char nEvent	奇偶校验位
	int nStop	停止位
返回值	成功：0	失败：-1

接口名称	int read_data(int fd,char* buf,int size)	
接口说明	读数据	
参数	int fd	数据读取的文件描述符
	char* buf	保存读取数据的buf
	int size	最大读取大小
返回值	实际读取数据的大小	

接口名称	int write_data(int fd,char* buf,int size)	
接口说明	写数据	
参数	int fd	数据写入的文件描述符
	char* buf	写入的数据buf
	int size	写入的数据大小
返回值	实际写入数据的大小	

7.4.2.3 运行程序

1、 获取 rs48 源码;

2、 进入案例目录/cases/rs485, 通过./compile.sh 编译案例

3、 按照图 7-5 连接好电路并把 USB 口插进 EAIDK-610 的 USB 接口，进入 build 目录，运行 sudo ./rs485，可实时检测 SHT20 的温度值。

8 图像采集

8.1 概述

本文档主要介绍如何对 EAIDK（包括 EAIDK-610 和 EAIDK-310）的视频图像进行采集，以及介绍 V4L2，RockchipISP 和 RockchipRGA 的概念及简单操作应用。

8.2 相关工具介绍

V4l2(Video for Linux two)为 linux 下视频设备程序提供了一套接口规范。包括一套数据结构和底层 V4L2 驱动接口。只能在 linux 下使用。它使程序有发现设备和操作设备的能力。它主要是用一系列的回调函数来实现这些功能。例如设置摄像头的频率、帧频、视频压缩格式和图像参数等等。当然也可以用于其他多媒体的开发，如音频等。

在 Linux 下，所有外设都被看成一种特殊的文件，称为“设备文件”，可以像访问普通文件一样对其进行读写。一般来说，采用 V4L2 驱动的摄像头设备文件是/dev/video*。V4L2 支持两种方式来采集图像：内存映射方式(mmap)和直接读取方式(read)。

V4L2 在/usr/include/linux/videodev.h 文件中定义了一些重要的数据结构，在采集图像的过程中，就是通过对这些数据的操作来获得最终的图像数据。Linux 系统 V4L2 的能力可在 Linux 内核编译阶段配置，默认情况下都有此开发接口。V4L2 从 Linux 2.5.x 版本的内核中开始出现。

V4L2 规范中不仅定义了通用 API 元素(Common API Elements)，图像的格式(Image Formats)，输入/输出方法(Input/Output)，还定义了 Linux 内核驱动处理视频信息的一系列接口(Interfaces)，这些接口主要有：

1. 视频采集接口——Video Capture Interface;
2. 视频输出接口—— Video Output Interface;
3. 视频覆盖/预览接口——Video Overlay Interface;
4. 视频输出覆盖接口——Video Output Overlay Interface;
5. 编解码接口——Codec Interface。

在应用程序获取视频数据的流程中，都是通过 ioctl 命令与驱动程序进行交互，常见的 ioctl 命令有：

表 8-1ioctl 命令

ioctl 命令	描述
VIDIOC_QUERYCAP	获取设备支持的操作
VIDIOC_G_FMT	获取设置支持的视频格式
VIDIOC_S_FMT	设置捕获视频的格式
VIDIOC_REQBUFS	向驱动提出申请内存的请求
VIDIOC_QUERYBUF	向驱动查询申请到的内存
VIDIOC_QBUF	将空闲的内存加入可捕获视频的队列
VIDIOC_DQBUF	将已经捕获好视频的内存拉出已捕获视频的队列
VIDIOC_STREAMON	打开视频流
VIDIOC_STREAMOFF	关闭视频流
VIDIOC_QUERYCTRL	查询驱动是否支持该命令
VIDIOC_G_CTRL	获取当前命令值
VIDIOC_S_CTRL	设置新的命令值
VIDIOC_G_TUNER	获取调谐器信息
VIDIOC_S_TUNER	设置调谐器信息
VIDIOC_G_FREQUENCY	获取调谐器频率
VIDIOC_S_FREQUENCY	设置调谐器频率

Rockchip ISP 和 Rockchip RGA 是 Rockchip 开发的针对 RK 硬件的软件包，RGA 是 RK 开发的图形处理单元，支持图像格式转换、图像裁剪、图像缩放等操作。ISP 主要是针对 mipi 摄像设计的一套 3A（自动白平衡,自动曝光,自动对焦）自动优化库，提供 3A 图像性能优化。

8.3 V4L2 视频采集原理

V4L2 支持内存映射方式(mmap)和直接读取方式(read)来采集数据，前者一般用于连续视频数据的采集，后者常用于静态图片数据的采集，本文重点讨论内存映射方式的视频采集。应用程序通过 V4L2 接口采集视频数据分为五个步骤：

1. 打开视频设备文件，进行视频采集的参数初始化，通过 V4L2 接口设置视频图像的采集窗口、采集的点阵大小和格式;
2. 其次，申请若干视频采集的帧缓冲区，并将这些帧缓冲区从内核空间映射到用户空间，便于应用程序读取/处理视频数据;
3. 将申请到的帧缓冲区在视频采集输入队列排队，并启动视频采集;
4. 驱动开始视频数据的采集，应用程序从视频采集输出队列取出帧缓冲区，处理完后，将帧缓冲区重新放入视频采集输入队列，循环往复采集连续的视频数据;
5. 停止视频采集。

