第3章 智慧照明平台终端软件设计

CHAPTER 3

智慧照明平台终端主要指本系统的实验箱，包括传感器板、中控制器即中控板和总控制器即大板，本章主要介绍传感器信息采集、本地控制和远程控制的程序设计方法和代码，包含网络传输、总控板实验程序设计、局域网网关程序设计等各模块的程序思路及主要代码的讲解。

3.1 开发软件介绍及使用

本实验箱采用 Keil uVision5 进行开发，即 Keil 的第 5 版。

Keil 公司是一家业界领先的微控制器(MCU)软件开发工具的独立供应商。Keil 公司由两家私人公司联合运营，分别是德国慕尼黑的 Keil Elektronik GmbH 和美国德克萨斯的 Keil Software Inc。Keil 公司制造和销售种类广泛的开发工具，包括 ANSI C 编译器、宏汇编程序、调试器、连接器、库管理器、固件和实时操作系统核心(Real-Time Kernel)。有超过 10 万名 MCU 开发人员在使用这种得到业界认可的解决方案。Keil C51 编译器自 1988 年引入市场以来已成为事实上的行业标准，并支持超过 500 种 8051 变种。MDK 即 RealView MDK 或 MDK-ARM(Microcontroller Development kit)，是 ARM 公司收购 Keil 公司以后，基于 uVision 界面推出的针对 ARM7、ARM9、Cortex-M0、Cortex-M1、Cortex-M2、Cortex-M3、Cortex-M4 等 ARM 处理器的嵌入式软件开发工具。MDK-ARM 集成了业内最领先的技术，包括 uVision5 集成开发环境与 RealView 编译器 RVCT。它支持 ARM7、ARM9 和最新的 Cortex-M3/M1/M0 核处理器，可自动配置启动代码，集成 Flash 烧写模块，具有强大的设备模拟、性能分析等功能，与 ARM 之前的工具包 ADS 等相比，RealView 编译器的最新版本可将性能改善超过 20%。

Keil 公司开发的 ARM 开发工具 MDK，是用来开发基于 ARM 核的系列 MCU 的嵌入式应用程序。它适合不同层次的开发者使用，包括专业的应用程序开发工程师和嵌入式软件开发的入门者。MDK 包含了工业标准的 Keil C 编译器、宏汇编器、调试器、实时内核等组件，支持所有基于 ARM 的设备，能帮助工程师按照计划完成项目。

3.1.1 Keil 5 的安装

1. 安装

双击如图 3-1 所示图标进行安装，进入如图 3-2 所示的安装界面，单击 Next。

图 3-1　Keil 5 安装包

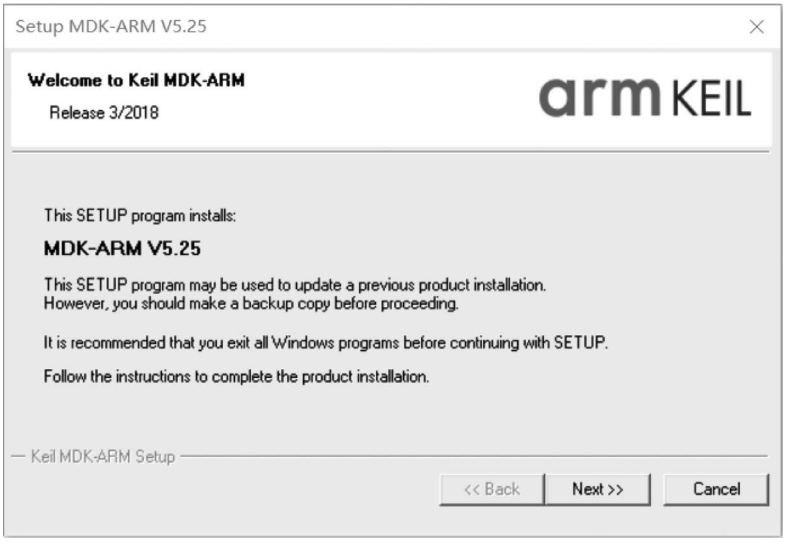

图 3-2　安装界面

在安装过程中出现以下界面。

(1) 选中同意软件使用条约,单击 Next(下一步);

(2) 选择安装路径(建议选择默认 C 盘路径),单击 Next(下一步);

(3) 填写用户名(First name)与邮箱(E-Mail)(任意填写),单击 Next(下一步);

(4) 正在安装,等待安装进度条完成;

(5) 去掉“Show Release Notes”对勾,安装完成,单击 Finish(完成)。

2. 添加器件库芯片包

(1) 下载芯片包。

官方下载界面如图 3-3 所示。找到 GD32F30x 系列的芯片包,如图 3-4 所示。将芯片包下载到 Keil 5 的安装根目录下。

(2) 安装芯片包。

双击芯片包. pack 文件,如图 3-5 所示。

(3) 进入添加器件库安装包界面,此步骤自动搜寻 MDK5 软件安装路径。如图 3-6 所示,完成本步骤后单击 Next。然后进入添加器件库安装包进度条,完成这一步后单击 Next。进入安装,等待完成后单击 Finish,如图 3-7 所示,则器件库安装完成。

(4) 打开 Keil 5,新建工程,芯片包已经安装好了,如图 3-8 所示。

3. 激活 MDK

(1) 右击桌面上的 Keil 图标,如图 3-9 所示。在弹出的选项卡中选择以管理员身份运行。

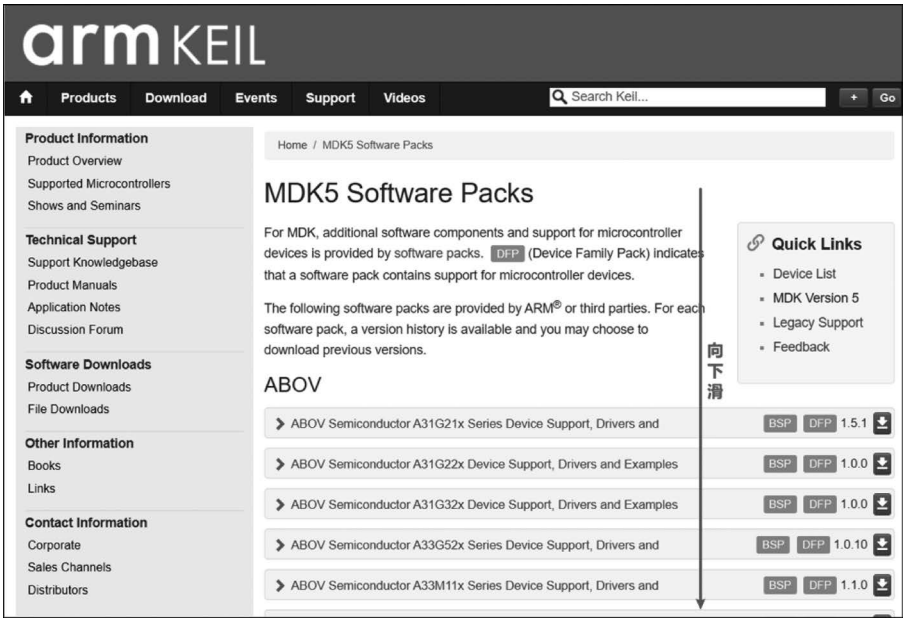

图 3-3　官方下载界面

图 3-4　GD32F30x 系列的芯片包

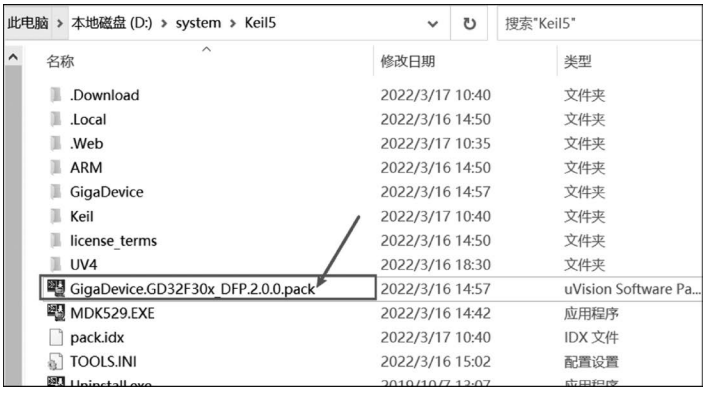

图 3-5　双击 GD32F30x 系列的芯片包

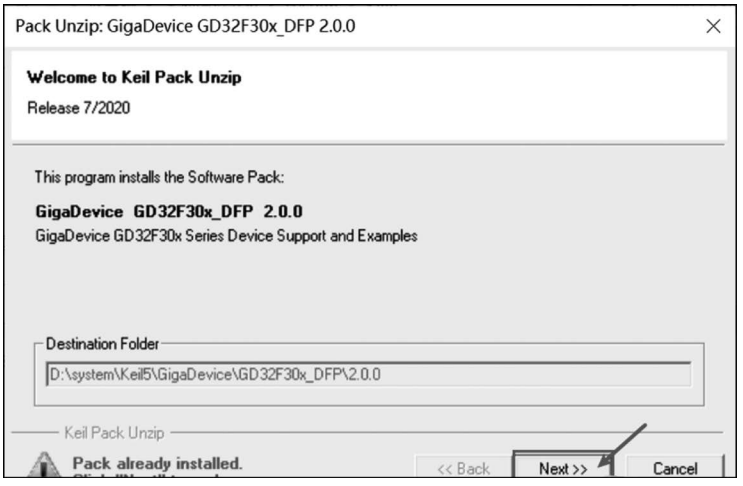

图 3-6　自动搜寻软件安装路径

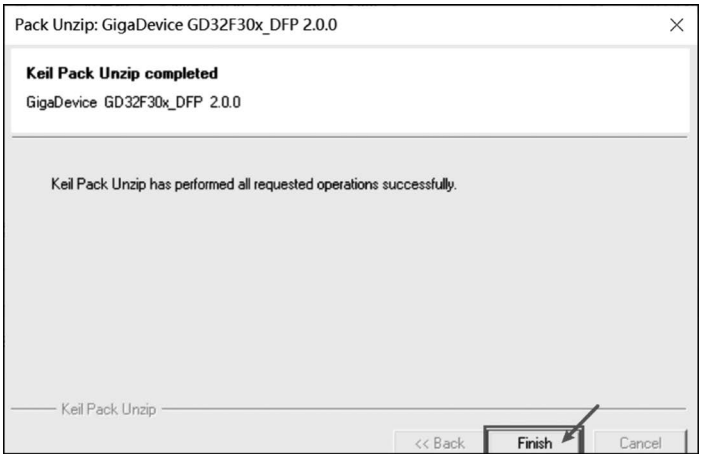

图 3-7　等待 GD32F30x 芯片包安装完成

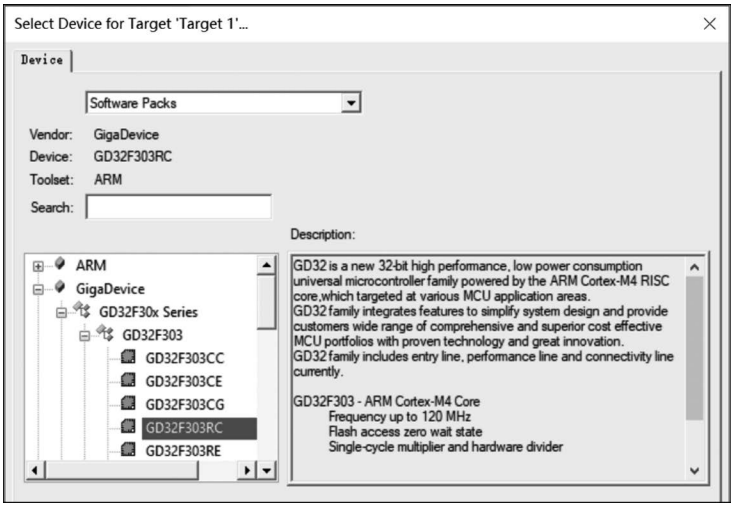

图 3-8　GD32F30x 芯片包安装完成

(2) 进入软件,选择 File→License Management,如图 3-10 所示。

图 3-9 右击 Keil 图标

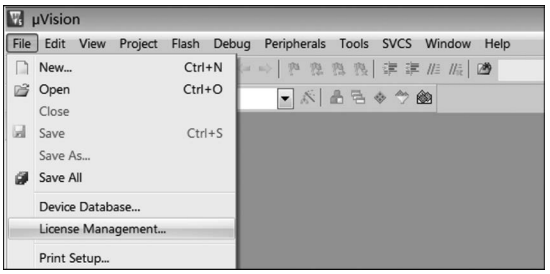

图 3-10 选择 License Management

(3) 复制 ID 号,如图 3-11 所示。

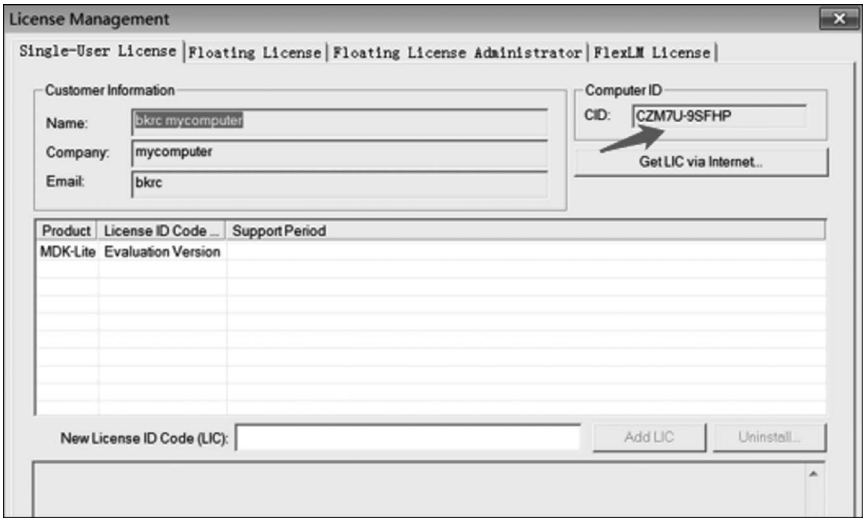

图 3-11 复制 ID 号

(4) 如需破解,需自行找到注册机,双击打开注册机软件,如图 3-12 所示。

(5) 粘贴 ID 号,选择 ARM,单击 Generate 按钮,获得注册号并复制,如图 3-13 所示。

图 3-12 注册机

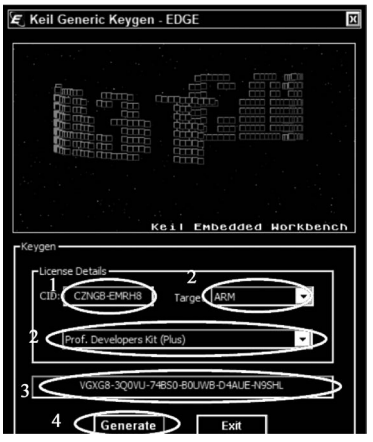

图 3-13 得到注册号

(6) 粘贴注册号,单击添加进行注册(出现如图 3-14 所示界面,即代表注册成功)

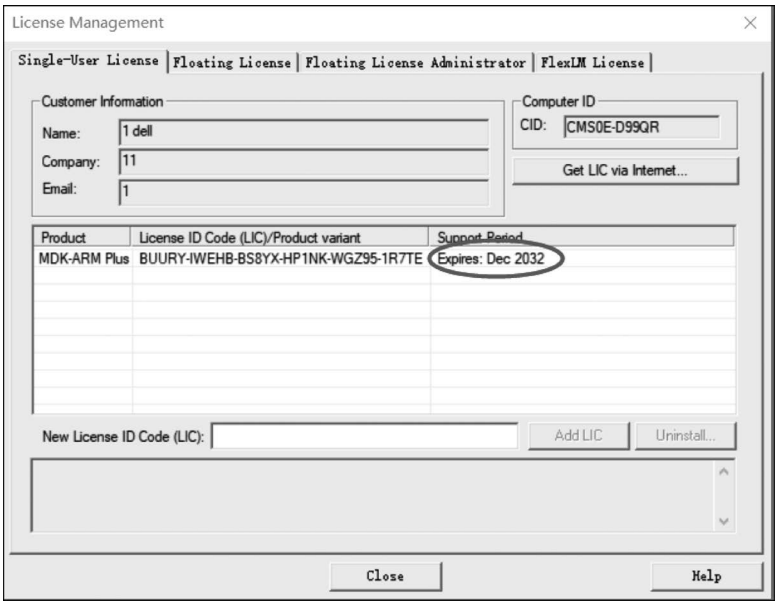

图 3-14　注册成功

至此,MDK5 安装完成。

3.1.2　下载器的安装与配置

GD32 下载程序需要用仿真器进行下载,可以用 ST-LINK 也可用 J-LINK,本实验平台采用 J-LINK 下载器。

1. J-LINK 下载器驱动安装

首先从官网下载最新的 J-LINK 驱动软件,J-Link ARM software and documentation pack,内含 USB driver、J-Mem、J-Link. exe and DLL for ARM、J-Flash and J-Link RDI。注意:SEGGER 公司升级比较频繁,请密切留意 SEGGER 公司网站,下载最新驱动,以支持更多器件。安装驱动很简单,只要将下载的 ZIP 包解压,然后直接安装即可。默认安装是一路单击“Yes”即可,如图 3-15 所示。

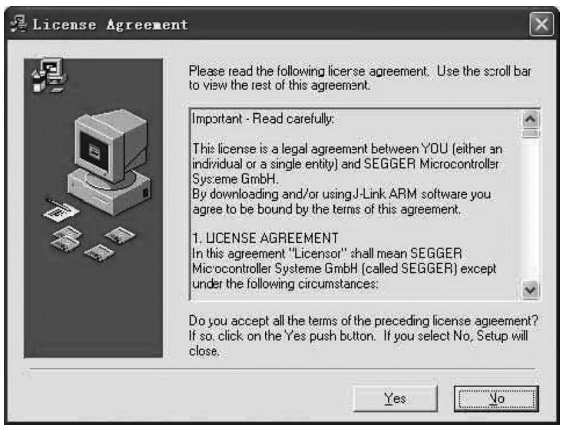

图 3-15　J-LINK 驱动安装

安装完成后，插入 J-LINK 硬件，系统会提示发现新硬件。一般情况下会自动安装驱动，如果没有自动安装，请选择手动指定驱动程序位置（安装目录），然后将驱动程序位置指向 J-LINK 驱动软件安装目录下的 Driver 文件夹，驱动程序就在该文件夹下。安装完成后桌面出现两个快捷图标，J-Link ARM 可以用来进行设置和测试。

