百科荣创

BECSTRONG

EXPERIMENT GUIDE BOOK | 实验指导书

# 百科荣创（北京）科技发展有限公司
# 嵌入式技术应用开发实战

# 前 言

平台为车型移动机器人与实训沙盘结合的阶梯化新型教学载体，按照“项目引领、任务驱动”的教学模式，采用模块化、积木式设计理念，可根据不同教学实训需求进行功能模块单元组合完成不同开发难度、不同系统框架与功能实验实训系统的自主设计与搭建。平台支持选配与定制不同性能与开发难度的核心控制板(Arduino、MCS51、Cortex-M0/M3/M4/A7、FPGA、DSP、MSP430 等)，系统标准功能单元支持汽车 CAN 总线通信，可完成真实汽车控制系统相关技术技能教学与实训。

平台可选配与定制多种传感器、执行器、自动识别及创新应用模块完成功能扩展。支持选配全系列物联网通信单元完成无线组网与智能互联，可接入多种云平台（百科荣创物联网云平台、百度天工、华为云、微信云等）完成云端数据交互与互联控制。平台支持视觉（图像分类、目标检测、图像分割）和语音两大典型 AI 方向应用开发，支持边缘智能和云端智能两类 AI 技术路线，可实现智能驾驶与车联网应用。本平台完全满足电子信息、嵌入式、物联网、人工智能、移动互联、机器人等电子信息大类专业核心课程日常教学、实践实训及竞赛创新使用。

本实验指导书主要对小车核心板 STM32F4 基础实验进行讲解，针对 STM32 核心板可实现各种基础驱动功能，如通用功能输入输出(GPIO)、中断、串行通信等。实验通过阐述其结构及应用设计方法，可以使学生逐步了解单片机编程的基础知识，并掌握单片机的原理和应用。同时，为学生深入学习和应用提供了坚实的基础。实验内容注重过程，从知识点出发，逐步深入到程序编写，从而积累编程经验，提升技能水平。

本实验指导书虽然在主观上力求谨慎从事，但限于时间和编者的学识、经验，疏漏之处，在所难免。恳请广大读者不吝赐教以便今后修改提高。

为了给您提供更加优质的服务，公司设有 400 开头的免费电话，在使用过程中，如果您有任何问题需要咨询或者需要服务，请与本公司相关部门联系，我们定当竭诚为您服务。服务电话：400-067-7899。

欢迎您在使用的过程中为我们提出宝贵的意见和建议。

# 目 录

第 1 章 嵌入式智能车开发平台介绍---------- 1
1.1 平台整体结构介绍---------- 1
1.2 平台硬件资源概述---------- 3
第 2 章 嵌入式开发与调试---------- 9
2.1 开发环境搭建---------- 9
2.2 新建工程管理----------16
2.3 程序下载调试----------33
第 3 章 嵌入式智能车功能介绍----------37
3.1 设备简介----------37
3.2 设备功能介绍----------39
3.3 设备基本操作----------47
第 4 章 嵌入式微控制器基础实验----------49
4.1 LED 灯控制实验----------49
4.2 按键输入实验----------71
4.3 蜂鸣器报警实验----------81
4.4 外部中断按键输入实验----------88
4.5 串口数据发送与接收实验---------- 108
4.6 DMA 数据传输实验---------- 128
4.7 RTC 实时时钟实验---------- 148
4.8 LED 交替闪烁实验---------- 173
4.9 呼吸灯实验---------- 184
4.10 电池电量测量实验---------- 197
4.11 内部温度传感器实验---------- 215
4.12 输出三角波实验---------- 224
4.13 SPI 总线驱动 OLED 显示实验---------- 235
4.14 CAN 总线通信实验---------- 250
4.15 SD 卡数据读写实验---------- 270

4.16 SD 卡文件系统实验 ---------- 294
4.17 FSMC-LCD 显示实验 ---------- 300
第 5 章 常见问题解决方案 ---------- 328
5.1 常见下载问题及解决方案 ---------- 328

# 第1章 嵌入式智能车开发平台介绍

## 1.1 平台整体结构介绍

### 1.1.1 新型实验实训操作台

平台采用磁吸抽屉式固定结构，集实训、收纳于一体，便于存放和组织各种模块和组件，方便读者快速更换和调整硬件资源。此外，平台还配备了多种实用接口和工具，包括 5V/12V 电源接口、逻辑分析仪、SWD 下载接口、串口调试接口和 USB 接口，以实现与外部设备的通信和文件烧录。此外，嵌入式平台还提供电压电流表，方便读者进行电路测量和调试，提供高精度的测试功能。

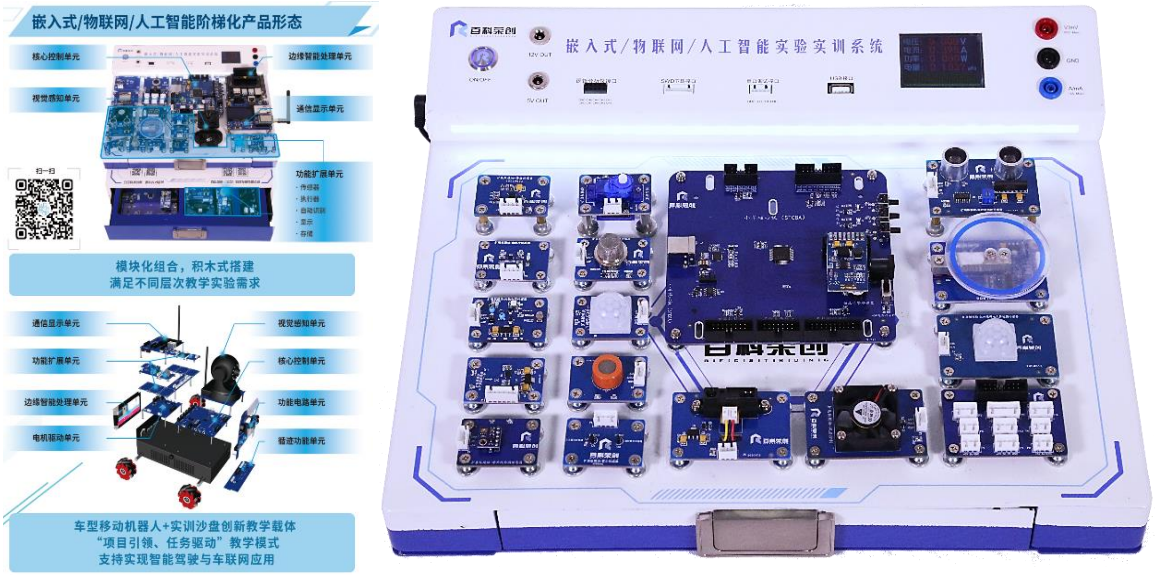



图 1.1.1 平台示意图

平台支持不同性能和开发难度的核心控制板（如 MCS-51 和 Arm Cortex-M4），同时与传感器、执行器、自动识别和通信单元等创新应用模块配合使用。采用模块化、积木式设计理念，可根据不同的教学实训需求，通过灵活组合功能模块单元来自主设计和搭建各种系统框架和功能实验实训系统。

### 1.1.2 嵌入式开发应用场景

图 1.1.2 平台开发应用场景示意图

平台不仅支持嵌入式基础实验的学习，还可以根据需求自由组合扩展模块，以构建多样化的综合应用场景。此外，平台还提供了多种显示单元供选择，用于实现数据展示和人机交互。这样的平台设计能够提供广泛的灵活性和适用性，满足用户的不同需求与创意。

### 1.1.3 嵌入式智能车应用场景

图 1.1.3 智能车示意图

通过组装智能小车可完成模拟真实汽车控制系统相关技术技能教学与实训；支持物联网通信，可完成无线组网与智能互联，可接入多种云平台完成云端数据交互与互联控制。

# 1.2 平台硬件资源概述

## 1.2.1 核心控制单元

**Cortex-M4 核心控制单元：**

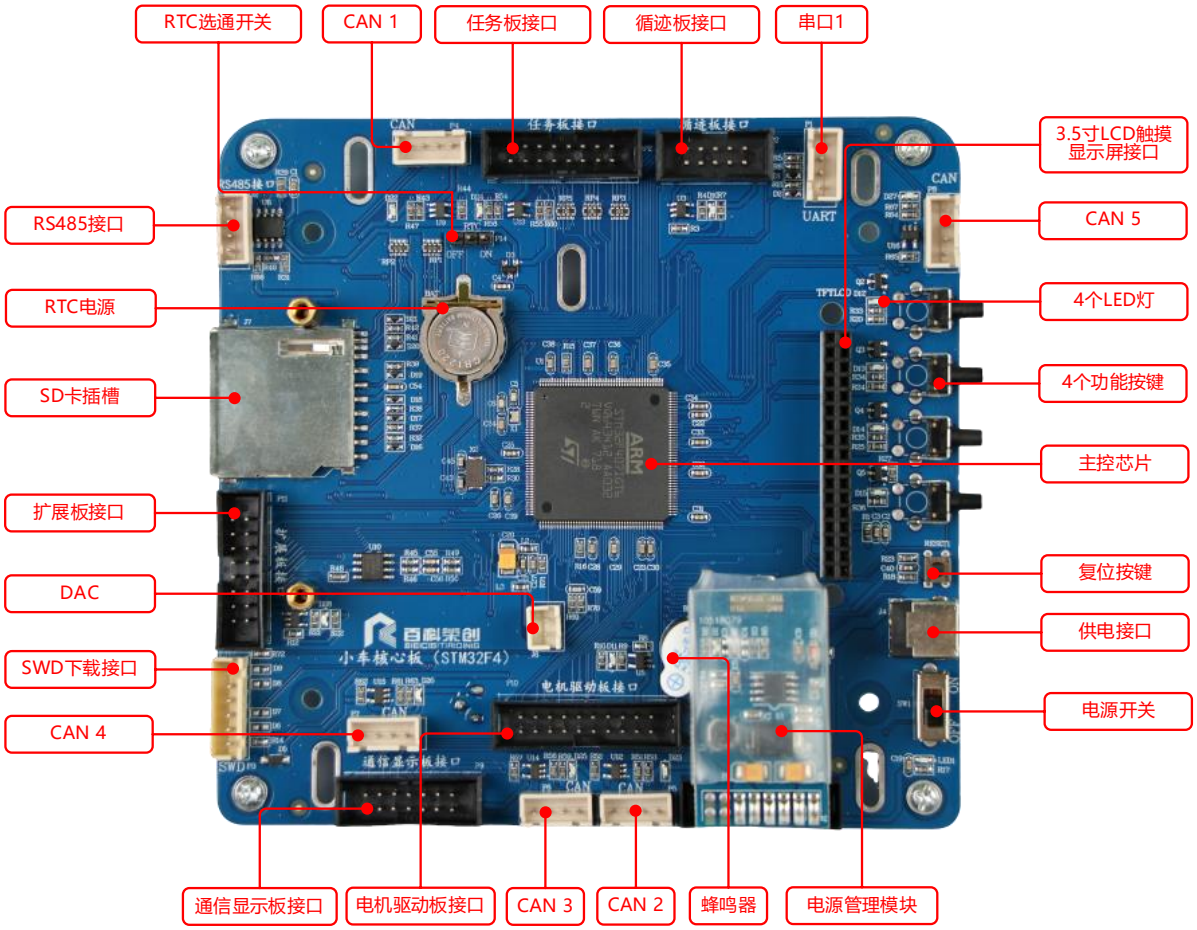



图 1.2.1 核心板示意图

**1、核心处理器：**

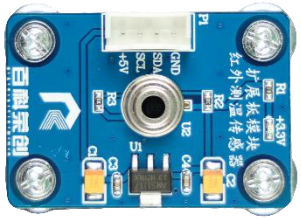
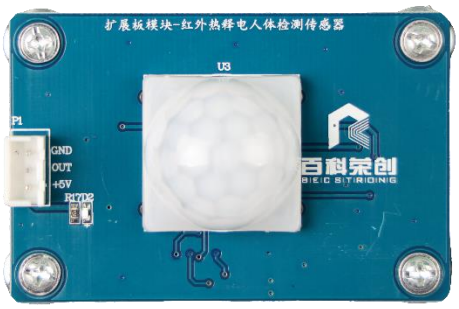
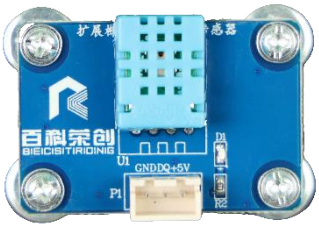
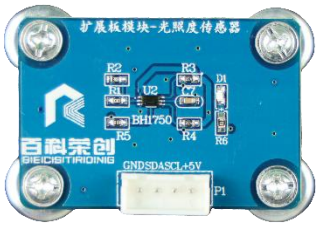
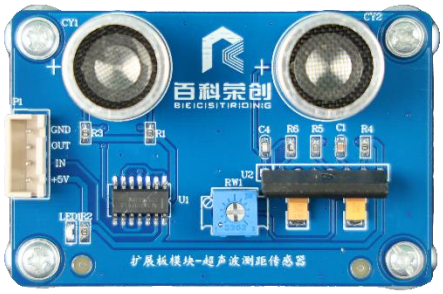
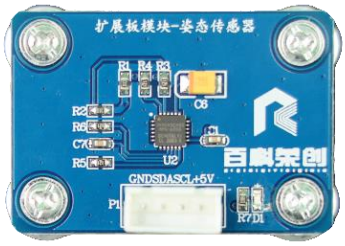
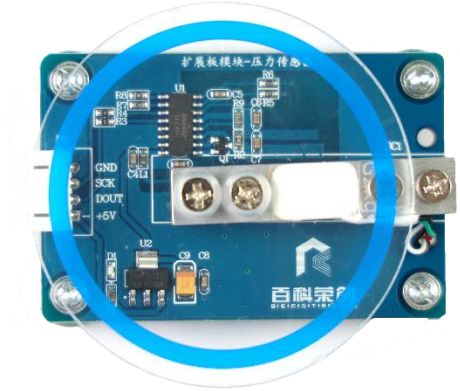
- 芯片型号：STM32F407IGT6 Arm Cortex-M4 内核系列核心处理器
- 主频可达 168MHz
- 内置 1MB Flash
- 内置 192KB SRAM

**2、板载资源及扩展接口：**

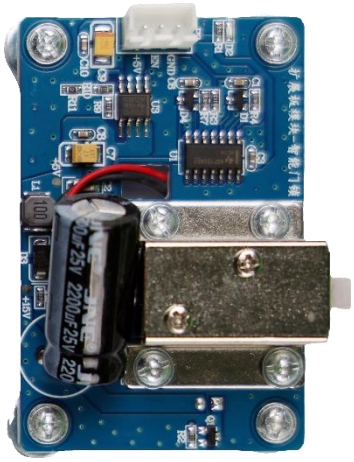
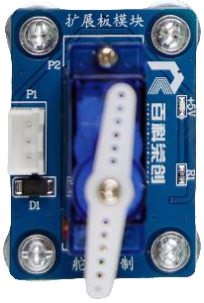
- 1 路 12V 供电接口
- 1 路硬件复位按键

- 1 路电源管理模块接口
- 4 个功能按键
- 4 个 LED 灯
- 1 路任务板接口
- 1 路循迹板接口
- 1 路通信显示板接口
- 1 路扩展板接口
- 5 路 CAN 总线接口
- 1 个 SD 卡插槽
- 1 路 3.5 寸 TFT 显示屏接口
- 1 路 RS485 通信接口
- 1 路 4Pin 串口
- 1 路 DAC 接口

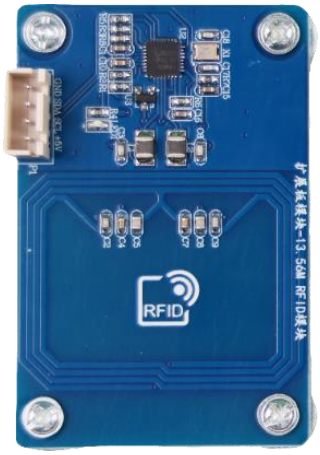
## 1.2.2 智能感知单元

| 霍尔传感器单元 | 红外测温传感器单元 | 红外热释电人体检测传感器单元 |
|---|---|---|
| |  |  |
| 温湿度传感器单元 | 光照度传感器单元 | 超声波传感器单元 |
|  |  |  |
| 姿态传感器单元 | 烟雾传感器单元 | 压力传感器单元 |
|  | |  |

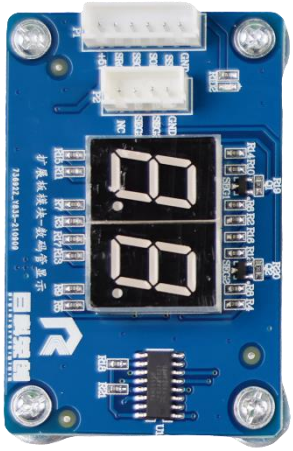
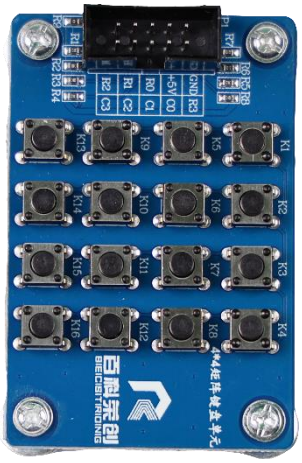
### 1.2.3 执行控制单元

| 风扇单元 | 智能门锁单元 |
| --- | --- |
| |  |
| 舵机单元 | 可调光 LED 显示单元 |
|   | |

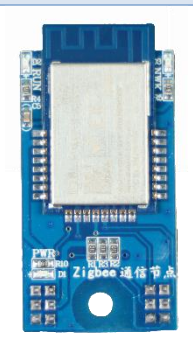
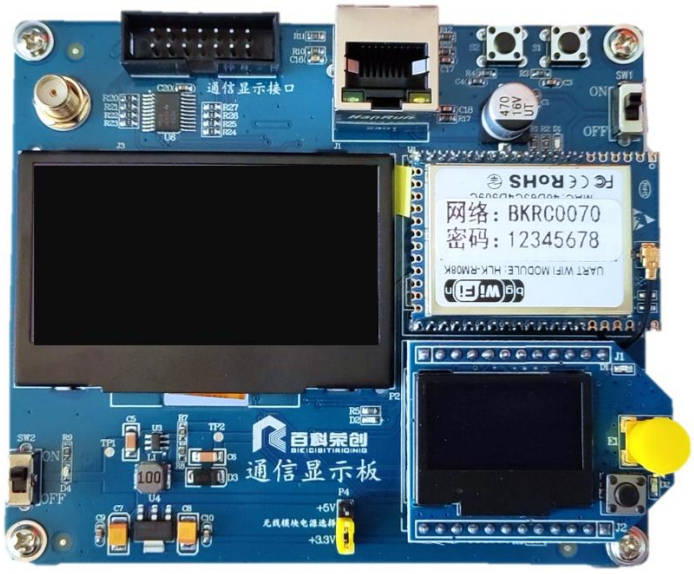
### 1.2.4 自动识别单元

| 高频 RFID 识别单元 | 智能语音识别单元 | 指纹识别单元 |
| --- | --- | --- |
|   | | |

## 1.2.5 人机交互单元

| 数码管显示单元 | 点阵显示单元 | 矩阵键盘单元 |
| --- | --- | --- |
|  | |   |
| 3.5 寸液晶屏单元 | | |
| | | |

## 1.2.6 通信应用单元

| WiFi 无线<br>通信单元 | ZigBee 无线<br>通信单元 |
|---|---|
| |  |
| 通信显示单元 | |
|  | |

# 第2章 嵌入式开发与调试

本章重点介绍嵌入式开发环境搭建、工程管理和程序调试的基本方法，旨在帮助读者快速掌握本实验产品的开发流程。平台支持多种类型的核心板接入，本实验指导书主要对核心板（STM32F4）开发进行讲解。

## 2.1 开发环境搭建

STM32 核心板开发支持多种开发环境（Keil MDK、STM32CubeIDE、IAR 等），这里主要以 Keil MDK 开发环境为例进行讲解。

### 2.1.1 开发工具下载

**1. KEIL MDK 下载**

本小节主要介绍如何下载 Keil MDK 和相关芯片支持包。读者可以在 Keil 官方网站上找到 Keil MDK 的下载链接。

其官方下载地址：https://www.keil.com/update/rvmdk.asp

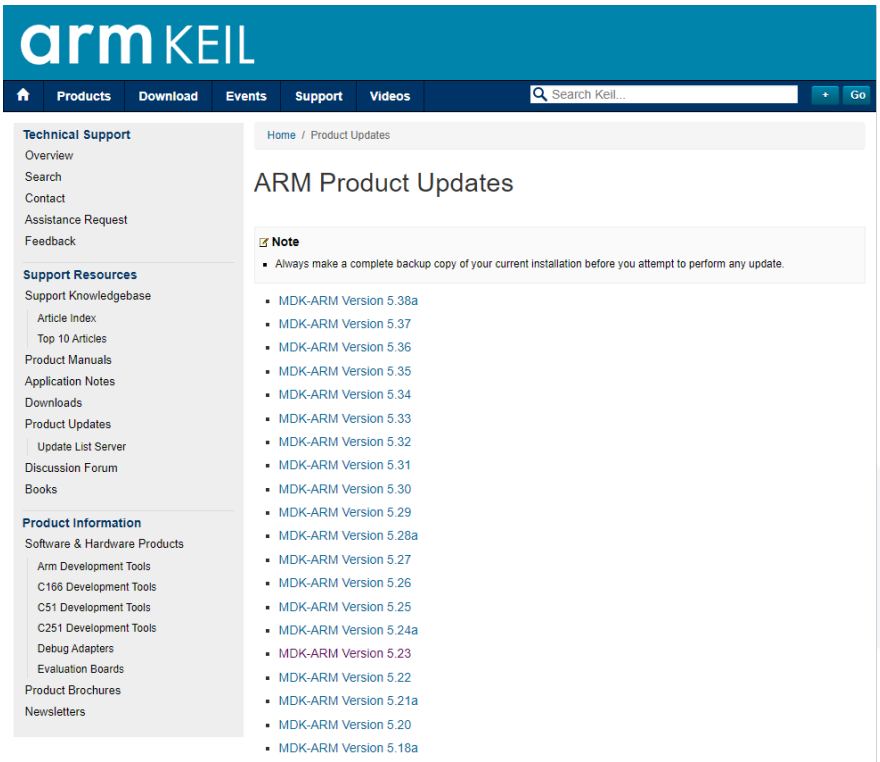



图 2.1.1 下载界面

进入下载界面，可以看到如上图所示的 MDK 版本列表。

下面以 MDK-ARM 5.23 版本为例，点击选中要下载的版本后进入下载界面，如下

图所示，填写相关信息并点击提交。

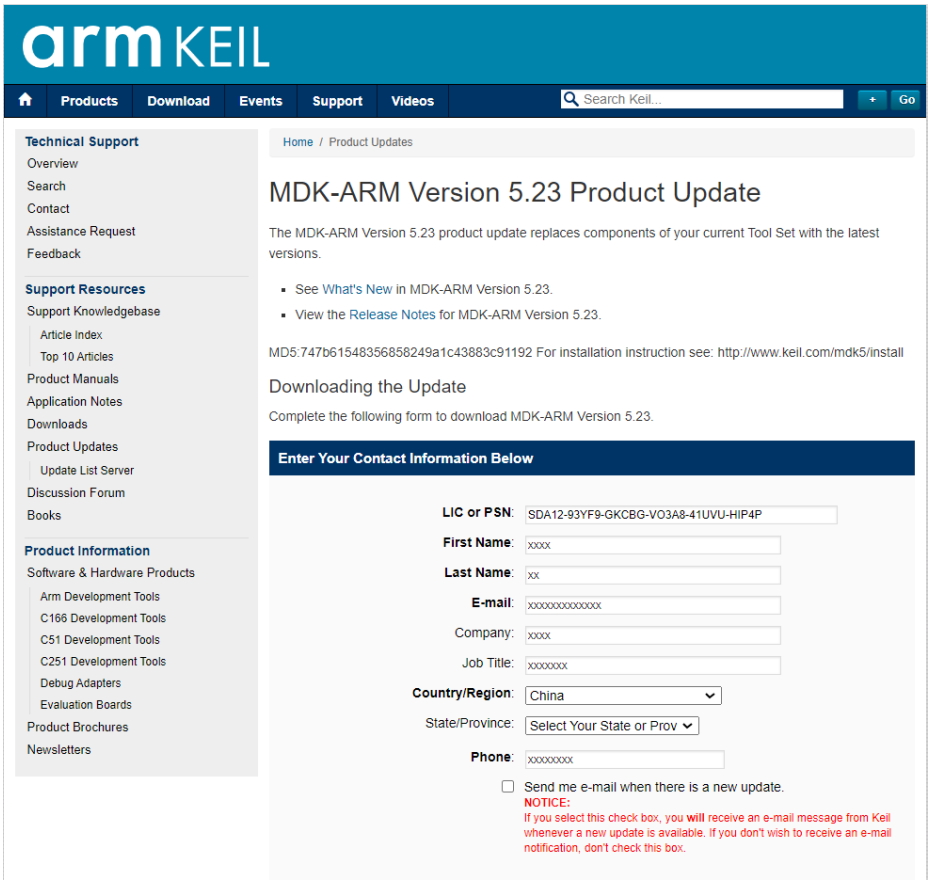



图 2.1.2 输入正确信息

将相关的信息正确填写后，点击如下图所示的红色方框内的网页链接，即可开始下载 MDK523.EXE 文件。

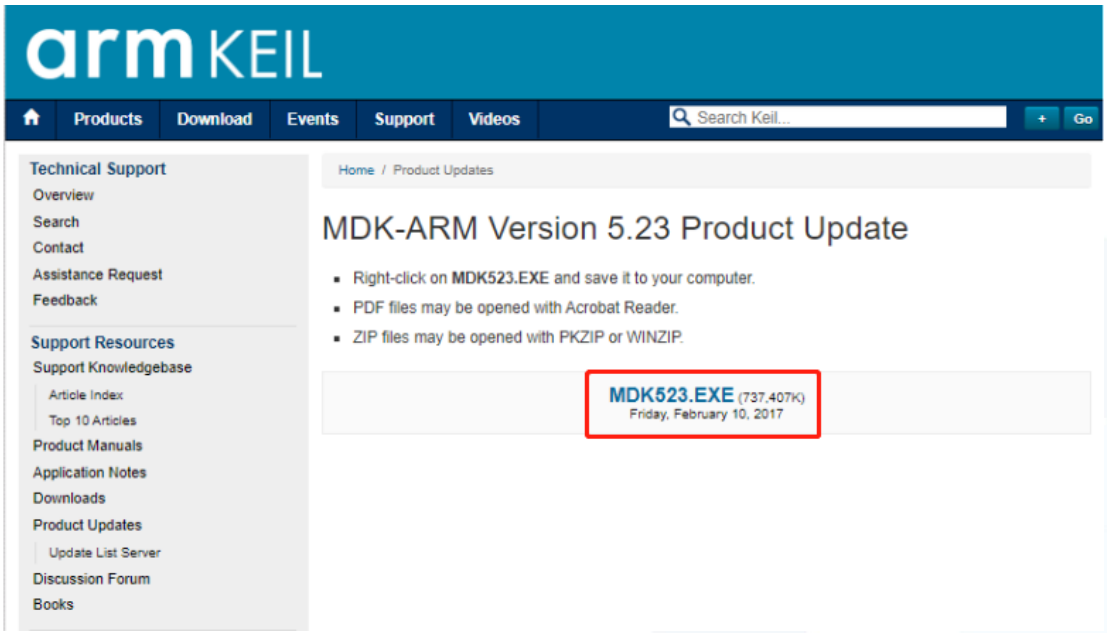



图 2.1.3 点击下载

读者还可以通过直接访问下载链接进行下载：https://armkeil.blob.core.windows.net/eval/MDK523.EXE（更改后缀 MDK523→MDK536 可进行其他版本下载）。

**2. 芯片器件库下载**

在进行程序开发时还需安装相关 Keil 芯片 DFP 包，可通过 Keil 官网进行下载，首先进入官网：https://www.keil.arm.com/，然后点击页面上方的 CMSIS Packs。

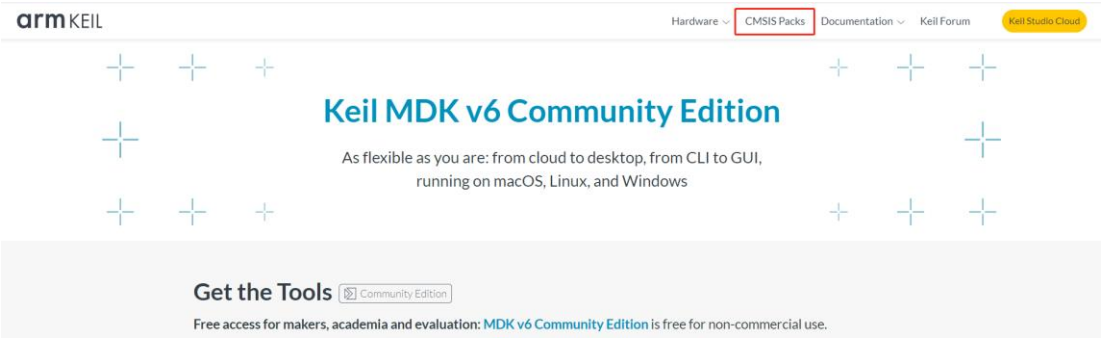



图 2.1.4 arm Keil

在搜索栏输入 STM32F4，点击回车后可选择 STM32F4xx_DFP 最新版进行下载，或点击进入进行历史版本下载。也可根据需要搜索安装其他芯片系列 DFP 文件。

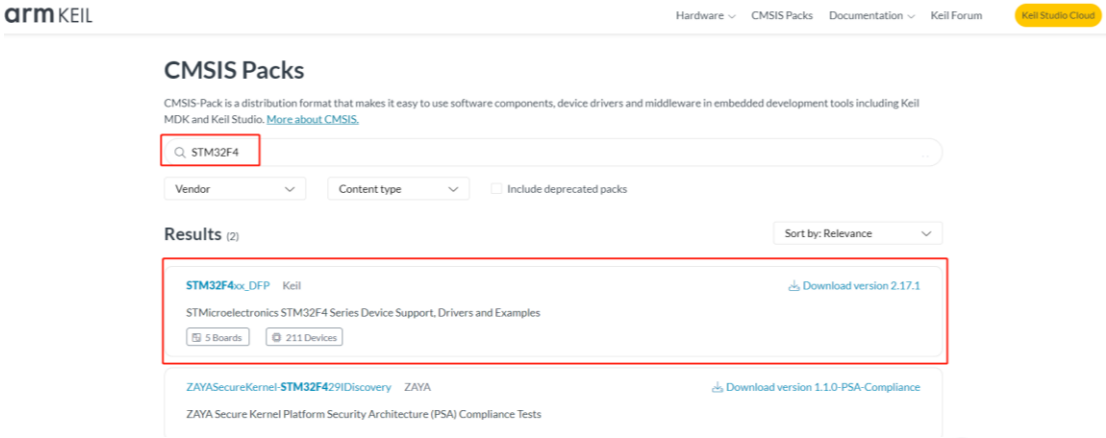



图 2.1.5 DFP 包下载

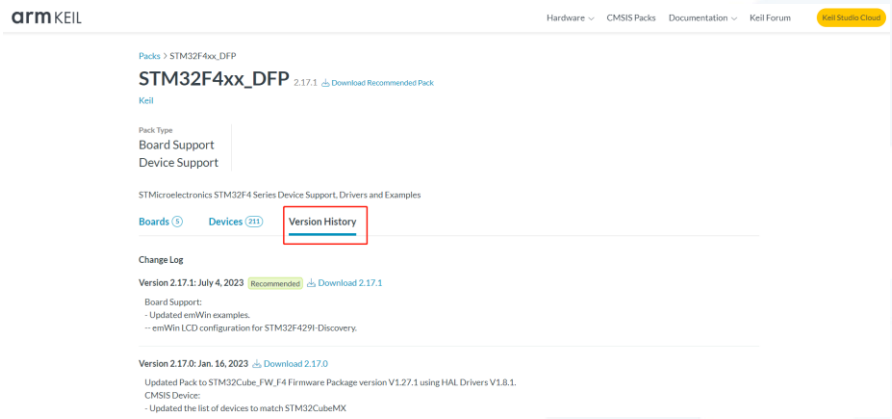



图 2.1.6 历史版本下载

## 2.1.2 软件安装说明

1. 安装路径必须是英文路径。

2. 安装目录不能与 51 的 KEIL 或是 KEIL4 相同。

3. Keil5 安装后，还需添加对应芯片的器件库。

4. 关闭防病毒软件，在安装过程中，安全软件可能会干扰安装过程。

## 2.1.3 软件安装步骤

1．双击图标进行安装。

图 2.1.7 软件图标

2．进入安装界面——点击 Next（下一步）。

图 2.1.8 安装界面

3．选中同意软件使用条约，点击 Next（下一步）。

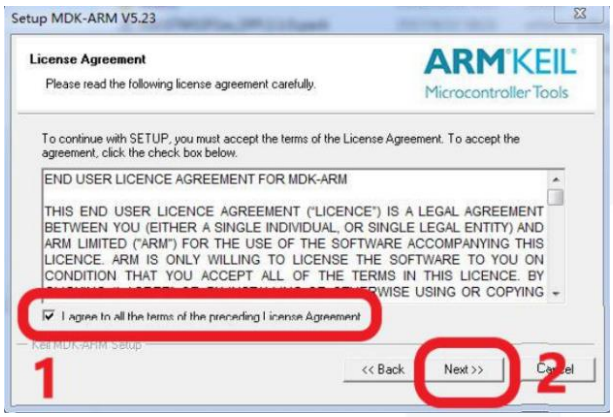



图 2.1.9 使用条约

4．选择安装路径（以 D 盘 Keil_V5 为例）——点击 Next（下一步）。

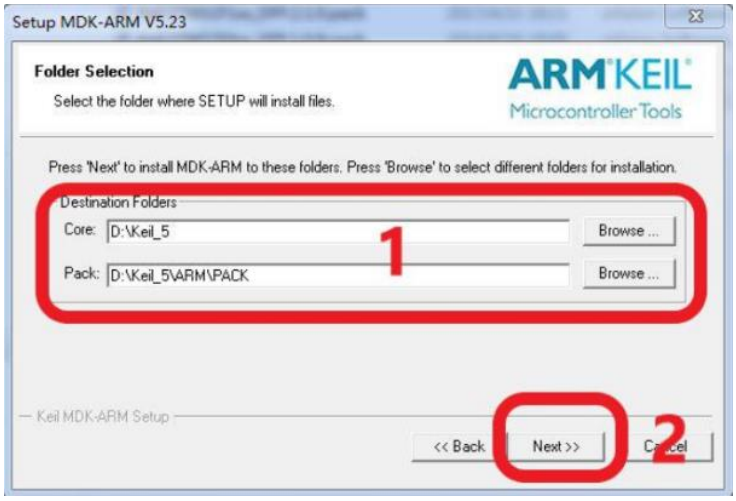



图 2.1.10 安装路径

5．填写用户名(First name)与邮箱(E-Mail)（任意填写）——点击 Next（下一步）。

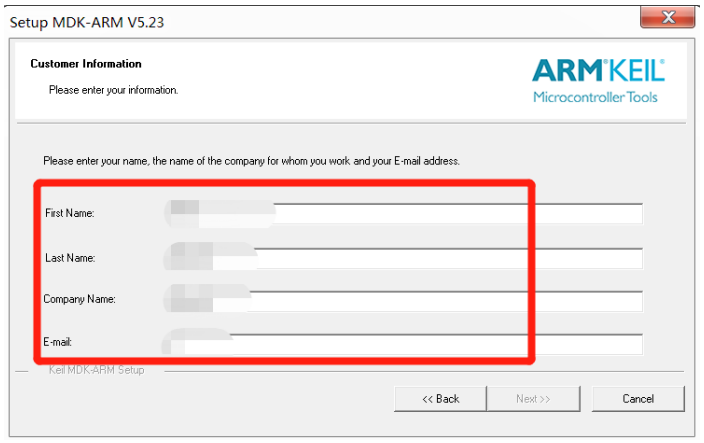



图 2.1.11 填写信息

6．正在安装——等待安装进度条完成。

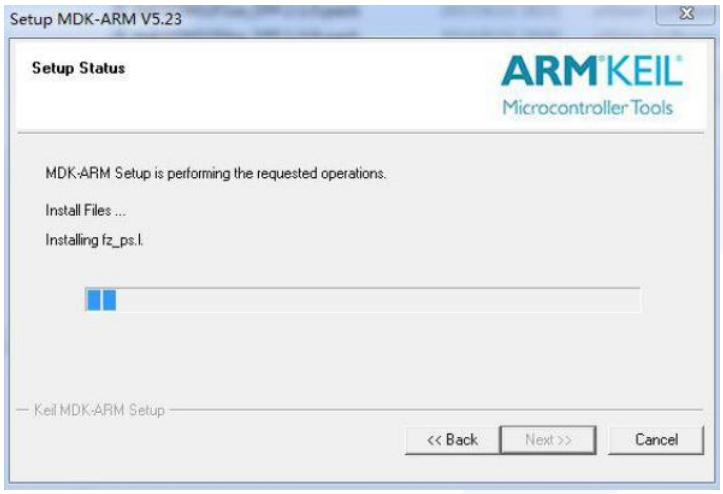



图 2.1.12 等待安装完成

7．去掉对勾，安装完成——点击 Finish（完成）。

图 2.1.13 安装完成

8．添加器件库安装包——双击 Keil.STM32F4xx_DFP.1.0.8.pack 安装包（STM32F4 系列，根据使用的芯片型号，添加对应的器件库包，因我们在这里所使用的是 STM32F407IGT6 型号的芯片，故选择添加器件库对应 F4 系列安装包。（若使用其他系列芯片，则需要添加其对应区间的库包；可安装不同型号的器件库安装包。）

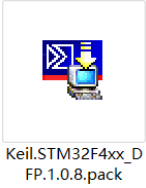



图 2.1.14 pack 包

9．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）点击 Next。

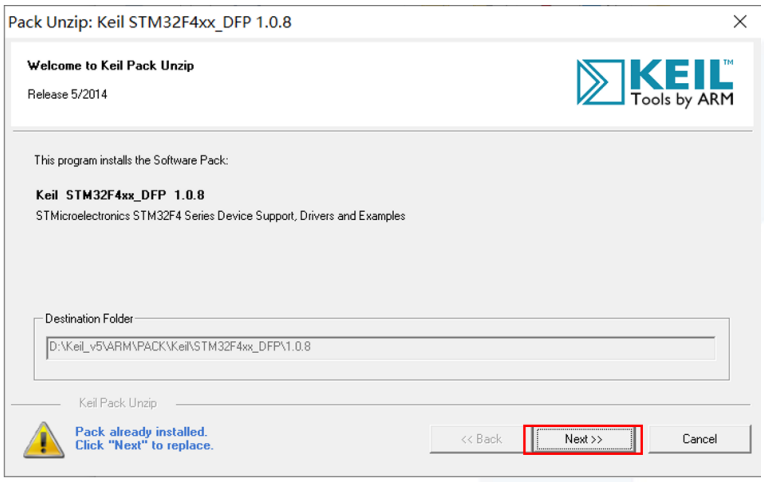



图 2.1.15 添加器件库包

10．添加器件库安装包进度条（等待进度条完成）。

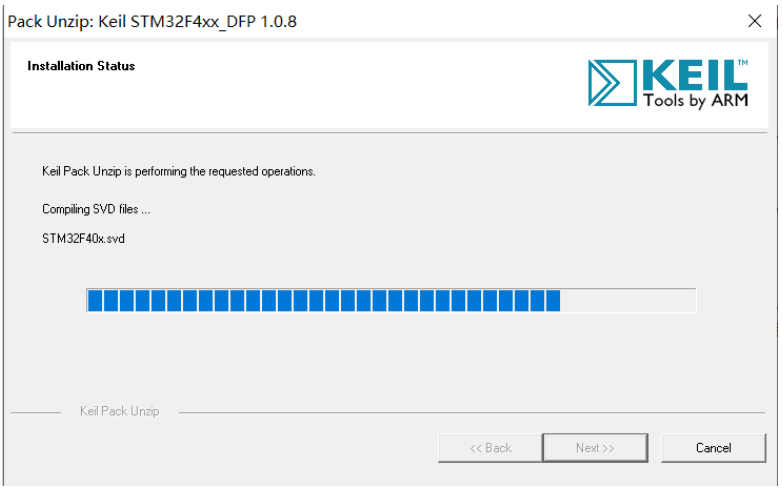



图 2.1.16 等待安装完成

11．如下图所示，添加成功——点击 Finish（完成）。

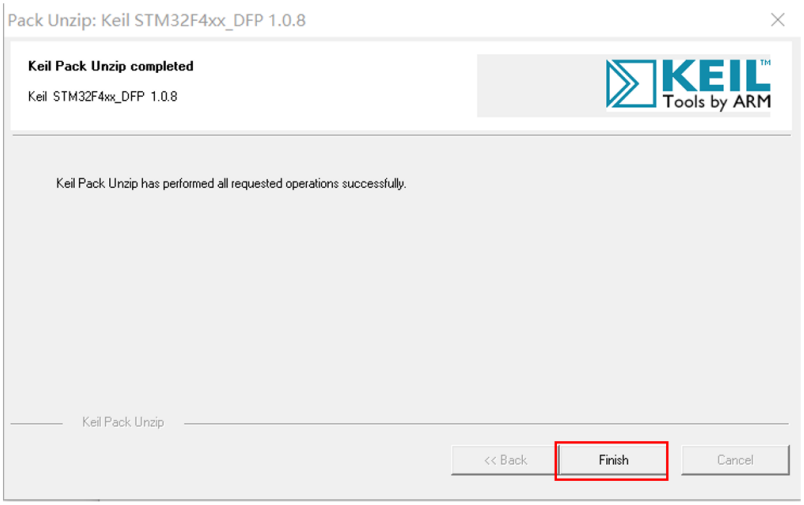



图 2.1.17 添加成功

12．至此软件安装完成。

## 2.2 新建工程管理

本节的主要内容是介绍如何新建和管理 MDK Keil 工程。新建工程的步骤可能较为复杂，建议创建一个工程模板，以便以后的程序开发可以基于该模板进行。

### 2.2.1 工程文件目录

工程文件管理有助于有效组织和管理项目中的各种文件和文档。通过建立清晰的目录结构和命名规范，可以使文件易于分类、检索和访问。

在提供的 MDK 标准库函数开发案例中均以以下目录进行工程管理。

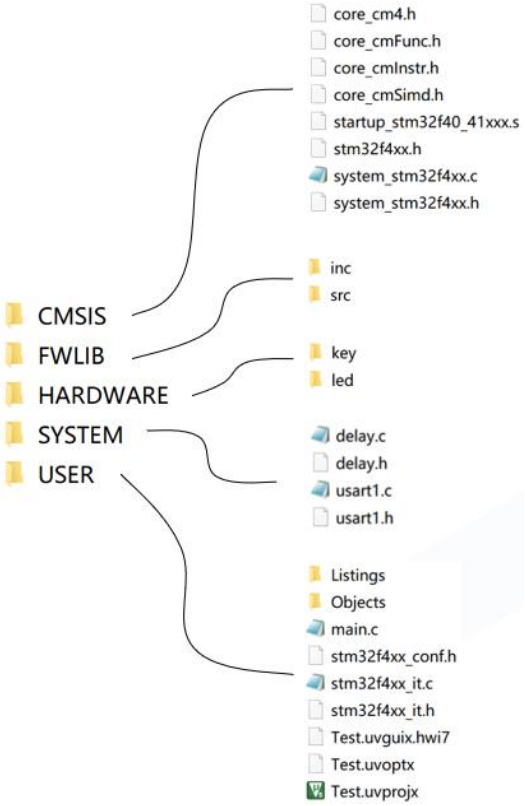



图 2.2.1 KEIL 工程文件结构示意图

工程中文件夹的功能分别为：

1. CMSIS：用来存放库为用户自带的启动文件和一些位于 CMSIS 层的文件。

2. FWLIB：用来存放 STM32 库里面的 inc 和 src 这两个文件夹，这两个文件夹包含了芯片上的所有驱动，这两个文件夹中的文件一般不需要修改。

3. HARDWARE：用来存放连接芯片与其外部设备的驱动程序。

4. SYSTEM：用来存放延时、系统等函数文件。

5. USER：用来存放工程文件和用户层代码，包括主函数 main。

表 2.2.1 各文件夹作用及文件清单

| 名称 | 作用 |
|---|---|
| CMSIS | 内核功能的定义：core_cm4.h<br>内核核心功能接口头文件：ore_cmFunc.h<br>包含内核核心专用指令：core_cmInstr.h<br>包含与编译器相关的处理：core_cmSimd.h<br>启动文件：startup_stm32f40_41xxx.s<br>头文件：stm32f4xx.h<br>系统源文件：system_stm32f4xx.c<br>系统.h 文件：system_stm32f4xx.h |
| FWLIB | 库函数对应头文件：inc<br>库函数源文件：src |
| HARDWARE | 用户编写的驱动文件，暂时为空。例如：<br>LED 灯驱动文件：led.c、led.h<br>按键驱动文件：key.c、key.h |
| SYSTEM | delay 文件“delay.c、delay.h”、usart 文件“usart1.c、usart1.h” |
| USER | 工程文件“XXX”、主源文件“main.c”、相关中断源文件“stm32f4xx_it.c”、相关中断.h 文件“stm32f4xx_it.h” |

## 2.2.2 新建工程文件

1．在工程新建前，先创建一个工程文件夹（STM32F4xx 库函数工程模板），根据工程管理目录创建 5 个子文件。

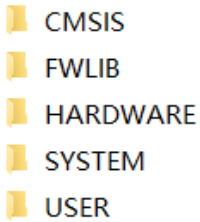



图 2.2.2 创建工程目录

2．打开 keil 软件，点击菜单栏 Project→New uvision Project，如果有其他工程，先点击 Project→Close Project，然后将目录定位到刚才建立的文件夹（STM32F4xx 库函数工程模板）中的子文件夹 USER 中，输入新建工程名称这里以 Test 为例，点击保存。

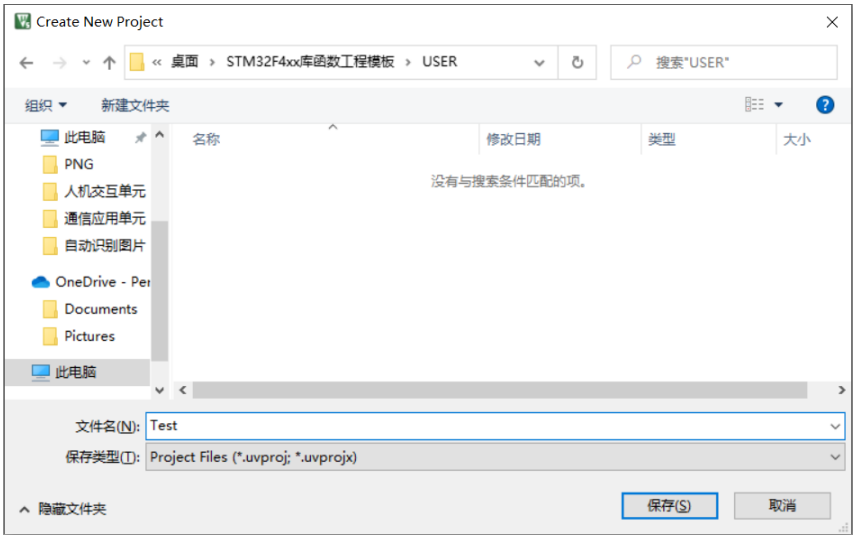



图 2.2.3 新建工程

3．接下来会出现一个选择 Device 的界面，以 STM32F407IGT6 芯片为例，这里选择 STMicroelectronics→STM32F4 Series→STM32F407→STM32F407IG。

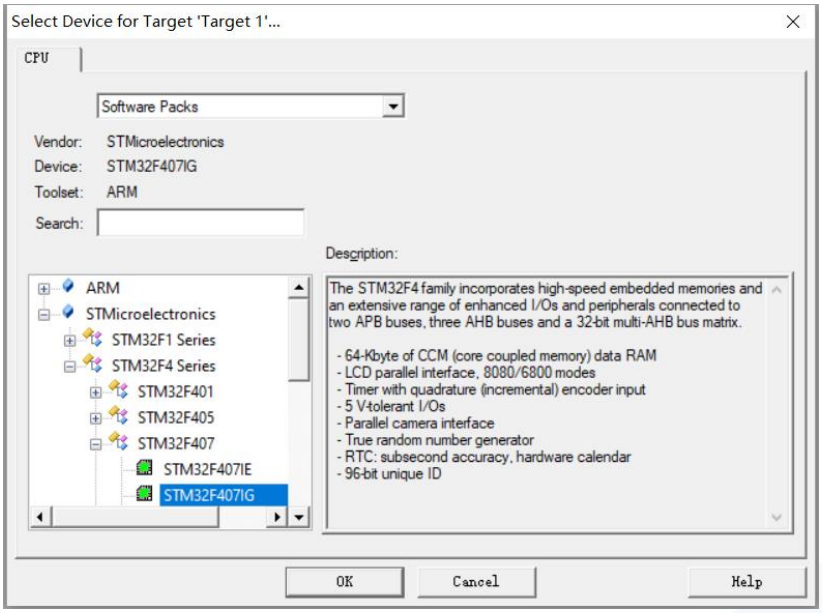



图 2.2.4 选择芯片型号

若未找到 STM32F4 Series 可能是未安装相关器件库，可参考开发环境搭建进行对应 Keil.STM32F4xx_DFP 包安装。

4．点击 OK，出现 Software Component 环境管理界面，可进行相关软件包添加，这里暂不进行更改，可直接点击 cancel。

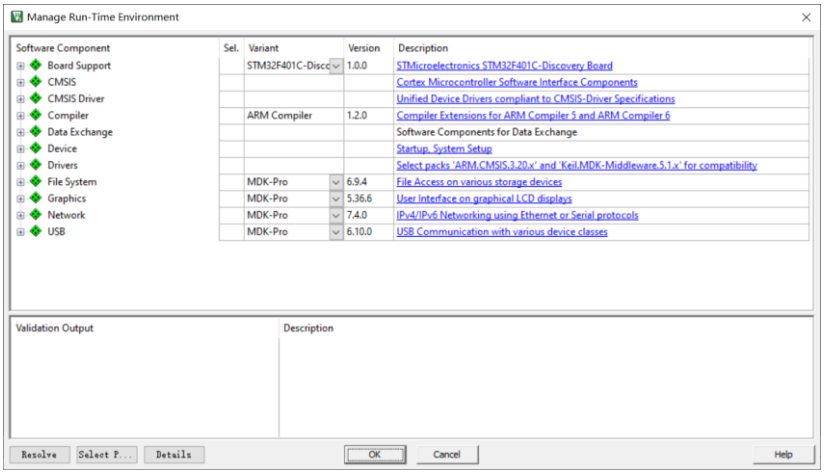


图 2.2.5 环境管理界面

5．接着进行固件包移植，将所需文件添加至工程管理子目录中，可在 ST 官网下载 STM32F4 开发固件包，或在提供的开发环境资料中的固件库进行查找。

ST 官方下载地址：[https://www.st.com/zh/embedded-software/stsw-stm32065.html](https://www.st.com/zh/embedded-software/stsw-stm32065.html)

图 2.2.6 下载固件包

本次新建工程均以 STM32F4xx_DSP_StdPeriph_Lib_V1.8.0 固件库为例。

（1）CMSIS 文件

该文件夹主要用来存放固件库为用户自带的启动文件和软件接口文件。

CMSIS
微控制器软件
接口标准文件

| | |
|---|---|
| core_cm4.h | 内核功能的定义 |
| core_cmFunc.h | 内核核心功能接口头文件 |
| core_cmInstr.h | 包含内核核心专用指令 |
| core_cmSimd.h | 包含与编译器相关的处理 |
| startup_stm32f40_41xxx.s | 启动文件 |
| stm32f4xx.h | 头文件 |
| system_stm32f4xx.c | 系统源文件 |
| system_stm32f4xx.h | 系统.h 文件 |

各个文件在固件库中位置如下：

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\CMSIS\Include

- core_cm4.h
- core_cmFunc.h
- core_cmInstr.h
- core_cmSimd.h

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\CMSIS\Device\ST\STM32F4xx\Source\Templates\arm

- startup_stm32f40_41xxx.s

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Templates

- system_stm32f4xx.c

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\CMSIS\Device\ST\STM32F4xx\Include

- stm32f4xx.h
- system_stm32f4xx.h

（2）FWLIB 文件

该文件夹主要用来存放 STM32 库中的芯片驱动文件 inc 和 src 文件夹。

- FWLIB
  - inc
    - misc.h
    - stm32f4xx_adc.h　　库函数对应头文件
    - stm32f4xx_can.h
    - ……
  - src
    - misc.c
    - stm32f4xx_adc.c　　库函数源文件
    - stm32f4xx_can.c
    - ……

在固件库中位置如下：

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Libraries\STM32F4xx_StdPeriph_Driver

- inc
- src

（3）HARDWARE 文件

该文件夹为硬件文件夹，用来存放连接芯片与其外部设备的驱动程序，通常存放用户新建的相关驱动文件，这里可以先不做文件添加。

（4）SYSTEM 文件

该文件为系统文件夹，通常用来存放延时、串口等系统函数文件，这里可以先不做文件添加。

（5）USER 文件

该文件夹为用户文件夹，用来存放工程文件和用户层代码，包括主函数 main 等。

USER 用户文件

| | |
|---|---|
| XXX | 工程文件 |
| main.c | 主源文件 |
| stm32f4xx_it.c | 相关中断源文件 |
| stm32f4xx_it.h | 相关中断.h 文件 |
| stm32f4xx_conf.h | 标准外设库头文件 |

由于新建 keil 工程时选中了 USER 文件夹，所以软件工程文件以及生成，还会有其他系统文件过程文件等，这里我们主要讲解需要从固件库中添加文件。

在固件库中位置如下：

STM32F4xx_DSP_StdPeriph_Lib_V1.8.0\Project\STM32F4xx_StdPeriph_Templates

- stm32f4xx_it.c
- stm32f4xx_it.h
- stm32f4xx_conf.h

main.c 文件可稍后新建一个简单测速函数。

6．工程文件导入，固件库文件添加成功后，还需将相关文件导入工程中，首先打开 Keil 工程文件，可直接点击 USER 文件夹中的 Test.uvprojx 文件打开工程。

右键点击 Target1，选择 Manage Project Items 进入工程管理。

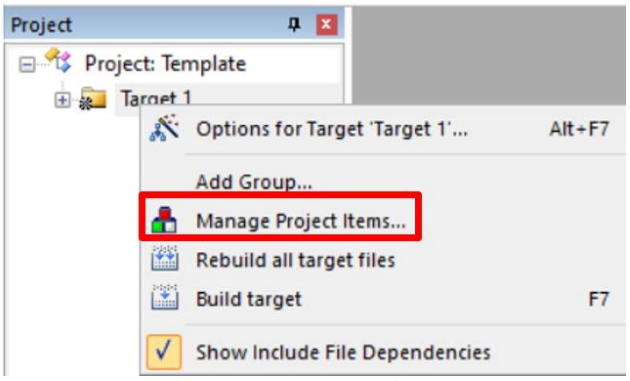



图 2.2.7 进入工程管理

7．在工程管理界面可双击更改文件名称，将 Source Group 1 更改为 CMSIS，点击 New 新建目录文件。

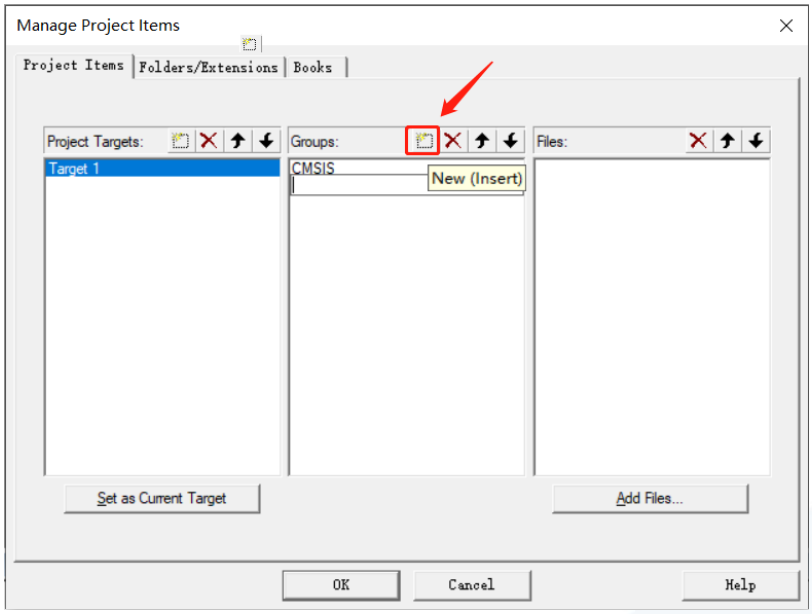



图 2.2.8 新建目录

按照顺序依次添加其他文件目录 CMSIS、FWLIB、HARDWARE、SYSTEM、USER。

创建完成后如下图所示，接着选中 CMSIS 目录点击 Add Files…进行相关文件添加。

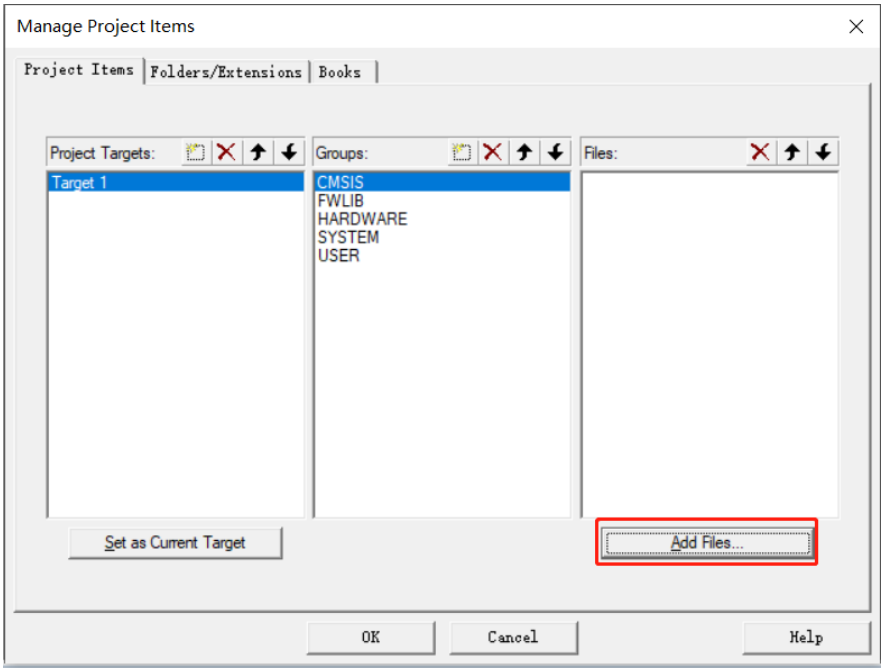



图 2.2.9 添加文件

8．首先找到工程文件目录 CMSIS 文件夹，选中 system_stm32f4xx.c 文件点击 Add 进行添加，接着还需要添加 startup_stm32f40_41xxx.s 启动文件，首先更改文件类型为 All files(*.*)，选中需要添加的文件，点击 Add 进行添加。

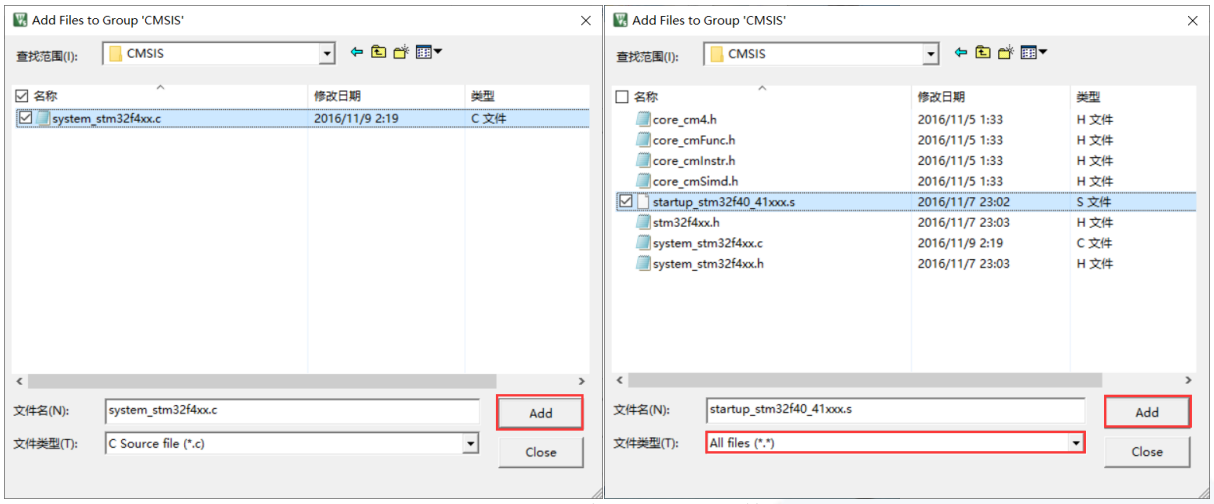



图 2.2.10 添加 CMSIS 文件

9．接着按顺序添加剩下目录文件（注意：添加完成后不要直接关闭，点击 OK 才会保存），选中 FWLIB 目录点击 Add Files…进行相关文件添加，首先找到 FWLIB 目录，打开 src 文件夹，可使用 Ctrl+A 全选，点击 Add 添加全部文件。

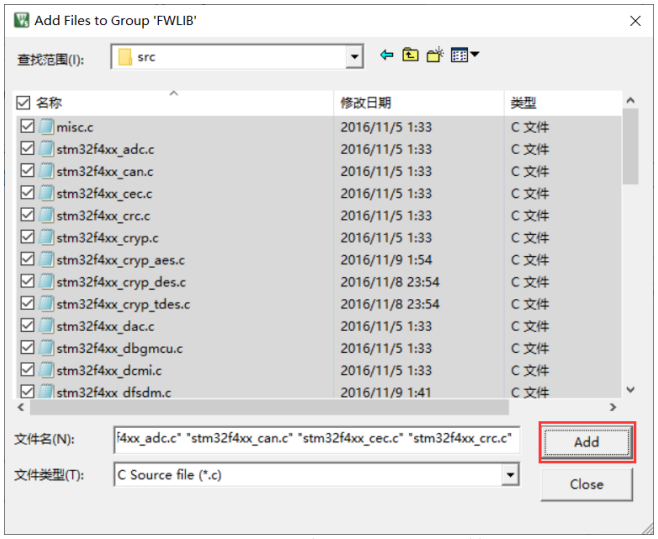


图 2.2.11 添加 FWLIB 文件

10．stm32f4xx_fmc.c 比较特殊。这个文件是 STM32F42 和 STM32F43 系列才用到，所以我们这里要把它删掉，选中点击删除按键。

（注意：要删掉 stm32f4xx_fmc.c，不要删掉 stm32f4xx_fsmc.c）

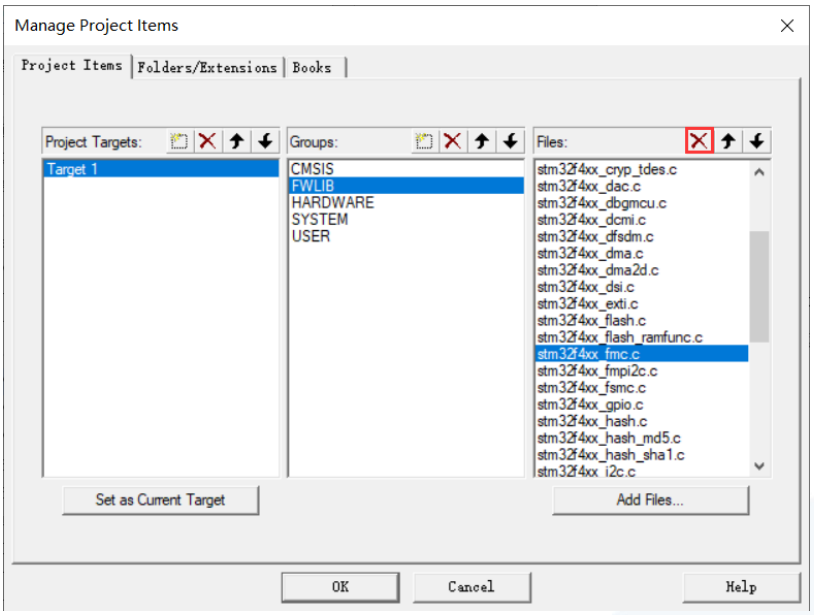


图 2.2.12 删除 fmc 文件

11．HARDWARE、SYSTEM 文件可后续新建或导入相关工程文件时进行导入，这里暂不做添加，接着开始添加 USER 目录文件，选中 USER 目录点击 Add Files…进行相关文件添加，这里只需要添加 stm32f4xx_it.c 文件，main.c 文件可以手动新建。

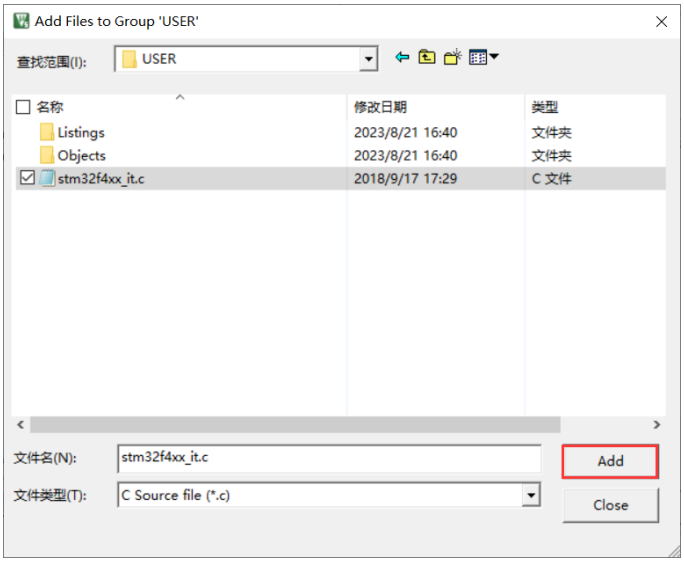



图 2.2.13 添加 USER 文件

12．固件库文件添加完成后，回到工程目录点击新建文件，会生成 Test1 文件。

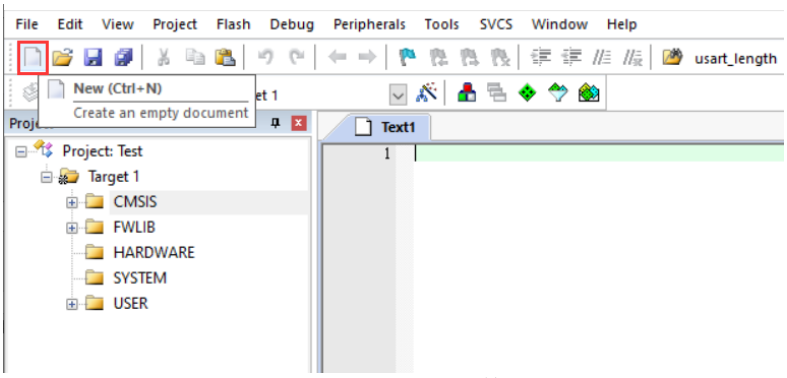



图 2.2.14 新建文件

13．接着点击保存按键，或按住 Ctrl+S 进行保存，进入保存界面后选择到 USER 文件夹下，设置文件名为 main.c，点击保存。

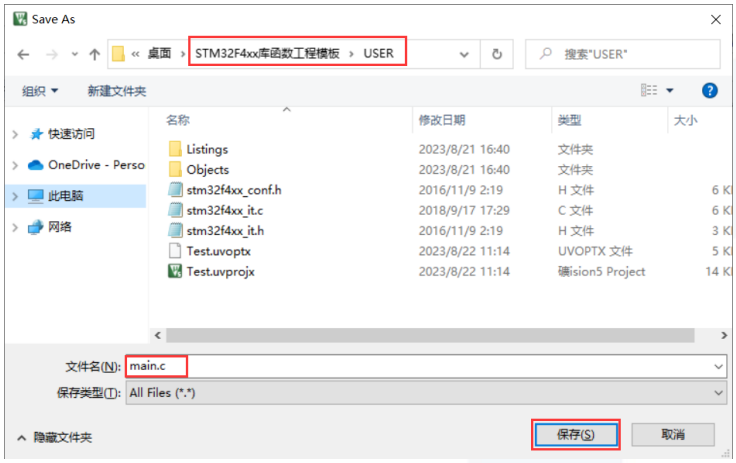



图 2.2.15 新建 main.c 文件

14．然后再根据之前步骤添加 main.c 文件到 USER 目录中，或直接在工程管理路径下双击 USER 文件夹进行添加，选中 main.c 文件点击 Add 添加。

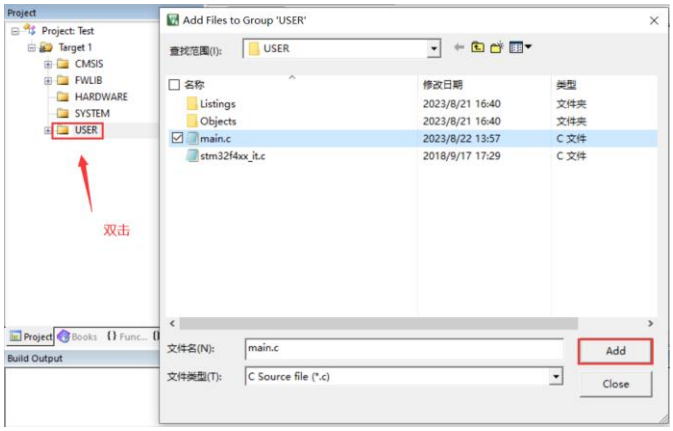



图 2.2.16 添加 main.c 文件

15．main.c 函数编写，可参考固件库提供案例源码或根据下方代码编写相关函数。

代码段 2.2.1 主程序函数（main.c）

```
#include "stm32f4xx.h"
void GPIO_Config(void) {
    GPIO_InitTypeDef GPIO_InitStruct; //使能 GPIOF 的时钟
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOF, ENABLE);
    GPIO_InitStruct.GPIO_Pin = GPIO_Pin_9; //配置 GPIOF Pin9 为输出
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
    GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOF, &GPIO_InitStruct);
}
int main(void)
{    int i;
    GPIO_Config();  //初始化 GPIO
    while (1) {
        GPIO_ToggleBits(GPIOF, GPIO_Pin_9);  //切换 GPIOF Pin9 的状态
        for (i = 0; i < 50000000; i++);   //延迟
    }
}
```

16．添加完成相关.c 源文件和相关.s 启动文件后，还需添加头文件路径，点击 Options for Target 按键，再选择 C/C++选择 Include Paths 添加头文件路径。

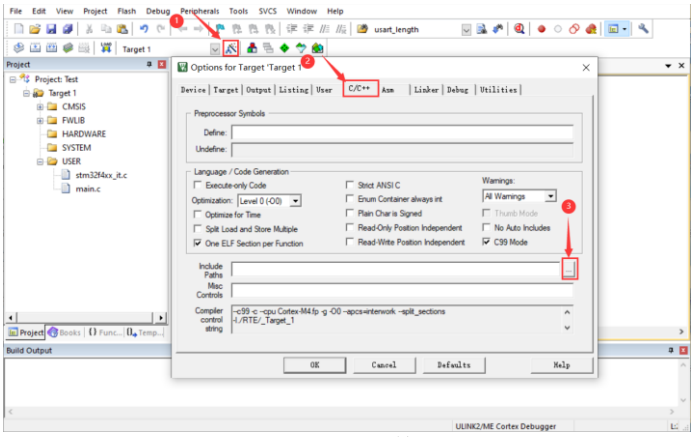



图 2.2.17 头文件路径

17．点击添加按键进行头文件路径添加，可点击按键进行路径选择。

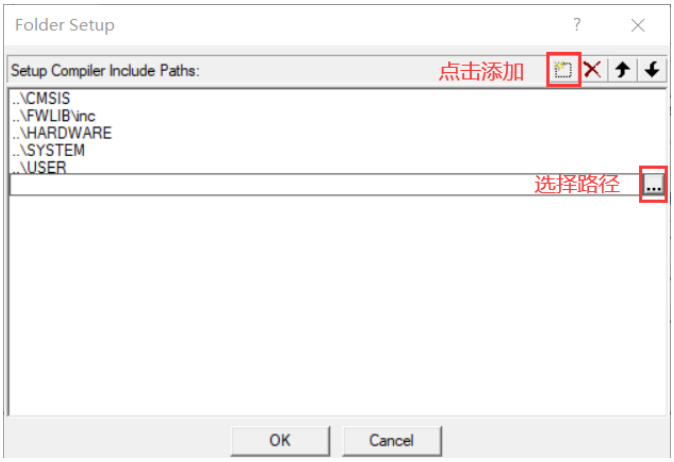



图 2.2.18 添加路径

接着按照顺序依次选择需要添加头文件的文件夹路径，点击选择文件夹进行添加，需要注意的是头文件路径仅包含当前路径下的头文件，不包含嵌套文件夹中的头文件，需根据实际情况进行添加，添加的路径设定为相对路径，这里依次添加的路径有：

..\CMSIS

..\FWLIB\inc

..\HARDWARE

..\SYSTEM

..\USER

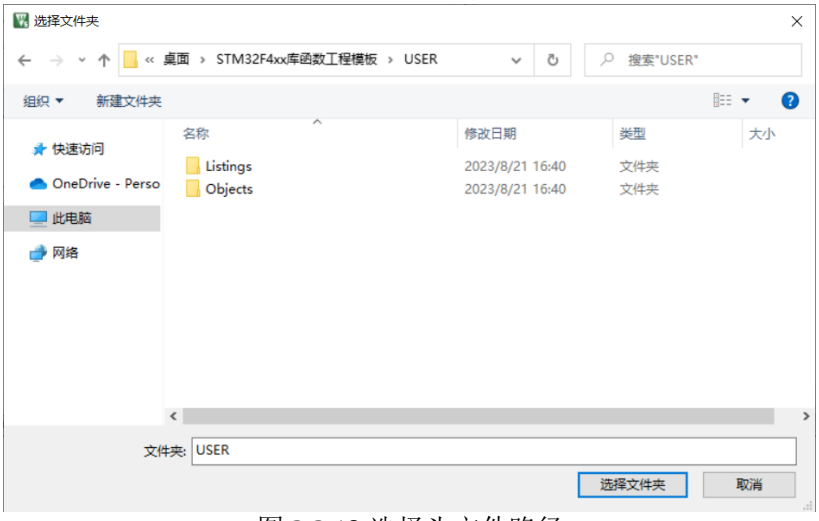



图 2.2.19 选择头文件路径

注意：FWLIB 目录下的头文件在 inc 文件夹下，所以选择 FWLIB\inc 文件路径。

18．接着还需要添加一个全局宏定义标识符，添加方法是点击 Options for Target 按键后，进入 C/C++选项卡，然后在 Define 输入框输入：“STM32F40_41xxx,USE_STDPERIPH_DRIVER”。

注意：这里是两个标识符。STM32F40_41xxx 和 USE_STDPERIPH_DRIVER，他们之间是用逗号隔开的。

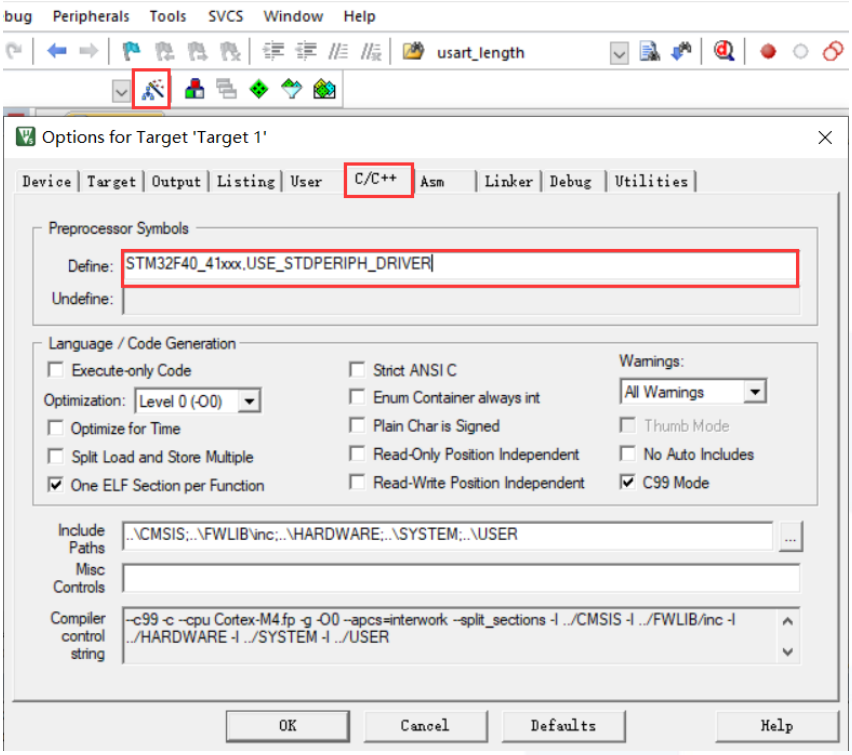



图 2.2.20 添加全局宏定义

19．配置完成后点击 或 按键进行编译， Build（构建）选项用于编译项目，并仅重新编译修改过的源文件及其依赖项， Rebuild（重建）选项用于完全重新编译项目的所有源文件，而不考虑修改的时间戳。编译结束后会提示"main.h": No such file or directory，这是没有定义 main.h 文件，可以在 USER 文件中进行定义用于相关头文件存放，或点击错误信息可定位到 stm32f4_it.c 文件，删除或注释#include "main.h"。

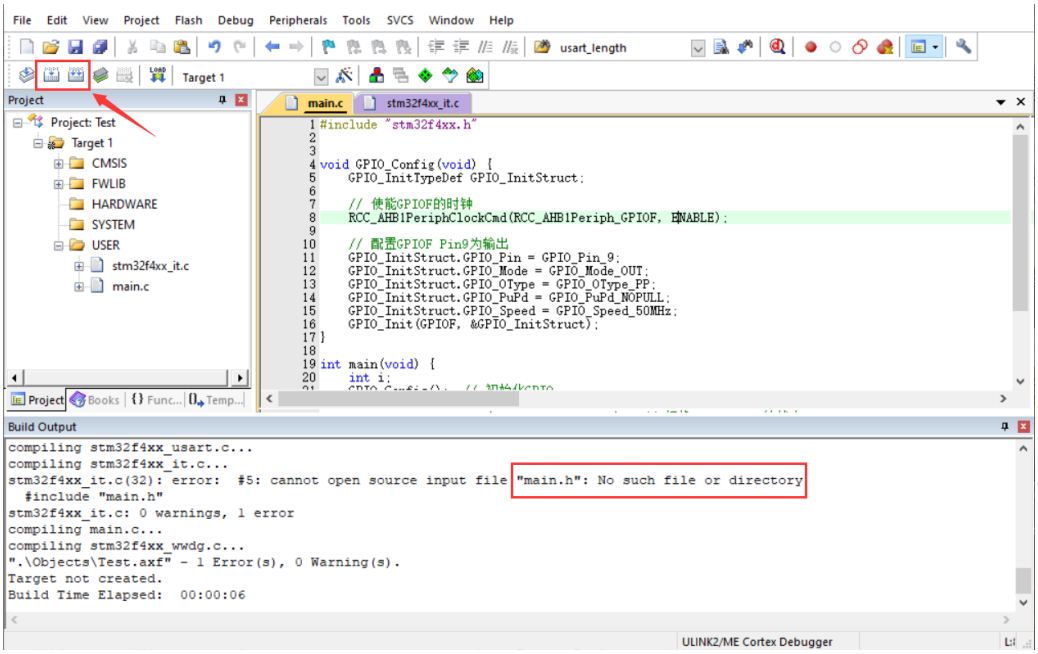


图 2.2.21 编译文件

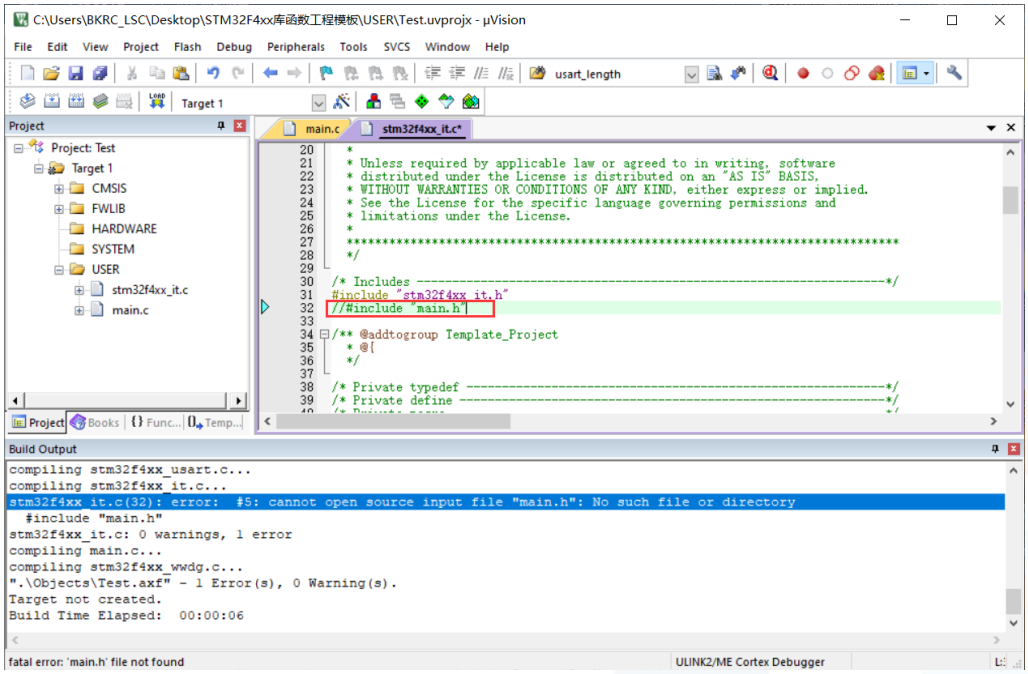


图 2.2.22 注释错误信息

20．再次点击 编译后提示还有一处错误，同样定位过去后将 TimingDelay_Decrement();进行注释或删除。

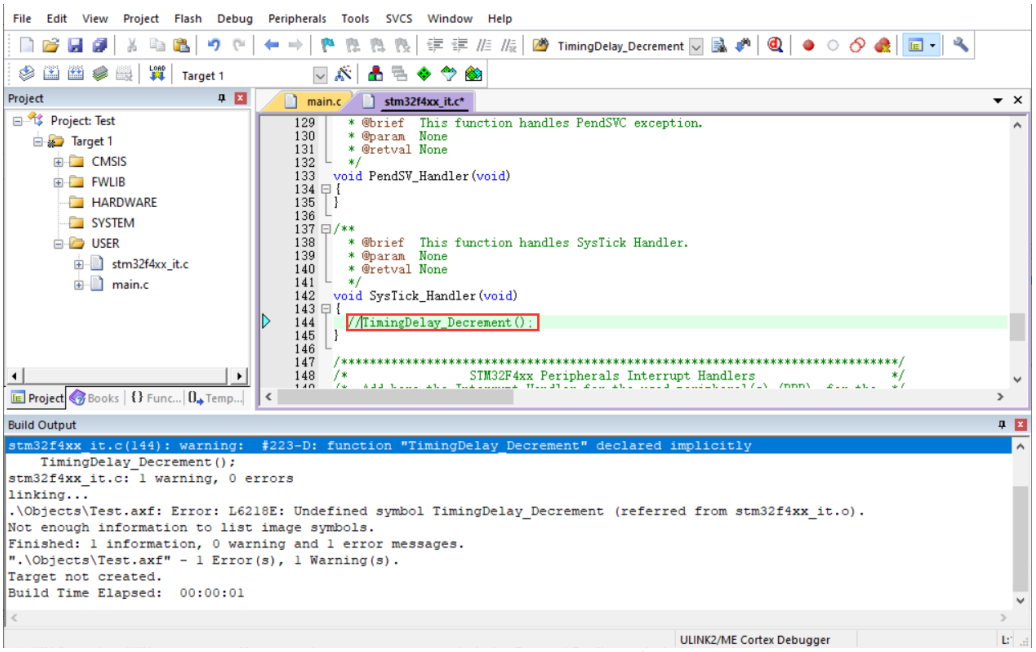


图 2.2.23 注释错误信息

到此 MDK Keil 工程创建完成，可以保存此工程模板，用于后续其他程序的开发。

## 2.2.3 开发工具配置

在软件开发过程中，可以通过配置 MDK 来实施代码格式规范，并进行常用配置选项的设置。这样可以确保项目的一致性，提高代码的可读性和可维护性。

**1．MDK 5 编码格式选择**

为了更便于使用并与串口助手、图像取模软件等工具通用，建议在使用 MDK5 时修改编码格式为 GB2312。

首先找到工具栏中的 Configuration 图标，在“Editor”中选择 GB2312 编码。

图 2.2.24 打开配置工具

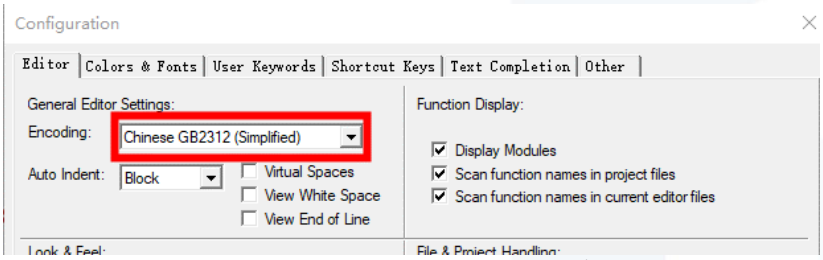


图 2.2.25 选择 Chinese GB2312 编码

### 2．MDK 5 设置 Tab Size 选项

为确保格式统一，在工具栏中的 Configuration 配置中，将 Tab Size 选项的值修改为 4，修改完毕后，点击 OK 即可。

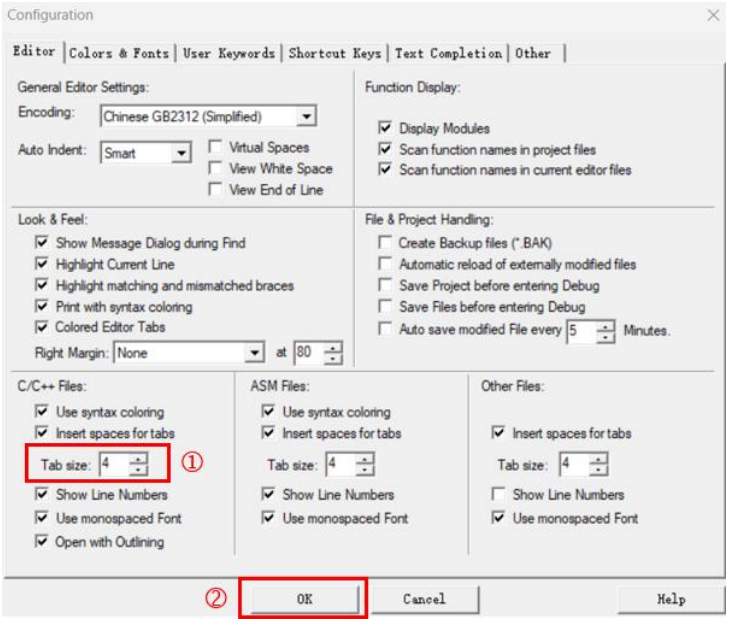


图 2.2.26 修改 tab size 选项

### 3．MDK 5 下载器配置

点击 Options for Target 按键后，进入 Debug 选项卡，点击 Use 下拉框，根据需要选择下载器，这里我们选择 ST-Link 下载器，然后点击 Settings，进入下一步配置。

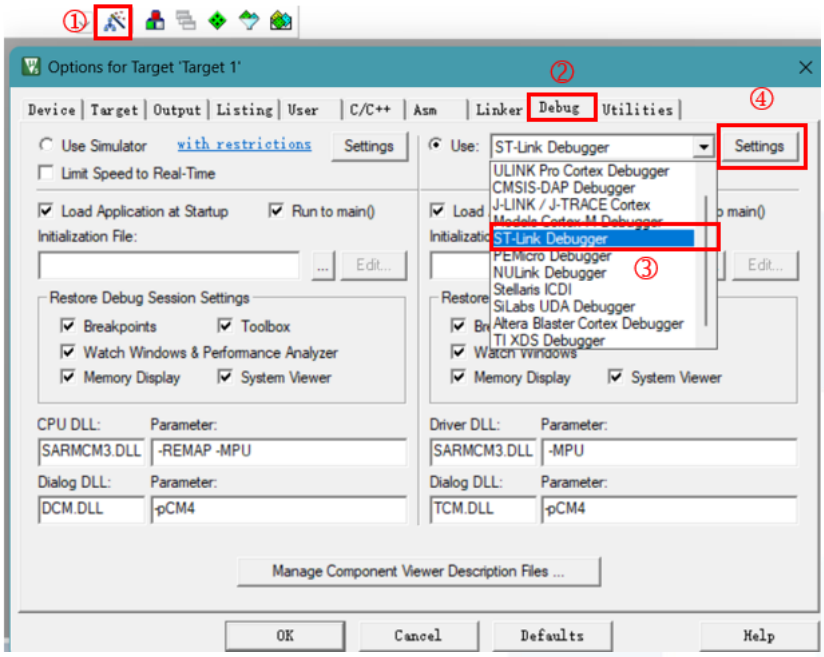


图 2.2.27 下载器设置界面

点击 Settings 按钮后，进入设置，如下图所示，其中的①和③所对应的区域会自动识别，若为空白，则证明电脑、下载器与核心板未连接，若连接正常，则会按照图中所识别的显示。这里只需确保②所对应的 Port 下拉选项为 SW 即可，点击 Flash Download 选项卡，进入下一步配置。

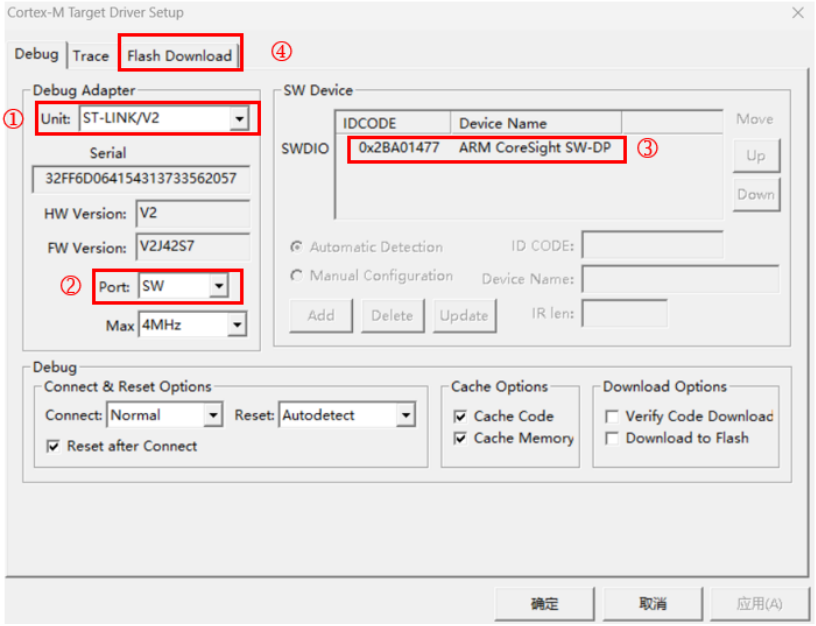



图 2.2.28 下载器设置界面

如下图，需确保 Reset and Run 选项被选中，②所对应区域的下载算法与图中一致，若为空白，需点击 Add（对应③），选择④所对应的算法，点击 Add（对应⑤）添加下载算法，完成后在每一个窗口中都应点击确定/OK 保存下载器的配置。

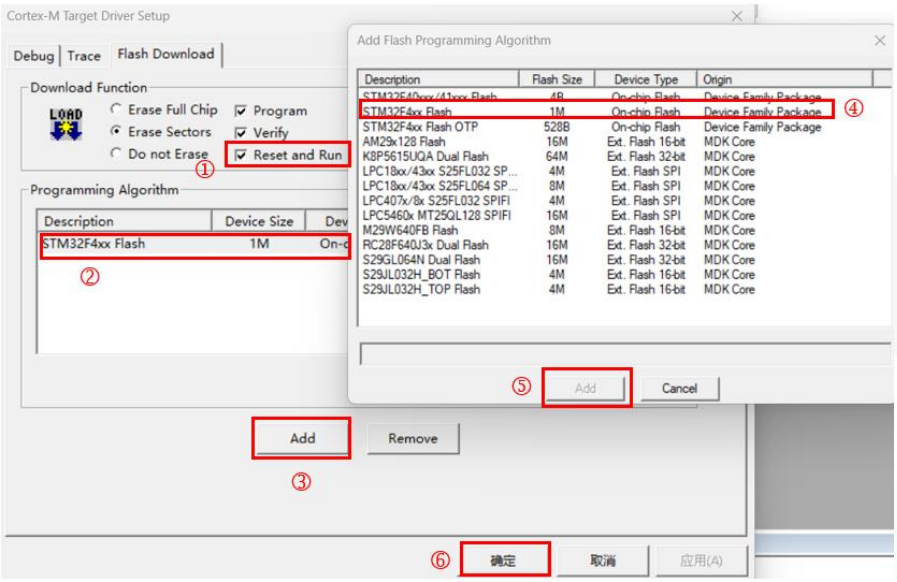



图 2.2.29 下载器设置界面

# 2.3 程序下载调试

## 2.3.1 下载连接示意图

在进行程序下载时，通常会选用 ST-Link 或 J-Link 下载器，以 ST-Link 下载器为例，首先将其连接至计算机。随后，通过 JTAG_SWD 转接板连接下载器 JTAG 接口，接着使用 6P 排线分别将下载器的 SWD_6P 接口与核心板的 SWD 接口相连接，如下图所示：