采集流程如下：

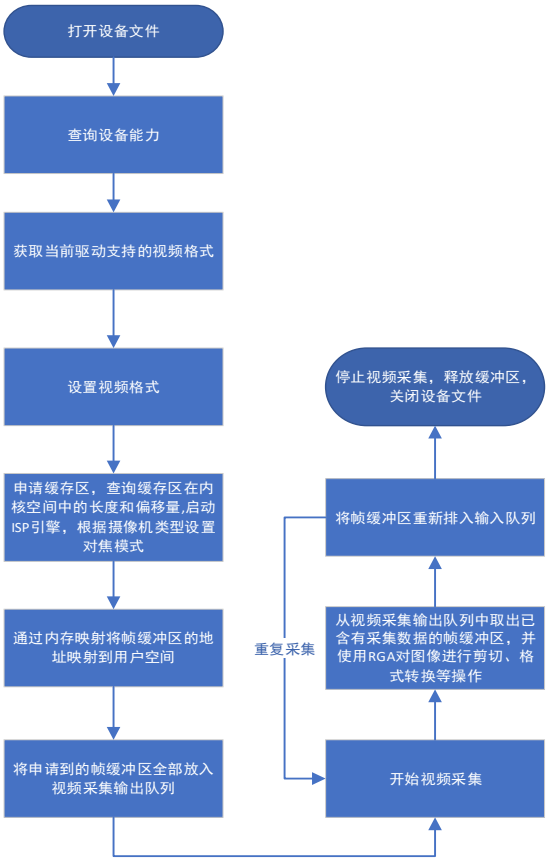

图 8-1 视频采集流程图

8.3.1 打开设备

打开视频设备非常简单，在 V4L2 中，视频设备被看做一个文件。可以使用 open 函数打开这个设备。应用程序能够使用阻塞模式或非阻塞模式打开视频设备，如果使用非阻塞模式调用视频设备，即使尚未捕获到信息，驱动依旧会把缓存（DQBUFF）里的东西返回给应用程序。

示例代码如下：

用非阻塞模式打开摄像头设备，

```
int cameraFd;
cameraFd = open("/dev/video0", O_RDWR | O_NONBLOCK);
```

用阻塞模式打开摄像头设备，上述代码变为：

```
cameraFd = open("/dev/video0", O_RDWR);
```

8.3.2 查询设备能力：VIDIOC_QUERYCAP

函数示例代码如下：

```
struct v4l2_capability capability;
int ret = ioctl(cameraFd, VIDIOC_QUERYCAP, &capability);
```

所有 V4L2 设备都支持 VIDIOC_QUERYCAP。它用于识别与此规范兼容的内核设备，并获取有关驱动程序和硬件功能的信息。ioctl 获取指向由驱动程序填充的 capability 的指针 。当驱动程序与此规范不兼容时，ioctl 将返回 EINVAL 错误代码。

struct v4l2_capability 结构体内容如下：

```
struct v4l2_capability
{
        u8 driver[16];      // 驱动名字
        u8 card[32];        // 设备名字
        u8 bus_info[32];    // 设备在系统中的位置
        u32 version;        // 驱动版本号
        u32 capabilities;   // 设备支持的操作
        u32 reserved[4];    // 保留字段
};
```

其中 capabilities 代表设备支持的操作模式，包含：

表 8-2 视频能力集

capabilities	对应的值	作用
V4L2_CAP_VIDEO_CAPTURE	0x00000001	视频采集设备
V4L2_CAP_VIDEO_OUTPUT	0x00000002	设备输出设备
V4L2_CAP_VIDEO_OVERLAY	0x00000004	可以进行视频叠加
V4L2_CAP_VBI_CAPTURE	0x00000010	原始的 VBI 捕获设备
V4L2_CAP_VBI_OUTPUT	0x00000020	原始的 VBI 输出设备
V4L2_CAP_SLICED_VBI_CAPTURE	0x00000040	限幅的 VBI 捕获设备
V4L2_CAP_SLICED_VBI_OUTPUT	0x00000080	限幅的 VBI 捕获设备
V4L2_CAP_RDS_CAPTURE	0x00000100	RDS 数据捕获
V4L2_CAP_VIDEO_OUTPUT_OVERLAY	0x00000200	可以进行视频输出叠加
V4L2_CAP_HW_FREQ_SEEK	0x00000400	可以进行硬件频率查找
V4L2_CAP_RDS_OUTPUT	0x00000800	RDS 编码器
V4L2_CAP_VIDEO_CAPTURE_MPLANE	0x00001000	支持多平面格式的视频捕获设备
V4L2_CAP_VIDEO_OUTPUT_MPLANE	0x00002000	支持多平面格式的视频输出设备
V4L2_CAP_VIDEO_M2M_MPLANE	0x00004000	支持多平面格式的内存到内存的视频设备
V4L2_CAP_VIDEO_M2M	0x00008000	内存到内存的视频设备
V4L2_CAP_TUNER	0x00010000	调谐器
V4L2_CAP_AUDIO	0x00020000	支持音频
V4L2_CAP_RADIO	0x00040000	无线设备
V4L2_CAP_MODULATOR	0x00080000	调制器
V4L2_CAP_SDR_CAPTURE	0x00100000	SDR 捕获设备
V4L2_CAP_EXT_PIX_FORMAT	0x00200000	支持扩展像素格式
V4L2_CAP_SDR_OUTPUT	0x00400000	SDR 输出设备
V4L2_CAP_META_CAPTURE	0x00800000	元数据捕获设备
V4L2_CAP_READWRITE	0x01000000	读/写的系统调用
V4L2_CAP_ASYNCIO	0x02000000	异步输入/输出
V4L2_CAP_STREAMING	0x04000000	流输入输出控制
V4L2_CAP_TOUCH	0x10000000	触摸设备
V4L2_CAP_DEVICE_CAPS	0x80000000	设置设备能力字段

常见的值有 V4L2_CAP_VIDEO_CAPTURE | V4L2_CAP_STREAMING 表示是一个视频捕获设备并且具有数据流控制模式。

8.3.3 获取当前驱动支持的视频格式：VIDIOC_ENUM_FMT

示例代码如下：

```
struct v4l2_fmtdesc fmtdesc;
fmtdesc.index = 0;
fmtdesc.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ret = ioctl(cameraFd, VIDIOC_ENUM_FMT, &fmtdesc);
```