2. Keil 5 开发环境对 J-LINK 的配置方法

本实验平台的开发环境是 Keil 5，要想利用 J-LINK 向电路板中的 MCU 下载 Keil 5 中编写的程序，就需要配置 J-LINK。下面是 Keil 5 的 J-LINK 配置流程。

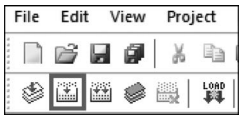

图 3-16 编译项目

（1）首先用 J-LINK 连接电路板，打开一个项目并编译项目，如图 3-16 所示。

（2）单击 Option for target→Debug，选择 J-LINK/J-TRACE Cortex，如图 3-17 所示，勾选 Run to main()，如图 3-18 所示。

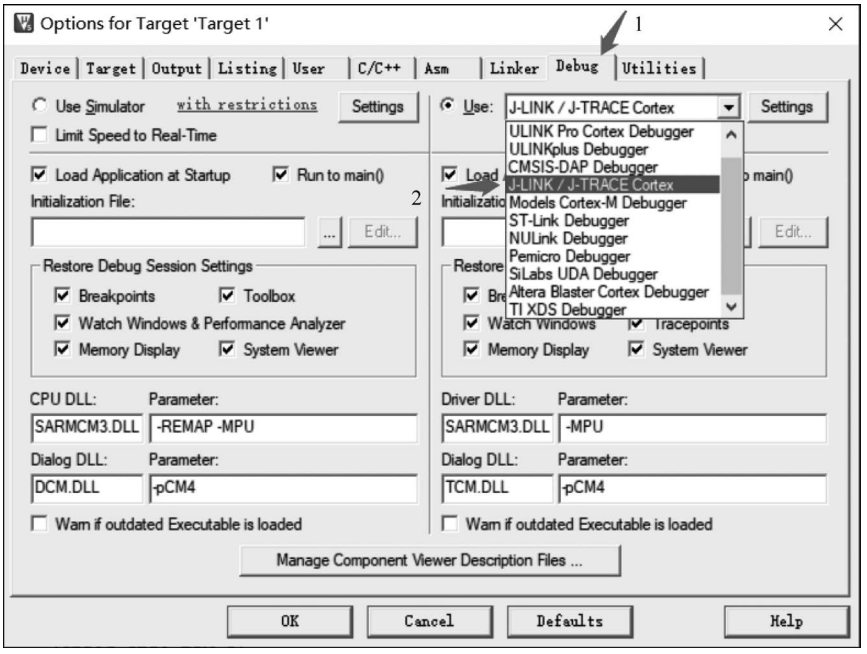

图 3-17 J-LINK 目标资源配置

（3）单击 Setting，会弹出另一个页面，在 Debug 菜单中选择 Port：SW 和 Max：10MHz。

这里要注意 Prot：SW 与 JTAG 模式。使用 JTAG 模式这个可以不做改动，但使用 SW 模式的情况下必须将 Port 选择为 SW，如果正确，在链接成功后会在 SWDIO 中显示图 3-19 所示内容。

（4）在 Flash Download 菜单中勾选 Program、Verify、Reset and Run，然后单击 Add，添加芯片的支持包，如图 3-20 所示。

（5）选择型号和 Flash Size 相符合的选项，单击 Add 后单击确定，如图 3-21 所示。

（6）在 Option for target→Utilities 中勾选 Use Debug Driver 和 Update Target before Debugging，实现芯片支持包的添加，如图 3-22 所示。

（7）J-LINK 配置完成，可以单击下载程序了，如图 3-23 所示。

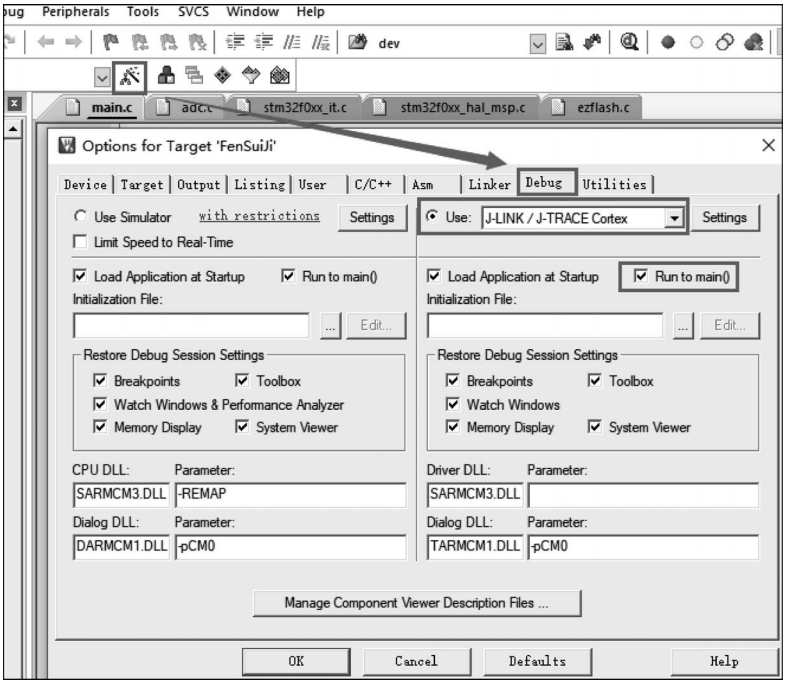

图 3-18　选择 J-LINK/J-TRACE Cortex

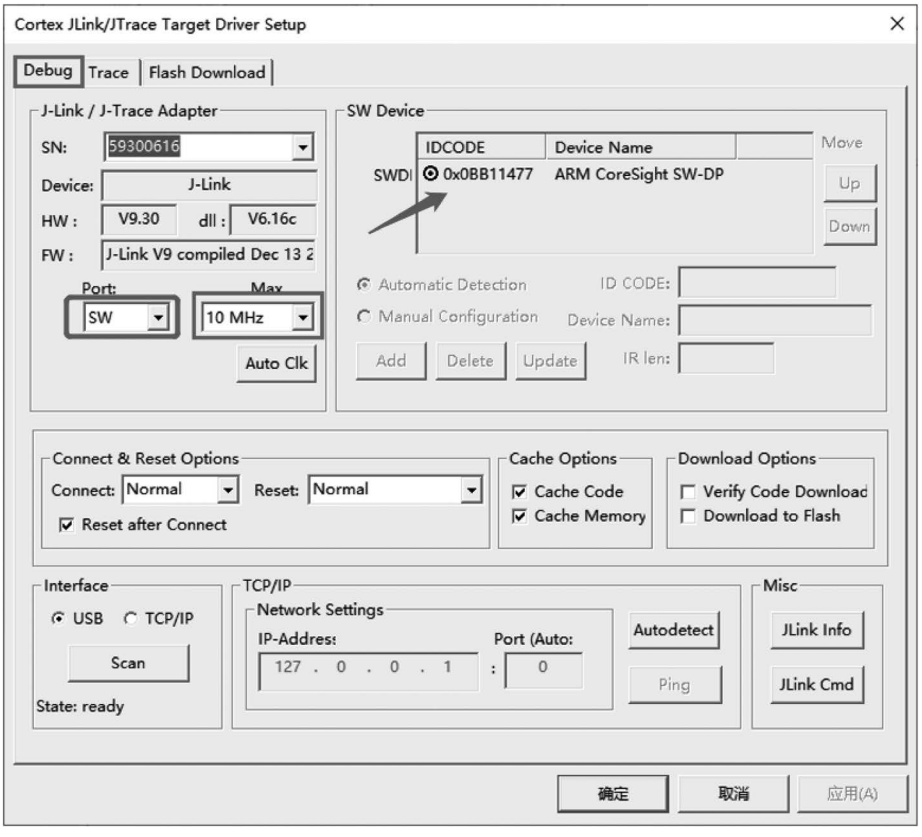

图 3-19　选择 Port:SW 和 Max:10MHz

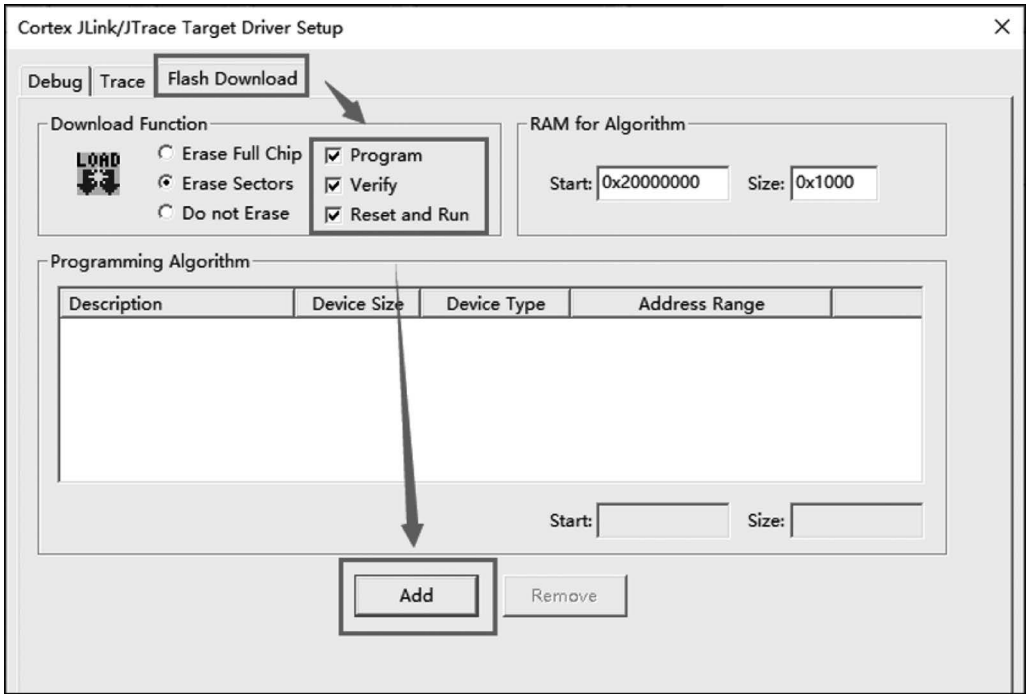

图 3-20　选择 Program、Verify、Reset and Run

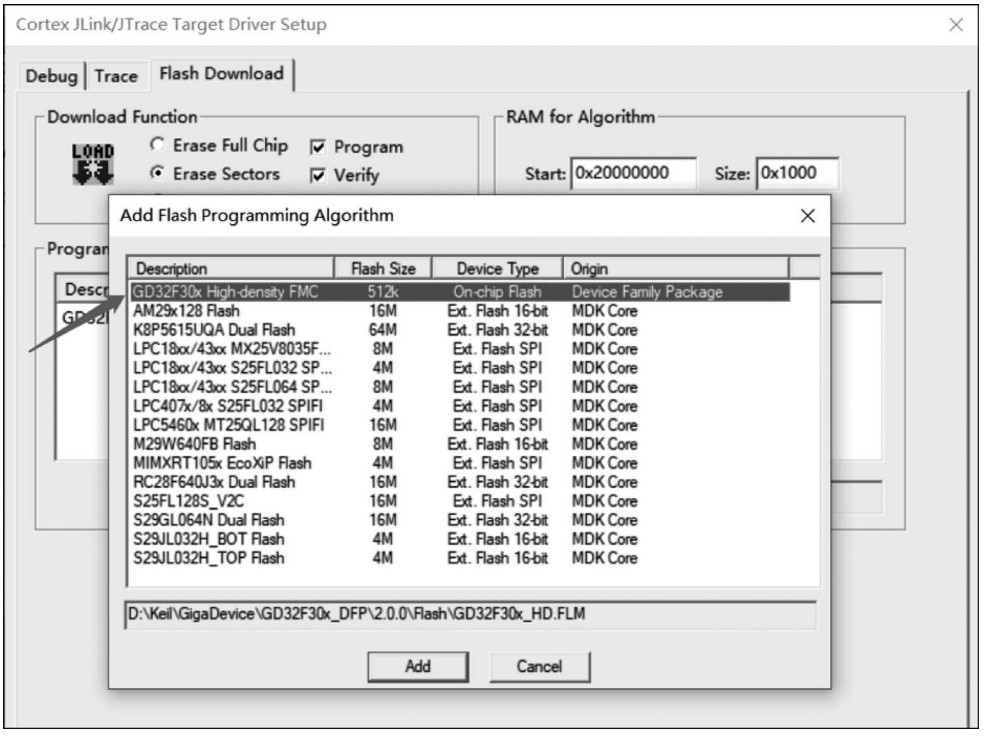

图 3-21　添加芯片的支持包

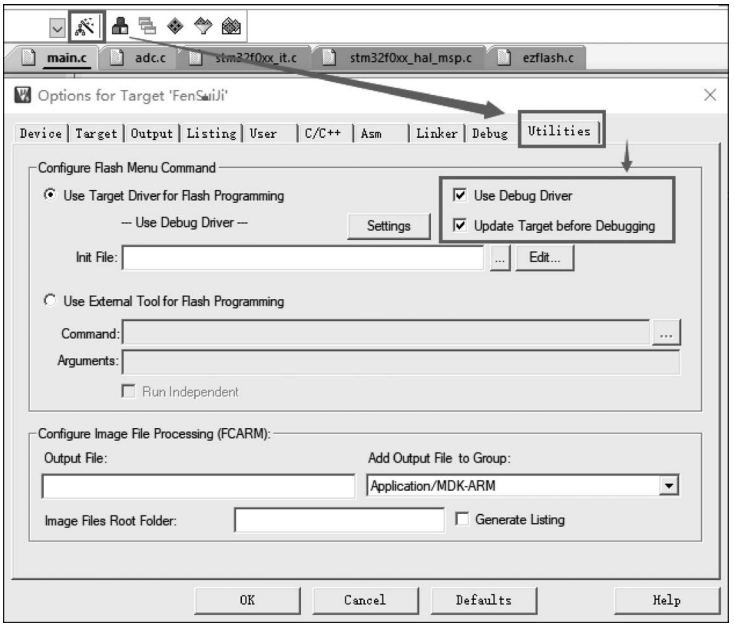

图 3-22　勾选 Use Debug Driver 和 Update Target before Debugging

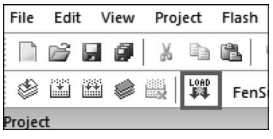

图 3-23　下载程序

3.1.3　Keil 5 的使用

1. 打开软件程序工程

在硬件连接没问题之后，打开任意一个工程，双击 Project 文件夹下的 template 可执行程序(如图 3-24 所示)，如图 3-25 所示就表示成功打开了一个工程。

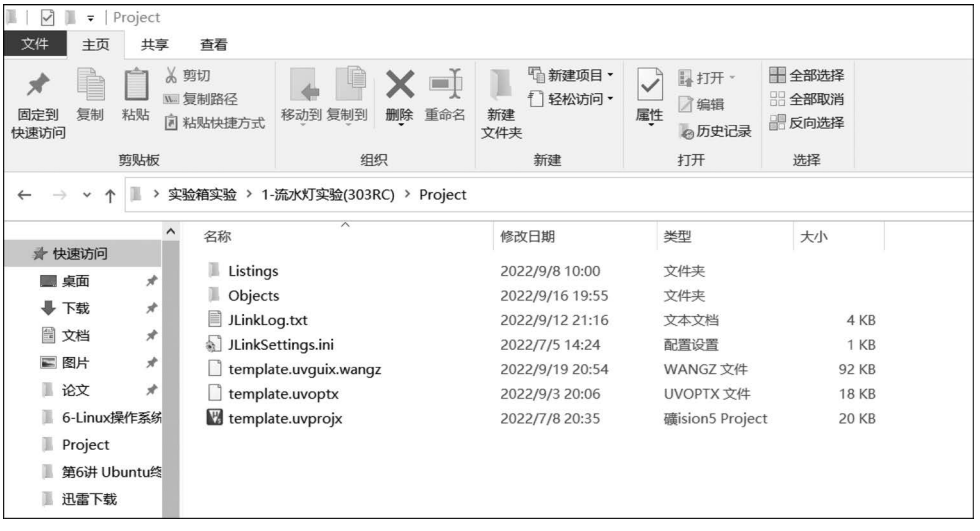

图 3-24　双击可执行程序

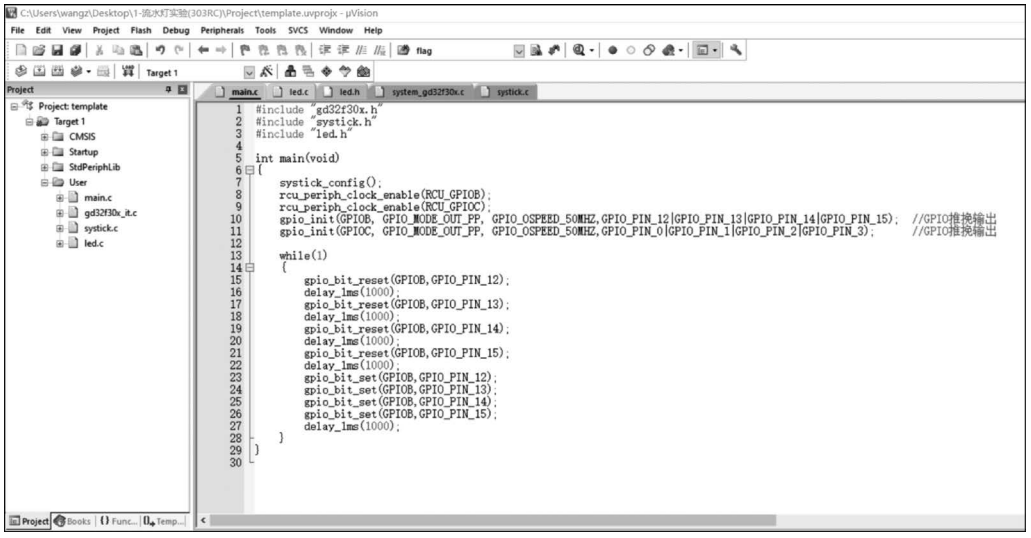

图 3-25　成功打开工程

2. 工程编辑、编译和调试下载

每个工程示例程序都是配置好的，采用 J-LINK 下载程序调试，查看是否配置好下载工具，可以按照以下步骤。

1）编辑

示例程序已经都编辑好，不需要再编辑代码。如果需要自己修改，需按图 3-26 所示在 Project 中单击需要编辑的文件，然后在主框图中编写和修改，最后单击保存按钮。