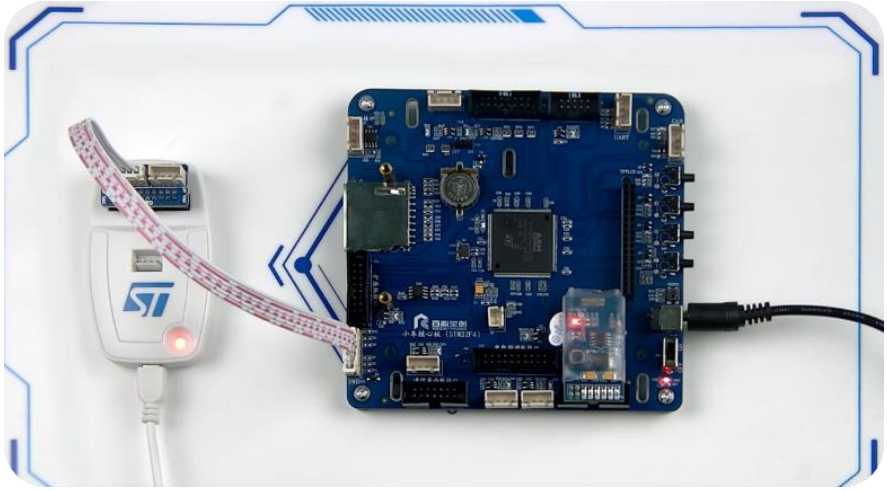


图 2.3.1 下载器与主机连接示意图

注意：程序下载前需确保安装相关下载器驱动，相关驱动工具可在开发工具中查找。

## 2.3.2 程序下载与调试

如下图所示，展示了 keil 项目的启动页面示意图，在图中的工具栏区域中：

（1） 和 是编译程序按钮。它的作用是构建应用程序，编译并链接相关源文件。

（2） 是下载程序按钮。它的作用是下载应用程序，通常是下载到设备的片上 Flash 闪存中。

（3） 是调试按钮。它的作用是使用调试器在目标硬件上运行应用程序。

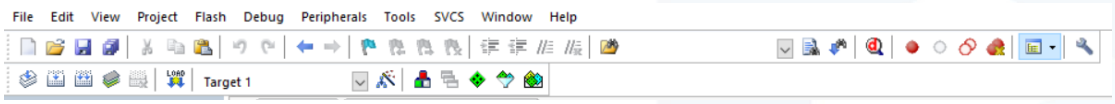



图 2.3.2 工具栏页面示意图

1．项目编译（构建应用）

在打开一个项目后，使用工具栏上的"Rebuild"按钮构建应用程序。在 Build Output 窗口显示有关编译过程的信息，无错误的编译还会显示有关程序大小的信息。

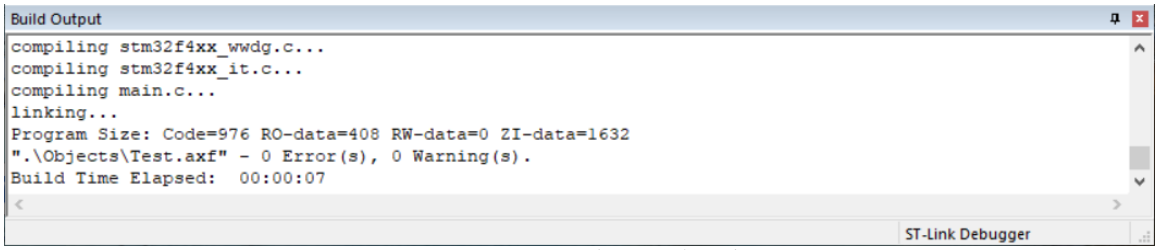



图 2.3.3 keil 项目编译成功示意图

2．项目下载

使用通过 USB 连接的调试适配器将目标硬件连接到计算机，并通过对下载器进行配置后，点击工具栏上的“Download”按钮，将应用程序下载到目标硬件上。

在 Build Output 窗口显示有关下载进度的信息。

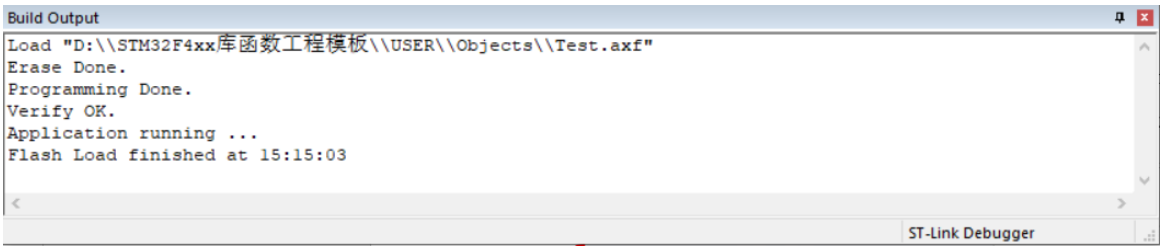



图 2.3.4 keil 项目下载成功示意图

3．项目调试

点击工具栏上的“Start/Stop Debug Session”按钮开始调试硬件上的应用程序，此时 IDE 会打开调试会话窗口，如下图所示。再次点击按钮就可以退出调试模式。

注意：项目调试需要使用通过 USB 连接的调试适配器将目标硬件连接到计算机。

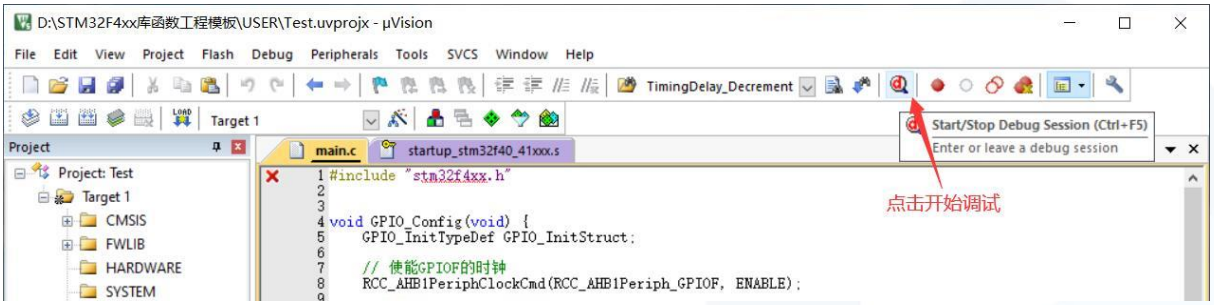



图 2.3.5 进入调试窗口

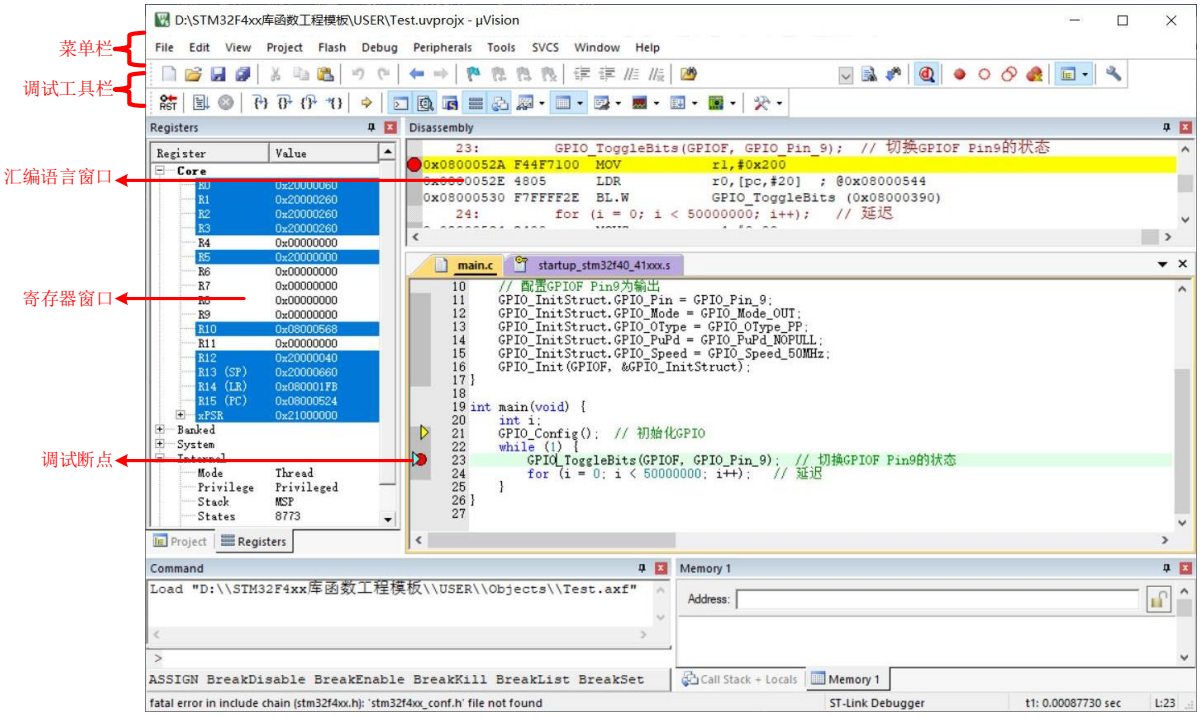



图 2.3.6 keil 项目调试会话窗口示意图

在调试会话开始期间，IDE 会加载应用程序、执行启动代码并在主函数（main 函数）处停止。

在调试工具栏上可快速使用许多调试命令，如下图所示：

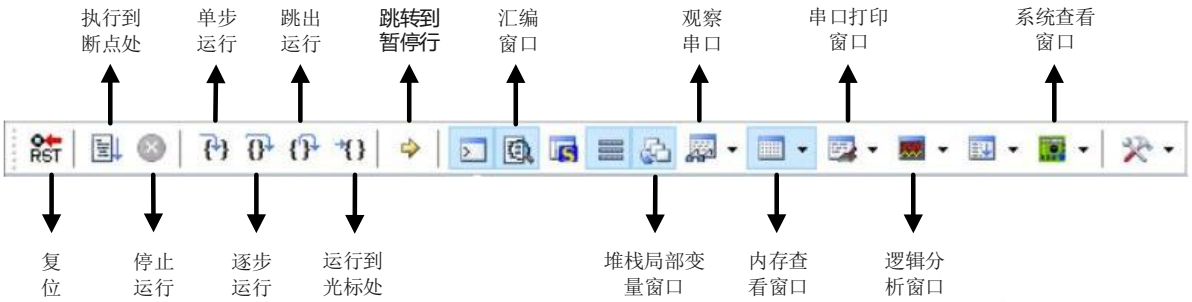



图 2.3.7 调试工具栏

：“Step”，单步运行。每点一次按钮，程序运行一步（单步），遇到函数会跳进函数执行。

：“Step Over”，逐步运行。每点一次按钮，程序运行一行（单行），遇到函数跳过函数执行。

：“Step Out”，跳出运行。每点一次按钮，程序跳出当前函数执行，直到跳出最外面的函数（main 函数）。

：“Run to Cursor Line”，运行到光标行。将光标放在某一行，点击该按钮，程

序全速运行，直到运行到光标行的位置停止下来。

注：如下图所示，黄色箭头表示程序运行至当前行，蓝色箭头表示光标行。当想要运行到某处时，只需点击该处的代码，蓝色箭头便会随之移动到该处所在的行，然后点击 {} 即可将程序运行到该处。

```
13     GPIO_InitStruct.GPIO_OType = GPIO_OType_PP;
14     GPIO_InitStruct.GPIO_PuPd = GPIO_PuPd_NOPULL;
15     GPIO_InitStruct.GPIO_Speed = GPIO_Speed_50MHz;
16     GPIO_Init(GPIOF, &GPIO_InitStruct);
17 }
18
19 int main(void) {
20     int i;
21     GPIO_Config();  // 初始化GPIO
22     while (1)
23     {
24         GPIO_ToggleBits(GPIOF, GPIO_Pin_9);  // 切换GPIOF Pin9的状态
25         for (i = 0; i < 50000000; i++);   // 延迟
26     }
27 }
28
```

图 2.3.8 光标行示意图

：“Run”，全速运行程序，或到下一个断点处。

：“Stop”，停止程序执行。

：“Reset”，执行 CPU 复位，复位之后程序回到最开始处。

：“Show next statement”，跳转到暂停行。点击该按钮，跳转到程序暂停（停止）所在行。

我们可以合理的使用以上调试命令，帮助我们识别和修复程序中的错误、问题和异常，从而对程序中的这些错误、问题和异常进行修改。

# 第3章 嵌入式智能车功能介绍

## 3.1 设备简介

嵌入式和人工智能技术的发展和不断落地的场景化应用，极大地促进了相关产业的发展，嵌入式技术与人工智能、物联网、机器人等新技术、新产业、新业态深度融合，已成为各领域数字化、智能化增值的关键性带动技术，如：智能家居、智能医疗、智能交通、智能电网等。智能化的升级改造必将是未来发展的主要趋势，使用嵌入式技术可以在社会的各行各业中大放异彩，由此更加体现出嵌入式技术结合行业落地应用的重要性。

嵌入式智能车综合应用创新实训开发单元（以下简称平台）是一款适用于嵌入式微控制器技术、传感器技术、RFID 技术、无线传感网技术、移动应用开发、嵌入式人工智能与边缘计算等相关技术的教学落地应用，能够满足智能语音、移动互联、智能机器人控制等相关课程的实验实训使用，具有自主知识产权。平台整体试图如下图所示：

图 3.1.1 嵌入式智能车综合应用创新实训开发平台视图

该平台模仿现代自动智能汽车设计，本身具有主动的环境感知能力，3.5 寸 TFT 真彩屏提供了优良的人机交互界面，用于显示平台的状态信息。平台采用 CAN 总线通信，多个处理器高效协同工作，数据处理更加流畅稳定。采用多通道无线技术，相关信息通过 OLED 显示屏显示。并且完全满足基于移动互联控制系统的智能车运动控制，支持进行视频图像采集、图像处理与识别等程序的开发、测试和应用，软硬件资源全部

开放，适合扩展教学、实训教学以及二次开发等。

平台配套语音识别处理单元、语音识别交互单元、智能语音交互开发系统，内置了离线语音识别引擎，自研基于神经网络深度学习的本地唤醒、回声消除和前端降噪算法，可完成在线语音识别、离线语音识别、语音交互控制等应用与实验。

平台支持嵌入式核心技术线上课程，包含 C 语言程序设计、嵌入式竞赛平台 Android 应用开发、嵌入式竞赛平台无线组网通信，配套完整的参考资源，包含文档教程、器件手册、参考案例代码、开发环境及软件工具等。

平台配套了超声波测距实验、语音识别实验、RFID 检测实验、光强度传感器检测实验、红外发送实验、综合程序案例等资源。

# 3.2 设备功能介绍

## 3.2.1 硬件介绍

**1. 设备整体试图**

嵌入式智能车综合应用创新实训开发平台整体试图如下图所示：包括通讯显示板、电机驱动板、核心板、云台摄像头、任务板、循迹板。

图 3.2.1 嵌入式智能车综合应用创新实训开发平台整体视图

**2. 模块资源介绍**

**1）任务板**

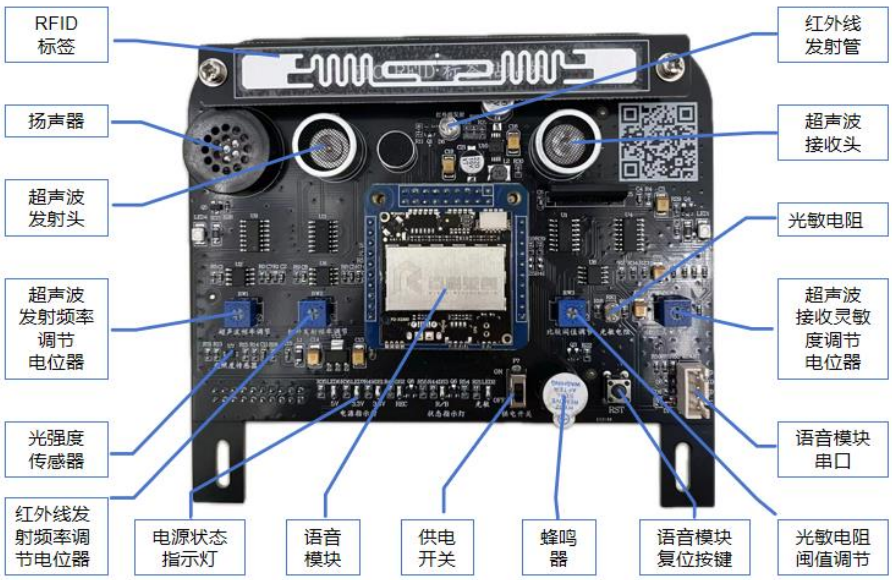



图 3.2.2 任务板

任务板搭载多种传感器及驱动电路，能够实时感知当前环境并迅速做出响应，包括光照强度传感器、超声波传感器、光敏电阻传感器、语音识别单元、红外发射驱动电路、转向灯驱动电路、蜂鸣器驱动电路等。

核心功能介绍如下：

- RFID 标签：用于 ETC 标志物检测标签。
- 超声波发射探头：超声波发送端。
- 超声波接收头：超声波接收端。
- 超声波发射频率调节电位器：超声波发送载波频率调节。
- 光强度传感器：型号为 BH1750 的光强度传感器，可直接将光强度物理量转化为具体的光强度数值。
- 红外线发射频率调节电位器：红外发送载波频率调节。
- 电源状态指示灯：用于任务板是否供电指示。
- 语音模块：用于语音识别。
- 供电开关：用于任务板供电。
- 蜂鸣器：用于警示功能（开启、关闭）。
- 语音模块复位按键：语音交互模块硬件复位按键。
- 语音模块串口：语音交互模块串口接口，用于语音交互模块调试、词条烧录。
- 红外发送管：通过处理器实现发送红外数据。
- 超声波接收灵敏度调节电位器：用于超声波接收调节。
- 光敏电阻：用于检测光线强度（只能输出 0 或 1）。

**2）网络摄像头**

图 3.2.3 新版摄像头

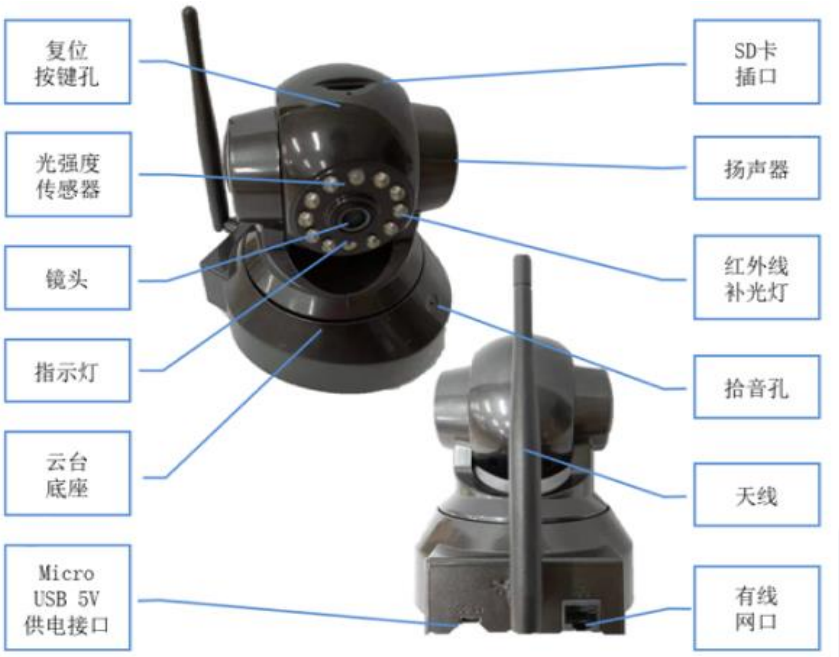



图 3.2.4 旧版网络摄像头

摄像头像素为 100 万像素，最大支持 1920*1080 分辨率。云台转角水平最大 355°，垂直最大 120°。连接方式包括：WiFi 无线连接和有线连接。

摄像头开机一分钟内会有自检过程，自检过程中摄像头水平旋转一周，垂直旋转 120°，请在自检完成后再进行操作。

摄像头自带恢复出厂设置按键，出现连接故障时可直接恢复出厂设置，根据教程

重新配置。

核心功能介绍如下：

- 镜头：摄像头镜头，可调整焦距。
- 天线：摄像头天线，用于摄像头无线通讯。
- 供电接口：摄像头供电口（Micro USB，5V 供电）。
- 复位按键孔：摄像头复位，可通过按键触发孔内复位按键。
- 有线网口：摄像头网线接口，用于与 Wifi 模块或 A72 开发板连接，以及摄像头配置。
- 光强度传感器：摄像头检测光线强度传感器，用于确定光线强度，从而确定是否打开红外灯。

**3）核心板**

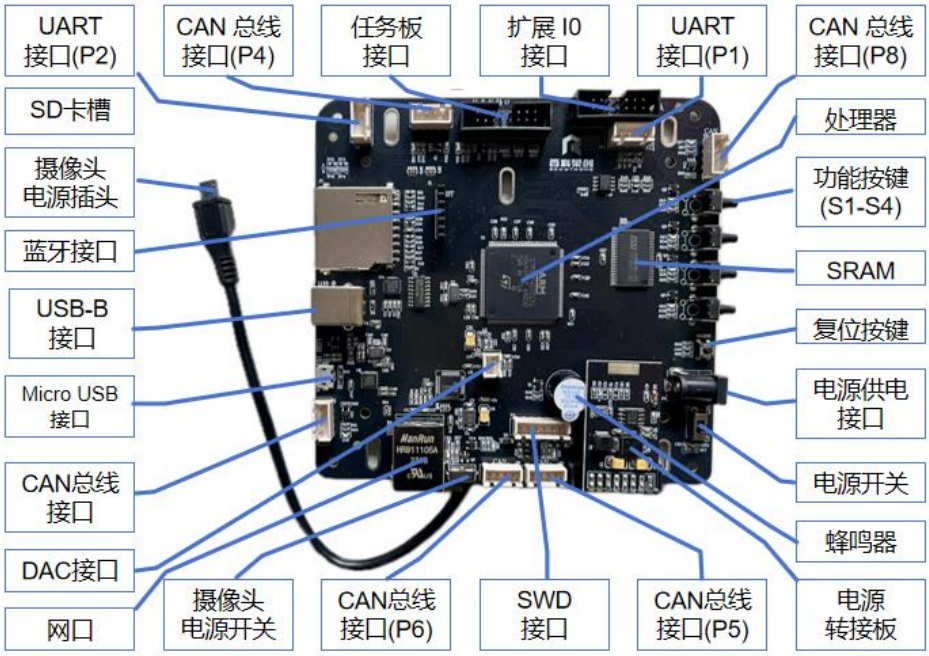



图 3.2.5 核心板

主车核心板采用 ARM Cortex-M4 内核的 STM32F407IGT6 为主控芯片，最高时钟频率可达 168MHz，满足实时数据处理要求。核心板预留多个通讯接口，方便二次开发学习使用，采用防反插封装接口，防止用户反插断路。

核心板板载硬件接口：蓝牙模块接口、USB-B 接口、Micro USB 接口、CAN 总线接口、有线网口、DAC 接口、 SWD 下载接口、扩展 IO 接口、SD 卡槽等。

核心功能介绍如下：

- UART（P2）接口：处理器串口，用于连接 A72 开发板串口。
- CAN 总线接口（P6）：连接电机驱动板。
- SWD 接口：处理器下载程序接口。
- CAN 总线接口（P5）：连接通讯显示板。
- 蜂鸣器：用于警示功能（开启、关闭）。
- 电源管理模块：核心板供电保护、降压功能。
- 拨码开关 SW1：核心板供电开关。
- 电源供电接口：连接电池。
- 摄像头开关电源：为摄像头供电。
- RESET 按键：处理器硬件复位按键。
- 功能按键：包含（S1、S2、S3、S4）。作为核心板处理器功能按键。
- UART（P1）接口：处理器串口，用于连接 RFID 读卡器。
- 任务板接口：处理器 IO 口，用于连接任务板。
- CAN 总线接口（P4）：用于连接循迹板。

**4）通讯显示板**

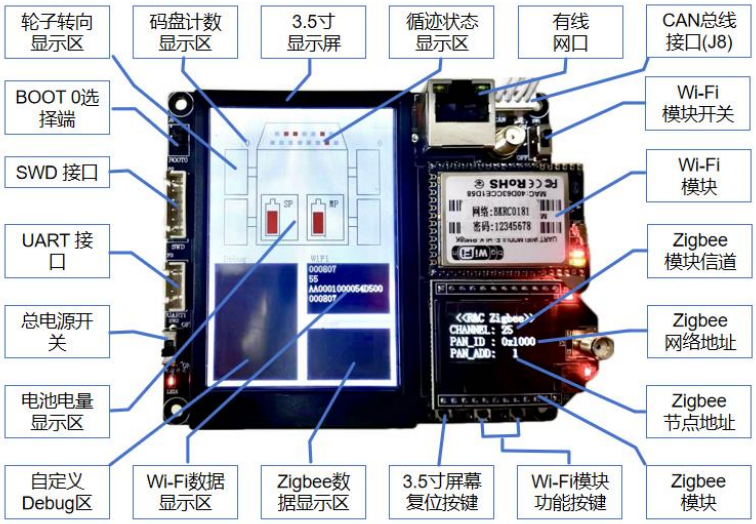



图 3.2.6 通讯显示板正面图

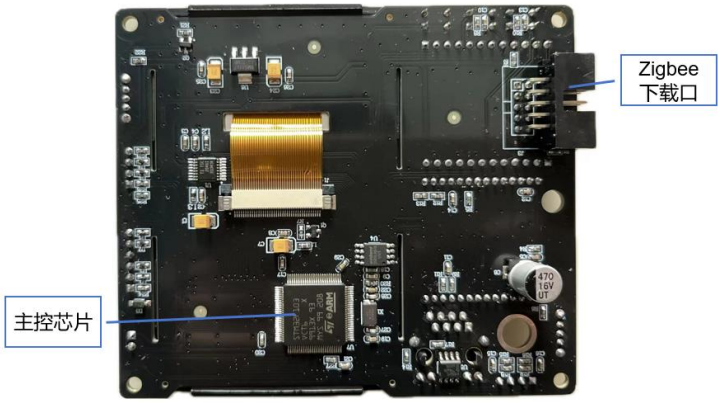



图 3.2.7 通讯显示板背面图

通信显示板采用 STM32F103VCT6 处理器，板载 3.5 寸 LCD 显示屏，可显示实时循迹状态、码盘数据（仅显示左前轮与右前轮）、WiFi 接收和发送数据、ZigBee 接收和发送数据、用户自定义调试信息。

通讯显示板板载 ZigBee 无线通信模块和 WiFi 无线通信模块，实现基于智能车与综合实训沙盘标志物、移动终端、网络摄像头和智能移动机器人之间交互控制。

核心功能介绍：

- SWD 接口：处理器下载程序接口。
- 拨码开关（SW2）：通讯显示板供电开关。
- 3.5 寸 LCD 显示屏：用于显示平台信息（循迹状态、电量、转动方向、码盘）。
- RESET 按键：处理器硬件复位按键。
- ES、WPS 按键：WIFI 模块硬件复位按键。
- ZigBee 模块：用于 ZigBee 无线通讯。
- WIFI 模块：用于与移动设备无线通讯，以及与摄像头进行有线或无线通讯。
- 拨码开关（SW1）：WIFI 模块供电开关。
- CAN 总线接口：连接核心板。
- 网口：WIFI 模块网口，用于连接摄像头或进行 WIFI 模块的配置。
- ZigBee 下载口：用于更新 ZigBee 模块的 HEX 文件。

5）电机驱动板

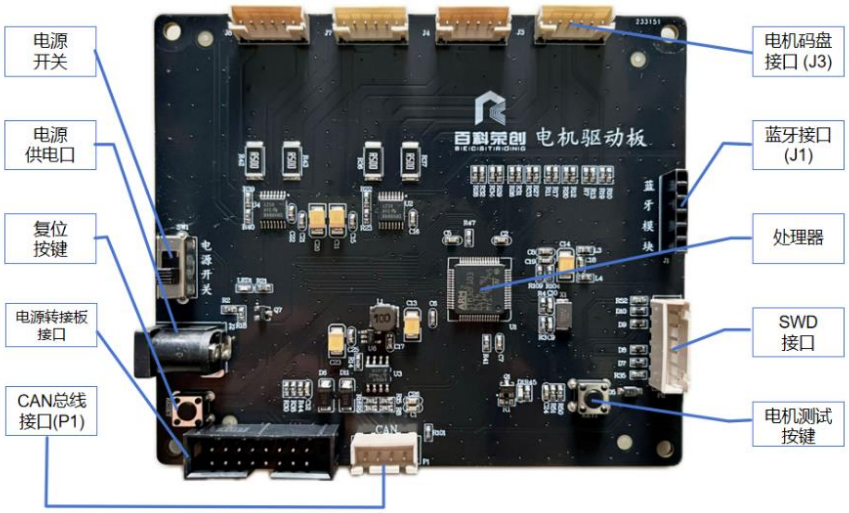



图 3.2.8 电机驱动板

电机驱动单元与核心控制单元分离，使四路电机驱动更加高效，并且在电机启动瞬间产生的浪涌不会对核心板处理器造成损伤有效提升了系统安全性。四路电机采用独立测速系统，实现四轮差速互补控制，提高了车辆运动控制的精度和稳定性。

核心功能介绍：

- 电源开关：电机驱动板供电开关。
- 电源供电接口：连接电池。
- 复位按键：处理器硬件复位按键。
- CAN 总线接口（P1）：连接核心板。
- 电机测试按键：用于测试电机是否正常，按键逐次按下，电机由左到右依次开启。
- SWD 接口：处理器下载程序接口。
- 电机码盘接口：连接电机码盘，J3-左前轮、J4-左后轮、J7-右前轮、J8-右后轮。

**6）循迹板**

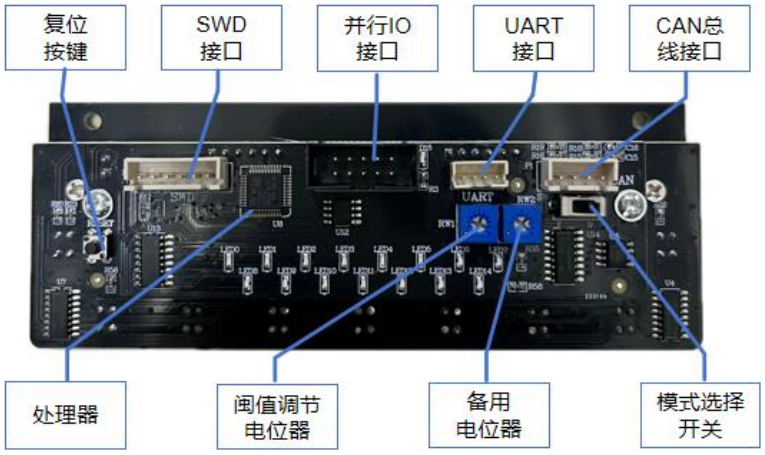



图 3.2.9 循迹板

循迹板采用反射式光电传感器，采用前七后八交叉排列方式，实时检测当前循迹状态，并将数据传回至核心板主控制器进行处理，从而实现自适应循迹，使智能车能够安全平稳行进。

循迹板前七路循迹灯可实现对前进方向道路预判，提高了循迹的精准度，使智能车行进的稳定性得到增强，实现循迹数据实时采集，实时处理，实时传输。

核心功能介绍：

- SWD 接口：用于处理器程序下载接口。
- 复位按键：为处理器硬件复位按键。
- 处理器：使用 STM32F103C8T6 型号。
- 并行 IO：用于并行输出循迹数据。
- CAN 总线接口：通过 CAN 总线传输循迹数据。
- 模式选择开关：用于选择设置循迹板为前置循迹板或后置循迹板。（F 前置，B 后置）。
- 阈值调节电位器：调节红外发送管发送功率。

# 3.3 设备基本操作

## 3.3.1 设备操作注意事项

为了保证平台能够正常的进行开发使用，需要对平台的常用按键及操作进行初步的了解，并注意以下注意事项：

**1. 平台通电注意事项：**

该平台为两块 12.6V 锂电池供电，因此平台配置了 2 个电源供电接口。核心板右侧为主电源供电开关，控制平台的各个模块电路；电机驱动板左侧为电机供电开关，控制四个轮胎电机正常运转。在使用时，需要将两个供电开关全部处于开启（ON）状态。

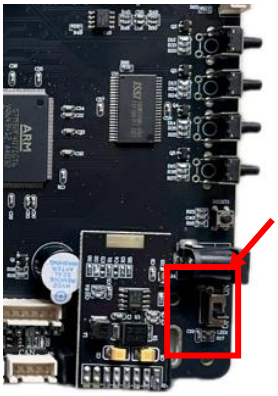
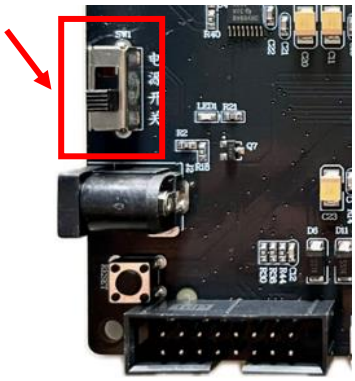


图 3.3.1 核心板供电开关



电机驱动板供电开关

在通信显示板的 LCD 显示屏上，有电池电量显示区，左侧电量为电机驱动板电池电量，右侧为核心板主电源电池电量。剩余电量低于 1/3 时，应使用 12.6V 电池充电器及时充电。该平台金属车架的左右两侧各有一个 12.6V 充电口，左侧为电机驱动板充电口，右侧为核心板充电口。

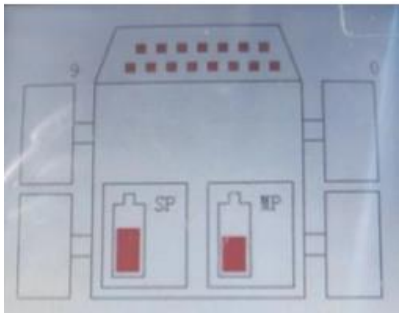



图 3.3.2 电量显示区

充电口

**2. 循迹板使用注意事项：**

该平台循迹板采用反射式光电传感器，前七后八交叉排列方式，实时检测当前循迹状态，其中模式选择开关应拨到 F 档（F 档为前置，B 档为后置），若拨到 B 档，循迹板将无法正常使用。

如果在使用时，确认模式选择开关在 F 档，但因地图材质、光线等原因的影响下，平台无法正常循迹，应使用小型一字螺丝刀转动 RW1（阈值调节电位器），来调节循迹灯的灵敏度，使之循迹恢复正常。

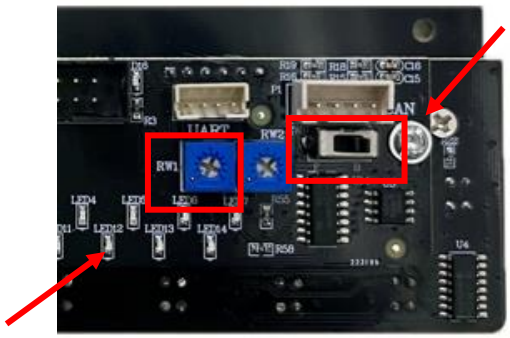



图 3.3.3 循迹板的模式选择开关和阈值调节电位器

**3. 电机驱动板使用注意事项：**

该平台采用 4 个独立电机控制轮胎运转，在电机驱动板上有 4 个电机码盘接口，J3-左前轮、J4-左后轮、J7-右前轮、J8-右后轮，注意连接顺序。

在使用过程中，可通过电机测试按键来测试电机驱动芯片（DRV8848），独立电机是否正常。正常情况下，每按下一次电机测试按键，左前轮、左后轮、右前轮、右后轮会依次转动。若某一轮胎无法正常转动，初步判断为控制该轮胎的电机存在故障。

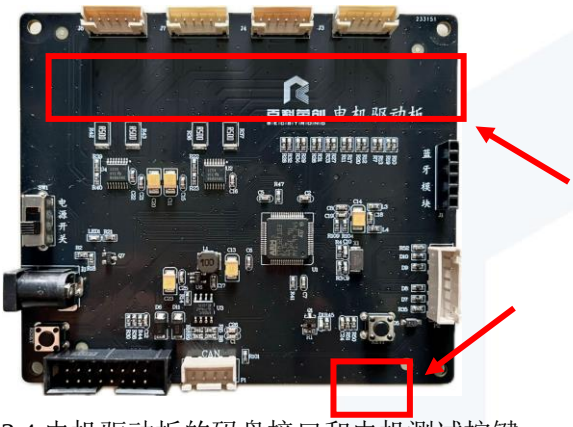


图 3.3.4 电机驱动板的码盘接口和电机测试按键

# 第4章 嵌入式微控制器基础实验

本章主要对 STM32F4 核心板的基础实验进行讲解，其中包括 LED 灯控制、按键检测、外部中断、定时器中断、PWM 输出、串口通信及 ADC 采样等实验。通过这些实验，读者将对 STM32F4 核心板的基础功能和应用有一个全面的了解，为后续的深入学习和实战开发打下坚实的基础。

## 4.1 LED 灯控制实验

本实验通过控制 LED 灯从而实现流水灯效果，让读者熟悉 STM32 最基本的外设——GPIO（通用输入输出）。

通过对本实验的学习，读者将学会如何配置和操作 GPIO 外设，并了解 STM32 的 GPIO 引脚的基本特性和功能。

### 4.1.1 LED 介绍

LED（Light Emitting Diode），发光二极管，是一种能够将电能转化为可见光的固态半导体器件，它可以直接把电转化为光。LED 的心脏是一个半导体晶片，晶片的一端附在一个支架上，一端是负极，另一端连接电源的正极，使整个晶片被环氧树脂封装起来。

如图 4.1.1 所示，展示了常见的几款 LED 灯：

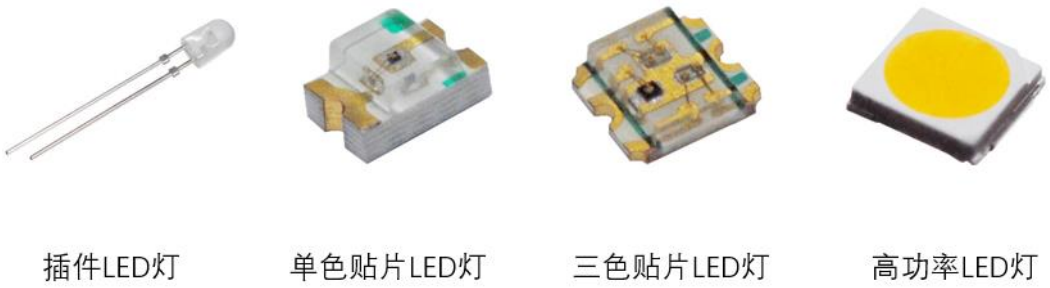



图 4.1.1 常见 LED 灯

其中贴片 LED 灯具备以下特点：

（1）发光原理是属于冷性发光，而非藉由加热或放电发光，所以元件寿命比钨丝灯泡长约 50~100 倍（约十万小时）。

（2）无需暖灯时间，点亮响应速度比一般电灯快（约 3~400ns）。

（3）电光转换效率高，耗电量小，比灯泡省约 1/3~1/20 的能源消耗。

（4）耐震性佳、可靠度高、系统运转成本低。

（5）易小型、薄型、轻量化、无形状限制、容易制成各式应用。

## 4. 1. 2 GPIO 介绍

**1. GPIO 的概念**

GPIO（General-purpose input/output），通用输入/输出端口，可以由 STM32 直接驱动从而实现与外部设备通信、控制外部硬件以及采集和捕获外部硬件数据的功能。如图 4.1.2 所示，芯片四周的大多数引脚都是 GPIO。

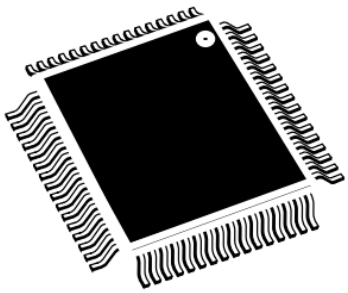


图 4.1.2 芯片示意图

在嵌入式开发过程中，经常用到一些外部功能模块，如 LED、按键、蜂鸣器等，这些外设模块原理简单，只需要与微控制器的 GPIO 引脚相连接，即可通过控制输入/输出电平信号，控制/获取模块状态。

当然还有一些外部模块，需要使用多个引脚与之进行数据交换，因此就需要约定传输数据的方式---通信协议，比如常见的通信协议有 UART、I2C、SPI 和 CAN 等等。

**2. STM32 的 GPIO**

以 STM32F407I 系列的芯片为例，它有 176 个引脚，除去电源引脚、晶振时钟引脚、复位引脚、启动选择引脚、程序下载引脚（大部分为最小系统必须引脚），剩下的则是 GPIO 引脚。

STM32 将这些 GPIO 引脚按 GPIOx（x 为 A、B、C、D、E、F、G、H、I）分组，除 GPIOI 只有 12 个引脚外（编号为 0~11），其他每个分组包含 16 个引脚（编号为 0~15）。即 140 个 GPIO 引脚。

通常来说，对 PA0 引脚，表示 GPIOA 组的 0 号引脚。

对于其他 STM32 的芯片，也适用于本规则。

如图 4.1.3 所示，列出了 STM32 产品型号的规则，从中可以看出，STM32F407I 系列的芯片为带 DSP 和 FPU 的高性能基础系列的 ARM Cortex-M4 MCU，其引脚数为 176&201（176+25）。

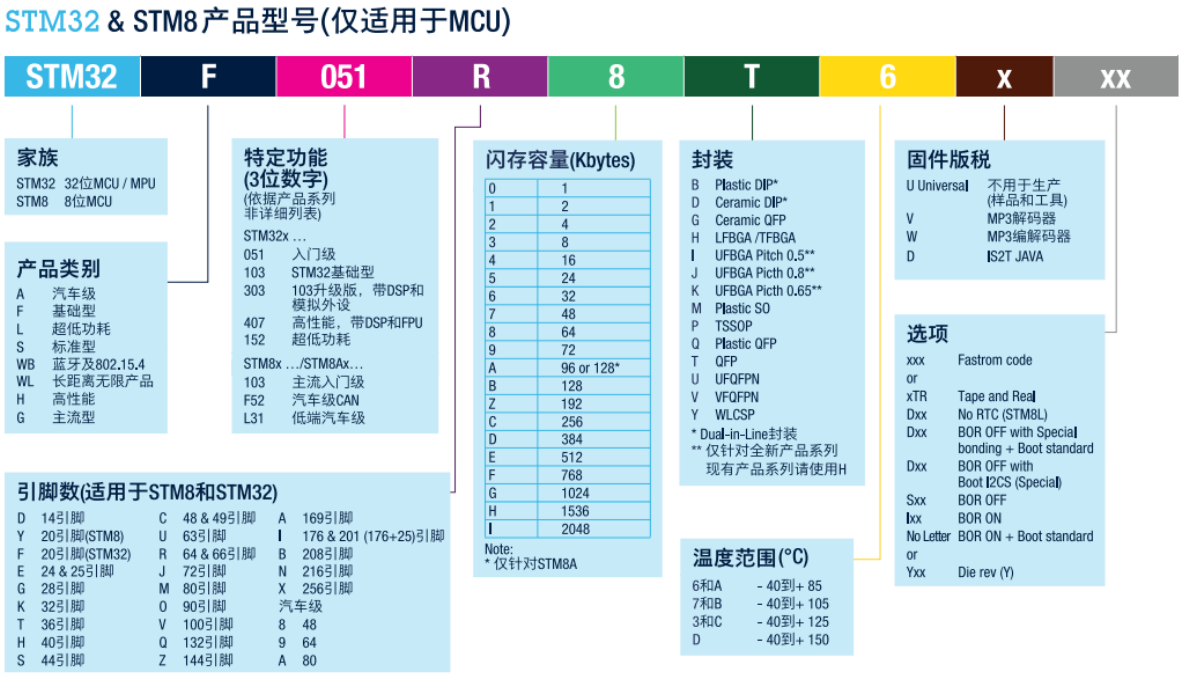



图 4.1.3 STM32 产品型号

### 3. STM32 的 GPIO 的主要特性

对于 STM32F4xx 系列的芯片，主要有以下特性：

（1）受控 I/O（每组）多达 16 个；

（2）输入状态：浮空、上拉/下拉、模拟；

（3）将数据输入到输入数据寄存器或外设（复用功能输入）；

（4）输出状态：推挽/开漏+上拉/下拉；

（5）从输出数据寄存器或外设（复用功能输出）输出数据；

（6）可为每个 I/O 选择不同的速度；

（7）置位和复位寄存器对输出数据寄存器具有按位写权限；

（8）锁定机制，可冻结 I/O 配置；

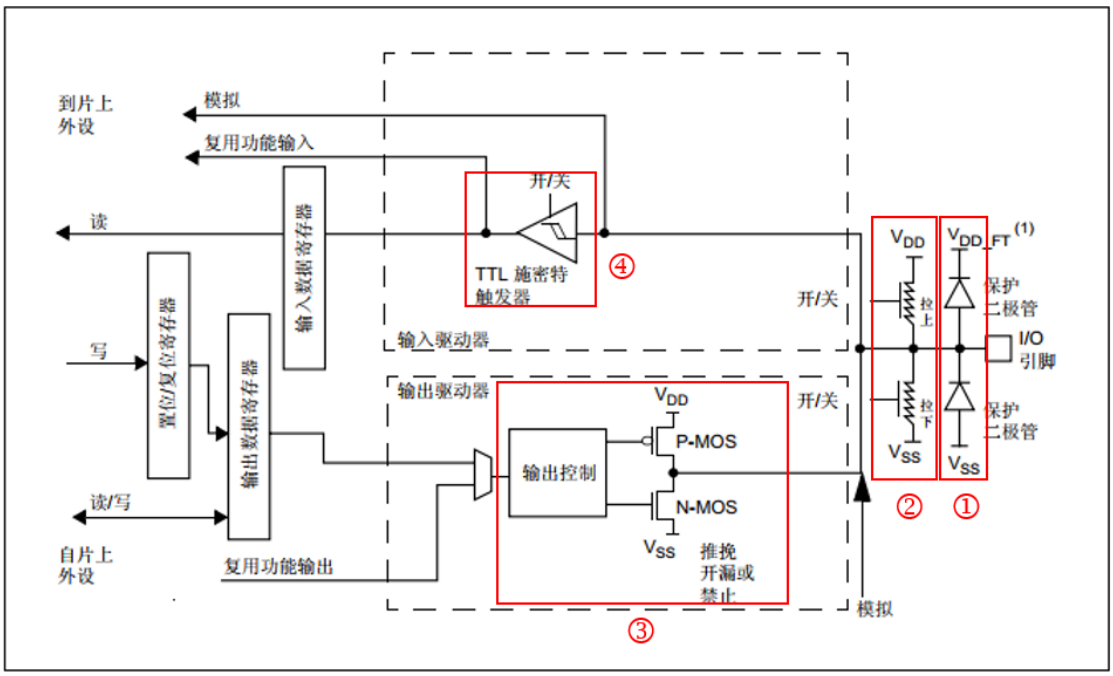
（9）模拟功能；

（10）复用功能输入/输出选择寄存器（一个 I/O 最多可具有 16 个复用功能）；

（11）快速翻转，每次翻转最快只需要两个时钟周期；

（12）引脚复用非常灵活，允许将 I/O 引脚用作 GPIO 或多种外设功能中的一种。

**4. GPIO 基本结构**




1. $V_{DD_FT}$ 是和 5 V 容忍 I/O 相关的电位，与 $V_{DD}$ 不同。

图 4.1.4 5V 容忍 I/O 端口位的基本结构

如图 4.1.4 所示，展示了 STM32F4xx 系列 GPIO 的基本结构，左侧部分连接 MCU 内部，中间上半部分为输入，中间下半部分为输出，右侧为 MCU 引出的外设 I/O 引脚。

以下是图中编号所对应区域的说明：

①保护二极管：I/O 引脚的上下两个二极管用于防止引脚外部过高/过低的电压输入，当引脚电压高于 VDD_FT 时，上方的二极管导通，当引脚电压低于 VSS 时，下方的二极管导通，防止不正常电压引入芯片导致芯片烧毁。

②上拉/下拉电阻：控制引脚默认状态的电压，开启上拉的时候引脚默认电压为高电平，开启下拉的时候引脚默认电压为低电平。

③P-MOS 管和 N-MOS 管：P-MOS 管高电平导通，低电平关闭；N-MOS 管低电平导通，高电平关闭。

④TTL 施密特触发器：基本原理是当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；当输入在正负向阈值电压之间，输出不改变。

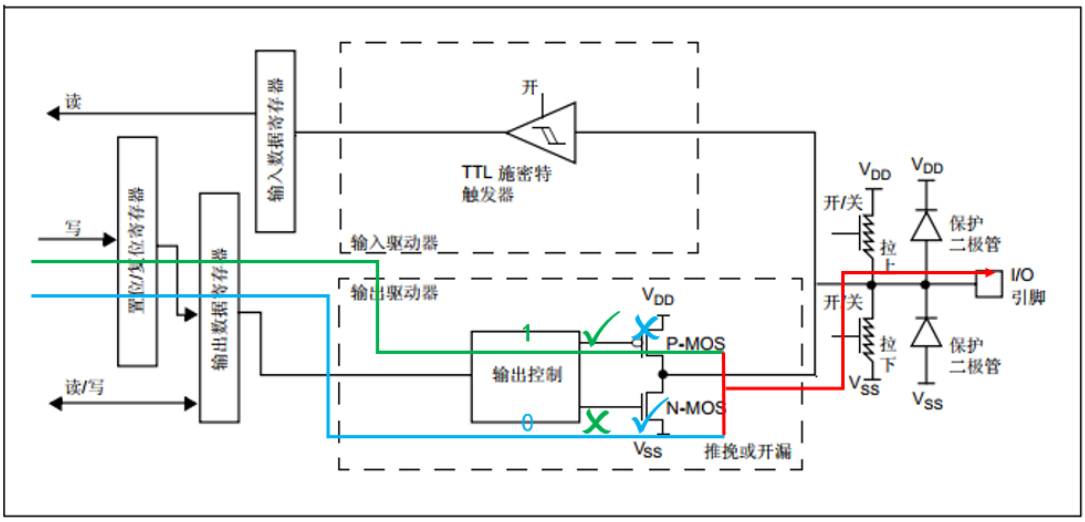
**5. GPIO 工作模式**

STM32F4xx 系列芯片的 I/O 引脚共有 8 种工作模式，其中输出模式有四种：推挽输出、开漏输出、复用推挽输出、复用开漏输出；输入模式有四种：上拉输入、下拉

输入、浮空输入、模拟输入。以下是对这 8 种工作模式的基本讲解：

（1）推挽输出（Push-Pull，PP）（带上拉或下拉）

推挽输出模式下，P-MOS 和 N-MOS 均有效。推挽输出模式下，STM32 对 I/O 口具有绝对的控制权，引脚可以输出高低电平。

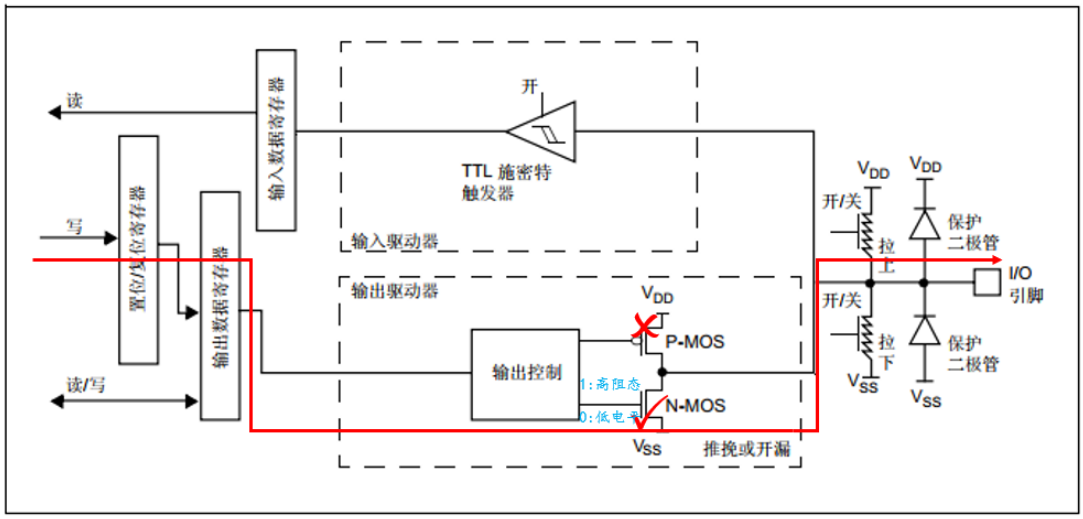



•当输出数据寄存器写1，上管导通，下管断开，输出控制接到VDD,输出为高电平；
•当输出数据寄存器写0，上管断开，下管导通，输出控制接到VSS,输出为低电平。

图 4.1.5 推挽输出配置

（2）开漏输出（Open-Drain，OD）（带上拉或下拉）

开漏输出模式下，P-MOS 无效，N-MOS 有效。开漏输出模式下，只有低电平才有驱动能力。高电平无驱动能力，需要借助外部上拉电阻完成对外驱动。引脚只能输出低电平，可以作为通信协议的驱动方式，比如 I2C 通信的引脚，使用开漏输出模式。




•当输出数据寄存器写1，N-MOS断开，输出控制相当于断开，也就是高阻态；当输出数据寄存器写0，N-MOS导通，输出控制相当于接VSS，输出为低电平。

图 4.1.6 开漏输出配置

（3）复用功能推挽/开漏输出（Alternate Function，AF）（带上拉或下拉）

复用推挽输出：引脚不再是普通的 I/O 口，不仅具有推挽输出的特点，还具有片上外设的功能。例如 I/O 口用作 I2C 的 SCL 或者 SDA 时，使用该模式。

复用开漏输出：引脚不再是普通的 I/O 口，不仅具有开漏输出的特点，还具有片上外设的功能。例如 I/O 口用作 USART 的数据发送端 TX 或者接收端 RX 时，使用该模式。

复用推挽输出和复用开漏输出的高低电平的来源，不再是输出数据寄存器，取而代之的是利用片上外设模块的复用功能输出来决定的。

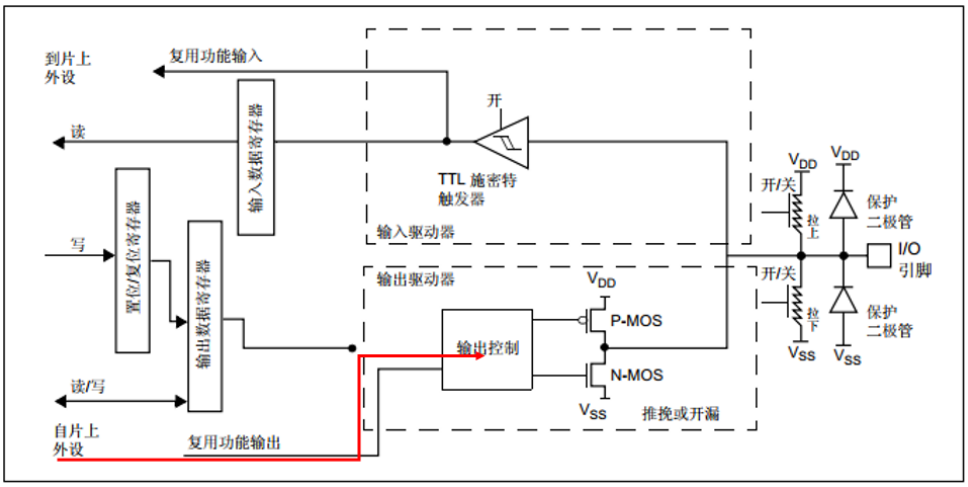



图 4.1.7 复用推挽输出/复用开漏输出配置

（4）上拉输入模式（Input Pull-up）

上拉输入就是 GPIO 内部有上拉电阻。GPIO 内部的上拉电阻开关打开，下拉电阻断开，该模式下，引脚上拉，当没有外部输入时默认输入高电平。

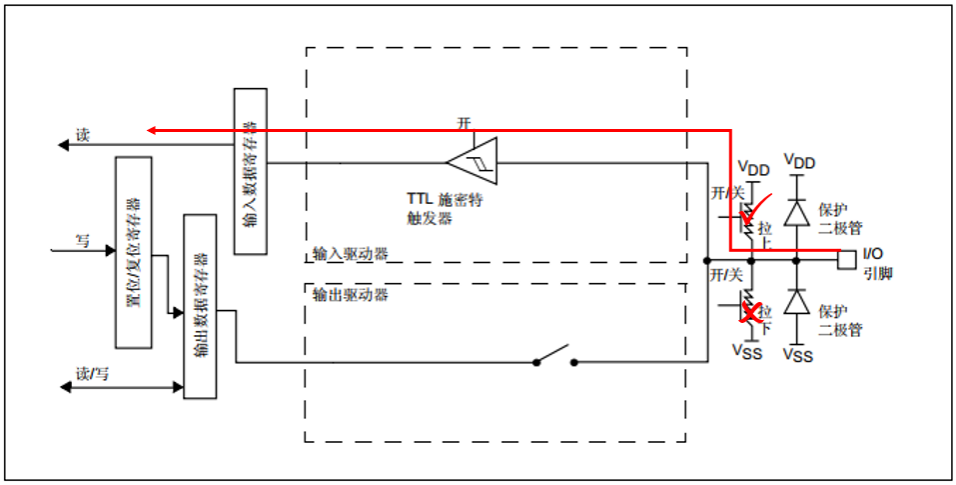



图 4.1.8 上拉输入配置

（5）下拉输入模式（Input Pull-down）

下拉输入就是 GPIO 内部有下拉电阻。GPIO 内部的下拉电阻开关打开，上拉电阻断开，该模式下，引脚下拉，当没有外部输入时默认输入低电平。

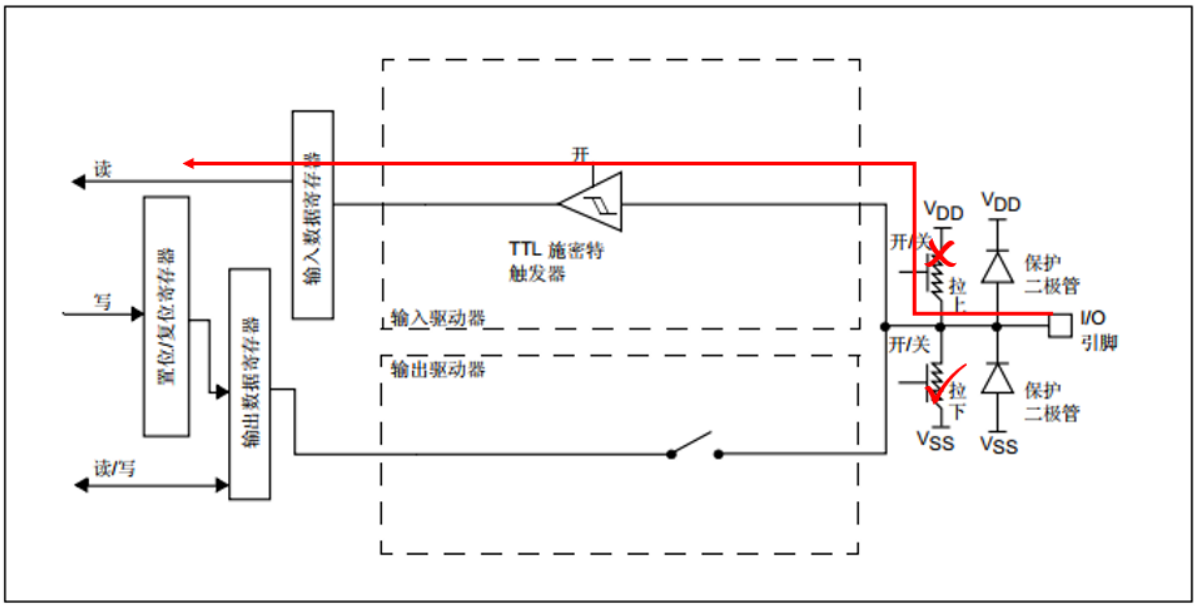



图 4.1.9 下拉输入配置

（6）浮空输入模式（Floating Input）

浮空输入就是 GPIO 内部既无上拉电阻也无下拉电阻。GPIO 内部的上拉电阻和下拉电阻开关都断开。该模式下，引脚默认输入位高阻态，即浮空，不确定。I/O 口电平由外部输入信号决定。

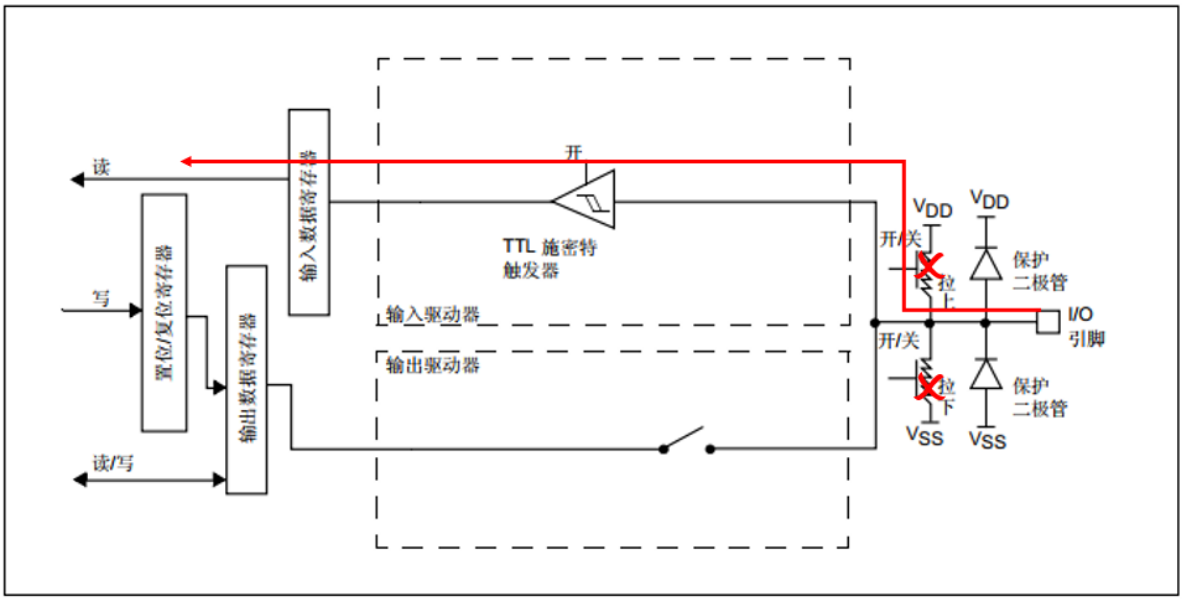


图 4.1.10 浮空输入配置

（7）模拟输入模式（Analog mode）

输入信号不经施密特触发器直接接入，输入信号为模拟量而非数字量，用于外部模拟信号的输入。

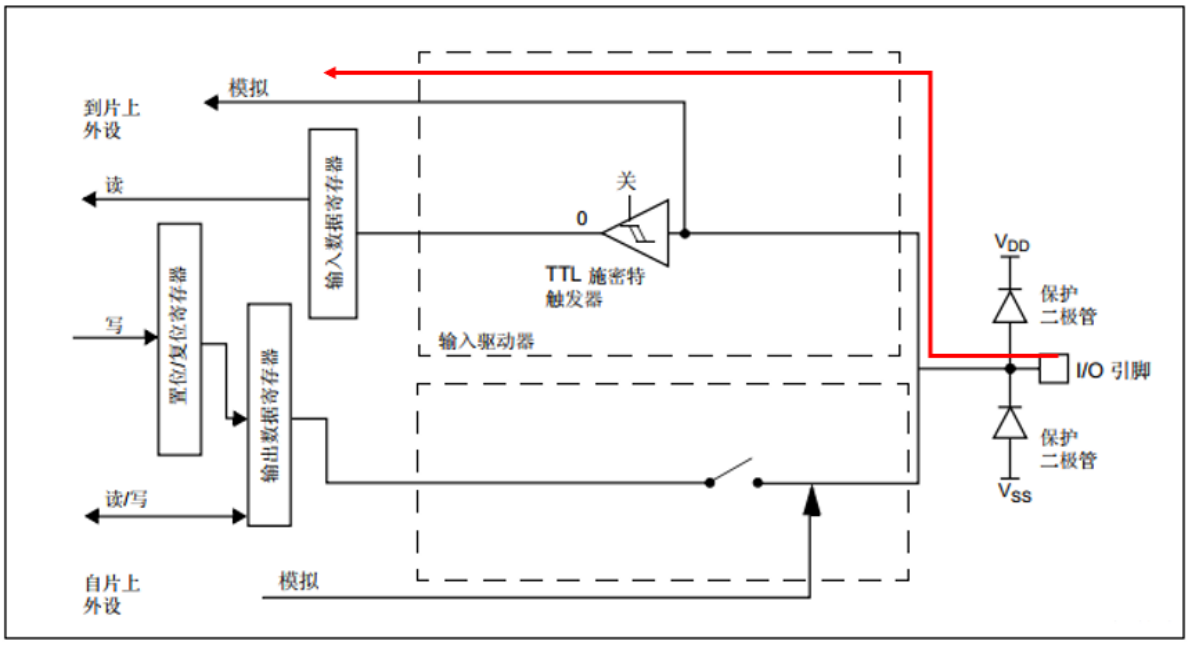



图 4.1.11 模拟输入配置

**6. GPIO 输出速度**

STM32 的 I/O 引脚工作在输出模式下时，需要配置 I/O 引脚的输出速度。该输出速度不是输出信号的速度，而是 I/O 口驱动电路的响应速度。

STM32F4xx 系列的芯片提供四个输出速度：2MHz、25MHz、50MHz 和 100MHz。实际开发中需要结合实际情况选择合适的相应速度，以兼顾信号的稳定性和低功耗。

通常，当设置为高速时，功耗高、噪声大、电磁干扰强；当设备为低速时，功耗低、噪声小、电磁干扰弱。

通常简单的外设，比如 LED 灯、蜂鸣器灯，建议使用较低的输出速度，而复用为 I2C、SPI 等通信信号引脚时，建议使用较高的输出速度以提高响应速度。

### 7. GPIO 相关库函数讲解

（1）GPIO 初始化结构体定义

代码段 4.1.1 GPIO 初始化结构体定义

```
typedef struct
{
    uint32_t            GPIO_Pin;
    GPIOMode_TypeDef    GPIO_Mode;
    GPIOSpeed_TypeDef   GPIO_Speed;
    GPIOOType_TypeDef   GPIO_OType;
    GPIOPuPd_TypeDef    GPIO_PuPd;
}GPIO_InitTypeDef;
```

关于 GPIO 初始化结构体各个成员的详细描述，如表 4.1.1 所示。

表 4.1.1 GPIO 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| GPIO_Pin | GPIO_Pin_0~GPIO_Pin_15<br>GPIO_Pin_All | 指定要配置的 GPIO 引脚 |
| GPIO_Mode | GPIO_Mode_IN：输入模式<br>GPIO_Mode_OUT：输出模式<br>GPIO_Mode_AF：复用模式<br>GPIO_Mode_AN：模拟模式 | 指定所选引脚的运行模式 |
| GPIO_Speed | GPIO_Low_Speed：低速（2MHz）<br>GPIO_Medium_Speed：中速（25MHz）<br>GPIO_Fast_Speed：快速（50MHz）<br>GPIO_High_Speed：高速（100MHz） | 指定所选引脚的速度 |
| GPIO_Otype | GPIO_OType_PP：推挽输出<br>GPIO_OType_OD：开漏输出 | 指定所选引脚的工作输出类型 |
| GPIO_PuPd | GPIO_PuPd_NOPULL：浮空<br>GPIO_PuPd_UP：上拉<br>GPIO_PuPd_DOWN：下拉 | 指定所选引脚的上拉/下拉操作 |

大致的使用方法为：首先定义一个 GPIO 初始化结构体变量，然后根据需求设置该结构体的成员的值。

其示例代码如代码段 4.1.2 所示，后续实验所涉及的有关结构体成员赋值的流程基本一致。

代码段 4.1.2 GPIO 初始化结构体使用示例

```
//GPIO 初始化结构体变量定义
GPIO_InitTypeDef GPIO_InitStructure;

//引脚选择
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
//输出模式
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
//推挽输出
GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
//100MHz
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
//带上拉
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
```

（2）GPIO 初始化函数

表 4.1.2 GPIO 初始化函数

| 接口函数：GPIO_Init | |
|---|---|
| 函数原型 | void GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_InitStruct) |
| 功能描述 | 根据初始化结构体变量 GPIO_InitStruct 中的指定参数初始化 GPIOx 外设。 |
| 入口参数 | GPIOx：指定的 GPIO 端口。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，x 可以为 A~I。 |
| | GPIO_InitStruct：指向 GPIO_InitTypeDef 结构体的指针，该结构体包含 GPIO 指定的外设配置信息。 |
| 返回值 | 无 |
| 使用示例 | GPIO_Init(GPIOA, &GPIO_InitStructure); |
| 注意事项 | 无 |

（3）设置指定 GPIO 引脚的状态函数

表 4.1.3 设置指定 GPIO 引脚的状态函数

| 接口函数：GPIO_WriteBit | |
|---|---|
| 函数原型 | void GPIO_WriteBit(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin, BitAction BitVal) |
| 功能描述 | 将指定 GPIO 引脚设置为指定的状态。 |
| 入口参数 | GPIOx：指定的 GPIO 端口。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，x 可以为 A~I。 |
| | GPIO_Pin：指定的 GPIO 引脚。其范围为 GPIO_Pin_0~GPIO_Pin_15。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，若指定端口为 GPIOI，则其范围为 GPIO_Pin_0~GPIO_Pin_11。 |
| | BitVal：指定 GPIO 引脚的状态。其值可以为：Bit_RESET 或 Bit_SET。 |
| 返回值 | 无 |
| 使用示例 | GPIO_WriteBit(GPIOA, GPIO_Pin_0, Bit_SET);<br>等效于 GPIO_WriteBit(GPIOA, GPIO_Pin_0, (BitAction)1); |
| 注意事项 | 无 |

（4）切换指定 GPIO 引脚的状态函数

表 4.1.4 切换指定 GPIO 引脚的状态函数

| 接口函数：GPIO_ToggleBits | |
|---|---|
| 函数原型 | void GPIO_ToggleBits(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin) |
| 功能描述 | 切换指定 GPIO 引脚的状态。 |
| 入口参数 | GPIOx：指定的 GPIO 端口。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，x 可以为 A~I。 |
| | GPIO_Pin：指定的 GPIO 引脚。其范围为 GPIO_Pin_0~GPIO_Pin_15。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，若指定端口为 GPIOI，则其范围为 GPIO_Pin_0~GPIO_Pin_11。 |
| 返回值 | 无 |
| 使用示例 | GPIO_ToggleBits(GPIOA, GPIO_Pin_0); |
| 注意事项 | 无 |

（5）读取指定 GPIO 输入引脚的状态函数

表 4.1.5 读取指定 GPIO 输入引脚的状态函数

| 接口函数：GPIO_ReadInputDataBit |
|---|

| | |
|---|---|
| 函数原型 | uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef *GPIOx, uint16_t GPIO_Pin) |
| 功能描述 | 读取指定 GPIO 输入引脚的状态，并将状态值返回。 |
| 入口参数 | GPIOx：指定的 GPIO 端口。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，x 可以为 A~I。 |
| | GPIO_Pin：指定的 GPIO 引脚。其范围为 GPIO_Pin_0~GPIO_Pin_15。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，若指定端口为 GPIOI，则其范围为 GPIO_Pin_0~GPIO_Pin_11。 |
| 返回值 | 指定 GPIO 输入引脚的状态。 |
| 使用示例 | uint8_t status = GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0); |
| 注意事项 | 无 |

**8. RCC 相关库函数讲解**

（1）启用或禁用 AHB1 外设时钟函数（AHB2、AHB3、APB1、APB2 使用方法与此类似）

表 4.1.6 启用或禁用 AHB1 外设时钟函数

| 接口函数：RCC_AHB1PeriphClockCmd | |
|---|---|
| 函数原型 | void RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph, FunctionalState NewState) |
| 功能描述 | 启用或禁用 AHB1 外设时钟。 |
| 入口参数 | RCC_AHBPeriph：指定的 AHB1 外设时钟。对于 GPIO 外设，其参数可以为 RCC_AHB1Periph_GPIOx，在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，x 可以为 A~I。 |
| | NewState：指定外设时钟的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA, ENABLE); |
| 注意事项 | 复位后，外设时钟（用于寄存器读/写访问）被禁用，因此需要应用软件在使用之前必须启用该时钟。 |

## 4.1.3 延时函数介绍

在 STM32 中，经常需要用到延时函数，最常遇到的便是微秒级延时和毫秒级延时。

通常情况下，使用循环的方式进行延时，但在某些精度较高的场景中，循环这种延时方法就无法满足其应用，因此需要采用高精度的延时方法。

SysTick 定时器延时就可以满足对高精度要求，其主要分为两种：一种为采用中断的方式；另一种为非中断方式。

本实验指导书则采用的是非中断的方式。

其延时函数实现方法如代码段 4.1.3 所示，主要是对延时初始化、微秒级延时及毫秒级延时的函数实现。

代码段 4.1.3 延时函数实现（SysTick 非中断代码实现）

```
static uint32_t fam_nus = 0;
static uint32_t fam_nms = 0;
/*
功　能：　延时初始化
参　数：　无
返回值：　无
*/
void Delay_Init(void)
{
    //关闭计时、不触发中断、计数时钟为 AHB/8
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8);
    SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;

    //微秒、毫秒级初值
    fam_nus = SystemCoreClock / 8000000;
    fam_nms = fam_nus * 1000;

    //计数器清零
    SysTick->VAL = 0;
}

/*
功　能：　微秒级延时
参　数：　ctr　　微秒数
返回值：　无
说　明：　由于 SysTick->LOAD 为 24 位的寄存器，
          赋值时不要大于 0xffffff/(SystemCoreClock/8000000)，
          对于 SystemCoreClock=168M 的情况下，ctr 应小于 798915us
*/
void Delay_us(uint32_t ctr)
{
```

```
    uint32_t tick_flag = 0;

    //定时器初值(SysTick->LOAD 为 24bit)
    SysTick->LOAD = ctr * fam_nus - 1;
    //计数器清零
    SysTick->VAL = 0x00;
    //开启定时器
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;

    do
    {
        //获取定时器状态
        tick_flag = SysTick->CTRL;
    }
    while (!(tick_flag & (1 << 16)) && (tick_flag & 0x01)); //状态判断

    //关闭计时
    SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;
    SysTick->VAL = 0x00;
}

/*
功   能:  毫秒级延时
参   数:  ctr     毫秒数
返回值:  无
说   明:  由于 SysTick->LOAD 为 24 位的寄存器,
          赋值时不要大于 0xffffff/(SystemCoreClock/8000000*1000),
          对于 SystemCoreClock=168M 的情况下，ctr 应小于 798ms.
          对于大于 798ms 的情况，可以多次调用本函数进行数值叠加,
          也可将本函数的内容全部注释，采用函数末端所提供的 while 循环即可（随着时间的叠加，可能会有较
大误差）
*/
void Delay_ms(uint32_t ctr)
{
    uint32_t tick_flag = 0;

    //定时器初值(SysTick->LOAD 为 24bit)
    SysTick->LOAD = (uint32_t)ctr * fam_nms - 1;
    //计数器清零
    SysTick->VAL = 0x00;
    //开启定时器
```

```
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;

    do
    {
        //获取定时器状态
        tick_flag = SysTick->CTRL;

    }
    while ((tick_flag & 0x01) && !(tick_flag & (1 << 16))); //状态判断

    //关闭计时
    SysTick->CTRL &= ~SysTick_CTRL_ENABLE_Msk;
    SysTick->VAL = 0X00;

    /*
    while(ctr--)
    {
        Delay_us(1000);
    }
    */
}
```

---

需要注意的是，由于 SysTick->LOAD 为 24 位的寄存器，其最大值为 0xFFFFFF，即 16777215，微秒函数中对该寄存器赋值为：（SysTick->LOAD=ctr*fam_nus-1;），又已知 fam_nus=SystemCoreClock/8000000（SystemCoreClock 在 STM32F40xxx 或 41xxx 中为 168M），带入可得 ctr 最大值为 798915us，因此在赋值的时候，微秒函数 Delay_us(uint32_t ctr)中的 ctr 不能超过该最大值。

同样的，通过计算可以得出毫秒延时函数 Delay_ms(uint32_t ctr)中的 ctr 不能大于 798ms。而对于大于 798ms 的情况，可以多次调用 Delay_ms 函数进行数值叠加，也可将此函数的内容全部注释，采用此函数末端所提供的 while 循环即可（随着时间的叠加，可能会有较大误差）。

## 4. 1. 4 硬件设计

**1. LED 灯驱动原理**

如图 4.1.12 所示，发光二极管正向导通则 LED 灯被点亮。

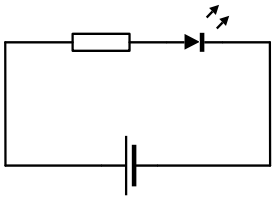


图 4.1.12 LED 灯正向导通电路示意图

如图 4.1.13 所示，微控制驱动 LED 灯的电路模型包括：控制器、驱动器和执行器三部分。其中控制器提供控制信号，再由驱动器驱动执行器。

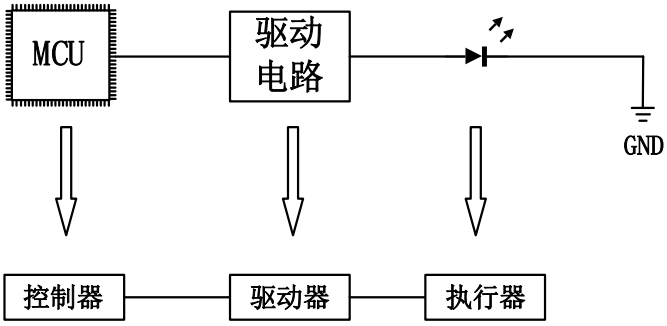



图 4.1.13 微控制器驱动电路模型

如图 4.1.14 所示 LED 灯通过 N-MOS 管驱动，MCU I/O 输出高电平时 N-MOS 管漏极和源极导通 LED 灯被点亮，反之 MCU 输出低电平时 N-MOS 管漏极和源极截止，LED 灯不能被点亮。LED 驱动电路中为避免损坏器件还增加了电阻（R），以此保证通过 LED 灯的电流不超过额定电流，故将电阻 R 称为限流电阻。

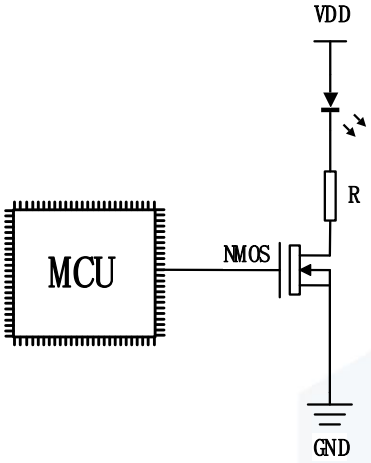



图 4.1.14 N-MOS 管驱动 LED 灯电路

**2. 实验电路分析**

根据图 4.1.14 所展示的 LED 驱动电路，可以分析出 LED 灯的电路是由 N-MOS 管驱动的，而 N-MOS 管漏极和源极的导通和截止则是由 MCU 所对应的 GPIO 引脚控制

的。由此可以得出，通过控制 MCU 相应的 GPIO 引脚输出高低电平，即可控制对应 LED 的亮灭状态。

在图 4.1.15 中，MP_LED1、MP_LED2、MP_LED3 和 MP_LED4 对应的 GPIO 引脚分别为 PF9、PF10、PH14 和 PH15，根据对 LED 驱动电路的分析，可以得出 GPIO 引脚 PF9、PF10、PH14 和 PH15 控制的 LED 灯分别为 D12、D13、D14 和 D15（常以 LED1、LED2、LED3、LED4 来表示）。

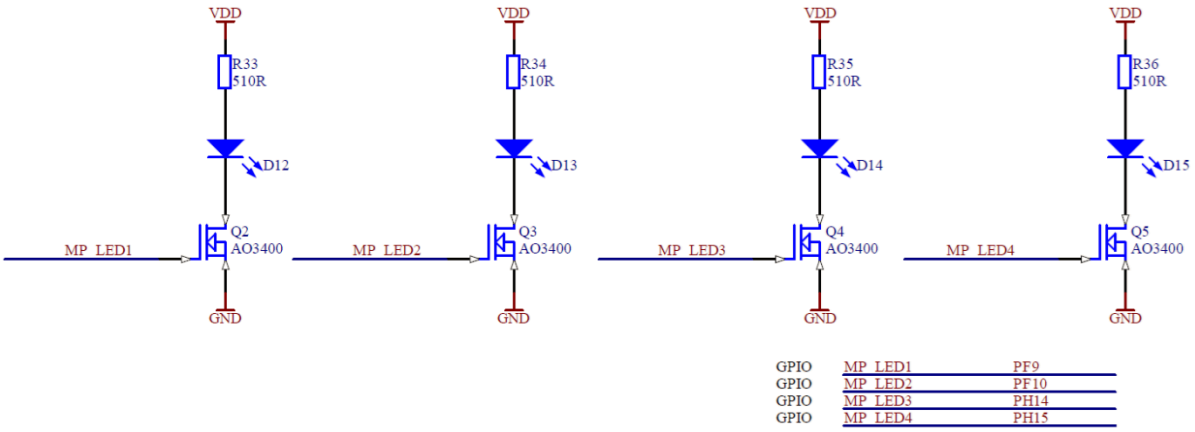



图 4.1.15 LED 灯电路原理图

## 4. 1. 5 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 LED 部分的驱动代码独立开来，源文件（led.c）和头文件（led.h）存放在实验案例根目录中的 HARDWARE/LED 下。

**1. 软件设计流程**

如图 4.1.16，展示了本实验的软件设计流程图，它是展示了整体程序的设计流程，能够反映程序设计的通用思维。

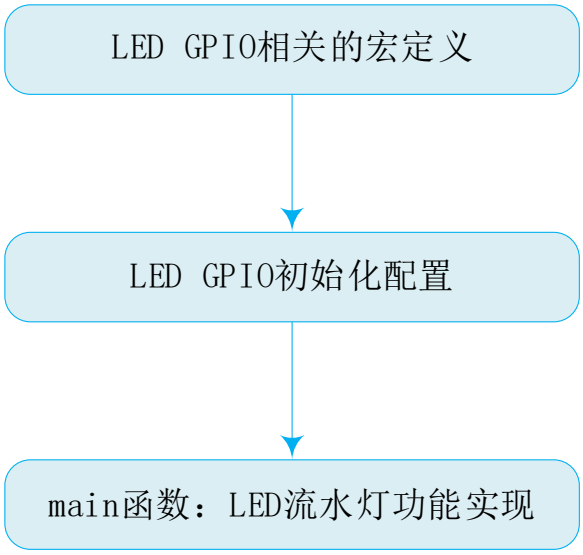



图 4.1.16 软件设计流程图

**2. 软件代码讲解**

（1）定义 LED GPIO 引脚和接口相关的宏，其一是为了当硬件中对应的 GPIO 发生变化的时候，只需修改少量的宏定义值即可完成整个实验项目的修改，增强了程序的可移植性；其二是通过宏定义，能让代码的结构清晰易懂，方便读者理解。

代码段 4.1.4 引脚宏定义（led.h）

```
//LED 对应的 GPIO 时钟挂载在 AHB1 上
#define LED_ALL_CLOCK        RCC_AHB1Periph_GPIOF | RCC_AHB1Periph_GPIOH
//LED 对应的 GPIO 时钟使能，其为宏函数定义
#define LED_CLK_INIT         RCC_AHB1PeriphClockCmd
/***************LED 端口引脚定义***************/
//LED1 --- PF9
#define LED1_PORT            GPIOF
#define LED1_PIN             GPIO_Pin_9
//LED2 --- PF10
#define LED2_PORT            GPIOF
#define LED2_PIN             GPIO_Pin_10

//LED3 --- PH14
#define LED3_PORT            GPIOH
#define LED3_PIN             GPIO_Pin_14

//LED4 -- PH15
#define LED4_PORT            GPIOH
#define LED4_PIN             GPIO_Pin_15
```

```
/***************控制 LED 亮灭状态***************/
//LED 设置灯的亮灭状态，BitAction 就是一个枚举类型的变量,表示这个位是高电平还是低电平
#define LED1(X)             GPIO_WriteBit(LED1_PORT, LED1_PIN, (BitAction)X);
#define LED2(X)             GPIO_WriteBit(LED2_PORT, LED2_PIN, (BitAction)X);
#define LED3(X)             GPIO_WriteBit(LED3_PORT, LED3_PIN, (BitAction)X);
#define LED4(X)             GPIO_WriteBit(LED4_PORT, LED4_PIN, (BitAction)X);
#define LED_ALL(X)          LED1(X);LED2(X);LED3(X);LED4(X);
//LED 设置灯的亮灭状态的切换
#define LED1_TOGGLE()       GPIO_ToggleBits(LED1_PORT, LED1_PIN);
#define LED2_TOGGLE()       GPIO_ToggleBits(LED2_PORT, LED2_PIN);
#define LED3_TOGGLE()       GPIO_ToggleBits(LED3_PORT, LED3_PIN);
#define LED4_TOGGLE()       GPIO_ToggleBits(LED4_PORT, LED4_PIN);
#define LED_ALL_TOGGLE()    LED1_TOGGLE();LED2_TOGGLE();LED3_TOGGLE();LED4_TOGGLE();
```

（2）对 LED GPIO 引脚进行初始化，初始化分为四个过程：首先定义 GPIO 初始化的结构体；使能 GPIO 时钟；然后设置 GPIO 的相关参数（输入输出模式，端口的速度，带上拉还是下拉等）；最后调用初始化函数对 GPIO 进行初始化。

代码段 4.1.5 引脚初始化函数（led.c）

```
/*
功   能:  LED GPIO 初始化
参   数:  无
返回值:  无
*/
void Led_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;                //定义 GPIO 初始化结构体变量
    LED_CLK_INIT(LED_ALL_CLOCK, ENABLE);                //开启 LED 相关的 GPIO 时钟
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;       //输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;      //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;   //25MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;        //带上拉

    GPIO_InitStructure.GPIO_Pin = LED1_PIN;             //选择 LED1 GPIO 引脚
    GPIO_Init(LED1_PORT, &GPIO_InitStructure);          //初始化 LED1 GPIO

    GPIO_InitStructure.GPIO_Pin = LED2_PIN;             //选择 LED2 GPIO 引脚
    GPIO_Init(LED2_PORT, &GPIO_InitStructure);          //初始化 LED2 GPIO
```

```
    GPIO_InitStructure.GPIO_Pin = LED3_PIN;             //选择 LED3 GPIO 引脚
    GPIO_Init(LED3_PORT, &GPIO_InitStructure);          //初始化 LED3 GPIO

    GPIO_InitStructure.GPIO_Pin = LED4_PIN;             //选择 LED4 GPIO 引脚
    GPIO_Init(LED4_PORT, &GPIO_InitStructure);          //初始化 LED4 GPIO
}
```

（3）主函数代码实现。首先需要在 while 循环之前进行必要的初始化，包括对延时函数和相关外设进行设置。然后，在 while 循环中则需要实现流水灯功能，通过适当的控制流程和延时，进而控制 LED 的亮灭状态。

代码段 4.1.6 主函数（main.c）

```
int main(void)
{
    Delay_Init();       //延时初始化
    Led_Init();         //LED GPIO 初始化

    while(1)
    {
        //LED 流水灯实现
        LED1(1);
        Delay_ms(100);
        LED1(0);

        LED2(1);
        Delay_ms(100);
        LED2(0);

        LED3(1);
        Delay_ms(100);
        LED3(0);

        LED4(1);
        Delay_ms(100);
        LED4(0);
    }
}
```

### 4.1.6 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器。

| PC | STM32F4 核心板 | 12V 电源适配器 | ST Link 下载器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

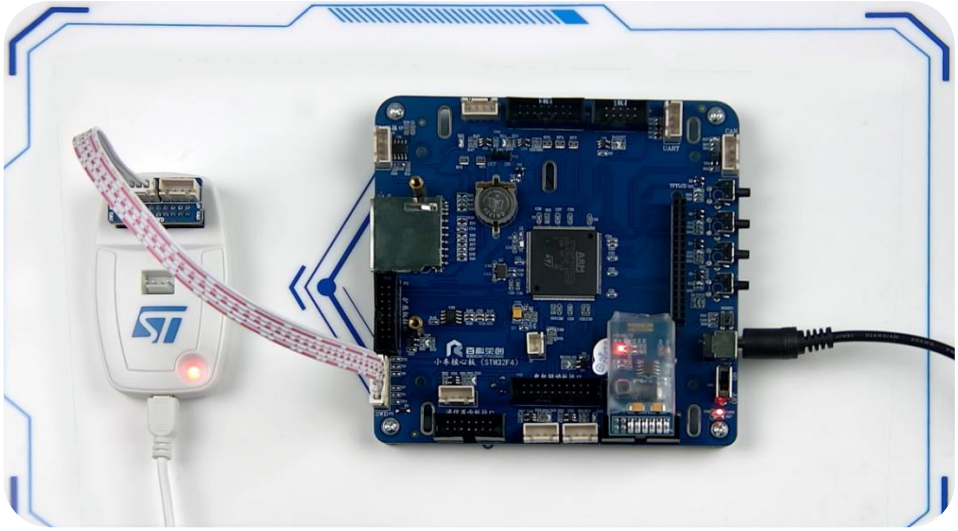


图 4.1.17 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，并观察核心板 LED 灯出现的实验现象。

### 4.1.7 结果展示

程序下载至核心板后，观察到核心板上 LED1~LED4 间隔 100ms 依次点亮熄灭，实现了流水灯效果。

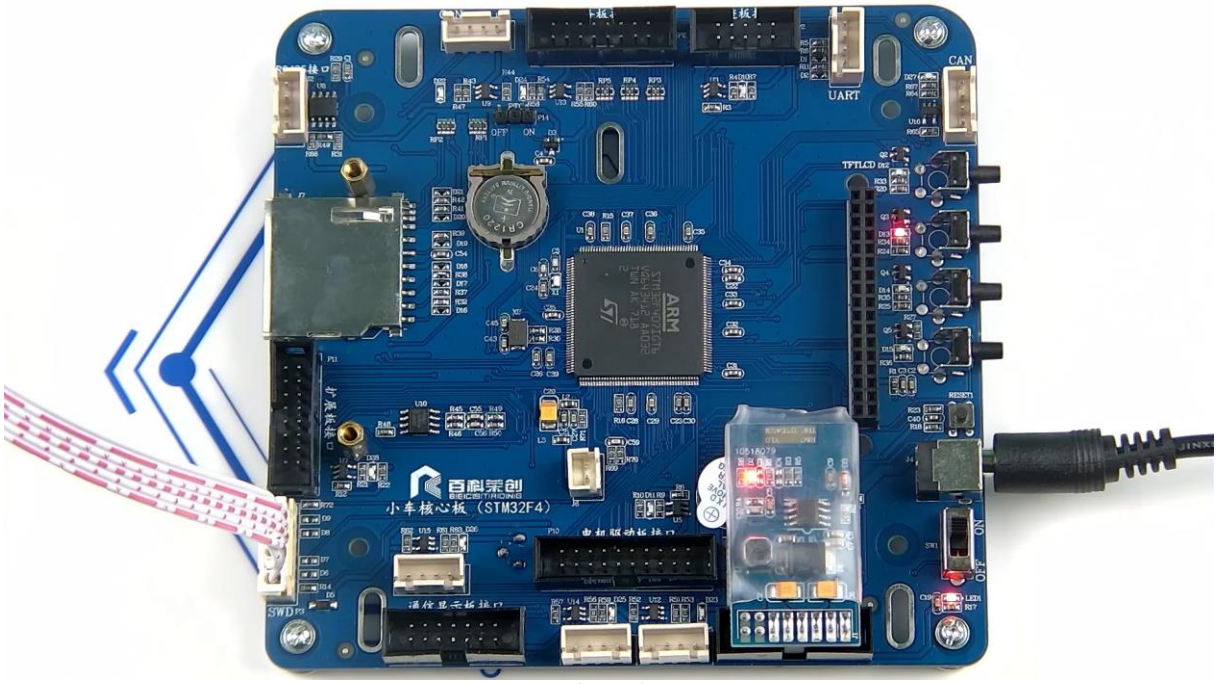



图 4.1.18 实验效果图展示

# 4.2 按键输入实验

在本实验中，需要对按键状态进行检测，以实现对 LED 灯亮灭状态的控制。

通过本实验，读者将学会如何使用 GPIO 作为输入口，并进一步掌握 STM32 的 GPIO 引脚的特性和功能，为后续其他外设的应用打下坚实的基础。

## 4.2.1 按键介绍

**1. 常用按键概念**

按键本质上就是一个开关，这个开关的下部还有一个用于复位的弹簧。按下按键时可以导通 A 端与 B 端；松开后弹簧又将按键弹开保持电路开路状态，按键开闭状态如图 4.2.1 所示：

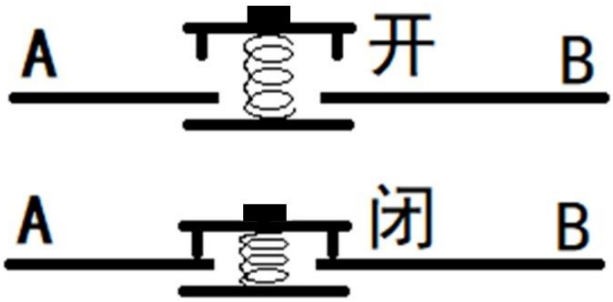



图 4.2.1 按键开闭状态

**2. 按键抖动**

按键开关为机械弹性开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个按键开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开。因而在闭合及断开的瞬间均伴随有一连串的抖动。抖动时间的长短由按键的机械特性决定，一般为 5ms～10ms。