首先初始化 v4l2_fmtdesc 的 index 和 type 字段，调用 VIDIOC_ENUM_FMT 后，驱动将会填充 v4l2_fmtdesc 剩余部分,否则返回-1。Index 从 0 开始，依次增加，直到返回-1。

struct v4l2_fmtdesc 具体内容如下：

```
struct v4l2_fmtdesc
{
        __u32 index;       // 需要填充，从 0 开始，依次上升。
        enum v4l2_buf_type type;  //应用程序设置的数据流的类型
        __u32 flags;  //视频格式标志，见下表
        __u8 description[32];// 格式说明，以 null 结尾的 ASCII 字符串
        __u32 pixelformat;  //图像格式标识符，由 v4l2_fourcc()计算的四字符代码
        __u32 reserved[4]; //保留字段
};
```

flags 支持的格式，如下表：

表 8-3flags 支持的格式

flags	作用
V4L2_FMT_FLAG_COMPRESSED	一种压缩格式
V4L2_FMT_FLAG_EMULATED	通过软件 libv4l2 模拟的，不是设备的本机格式。推荐使用本机格式

8.3.4 获取和设置视频制式：VIDIOC_G_FMT, VIDIOC_S_FMT

利用 VIDIOC_G_FMT 得到当前设置。因为 Camera 为 CAPTURE 设备，所以需要设置 type 为： V4L2_BUF_TYPE_VIDEO_CAPTURE,然后 Driver 会填充其它内容。然后通过 VIDIOC_S_FMT 操作命令设置视频捕捉格式。v4l2_format 结构体用来设置摄像头的视频制式、帧格式等，在设置这个参数时应先填好 v4l2_format 的各个字段，如 type（传输流类型），fmt.pix.width(宽)，fmt.pix.heigth(高)，fmt.pix.field(采样区域，如隔行采样)，fmt.pix.pixelformat(采样类型，如 YUV4:2:2)。

示例代码如下：

```
struct v4l2_format fmt;
memset(&fmt, 0, sizeof(struct v4l2_format));
fmt.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
ioctl(cameraFd, VIDIOC_G_FMT, &fmt);
memset(&fmt, 0, sizeof(fmt));
fmt.type                = V4L2_BUF_TYPE_VIDEO_CAPTURE;
fmt.fmt.pix.width       = g_display_width;
fmt.fmt.pix.height      = g_display_height;
fmt.fmt.pix.pixelformat = g_fmt;
fmt.fmt.pix.field       = V4L2_FIELD_INTERLACED;
ret = ioctl(cameraFd, VIDIOC_S_FMT, &fmt);
```

struct v4l2_format 结构体所包含的信息如下：

```
struct v4l2_format
{
        enum v4l2_buf_type type; //数据流类型
        union
        {
          struct v4l2_pix_format pix;   // 视频捕获/输出设备的图像格式定义
          struct v4l2_window win; //视频叠加设备的定义
          struct v4l2_vbi_format vbi; //原始 VBI 捕获/输出参数
          struct v4l2_sliced_vbi_format sliced;//限幅 VBI 捕获/输出参数
          __u8 raw_data[200];        //保留字段
        }fmt;
};
```

因为是 Camera， 所以采用像素格式，其结构如下：

```
struct v4l2_pix_format
  {
        __u32 width;    //图像像素的宽
        __u32 height;   // 图像像素的宽
        __u32 pixelformat;  // 像素格式或者压缩类型
```

```
    enum v4l2_field field; //是否逐行扫描，是否隔行扫描
    __u32 bytesperline; //每行的 byte 数
    __u32 sizeimage;  //总共的 byte 数，bytesperline * height
    enum v4l2_colorspace colorspace;
    __u32 priv;
};
```

为了在驱动和应用程序之间交换图像数据，在两边都有标准图像数据是很有必要的。Pixelformat 一般是指 V4L2 中支持的标准图像格式。其中包括 RGB 格式，YUV 格式，压缩格式等，部分举例如下：

1. RGB 格式

定义	码	Byte0								Byte1								Byte2								Byte3							
V4L2_PIX_FMT_RGB332	RGB1	r2	r1	r0	g2	g1	g0	b1	b0	-								-								-							
V4L2_PIX_FMT_RGB444	R444	g3	g2	g1	g0	b3	b2	b1	b0	a3	a2	a1	a0	r3	r2	r1	r0	-								-							
V4L2_PIX_FMT_RGB555	RGBO	g2	g1	g0	b4	b3	b2	b1	b0	a	r4	r3	r2	r1	r0	g4	g3	-								-							
V4L2_PIX_FMT_RGB565	RGBP	g2	g1	g0	b4	b3	b2	b1	b0	r4	r3	r2	r1	r0	g5	g4	g3	-								-							
V4L2_PIX_FMT_RGB555X	RGBQ	a	r4	r3	r2	r1	r0	g4	g3	g2	g1	g0	b4	b3	b2	b1	b0	-								-							
V4L2_PIX_FMT_RGB565X	RGBR	r4	r3	r2	r1	r0	g5	g4	g3	g2	g1	g0	b4	b3	b2	b1	b0	-								-							
V4L2_PIX_FMT_BGR666	BGRH	b5	b4	b3	b2	b1	b0	g5	g4	g3	g2	g1	g0	r5	r4	r3	r2	r1	r0							-							
V4L2_PIX_FMT_BGR24	BGR3	b7	b6	b5	b4	b3	b2	b1	b0	g7	g6	g5	g4	g3	g2	g1	g0	r7	r6	r5	r4	r3	r2	r1	r0	-							
V4L2_PIX_FMT_RGB24	RGB3	r7	r6	r5	r4	r3	r2	r1	r0	g7	g6	g5	g4	g3	g2	g1	g0	b7	b6	b5	b4	b3	b2	b1	b0	-							
V4L2_PIX_FMT_BGR32	RGR4	b7	b6	b5	b4	b3	b2	b1	b0	g7	g6	g5	g4	g3	g2	g1	g0	r7	r6	r5	r4	r3	r2	r1	r0	a7	a6	a5	a4	a3	a2	a1	a0
V4L2_PIX_FMT_RGB32	RGB4	a7	a6	a5	a4	a3	a2	a1	a0	r7	r6	r5	r4	r3	r2	r1	r0	g7	g6	g5	g4	g3	g2	g1	g0	b7	b6	b5	b4	b3	b2	b1	b0