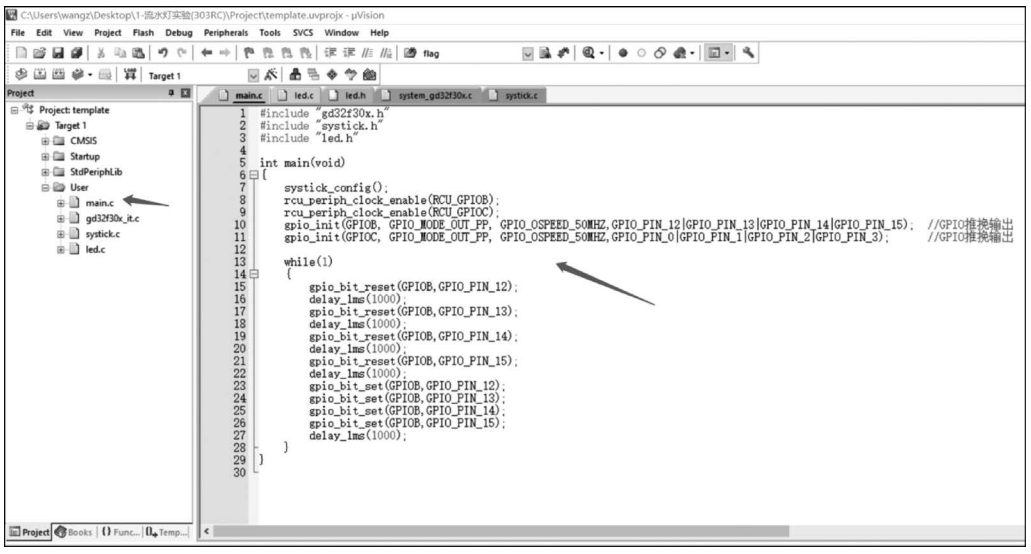

图 3-26　在箭头处编辑代码

2）编译

如图 3-27 所示，单击编译按钮，其中，箭头 1 所指是部分编译按钮，箭头 2 所指是全部编译按钮，如果箭头 3 所示 Error 值为 0，说明编译通过，可以进入下载步骤，否则还需把错误的地方改正才能进入下载步骤。

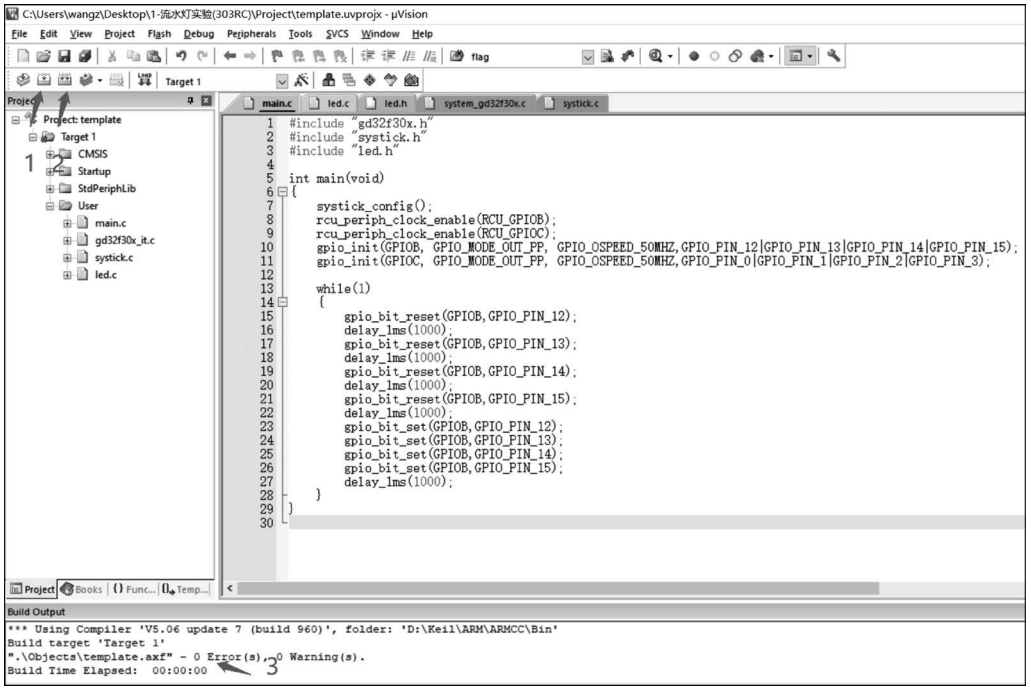

图 3-27　编译代码

3）下载

如图 3-28 所示，单击（箭头 1 所指）Load，Build Output 中会提示 Programming Done，Verify OK，说明下载成功。

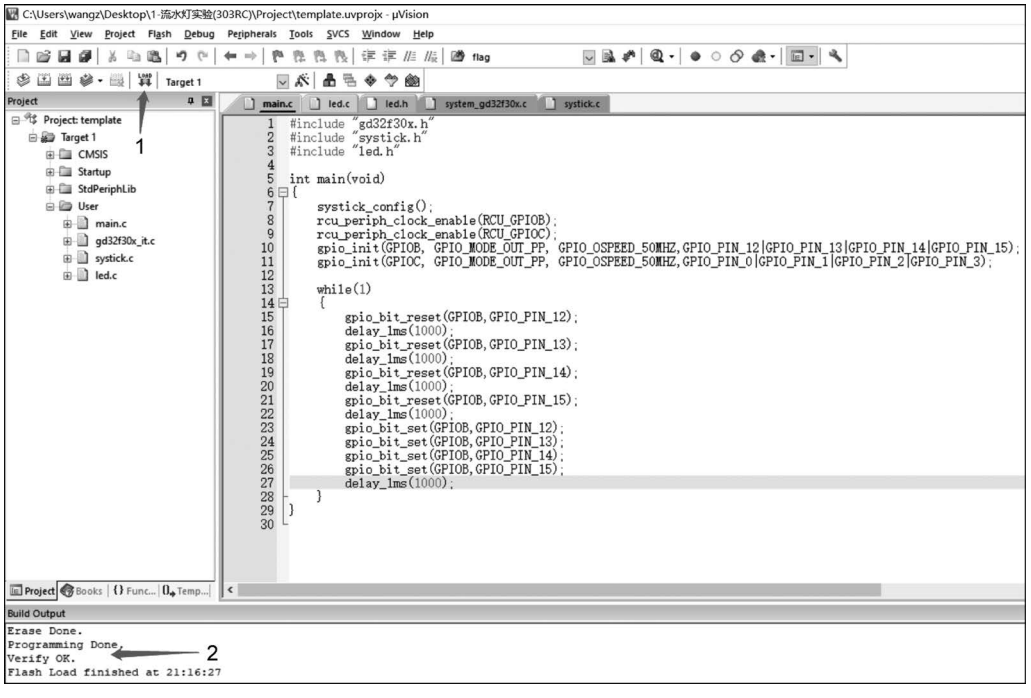

图 3-28　下载程序

4）调试运行

如图 3-29 所示，先按箭头 1 所指按钮（Start/Stop Debug Session），光标就会跳到 main 函数，再按箭头 2 所指按钮（全速运行），程序就完全跑起来了，比如流水灯例程可以在主板上看到 LED 灯每秒依次点亮，箭头 3 所指按钮是当全速运行起来之后，按了 Stop 按钮，单步调试时才用得上。

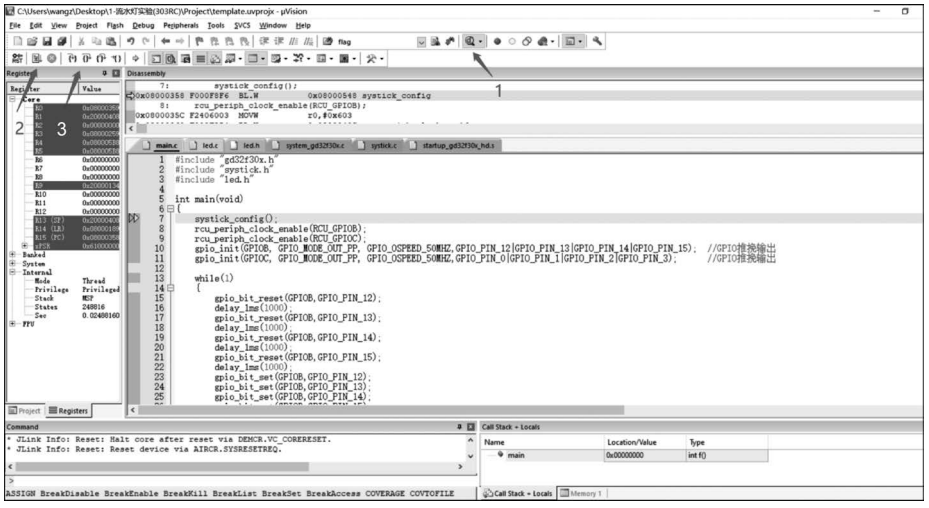

图 3-29　调试运行

3.1.4　Keil 5 的常见问题排查

1. J-LINK 下载程序找不到 Cortex-M 器件的解决方法

J-LINK 下载程序的时候显示“No Cortex-M Device found in JTAG chain. Please check the JTAG cable and the connected devices.”，如图 3-30 所示。

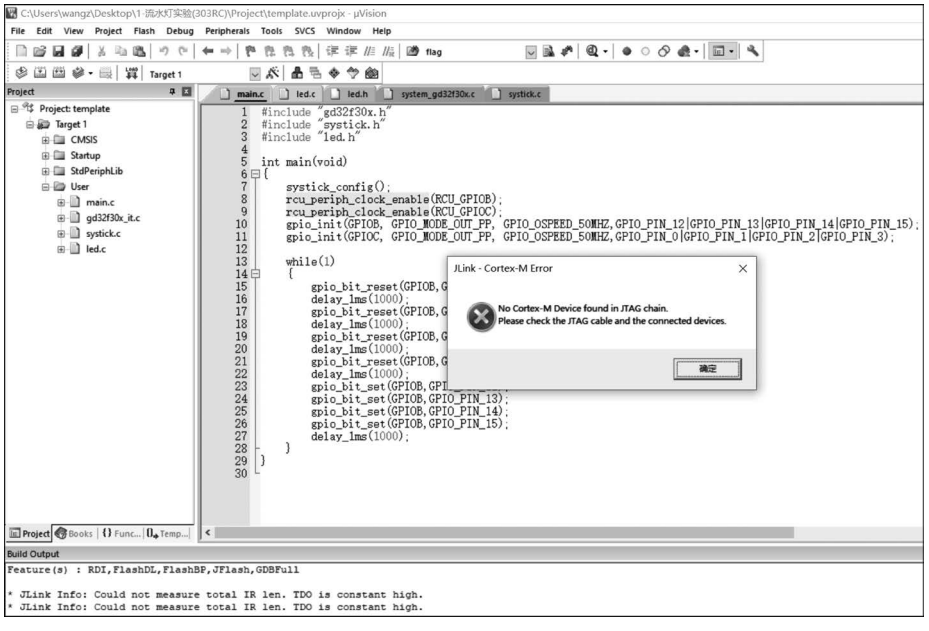

图 3-30　J-LINK 下载程序常见问题

解决办法：单击 Option for target→Debug，选择 J-LINK/J-TRACE Cortex，单击 Settings，Port 选择 SW，如图 3-31 所示。

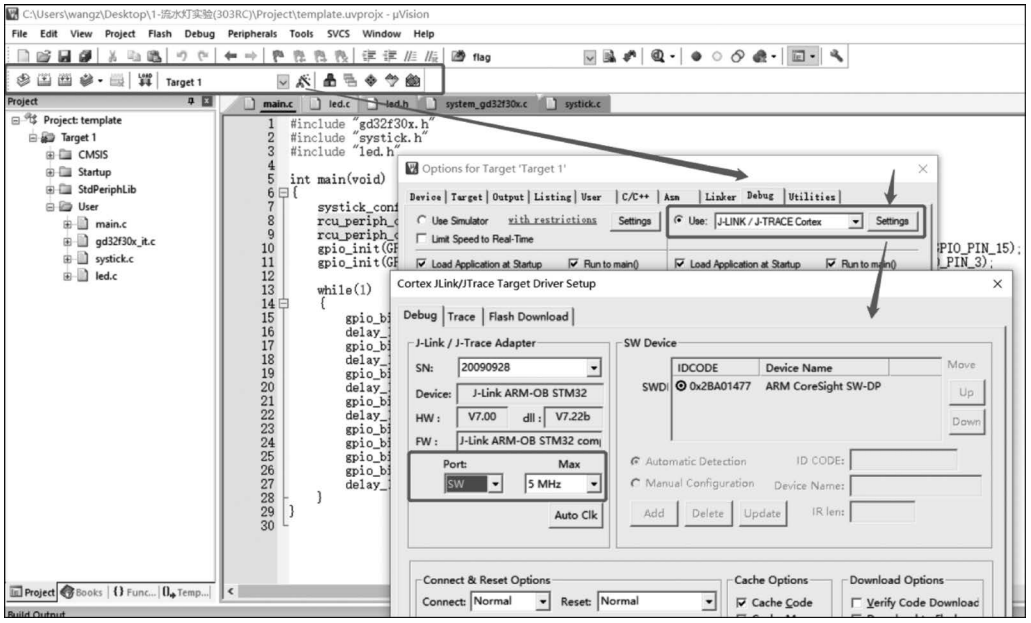

图 3-31　J-LINK 下载程序常见问题解决方法

2. J-LINK 下载提示 NO J-Link found 问题解决方法

如图 3-32 所示，J-LINK 下载过程中有时会提示 NO J-Link found 问题，解决办法：检查 J-LINK 与计算机之间接线，连线没问题的话，检查驱动，重新安装驱动。

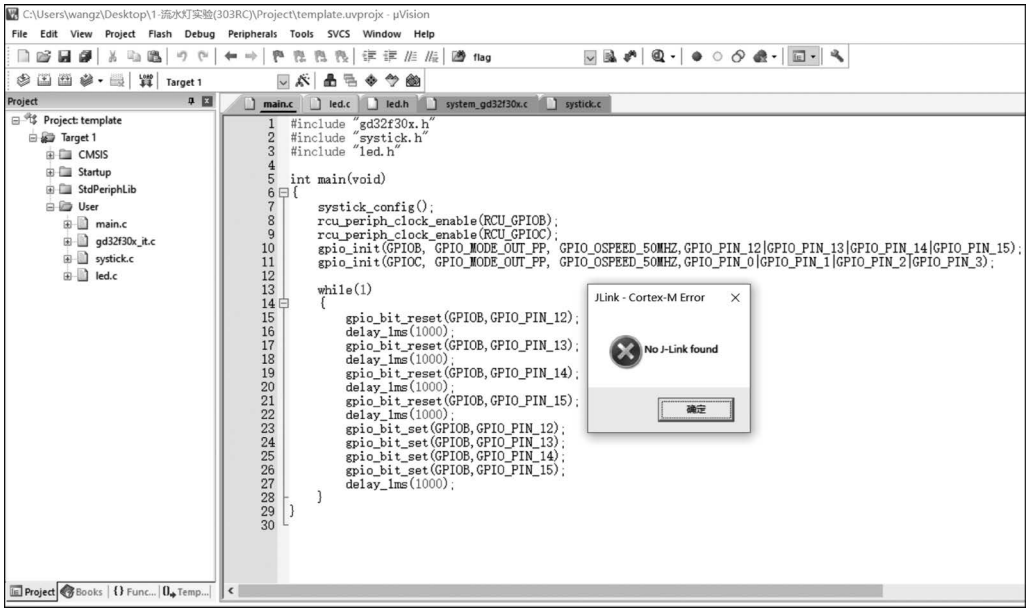

图 3-32　NO J-Link found

3. J-LINK 下载提示 NO ST-LINK detected 问题的解决方法

J-LINK 下载过程中，有时会提示 NO ST-LINK detected，如图 3-33 所示。出现问题的原因是下载 J-LINK 的时候没有正确勾选 J-LINK 下载器，而是选择了 ST-Link 下载器。

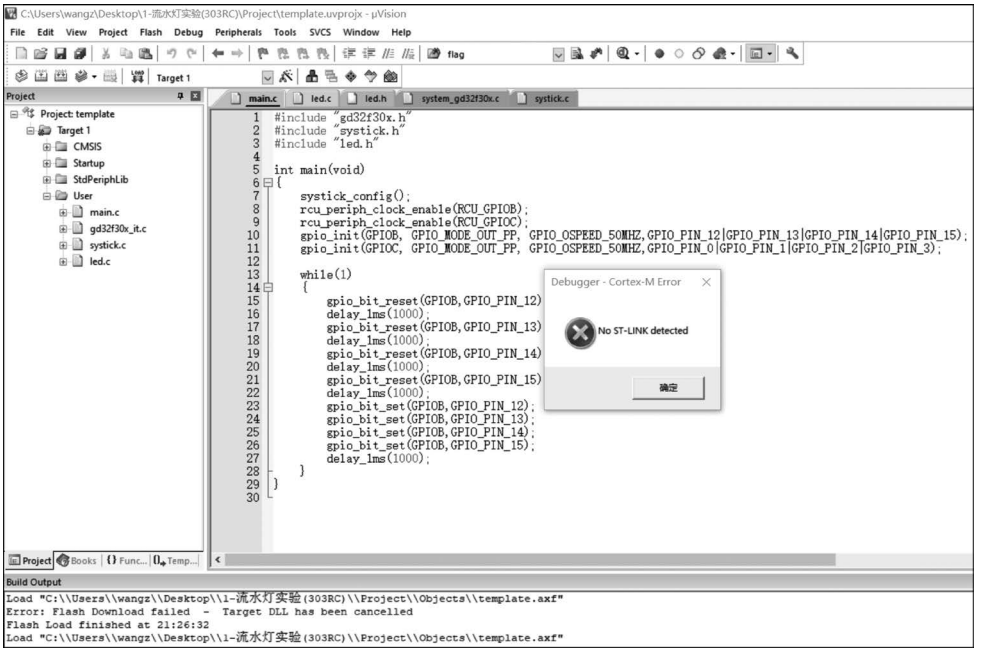

图 3-33　NO ST-LINK detected

解决办法：单击 Option for target→Debug，选择 J-LINK/J-TRACE Cortex，勾选 Run to main()，如图 3-34 所示。