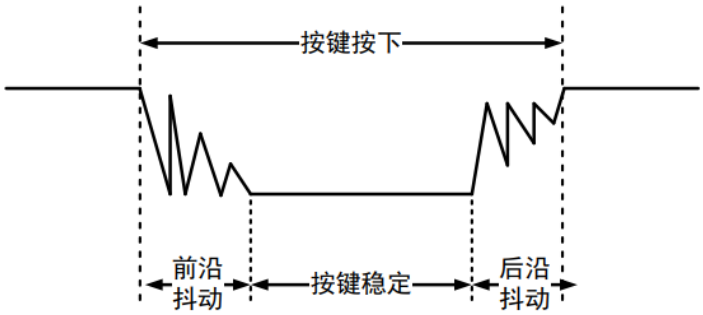



图 4.2.2 机械按键抖动示意图

按键稳定的时间的长短则是由操作人员的按键动作决定的，一般为零点几秒至数

秒。按键抖动会引起按键被误读多次。为确保微控制器在按下按键时仅作一次处理，必须对按键进行消抖。

**3. 按键消抖**

按键消抖可以硬件上处理，即在按键上并联一个电容，利用电容的充放电特性来对抖动过程中产生的电压毛刺进行平滑处理，从而实现消抖。

也可以软件处理，即通过延时，避开抖动。

但实际应用中，硬件消抖的方式效果并不是很好，而且还增加了成本和电路复杂度，因此在大多数情况下，还是用的软件消抖。

### 4.2.2 硬件设计

**1. 按键驱动原理**

按键在 MCU 中一般占用一个 GPIO，通过检测该 GPIO 的电平变化即可知晓按键是否被按下，常见的电路连接如图 4.2.3 和图 4.2.4 所示。

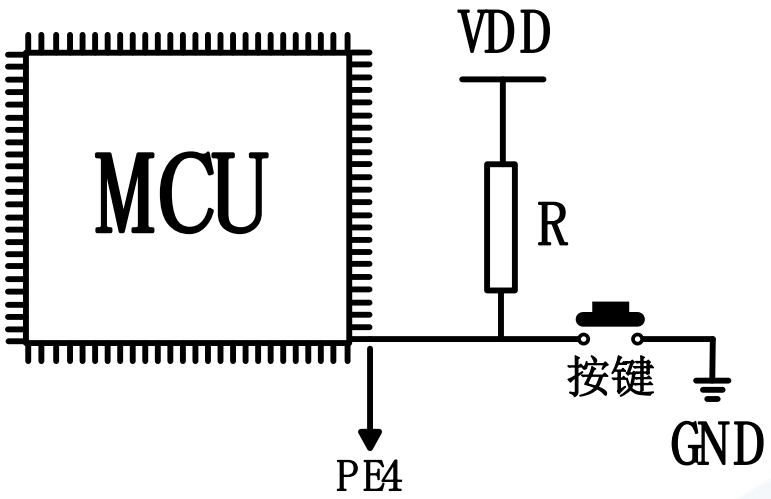



图 4.2.3 按键与 MCU 连接示意图 1

由图 4.2.3 可知，以 PE4 引脚为例（实际需参照原理图分析），当没有按键按下时，电源通过电阻直接连接到了 MCU 的 PE4 引脚，此时 PE4 处读取的电平即为高电平；而当按键按下时，此时电源通过电阻再到按键直接接地，此时 PE4 处读取的电平即为低电平。

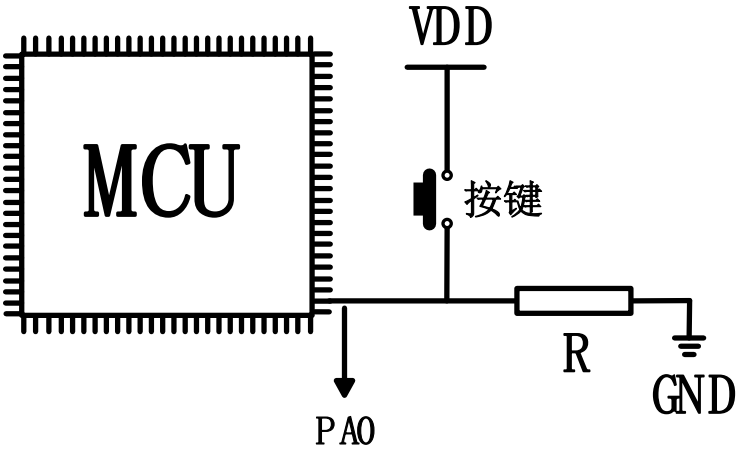



图 4.2.4 按键与 MCU 连接示意图 2

相反，以 PA0 引脚为例（实际需参照原理图分析），如图 4.2.4，当没有按键按下时，PA0 引脚通过电阻直接接地，此时 PA0 处读取的电平即为低电平；而当按键按下时，此时 PA0 引脚与电源相连接，则 PA0 处读取的电平即为高电平。

**2. 微控制器 I/O 输入检测原理**

如图 4.2.5 所示，微控制器 I/O 输入具备一套上下拉的功能以及保护机制，包括输入驱动器的 TTL 施密特触发器。

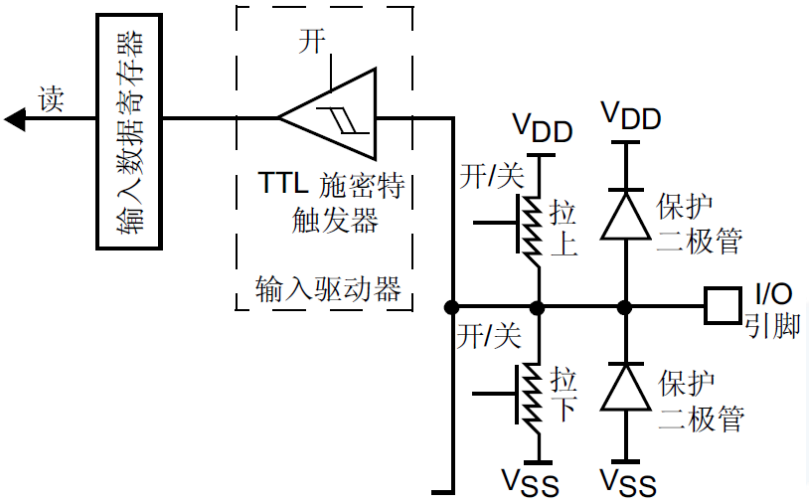



图 4.2.5 基础 I/O 的输入检测结构

对于 TTL 施密特触发器，如当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；当输入在正负向阈值电压之间，输出不改变，也就是说输出由高电准位翻转为低电准位，或是由低电准位翻转为高电准位时所对应的阈值电压是不同的。只有当输入电压发生足够的变化时，输出才会变化，因此将这种元

件命名为触发器。这种双阈值动作被称为迟滞现象，表明施密特触发器有记忆性。从本质上来说，施密特触发器是一种双稳态多谐振荡器。施密特触发器输出的信号时钟是数字信号且会被微控制器处理成正确的输入电平状态。

**3. 实验电路分析**

如图 4.2.6 所示，显示了核心板中按键的电路原理图，其中按键 MP_S1、MP_S2、MP_S3 和 MP_S4 分别对应 GPIO 引脚 PE4、PE3、PE2 和 PA0。

（注：MP_S1、MP_S2、MP_S3 和 MP_S4 常以 KEY1、KEY2、KEY3 和 KEY4 代替）。

其中 PE4、PE3、PE2 对应图 4.2.3 的驱动原理（按下为低电平），PA0 则对应图 4.2.4 的驱动原理（按下为高电平），在编写程序时要注意这一点。

当其中某一个按键按下时，通过读取对应 GPIO 引脚的高低电平状态即可判断是否被按下。

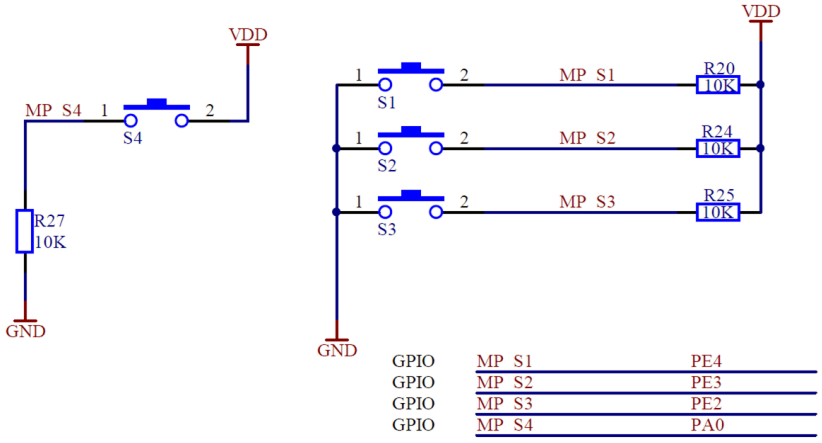



图 4.2.6 按键电路原理图

## 4. 2. 3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将按键部分的驱动代码独立开来，源文件（key.c）和头文件（key.h）存放在实验案例根目录中的 HARDWARE/KEY 下。

**1. 软件设计流程**

如图 4.2.7，展示了本实验的软件设计流程图。

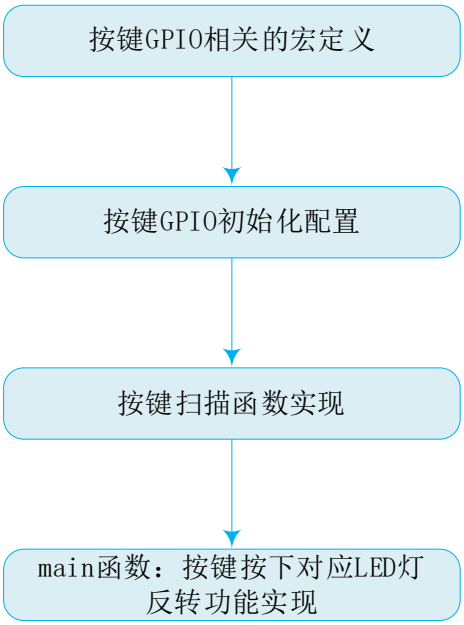



图 4.2.7 软件设计流程图

**2. 软件代码讲解**

（1）定义按键 GPIO 引脚和接口相关的宏。

代码段 4.2.1 引脚宏定义（key.h）

```
//按键对应的 GPIO 时钟挂载在 AHB1 上
#define KEY_ALL_CLOCK    RCC_AHB1Periph_GPIOE | RCC_AHB1Periph_GPIOA

//按键对应的 GPIO 时钟使能，其为宏函数定义
#define KEY_CLK_INIT     RCC_AHB1PeriphClockCmd
/***************按键端口引脚定义***************/
//KEY1 --- PE4
#define KEY1_PORT        GPIOE
#define KEY1_PIN         GPIO_Pin_4

//KEY2 --- PE3
#define KEY2_PORT        GPIOE
#define KEY2_PIN         GPIO_Pin_3

//KEY3 --- PE2
#define KEY3_PORT        GPIOE
#define KEY3_PIN         GPIO_Pin_2
```

```
//KEY4 --- PA0
#define KEY4_PORT       GPIOA
#define KEY4_PIN        GPIO_Pin_0

/***************读取按键 GPIO 引脚状态***************/
#define KEY1()          GPIO_ReadInputDataBit(KEY1_PORT, KEY1_PIN)
#define KEY2()          GPIO_ReadInputDataBit(KEY2_PORT, KEY2_PIN)
#define KEY3()          GPIO_ReadInputDataBit(KEY3_PORT, KEY3_PIN)
#define KEY4()          GPIO_ReadInputDataBit(KEY4_PORT, KEY4_PIN)
```

（2）对按键 GPIO 引脚进行初始化。需要注意的是：KEY1、KEY2 和 KEY3 按键的驱动原理如图 4.2.3 所示，因此需将其工作模式设置为上拉输入模式；而 KEY4 按键的驱动原理如图 4.2.4 所示，因此需将其工作模式设置为下拉输入模式。

代码段 4.2.2 引脚初始化函数（key.c）

```
/*
功   能：  按键初始化函数
参   数：  无
返回值：  无
*/
void Key_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;                //定义 GPIO 初始化结构体变量

    KEY_CLK_INIT(KEY_ALL_CLOCK, ENABLE);                //开启按键相关的 GPIO 时钟

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;        //输入模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;        //带上拉

    GPIO_InitStructure.GPIO_Pin = KEY1_PIN;             //选择 KEY1 GPIO 引脚
    GPIO_Init(KEY1_PORT, &GPIO_InitStructure);          //初始化 KEY1 GPIO

    GPIO_InitStructure.GPIO_Pin = KEY2_PIN;             //选择 KEY2 GPIO 引脚
    GPIO_Init(KEY2_PORT, &GPIO_InitStructure);          //初始化 KEY2 GPIO

    GPIO_InitStructure.GPIO_Pin = KEY3_PIN;             //选择 KEY3 GPIO 引脚
    GPIO_Init(KEY3_PORT, &GPIO_InitStructure);          //初始化 KEY3 GPIO

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;        //输入模式
```

```
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;      //带下拉

    GPIO_InitStructure.GPIO_Pin = KEY4_PIN;             //选择 KEY4 GPIO 引脚
    GPIO_Init(KEY4_PORT, &GPIO_InitStructure);          //初始化 KEY4 GPIO
}
```

（3）按键功能函数实现。主要是按键扫描函数的实现，其具体功能是检测按键状态并返回哪个按键被按下了。在这个过程中，还实现了按键的软件消抖，以确保按键信号的稳定性。

代码段 4.2.3 按键功能函数（key.c）

```
/*
功   能:  按键扫描函数
参   数:  mode    0,不支持连续按
                  1,支持连续按
返回值:  0，没有任何按键按下
         1，KEY1 按下
         2，KEY2 按下
         3，KEY3 按下
         4，KEY4 按下
         注意此函数有响应优先级,KEY1>KEY2>KEY3>KEY4!!!
*/
uint8_t Key_Scan(uint8_t mode)
{
    static uint8_t key_up = 1;                          //按键按松开标志

    if(mode) key_up = 1;                                //支持连按

    //判断按键是否被按下
    if(key_up&&(!KEY1() || !KEY2() || !KEY3()|| KEY4()))
    {
        Delay_ms(10);                                   //软件延时去抖动
        key_up=0;                                       //按键被按下

        if     (KEY1() == 0)    return 1;
        else if(KEY2() == 0)    return 2;
        else if(KEY3() == 0)    return 3;
        else if(KEY4() == 1)    return 4;
```

```
    }else if(KEY1() && KEY2() && KEY3() && !KEY4())
    {
        key_up = 1;                                     //按键松开
    }

    return 0;                                           //无按键按下
}
```

（4）主函数代码实现。首先需要在 while 循环之前进行必要的初始化，包括对延时函数和相关外设进行设置。然后，在 while 循环中则需要实现按键控制对应 LED 反转的功能。

代码段 4.2.4 主函数（main.c）

```
int main(void)
{
    Delay_Init();               //延时初始化
    Led_Init();                 //LED 初始化
    Key_Init();                 //按键初始化

    while(1)
    {
        uint8_t key_value = 0;

        //确定是哪个按键被按下
        key_value = Key_Scan(0);

        switch(key_value)
        {
            case 1:
                LED1_TOGGLE();
                break;
            case 2:
                LED2_TOGGLE();
                break;
            case 3:
                LED3_TOGGLE();
                break;
            case 4:
                LED4_TOGGLE();
```

```
                break;
            default:
                break;
        }
    }
}
```

---

## 4.2.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器。

| PC | STM32F4 核心板 | 12V 电源适配器 | ST Link 下载器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

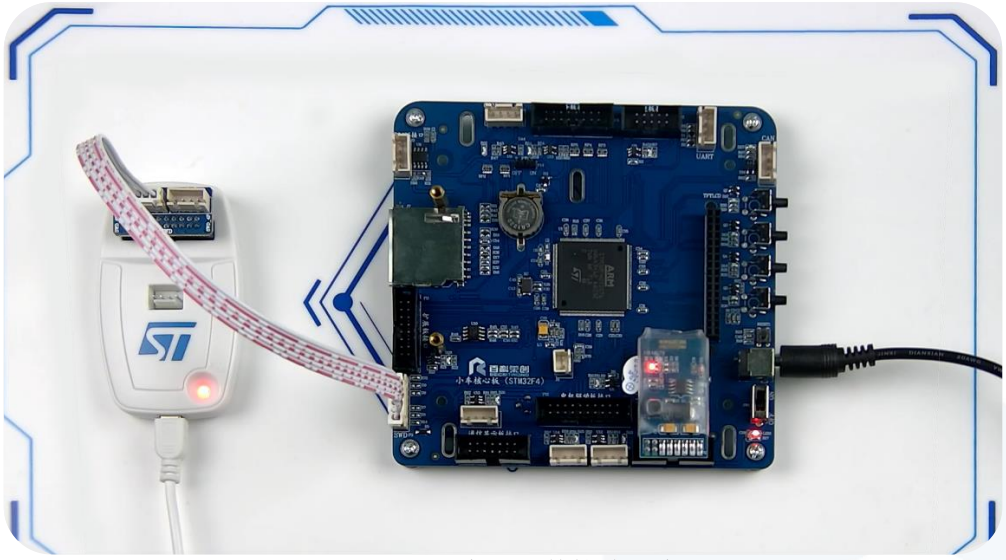


图 4.2.8 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，通过按下不同按键，观察对应 LED 灯的亮灭状态。

### 4.2.5 结果展示

将程序下载到核心板后，按下按键 KEY1，LED1 状态取反；按下按键 KEY2，LED2 状态取反；按下按键 KEY3，LED3 状态取反；按下按键 KEY4，LED4 状态取反。

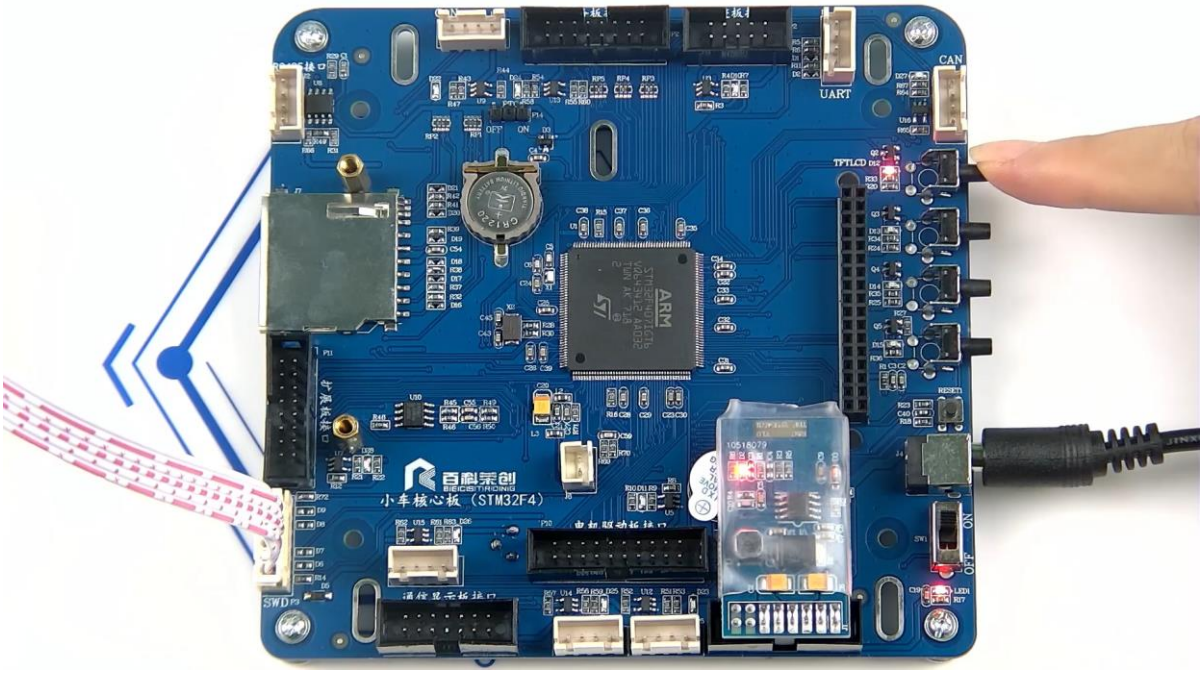



图 4.2.9 实验效果图展示

# 4.3 蜂鸣器报警实验

本实验通过使用 GPIO 控制蜂鸣器，旨在加深对 GPIO 驱动原理的理解。

## 4.3.1 蜂鸣器介绍

**1. 蜂鸣器概念**

蜂鸣器是一种一体化结构的电子讯响器，采用直流电压供电，广泛应用于计算机、打印机、复印机、报警器、电子玩具、汽车电子设备、电话机、定时器等电子产品中。蜂鸣器主要分为压电式的有源蜂鸣器和电磁式的无源蜂鸣器两种类型。如图 4.3.1 所示，分别展示了有源蜂鸣器和无源蜂鸣器。

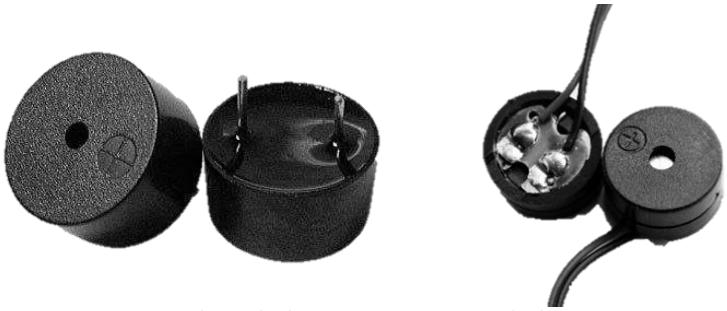


图 4.3.1 有源蜂鸣器（左）与无源蜂鸣器（右）

**2. 蜂鸣器结构和发声原理**

压电式的有源蜂鸣器主要由多谐振荡器、压电蜂鸣片、阻抗匹配器及共鸣箱、外壳等组成。多谐振荡器由晶体管或集成电路构成。当接通电源后（1.5~15V 直流工作电压）,直流电源输入经过振荡系统的放大取样电路在谐振装置作用下产生声音信号，有源蜂鸣器的工作发声如图 4.3.2 所示。

图 4.3.2 有源蜂鸣器的驱动原理

而电磁式的无源蜂鸣器由振荡器、电磁线圈、磁铁、振动膜片及外壳等组成。接通电源后，振荡器产生的音频信号电流通过电磁线圈，使电磁线圈产生磁场。振动膜片在电磁线圈和磁铁的相互作用下周期性的振动发声，如图 4.3.3 所示。

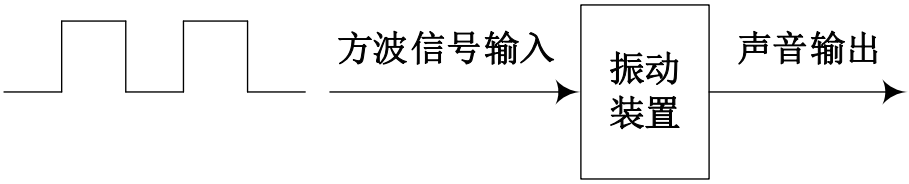



图 4.3.3 无源蜂鸣器的驱动原理

“有源”与“无源”的“源”不是指电源，而是指振荡源。也就是说，有源蜂鸣器内部带振荡源，通电就能起振并鸣叫。而无源蜂鸣器内部不带振荡源，所以如果用直流信号是无法令其鸣叫，只有用方波去驱动它。又由于人的耳朵无法听到次声波与超声波，所以驱动无源蜂鸣器频率的合适范围为 20~20kHz。

## 4.3.2 硬件设计

**1. 蜂鸣器驱动原理**

如图 4.3.4 所示，蜂鸣器通过 N-MOS 管驱动，MCU IO 输出高电平时 MOS 管漏极和源极导通，蜂鸣器发出声音，反之 MCU 输出低电平时 MOS 管漏极和源极截止，蜂鸣器停止发声。

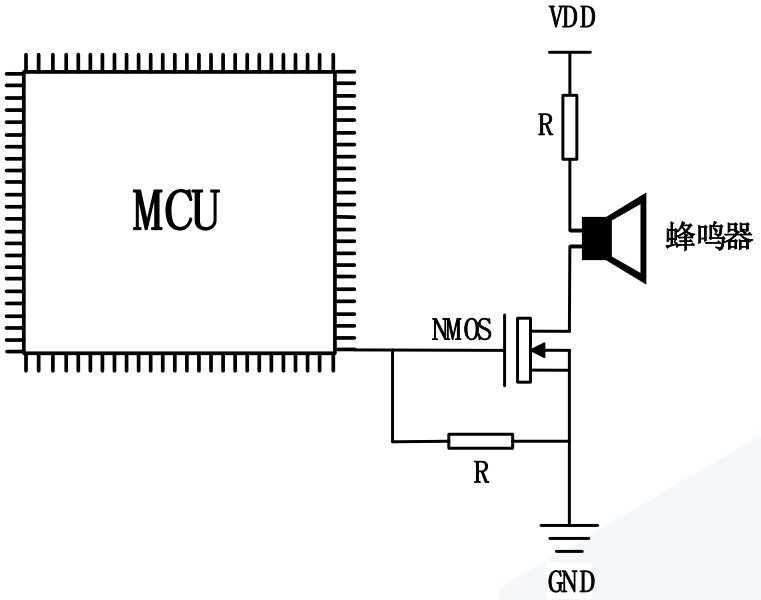



图 4.3.4 蜂鸣器的驱动原理

**2. 实验电路分析**

如图 4.3.5 所示，显示了核心板中蜂鸣器的电路原理图，其中控制蜂鸣器的 GPIO 引脚为 PH7。因此只需控制 GPIO 引脚输出高低电平，即可控制蜂鸣器发声和停止发声。

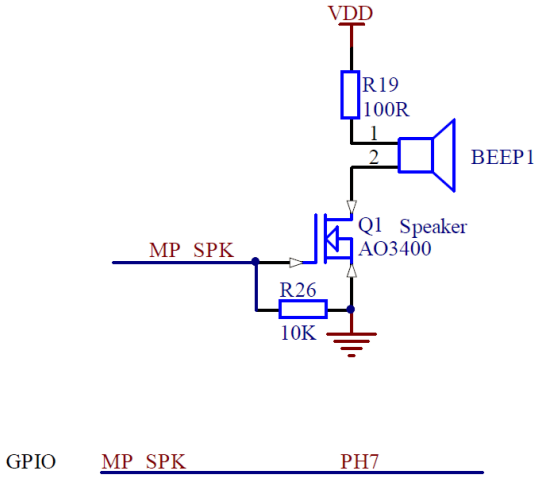



图 4.3.5 蜂鸣器电路原理图

## 4. 3. 3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将蜂鸣器部分的驱动代码独立开来，源文件（beep.c）和头文件（beep.h）存放在实验案例根目录中的 HARDWARE/BEEP 下。

**1. 软件设计流程**

如图 4.3.6，展示了本实验的软件设计流程图。

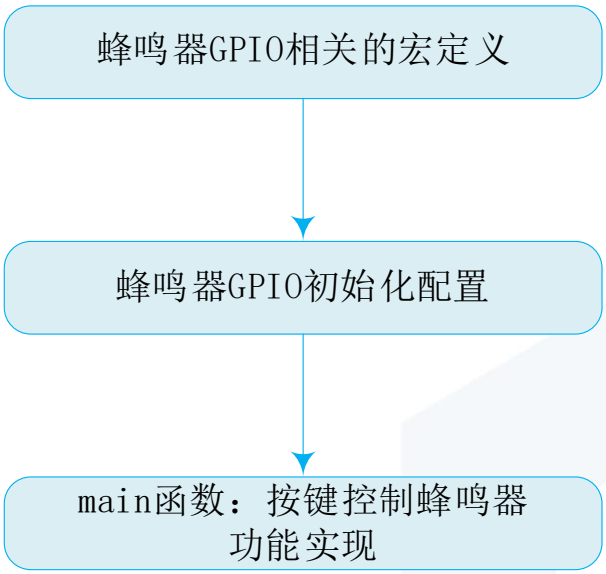



图 4.3.6 软件设计流程图

**2. 软件代码讲解**

（1）定义蜂鸣器 GPIO 引脚和接口相关的宏。

代码段 4.3.1 引脚宏定义（beep.h）

```
//蜂鸣器对应的 GPIO 时钟挂载在 AHB1 上
#define BEEP_CLOCK       RCC_AHB1Periph_GPIOH
//蜂鸣器对应的 GPIO 时钟使能，其为宏函数定义
#define BEEP_CLK_INIT    RCC_AHB1PeriphClockCmd

//蜂鸣器端口
#define BEEP_PORT        GPIOH
//蜂鸣器引脚
#define BEEP_PIN         GPIO_Pin_7
//控制蜂鸣器 GPIO 引脚状态
#define BEEP(X)          GPIO_WriteBit(BEEP_PORT, BEEP_PIN, (BitAction)X)
```

（2）对蜂鸣器 GPIO 引脚进行初始化。

代码段 4.3.2 引脚初始化函数（beep.c）

```
/*
功　能：  BEEP IO 初始化
参　数：  无
返回值：  无
*/
void Beep_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    BEEP_CLK_INIT(BEEP_CLOCK, ENABLE);                  //使能 GPIO 时钟

    GPIO_InitStructure.GPIO_Pin = BEEP_PIN;             //初始化蜂鸣器对应引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;       //输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;      //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;   //25MHz
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;      //下拉
    GPIO_Init(BEEP_PORT, &GPIO_InitStructure);          //初始化 GPIO
}
```

（3）主函数代码实现。首先需要在 while 循环之前进行必要的初始化，包括对延时函数和相关外设进行设置。然后，在 while 循环中则需要实现按键 KEY1 控制蜂鸣器发声且 LED1 闪烁一下的功能。

代码段 4.3.3 主函数（main.c）

```
int main(void)
{
    Delay_Init();        //延时初始化
    Led_Init();          //LED 初始化
    Key_Init();          //按键初始化
    Beep_Init();         //蜂鸣器初始化

    while(1)
    {
        uint8_t key_value = 0;

        //确定是哪个按键被按下
        key_value = Key_Scan(0);

        switch(key_value)
        {
            case 1:
                LED1(1);
                BEEP(1);
                Delay_ms(100);

                LED1(0);
                BEEP(0);
                Delay_ms(100);
                break;
            default:
                break;
        }
    }
}
```

### 4.3.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器。

| PC | STM32F4 核心板 | 12V 电源适配器 | ST Link 下载器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

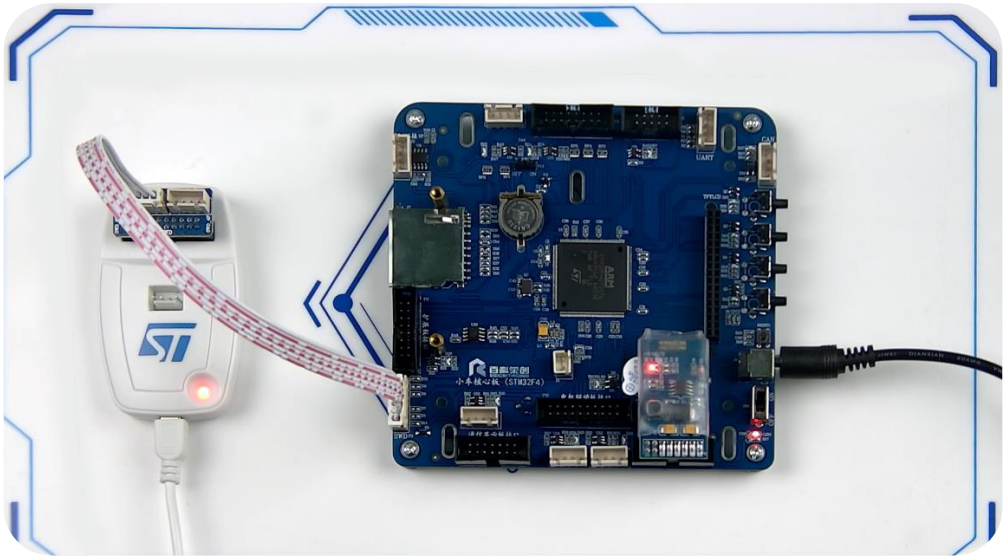


图 4.3.7 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，通过按下按键检测蜂鸣器是否响。

### 4.3.5 结果展示

将程序下载到核心板后，按下按键 KEY1，蜂鸣器响一声，LED1 灯闪烁一下。

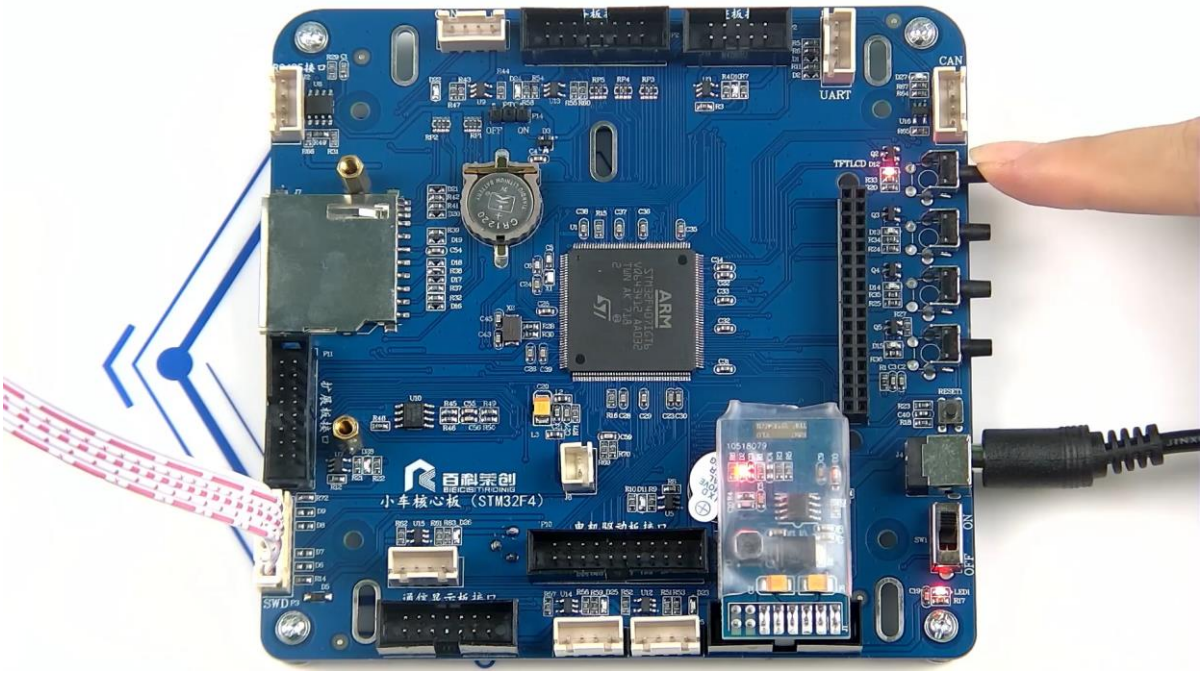



图 4.3.8 实验效果图展示

# 4.4 外部中断按键输入实验

本实验旨在理解并学习 STM32 中外部中断的概念和作用。在嵌入式应用中，中断具有重要地位，几乎每个控制器都支持中断功能。

我们通过设计按键作为触发源，使控制器产生中断，并在中断服务函数中控制按键对应的 LED 产生闪烁效果。

相对于通过轮询方式获取按键状态，使用中断可以提高效率。

## 4.4.1 中断介绍

**1. 异常和中断**

异常主要是指来自 CPU 内部的意外事件，比如执行了未定义指令、算术溢出、除零运算等发生在 CPU 内部的意外事件，这些异常的发生，会引起 CPU 运行相应的异常处理程序。

中断一般来自硬件（如片上外设、外部 I/O 输入等）发生的事件，当这些硬件产生中断信号时，CPU 会暂停当前运行的程序，转而去处理相关硬件的中断服务程序。但无论是异常还是中断，都会引起程序执行偏离正常的流程，转而去执行异常/中断的处理函数。

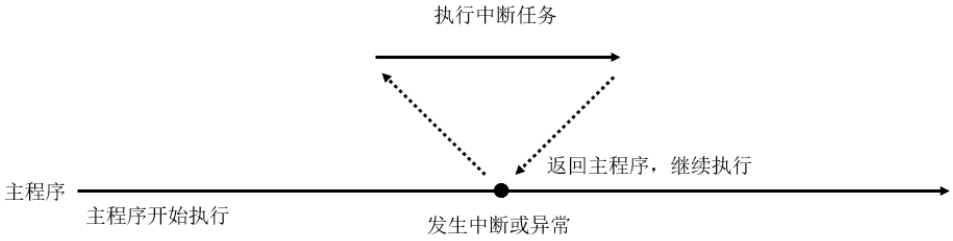



图 4.4.1 中断或异常处理示意图

**2. 异常和中断向量表**

STM32F405xx/07xx 和 STM32F415xx/17xx 在内核的基础上搭载了一个异常响应系统，支持众多的系统异常和外部中断（其中系统异常有 10 个，外部中断 82 个）。可在标准库头文件 stm32f4xx.h 中的 IRQn_Type 结构体中查询到 STM32F4xx 系列的全部异常声明。

如表 4.4.1，展示了 STM32F405xx/07xx 和 STM32F415xx/17xx 系统异常向量表。要注意其中 Reset（复位）、NMI（不可屏蔽中断）和 HardFault（所有类型的错误）系统异常的优先级是固定的、不可设置的。且异常优先级值越小，代表其优先级越高。

表 4.4.1 STM32F405xx/07xx 和 STM32F415xx/17xx 系统异常向量表

| 位置 | 优先级 | 优先级类型 | 名称 | 说明 | 地址 |
|---|---|---|---|---|---|
| - | - | - | - | 保留 | 0x0000 0000 |
| - | -3 | 固定 | Reset | 复位 | 0x0000 0004 |
| - | -2 | 固定 | NMI | 不可屏蔽中断。RCC 时钟安全系统(CSS)连接到 NMI 向量。 | 0x0000 0008 |
| - | -1 | 固定 | HardFault | 所有类型的错误 | 0x0000 000C |
| - | 0 | 可设置 | MemManage | 存储器管理 | 0x0000 0010 |
| - | 1 | 可设置 | BusFault | 预取指失败，存储器访问失败 | 0x0000 0014 |
| - | 2 | 可设置 | UsageFault | 未定义的指令或非法状态 | 0x0000 0018 |
| - | - | - | - | 保留 | 0x0000 001C - 0x0000 002B |
| - | 3 | 可设置 | SVCall | 通过 SWI 指令调用的系统服务 | 0x0000 002C |
| - | 4 | 可设置 | Debug Monitor | 调试监控器 | 0x0000 0030 |
| - | - | - | - | 保留 | 0x0000 0034 |
| - | 5 | 可设置 | PendSV | 可挂起的系统服务 | 0x0000 0038 |
| - | 6 | 可设置 | SysTick | 系统嘀嗒定时器 | 0x0000 003C |

如表 4.4.2，展示了 STM32F405xx/07xx 和 STM32F415xx/17xx 部分外部中断异常的向量表，完整向量表请参考《STM32F4xx 中文参考手册》10.2 外部中断/事件控制器(EXTI)。

表 4.4.2 STM32F405xx/07xx 和 STM32F415xx/17xx 外部中断异常向量表

| 位置 | 优先级 | 优先级类型 | 名称 | 说明 | 地址 |
|---|---|---|---|---|---|
| 0 | 7 | 可设置 | WWDG | 窗口看门狗中断 | 0x0000 0040 |
| 1 | 8 | 可设置 | PVD | 连接到 EXTI 线的可编程电压检测(PVD)中断 | 0x0000 0044 |
| 2 | 9 | 可设置 | TAMP_STAMP | 连接到 EXTI 线的入侵和时间戳中断 | 0x0000 0048 |
| 3 | 10 | 可设置 | RTC_WKUP | 连接到 EXTI 线的 RTC 唤醒中断 | 0x0000 004C |
| 4 | 11 | 可设置 | FLASH | Flash 全局中断 | 0x0000 0050 |
| 5 | 12 | 可设置 | RCC | RCC 全局中断 | 0x0000 0054 |
| 6 | 13 | 可设置 | EXTI0 | EXTI 线 0 中断 | 0x0000 0058 |
| 7 | 14 | 可设置 | EXTI1 | EXTI 线 1 中断 | 0x0000 005C |
| 8 | 15 | 可设置 | EXTI2 | EXTI 线 2 中断 | 0x0000 0060 |
| 9 | 16 | 可设置 | EXTI3 | EXTI 线 3 中断 | 0x0000 0064 |
| 10 | 17 | 可设置 | EXTI4 | EXTI 线 4 中断 | 0x0000 0068 |

| | | | | | |
|---|---|---|---|---|---|
| … | … | … | … | … | … |
| 80 | 87 | 可设置 | HASH_RNG | 哈希和随机数发生器全局中断 | 0x0000 0180 |
| 81 | 88 | 可设置 | FPU | FPU 全局中断 | 0x0000 0184 |

**3 嵌套向量中断控制器**

在内核中有如此多的中断，就不得不面临一些问题，比如“两个中断同时发生，应该先执行哪个中断任务？”。又比如“一个中断发生了，又来了一个更紧急的中断，是该继续执行原来的中断，还是执行新的紧急中断？”等，针对这些问题，内核中提供了一个专门的外设：嵌套向量中断控制器（Nested Vectored Interrupt Controller，简称 NVIC）。

它控制着整个芯片中断相关的功能，跟内核是紧密耦合的。它通过优先级来控制中断的嵌套和调度。NVIC 是一个总的中断控制器，无论是来在内核的异常还是外设的外部中断，都是由 NVIC 统一进行管理的。

但是各个芯片厂商在设计芯片的时候会对内核里面的 NVIC 进行裁剪，把不需要的部分去掉，所以说 STM32F4xx 的 NVIC 是 Cortex-M4 的 NVIC 的一个子集。

**4. 中断优先级**

在 Cortex-M4 中，将中断的优先级拆分为抢占优先级和响应优先级（也称子优先级），每个中断都需要指定这两个优先级。其中抢占优先级决定是否可以产生中断嵌套，响应优先级决定中断响应顺序。

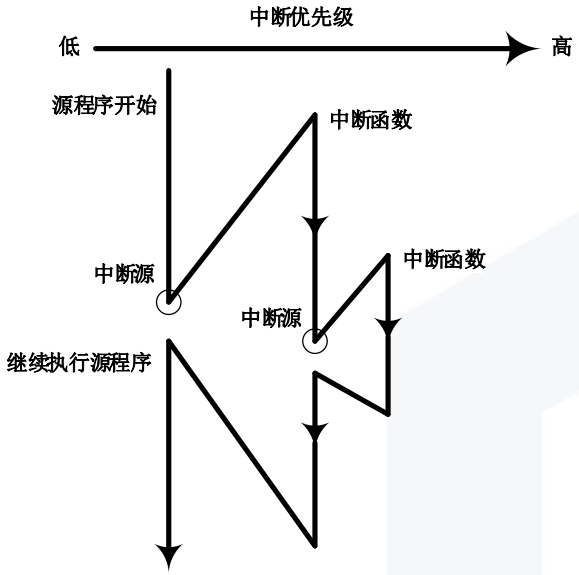



图 4.4.2 中断嵌套示意图

抢占优先级高（值小）的中断可以中断抢占优先级低（值大）的中断处理函数，进而执行高优先级的中断处理函数，执行完毕后再继续执行被中断的低优先级的处理函数。

当两个中断的抢占优先级相同时，即这两个中断将没有嵌套关系，当一个中断到来后，若此时 CPU 正在处理另一个中断，则这个后到来的中断就要等到前一个中断处理函数处理完毕后才能被处理，当这两个中断同时到达，则中断控制器会根据它们的响应优先级决定先处理哪个。

如果两个中断的优先级都设置为一样，那么谁先触发的就谁先执行；如果是同时触发的，那么就根据中断异常表的位置（靠前）来决定谁先执行。

在 STM32 中设置了中断分组的概念。可以使用 NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)函数将其分为 0~4 组，每组的分配结果如表 4.4.3 所示。

表 4.4.3 STM32F4xx 中断分组介绍

| 中断分组输入参数 | bit 分配情况 | 中断优先级分配结果 |
|---|---|---|
| NVIC_PriorityGroup_0 | 0:4 | 0 位抢占优先级（0），4 位响应优先级（0~15） |
| NVIC_PriorityGroup_1 | 1:3 | 1 位抢占优先级（0~1），3 位响应优先级（0~7） |
| NVIC_PriorityGroup_2 | 2:2 | 2 位抢占优先级（0~3），2 位响应优先级（0~3） |
| NVIC_PriorityGroup_3 | 3:1 | 3 位抢占优先级（0~7），1 位响应优先级（0~1） |
| NVIC_PriorityGroup_4 | 4:0 | 4 位抢占优先级（0~15），0 位响应优先级（0） |

**5. 外部中断/事件控制器**

STM32F4xx 系列的外部中断/事件控制器（EXTI）包含多达 23 个用于产生事件/中断请求的边沿检测器。每根输入线都可单独进行配置，以选择类型（中断或事件）和相应的触发事件（上升沿触发、下降沿触发或边沿触发），每根输入线还可单独屏蔽。其中挂起寄存器用于保持中断请求的状态线。

如图 4.4.3，展示了 STM32F4xx 系列的外部中断/事件控制器框图。可以看到在信号线上会有一个斜杠并标注“23”字样，这表示在控制器内部类似的信号线路有 23 个。

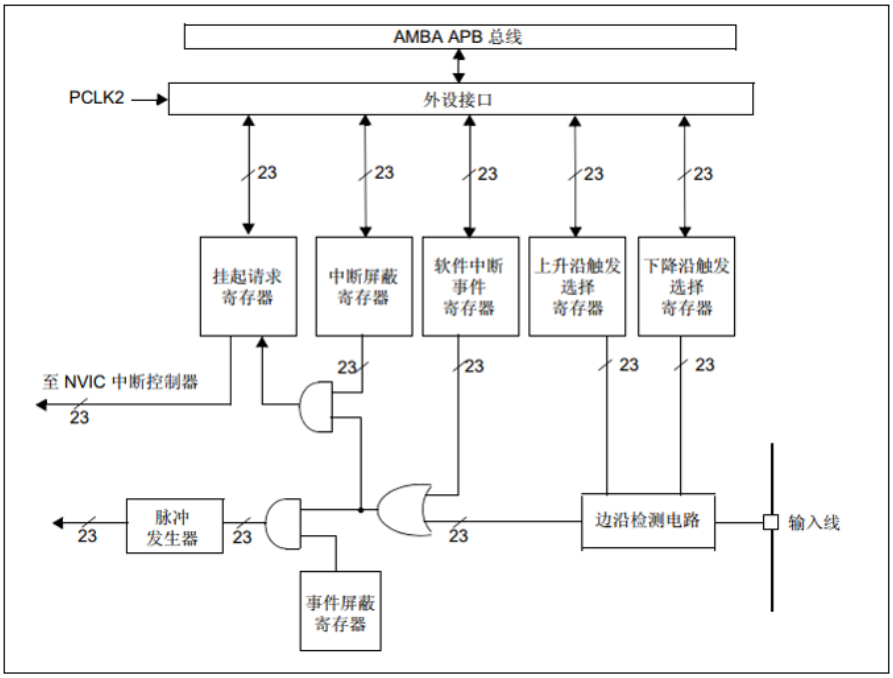



图 4.4.3 外部中断/事件控制器框图

**6. 外部中断/事件线映射**

如表 4.4.4，展示了 STM32F4xx 系列的 23 条线路的外部中断/事件线映射表，中断线与输入源的对应关系参考《STM32F4xx 中文参考手册》10.2.5 外部中断/事件线映射，外部中断通道和服务函数需分别参照标准库头文件 stm32f4xx.h 的对应芯片型号和 startup_stm32f40_41xxx.s。

表 4.4.4 外部中断/事件线映射表

| 中断/事件线 | 输入源 | 外部中断通道 | 外部中断服务函数 |
|---|---|---|---|
| EXTI0 | PX0(X 可为 A,B,C,D,E,F,G,H,I) | EXTI0_IRQn | EXTI0_IRQHandler |
| EXTI1 | PX1(X 可为 A,B,C,D,E,F,G,H,I) | EXTI1_IRQn | EXTI1_IRQHandler |
| EXTI2 | PX2(X 可 A,B,C,D,E,F,G,H,I) | EXTI2_IRQn | EXTI2_IRQHandler |
| EXTI3 | PX3(X 可为 A,B,C,D,E,F,G,H,I) | EXTI3_IRQn | EXTI3_IRQHandler |
| EXTI4 | PX4(X 可为 A,B,C,D,E,F,G,H,I) | EXTI4_IRQn | EXTI4_IRQHandler |
| EXTI5 | PX5(X 可为 A,B,C,D,E,F,G,H,I) | EXTI9_5_IRQn | EXTI9_5_IRQHandler |
| EXTI6 | PX6(X 可为 A,B,C,D,E,F,G,H,I) | | |
| EXTI7 | PX7(X 可为 A,B,C,D,E,F,G,H,I) | | |
| EXTI8 | PX8(X 可为 A,B,C,D,E,F,G,H,I) | | |
| EXTI9 | PX9(X 可为 A,B,C,D,E,F,G,H,I) | | |
| EXTI10 | PX10(X 可为 A,B,C,D,E,F,G,H,I) | EXTI15_10_IRQn | EXTI15_10_IRQHandler |
| EXTI11 | PX11(X 可为 A,B,C,D,E,F,G,H,I) | | |

| | | | |
|---|---|---|---|
| EXTI12 | PX12(X 可为 A,B,C,D,E,F,G,H) | | |
| EXTI13 | PX13(X 可为 A,B,C,D,E,F,G,H) | | |
| EXTI14 | PX14(X 可为 A,B,C,D,E,F,G,H) | | |
| EXTI15 | PX15(X 可为 A,B,C,D,E,F,G,H) | | |
| EXTI16 | 可编程电压检测器(PVD)输出 | PVD_IRQn | PVD_IRQHandler |
| EXTI17 | RTC 闹钟事件 | RTC_Alarm_IRQn | RTC_Alarm_IRQHandler |
| EXTI18 | USB OTG FS 唤醒事件 | OTG_FS_WKUP_IRQn | OTG_FS_WKUP_IRQHandler |
| EXTI19 | 以太网唤醒事件 | ETH_WKUP_IRQn | ETH_WKUP_IRQHandler |
| EXTI20 | USB OTG HS(在 FS 中配置)唤醒事件 | OTG_HS_WKUP_IRQn | OTG_HS_WKUP_IRQHandler |
| EXTI21 | 入侵和时间戳事件 | TAMP_STAMP_IRQn | TAMP_STAMP_IRQHandler |
| EXTI22 | RTC 唤醒事件 | RTC_WKUP_IRQn | RTC_WKUP_IRQHandler |

**7. 中断相关库函数讲解**

（1）外部中断初始化结构体定义

代码段 4.4.1 外部中断初始化结构体定义

```
typedef struct
{
    uint32_t            EXTI_Line;
    EXTIMode_TypeDef    EXTI_Mode;
    EXTITrigger_TypeDef EXTI_Trigger;
    FunctionalState     EXTI_LineCmd;
}EXTI_InitTypeDef;
```

关于外部中断初始化结构体各个成员的详细描述，如表 4.4.5 所示。

表 4.4.5 外部中断初始化结构体成员

| 结构体成员名称 | 结构体成员参数 (即可选的参数) | 结构体成员说明 |
|---|---|---|
| EXTI_Line | EXTI_Line0~23 | 指定要启用或禁用的中断线 |
| EXTI_Mode | EXTI_Mode_Interrupt：中断模式<br>EXTI_Mode_Event：事件事件 | 指定所选中断线的模式 |

| | | |
|---|---|---|
| EXTI_Trigger | EXTI_Trigger_Rising：上升沿触发<br>EXTI_Trigger_Falling：下降沿触发<br>EXTI_Trigger_Rising_Falling：上升沿和下降沿触发 | 指定所选中断线的触发类型 |
| EXTI_LineCmd | ENABLE<br>DISABLE | 指定所选中断线的状态 |

（2）外部中断初始化函数

表 4.4.6 外部中断初始化函数

| 接口函数：EXTI_Init | |
|---|---|
| 函数原型 | void EXTI_Init(EXTI_InitTypeDef *EXTI_InitStruct) |
| 功能描述 | 根据 EXTI_InitStruct 中的指定参数初始化 EXTI 外设。 |
| 入口参数 | EXTI_InitStruct：指向 EXTI_InitStruct 结构体的指针，该结构体包含 EXTI 指定的外设配置信息。 |
| 返回值 | 无 |
| 使用示例 | EXTI_Init(&EXTI_InitStructure); |
| 注意事项 | 无 |

（3）选择 GPIO 引脚用于外部中断线的函数

表 4.4.7 选择 GPIO 引脚用于外部中断线的函数

| 接口函数：SYSCFG_EXTILineConfig | |
|---|---|
| 函数原型 | void SYSCFG_EXTILineConfig(uint8_t EXTI_PortSourceGPIOx, uint8_t EXTI_PinSourcex) |
| 功能描述 | 将指定的 GPIO 端口和引脚用作外部中断线。 |
| 入口参数 | EXTI_PortSourceGPIOx：指定用作外部中断线的 GPIO 端口。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，x 可以为 A~I。 |
| | EXTI_PinSourcex：指定用作外部中断线的 GPIO 引脚。其范围为 GPIO_Pin_0~GPIO_Pin_15。在 STM32F405xx/407xx 和 STM32F415xx/417xx 中，若指定端口为 GPIOI，则其范围为 GPIO_Pin_0~GPIO_Pin_11。 |
| 返回值 | 无 |
| 使用示例 | SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOA, EXTI_PinSource0); |
| 注意事项 | 无 |

（4）获取中断线标志的状态函数

表 4.4.8 获取中断线标志的状态函数

| 接口函数：EXTI_GetFlagStatus | |
|---|---|
| 函数原型 | FlagStatus EXTI_GetFlagStatus(uint32_t EXTI_Line) |
| 功能描述 | 获取指定的 EXTI_Line 的标志状态。 |
| 入口参数 | EXTI_Line：指定的中断线。其参数可以为 EXTI_Linex（x 为 0~22）。 |
| 返回值 | 中断线（EXTI_Line）的状态，返回值可以为 SET 或 RESET。 |
| 使用示例 | FlagStatus status = EXTI_GetITStatus(EXTI_Line0); |
| 注意事项 | 无 |

（5）清除中断线的待处理位函数

表 4.4.9 清除中断线的待处理位函数

| 接口函数：EXTI_ClearITPendingBit | |
|---|---|
| 函数原型 | void EXTI_ClearITPendingBit(uint32_t EXTI_Line) |
| 功能描述 | 清除指定 EXTI_Line 的待处理位。 |
| 入口参数 | EXTI_Line：指定的中断线。其参数可以为 EXTI_Linex（x 为 0~22）。 |
| 返回值 | 无 |
| 使用示例 | EXTI_ClearITPendingBit(EXTI_Line0); |
| 注意事项 | 无 |

（6）配置优先级分组函数

表 4.4.10 配置优先级分组函数

| 接口函数：NVIC_PriorityGroupConfig | |
|---|---|
| 函数原型 | void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup) |
| 功能描述 | 配置优先级分组。 |
| 入口参数 | NVIC_PriorityGroup：指定优先级分组的比特长度。<br>其参数可以为 NVIC_PriorityGroup_x（x 为 0~4）。<br>注意：根据不同的优先级分组，抢占优先级和响应优先级有不同的分配情况。具体的分配情况见表 4.4.3。 |
| 返回值 | 无 |
| 使用示例 | NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); |
| 注意事项 | 当函数参数为 NVIC_PriorityGroup_0 时，中断抢占将不再可能，待处理的中断优先级将仅由子优先级管理。 |

（7）嵌套中断向量控制器初始化结构体定义

代码段 4.4.2 嵌套中断向量控制器初始化结构体定义

```
typedef struct
{
    uint8_t         NVIC_IRQChannel;
    uint8_t         NVIC_IRQChannelPreemptionPriority;
    uint8_t         NVIC_IRQChannelSubPriority;
    FunctionalState NVIC_IRQChannelCmd;
} NVIC_InitTypeDef;
```

关于嵌套中断向量控制器初始化结构体各个成员的详细描述，如表 4.4.11 所示。

表 4.4.11 嵌套中断向量控制器初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| NVIC_IRQChannel | EXTI0_IRQn<br>EXTI1_IRQn<br>……<br>常见中断通道详见表 4.4.4 中的外部中断通道 | 指定要启用或禁用的中断通道 |

| | | |
|---|---|---|
| NVIC_IRQChannelPreemptionPriority | 详见表 4.4.3 关于中断优先级的分配结果 | 指定中断通道的抢占优先级 |
| NVIC_IRQChannelSubPriority | 详见表 4.4.3 关于中断优先级的分配结果 | 指定中断通道的子优先级 |
| NVIC_IRQChannelCmd | ENABLE<br>DISABLE | 指定是否启动所选用的中断通道 |

（8）嵌套中断向量控制器初始化函数

表 4.4.12 嵌套中断向量控制器初始化函数

| 接口函数：NVIC_Init | |
|---|---|
| 函数原型 | void NVIC_Init(NVIC_InitTypeDef* NVIC_InitStruct) |
| 功能描述 | 根据 NVIC_InitStruct 中的指定参数初始化 NVIC 外设。（注意：NVIC 属于内核外设，EXTI 属于片上外设） |
| 入口参数 | NVIC_InitStruct：指向 NVIC_InitStruct 结构体的指针，该结构体包含 NVIC 指定的外设配置信息。 |
| 返回值 | 无 |
| 使用示例 | NVIC_Init(&NVIC_InitStructure); |
| 注意事项 | 无 |

## 4.4.2 硬件设计

中断相关内容不涉及外部硬件电路。

## 4.4.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将中断部分的驱动代码独立开来，源文件（exti.c）和头文件（exti.h）存放在实验案例根目录中的 HARDWARE/EXTI 下。

**1. 软件设计流程**

如图 4.4.4，展示了本实验的软件设计流程图。

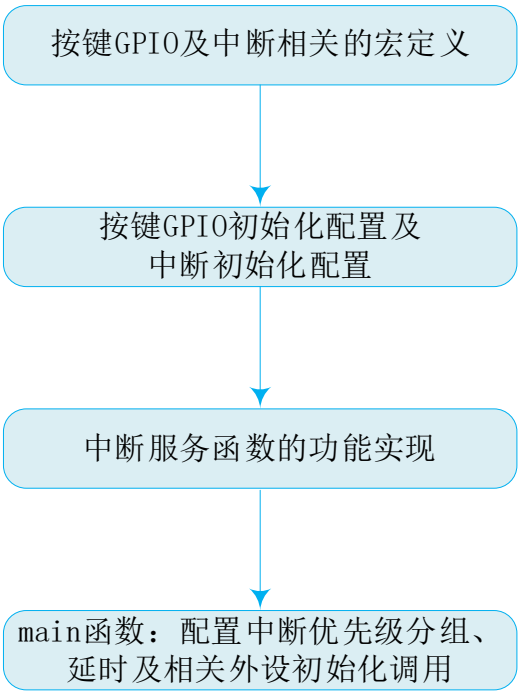



图 4.4.4 软件设计流程图

**2. 软件代码讲解**

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将中断部分的驱动代码独立开来，源文件（exti.c）和头文件（exti.h）存放在实验案例根目录中的 HARDWARE/EXTI 下。

（1）定义按键 GPIO 引脚及外部中断相关的宏。

代码段 4.4.3 按键 GPIO 引脚及外部中断相关定义（exti.h）

```
//按键对应的 GPIO 时钟挂载在 AHB1 上
#define KEY_ALL_CLOCK           RCC_AHB1Periph_GPIOE | RCC_AHB1Periph_GPIOA

//按键对应的 GPIO 时钟使能，其为宏函数定义
#define KEY_CLK_INIT            RCC_AHB1PeriphClockCmd

//SYSCFG 时钟挂载在 APB2 上，外部中断必须使能该时钟
#define SYSCFG_CLOCK            RCC_APB2Periph_SYSCFG

//SYSCFG 时钟使能，其为宏函数定义
```

```
#define SYSCFG_CLK_INIT         RCC_APB2PeriphClockCmd

/***************按键端口引脚定义***************/
//KEY1 --- PE4
#define KEY1_PORT               GPIOE
#define KEY1_PIN                GPIO_Pin_4

//KEY2 --- PE3
#define KEY2_PORT               GPIOE
#define KEY2_PIN                GPIO_Pin_3

//KEY3 --- PE2
#define KEY3_PORT               GPIOE
#define KEY3_PIN                GPIO_Pin_2

//KEY4 --- PA0
#define KEY4_PORT               GPIOA
#define KEY4_PIN                GPIO_Pin_0

/***************按键端口复用引脚定义***************/
//KEY1 --- PE4
#define KEY1_PORT_Sources       EXTI_PortSourceGPIOE
#define KEY1_PIN_Sources        EXTI_PinSource4

//KEY2 --- PE3
#define KEY2_PORT_Sources       EXTI_PortSourceGPIOE
#define KEY2_PIN_Sources        EXTI_PinSource3

//KEY3 --- PE2
#define KEY3_PORT_Sources       EXTI_PortSourceGPIOE
#define KEY3_PIN_Sources        EXTI_PinSource2

//KEY4 --- PA0
#define KEY4_PORT_Sources       EXTI_PortSourceGPIOA
#define KEY4_PIN_Sources        EXTI_PinSource0

/***************按键中断线、中断通道、中断服务函数对应关系及其定义***************/
//按键名   输入源      中断线          中断通道          中断服务函数
//KEY1     PE4       EXTI_Line4      EXTI4_IRQn       EXTI4_IRQHandler
#define KEY1_EXTI_Line          EXTI_Line4
#define KEY1_EXTI_IRQn          EXTI4_IRQn
```

```
#define KEY1_EXTI_IRQHandler    EXTI4_IRQHandler

//KEY2    PE3       EXTI_Line3     EXTI3_IRQn      EXTI3_IRQHandler
#define KEY2_EXTI_Line          EXTI_Line3
#define KEY2_EXTI_IRQn          EXTI3_IRQn
#define KEY2_EXTI_IRQHandler    EXTI3_IRQHandler

//KEY3    PE2       EXTI_Line2     EXTI2_IRQn      EXTI2_IRQHandler
#define KEY3_EXTI_Line          EXTI_Line2
#define KEY3_EXTI_IRQn          EXTI2_IRQn
#define KEY3_EXTI_IRQHandler    EXTI2_IRQHandler

//KEY4    PA0       EXTI_Line0     EXTI0_IRQn      EXTI0_IRQHandler
#define KEY4_EXTI_Line          EXTI_Line0
#define KEY4_EXTI_IRQn          EXTI0_IRQn
#define KEY4_EXTI_IRQHandler    EXTI0_IRQHandler
```

（2）外部中断初始化配置函数。其配置步骤主要为：定义按键 GPIO 和中断的相关结构体；使能按键 GPIO 和中断相关的时钟；按键 GPIO 初始化配置（配置引脚为输入模式）；中断线关联；中断初始化配置（包括外部中断配置和中断优先级配置）；初始化中断。

代码段 4.4.4 外部中断初始化配置函数（exti.c）

```
/*
功   能：  外部中断初始化配置函数
参   数：  无
返回值：  无
*/
void Key_EXTI_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;            //定义 GPIO 初始化结构体变量
    EXTI_InitTypeDef EXTI_InitStructure;            //定义 EXTI 初始化结构体变量
    NVIC_InitTypeDef NVIC_InitStructure;            //定义 NVIC 初始化结构体变量

    KEY_CLK_INIT(KEY_ALL_CLOCK, ENABLE);            //开启中断输入端口时钟
    SYSCFG_CLK_INIT(SYSCFG_CLOCK, ENABLE);          //开启外部中断关联时钟

    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;    //输入模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;    //带上拉
```

```
GPIO_InitStructure.GPIO_Pin = KEY1_PIN;          //选择 KEY1 GPIO 引脚
GPIO_Init(KEY1_PORT, &GPIO_InitStructure);       //初始化 KEY1 GPIO

GPIO_InitStructure.GPIO_Pin = KEY2_PIN;          //选择 KEY2 GPIO 引脚
GPIO_Init(KEY2_PORT, &GPIO_InitStructure);       //初始化 KEY2 GPIO

GPIO_InitStructure.GPIO_Pin = KEY3_PIN;          //选择 KEY3 GPIO 引脚
GPIO_Init(KEY3_PORT, &GPIO_InitStructure);       //初始化 KEY3 GPIO

GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;     //输入模式
GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;   //带下拉

GPIO_InitStructure.GPIO_Pin = KEY4_PIN;          //选择 KEY4 GPIO 引脚 O
GPIO_Init(KEY4_PORT, &GPIO_InitStructure);       //初始化 KEY4 GPI

//中断线关联
SYSCFG_EXTILineConfig(KEY1_PORT_Sources, KEY1_PIN_Sources);
SYSCFG_EXTILineConfig(KEY2_PORT_Sources, KEY2_PIN_Sources);
SYSCFG_EXTILineConfig(KEY3_PORT_Sources, KEY3_PIN_Sources);
SYSCFG_EXTILineConfig(KEY4_PORT_Sources, KEY4_PIN_Sources);

//中断线选择
EXTI_InitStructure.EXTI_Line = KEY1_EXTI_Line | KEY2_EXTI_Line | KEY3_EXTI_Line;

EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;          //中断触发
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;      //下降沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;                    //中断线使能
EXTI_Init(&EXTI_InitStructure);                              //初始化配置

EXTI_InitStructure.EXTI_Line = KEY4_EXTI_Line;               //中断线选择
EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;          //中断触发
EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;       //上升沿触发
EXTI_InitStructure.EXTI_LineCmd = ENABLE;                    //中断线使能
EXTI_Init(&EXTI_InitStructure);                              //初始化配置

NVIC_InitStructure.NVIC_IRQChannel = KEY1_EXTI_IRQn;         //选择中断通道
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;    //设置先占优先级
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;           //设置从优先级
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;              //设置中断通道开启
NVIC_Init(&NVIC_InitStructure);                              //初始化配置中断优先级
```

```
    NVIC_InitStructure.NVIC_IRQChannel = KEY2_EXTI_IRQn;        //选择中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;   //设置先占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 2;          //设置从优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             //设置中断通道开启
    NVIC_Init(&NVIC_InitStructure);                             //初始化配置中断优先级

    NVIC_InitStructure.NVIC_IRQChannel = KEY3_EXTI_IRQn;        //选择中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;   //设置先占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;          //设置从优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             //设置中断通道开启
    NVIC_Init(&NVIC_InitStructure);                             //初始化配置中断优先级

    NVIC_InitStructure.NVIC_IRQChannel = KEY4_EXTI_IRQn;        //选择中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;   //设置先占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;          //设置从优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;             //设置中断通道开启
    NVIC_Init(&NVIC_InitStructure);                             //初始化配置中断优先级
}
```

（3）编写 EXTI 中断服务函数。指定按键按下后的中断触发事件为对应 LED 灯闪烁 5 次。

代码段 4.4.5 EXTI 中断服务函数（exti.c）

```
/*
功　能：　外部中断向量服务函数(KEY1)
参　数：　无
返回值：　无
*/
void KEY1_EXTI_IRQHandler()
{
    //消抖延时，实际应用中不建议在中断函数延时
    Delay_ms(10);
    //检查指定的 EXTI 线路触发请求发生与否
    if (EXTI_GetFlagStatus(KEY1_EXTI_Line) == SET)
    {
        //LED1 闪烁 5 次
        for(int i=0; i<5; i++)
        {
```

```
            LED1(1);
            Delay_ms(300);
            LED1(0);
            Delay_ms(300);
        }
        EXTI_ClearITPendingBit(KEY1_EXTI_Line);          //清除 EXTI 线路挂起位
    }
}

/*
功   能:  外部中断向量服务函数(KEY2)
参   数:  无
返回值:  无
*/
void KEY2_EXTI_IRQHandler()
{
    //消抖延时，实际应用中不建议在中断函数延时
    Delay_ms(10);
    //检查指定的 EXTI 线路触发请求发生与否
    if (EXTI_GetITStatus(KEY2_EXTI_Line) == SET)
    {
        //LED2 闪烁 5 次
        for(int i=0; i<5; i++)
        {
            LED2(1);
            Delay_ms(300);
            LED2(0);
            Delay_ms(300);
        }
        EXTI_ClearITPendingBit(KEY2_EXTI_Line);         //清除 EXTI 线路挂起位
    }
}
/*
功   能:  外部中断向量服务函数(KEY3)
参   数:  无
返回值:  无
*/
void KEY3_EXTI_IRQHandler()
{
    //消抖延时，实际应用中不建议在中断函数延时
    Delay_ms(10);
```

```
    //检查指定的 EXTI 线路触发请求发生与否
    if (EXTI_GetITStatus(KEY3_EXTI_Line) == SET)
    {
        //LED3 闪烁 5 次
        for(int i=0; i<5; i++)
        {
            LED3(1);
            Delay_ms(300);
            LED3(0);
            Delay_ms(300);
        }
        EXTI_ClearITPendingBit(KEY3_EXTI_Line);         //清除 EXTI 线路挂起位
    }
}
/*
功   能:  外部中断向量服务函数(KEY4)
参   数:  无
返回值:  无
*/
void KEY4_EXTI_IRQHandler(void)
{
    //消抖延时，实际应用中不建议在中断函数延时
    Delay_ms(10);
    //检查指定的 EXTI 线路触发请求发生与否
    if (EXTI_GetITStatus(KEY4_EXTI_Line) == SET)
    {
        //LED4 闪烁 5 次
        for(int i=0; i<5; i++)
        {
            LED4(1);
            Delay_ms(300);
            LED4(0);
            Delay_ms(300);
        }
        EXTI_ClearITPendingBit(KEY4_EXTI_Line);         //清除 EXTI 线路挂起位
    }
}
```

---

（5）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了中断优先级分组、延时和相关外设初始化的流程，而 while 循环中则无需任何操作。

代码段 4.4.6 主函数（main.c）

```
int main(void)
{
    //配置优先级组别 2
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);
    Delay_Init();                    //延时初始化
    Led_Init();                      //LED 初始化
    Key_EXTI_Init();                 //外部中断初始化
    while(1){}
}
```

## 4. 4. 4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器。

| PC | STM32F4 核心板 | 12V 电源适配器 | ST Link 下载器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

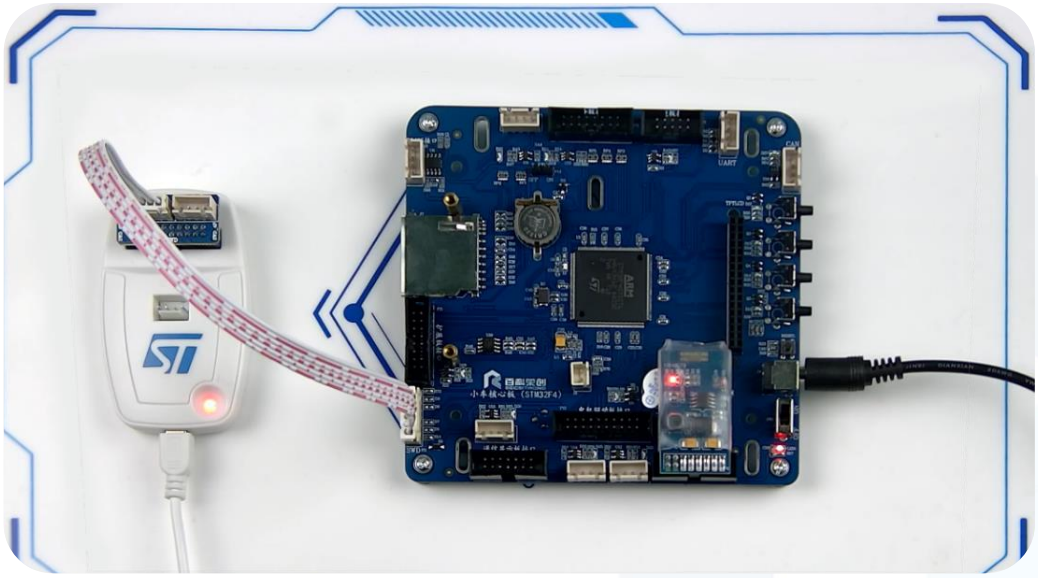


图 4.4.5 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，通过按下按键观察是否出现 LED 流水灯现象。

### 4.4.5 结果展示

将程序下载至核心板后，按下任意一个按键，其对应的 LED 灯闪烁 5 次。

本实验中，中断的优先级是根据表 4.4.3 配置了不同的中断分组、抢占优先级和响应优先级。如表 4.4.13 所示，展示了本实验的中断配置。

表 4.4.13 实验中断配置

| 中断分组配置 | 按键 | 对应中断抢占优先级 | 对应中断响应优先级 |
|---|---|---|---|
| NVIC_PriorityGroup_2<br>2 位抢占优先级<br>（0~3）<br>2 位响应优先级<br>（0~3） | KEY1 | 1 | 1 |
| | KEY2 | 1 | 2 |
| | KEY3 | 1 | 3 |
| | KEY4 | 0 | 1 |

因此，快速按下 KEY1、KEY4，可以发现，当 LED1 闪烁事件执行一段时间后，LED4 的闪烁事件打断了 LED1 的闪烁事件，且等待 LED4 的闪烁事件完成后，继续执行剩余 LED1 未完成的闪烁事件，这一现象体现了中断的抢占优先级特性。

同理，也可快速按下 KEY4、KEY3、KEY2、KEY1，观察实验现象，发现 LED4 的闪烁事件执行完毕后，依次执行 LED1、LED2、LED3 的闪烁事件，这一现象则体现了中断的响应优先级特性。

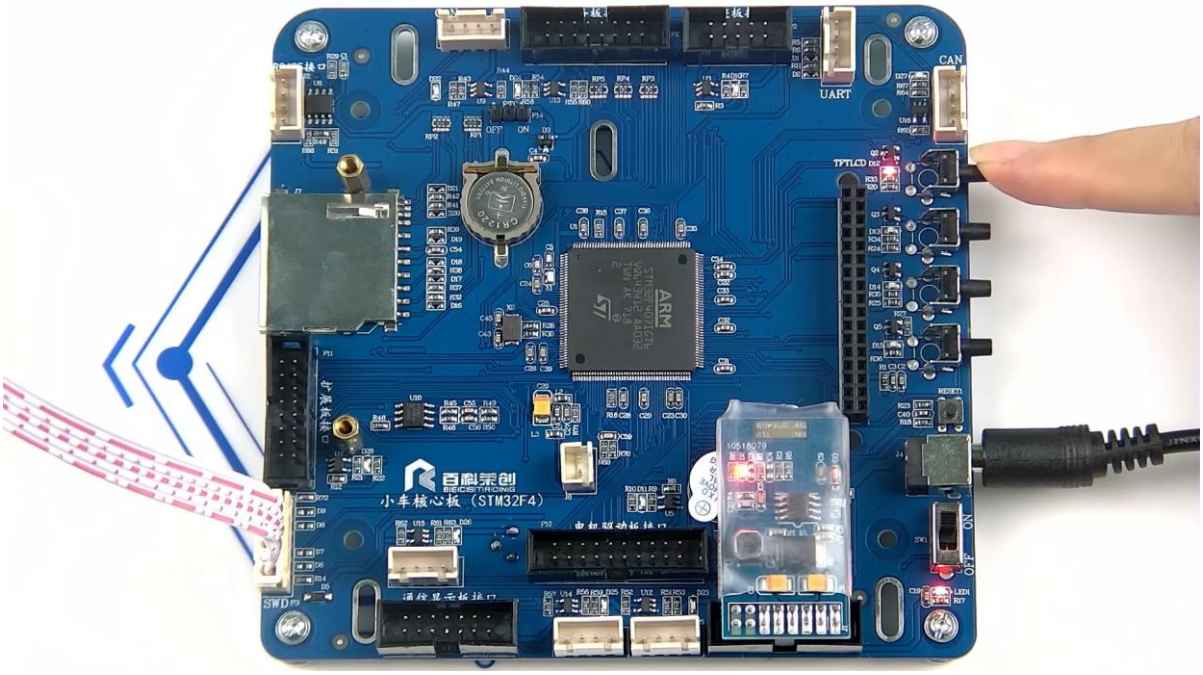



图 4.4.6 实验效果图展示

# 4.5 串口数据发送与接收实验

本实验旨在介绍嵌入式系统中一个关键的外设——串口通信。

通过完成本实验，读者将学习如何配置和操作 STM32F4 核心板的串口功能，并更深入地了解串口通信的原理和应用，使读者具备在嵌入式系统中利用串口进行数据交互的能力。

## 4.5.1 串口通信介绍

### 1. 设备通信方式

设备之间的通信方式可以分成并行通信和串行通信两种，它们之间的区别如图 4.5.1 和表 4.5.1 所示。

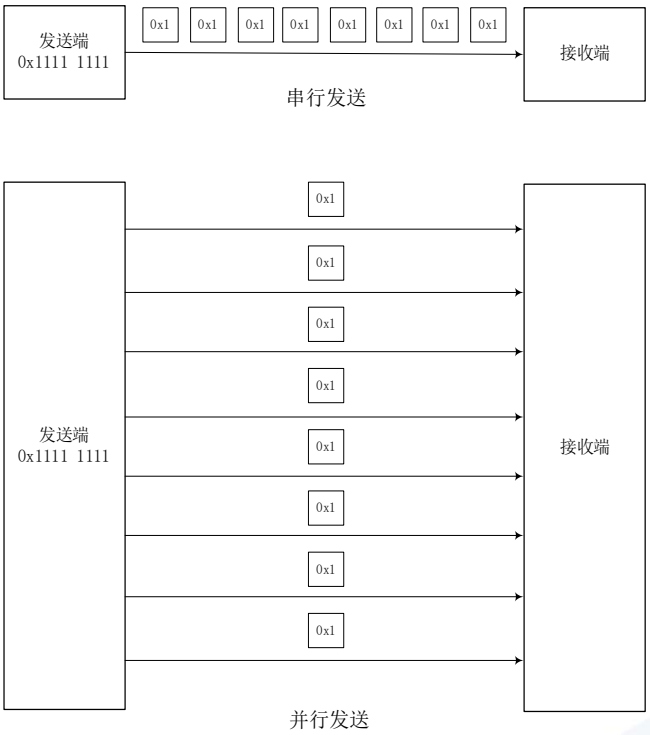



图 4.5.1 串行发送与并行发送

表 4.5.1 并行通信与串行通信的对比

| | 并行通信 | 串行通信 |
|---|---|---|
| 传输原理 | 数据各个位同时传输 | 数据按位顺序传输 |
| 优点 | 速度快 | 占用引脚资源少 |
| 缺点 | 占用引脚资源多 | 速度相对较慢 |

### 2. 串行通信分类

按照数据的传输方向，可将其分为：

（1）单工：数据传输只支持数据在一个方向上传输；

（2）半双工：允许数据在两个方向上传输，但是在某一时刻只允许数据在一个方向上传输，它实际上是一种能切换方向的单工通信。它不需要独立的接收端和发送端，两者可以合并在一起使用一个端口；

（3）全双工：允许数据同时在两个方向上传输。因此，全双工通信是两个单工通信方式的结合，需要独立的接收端和发送端。

如图 4.5.2 所示，展示了串行通讯中单工、半双工及全双工的工作特点。

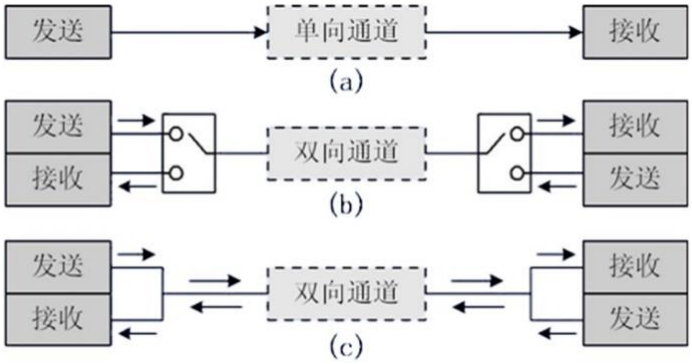



图 4.5.2 串行通讯中单工、半双工及全双工的工作特点

按照通信方式分，可将其分为：

（1）同步通信：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讯方式。

（2）异步通信：发送方发出数据后，不等接收方发回响应，接着发送下个数据包的通讯方式。

其中，同步通信要求接收端时钟频率和发送端时钟频率一致；异步通信时不要求接收端时钟和发送端时钟同步，但需要双方规约好数据的传输速率（也就是波特率）等信息，以便更好地同步。（常用的波特率有 4800bps、9600bps、115200bps 等。）

同步通信数据传输是以字节块(多个字节)传输的；异步通信数据传输是以字符(一个字节）传输的。

同步通信传输效率高，但复杂、要求高，双方时钟的允许误差较小；异步通信传输效率低，但简单、要求低，双方时钟可允许一定误差。

同步通信的字节传输是没有间隔的发送端发送连续的比特流；异步通信字节传送的间隔是任意的，发送端发送完一个字节后，可经过任意长的时间间隔再发送下一个字节。

如表 4.5.2 所示，展示了同步通信与异步通信之间的差异。

表 4.5.2 同步通信与异步通信的对比

| | 同步通信 | 异步通信 |
|---|---|---|
| 传送单位 | 信息帧（由若干字符组成的数据块） | 字符（由若干 bit 组成） |
| 单位格式 | 同步字符+数据+校验字符 CRC | 起始位+数据位+奇偶校验位+停止位 |
| 传送间隔 | 一个数据块（信息帧）内，字符与字符间无间隔 | 相邻两字符之间间隔任意长度 |
| 时钟信号 | 时序要求高，使用同频同相的时钟线路 | 时序要求较低，使用各自的时钟信号 |
| 优点 | 效率高 | 简单，要求低 |
| 缺点 | 复杂，要求高 | 效率低（传送一个字符，要增加约 20%的附加信息位） |

**3. STM32 常见的串行通信**

在 STM32 微控制器上，SPI 和 IIC 为同步通信，UART 为异步通信，USART 为同步和异步通信。如表 4.5.3，展示了 STM32 微控制器上常用的串行通信接口。

表 4.5.3 常用的串行通信接口

| 通信标准 | 引脚说明 | 通信方式 | 通信方向 |
|---|---|---|---|
| UART（通用异步收发器） | TXD 发送端，RXT 接收端 | 异步通信 | 全双工 |
| 1-wire（单总线） | DQ：发送/接收端 | 异步通信 | 半双工 |
| SPI | SCK：同步时钟<br>MISO：主机输入，从机输出<br>MOSI：主机输出，从机输入 | 同步通信 | 全双工 |
| I2C | SCK：同步时钟<br>SDA：数据输入/输出端 | 同步通信 | 半双工 |

**4. STM32 的串口通信**

STM32F405xx/STM32F407xx 的串口通信接口有两种，分别是：UART（通用异步收发器）和 USART（通用同步异步收发器）。其片上有 4 个 USART 和 2 个 UART。分别为：USART1，USART2，USART3，UART4，UART5 和 USART6。

**5. 串口异步通信协议**

由表 4.5.1 可知，串行通信的数据是按位顺序传输的。

在 STM32 中串口异步通信定义的数据有：起始位、数据位（8 位或者 9 位）、奇偶校验位（第 9 位）和停止位（0.5 位，1 位，1.5 位，2 位）。

首先，起始位是异步通信中用于同步的初始信号位，恒为低电平。

其次，数据位就是需要传输的数据，最高可设置九位。可以选择九位全数据、八位全数据、八位数据+一位奇偶校验位。奇偶校验位可用于判断传输数据完整性，可以用于检测数据传输是否错误。

最后是停止位，停止位用于向接收方确定发送数据结束，恒为高电平。可以自由选择停止位的时长，可以使用半个数据位、一个数据位、一个半数据位、两个数据位。

设置完成后，需要接收方和发送方都要改成相同的通信协议才能正常通信。

异步串口通信的数据包以帧为单位，常用的帧结构包含：

（1）9 位字长（1 个停止位）：1 位起始位+<8 位数据位+1 位奇偶校验位（可选）>+1 位停止位。

（2）8 位字长（1 个停止位）：1 位起始位+<7 位数据位+1 位奇偶校验位（可选）>+1 位停止位。

如图 4.5.3，展示了常用的数据帧结构（9 位字长和 8 位字长）。

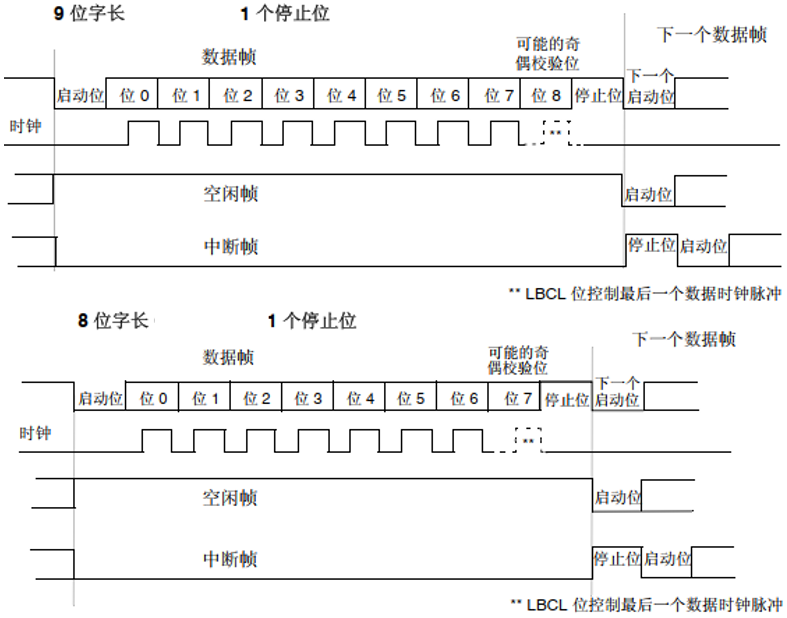



图 4.5.3 常用数据帧结构（9 位字长和 8 位字长）

**6. 串口波特率**

串口波特率用于确定串口传输速度的快慢，单位为 bps，即每秒发送的位数，每个位（bit）所占用的时间为（1/波特率）秒。常用器件与微控制器之间常用的波特率有 115200 和 9600 这两个波特率。

波特率计算公式：

$$\text{Tx/Rx baud} = \text{fck} / (16 * \text{USARTDIV})$$

其中 fck（串口时钟源）由微控制器总线决定。比如：STM32F405xx/STM32F407xx 微控制器的 USART1 和 USART6 搭载在 APB2 总线上，USART2、USART3、UART4 和 UART5 搭载在 APB1 总线上。所以串口的波特率可通过设置对应的分频系数来控制。而使用库函数会自动帮助设置分频系数。

使用异步通信时必须将接收方和发送方设置成相同波特率才能正常通信。

**7. 串口通信的连接**

（1）如图 4.5.4 所示，展示了两个芯片之间的串口连接方法，两个芯片之间的连接除 TXD 和 RXD 交叉连接外，还需要进行共地连接。这样，两个芯片之间就可以进行正常的 TTL 电平通信了。其中 RXD 是数据输入端口，用于数据接收；TXD 是数据发送端口，用于数据发送。

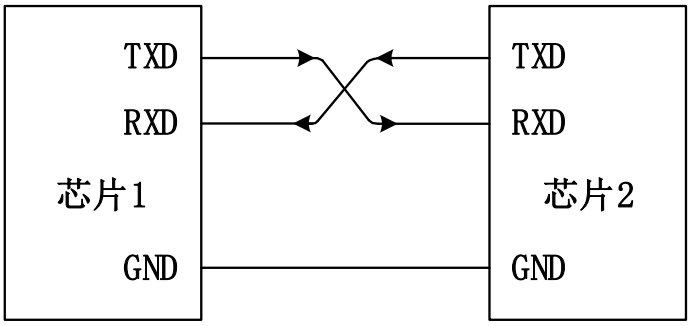



图 4.5.4 UART 端口的连接方法

（2）使用 USB 转 TTL 模块将主机（PC）与芯片进行连接，其作用就是将电平转换为双方都能识别的方式进行通信。

TTL 电平一般是单片机或芯片发出的电平，其数据表示采用二进制：其中，逻辑高电平 1 通常对应 3.3V 或 5V，逻辑低电平 0 通常对应 0V。

其接线示意图如图 4.5.5 所示。

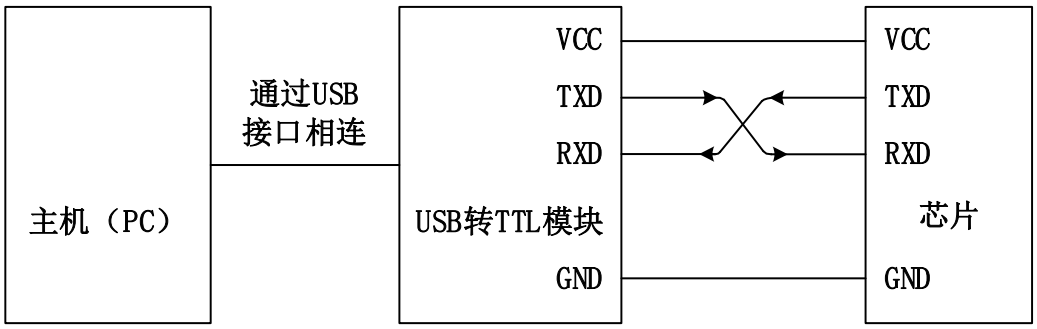



图 4.5.5 USB 转 TTL 接线示意图

**8. USART 中断**

如表 4.5.4，列出了 USART 的中断请求。

表 4.5.4 USART 中断请求

| 中断事件 | 事件标志 | 使能控制位 |
|---|---|---|
| 发送数据寄存器为空 | TXE | TXEIE |
| CTS 标志 | CTS | CTSIE |
| 发送完成 | TC | TCIE |
| 准备好读取接收到的数据 | RXNE | RXNEIE |
| 检测到上溢错误 | ORE | |
| 检测到空闲线路 | IDLE | IDLEIE |
| 奇偶校验错误 | PE | PEIE |
| 断路标志 | LBD | LBDIE |
| 多缓冲区通信中的噪声标志、上溢错误和帧错误 | NF 或 ORE 或 FE | EIE |

USART 中断事件被连接到相同的中断向量，USART 中断具体分为：

（1）在发送期间包括：发送完成、清除已发送或发送数据寄存器为空中断；

（2）在接收期间包括：空闲线路检测、上溢错误、接收数据寄存器不为空、奇偶校验错误、LIN 断路检测、噪声标志（仅限多缓冲区通信）和帧错误（仅限多缓冲区通信）。

如果相应的使能控制位置 1，则这些事件会生成中断。

如图 4.5.6，展示了 USART 的中断映射图。

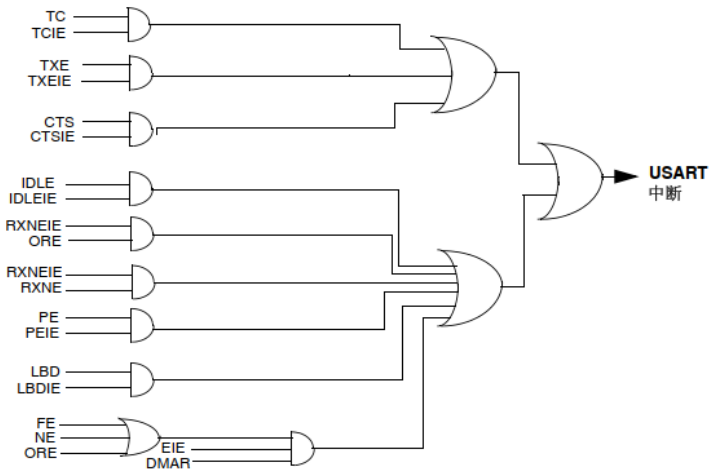



图 4.5.6 USART 的中断映射图

**9. 串口相关函数讲解**

（1）串口初始化结构体定义

代码段 4.5.1 串口初始化结构体定义

```
typedef struct
{
  uint32_t  USART_BaudRate;
  uint16_t  USART_WordLength;
  uint16_t  USART_StopBits;
  uint16_t  USART_Parity;
  uint16_t  USART_Mode;
  uint16_t  USART_HardwareFlowControl;
} USART_InitTypeDef;
```

关于串口初始化结构体各个成员的详细描述，如表 4.5.5 所示。

表 4.5.5 串口初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| USART_BaudRate | 常用串口波特率为：9600、115200 等 | 串口波特率① |
| USART_WordLength | USART_WordLength_8b：8 个<br>USART_WordLength_9b：9 个 | 指定帧中传输或接收的数据位数 |

| | | |
|---|---|---|
| USART_StopBits | USART_StopBits_0_5：0.5 个<br>USART_StopBits_1：1 个<br>USART_StopBits_1_5：1.5 个<br>USART_StopBits_2：2 个 | 指定传输的停止位数 |
| USART_Parity | USART_Parity_Even：偶校验<br>USART_Parity_No：无奇偶校验<br>USART_Parity_Odd：奇校验 | 指定奇偶校验模式 |
| USART_Mode | USART_Mode_Rx：接受模式<br>USART_Mode_Tx：发送模式<br>USART_Mode_Rx\|USART_Mode_Tx：发送和接受模式 | 指定启用或禁用接收/发送模式 |
| USART_Hardware FlowControl | USART_HardwareFlowControl_CTS<br>USART_HardwareFlowControl_None：禁用硬件流控制模式<br>USART_HardwareFlowControl_RTS<br>USART_HardwareFlowControl_RTS_CTS | 指定启用或禁用硬件流控制模式 |
| ①串口波特率的计算公式如下：<br>(1)IntegerDivider = ((PCLKx) / (8 * (OVR8+1) * (USART_InitStruct->USART_BaudRate)))<br>(2)FractionalDivider = ((IntegerDivider - ((u32) IntegerDivider)) * 8 * (OVR8+1)) + 0.5<br>其中 OVR8 是 USART_CR1 寄存器中 8 倍过采样模式的配置位。 | | |