图 8-2 RGB 格式

2. YUV 格式

定义	码	Byte0								Byte1								Byte2								Byte3							
V4L2_PIX_FMT_YUV444	Y444	Cb3	Cb2	Cb1	Cb	Cr3	Cr2	Cr1	Cr0	a3	a2	a1	a0	Y'3	Y'2	Y'1	Y'0	-								-							
V4L2_PIX_FMT_YUV555	YUVO	Cb2	Cb1	Cb0	Cr4	Cr3	Cr2	Cr1	Cr0	a	Y'4	Y'3	Y'2	Y'1	Y'0	Cb4	Cb3	-								-							
V4L2_PIX_FMT_YUV565	YUVP	Cb2	Cb1	Cb0	Cr4	Cr3	Cr2	Cr1	Cr0	Y'4	Y'3	Y'2	Y'1	Y'0	Cb5	Cb4	Cb3	-								-							
V4L2_PIX_FMT_YUV32	YUV4	a7	a6	a5	a4	a3	a2	a1	a0	Y'7	Y'6	Y'5	Y'4	Y'3	Y'2	Y'1	Y'0	Cb7	Cb6	Cb5	Cb4	Cb3	Cb2	Cb1	Cb0	Cr7	Cr6	Cr5	Cr4	Cr3	Cr2	Cr1	Cr0

图 8-3 YUV 格式

3. 压缩格式

表 8-4 压缩格式

定义	码	描述
V4L2_PIX_FMT_H264	'H264'	不带起始码的 H264 视频基本码流
V4L2_PIX_FMT_H264_NO_SC	'AVC1'	带起始码的 H264 视频基本码流
V4L2_PIX_FMT_MVC	'MVC'	H264 MVC 视频流
V4L2_PIX_FMT_H263	'H263'	H263 视频流
V4L2_PIX_MPEG1	'MPG1'	MPEG1 视频流
V4L2_PIX_MPEG2	'MPG2'	MPEG2 视频流
V4L2_PIX_MPEG4	'MPG4'	MPEG4 视频流
V4L2_PIX_FMT_XVID	'XVID'	XVID 视频流
V4L2_PIX_FMT_VC1_ANNEX_G	'VC1G'	VC1, SMPTE 421M Annex G 兼容流

V4L2_PIX_FMT_VC1_ANNEX_L	'VC1L'	VC1, SMPTE 421M Annex L 兼容流
V4L2_PIX_FMT_VP8	'VP8'	VP8 视频流

8.3.5 申请缓存区：VIDIOC_REQBUFS

请求 V4L2 驱动分配视频缓冲区（申请 V4L2 视频驱动分配内存），V4L2 是视频设备的驱动层，位于内核空间，所以通过 VIDIOC_REQBUFS 控制命令字申请的内存位于内核空间，应用程序不能直接访问，需要通过调用 mmap 内存映射函数把内核空间内存映射到用户空间后，应用程序通过访问用户空间地址来访问内核空间。示例代码如下：

```
struct v4l2_requestbuffers req;
memset(&req, 0, sizeof(req));
req.count = 4;  //申请一个拥有四个缓冲帧的缓冲区
req.type = V4L2_BUF_TYPE_VIDEO_CAPTURE;
req.memory = V4L2_MEMORY_MMAP;
ret = ioctl(cameraFd, VIDIOC_REQBUFS, &req)
```

v4l2_requestbuffers 结构中定义了缓存的数量，驱动会据此申请对应数量的视频缓存。多个缓存可以用于建立 FIFO，来提高视频采集的效率。v4l2_requestbuffers 结构体如下：

```
struct v4l2_requestbuffers
{
        __u32 count; //请求的缓存区数量
        enum v4l2_buf_type type;  //缓冲区的格式
        enum v4l2_memory memory; //采集图像的方式
        __u32 reserved[2];  //保留字段
};
```

V4L2 支持使用内存映射，用户指针和 DMABUF 来进行视频流的读写。

1.内存映射：V4L2_MEMORY_MMAP

在应用程序设计中通过调 VIDIOC_QUERYBUF 来获取内核空间的视频缓冲区信息（视频缓冲区的使用状态、在内核空间的偏移地址、缓冲区长度等），然后调用函数 mmap 将申请到的内核空间帧缓冲区的地址映射到用户空间地址，这样就可以直接处理帧缓冲区的数据。在驱动程序处理视频的过程中，定义了两个队列：视频采集输入队列(incoming queues)和视频采集输

出队列(outgoing queues)，前者是等待驱动存放视频数据的队列，后者是驱动程序已经放入了视频数据的队列。应用程序需要将上述帧缓冲区在视频采集输入队列排队(VIDIOC_QBUF)，然后可启动视频采集。启动视频采集后，驱动程序开始采集一帧数据，把采集的数据放入视频采集输入队列的第一个帧缓冲区，一帧数据采集完成，也就是第一个帧缓冲区存满一帧数据后，驱动程序将该帧缓冲区移至视频采集输出队列，等待应用程序从输出队列取出。驱动程序接下来采集下一帧数据，放入第二个帧缓冲区，同样帧缓冲区存满下一帧数据后，被放入视频采集输出队列。应用程序从视频采集输出队列中取出含有视频数据的帧缓冲区，处理帧缓冲区中的视频数据，如存储或压缩。最后，应用程序将处理完数据的帧缓冲区重新放入视频采集输入队列,这样可以循环采集。

2. 用户指针：V4L2_MEMORY_USERPTR

此 I/O 方法结合了高级读写以及内存映射方法。缓存通过应用程序本身进行申请，可以贮存在虚拟或共享内存中，需要交换的只是数据指针。这些指针和元数据在 struct v4l2_buffer（对多平面 API 来说是 struct v4l2_plane）中。若调用 VIDIOC_REQBUFS 且带有相应缓存类型，则驱动必须切换到用户指针 I/O 模式。不需要预先分配缓存，因此他们没有索引，也不能像映射缓存那样通过 VIDIOC_QUERYBUF ioctl 进行查询。