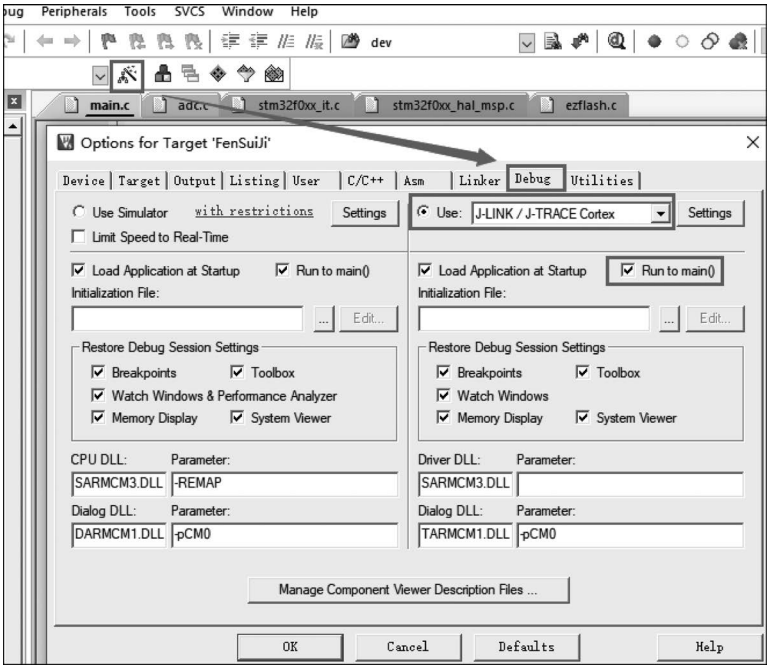

图 3-34　选择 J-LINK/J-TRACE Cortex

3.2　传感器端

传感器端即数据采集端，传感器采集的信息经模数转换后传送给处理器，根据传感器采集的不同数值对 LED 灯进行调光调色的控制。其中传感器和 LED 灯的颜色和亮度的值，是预置的数字，可以根据实际应用场景进行调整。

本实验套件传感器包括温湿度、红外测距、人感、声音和光敏传感器。温湿度传感器有自己特有的时序，所以单独讲解，其他四种信号均接到传感器板上处理器的 ADC 引脚，程序设计类似，所以统一设计和讲解。

3.2.1　温湿度传感器

1. DHT11 温湿度传感器

1) DHT11 温湿度传感器简介

温湿度数据采集使用 DHT11 数字温湿度传感器。DHT11 是一款温湿度一体化的数字传感器。该传感器包括一个电阻式测湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。通过单片机等微处理器简单的电路连接就能够实时采集本地湿度和温度。DHT11 与单片机之间能采用简单的单总线进行通信，仅仅需要一个 I/O 口。传感器内部湿度和温度数据以 40bit 的数据一次性传给单片机，数据采用校验和方式进行校验，有效地保证数据传输的准确性。DHT11 功耗很低，5V 电源电压下，工作平均最大电流为 0.5mA。

2) DHT11 温湿度传感器数据传输

DHT11 温湿度传感器采用单总线数据格式。单个数据引脚端口完成输入输出双向传输。其数据包由 5 字节(40 位)组成。数据分小数部分和整数部分，一次完整的数据传输为 40 位，高位先出。

DHT11 的数据格式为：8 位湿度整数数据＋8 位湿度小数数据＋8 位温度整数数据＋8 位温度小数数据＋8 位校验和。其中，校验和数据为前四个字节相加，如图 3-35 所示。

传感器数据输出的是未编码的二进制数据。数据(湿度、温度、整数、小数)之间应该分开处理。

byte4	byte3	byte2	byte1	byte0
00101101	00000000	00011100	00000000	01001001
整数	小数	整数	小数	校验和
湿度		温度		校验和

图 3-35　DHT11 的数据格式

由以上数据就可得到湿度和温度的值，计算方法：

湿度＝byte4．byte3＝45.0（%RH）

温度＝byte2．byte1＝28.0（℃）

校验＝byte4＋byte3＋byte2＋byte1＝73(校验正确)

3) DHT11 数据发送流程

首先主机发送开始信号，即拉低数据线，保持 t1(至少 18ms)时间，然后拉高数据线

t2(20～40μs)时间,然后读取 DHT11 的响应。正常情况下,DHT11 会拉低数据线,保持 t3(40～50μs)时间,作为响应信号,然后 DHT11 拉高数据线,保持 t4(40～50μs)时间后,开始输出数据。DHT11 数据发送流程如图 3-36 所示。

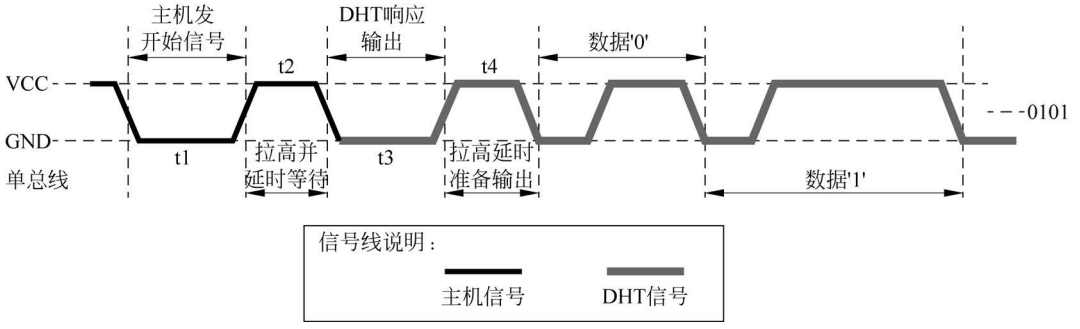

图 3-36 DHT11 数据发送流程

DHT11 输出数字'0'的时序,如图 3-37 所示。

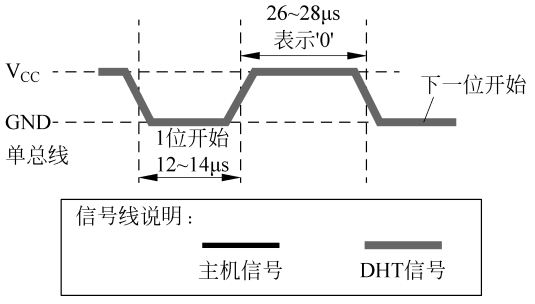

图 3-37 DHT11 输出数字'0'的时序

DHT11 输出数字'1'的时序,如图 3-38 所示。

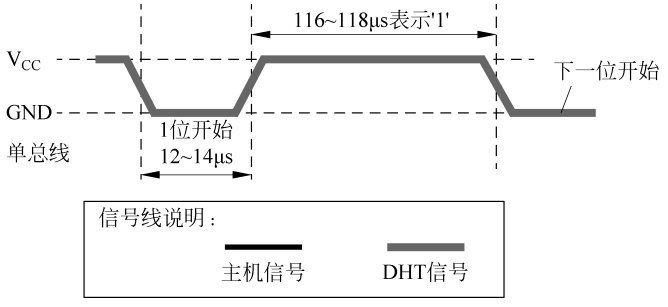

图 3-38 DHT11 输出数字'1'的时序

2. 程序实现

(1) DHT11 的输入输出模式的配置关键代码。

```
cmd = 1,配置为输出模式;cmd = 0,配置为输入模式.
void DHT11_Read_Out_Input(uint8_t cmd)
{
    if(cmd == 1)
    {
    gpio_mode_set(DHT11_GPIO, GPIO_MODE_OUTPUT, GPIO_PUPD_PULLUP, DHT11_
```

```
Pin);
    gpio_output_options_set(DHT11_GPIO,GPIO_OTYPE_PP,GPIO_OSPEED_50MHZ,DHT11_
Pin);
    }
    else
    {
    gpio_mode_set(DHT11_GPIO,GPIO_MODE_INPUT,GPIO_PUPD_PULLUP,DHT11_Pin);
    }
}
```

(2) 读取温湿度数据。

```
uint8_t DHT11_Read_Huim_Temp(uint8_t * HuimH,uint8_t * HuimL,uint8_t * TempH,uint8_t
* TempL)
{
    uint8_t i;
    uint8_t Data[5];
    //主机发送开始信号
    DHT11_Read_Out_Input(1);
    DHT11_Low;
    delay_1ms(20);      //20ms
    DHT11_High;
    delay_1us(40);
    //DHT11 响应信号
    DHT11_Read_Out_Input(0);
    while((DHT11_Input == RESET) && (++Time<1000)); Time = 0;
    while((DHT11_Input == SET) && (++Time<1000)); Time = 0;
    for(i=0;i<5;i++){
        Data[i] = DHT11_Read_Byte();
    }
    delay_1ms(3);
    if((Data[0]+Data[1]+Data[2]+Data[3]) == Data[4]){
        * HuimH = Data[0];
        * HuimL = Data[1];
        * TempH = Data[2];
        * TempL = Data[3];
    }
    else{
        return 1;
    }
    return 0;
}
```

3.2.2　红外、人感、声音和光敏传感器

1. 数据采集简介

数据采集模块采用兆易创新公司的 GD32F330F6P6 微控制器作为主控芯片。GD32F330 系列器件是基于 Arm Cortex-M4 处理器的 32 位通用微控制器。Arm Cortex-M4 处理器包括三条 AHB 总线，分别称为 I-CODE 总线、D-Code 总线和系统总线。Cortex-M4 处理器的所有存储访问，根据不同的目的和目标存储空间，都会在这三条总线上执行。存储器组织采用哈佛结构，预先定义的存储器映射和高达 4GB 的存储空间，充分保证了系统的灵活性和可扩展性。

GD32F330F6P6 主控芯片自带 1 个 12 位 ADC，此处数据采集使用 ADC 的通道 4 作为 A/D 采集输入端，采集红外、人感、声音和光敏传感器的模拟量输出，然后再转换成数字量。

2. 程序实现

(1) 配置 ADC 的 GPIO 端口为模拟输入。

```
void adc_gpio_config(void)
{
    gpio_mode_set(GPIOA,GPIO_MODE_ANALOG,GPIO_PUPD_NONE,GPIO_PIN_1);
}
```

(2) ADC 配置。

```
void adc_config(void)
{
adc_deinit();
    adc_channel_length_config(ADC_REGULAR_CHANNEL, 1);
    adc_regular_channel_config(0, ADC_CHANNEL_1, ADC_SAMPLETIME_55POINT5);
    adc_external_trigger_config(ADC_REGULAR_CHANNEL, ENABLE);
    adc_external_trigger_source_config(ADC_REGULAR_CHANNEL,
ADC_EXTTRIG_REGULAR_NONE);
    adc_data_alignment_config(ADC_DATAALIGN_RIGHT);
    adc_enable();
    adc_calibration_enable();
    adc_special_function_config(ADC_SCAN_MODE, ENABLE);
}
```

3. 获得 ADC 采集的模拟量后转换得到的数字量

```
adc_value = (ADC_RDATA * 3.3 / 4096);
```

3.3 网络传输

3.3.1 DALI

1. DALI 协议

1) 概述

数字可寻址照明接口(digital addressable lighting interface，DALI)协议是照明控制的一个标准，协议编码简单明了，通信结构可靠。DALI 协议是用于满足智能化照明控制需要的非专有标准，它定义了电子镇流器与控制模块之间进行数字化通信的接口标准。DALI 协议被编入欧洲电子镇流器标准“EN60929 附录 E 中”，利用数字化控制方式调节荧光灯的输出光通量。该协议支持“开放系统”的概念，不同制造厂商的产品只要都遵守 DALI 协议就可以相互连接，保证不同制造厂商生产的 DALI 设备能全部兼容。

DALI 系统包含分布式智能模块，各个智能化 DALI 模块都具有数字控制和数字通信能力，地址和灯光场景信息等都存储在各个 DALI 模块的存储器内。DALI 模块通过 DALI 总线进行数字通信，传递指令和状态信息，实现开/关灯、调光控制和系统设置等功能。因此，当 DALI 控制器位置改变时，不需要改动灯的电源线。

DALI 协议是基于主从式控制模型建立的，主从设备通过 DALI 接口连接到 2 芯的总线上。操作人员通过主控制器(荧光灯调光控制器)操作整个系统，可对每个从控制器(电子

镇流器)分别寻址,能够实现对连在同一条控制线上的每个荧光灯的亮度分别进行调光。

2) 电气特性

(1) 异步串行通信协议。

(2) 信息传送速率 1200bit/s,半双工,双向编码。

(3) 双线连接方式。

(4) 电平标准见图 3-39。

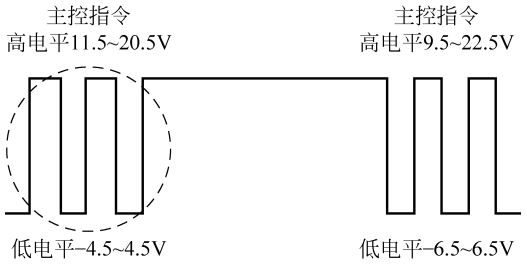

图 3-39　DALI 电平标准

根据 IEC60929 标准,DALI 总线上的最大电流限制为 250mA,每个电子镇流器的电流消耗设定在 2mA; DALI 总线的线路长度不得超过 300m。DALI 线路上最大的电压降应确保不超过 2V,如图 3-40 所示。任何时候,系统都应该保证不能超过这些限制值,否则会降低信号的安全性和完整性,系统运行也变得不稳定。出于这个原因,系统设计者不仅应考虑寻址的方便,也要考虑每个器件的电能消耗,并留有一定的余量以便日后可以进行扩展。

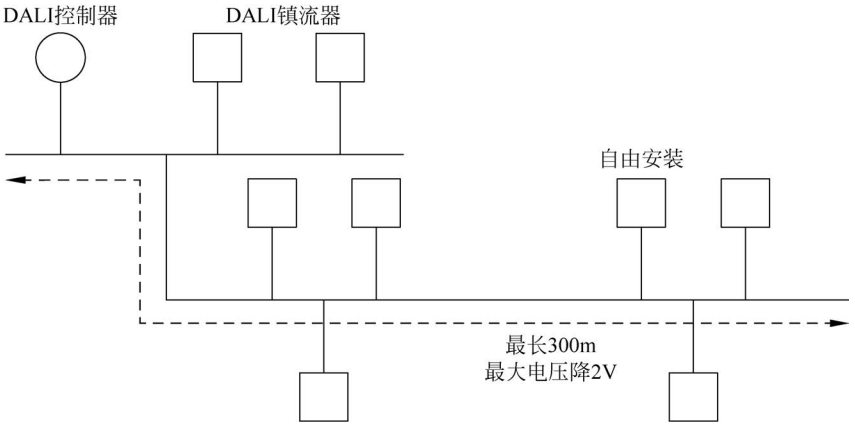

图 3-40　DALI 安装布线图

2. DALI 协议的数据通信

1) DALI 协议的编码

DALI 协议采用双向曼彻斯特编码,如图 3-41 所示。值“1”和“0”表示两种不同的电平跃变,从逻辑低电平转变到高电平表示值“1”,从逻辑高电平转变到低电平表示值“0”。

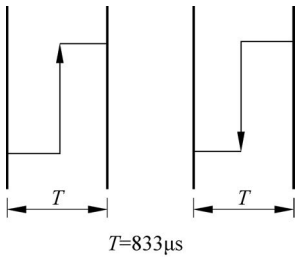

图 3-41　DALI 协议的编码方式

DALI 协议的主控单元向从控单元发出的指令数据由 19 位数据组成,如图 3-42 所示。第 1 位是起始位,第 2～9 位是地址位(这就决定了只能对 64 个从控单元进行单独编

址),第 10～17 位是数据位,第 18、19 位为停止位。

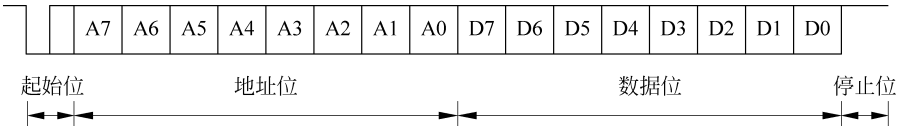

图 3-42　DALI 主控命令

DALI 协议中,只有在主控单元查询时,从控单元才向主控单元发送数据。从控单元向主控单元发送的数据由 11 位数据组成,如图 3-43 所示。第 1 位是起始位,第 2～9 位是数据位,第 10、11 位是停止位。

图 3-43　DALI 从控命令

只有符合上述指令标准的信息,DALI 设备才对其做出反应,否则将不予执行。

2) DALI 协议的指令信息

DALI 的调光指令分为普通指令和专有指令,普通指令用于单播、组播、广播调光、地址分配、镇流器状态查询等;专有指令主要用于整个 DALI 系统的参数配置,如表 3-1 所示。

表 3-1　DALI 地址信息

地 址 形 式	字 节 形 式	使 用 说 明
短地址	YAAAAAAS(AAAAAA=0～63,S=0/1)	单独控制某个从控单元的个体地址
组地址	Y00AAAAS(AAAA=0～15,S=0/1)	成组控制的组地址指令
广播地址	Y111111S(S=0/1)	对所控制的所有从控单元的统一指令
专用命令	Y01CCCCS(CCCC=命令码,S=0/1)	专用指令,可执行特殊的命令

DALI 前向帧的 8 位地址结构为“YAAA AAAS”。AAAAAA 表示具体地址。Y 为地址标志位:Y=0 时,具体地址表示 16 位短地址;Y=1 时,具体地址表示 4 位组地址或广播地址。S 为功能标志位:S=0 时,前向帧的 8 位命令位为调光指令,调光范围 1～255;S=1 时,8 位命令位表示常规控制指令。

在 DALI 信息中,用 8bit 数据表示调光的亮度水平。值“00000000”表示灯没有点亮,DALI 协议按对数调节规则决定灯光亮度水平,在最亮和最暗之间包含 256 级灯光亮度,按对数调光曲线分布。在高亮度具有高增量,在低亮度具有低增量。这样整个调光曲线在人眼里看起来像线性变化。DALI 协议确定的灯光亮度水平在 0.1%～100%,值“00000001”对应 0.1%的亮度水平,值“11111111”对应 100%的亮度水平。

3. 程序实现

(1) 程序发送函数。

```
void key_scan(uint8_t mode)
{
    static uint8_t key_up = 1;
    if(key_up && ((RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_4))\
        ||(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_5))\
        ||(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_6))\
```

```
        ||(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_7))))
    {
        delay_1ms(10);      //按键消抖
        key_up = 0;
        if(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_4))
        {
            addr = 0x01;
            data1 += 10;
            dali_send_command(addr,data1);
        }
        if(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_5))
        {
            addr = 0x02;
            data2 += 10;
            dali_send_command(addr,data2);
        }
        if(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_6))
        {
            addr = 0x03;
            data3 += 10;
            dali_send_command(addr,data3);
        }
        if(RESET == gpio_input_bit_get(GPIOB, GPIO_PIN_7))
        {
            addr = 0x04;
            数据清 0;
            dali_send_command(addr,data);
        }
    }
    return ;
}
```