（2）启用或禁用指定的串口中断函数

表 4.5.6 启用或禁用指定的串口中断函数

| 接口函数：USART_ITConfig | |
|---|---|
| 函数原型 | void USART_ITConfig(USART_TypeDef *USARTx, uint16_t USART_IT, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 USARTx 中断。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| | USART_IT：指定要启用或禁用的 USART 中断源。<br>其参数可以为：USART_IT_CTS、USART_IT_LBD、USART_IT_TXE（发送数据寄存器为空中断源）、USART_IT_TC（发送完成中断源）、USART_IT_RXNE（接受数据寄存器非空中断源）、USART_IT_IDLE（检测到空闲线路中断源）、USART_IT_PE（奇偶校验错误中断源）、USART_IT_ERR（错误中断源）。 |
| | NewState：指定 USARTx 外设中断的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | USART_ITConfig(USART3, USART_IT_RXNE, ENABLE); |
| 注意 | 无 |

| 事项 | |
|---|---|

（3）启用或禁用指定的串口外设函数

表 4.5.7 启用或禁用指定的串口外设函数

| 接口函数：USART_Cmd | |
|---|---|
| 函数原型 | void USART_Cmd(USART_TypeDef *USARTx, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 USARTx 外设。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| | NewState：指定 USARTx 外设的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | USART_Cmd(USART3, ENABLE); |
| 注意事项 | 无 |

（4）通过串口外设发送单个数据函数

表 4.5.8 通过串口外设发送单个数据函数

| 接口函数：USART_SendData | |
|---|---|
| 函数原型 | void USART_SendData(USART_TypeDef *USARTx, uint16_t Data) |
| 功能描述 | 通过指定的 USARTx 外设发送单个数据。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| | Data：发送的数据。 |
| 返回值 | 无 |
| 使用示例 | USART_SendData(USART3, “A”); |
| 注意事项 | 无 |

（5）返回串口外设最近获取到的数据函数

表 4.5.9 返回串口外设最近获取到的数据函数

| 接口函数：USART_ReceiveData | |
|---|---|
| 函数原型 | uint16_t USART_ReceiveData(USART_TypeDef *USARTx) |
| 功能描述 | 返回指定的 USARTx 外设最近获取到的数据。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| 返回值 | 接受到的数据。 |
| 使用示例 | uint16_t receive_data = USART_ReceiveData(USART3); |
| 注意事项 | 无 |

（6）获取串口指定中断的状态函数

表 4.5.10 获取串口指定中断的状态函数

| 接口函数：USART_GetITStatus | |
|---|---|
| 函数原型 | ITStatus USART_GetITStatus(USART_TypeDef *USARTx, uint16_t USART_IT) |
| 功能描述 | 检查 USARTx 上的指定的中断是否发生。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| | USART_IT：指定的 USART 中断源。<br>其参数可以为：USART_IT_CTS、USART_IT_LBD、USART_IT_TXE（发送数据寄存器为空中断源）、USART_IT_TC（发送完成中断源）、USART_IT_RXNE（接受数据寄存器非空中断源）、USART_IT_IDLE（检测到空闲线路中断源）、USART_IT_PE（奇偶校验错误中断源）、USART_IT_ORE_RX、USART_IT_ORE_ER、USART_IT_NE、USART_IT_FE。 |
| 返回值 | 串口中断源（USART_IT）的状态，返回值可以为 SET 或 RESET。 |
| 使用示例 | ITStatus status = USART_GetITStatus(USART3, USART_IT_RXNE); |
| 注意事项 | 无 |

（7）清除串口的中断待处理位函数

表 4.5.11 清除串口的中断待处理位函数

| 接口函数：USART_ClearITPendingBit | |
|---|---|
| 函数原型 | void USART_ClearITPendingBit(USART_TypeDef *USARTx, uint16_t USART_IT) |
| 功能描述 | 清除 USARTx 外设指定中断的待处理位。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| | USART_IT：指定的 USART 中断源。<br>其参数可以为：USART_IT_CTS、USART_IT_LBD、USART_IT_TC（发送完成中断源）、USART_IT_RXNE（接受数据寄存器非空中断源）。 |
| 返回值 | 无 |
| 使用示例 | USART_ClearITPendingBit(USART3, USART_IT_RXNE); |
| 注意事项 | （1）PE(Parity error)、FE(Framing error)、NE(Noise error)、ORE(OverRun error) 和 IDLE(Idle line detected) 的待处理位需通过先调用 USART_GetITStatus()，然后调用 USART_ReceiveData()进行清除。<br>（2）RXNE 的待处理位也可以通过调用 USART_ReceiveData()进行清除。<br>（3）TC 的待处理位也可以通过先调用 USART_GetITStatus()，然后调用 USART_SendData()进行清除。<br>（4）TXE 的待处理位只能通过调用 USART_SendData()进行清除。 |

## 4.5.2 硬件设计

**1. 实验电路分析**

如图 4.5.7，显示了串口的接口原理图，其中 PC11 对应了 USART3_RX，PC10 对应了 USART3_TX，在程序上则需将这两个 GPIO 引脚进行复用。

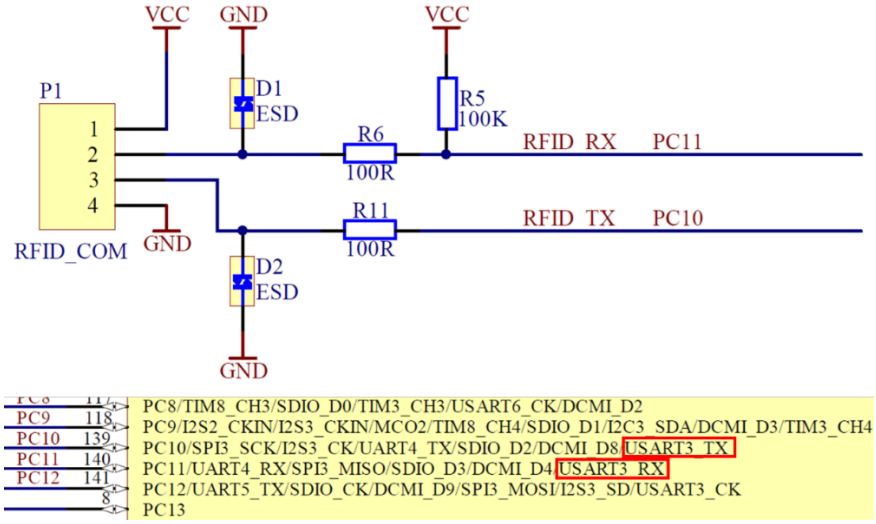


图 4.5.7 串口接口电路原理图

## 4.5.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将串口 3 部分的驱动代码独立开来，源文件（usart3.c）和头文件（usart3.h）存放在实验案例根目录中的 SYSTEM/usart3 下。

**1. 软件设计流程**

如图 4.5.8，展示了本实验的软件设计流程图。

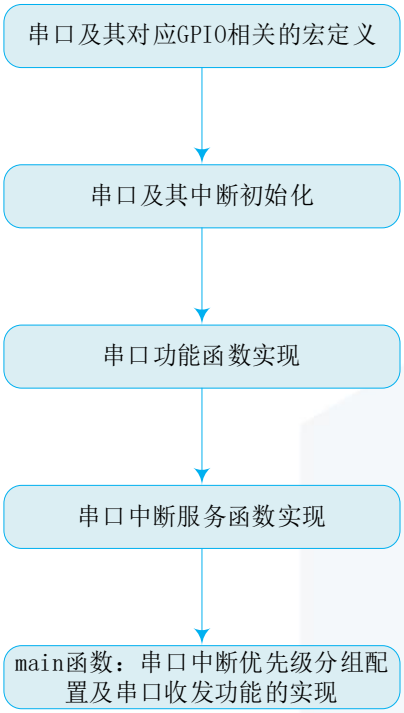


图 4.5.8 软件设计流程图

## 2. 软件代码讲解

（1）定义串口和 GPIO 引脚相关的宏，并包含相关变量的定义。

代码段 4.5.2 串口和 GPIO 引脚宏定义（usart3.h）

```
//定义串口接受最大存储的数据为 200 个字节,可根据实际需要修改定义
#define USART3_RX_SIZE              200
//储存串口接收数据的数组
extern uint8_t                      usart3_buffer[];
//储存串口接收数据数组的长度
extern uint16_t                     usart3_length;

/***************串口相关定义***************/
//串口时钟挂载在 APB1 上
#define USART3_CLOCK                RCC_APB1Periph_USART3

//串口时钟使能，其为宏函数定义
#define USART3_CLK_INIT             RCC_APB1PeriphClockCmd

//串口对应的 GPIO 时钟挂载在 AHB1 上
#define USART3_GPIO_CLOCK           RCC_AHB1Periph_GPIOC

//串口对应的 GPIO 时钟使能，其为宏函数定义
#define USART3_GPIO_CLK_INIT        RCC_AHB1PeriphClockCmd

//串口发送引脚（USART TX）
#define USART3_TX_PORT              GPIOC
#define USART3_TX_PIN               GPIO_Pin_10
#define USART3_TX_GPIO_PinSource    GPIO_PinSource10

//串口接受引脚（USART RX）
#define USART3_RX_PORT              GPIOC
#define USART3_RX_PIN               GPIO_Pin_11
#define USART3_RX_GPIO_PinSource    GPIO_PinSource11

/***************串口中断通道、串口中断服务函数定义***************/
//串口        串口中断通道          串口中断服务函数
//USART3      USART3_IRQn           USART3_IRQHandler
#define USART3_INT_IRQn             USART3_IRQn
#define USART3_INT_IRQHandler       USART3_IRQHandler
```

（2）串口及其中断进行初始化。

代码段 4.5.3 串口始化函数（usart3.c）

```
/*
功　能：　串口的初始化函数
参　数：　baudrate 串口的波特率
返回值：　无
*/
void USART3_Init(uint32_t baudrate)
{
    GPIO_InitTypeDef GPIO_InitStructure;        //定义 GPIO 初始化结构体变量
    USART_InitTypeDef USART_InitStructure;      //定义串口初始化结构体变量
    NVIC_InitTypeDef NVIC_InitStructure;        //定义 NVIC 初始化结构体变量

    USART3_GPIO_CLK_INIT(USART3_GPIO_CLOCK, ENABLE);        //开启外设 GPIO 时钟
    USART3_CLK_INIT(USART3_CLOCK, ENABLE);                  //开启外设时钟 USART
    //GPIO 引脚复用到串口上
    GPIO_PinAFConfig(USART3_TX_PORT, USART3_TX_GPIO_PinSource, GPIO_AF_USART3);
    GPIO_PinAFConfig(USART3_RX_PORT, USART3_RX_GPIO_PinSource, GPIO_AF_USART3);

    //GPIO 配置
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;            //复用模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;          //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;      //输出速度

    //USART Tx 初始化
    GPIO_InitStructure.GPIO_Pin = USART3_TX_PIN;            //选择引脚
    GPIO_Init(USART3_TX_PORT, &GPIO_InitStructure);         //初始化配置

    //USART Rx 初始化
    GPIO_InitStructure.GPIO_Pin = USART3_RX_PIN;            //选择引脚
    GPIO_Init(USART3_RX_PORT, &GPIO_InitStructure);         //初始化配置

    //串口参数设置
    USART_InitStructure.USART_BaudRate = baudrate;                  //设置波特率
    USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx; //设置串口模式
    USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None; //设置无
硬件流控制
    USART_InitStructure.USART_WordLength = USART_WordLength_8b;     //8bit 数据位
    USART_InitStructure.USART_Parity = USART_Parity_No;             //无校验位
```

```
    USART_InitStructure.USART_StopBits = USART_StopBits_1;          //1 停止位
    USART_Init(USART3, &USART_InitStructure);                       //初始化配置串口

    //串口中断响应优先级设置
    NVIC_InitStructure.NVIC_IRQChannel = USART3_INT_IRQn;        //选择中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;    //设置先占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 8;           //设置从优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;              //设置中断通道开启
    NVIC_Init(&NVIC_InitStructure);                              //初始化配置中断优先级

    USART_ITConfig(USART3, USART_IT_RXNE, ENABLE);               //开启串口接受中断使能

    USART_Cmd(USART3, ENABLE);                                   //开启串口
}
```

（4）串口功能函数实现，包括串口发送数据等函数。

代码段 4.5.4 串口功能函数（usart3.c）

```
uint16_t usart3_length = 0;                     //储存串口一接收数据数组的长度
uint8_t usart3_buffer[USART3_RX_SIZE] = { 0 };  //储存串口一接收数据的数组
uint8_t usart3_tc = 0;                          //串口接收数据时的计数器
uint8_t usart3_flag = 0;                        //串口标志位,1 表示接收完成,0 表示可以接收数据

/*
功   能:  串口发送单字节数据
参   数:  src 发送的单字节数据
返回值:  无
*/
void USART3_Send_Byte(uint8_t src)
{
    USART_SendData(USART3, src);                                    //发送数据
    while (USART_GetFlagStatus(USART3, USART_FLAG_TXE) == RESET);   //等待串口发送完成
}

/*
功   能:  串口发送一定长度的字节数据
参   数:  src        发送的字符串数据
          length     发送字节/字符的长度
返回值:  无
*/
```

```
void USART3_Send_Length_String(uint8_t* src, uint16_t length)
{
    uint16_t Tx_cut = 0;
    uint16_t Tx_length = 0;
    Tx_length = length;

    while (Tx_cut != Tx_length)             //循环发送指定长度的数据
    {
        USART3_Send_Byte(*(src + Tx_cut));  //发送一个字节信息
        Tx_cut++;                           //指针自增
    }
}
/*
功  能:  串口发送字符串数据
参  数:  src     发送的字符串数据,以'\0'即 0x00 标识结束发送数据
返回值:  无
*/
void USART3_Send_String(char* src)
{
    uint16_t Tx_cut = 0;

    while (*(src + Tx_cut) != '\0')         //检测到下一个发送字节为'\0'时停止发送
    {
        USART3_Send_Byte(*(src + Tx_cut));  //发送一个字节信息
        Tx_cut++;                           //指针自增
    }
}
/*
功  能:  串口清除储存数组的数据
参  数:  无
返回值:  无
*/
void USART3_Clear_Buffer(void)
{
    while (usart3_length)                   //清空接收到的数据
    {
        usart3_length--;                    //指针自减
        usart3_buffer[usart3_length] = 0;   //数据清零
    }
}
```

---

（5）串口中断服务函数实现，主要是串口接受数据的功能实现。

代码段 4.5.5 串口中断服务函数（usart3.c）

```
/*
功　能：　串口的接收中断服务函数
参　数：　无
返回值：　无
*/
void USART3_INT_IRQHandler(void)
{
    if (USART_GetITStatus(USART3, USART_IT_RXNE) == SET)            //判断串口接收中断标志位
    {
        usart3_buffer[usart3_length++] = USART_ReceiveData(USART3);  //读取串口数据
        if (usart3_length >= USART3_RX_SIZE)                        //如果储存超出长度
        {
            usart3_length = 0;                                      //重新从数组 0 开始储存字
节
        }
    }
    USART_ClearITPendingBit(USART3, USART_IT_RXNE);               //清除串口中断接收标志位
}
```

（6）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而 while 循环中则需实现串口发送数据的功能。

代码段 4.5.6 主函数（main.c）

```
int main(void)
{
    uint8_t key_value = 0;

    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0); //使用中断,配置优先级组别 0
    Delay_Init();                                  //延时初始化
    Led_Init();                                    //LED 初始化
    Key_Init();                 //按键初始化
    USART3_Init(115200);                           //串口初始化

    while (1)
    {
        //确定是哪个按键被按下
```

```
        key_value = Key_Scan(0);

        switch(key_value)
        {
            case 1:
                LED1_TOGGLE();
                USART3_Send_String("LED1 OK!\r\n");
                break;
            case 2:
                LED2_TOGGLE();
                USART3_Send_String("LED2 OK!\r\n");
                break;
            case 3:
                LED3_TOGGLE();
                USART3_Send_String("LED3 OK!\r\n");
                break;
            case 4:
                LED4_TOGGLE();
                USART3_Send_String("LED4 OK!\r\n");
                break;
            default:
                break;
        }
    }
  }
```

---

## 4.5.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |

| ST Link 下载器 | USB 转 TTL 模块 |
| --- | --- |
| | |

2. 实验硬件接线：可使用操作台串口调试助手连接核心板 UART 接口，也可使用 USB 转 TTL 模块进行连接，注意：（使用 USB 转 TTL 模块连接时 TXD 与 RXD 交叉连接，即：TXD 接 RXD，RXD 接 TXD）

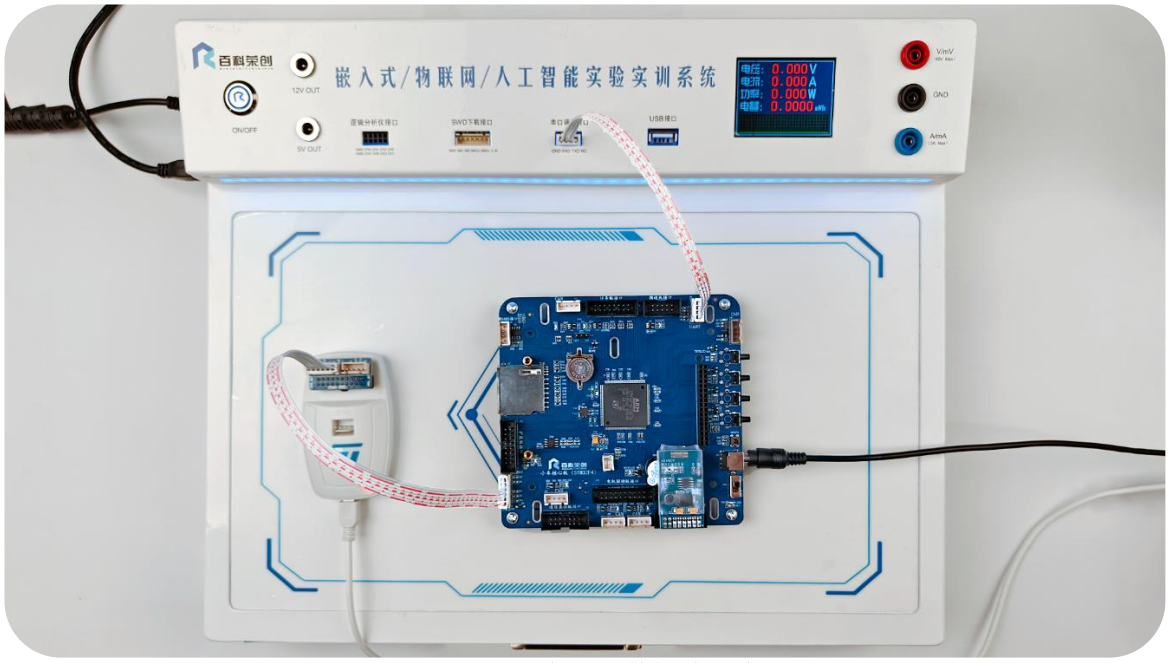



图 4.5.9 实验硬件接线示意图

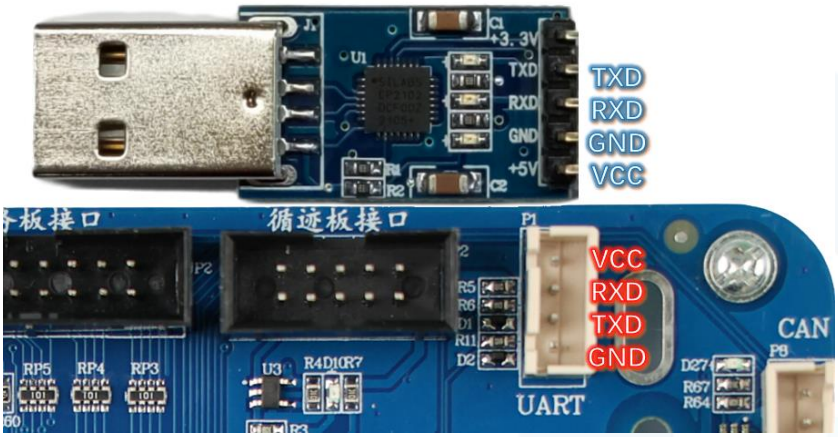



图 4.5.10 USB 转 TTL 接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，按下核心板任意按键，观察串口调试助手是否打印发送的数据。

## 4.5.5 结果展示

按下按键 KEY1，LED1 状态取反；按下按键 KEY2，LED2 状态取反；按下按键 KEY3，LED3 状态取反；按下按键 KEY4，LED4 状态取反，同时串口发送对应字符串“LEDx OK!”,x 为对应 LED 灯。

如图 4.5.11 所示，展示了串口调试助手的实验效果界面。

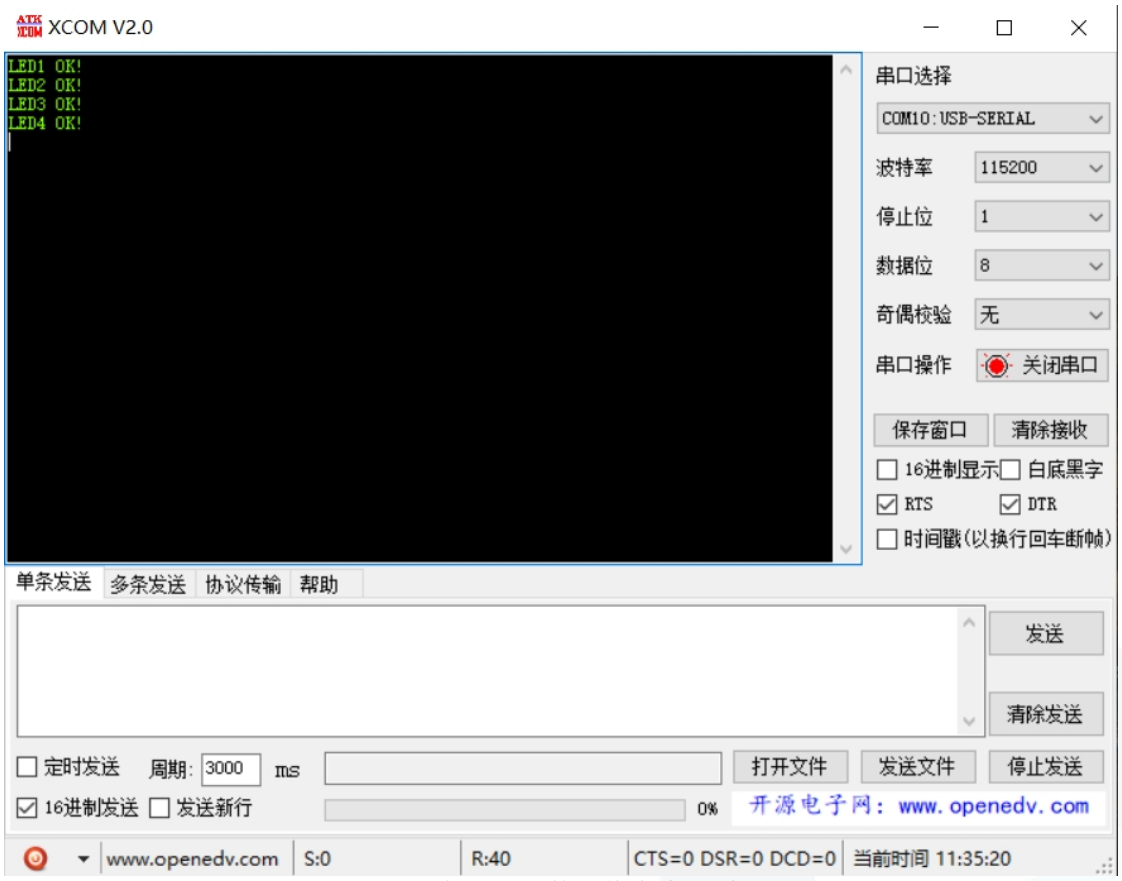



图 4.5.11 串口通信数据收发实验效果图

# 4.6 DMA 数据传输实验

本实验旨在介绍 DMA（直接内存访问）的相关原理，并通过利用 DMA 的传输功能，实现将数据有效地从外设传输至内存、内存传输至外设。

在本实验中，我们将详细讲解 DMA 的工作原理和机制，包括如何配置和启动 DMA 传输通道。通过使用 DMA 传输功能，我们可以实现快速、可靠的数据传输，提高系统性能并降低处理器的负载。

通过完成本实验，读者将深入理解 DMA 的工作原理，并学会如何有效利用 DMA 的传输功能，从而在数据处理和外设交互方面实现更好的性能和效果。

## 4.6.1 DMA 介绍

**1. DMA 简介**

直接存储器访问(Direct memory access，DMA)用于在外设与存储器之间以及存储器与存储器之间提供高速数据传输。可以在无需任何 CPU 操作的情况下通过 DMA 快速移动数据。这样节省的 CPU 资源可供其它操作使用。

两个 DMA 控制器（DMA1、DMA2）总共有 16 个数据流（每个控制器 8 个），每一个 DMA 控制器都用于管理一个或多个外设的存储器访问请求。每个数据流总共可以有多达 8 个通道（或称请求）。每个通道都有一个仲裁器，用于处理 DMA 请求间的优先级。

**2. DMA 主要特性**

（1）双 AHB 主总线架构，一个用于存储器访问，另一个用于外设访问。

（2）仅支持 32 位访问的 AHB 从编程接口

（3）每个 DMA 控制器有 8 个数据流。可供每个数据流选择的通道请求多达 8 个，此选择可由软件配置，允许几个外设启动 DMA 请求。

（4）通过硬件可以将每个数据流配置为：

——支持外设到存储器、存储器到外设和存储器到存储器传输的常规通道；

——也支持在存储器方双缓冲的双缓冲区通道。

（5）8 个数据流中的每一个都连接到专用硬件 DMA 通道（请求）。

（6）DMA 数据流请求之间的优先级可用软件编程（4 个级别：非常高、高、中、低），在软件优先级相同的情况下可以通过硬件决定优先级（例如，请求 0 的优先级高

于请求 1）。

（7）每个数据流也支持通过软件触发存储器到存储器的传输（仅限 DMA2 控制器）。

（8）要传输的数据项的数目可以由 DMA 控制器或外设管理：

——DMA 流控制器：要传输的数据项的数目是 1 到 65535，可用软件编程；

——外设流控制器：要传输的数据项的数目未知并由源或目标外设控制，这些外设通过硬件发出传输结束的信号。

（9）对源和目标的增量或非增量寻址。

（10）每个数据流都支持循环缓冲区管理。

（11）5 个事件标志（DMA 半传输、DMA 传输完成、DMA 传输错误、DMA FIFO 错误、直接模式错误），进行逻辑或运算，从而产生每个数据流的单个中断请求。

**3. 外设通道选择**

每个数据流都与一个 DMA 请求相关联，此 DMA 请求可以从 8 个可能的通道请求中选出。

来自外设的 8 个请求（TIM、ADC、SPI、I2C 等）独立连接到每个通道，具体的连接取决于产品实现情况。如表 4.6.1 和表 4.6.2 所示，分别展示了 DMA1 请求映射表和 DMA2 请求映射表。

表 4.6.1 DMA1 请求映射表

| 外设请求 | 数据流 0 | 数据流 1 | 数据流 2 | 数据流 3 |
|---|---|---|---|---|
| 通道 0 | SPI3_RX | - | SPI3_RX | SPI2_RX |
| 通道 1 | I2C1_RX | - | TIM7_UP | - |
| 通道 2 | TIM4_CH1 | - | I2S3_EXT_RX | TIM4_CH2 |
| 通道 3 | I2S3_EXT_RX | TIM2_UP<br>TIM2_CH3 | I2C3_RX | I2S2_EXT_RX |
| 通道 4 | UART5_RX | USART3_RX | UART4_RX | USART3_TX |
| 通道 5 | UART8_TX(1) | UART7_TX(1) | TIM3_CH4<br>TIM3_UP | UART7_RX(1) |
| 通道 6 | TIM5_CH3<br>TIM5_UP | TIM5_CH4<br>TIM5_TRIG | TIM5_CH1 | TIM5_CH4<br>TIM5_TRIG |
| 通道 7 | - | TIM6_UP | I2C2_RX | I2C2_RX |
| 外设请求 | 数据流 4 | 数据流 5 | 数据流 6 | 数据流 7 |
| 通道 0 | SPI2_TX | SPI3_TX | - | SPI3_TX |
| 通道 1 | TIM7_UP | I2C1_RX | I2C1_TX | I2C1_TX |

| | | | | |
|---|---|---|---|---|
| 通道 2 | I2S2_EXT_TX | I2S3_EXT_TX | TIM4_UP | TIM4_CH3 |
| 通道 3 | I2C3_TX | TIM2_CH1 | TIM2_CH2<br>TIM2_CH4 | TIM2_UP<br>TIM2_CH4 |
| 通道 4 | UART4_TX | USART2_RX | USART2_TX | UART5_TX |
| 通道 5 | TIM3_CH1<br>TIM3_TRIG | TIM3_CH2 | UART8_RX(1) | TIM3_CH3 |
| 通道 6 | TIM5_CH2 | - | TIM5_UP | - |
| 通道 7 | USART3_TX | DAC1 | DAC2 | I2C2_TX |
| (1)这些请求仅在 STM32F42xxx 和 STM32F43xxx 上可用。 | | | | |

表 4.6.2 DMA2 请求映射表

| 外设请求 | 数据流 0 | 数据流 1 | 数据流 2 | 数据流 3 |
|---|---|---|---|---|
| 通道 0 | ADC1 | - | TIM8_CH1<br>TIM8_CH2<br>TIM8_CH3 | - |
| 通道 1 | - | DCMI | ADC2 | ADC2 |
| 通道 2 | ADC3 | ADC3 | - | SPI5_RX(1) |
| 通道 3 | SPI1_RX | SPI4_TX(1) | SPI1_RX | SPI1_TX |
| 通道 4 | SPI4_RX(1) | USART6_RX | USART1_RX | SDIO |
| 通道 5 | TIM1_TRIG | TIM1_CH1 | USART6_RX | SPI4_RX(1) |
| 通道 6 | - | TIM8_UP | TIM1_CH2 | TIM1_CH1 |
| 通道 7 | - | - | TIM8_CH1 | TIM8_CH2 |
| 外设请求 | 数据流 4 | 数据流 5 | 数据流 6 | 数据流 7 |
| 通道 0 | ADC1 | - | TIM1_CH1<br>TIM1_CH2<br>TIM1_CH3 | - |
| 通道 1 | - | SPI6_TX(1) | SPI6_RX(1) | DCMI |
| 通道 2 | SPI5_TX(1) | CRYP_OUT | CRYP_IN | HASH_IN |
| 通道 3 | - | SPI1_TX | - | - |
| 通道 4 | - | USART1_RX | SDIO | USART1_TX |
| 通道 5 | SPI4_TX(1) | - | USART6_TX | USART6_TX |
| 通道 6 | TIM1_CH4<br>TIM1_TRIG<br>TIM1_COM | TIM1_UP | TIM1_CH3 | - |
| 通道 7 | TIM8_CH3 | SPI5_RX(1) | SPI5_TX(1) | TIM8_CH4<br>TIM8_TRIG<br>TIM8_COM |
| (1)这些请求仅在 STM32F42xxx 和 STM32F43xxx 上可用。 | | | | |

**4. 仲裁器**

一个 DMA 控制器对应 8 个数据流，数据流包含要传输数据的源地址、目标地址、数据等等信息。

如果我们需要同时使用同一个 DMA(DMA1 或 DMA2)控制多个外设请求时，那必然需要同时使用多个数据流，那究竟哪一个数据流具有优先传输的权利呢？这就需要仲裁器来管理判断了。

仲裁器管理数据流方法分为两个阶段：

（1）第一阶段属于软件阶段，我们在配置数据流时可以设定它的优先级别，可以设置为非常高、高、中和低四个优先级。

（2）第二阶段属于硬件阶段，如果两个或以上数据流软件设置优先级一样，则他们优先级取决于数据流编号，编号越低越具有优先权，比如数据流 2 的优先级要高于数据流 3。

**5. 可能的 DMA 配置**

如表 4.6.3 所示，展示了可能的 DMA 配置。

表 4.6.3 可能的 DMA 配置

<table>
<tr><th>DMA 传输模式</th><th>源</th><th>目标</th><th>流控制器</th><th>循环模式</th><th>传输类型</th><th>直接模式</th><th>双缓冲区模式</th></tr>
<tr><td rowspan="4">外设到存储器</td><td rowspan="4">AHB 外设端口</td><td rowspan="4">AHB 存储器端口</td><td rowspan="2">DMA</td><td rowspan="2">允许</td><td>单独</td><td>允许</td><td rowspan="2">允许</td></tr>
<tr><td>突发</td><td>禁止</td></tr>
<tr><td rowspan="2">外设</td><td rowspan="2">禁止</td><td>单独</td><td>允许</td><td rowspan="2">禁止</td></tr>
<tr><td>突发</td><td>禁止</td></tr>
<tr><td rowspan="4">存储器到外设</td><td rowspan="4">AHB 存储器端口</td><td rowspan="4">AHB 外设端口</td><td rowspan="2">DMA</td><td rowspan="2">允许</td><td>单独</td><td>允许</td><td rowspan="2">允许</td></tr>
<tr><td>突发</td><td>禁止</td></tr>
<tr><td rowspan="2">外设</td><td rowspan="2">禁止</td><td>单独</td><td>允许</td><td rowspan="2">禁止</td></tr>
<tr><td>突发</td><td>禁止</td></tr>
<tr><td rowspan="2">存储器到存储器</td><td rowspan="2">AHB 外设端口</td><td rowspan="2">AHB 存储器端口</td><td rowspan="2">仅 DMA</td><td rowspan="2">禁止</td><td>单独</td><td rowspan="2">禁止</td><td rowspan="2">禁止</td></tr>
<tr><td>突发</td></tr>
</table>

注意：（1）使用存储器到存储器模式时，不允许循环模式和直接模式。

（2）只有 DMA2 控制器能够执行存储器到存储器的传输。

**6. 流配置过程**

配置 DMA 数据流 x（其中 x 是数据流编号）时应遵守下面的顺序：

（1）如果使能了数据流，要关闭连接到 DMA 数据流请求的外设，必须首先关闭外设连接的 DMA 数据流，重置 DMA_SxCR 寄存器中的 EN 位将其禁止，使用

DMA_Cmd(DMA_Stream_TypeDef* DMAy_Streamx, FunctionalState NewState)函数即可实现，其中 DMAy_Streamx 选择对应的 DMA 和 DMA 数据流，NewState 设为 DISABLE，然后等待 EN 位=0。只有这样才能安全地禁止外设。

（2）设置外设端口地址。外设事件发生后，数据会从此地址移动到外设端口或从外设端口移动到此地址。

（3）设置存储器地址。外设事件发生后，将从此存储器读取数据或将数据写入此存储器。

（4）配置要传输的数据项的总数。每出现一次外设事件或每出现一个节拍的突发传输，该值都会递减。

（5）选择 DMA 通道（请求）。

（6）如果外设用作流控制器而且支持此功能，请使用以下函数进行配置：

DMA_FlowControllerConfig(DMA_Stream_TypeDef* DMAy_Streamx, uint32_t DMA_FlowCtrl)，其中：DMA_FlowCtrl 可以为（DMA_FlowCtrl_Memory 和 DMA_FlowCtrl_Peripheral）

（7）配置数据流优先级。

（8）配置 FIFO 的使用情况（使能或禁止，发送和接收阈值）。

（9）配置数据传输方向、外设和存储器增量/固定模式、单独或突发事务、外设和存储器数据宽度、循环模式、双缓冲区模式和传输完成一半和/或全部完成，和/或错误的中断。

（10）使能数据流。

一旦使能了数据流，即可响应连接到数据流的外设发出的任何 DMA 请求。

一旦在 AHB 目标端口上传输了一半数据，传输一半标志(HTIF)便会置 1，如果传输一半中断使能位(HTIE)置 1，还会生成中断。传输结束时，传输完成标志(TCIF)便会置 1，如果传输完成中断使能位(TCIE)置 1，还会生成中断。

**7. DMA 中断**

对于每个 DMA 数据流，可在发生以下事件时产生中断：

达到半传输；传输完成；传输错误；FIFO 错误（上溢、下溢或 FIFO 级别错误）；直接模式错误。

可以使用单独的中断使能位以实现灵活性，如表 4.6.4 所示。

注意：在将使能控制位置‘1’前，应将相应的事件标志清零，否则会立即产生中断。

表 4.6.4 DMA 中断请求

| 中断事件 | 事件标志 | 使能控制位 |
|---|---|---|
| 半传输 | HTIF | HTIE |
| 传输完成 | TCIF | TCIE |
| 传输错误 | TEIF | TEIE |
| FIFO 上溢/下溢 | FEIF | FEIE |
| 直接模式错误 | DMEIF | DMEIE |

**8. DMA 相关函数讲解**

（1）DMA 初始化结构体定义

代码段 4.6.1 DMA 初始化结构体定义

```
typedef struct
{
  uint32_t  DMA_Channel;
  uint32_t  DMA_PeripheralBaseAddr;
  uint32_t  DMA_Memory0BaseAddr;
  uint32_t  DMA_DIR;
  uint32_t  DMA_BufferSize;
  uint32_t  DMA_PeripheralInc;
  uint32_t  DMA_MemoryInc;
  uint32_t  DMA_PeripheralDataSize;
  uint32_t  DMA_MemoryDataSize;
  uint32_t  DMA_Mode;
  uint32_t  DMA_Priority;
  uint32_t  DMA_FIFOMode;
  uint32_t  DMA_FIFOThreshold;
  uint32_t  DMA_MemoryBurst;
  uint32_t  DMA_PeripheralBurst;
}DMA_InitTypeDef;
```

关于 DMA 初始化结构体各个成员的详细描述，如表 4.6.5 所示。

表 4.6.5 DMA 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| DMA_Channel | DMA_Channel_x（x 可以为 0~7） | 指定用于指定数据流的通道 |
| DMA_Peripheral BaseAddr | 外设基地址 | 指定 DMAy Streamx 的外设基地址 |
| DMA_Memory0 BaseAddr | 内存基地址 | 指定 DMAy Streamx 的内存基地址，该内存是未启用双缓冲模式时使用的默认内存 |
| DMA_DIR | DMA_DIR_MemoryToMemory<br>内存传输到内存<br>DMA_DIR_MemoryToPeripheral<br>内存传输到外设<br>DMA_DIR_PeripheralToMemory<br>外设传输到内存 | 指定数据是从内存传输到外设、从内存传输到内存还是从外设传输到内存 |
| DMA_BufferSize | 缓冲区大小 | 以数据单位指定指定数据流的缓冲区大小① |
| DMA_PeripheralInc | DMA_PeripheralInc_Disable<br>DMA_PeripheralInc_Enable | 指定是否递增外设地址寄存器 |
| DMA_MemoryInc | DMA_MemoryInc_Disable：不递增<br>DMA_MemoryInc_Enable：递增 | 指定是否递增内存地址寄存器 |
| DMA_Peripheral DataSize | DMA_PeripheralDataSize_Byte<br>DMA_PeripheralDataSize_HalfWord<br>DMA_PeripheralDataSize_Word | 指定外设数据宽度 |
| DMA_Memory DataSize | DMA_MemoryDataSize_Byte<br>DMA 传输时的内存数据宽度为 1 个字节<br>DMA_MemoryDataSize_HalfWord<br>DMA 传输时的内存数据宽度为半个字节<br>DMA_MemoryDataSize_Word<br>DMA 传输时的内存数据宽度为 4 个字节 | 指定内存数据宽度 |
| DMA_Mode | DMA_Mode_Circular：循环模式<br>DMA_Mode_Normal：普通模式 | 指定 DMAy Streamx 的运行模式 |
| DMA_Priority | DMA_Priority_High：高<br>DMA_Priority_Low：低<br>DMA_Priority_Medium：中<br>DMA_Priority_VeryHigh：非常高 | 指定 DMAy Streamx 的软件优先级 |

| | | |
|---|---|---|
| DMA_FIFOMode | DMA_FIFOMode_Disable<br>直接模式<br>DMA_FIFOMode_Enable<br>FIFO 模式 | 指定指定数据流使用 FIFO 模式还是直接模式 |
| DMA_FIFOThreshold | DMA_FIFOThreshold_1QuarterFull<br>FIFO 缓冲区在填充到 1/4 时触发传输请求<br>DMA_FIFOThreshold_3QuartersFull<br>FIFO 缓冲区在填充到 3/4 时触发传输请求<br>DMA_FIFOThreshold_Full<br>FIFO 缓冲区在填充满时触发传输请求<br>DMA_FIFOThreshold_HalfFull<br>FIFO 缓冲区在填充到 1/2 时触发传输请求 | 指定先进先出阈值级别 |
| DMA_MemoryBurst | DMA_MemoryBurst_INC16<br>单次 16 个字节的传输量<br>DMA_MemoryBurst_INC8<br>单次 8 个字节的传输量<br>DMA_MemoryBurst_INC4<br>单次 4 个字节的传输量<br>DMA_MemoryBurst_Single<br>单次 1 个字节的传输量 | 指定内存传输的突发传输配置，它指定了单次非中断事务中传输的数据量 |
| DMA_PeripheralBurst | DMA_PeripheralBurst_INC16<br>DMA_PeripheralBurst_INC8<br>DMA_PeripheralBurst_INC4<br>DMA_PeripheralBurst_Single | 指定外设传输的突发传输配置，它指定了单次非中断事务中传输的数据量 |
| ①数据单位等于 DMA_PeripheralDataSize 或 DMA_MemoryDataSize 成员中设置的配置，具体取决于传输方向。 | | |

（2）DMA 初始化函数

表 4.6.6 DMA 初始化函数

| 接口函数：DMA_Init | |
|---|---|
| 函数原型 | void DMA_Init(DMA_Stream_TypeDef *DMAy_Streamx, DMA_InitTypeDef *DMA_InitStruct) |
| 功能描述 | 根据 DMA_InitStruct 结构体中的指定参数初始化 DMAy Streamx。 |
| 入口参数 | DMAy_Streamx：其中，y 可以是 1 或 2，用于选择 DMA；x 可以是 0 至 7，用于选择 DMA 数据流。 |
| | DMA_InitStruct：指向 DMA_InitTypeDef 结构体的指针，该结构体包含指定的 DMA 数据流的配置信息。 |

| | |
|---|---|
| 返回值 | 无 |
| 使用示例 | DMA_Init(DMA2_Stream0, &DMA_InitStructure); |
| 注意事项 | 在调用该函数之前，建议使用函数 DMA_GetCmdStatus()检查数据流是否已禁用。 |

（3）获取 DMAy Streamx 的启用状态函数

表 4.6.7 获取 DMAy Streamx 的启用状态函数

| 接口函数：DMA_GetCmdStatus | |
|---|---|
| 函数原型 | FunctionalState DMA_GetCmdStatus(DMA_Stream_TypeDef *DMAy_Streamx) |
| 功能描述 | 获取指定 DMAy Streamx 的 EN 位状态。 |
| 入口参数 | DMAy_Streamx：其中，y 可以是 1 或 2，用于选择 DMA；x 可以是 0 至 7，用于选择 DMA 数据流。 |
| 返回值 | 返回当前 DMAy Streamx 的状态，返回值可以为 ENABLE 或 DISABLE。 |
| 使用示例 | FunctionalState state = DMA_GetCmdStatus(DMA2_Stream0); |
| 注意事项 | （1）配置 DMA 数据流（DMA_Init()函数）并启用数据流后，建议检查（或等待）DMA 数据流是否有效启用。如果配置参数有误，数据数据流可能会一直处于禁用状态。<br>（2）禁用 DMA 数据流后，也建议检查（或等待）DMA 数据流是否已有效禁用。如果在数据传输过程中禁用了数据流，则将传输当前数据，只有在传输完该单个数据后才会有效禁用该数据流。 |

（4）启用或禁用指定的 DMAy Streamx 函数

表 4.6.8 启用或禁用指定的 DMAy Streamx 函数

| 接口函数：DMA_Cmd | |
|---|---|
| 函数原型 | void DMA_Cmd(DMA_Stream_TypeDef *DMAy_Streamx, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 DMAy Streamx。 |
| 入口参数 | DMAy_Streamx：其中，y 可以是 1 或 2，用于选择 DMA；x 可以是 0 至 7，用于选择 DMA 数据流。 |
| | NewState：DMAy Streamx 的状态，该参数可以是 ENABLE 或 DISABLE。 |
| 返回值 | 无 |

| | |
|---|---|
| 使用示例 | DMA_Cmd(DMA2_Stream0, DISABLE); |
| 注意事项 | 此函数可用于执行暂停-恢复操作。当传输正在进行时，调用此函数禁用数据流将导致传输暂停，所有配置寄存器和剩余数据数将被保留。再次调用该函数重新启用数据流时，传输将从暂停的位置重新开始。 |

（5）检查指定 DMAy Streamx 的标志设置与否函数

表 4.6.9 检查指定 DMAy Streamx 的标志设置与否函数

| 接口函数：DMA_GetFlagStatus | |
|---|---|
| 函数原型 | FlagStatus DMA_GetFlagStatus(DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_FLAG) |
| 功能描述 | 检查指定的 DMAy Streamx 标志是否已设置。 |
| 入口参数 | DMAy_Streamx：其中，y 可以是 1 或 2，用于选择 DMA；x 可以是 0 至 7，用于选择 DMA 数据流。 |
| | DMA_FLAG：指定要检查的标志，其参数可以为 DMA_FLAG_TCIFx（Streamx 传输完成标志）、DMA_FLAG_HTIFx（Streamx 半传输完成标志）、DMA_FLAG_TEIFx（Streamx 传输错误标志）、DMA_FLAG_DMEIFx（Streamx 直接模式错误标志）、DMA_FLAG_FEIFx（Streamx FIFO 错误标志），其中 x 可以为 0~7。 |
| 返回值 | 指定 DMA_FLAG 的状态，其返回值可以为 SET 或 RESET。 |
| 使用示例 | FlagStatus status = DMA_GetFlagStatus(DMA2_Stream0, DMA_FLAG_TCIF0); |
| 注意事项 | 无 |

（6）清除 DMAy Streamx 的待处理标志函数

表 4.6.10 清除 DMAy Streamx 的待处理标志函数

| 接口函数：DMA_ClearFlag | |
|---|---|
| 函数原型 | void DMA_ClearFlag(DMA_Stream_TypeDef *DMAy_Streamx, uint32_t DMA_FLAG) |
| 功能描述 | 清除 DMAy Streamx 的指定待处理标志。 |
| 入口参数 | DMAy_Streamx：其中，y 可以是 1 或 2，用于选择 DMA；x 可以是 0 至 7，用于选择 DMA 数据流。 |
| | DMA_FLAG：指定要清除的待处理标志，其参数可以为 DMA_FLAG_TCIFx（Streamx 传输完成标志）、DMA_FLAG_HTIFx（Streamx 半传输完成标志）、DMA_FLAG_TEIFx（Streamx 传输错误标志）、DMA_FLAG_DMEIFx（Streamx 直接模式错误标志）、DMA_FLAG_FEIFx（Streamx FIFO 错误标志），其中 x 可以为 0~7。 |

| | |
|---|---|
| 返回值 | 无 |
| 使用示例 | DMA_ClearFlag(DMA2_Stream0, DMA_FLAG_TCIF0); |
| 注意事项 | 无 |

**9. ADC DMA 请求相关函数讲解**

（1）启用或禁用指定的 ADC DMA 请求函数

表 4.6.11 启用或禁用指定的 ADC DMA 请求函数

| 接口函数：ADC_DMACmd | |
|---|---|
| 函数原型 | void ADC_DMACmd(ADC_TypeDef *ADCx, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 ADC DMA 请求。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| | NewState：所选 ADC DMA 传输状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | ADC_DMACmd(ADC1, ENABLE); |
| 注意事项 | 无 |

**10. USART DMA 请求相关函数讲解**

（1）启用或禁用 USART 的 DMA 接口函数

表 4.6.12 启用或禁用 USART 的 DMA 接口函数

| 接口函数：USART_DMACmd | |
|---|---|
| 函数原型 | void USART_DMACmd(USART_TypeDef *USARTx, uint16_t USART_DMAReq, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定 USART/UART 的 DMA 接口函数。 |
| 入口参数 | USARTx：指定的 USART/UART 外设，其中 x 可以为 1~8。 |
| | USART_DMAReq：指定 DMA 请求，其参数可以为 USART_DMAReq_Tx（USART DMA 发送请求）、USART_DMAReq_Rx（USART DMA 接收请求）。 |
| | NewState：DMA 请求源的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | USART_DMACmd(USART3, USART_DMAReq_Tx, ENABLE); |
| 注意事项 | 无 |

## 4.6.2 硬件设计

DMA 相关内容不涉及外部硬件电路。

## 4.6.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 DMA 部分的驱动代码独立开来，源文件（dma.c）和头文件（dma.h）存放在实验案例根目录中的 HARDWARE/DMA 下。

**1. 软件设计流程**

如图 4.6.1，展示了本实验的软件设计流程图。

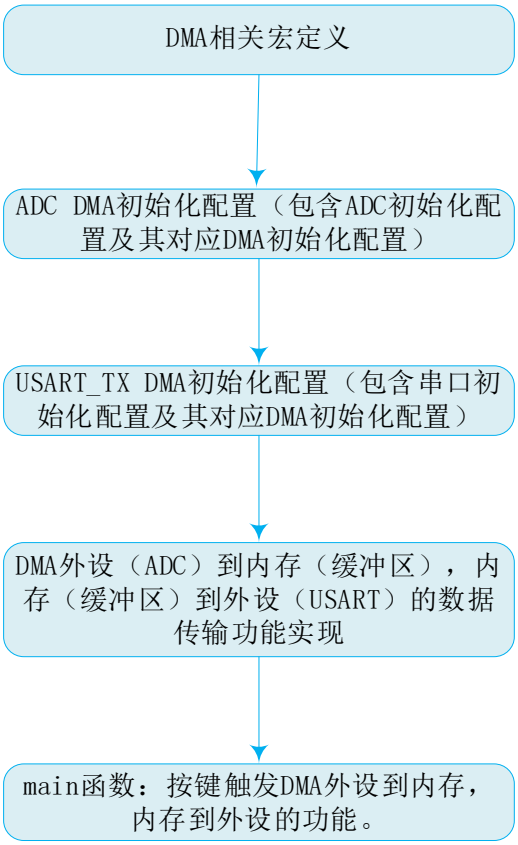



图 4.6.1 软件设计流程图

**2. 软件代码讲解**

（1）定义 DMA 相关的宏。

代码段 4.6.2 DMA 宏定义（dma.h）

```
//缓冲区大小
#define BUFFERSIZE                  100

/********************ADC DMA 相关定义********************/
//ADC      对应 DMA      对应通道      对应数据流
//ADC1     DMA2          通道 0        数据流 0
#define ADC_DMA                     DMA2
#define ADC_DMA_CHANNEL             DMA_Channel_0
#define ADC_DMA_STREAM              DMA2_Stream0

//DMA 时钟挂载在 AHB1 上
#define ADC_DMA_CLK                 RCC_AHB1Periph_DMA2
```

```
//DMA 时钟使能，其为宏函数定义
#define ADC_DMA_CLK_INIT            RCC_AHB1PeriphClockCmd

/*******************USART_TX DMA*******************/
//USART          对应 DMA        对应通道          对应数据流
//USART3_TX      DMA1            通道 4             数据流 3
#define USART_DMA                   DMA1
#define USART_TX_DMA_CHANNEL        DMA_Channel_4
#define USART_TX_DMA_STREAM         DMA1_Stream3

//DMA 时钟挂载在 AHB1 上
#define USART_DMA_CLK               RCC_AHB1Periph_DMA1

//DMA 时钟使能，其为宏函数定义
#define USART_DMA_CLK_INIT          RCC_AHB1PeriphClockCmd

//DMA 传输完成标志，对应数据流 3
#define USART_TX_DMA_FLAG_TCIF      DMA_FLAG_TCIF3
```

（2）ADC DMA 初始化配置函数实现。

代码段 4.6.3 ADC DMA 初始化配置函数（dma.c）

```
uint16_t adcx = 0;                                  //用于存放数据的变量
char show[BUFFERSIZE] = {0};                        //用于存放打印数据的数组

/*
功   能：  DMA 初始化配置函数(ADC 通道)
参   数：  无
返回值：  无
*/
void DMA_ADC_Init(void)
{
    DMA_InitTypeDef DMA_InitStructure;

    ADC_DMA_CLK_INIT(ADC_DMA_CLK, ENABLE);          //开启 DMA 时钟
```

```
    ADC1_Init();                                          //ADC 初始化配置

    while(DMA_GetCmdStatus(ADC_DMA_STREAM) != DISABLE);   //等待 DMA 可以配置

    /*******************ADC DMA 初始化配置*******************/
    DMA_InitStructure.DMA_BufferSize = BUFFERSIZE;        //DMA 缓冲器大小
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;  //关闭 FIFO 模式
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;//FIFO 阈值
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single ;//内存突发传输每次转移单个数据
    DMA_InitStructure.DMA_MemoryDataSize = DMA_PeripheralDataSize_HalfWord;//外设单个数据的位
宽,长度 16 位
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable; //内存地址不递增
    DMA_InitStructure.DMA_Mode = DMA_Mode_Circular;       //DMA 运行模式，持续传输
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;//外设突发传输每次转移
单个数据
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_MemoryDataSize_HalfWord;//内存单个数据的位
宽,长度 16 位
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设寄存器地址不递增
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;   //DMA 优先级

    DMA_InitStructure.DMA_Channel = ADC_DMA_CHANNEL;      //选择传输通道
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralToMemory; //选择传输方式,数据由内存传输到外
设
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) (&(ADC1->DR));//选择传输数据外设的
地址
    DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t) &adcx;//选择传输数据内存的地址
    DMA_Init(ADC_DMA_STREAM, &DMA_InitStructure);         //初始化 ADC DMA 配置

    ADC_RegularChannelConfig(ADC1,ADC1_CH,1,ADC_SampleTime_480Cycles);//ADCX,ADC 通道,规则序
列,采样时间
    ADC_DMARequestAfterLastTransferCmd(ADC1, ENABLE);     //源数据变化时开启 DMA 传输
    ADC_DMACmd(ADC1, ENABLE);                             //使能 ADC 的 DMA 传输
    ADC_SoftwareStartConv(ADC1);                          //开启软件 ADC 功能
}
```

---

（3） USART_TX DMA 初始化配置函数实现。

代码段 4.6.4 USART_TX DMA 初始化配置函数（dma.c）

---

```
/*
功   能:  DMA 初始化配置函数(USART_TX)
```

```
参　数：  baudrate        串口波特率
返回值：  无
*/
void DMA_USART_Init(uint32_t baudrate)
{
    DMA_InitTypeDef  DMA_InitStructure;

    USART_DMA_CLK_INIT(USART_DMA_CLK, ENABLE);                  //开启 DMA 时钟

    USART3_Init(baudrate);                                      //串口初始化配置

    while(DMA_GetCmdStatus(USART_TX_DMA_STREAM) != DISABLE);    //等待 DMA 可以配置

    /*******************USART_TX DMA 初始化配置*******************/
    DMA_InitStructure.DMA_BufferSize = BUFFERSIZE;              //DMA 缓冲器大小
    DMA_InitStructure.DMA_FIFOMode = DMA_FIFOMode_Disable;      //关闭 FIFO 模式
    DMA_InitStructure.DMA_FIFOThreshold = DMA_FIFOThreshold_HalfFull;//FIFO 阈值
    DMA_InitStructure.DMA_MemoryBurst = DMA_MemoryBurst_Single; //内存突发传输每次转移单个数据
    DMA_InitStructure.DMA_MemoryDataSize = DMA_MemoryDataSize_Byte;//内存单个数据的位宽,长度 8
位
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;     //内存地址递增
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;               //DMA 运行模式,正常传输
    DMA_InitStructure.DMA_PeripheralBurst = DMA_PeripheralBurst_Single;//外设突发传输每次转移
单个数据
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_Byte;//内存单个数据的位
宽,长度 8 位
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;//外设寄存器地址不递增
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;         //DMA 优先级

    DMA_InitStructure.DMA_Channel = USART_TX_DMA_CHANNEL;       //选择传输通道
    DMA_InitStructure.DMA_DIR = DMA_DIR_MemoryToPeripheral;     //选择传输方式,数据由内存传输
到外设
    DMA_InitStructure.DMA_PeripheralBaseAddr = (uint32_t) (&(USART3->DR));//选择传输数据外设
的地址
    DMA_InitStructure.DMA_Memory0BaseAddr = (uint32_t) &show;   //选择传输数据内存的地址
    DMA_Init(USART_TX_DMA_STREAM,&DMA_InitStructure);           //初始化 USART_TX DMA 配置
}
-----------------------------------------------------------------------------------------------------------------------------------
```

（4） DMA 数据传输功能函数实现，包括从外设到内存（ADC 采集内部温度，将采集到的转换值传输至内存缓冲区）、从内存到外设（将计算好的转换值数据从内存缓冲区发送至串口）。

代码段 4.6.5 DMA 数据传输功能函数（dma.c）

```
/*
功　能：  ADC DMA 数据传输函数
          外设 --> 内存 （ADC 采集内部温度，将采集到的转换值传输至内存缓冲区）
参　数：  无
返回值：  无
*/
void DMA_ADC_Cmd(void)
{
    DMA_Cmd(ADC_DMA_STREAM, DISABLE);    //关闭 ADC DMA 数据流传输
    DMA_Cmd(ADC_DMA_STREAM, ENABLE);     //开启 ADC DMA 数据流传输
    ADC_SoftwareStartConv(ADC1);         //开启软件 ADC 功能
}
/*
功　能：  USART_TX DMA 数据传输函数
          内存 --> 外设 （将计算好的转换值数据从内存缓冲区发送至串口）
参　数：  无
返回值：  无
*/
void DMA_USART_Tx_Cmd(void)
{
    //ADC 采集内部温度，将采集到的转换值通过 DMA 传输至内存缓冲区
    DMA_ADC_Cmd();
    //将内存缓存区的 ADC 转换值计算并存放至串口发送缓冲区，等待串口 DMA 发送
     sprintf(show, "Temperature:%.2f ℃ \r\n", (double)(((adcx * (3.3 / 4096)) - 0.76) /
0.0025 + 25));
    USART_DMACmd(USART3, USART_DMAReq_Tx, ENABLE);//使能串口发送的 DMA 通道
    DMA_Cmd(USART_TX_DMA_STREAM, DISABLE);//关闭 USART_TX DMA 所指示的通道
    DMA_Cmd(USART_TX_DMA_STREAM, ENABLE);//开启 USART_TX DMA 所指示的通道
    while(1)                             //等待传输完成
    {
        //等待通道传输完成
        if(DMA_GetFlagStatus(USART_TX_DMA_STREAM, USART_TX_DMA_FLAG_TCIF) != RESET)
        {
            LED1(1);
```

```
            memset(show, 0x00, BUFFERSIZE);//清空字符串
            DMA_ClearFlag(USART_TX_DMA_STREAM, USART_TX_DMA_FLAG_TCIF); //清除传输完成标志
            break;
        }
    }
    Delay_ms(30);
    LED1(0);
}
```

（5） 主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而 while 循环中需要实现按键触发 DMA 外设到内存，内存到外设的功能。

代码段 4.6.6 主函数（main.c）

```
int main(void)
{
    uint8_t key_value = 0;
    Delay_Init();               //延时初始化
    Led_Init();                 //LED 初始化
    Key_Init();                 //按键初始化
    DMA_ADC_Init();             //DMA 初始化(外设到内存)
    DMA_USART_Init(115200);     //DMA 初始化(内存到外设)
    while(1)
    {
        key_value = Key_Scan(0);//检测按键
        switch(key_value)       //判断按键
        {
            //按键 KEY1 按下
            case 1:
                //使用 DMA 将 ADC 采集的数据传输至缓冲区（外设到内存），然后将缓冲区中的数据传输到串口（内存到外设）
                DMA_USART_Tx_Cmd();
                break;
            default:
                break;
        }
    }
}
```

## 4.6.1 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |

| ST Link 下载器 | USB 转 TTL 模块 |
|---|---|
| | |

2. 实验硬件接线：

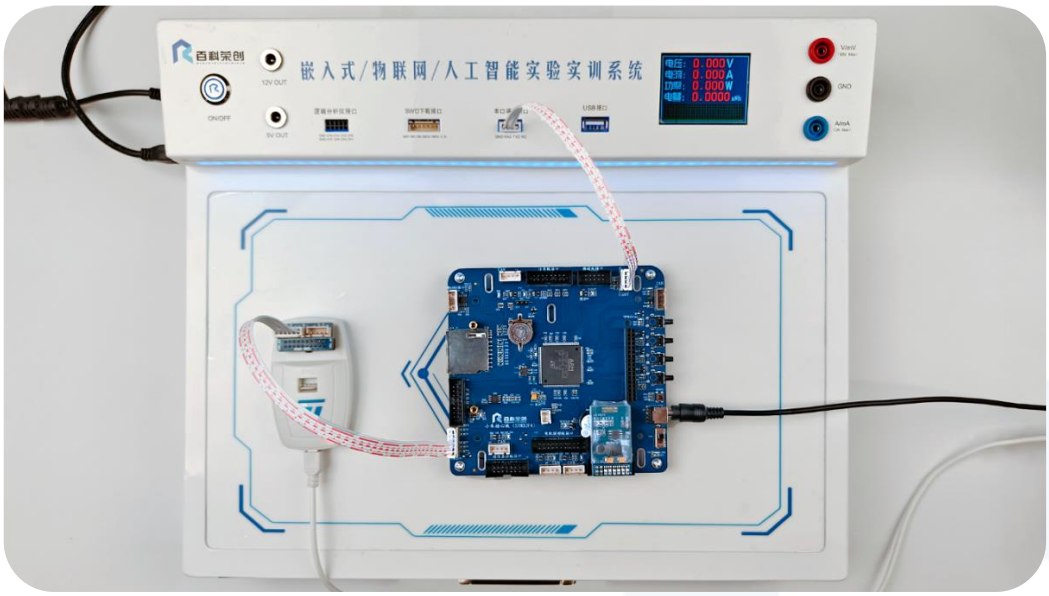



图 4.6.2 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英

文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，按下核心板按键 KEY1，观察串口调试助手是否打印 ADC 采集的内部温度。

## 4.6.2 结果展示

按下按键 KEY1，串口打印采集的芯片内部温度，LED1 闪烁一下。

其过程为：

外设-->内存（ADC 采集内部温度，将采集到的转换值传输至内存缓冲区）；

内存-->外设（将计算好的转换值数据从内存缓冲区发送至串口）。

如图 4.6.3 所示，展示了串口调试工具上显示的数据示意图。

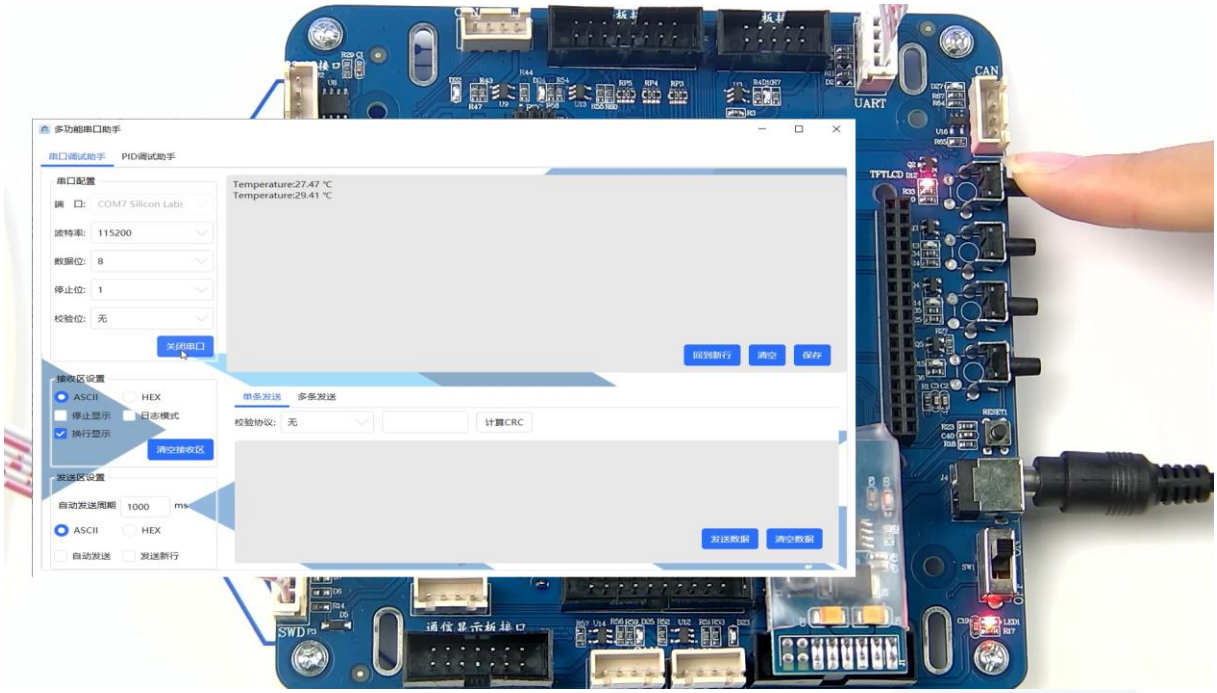



图 4.6.3 串口打印芯片内部温度值数据示意图

# 4.7 RTC 实时时钟实验

本实验旨在利用 STM32F4xx 内部的实时时钟（RTC）模块记录和管理精确时间。它通过配置和利用 RTC 模块的功能，来进行时间计数和时间信息获取。此外，还可以使用 RTC 模块的闹钟功能，实现定时触发事件的操作。

通过学习本实验，读者将能够深入了解 STM32F4 中 RTC 模块的工作原理和功能，并且掌握如何在 STM32F4 上进行精确的时间管理和记录。

## 4.7.1 实时时钟介绍

**1. RTC 概念**

实时时钟（RTC, real time clock），是一个独立的 BCD（Binary-Coded Decimal）定时器/计数器。RTC 提供了一个日历时钟、两个可编程闹钟中断，以及一个具有中断功能的周期性可编程唤醒标志。RTC 还包含用于管理低功耗模式的自动唤醒单元。

在 RTC 中，两个 32 位寄存器包含“二进制编码的十进制数”格式(BCD)的秒、分钟、小时（12 或 24 小时制）、星期、日期、月份和年份。此外，还可提供二进制格式的亚秒值。

系统也可以自动将月份的天数补偿为 28、29（闰年）、30 和 31 天。并且还可以进行夏令时补偿。其它 32 位寄存器还包含可编程的闹钟亚秒、秒、分钟、小时、星期和日期。

此外，还可以使用数字校准功能对晶振精度的偏差进行补偿。

在上电复位后，所有 RTC 寄存器都会受到保护，以防止可能的非正常写访问。无论器件状态如何（运行模式、低功耗模式或处于复位状态），只要电源电压保持在工作范围内，RTC 便不会停止工作。

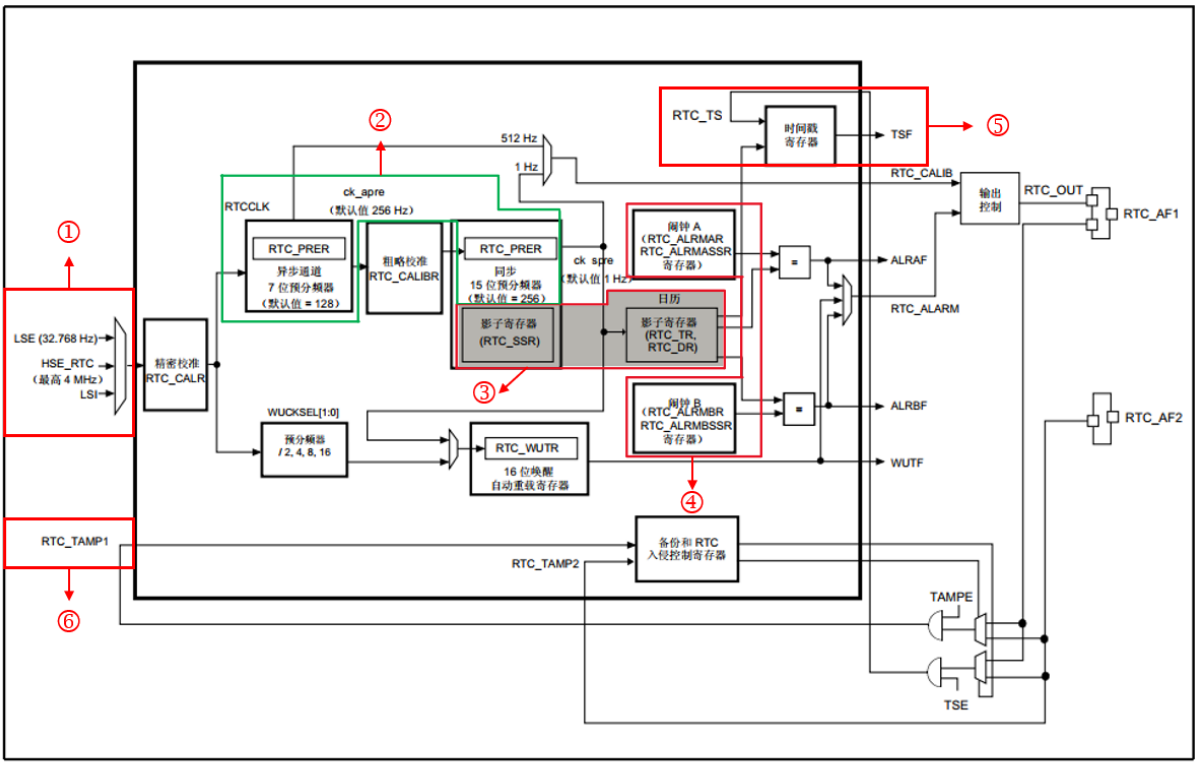
注：BCD 码，用 4 位二进制数来表示 1 位十进制中的 0~9 这 10 个数码，是一种二进制的数字编码形式，是用二进制编码的十进制数。

表 4.7.1 十进制数字对应的 BCD 表达式

| 十进制 | BCD 表达式 |
|---|---|
| 0 | 0000 |
| 1 | 0001 |
| 2 | 0010 |
| 3 | 0011 |
| 4 | 0100 |

| | |
|---|---|
| 5 | 0101 |
| 6 | 0110 |
| 7 | 0111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 0001 0000 |
| 11 | 0001 0001 |
| … | … |

**2. RTC 功能框图解析**




1. 对于 STM32F4xx 器件，RTC_AF1 和 RTC_AF2 复用功能分别与 PC13 和 PI8 相连。

图 4.7.1 RTC 功能框图

如图 4.7.1，展示了 RTC 的功能框图，其中：

**①时钟源：**RTC 时钟源(RTCCLK)通过时钟控制器从 LSE 时钟、LSI 振荡器时钟以及 HSE 时钟三者中选择。其中使用最多的是 LSE，LSE 由一个外部的 32.768KHZ 的晶振提供，精度高，稳定，是 RTC 的首选。LSI 是芯片内部的 30KHZ 晶体，精度较低，会有温漂，一般不建议使用。HSE_RTC 由 HSE 分频得到，最高是 4M，使用的也较少。

**②预分频器：**

可编程的预分频器阶段可生成 1Hz 的时钟，用于更新日历。

为最大程度地降低功耗，预分频器分为 2 个可编程的预分频器：一个 7 位异步预

分频器（通过 RTC_PRER 寄存器的 PREDIV_A 位配置）；一个 15 位同步预分频器（通过 RTC_PRER 寄存器的 PREDIV_S 位配置）。

当使用两个预分频器时，推荐将异步预分频器配置为较高的值，以最大程度降低功耗。其中：

（1）$f_{ck_apre}$ 可根据以下公式得出：

$$f_{CK_APRE} = \frac{f_{RTCCLK}}{PREDIV_A + 1}$$

ck_apre 时钟（异步预分频器时钟）用于为二进制 RTC_SSR 亚秒递减计数器提供时钟。当该计数器计数到 0 时，会使用 PREDIV_S 的内容重载 RTC_SSR。

（2）$f_{ck_spre}$ 可根据以下公式得出：

$$f_{CK_SPRE} = \frac{f_{RTCCLK}}{(PREDIV_S + 1) \times (PREDIV_A + 1)}$$

要使用频率为 32.768kHz 的 LSE 获得频率为 1 Hz 的内部时钟(ck_spre)，需要将异步预分频系数设置为 128，并将同步预分频系数设置为 256。

因此根据公式 $f_{RTCCLK}$=32.768kHz，PREDIV_S=255，PREDIV_A=127，得 $f_{ck_spre}$=1Hz。

ck_spre 时钟（同步预分频器时钟）既可以用于更新日历，也可以用作 16 位唤醒自动重载定时器的时基。为获得较短的超时周期，还可以将 16 位唤醒自动重载定时器与经可编程的 4 位异步预分频器分频的 RTCCLK 一同运行。

**③实时时钟和日历：**

RTC 日历时间和日期寄存器可通过与 PCLK1（APB1 时钟）同步的影子寄存器来访问。这些时间和日期寄存器也可以直接访问，这样可避免等待同步的持续时间。其中：RTC_SSR 寄存器对应于亚秒；RTC_TR 寄存器对应于时间（时分秒）；RTC_DR 寄存器对应于日期。

当应用程序读取日历寄存器时，默认情况是读取影子寄存器（可通过系统复位来复位）的内容，每隔两个 RTCCLK 周期，便将当前日历值复制到影子寄存器。

我们也可以通过将 RTC_CR 寄存器的 BYPSHAD 控制位置 1 来直接访问日历寄存器，这样可避免等待同步的持续时间。

注意：在 BYPSHAD=0 模式下读取 RTC_SSR、RTC_TR 或 RTC_DR 寄存器时，APB 时钟频率($f_{APB}$)必须至少为 RTC 时钟频率($f_{RTCCLK}$)的 7 倍。

RTCCLK 经过预分频器后，有 1 个 256Hz 的 ck_apre 和 1 个 1Hz 的 ck_spre，这两个时钟可以成为校准的时钟输出 RTC_CALIB，RTC_CALIB 最终要输出则需映射到 RTC_AF1 引脚，即 PC13 输出，用来对外部提供时钟。

**④可编程闹钟：**

RTC 单元提供两个可编程闹钟，即闹钟 A 和闹钟 B。

可通过将 RTC_CR 寄存器中的 ALRAE 和 ALRBE 位置 1 来使能可编程闹钟功能。如果日历亚秒、秒、分钟、小时、日期或日分别与预设的闹钟时间相同的时候，则 ALRAF 和 ALRBF 标志会被置为 1。

可通过 RTC_CR 寄存器中的 ALRAIE 和 ALRBIE 位使能闹钟中断。

闹钟 A 和闹钟 B（如果已通过 RTC_CR 寄存器中的位 OSEL[0:1]使能）可连接到 RTC_ALARM 输出。RTC_ALARM 最终连接到 RTC 的外部引脚 RTC_AF1（即 PC13），输出的极性由 RTC_CR 寄存器的 POL 位配置，可以是高电平或者低电平。

**⑤时间戳功能：**

时间戳即时间点的意思，就是某一个时刻的时间。时间戳复用功能(RTC_TS)可映射到 RTC_AF1 或 RTC_AF2，当发生外部的入侵事件时，即发生时间戳事件时，RTC_ISR 寄存器中的时间戳标志位（TSF）将置 1，日历会保存到时间戳寄存器（RTC_TSSSR、RTC_TSTR 和 RTC_TSDR）中。时间戳往往用来记录危及时刻的时间，以供事后排查问题时查询

**⑥入侵检测：**

RTC 自带两个入侵检测引脚 RTC_AF1（PC13）和 RTC_AF2（PI8），这两个输入既可配置为边沿检测，也可配置为带过滤的电平检测。当发生入侵检测时，备份寄存器将被复位。

**3. RTC 中断**

所有 RTC 中断均与 EXTI 控制器相连，其中断/事件线映射表参见表 4.7.2。

表 4.7.2 外部中断/事件线映射表（RTC 部分）

| 中断/事件线 | 输入源 | 外部中断通道 | 外部中断服务函数 |
|---|---|---|---|
| EXTI17 | RTC 闹钟事件 | RTC_Alarm_IRQn | RTC_Alarm_IRQHandler |
| EXTI21 | 入侵和时间戳事件 | TAMP_STAMP_IRQn | TAMP_STAMP_IRQHandler |
| EXTI22 | RTC 唤醒事件 | RTC_WKUP_IRQn | RTC_WKUP_IRQHandler |

其中：

（1）要使能 RTC 闹钟中断，需按照以下顺序操作：

将 EXTI 线 17 配置为中断模式并将其使能，然后选择上升沿有效——>配置 NVIC 中的 RTC_Alarm IRQ 通道并将其使能——>配置 RTC 以生成 RTC 闹钟（闹钟 A 或闹钟 B）

（2）要使能 RTC 唤醒中断，需按照以下顺序操作：

将 EXTI 线 22 配置为中断模式并将其使能，然后选择上升沿有效——>配置 NVIC 中的 RTC_WKUPIRQ 通道并将其使能——>配置 RTC 以生成 RTC 唤醒定时器事件。

（3）要使能 RTC 入侵中断，需按照以下顺序操作：

将 EXTI 线 21 配置为中断模式并将其使能，然后选择上升沿有效——>配置 NVIC 中的 TAMP_STAMPIRQ 通道并将其使能——>配置 RTC 以检测 RTC 入侵事件。

（4）要使能 RTC 时间戳中断，需按照以下顺序操作：

将 EXTI 线 21 配置为中断模式并将其使能，然后选择上升沿有效——>配置 NVIC 中的 TAMP_STAMPIRQ 通道并将其使能——>配置 RTC 以检测 RTC 时间戳事件。

**4. PWR（电源控制器）相关函数讲解**

（1）启用或禁用备份调压器函数

表 4.7.3 启用或禁用备份调压器函数

| 接口函数：PWR_BackupRegulatorCmd | |
|---|---|
| 函数原型 | void PWR_BackupRegulatorCmd(FunctionalState NewState) |
| 功能描述 | 启用或禁用备份调压器（Backup Regulator）。 |
| 入口参数 | NewState：备份调压器（Backup Regulator）的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | PWR_BackupRegulatorCmd(ENABLE); |
| 注意事项 | 无 |

**5. RCC 相关函数讲解**

（1）配置外部低速（LSE）振荡器函数

表 4.7.4 配置外部低速（LSE）振荡器函数

| 接口函数：RCC_LSEConfig | |
|---|---|
| 函数原型 | void RCC_LSEConfig(uint8_t RCC_LSE) |
| 功能描述 | 配置外部低速振荡器（LSE）。 |
| 入口参数 | RCC_LSE：指定 LSE 的状态，其参数可以为：<br>RCC_LSE_OFF：关闭 LSE 振荡器，LSERDY 标志位在 6 个 LSE 振荡器时钟周期后变为低电平。<br>RCC_LSE_ON：打开 LSE 振荡器。<br>RCC_LSE_Bypass：带外部时钟的旁路 LSE 振荡器。 |
| 返回值 | 无 |
| 使用示例 | RCC_LSEConfig(RCC_LSE_ON); |
| 注意事项 | 由于 LSE 位于备份域（Backup domain）中，且复位后该域会拒绝写入访问，因此在配置 LSE 之前，必须使用 PWR_BackupAccessCmd(ENABLE)函数启用写入访问（复位后必须执行一次）。<br>启用 LSE（RCC_LSE_ON 或 RCC_LSE_Bypass）后，应用软件应等待 LSERDY 标志置位，表明 LSE 时钟稳定，可用于 RTC 时钟。 |

（2）获取指定 RCC 标志位的状态函数

表 4.7.5 获取指定 RCC 标志位的状态函数

| 接口函数：RCC_GetFlagStatus | |
|---|---|
| 函数原型 | FlagStatus RCC_GetFlagStatus(uint8_t RCC_FLAG) |
| 功能描述 | 检查指定的 RCC 标志位是否被设置。 |
| 入口参数 | RCC_FLAG：被检查的指定 RCC 标志位，其参数可以为：<br>RCC_FLAG_HSIRDY（HSI 振荡器时钟就绪）、<br>RCC_FLAG_HSERDY（HSE 振荡器时钟就绪）、<br>RCC_FLAG_LSERDY（LSE 振荡器时钟就绪）、<br>RCC_FLAG_LSIRDY（LSI 振荡器时钟就绪）…… |
| 返回值 | 指定 RCC 标志位（RCC_FLAG）的状态，返回值可以为 SET 或 RESET。 |
| 使用示例 | FlagStatus status = RCC_GetFlagStatus(RCC_FLAG_LSERDY); |
| 注意事项 | 无 |

（3）配置 RTC 时钟（RTCCLK）函数

表 4.7.6 配置 RTC 时钟（RTCCLK）函数

| 接口函数：RCC_RTCCLKConfig | |
|---|---|
| 函数原型 | void RCC_RTCCLKConfig(uint32_t RCC_RTCCLKSource) |
| 功能描述 | 根据指定的 RTC 时钟源配置 RTC 时钟。 |
| 入口参数 | RCC_RTCCLKSource：指定的 RTC 时钟源，其参数可以为：<br>RCC_RTCCLKSource_LSE（将 LSE 作为 RTC 时钟源）、<br>RCC_RTCCLKSource_LSI（将 LSI 作为 RTC 时钟源）、<br>RCC_RTCCLKSource_HSE_Divx（将 HSE x 分频后作为 RTC 时钟源，x 可以为 2~31）。 |
| 返回值 | 无 |
| 使用示例 | RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE); |
| 注意事项 | （1）由于“RTC 时钟配置位”位于备份域（Backup domain）中，且复位后该域会拒绝写入访问，因此在配置 RTC 时钟源之前，必须使用 PWR_BackupAccessCmd(ENABLE)函数启用写入访问（复位后必须执行一次）。<br>（2）一旦配置了 RTC 时钟，除非使用 RCC_BackupResetCmd()函数复位备份域，或使用上电复位（POR），否则无法更改。<br>（3）如果将 LSE 或 LSI 用作 RTC 时钟源，RTC 将在 STOP 和 STANDBY 模式下继续工作，并可用作唤醒源。但是，当 HSE 时钟用作 RTC 时钟源时，RTC 不能在 STOP 和 STANDBY 模式下使用。<br>（4）使用 HSE 作为 RTC 时钟源时，RTC 的最大输入时钟频率为 1MHz。 |

（4）启用或禁用 RTC 时钟函数

表 4.7.7 启用或禁用 RTC 时钟函数

| 接口函数：RCC_RTCCLKCmd | |
|---|---|
| 函数原型 | void RCC_RTCCLKCmd(FunctionalState NewState) |
| 功能描述 | 启用或禁用 RTC 时钟。 |
| 入口参数 | NewState：RTC 时钟的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | RCC_RTCCLKCmd(ENABLE); |
| 注意事项 | 该函数只能在使用 RCC_RTCCLKConfig 函数选择完 RTC 时钟源后才能被使用。 |

## 6. RTC 相关函数讲解

（1）RTC 初始化结构体定义

代码段 4.7.1 RTC 初始化结构体定义

```
typedef struct
{
  uint32_t  RTC_HourFormat;
  uint32_t  RTC_AsynchPrediv;
  uint32_t  RTC_SynchPrediv;
}RTC_InitTypeDef;
```

关于 RTC 初始化结构体各个成员的详细描述，如表 4.7.8 所示。

表 4.7.8 RTC 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| RTC_HourFormat | RTC_HourFormat_12<br>RTC_HourFormat_24 | 指定 RTC 小时格式 |
| RTC_AsynchPrediv | 该参数值必须小于 0x7F | 指定 RTC 异步分频器值 |
| RTC_SynchPrediv | 该参数值必须小于 0x7FFF | 指定 RTC 同步分频器值 |

（2）从 RTC 备份数据寄存器中读取数据函数

表 4.7.9 从 RTC 备份数据寄存器中读取数据函数

| 接口函数：RTC_ReadBackupRegister | |
|---|---|
| 函数原型 | uint32_t RTC_ReadBackupRegister(uint32_t RTC_BKP_DR) |
| 功能描述 | 从指定的 RTC 备份数据寄存器读取数据。 |
| 入口参数 | RTC_BKP_DR：RTC 备份数据寄存器编号，其参数可以为 RTC_BKP_DRx（其中 x 可以为 0~19）。 |
| 返回值 | 返回从指定 RTC 备份数据寄存器中读取的数据。 |
| 使用示例 | uint32_t bkp_data = RTC_ReadBackupRegister(RTC_BKP_DR0); |
| 注意事项 | 无 |

（3）向指定 RTC 备份数据寄存器写入数据函数

表 4.7.10 向指定 RTC 备份数据寄存器写入数据函数

| 接口函数：RTC_WriteBackupRegister | |
|---|---|
| 函数原型 | void RTC_WriteBackupRegister(uint32_t RTC_BKP_DR, uint32_t Data) |
| 功能描述 | 向指定 RTC 备份数据寄存器写入数据。 |
| 入口参数 | RTC_BKP_DR：RTC 备份数据寄存器编号，其参数可以为 RTC_BKP_DRx（其中 x 可以为 0~19）。 |
| | Data：要写入指定的 RTC 备份数据寄存器的数据。 |
| 返回值 | 无 |
| 使用示例 | RTC_WriteBackupRegister(RTC_BKP_DR0, 0x1234); |
| 注意事项 | 无 |

（4）RTC 时间结构体定义

代码段 4.7.2 RTC 时间结构体定义

```
typedef struct
{
  uint8_t   RTC_Hours;
  uint8_t   RTC_Minutes;
  uint8_t   RTC_Seconds;
  uint8_t   RTC_H12;
}RTC_TimeTypeDef;
```

关于 RTC 时间结构体各个成员的详细描述，如表 4.7.11 所示。

表 4.7.11 RTC 时间结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| RTC_Hours | （1）如果 RTC 小时格式为 RTC_HourFormat_12，则参数值为 0~12。<br>（2）如果 RTC 小时格式为 RTC_HourFormat_24，则参数值为 0~23。 | 指定 RTC 时间的小时 |
| RTC_Minutes | 0~59 | 指定 RTC 时间的分钟 |

| | | |
|---|---|---|
| RTC_Seconds | 0~59 | 指定 RTC 时间的秒钟 |
| RTC_H12 | RTC_H12_AM<br>RTC_H12_PM | 指定 RTC AM/PM 时间 |

（5）设置 RTC 当前时间函数

表 4.7.12 设置 RTC 当前时间函数

| 接口函数：RTC_SetTime | |
|---|---|
| 函数原型 | ErrorStatus RTC_SetTime(uint32_t RTC_Format, RTC_TimeTypeDef *RTC_TimeStruct) |
| 功能描述 | 设置 RTC 当前时间。 |
| 入口参数 | RTC_Format：指定输入参数的格式，其参数可以为 RTC_Format_BIN（二进制数据格式）或 RTC_Format_BCD（BCD 数据格式）。 |
| | RTC_TimeStruct：指向 RTC_TimeTypeDef 结构体的指针，该结构体包含 RTC 时间配置信息。 |
| 返回值 | ErrorStatus 枚举值，返回值可以为 SUCCESS（RTC 时间寄存器已配置）或 ERROR（RTC 时间寄存器未被配置）。 |
| 使用示例 | ErrorStatus status = RTC_SetTime(RTC_Format_BIN, &RTC_TimeTypeInitStructure); |
| 注意事项 | 无 |

（6）RTC 日期结构体定义

代码段 4.7.3 RTC 日期结构体定义

```
typedef struct
{
  uint8_t   RTC_WeekDay;
  uint8_t   RTC_Month;
  uint8_t   RTC_Date;
  uint8_t   RTC_Year;
}RTC_DateTypeDef;
```

关于 RTC 日期结构体各个成员的详细描述，如表 4.7.13 所示。

表 4.7.13 RTC 日期结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| RTC_WeekDay | RTC_Weekday_x（x 可以为 Monday、Tuesday、Wednesday、Thursday、Friday、Saturday、Sunday） | 指定 RTC 日期——星期 |
| RTC_Month | RTC_Month_x（x 可以为 January、February、March、April、May、June、July、August、September、October、November、December） | 指定 RTC 日期——月份（BCD 格式） |
| RTC_Date | 1~31 | 指定 RTC 日期 |
| RTC_Year | 0~99 | 指定 RTC 日期——年份 |

（7）设置 RTC 当前日期函数

表 4.7.14 设置 RTC 当前日期函数

| 接口函数：RTC_SetDate | |
|---|---|
| 函数原型 | ErrorStatus RTC_SetDate(uint32_t RTC_Format, RTC_DateTypeDef *RTC_DateStruct) |
| 功能描述 | 设置 RTC 当前日期。 |
| 入口参数 | RTC_Format：指定输入参数的格式。其参数可以为 RTC_Format_BIN（二进制数据格式）或 RTC_Format_BCD（BCD 数据格式）。 |
| | RTC_DateStruct：指向 RTC_DateTypeDef 结构体的指针，该结构体包含 RTC 日期配置信息。 |
| 返回值 | ErrorStatus 枚举值，返回值可以为 SUCCESS（RTC 日期寄存器已配置）或 ERROR（RTC 日期寄存器未被配置）。 |
| 使用示例 | ErrorStatus status = RTC_SetTime(RTC_Format_BIN, &RTC_DateTypeInitStructure); |
| 注意事项 | 无 |

（8）启用或禁用 RTC 唤醒定时器函数

表 4.7.15 启用或禁用 RTC 唤醒定时器函数

| 接口函数：RTC_WakeUpCmd | |
|---|---|
| 函数原型 | ErrorStatus RTC_WakeUpCmd(FunctionalState NewState) |
| 功能描述 | 启用或禁用 RTC 唤醒定时器。 |
| 入口参数 | NewState：RTC 唤醒定时器的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | ErrorStatus 枚举值，返回值可以为 SUCCESS 或 ERROR。 |
| 使用示例 | RTC_WakeUpCmd(ENABLE); |
| 注意事项 | 无 |

（9）配置 RTC 唤醒时钟源函数

表 4.7.16 配置 RTC 唤醒时钟源函数

| 接口函数：RTC_WakeUpClockConfig | |
|---|---|
| 函数原型 | void RTC_WakeUpClockConfig(uint32_t RTC_WakeUpClock) |
| 功能描述 | 配置 RTC 唤醒时钟源。 |
| 入口参数 | RTC_WakeUpClock：唤醒时钟源，其参数可以为：<br>（1）RTC_WakeUpClock_RTCCLK_Divx（其中 x 可以为 16、8、4、2），RTC 唤醒计数器时钟计算公式：RTC Wakeup Counter Clock = RTCCLK/x。<br>（2）RTC_WakeUpClock_CK_SPRE_16bits（或 17bits），RTC 唤醒计数器时钟计算公式：RTC Wakeup Counter Clock = CK_SPRE（频率为 1Hz 的内部时钟）。 |
| 返回值 | 无 |
| 使用示例 | RTC_WakeUpClockConfig(RTC_WakeUpClock_CK_SPRE_16bits); |
| 注意事项 | 只有禁用 RTC 唤醒功能（使用 RTC_WakeUpCmd(DISABLE)）时，才能更改唤醒时钟源。 |

（10）配置 RTC 唤醒计数器值函数

表 4.7.17 配置 RTC 唤醒计数器函数

| 接口函数：RTC_SetWakeUpCounter | |
|---|---|
| 函数原型 | void RTC_SetWakeUpCounter(uint32_t RTC_WakeUpCounter) |
| 功能描述 | 配置 RTC 唤醒计数器值。 |
| 入口参数 | RTC_WakeUpCounter：指定唤醒计数器值。该参数的取值范围为 0x0000 至 0xFFFF。 |
| 返回值 | 无 |
| 使用示例 | RTC_SetWakeUpCounter(0); |
| 注意事项 | 只有禁用 RTC 唤醒功能（使用 RTC_WakeUpCmd(DISABLE)）时，才能写入 RTC 唤醒计数器值。 |

（11）清除 RTC 中断待处理位函数

表 4.7.18 清除 RTC 中断待处理位函数

| 接口函数：RTC_ClearITPendingBit | |
|---|---|
| 函数原型 | void RTC_ClearITPendingBit(uint32_t RTC_IT) |
| 功能描述 | 清除指定 RTC 中断的待处理位。 |
| 入口参数 | RTC_IT：指定要清除的 RTC 中断待处理位。<br>该参数可以是以下值的任意组合：<br>RTC_IT_TS（时间戳中断）、RTC_IT_WUT（唤醒定时器中断）、RTC_IT_ALRA（闹钟 A 中断）、RTC_IT_ALRB（闹钟 B 中断）、RTC_IT_TAMP1（Tamper 1 事件中断）、RTC_IT_TAMP2（Tamper 2 事件中断）。 |
| 返回值 | 无 |
| 使用示例 | RTC_ClearITPendingBit(RTC_IT_WUT); |
| 注意事项 | 无 |

（12）启用或禁用 RTC 中断函数

表 4.7.19 启用或禁用 RTC 中断函数

| 接口函数：RTC_ITConfig | |
|---|---|
| 函数原型 | void RTC_ITConfig(uint32_t RTC_IT, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 RTC 中断。 |
| 入口参数 | 指定要启用或禁用的 RTC 中断源。该参数可以是以下值的任意组合：RTC_IT_TS（时间戳中断）、RTC_IT_WUT（唤醒定时器中断）、RTC_IT_ALRA（闹钟 A 中断）、RTC_IT_ALRB（闹钟 B 中断）、RTC_IT_TAMP（入侵事件中断）。 |
| | NewState：指定 RTC 中断的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | RTC_ITConfig(RTC_IT_WUT, ENABLE); |
| 注意事项 | 无 |

（13）检查 RTC 标志是否设置函数

表 4.7.20 检查 RTC 标志是否设置函数

| 接口函数：RTC_GetFlagStatus | |
|---|---|
| 函数原型 | FlagStatus RTC_GetFlagStatus(uint32_t RTC_FLAG) |
| 功能描述 | 检查指定的 RTC 标志是否设置。 |
| 入口参数 | RTC_FLAG：指定要检查的 RTC 标志，其参数可以为：RTC_FLAG_WUTF（唤醒定时器标志）、RTC_FLAG_RECALPF（RECALPF 事件标志）、RTC_FLAG_ALRAF（闹钟 A 标志）、RTC_FLAG_ALRBF（闹钟 B 标志）、RTC_FLAG_INITF（初始化模式标志）、RTC_FLAG_WUTWF（唤醒定时器写标志）…… |
| 返回值 | 指定 RTC_FLAG 的状态，其参数可以为 SET 或 RESET。 |
| 使用示例 | FlagStatus status = RTC_GetFlagStatus(RTC_FLAG_WUTF); |
| 注意事项 | 无 |