3. DMA 缓存引用：V4L2_MEMORY_DMABUF

DMABUF 框架提供了在多设备见共享缓存的通用方法，支持 DMABUF 的设备驱动可以将一个 DMA 缓存以文件句柄的方式输出到用户空间（输出者规则），以文件句柄的方式从用户空间获取一个 DMA 缓存，这个文件句柄是之前其他或相同的设备所输出的（引入者规则），或都是。此章节描述在 V4L2 中 DMABUF 的引入者规则。V4L2 缓存以 DMABUF 文件句柄方式进行 DMABUF 输出。支持 I/O 流方法的输入和输出设备在通过 VIDIOC_QUERYCAP ioctl 查询时，返回的 struct v4l2_capability 中 capabilities 成员会包含 V4L2_CAP_STREAMING 标签。是否支持通过 DMABUF 文件句柄引入 DMABUF 缓存决定了在调用 VIDIOC_REQBUFS 时内存类型是否要设定为 V4L2_MEMORY_DMABUF。这种 I/O 方法专用于在不同设备间共享 DMA 缓存，这些设备可以是 V4L 设备或是其他视频相关设备（如 DRM）。缓存通过应用程序控制驱动来申请。然后，这些缓存通过使用特殊 API 以文件句柄的方式输出给应用程序，交换的只有文件句柄。句柄和信息存在于 struct v4l2_buffer。在 VIDIOC_REQBUFS ioctl 执行后驱动必须切换到 DMABUF I/O 模式中。

8.3.6 开始/停止采集：VIDIOC_STREAMON，VIDIOC_STREAMOFF

VIDIOC_STREAMON 和 VIDIOC_STREAMOFF 在视频流(内存映射、用户指针或 DMABUF) I/O 期间启动和停止捕获/输出进程。在调用 VIDIOC_STREAMON 之前，视频捕获是禁用的，并且没有输入缓冲区被填充，而且输出设备也是禁用的，并且不产生任何视频信号。直到捕获流和输出流类型都调用了 VIDIOC_STREAMON，设备才会启动。如果 VIDIOC_STREAMON 失败，那么输入缓冲区将保持不变。VIDIOC_STREAMOFF 除了中止或完成正在进行的任何 DMA 之外，还会解锁物理内存中锁定的任何用户指针缓冲区，并从传入和传出队列中删除所有缓冲区。这意味着所有捕获但未脱队列的图像将丢失，同样地，所有排队等待输出但尚未传输的图像也将丢失。

8.4 Rockchip RGA 接口介绍

Rockchip RGA 主要是图像处理单元，支持一些图像相关的操作，具体接口如下：

1. RgaCreate:

接口名称	RgaCreate
接口说明	创建 RGA 实例,返回 RGA 结构指针
参数	无
返回值	RGA 结构指针
接口声明	RockchipRga * RgaCreate(void);
示例	RockchipRga *rga = rgaCreate();

2. RgaDestory:

接口名称	RgaDestroy	
接口说明	销毁 RGA 实例	
参数	RockchipRga *rga	需要销毁的RGA实例
返回值	无	
接口声明	void RgaDestroy(RockchipRga *rga);	
示例	RgaDestory(rga);	

3. initCtx

接口名称	initCtx	
接口说明	清空 RGA 上下文	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
返回值	无	
接口声明	void (*initCtx)(struct _RockchipRga *rga);	
示例	rga->ops->initCtx(rga);	

注意:如果不清空上下文,下次执行 RGA 操作时会沿用之前设置图像参数。

4. setRotate

接口名称	setRotate	
接口说明	设置选择旋转角度	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	RgaRotate rotate	旋转角度 RGA_ROTATE_NONE:不旋转 RGA_ROTATE_90:逆时针旋转 90 度 RGA_ROTATE_180:逆时针旋转 180 度 RGA_ROTATE_270:逆时针旋转 270 度 RGA_ROTATE_VFLIP:垂直翻转 RGA_ROTATE_HFLIP:水平翻转
返回值	无	
接口声明	void (*setRotate)(struct _RockchipRga *rga, RgaRotate rotate);	
示例	rga->ops->setRotate(rga, rotate);	

4. setFillColor:

接口名称	setFillColor	
接口说明	设置色彩填充	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	int color	颜色值 蓝色:0xffff0000 绿色:0xff00ff00 红色:0xff0000ff
返回值	无	
接口声明	void (*setFillColor)(struct _RockchipRga *rga, int color);	
示例	rga->ops->setFillColor(rga, color);	

6. setCrops

接口名称	setSrcCrop，setDstCrop	
接口说明	setSrcCrop：设置源图像剪切窗口 setDstCrop：设置目标图像剪切窗口	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	cropX	原点横坐标
	cropY	原点纵坐标
	cropW	窗口宽度
	cropH	窗口高度
返回值	无	
接口声明	设置源图像剪切窗口函数: void (*setSrcCrop)(struct _RockchipRga *rga, __u32 cropX, __u32 cropY,__u32 cropW, __u32 cropH); 设置目标图像剪切窗口函数: void (*setDstCrop)(struct _RockchipRga *rga, __u32 cropX, __u32 cropY,__u32 cropW, __u32 cropH);	
示例	设置源图像剪切窗口： rga->ops->setSrcCrop(rga, cropX, cropY, cropW, cropH); 设置目标图像剪切窗口： rga->ops->setSrcCrop(rga, cropX, cropY, cropW, cropH);	

7. setFormat

接口名称	setSrcFormat，setDstFormat	
接口说明	setSrcFormat：设置源图像格式 setDstFormat：设置目标图像格式	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	__u32 v4l2Format	v4l2 图像格式,支持的格式见 /usr/include/rockchip/rockchip_rga.h
	__u32 width	图像宽度