(2) 接收数据处理函数。

```
void light_process(void)
{
    if(flag == 0)
        return ;
    flag = 0;
    switch(addr)
    {
        case 01:num = (((double)data/ 0xFF) * 10000);
    timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_3,num);
            break;
        case 02:num = (((double)data/ 0xFF) * 10000);
    timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_2,num);
            break;
        case 03:num = (((double)data/ 0xFF) * 10000);
    timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_1,num);
            break;
        case 04:
        timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_1,0);
        timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_2,0);
        timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_3,0);
        break;
    }
}
```

3.3.2 DMX512

1. DMX512 协议定义及格式

DMX512(digital multiplex with 512 pieces of information)。根据 DMX512 数据传输速率的要求及控制网络分散的特点,其物理层的设计采用 RS485 收发器,总线用一对双绞线实现调光台与调光器的相接。数据传输的方式是 DMX512 协议,但数据的交换采用 RS485 通信。

1) 帧结构

一个 DMX 控制字节称为一个指令帧,数据传输速率为 250kbit/s,4μs/bit,44μs/帧。1 帧数据长度为 11 位。第 1 位:起始位,低电平(SPACE);第 2～9 位:数据位(亮度数据,表示 0～255 的 256 级亮度),从最低位到最高位(LSB～MSB);第 10、11 位:停止位,高电平(MARK)。DMX512 协议的帧结构如图 3-44 所示。

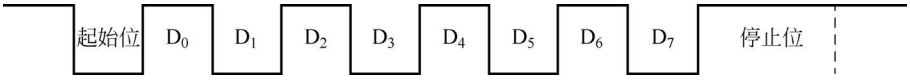

图 3-44 DMX512 协议的帧结构

2) 信息包

DMX512 协议的信息包由一个 MTBP 位,一个 Break 位,一个 MAB 位,一个 SC 和 512 个数据帧组成,数据传输采用异步串行格式。DMX512 字段采取顺序传输,以第 0 字段开始,以最后第 512 字段结束(最大字段数量为 513)。在第一个数据字段开始发送之前,应传输暂停标志,接着传输暂停结束标志和开始码。信息包的格式如图 3-45 所示,信息包电平时间见表 3-2。

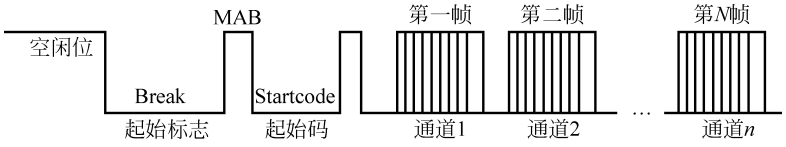

图 3-45 DMX512 协议信息包格式

表 3-2 信息包电平时间表

描　述	最　小　值	典　型　值	最　大　值	单　位
Break	88	88	1000000	μs
MAB	4	8	12	μs
指令帧		44		μs
起始位		4		μs
停止位		8		μs
数据位		4		μs
MTBP	0	NS	1000000	μs

2. 程序实现

(1) DMX512 信息包起始码程序。

```
void dmx512_init(void)
{
```

```
    int i;
    TXDData[0] = 0;
    for(i = 1; i<=512; i++)
    {
        TXDData[i] = i;
    }
    TXDData[1] = 0x01;
    TXDData[2] = 0x09;
    TXDData[3] = value_send_R;
    TXDData[4] = value_send_G;
    TXDData[5] = value_send_B;
}
```

(2) 主控制器发送数据。

```
void dmx_sendpacket(void)
{
    dmx512_init();
    pDMX_buf = 0;
    gpio_tx_config(0);
    gpio_bit_reset(GPIOB,GPIO_PIN_10);
    delay_lus(88);                          //break
    gpio_bit_set(GPIOB,GPIO_PIN_10);
    delay_lus(13);                          //MAB
    gpio_tx_config(1);
    usart_data_transmit(USART2, 0x00);      // SC
    while(RESET == usart_flag_get(USART2, USART_FLAG_TBE)){};

    while(pDMX_buf < 10)                    //1~512
    {
        usart_data_transmit(USART2, 0x0100|TXDData[pDMX_buf]);
        while(SET != usart_flag_get(USART2, USART_FLAG_TBE));
        pDMX_buf++;
    }
}
```

(3) 接收完成处理函数。

```
void dmx512_process(void)
{
    uint32_t tick;
    int len;
    len = dmx_data_idx;
    if(len == 0)
    {
        return ;
    }
    tick = get_Tick();
    if(tick - dmx_last_time > 4)
    {
        if(dmx_data[1] == 0x01 && dmx_data[2] == 0x09)
        {
timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_3,dmx_data[3]);
timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_2,dmx_data[4]);
timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_1,dmx_data[5]);
        }
```

```
            dmx_data_idx = 0;
        }
    }
```

3.3.3 WiFi

1. WiFi 简介

WiFi 是一种可以将 PC、手持设备(如 PDA、手机)等终端以无线方式互相连接的技术。简单来说就是 IEEE 802.11b 的别称,是由一个名为“无线以太网相容联盟”(Wireless Ethernet Compatibility Alliance,WECA)的组织所发布的业界术语,它是一种短程无线传输技术,能够在数百米范围内支持互联网接入的无线电信号。

WiFi 无线网络包括两种类型的拓扑形式:基础网(Infra)和自组网(Ad-hoc)。网络中包括 AP 设备和 STA(站点)。其中,AP 是无线接入点,是一个无线网络的创建者,是网络的中心节点,一般家庭或办公室使用的无线路由器就是一个 AP。每一个连接到无线网络中的终端(如笔记本电脑、PDA 及其他可以联网的用户设备)都可称为一个 STA。

1) 基于 AP 组建的基础无线网络(Infra)

由 AP 创建,众多 STA 加入所组成的无线网络为基础无线网络,这种类型网络的特点是 AP 是整个网络的中心,网络中所有的通信都通过 AP 来转发完成,拓扑结构如图 3-46 所示。

2) 基于自组网的无线网络(Ad-hoc)

自组网仅由两个及以上 STA 组成,网络中不存在 AP,这种类型的网络是一种松散的结构,网络中所有的 STA 都可以直接通信,拓扑结构如图 3-47 所示。

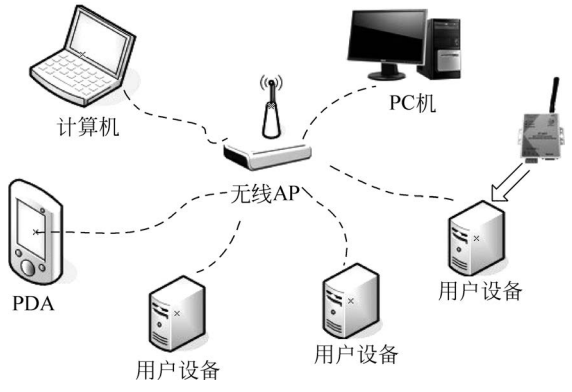

图 3-46 WiFi 基础无线网络

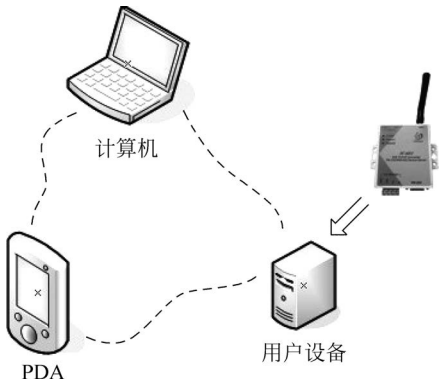

图 3-47 WiFi 自组网网络

2. WiFi 通信协议

根据实际照明控制需求,自行定义 WiFi 照明控制的通信数据协议,如图 3-48 所示。

地址码1字节		控制码1字节				调光数据1字节
箱号	板号	群/单发	同/异协议	串行端口地址	指令形式	调光数据
5位	3位	1位	1位	3位	3位	8位

图 3-48 WiFi 通信数据协议

其中:

（1）地址码：包括5位箱号和3位板号，网络内的最大节点数为32×8。

（2）控制码：包括1位群/单发、1位同/异协议、3位串行端口地址和3位指令形式。控制码默认为0x00。开发者可以根据实际需要，自行定义控制码。

（3）调光数据：包括8位调光信息。00000000表示PWM的占空比为0%，即为关灯指令；11111表示PWM的占空比为100%，即为全亮指令；在最亮和最暗之间包含256级灯光亮度。

3. 程序实现

（1）主控制器发送数据。

```
void KEY_Send(uint8_t mode)
{
    int i = 0;
    static uint8_t key_up=1;
    if(mode)
    {
        key_up=1;
    }
if(key_up==1&&(gpio_input_bit_get(GPIOB,GPIO_PIN_4)==RESET||gpio_input_bit_get
(GPIOB,GPIO_PIN_5)==RESET||gpio_input_bit_get(GPIOB,GPIO_PIN_6)==RESET||gpio_
input_bit_get(GPIOB,GPIO_PIN_7)==RESET))
    {
        delay_1ms(50);
        key_up=0;
        if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
        {
            value_R += 10;
            command[0] = 0x01;
            command[1] = 0x09;
            command[2] = value_R;
            command[3] = value_G;
            command[4] = value_B;
            command[5] = 0x00;
            for(i = 0; i < 6; i++)
            {
                usart_data_transmit(USART2,(uint8_t)command[i]);
                delay_1ms(1);
            }
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_5) == RESET)
        {
            value_G += 10;
```

其余代码同上。

```
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_6) == RESET)
        {
            value_B += 10;
```

其余代码同上。

```
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_7) = = RESET)
        {
            value_R = 0;
            value_G = 0;
```

```
            value_B = 0;
```

其余代码同上。

```
        }
    }
```

(2) WiFi 控制器接收数据处理函数。

```
void USART2_IRQHandler(void)
{
    if(usart_interrupt_flag_get(USART2,USART_INT_FLAG_RBNE) != RESET)
    {
        uint8_t data;
        data = usart_data_receive(USART2);
        if(data == 0x01)
            uart_data_idx = 0;
        uart_data[uart_data_idx++] = data;
        if(uart_data_idx == 5)
        {
            if(uart_data[0] == 0x01 || uart_data[0] == Add_rec)
            {
                if(uart_data[1] == 0x09)
                {
                    value_rec_R = uart_data[2];
                    value_rec_G = uart_data[3];
                    value_rec_B = uart_data[4];
                }
            }
            uart_data_idx = 0;
        }
        usart_interrupt_flag_clear(USART2,USART_INT_FLAG_RBNE);
    }
}
```

3.3.4 ZigBee

1. ZigBee 简介

ZigBee 是一种提供固定、便携或移动设备使用的低复杂度、低成本、低功耗、低速率的无线连接技术。ZigBee 主要适用于自动控制和远程控制领域，可以嵌入在各种设备中，同时支持地理定位功能，非常适合无线传感器网络的通信协议。

ZigBee 通信协议分为 4 层，包括应用层、网络/安全层、介质访问控制层和物理层，如图 3-49 所示。

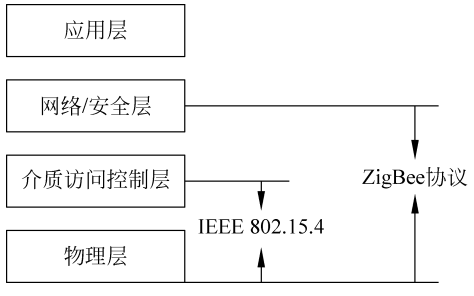

图 3-49 ZigBee 的协议栈

2. ZigBee 通信协议

根据实际照明控制需求，自行定义 ZigBee 照明控制的通信数据协议，如图 3-50 所示。

其中：

(1) 地址码：包括 5 位箱号和 3 位板号，网络内的最大节点数为 32×8。

(2) 控制码：包括 1 位群/单发、1 位同/异协议、3 位串行端口地址和 3 位指令形式。控制码默

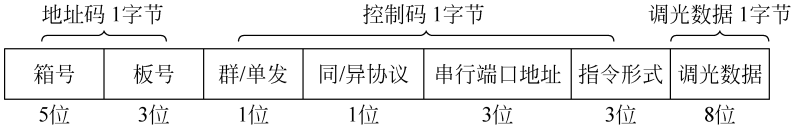

图 3-50　ZigBee 通信数据协议

认为 0x00。开发者可以根据实际需要，自行定义控制码。

（3）调光数据：包括 8 位调光信息。00000000 表示 PWM 的占空比为 0%，即为关灯指令；11111111 表示 PWM 的占空比为 100%，即为全亮指令；在最亮和最暗之间包含 256 级灯光亮度。

3. ZigBee 网络结构

ZigBee 支持星型、树型和网格型等多种拓扑结构，如图 3-51 所示。ZigBee 网络中包括三种节点：协调器、路由器和终端节点。其中协调器和路由器均为全功能设备（FFD），而终端结点选用精简功能设备（RFD）。

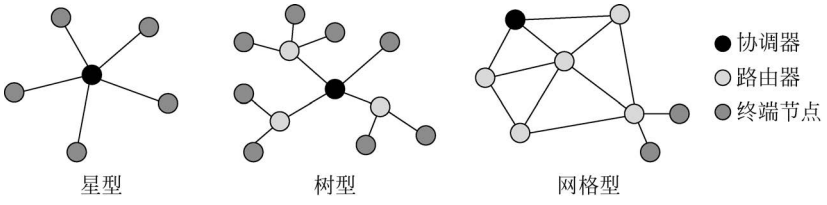

图 3-51　ZigBee 支持的网络拓扑

1）协调器（Coordinator）

一个网络有且只有一个协调器，该设备负责启动网络、配置网络成员地址、维护网络、维护节点的绑定关系表等，需要最多的存储空间和计算能力。

2）路由器（Router）

主要实现扩展网络及路由消息的功能，扩展网络作为网络中的潜在父节点，允许更多的设备接入网络，路由节点只有在树型网络和网格型网络中存在。

3）终端节点（End Device）

不具备成为父节点或路由器的能力，一般作为网络的边缘设备，负责与实际的监控对象相连，这种设备只与自己的父节点主动通信，具体的信息路由则全部交由其父节点及网络中具有路由功能的协调器和路由器完成。

4. 程序实现

（1）主控制器发送数据。

```
void KEY_Send(uint8_t mode)
{
    int i = 0;
    static uint8_t key_up=1;
    if(mode)
    {
        key_up=1;
    }
    if(key_up==1&&(gpio_input_bit_get(GPIOB,GPIO_PIN_4)==RESET||gpio_input_bit_get
(GPIOB,GPIO_PIN_5)==RESET||gpio_input_bit_get(GPIOB,GPIO_PIN_6)==RESET||gpio_
input_bit_get(GPIOB,GPIO_PIN_7)==RESET))
```

```
    {
        delay_1ms(50);
        key_up=0;
        if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
        {
            value_R += 10;
            command[0] = 0x01;
            command[1] = 0x09;
            command[2] = value_R;
            command[3] = value_G;
            command[4] = value_B;
            command[5] = 0x00;
            for(i = 0; i < 6; i++)
            {
                usart_data_transmit(USART2,(uint8_t)command[i]);
                delay_1ms(1);
            }
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_5) == RESET)
        {
            value_G += 10;
```

其余代码同上。

```
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_6) = = RESET)
        {
            value_B + = 10;
```

其余代码同上。

```
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_7) = = RESET)
        {
            value_R = 0;
            value_G = 0;
            value_B = 0;
```

其余代码同上。

```
        }
    }
```

(2) ZigBee 控制器接收数据处理函数。

```
void USART2_IRQHandler(void)
{
    if(usart_interrupt_flag_get(USART2,USART_INT_FLAG_RBNE) != RESET)
    {
        uint8_t data;
        data = usart_data_receive(USART2);
        if(data == 0x01)
            uart_data_idx = 0;
        uart_data[uart_data_idx++] = data;
        if(uart_data_idx == 5)
        {
            if(uart_data[0] == 0x01 || uart_data[0] == Add_rec)
            {
                if(uart_data[1] == 0x09 )
                {
```

```
                    value_rec_R = uart_data[2];
                    value_rec_G = uart_data[3];
                    value_rec_B = uart_data[4];
                }
            }
            uart_data_idx = 0;
        }
        usart_interrupt_flag_clear(USART2,USART_INT_FLAG_RBNE);
    }
}
```

3.3.5　RS485

1. RS485 协议介绍

RS485 协议只是一个物理层协议，数据通信协议需要另行规定。硬件通信接口建立后，在进行数据传输的仪表之间需要约定一个数据协议，以使接收端能够解析收到的数据，这便是“协议”的概念。

RS485 通信接口是一个对通信接口硬件的描述，它只需要两根通信线，就可以在两个或两个以上数据进行传输。对于这种数据传输的方式，某些芯片可以是半双工的通信方式，即在某一时刻，某设备只能进行数据的发送或者接收，采用分时复用原则。

它是能进行联网的通信接口。在 RS485 通信网络中一般采用主从通信方式，即一个主机带多个从机。很多情况下，连接 RS485 通信链路时只是简单地用一对双绞线将各个接口的“A”“B”端连接起来，即可实现主从设备的数据通信。

RS485 的电气特性：采用差分信号负逻辑，逻辑“1”以两线间的电压差为＋(2～6)V 表示；逻辑"0"以两线间的电压差为－(2～6)V 表示。该电平与 TTL 电平兼容，可方便与 TTL 电路连接。

2. RS485 通信协议格式

根据实际照明控制需求，自行定义 RS485 照明控制的通信数据协议，如图 3-52 所示。

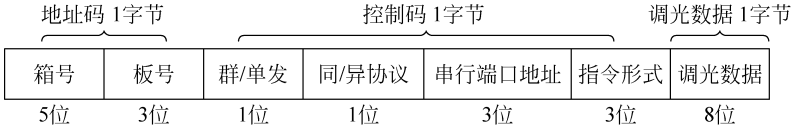

图 3-52　RS485 通信数据协议

其中：

(1) 地址码：包括 5 位箱号和 3 位板号，网络内的最大节点数为 32×8。

(2) 控制码：包括 1 位群/单发、1 位同/异协议、3 位串行端口地址和 3 位指令形式。控制码默认为 0x00。开发者可以根据实际需要，自行定义控制码。

(3) 调光数据：包括 8 位调光信息。00000000 表示 PWM 的占空比为 0%，即为关灯指令；11111111 表示 PWM 的占空比为 100%，即为全亮指令；在最亮和最暗之间包含 256 级灯光亮度。