（14）清除 RTC 的待处理标志函数

表 4.7.21 清除 RTC 的待处理标志函数

| 接口函数：RTC_ClearFlag | |
|---|---|
| 函数原型 | void RTC_ClearFlag(uint32_t RTC_FLAG) |
| 功能描述 | 清除指定 RTC 的待处理标志。 |
| 入口参数 | RTC_FLAG：指定要清除的 RTC 标志，该参数可以是以下值的任意组合：RTC_FLAG_RSF（寄存器同步标志）、RTC_FLAG_ALRAF（闹钟 A 标志）、RTC_FLAG_ALRBF（闹钟 B 标志）、RTC_FLAG_WUTF（唤醒定时器标志）、RTC_FLAG_TSF（时间戳事件标志）、RTC_FLAG_TSOVF（时间戳溢出标志）、RTC_FLAG_TAMP1F（Tamper 1 事件标志）、RTC_FLAG_TAMP2F（Tamper 2 事件标志）。 |
| 返回值 | 无 |
| 使用示例 | RTC_ClearFlag(RTC_FLAG_WUTF); |
| 注意事项 | 无 |

（15）获取 RTC 当前日期函数

表 4.7.22 获取 RTC 当前日期函数

| 接口函数：RTC_GetDate | |
|---|---|
| 函数原型 | void RTC_GetDate(uint32_t RTC_Format, RTC_DateTypeDef *RTC_DateStruct) |
| 功能描述 | 获取 RTC 当前日期。 |
| 入口参数 | RTC_Format：指定返回参数的格式，其参数可以为 RTC_Format_BIN（二进制数据格式）或 RTC_Format_BCD（BCD 数据格式）。 |
| | RTC_DateStruct：指向 RTC_DateTypeDef 结构体的指针，该结构体包含返回的当前日期配置。 |
| 返回值 | 无 |
| 使用示例 | RTC_GetDate(RTC_Format_BIN, &RTC_DateStruct); |
| 注意事项 | 无 |

（16）获取 RTC 当前时间函数

表 4.7.23 获取 RTC 当前时间函数

| 接口函数：RTC_GetTime | |
|---|---|
| 函数原型 | void RTC_GetTime(uint32_t RTC_Format, RTC_TimeTypeDef *RTC_TimeStruct) |
| 功能描述 | 获取 RTC 当前时间。 |
| 入口参数 | RTC_Format：指定返回参数的格式，其参数可以为 RTC_Format_BIN（二进制数据格式）或 RTC_Format_BCD（BCD 数据格式）。 |
| | RTC_TimeStruct：指向 RTC_TimeTypeDef 结构体的指针，该结构体包含返回的当前时间配置。 |
| 返回值 | 无 |
| 使用示例 | RTC_GetTime(RTC_Format_BIN, &RTC_TimeStruct); |
| 注意事项 | 无 |

## 4.7.2 硬件设计

### 1. 实验电路分析

RTC 的功能部分不涉及外部硬件电路。如图 4.7.2 所示，为 RTC 备用电源原理图。

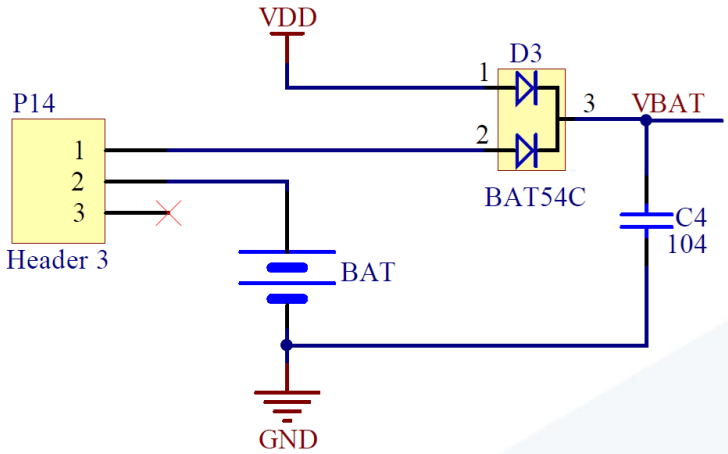



图 4.7.2 RTC 备用电源原理图

## 4.7.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 RTC 部分的驱动代码独立开来，源文件（rtc.c）和头文件（rtc.h）存放在实验案例根目录中的 HARDWARE/RTC 下。

**1. 软件设计流程**

如图 4.7.3，展示了本实验的软件设计流程图。

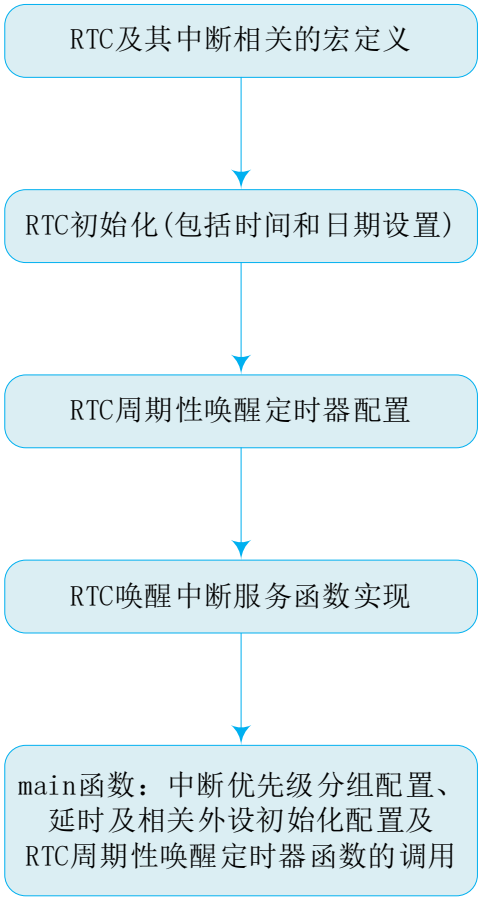



图 4.7.3 软件设计流程图

**2. 软件代码讲解**

（1）定义 RTC 及初始时分秒、日期、星期等相关的宏。

代码段 4.7.4 RTC 相关宏定义（rtc.h）

```
//PWR 时钟挂载在 APB1 上
#define PWR_CLOCK                    RCC_APB1Periph_PWR

//PWR 时钟使能，其为宏函数定义
#define PWR_CLK_INIT                 RCC_APB1PeriphClockCmd

/********************RTC 唤醒事件********************/
//中断线          中断通道            中断服务函数
//EXTI_Line22    RTC_WKUP_IRQn       RTC_WKUP_IRQHandler
```

```
#define RTC_WKUP_EXTI_Line          EXTI_Line22
#define RTC_WKUP_EXTI_IRQn          RTC_WKUP_IRQn
#define RTC_WKUP_EXTI_IRQHandler    RTC_WKUP_IRQHandler

//自定义小时、分钟、秒、年、月、日、星期
#define SET_HOUR                    23
#define SET_MIN                     58
#define SET_SEC                     14
#define SET_YEAR                    22
#define SET_MONTH                   12
#define SET_DATE                    31
#define SET_WEEK                    6

//修改初始日期需要将 SET_BKP_KEY 的宏定义值一并修改
#define SET_BKP_KEY                 0x2345
```

---

（2）RTC 初始化配置（包括 RTC 时间和日期设置）。

代码段 4.7.5 初始化函数（rtc.c）

---

```
/*
功   能:  RTC 初始化配置函数
参   数:  无
返回值:  0       初始化成功
         1       LSE 开启失败
*/
uint8_t RTC_Time_Init(void)
{
    RTC_InitTypeDef RTC_InitStructure;

    uint16_t retry = 0XFFF;

    uint16_t bkp_key = SET_BKP_KEY;     //加上电池后修改日期需要将 bkp_key 改写

    PWR_CLK_INIT(PWR_CLOCK, ENABLE);    //使能 PWR 时钟
    PWR_BackupAccessCmd(ENABLE);        //使能后备寄存器访问

    if (RTC_ReadBackupRegister(RTC_BKP_DR0) != bkp_key)         //判断是否第一次配置
    {
        RCC_LSEConfig(RCC_LSE_ON);          //LSE 开启外部低速时钟(32k)
```

```
        while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET)      //检查指定的 RCC 标志位设置与
否,等待低速晶振就绪
        {
            retry--;
            if (retry == 0)                                      //判断是否超时
                return 1;                                        //LSE 开启失败,返回 1
            Delay_ms(1);                                         //延时
        }

        RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);                  //设置 RTC 时钟(RTCCLK),选择
LSE 作为 RTC 时钟
        RCC_RTCCLKCmd(ENABLE);                                   //RTC 时钟使能

        RTC_InitStructure.RTC_AsynchPrediv = 0x7F;               //RTC 异步分频系数(1~0X7F)
        RTC_InitStructure.RTC_SynchPrediv = 0xFF;                //RTC 同步分频系数(0~0x7FFF)
        RTC_InitStructure.RTC_HourFormat = RTC_HourFormat_24;    //RTC 设置为,24 小时格式
        RTC_Init(&RTC_InitStructure);                            //初始化配置

        RTC_Set_Time(SET_HOUR, SET_MIN, SET_SEC, RTC_H12_AM);    //RTC 时钟设置
        RTC_Set_Date(SET_YEAR, SET_MONTH, SET_DATE, SET_WEEK);   //RTC 日期设置

        RTC_WriteBackupRegister(RTC_BKP_DR0, bkp_key);           //后备寄存器标记已经初始化过了
    }
    return 0;                                                    //初始化完成返回 0
}

/*
功   能:  RTC 时间设置函数
参   数:  hour          小时
          min           分钟
          sec           秒钟
返回值:  ERROR(0)       初始化失败
          SUCCESS(1)    初始化成功
*/
ErrorStatus RTC_Set_Time(uint8_t hour, uint8_t min, uint8_t sec, uint8_t ampm)
{
    RTC_TimeTypeDef RTC_TimeTypeInitStructure;

    RTC_TimeTypeInitStructure.RTC_Hours = hour;                          //设置小时
    RTC_TimeTypeInitStructure.RTC_Minutes = min;                         //设置分钟
    RTC_TimeTypeInitStructure.RTC_Seconds = sec;                         //设置秒钟
```

```
    RTC_TimeTypeInitStructure.RTC_H12 = ampm;                          //设置时间模式

    return RTC_SetTime(RTC_Format_BIN, &RTC_TimeTypeInitStructure);    //初始化时钟
}

/*
功   能:  RTC 日期设置函数
参   数:  year          年份
          month         月份
          date          日
          week          星期
返回值:  ERROR(0)       初始化失败
          SUCCESS(1)    初始化成功
*/
ErrorStatus RTC_Set_Date(uint8_t year, uint8_t month, uint8_t date, uint8_t week)
{
    RTC_DateTypeDef RTC_DateTypeInitStructure;

    RTC_DateTypeInitStructure.RTC_Date = date;                         //设置日
    RTC_DateTypeInitStructure.RTC_Month = month;                       //设置月份
    RTC_DateTypeInitStructure.RTC_WeekDay = week;                      //设置星期
    RTC_DateTypeInitStructure.RTC_Year = year;                         //设置年份

    return RTC_SetDate(RTC_Format_BIN, &RTC_DateTypeInitStructure);    //初始化日期
}
```

（3）RTC 周期性唤醒定时器配置函数实现。

代码段 4.7.6 RTC 周期性唤醒定时器配置函数（rtc.c）

```
/*
功   能:  RTC 周期性唤醒定时器配置函数
参   数:  wksel       分频系数
          cnt         自动重装载值
返回值:  无
*/
void RTC_Set_Wake_Up(uint32_t wksel, uint16_t cnt)
{
    EXTI_InitTypeDef EXTI_InitStructure;
    NVIC_InitTypeDef NVIC_InitStructure;
```

```
    RTC_WakeUpCmd(DISABLE);                          //关闭 WAKE UP
    RTC_WakeUpClockConfig(wksel);                    //唤醒时钟配置
    RTC_SetWakeUpCounter(cnt);                       //设置自动重装载寄存器

    RTC_ClearITPendingBit(RTC_IT_WUT);               //清除 RTC WAKE UP 的标志
    EXTI_ClearITPendingBit(RTC_WKUP_EXTI_Line);      //清除 RTC 唤醒中断线上的中断标志位
    RTC_ITConfig(RTC_IT_WUT, ENABLE);                //开启 WAKE UP 定时器中断
    RTC_WakeUpCmd(ENABLE);                           //开启 WAKE UP 定时器

    EXTI_InitStructure.EXTI_Line = RTC_WKUP_EXTI_Line;                    //中断线选择
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;                   //中断触发
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Rising;                //上升沿触发
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;                             //中断线使能
    EXTI_Init(&EXTI_InitStructure);                                       //初始化配置

    NVIC_InitStructure.NVIC_IRQChannel = RTC_WKUP_EXTI_IRQn;              //选择中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x00;          //设置先占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x0e;                 //设置从优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                        //设置中断通道开
启
    NVIC_Init(&NVIC_InitStructure);                                       //初始化配置中断优
先级
}
```

（4）RTC 唤醒中断服务函数实现（包含所需变量的定义），每触发一次中断，向串口发送时间及日期等信息。

代码段 4.7.7 RTC 唤醒中断服务函数（rtc.c）

```
//全局变量
static                    uint8_t                    showweek[7][10]                    =
{ "Monday","Tuesday","Wednesday","Thursday","Friday","Saturday","Sunday" };

/*
功   能:  RTC 唤醒中断服务函数
参   数:  无
返回值:  无
*/
void RTC_WKUP_EXTI_IRQHandler(void)
{
```

```
    RTC_TimeTypeDef RTC_TimeStruct;
    RTC_DateTypeDef RTC_DateStruct;
    char tbuf[60];

    if (RTC_GetFlagStatus(RTC_FLAG_WUTF) == SET)                //判断 ALARM WKUP 中断
    {
        RTC_ClearFlag(RTC_FLAG_WUTF);                           //清除中断标志
        RTC_GetDate(RTC_Format_BIN, &RTC_DateStruct);           //获取 RTC 日期

                    sprintf(tbuf,    "Date:20%02d-%02d-%02d\r\n",    RTC_DateStruct.RTC_Year,
RTC_DateStruct.RTC_Month, RTC_DateStruct.RTC_Date);

        USART3_Send_String(tbuf);

        sprintf(tbuf, "Week:%s\r\n", showweek[RTC_DateStruct.RTC_WeekDay - 1]);

        USART3_Send_String(tbuf);

        RTC_GetTime(RTC_Format_BIN, &RTC_TimeStruct);           //获取 RTC 时间

                  sprintf(tbuf,    "Time:%02d:%02d:%02d\r\n\r\n",    RTC_TimeStruct.RTC_Hours,
RTC_TimeStruct.RTC_Minutes, RTC_TimeStruct.RTC_Seconds);
        USART3_Send_String(tbuf);
    }

    EXTI_ClearITPendingBit(RTC_WKUP_EXTI_Line);                 //清除中断线的中断标志
}
```

（5）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了中断优先级分组配置、延时和相关外设初始化的流程，并调用 RTC 周期性唤醒定时器配置函数，而 while 循环中可以添加 LED1 状态翻转的功能（以表示程序正常运行）。

代码段 4.7.8 主函数（main.c）

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0);         //设置系统中断优先级分组 0
    Delay_Init();                                           //延时初始化
    Led_Init();                                             //LED 初始化
    RTC_Time_Init();                                        //实时时钟初始化
```

```
    USART3_Init(115200);                                    //串口初始化

    /*************************************************************************
        rtc 默认设置断电复位时间为 22 年 12 月 31 日
        (-----无纽扣电池)

        可将纽扣电池加上并修改设置正确的日期和时间
        (-----有纽扣电池)

        修改日期时的同时需要将 SET_BKP_KEY(在 rtc.h 中)的值改写
    *************************************************************************/

    //RTC 周期性唤醒定时器配置函数,1 秒钟中断一次
    RTC_Set_Wake_Up(RTC_WakeUpClock_CK_SPRE_16bits, 1 - 1);

    /*************************************************************************
        rtc 唤醒定时器设置:
        RTC_WakeUpClock_CK_SPRE_16bits      分频为 1Hz
        1                                   定时器重装载值
    *************************************************************************/

    while (1)
    {
        LED1_TOGGLE();      //LED1 状态翻转表示程序正常运行
        Delay_ms(500);      //延时
    }
}
```

---

## 4.7.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |
| ST Link 下载器 | USB 转 TTL 模块 | |
| | | |

2. 实验硬件接线：

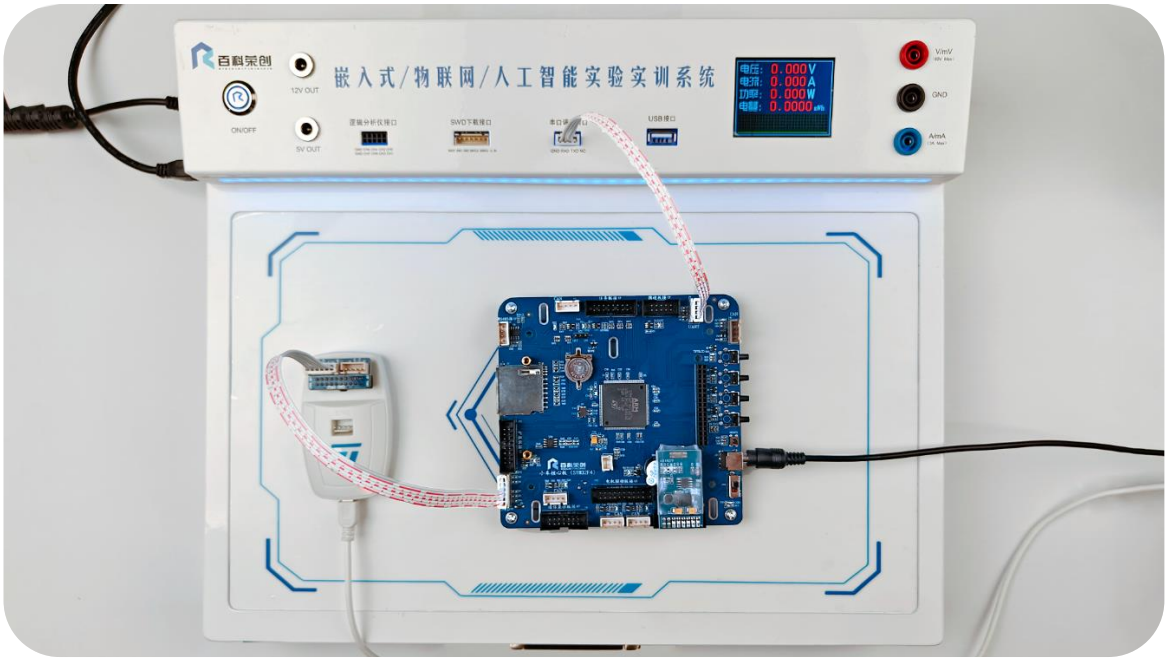



图 4.7.4 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，观察串口调试助手每秒是否打印日期和时间。

### 4.7.5 结果展示

串口调试助手每秒打印日期和时间。其串口打印格式如图 4.7.5 所示：

图 4.7.5 串口打印时间数据格式示意图

注：

（1）**无纽扣电池：**rtc 默认设置断电复位时间为 22 年 12 月 31 日 23：58：14（星期六）。

（2）**有纽扣电池：**可将纽扣电池加上并修改默认时间为当前正确的日期、时间和星期，编译下载，再让核心板断电一段后时间后重新上电，则时间打印与当前实际时间一致。

（3）修改默认的日期时的同时需要将 SET_BKP_KEY(在 rtc.h 中)的值改写，否则编译下载后默认初始时间不会改变。

# 4.8 LED 交替闪烁实验

本实验通过配置定时器中断和延时方式，实现 LED1（定时器中断）和 LED2（延时方式）状态的同时反转，并进一步观察其实验现象。

通过定时器中断和延时方式的组合应用，有助于读者深入理解定时器中断和延时在实际应用中的作用，并探索它们的优势和适用场景。

## 4.8.1 定时器介绍

**1. STM32 的定时器**

STM32F4xx 的定时器包含基本定时器（TIM6 和 TIM7）、通用定时器（TIM2 到 TIM5）、通用定时器（TIM9 到 TIM14）以及高级控制定时器（TIM1 和 TIM8），共计 14 个。

**2. 定时器之间的区别**

如表 4.8.1，展示了 STM32F4xx 不同定时器的区别。

表 4.8.1 STM32F4xx 定时器区别

| 定时器种类 | 位数 | 计数模式 | 产生 DMA 请求 | 捕获/比较通道 | 互补输出 | 特殊应用场景 |
|---|---|---|---|---|---|---|
| 高级定时器（TIM1、TIM8） | 16 | 向上，向下，向上/下 | 有 | 4 | 有 | 带可编程死区的互补输出… |
| 通用定时器（TIM2、TIM5） | 32 | 向上，向下，向上/下 | 有 | 4 | 无 | 通用。定时计数，PWM 输出，输入捕获，输出比较… |
| 通用定时器（TIM3、TIM4） | 16 | 向上，向下，向上/下 | 有 | 4 | 无 | 通用。定时计数，PWM 输出，输入捕获，输出比较… |
| 通用定时器（TIM9~TIM14） | 16 | 向上 | 无 | 2~1 | 无 | 通用。定时计数，PWM 输出，输入捕获，输出比较… |
| 基本定时器（TIM6、TIM7） | 16 | 向上 | 有 | 0 | 无 | 主要应用于驱动 DAC… |

**3. 定时器时基单元**

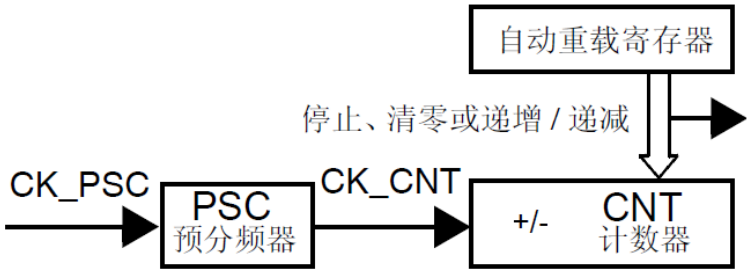



图 4.8.1 定时器时基单元

定时器中最核心的是时基单元，这是由三个器件组成的一个基础定时器。就定时器的向上计数模式而言，计数器获得分频后的时钟信号后可从 0 开始递增计数，当计数器中的数值自增至等于重装载值后，会立刻将更新中断位置 1，且计数器重新从 0 开始计数。

**4. 通用定时器（TIM2 到 TIM5）**

通用定时器（TIM2 到 TIM5）包含一个 16 位或 32 位自动重载计数器，该计数器由可编程预分频器驱动。它们可用于多种用途，包括测量输入信号的脉冲宽度（输入捕获）或生成输出波形（输出比较和 PWM）。

使用定时器预分频器和 RCC 时钟控制器预分频器，可将脉冲宽度和波形周期从几微秒调制到几毫秒。

并且这些定时器彼此完全独立，不共享任何资源。

**5. 计数模式**

在递增计数模式下：计数器从 0 计数到自动重载值，然后重新从 0 开始计数并生成计数器上溢事件。

在递减计数模式下：计数器从自动重载值开始递减计数到 0，然后重新从自动重载值开始计数并生成计数器下溢事件。

在中心对齐模式下：计数器从 0 开始计数到自动重载值减一（比如自动重载值为 100 那么计数到 99）并生成计数器上溢事件；然后从自动重载值开始向下计数到 1 并生成计数器下溢事件。之后从 0 开始重新计数。

如图 4.8.2 所示，可以更为简单的理解三种计数模式。

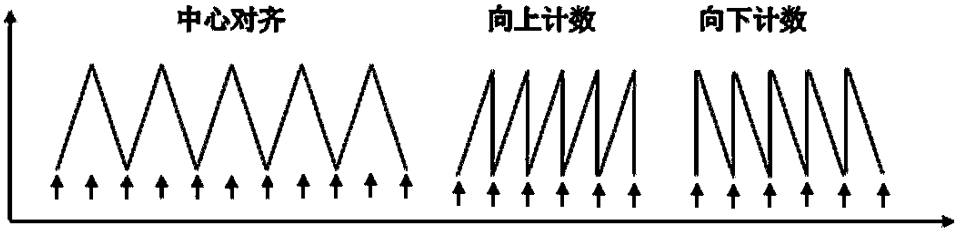



图 4.8.2 三种计数模式

**6. 定时器时钟信号分频与自动重装载值**

设置定时器时钟分频与装载值尤为重要。首先 STM32F4xx 的每个定时器时钟都挂载在不同的时钟总线上。如图 4.8.3 所示，展示了 STM32F405xx/STM32F407xx 挂载在 APB2 上的定时器，包括 TIM1、TIM8、TIM9、TIM10 和 TIM11。

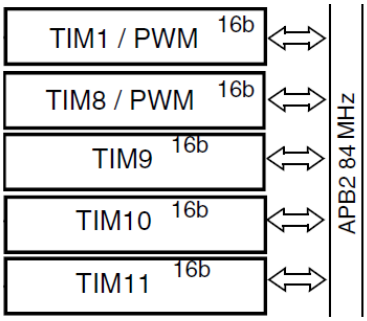



图 4.8.3 挂载在 APB2 上的定时器

其他定时器则挂载在 APB1 总线上，如图 4.8.4 所示。

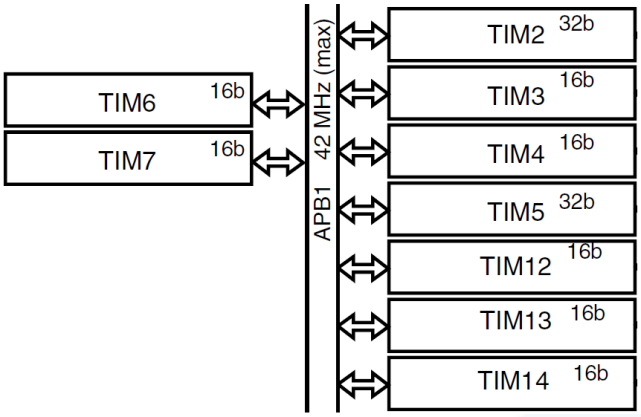



图 4.8.4 挂载在 APB1 上的定时器

虽然 APB1 与 APB2 总线的时钟频率有上限（42MHz 与 84MHz），但 APB1/APB2 总线分频系数（APB Prescaler）不为“/1”时进入定时器的时钟信号会被默认设为两倍频（由硬件控制），所以对应 APB 总线输入的定时器时钟信号由 42MHz、84MHz 变为了 84MHz、168MHz。情况如图 4.8.5 所示。

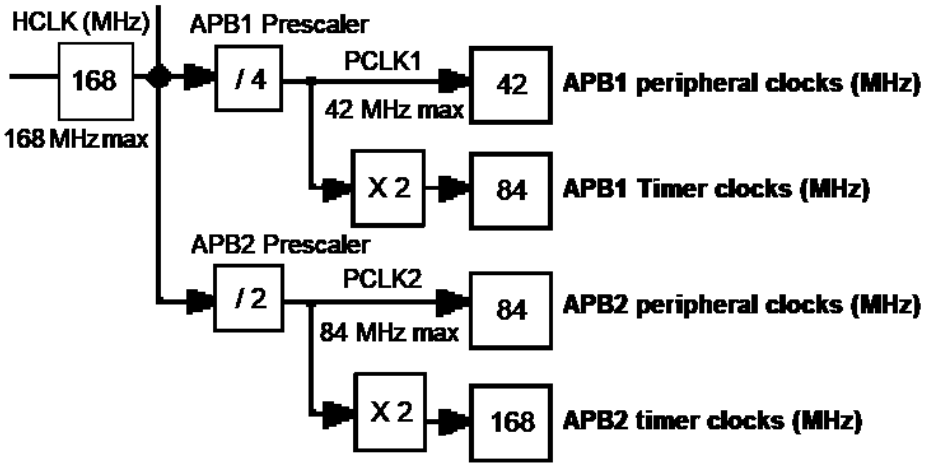



图 4.8.5 STM32F4 时钟信号

以定时器三向上计数模式作为例子。由图 4.8.4 可知 TIM3 的时钟信号最大为 84MHz，而时钟信号需要先经过时基单元的 PSC 预分频器才能进入计数器。已知预分频器可以进一步地对时钟信号进行分频。为更方便理解，可先设一个分频系数 n，当预分频器分频为“/n”时，计数器计数一次的耗时为“n/84M”秒。每当计数器计数到自动重装载值 m 时，计数器置位更新中断位后清零重新计数。所以每次重装载之间的耗时为 m*(n/84M)秒。所以可知：分频系数与计数值的乘积可以控制中断的触发间隔。设定时器三进行一毫秒中断一次，可得公式：

$$m * n = 84M * 0.001 = 84K = 84 * 1000$$

所以分频系数设为 84 是 1us 计数一次，计数 1000 次时长到达 1ms，产生一次定时器更新中断。

**7. 定时器相关库函数讲解**

（1）定时器时基单元初始化结构体

代码段 4.8.1 定时器时基单元初始化结构体定义

```
typedef struct
{
  uint16_t  TIM_Prescaler;
  uint16_t  TIM_CounterMode;
  uint32_t  TIM_Period;
  uint16_t  TIM_ClockDivision;
  uint8_t   TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;
```

关于定时器时基单元初始化结构体各个成员的详细描述，如表 4.8.2 所示。

表 4.8.2 定时器时基单元初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| TIM_Prescaler | 0x0000~0xFFFF | 指定预分频器值 |
| TIM_CounterMode | TIM_CounterMode_Up：<br>向上计数模式<br>TIM_CounterMode_Down：<br>向下计数模数<br>TIM_CounterMode_CenterAligned1<br>：<br>中心对齐模式 1<br>TIM_CounterMode_CenterAligned2<br>：<br>中心对齐模式 2<br>TIM_CounterMode_CenterAligned3<br>：<br>中心对齐模式 3 | 指定计数器模式 |
| TIM_Period | 0x0000~0xFFFF | 指定自动重装载值 |
| TIM_ClockDivision | TIM_CKD_DIV1<br>TIM_CKD_DIV2<br>TIM_CKD_DIV4 | 指定时钟分频 |
| TIM_RepetitionCounter | 0x00~0xFF | 指定重复计数器值<br>(仅限 TIM1、TIM8) |

（2）初始化定时器时基单元外设函数

表 4.8.3 初始化定时器时基单元外设函数

| 接口函数：TIM_TimeBaseInit | |
|---|---|
| 函数原型 | void TIM_TimeBaseInit(TIM_TypeDef *TIMx, TIM_TimeBaseInitTypeDef *TIM_TimeBaseInitStruct) |
| 功能描述 | 根据 TIM_TimeBaseInitStruct 中的指定参数初始化 TIMx 时基单元外设。 |
| 入口参数 | TIMx：x 可以为 1~14，表示不同的定时器外设。 |
| | TIM_TimeBaseInitStruct：指向 TIM_TimeBaseInitTypeDef 结构体的指针，该结构体包含 TIM 指定的外设配置信息。 |
| 返回值 | 无 |
| 使用示例 | TIM_TimeBaseInit(TIM2, &TIM_TimeBaseInitStructure); |

| | |
|---|---|
| 注意事项 | 无 |

（3）启用或禁用定时器中断函数

表 4.8.4 启用或禁用定时器中断函数

| 接口函数：TIM_ITConfig | |
|---|---|
| 函数原型 | void TIM_ITConfig(TIM_TypeDef *TIMx, uint16_t TIM_IT, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 TIMx 中断。 |
| 入口参数 | TIMx：x 可以为 1~14，表示不同的定时器外设。 |
| | TIM_IT：指定要启用或禁用的 TIM 中断源。<br>其参数可以为 TIM_IT_Update（更新中断源）、TIM_IT_CC1(2,3,4)（定时器捕获/比较中断源）、TIM_IT_COM、TIM_IT_Trigger（定时器触发中断源）、TIM_IT_Break（定时器中断中断源）。 |
| | NewState：定时器中断状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | TIM_ITConfig(TIM2, TIM_IT_Update, ENABLE); |
| 注意事项 | （1）对于 TIM6、TIM7 只有 TIM_IT_Update 中断源可用；<br>（2）对于 TIM9、TIM12 只有 TIM_IT_Update、TIM_IT_CC1、TIM_IT_CC2 和 TIM_IT_Trigger 中断源可用；<br>（3）对于 TIM10、TIM11、TIM13 和 TIM14 只有 TIM_IT_Update 和 TIM_IT_CC1 中断源可用；<br>（4）TIM_IT_COM 和 TIM_IT_Break 中断源只有 TIM1 和 TIM8 能被使用。 |

（4）启用或禁用定时器外设函数

表 4.8.5 启用或禁用定时器外设函数

| 接口函数：TIM_Cmd | |
|---|---|
| 函数原型 | void TIM_Cmd(TIM_TypeDef *TIMx, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 TIMx 外设。 |
| 入口参数 | TIMx：x 可以为 1~14，表示不同的定时器外设。 |
| | NewState：定时器外设状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | TIM_Cmd(TIM2, ENABLE); |
| 注意事项 | 无 |

（5）获取定时器指定中断的状态函数

表 4.8.6 获取定时器指定中断的状态函数

| 接口函数：TIM_GetITStatus | |
|---|---|
| 函数原型 | ITStatus TIM_GetITStatus(TIM_TypeDef *TIMx, uint16_t TIM_IT) |
| 功能描述 | 检查 TIMx 上的指定的中断是否发生。 |
| 入口参数 | TIMx：x 可以为 1~14，表示不同的定时器外设。 |
| | TIM_IT：指定的 TIM 中断源。其参数可以为 TIM_IT_Update（更新中断源）、TIM_IT_CC1(2,3,4)（定时器捕获/比较中断源）、TIM_IT_COM、TIM_IT_Trigger（定时器触发中断源）、TIM_IT_Break（定时器中断中断源）。 |
| 返回值 | 定时器中断源（TIM_IT）的状态，返回值可以为 SET 或 RESET。 |
| 使用示例 | ITStatus status = TIM_GetITStatus(TIM2, TIM_IT_Update); |
| 注意事项 | 无 |

（6）清除定时器的中断待处理位函数

表 4.8.7 清除定时器的中断待处理位函数

| 接口函数：TIM_ClearITPendingBit | |
|---|---|
| 函数原型 | void TIM_ClearITPendingBit(TIM_TypeDef *TIMx, uint16_t TIM_IT) |
| 功能描述 | 清除 TIMx 外设指定中断的待处理位。 |
| 入口参数 | TIMx：x 可以为 1~14，表示不同的定时器外设。 |
| | TIM_IT：指定的 TIM 中断源。其参数可以为 TIM_IT_Update（更新中断源）、TIM_IT_CC1(2,3,4)（定时器捕获/比较中断源）、TIM_IT_COM、TIM_IT_Trigger（定时器触发中断源）、TIM_IT_Break（定时器中断中断源）。 |
| 返回值 | 无 |
| 使用示例 | TIM_ClearITPendingBit(TIM2, TIM_IT_Update); |
| 注意事项 | 无 |

## 4.8.2 硬件设计

定时器相关内容不涉及外部硬件电路。

## 4.8.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将定时器部分的驱动代码独立开来，下面以定时器 3 为例进行定时器初始化讲解，源文件（timer.c）和头文件（timer.h）存放在实验案例根目录中的 HARDWARE/TIMER 下。

**1. 软件设计流程**

如图 4.8.6，展示了本实验的软件设计流程图。

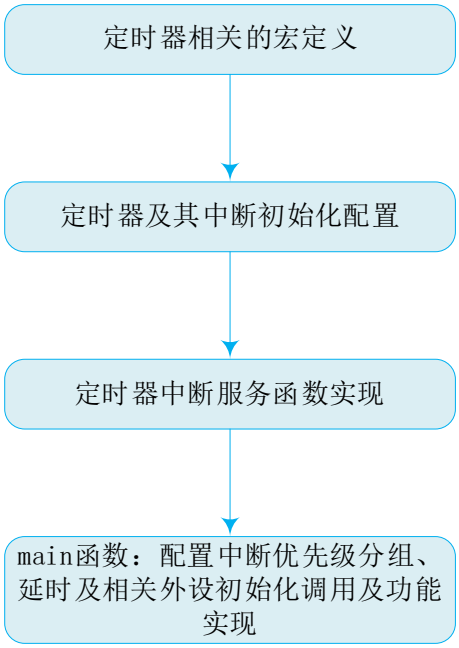



图 4.8.6 软件设计流程图

**2. 软件代码讲解**

（1）对定时器及其中断进行初始化配置。

代码段 4.8.2 定时器初始化函数（timer.c）

```
void Timer3_Init(uint16_t arr, uint16_t psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;                    //定义定时器时基单元初始化结构体变量
    NVIC_InitTypeDef NVIC_InitStructure;                                  //定义 NVIC 初始化结构体变量
```

```
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);                //使能定时器时钟

    TIM_TimeBaseInitStructure.TIM_Period = arr;                         //设置自动重装载值
    TIM_TimeBaseInitStructure.TIM_Prescaler = psc;                      //设置预分频器值
    TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up;     //设置计数器模式为向
上计数模式
    TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;         //设置时钟分频

    TIM_TimeBaseInit(TIM3, &TIM_TimeBaseInitStructure);               //初始化定时器

    TIM_ITConfig(TIM3, TIM_IT_Update, ENABLE);                        //启用定时器的更新中断
    TIM_Cmd(TIM3, ENABLE);                                            //使能定时器

    NVIC_InitStructure.NVIC_IRQChannel = TIM3_IRQn;                   //选用定时器中断通道
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;          //抢占优先级 0
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 4;                //子优先级 4
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                   //启动中断通道
    NVIC_Init(&NVIC_InitStructure);
  }
```

（2）定时器中断服务函数实现，每产生一次中断，LED1 反转状态。

代码段 4.8.3 定时器中断服务函数（timer.c）

```
/*
功   能：  定时器中断服务函数
参   数：  无
返回值：  无
*/
void TIMER3_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM3, TIM_IT_Update) == SET)                  //溢出中断
    {
        LED3_TOGGLE();
    }

    TIM_ClearITPendingBit(TIM3, TIM_IT_Update);                       //清除中断标志位
}
```

（3）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了

中断优先级分组、延时和相关外设初始化的流程。

代码段 4.8.4 主函数（main.c）

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1);   //配置优先级组别 1

    Delay_Init();                    //延时初始化
    Led_Init();                      //LED 初始化
    Timer1_Init(10000-1, 16800-1);   //定时器 1，1s
    Timer2_Init(10000-1, 8400-1);    //定时器 2，1s
    Timer3_Init(10000-1, 840-1);     //定时器 3，100ms
    Timer9_Init(5000-1, 8400-1);     //定时器 9，500ms

    while(1);
}
```

## 4. 8. 4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器。

| PC | STM32F4 核心板 | 12V 电源适配器 | ST Link 下载器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

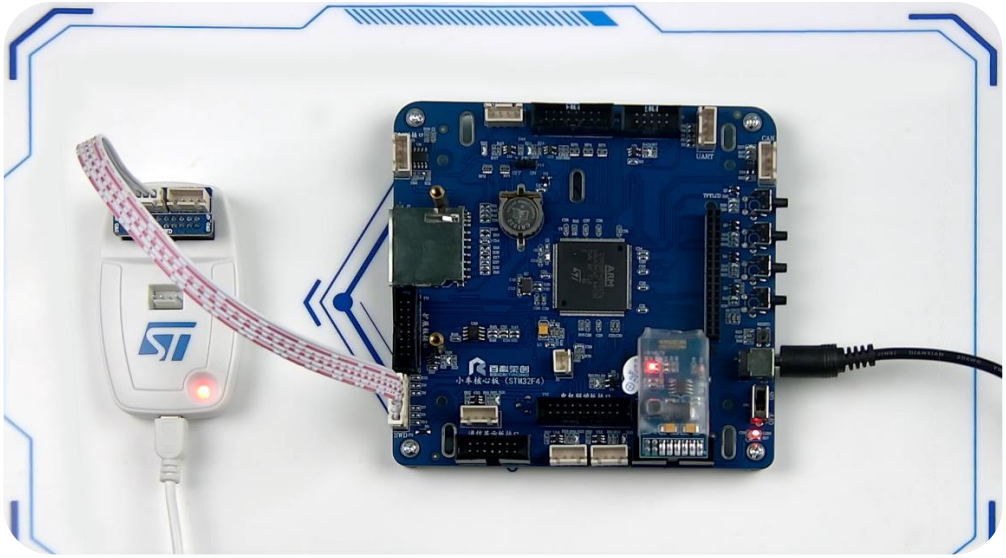


图 4.8.7 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板。

## 4.8.5 结果展示

将程序下载至核心板后，LED1 每隔 1000ms 点亮熄灭，LED2 每隔 1000ms 点亮熄灭，LED3 每隔 100ms 点亮熄灭，LED4 每隔 500ms 点亮熄灭。

# 4.9 呼吸灯实验

本实验旨在通过学习使用定时器的 PWM（脉冲宽度调制）输出功能原理，以控制 LED 实现呼吸灯效果（所谓的呼吸灯效果，就是让 LED 的亮度逐渐增强，然后再逐渐减弱，如同人类的呼吸一样）。

## 4.9.1 PWM 介绍

**1. PWM 概念**

PWM(Pulse Width Modulation，脉冲宽度调制)，是利用微处理器的数字输出来对模拟电路进行控制的一种非常有效的技术，广泛应用在从测量、通信到功率控制与变换的许多领域中。

**2. STM32 的 PWM**

在 STM32F4xx 中，除基本定时器（TIM6、TIM7）外，其它定时器都支持 PWM 输出，其中高级定时器（TIM1、TIM8）可以同时产生多达 7 路的 PWM 输出，通用定时器（TIM2 到 TIM5）也能同时产生 4 路的 PWM 输出。

**3. PWM 频率**

是指在 1 秒钟内，信号从高电平到低电平再回到高电平的次数，单位为 Hz。

**4. PWM 周期**

T = 1（s） / f（Hz），T 是周期，f 是频率。如果频率为 100Hz，也就是说一个周期是 10ms，那么一秒钟就有 100 个 PWM 周期。

在 STM32 中，定时器计数周期就是 PWM 输出波形的周期。设 TIM3 输入时钟信号频率为 84Mhz，分频系数为“/84”,计数器值为 1000，那么 PWM 波形周期为:

$$1s * 1000 * 84/84Mhz = 0.001s = 1ms$$

当波形的占空比为 50%时，那么一个周期内 PWM 输出高低电平的时长各是 500us。

**5. 占空比（Duty Cycle）**

占空比就是在一个脉冲周期内，高电平的时间占整个脉冲周期时间的比例，单位是%(0%-100%)。例如：一个脉冲周期为 10μs，其脉冲宽度为 4μs，则其占空比则为 4/10=0.4（40%）。

PWM 的调节作用来源于对脉冲宽度的控制，脉冲宽度变宽，输出的能量就会提高，通过阻容变换电路所得到的平均电压值也会上升；反之脉冲宽度变窄，输出的电压信

号的电压平均值就会降低，通过阻容变换电路所得到的平均电压值也会下降。

因此，在一定频率下，输出控制不同的占空比在理想状态下可以视为输出不同的模拟电压。

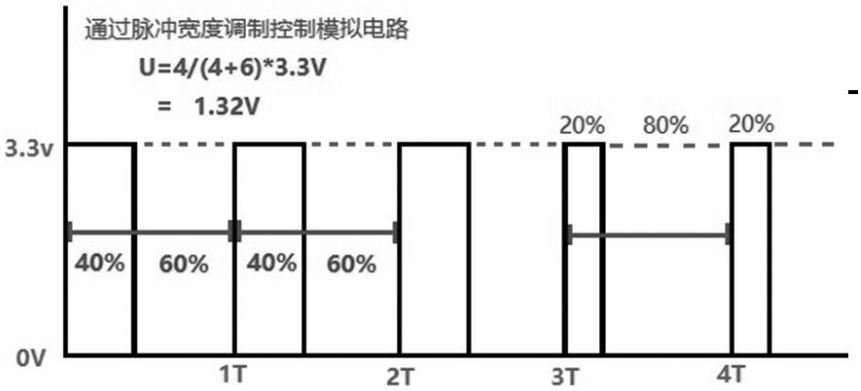



图 4.9.1 PWM 调试占空比

**6. PWM 占空比控制**

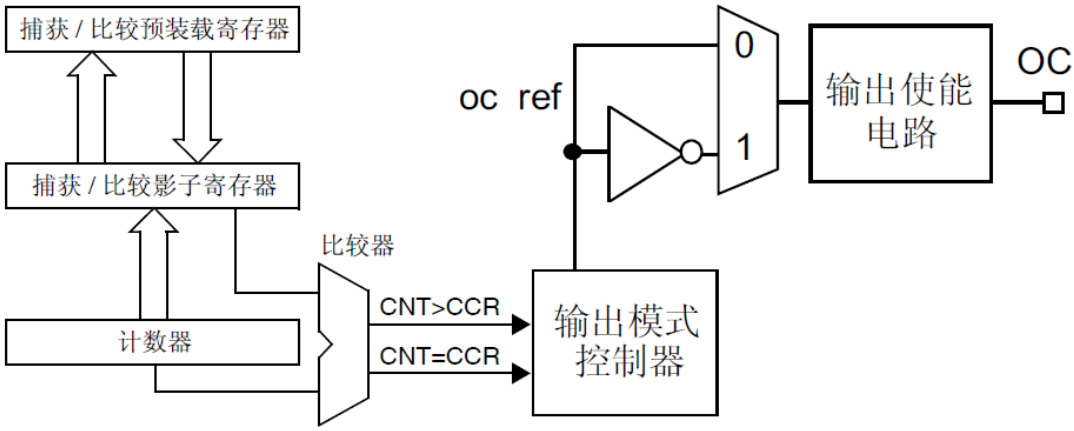



图 4.9.2 PWM 输出通道

已知每个定时器的时基单元都有一个计数器。如图 4.9.2 所示，当比较器判断到“CNT>CCR”或“CNT=CCR”时比较器会输入有效信号驱动输出模式控制器。其中 CNT 是指定时器的计数器值，开启后会定时自增或自减。CCR 是比较寄存器中的比较值，用于与计数值进行比较。所以可随时根据需求控制 CCR 值，从而控制输出 PWM 占空比。

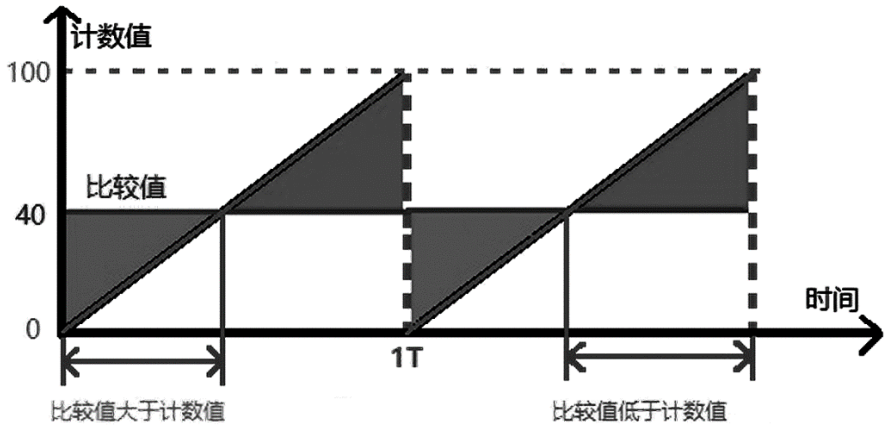



图 4.9.3 CNT 与 CCR

以图 4.9.3 作为例子，设置定时器自动重装载值为 99、向上计数模式、比较值设为 40。当计数器计数为 0~39 时，CCR 值（比较值）恒大于 CNT 值（计数值），不满足比较器有效输出条件，输出模式控制器输出低电平。反之满足比较器有效输出条件，则输出模式控制器输出高电平。同一输出状态在维持多个周期后最终形成 PWM。

**7. PWM 输出模式**

（1）向上计数模式

在向上计数模式中，计数器从 0 开始递增，一直增加到一个预设的上限值。一旦计数器达到上限值，它将被清零并重新开始递增。这种模式下生成的 PWM 信号的占空比由比较值决定。当计数器的值小于比较值时，输出为高电平，否则为低电平。因此，高电平的持续时间取决于比较值。

（2）向下计数模式

在向下计数模式中，计数器从一个预设的上限值开始递减，递减到 0 后重新从上限值开始递减。一旦计数器达到 0，它将被重新加载为上限值并重新开始递减。与向上计数模式类似，向下计数模式下的 PWM 信号的占空比也由比较值决定。当计数器的值大于比较值时，输出为高电平，否则为低电平。因此，高电平的持续时间取决于比较值。

（3）中心对齐模式

中心对齐模式是一种特殊的计数模式，它将 PWM 信号的周期分为上升和下降两个阶段。计数器从 0 开始递增，一直增加到一个预设的上限值。然后，计数器会从上限值减少到 0。这就导致输出的高电平和低电平时间会相等，从而以中心位置为基准对齐。在中心对齐模式下，高电平的持续时间取决于计数器上限的一半。

**8. 定时器通道功能映射**

STM32F4xx 系列的定时器通道功能映射表如所示。完整映射表参考《STM32F4xx 数据手册》Table 9. Alternate function mapping。

表 4.9.1 定时器通道功能映射表

| Port | AF1 | AF2 | AF3 | AF9 |
|---|---|---|---|---|
| | TIM1/2 | TIM3/4/5 | TIM8/9/10/11 | TIM12/13/14 |
| PA0 | TIM2_CH1_ETR | TIM5_CH1 | TIM8_ETR | |
| PA1 | TIM2_CH2 | TIM5_CH2 | | |
| PA2 | TIM2_CH3 | TIM5_CH3 | TIM9_CH1 | |
| PA3 | TIM2_CH4 | TIM5_CH4 | TIM9_CH2 | |
| PA5 | TIM2_CH1_ETR | | TIM8_CH1N | |
| PA6 | TIM1_BKIN | TIM3_CH1 | TIM8_BKIN | TIM13_CH1 |
| PA7 | TIM1_CH1N | TIM3_CH2 | TIM8_CH1N | TIM14_CH1 |
| PA8 | TIM1_CH1 | | | |
| PA9 | TIM1_CH2 | | | |
| PA10 | TIM1_CH3 | | | |
| PA11 | TIM1_CH4 | | | |
| PA12 | TIM1_ETR | | | |
| PA15 | TIM2_CH1/TIM2_ETR | | | |
| PB0 | TIM1_CH2N | TIM3_CH3 | TIM8_CH2N | |
| PB1 | TIM1_CH3N | TIM3_CH4 | TIM8_CH3N | |
| PB3 | TIM2_CH2 | | | |
| PB4 | | TIM3_CH1 | | |
| PB5 | | TIM3_CH2 | | |
| PB6 | | TIM4_CH1 | | |
| PB7 | | TIM4_CH2 | | |
| PB8 | | TIM4_CH3 | TIM10_CH1 | |
| PB9 | | TIM4_CH4 | TIM11_CH1 | |
| PB10 | TIM2_CH3 | | | |
| PB11 | TIM2_CH4 | | | |
| PB12 | TIM1_BKIN | | | |
| PB13 | TIM1_CH1N | | | |
| PB14 | TIM1_CH2N | | TIM8_CH2N | TIM12_CH1 |
| PB15 | TIM1_CH3N | | TIM8_CH3N | TIM12_CH2 |
| PC6 | | TIM3_CH1 | TIM8_CH1 | |
| PC7 | | TIM3_CH2 | TIM8_CH2 | |
| PC8 | | TIM3_CH3 | TIM8_CH3 | |

| | | | | |
|---|---|---|---|---|
| PC9 | | TIM3_CH4 | TIM8_CH4 | |
| PD2 | | TIM3_ETR | | |
| PD12 | | TIM4_CH1 | | |
| PD13 | | TIM4_CH2 | | |
| PD14 | | TIM4_CH3 | | |
| PD15 | | TIM4_CH4 | | |
| PE0 | | TIM4_ETR | | |
| PE5 | | | TIM9_CH1 | |
| PE6 | | | TIM9_CH2 | |
| PE7 | TIM1_ETR | | | |
| PE8 | TIM1_CH1N | | | |
| PE9 | TIM1_CH1 | | | |
| PE10 | TIM1_CH2N | | | |
| PE11 | TIM1_CH2 | | | |
| PE12 | TIM1_CH3N | | | |
| PE13 | TIM1_CH3 | | | |
| PE14 | TIM1_CH4 | | | |
| PE15 | TIM1_BKIN | | | |
| PF6 | | | TIM10_CH1 | |
| PF7 | | | TIM11_CH1 | |
| PF8 | | | | TIM13_CH1 |
| PF9 | | | | TIM14_CH1 |
| PH6 | | | | TIM12_CH1 |
| PH9 | | | | TIM12_CH2 |
| PH10 | | TIM5_CH1 | | |
| PH11 | | TIM5_CH2 | | |
| PH12 | | TIM5_CH3 | | |
| PH13 | | | TIM8_CH1N | |
| PH14 | | | TIM8_CH2N | |
| PH15 | | | TIM8_CH3N | |
| PI0 | | TIM5_CH4 | | |
| PI2 | | | TIM8_CH4 | |
| PI3 | | | TIM8_ETR | |
| PI4 | | | TIM8_BKIN | |
| PI5 | | | TIM8_CH1 | |
| PI6 | | | TIM8_CH2 | |
| PI7 | | | TIM8_CH3 | |

## 9. PWM 相关库函数讲解

（1）定时器输出比较初始化结构体定义

代码段 4.9.1 定时器输出比较初始化结构体定义

```
typedef struct
{
  uint16_t  TIM_OCMode;
  uint16_t  TIM_OutputState;
  uint16_t  TIM_OutputNState;
  uint32_t  TIM_Pulse;
  uint16_t  TIM_OCPolarity;
  uint16_t  TIM_OCNPolarity;
  uint16_t  TIM_OCIdleState;
  uint16_t  TIM_OCNIdleState;
} TIM_OCInitTypeDef;
```

关于定时器输出比较初始化结构体各个成员的详细描述，如表 4.9.2 所示。

表 4.9.2 定时器输出比较初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| TIM_OCMode | TIM_OCMode_Active<br>TIM_OCMode_Inactive<br>TIM_OCMode_PWM1<br>TIM_OCMode_PWM2<br>TIM_OCMode_Timing<br>TIM_OCMode_Toggle | 指定 TIM 的模式，该参数可以是输出比较模式和 PWM 模式 |
| TIM_OutputState | TIM_OutputState_Disable<br>TIM_OutputState_Enable | 指定 TIM 的输出比较状态 |
| TIM_OutputNState① | TIM_OutputNState_Disable<br>TIM_OutputNState_Enable | 指定 TIM 的互补输出比较状态 |
| TIM_Pulse | 0x0000~0xFFFF | 指定载入捕获比较寄存器的脉冲宽度 |
| TIM_OCPolarity | TIM_OCPolarity_High<br>TIM_OCPolarity_Low | 指定输出极性 |
| TIM_OCNPolarity① | TIM_OCNPolarity_High<br>TIM_OCNPolarity_Low | 指定互补输出极性 |
| TIM_OCIdleState① | TIM_OCIdleState_Reset<br>TIM_OCIdleState_Set | 指定空闲状态下 TIM 输出比较引脚的状态 |
| TIM_OCNIdleState① | TIM_OCNIdleState_Reset<br>TIM_OCNIdleState_Set | 指定空闲状态下 TIM 的互补输出比较引脚的状态 |

| ①该成员参数仅对 TIM1 和 TIM8 有效。 |
|---|

（2）根据输出比较结构体指定的参数初始化定时器通道 1 函数

表 4.9.3 根据输出比较结构体指定的参数初始化定时器通道 1 函数

| 接口函数：TIM_OC1Init | |
|---|---|
| 函数原型 | void TIM_OC1Init(TIM_TypeDef *TIMx, TIM_OCInitTypeDef *TIM_OCInitStruct) |
| 功能描述 | 根据 TIM_OCInitStruct 中的指定参数初始化 TIMx 通道 1。 |
| 入口参数 | TIMx：x 可以为 1~5、8~14，表示不同的定时器外设。 |
| | TIM_OCInitStruct：指向 TIM_OCInitTypeDef 结构体的指针，该结构体包含指定 TIMx 外设的配置信息。 |
| 返回值 | 无 |
| 使用示例 | TIM_OC1Init(TIM2, &TIM_OCInitStructure); |
| 注意事项 | 无 |

（3）启用或禁止 TIMx_CCR1 寄存器的预装载功能函数

表 4.9.4 启用或禁止 TIMx_CCR1 寄存器的预装载功能函数

| 接口函数：TIM_OC1PreloadConfig | |
|---|---|
| 函数原型 | void TIM_OC1PreloadConfig(TIM_TypeDef *TIMx, uint16_t TIM_OCPreload) |
| 功能描述 | 启用或禁用 TIMx（除 TIM6 和 TIM7）外设 CCR1 寄存器上的预装载功能。 |
| 入口参数 | TIMx：x 可以为 1~5、8~14，表示不同的定时器外设。 |
| | TIM_OCPreload：TIMx 外设预加载寄存器的状态，其参数可以为 TIM_OCPreload_Enable 或 TIM_OCPreload_Disable。 |
| 返回值 | 无 |
| 使用示例 | TIM_OC1PreloadConfig(TIM2, TIM_OCPreload_Enable); |
| 注意事项 | 无 |

（4）启用或禁止 TIMx_ARR 寄存器的预装载功能函数

表 4.9.5 启用或禁止 TIMx_ARR 寄存器的预装载功能函数

| 接口函数：TIM_ARRPreloadConfig | |
|---|---|
| 函数原型 | void TIM_ARRPreloadConfig(TIM_TypeDef *TIMx, FunctionalState NewState) |
| 功能描述 | 启用或禁用 TIMx 外设 ARR 寄存器上的预装载功能。 |
| 入口参数 | TIMx：x 可以为 1~14，表示不同的定时器外设。 |
| | NewState：TIMx 外设预加载寄存器的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | TIM_ARRPreloadConfig(TIM2, ENABLE); |
| 注意事项 | 无 |

## 4.9.2 硬件设计

定时器 PWM 输出相关内容不涉及外部硬件电路。

## 4.9.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将定时器 PWM 部分的驱动代码独立开来，源文件（pwm.c）和头文件（pwm.h）存放在实验案例根目录中的 HARDWARE/PWM 下。

**1. 软件设计流程**

如图 4.9.4，展示了本实验的软件设计流程图。

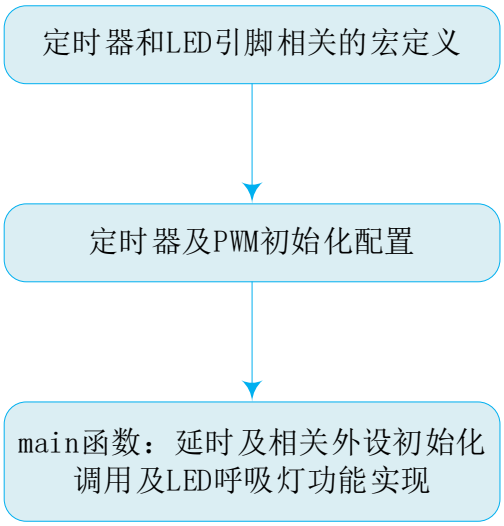



图 4.9.4 软件设计流程图

**2. 软件代码讲解**

（1）定义定时器与 LED 相关的宏。

代码段 4.9.2 定时器与 LED 宏定义（pwm.h）

```
/***************LED1 相关定义***************/
//LED1 对应的 GPIO 时钟挂载在 AHB1 上
#define LED1_CLOCK          RCC_AHB1Periph_GPIOF

//LED1 对应的 GPIO 时钟使能，其为宏函数定义
#define LED1_CLK_INIT       RCC_AHB1PeriphClockCmd

//LED 名   LED 端口引脚      对应定时器通道
//LED1    PF9              TIM14_CH1
#define LED1_PORT           GPIOF
#define LED1_PIN            GPIO_Pin_9

//LED1 复用引脚
#define LED1_PIN_SOURCE     GPIO_PinSource9

//GPIO 复用至定时器
#define GPIO_AF_TIMER14     GPIO_AF_TIM14

/***************定时器相关定义***************/
```

```
//定时器 14
#define TIMER14             TIM14

//定时器 14 时钟挂载在 APB1 上
#define TIMER14_CLOCK       RCC_APB1Periph_TIM14

//定时器时钟使能，其为宏函数定义
#define TIMER14_CLK_INIT    RCC_APB1PeriphClockCmd
```

---

（2）对定时器进行初始化。包括了定时器与 LED 灯的时钟使能、LED 灯引脚的复用、定时器配置、PWM 模式配置等初始化流程。

代码段 4.9.3 定时器初始化函数（pwm.c）

---

```
/*
功   能:  通用定时器初始化（PWM 配置）
参   数:  arr     自动重装值
          psc     时钟预分频数
返回值:  无
说   明:  定时器溢出时间计算方法: Tout=((arr+1)*(psc+1))/Ft us
          Ft: 定时器工作频率（单位: Mhz）
*/
void PWM_Init(uint16_t arr, uint16_t psc)
{
    TIM_TimeBaseInitTypeDef TIM_TimeBaseInitStructure;      //定义定时器时基单元初始化结构体变量
    TIM_OCInitTypeDef TIM_OCInitStructure;                  //定义定时器输出比较初始化结构体变量
    GPIO_InitTypeDef GPIO_InitStructure;                    //定义 LED1 的 GPIO 初始化结构体变量

    TIMER14_CLK_INIT(TIMER14_CLOCK, ENABLE);                //使能定时器时钟
    LED1_CLK_INIT(LED1_CLOCK, ENABLE);                      //使能 LED1 的 GPIO 时钟

    GPIO_PinAFConfig(LED1_PORT, LED1_PIN_SOURCE, GPIO_AF_TIMER14);     //将 LED1 的 GPIO 引脚复用

    GPIO_InitStructure.GPIO_Pin = LED1_PIN;                     //选择 LED1 的 GPIO 引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;                //复用功能
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;              //推挽输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_25MHz;           //25MHz
```

```
    GPIO_Init(LED1_PORT, &GPIO_InitStructure);                    //初始化 GPIO

    TIM_TimeBaseInitStructure.TIM_Period = arr;                   //设置自动重装载值
    TIM_TimeBaseInitStructure.TIM_Prescaler = psc;                //设置计数器模式为向上计数模
式
    TIM_TimeBaseInitStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式
    TIM_TimeBaseInitStructure.TIM_ClockDivision = TIM_CKD_DIV1;  //设置时钟分频

    TIM_TimeBaseInit(TIMER14, &TIM_TimeBaseInitStructure);        //初始化定时器

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;    //选择定时器模式:TIM 脉冲宽度调制模式 1
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //输出比较使能
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;     //输出极性:TIM 输出比较极性
高
    TIM_OC1Init(TIMER14, &TIM_OCInitStructure);            //根据指定的参数初始化外设 TIM_OC1

    TIM_OC1PreloadConfig(TIMER14, TIM_OCPreload_Enable);  //启用定时器外设 CCR1 上的预装载寄存
器

    TIM_ARRPreloadConfig(TIMER14, ENABLE);                 //启用定时器外设 ARR 上的预装载寄存器

    TIM_Cmd(TIMER14, ENABLE);                               //使能定时器
}
```

（3）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而 while 循环中则需实现 LED 呼吸灯功能。

代码段 4.9.4 主函数（main.c）

```
int main(void)
{
    uint8_t led1_pwm_val = 0;    //LED1 占空比的值
    uint8_t direction = 1;       //占空比值是增加还是减小

    Delay_Init();                //延时初始化
    Led_Init();                  //LED 初始化
    PWM_Init(100-1,84-1);        //定时器初始化，84M/84=1Mhz 的计数频率,重装载值 100，所以 PWM 频
率为 1M/100=10Khz

    while(1)
```

```
    {
        if(direction) led1_pwm_val++;           //direction=1，led1_pwm_val 递增
        else          led1_pwm_val--;           //direction=0，led1_pwm_val 递减

        if(led1_pwm_val==100)   direction = 0;  //led1_pwm_val 递增至 100 后，方向改为递减
        if(led1_pwm_val==0)     direction = 1;  //led1_pwm_val 递减至 0 后，方向改为递增

        TIM_SetCompare1(TIMER14, led1_pwm_val); //修改占空比，占空比=led1_pwm_val/100

        Delay_ms(10);
    }
}
```

---

## 4.9.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器。

| PC | STM32F4 核心板 | 12V 电源适配器 | ST Link 下载器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

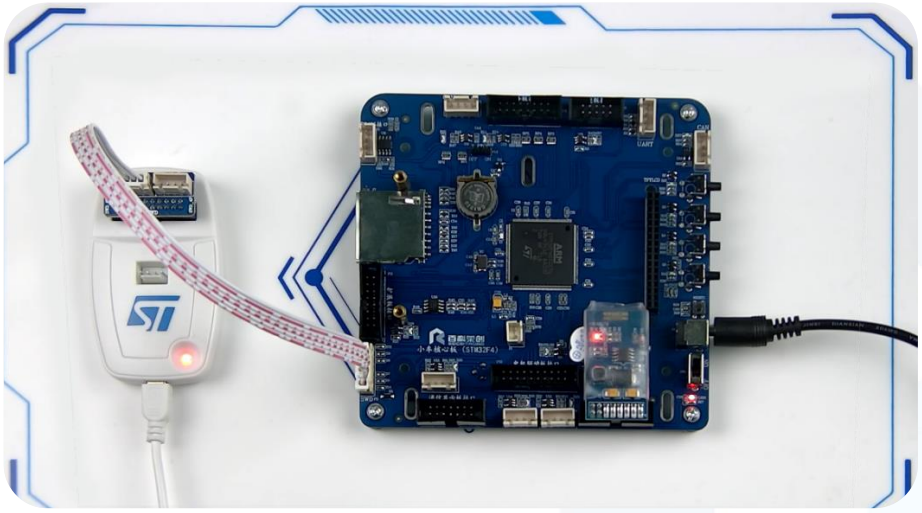


图 4.9.5 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，观察 LED1 是否出现呼吸灯效果。

## 4.9.5 结果展示

将程序下载至核心板后，观察到 LED1 灯出现呼吸灯效果。

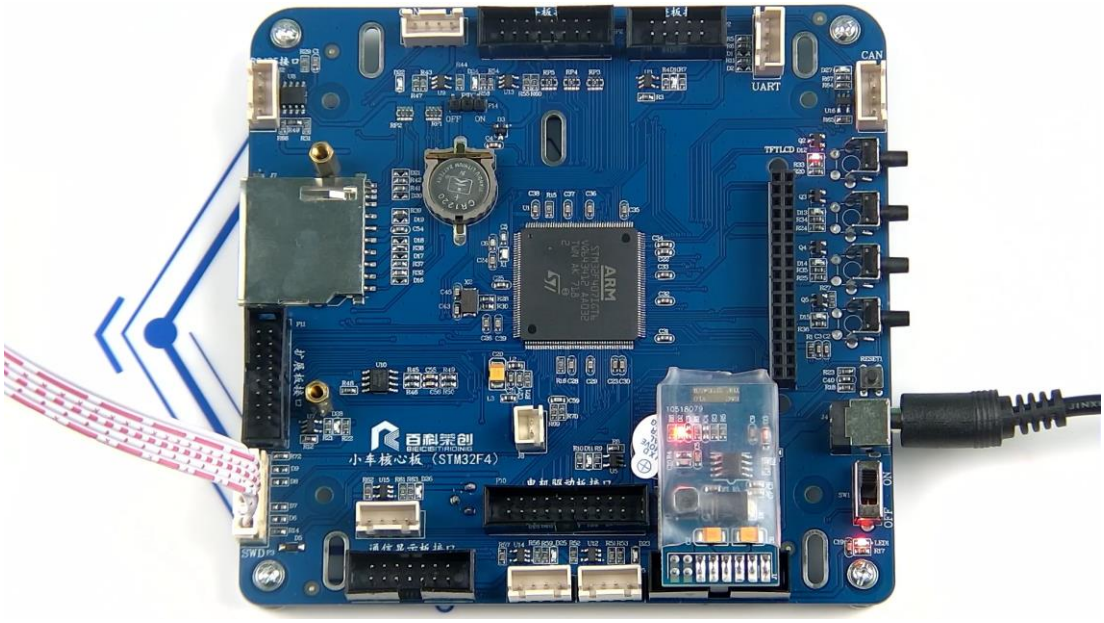



图 4.9.6 实验效果图展示

# 4.10 电池电量测量实验

本实验使用 STM32F4xx 系列微控制器进行模数转换实验。

通过配置和利用 STM32 内置的 ADC 模块，实验将模拟输入信号转换为数字信号，并测量特定引脚的电压值。本实验旨在帮助读者理解和学习 STM32 微控制器的 ADC 功能，以及如何应用该功能进行模数转换操作。通过本实验，读者可以深入了解模数转换原理，并掌握在嵌入式系统开发中应用 ADC 的实际技能。

## 4.10.1 ADC 介绍

**1. ADC 概念**

模数转换器（Analog-to-Digital Converter, ADC）是将连续变化的模拟信号转换为离散的数字信号的器件。在自然界中的信号几乎都是模拟信号，比如光亮、温度、压力、声音，而为了方便存储、处理，需要转换成更容易储存、处理和发射的数字信号，模数转换器就可以实现这个功能。

按原理可分为：并行比较型 A/D 转换器(FLASH ADC)、逐次比较型 A/D 转换器(SAR ADC)和双积分式 A/D 转换器(Double Integral ADC)。

典型的模拟数字转换器是将模拟信号转换为表示一定比例电压值的数字信号。

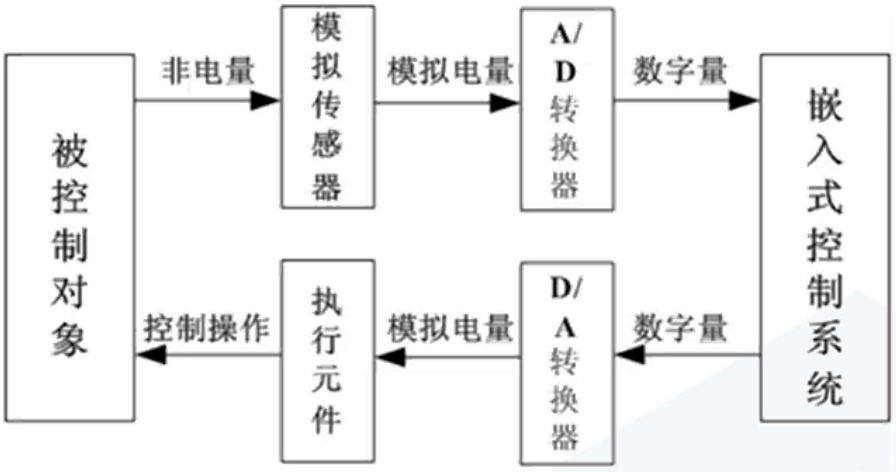



图 4.10.1 数据采集系统

**2. ADC 转换原理**

A/D 转换过程通常为 4 步：采样、保持、量化和编码。如图 4.10.2 所示，展示了 ADC 的采样过程。

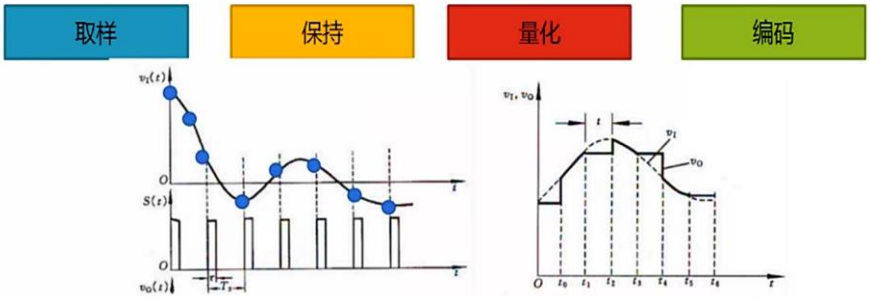



图 4.10.2 ADC 采样过程

其中采样是对模拟信号周期性地抽取样值，使模拟信号转化为时间上离散的脉冲信号。采样频率（$f_S$）越高，采样越密集，采样值越多，也就越接近模拟信号。为确保采样后的信号能够还原模拟信号，根据香农-奈奎斯特(Shannon & Nyquist)采样定理，采样频率必须大于等于 2 倍输入模拟信号的最高截止频率（$f_{Imax}$）：

$$f_S \geq 2f_{Imax}$$

ADC 的主要有三个性能指标：分辨率、转换时间和转换精度。

分辨率：指 ADC 能分辨的最小电压，通常使用二进制有效位表示，反应了 ADC 对输入模拟量微小变化的分辨能力。当最大输入电压一定时，位数越多，量化单位越小，误差越小，分辨率越高。比如一个 12 位的 ADC，参考电压为 3.3V，则其能分辨的最小电压为：$3.3V/2^{12}=3.3V/4096=0.0008V=0.8mV$。

转换时间：其倒数为转换速率，指 ADC 从控制信号到来开始，到输出端得到稳定的数字信号所经历的时间。转换时间通常与 ADC 类型有关，双积分型 ADC 的转换时间一般为几十毫秒，属于低速 ADC；逐次逼近型 ADC 的转换时间一般为几十微妙，属于中速 ADC；并联比较型 ADC 的转换时间一般为几十纳秒，属于高速 ADC。

转换精度：指 ADC 输出的数字量所表示的模拟值与实际输入的模拟量之间的偏差，通常为 1 个或半个最小数字量的模拟变化量，表示为 1LSB 或 1/2LSB。

**3. STM32 的 ADC**

STM32F4xx 系列一般都有 3 个 ADC，这些 ADC 可以独立使用，也可以使用双重/三重模式（提高采样率）。STM32F4xx 的 ADC 是 12 位逐次逼近型的模拟数字转换器。它具有多达 19 个复用通道，可测量来自 16 个外部源、两个内部源和 VBAT 通道的信

号。这些通道的 A/D 转换可在单次、连续、扫描或不连续采样模式下进行。

ADC 的结果存储在一个左对齐或右对齐的 16 位数据寄存器中。

ADC 还具有模拟看门狗特性，允许应用检测输入电压是否超过了用户自定义的阈值上限或下限。

**4. ADC 参考电压引脚**

$V_{DDA}$ 和 $V_{SSA}$ 为专门为模拟电路设计的独立电源，以过滤和屏蔽来自 PCB 上的毛刺和干扰。ADC 为提高转换精确度，使用 $V_{DDA}$ 和 $V_{SSA}$ 供电。

$V_{REF+}$和 $V_{REF-}$为 ADC 基准参考电压（Voltage Reference），将输入的待测量电压与基准参考电压比较，从而得知测量电压的大小。基准参考电压的准确度，直接影响测量结果，在高要求场合，通常使用独立的电压基准芯片提供参考电压。一般情况下，将 $V_{REF+}$与 $V_{DDA}$ 连接，$V_{REF-}$与 $V_{SSA}$ 连接，得到 3.3V 的参考电压。此时，输入电压 $V_{IN}$ 与数字量 $V_{IN}$ 关系为：

$$模拟量V_{IN} = \frac{数字量V_{IN}}{2^{12}} * 3.3V$$

如果想测量更高电压，比如 5V 电压。可以使用输入范围较大的 ADC 外置 ADC 芯片，通过 I2C 等接口将测量结果返回给 MCU。也可以使用简单的电阻分压，分压到 3.3V 以下，将采集的电压再按比例换算，得到实际电压。

ADC 电源要求：全速运行时为 2.4V 到 3.6V，慢速运行时为 1.8 V。

ADC 输入范围：$V_{REF-} \leq V_{IN} \leq V_{REF+}$。

表 4.10.1 ADC 参考电压引脚

| 名称 | 信号类型 | 备注 |
|---|---|---|
| VREF+ | 正模拟参考电压输入 | ADC 高/正参考电压，1.8V≤VREF+≤VDDA |
| VDDA | 模拟电源输入 | 模拟电源电压等于 VDD，<br>全速运行时，2.4V≤ VDDA≤ VDD(3.6V)<br>低速运行时，1.8V≤ VDDA≤ VDD(3.6V) |
| VREF- | 负模拟参考电压输入 | ADC 低/负参考电压，VREF-=VSSA |
| VSSA | 模拟电源接地输入 | 模拟电源接地电压等于 VSS |
| ADCx_IN[15:0] | 模拟输入信号 | 16 个模拟输入通道 |

注意：$V_{REF-}$如果可用（取决于封装），则必须将其连接到 $V_{SSA}$。

**5. ADC 输入引脚**

ADCx_IN0~ADCx_IN15 为 ADC 的输入信号通道，每个输入通道连接一个 GPIO 引脚。需要 GPIO 设置为对应模拟输入的复用模式，如表 4.10.2 为 STM32F405xx/STM32F407xx 的 ADC 通道所对应引脚。数据参考《STM32F4xx 数据手册》Table 7. STM32F40xxx pin and ball definitions。

表 4.10.2 ADC 通道对应复用引脚

| 通道 | ADC1（复用引脚） | ADC2（复用引脚） | ADC3（复用引脚） |
|---|---|---|---|
| 通道 0 | PA0 | PA0 | PA0 |
| 通道 1 | PA1 | PA1 | PA1 |
| 通道 2 | PA2 | PA2 | PA2 |
| 通道 3 | PA3 | PA3 | PA3 |
| 通道 4 | PA4 | PA4 | PF6 |
| 通道 5 | PA5 | PA5 | PF7 |
| 通道 6 | PA6 | PA6 | PF8 |
| 通道 7 | PA7 | PA7 | PF9 |
| 通道 8 | PB0 | PB0 | PF10 |
| 通道 9 | PB1 | PB1 | PF3 |
| 通道 10 | PC0 | PC0 | PC0 |
| 通道 11 | PC1 | PC1 | PC1 |
| 通道 12 | PC2 | PC2 | PC2 |
| 通道 13 | PC3 | PC3 | PC3 |
| 通道 14 | PC4 | PC4 | PF4 |
| 通道 15 | PC5 | PC5 | PF5 |
| 通道 16 | 温度传感器 | — | — |
| 通道 17 | 内部参考电压 | — | — |

**6. ADC 时钟**

ADC 具有两个时钟方案：

（1）用于模拟电路的时钟：ADCCLK，所有 ADC 共用。此时钟来自于经可编程预分频器分频的 APB2 时钟，该预分频器允许 ADC 在 $f_{PCLK2}$/2、/4、/6 或/8 下工作。

AHB 域的最大频率为 168MHz。高速 APB2（PLCK2）域的最大允许频率为 84MHz。低速 APB1 域的最大允许频率为 42MHz。

对于 STM32F405xx/STM32F407xx，ADCCLK 在全速运行下，最高时钟频率为 36MHz，低速运行下最高时钟频率为 18MHz。因此在全速运行下通常设置分频值为 4，

则 ADCCLK 为 $f_{PCLK2}$/4=84/4=21MHz。

（2）用于数字接口的时钟（用于寄存器读/写访问）。此时钟等效于 APB2 时钟。可以通过 RCC APB2 外设时钟使能寄存器(RCC_APB2ENR)分别为每个 ADC 使能/禁止数字接口时钟。

**7. 转换模式**

（1）单次模式：ADC 只执行一次转换；

（2）连续模式：当前 ADC 转换结束后，立即进入下一个转换；

（3）扫描模式：用来扫描一组通道。通道可以来自规则通道组，也可来自注入通道组。开启扫描模式后，ADC 将自动扫描该组所有通道，如此时转换模式设置为单次转换，则扫描本组所有通道后，ADC 自动停止；若将转换模式设置为连续模式，则在扫描本组所有通道后，再从第一个通道开始扫描；

（4）不连续采样模式：用来间歇转换一组通道。通道可以来自规则通道组，也可来自注入通道组。假设该组包含 0、1、2、3、5、6、7、8，共 8 个通道，而设置转换通道数为 3，则第一次触发，转换 0、1、2 通道；第二次触发，转换 3、5、6 通道；第三次触发，转换 7、8 通道，并产生 EOC 中断。

**8. 数据对齐**

ADC_CR2 寄存器中的 ALIGN 位用于选择转换后存储的数据的对齐方式。可选择左对齐和右对齐两种方式。如图 4.10.3 所示，为 12 位数据的左右对齐方式。

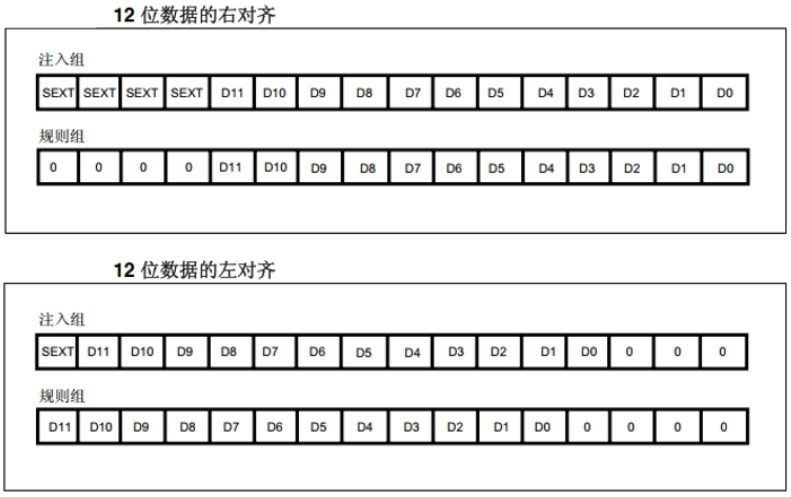



图 4.10.3 12 位数据的左右对齐方式

**9. ADC 采样时间**

A/D 转换在采样时信号需要保持一段时间，采样时间越长，转换结果越稳定，但转换速率也就越慢。STM32F4xx 的 ADC 每个通道的采样时间都可以进行设置，可设置为 3 个周期、15 个周期、28 个周期、56 个周期、84 个周期、112 个周期、144 个周期、480 个周期。

ADC 转换总时间可以表示为：

$$\text{ADC 转换总时间} = \text{采样时间} + 12 \text{ 个周期}$$

如采样时间为 480 个周期，ADCCLK 为 21MHz，则

$$\text{ADC 转换总时间} = \frac{480 + 12}{21\text{MHz}} = 23.43 \text{ 单位：us}$$

可通过降低 ADC 分辨率来执行快速转换，每种分辨率的最小转换时间如下：

（1）12 位：3（采样时间）+12 = 15（ADCCLK 周期）

（2）10 位：3（采样时间）+10 = 13（ADCCLK 周期）

（3）8 位：3（采样时间）+8 = 11（ADCCLK 周期）

（4）6 位：3（采样时间）+6 = 9（ADCCLK 周期）

**10. ADC 中断**

ADC 在每个通道转换结束后，可产生相应的中断请求。如所示，为 ADC 中断的事件对应的事件标志和使能控制位。

表 4.10.3 ADC 中断

| 中断事件 | 事件标志（ADC_SR） | 使能控制位（ADC_CR1） |
|---|---|---|
| 结束规则组的转换 | EOC | EOCIE |
| 结束注入组的转换 | JEOC | JEOCIE |
| 模拟看门狗状态位置 1 | AWD | AWDIE |
| 溢出(Overrun) | OVR | OVRIE |

## 11. ADC 相关函数讲解

（1）ADC 通用初始化结构体定义

代码段 4.10.1 ADC 通用初始化结构体定义

```
typedef struct
{
  uint32_t  ADC_Mode;
  uint32_t  ADC_Prescaler;
  uint32_t  ADC_DMAAccessMode;
  uint32_t  ADC_TwoSamplingDelay;
}ADC_CommonInitTypeDef;
```

关于 ADC 通用初始化结构体各个成员的详细描述，如表 4.10.4 所示。

表 4.10.4 ADC 通用初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| ADC_Mode | （1）ADC_Mode_Independent：独立模式<br>（2）ADC_DualMode_x（x 可以为 RegSimult_InjecSimult、RegSimult_AlterTrig、InjecSimult、RegSimult、Interl、AlterTrig）：双重模式<br>（2）ADC_TripleMode_x（x 可以为 RegSimult_InjecSimult、RegSimult_AlterTrig、InjecSimult、RegSimult、Interl、AlterTrig）：三重模式 | 将 ADC 配置为独立模式或多重模式 |
| ADC_Prescaler | ADC_Prescaler_Divx（x 可以为 2,4,6,8） | 选择 ADC 的时钟频率，所有 ADC 都使用相同的时钟 |
| ADC_DMAAccessMode | ADC_DMAAccessMode_x（x 可以为 1,2,3,Disabled） | 配置 ADC 多重模式下的 DMA 模式 |
| ADC_TwoSamplingDelay | ADC_TwoSamplingDelay_xCycles（x 可以为 5~20） | 配置两个采样阶段之间的延迟 |

（2）ADC 通用参数初始化函数

表 4.10.5 ADC 通用参数初始化函数

| 接口函数：ADC_CommonInit | |
|---|---|
| 函数原型 | void ADC_CommonInit(ADC_CommonInitTypeDef *ADC_CommonInitStruct) |
| 功能描述 | 根据 ADC_CommonInitStruct 中的指定参数初始化 ADC 外设。 |
| 入口参数 | ADC_CommonInitStruct：指向 ADC_CommonInitTypeDef 结构体的指针，该结构体包含所有 ADC 外设的通用配置信息。 |
| 返回值 | 无 |
| 使用示例 | ADC_CommonInit(&ADC_CommonInitStructure); |
| 注意事项 | 无 |

（3）ADC 初始化结构体定义

代码段 4.10.2 ADC 初始化结构体定义

```
typedef struct
{
  uint32_t          ADC_Resolution;
  FunctionalState   ADC_ScanConvMode;
  FunctionalState   ADC_ContinuousConvMode;
  uint32_t          ADC_ExternalTrigConvEdge;
  uint32_t          ADC_ExternalTrigConv;
  uint32_t          ADC_DataAlign;
  uint8_t           ADC_NbrOfConversion;
}ADC_InitTypeDef;
```

关于 ADC 初始化结构体各个成员的详细描述，如表 4.10.6 所示。

表 4.10.6 ADC 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| ADC_Resolution | ADC_Resolution_xb（x 可以为 6，8,10,12） | 配置 ADC 分辨率 |
| ADC_ScanConv Mode | ENABLE：扫描模式<br>DISABLE：单通道模式 | 指定转换是在扫描（多通道）还是单通道（一个通道）模式下进行 |
| ADC_Continuous ConvMode | ENABLE：连续模式<br>DISABLE：单次模式 | 指定转换是以连续模式还是单次模式执行 |
| ADC_ExternalTrig ConvEdge | ADC_ExternalTrigConvEdge_x<br>x 为 Falling 代表下降沿触发；<br>x 为 None 代表禁止外部触发；<br>x 为 Rising 代表上升沿触发；<br>x 为 RisingFalling 代表上升沿和下降沿触发 | 选择外部触发边沿并启用规则通道组的触发 |
| ADC_ExternalTrig Conv | ADC_ExternalTrigConv_x（x 可以为 Ext_IT11、T1_CC1、T1_CC2、T1_CC3、T2_CC2、T2_CC3、T2_CC4、T2_TRGO、T3_CC1、T3_TRGO、T4_CC4、T5_CC、T5_CC2、T5_CC3、T8_CC1、T8_TRGO） | 选择用于触发规则通道组转换开始的外部事件 |
| ADC_DataAlign | ADC_DataAlign_Left：左对齐<br>ADC_DataAlign_Right：右对齐 | 指定 ADC 数据对齐方式（左对齐或右对齐） |
| ADC_NbrOf Conversion | 1~16 | 规则通道组序列器中进行 ADC 转换的次数 |

（4）ADC 初始化函数

表 4.10.7 ADC 初始化函数

| 接口函数：ADC_Init | |
|---|---|
| 函数原型 | void ADC_Init(ADC_TypeDef *ADCx, ADC_InitTypeDef *ADC_InitStruct) |
| 功能描述 | 根据 ADC_InitStruct 中的指定参数初始化 ADCx 外设。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| | ADC_InitStruct：指向 ADC_InitTypeDef 结构体的指针，该结构体包含指定 ADC 外设的配置信息。 |
| 返回值 | 无 |
| 使用示例 | ADC_Init(ADC1, &ADC_InitStructure); |
| 注意事项 | 该功能用于配置 ADC 的全局特性（分辨率和数据对齐），但其余配置参数则针对规则通道组（扫描模式激活、连续模式激活、外部触发源和触发边沿、规则通道组序列器中的转换次数）。 |

（5）启用或禁用 ADC 外设函数

表 4.10.8 启用或禁用 ADC 外设函数

| 接口函数：ADC_Cmd | |
|---|---|
| 函数原型 | void ADC_Cmd(ADC_TypeDef *ADCx, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 ADCx 外设。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| | NewState：指定 ADCx 外设的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | ADC_Cmd(ADC1, ENABLE); |
| 注意事项 | 无 |

（6）配置所选 ADC 规则通道的等级和采样时间函数

表 4.10.9 配置所选 ADC 规则通道的等级和采样时间函数

| 接口函数：ADC_RegularChannelConfig | |
|---|---|
| 函数原型 | void ADC_RegularChannelConfig(ADC_TypeDef *ADCx, uint8_t ADC_Channel, uint8_t Rank, uint8_t ADC_SampleTime) |
| 功能描述 | 为选定的 ADC 规则通道配置其在序列器中的相应等级和其采样时间。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| | ADC_Channel：配置所选 ADC 的通道，其参数可以为 ADC_Channel_x（x 可以为 0~18）。 |
| | Rank：在规则通道组中的等级，其参数必须在 1 到 16 之间。 |
| | ADC_SampleTime：为所选通道设置采样时间，其参数可以为 ADC_SampleTime_xCycles（x 可以为 3，15，28，56，84，112，144，480）。 |
| 返回值 | 无 |
| 使用示例 | ADC_RegularChannelConfig(ADC1 ，ADC_Channel_1 , 1 , ADC_SampleTime_480Cycles); |
| 注意事项 | 无 |

（7）所选 ADC 软件开始规则通道的转换函数

表 4.10.10 所选 ADC 软件开始规则通道的转换函数

| 接口函数：ADC_SoftwareStartConv | |
|---|---|
| 函数原型 | void ADC_SoftwareStartConv(ADC_TypeDef *ADCx) |
| 功能描述 | 所选 ADC 软件开始规则通道的转换。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| 返回值 | 无 |
| 使用示例 | ADC_SoftwareStartConv(ADC1); |
| 注意事项 | 无 |

（8）检查 ADC 标志是否设置函数

表 4.10.11 检查 ADC 标志是否设置函数

| 接口函数：ADC_GetFlagStatus | |
|---|---|
| 函数原型 | FlagStatus ADC_GetFlagStatus(ADC_TypeDef *ADCx, uint8_t ADC_FLAG) |
| 功能描述 | 检查指定 ADC 的标志是否设置。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| | ADC_FLAG：指定要检查的标志，其参数可以为：ADC_FLAG_AWD（模拟看门狗标志）、ADC_FLAG_EOC（转换结束标志）、ADC_FLAG_JEOC（注入组转换结束标记）、ADC_FLAG_JSTRT（注入组转换开始标记）、ADC_FLAG_STRT（规则组转换开始标志）、ADC_FLAG_OVR（溢出标志） |
| 返回值 | 指定 ADC 标志（ADC_FLAG）的状态，其值可以为 SET 或 RESET。 |
| 使用示例 | FlagStatus status = ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC); |
| 注意事项 | 无 |

（9）获取规则通道最后一次 ADCx 转换的结果函数

表 4.10.12 获取规则通道最后一次 ADCx 转换的结果函数

| 接口函数：ADC_GetConversionValue | |
|---|---|
| 函数原型 | uint16_t ADC_GetConversionValue(ADC_TypeDef *ADCx) |
| 功能描述 | 返回规则通道最后一次 ADCx 转换的结果数据。 |
| 入口参数 | ADCx：ADC 外设，x 可以为 1，2，3。 |
| 返回值 | 转换值数据。 |
| 使用示例 | uint16_t adcx = ADC_GetConversionValue(ADC1); |
| 注意事项 | 无 |

## 4.10.2 硬件设计

ADC 相关内容不涉及外部硬件电路。

## 4.10.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 ADC1 部分的驱动代码独立开来，源文件（adc1.c）和头文件（adc1.h）存放在实验案例根目录中的 HARDWARE/ADC1 下。

**1. 软件设计流程**

如图 4.10.4，展示了本实验的软件设计流程图。

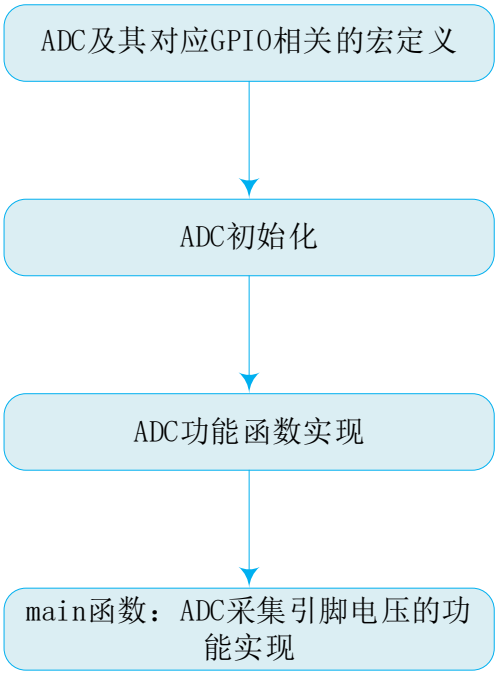



图 4.10.4 软件设计流程图

**2. 软件代码讲解**

（1）定义 ADC 及其对应 GPIO 相关的宏。

代码段 4.10.3 ADC 宏定义（adc1.h）

```
/********************ADC 所需 GPIO 引脚定义********************/
#define ADC1_GPIO_PORT        GPIOA
#define ADC1_PIN              GPIO_Pin_5
//ADC 对应的 GPIO 时钟挂载在 AHB1 上
#define ADC1_GPIO_CLOCK       RCC_AHB1Periph_GPIOA
```

```
//ADC 对应的 GPIO 时钟使能，其为宏函数定义
#define ADC1_GPIO_CLK_INIT  RCC_AHB1PeriphClockCmd
/************************ADC 定义************************/
//通道 1
#define ADC1_CH             ADC_Channel_5
//ADC 时钟挂载在 APB2 上
#define ADC1_CLOCK          RCC_APB2Periph_ADC1
//ADC 时钟使能，其为宏函数定义
    #define ADC1_CLK_INIT       RCC_APB2PeriphClockCmd
```