	__u32 height	图像高度
返回值	无	
接口声明	设置源图像格式函数: void (*setSrcFormat)(struct _RockchipRga *rga, __u32 v4l2Format,__u32 width, __u32 height); 设置目标图像格式函数: void (*setDstFormat)(struct _RockchipRga *rga, __u32 v4l2Format,__u32 width, __u32 height);	
示例	设置源图像格式：rga->ops->setSrcFormat(rga, v4l2Format, width, height); 设置目标图像格式：rga->ops->setDstFormat(rga, v4l2Format, width, height);	

8. setBuffer

接口名称	setSrcBufferFd，setSrcBufferPtr，setDstBufferFd，setDstBufferPtr	
接口说明	setSrcBufferFd：设置源图像 DMA Buffer函数: setSrcBufferPtr：设置源图像用户空间Buffer函数： setDstBufferFd:设置目标图像DMA Buffer函数: setDstBufferPtr:设置目标图像 用户空间Buffer函数	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	int dmaFd	DMA 内存 fd
	void *ptr	用户空间内存指针
返回值	无	
接口声明	设置源图像 DMA Buffer函数: void (*setSrcBufferFd)(struct _RockchipRga *rga, int dmaFd); 设置源图像用户空间Buffer函数： void (*setSrcBufferPtr)(struct _RockchipRga *rga, void *ptr); 设置目标图像DMA Buffer函数: void (*setDstBufferFd)(struct _RockchipRga *rga, int dmaFd); 设置目标图像 用户空间Buffer函数：	

	void (*setDstBufferPtr)(struct _RockchipRga *rga, void *ptr);
示例	设置源图像 DMA Buffer示例:rga->ops->setSrcBufferFd(rga, fd); 设置源图像用户空间Buffer示例:rga->ops->setSrcBufferPtr(rga, ptr); 设置目标图像DMA Buffer示例:rga->ops->setDstBufferFd(rga, fd); 设置目标图像用户空间Buffer示:rga->ops->setDstBufferPtr(rga, ptr);

9. allBuffer:分配 DMA 内存

接口名称	allocDmaBuffer	
接口说明	分配 DMA 内存	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	v4l2Format	v4l2 图像格式,支持的格式见 /usr/include/rockchip/rockchip_rga.h
	__u32 width, __u32 height	图像宽高
返回值	无	
接口声明	RgaBuffer *(*allocDmaBuffer)(struct _RockchipRga *rga,__u32 v4l2Format, __u32 width, __u32 height);	
示例	RgaBuffer *buf = rga->ops->allocBuffer(rga, v4l2Format, width, height);	

10. 释放 DMA 内存

接口名称	freeDmaBuffer	
接口说明	释放 DMA 内存	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
	RgaBuffer *buf	DMA内存指针
返回值	无	
接口声明	void (*freeDmaBuffer)(struct _RockchipRga *rga, RgaBuffer *buf);	
示例	rga->ops->deinitBuffer(rga, buf);	

11. 执行图像处理操作

接口名称	go	
接口说明	执行图像处理操作	
参数	struct _RockchipRga *rga	需要初始化的RGA实例
返回值	无	
接口声明	int (*go)(struct _RockchipRga *rga);	
示例	rga->ops->go(rga);	

8.5 Rockchip ISP 接口介绍

1. rkisp_start: 创建 ISP 引擎实例

接口名称	rkisp_start	
接口说明	创建 ISP 引擎实例	
参数	void* &engine	ISP 引擎实例地址,rkisp_start 创建引擎实例后,由该参数返回实例地址
	int vidFd	Video stream fd,用户捕获图像时,使用的 video 节点句柄
	const char* ispNode	ISP 节点名,例如”/dev/video1”,该节点与用户捕获图像的节点不同
	const char* tuningFile	IQ xml 文件。例如”/etc/cam_iq_ov9750.xml”,里面记录了摄像头 3A(自动白平衡,自动曝光,自动对焦)操作所需的参数
返回值	成功返回 0 且 engine 不为 NULL	
接口声明	int rkisp_start(void* &engine, int vidFd, const char* ispNode, const char* tuningFile);	
示例	rkisp_start(rkengine, fd, isp_dev, "/etc/cam_iq_imx258.xml");	

注意:

对于内置摄像头而言, n 个摄像头会枚举出 4 个 video 节点。

video(4*(n-1)+2) 节点为用户捕获图像操作的 video 节点,如 video2，video6 。

video(4*(n-1)+1) 节点为 ISP 库进行 3A 操作的 video 节点,如 video1，video5 。

使用结束后必须使用 rkisp_stop 停止 ISP 引擎,释放占用的资源。

必须选择与摄像头相互匹配的 tuningFile ,否则 ISP 引擎无法工作。

目前 ISP 摄像头获取的图像格式 V4L2_PIX_FMT_NV12

2. rkisp_stop:停止并销毁 ISP 引擎实例,释放占用的资源

接口名称	rkisp_stop	
接口说明	停止并销毁 ISP 引擎实例,释放占用的资源	
参数	void* &engine	ISP 引擎实例地址,rkisp_start 创建引擎实例后,由该参数返回实例地址
返回值	成功返回 0	
接口声明	int rkisp_stop(void* &engine)	
示例	rkisp_stop(rkengine);	

3. rkisp_setFocusMode:设置对焦模式

接口名称	rkisp_setFocusMode	
接口说明	设置对焦模式	
参数	void* &engine	ISP 引擎实例地址,rkisp_start 创建引擎实例后,由该参数返回实例地址
	enum HAL_AF_MODE fcMode	对焦模式设置 HAL_AF_MODE_NOT_SET:关闭对焦功能 HAL_AF_MODE_AUTO:自动对焦模式 HAL_AF_MODE_MACRO:特写对焦模式 HAL_AF_MODE_FIXED:固定焦距 HAL_AF_MODE_EDOF:扩展景深 HAL_AF_MODE_CONTINUOUS_VIDEO: 用于