3. 程序实现

(1) 主控制器发送数据。

```
void KEY_Send(uint8_t mode)
```

```
{
    int i = 0;
    static uint8_t key_up=1;
    if(mode)
    {
        key_up=1;
    }
    if(key_up==1&&(gpio_input_bit_get(GPIOB,GPIO_PIN_4)==RESET||gpio_input_bit_get
(GPIOB,GPIO_PIN_5)==RESET||gpio_input_bit_get(GPIOB,GPIO_PIN_6)==RESET||gpio_
input_bit_get(GPIOB,GPIO_PIN_7)==RESET))
    {
        delay_1ms(50);
        key_up=0;
        if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
        {
            value_R += 10;
            command[0] = 0x01;
            command[1] = 0x09;
            command[2] = value_R;
            command[3] = value_G;
            command[4] = value_B;
            command[5] = 0x00;
            RS485_TX;
            for(i = 0; i<6; i++)
            {
                usart_data_transmit(USART2,(uint8_t)command[i]);
                delay_1ms(1);
            }
            RS485_RX;
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_5) == RESET)
        {
            value_G += 10;
```

其余代码同上。

```
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_6) = = RESET)
        {
            value_B + = 10;
```

其余代码同上。

```
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_7) = = RESET)
        {
            value_R = 0;
            value_G = 0;
            value_B = 0;
```

其余代码同上。

```
        }
    }
```

(2) RS485 控制器接收数据处理函数。

```
void USART2_IRQHandler(void)
{
```

```
    if(usart_interrupt_flag_get(USART2,USART_INT_FLAG_RBNE) != RESET)
    {
        uint8_t data;
        data = usart_data_receive(USART2);
        if(data == 0x01)
            uart_data_idx = 0;
        uart_data[uart_data_idx++] = data;
        if(uart_data_idx == 5)
        {
            if(uart_data[0] == 0x01 || uart_data[0] == Add_rec)
            {
                if(uart_data[1] == 0x09)
                {
                    value_rec_R = uart_data[2];
                    value_rec_G = uart_data[3];
                    value_rec_B = uart_data[4];
                }
            }
            uart_data_idx = 0;
        }
        usart_interrupt_flag_clear(USART2,USART_INT_FLAG_RBNE);
    }
}
```

3.4　主控制器程序设计

主控制器即智慧照明实验箱中的大板，它支撑 GD32 的基础性实验，可以实现处理器和编程环境的认知实验，同时它还是实验箱级的网关，同时作为 AP＋STA，接收综合控制器的控制信息，完成控制大功率 LED 的功能，同时又作为网关实现 WiFi 到其他协议的转换。

3.4.1　流水灯程序设计

1. 流水灯的设计思路

流水灯的概念是一组灯在系统控制下按照设定的顺序和时间发亮和熄灭，形成一定的视觉效果。

本实验平台流水灯的设计思路是让主控制器流水灯模块中的 LED 灯按设定时间依次点亮，然后同时熄灭。之后再依次全部点亮，同时熄灭，循环往复。

2. 流水灯的实现方法

LED 灯即发光二极管，是一种半导体固体发光器件，它是利用固体半导体芯片作为发光材料，当两端加上正向电压，半导体中的载流子发生复合引起光子发射而产生光。

本实验平台流水灯模块的 LED 灯，在电路中其一端已经连接上 3.3V 电源，另一端需要接在 GD32 单片机的 I/O 口。如果要让接在单片机某 I/O 口的 LED 灯点亮，只需让该 I/O 口的电平变为低电平，此时 LED 两端形成正向电压，LED 灯点亮；相反，如果要让接在单片机某 I/O 口的 LED 灯熄灭，只需让该 I/O 口的电平变为高电平，此时 LED 两端没有正向电压，LED 灯熄灭。

本实验平台流水灯的实现方法是利用 GD32 单片机 I/O 口的 8 个引脚分别控制 8 个

LED灯，通过循环控制I/O口的高低电平变换，从而实现LED1～LED8流水灯点亮，再同时熄灭，循环往复。灯的点亮间隔时间通过延时函数自定。

在此，我们还应注意一点，由于人眼的视觉暂留效应以及单片机执行每条指令的时间很短，我们在控制LED灯亮灭的时候应该延时一段时间，否则我们就看不到“流水”效果了。

3. 程序实现

(1) LED的GPIO端口初始化。

```
void LED_init(void)
    {
rcu_periph_clock_enable(RCU_GPIOB);
rcu_periph_clock_enable(RCU_GPIOC);
gpio_init(GPIOB, GPIO_MODE_OUT_PP,
GPIO_OSPEED_50MHZ,GPIO_PIN_12|GPIO_PIN_13|GPIO_PIN_14|GPIO_PIN_15);
gpio_init(GPIOC, GPIO_MODE_OUT_PP,
GPIO_OSPEED_50MHZ,GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
}
```

(2) 主函数实现。

```
while(1)
{
    gpio_bit_reset(GPIOB,GPIO_PIN_12);
    delay_1ms(1000);
    gpio_bit_reset(GPIOB,GPIO_PIN_13);
    delay_1ms(1000);
```

其他GPIO端口同上；

```
    gpio_bit_set(GPIOB,GPIO_PIN_12);
    gpio_bit_set(GPIOB,GPIO_PIN_13);
```

其他GPIO端口同上；

```
    delay_1ms(1000);
}
```

3.4.2 按键输入程序设计

1. 按键简介

按键开关指轻触式按键开关，也称为轻触开关，本实验平台所用按键开关如图3-53所示。按键开关是一种电子开关，按键按下，此时按键开关内部保持闭合状态，如果撤销压力即手拿开，则在内部弹片作用下按键弹开，按键开关内部断开。

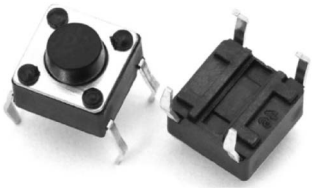

图3-53 本实验平台所用按键开关

按键开关的工作方法本质就是按键的按下与弹开，分别对应GD32单片机检测GPIO输入的两种电平状态。本实验平台按键开关4个引脚分为两组，即对角线为一组。按键的一组引脚接到单片机的I/O口上，另一组与GND连接。当按键按下时，单片机的I/O口与GND连接，端口电平被拉低。因此通过读取端口电平即可获知按键状态。

当按键按下，按键开关内部闭合，按键电路导通，此时 GD32 单片机检测 GPIO 输入为低电平。按键弹开，按键开关内部断开，按键电路不导通，此时 GD32 单片机检测 GPIO 输入为高电平。单片机内部可以通过检测 GPIO 输入的电平高低来判断按键是否被按下，这个判断结果即可作为单片机的输入信号。

2. 按键消抖

按键这种物理器件本身会有抖动信号，抖动信号指的是在电平由高到低也就是在按键按下时，或者电平由低到高也就是在按键抬起过程中，电平的变化不是立刻发生的，而是经过了一段时间的不稳定期才完成，在这个不稳定期间电平可能会时高时低反复变化，这个不稳定期称之为抖动，抖动期内获取按键信息是不可靠的，要想办法消抖。

消抖就是用硬件或软件方法来尽量减少抖动期对获取按键信息的影响。消抖常用两种思路：第一种是硬件消抖，就是尽量减少抖动时间，方法是通过添加电容等元件来减少抖动；第二种是软件消抖，就是发现一次按键按下或抬起事件后，不立即处理按键，而是延时一段时间（一般 10～20ms，这就是消抖时间）后再次获取按键键值，如果此次获取的信息和上次一样是按下/抬起，那就认为真的按下/抬起了。

3. 程序实现

（1）按键的 GPIO 端口初始化。

```
void key_init(void)
{
    rcu_periph_clock_enable(RCU_GPIOA);
    gpio_init(GPIOA, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ, KEYPORT);
                                          //浮空输入模式
}
```

（2）按键扫描函数。

```
uint8_t KEY_Scan(void)
{
    if(gpio_input_bit_get(GPIOA,GPIO_PIN_1) == RESET || gpio_input_bit_get(GPIOA,GPIO
_PIN_4) == RESET
    || gpio_input_bit_get(GPIOA,GPIO_PIN_5) == RESET || gpio_input_bit_get(GPIOA,
GPIO_PIN_6) == RESET
    || gpio_input_bit_get(GPIOA,GPIO_PIN_7) == RESET || gpio_input_bit_get(GPIOA,
GPIO_PIN_8) == RESET
    || gpio_input_bit_get(GPIOA,GPIO_PIN_11) == RESET|| gpio_input_bit_get(GPIOA,
GPIO_PIN_12) == RESET)                    //按键按下
    {
        delay_1ms(20);                    //消抖
        if(gpio_input_bit_get(GPIOA,GPIO_PIN_1) == RESET)return KEY1_PRES;
                                          //KEY1 按下
        else if(gpio_input_bit_get(GPIOA,GPIO_PIN_4) == RESET)return KEY2_PRES;
                                          //KEY2 按下
        //…KEY3,KEY4,KEY5,KEY6,KEY6,KEY7 同上。
    }
    return KEY_UNPRES;                    //无按键按下
}
```

（3）主函数实现。

```
while(1)
```

```
{
    key = KEY_Scan();
    switch(key)
    {
        case KEY1_PRES:
             gpio_bit_write(GPIOB, GPIO_PIN_12, (bit_status)(1 - gpio_output_bit_get
(GPIOB, GPIO_PIN_12)));                    //翻转 LED2 的输出状态
            while(gpio_input_bit_get(GPIOA,GPIO_PIN_1) == RESET);
            break;
        case KEY2_PRES:
             gpio_bit_write(GPIOB, GPIO_PIN_13, (bit_status)(1 - gpio_output_bit_get
(GPIOB, GPIO_PIN_13)));                    //翻转 LED3 的输出状态
            while(gpio_input_bit_get(GPIOA,GPIO_PIN_4) == RESET);
            break;
        //…KEY3、KEY4、KEY5、KEY6、KEY7、KEY8 按键同上。
        case KEY_UNPRES:
            break;
    }
}
```

3.4.3 数码管显示程序设计

1. 数码管简介

本实验平台采用 4 位共阴极数码管。数码管按发光二极管单元连接方式可分为共阳极数码管和共阴极数码管。共阳极数码管是指将所有发光二极管的阳极接到一起形成公共阳极(COM)的数码管,共阳极数码管在应用时应将公共极 COM 接到+5V,当某一字段发光二极管的阴极为低电平时,相应字段就点亮,当某一字段发光二极管的阴极为高电平时,相应字段就不亮。共阴极数码管是指将所有发光二极管的阴极接到一起形成公共阴极(COM)的数码管,共阴极数码管在应用时应将公共极 COM 接到地线(GND)上,当某一字段发光二极管的阳极为高电平时,相应字段就点亮,当某一字段发光二极管的阳极为低电平时,相应字段就不亮。

4 位数码管是将 4 个 1 位数码管的 8 个显示笔画“a、b、c、d、e、f、g、dp”的同名端连在一起,另外每个数码管的公共极 COM 增加位选通控制电路,位选通由各自独立的 I/O 线控制。

2. 数码管显示

当单片机输出字形码时,所有数码管都接收到相同的字形码,但究竟哪个数码管会显示出字形,取决于单片机对位选通 COM 端电路的控制,所以我们只要将需要显示的数码管的位选通控制打开,该位就显示出字形,没有位选通的数码管就不会亮。通过分时轮流控制各个数码管的位选通 COM 端,就可使各个数码管轮流受控显示。

在轮流显示过程中,每位数码管的点亮时间为 1～2ms,由于人的视觉暂留现象及发光二极管的余晖效应,尽管各位数码管并非同时点亮,但只要扫描的速度足够快,整体扫描时间(单个数码管点亮时间×数码管个数)小于 10ms,给人的印象就是一组稳定的显示数据,感觉不到闪烁。

3. 程序实现

(1) 数码管端口配置函数。

```
void SMG_UserConfig(void)
```

```
{
    rcu_periph_clock_enable(RCU_AF);
    rcu_periph_clock_enable(RCU_GPIOA);
    rcu_periph_clock_enable(RCU_GPIOB);
    gpio_pin_remap_config(GPIO_SWJ_SWDPENABLE_REMAP,ENABLE);
    gpio_init(SMG_PORT, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ, SMG_PIN);
    gpio_init(WEI_PORT, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ, WEI_PIN);
    gpio_bit_reset(WEI_PORT,WEI_PIN);
}
```

(2) 数码管显示函数。

```
void SMG_Display(uint8_t dat)
{
    gpio_port_write(SMG_PORT,data[dat]);
}
```

(3) 主函数实现。

```
while(1)
{
    if(RESET == gpio_input_bit_get(GPIOB,GPIO_PIN_8))
    {
        //延时 20ms 用于消除抖动
        delay_1ms(20);
        if(RESET == gpio_input_bit_get(GPIOB,GPIO_PIN_8))
        {
            for(i = 0; i < 17; i++)
            {
                SMG_Display(i);
                delay_1ms(1000);
            }
            while(RESET == gpio_input_bit_get(GPIOB,GPIO_PIN_8));
        }
    }
}
```

3.4.4　串口通信程序设计

1. 串口简介

串口,也称串行通信接口,是采用串行通信方式的扩展接口。串行接口是指数据一位一位地顺序传送,其特点是通信线路简单,只要一对传输线就可以实现双向通信。

本实验平台的主控制器芯片 GD32F303RCT6 最多支持 3 个 USART 和 2 个 UART,工作频率高达 7.5Mbps,支持异步和时钟同步串行通信模式。USART(USART0、USART1、USART2)和 UART(UART3、UART4)用于在并行和串行接口之间转换数据,数据帧可以通过全双工或半双工,同步或异步的方式进行传输。USART/UART 包括一个可编程波特率发生器,能够分割系统时钟,为 USART 发射机和接收机生成专用时钟。

USART 不仅支持标准的异步收发模式,还实现了一些其他类型的串行数据交换模式,如红外编码规范,SIR,智能卡协议,LIN,半双工以及同步模式。它还支持多处理器通信和 Modem 流控操作(CTS/RTS)。数据帧支持从 LSB 或者 MSB 开始传输。数据位的极性和 TX/RX 引脚都可以灵活配置。

2. 串口通信

串口通信一般是以帧格式传输数据，即一帧一帧传输，每帧包含起始信号、数据信息、停止信息，可能还有校验信息。所以，USART 数据格式一般分为启动位、数据帧、可能的奇偶校验位、停止位，如图 3-54 和图 3-55 所示。

启动位：发送方想要发送串口数据时，必须先发送启动位。

数据帧：发送的数据内容，数据的 bit 位。有 8 位字长和 9 位字长两种。

可能的奇偶校验位：在串口通信中一种简单的检错方式，没有校验位也是可以的。对于偶和奇校验的情况，串口会设置校验位(数据位后面的一位)，用一个值确保传输的数据有偶数个或者奇数个逻辑高位。

停止位：停止位不仅表示传输的结束，并且还提供计算机校正时钟同步的机会。

通常情况下，默认选择的 USART 数据格式为 8 位数据字长、无奇偶校验位、1 位停止位。

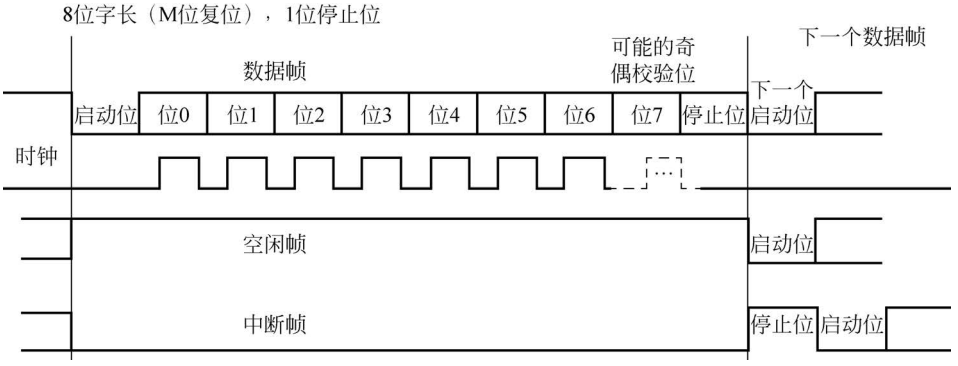

图 3-54　串口数据格式(8 位字长)

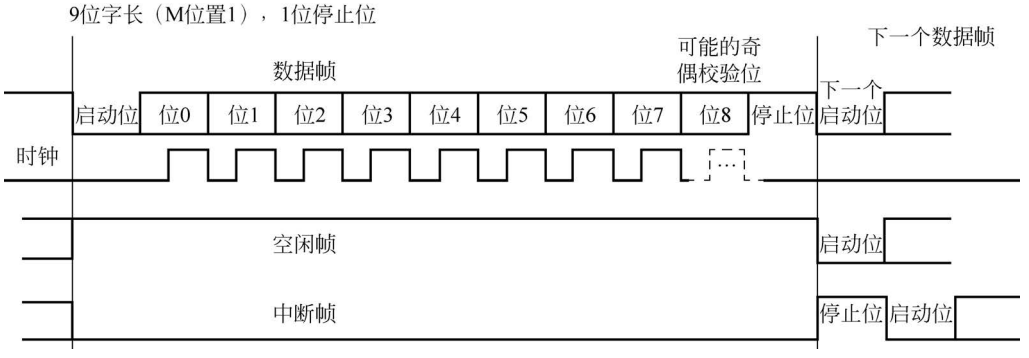

图 3-55　串口数据格式(9 位字长)

两个串口之间的连接方式，如图 3-56 所示。

3. USART 和 UART 的区别

USART：通用同步和异步收发器；

UART：通用异步收发器。

当进行异步通信时，这两者是没有区别的。区别在于 USART 比 UART 多了同步通信功能。

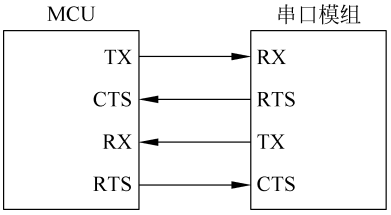

图 3-56　串口连接方式

同步是指发送方发出数据后，等接收方发回响应以后才发下一个数据包的通信方式。

异步是指发送方发出数据后，不等接收方发回响应，就发送下个数据包的通信方式。

4. 程序实现

(1) 串口 1 配置函数。

```
void usart1_init(void)
{
    rcu_periph_clock_enable(RCU_AF);
    rcu_periph_clock_enable(RCU_GPIOA);
    rcu_periph_clock_enable(RCU_USART1);
    gpio_init(GPIOA, GPIO_MODE_AF_PP, GPIO_OSPEED_50MHZ, GPIO_PIN_2);
    gpio_init(GPIOA, GPIO_MODE_IN_FLOATING, GPIO_OSPEED_50MHZ, GPIO_PIN_3);
        usart_deinit(USART1);
        usart_baudrate_set(USART1,115200);
        usart_parity_config(USART1,USART_PM_NONE);
        usart_word_length_set(USART1,USART_WL_8BIT);
        usart_stop_bit_set(USART1,USART_STB_1BIT);
        usart_transmit_config(USART1,USART_TRANSMIT_ENABLE);
        usart_receive_config(USART1,USART_RECEIVE_ENABLE);
        usart_enable(USART1);
}
```