---

（2）对 ADC 相关配置进行初始化。

代码段 4.10.4 ADC 初始化函数（adc1.c）

---

```
/*
功   能：  ADC 初始化配置函数
参   数：  无
返回值：  无
*/
void ADC1_Init()
{
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;
    ADC_CommonInitTypeDef ADC_CommonInitStructure;

    ADC1_GPIO_CLK_INIT(ADC1_GPIO_CLOCK, ENABLE);        //开启时钟
    ADC1_CLK_INIT(ADC1_CLOCK, ENABLE);

    GPIO_InitStructure.GPIO_Pin = ADC1_PIN;             //选择端口
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AN;        //模拟输入
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;    //内部无上下拉
    GPIO_Init(ADC1_GPIO_PORT, &GPIO_InitStructure);     //初始化配置

    /********************ADC 通道设置********************/
    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;                    //独立模式
     ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_20Cycles;//两个采样
阶段 20 个时钟
    ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;     //DMA 传输失
能
    //预分频 8 分频 ADCCLK = APB2CLK/8 = 84/8 = 10.5MHz;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div8;
```

```
    ADC_CommonInit(&ADC_CommonInitStructure);                                    //初始化配置

    /*******************ADC 设置*******************/
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;                       //ADC 精度 12
位
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;                                //扫描模式
    ADC_InitStructure.ADC_ContinuousConvMode = ENABLE;                           //连续转换
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None; //禁止触发检
测
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;                       //转换结果右
对齐
    ADC_InitStructure.ADC_NbrOfConversion = 1;                                   //1 个转换在
序列规则中
    ADC_Init(ADC1, &ADC_InitStructure);                                          //初始化配置
    ADC_Cmd(ADC1, ENABLE);                                                       //ADC 使能
}
```

（3）ADC 功能函数实现，包括获取 ADC 采集函数和平均滤波函数。

代码段 4.10.5 ADC 功能函数（adc1.c）

```
/*
功   能:  ADC 采集函数
参   数:  ch      采集通道
返回值:  最近一次的转换结果
*/
uint16_t Get_ADC1(uint8_t ch)
{
    ADC_RegularChannelConfig(ADC1, ch, 1, ADC_SampleTime_480Cycles);     //ADCX,ADC 通道,规则
序列,采样时间
    ADC_SoftwareStartConv(ADC1);                                         //开启软件 ADC 功能
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);              //等待转换结束
    return ADC_GetConversionValue(ADC1);                                  //返回最近一次转换结
果
}

/*
功   能:  获取通道 ch 的转换值，取 times 次,然后平均
参   数:  ch       通道编号
          times    获取次数
返回值:  通道 ch 的 times 次转换结果平均值
```

```
*/
uint16_t Get_ADC1_Average(uint8_t ch, uint8_t times)
{
    uint32_t temp_val = 0;
    uint8_t t;

    for (t = 0; t < times; t++)
    {
        temp_val += Get_ADC1(ch);
    }
    return temp_val / times;
}
```

（4）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而 while 循环中需要实现获取电源电量百分比，并将其发送至串口的功能。

代码段 4.10.6 主函数（main.c）

```
int main(void)
{
    char show[20];
    Delay_Init();                   //延时初始化
    Led_Init();                     //LED 初始化
    Electricity_Init();             //ADC 初始化
    USART3_Init(115200);            //串口初始化
    while(1)
    {
        Power_Check();  //检测电源电量
        //将 ADC 转换值进行计算并发送至串口
        sprintf(show,"电量: %d%% \r\n",pwr_ck_l);
        USART3_Send_String(show);
        LED1_TOGGLE();
        Delay_ms(500);
    }
}
```

## 4.10.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块、杜邦线若干。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |

| ST Link 下载器 | USB 转 TTL 模块 |
|---|---|
| | |

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，ADC 将采集 PA1 端口的电压数据，观察串口调试助手打印的电压数据。

## 4.10.5 结果展示

将核心板使用 12V 锂电池供电，通过串口即可观察到电量百分比数据。

如图 4.10.5 所示，展示了串口调试助手打印电量数据。

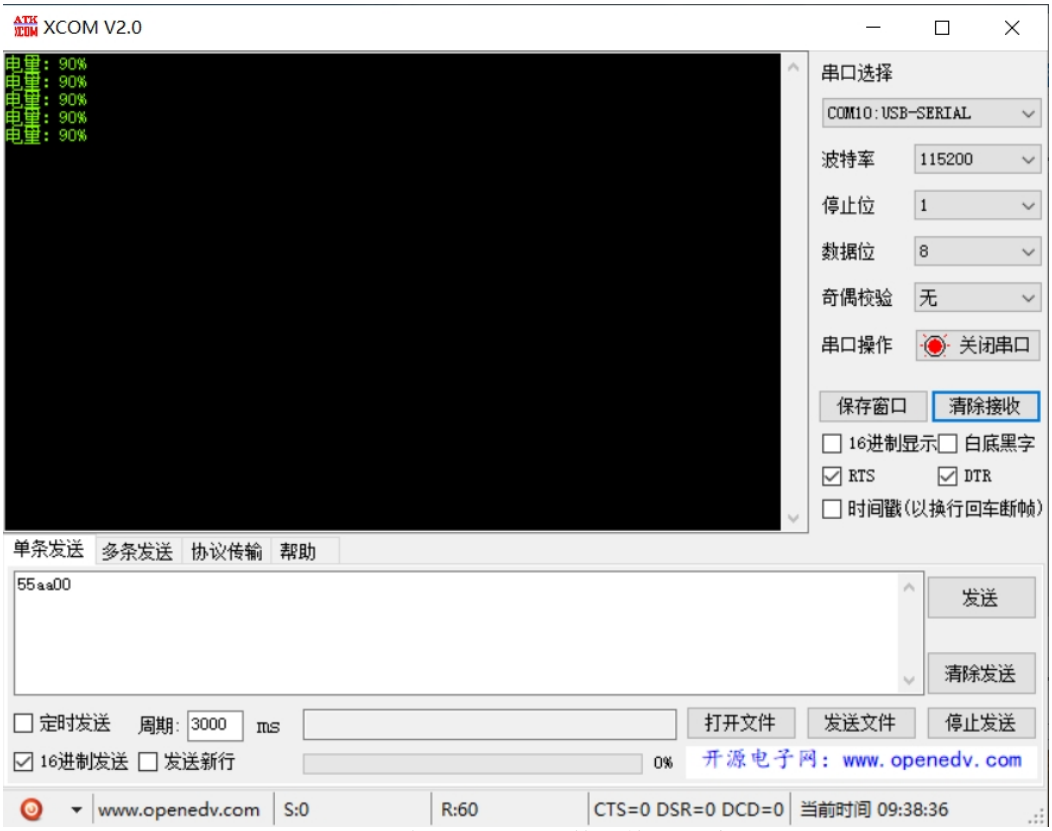


图 4.10.5 串口打印电量数据格式示意图

# 4.11 内部温度传感器实验

本实验通过介绍 STM32 中的内部温度传感器，使读者学会使用 ADC 读取温度传感器转换值，并通过公式转化为温度值。通过本实验，读者将了解到内部温度传感器的使用方法、ADC 模块的工作原理，以及如何实现实时的温度数据传输。

## 4.11.1 内部温度传感器介绍

**1. STM32F4xx 的内部温度传感器**

内部温度传感器可用于测量芯片的内部环境温度($T_A$)；不使用时可将传感器置于掉电模式。

（1）对于 STM32F40x 和 STM32F41x 系列芯片，温度传感器内部连接到 ADC1_IN16 通道，而 ADC1 用于将传感器输出电压转换为数字值。

（2）对于 STM32F42x 和 STM32F43x 器件，温度传感器内部连接到与 VBAT 共用的输入通道 ADC1_IN18：ADC1_IN18 用于将传感器输出电压或 VBAT 转换为数字值。一次只能选择一个转换（温度传感器或 VBAT）。同时设置了温度传感器和 VBAT 转换时，将只进行 VBAT 转换。

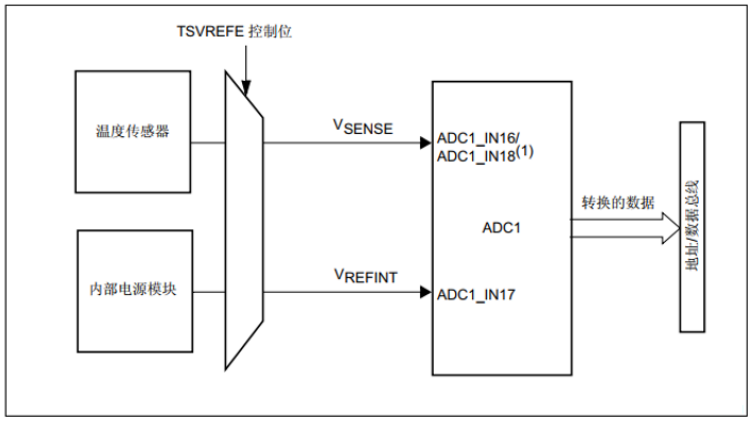
**2. 主要特性**

（1）支持的温度范围：-40℃到 125℃。

（2）精度：±1.5℃。

**3. 通道框图**

如图 4.11.1，展示了温度传感器和 $V_{REFINT}$ 的通道框图。




1. $V_{SENSE}$ 是至 ADC1_IN16 的输入（对于 STM32F40x 和 STM32F41x 器件），也是 ADC1_IN18 的输入（对于 STM32F42x 和 STM32F43x 器件）。

图 4.11.1 温度传感器和 $V_{REFINT}$ 通道框图

**4. 读取温度流程**

要使用 STM32F405xx/STM32F407xx 的内部温度传感器，需要执行以下步骤：

（1）根据芯片型号选择 ADC1_IN16（或 ADC1_IN18）输入通道。

（2）选择一个采样时间，该采样时间要大于指定的最低采样时间 10us。

（3）将温度传感器从掉电模式中唤醒。

相关函数：ADC_TempSensorVrefintCmd(FunctionalState NewState)：函数参数选择 ENABLE。

（4）开启 ADC1 转换。

相关函数：ADC_SoftwareStartConv(ADC_TypeDef* ADCx)：函数参数选择 ADC1。

（5）读取 ADC1 通道 16（或 18）的 $V_{SENSE}$ 数据。

（6）计算温度值，公式如下：

$$温度值(单位：℃) = \frac{V_{SENSE} - V_{25}}{AvgSlope} + 25$$

其中：

$V_{25}$：25℃时的 $V_{SENSE}$ 值。常规值为：0.76 V。

AvgSlope：温度与 $V_{SENSE}$ 曲线的平均斜率（以 mV/℃或 μV/℃表示）。常规值为：2.5 mV/℃。

（数据来源：《STM32F4xx 数据手册》Table 69. Temperature sensor characteristics）。

**5. 注意事项**

（1）传感器从掉电模式中唤醒需要一个启动时间，启动时间过后其才能正确输出 $V_{SENSE}$。ADC 在上电后同样需要一个启动时间。

因此，为尽可能减少延迟间，应使能：

温度传感器和 VREFINT 通道：“ADC_TempSensorVrefintCmd(ENABLE);”；

ADC1 的外设：“ADC_Cmd(ADC1,ENABLE);”。

（2）温度传感器的输出电压随温度线性变化。由于工艺不同，该线性函数的偏移量取决于各个芯片（芯片之间的温度变化可达 45℃）。

（3）内部温度传感器更适用于对温度变量而非绝对温度进行测量的应用情况。如果需要读取精确温度，则应使用外部温度传感器。

（4）由于内部温度传感器需要使用到模拟复用器(Analog mux)的功能，将一个

GPIO 端口进行复用并初始化即可。

**6. ADC 中内部温度传感器相关函数讲解**

启用或禁用温度传感器和 $V_{REFINT}$ 通道函数

表 4.11.1 启用或禁用温度传感器和 $V_{REFINT}$ 通道函数

| 接口函数：ADC_TempSensorVrefintCmd | |
|---|---|
| 函数原型 | void ADC_TempSensorVrefintCmd(FunctionalState NewState) |
| 功能描述 | 启用或禁用温度传感器和 $V_{REFINT}$ 通道。 |
| 入口参数 | 温度传感器和 $V_{REFINT}$ 通道的状态，该参数可以是 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | ADC_TempSensorVrefintCmd(ENABLE); |
| 注意事项 | 无 |

## 4.11.2 硬件设计

ADC 内部温度传感器相关内容不涉及外部硬件电路。

## 4.11.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 ADC1 部分的驱动代码独立开来，源文件（adc1.c）和头文件（adc1.h）存放在实验案例根目录中的 HARDWARE/ADC1 下。

**1. 软件设计流程**

如图 4.11.2，展示了本实验的软件设计流程图。

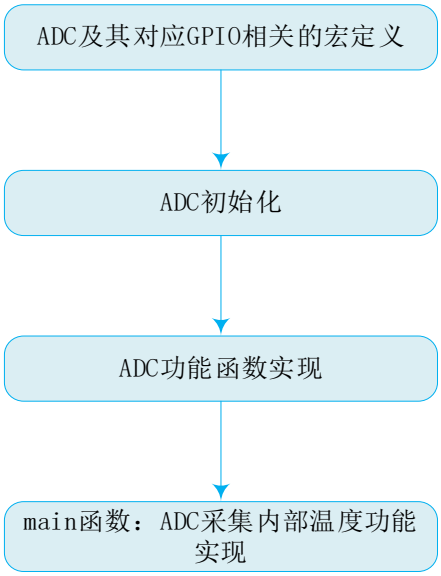



图 4.11.2 软件设计流程图

**2. 软件代码讲解**

（1）定义 GPIO 端口和 ADC 内部温度接口相关的宏。

代码段 4.11.1 引脚和接口宏定义（adc1.h）

```
/********************ADC 所需端口定义********************/
#define ADC1_GPIO_PORT       GPIOA
//ADC 对应的 GPIO 时钟挂载在 AHB1 上
#define ADC1_GPIO_CLOCK      RCC_AHB1Periph_GPIOA
//ADC 对应的 GPIO 时钟使能，其为宏函数定义
#define ADC1_GPIO_CLK_INIT   RCC_AHB1PeriphClockCmd
/************************ADC 定义************************/
//内部温度采集对应的 ADC 通道为 ADC1_CH16
//通道 1
#define ADC1_CH              ADC_Channel_16
//ADC 时钟挂载在 APB2 上
#define ADC1_CLOCK           RCC_APB2Periph_ADC1
//ADC 时钟使能，其为宏函数定义
#define ADC1_CLK_INIT        RCC_APB2PeriphClockCmd
```

（2）对 GPIO 端口和 ADC 内部温度相关配置进行初始化。

代码段 4.11.2 初始化函数（adc1.c）

```
/*
功　能：ADC 初始化配置函数
参　数：无
返回值：无
*/
void ADC1_Init()
{
    GPIO_InitTypeDef GPIO_InitStructure;
    ADC_InitTypeDef ADC_InitStructure;
    ADC_CommonInitTypeDef ADC_CommonInitStructure;

    ADC1_GPIO_CLK_INIT(ADC1_GPIO_CLOCK, ENABLE);                    //开启时钟
    ADC1_CLK_INIT(ADC1_CLOCK, ENABLE);

    //需要使用到模拟复用器(Analog mux)的功能，设置一个 GPIO 端口进行初始化即可
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_Init(ADC1_GPIO_PORT, &GPIO_InitStructure);
    /********************ADC 通道设置********************/
    ADC_CommonInitStructure.ADC_Mode = ADC_Mode_Independent;        //独立模式
    ADC_CommonInitStructure.ADC_TwoSamplingDelay = ADC_TwoSamplingDelay_5Cycles;//两个采样阶
段 5 个时钟
    ADC_CommonInitStructure.ADC_DMAAccessMode = ADC_DMAAccessMode_Disabled;     //DMA 传输失
能
    //预分频 4 分频 ADCCLK = APB2CLK/8 = 84/4 = 21MHz;
    ADC_CommonInitStructure.ADC_Prescaler = ADC_Prescaler_Div4;     //初始化配置
    ADC_CommonInit(&ADC_CommonInitStructure);
    /********************ADC 设置********************/
    ADC_InitStructure.ADC_Resolution = ADC_Resolution_12b;          //ADC 精度 12
位
    ADC_InitStructure.ADC_ScanConvMode = DISABLE;                   //扫描模式
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;             //非连续转换
    ADC_InitStructure.ADC_ExternalTrigConvEdge = ADC_ExternalTrigConvEdge_None;//禁止触发检
测
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;          //转换结果右
对齐
    ADC_InitStructure.ADC_NbrOfConversion = 1;          //1 个转换在序列规则中
    ADC_Init(ADC1, &ADC_InitStructure);                 //初始化配置
```

```
    ADC_TempSensorVrefintCmd(ENABLE);                          //使能温度传感器和 VREFINT 通
道
    ADC_Cmd(ADC1, ENABLE);                                                    //ADC 使能
}
```

---

（3）ADC 功能函数实现。包括读取内部温度转换值、对转换值进行平均滤波处理及通过公式计算温度的功能函数的实现。

代码段 4.11.3 功能函数（adc1.c）

---

```
/*
功   能:  ADC 采集函数
参   数:  ch       采集通道
返回值:  最近一次的转换结果
*/
uint16_t Get_ADC1(uint8_t ch)
{
    ADC_RegularChannelConfig(ADC1, ch, 1, ADC_SampleTime_480Cycles);    //ADCX,ADC 通道,规则
序列,采样时间
    ADC_SoftwareStartConv(ADC1);                                        //开启软件 ADC 功能
    while (ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC) == RESET);          //等待转换结束

    return ADC_GetConversionValue(ADC1);                              //返回最近一次转换结果
}
/*
功   能:  获取通道 ch 的转换值，取 times 次,然后平均
参   数:  ch       通道编号
          times    获取次数
返回值:  通道 ch 的 times 次转换结果平均值
*/
uint16_t Get_ADC1_Average(uint8_t ch, uint8_t times)
{
    uint32_t temp_val = 0;
    uint8_t t;

    for (t = 0; t < times; t++)
    {
        temp_val += Get_ADC1(ch);
    }
    return temp_val / times;
```

```
}
/*
功   能:  获取 ADC 采集的转换值，并进行计算得出温度
参   数:  无
返回值:  温度值(单位:℃)
*/
float Get_Temprate(void)
{
  uint32_t adcx;
  float temperate;

  adcx = Get_ADC1_Average(ADC1_CH, 20);          //读取内部温度传感器数据
  temperate = (float)adcx*(3.3/4096);            //电压值
  temperate = ((temperate-0.76f)/0.0025f) + 25;  //转换为温度值
  return temperate;
}
```

（4）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而 while 循环中需要实现获取内部温度数据并将获取的值发送至串口的功能。

代码段 4.11.4 主函数（main.c）

```
int main(void)
{
    float temp;
    char show[20]={0};
    Delay_Init();                  //延时初始化
    Led_Init();                    //LED 初始化
    USART3_Init(115200);           //串口初始化
    ADC1_Init();                   //ADC 初始化

    while(1)
    {
        temp = Get_Temprate();     //获取 ADC 采集的内部温度值
        if(temp < 0)
        {
            temp=-temp;
        }
        sprintf(show, "Temperature: %.2f ℃\r\n ", temp);
```

```
        USART3_Send_String(show);
        LED1_TOGGLE();
        Delay_ms(500);
    }
}
```

---

## 4.11.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |

| ST Link 下载器 | USB 转 TTL 模块 |
|---|---|
| | |

2. 实验硬件接线：

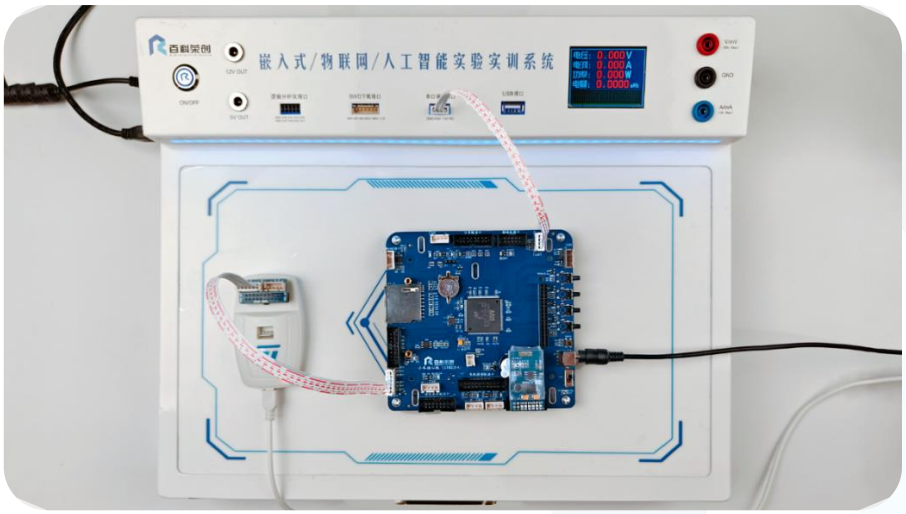



图 4.11.3 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，ADC 将采集芯片内部的电压数据，观察串口调试助手打印的电压数据。

## 4.11.5 结果展示

串口调试助手上实时显示当前芯片内部的温度值。如图 4.11.4 所示，展示了串口调试工具上显示的数据示意图。

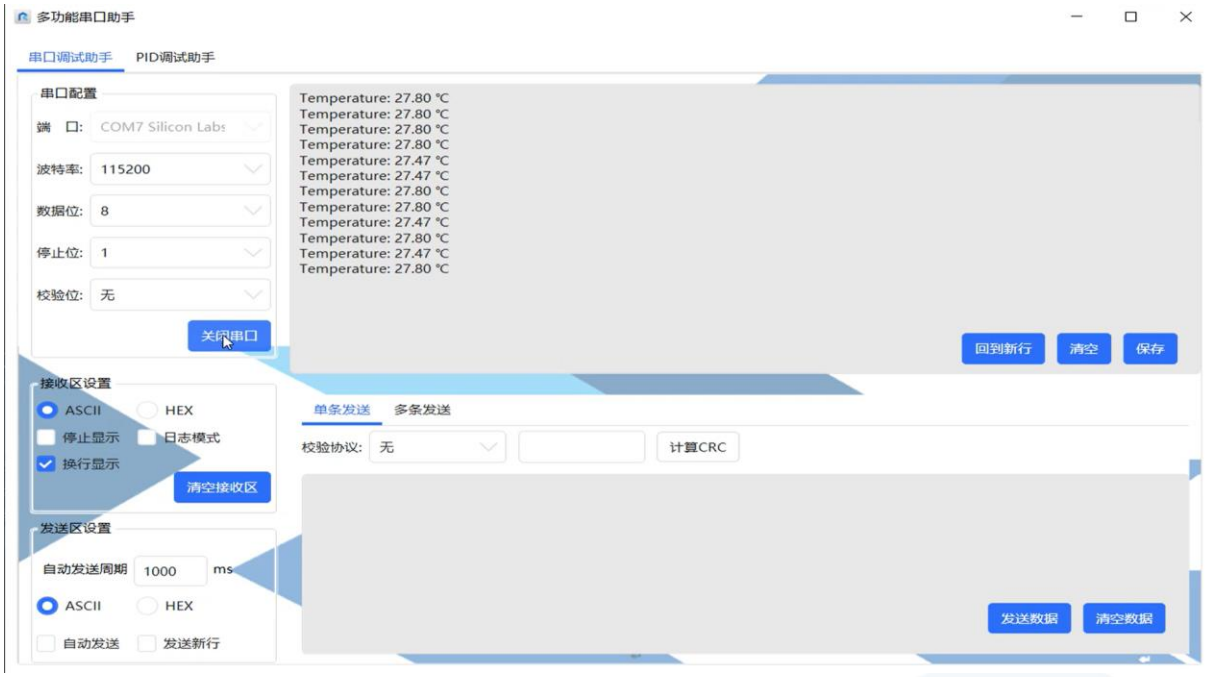



图 4.11.4 串口打印芯片内部温度值数据示意图

# 4.12 输出三角波实验

本实验通过使用 STM32 的 DAC 功能生成三角波形，旨在帮助读者学习如何有效利用 STM32 芯片中强大的模拟输出功能。

## 4.12.1 DAC 介绍

**1. DAC 概念**

数字模拟转换器（Digital-to-Analog Converter，简称 DAC），是一种电子设备或电路，可将数字信号转换为模拟信号。DAC 常见于许多电子系统中，例如音频设备、通信系统、工业控制和仪器仪表等。

DAC 的作用是将数字信号转换为模拟信号，使数字系统能够与模拟系统进行交互。数字信号在计算机和数字系统中以离散形式存在，而模拟信号是连续的。DAC 的主要功能是将来自数字系统的离散值转化为连续的模拟信号，以便能够在模拟系统中进行处理、放大、传输或控制。

**2. DAC 基本工作原理**

DAC 的基本工作原理是通过使用参考电压或参考电流，根据数字输入信号的二进制编码，生成相应的模拟输出。

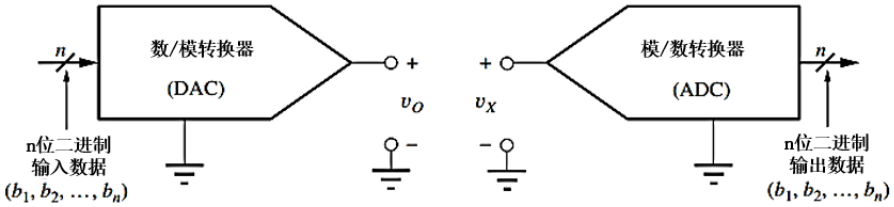



图 4.12.1 D/A 及 A/D 转换器的框图表示

在图 4.12.1 所示的 D/A 转换器中，其输入是来自于 n 位的二进制数（$b_1, b_2, \ldots, b_n$），输出可以看作为该二进制数的分式形式乘以满量程的参考电压，即生成对应的模拟输出电压 $v_o$。

**3. STM32 中的 DAC**

在 STM32F4xx 系列的芯片中，DAC 模块是 12 位电压输出数模转换器。DAC 可以按 8 位或 12 位模式进行配置，并且可与 DMA 控制器配合使用。在 12 位模式下，数据可以采用左对齐或右对齐。

DAC 有两个输出通道，每个通道各有一个转换器。

在 DAC 双通道模式下，每个通道可以单独进行转换；当两个通道组合在一起同步执行更新操作时，也可以同时进行转换。可通过一个输入参考电压引脚 $V_{REF+}$（与 ADC 共享）来提高分辨率。

**4. DAC 通道框图**

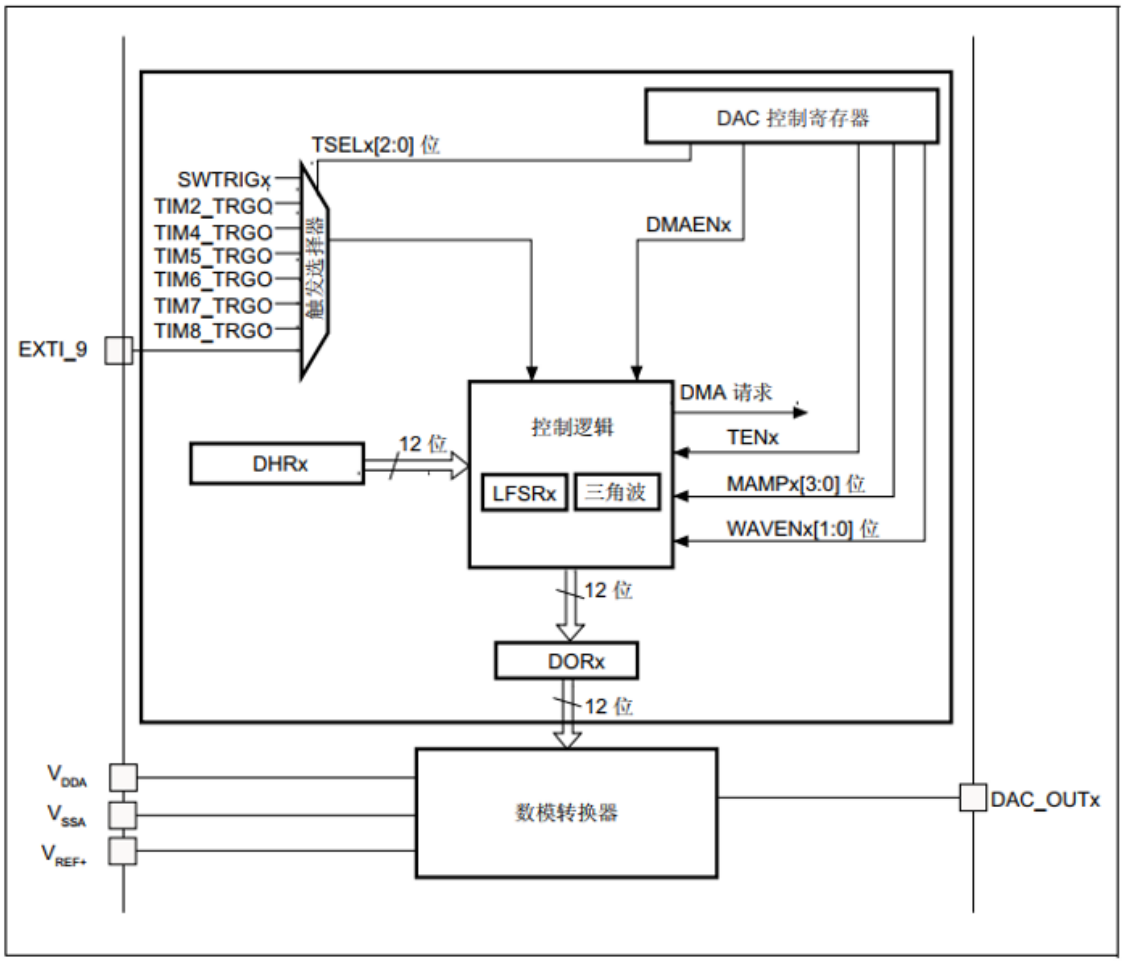



图 4.12.2 DAC 通道框图

STM32F4xx 中存在两个 DAC 转换器，其各对应一个输出通道，在使能 DAC 通道 x 后，相应 GPIO 引脚（PA4 或 PA5）将自动连接到模拟转换器输出（DAC_OUTx）。为了避免寄生电流消耗，应首先将 PA4 或 PA5 引脚配置为模拟模式（AIN）。

其中 PA4 对应 DAC_OUT1（通道 1），PA5 对应 DAC_OUT2（通道 2）。

如表 4.12.1 所示，展示了框图中模数转换器的引脚说明。

表 4.12.1 DAC 引脚说明

| 名称 | 信号类型 | 备注 |
|---|---|---|
| $V_{REF+}$ | 正模拟参考电压输入 | DAC 高/正参考电压<br>1.8V≤ $V_{REF+}$≤ $V_{DDA}$ |
| $V_{DDA}$ | 模拟电源输入 | 模拟电源 |
| $V_{SSA}$ | 模拟电源接地输入 | 模拟电源接地 |
| DAC_OUTx | 模拟输出信号 | DAC 通道 x 模拟输出 |

**5. DAC 功能说明**

（1）DAC 通道使能

将 DAC_CR 寄存器中的相应 ENx 位置 1，即可接通对应 DAC 通道。经过一段启动时间 $t_{WAKEUP}$ 后，DAC 通道被真正使能。

（2）DAC 数据格式

对于 DAC 单通道，可选数据格式为：8 位右对齐、12 位左对齐、12 位右对齐。

如所示，展示了 DAC 单通道模式下的数据寄存器。

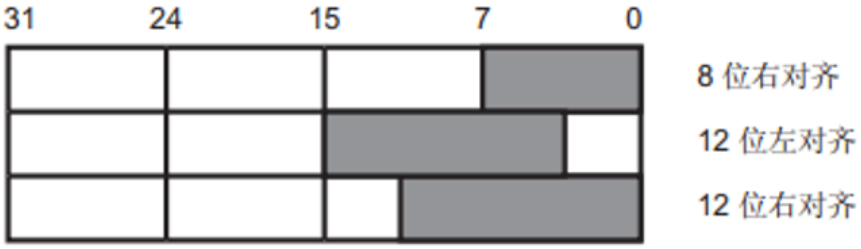



图 4.12.3 DAC 单通道模式下的数据寄存器

（3）DAC 转换

DAC_DORx 无法直接写入，任何数据都必须通过加载 DAC_DHRx 寄存器才能传输到 DAC 通道 x。

如果未选择硬件触发（DAC_CR 寄存器中的 TENx 位复位），那么经过一个 APB1 时钟周期后，DAC_DHRx 寄存器中存储的数据将自动转移到 DAC_DORx 寄存器。但是，如果选择硬件触发（置位 DAC_CR 寄存器中的 TENx 位）且触发条件到来，将在三个 APB1 时钟周期后进行转移。

如图 4.12.4 所示，展示了未选择硬件触发时的 DAC 转换时序图。

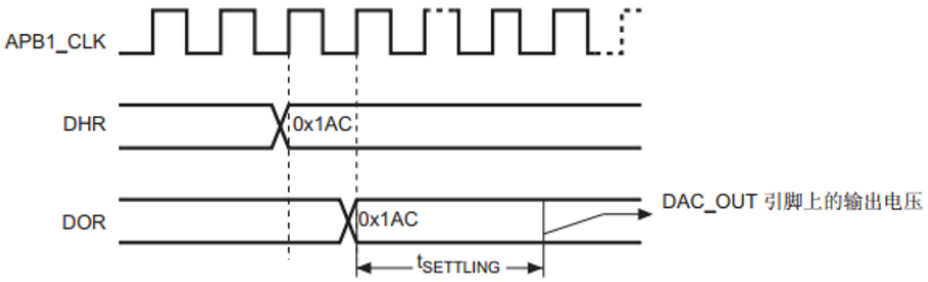



图 4.12.4 关闭触发(TEN=0)时的转换时序图

（4）DAC 输出电压

经过线性转换后，数字输入会转换为 0 到 VREF+之间的输出电压。各 DAC 通道引脚的模拟输出电压通过以下公式确定：

$$DAC_{output} = V_{REF} \times \frac{DOR}{4095}$$

**6. DAC 相关函数讲解**

（1）DAC 初始化结构体定义

代码段 4.12.1 DAC 初始化结构体定义

```
typedef struct
{
  uint32_t  DAC_Trigger;
  uint32_t  DAC_WaveGeneration;
  uint32_t  DAC_LFSRUnmask_TriangleAmplitude;
  uint32_t  DAC_OutputBuffer;
}DAC_InitTypeDef;
```

关于 DAC 初始化结构体各个成员的详细描述，如表 4.12.2 所示。

表 4.12.2 DAC 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| DAC_Trigger | （1）DAC_Trigger_Ext_IT9<br>（2）DAC_Trigger_None<br>（3）DAC_Trigger_Software<br>（4）DAC_Trigger_Tx_TRGO（x 可以为 2,4,5,6,7,8） | 指定所选 DAC 通道的外部触发器 |
| DAC_WaveGeneration | DAC_WaveGeneration_Noise：噪声波<br>DAC_WaveGeneration_None：不产生波<br>DAC_WaveGeneration_Triangle：三角波 | 指定是否产生 DAC 通道噪声波、三角波或不产生波。 |

| | | |
|---|---|---|
| DAC_LFSRUnmask_TriangleAmplitude | （1）DAC_LFSRUnmask_Bit0<br>（2）DAC_LFSRUnmask_Bitsx_0（x 可以为 1~11）<br>（3）DAC_TriangleAmplitude_x（x 可以为 1,3,7,15,31,63,127,255,511,1023,2047,4095） | 为 DAC 通道指定用于生成噪声波或最大振幅三角波的 LFSR 掩码 |
| DAC_OutputBuffer | DAC_OutputBuffer_Disable：禁用<br>DAC_OutputBuffer_Enable：启用 | 指定 DAC 通道输出缓冲器是启用还是禁用 |

（2）DAC 初始化函数

表 4.12.3 DAC 初始化函数

| 接口函数：DAC_Init | |
|---|---|
| 函数原型 | void DAC_Init(uint32_t DAC_Channel, DAC_InitTypeDef *DAC_InitStruct) |
| 功能描述 | 根据 DAC_InitStruct 中指定的参数初始化 DAC 外设。 |
| 入口参数 | DAC_Channel：所选 DAC 通道，其参数可以为 DAC_Channel_1 或 DAC_Channel_2。 |
| | DAC_InitStruct：指向 DAC_InitTypeDef 结构体的指针，该结构体包含指定 DAC 通道的配置信息。 |
| 返回值 | 无 |
| 使用示例 | DAC_Init(DAC_Channel_1, &DAC_InitStructure); |
| 注意事项 | 无 |

（3）启用或禁用指定的 DAC 通道函数

表 4.12.4 启用或禁用指定的 DAC 通道函数

| 接口函数：DAC_Cmd | |
|---|---|
| 函数原型 | void DAC_Cmd(uint32_t DAC_Channel, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 DAC 通道。 |
| 入口参数 | DAC_Channel：所选 DAC 通道，其参数可以为 DAC_Channel_1 或 DAC_Channel_2。 |
| | NewState：DAC 通道的状态，该参数可以是 ENABLE 或 DISABLE。 |

| | |
|---|---|
| 返回值 | 无 |
| 使用示例 | DAC_Cmd(DAC_Channel_1, ENABLE); |
| 注意事项 | 启用 DAC 通道后，就不能再修改触发源了。 |

（4）为 DAC 通道 1 设置数据函数

表 4.12.5 为 DAC 通道 1 设置数据函数

| 接口函数：DAC_SetChannel1Data | |
|---|---|
| 函数原型 | void DAC_SetChannel1Data(uint32_t DAC_Align, uint16_t Data) |
| 功能描述 | 为 DAC 通道 1 设置指定的数据保持寄存器值 |
| 入口参数 | DAC_Align：指定 DAC 通道 1 的数据对齐方式，其参数可以为 DAC_Align_8b_R（8 位数据右对齐方式）、DAC_Align_12b_L（12 位数据左对齐方式）、DAC_Align_12b_R（12 位数据右对齐方式）。 |
| | Data：所选数据寄存器中要加载的数据。 |
| 返回值 | 无 |
| 使用示例 | DAC_SetChannel1Data(DAC_Align_12b_R, 0); |
| 注意事项 | 无 |

## 4.12.2 硬件设计

**1. 实验电路分析**

DAC 的功能部分不涉及外部硬件电路。如图 4.12.5 所示，为 DAC 通道 1 对应测量引脚的电路原理图，其引脚位于核心板 J6 的引脚 2。

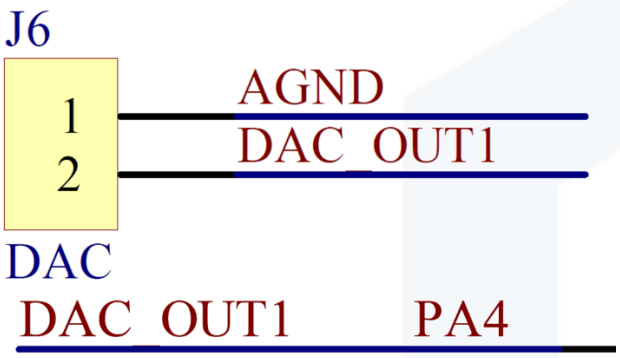



图 4.12.5 DAC 测量引脚原理图

## 4.12.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 DAC1 部分的驱动代码独立开来，源文件（dac1.c）和头文件（dac1.h）存放在实验案例根目录中的 HARDWARE/DAC1 下。

**1. 软件设计流程**

如图 4.12.6，展示了本实验的软件设计流程图。

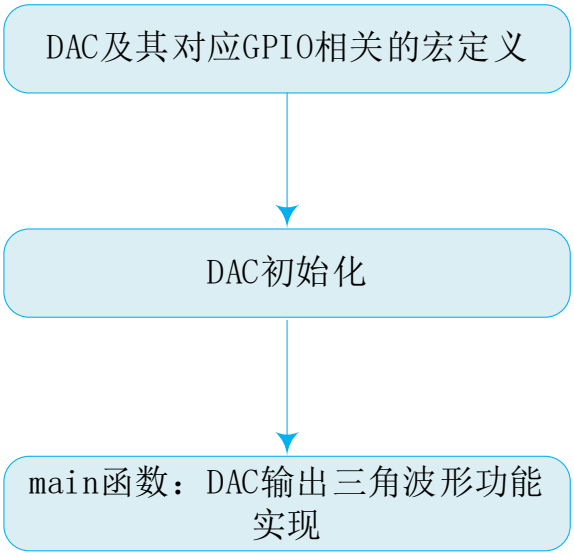



图 4.12.6 软件设计流程图

**2. 软件代码讲解**

（1）定义 DAC 及其对应 GPIO 相关的宏。

代码段 4.12.2 DAC 引脚和接口宏定义（dac1.h）

```
/********************GPIO 相关定义********************/
//DAC 通道            对应 GPIO 引脚
//DAC_Channel_1       PA4
//DAC_Channel_2       PA5
#define DAC_CHANNEL1_PORT        GPIOA
#define DAC_CHANNEL1_PIN         GPIO_Pin_4

//DAC 对应的 GPIO 时钟挂载在 AHB1 上
#define DAC_GPIO_CLOCK           RCC_AHB1Periph_GPIOA

//DAC 对应的 GPIO 时钟使能，其为宏函数定义
```

```
#define DAC_GPIO_CLK_INIT        RCC_AHB1PeriphClockCmd

/*******************DAC 相关定义*******************/
//DAC 通道
#define DAC_CHANNEL              DAC_Channel_1

//DAC 时钟挂载在 APB1 上
#define DAC_CLOCK                RCC_APB1Periph_DAC

//DAC 时钟使能，其为宏函数定义
#define DAC_CLK_INIT             RCC_APB1PeriphClockCmd
```

---

（2）对 GPIO 端口和 DAC 相关配置进行初始化。

代码段 4.12.3 初始化函数（dac1.c）

---

```
/*
功 能：DAC 初始化
参 数：无
返回值：无
*/
void DAC1_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    DAC_InitTypeDef DAC_InitStructure;

    DAC_GPIO_CLK_INIT(DAC_GPIO_CLOCK, ENABLE);          //时钟使能
    DAC_CLK_INIT(DAC_CLOCK, ENABLE);

    /* GPIO 初始化 */
    GPIO_InitStructure.GPIO_Pin = DAC_CHANNEL1_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;       //配置为模拟输入模式
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_100MHz;
    GPIO_Init(DAC_CHANNEL1_PORT, &GPIO_InitStructure);

    /* DAC 初始化 */
    DAC_InitStructure.DAC_Trigger = DAC_Trigger_None;                    //不使用触发功能,对应
寄存器中 TEN1=0,TSEL1[2:0]=000
    DAC_InitStructure.DAC_WaveGeneration = DAC_WaveGeneration_None;     //不使用自带的波形生成
功能
```

```
    //通过线性反馈移位寄存器生成伪噪声，仅当 DAC_WaveGeneration 配置为 DAC_WaveGeneration_Noise 时有效
    DAC_InitStructure.DAC_LFSRUnmask_TriangleAmplitude = DAC_LFSRUnmask_Bit0;
    DAC_InitStructure.DAC_OutputBuffer = DAC_OutputBuffer_Disable;    //禁用输出缓存
    DAC_Init(DAC_CHANNEL, &DAC_InitStructure);

    DAC_Cmd(DAC_CHANNEL, ENABLE);                        //使能 DAC

    DAC_SetChannel1Data(DAC_Align_12b_R, 0);             //输出 0V
}
```

（3）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而 while 循环中需要实现 DAC 输出三角波的功能。

代码段 4.12.4 主函数（main.c）

```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2); //使用中断,配置优先级组别 2
    Delay_Init();                                  //延时初始化
    DAC1_Init();                                   //DAC 通道 1 初始化

    while (1)                                      //三角波
    {
        for(int i=0; i<=100; i++)                  //递增
        {
            //DAC 输出数据
            DAC_SetChannel1Data(DAC_Align_12b_R, (uint16_t)((4000 / 100) * i));
        }

        for(int i=100; i>=0; i--)                  //递减
        {
            //DAC 输出数据
            DAC_SetChannel1Data(DAC_Align_12b_R, (uint16_t)((4000 / 100) * i));
        }
    }
}
```

## 4.12.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、示波器。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |

| ST Link 下载器 | 示波器 |
|---|---|
| | |

2. 实验硬件接线：

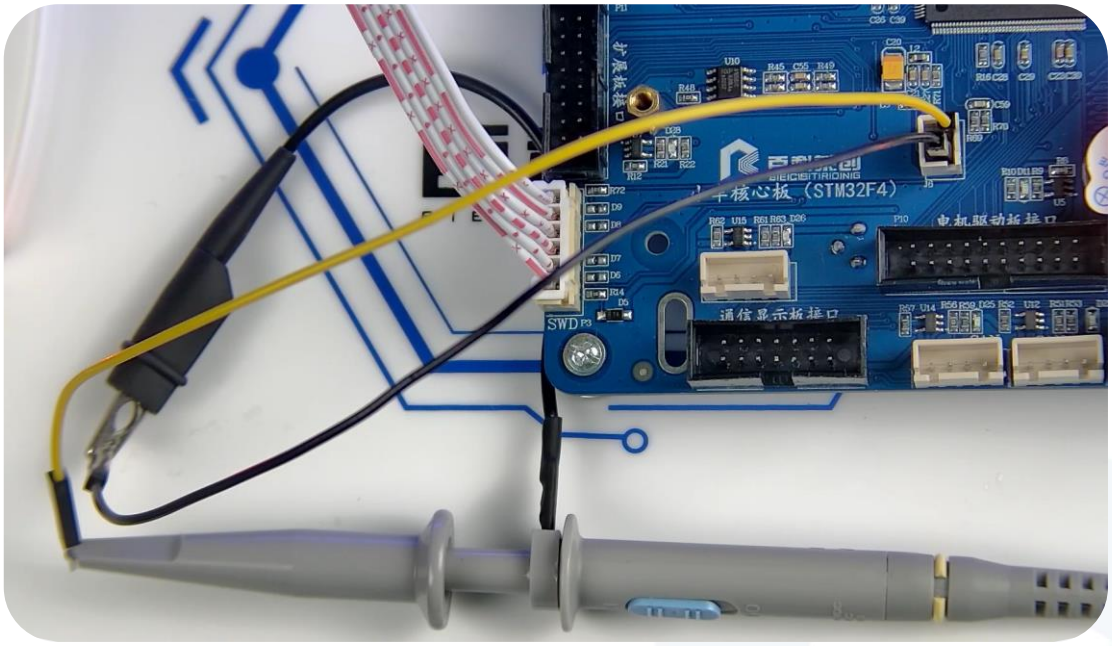



图 4.12.7 示波器接线示意图

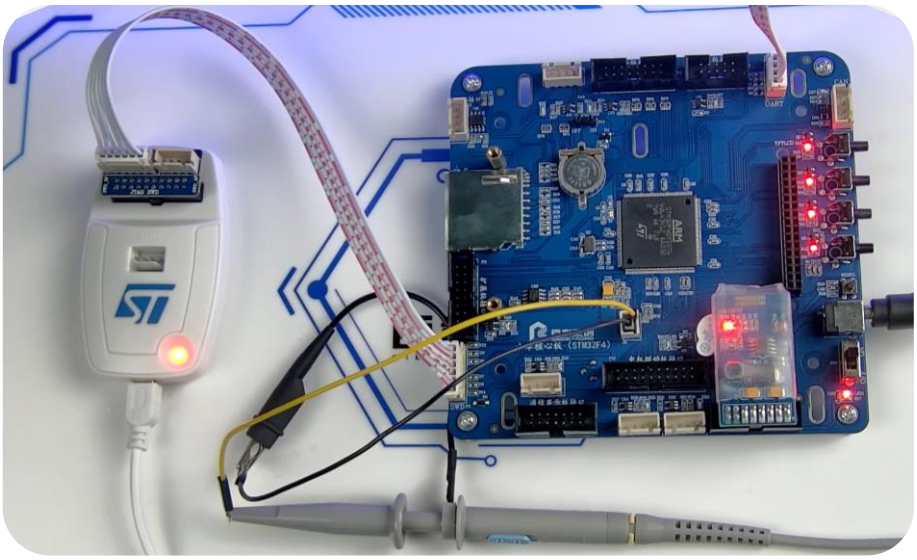


图 4.12.8 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 使用 IDE 将程序编译下载至核心板，将示波器测量探头接至测量引脚处（示波器与核心板的接地端也需相接），通过校准示波器，观察是否有三角波形。

## 4.12.5 结果展示

将程序下载至核心板后，连接示波器至核心板的 PA4 引脚（确保示波器的接地端与核心板接地端连接），配置好示波器，即可在示波器上观察到完整的三角波形。核心板的接地引脚和 PA4 引脚分别位于核心板 J6 的引脚 1 和引脚 2 位置。

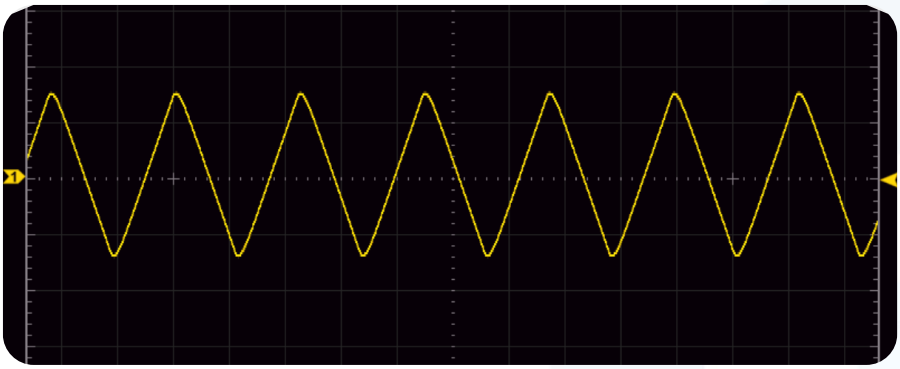


图 4.12.9 示波器波形

# 4. 13 SPI 总线驱动 OLED 显示实验

## 4. 13. 1 OLED 屏幕介绍

图 4.13.1 OLED 显示

OLED（有机发光二极管）屏幕以其自发光特性、高对比度、广视角、快速响应时间、薄型设计和鲜艳的色彩表现而著称。这些特点使得 OLED 屏幕非常适合用于高端智能手机、平板电脑、电视和显示器，以及在可穿戴设备、汽车内部显示屏和创新型弯曲或卷曲显示技术中的应用。

1. OLED 屏幕介绍

通信显示板采用 2.42 寸 OLED 屏幕。屏幕分辨率为 128*64，采用 SSD1309 驱动 IC，该芯片内部集成 DCDC 升压，仅需 3.3V 供电，即可正常工作，无需用户再添加升压 DCDC 电路。

图 4.13.2 OLED 屏幕

图 4.13.2 是屏幕的实物图，能直观看到的是有 FPC 排线和液晶显示板。然而内部有着很复杂的电路。图 4.13.3 中就是 OLED 屏幕内部的控制电路结构框图，由 MCU 控制器、图形显示数据缓存区（RAM\GDDRAM）、显示控制器、命令译码器、电源控

制器、段控制器等组成。微控制器无法直接对内部的控制器进行设置，需要于基础的 MCU 控制器通过总线接口进行通信，实现参数配置和数据读写。

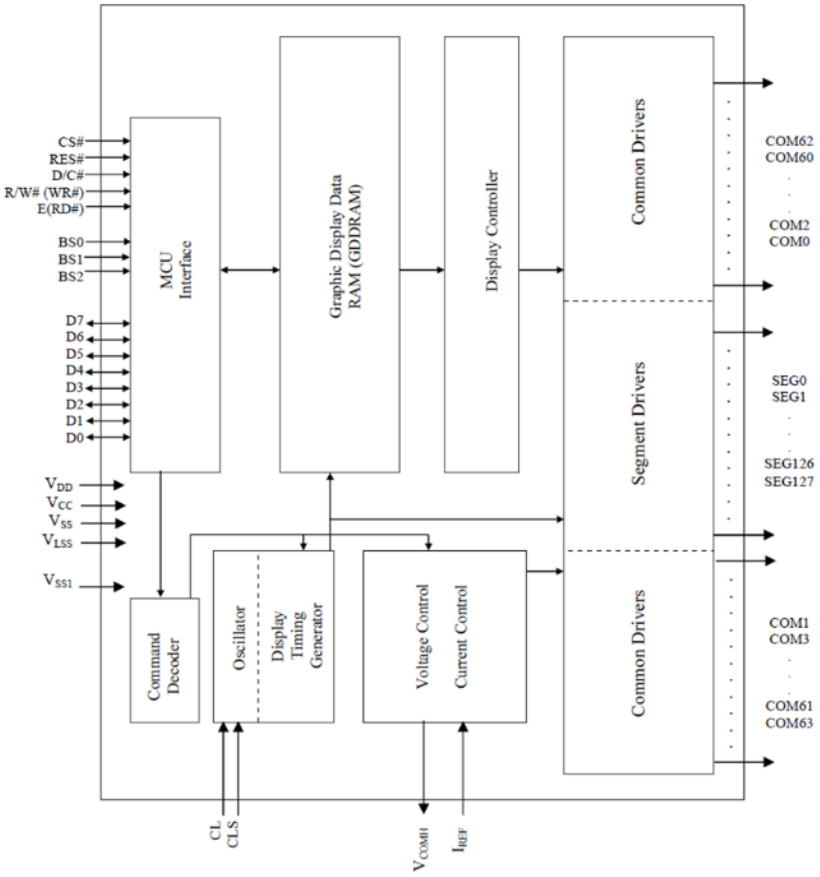



图 4.13.3 SSD1309 结构框图

（1）SSD1309 引脚说明

SD1309 内部引脚有 249 个引脚，其中 1~31 好引脚是由用户设计相关电路连接，32~249 则是由屏幕内部连接，实现对屏幕中的 OLED 像素点的控制。

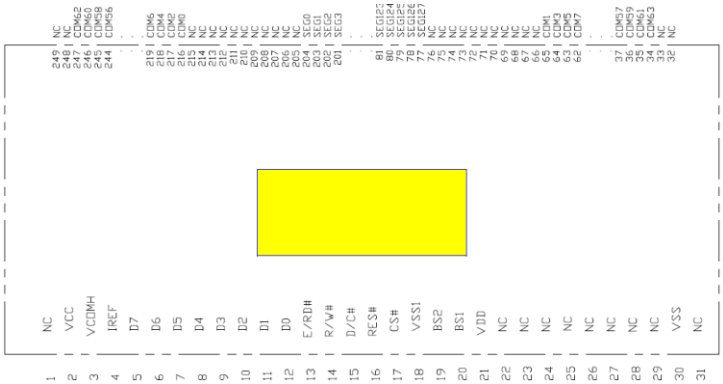



图 4.13.4 引脚分布

（2）SSD1309 接口选择

通过阅读数据手册会发现，OLED 屏幕一共有 5 总接口模式。如图 4.13.5 所示。其中 BS0~2 的设定于硬件电路设计有关。

| SSD1306 Pin Name | I²C Interface | 6800-parallel interface (8 bit) | 8080-parallel interface (8 bit) | 4-wire Serial interface | 3-wire Serial interface |
|---|---|---|---|---|---|
| BS0 | 0 | 0 | 0 | 0 | 1 |
| BS1 | 1 | 0 | 1 | 0 | 0 |
| BS2 | 0 | 1 | 1 | 0 | 0 |

图 4.13.5 微控制器总线接口引脚选择

结合项目电路原理图，如图 4.13.6 所示。其中 BS1、2 被直接拉低,BS0 被默认连接到内部 V_SS，这里我们硬件设计的是 4 线 SPI 工作模式。

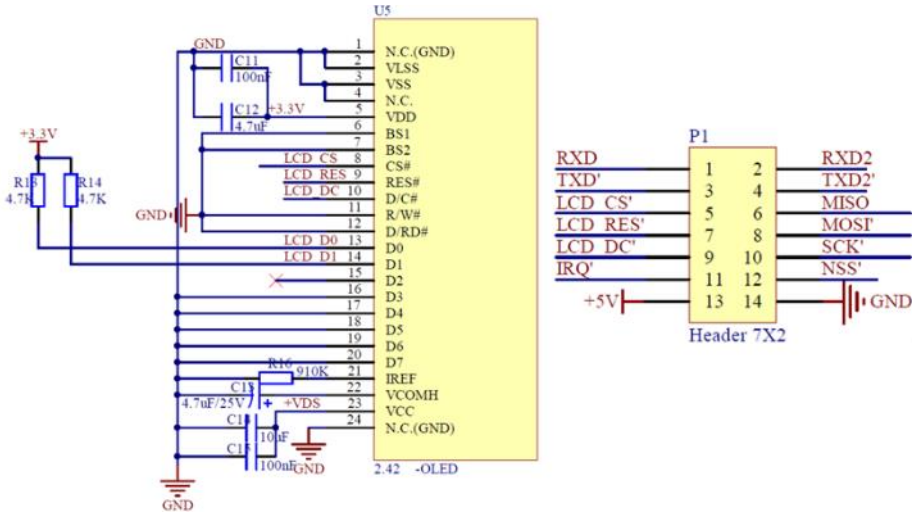



图 4.13.6 OLED 原理图

（3）4 线 SPI 接口要求

4 线 SPI 接口由串行时钟 SCK、串行数据引脚 SDIN、DC、CS 组成。D0 扮演 SCK 角色。D1 作为 MOSI，对于没有使用到 D2 引脚，保持悬空状态。从 D3~7、R/W#和 E 都需要接地。RST 引脚手册这部分中没有进行列举，该引脚用于控制 OLED 复位。

| Function | E | R/W# | CS# | D/C# | D0 |
|---|---|---|---|---|---|
| Write command | Tie LOW | Tie LOW | L | L | ↑ |
| Write data | Tie LOW | Tie LOW | L | H | ↑ |

图 4.13.7 4 线 SPI 控制引脚

注：H 代表高电平，L 代表低电平，↑代表上升沿。

首先 CS#引脚拉低选中设备，微控制器可与开始于 SSD1309 开始进行数据传输。D/C#引脚对传输的数据进行设定，为低电平则表示写命令，为高表示写数据。每次传

输以一个字节为单位进行传输，数据在 SCK 上升沿进行被采样，并且高位在前优先发送。在 CS#没有变为高电平之前，支持进行连续数据传输。直至取消选中设备，完成数据传输。如图 4.13.8 所示。

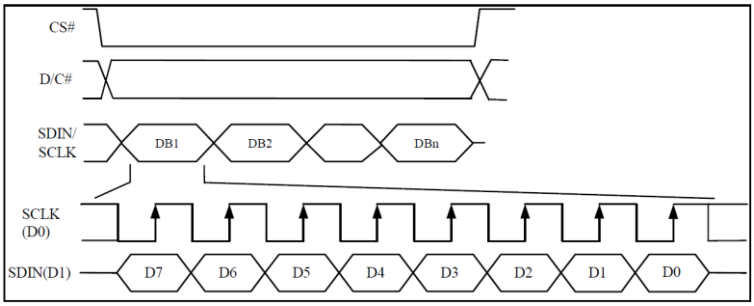



图 4.13.8 4 线 SPI 接口模式下的写时序

（4）显存与指令

1）显存

SSD1309 的显存总共为 128*64bit 大小，SSD1309 将这些显存分为了 8 页，从 Page0 到 Page7，用于单色 128x64 点阵显示，如图 4.13.9 所示。

| | | Row re-mapping |
|---|---|---|
| PAGE0 (COM0-COM7) | Page 0 | PAGE0 (COM 63-COM56) |
| PAGE1 (COM8-COM15) | Page 1 | PAGE1 (COM 55-COM48) |
| PAGE2 (COM16-COM23) | Page 2 | PAGE2 (COM47-COM40) |
| PAGE3 (COM24-COM31) | Page 3 | PAGE3 (COM39-COM32) |
| PAGE4 (COM32-COM39) | Page 4 | PAGE4 (COM31-COM24) |
| PAGE5 (COM40-COM47) | Page 5 | PAGE5 (COM23-COM16) |
| PAGE6 (COM48–COM55) | Page 6 | PAGE6 (COM15-COM8) |
| PAGE7 (COM56-COM63) | Page 7 | PAGE7 (COM 7-COM0) |
| | SEG0 ----------SEG127 | |
| Column re-mapping | SEG127 ----------SEG0 | |

图 4.13.9 SSD1309 的 GDDRAM 分页

当一个数据字节被写入显存当中，该数据将会写入设定的页和列地址中，将该列的数据填满。如图 4.13.10 所示，D0 为写入最上面一行，D7 写入最下面一行。这里读者可以理解为，我们无法直接控制单独的像素点，而是控制 8 个像素点为整体的这一列。

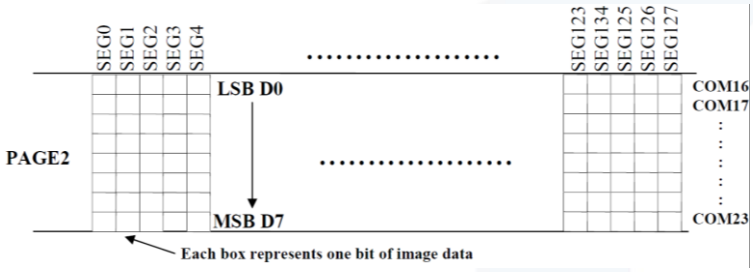



图 4.13.10 显存数据显示的方式（没有开启行、列重映射）

2）指令

SSD1309 的指令比较多，这里我们仅介绍几个常用的命令，这些命令如图 4.13.11 所示：

| 序号 | HEX | 各位描述 | | | | | | | | 指令 | 说明 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | | |
| 0 | 81 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 设置对比度 | A 的值越大，屏幕越亮 |
| | A[7:0] | A7 | A6 | A5 | A4 | A3 | A2 | A1 | A0 | | A[7:0]：0X00~0XFF |
| 1 | AE/AF | 1 | 0 | 1 | 0 | 1 | 1 | 1 | X0 | 设置显示开关 | X0=0，关闭显示<br>X0=1，开启显示 |
| 2 | 8D | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 电荷泵设置 | A2=0，关闭电荷泵 |
| | A[7:0] | * | * | 0 | 1 | 0 | A2 | 0 | 0 | | A2=1，开启电荷泵 |
| 3 | B0~B7 | 1 | 0 | 1 | 1 | 0 | X2 | X1 | X0 | 设置页地址 | X[2:0]：0~7 对应页 0~7 |
| 4 | 00~0F | 0 | 0 | 0 | 0 | X3 | X2 | X1 | X0 | 设置列地址（低四位） | 设置 8 位起始列地址的低四位 |
| 5 | 10~1F | 0 | 0 | 0 | 1 | X3 | X2 | X1 | X0 | 设置列地址（高四位） | 设置 8 位起始列地址的高四位 |

图 4.13.11 SSD1309 常用命令表

第一个命令为 0X81，用于设置对比度的，这个命令包含了两个字节，第一个 0X81 为命令，随后发送的一个字节为要设置的对比度的值。这个值设置得越大屏幕就越亮。

第二个命令为 OXAE/OXAF。0XAE 为关闭显示命令；0XAF 为开启显示命令。

第三个命令为 0X8D，该指令也包含 2 个字节，第一个为命令字，第二个为设置值，第二个字节的 BIT2 表示电荷泵的开关状态，该位为 1，则开启电荷泵，为 0 则关闭。在模块初始化的时候，这个必须要开启，否则看不到屏幕显示。

第四个命令为 0XBO~B7，该命令用于设置页地址，其低三位的值对应着 GRAM 的页地址。

第五个指令为 0X00~0XOF，该指令用于设置显示时的起始列地址低四位。第六个指令为 OX10~0X1F，该指令用于设置显示时的起始列地址高四位。

其他命令，我们就不在这里一一介绍了，大家可以参考 SSD1309 datasheet 的第 27 页。从这页开始，对 SSD1309 的指令有详细的介绍。

（5）OLED 初始化指令

在使用 OLED 进行显示时，需要对 OLED 参数配置，其实质就是对 SSD1309 的初始化，SSD1309 典型的初始化框图，如图 4.13.12 所示：

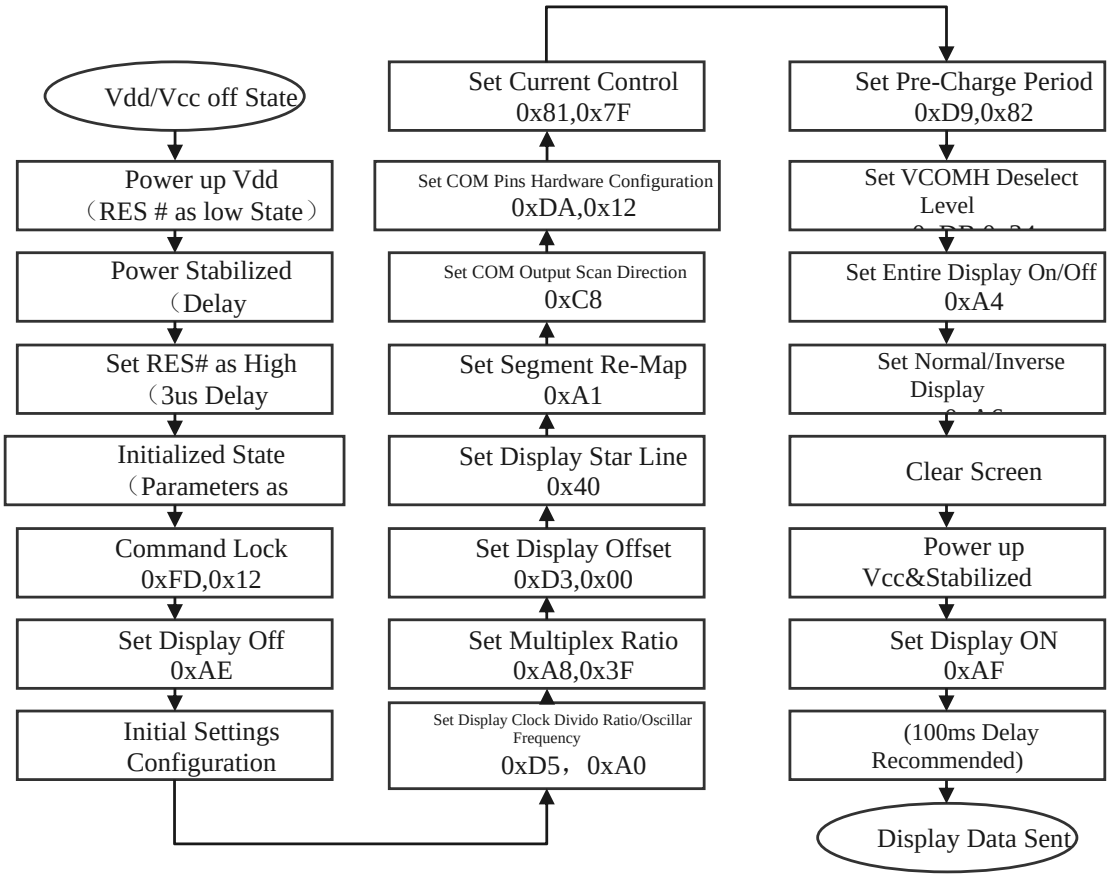



图 4.13.12 SSD1309 软件初始化流程

整个过程比较复杂，主要是 SSD1309 的初始化序列，这部分代码，我是直接采用厂家推荐的初始化代码，没有做相关参数调整，读者若有其它需求时可细致研究参数意义，在进行相关修订。

2. 字符取模软件的使用

软件包下载链接：https://pan.baidu.com/s/1m9DqsO26NBx4KQtYwqTA4A 提取码：bkrc。下载完成后，进行解压。点击图标，进入取模软件界面，如图 4.13.13 所示，这里以 PCtoLCD2002 为例：

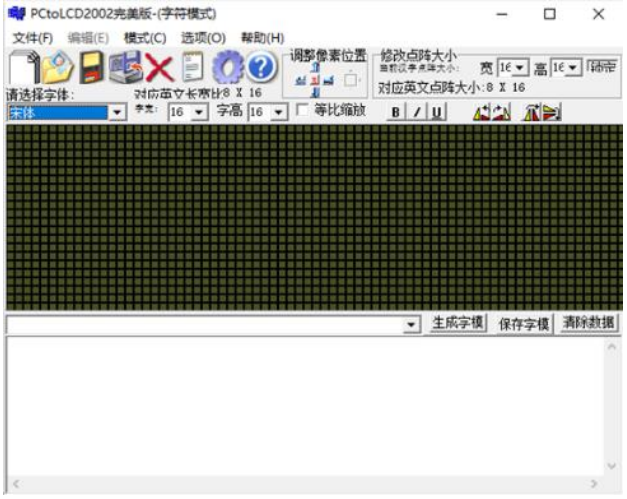



图 4.13.13 字符取模软件

进入取模软件后，在下图选中的区域内，可以进行字体和大小设置。这里我们以宋体，宽和高都为 16 为例进行讲解。结合 OLED 特点，字体大小采用 8 的整数倍最佳。

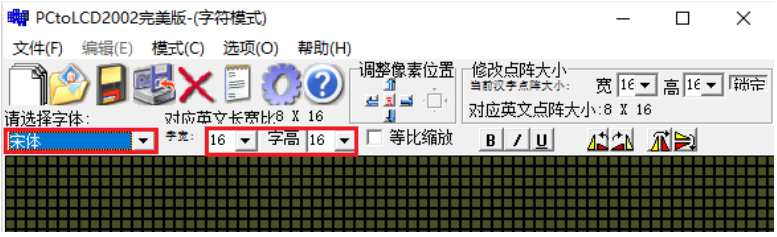



图 4.13.14 字体大小设置

完成字体和大小设置后，点击软件选项栏目，会弹出字模选项配置界面，如下图所示。

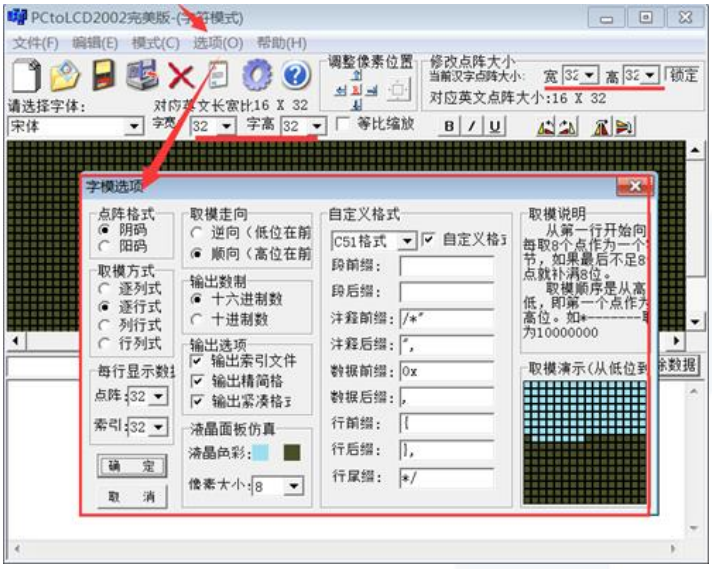



图 4.13.15 字模选项配置界面

在字模选项配置中，我们可以设置点阵格式、取模方向、每行显示数据长度、取

模走向、输出数制和自定义格式。结合项目使用情况，下面给大家讲解每个参数的设置方式：

（1）点阵格式：阴阳码和数码管共阴共阳类似，这里我们使用阴码格式；

（2）取模走向：选项的设置，于发送数据的高低位有关。这里设置为逆向；

（3）取模方式：取模方式有四种，其效果如下图所示，选择不同的模式，会在取模演示进行预演。读者根据自己字符填充算法进行选择，我们设置为列行式；

图 4.13.16 取模方式设置效果图

（4）输出数制：选择十六进制数；

（5）每行显示数据：如下图所示，该数值设定的生成的字模数据每行的长度；

图 4.13.17 每行显示数据长度

（6）自定义格式：将格式设置为 C51，勾选中自定义格式栏。删除行前后缀的{}符号，其它保持默认即可。最终设置效果如下图所示：

图 4.13.18 自定义格式

参数设置完成以后，点击确认即可，输入需要取模汉字，点击生成字模，软件会自动生成相应的数据。将自摸数据添加到 oledfont.h 文件中的 tfont16[]数组内。

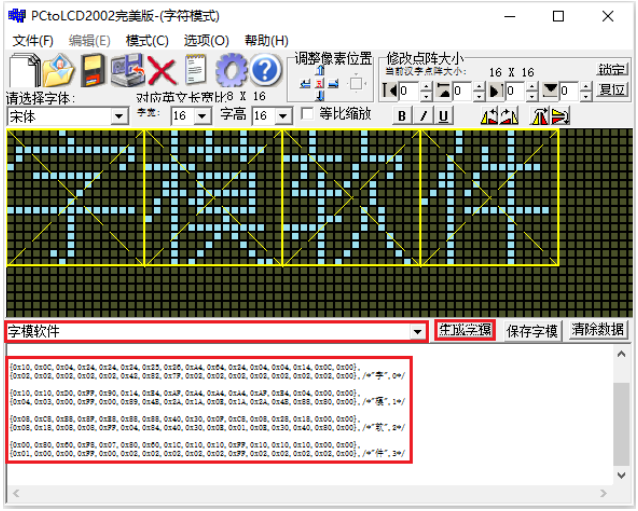



图 4.13.19 生成字模

软件生成的数据拷贝在 tfont16[]中，并无法直接使用，需要进行如下图所示的修改。拷贝的字符信息在数据前面，是在给 tfont16[]这个结构体对象，添加第一个数组的数据。完成字模数据添加后，在调用字符显示函数即可完成验证。

```
"字",
0x10,0x0C,0x04,0x24,0x24,0x24,0x25,0x26,0xA4,0x64,0x24,0x04,0x04,0x14,0x0C,0x00,
0x02,0x02,0x02,0x02,0x02,0x42,0x82,0x7F,0x02,0x02,0x02,0x02,0x02,0x02,0x02,0x00,/*"字",0*/
"模",
0x10,0x10,0xD0,0xFF,0x90,0x14,0xE4,0xAF,0xA4,0xA4,0xA4,0xAF,0xE4,0x04,0x00,0x00,
0x04,0x03,0x00,0xFF,0x00,0x89,0x4B,0x2A,0x1A,0x0E,0x1A,0x2A,0x4B,0x88,0x80,0x00,/*"模",1*/
"软",
0x08,0xC8,0xB8,0x8F,0xE8,0x88,0x88,0x40,0x30,0x0F,0xC8,0x08,0x28,0x18,0x00,0x00,
0x08,0x18,0x08,0x08,0xFF,0x04,0x84,0x40,0x30,0x0E,0x01,0x0E,0x30,0x40,0x80,0x00,/*"软",2*/
"件",
0x00,0x80,0x60,0xF8,0x07,0x80,0x60,0x1C,0x10,0x10,0xFF,0x10,0x10,0x10,0x00,0x00,
0x01,0x00,0x00,0xFF,0x00,0x02,0x02,0x02,0x02,0x02,0xFF,0x02,0x02,0x02,0x02,0x00,/*"件",3*/
```

图 4.13.20 修订文件内容

3. 图片取模软件的使用

OLED 显示屏还可以进行图片显示，需要注意的是 OLED 显示图片范围为 128*64 像素，OLED 高度范围为 0~8，每行 8 个像素点，选择图片时尽量选择 8 为倍数的图片格式。

图片取模软件不支持 PNG 类型图片，进行使用时需要先进行图片内容格式优化，打开图片取模软件后，按图 4.13.21 所示配置进行设置，输出类型为 C 语言数组，选择扫描模式：数据水平，字节垂直。输出灰度为单色，按照输入图像分辨率设置最大宽度和最大高度，取消勾选包含图像头数据，勾选字节内象素数据反序，取消勾选自右至左扫描，取消勾选自底至顶扫描，取消勾选高位在前。

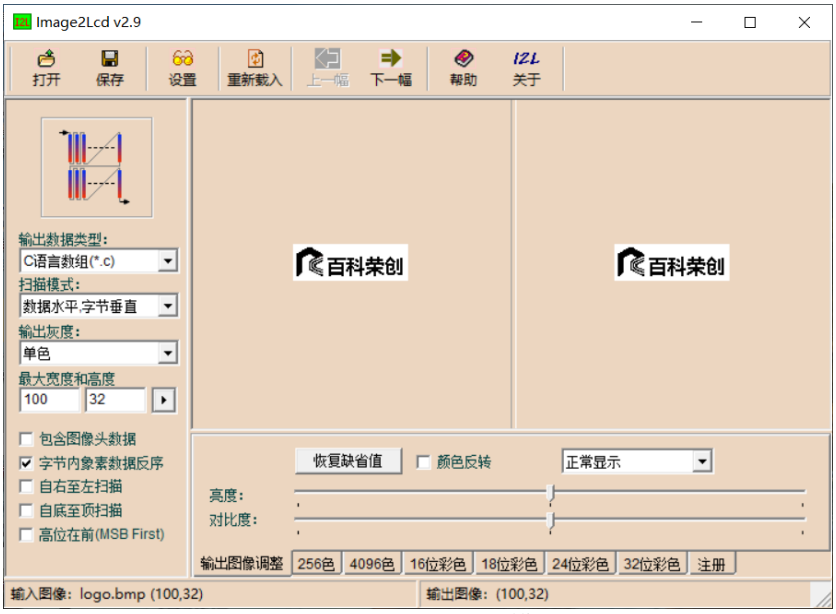



图 4.13.21 图片取模

设置完成后点击重新载入，点击保存，设置保存文件名称为 c 语言文件。

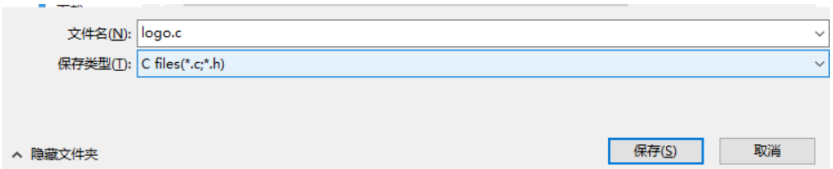



图 4.13.22 文件保存

图 4.13.23 取模文件

将取模后生成的数组文件放置在程序 pic.h 文件中。

main.c | pic.h | oled.c | oledfont.h

```
 9  *输出灰度：单色
10  *不勾选 包含头数据
11  *勾   选 字节内象素数据反序
12  *不勾选 自右至左扫描
13  *不勾选 自底至顶扫描
14  *不勾选 高位在前
15  */
16
17  unsigned char gImage_logo[400] = { /* 0X22,0X01,0X64,0X00,0X20,0X00, */
18  0X00,0X00,0X00,0X00,0X00,0X80,0X80,0XC0,0XE0,0XE0,0XF0,0XF0,0XF8,0XF8,0XF8,0XF8,
19  0XF8,0XF0,0XE0,0XE0,0XC0,0XC0,0X80,0X80,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,
20  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,
21  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,
22  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,
23  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,
24  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0XFF,0XFF,0XFF,0XFF,0XFF,0X0F,0X07,0X07,0X03,
25  0X03,0X01,0X81,0XC1,0X63,0XA7,0XB7,0XDF,0X5F,0X4F,0XAF,0XA7,0XD7,0X56,0X00,0X00,
26  0X00,0X00,0X30,0X30,0X30,0X30,0X30,0X30,0XF0,0XF0,0X30,0X30,0X30,0X30,0X30,0X30,
27  0X30,0X30,0X00,0X30,0X30,0XF0,0XF0,0X30,0X30,0X00,0X60,0XE0,0XC0,0X00,0XF8,0XF8,
28  0X00,0X00,0X00,0X00,0X60,0X60,0X60,0XF8,0XF8,0X60,0X60,0X60,0X60,0XF8,0XF8,0X60,
29  0X60,0X60,0X00,0X00,0X00,0XC0,0XE0,0X70,0X38,0X70,0XE0,0XC0,0X80,0X00,0XE0,0XE0,
30  0X00,0XF0,0XF0,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0XFF,0XFF,0XFF,0XFF,0XFF,
31  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X7F,0X40,0XDF,0XB1,0X6E,0X5E,0XD3,0XA1,0XA1,0XA0,
32  0X40,0X40,0X00,0X00,0X00,0X00,0X00,0XFF,0XFF,0X33,0X33,0X33,0X33,0X33,0X33,0X33,
33  0X33,0X33,0XFF,0XFF,0X00,0X00,0X00,0XC3,0XF3,0X3B,0XFF,0XFF,0X1B,0X33,0XC4,0XC6,0XCC,
34  0XCC,0XE0,0XFF,0XFF,0X60,0X60,0X00,0X07,0X37,0X37,0XB3,0XB3,0XF3,0XF3,0XFF,0XFF,
35  0XF3,0XF3,0XB3,0XB3,0X33,0X37,0X07,0X00,0X03,0XFF,0XFE,0X06,0X66,0X66,0X7E,0XBF,
36  0X80,0X00,0X7F,0X7F,0X00,0XFF,0XFF,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X0F,
37  0X0F,0X07,0X03,0X01,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X01,0X01,
38  0X02,0X02,0X04,0X05,0X0D,0X0D,0X00,0X00,0X00,0X00,0X00,0X0F,0X0F,0X07,0X07,0X07,
39  0X07,0X07,0X07,0X07,0X07,0X07,0X0F,0X0F,0X00,0X00,0X00,0X00,0X00,0X0F,0X0F,0X00,
40  0X00,0X00,0X00,0X00,0X00,0X00,0X00,0X0F,0X0F,0X00,0X00,0X00,0X06,0X07,0X03,0X03,0X01,
41  0X00,0X00,0X0F,0X0F,0X00,0X00,0X01,0X03,0X03,0X06,0X06,0X00,0X00,0X03,0X07,0X06,
42  0X06,0X06,0X06,0X07,0X03,0X00,0X02,0X06,0X06,0X07,0X07,0X00,0X00,0X00,0X00,0X00,
43  };
```

图 4.13.24 取模文件

## 4.13.2 硬件设计

结合项目电路原理图，如下图所示。其中 BS1、2 被直接拉低，BS0 被默认连接到内部 V_SS，这里我们硬件设计的是 4 线 SPI 工作模式。

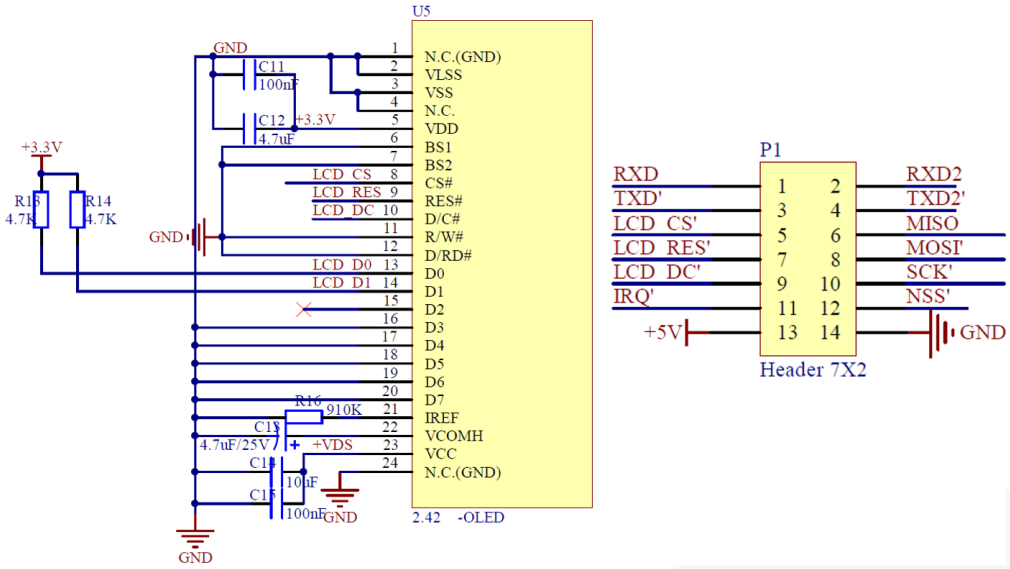



图 4.13.25 硬件原理图

## 4.13.3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将实验部分的驱动代码独立开来，源文件（oled.c）和头文件（oled.h）存放在实验案例根目录中的 HARDWARE/OLED 下。

### 1. 软件设计流程

如下图所示图展示了本实验的软件设计流程图

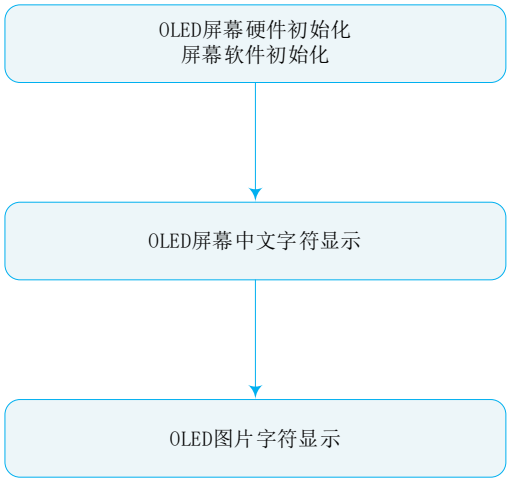



图 4.13.26 软件设计流程图

### 2. 软件代码讲解

（1）OLED 初始化，端口初始化，发送相关软件协议。

代码段 4.13.1 OLED 初始化

```
void OLED_Init(void)
{
    OLED_Hardware_Init();
    OLED_RST_Set();
    Delay_ms(100);
    OLED_RST_Clr();
    Delay_ms(200);
    OLED_RST_Set();
    OLED_WR_Byte(0xAE,OLED_CMD);//--turn off oled panel
    OLED_WR_Byte(0x02,OLED_CMD);//---set low column address
    OLED_WR_Byte(0x10,OLED_CMD);//---set high column address
     OLED_WR_Byte(0x40,OLED_CMD);//--set start line address  Set Mapping RAM Display Start
Line (0x00~0x3F)
    OLED_WR_Byte(0x81,OLED_CMD);//--set contrast control register
    OLED_WR_Byte(0xCF,OLED_CMD); // Set SEG Output Current Brightness
    OLED_WR_Byte(0xA1,OLED_CMD);//--Set SEG/Column Mapping     0xa0 左右反置 0xa1 正常
    OLED_WR_Byte(0xC8,OLED_CMD);//Set COM/Row Scan Direction   0xc0 上下反置 0xc8 正常
```

```
    OLED_WR_Byte(0xA6,OLED_CMD);//--set normal display
    OLED_WR_Byte(0xA8,OLED_CMD);//--set multiplex ratio(1 to 64)
    OLED_WR_Byte(0x3f,OLED_CMD);//--1/64 duty
    OLED_WR_Byte(0x81,OLED_CMD);    //对比度设置
    OLED_WR_Byte(0xfF,OLED_CMD);    //1~255;默认 0X7F (亮度设置,越大越亮)
      OLED_WR_Byte(0xD3,OLED_CMD);//-set  display  offset     Shift  Mapping  RAM  Counter
(0x00~0x3F)
    OLED_WR_Byte(0x00,OLED_CMD);//-not offset
    OLED_WR_Byte(0xd5,OLED_CMD);//--set display clock divide ratio/oscillator frequency
    OLED_WR_Byte(0x80,OLED_CMD);//--set divide ratio, Set Clock as 100 Frames/Sec
    OLED_WR_Byte(0xD9,OLED_CMD);//--set pre-charge period
    OLED_WR_Byte(0xF1,OLED_CMD);//Set Pre-Charge as 15 Clocks & Discharge as 1 Clock
    OLED_WR_Byte(0xDA,OLED_CMD);//--set com pins hardware configuration
    OLED_WR_Byte(0x12,OLED_CMD);
    OLED_WR_Byte(0xDB,OLED_CMD);//--set vcomh
    OLED_WR_Byte(0x40,OLED_CMD);//Set VCOM Deselect Level
    OLED_WR_Byte(0x20,OLED_CMD);//-Set Page Addressing Mode (0x00/0x01/0x02)
    OLED_WR_Byte(0x02,OLED_CMD);//
    OLED_WR_Byte(0x8D,OLED_CMD);//--set Charge Pump enable/disable
    OLED_WR_Byte(0x14,OLED_CMD);//--set(0x10) disable
    OLED_WR_Byte(0xA4,OLED_CMD);// Disable Entire Display On (0xa4/0xa5)
    OLED_WR_Byte(0xA6,OLED_CMD);// Disable Inverse Display On (0xa6/a7)
    OLED_WR_Byte(0xAF,OLED_CMD);//--turn on oled panel
    OLED_WR_Byte(0xAF,OLED_CMD); /*display ON*/
    OLED_Clear();
    OLED_Set_Pos(0,0);
}
```

---

（2）OLED 显示，调用文字显示函数，设置起点坐标和字号大小即可进行文本显示，根据图片像素大小设置图片显示坐标位置，填写显示数组即可进行 OLED 图片显示（注意：OLED 显示 y 轴坐标范围为 0~8 行，每行 8 个像素点）。

代码段 4.13.2 OLED 初始化

---

```
    OLED_Init();                            //OLED 初始化
    OLED_Show_Str(16*1,0,"OLED 显示实验",16);//显示举例,需要文字取模保存于 oledfont.h
   OLED_DrawBMP(14,4,114,8,gImage_logo);    //OLED 显示图片
```

## 4.13.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、通信显示板、14P 排线一根。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |
| ST Link 下载器 | 通信显示板 | 实验导线 |
| | | |

2. 实验硬件接线：使用 14P 排线，连接通信显示板和核心板接口。

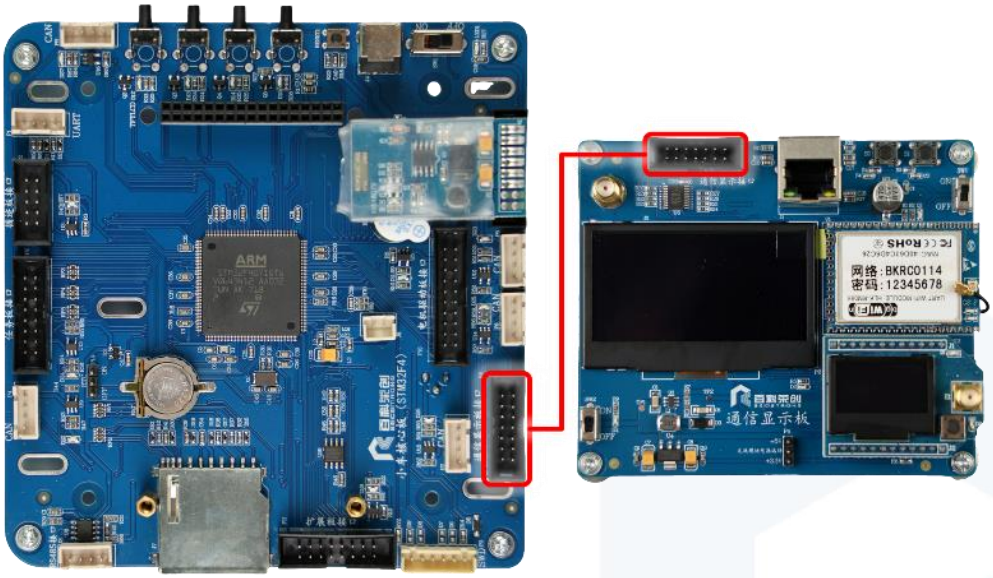



图 4.13.27 实验硬件接线示意图

表 4.13.1 实验接线表

| 小车核心板（STM32F4） | 通信显示板 |
|---|---|
| P9 | P1 |

3. 使用 IDE 将程序编译下载至核心板，观察 OLED 屏幕显示。

## 4.13.5 结果展示

程序运行后，可在通信显示板上看到 OLED 屏幕上显示“OLED 显示实验”字符字样和“百科荣创”图片显示等。

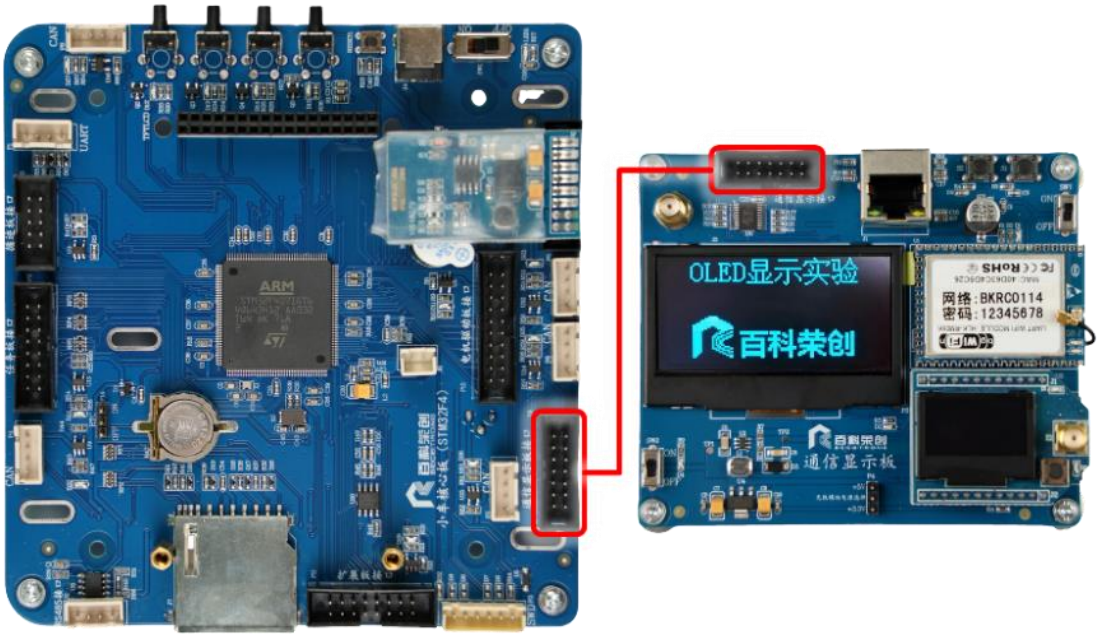



图 4.13.28 实验现象

# 4.14 CAN 总线通信实验

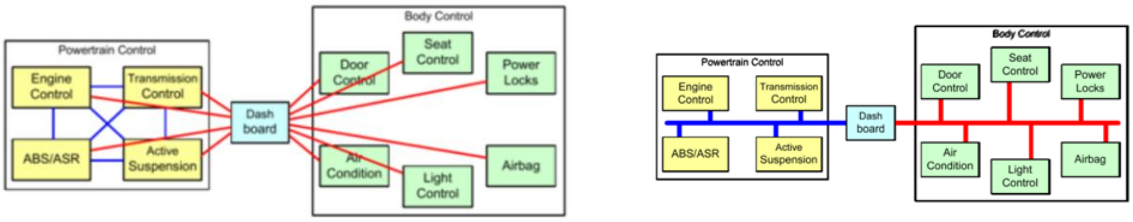
本实验旨在介绍嵌入式中 CAN 总线通信设备的应用。通过完成本实验，读者将深入理解 CAN 通信协议的工作机制，掌握 CAN 总线通信驱动程序的编写方法，并最终实现主机与远程节点之间的可靠通信。

## 4.14.1 基本介绍和应用

### 1．CAN 通信基础

CAN 通信是一种基于串行通信协议的高度可靠性数据传输技术，通过差分信号传输数据，具有抗干扰能力强、数据传输可靠等特点，被广泛应用于汽车、工业自动化、航空航天等领域。

CAN 总线：CAN（Controller Area Network），中文名称为控制器局域网，是一种常用的现场总线标准。它由欧洲汽车工业联合会（ACEA）于 1994 年提出。




（a）点对点通信　　（b）总线通信

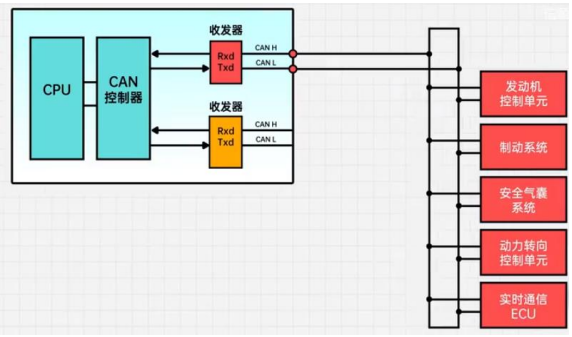
图 4.14.1 通信布线

### 2．CAN 通信应用领域

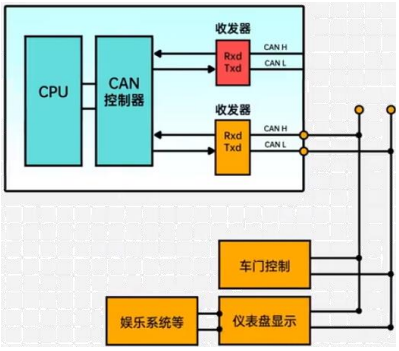
CAN 通信在汽车领域中被广泛应用于车辆网络、车身控制、发动机管理等方面。在工业自动化中，CAN 通信用于 PLC 控制、传感器网络等。此外，航空航天、医疗设备、军事应用等领域也采用 CAN 通信技术来满足其特定的通信需求。

CAN 总线主要应用在汽车电子控制系统中，如发动机管理系统、自动空调控制系统、安全气囊控制系统等。CAN 总线具有可靠性高、抗干扰能力强、传输速率较快等优点。

高速 CAN 总线 CAN2.0B 协议，用于精密控制部分；低速 CAN 总线 CAN2.0A 协议，用于简单设备控制。




（a）高速协议应用




（b）低速协议应用

图 4.14.2 CAN 通信应用

## 4.14.2 总线架构、节点组成和信号传输

### 3．CAN 总线架构

CAN 总线采用多个节点连接在一条共享总线上的方式实现通信。每个节点都包含一个 CAN 控制器，可以通过总线上的仲裁机制实现优先级和发送顺序的决定。总线结构可以是单总线或双总线，具体的结构取决于应用需求。

CAN 总线网络拓扑结构采用总线式结构。这种网络结构简单、成本低，并且采用无源抽头连接，系统可靠性高。通过 CAN 总线连接各个网络节点，形成多主机控制器局域网（CAN）。

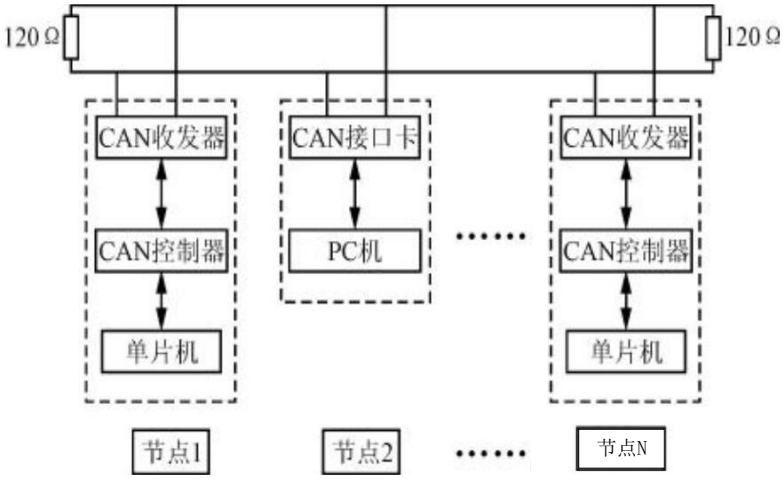



图 4.14.3 CAN 通信总线式结构

信息的传输采用 CAN 通信协议，通过 CAN 控制器来完成。

### 4．CAN 节点组成

每个 CAN 节点需要以下三个部分构成。

（1）中央处理器、微处理器或主处理器

处理主机决定收到的信息的意思以及想要传输的信息。

传感器、驱动器和控制设备可以与主处理器连接。

（2）CAN 控制器；通常是集成单片机的一部分

接收：CAN 控制器将从总线上接收的串位字节存储直到整个消息可用，之后主处理器可以获取这个消息（通常由于 CAN 控制器触发一个中断）。

发送：主处理器发送传递信息到 CAN 控制器，之后当总线空闲时将串位信息传递至总线。

（3）收发器；由 ISO11898-2/3 介质访问单元（MAU）标准定义

接收：把数据流从 CAN 总线层转换成 CAN 控制器可以使用的标准。CAN 控制器通常配有保护电路。

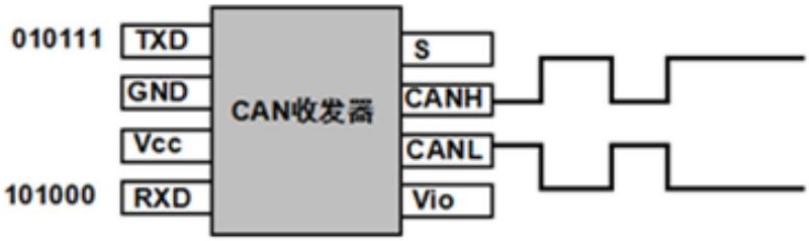
传输：把来自 CAN 控制器的数据流转换至 CAN 总线层。

CAN 总线分高速 CAN 和低速 CAN，收发器也分为高速 CAN 收发器（1Mbps）和低速 CAN 收发器（125Kbps）。低速 CAN 也叫 Fault Tolerance CAN，指的是即使总线上一根线失效，总线依然可以通信。如同串口中的 MAX3232 用作电平转换，CAN 收发器的作用则是把逻辑信号转换为差分信号。

**5．信号传输**

（一）差分信号

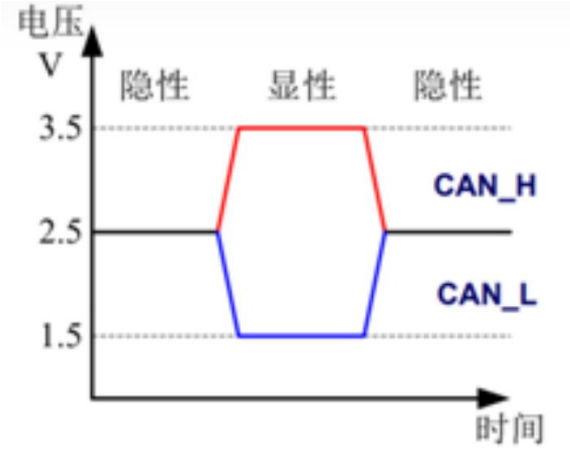
差分信号在 CAN 通信中起着重要作用。差分信号通过同时使用一对相互反向的电压信号来传输数据，有效抵抗外界干扰，并提供较好的信号完整性和抗噪声性能。差分信号的设计和布线需要遵循一定的原则和技巧，以确保可靠的通信连接。




ECU 见通信的 CAN 协议组成局域网通信，单片机+CAN 收发器，输出差分信号，收到干扰压差不变，保证信息不受干扰。

CAN 总线采用差分信号传输，通常情况下只需要两根信号线就可以进行正常的通信。在差分信号中，逻辑 0 和逻辑 1 是用两根差分信号线的电压差来表示。当处于逻

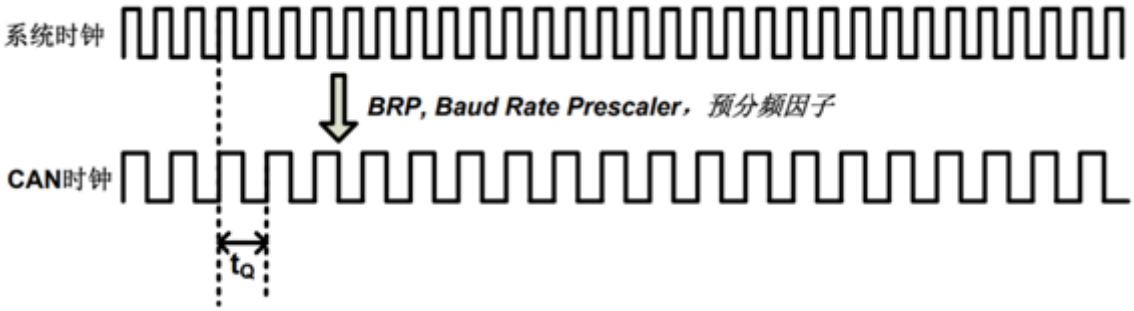
辑 1，CAN_High 和 CAN_Low 的电压差小于 0.5V 时，称为隐性电平（Recessive）；当处于逻辑 0，CAN_High 和 CAN_Low 的电压差大于 0.9V，称为显性电平（Dominant）。



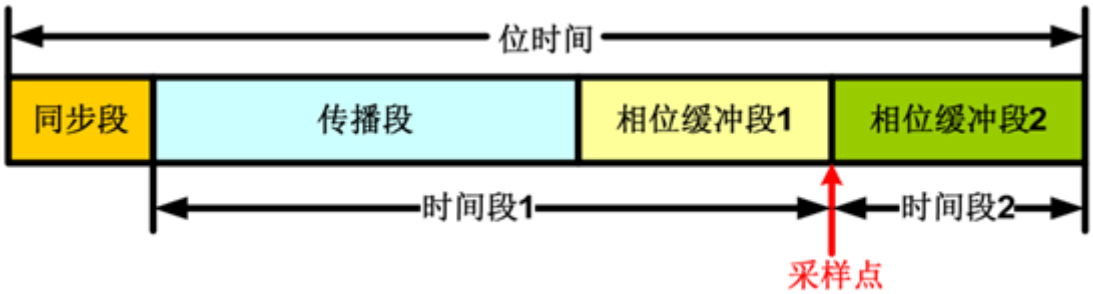

（二）时序同步

CAN 总线使用位同步的方式来确保通信时序，以及对总线的电平进行正确采样。

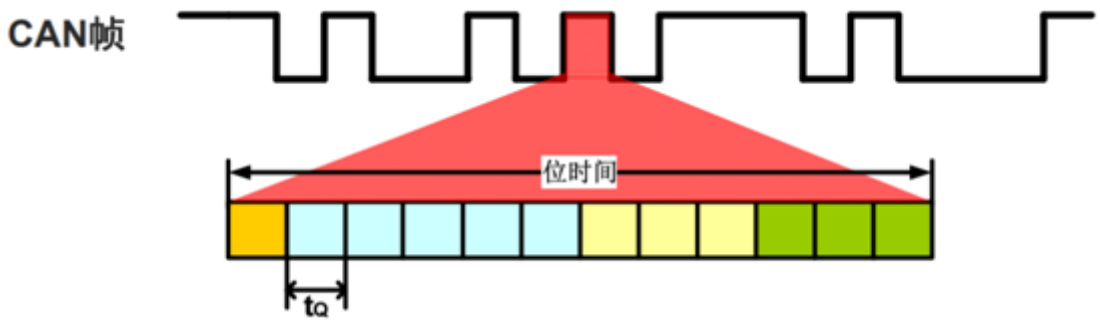
Time Quantum 时间份额 Tq ：CAN 控制器工作的最小时间单位，通常对系统时钟分频得到。




波特率：单位时间内（1s）传输的数据位，公式：1/位时间。举个栗子，系统时钟频率 36MHz，预分频因子为 4，则 CAN 时钟频率 9MHz，则 Tq=1/9M。假设一个 CAN 位包含 10 个 Tq，则一个位周期 T=10Tq，从而波特率为 1/T=0.9MHz.