		视频记录的连续自动对焦模式 HAL_AF_MODE_CONTINUOUS_PICTURE，用于拍照的连续自动对焦模式
返回值	成功返回 0	
接口声明	int rkisp_setFocusMode(void* &engine, enum HAL_AF_MODE fcMode);	
示例	rkisp_setFocusMode(rkengine, HAL_AF_MODE_CONTINUOUS_VIDEO);	

注意: OV9750 不支持自动对焦,不允许设置对焦模式(默认关闭即可),否则会出错。

8.6 案例介绍

8.6.1 采集流程

1. USB 摄像头

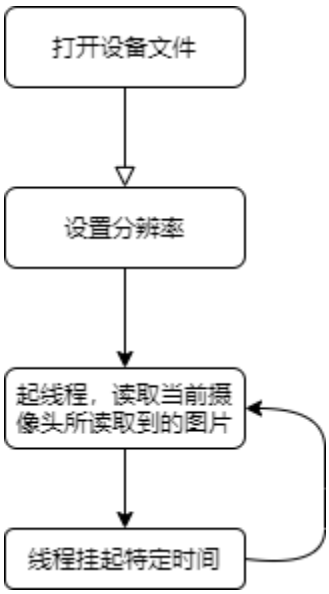

图 8-4 USB 摄像头图像（视频）采集流程

8.6.2 案例文件说明

图像采集案例程序包括如下源文件：

main.cpp：案例主程序

摄像头类说明

1.6.3.2 USB 相机

1. 构造函数

接口名称	UsbCamera	
接口说明	初始化摄像机设备，设置界面宽和高，并且记录摄像机IP	
参数	int cam	摄像机编号
	const __u32 w=DEFAULT_WIDTH	初始化宽度
	__u32 h=DEFAULT_HEIGHT	初始化高度
返回值		
接口声明	UsbCamera(int cam, const __u32 w=DEFAULT_WIDTH, __u32 h=DEFAULT_HEIGHT)	

2. 打开摄像头

接口名称	startCamera	
接口说明	打开摄像头并且根据宽和高的初始值设置分辨率	
参数	无	
返回值	true: 成功	false: 失败
接口声明	bool UsbCamera::startCamera()	

3. 关闭摄像头

接口名称	stopCamera
接口说明	关闭摄像头，删除rtspClient、decoder以及RGA实例
参数	无
返回值	无
接口声明	void IpcCamera::stopCamera()

4. USB 摄像头线程

接口名称	usbThreadFunction
接口说明	循环捕捉当前图片
参数	无
返回值	无
接口声明	void UsbCamera::usbThreadFunction()

5. 设置分辨率

接口名称	setResolution	
接口说明	设置摄像头分辨率	
参数	float w	宽度
	float h	高度
返回值	无	
接口声明	void UsbCamera::setResolution(float w, float h)	

8.6.3 运行程序

1. 进入 home/openailab/cases/camera/目录下，用./compile.sh 命令编译案例，成功编译后会生成一个 build 文件夹。
2. 在 home/openailab/cases/camera/build 目录下，用./camera 运行案例。

注意，本案例仅可以打开 USB 摄像头。如打开摄像头设备路径与程序中默认路径不同，请修改主程序 capture()中的路径。

9 音频采集

9.1 文件说明

音频采集与播放代码包括如下源文件：

- CMakeLists.txt 是控制程序编译的文件，会根据此文件生成 Makefile 以及编译。
- compile.sh 为编译本案例的脚本
- main.cpp 是录音的 C++源码
- README.txt 为运行本案例的指导文档

9.2 接口说明

接口名称	`int snd_pcm_open(snd_pcm_t **pcm, const char *name, snd_pcm_stream_t stream, int mode)`	
接口说明	打开采集pcm音频的设备	
参数	`snd_pcm_t **pcm`	打开的pcm句柄
	`const char *name`	要打开的pcm设备名字
	`snd_pcm_stream_t stream`	SND_PCM_STREAM_PLAYBACK 或 SND_PCM_STREAM_CAPTURE，分别表示播放和录音的 PCM流
	`int mode`	打开pcm句柄时的一些附加参数 SND_PCM_NONBLOCK 非阻塞打开（默认阻塞打开），SND_PCM_ASYNC 异步模式打开

返回值	0 表示打开成功，小于0表示失败

接口名称	snd_pcm_hw_params_alloca(snd_pcm_hw_params_t **params)	
接口说明	申请一段snd_pcm_hw_params_t *params结构体空间。使用该结构体，来配置底层ALSA的相关硬件参数，比如：声道，采样率，位宽，buffer大小等	
参数	snd_pcm_hw_params_t **params	申请的snd_pcm_hw_params_t *params结构体空间
返回值	宏定义，无返回值	

接口名称	int snd_pcm_hw_params_any(snd_pcm_t *pcm, snd_pcm_hw_params_t *params)	
接口说明	获取设备句柄的参数	
参数	snd_pcm_t *pcm	pcm句柄
	snd_pcm_hw_params_t *params	获取的params结构体
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_hw_params_set_access(snd_pcm_t *pcm, snd_pcm_hw_params_t *params, snd_pcm_access_t _access)	
接口说明	设置交错方式	
参数	snd_pcm_t *pcm	设置的pcm句柄
	snd_pcm_hw_params_t *params	设置的params结构体
	snd_pcm_access_t _access	SND_PCM_ACCESS_RW_INTERLEAVED表示交错方式

返回值	0 表示成功，小于0表示失败

接口名称	int snd_pcm_hw_params_set_format(snd_pcm_t *pcm, snd_pcm_hw_params_t *params, snd_pcm_format_t val)	
接口说明	设置采样位宽格式	
参数	snd_pcm_t *pcm	设置的pcm句柄
	snd_pcm_hw_params_t *params	设置的params结构体
	snd_pcm_format_t val	采样位宽格式。SND_PCM_FORMAT_S16_LE表示有符号16位，低位在前
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_hw_params_set_channels (snd_pcm_t *pcm, snd_pcm_hw_params_t *params, unsigned int val)	
接口说明	设置声道数	
参数	snd_pcm_t *pcm	设置的pcm句柄
	snd_pcm_hw_params_t *params	设置的params结构体
	unsigned int val	声道数
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_hw_params_set_rate_near(snd_pcm_t *pcm, snd_pcm_hw_params_t *params, unsigned int *val, int *dir)	
接口说明	设置通采样率	
参数	snd_pcm_t *pcm	设置的pcm句柄
	snd_pcm_hw_params_t *params	设置的params结构体
	unsigned int* val	采样率
	int *dir	默认为0
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_hw_params_set_period_size_near(snd_pcm_t *pcm, snd_pcm_hw_params_t *params, snd_pcm_uframes_t *val, int