(2) 将 C 语言库函数 printf 重新定向到 USART。

```
int fputc(int ch, FILE *f)
{
    usart_data_transmit(USART1, (uint8_t)ch);
    while (RESET == usart_flag_get(USART1,USART_FLAG_TBE));
                                                //USART_FLAG_TBE 传输数据缓冲区为空
    return ch;
}
```

3.4.5　中断程序设计

1. 中断原理

中断是实现多任务的基础，也是 I/O 的一种基本工作方式。中断包括可屏蔽中断和非屏蔽中端，用户可以使用的是可屏蔽中断。可屏蔽中断包括内部中断和外部中断两种，其中定时器中断和串行口中断属于内部中断。不同的中断由中断服务程序实现其功能。

GD32 集成了嵌套式矢量型中断控制器(nested vectored interrupt controller，NVIC)来实现高效的异常和中断处理。NVIC 实现了低延迟的异常和中断处理，以及电源管理控制。它和内核是紧密耦合的。

EXTI(中断/事件控制器)包括 20 个相互独立的边沿检测电路并且能够向处理器内核产生中断请求或唤醒事件。EXTI 有三种触发类型：上升沿触发、下降沿触发和任意沿触发。EXTI 中的每一个边沿检测电路都可以独立配置和屏蔽。

中断触发源包括来自 I/O 引脚的 16 根线以及来自内部模块的 4 根线(包括 LVD、RTC 闹钟、USB 唤醒、以太网唤醒)。通过配置 GPIO 模块的 AFIO_EXTISSx 寄存器，所有的 GPIO 引脚都可以被选作 EXTI 的触发源。

MCU 通过检测中断寄存器来判断是否有外部中断请求。如果有中断产生，则 MCU 暂停执行正在执行的进程，优先处理中断事件。对不同的中断事件，由于它们的性质不同，对应的处理程序不同。

2. 中断过程

中断过程：中断的完整过程包括中断申请、中断响应、中断处理和中断返回。每一个过程的实现都有对应的寄存器支撑。例如，中断请求寄存器、中断屏蔽寄存器、中断优先级寄存器、中断标志寄存器等。

若出现中断事件，硬件就把它记录在中断请求寄存器中。中断请求寄存器的每一位与一个中断事件对应(通过引脚)，当出现某中断事件后，对应的中断寄存器的对应位就被置成“1”。然后根据中断使能寄存器去查看此中断是否被使能，假设没有被使能则不能响应此中断请求；假设中断被使能，然后看该中断优先级，假设此中断正好处于最高级，则 MCU 为该中断进行服务，即根据中断类型号去中断矢量表找中断服务程序的入口地址，然后去执行，完成中断服务。

在中断服务程序的最后是中断返回指令，即完成中断服务后，返回到调用中断的地方继续执行原来的任务。假设该中断优先级较低，则处于等待状态，直到该中断处于高优先级状态。本实验第一个内容是用按键按下动作模拟中断事件。第二个内容为定时器中断，通过时间计数产生中断。对其他中断事件的模拟处理，可根据各中断事件的性质确定处理原则，制定算法。

3. 外部中断程序实现

(1) 启动外部中断。

```
void Key_ExitConfig()
{
    nvic_irq_enable(EXTI5_9_IRQn, 2U, 0U);
    gpio_exti_source_select(GPIO_PORT_SOURCE_GPIOB, GPIO_PIN_SOURCE_8);
    gpio_exti_source_select(GPIO_PORT_SOURCE_GPIOB, GPIO_PIN_SOURCE_9);
    exti_init(EXTI_8, EXTI_INTERRUPT, EXTI_TRIG_FALLING);      //下降沿触发
    exti_init(EXTI_9, EXTI_INTERRUPT, EXTI_TRIG_FALLING);      //下降沿触发
}
```

(2) 中断实现数字的加减函数。

```
void EXTI5_9_IRQHandler(void)
{
    if(RESET != exti_interrupt_flag_get(EXTI_8))
    {
        i = i + 1;
        exti_interrupt_flag_clear(EXTI_8);
    }
```

```
    if(RESET != exti_interrupt_flag_get(EXTI_9))
    {
        i = i - 1;
            exti_interrupt_flag_clear(EXTI_9);
    }
}
```

4. 定时器中断程序实现

(1) 启动定时器中断。

```
void timer_config(void)
{
timer_parameter_struct timer_initpara;
    rcu_periph_clock_enable(RCU_TIMER1);
    timer_deinit(TIMER1);
    timer_initpara.prescaler= 11999;
    timer_initpara.alignedmode= TIMER_COUNTER_EDGE;
    timer_initpara.counterdirection= TIMER_COUNTER_UP;
    timer_initpara.period= 9999;
    timer_initpara.clockdivision= TIMER_CKDIV_DIV1;
    timer_initpara.repetitioncounter = 0;
    timer_init(TIMER1,&timer_initpara);
        //开启定时器中断
timer_interrupt_enable(TIMER1,TIMER_INT_UP);
            nvic_priority_group_set(NVIC_PRIGROUP_PRE1_SUB3);
nvic_irq_enable(TIMER1_IRQn, 0, 1);
    timer_primary_output_config(TIMER1,ENABLE);
    timer_auto_reload_shadow_enable(TIMER1);
    timer_enable(TIMER1);
}
```

(2) 定时器计数函数。

```
void TIMER1_IRQHandler()
{
            if(timer_interrupt_flag_get(TIMER1,TIMER_INT_FLAG_UP) != RESET)
            {
                i++;
                if(i > 60)
                    i = 1;
            }
            timer_interrupt_flag_clear(TIMER1,TIMER_INT_FLAG_UP);
}
```

3.4.6　A/D 转换程序设计

1. A/D 转换原理

本实验平台的主控制器采用 GD32F303RCT6 芯片,MCU 自带模数转换器(ADC)。所以不需扩展 AD 专用芯片。

ADC 是一种采用逐次逼近方式的模拟数字转换器。它有 18 个多路复用通道,可以转换来自 16 个外部通道和 2 个内部通道的模拟信号。各种通道的 A/D 转换可以配置成单次、连续、扫描或间断转换模式。ADC 转换的结果可以按照左对齐或右对齐的方式存储在

16 位数据寄存器中。片上的硬件过采样机制可以通过减少来自 MCU 的相关计算负担来提高性能。

2. ADC 主要特征

(1) 高性能。

可配置 12 位、10 位、8 位或 6 位分辨率；自校准；可编程采样时间；数据寄存器可配置数据对齐方式；支持规则数据转换的 DMA 请求。

(2) 有 18 个多路复用通道。

包括 16 个外部模拟输入通道、1 个内部温度传感通道(VSENSE)和 1 个内部参考电压输入通道(VREFINT)。

(3) 两种转换方式。

可以通过软件触发，也可以通过硬件触发。

(4) 包含多种转换模式。

有转换单个通道，或者扫描一序列的通道；单次模式，每次触发转换一次选择的输入通道；连续模式，连续转换所选择的输入通道；间断模式；同步模式(适用于具有两个或多个 ADC 的设备)。

(5) 可产生中断。

中断的产生方式有规则组转换完成、注入组转换完成和模拟看门狗事件。

(6) 过采样。

包括 16 位的数据寄存器；可调整的过采样率，从 2x 到 256x；高达 8 位的可编程数据移位。

(7) ADC 供电要求。

2.6～3.6V，一般电源电压为 3.3V。

3. 程序实现

实验采用 ADC0 通道 1、通道 4 和通道 5 作为 A/D 采集输入端。

(1) AD 转换器配置。

```
void adc_config(void)
{
    adc_deinit(ADC0);
    adc_mode_config(ADC_MODE_FREE);
    adc_data_alignment_config(ADC0,ADC_DATAALIGN_RIGHT);
    adc_special_function_config(ADC0,ADC_SCAN_MODE,ENABLE);
    adc_channel_length_config(ADC0,ADC_INSERTED_CHANNEL,3);
     adc_inserted_channel_config(ADC0, 0, ADC_CHANNEL_1, ADC_SAMPLETIME_
239POINT5);
     adc_inserted_channel_config(ADC0, 1, ADC_CHANNEL_4, ADC_SAMPLETIME_
239POINT5);
     adc_inserted_channel_config(ADC0, 2, ADC_CHANNEL_5, ADC_SAMPLETIME_
239POINT5);
    adc_external_trigger_config(ADC0,ADC_INSERTED_CHANNEL,ENABLE);
     adc_external_trigger_source_config(ADC0, ADC_INSERTED_CHANNEL, ADC0_1_2_
EXTTRIG_INSERTED_NONE);
    adc_enable(ADC0);
    delay_1ms(1);
```

```
    adc_calibration_enable(ADC0);
}
```

(2) PWM 输出控制主控制器 LED 灯的亮度或颜色。

```
dac_value1 = (ADC_IDATA0(ADC0) * 3.3 / 4096);
dac_value2 = (ADC_IDATA1(ADC0) * 3.3 / 4096);
dac_value3 = (ADC_IDATA2(ADC0) * 3.3 / 4096);
timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_3,dac_value1 * 100);
timer_channel_output_pulse_value_config(TIMER2,TIMER_CH_2,dac_value2 * 100);timer_channel
_output_pulse_value_config(TIMER2,TIMER_CH_1,dac_value3 * 100);
```

3.5　局域网网关程序设计

3.5.1　单控程序设计

1. LED 单灯控制简介

本开发平台支持 DALI、DMX12、RS485、ZigBee 和 WiFi 五种通信协议，能够实现主控制器通过某个协议对相应中控制器进行控制的功能。

LED 单灯控制是指主控制器对单个中控制器的 LED 灯的控制，将主控制器的按键信息发送给中控制器。中控制器需要设置为接收模式，中控制器根据接收的数据，控制 RGB LED 灯的开关、亮度和颜色。每按下一次主控制器的通信按键，发送一次数据，并且数据递加 10。

LED 单灯控制结构如图 3-57 所示。

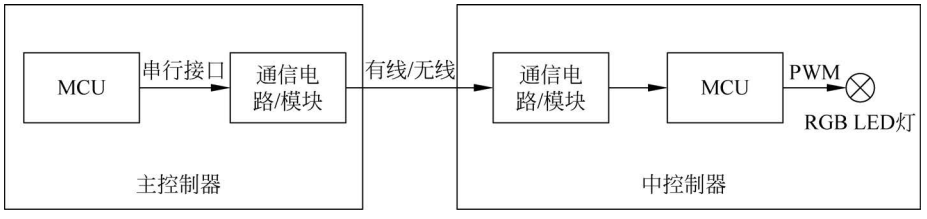

图 3-57　LED 单灯控制结构

2. LED 单灯控制流程

1) LED 单灯控制过程

首先，通过按下主控制器的按键，使主控制器的 MCU 发送相应的控制信息，经过 DALI、DMX12、RS485、ZigBee 和 WiFi 五种通信方式中的某一通信模块传输。然后，中控制器设置为接收模式，中控制器通过对应的通信模块接收该控制信息，经过中控制器的 MCU 处理后，实现对中控制器上 RGB LED 灯的开关、亮度和颜色控制。

2) LED 单灯控制程序流程

通过按键选择灯的颜色和亮度，每按下一次按键，亮度增加 10。LED 单灯控制程序流程如图 3-58 所示。

3. 数据格式

根据实际照明控制需求，自行定义照明控制的通信数据协议，如图 3-59 所示。

其中：

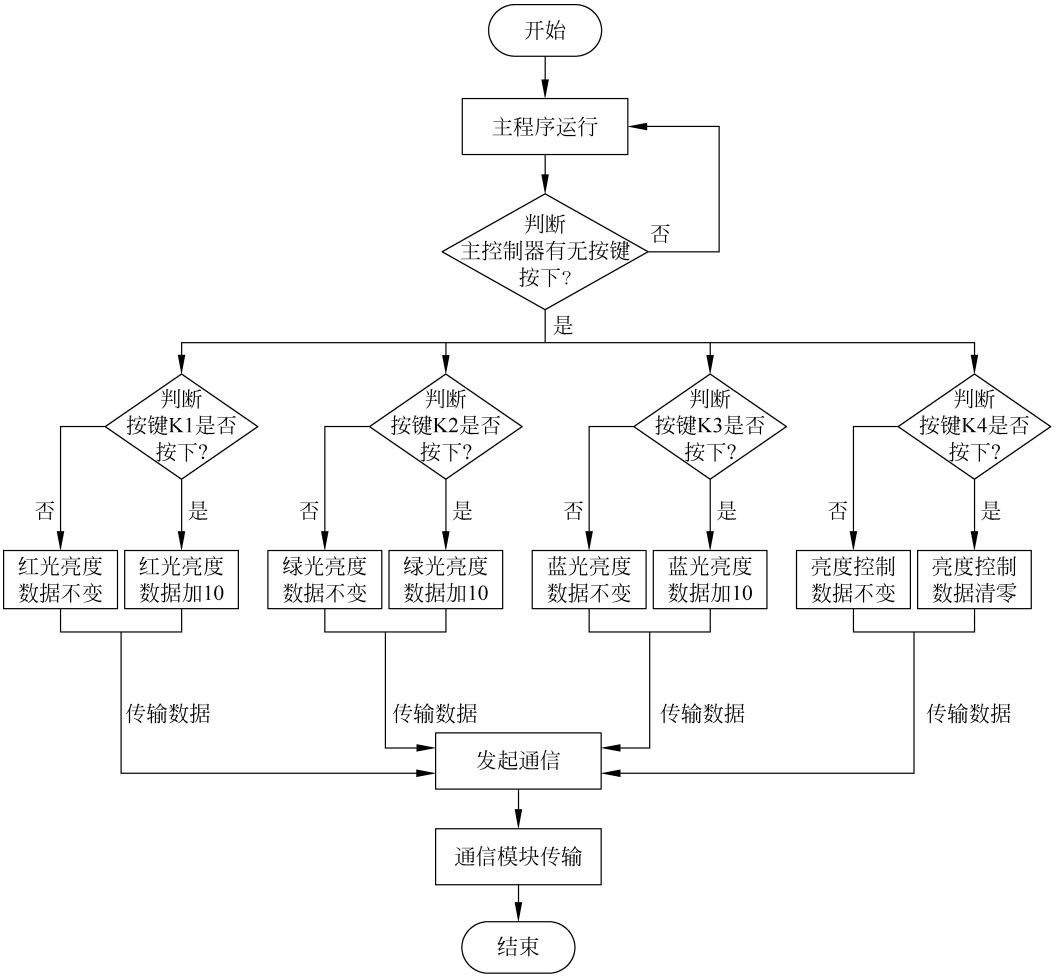

图 3-58　LED 单灯控制程序流程

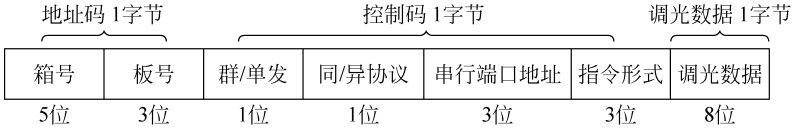

图 3-59　LED 单灯控制通信数据格式

(1) 地址码：包括 5 位箱号和 3 位板号，网络内的最大节点数为 32×8。

(2) 控制码：包括 1 位群/单发、1 位同/异协议、3 位串行端口地址和 3 位指令形式。控制码默认为 0x00。开发者可以根据实际需要，自行定义控制码。

(3) 调光数据：包括 8 位调光信息。00000000 表示 PWM 的占空比为 0%，即为关灯指令；11111111 表示 PWM 的占空比为 100%，即为全亮指令；在最亮和最暗之间包含 256 级灯光亮度。

4. 程序关键代码

(1) 主控制器发送数据函数，代码如下所示。

```
void KEY_Send()
```

```
{
    if(gpio_input_bit_get(GPIOB,GPIO_PIN_4)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_5)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_6)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_7)==RESET)      //有按键按下
    {
        delay_1ms(50);                                  //消抖
        if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
        {
            value_R += 10;                              //K1 按下,红光数据加 10
            command[0] = 0x01;
            command[1] = 0x09;
            command[2] = value_R;
            command[3] = value_G;
            command[4] = value_B;
            command[5] = 0x00;                          //结束标志
            RS485_TX;
            for(i = 0; i < 6; i++)
            {
                usart_data_transmit(USART2,(uint8_t)command[i]);
                delay_1ms(1);
            }
            RS485_RX;
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_5) == RESET)
        {
            value_G += 10;                              //K2 按下,绿光数据加 10
            …;                                          //其他代码同上
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_6) == RESET)
        {
            value_B += 10;                              //K3 按下,蓝光数据加 10
            …;                                          //其他代码同上
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_7) == RESET)
        {
            value_R = 0;value_G = 0;value_B = 0;        //K4 按下,数据清零
            …;                                          //其他代码同上
        }
    }
}
```

(2) 中控制器接收数据函数,代码如下所示。

```
void USART2_IRQHandler(void)
{
    if(usart_interrupt_flag_get(USART2,USART_INT_FLAG_RBNE) != RESET)
    {
        uint8_t data;
        data = usart_data_receive(USART2);
        if(data == 0x01)
            uart_data_idx = 0;
        uart_data[uart_data_idx++] = data;
        if(uart_data_idx == 5)
        {
```

```
                if(uart_data[0] == 0x01)
                {
                    if(uart_data[1] == 0x09)
                    {
                        value_rec_R = uart_data[2];
                        value_rec_G = uart_data[3];
                        value_rec_B = uart_data[4];
                    }
                }
                uart_data_idx = 0;
            }
            usart_interrupt_flag_clear(USART2,USART_INT_FLAG_RBNE);
        }
    }
```