为了实现位同步，CAN 协议把每一位的时序分解成下图所示的四段。这四段的长度加起来即为一个 CAN 数据位的长度。一个完整的位由 8-25 个 Tq 组成。




同步段（SS，Synchronization Segment）

一个位的输出从同步段开始。若总线的跳变沿被包含在 SS 段的范围之内，则表示节点与总线的时序同步。节点与总线同步时，采样点采集到的总线电平即可被确定为该电平的电位。SS 段的大小为 1Tq。

传播段（PTS，Propagation Time Segment）

用于补偿信号在网络和节点传播的物理延时时间，是总线上输入比较器延时和输出驱动器延时总和的两倍。通常 1-8Tq。

相位缓冲段 1（PBS1， Phase Buffer Segment 1）

主要用于补偿边沿阶段的误差，其时间长度在重新同步时可以加长。初始大小 1-8Tq。

相位缓冲段 2（PBS2，Phase Buffer Segment 2）

也是用于补偿边沿阶段的误差，其时间长度在重新同步时可以缩短。初始大小 2-8Tq。

## 4.14.3 协议介绍、仲裁和通信模式

**6．CAN 协议**

CAN 协议定义了 CAN 帧的格式，包括标准帧和扩展帧。其中，标识符用于区分数据的优先级，数据字段用于传输实际数据，CRC 用于进行差错检测。CAN 2.0 协议和 CAN FD 协议是两种常见的 CAN 协议，各自具有不同的特点和适用性。

介绍 CAN 消息的帧格式，包括标准帧和扩展帧的区别，以及帧的结构和字段。

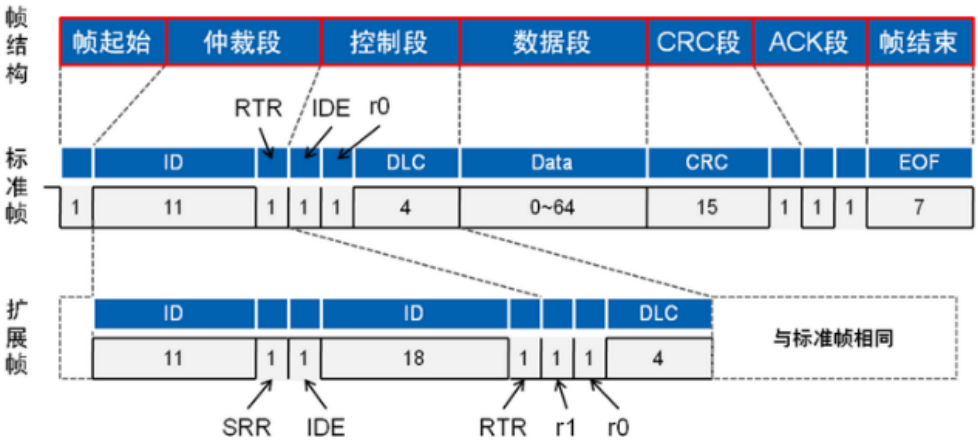



图 4.14.4 CAN 帧结构

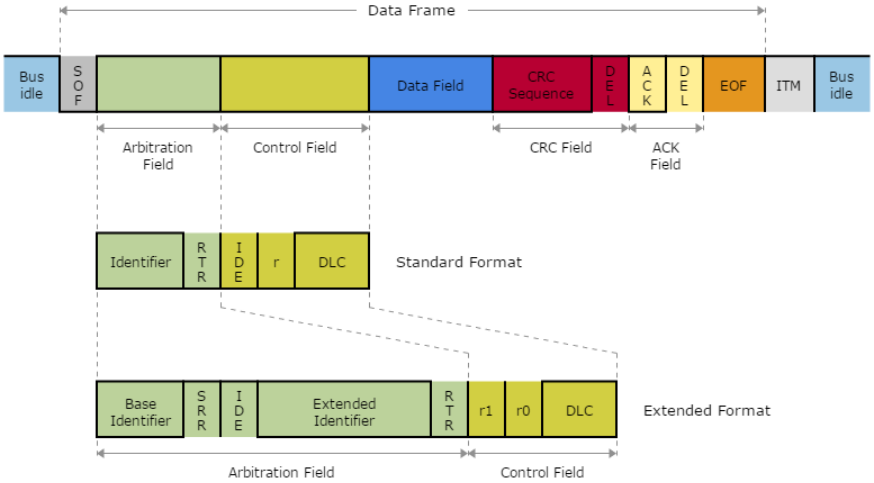



图 4.14.5 CAN 帧结构

**SOF**：表示数据帧开始；（1bit）

**Identifier**：标准格式 11bit，扩展格式 29bit 包括 Base Identifier（11bit）和 Extended Identifier（18bit），该区段标识数据帧的优先级，数值越小，优先级越高；

**RTR**：远程传输请求位，0 时表示为数据帧，1 表示为远程帧，也就是说 RTR=1 时，消息帧的 Data Field 为空；（1bit）

**IDE**：标识符扩展位，0 时表示为标准格式，1 表示为扩展格式；（1bit）

**DLC**：数据长度代码，08 表示数据长度为 08Byte；（4bit）

**Data Field**：数据域；（0~8Byte）

**CRC Sequence**：校验域，校验算法 $G(x)=x^{15}+x^{14}+x^{10}+x^{8}+x^{7}+x^{4}+x^{3}+1$；（15bit）

**DEL**：校验域和应答域的隐性界定符；（1bit）

**ACK**：应答，确认数据是否正常接收，所谓正常接收是指不含填充错误、格式错误、CRC 错误。发送节点将此位为 1，接收节点正常接收数据后将此位置为 0；（1bit）

**SRR**：替代远程请求位，在扩展格式中占位用，必须为 1；（1bit）

**EOF**：连续 7 个隐性位（1）表示帧结束；（7bit）

**ITM**：帧间空间，Intermission(ITM)，又称 Interframe Space(IFS)，连续 3 个隐性位，但它不属于数据帧。帧间空间是用于将数据帧和远程帧与前面的帧分离开来的帧。数据帧和远程帧可通过插入帧间空间将本帧与前面的任何帧（数据帧、遥控帧、错误帧、过载帧）分开。过载帧和错误帧前不能插入帧间空间。

**Bus Idle**：总线空闲。

**7．仲裁机制**

CAN 通信中使用仲裁机制解决多个节点同时发送数据引起的冲突。节点通过比较标识符和优先级来确定发送顺序，遵循位定时和位同步的方式进行仲裁。仲裁机制保证了总线上数据的有序传输，并提供了协调多节点间通信的能力。

解释 CAN 消息中的标识符，它用于区分不同类型的消息和消息的优先级。

CAN 总线数据帧分位标准格式与扩展格式，其组成如下所示。

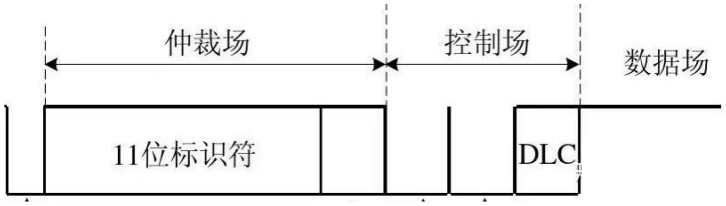



图 4.14.6 标准格式数据帧

图 4.14.7 扩展格式数据帧

由以上数据帧可看出，标准格式与扩展格式的区别在于，不同的位数的标识符，标准格式包含 11 位标识符，扩展格式包含 29 位标识符。

CAN 总线的标识符组成如下所示。

| | 11位标识符 | | | | | | | | | | | | | | | 18位标识符 | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | IDENTIFIER(11BITS) | | | | | | | | | | | SRR | IDE | IDENTIFIER EXTENSION(18 BITS) | | | | | | | | | | | | | | | | | |
| 编码组成 | PRIORITY | | | R | DP | PDU FORMAT(PF) | | | | | | SRR | IDE | PF | | PDU SPECIFIC(PS) | | | | | | | | SOURCE ADDRESS(SA) | | | | | | | |
| 二进制 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 0 | 0 | | | 0 | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 十六进制 | C | | | | | F1 | | | | | | | | | | 1A | | | | | | | | D0 | | | | | | | |
| 扩展帧ID | 0x0CF11AD0 | | | | | | | | | | | | | | | | | | | | | | | | | | | | | | |

图 4.14.8 标识符组成方式

其中：

PRIORITY：此报文的优先级；

R：保留位；

DP：此报文数据页位；

PDU FORMAT(PF)：此报文的数据格式；

PDU SPECIFIC(PS)：此报文的目标地址或者组扩展；

SOURCE ADDRESS(SA)：发送此报文的源地址。

**8．通信模式**

CAN 控制器可以对寄存器 CAN_CTLR 中的 SLEEP 或 INRQ 位进行操作，实现在初始化模式、睡眠模式和正常模式 3 个工作模式下切换。正常模式分为静默模式、环回模式和静默环回模式三种。

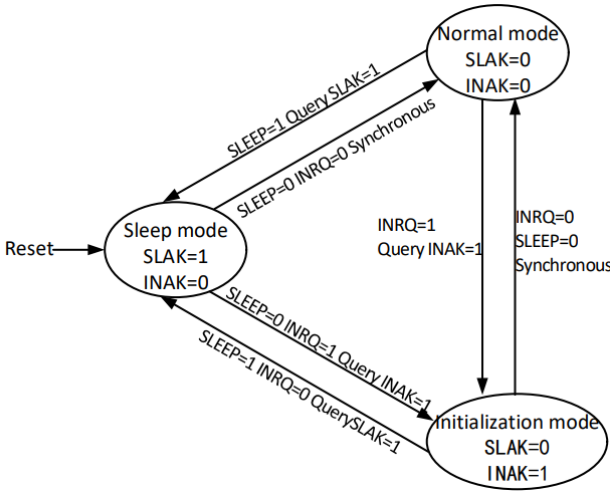



图 4.14.9 CAN 工作模式切换

静默模式

对寄存器 CAN_BTIMR 的 SILM 位置 1，可选择进入静默模式。该模式下，CAN 控制器可以接收，不能对外发送报文，对外总是处于隐性位，可以避免对总线产生影响，但是报文能够被所在节点的控制器所接收。通常静默模式被用于 CAN 总线的状态分析。

环回模式

对寄存器 CAN_BTIMR 的 LBKM 位置 1，可选择进入环回模式。该模式下，CAN 控制器可以对外发送报文，不能接收外部报文，但是发送报文能够被所在节点的控制器所接收，接收过滤机制有效。通常环回模式被用于 CAN 控制器的收发测试。

静默环回模式

对寄存器 CAN_BTIMR 的 SILM 和 LBKM 位置 1，可选择进入静默环回模式。该

模式通常用于 CAN 控制器封闭自测试，在该模式下，对 CAN 总线无影响，RX 引脚与总线断开，TX 引脚置隐性位。

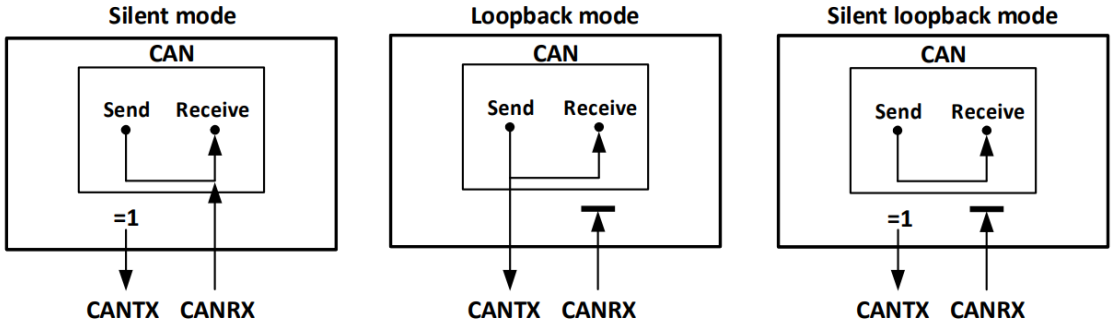



图 4.14.10 CAN 总线的三种测试模式

## 4.14.4 硬件设计

**实验电路分析**

CAN 总线通信的电路原理图如下所示，其中 CANRX 对应 PA11、CANTX 对应 PA12。

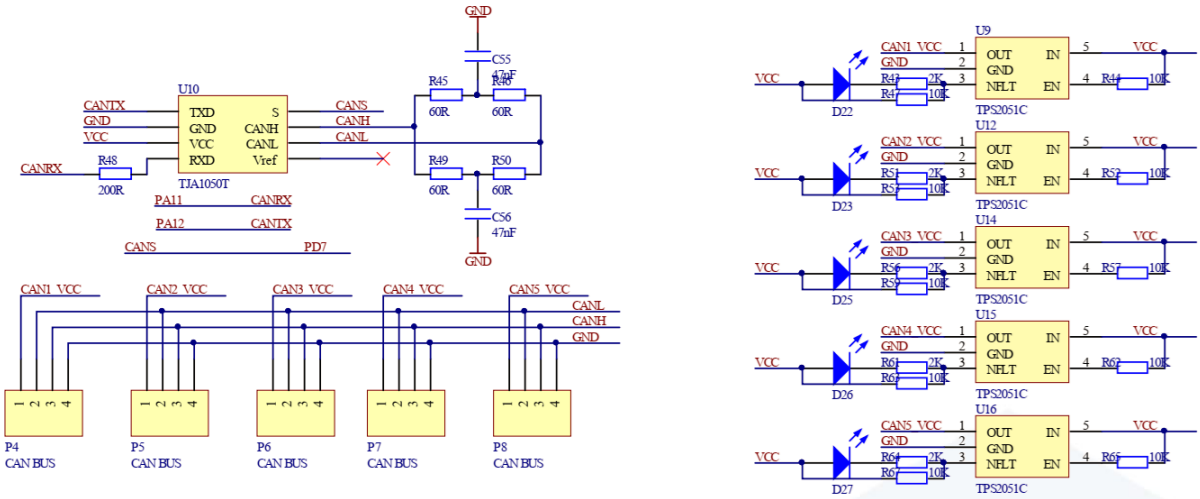
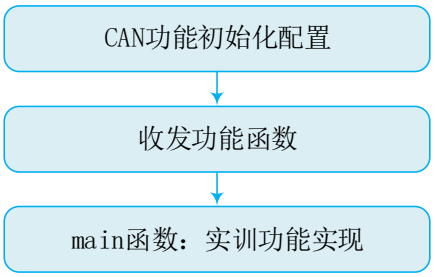



图 4.14.11 CAN 总线通信电路原理图

## 4.14.5 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时为了方便代码的移植和对代码结构的理解，案例中将 CAN 总线通信部分的驱动代码独立出来，源文件（can.c）和头文件（can.h）存放在实验案例根目录中的 HARDWARE/CAN 下。

**1．软件设计流程图**




**2．结构体讲解**

（1）CAN 初始化结构体的内容。

1）Prescaler：设置 CAN 外设的时钟分频，它可控制时间片 Tq 的时间长度，这里设置的值最终会减 1 后再写入 BRP 寄存器位，即前面介绍的 Tq 计算公式：

$Tq=(BRP[9:0]+1)xT_{PCLK}$ 等效于：$Tq=CAN_PrescalerxT_{PCLK}$

2）Model：设置 CAN 的工作模式，可设置为正常模式(CAN_MODE_NORMAL)、回环模式(CAN_MODE_LOOPBACK)、静默模式(CAN_MODE_SILENT)以及回环静默模式(CAN_MODE_SILENT_LOOPBACK)。

3）SJW：可以配置 SJW 的极限长度，即 CAN 重新同步时单次可增加或缩短的最大长度，它可以被配置为 1-4Tq(CAN_SJW_1/2/3/4tq)。

4）BS1：用于设置 CAN 位时序中的 BS1 段的长度，它可以被配置为 1-16 个 Tq 长度。

5）BS2：用于设置 CAN 位时序中的 BS2 段的长度，它可以被配置为 1-8 个 Tq 长度。

SYNC_SEG、BS1 段及 BS2 段的长度加起来即一个数据位的长度，即前面介绍的原来计算公式：

$$T^{1bit}=1Tq+T^{S1}+T^{S2}=1+(TS1[3:0]+1)+(TS2[2:0]+1)$$

等效于：$T_{1bit}=1Tq+CAN_BS1+CAN_BS2$

6）TTCM：用于设置是否使用时间触发功能(ENABLE/DISABLE)，时间触发功能在某些 CAN 标准中会使用到。

7）ABOM：用于设置是否使用自动离线管理(ENABLE/DISABLE)，使用自动离

线管理可以在节点出错离线后适时自动恢复，不需要软件干预。

8）AWUM：用于设置是否使用自动唤醒功能(ENABLE/DISABLE)，使能自动唤醒功能后它会在监测到总线活动后自动唤醒。

9)ABOM：用于设置是否使用自动离线管理功能(ENABLE/DISABLE)，使用自动离线管理可以在出错时离线后适时自动恢复，不需要软件干预。

10）NART：用于设置是否使用自动重传功能(ENABLE/DISABLE)，使用自动重传功能时，会一直发送报文直到成功为止。

11）RFLM：用于设置是否使用锁定接收 FIFO(ENABLE/DISABLE)，锁定接收 FIFO 后，若 FIFO 溢出时会丢弃新数据，否则在 FIFO 溢出时以新数据覆盖旧数据。

12）TXFP：用于设置发送报文的优先级判定方法(ENABLE/DISABLE)，使能时，以报文存入发送邮箱的先后顺序来发送，否则按照报文 ID 的优先级来发送。

代码段 4.14.1 CAN 初始化结构体（stm32f4xx_can.h）

```
/* CAN init structure definition */
typedef struct
{
  uint16_t CAN_Prescaler;    /*配置CAN外设的时钟分频，可设置为1-1024*/
  uint8_t CAN_Mode;          /*配置CAN的工作模式，回环或正常模式*/
  uint8_t CAN_SJW;           /*配置SJW极限值*/
  uint8_t CAN_BS1;           /*配置BS1段长度*/
  uint8_t CAN_BS2;           /*配置BS2段长度*/
  FunctionalState CAN_TTCM; /*是否使能TTCM时间触发功能*/
  FunctionalState CAN_ABOM;  /*是否使能ABOM自动离线管理功能*/
  FunctionalState CAN_AWUM;  /*是否使能AWUM自动唤醒功能*/
  FunctionalState CAN_NART;  /*是否使能NART自动重传功能*/
  FunctionalState CAN_RFLM;  /*是否使能RFLM锁定FIFO功能*/
  FunctionalState CAN_TXFP;  /*配置TXFP报文优先级的判定方法*/
} CAN_InitTypeDef;
```

（2）在发送或接收报文时，需要往发送邮箱中写入报文信息或从接收 FIFO 中读取报文信息，利用外设发送及接收结构体可以方便地完成这样的工作。这些结构体成员，说明如下：

1）StdId：存储的是报文的 11 位标准标识符，范围是 0-0x7FF。

2）ExtId：存储的是报文的 29 位扩展标识符，范围是 0-0x1FFFFFFF。ExtId 与 StdId 这两个成员根据下面的 IDE 位配置，只有一个是有效的。

3）IDE：存储的是扩展标志 IDE 位，当它的值为宏 CAN_ID_STD 时表示本报文是标准帧，使用 StdId 成员存储报文 ID；当它的值为宏 CAN_ID_EXT 时表示本报文是扩展帧，使用 ExtId 成员存储报文 ID。

4）RTR：存储的是报文类型标志 RTR 位，当它的值为宏 CAN_RTR_Data 时表示本报文是数据帧；当它的值为宏 CAN_RTR_Remote 时表示本报文是遥控帧，由于遥控帧没有数据段，所以当报文是遥控帧时，下面的 Data[8]成员的内容是无效的。

5）DLC：存储的是数据帧数据段的长度，它的值的范围是 0-8，当报文是遥控帧时 DLC 值为 0。

6）Data[8]：存储的就是数据帧中数据段的数据。

7）FMI：只存在于接收结构体，它存储了筛选器的编号，表示本报文是经过哪个筛选器存储进接收 FIFO 的，可以用它简化软件处理。

当需要使用 CAN 发送报文时，先定义一个发送类型的结构体，然后把报文的内容按成员赋值到该结构体中，最后函数 CAN_Transmit 把这些内容写入到发送邮箱即可把报文发送出去。

接收报文时，通过检测标志位获知接收 FIFO 的状态，若收到报文，可调用库函数 CAN_Receive 把接收 FIFO 中的内容读取到预先定义的接收类型结构体中，然后再访问该结构体即可利用报文了。

代码段 4.14.2 CAN 发送及接收结构体（stm32f4xx_can.h）

```
/* CAN Tx message structure definition */
typedef struct
{
  uint32_t StdId;   /*存储报文的标准标识符11位，0-0x7FF.*/
  uint32_t ExtId;   /*存储报文的扩展标识符29位，O-Ox1 FFFFFFF.*/
  uint8_t IDE;      /*存储IDE扩展标志*/
  uint8_t RTR;      /*存储RTR远程帧标志*/
  uint8_t DLC;      /*存储报文数据段的长度，0-8*/
  uint8_t Data[8]; /*存储报文数据段的内容*/
} CanTxMsg;
```

```
/* CAN Rx message structure definition  */
typedef struct
{
  uint32_t StdId;  /*存储了报文的标准标识符11位，0-0x7FF.*/
  uint32_t ExtId;  /*存储了报文的扩展标识符29位，0-0x1 FFFFFFF.*/
  uint8_t IDE;     /*存储了IDE扩展标志*/
  uint8_t RTR;     /*存储了RTR远程帧标志*/
  uint8_t DLC;     /*存储了报文数据段的长度，0-8*/
  uint8_t Data[8]; /*存储了报文数据段的内容*/
  uint8_t FMI;     /*存储了本报文是由经过筛选器存储进FIFO的，0-0xFF*/
} CanRxMsg;
```

---

（3）CAN 的筛选器有多种工作模式，利用筛选器结构体可方便配置。

1）FilterIdHigh：FilterIdHigh 成员用于存储要筛选的 ID，若筛选器工作在 32 位模式，它存储的是所筛选 ID 的高 16 位；若筛选器工作在 16 位模式，它存储的就是一个完整的要筛选的 ID。

2）FilterIdLow：类似地，FilterIdLow 成员也是用于存储要筛选的 ID，若筛选器工作在 32 位模式，它存储的是所筛选 ID 的低 16 位；若筛选器工作在 16 位模式，它存储的就是一个完整的要筛选的 ID。

3）FilterMaskIdHigh：FilterMaskIdHigh 存储的内容分两种情况，当筛选器工作在标识符列表模式时，它的功能与 FilterIdHigh 相同，都是存储要筛选的 ID；而当筛选器工作在掩码模式时，它存储的是 FilterIdHigh 成员对应的掩码，与 FilterIdLow 组成一组筛选器。

4）FilterMaskIdLow：类似地，FilterMaskIdLow 存储的内容也分两种情况，当筛选器工作在标识符列表模式时，它的功能与 FilterIdLow 相同，都是存储要筛选的 ID；而当筛选器工作在掩码模式时，它存储的是 FilterIdLow 成员对应的掩码，与 FilterIdLow 组成一组筛选器。

5）FilterFIFOAssignment：用于设置当报文通过筛选器的匹配后，该报文会被存储到哪一个接收 FIFO，它的可选值为 FIFO0 或 FIFO1(宏 CAN_FILTER_FIFO0/1)。

6）FilterNumber：用于设置筛选器的编号，即本过滤器结构体配置的是哪一组筛选器，CAN 一共有 28 个筛选器，所以它的可输入参数范围为 0-27。

7）FilterMode：用于设置筛选器的工作模式，可以设置为列表模式(宏

CAN_FILTERMODE_IDLIST)及掩码模式(宏 CAN_FILTERMODE_IDMASK)。

8）FilterScale：用于设置筛选器的尺度，可以设置为 32 位长(宏 CAN_FILTERSCALE_32BIT)及 16 位长(宏 CAN_FILTERSCALE_16BIT)。

9）FilterActivation：用于设置是否激活这个筛选器(宏 ENABLE/DISABLE)。

配置完这些结构体成员后，调用库把这些参数写入到筛选控制寄存器中，从而使用筛选器。

代码段 4.14.3 CAN 总线通信中断服务函数实现（stm32f4xx_can.h）

```
/* CAN filter init structure definition */
typedef struct
{
  uint16_t CAN_FilterIdHigh;          /*CAN_FXR1寄存器的高16位*/
  uint16_t CAN_FilterIdLow;           /*CAN_FXR1寄存器的低16位*/
  uint16_t CAN_FilterMaskIdHigh;      /*CAN_FXR2寄存器的高16位*/
  uint16_t CAN_FilterMaskIdLow;       /*CAN_FXR2寄存器的低16位*/
  uint16_t CAN_FilterFIFOAssignment; /*设置经过筛选后数据存储到哪个接收FIFO*/
  uint8_t CAN_FilterNumber;           /*筛选器编号，范围0-27*/
  uint8_t CAN_FilterMode;             /*筛选器模式*/
  uint8_t CAN_FilterScale;            /*设置筛选器的尺度*/
  FunctionalState CAN_FilterActivation; /*是否使能本筛选器*/
   } CAN_FilterInitTypeDef;
```

### 3. 软件代码讲解

（1）编写 CAN 的初始化函数，包括 GPIO、工作模式和这段代码主要是把 CAN 的模式设置成了正常工作模式，代码中还把位时序中的 SJW、BS1 和 BS2 段分别设置成了 SJW_1tq、BS1_9tq 和 BS2_8Tq。

代码段 4.14.4 CAN 总线通信引脚初始化（can.c）

```
void CAN_Port_Configure(void)
{
    GPIO_InitTypeDef        GPIO_InitStructure;
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA|RCC_AHB1Periph_GPIOF, ENABLE);

    GPIO_PinAFConfig(GPIOA, GPIO_PinSource11, GPIO_AF_CAN1);
```

```
    GPIO_PinAFConfig(GPIOA, GPIO_PinSource12, GPIO_AF_CAN1);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
    GPIO_InitStructure.GPIO_PuPd  = GPIO_PuPd_UP;
    GPIO_Init(GPIOF, &GPIO_InitStructure);
    GPIO_ResetBits(GPIOF,GPIO_Pin_10);      //硬件 CAN 总线速度
}
```

代码段 4.14.5 CAN 总线通信工作配置（can.c）

```
void CAN_Configure(void)
{
    CAN_InitTypeDef  CAN_InitStructure;
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_CAN1, ENABLE);
    //CAN 总线寄存器初始化
    CAN_DeInit(CAN1);

    //关闭时间触发通讯模式
    CAN_InitStructure.CAN_TTCM = DISABLE;
    //自动离线管理模式使能
    CAN_InitStructure.CAN_ABOM = ENABLE;
    //使用自动唤醒模式
    CAN_InitStructure.CAN_AWUM = DISABLE;
    //禁止报文自动重传 DISABLE-自动重传
    CAN_InitStructure.CAN_NART = DISABLE;
    //接收 FIFO 锁定模式 DISABLE-溢出时新报文会覆盖原有报文
    CAN_InitStructure.CAN_RFLM = DISABLE;
    //发送 FIFO 优先级 DISABLE-优先级取决于报文标示符
    CAN_InitStructure.CAN_TXFP = ENABLE;
    //正常模式
```

```
    CAN_InitStructure.CAN_Mode = CAN_Mode_Normal;//CAN_Mode_LoopBack;//CAN_Mode_Normal;
    //重新同步数据宽度 2 个时间单位
    CAN_InitStructure.CAN_SJW = CAN_SJW_1tq;
    //时间段 1 占用 6 个时间单位
    CAN_InitStructure.CAN_BS1 = CAN_BS1_2tq;
    //时间段 2 占用 3 个时间单位
    CAN_InitStructure.CAN_BS2 = CAN_BS2_3tq;
    //波特率分频器  定义了时间单元的时间长度 42/(1+2+3)/7=1Mbps
    CAN_InitStructure.CAN_Prescaler = 7;
    CAN_Init(CAN1, &CAN_InitStructure);
}
```

---

代码段 4.14.6 CAN 总线通信工作配置（can.c）

---

```
void CAN_Filter_Configure(void)
{
    CAN_FilterInitTypeDef  CAN_FilterInitStructure;

    //选择过滤器号 取值范围 0-13
    CAN_FilterInitStructure.CAN_FilterNumber = 0;
    //标识符屏蔽模式
    CAN_FilterInitStructure.CAN_FilterMode = CAN_FilterMode_IdMask;
    //一个 32bit 的过滤器
    CAN_FilterInitStructure.CAN_FilterScale = CAN_FilterScale_32bit;
    //过滤器  过滤标识符设定   高 16bit / 第一个 16bit 的 过滤器
    CAN_FilterInitStructure.CAN_FilterIdHigh = 0x0000;
    //过滤器  过滤标识符设定   低 16bit / 第二个 16bit 的 过滤器
    CAN_FilterInitStructure.CAN_FilterIdLow = 0x0000;
    //过滤器  屏蔽标识符设定 / 过滤标识符  高 16bit / 第一个 16bit 的过滤器
    CAN_FilterInitStructure.CAN_FilterMaskIdHigh = 0x0000;
    //过滤器  屏蔽标识符设定 / 过滤标识符   低 16bit / 第二个 16bit 的过滤器
    CAN_FilterInitStructure.CAN_FilterMaskIdLow = 0x0000;
    //过滤器关联到 FIFO0
    CAN_FilterInitStructure.CAN_FilterFIFOAssignment = 0;
    //使能过滤器
    CAN_FilterInitStructure.CAN_FilterActivation = ENABLE;
    CAN_FilterInit(&CAN_FilterInitStructure);
    //使能 CAN 通讯中断
    CAN_ITConfig(CAN1, CAN_IT_FMP0, ENABLE);
}
```

（2）设置发送报文

要使用 CAN 发送报文时，需要先定义一个发送报文结构体并向它赋值。这段代码是我们为了方便演示而自己定义的设置报文内容的函数，标准标识符为 0x13，扩展标识符为 0x12，数据段的长度为 8，且数据内容分别为 0-7，实际应用中读者可根据需求设置报文内容。

代码段 4.14.7 CAN 总线通信设置要发送的报文（can.c）

```
void CAN_Send_Msg(uint8_t length, uint8_t *data)
{
    uint8_t i = 0;

    //设定标识符 共 11 位 取值范围 0~0x7FF
    //决定该帧优先级 数值越小 优先级越高
    TxMessage.StdId = 0x200;
    TxMessage.ExtId = 0x000;            //设定扩展标识符 共 18bit 取值范围 0~0x3FFFF
    TxMessage.IDE = CAN_ID_STD;        //标准格式
    TxMessage.RTR = CAN_RTR_DATA;      //数据帧
    //DLC 长度 取值 0~8
    TxMessage.DLC = length;
    for(i = 0;i<length;i++)
    {
        TxMessage.Data[i] = data[i];        //CAN 总线装载数据
    }
    CAN_Transmit(CAN1, &TxMessage);         //CAN 总线发送数据
}
```

（3）设置接收报文

代码段 4.14.8 CAN 总线通信数据接收函数（can.c）

```
void CAN1_RX0_IRQHandler(void)
{
    if(Rx_finish_Flag == 0)
    {
        if( CAN_GetFlagStatus(CAN1,CAN_FLAG_FMP0) == SET)
```

```
        {
            CAN_Receive(CAN1,CAN_FIFO0,&RxMessage);
            Rx_finish_Flag = 1;
        }
    }
    CAN_ClearITPendingBit(CAN1,CAN_FLAG_FMP0);
}
```

---

（4）主函数代码实现

在 main 函数里，调用了 CAN_Device_Configure 函数初始化 CAN 外设，调用 CAN_Send_Msg g 函数发送数据,通过中断服务函数接收数据控制 LED 灯点亮。

代码段 4.14.9 CAN 总线通信中断服务函数实现（main.c）

---

```
int main(void)
{
    Delay_Init();
    LED_Configuration();
    CAN_Device_Configure();
    Timer_Init(83,999);
    LED_Run_times = gt_get() + 500;
    while(1)
    {
        if(gt_get_sub(LED_Run_times) == 0)
        {
            LED_Run_times = gt_get() + 500;
            LED1 = !LED1;
            CAN_Send_Msg(6,CAN_TxBuf);
        }
        if(Rx_finish_Flag == 1)
        {
            if(RxMessage.Data[0] == 0x55)
            {
                if(RxMessage.Data[1] == 0xAA)
                {
                    LED2 = !LED2;
                }
            }
            Rx_finish_Flag = 0;
```

```
        }
    }
}
```

---

## 4.14.6 实验流程

**1. 实验设备准备：**

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、4P 排线根。

| PC | STM32F4 核心板 | ST Link 下载器 | 12V 电源适配器 |
|---|---|---|---|
| | | | |

2. 实验硬件接线：

（2）CAN 通信为总线通信，所以任意连接两个核心板直接的 CAN 接口即可通信，核心板与核心板连接示意图如下所示。

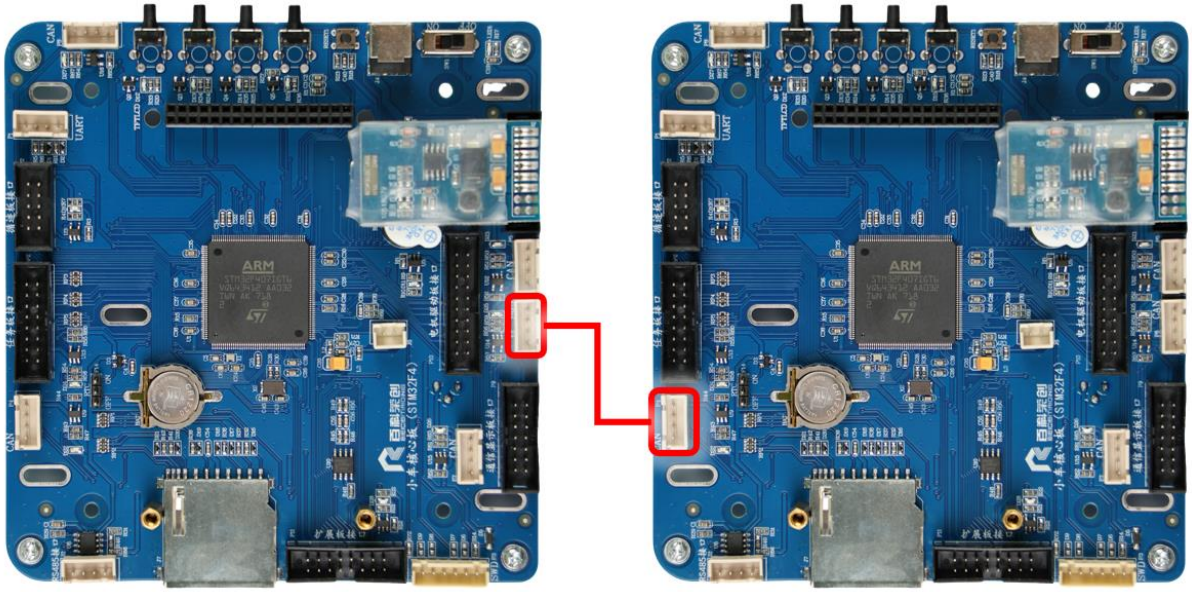


图 4.14.12 实验硬件接线示意图

表 4.14.1 实验接线表

| 小车核心板（STM32F4） | 小车核心板（STM32F4） |
|---|---|
| P4、P5、P6、P7、P8 | P4、P5、P6、P7、P8 |

3. 使用 IDE 将程序编译下载至核心板。

### 4.14.7 实验结果

使用两个核心板用 4P 小白线连接核心板的 CAN 接口，核心板 LED1 灯闪烁，表示正在发送数据，LED2 灯闪烁已经接收到数据。

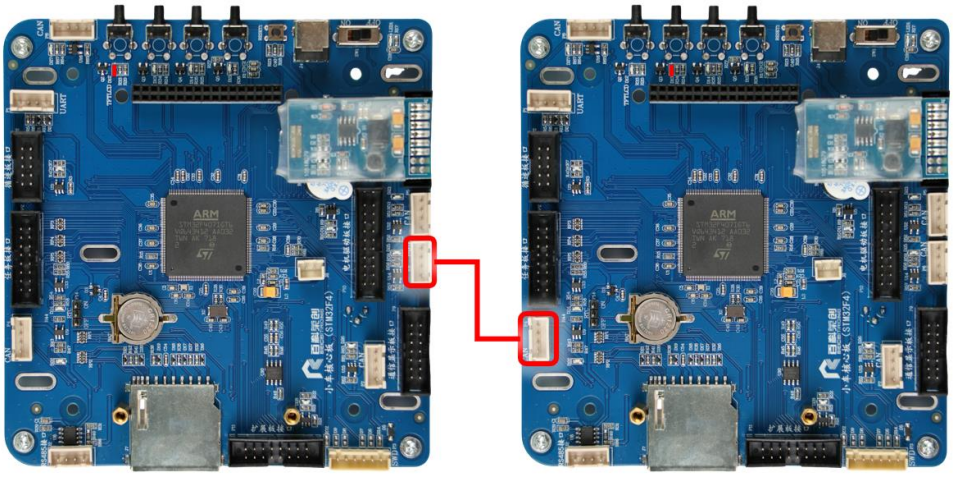


图 4.14.13 CAN 总线通信效果展示图

# 4.15 SD 卡数据读写实验

SD 卡作为一种常用的存储设备，可通过连接 STM32 的 SDIO 接口进行配置和初始化，以实现对 SD 卡的读写功能。

在本实验中，读者将学会 STM32 的 SDIO 接口在 SD 卡中进行数据读写的应用，从而熟悉嵌入式系统中存储设备的使用。

## 4.15.1 SD 卡介绍

**1. SD 卡概念**

SD 存储卡(Secure Digital Memory Card)是一种基于半导体快闪记忆器的新一代记忆设备，由于它体积小、数据传输速度快、可热插拔等优良的特性，被广泛地于便携式装置上使用，例如数码相机、平板电脑和多媒体播放器等。

SD 存储卡的技术是从 MMC 卡(MultiMedia Card）格式上发展而来，在兼容 SD 存储卡基础上发展了 SDIO(SD Input/ Output)卡，此兼容性包括机械、电子、电力、信号和软件，通常将 SD、SDIO 卡俗称 SD 存储卡。

SD 非常小巧，可以嵌入到设备中，在嵌入式开发中，产品需要存储一些容量叫大的文件，就可能会运用到 SD 卡。SD 卡按容量分类，可以分为 3 类：SD 卡、SDHC 卡、SDXC 卡。

图 4.15.1 SD 卡

**2. SD 卡物理结构**

SD 卡有存储单元、存储单元接口、电源检测、卡及接口控制器和接口驱动器 5 个部分。

如图 4.15.2 存储单元是存储数据部件，其中：

（1）存储单元通过存储单元接口与卡及接口控制单元进行数据传输；

（2）电源检测单元保证 SD 卡工作在合适的电压下，如出现掉电或上状态时，它会使控制单元和存储单元接口复位；

（3）卡及接口控制单元控制 SD 卡的运行状态，它包括有 8 个寄存器；

（4）接口驱动器控制 SD 卡引脚的输入输出。

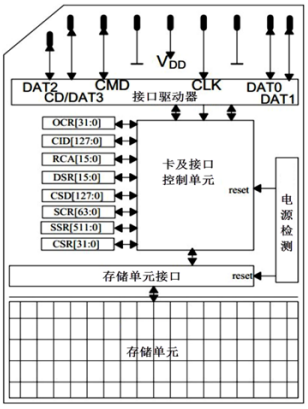



图 4.15.2 SD 卡物理结构

**3. SD 卡寄存器**

SD 卡总共有 8 个寄存器，用于设定或表示 SD 卡信息。这些寄存器只能通过对应的命令访问，对 SD 卡进行控制操作并不是像操作控制器 GPIO 相关寄存器那样一次读写一个寄存器的，它是通过命令来控制，SDIO 定义了 64 个命令，每个命令都有特殊意义，可以实现某一特定功能，SD 卡接收到命令后，根据命令要求对 SD 卡内部寄存器进行修改，程序控制中只需要发送组合命令就可以实现 SD 卡的控制以及读写操作。

如表 4.15.1 所示，列出了 SD 卡的 bit 宽度和寄存器功能。

表 4.15.1 SD 卡寄存器

| 名称 | bit 宽度 | 描述 |
|---|---|---|
| CID | 128 | 卡识别号（Card Identification Number）:用来识别的卡的个体号码（唯一的） |
| RCA | 16 | 相对地址（Relative Card Address）:卡的本地系统地址，初始化时，动态地由卡建议，主机核准。 |
| DSR | 16 | 驱动级寄存器（Driver Stage Register）:配置卡的输出驱动 |
| CSD | 128 | 卡的特定数据（Card Specific Data）:卡的操作条件信息 |
| SCR | 64 | SD 配置寄存器（SD Configuration Register）:SD 卡特殊特性信息 |
| OCR | 32 | 操作条件寄存器（Operation conditions register） |
| SSR | 512 | SD 状态（SD Status）:SD 卡专有特征信息 |
| CSR | 32 | 卡状态（Card Status）:卡状态信息 |

## 4. 15. 2 SDIO 介绍

**1. SDIO 概念**

安全数字输入/输出接口(Secure digital input/output interface，SDIO)，是一种用于连

接外部设备的接口标准，它基于 SD 卡（Secure Digital Card）标准，并扩展了其功能。SDIO 接口允许设备通过 SD 卡插槽进行数字输入和输出操作，以满足各种应用需求，如无线通信、存储扩展、音频/视频流传输等。

SDIO 接口可以与各种设备集成，包括移动电话、便携式媒体播放器、相机、无线局域网（WLAN）适配器、蓝牙适配器、GPS 接收器等。通过使用 SDIO 接口，这些设备可以实现与主机设备之间的高速数据传输和功能扩展，提供更多的功能和灵活性。

**2. SDIO 主要特性**

SD/SDIO MMC 卡主机接口(SDIO)提供 APB2 外设总线与多媒体卡(MMC)、SD 卡、SDIO 卡以及 CE-ATA 设备之间的接口。

SDIO 具有以下特性：

（1）完全兼容多媒体卡系统规范版本 4.2。卡支持三种不同数据总线模式：1 位（默认）、4 位和 8 位。

（2）完全兼容先前版本的多媒体卡（向前兼容性）。

（3）完全兼容 SD 存储卡规范版本 2.0。

（4）完全兼容 SD I/O 卡规范版本 2.0：卡支持两种不同数据总线模式：1 位（默认）和 4 位。

（5）完全支持 CE-ATA 功能（完全符合 CE-ATA 数字协议版本 1.1）。

（6）对于 8 位模式，数据传输高达 50MHz。

（7）数据和命令输出使能信号，控制外部双向驱动程序。

注意：SDIO 不具备兼容 SPI 的通信模式；多媒体卡/SD 总线将卡连接到控制器；当前版本的 SDIO 每次只支持一个 SD/SDIO/MMC4.2 卡，但支持多个 MMC4.1 或之前版本的卡。

**3. SDIO 功能**

SDIO 由两部分组成：

（1）SDIO 适配器模块提供特定于 MMC/SD/SD I/O 卡的所有功能，如时钟生成单元（SDIO_CK）、命令（SDIO_CMD）和数据传输（SDIO_D[7:0]）；

（2）APB2 接口通过访问 SDIO 适配器寄存器，生成中断和 DMA 请求信号。

如图 4.15.3，展示了 SDIO 框图：

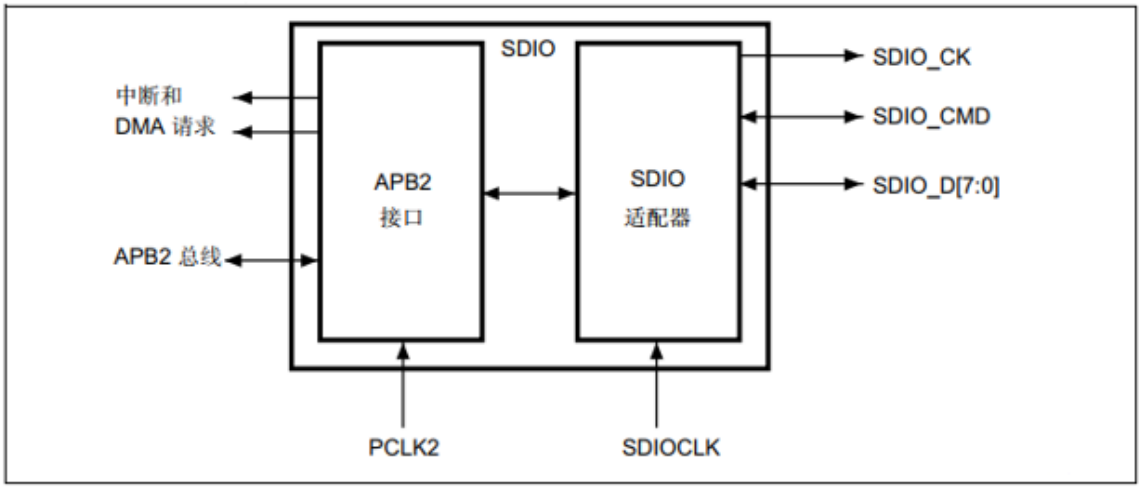



图 4.15.3 SDIO 框图

其中：

（1）在默认情况下，SDIO_D0 用于数据传输。初始化后，主机可以更改数据总线宽度。

如果 SD 或 SD I/O 卡连接到总线，则主机可以将数据传输配置为使用 SDIO_D0 或 SDIO_D[3:0]。所有数据线均以推挽模式运行。

（2）SDIO_CMD 有两种操作模式：

开漏，用于初始化（仅限于 MMC V3.31 或更低版本）；

推挽，用于命令传输（SD/SD I/O 卡 MMC4.2 还将推挽驱动程序用于初始化）。

（3）SDIO_CK 是连接到卡的时钟：每个时钟周期，指令线和数据线都传输一个比特。

SDIO 使用两个时钟信号：

—SDIO 适配器时钟（SDIOCLK），高达 50MHz（与 USB 一起使用时为 48MHz）；

—APB2 总线时钟(PCLK2)。

PCLK2 和 SDIO_CK 时钟频率必须符合以下条件：

$$\text{Frequency(PCLK2)} \geq 3 / 8 \times \text{Frequency(SDIO_CK)}$$

如表 4.15.2，对 SDIO 的几个 I/O 定义进行说明。

表 4.15.2 SDIO I/O 定义

| 引脚 | 方向 | 说明 |
|---|---|---|
| SDIO_CK | 输出 | 多媒体卡/SD/SDIO 卡时钟。该引脚是从主机到卡之间的时钟。 |

| | | |
|---|---|---|
| SDIO_CMD | 双向 | 多媒体卡/SD/SDIO 卡命令。该引脚是双向命令/响应信号。 |
| SDIO_D[7:0] | 双向 | 多媒体卡/SD/SDIO 卡数据。这些引脚是双向数据总线。 |

**4. 卡功能说明**

（1）卡识别模式：在卡识别模式下，主机复位所有卡，验证运行电压范围，识别卡并在总线上为每个卡设置相对卡地址(RCA)。卡识别模式中的所有数据通信都仅使用命令行(CMD)。

（2）智能卡复位：GO_IDLE_STATE 命令(CMD0)是软件复位命令，它会将多媒体卡和 SD 存储器置于空闲状态。IO_RW_DIRECT 命令(CMD52)可复位 SD I/O 卡。在上电或 CMD0 之后，所有卡输出总线驱动程序都处于高阻态状态，并且这些卡将使用默认的相对卡地址(RCA=0x0001)和默认的驱动程序阶段寄存器设置（最低速度，最高驱动电流容量）进行初始化。

（3）工作电压范围验证：所有卡都可以使用规范范围内的任何工作电压与 SDIO 卡主机进行通信。卡上的运行条件寄存器(OCR)定义了支持的最小和最大 VDD 值。

（4）卡识别过程：

对于 SD 卡，识别过程是以时钟速率 Fod 开始，并且 SDIO_CMD 线输出驱动是推挽驱动器，而非开漏。对准过程以如下方式完成：

—激活总线；

—SDIO 卡的主机广播：SD_APP_OP_COND(ACMD41)；

—卡将以其运行条件寄存器的内容作出响应。

—不兼容的卡将被置于无效状态。

—SDIO 卡的主机向所有有效卡广播：ALL_SEND_CID(CMD2)。

—有效卡将发回其唯一的卡识别号(CID)并进入识别状态。

—SDIO 卡主机向某个地址的有效卡发出 SET_RELATIVE_ADDR(CMD3)。这一新地址称为相对卡地址(RCA)；它比 CID 更短，可对卡进行寻址。分配的卡变为待机状态。SDIO 卡主机可以重新发出此命令以更改 RCA。卡的 RCA 是最后分配的值。

—SDIO 卡主机对所有有效卡重复步骤 5 到 7。

注：对于多媒体卡与 SD I/O 卡，请参考《STM32F4xx 中文参考手册》28.4.4 卡识别过程。

（5）块写入和块读取

总线的通信是基于命令和数据传输的。

与 SD/SDIO 存储卡/CE-ATA 器件的数据相互传输在数据块中执行。

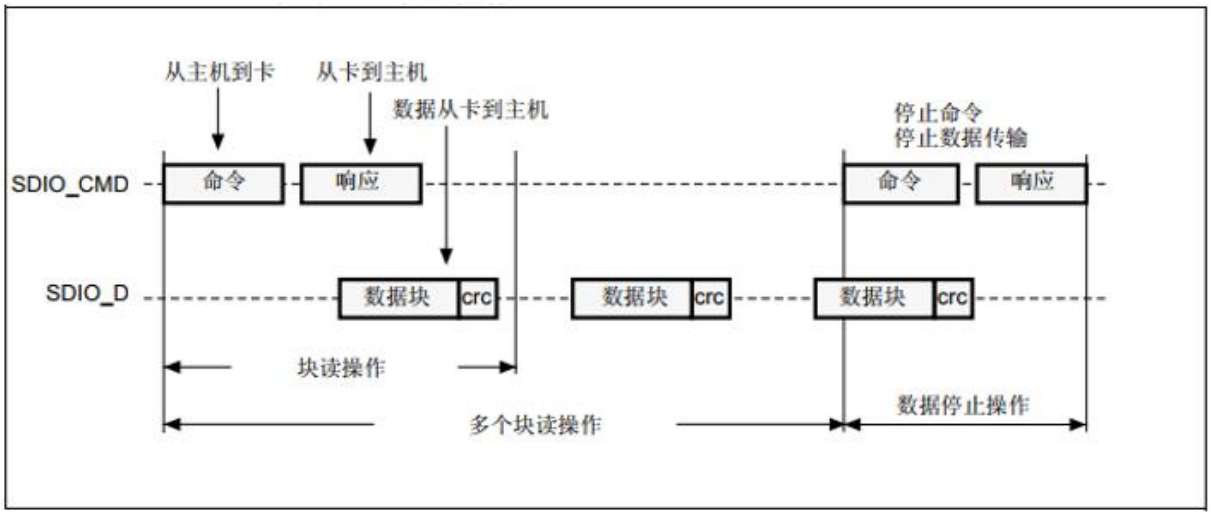



图 4.15.4 SDIO（多个）块读取操作

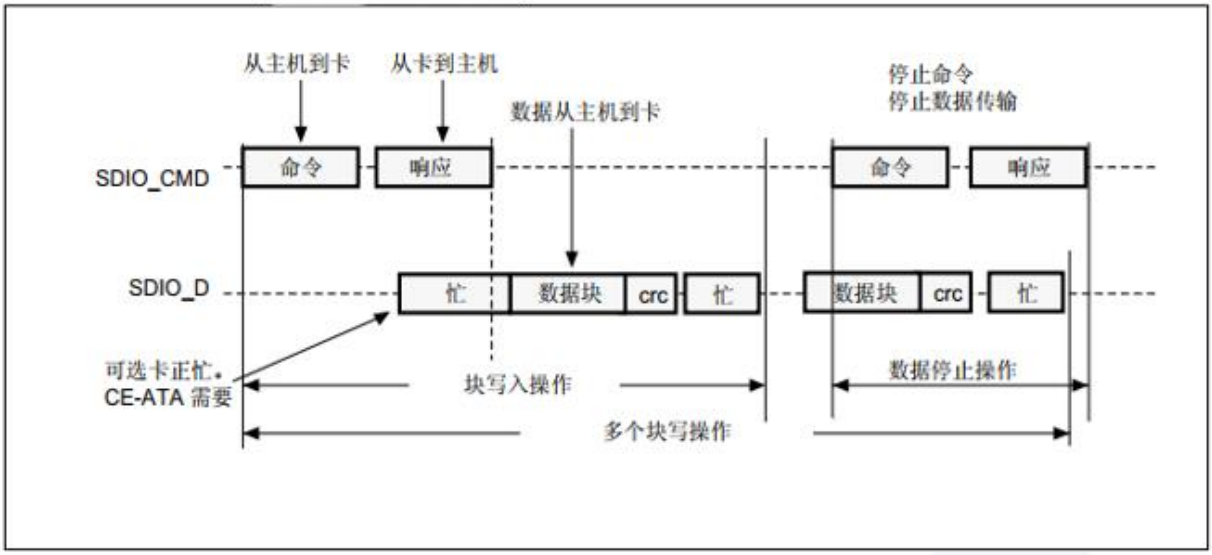



图 4.15.5 SDIO（多个）块写入操作

### 5. SDIO 相关函数讲解

（1）SDIO 初始化结构体定义

代码段 4.15.1 SDIO 初始化结构体定义

```
typedef struct
{
  uint32_t  SDIO_ClockEdge;
  uint32_t  SDIO_ClockBypass;
  uint32_t  SDIO_ClockPowerSave;
  uint32_t  SDIO_BusWide;
  uint32_t  SDIO_HardwareFlowControl;
  uint8_t   SDIO_ClockDiv;
} SDIO_InitTypeDef;
```

关于 SDIO 初始化结构体各个成员的详细描述，如表 4.15.3 所示。

表 4.15.3 SDIO 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| SDIO_ClockEdge | SDIO_ClockEdge_Falling<br>SDIO_ClockEdge_Rising | 指定进行位捕获的时钟边沿（上升沿或下降沿） |
| SDIO_ClockBypass | SDIO_ClockBypass_Disable<br>SDIO_ClockBypass_Enable | 指定是否启用 SDIO 时钟分频旁路 |
| SDIO_Clock PowerSave | SDIO_ClockPowerSave_Disable<br>SDIO_ClockPowerSave_Enable | 指定总线空闲时是否启用或禁用 SDIO 时钟输出 |
| SDIO_BusWide | SDIO_BusWide_1b<br>SDIO_BusWide_4b<br>SDIO_BusWide_8b | 指定 SDIO 总线宽度 |
| SDIO_Hardware FlowControl | SDIO_HardwareFlowControl_Disable<br>SDIO_HardwareFlowControl_Enable | 指定启用或禁用 SDIO 硬件流量控制 |
| SDIO_ClockDiv | 0x00~0xFF | 指定 SDIO 控制器的时钟频率 |

（2）SDIO 初始化函数

表 4.15.4 SDIO 初始化函数

| 接口函数：SDIO_Init | |
|---|---|
| 函数原型 | void SDIO_Init(SDIO_InitTypeDef *SDIO_InitStruct) |

| | |
|---|---|
| 功能描述 | 根据 SDIO_InitStruct 中的指定参数初始化 SDIO 外设。 |
| 入口参数 | SDIO_InitStruct：指向 SDIO_InitTypeDef 结构体的指针，该结构体包含 SDIO 外设的配置信息。 |
| 返回值 | 无 |
| 使用示例 | SDIO_Init(&SDIO_InitStructure); |
| 注意事项 | 无 |

（3）设置控制器的电源状态函数

表 4.15.5 设置控制器的电源状态函数

| 接口函数：SDIO_SetPowerState | |
|---|---|
| 函数原型 | void SDIO_SetPowerState(uint32_t SDIO_PowerState) |
| 功能描述 | 设置控制器的电源状态。 |
| 入口参数 | SDIO_PowerState：电源状态，其参数可以为 SDIO_PowerState_OFF 或 SDIO_PowerState_ON。 |
| 返回值 | 无 |
| 使用示例 | SDIO_SetPowerState(SDIO_PowerState_ON); |
| 注意事项 | 无 |

（4）启用或禁用 SDIO 时钟函数

表 4.15.6 启用或禁用 SDIO 时钟函数

| 接口函数：SDIO_ClockCmd | |
|---|---|
| 函数原型 | void SDIO_ClockCmd(FunctionalState NewState) |
| 功能描述 | 启用或禁用 SDIO 时钟。 |
| 入口参数 | NewState：SDIO 时钟的状态，其参数可以为 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | SDIO_ClockCmd(ENABLE); |

| | |
|---|---|
| 注意事项 | 无 |

（5）获取控制器的电源状态函数

表 4.15.7 获取控制器的电源状态函数

| 接口函数：SDIO_GetPowerState | |
|---|---|
| 函数原型 | uint32_t SDIO_GetPowerState(void) |
| 功能描述 | 获取控制器的电源状态 |
| 入口参数 | 无 |
| 返回值 | 控制器的电源状态，其返回值可以为：<br>0x00：电源关闭；0x02：开机；0x03：电源接通。 |
| 使用示例 | uint32_t power_state = SDIO_GetPowerState(); |
| 注意事项 | 无 |

（6）清除 SDIO 的待处理标记函数

表 4.15.8 清除 SDIO 的待处理标记函数

| 接口函数：SDIO_ClearFlag | |
|---|---|
| 函数原型 | void SDIO_ClearFlag(uint32_t SDIO_FLAG) |
| 功能描述 | 清除 SDIO 指定的待处理标记。 |
| 入口参数 | SDIO_FLAG：指定要清除的标志，其参数可以为：<br>SDIO_FLAG_CCRCFAIL（收到命令响应，CRC 校验失败）、<br>SDIO_FLAG_DCRCFAIL（发送/接收到的数据块，CRC 校验失败）、<br>SDIO_FLAG_CTIMEOUT（命令响应超时）、<br>SDIO_FLAG_DTIMEOUT（数据超时）、<br>SDIO_FLAG_TXUNDERR（发送 FIFO 未满错误）、<br>SDIO_FLAG_RXOVERR（接收 FIFO 超限错误）、<br>SDIO_FLAG_CMDREND（收到命令响应，通过 CRC 校验）、<br>SDIO_FLAG_CMDSENT（已发送命令，无需响应）、<br>SDIO_FLAG_DATAEN（数据结束，数据计数器 SDIDCOUNT 为零）、<br>SDIO_FLAG_STBITERR（在宽总线模式下，未检测到所有数据信号的起始位）、<br>SDIO_FLAG_DBCKEND（发送/接收数据块，通过 CRC 校验）、<br>SDIO_FLAG_SDIOIT（收到 SD I/O 中断）、<br>SDIO_FLAG_CEATAEND（收到 CMD61 的 CE-ATA 命令完成信号）。 |

| | |
|---|---|
| 返回值 | 无 |
| 使用示例 | SDIO_ClearFlag(SDIO_FLAG_CTIMEOUT); |
| 注意事项 | 无 |

（7）检查 SDIO 标志是否已设置函数

表 4.15.9 检查 SDIO 标志是否已设置函数

| 接口函数：SDIO_GetFlagStatus | |
|---|---|
| 函数原型 | FlagStatus SDIO_GetFlagStatus(uint32_t SDIO_FLAG) |
| 功能描述 | 检查指定的 SDIO 标志是否已设置。 |
| 入口参数 | SDIO_FLAG：指定要检查的标志，其参数可以为：<br>SDIO_FLAG_CCRCFAIL（收到命令响应，CRC 校验失败）、<br>SDIO_FLAG_DCRCFAIL（发送/接收到的数据块，CRC 校验失败）、<br>SDIO_FLAG_CTIMEOUT（命令响应超时）、<br>SDIO_FLAG_DTIMEOUT（数据超时）、<br>SDIO_FLAG_TXUNDERR（发送 FIFO 未满错误）、<br>SDIO_FLAG_RXOVERR（接收 FIFO 超限错误）、<br>SDIO_FLAG_CMDREND（收到命令响应，通过 CRC 校验）、<br>SDIO_FLAG_CMDSENT（已发送命令，无需响应）、<br>SDIO_FLAG_DATAEND（数据结束，数据计数器 SDIDCOUNT 为零）、<br>SDIO_FLAG_STBITERR（在宽总线模式下，未检测到所有数据信号的起始位）、<br>SDIO_FLAG_DBCKEND（发送/接收数据块，通过 CRC 校验）、<br>SDIO_FLAG_CMDACT（正在进行命令传输）、<br>SDIO_FLAG_TXACT（数据传输中）、<br>SDIO_FLAG_RXACT（数据接收中）、<br>SDIO_FLAG_TXFIFOHE（传输 FIFO 半空）、<br>SDIO_FLAG_RXFIFOHF（接收 FIFO 半满）、<br>SDIO_FLAG_TXFIFOF（发送 FIFO 满）、<br>SDIO_FLAG_RXFIFOF（接收 FIFO 满）、<br>SDIO_FLAG_TXFIFOE（发送 FIFO 空）、<br>SDIO_FLAG_RXFIFOE（接收 FIFO 空）、<br>SDIO_FLAG_TXDAVL（发送 FIFO 中的可用数据）、<br>SDIO_FLAG_RXDAVL（接收 FIFO 中的可用数据）、<br>SDIO_FLAG_SDIOIT（收到 SDI/O 中断）、<br>SDIO_FLAG_CEATAEND（接收到 CMD61 的 CE-ATA 命令完成信号）。 |
| 返回值 | SDIO_FLAG 的状态，返回值可以为 SET 或 RESET。 |

| | |
|---|---|
| 使用示例 | FlagStatus status = SDIO_GetFlagStatus(SDIO_FLAG_CCRCFAIL); |
| 注意事项 | 无 |

（8）SDIO 命令初始化结构体定义

代码段 4.15.2 SDIO 命令初始化结构体定义

```
typedef struct
{
  uint32_t  SDIO_Argument;
  uint32_t  SDIO_CmdIndex;
  uint32_t  SDIO_Response;
  uint32_t  SDIO_Wait;
  uint32_t  SDIO_CPSM;
} SDIO_CmdInitTypeDef;
```

关于 SDIO 命令初始化结构体各个成员的详细描述，如表 4.15.10 所示。

表 4.15.10 SDIO 命令初始化结构体成员

| 结构体成员名称 | 结构体成员参数(即可选的参数) | 结构体成员说明 |
|---|---|---|
| SDIO_Argument | - | 指定作为命令信息的一部分发送到卡的 SDIO 命令参数。如果命令中包含参数，则必须在将命令写入命令寄存器之前将其加载到该寄存器中 |
| SDIO_CmdIndex | 必须小于 0x40 | 指定 SDIO 命令索引 |
| SDIO_Response | SDIO_Response_Long<br>SDIO_Response_No<br>SDIO_Response_Short | 指定 SDIO 响应类型 |
| SDIO_Wait | SDIO_Wait_IT<br>SDIO_Wait_No<br>SDIO_Wait_Pend | 指定是启用还是禁用 SDIO 等待中断请求 |
| SDIO_CPSM | SDIO_CPSM_Disable<br>SDIO_CPSM_Enable | 指定是启用还是禁用 SDIO 命令路径状态机(CPSM) |

（9）初始化 SDIO 命令并发送命令函数

表 4.15.11 初始化 SDIO 命令并发送命令函数

| 接口函数：SDIO_SendCommand | |
|---|---|
| 函数原型 | void SDIO_SendCommand(SDIO_CmdInitTypeDef *SDIO_CmdInitStruct) |
| 功能描述 | 根据 SDIO_CmdInitStruct 中指定的参数初始化 SDIO 命令并发送命令。 |
| 入口参数 | SDIO_CmdInitStruct：指向 SDIO_CmdInitTypeDef 结构体的指针，该结构体包含 SDIO 命令的配置信息。 |
| 返回值 | 无 |
| 使用示例 | SDIO_SendCommand(&SDIO_CmdInitStructure); |
| 注意事项 | 无 |

（10）SDIO 数据初始化结构体定义

代码段 4.15.3 SDIO 数据初始化结构体定义

```
typedef struct
{
  uint32_t  SDIO_DataTimeOut;
  uint32_t  SDIO_DataLength;
  uint32_t  SDIO_DataBlockSize;
  uint32_t  SDIO_TransferDir;
  uint32_t  SDIO_TransferMode;
  uint32_t  SDIO_DPSM;
} SDIO_DataInitTypeDef;
```

关于 SDIO 数据初始化结构体各个成员的详细描述，如表 4.15.12 所示。

表 4.15.12 SDIO 数据初始化结构体成员

| 结构体成员名称 | 结构体成员参数(即可选的参数) | 结构体成员说明 |
|---|---|---|
| SDIO_DataTimeOut | - | 以卡总线时钟周期为单位指定数据超时时间 |
| SDIO_DataLength | - | 指定要传输的数据字节数 |

| | | |
|---|---|---|
| SDIO_DataBlockSize | SDIO_DataBlockSize_xb（x 可以为 1，2，4，8，16，32，64，128，256，512，1024，2048，4096，8192，16384） | 指定数据块传输的数据块大小 |
| SDIO_TransferDir | SDIO_TransferDir_ToCard（写）<br>SDIO_TransferDir_ToSDIO（读） | 指定数据传输方向，即传输是读还是写 |
| SDIO_TransferMode | SDIO_TransferMode_Block（块）<br>SDIO_TransferMode_Stream（流） | 指定数据传输是流模式还是块模式 |
| SDIO_DPSM | SDIO_DPSM_Disable<br>SDIO_DPSM_Enable | 指定启用或禁用 SDIO 数据路径状态机(DPSM) |

（11）初始化 SDIO 数据路径函数

表 4.15.13 初始化 SDIO 数据路径函数

| 接口函数：SDIO_DataConfig | |
|---|---|
| 函数原型 | void SDIO_DataConfig(SDIO_DataInitTypeDef *SDIO_DataInitStruct) |
| 功能描述 | 根据 SDIO_DataInitStruct 中的指定参数初始化 SDIO 数据路径。 |
| 入口参数 | SDIO_DataInitStruct：指向 SDIO_DataInitTypeDef 结构体的指针，该结构体包含 SDIO 命令的配置信息。 |
| 返回值 | 无 |
| 使用示例 | SDIO_DataConfig(&SDIO_DataInitStructure); |
| 注意事项 | 无 |

## 4.15.3 硬件设计

**1. 实验电路分析**

如图 4.15.6 所示，展示了 SD 的电路原理图，其中包含了 SD 所需的 GPIO 引脚。

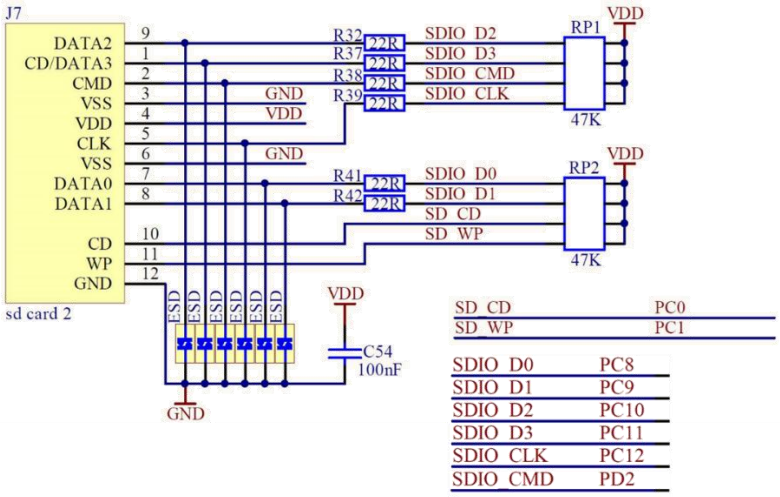



图 4.15.6 SD 电路原理图

## 4. 15. 4 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 SD 卡部分的驱动代码独立开来，源文件（sdio.c）和头文件（sdio.h）存放在实验案例根目录中的 HARDWARE/SD 下。

**1. 软件设计流程**

如图 4.15.7，展示了本实验的软件设计流程图。

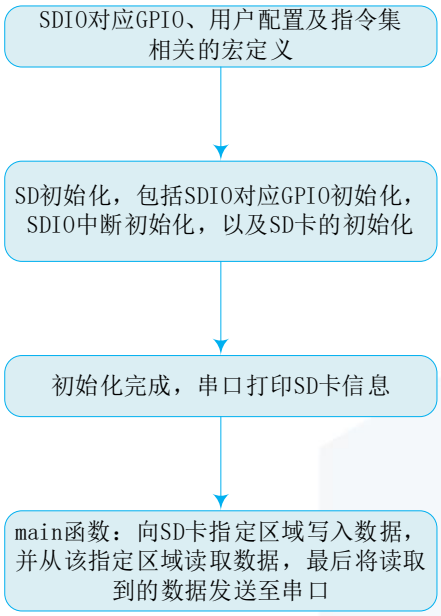



图 4.15.7 软件设计流程图

其中，对于 SD 卡的初始化流程，如图 4.15.8 所示。

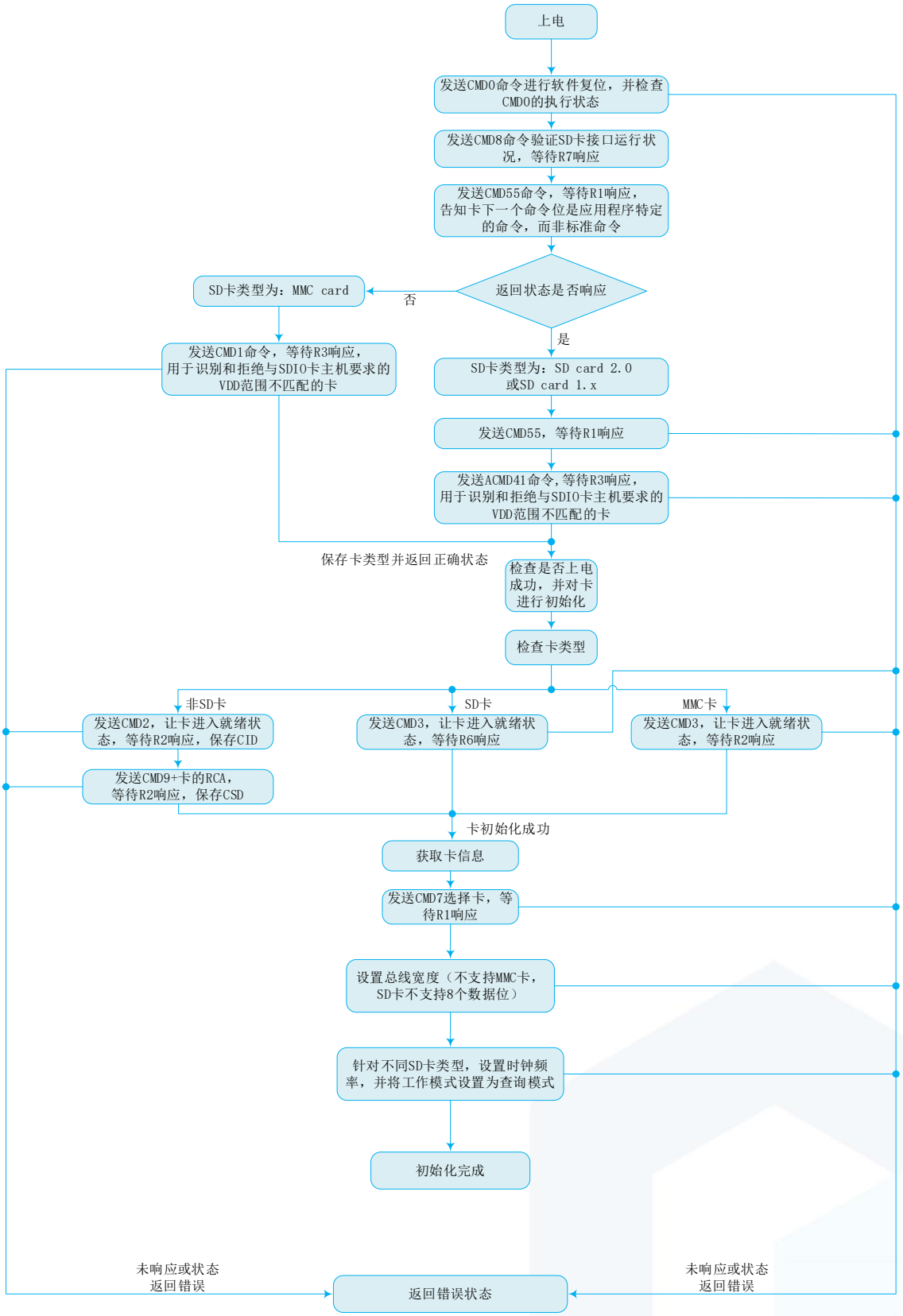



图 4.15.8 SD 卡初始化流程图

**2. 软件代码讲解**

（1）定义 SDIO 对应 GPIO、用户配置及指令集相关的宏，这里只展示 GPIO 相关的宏定义，其余宏定义请参考实验案例根目录下的 HARDWARE/SD/sdio.h。

代码段 4.15.4 SDIO 相关的宏定义（部分）（sdio.h）

```
/*
SDIO 接口            对应 GPIO 引脚
SDIO_D0                PC8
SDIO_D1                PC9
SDIO_D2                PC10
SDIO_D3                PC11
SDIO_CLK               PC12
SDIO_CMD               PD2
*/
//端口
#define SDIO_PORT_1             GPIOC
#define SDIO_PORT_2             GPIOD

//引脚
#define SDIO_D0_PIN             GPIO_Pin_8
#define SDIO_D1_PIN             GPIO_Pin_9
#define SDIO_D2_PIN             GPIO_Pin_10
#define SDIO_D3_PIN             GPIO_Pin_11
#define SDIO_CLK_PIN            GPIO_Pin_12
#define SDIO_CMD_PIN            GPIO_Pin_2

//复用引脚
#define SDIO_D0_PIN_SOURCE      GPIO_PinSource8
#define SDIO_D1_PIN_SOURCE      GPIO_PinSource9
#define SDIO_D2_PIN_SOURCE      GPIO_PinSource10
#define SDIO_D3_PIN_SOURCE      GPIO_PinSource11
#define SDIO_CLK_PIN_SOURCE     GPIO_PinSource12
#define SDIO_CMD_PIN_SOURCE     GPIO_PinSource2

//SDIO 对应 GPIO 时钟定义，时钟挂载在 AHB1 上
#define SDIO_ALL_GPIO_CLOCK     RCC_AHB1Periph_GPIOC | RCC_AHB1Periph_GPIOD

//SDIO 对应 GPIO 时钟使能定义，其为宏函数定义
#define SDIO_GPIO_CLOCK_INIT    RCC_AHB1PeriphClockCmd
```

```
//SDIO 时钟定义，时钟挂载在 APB2 上
#define SDIO_CLOCK              RCC_APB2Periph_SDIO

//SDIO 时钟使能定义，其为宏函数定义
#define SDIO_CLOCK_INIT         RCC_APB2PeriphClockCmd

//SDIO 时钟复位定义，其为宏函数定义
#define SDIO_CLOCK_RESET        RCC_APB2PeriphResetCmd
……
```

---

（2）SD 初始化，这里只展示关键函数 SD_Init()，其中有关初始化步骤中的具体函数的实现请参考 HARDWARE/SD/sdio.c。

代码段 4.15.5 SD 初始化（部分）（sdio.c）

---

```
/*
功　能：　初始化 SD 卡
参　数：　无
返回值：　错误代码(0,无错误)
*/
SD_Error SD_Init(void)
{
    SD_Error errorstatus = SD_OK;
    uint8_t clkdiv = 0;

    SDIO_Port_Configure();
    SDIO_NVIC_Configure();
    errorstatus = SD_PowerON();                                        //SD 卡上电
    if (errorstatus == SD_OK)errorstatus = SD_InitializeCards();       //初始化 SD 卡
    if (errorstatus == SD_OK)errorstatus = SD_GetCardInfo(&SDCardInfo); //获取卡信息
     if (errorstatus == SD_OK)errorstatus = SD_SelectDeselect((uint32_t)(SDCardInfo.RCA << 16)); //选中 SD 卡
                    if (errorstatus == SD_OK)errorstatus = SD_EnableWideBusOperation(SDIO_BusWide_4b);  //4 位宽度,如果是 MMC 卡,则不能用 4 位模式
    if ((errorstatus == SD_OK) || (SDIO_MULTIMEDIA_CARD == CardType))
    {
        if (SDCardInfo.CardType == SDIO_STD_CAPACITY_SD_CARD_V1_1 || SDCardInfo.CardType == SDIO_STD_CAPACITY_SD_CARD_V2_0)
        {
```

```
            clkdiv = SDIO_TRANSFER_CLK_DIV + 2;                //V1.1/V2.0 卡，设置最高
48/4=12Mhz
        }
         else clkdiv = SDIO_TRANSFER_CLK_DIV;                  //SDHC 等其他卡，设置最高
48/2=24Mhz
        SDIO_Clock_Set(clkdiv);                                //设置时钟频率,SDIO 时钟计算公
式:SDIO_CK 时钟=SDIOCLK/[clkdiv+2];其中,SDIOCLK 固定为 48Mhz
        errorstatus = SD_SetDeviceMode(SD_POLLING_MODE);    //设置为查询模式
    }
    return errorstatus;
}
```

---

（3）初始化成功后，串口打印 SD 卡信息函数实现。

代码段 4.15.6 打印 SD 卡信息函数实现（sdio.c）

---

```
/*
功   能:  通过串口打印 SD 卡相关信息
参   数:  无
返回值:  无
*/
void show_sdcard_info(void)
{
    char SD_InfoTab[1024], Empty_Array[1024];

    switch (SDCardInfo.CardType)
    {
        case SDIO_STD_CAPACITY_SD_CARD_V1_1:
            USART6_Send_String("Card Type:SDSC V1.1\r\n");
            break;
        case SDIO_STD_CAPACITY_SD_CARD_V2_0:
            USART6_Send_String("Card Type:SDSC V2.0\r\n");
            break;
        case SDIO_HIGH_CAPACITY_SD_CARD:
            USART6_Send_String("Card Type:SDHC V2.0\r\n");
            break;
        case SDIO_MULTIMEDIA_CARD:
            USART6_Send_String("Card Type:MMC Card\r\n");
            break;
    }
sprintf(SD_InfoTab,"Card ManufacturerID:%d\r\n",
```

```
                                                SDCardInfo.SD_cid.ManufacturerID); //制造商
ID
    USART6_Send_String(SD_InfoTab);
    memcmp(SD_InfoTab, Empty_Array, 1024);
    sprintf(SD_InfoTab, "Card RCA:%d\r\n", SDCardInfo.RCA);          //卡相对地址
    USART6_Send_String(SD_InfoTab);
    memcmp(SD_InfoTab, Empty_Array, 1024);
     sprintf(SD_InfoTab, "Card Capacity:%d MB\r\n", (uint32_t)(SDCardInfo.CardCapacity >>
20)); //显示容量
    USART6_Send_String(SD_InfoTab);
    memcmp(SD_InfoTab, Empty_Array, 1024);
                              sprintf(SD_InfoTab,"Card              BlockSize:%d\r\n\r\n",
SDCardInfo.CardBlockSize);                //显示块大小
    USART6_Send_String(SD_InfoTab);
    memcmp(SD_InfoTab, Empty_Array, 1024);
}
```

（4）主函数代码实现。需要在 while 循环之前进行必要的配置和初始化，包括了延时和相关外设初始化的流程，而对于 SD 卡初始化，若初始化成功，则会打印 SD 卡相关信息，若初始化失败，则所有 LED 灯闪烁。在 while 循环中则实现了 SD 卡的数据读写功能，并将读取到的数据发送至串口。

如代码段 4.15.7 所示，展示了 SD 卡数据读写函数的实现，对于其中有关数据读写步骤的具体函数的实现请参考 HARDWARE/SD/sdio.c。

代码段 4.15.7 SD 卡数据读写函数实现（部分）（sdio.c）

```
/*
功   能:  读 SD 卡
参   数:  buf        读数据缓存区
          sector     扇区地址
          cnt        扇区个数
返回值:  错误状态;0,正常;其他,错误代码
*/
uint8_t SD_ReadDisk(uint8_t* buf, uint32_t sector, uint8_t cnt)
{
    uint8_t sta = SD_OK;
    long long lsector = sector;
    uint8_t n;
    lsector <<= 9;
```

```
    if ((uint32_t)buf % 4 != 0)
    {
        for (n = 0; n < cnt; n++)
        {
            sta = SD_ReadBlock(SDIO_DATA_BUFFER, lsector + 512 * n, 512);   //单个 sector 的
读操作
            memcpy(buf, SDIO_DATA_BUFFER, 512);
            buf += 512;
        }
    }
    else
    {
        if (cnt == 1)sta = SD_ReadBlock(buf, lsector, 512);                 //单个 sector 的
读操作
    }
    return sta;
}

/*
功   能:  写 SD 卡
参   数:  buf        写数据缓存区
          sector     扇区地址
          cnt        扇区个数
返回值:  错误状态;0,正常;其他,错误代码
*/
uint8_t SD_WriteDisk(uint8_t* buf, uint32_t sector, uint8_t cnt)
{
    uint8_t sta = SD_OK;
    uint8_t n;
    long long lsector = sector;
    lsector <<= 9;
    if ((uint32_t)buf % 4 != 0)
    {
        for (n = 0; n < cnt; n++)
        {
            memcpy(SDIO_DATA_BUFFER, buf, 512);
            sta = SD_WriteBlock(SDIO_DATA_BUFFER, lsector + 512 * n, 512);  //单个 sector 的
写操作
            buf += 512;
        }
    }
```

```
    else
    {
        if (cnt == 1)sta = SD_WriteBlock(buf, lsector, 512);                //单个 sector 的
写操作
    }
    return sta;
}
```