	*dir)	
接口说明	设置period区间大小	
参数	snd_pcm_t *pcm	设置的pcm句柄
	snd_pcm_hw_params_t *params	设置的params结构体
	unsigned int* val	period区间大小，以字节为单位
	int *dir	默认为0
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_hw_params_set_buffer_size_near(snd_pcm_t *pcm, snd_pcm_hw_params_t *params, snd_pcm_uframes_t *val)	
接口说明	设置缓存空间大小	
参数	snd_pcm_t *pcm	设置的pcm句柄
	snd_pcm_hw_params_t *params	设置的params结构体
	snd_pcm_uframes_t *val	缓存空间大小。大小为：2*period区间大小*声道数
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_hw_params(snd_pcm_t *pcm, snd_pcm_hw_params_t *params)	
接口说明	设置设备的参数	
参数	snd_pcm_t *pcm	被设置的pcm设备句柄
	snd_pcm_hw_params_t *params	设置的params结构体
返回值	0 表示成功，小于0表示失败	

接口名称	snd_pcm_sframes_t snd_pcm_readi(snd_pcm_t *pcm, void *buffer, snd_pcm_uframes_t size)	
接口说明	读取音频数据	
参数	snd_pcm_t *pcm	pcm设备句柄
	void *buffer	缓冲区，保存读取上来的数据

	snd_pcm_uframes_t size	采样帧数
返回值	读取的数据大小	

接口名称	snd_pcm_sframes_t snd_pcm_writei(snd_pcm_t *pcm, const void *buffer, snd_pcm_uframes_t size)	
接口说明	播放音频数据	
参数	snd_pcm_t *pcm	播放音频的pcm设备句柄
	void *buffer	缓冲区，保存要播放的的数据
	snd_pcm_uframes_t size	采样帧数
返回值	写入的数据大小	

接口名称	int snd_pcm_drain(snd_pcm_t *pcm);	
接口说明	等待缓存区中的数据清空	
参数	snd_pcm_t *pcm	pcm设备句柄
返回值	0 表示成功，小于0表示失败	

接口名称	int snd_pcm_close (snd_pcm_t *pcm);	
接口说明	关闭pcm设备句柄	
参数	snd_pcm_t *pcm	被关闭的pcm设备句柄
返回值	0 表示成功，小于0表示失败	

9.3 运行程序

EAIDK-610 支持使用板载麦克和耳机麦克（需要耳机插入板子上的 TRS3.5mm 接口）录制声音。

1. 点击左下角 LXTernimal（左下角第三个，黑色）打开 Terminal 窗口
2. 进入案例目录/cases/record/case/speech/record, 通过./compile.sh 编译案例
3. 进入 build 目录，运行录音程序：

使用板载 MIC 录音：

```
./record MainMicCapture
```

使用耳机 MIC 录音：

```
./record FreeMicCapture
```

4. 5 秒后录完完成，将在 build 目录下生成 save.pcm 文件

10 音频播放

10.1 实验介绍

最早出现在 Linux 上的音频编程接口是 OSS（Open Sound System），它由一套完整的内核驱动程序模块组成，可以为绝大多数声卡提供统一的编程接口。OSS 出现的历史相对较长，这些内核模块中的一部分（OSS/Free）是与 Linux 内核源码共同免费发布的，另外一些则以二进制的形式由 4Front Technologies 公司提供。由于得到了商业公司的鼎力支持，OSS 已经成为在 Linux 下进行音频编程的事实标准，支持 OSS 的应用程序能够在绝大多数声卡上工作良好。

虽然 OSS 已经非常成熟，但它毕竟是一个没有完全开放源代码的商业产品，ALSA（Advanced Linux Sound Architecture）恰好弥补了这一空白，它是在 Linux 下进行音频编程时另一个可供选择的声卡驱动程序。ALSA 除了像 OSS 那样提供了一组内核驱动程序模块之外，还专门为简化应用程序的编写提供了相应的函数库，与 OSS 提供的基于 ioctl 的原始编程接口相比，ALSA 函数库使用起来要更加方便一些。ALSA 的主要特点有：

- 支持多种声卡设备
- 模块化的内核驱动程序
- 支持 SMP 和多线程
- 提供应用开发函数库
- 兼容 OSS 应用程序

随着 Linux 平台下多媒体应用的逐渐深入，需要用到数字音频的场合必将越来越广泛。虽然数字音频牵涉到的概念非常多，但在 Linux 下进行最基本的音频编程却并不十分复

杂，关键是掌握如何与 OSS 或者 ALSA 这类声卡驱动程序进行交互，以及如何充分利用它们提供的各种功能，熟悉一些最基本的音频编程框架和模式对初学者来讲大有裨益。

10.2 实践说明

播放案例代码放于\home\openailab\cases\play 目录下

其中

- CMakeLists.txt 是控制程序编译的文件，会根据此文件生成 Makefile 以及编译。
- compile.sh 为编译本案例的脚本
- demo.pcm 为案例播放音频
- main.cpp 是播放案例的 C++源码
- README.txt 为运行本案例的指导文档

10.3 程序运行

1. 进入/home/openailab/cases/play 目录下，用./compile.sh 编译案例
2. 运行前，将音箱通过 3.5mm 耳机口连接至 610
3. 进入/home/openailab/cases/play/build 目录下，用./play demo.pcm 运行案例