3.5.2 群组控制程序设计

1. LED 群组控制简介

本开发平台支持 DALI、DMX12、RS485、ZigBee 和 WiFi 五种通信协议，能够实现主控制器通过某个协议对相应中控制器进行控制的功能。

LED 群组控制，即设置主控板为集控中心，实现对同种协议多个板子的 LED 灯的调光调色控制。

将主控制器设置为主设备，中控制器设置为从设备，为多个中控制器分配不同的从地址。主控制器发出控制指令，各个中控制器接收并执行指令，通过各个中控板的 RGB LED 灯显示被控效果，进而实现基于同种协议的 LED 群组控制的目的。

LED 群组控制中，主控制器和中控制器的通信结构如图 3-60 所示。

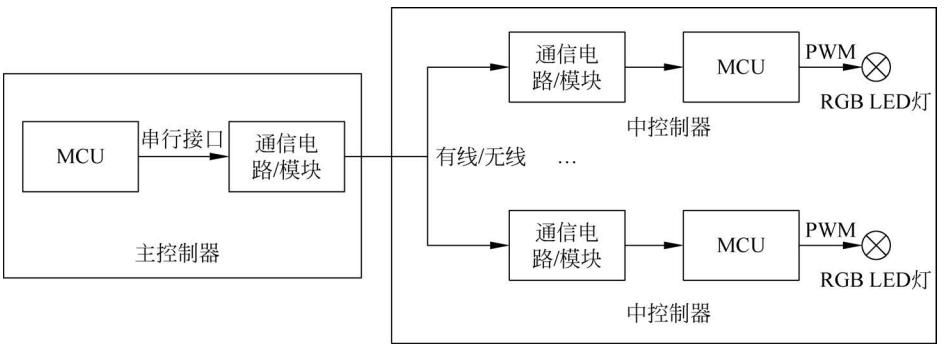

图 3-60 LED 群组控制中的通信结构

2. LED 群组控制流程

1）LED 群组控制过程

LED 群组控制是指主控制器对多个中控制器的 LED 灯的控制，将主控制器的按键信息发送给多个中控制器。中控制器设置为接收模式，多个中控制器设置不同的地址值，中控制器根据接收的数据，控制 RGB LED 灯的开关、亮度和颜色。

首先，通过按下主控制器的按键，使主控制器的 MCU 发送相应的控制信息，经过 DALI、DMX12、RS485、ZigBee 和 WiFi 五种通信方式中的某一通信模块传输。然后，多个中控制器设置为接收模式并分配不同的地址，多个中控制器通过对应的通信模块接收该控

制信息，经过 MCU 处理后，实现对板子上 RGB LED 灯的控制。

2）LED 群组控制程序流程

通过按键选择灯的颜色和亮度，每按下一次按键，亮度增加 10。LED 群组控制程序流程如图 3-61 所示。

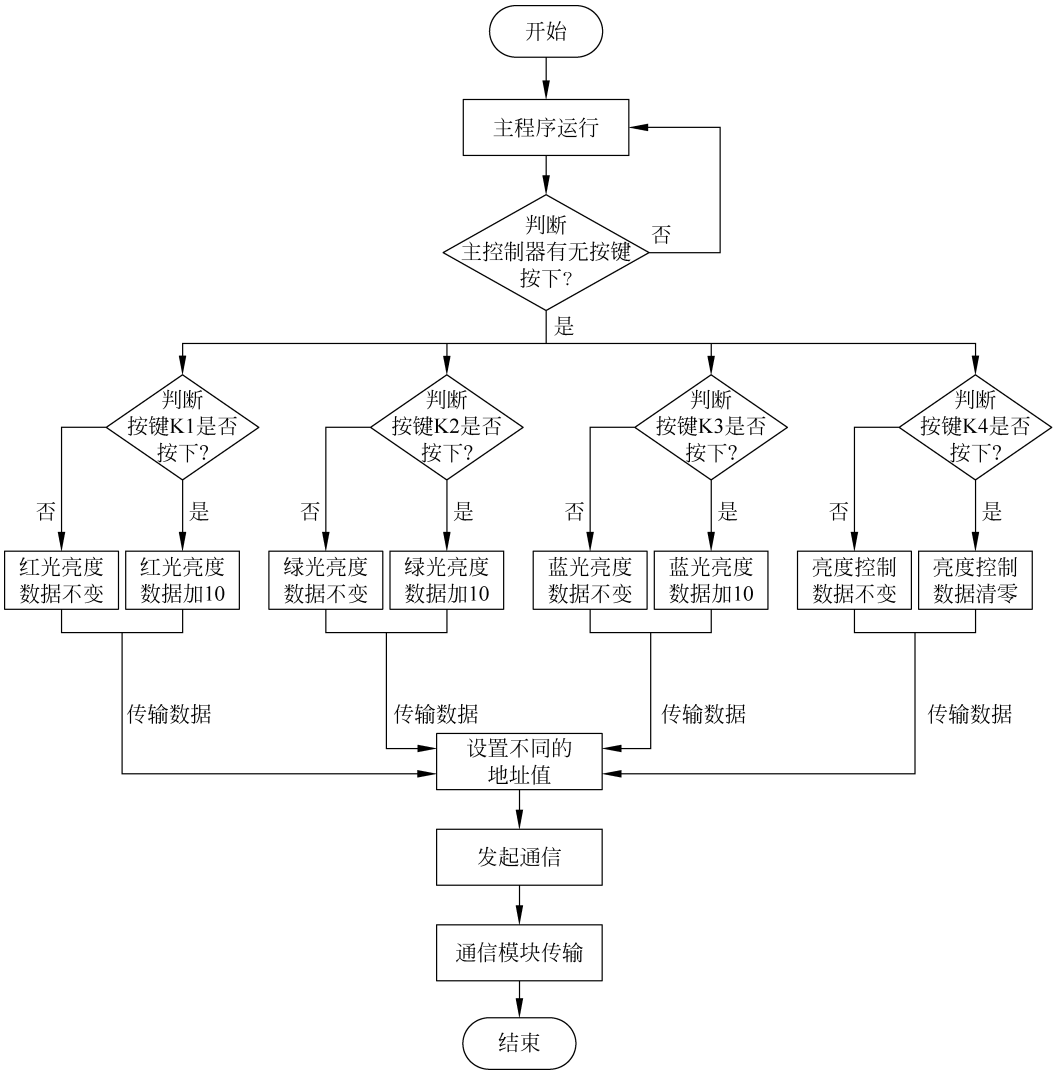

图 3-61　LED 群组控制程序流程

3. 程序关键代码

（1）主控制器发送数据函数，代码如下所示。

```
int add = 0;
void KEY_Send()
{
    if(gpio_input_bit_get(GPIOB,GPIO_PIN_4)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_5)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_6)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_7)==RESET)                    //按键按下
```

```
    {
        delay_1ms(50);                                          //消抖
        if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
        {
            value_R += 10;                                      //K1 按下,红光数据加 10
            //群组发送
            for( add = 1 ; add < 3 ;add++)
            {
                command[0] = add;
                command[1] = 0x09;
                command[2] = value_R;
                command[3] = value_G;
                command[4] = value_B;
                command[5] = 0x00;                              //结束标志
                RS485_TX;
                for(i = 0; i < 6; i++)
                {
                    usart_data_transmit(USART2,(uint8_t)command[i]);
                    delay_1ms(1);
                }
                RS485_RX;
            }
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_5) == RESET)
        {
            value_G += 10;                                      //K2 按下,绿光数据加 10
            …;                                                  //其他代码同上
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_6) == RESET)
        {
            value_B += 10;                                      //K3 按下,蓝光数据加 10
            …;                                                  //其他代码同上
        }
        else if(gpio_input_bit_get(GPIOB,GPIO_PIN_7) == RESET)
        {
            value_R = 0;value_G = 0;value_B = 0;                //K4 按下,数据清零
            …;                                                  //其他代码同上
        }
    }
}
```

(2) 从设备设置地址代码。

```
void KeyService(void)
{
    if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
    {
        if(menu_One == 4 && menu_Two == 1 && flag == receive)
        {
            Group_rec +=1;
        }
        if(menu_One == 4 && menu_Two == 2 && flag == receive)
        {
            Add_rec +=1;
        }
    }
```

(3) 从设备接收数据代码。

```
void USART2_IRQHandler(void)
{
    if(usart_interrupt_flag_get(USART2,USART_INT_FLAG_RBNE) != RESET)
    {
        uint8_t data;
        data = usart_data_receive(USART2);
        if(data == 0x01)
            uart_data_idx = 0;
        uart_data[uart_data_idx++] = data;
        if(uart_data_idx == 5)
        {
            if(uart_data[0] == 0x01 || uart_data[0] == Add_rec)
            {
                if(uart_data[1] == 0x09)
                {
                    value_rec_R = uart_data[2];
                    value_rec_G = uart_data[3];
                    value_rec_B = uart_data[4];
                }
            }
            uart_data_idx = 0;
        }
        usart_interrupt_flag_clear(USART2,USART_INT_FLAG_RBNE);
    }
}
```

3.5.3　网络融合程序设计

1. 网络融合简介

网络融合是指多个网络协议可以相互通信,实现数据互通和信息融合,在复杂环境下对于灯光的调光调色做出更精准的控制。主控制器在网络融合实验中起到网关的功能,即实现不同协议的相互转换。本实验箱支持 RS485、WiFi 和 ZigBee 三种协议的融合。主控制器对这 3 个中控制器进行控制,将主控制器的按键信息发送给多个中控制器,中控制器设置为接收模式,中控制器根据接收的数据控制 RGB LED 灯的开关、亮度和颜色,能够通过多个中控制器上的 RGB LED 灯显示控制效果。

各种中控制器支持的通信协议原理、硬件设计均在前面相关章节中介绍,可参考。不同的通信协议拓扑结构和包格式均符合 OSI 标准。由于 DALI 和 DMX512 具有明确的协议标准,与其他协议不兼容,因此,多网络融合实验只包括 RS485、ZigBee 和 WiFi 三种协议。本开发平台制定的协议格式均是对用户数据的定义,如图 3-62 所示。

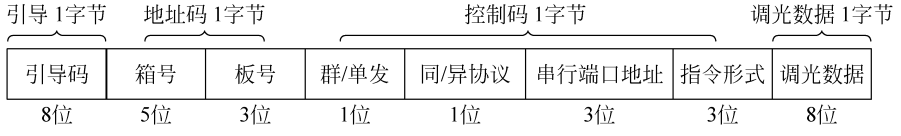

图 3-62　数据格式

2. 多网络融合实验网络构建

首先,通过杜邦线连接主控制器上的串口和 RS485、WiFi、ZigBee 这三个通信模块,实

现主控制器的 MCU 能够通过以上通信模块发送信息。

其次，连接主控制器和中控制器上对应的通信模块，使它们能够正常通信。RS485 是有线通信方式，需要用杜邦线连接主控制器和中控制器上的 RS485 模块，WiFi 和 ZigBee 是无线通信方式，需要单独配置，详情见“WiFi 模块配置方法”和“ZigBee 模块配置方法”。

最后，按下主控制器上的按键，发送控制信息，通过主控制器上的通信模块进行传输。各个中控制器设置为接收模式，通过相应的通信模块接收主控制器的控制信息。

多网络融合实验网络构建如图 3-63 所示，图中主控制器即实验箱内的总控制器，起到网关的功能，每个通信协议模块通过处理器的串口与主控制器相连，实现不同协议间的通信。

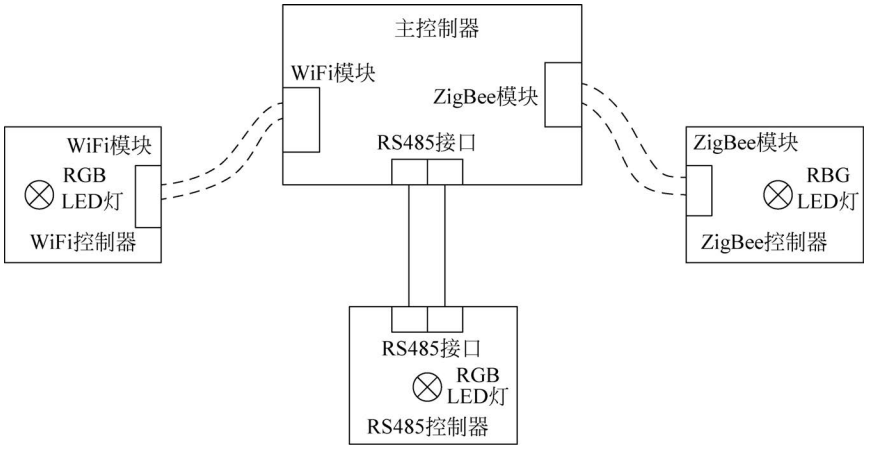

图 3-63　多网络融合实验网络构建

3. 网络融合控制流程

首先，通过按下主控制器的按键，使主控制器的 MCU 发送相应的 RGB LED 灯的颜色、亮度和开关的控制信息，经过 RS485、ZigBee 和 WiFi 三种通信方式传输。然后，多个中控制器设置为接收模式，中控制器通过对应的通信模块接收该控制信息，该控制信息经过中控制器 MCU 的解析及处理后，实现对板子上 RGB LED 灯的控制。

通过按键选择灯的颜色和亮度。网络融合控制流程如图 3-64 所示。

4. 程序关键代码

```
void key_data_send(uint8_t mode)
{
    int i = 0;
    if(gpio_input_bit_get(GPIOB,GPIO_PIN_4)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_5)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_6)==RESET
||gpio_input_bit_get(GPIOB,GPIO_PIN_7)==RESET))
    {
        delay_1ms(50);
        if(gpio_input_bit_get(GPIOB,GPIO_PIN_4) == RESET)
        {
            valueR += 10;                                    //K1 按下，红光数据加 10
            command[0] = 0x01;
            command[1] = 0x09;
```

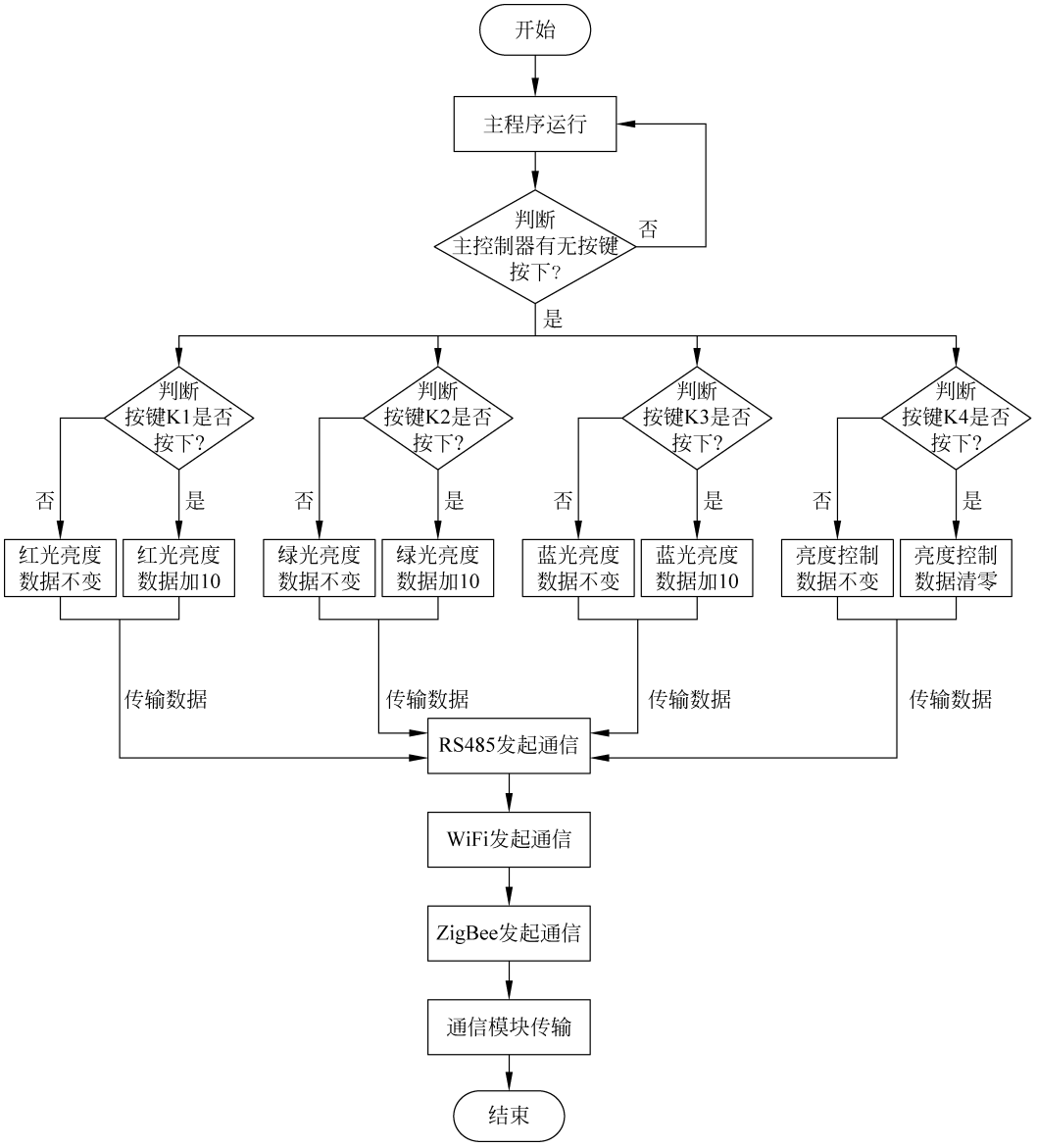

图 3-64　网络融合控制流程

```
command[2] = valueR;
command[3] = valueG;
command[4] = valueB;
command[5] = 0x00;
RS485_TX;
for(i = 0; i < 6; i++)
{
    usart_data_transmit(USART1,(uint8_t)command[i]);
    usart_data_transmit(USART0,(uint8_t)command[i]);
    usart_data_transmit(USART2,(uint8_t)command[i]);
    delay_1ms(1);
}
RS485_RX;
```

```
            }
            if(gpio_input_bit_get(GPIOB,GPIO_PIN_5) == RESET)
            {
                valueG += 10;                                    //K2 按下,绿光数据加 10
                … ;                                              //其他代码同上
            }
            else if(gpio_input_bit_get(GPIOB,GPIO_PIN_6) == RESET)
            {
                valueB += 10;                                    //K3 按下,蓝光数据加 10
                … ;                                              //其他代码同上
            }
            else if(gpio_input_bit_get(GPIOB,GPIO_PIN_7) == RESET)
            {
                valueR = 0;valueG = 0;valueB = 0;                //K4 按下,数据清零
                … ;                                              //其他代码同上
            }
        }
    }
```