如代码段 4.15.8 所示，展示了主函数代码的实现。

代码段 4.15.8 主函数实现（main.c）

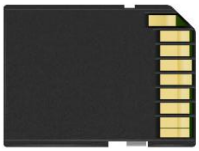
```
int main(void)
{
    Delay_Init();               //延时初始化
    Led_Init();                 //LED 初始化
    USART6_Init(115200);        //串口初始化

    while(SD_Init())            //SD 卡初始化，若初始化失败，则所有 LED 闪烁
    {
        Delay_ms(500);
        LED_ALL_TOGGLE();
    }

    show_sdcard_info();         //打印 SD 相关信息

    while(1)
    {
        Delay_ms(500);
        Delay_ms(500);
        LED1_TOGGLE();
        SD_WriteDisk(TxBuf, 0, 1);          //向 SD 卡指定位置写入数据
        SD_ReadDisk(RxBuf, 0, 1);           //读取 SD 卡指定位置的数据
        USART6_Send_String((char*)RxBuf);   //将读取到的数据发送至串口
    }
}
```

## 4.15.5 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块、SD 卡一张。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |
| ST Link 下载器 | USB 转 TTL 模块 | SD 卡 |
| | |  |

2. 实验硬件接线：

由于之前 USART3 的引脚与本次实验 SD 卡相关的数据引脚冲突，所以改为 USART6，相关引脚接线如下所示：注意：USB 转 TTL 与核心板 TX、RX 交叉连接。

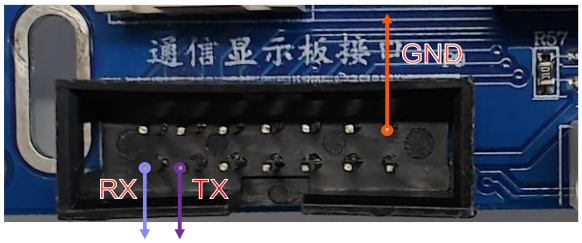



图 4.15.9 串口示意图

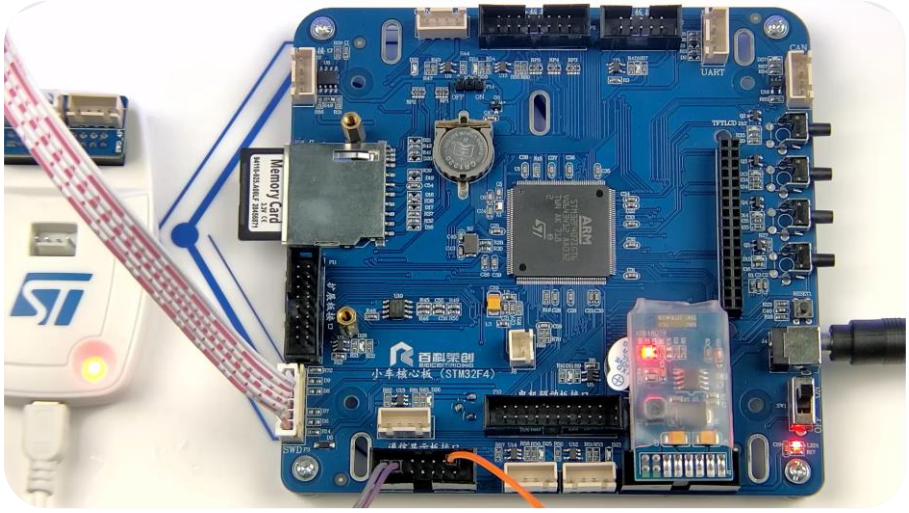


图 4.15.10 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 将 SD 卡插入核心板插槽，并使用 IDE 将程序编译下载至核心板，观察 LED 灯闪烁情况，若全部闪烁，证明 SD 卡初始化出错，若只有 LED1 闪烁，代表初始化成功，同时观察串口调试助手是否打印 SD 卡信息，且打印完 SD 卡信息后，是否会循环打印“Hello world!”。

## 4. 15. 6 结果展示

将 SD 卡插入插槽，并将程序编译下载至核心板后，核心板 LED1 闪烁，同时串口调试助手打印 SD 卡信息，打印完 SD 卡信息后，会循环打印“Hello world!”，这代表 SD 卡读写数据成功。

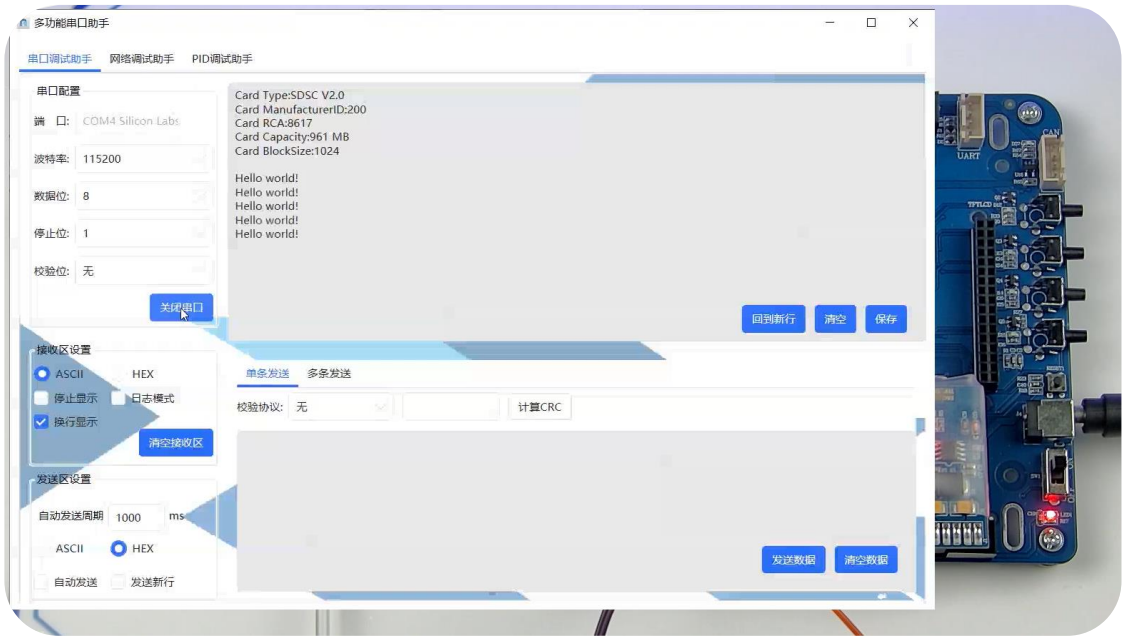



图 4.15.11 实验结果示意图

# 4.16 SD 卡文件系统实验

上一章，我们学习了 SD 卡的使用，不过仅仅是简单的实现读扇区而已，真正要好好应用 SD 卡，必须使用文件系统管理，本章，我们将使用 FATFS 来管理 SD 卡，实现 SD 卡文件的读 写等基本功能。

## 4.16.1 FATFS 简介

FATFS 是一个完全免费开源的 FAT 文件系统模块，专门为小型的嵌入式系统而设计。它完 全用标准 C 语言编写，所以具有良好的硬件平台独立性，可以移植到 8051、PIC、AVR、SH、Z80、H8、ARM 等系列单片机上而只需做简单的修改。它支持 FATl2、FATl6 和 FAT32，支持多个存储媒介；有独立的缓冲区，可以对多个文件进行读 / 写，并特别对 8 位单片机和 16 位 单片机做了优化。

**FATFS 的特点有：**

- Windows 兼容的 FAT 文件系统（支持 FAT12/FAT16/FAT32）
- 与平台无关，移植简单
- 代码量少、效率高
- 多种配置选项：

支持多卷（物理驱动器或分区，最多 10 个卷）

多个 ANSI/OEM 代码页包括 DBCS

支持长文件名、ANSI/OEM 或 Unicode

支持 RTOS

支持多种扇区大小

只读、最小化的 API 和 I/O 缓冲区等

FATFS 的这些特点，加上免费、开源的原则，使得 FATFS 应用非常广泛。FATFS 模块的层次结构如下图所示：

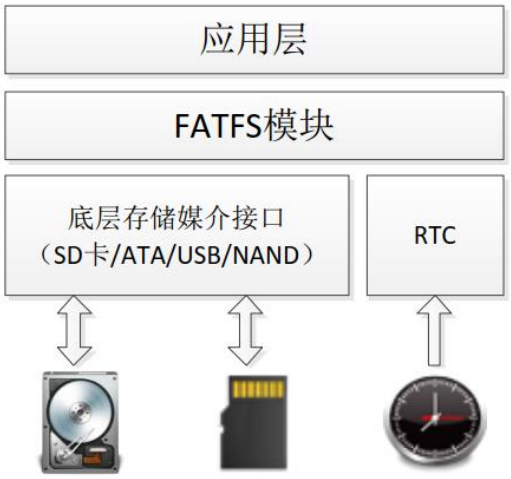



图 4.16.1 FATFS 层次结构图

最顶层是应用层，使用者无需理会 FATFS 的内部结构和复杂的 FAT 协议，只需要调用 FATFS 模块提供给用户的一系列应用接口函数，如 f_open，f_read，f_write 和 f_close 等，就可以像在 PC 上读 / 写文件那样简单。

中间层 FATFS 模块，实现了 FAT 文件读 / 写协议。FATFS 模块提供的是 ff.c 和 ff.h。除非 有必要，使用者一般不用修改，使用时将头文件直接包含进去即可。

需要我们编写移植代码的是 FATFS 模块提供的底层接口，它包括存储媒介读 / 写接口（disk I/O）和供给文件创建修改时间的实时时钟。

FATFS 的源码，大家可以在：http://elm-chan.org/fsw/ff/00index_e.html 这个网站下载到，目前最新版本为 R0.10b。本章我们就使用最新版本的 FATFS 作为介绍，下载最新版本的 FATFS 软件包，解压后可以得到两个文件夹：doc 和 src。doc 里面主要是对 FATFS 的介绍，而 src 里面才是我们需要的源码。

## 4. 16. 2 硬件设计

**1. 实验电路分析**

如图 4.15.6 所示，展示了 SD 的电路原理图，其中包含了 SD 所需的 GPIO 引脚。

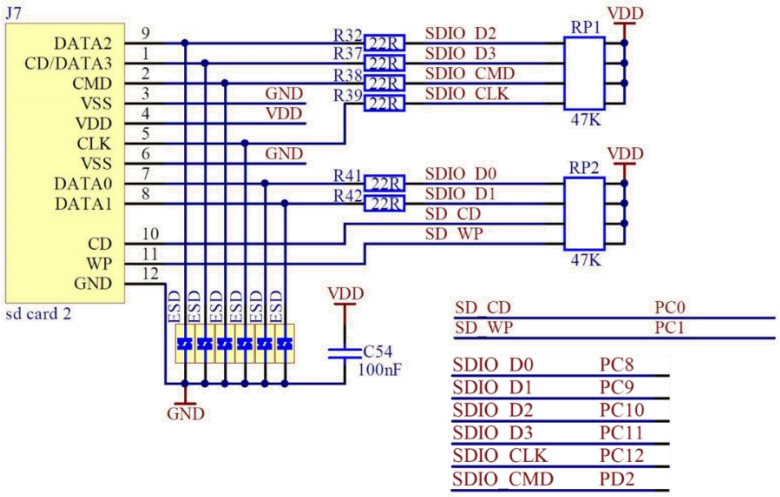



图 4.16.2 SD 电路原理图

## 4. 16. 3 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。

**1. 软件设计流程**

如图 4.15.7，展示了本实验的软件设计流程图。

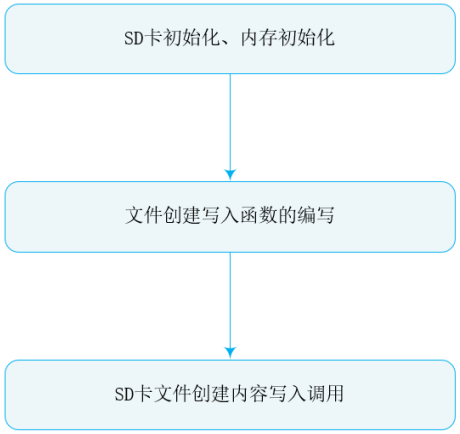



图 4.16.3 软件设计流程图

**2. 软件代码讲解**

（1）SD 初始化在上节实验有所讲解，本实验不在赘述，其中有关初始化步骤中的具体函数的实现请参考 HARDWARE/SD/sdio.c。

（2）SD 卡初始化成功后，SD 卡创建文件及写入数据函数实现。

代码段 4.16.1 创建文件写入函数实现（main.c）

```
FIL MyFile; //文件对象
static int fileSystemTest()
{
```

```
    FIL fil;
    UINT bw;
    FRESULT res = 0;
  /************* 打开文件，如果文件不存在则创建它 ***************************/
    res = f_open(&fil, "0:/2020.txt", FA_CREATE_ALWAYS | FA_WRITE);
    if (res == 0)
    {
        f_write(&fil, "SD 卡文件操作实验", sizeof("SD 卡文件操作实验"), &bw);
        f_close(&fil);//关闭文件
    }
  return 0;
}
```

（3）如代码段 4.15.8 所示，展示了主函数代码的实现。

代码段 4.16.2 主函数实现（main.c）

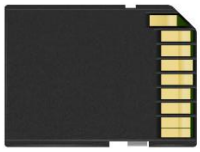
```
int main(void)
{
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);//设置系统中断优先级分组 2
    delay_init(168);     //初始化延时函数
    uart_init(115200);     //初始化串口波特率为 115200
    Led_Init();        //初始化 LED
    my_mem_init(SRAMIN);  //初始化内部内存池
    my_mem_init(SRAMCCM); //初始化 CCM 内存池

    while(SD_Init())//检测不到 SD 卡
    {
        LED_ALL_TOGGLE();
    delay_ms(300);
    }
    exfuns_init();              //为 fatfs 相关变量申请内存
    f_mount(fs[0],"0:",1);          //挂载 SD 卡
    fileSystemTest();

    while(1)
    {
    LED_ALL(1);
    }
}
```

## 4.16.4 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK、串口调试助手。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、USB 转 TTL 模块、SD 卡一张。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |
| **ST Link 下载器** | **USB 转 TTL 模块** | **SD 卡** |
| | |  |

2. 实验硬件接线：

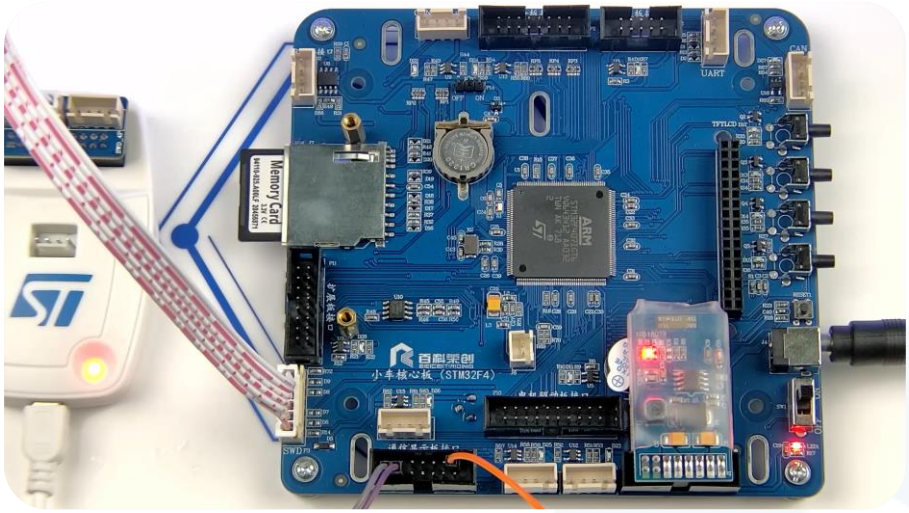



图 4.16.4 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

4. 将 SD 卡插入核心板插槽，并使用 IDE 将程序编译下载至核心板，观察 LED 灯闪烁情况，若全部闪烁，证明 SD 卡初始化出错，若所有 LED 灯点亮，代表初始化成功，SD 卡完成创建文件。

## 4.16.5 结果展示

将 SD 卡插入插槽，并将程序编译下载至核心板后，核心板所有 LED 点亮，，这代表 SD 卡创建文件成功。

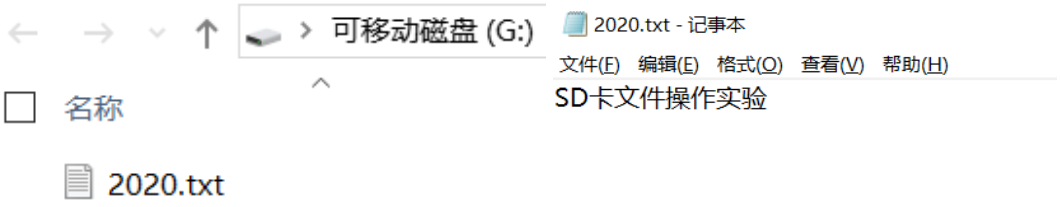



图 4.16.5 实验结果示意图

# 4. 17 FSMC-LCD 显示实验

本实验利用 STM32 微控制器的 FMSC 和 ILI9486 驱动器，实现 LCD 屏幕的高效显示。

其中 STM32 的 FMSC 提供了强大的内存控制功能，可实现快速可靠的数据传输到 LCD 屏幕。ILI9486 驱动器是常用的 LCD 控制器，与 FMSC 无缝集成，用于配置和控制 LCD 屏幕。

通过结合 FMSC 和 ILI9486 驱动器，可以实现快速、精确和高质量的 LCD 屏幕显示，提升读者体验和应用的可靠性。

## 4. 17. 1 TFT LCD 介绍

**1. TFT-LCD 概念**

TFT-LCD，全称为薄膜晶体管液晶显示器（Thin Film Transistor Liquid Crystal Display），是一种液晶显示技术。它采用薄膜晶体管作为每个像素点的开关，通过控制晶体管的状态来实现像素的亮度和颜色变化。

TFT-LCD 广泛应用于电子设备中，如计算机显示器、电视机、智能手机、平板电脑、车载显示器等。

TFT-LCD 相对于其他液晶显示技术的优势在于其高像素密度、色彩准确性和快速响应时间。每个像素点都由一个薄膜晶体管驱动，可以精确地控制每个像素的亮度和颜色，从而呈现出细腻、清晰的图像。快速的响应时间意味着在显示快速移动的图像或视频时，TFT-LCD 可以更好地显示细节和减少残影。

**2. ILI9486L**

（1）概念

ILI9486L 是一款常用的 TFT LCD 驱动器芯片，广泛用于嵌入式系统中驱动液晶显示屏，它支持 262144 种色彩，支持的显示分辨率为 320*480，内部 GRAM 显存为 345600bytes（320*480*18bit）。

（2）功能描述

- **MCU 接口**

ILI9486L 为 8080 系列提供 8 位、9 位、16 位或 18 位并行系统接口；为串行数据输入提供 3 线或 4 线串行系统接口（SPI）。

输入系统接口由外部引脚 IM[2:0]进行选择，每个像素的位正规颜色顺序由 DBI[2:0]位进行选择。

- **MCU 接口选择**

如表 4.17.1 所示，通过设置外部引脚 IM[2:0]，可以选择给定的接口。在实验中使用的是“8080 16 位总线接口”，因此其 IM[2:0]应设置为 010，需要使用数据引脚为 DB[15:0]。

表 4.17.1 ILI9486L MCU 接口选择

| IM2 | IM1 | IM0 | 接口 | 需要使用的数据引脚 |
|---|---|---|---|---|
| 0 | 0 | 0 | 8080 18 位总线接口 | DB[17:0] |
| 0 | 0 | 1 | 8080 9 位总线接口 | DB[8:0] |
| 0 | 1 | 0 | 8080 16 位总线接口 | DB[15:0] |
| 0 | 1 | 1 | 8080 8 位总线接口 | DB[7:0] |
| 1 | 0 | 0 | Prohibited（禁止） | - |
| 1 | 0 | 1 | 3 线 SPI | SDA |
| 1 | 1 | 0 | Prohibited（禁止） | - |
| 1 | 1 | 1 | 4 线 SPI | SDA |

- **8080 系列并行接口**

片选 CSX（低电平有效）用于启用或禁用 ILI9486L 芯片；RESX（低电平有效）用于外部复位信号；WRX 为并行数据写信号；RDX 为并行数据读信号；DB[17:0]为并行数据总线。

MCU 在 WRX 信号的上升沿锁存输入数据。D/CX 是数据/命令选择信号，当 D/CX=1 时，DB[17:0]位显示 RAM 数据或命令参数。当 D/CX=0 时，DB[17:0]位为命令。

MCU 控制器与 LCD 驱动芯片之间的通信可使用 8080 系列双向接口，其选择如表 4.17.2 所示。

表 4.17.2 并行接口控制表

| WRX | RDX | D/CX | 功能 |
|---|---|---|---|
| ↑ | “H” | “L” | 写入命令代码 |
| “H” | ↑ | “H” | 读取内部状态 |
| ↑ | “H” | “H” | 写入参数或显示数据 |
| “H” | ↑ | “H” | 读取参数或显示数据 |

- **8080 系列并行接口写时序**

在写入周期内，WRX 信号从高电平驱动到低电平，然后拉回到高电平。当显示模块在 WRX 上升沿捕捉到来自主机处理器的信息时，主机处理器就会在写入周期内提供信息。当 D/CX 信号被驱动为低电平时，接口上的输入数据将被解释为命令信息。当接口上的数据是 RAM 数据或命令参数时，D/CX 信号也可以被拉到高电平。

注意：WRX 是非同步信号（其可以被停止）。

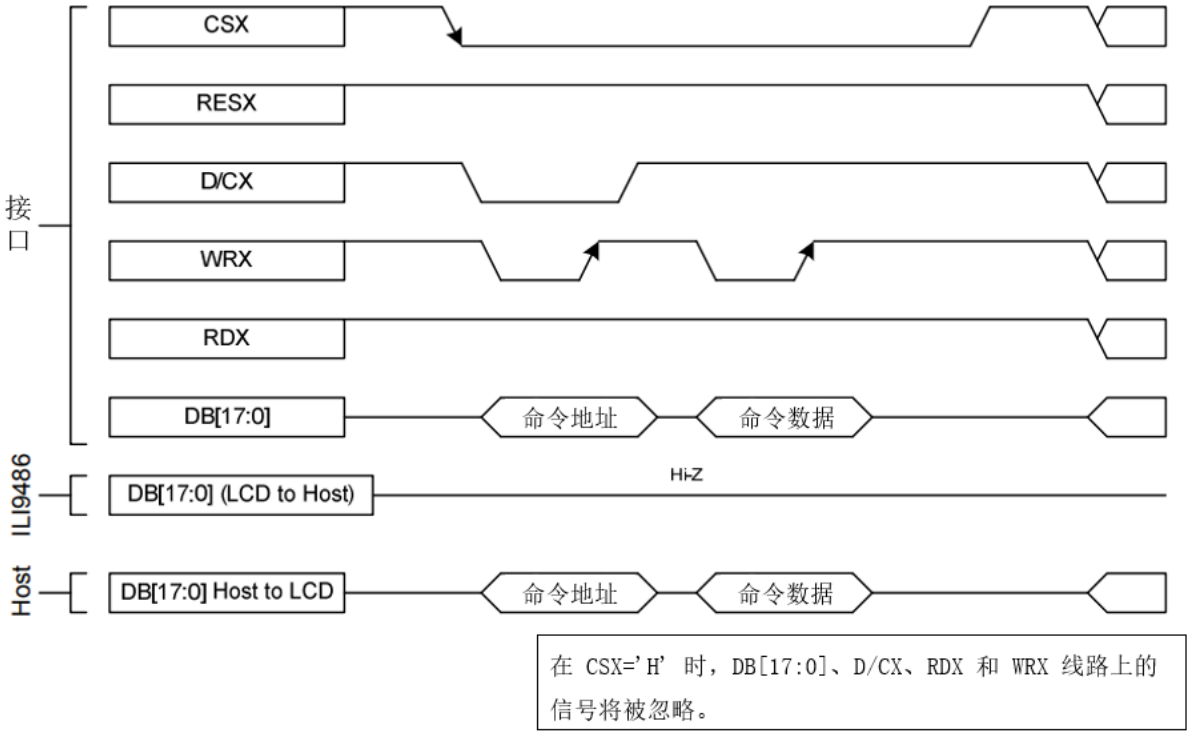



图 4.17.1 8080 系列并行接口写时序

## 4. 17. 2 FMSC 介绍

**1. FMSC 概念**

FMSC（Flexible Static Memory Controller），灵活的静态存储控制器。用于连接外部存储器，它提供了高度灵活的接口和功能，以满足各种应用需求。

FSMC 能够连接同步、异步存储器和 16 位 PC 存储卡。其主要用途如下：

（1）将 AHB 数据通信事务转换为适当的外部器件协议；

（2）满足外部器件的访问时序要求。

所有外部存储器共享地址、数据和控制信号，但有各自的片选信号。FSMC 一次只能访问一个外部器件。

FSMC 可连接的外部器件包括：静态随机访问存储器(SRAM)、只读存储器(ROM)、NOR Flash/OneNAND Flash、PSRAM（4 个存储区域）。

FSMC 也可对时序进行编程，以支持各种器件，特别是：等待周期可编程（最多 15 个时钟周期）；总线周转周期可编程（最多 15 个时钟周期）；输出使能和写入使能延迟可编程（最多 15 个时钟周期）；独立的读和写时序和协议，以支持各种存储器和时序。

注：定义外部器件类型和其特性的 FSMC 寄存器通常在启动时进行设置，并且在下次上电或复位前保持不变。但也可随时更改这些设置。

**2. FSMC 框图**

如图 4.17.2 所示，展示了 FSMC 的框图。

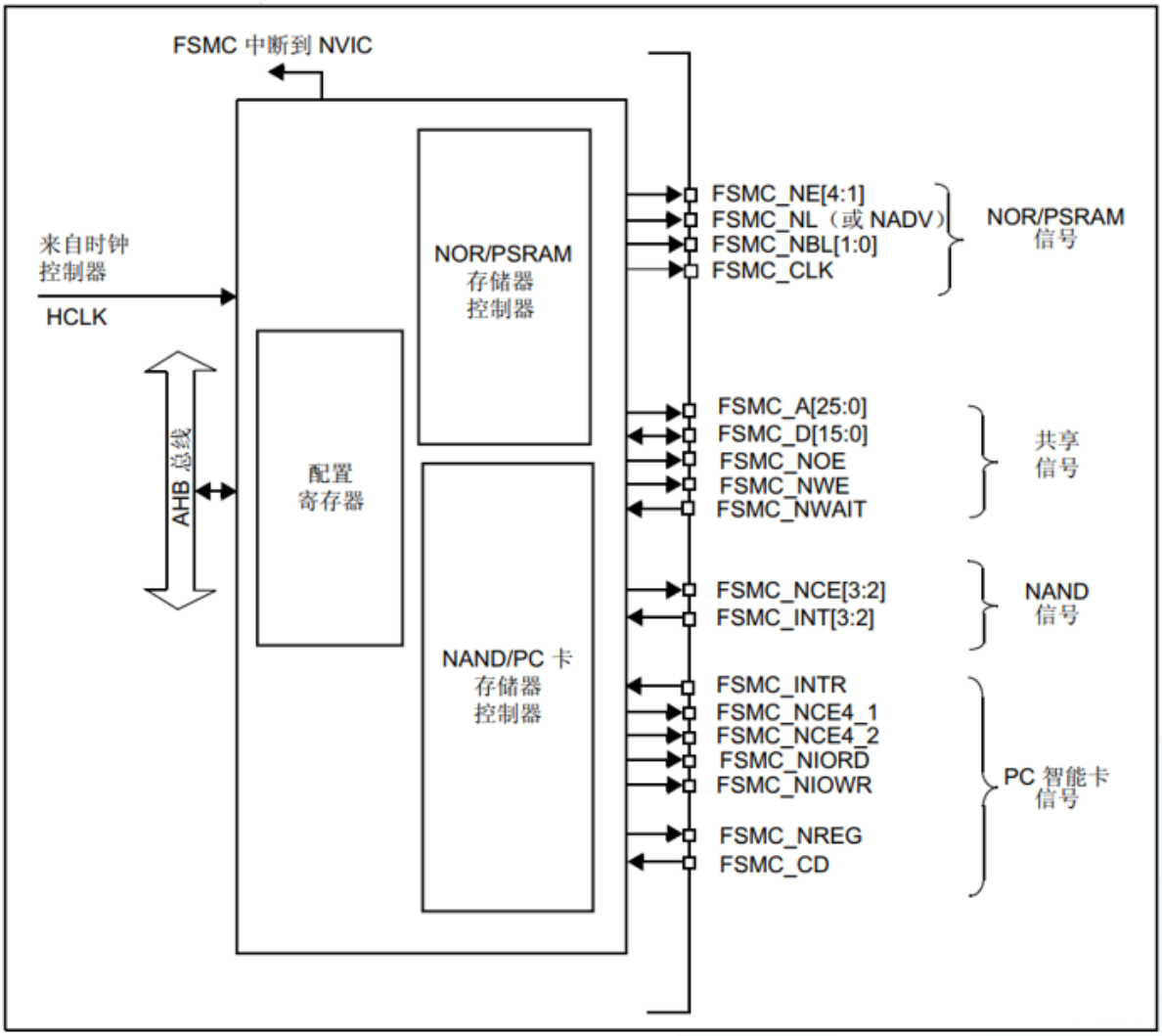



图 4.17.2 FSMC 框图

其中，包含了 FSMC 的四个主要模块：

（1）AHB 接口（包括 FSMC 配置寄存器）：AHB“从”接口允许内部 CPU 和其他总线“主”外设访问外部静态存储器。AHB 时钟(HCLK)是 FSMC 的参考时钟。

（2）NOR Flash/PSRAM 控制器。

（3）NAND Flash/PC 卡控制器。

（4）外部器件接口。

**3. 外部器件地址映射**

从 FSMC 的角度，外部存储器被划分为了 4 个固定大小的存储区域，每个存储区域的大小为 256MB，如图 4.17.3 所示，展示了 FSMC 的存储区域。

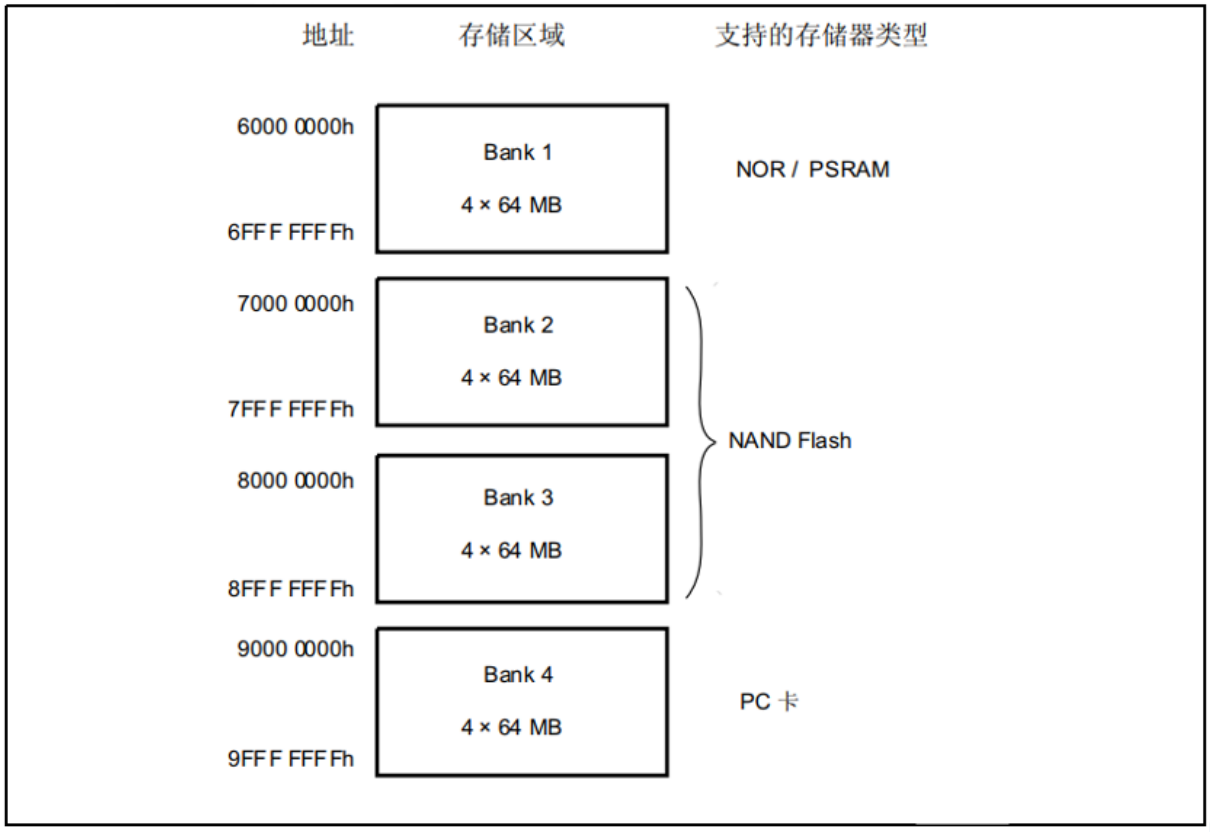



图 4.17.3 FSMC 存储区域

其中：

（1）存储区域 1 可连接多达 4 个 NOR Flash 或 PSRAM 存储器器件。此存储区域被划分为 4 个 NOR/PSRAM 区域，带 4 个专用片选信号。其 4 个区域所对应的地址范围如表 4.17.3 所示。

NOR/PSRAM 地址映射：HADDR[27:26]位用于从表 4.17.3 中所示的四个存储区域之中选择其中一个存储区域。

表 4.17.3 可选择存储器区域及区域对应的地址范围

| HADDR[27:26](1) | 可选存储器区域参数 | 对应地址范围 | 选择的存储区域 |
|---|---|---|---|
| 00 | FSMC_Bank1_NORSRAM1 | 0x6000 0000~0x63FF FFFF | 存储区域 1 NOR/PSRAM 1 |
| 01 | FSMC_Bank1_NORSRAM2 | 0x6400 0000~0x67FF FFFF | 存储区域 1 NOR/PSRAM 2 |
| 10 | FSMC_Bank1_NORSRAM3 | 0x6800 0000~0x6BFF FFFF | 存储区域 1 NOR/PSRAM 3 |
| 11 | FSMC_Bank1_NORSRAM4 | 0x6C00 0000~0x6FFF FFFF | 存储区域 1 NOR/PSRAM 4 |
| (1)HADDR 是 AHB 内部地址线，但也会参与对外部存储器的寻址。 | | | |

HADDR[25:0]包含外部存储器地址。由于 HADDR 为字节地址，而存储器按字寻址，所以根据存储器数据宽度不同，实际向存储器发送的地址也将有所不同，如表 4.17.4 所示。

表 4.17.4 外部存储器地址

| 存储器宽度(1) | 向存储器发出的数据地址 | 最大存储器容量（位） |
|---|---|---|
| 8 位 | HADDR[25:0] | 64MB*8 = 512Mb |
| 16 位 | HADDR[25:1]>>1 | 64MB/2*16 = 512Mb |
| (1)如果外部存储器的宽度为 16 位，FSMC 将使用内部的 HADDR[25:1]地址来作为对外部存储器的寻址地址 FSMC_A[24:00]。无论外部存储器的宽度为 16 位还是 8 位，FSMC_A[0]都应连接到外部存储器地址 A[0]。 | | |

（2）存储区域 2 和 3 用于连接 NAND Flash 器件（每个存储区域一个器件）。

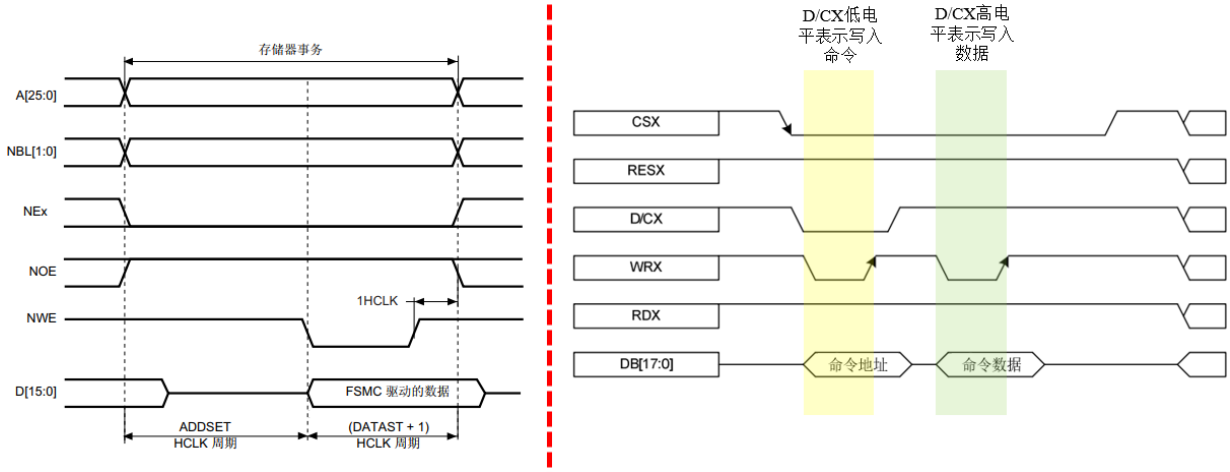
（3）存储区域 4 用于连接 PC 卡设备。

对于每个存储区域，所要使用的存储器类型由用户在配置寄存器中定义。

**4. STM32 使用 FSMC 接口实现 8080 时序**

对于 STM32F4xx 系列，可以使用 STM32 的 FSMC 接口来与一个遵循 8080 总线协议的设备（如 LCD 显示屏）进行通信。

结合图 4.17.4 和表 4.17.5 对照着看，可以发现 FSMC NOR FLASH 中的模式 A 与 8080 总线接口时序是十分相似的（除开 FSMC 的地址总线 A 和 8080 的 D/CX 线，其余接口是完全一样的）。




NOR Flash控制器异步事务 模式A写入时序　　LCD的8080时序（含命令和数据两个写入周期）

图 4.17.4 FSMC NOR FLASH A 模式写入时序与 8080 总线接口写入时序的对比

表 4.17.5 FSMC NOR FLASH 接口与 8080 总线接口对比

| FSMC NOR 接口 | 功能 | 8080 总线接口 | 功能 |
|---|---|---|---|
| NEx | 片选信号，x 可以为 1~4 | CSX | 片选信号 |
| NWE | 写入使能 | WRX | 并行数据写信号 |
| NOE | 输出（读）使能 | RDX | 并行数据读信号 |
| D[15:0] | 双向数据总线 | DB[15:0]（对于 16 位总线接口） | 并行数据总线 |
| A[25:0] | 地址总线 | D/CX | 数据/命令选择信号 |

对于 D/CX 线，它为高电平的时候表示数值，为低电平的时候表示命令，如果能使用 FSMC 的 A 地址线根据不同的情况产生对应的电平，那么就完全可以使用 FSMC 来模拟 8080 总线接口需要的时序。

为了模拟出 8080 总线时序，我们可以把 FSMC 的 A6 地址线(也可以使用 A0/A1/A2 等地址线，具体需要根据对应硬件确定)与 ILI9486L 芯片 8080 总线接口的 D/CX 信号线连接，那么当 A6 为高电平时(即 D/CX 为高电平)，数据线 DB[15:0]的信号会被 ILI9486L 理解为数值，若 A6 为低电平时(即 D/CX 为低电平)，传输的信号则会被理解为命令。

注意：在实际控制时，还需要注意 HADDR 内部地址与 FSMC 地址信号线的转换。

### 3. FSMC 相关函数讲解

（1）FSMC NOR/SRAM 初始化结构体定义

代码段 4.17.1 FSMC NOR/SRAM 初始化结构体定义

```
typedef struct
{
  uint32_t  FSMC_Bank;
  uint32_t  FSMC_DataAddressMux;
  uint32_t  FSMC_MemoryType;
  uint32_t  FSMC_MemoryDataWidth;
  uint32_t  FSMC_BurstAccessMode;
  uint32_t  FSMC_AsynchronousWait;
  uint32_t  FSMC_WaitSignalPolarity;
  uint32_t  FSMC_WrapMode;
  uint32_t  FSMC_WaitSignalActive;
  uint32_t  FSMC_WriteOperation;
  uint32_t  FSMC_WaitSignal;
  uint32_t  FSMC_ExtendedMode;
  uint32_t  FSMC_WriteBurst;
  FSMC_NORSRAMTimingInitTypeDef* FSMC_ReadWriteTimingStruct;
  FSMC_NORSRAMTimingInitTypeDef* FSMC_WriteTimingStruct;
}FSMC_NORSRAMInitTypeDef;
```

关于 FSMC NOR/SRAM 初始化结构体各个成员的详细描述，如表 4.17.6 所示。

表 4.17.6 FSMC NOR/SRAM 初始化结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| FSMC_Bank | FSMC_Bank1_NORSRAMx<br>（x 可以为 1~4） | 指定将使用的 NOR/SRAM 存储区域 |
| FSMC_DataAddressMux | FSMC_DataAddressMux_Disable<br>FSMC_DataAddressMux_Enable | 指定地址和数据值是否在数据总线上多路复用 |
| FSMC_MemoryType | FSMC_MemoryType_NOR<br>FSMC_MemoryType_PSRAM<br>FSMC_MemoryType_SRAM | 指定连接到相应存储区域的外部存储器类型 |
| FSMC_MemoryDataWidth | FSMC_MemoryDataWidth_16b<br>FSMC_MemoryDataWidth_8b | 指定外部存储器设备的宽度 |

| | | |
|---|---|---|
| FSMC_BurstAccessMode | FSMC_BurstAccessMode_Disable<br>FSMC_BurstAccessMode_Enable | 启用或禁用闪存的突发访问模式，仅对同步突发闪存有效 |
| FSMC_AsynchronousWait | FSMC_AsynchronousWait_Disable 禁用等待信号<br>FSMC_AsynchronousWait_Enable 启用等待信号 | 在异步传输期间启用或禁用等待信号，仅对异步闪存有效 |
| FSMC_WaitSignalPolarity | FSMC_WaitSignalPolarity_High<br>FSMC_WaitSignalPolarity_Low | 指定等待信号极性，仅在以突发模式访问闪存时有效 |
| FSMC_WrapMode | FSMC_WrapMode_Disable<br>FSMC_WrapMode_Enable | 启用或禁用闪存的环回突发访问模式，仅在以突发模式访问闪存时有效 |
| FSMC_WaitSignalActive | FSMC_WaitSignalActive_BeforeWaitState<br>FSMC_WaitSignalActive_DuringWaitState | 指定存储器是在等待状态前一个时钟周期还是在等待状态期间发出等待信号，仅在以突发模式访问存储器时有效 |
| FSMC_WriteOperation | FSMC_WriteOperation_Disable<br>FSMC_WriteOperation_Enable | 启用或禁用 FSMC 在所选存储区域中的写操作 |
| FSMC_WaitSignal | FSMC_WaitSignal_Disable<br>FSMC_WaitSignal_Enable | 通过等待信号启用或禁用等待状态插入，在突发模式下访问闪存时有效 |
| FSMC_ExtendedMode | FSMC_ExtendedMode_Disable<br>FSMC_ExtendedMode_Enable | 启用或禁用扩展模式 |
| FSMC_WriteBurst | FSMC_WriteBurst_Disable<br>FSMC_WriteBurst_Enable | 启用或禁用写入突发操作 |
| FSMC_ReadWrite TimingStruct | 由 NOR/SRAM 存储区域的时序参数组成的结构体 | 不使用扩展模式时的写入和读取访问时序参数 |
| FSMC_WriteTimingStruct | 由 NOR/SRAM 存储区域的时序参数组成的结构体 | 使用扩展模式时的写入访问时序参数 |

（2）FSMC NOR/SRAM 存储区域的时序参数结构体定义

代码段 4.17.2 FSMC NOR/SRAM 存储区域的时序参数结构体定义

---

```
typedef struct
{
  uint32_t  FSMC_AddressSetupTime;
  uint32_t  FSMC_AddressHoldTime;
  uint32_t  FSMC_DataSetupTime;
  uint32_t  FSMC_BusTurnAroundDuration;
  uint32_t  FSMC_CLKDivision;
  uint32_t  FSMC_DataLatency;
  uint32_t  FSMC_AccessMode;
}FSMC_NORSRAMTimingInitTypeDef;
```

---

关于 FSMC NOR/SRAM 存储区域的时序参数结构体各个成员的详细描述，如表 4.17.7 所示。

表 4.17.7 FSMC NOR/SRAM 存储区域的时序参数结构体成员

| 结构体成员名称 | 结构体成员参数<br>(即可选的参数) | 结构体成员说明 |
|---|---|---|
| FSMC_AddressSetupTime | 0x0~0xF | 定义用于配置地址建立阶段的持续时间的 HCLK 周期数 |
| FSMC_AddressHoldTime | 0x0~0xF | 定义用于配置地址保持阶段的持续时间的 HCLK 周期数 |
| FSMC_DataSetupTime | 0x00~0xFF | 定义用于配置数据建立阶段的持续时间的 HCLK 周期数 |
| FSMC_BusTurnAround Duration | 0x0~0xF | 定义用于配置总线周转阶段的持续时间的 HCLK 周期数 |
| FSMC_CLKDivision | 0x1~0xF | 定义用于构建一个存储器时钟周期（CLK）所需的 AHB 时钟周期（HCLK）数量 |
| FSMC_DataLatency | 参数值取决于内存类型：<br>（1）如果是 CRAM，必须设置为 0；<br>（2）在异步 NOR、SRAM 或 ROM 访问中，该值无关紧要；<br>（3）在启用同步突发模式的 NOR 闪存中，它的值可能介于 0 和 0xF 之间。 | 定义在发出突发的第一个数据前向存储器发出的时钟周期（CLK）数量 |

| | | |
|---|---|---|
| FSMC_AccessMode | FSMC_AccessMode_x（x 可以为 A~D） | 指定异步访问模式 |

（3）FSMC NOR/SRAM 存储区域初始化函数

表 4.17.8 FSMC NOR/SRAM 存储区域初始化函数

| 接口函数：FSMC_NORSRAMInit | |
|---|---|
| 函数原型 | void FSMC_NORSRAMInit(FSMC_NORSRAMInitTypeDef *FSMC_NORSRAMInitStruct) |
| 功能描述 | 根据 FSMC_NORSRAMInitStruct 中指定的参数初始化 FSMC NOR/SRAM 存储区域。 |
| 入口参数 | FSMC_NORSRAMInitStruct：指向 FSMC_NORSRAMInitTypeDef 结构体的指针，该结构体包含 FSMC NOR/SRAM 指定存储区域的配置信息。 |
| 返回值 | 无 |
| 使用示例 | FSMC_NORSRAMInit(&FSMC_NORSRAMInitStructure); |
| 注意事项 | 无 |

（4）启用或禁用指定的 NOR/SRAM 存储区域函数

表 4.17.9 启用或禁用指定的 NOR/SRAM 存储区域函数

| 接口函数：FSMC_NORSRAMCmd | |
|---|---|
| 函数原型 | void FSMC_NORSRAMCmd(uint32_t FSMC_Bank, FunctionalState NewState) |
| 功能描述 | 启用或禁用指定的 NOR/SRAM 存储区域。 |
| 入口参数 | FSMC_Bank：指定要使用的 FSMC 存储区域，其参数可以为 FSMC_Bank1_NORSRAMx（x 可以为 1~4）。 |
| | NewState：存储区域的状态，该参数可以是 ENABLE 或 DISABLE。 |
| 返回值 | 无 |
| 使用示例 | FSMC_NORSRAMCmd(FSMC_Bank1_NORSRAM4, ENABLE); |
| 注意事项 | 无 |

## 4. 17. 3 TFT LCD 显示

### 1. ILI9486L 芯片初始化命令

根据表 4.17.10 对 TFT 液晶显示终端模块的控制芯片进行初始化。

首先将 RES 端口拉低 100ms 进行复位，再将 RES 拉高 100ms，其次再输入一系列的初始化指令，如表 4.17.10 所示。表中每一个功能都需要先输入一个指令，再输入若干个参数，其详细功能可以翻阅 ILI9486L 芯片的技术手册。

表 4.17.10 常用初始化指令表

| 功能简述 | 指令 | 从左到右输入的参数 |
|---|---|---|
| 电源控制 1 | 0xc0 | 0x19、0x1a |
| 电源控制 2 | 0xc1 | 0x45、0x00 |
| 电源控制 3 | 0xc2 | 0x33 |
| VCOM 控制 | 0xc5 | 0x00、0x28 |
| 帧速率控制 | 0xb1 | 0xa0、0x11 |
| 显示反转控制 | 0xb4 | 0x02 |
| 显示功能控制 | 0xb6 | 0x00、0x42、0x3b |
| 进入模式设置 | 0xb7 | 0x07 |
| 正伽马校正 | 0xe0 | 0x1f、0x25、0x22、0x0b、0x06、0x0a、0x4e、0xc6、0x39、0x00、0x00、0x00、0x00、0x00、0x00 |
| 负伽马校正 | 0xe1 | 0x1f、0x3f、0x3f、0x0f、0x1f、0x0f、0x46、0x49、0x31、0x05、0x09、0x03、0x1c、0x1a、0x00 |
| 内存访问控制 | 0x36 | 0xc8 |
| 接口像素格式 | 0x3a | 0x55 |
| 停止休眠指令 | 0x11 | （不需要参数，但是需要至少等待延时 120ms） |
| 开启液晶屏显示 | 0x29 | （不需要参数） |

注：部分指令与模块的硬件参数匹配，可根据不同需求翻阅手册来修改初始化指令。

**2. TFT-LCD 绘图步骤**

输入完成初始化指令并点亮背光，完成初始化后屏幕呈现出全屏的白色界面。然后就可以在屏幕上绘图，绘图需要两个步骤：

（1）设置绘图范围。

（2）连续输入颜色信息并在指定范围内完成绘图。

设置绘图范围其实就是设置光标移动范围。这个功能需要设置 0x2a，0x2b 两个寄存器，这两个寄存器可以分别用于设置横向和纵向两个方向上的光标移动范围。其中 SC[15:8]与 SC[7:0]可以组合成一个 16 位数值，最小值为 0x0000。输入 SC[15:0]的值可以设置光标横向移动的起始地址。EC[15:0]设置光标横向移动的结束地址，SP[15:0]设置光标纵向移动的起始地址，EP[15:0]设置光标是纵向移动的结束地址。参考图 4.17.5

和图 4.17.6 完成设置光标的移动范围。要求先输入指令再依次输入对应的八位的参数。

| 2Ah | CASET (Column Address Set) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | D/CX | RDX | WRX | D[15:8] | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | HEX |
| Command | 0 | 1 | ↑ | XXXXXXXX | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 2Ah |
| 1st Parameter | 1 | 1 | ↑ | XXXXXXXX | SC[15:8] | | | | | | | | XX |
| 2nd Parameter | 1 | 1 | ↑ | XXXXXXXX | SC[7:0] | | | | | | | | XX |
| 3rd Parameter | 1 | 1 | ↑ | XXXXXXXX | EC[15:8] | | | | | | | | XX |
| 4th Parameter | 1 | 1 | ↑ | XXXXXXXX | EC[7:0] | | | | | | | | XX |

图 4.17.5 设置 X 轴方向上的光标范围

| 2Bh | PASET (Page Address Set) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | D/CX | RDX | WRX | D[15:8] | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | HEX |
| Command | 0 | 1 | ↑ | XXXXXXXX | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 2Bh |
| 1st Parameter | 1 | 1 | ↑ | XXXXXXXX | SP[15:8] | | | | | | | | XX |
| 2nd Parameter | 1 | 1 | ↑ | XXXXXXXX | SP[7:0] | | | | | | | | XX |
| 3rd Parameter | 1 | 1 | ↑ | XXXXXXXX | EP[15:8] | | | | | | | | XX |
| 4th Parameter | 1 | 1 | ↑ | XXXXXXXX | EP[7:0] | | | | | | | | XX |

图 4.17.6 设置 Y 轴方向上的光标范围

可这样理解：在光标处写入颜色后光标都会自动后移一个位置等待下次写入。而上图两个寄存器则是控制了光标的移动范围，当光标后移的位置超出了设置的范围会自动跳往设置中下一行的起始位置或者第一行的起始位置。设置光标移动范围时要求横向和纵向的结束地址的值需要恒大于对应起始地址的值，否则会导致输入的颜色数据无效或者产生花屏。

设置完成绘图范围后可以填充颜色。如图 4.17.7 所示，输入 0x2c 指令后可以通过连续输入 16 位的颜色值完成在指定范围内的光标移动处多次写入颜色值，最终达到绘图的效果。其中，写入的数值并不是实时显示的而是存储在显示终端的内存中，再由定时刷新屏幕的方式将内存中的数据显示在屏幕上。

| 2Ch | RAMWR (Memory Write) | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | D/CX | RDX | WRX | D[15:8] | D7 | D6 | D5 | D4 | D3 | D2 | D1 | D0 | HEX |
| Command | 0 | 1 | ↑ | XXXXXXXX | 0 | 0 | 1 | 0 | 1 | 1 | 0 | 0 | 2Ch |
| 1st Parameter | 1 | 1 | ↑ | D1[15:0] | | | | | | | | | XX |
| : | 1 | 1 | ↑ | Dx[15:0] | | | | | | | | | XX |
| Nth Parameter | 1 | 1 | ↑ | Dn[15:0] | | | | | | | | | XX |

图 4.17.7 输入颜色的指令表

以“画绿色方块”为例：需要在坐标（0,0）到坐标（9,9）间绘制一个 10*10 大小的绿色方块。先使用指令 0x2a、0x2b 设置 0x0000 的横向起始地址、0x0009 的横向结束地址、0x0000 的纵向起始地址、0x0009 的纵向结束地址，然后再使用指令 0x2c 连续输入 100 个绿色值就能完成绘制一个绿色方块。

16 位颜色值的组成如下图 4.17.8 所示，由 5 位的红色值、6 位的绿色值、5 位的蓝

色值组成。显示时红色和蓝色的第六位都是由第一位补充得到的。

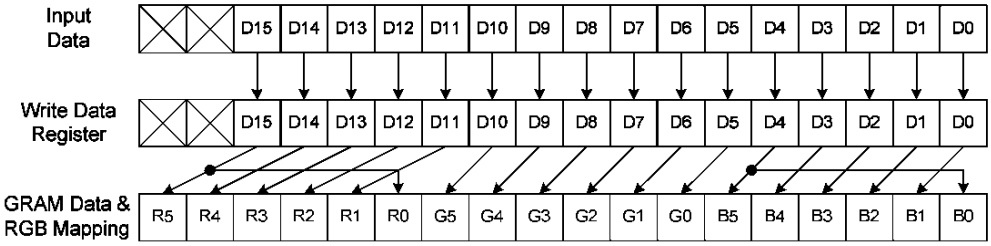



图 4.17.8 16 位颜色信息图

在现实中会使用 0 到 255 去描述一个 RGB 值，一共使用到了 24 位数据（RGB888）。而在显示终端中设置的颜色值长度为 16 位（RGB565），所以需要将 24 位中红色的后三位、绿色的后两位、蓝色的后三位舍去。计算后可以得到对应的 16 位颜色值，如：红色 0xf800，绿色 0x07e0，蓝色 0x001f，白色 0xffff，黑色 0x0000。

在了解完上述原理后，绘制文字的原理可大致概括为：先设置 32*32 像素大小的光标范围，再连续填入 1024 个 16 位颜色值就能完成在屏幕上绘制一个 32*32 大小的文字。同理，任意大小的字符、图形、图片都可以使用这种方法显示在屏幕上。

**3. 字符显示及取模工具的使用**

（1）字符显示机制

在 LCD 屏上显示一个字符，需要：

①占据屏幕上的一块地方，其大小由字符大小决定；

②在占据的地方上依次控制每个像素点是否显示。

以“好”字为例，假设我们需要将它以 32*32 大小在 LCD 屏幕上显示，则需要在屏幕上占用 32*32（水平 32 个像素点，垂直 32 个像素点）的空间像素点。

如图 4.17.9 所示，在这块 32*32 大小的空间里，每个像素点是否显示是由我们取的字模决定的，字模中的每 1 位数据表示一个像素点，如果该位为 0 则表示此处像素点不显示、为 1 则表示显示（阴码规则）。

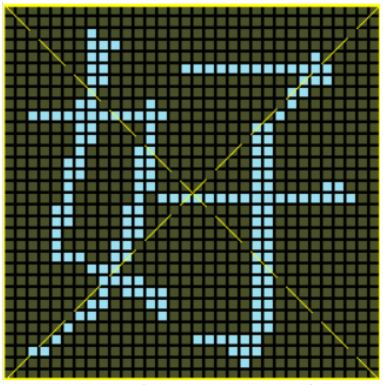


图 4.17.9 32*32 大小的“好”字显示示例

所以，字符显示的函数只需要使用画点函数即可实现。程序中需逐位的读取字模，如果该位为 0，则写入背景颜色；如果该位为 1，写入前景颜色。

（2）字模生成

①这里我们使用的取字模软件生成字模，在生成之前，我们需要对取字模软件进行设置，进入程序后，出现如图 4.17.10（左）所示的主界面，点击其中的齿轮状按钮即可进入右图的设置界面，按照图片中的设置修改即可，最后点击“确定”应用设置。

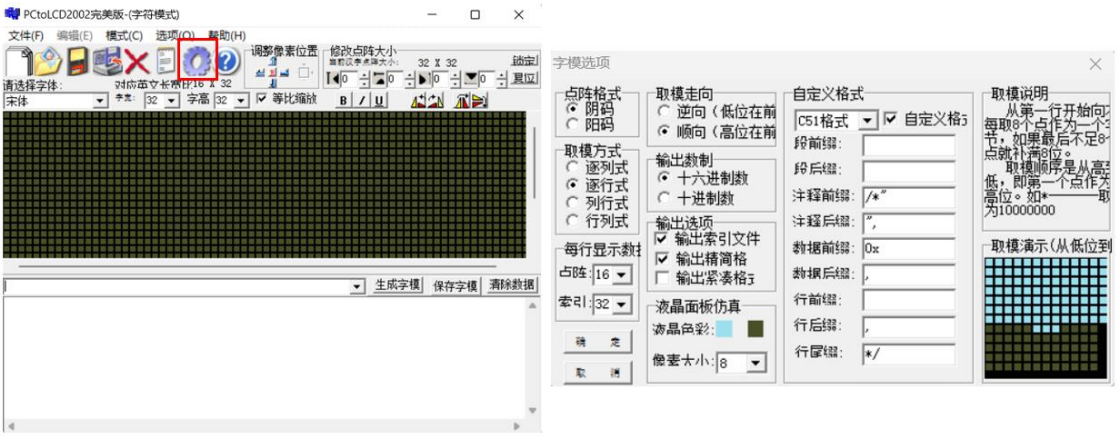



图 4.17.10 取字模软件的设置

②将软件设置完毕后，接下来就是对具体的字符进行取模了。以 32*32 大小的“好”字为例，如图 4.17.11，按照步骤完成即可在下方生成“好”字的字模数据。

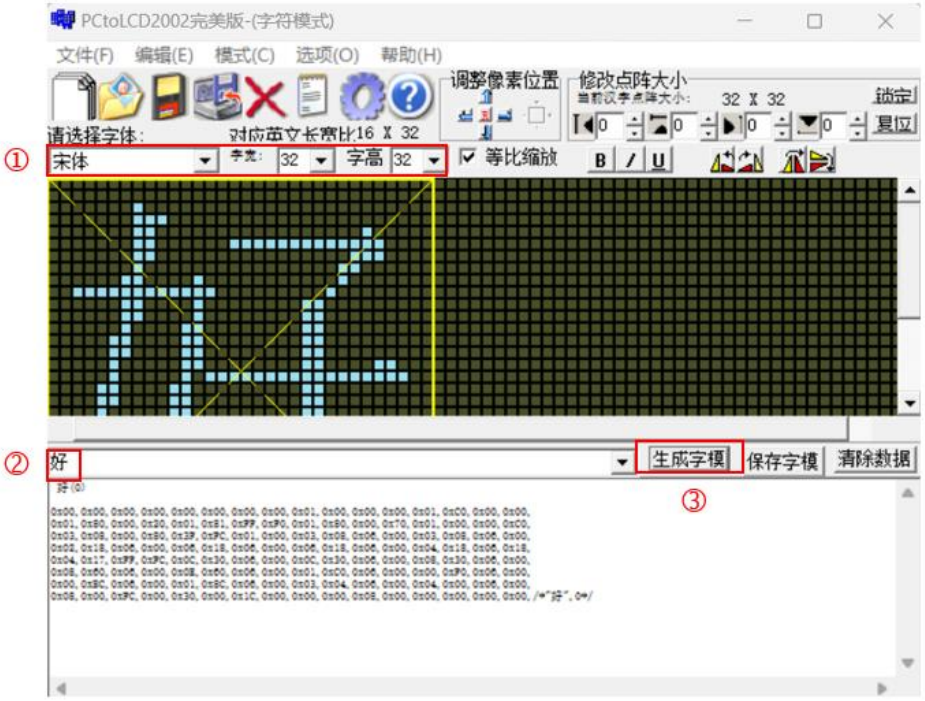



图 4.17.11 32*32 大小的“好”字取模步骤

③取完字模后，将其字模数据粘贴至 font.h（LCD 存放字模数据的头文件）的 tfont32[]数组（32*32 大小中文字模数据存放处）中，并将其格式修改为如图 4.17.12 所示即可，接下来就可以使用程序配套的 LCD 显示函数在 LCD 屏中显示“好”字了。

```
"好",
0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x01, 0x00, 0x00, 0x00, 0x01, 0xC0, 0x00, 0x00,
0x01, 0x80, 0x00, 0x20, 0x01, 0x81, 0xFF, 0xF0, 0x01, 0x80, 0x00, 0x70, 0x01, 0x00, 0x00, 0xC0,
0x03, 0x08, 0x00, 0x80, 0x3F, 0xFC, 0x01, 0x00, 0x03, 0x08, 0x06, 0x00, 0x03, 0x08, 0x06, 0x00,
0x02, 0x18, 0x06, 0x00, 0x06, 0x18, 0x06, 0x00, 0x06, 0x18, 0x06, 0x00, 0x04, 0x18, 0x06, 0x18,
0x04, 0x17, 0xFF, 0xFC, 0x0C, 0x30, 0x06, 0x00, 0x0C, 0x30, 0x06, 0x00, 0x08, 0x30, 0x06, 0x00,
0x08, 0x60, 0x06, 0x00, 0x0E, 0x60, 0x06, 0x00, 0x01, 0xC0, 0x06, 0x00, 0x00, 0xF0, 0x06, 0x00,
0x00, 0xBC, 0x06, 0x00, 0x01, 0x8C, 0x06, 0x00, 0x03, 0x04, 0x06, 0x00, 0x04, 0x00, 0x06, 0x00,
0x08, 0x00, 0xFC, 0x00, 0x30, 0x00, 0x1C, 0x00, 0x00, 0x00, 0x08, 0x00, 0x00, 0x00, 0x00, 0x00, /*"好",0*/
```

图 4.17.12 32*32 大小的“好”字取字模数据后的存放格式

## 4.17.4 硬件设计

### 1. 实验电路分析

如图 4.17.13 所示，展示了 TFT LCD 的电路原理图，其中包含了 LCD 所需的 GPIO 引脚。

其中：

（1）FSMC_NE4 代表片选引脚，对应 FSMC NOR 接口的 NEx（片选，其中 x 为 4）；

（2）FSMC_NWE 代表写入引脚，对应 FSMC NOR 接口的 NEW（写入使能）；

（3）FSMC_A6 代表地址总线引脚，对应 FSMC NOR 接口的 A[25:0]（地址总线，

选择 A6。已知 NEx 中 x 为 4，地址由 0x6C00 0000 开始，在程序中 LCD_BASE 地址 0x6C00 007E 则为 A6 的偏移地址。那么可得 LCD->LCD_REG 地址为 0x6C00 007E，对应 A6 状态为 0；LCD->LCD_DAT 的地址为 0x6C00 0080，对应 A6 的状态为 1）；

（4）FSMC_NOE 代表读取引脚，对应 FSMC NOR 接口的 NOE（输出使能）；

（5）FSMC_D0~D15 代表数据引脚，对应 FSMC NOR 接口的 D[15:0]（双向数据总线）；

（6）LCD_RESET 代表 LCD 复位引脚；

（7）LCD_BL 代表背光控制引脚。

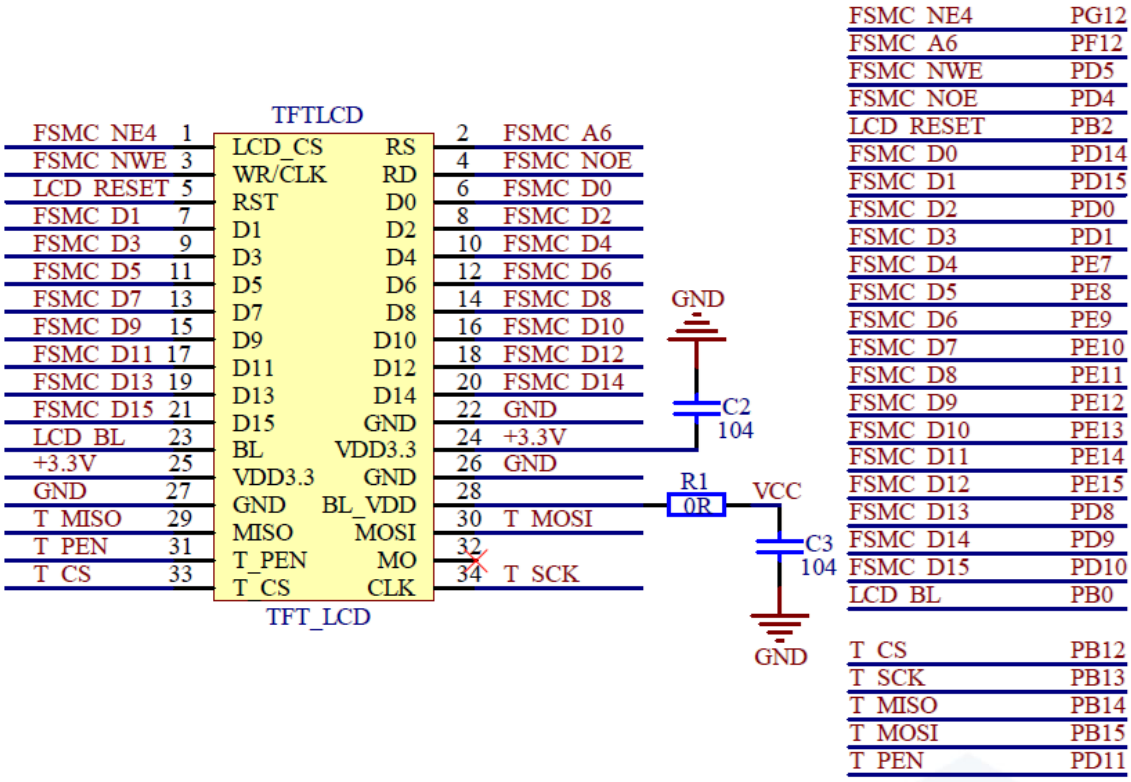



图 4.17.13 TFT LCD 电路原理图

## 4.17.5 软件设计

本小节是对实验核心代码的设计思路进行梳理，完整工程参见项目的参考案例。同时也为了方便代码的移植和对代码结构的理解，案例中将 LCD 部分的驱动代码独立开来，源文件（ili9486.c、gui.c）和头文件（font.h、pic.h、ili9486.h、gui.h）存放在实验案例根目录中的 HARDWARE/LCD 下。

其中 pic.h 存放了 LCD 显示的图片的取模数据；font.h 存放了 LCD 显示的文字的

取模数据；ili9486.c 和 ili9486.h 包含了 LCD 显示屏的驱动程序；gui.c 和 gui.h 则包含 LCD 的绘制函数（包括中英文、数字及简单图形的绘制函数）。

**1. 软件设计流程**

如图 4.17.14，展示了本实验的软件设计流程图。

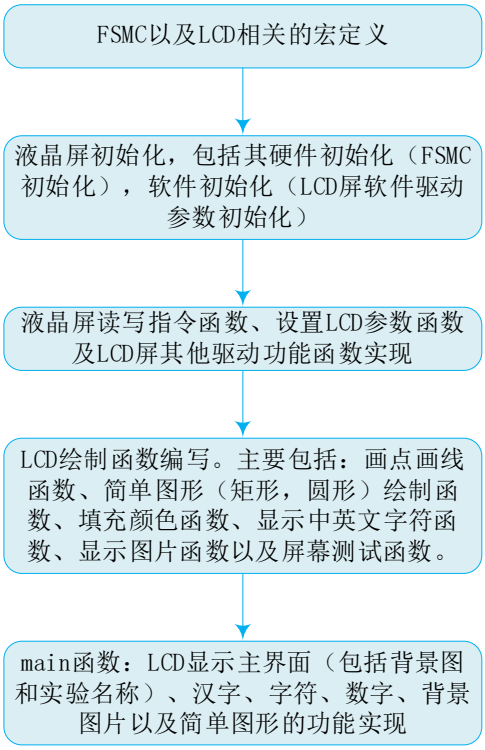



图 4.17.14 软件设计流程图

如图 4.17.15，展示了 LCD 屏的初始化流程图，其中包括软件初始化和硬件初始化。

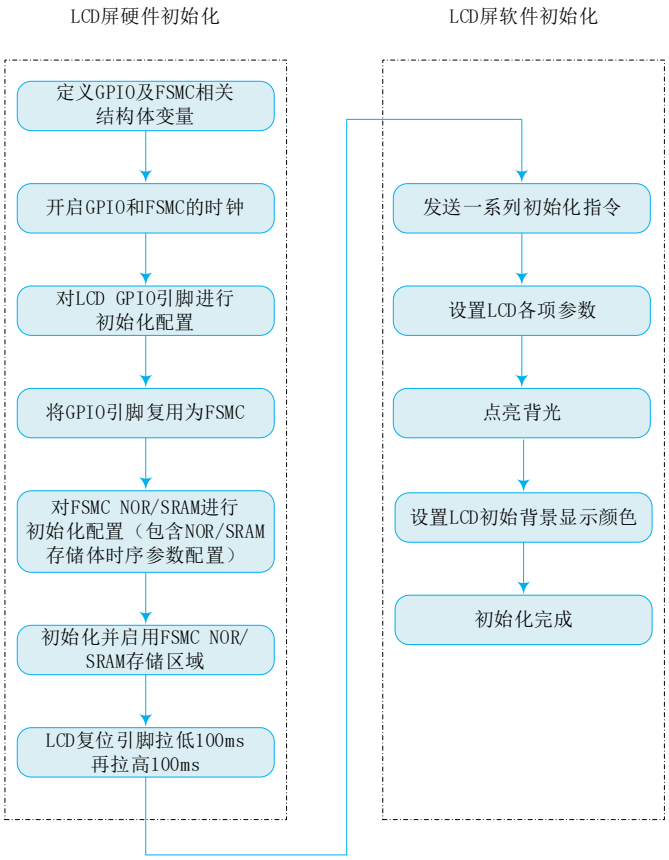



图 4.17.15 LCD 初始化流程图

**2. 软件代码讲解**

本小节主要是常用 LCD 驱动和部分 GUI 绘制的函数讲解，具体的代码请参考配套资料。

（1）LCD 初始化函数

表 4.17.11 LCD 初始化函数

| 接口函数：LCD_Init | |
|---|---|
| 函数原型 | void LCD_Init(void); |
| 功能描述 | 对 LCD 屏进行初始化，包括 FSMC 初始化（硬件初始化）和 LCD 屏驱动参数初始化（软件初始化）。 |
| 入口参数 | 无 |
| 返回值 | 无 |
| 使用示例 | LCD_Init(); |

| | |
|---|---|
| 注意事项 | 在进行屏幕绘制之前，必须调用 LCD 初始化函数。 |

（2）LCD 画点函数

表 4.17.12 LCD 画点函数

| 接口函数：GUI_Draw_Point | |
|---|---|
| 函数原型 | void GUI_Draw_Point(uint16_t x, uint16_t y, uint16_t c); |
| 功能描述 | 在 LCD 屏指定位置画一个点。 |
| 入口参数 | (x, y)：需要绘制的点的位置坐标。x 为横坐标位置，y 为纵坐标位置。 |
| | c：需要绘制的点的填充颜色。 |
| 返回值 | 无 |
| 使用示例 | GUI_Draw_Point(100, 100, RED); |
| 注意事项 | “ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。 |

（3）LCD 画线函数

表 4.17.13 LCD 画线函数

| 接口函数：GUI_Draw_Line | |
|---|---|
| 函数原型 | void GUI_Draw_Line(uint16_t x1, uint16_t y1,<br>uint16_t x2, uint16_t y2, uint16_t c); |
| 功能描述 | 在 LCD 屏指定位置画一条线。 |
| 入口参数 | (x1, y1)：需要绘制的线的起始坐标。 |
| | (x2, y2)：需要绘制的线的结束坐标。 |
| | c：需要绘制的线的填充颜色。 |
| 返回值 | 无 |
| 使用示例 | GUI_Draw_Line(100, 100, 200, 200, RED); |
| 注意事项 | “ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。 |

（4）LCD 绘制矩形函数

表 4.17.14 LCD 绘制矩形函数

| 接口函数：GUI_Draw_Line | |
|---|---|
| 函数原型 | void GUI_Draw_Rectangle(uint16_t x1, uint16_t y1, uint16_t x2, uint16_t y2, uint16_t c); |
| 功能描述 | 在 LCD 屏指定位置绘制一个矩形（非填充）。 |
| 入口参数 | (x1, y1)：需要绘制的矩形的左上角坐标。 |
| | (x2, y2)：需要绘制的矩形的右下角坐标。 |
| | c：需要绘制的矩形的四个边的填充颜色。 |
| 返回值 | 无 |
| 使用示例 | GUI_Draw_Rectangle(100, 100, 200, 200, RED); |
| 注意事项 | （1）“ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。<br>（2）GUI_Draw_Fill_Rectangle 函数也是绘制矩形函数，其参数是一致的，但其颜色参数是填充整个矩形区域的，而非四个边的颜色填充。 |

（5）LCD 绘制圆形函数

表 4.17.15 LCD 绘制圆形函数

| 接口函数：GUI_Circle | |
|---|---|
| 函数原型 | void GUI_Circle(int xc, int yc, uint16_t c, uint16_t r, int fill); |
| 功能描述 | 在 LCD 屏指定位置绘制一个圆形。 |
| 入口参数 | (xc, yc)：需要绘制的圆的圆心坐标。 |
| | c：需要绘制的圆的填充颜色。 |
| | r：需要绘制的圆的半径。 |
| | fill：填充标志，为 1 表示填充，0 表示不填充。 |
| 返回值 | 无 |
| 使用示例 | GUI_Circle(100,100,RED,50,0); |
| 注意事项 | “ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。 |

（6）LCD 显示数字函数

表 4.17.16 LCD 显示数字函数

| 接口函数：GUI_Show_Int_Num | |
|---|---|
| 函数原型 | void GUI_Show_Int_Num(uint16_t x, uint16_t y, uint16_t fc, uint16_t bc, int32_t num, uint8_t len, uint8_t size, uint8_t mode); |
| 功能描述 | LCD 显示数字变量值，非 32 位的变量可以用强制转换（int32_t）输入数值。 |
| 入口参数 | (x, y)：起点坐标。 |
| | fc：前置画笔颜色。 |
| | bc：背景颜色。 |
| | num：需要绘制的数值，范围（-2147483648~2147483647）。 |
| | len：显示的最少位数，若显示值的长度大于 len 正常显示，否则补位显示。 |
| | size：字体大小。 |
| | mode：0，填充模式；1，叠加模式。 |
| 返回值 | 无 |
| 使用示例 | GUI_Show_Int_Num(100,100,RED,WHITE,123,3,32,0); |
| 注意事项 | “ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。 |

（7）LCD 字符串显示（包含中英文显示）函数

表 4.17.17 LCD 字符串显示（包含中英文显示）函数

| 接口函数：GUI_Show_Str | |
|---|---|
| 函数原型 | void GUI_Show_Str(uint16_t x, uint16_t y, uint16_t fc, uint16_t bc, char* str, uint8_t size, uint8_t mode); |
| 功能描述 | LCD 显示字符串，包含了中英文字符串的显示。 |
| 入口参数 | (x, y)：起点坐标。 |
| | fc：前置画笔颜色。 |
| | bc：背景颜色。 |
| | *str：需要绘制的字符串。 |

| | |
|---|---|
| | size：字体大小。 |
| | mode：0，填充模式；1，叠加模式。 |
| 返回值 | 无 |
| 使用示例 | GUI_Show_Str(100,100,RED,WHITE,"中文测试",24,0); |
| 注意事项 | （1）“ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。<br>（2）对于数字或浮点数，可以使用 sprintf 函数（需包含 stdio.h 头文件）将其转为字符串的形式进行打印。使用方法：<br>float data = 10.23f;<br>char show_str[50] = {0};<br>sprintf(show_str, "数据：%5.2f", data);　//字符需提前进行取模<br>GUI_Show_Str(100, 100, RED, WHITE, show_str, 24, 0); |

（8）LCD 居中显示字符串函数

表 4.17.18 LCD 居中显示字符串函数

| 接口函数：GUI_Str_Center | |
|---|---|
| 函数原型 | void GUI_Str_Center(uint16_t x, uint16_t y, uint16_t fc, uint16_t bc, char* str, uint8_t size, uint8_t mode); |
| 功能描述 | LCD 居中显示字符串。 |
| 入口参数 | (x, y)：起点坐标。 |
| | fc：前置画笔颜色。 |
| | bc：背景颜色。 |
| | *str：需要绘制的字符串。 |
| | size：字体大小。 |
| | mode：0，填充模式；1，叠加模式。 |
| 返回值 | 无 |
| 使用示例 | GUI_Str_Center(0,100,RED,WHITE,"中文测试",24,0); |
| 注意事项 | “ili9486.h”存放了各种常用颜色的宏定义（GB565 格式）。 |

（9）LCD 绘制一幅 16 位 BMP 图像函数

表 4.17.19 LCD 绘制一幅 16 位 BMP 图像函数

| 接口函数：GUI_Drawbmp16 | |
|---|---|
| 函数原型 | void GUI_Drawbmp16(uint16_t x0, uint16_t y0, uint16_t x1, uint16_t y1, const uint8_t* p); |
| 功能描述 | LCD 显示一幅 16 位 BMP 图像。 |
| 入口参数 | (x0, y0)：需要绘制的图像的左上角坐标。 |
| | (x1, y1)：图片像素大小。 |
| | *p：图像数组。 |
| 返回值 | 无 |

| | |
|---|---|
| 使用示例 | GUI_Drawbmp16(0, 0, 320, 480, gImage_background); |
| 注意事项 | 图像数据需提前取模。 |

（10）实验主界面绘制函数

表 4.17.20 实验主界面绘制函数

| 接口函数：GUI_Main_Test | |
|---|---|
| 函数原型 | void GUI_Main_Test(void); |
| 功能描述 | 绘制综合测试程序主界面。 |
| 入口参数 | 无 |
| 返回值 | 无 |
| 使用示例 | GUI_Main_Test(); |
| 注意事项 | 本函数主要实现了实验主界面的绘制（主要是背景图片的绘制），程序中需先进行 LCD 初始化，再调用本函数。 |

（11）常用颜色的宏定义（GB565 格式）。

代码段 4.17.3 常用颜色的宏定义（GB565 格式）（ili9486.h）

```
/*******************RGB 565 颜色格式*******************/
#define WHITE       0xFFFF      //白色
#define BLACK       0x0000      //黑色
#define BLUE        0x001F      //蓝色
#define BRED        0XF81F      //红蓝色
#define GRED        0XFFE0      //红绿色
#define GBLUE       0X07FF      //蓝绿色
#define RED         0xF800      //红色
#define MAGENTA     0xF81F      //品红色
#define GREEN       0x07E0      //绿色
#define CYAN        0x7FFF      //青色
#define YELLOW      0xFFE0      //黄色
#define BROWN       0XBC40      //棕色
#define BRRED       0XFC07      //棕红色
#define GRAY        0X8430      //灰色
#define DARKBLUE    0X01CF      //深蓝色
#define LIGHTBLUE   0X7D7C      //浅蓝色
```

```
#define GRAYBLUE     0X5458      //灰蓝色
#define LIGHTGREEN   0X841F      //浅绿色
#define LIGHTGRAY    0XC618      //浅灰色(PANNEL),窗体背景色
#define LGRAYBLUE    0XA651      //浅灰蓝色(中间层颜色)
#define LBBLUE       0X2B12      //浅棕蓝色(选择条目的反色)
//自定义颜色
#define CONTENT_COLOR   0x6CBE  //文本背景颜色
#define TITLE_COLOR     0x1A14  //标题背景颜色
```

---

（12）实验主函数代码实现。

代码段 4.17.4 实验主函数代码实现（main.h）

---

```
int main(void)
{
    Delay_Init();        //延时初始化
    LCD_Init();          //液晶显示初始化
   //显示主界面背景图片及实验名称
    GUI_Main_Test();
    GUI_Str_Center(0, 140, WHITE, TITLE_COLOR, "LCD 显示实验", 32, 0);
    Delay_ms(500);
    //绘制文字
    GUI_Str_Center(0, 320, BLACK, CONTENT_COLOR, "中文测试", 24, 0);
    Delay_ms(500);
    GUI_Show_Str(100, 350, BLACK, CONTENT_COLOR, "123456*_*abcdef", 16, 0);
    Delay_ms(500);
    //绘制非填充矩形
    GUI_Draw_Rectangle(120, 370, 200, 410, BLACK);
    Delay_ms(500);
    //绘制线段
    GUI_Draw_Line(120, 370, 200, 410, BLACK);
    Delay_ms(500);
    GUI_Draw_Line(200, 370, 120, 410, BLACK);
    Delay_ms(500);
    //绘制圆形
    GUI_Circle(160, 390, BLACK, 18, 1);
    while(1){}
}
```

---

## 4.17.6 实验流程

1. 实验设备准备：

软件设备：PC 系统：Windows 7 及以上、集成开发环境（IDE）：Keil MDK。

硬件设备：PC（主机）、STM32F4 核心板、12V 电源适配器、ST Link 下载器、TFT LCD 显示屏。

| PC | STM32F4 核心板 | 12V 电源适配器 |
|---|---|---|
| | | |

| ST Link 下载器 | TFT LCD 显示屏 |
|---|---|
| | |

2. 实验硬件接线：

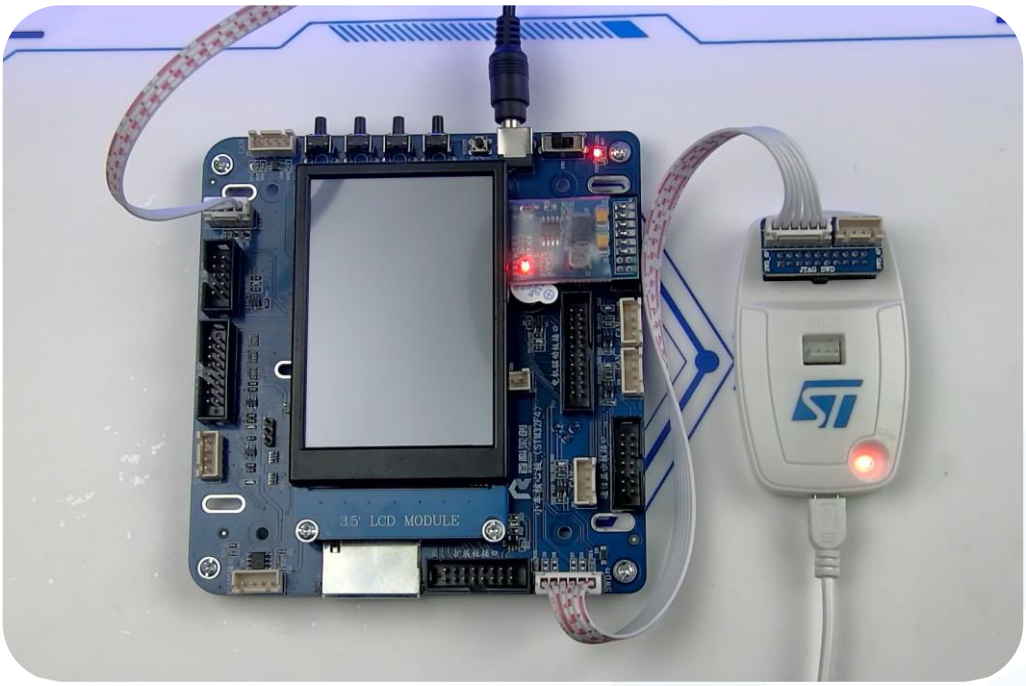



图 4.17.16 实验硬件接线示意图

3. 软件资源参考：

（1）配套资料提供了实验的案例，可供参考。

（2）实验有关的原理可结合《STM32F4xx 中文参考手册.pdf》或《STM32F4xx 英文参考手册.pdf》进行学习。

（3）实验相关的电路原理图可以参考《小车核心板（STM32F4）V3.7.pdf》。

（4）LCD 屏的相关参数可参考《ILI9486L 手册（液晶屏控制器）.pdf》。

4. 使用 IDE 将程序编译下载至核心板，观察 LCD 显示屏是否正常显示主界面。

## 4.17.7 结果展示

LCD 显示主界面（包括背景图和实验名称）、汉字、字符、数字、背景图片以及简单图形。

图 4.17.17 LCD 屏显示示意图

# 第5章 常见问题解决方案

## 5.1 常见下载问题及解决方案

### 5.1.1 NO ST-Link/No J-Link

未找到下载器：检查下载器是否连接或选择正确（JLink/ST-Link），是否安装相关下载器驱动。

图 5.1.1 未找到下载器

### 5.1.2 No target connected

未找到核心板：检查核心板是否上电，或是否连接下载器正常。

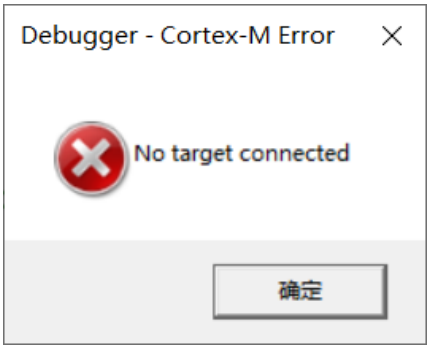



图 5.1.2 未找到核心板

### 5.1.3 Unknown target connected

未知目标连接：检查接口是否选择正确接口（SW/JTAG），本设备使用 SW 接口。

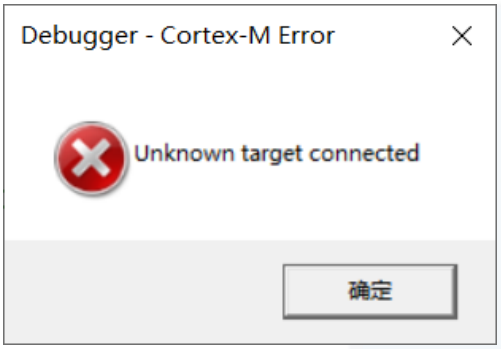



图 5.1.3 未知目标连接

## 5. 1. 4 Error: Flash Download failed - "Cortex-M4"

Flash 下载失败：添加 Flash Dowmlad 下载文件。

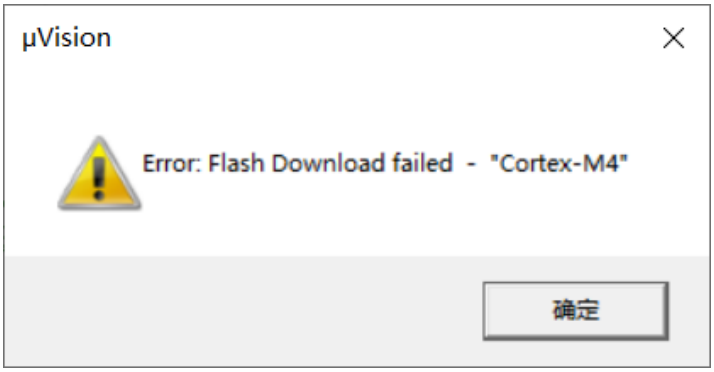



图 5.1.4 Flash 下载失败

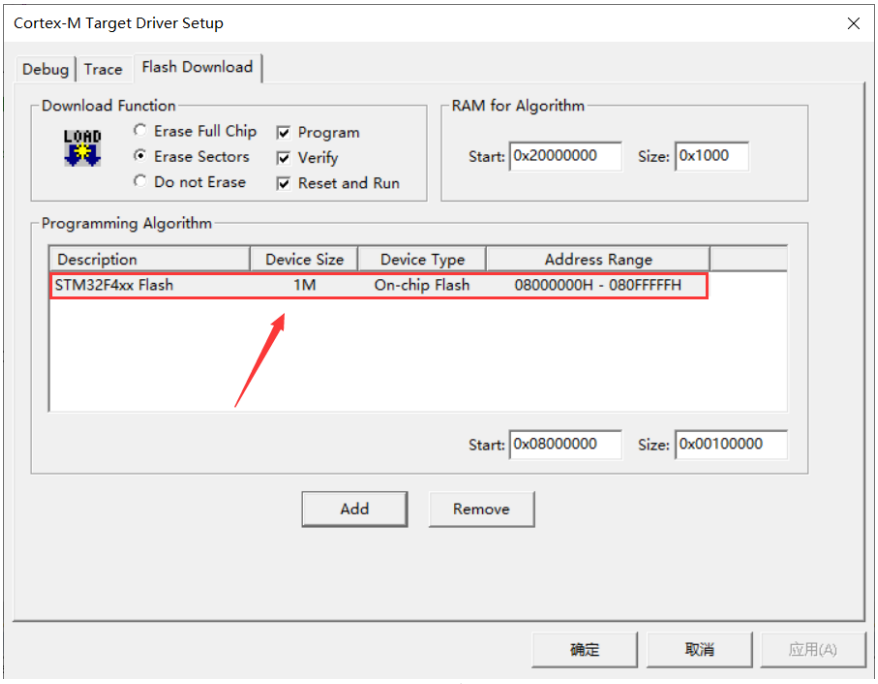



图 5.1.5 添加 Flash

## 5. 1. 5 Invalid ROM Table

下载时提示无效 ROM，可能是芯片保护进行锁死：

1、烧进去的工程对应器件与目标器件不一致；

2、烧进去的工程 HSE_VALUE 与目标板上晶振频率不一致；

3、其他违规操作…

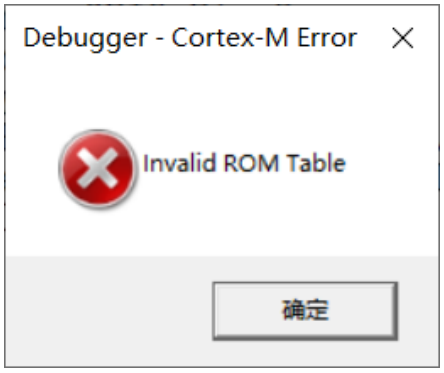



图 5.1.6 下载失败

解决方案：

点击进入下载设置，将 Connect 连接方式改成 under Reset，下载前复位，配置完成后点击下载或短按复位按键后松开进行下载，恢复下载。

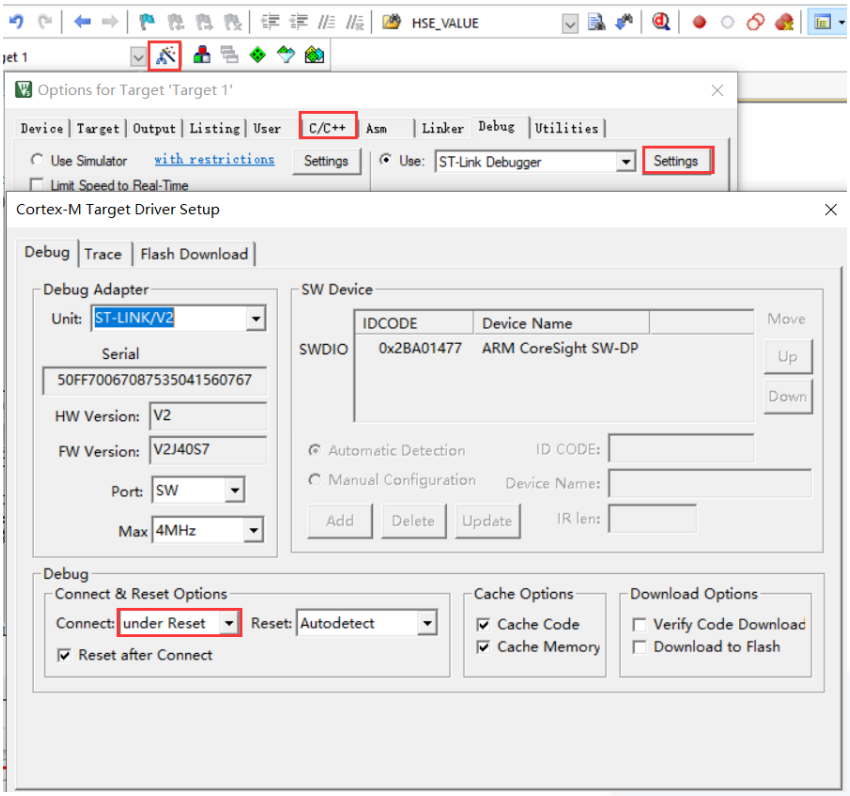



图 5.1.7 下载前复位