

STM32L1 系列

NB-IoT 物联网实验手册



中科四平物联网交流群
扫一扫二维码，入群聊。



中科四平
ZHONGKE SIPING

前言

物联网是新一代信息技术的重要组成部分，也是“信息化”时代的重要发展阶段，顾名思义，物联网就是物物相连的互联网。这有两层意思：一，物联网的核心和基础仍然是互联网，就是在互联网基础上的延伸和扩展的网络；二，其用户端延伸和扩展到了任何物体与物体之间，进行信息交互和通信，也就是物物相息。物联网通过智能感知、识别技术与普适计算等通信感知技术，广泛应用于网络的融合中，也因此被称为继计算机、互联网之后世界信息产业发展的第三次浪潮。物联网是互联网的应用拓展，与其说物联网是网络，不如说物联网是业务和应用。因此应用创新是物联网发展的核心。利用局部网络或互联网等通信技术把传感器、控制器、机器、人员和物体等通过新的方式连在一起，形成人与物、物与物相连，实现信息化、远程管理和智能化的网络。

物联网的学习是一个大的方向，学习物联网的知识以及相关的专业技能，要脚踏实地，一步一个脚印，多积累，多学习，慢慢的你就会攀登更高的山峰。我们通过很多年的物联网教学经验积累，结合现在学校教育的发展现状，制定了一系列科学有效的解决方案，最大限度降低物联网学习的门槛。使基础较弱的学生，也可以通过我们的方案和相应的配套教材和操作的视频，使之具备扎实的基础和熟练的专业技能，进入到物联网行业。基础好的学生也可以通过我们的方案和配套的教材，不断的提升自己的能力，不断利用自己想象力创造出新的东西，成为物联网开发的栋梁之才。

本书我们通过三篇带领大家认识我们物联网的开发过程，第一篇我们了解我们公司 NB-IoT 硬件开发套件，只有实战才能学到更多的东西，开发板就是我们强而有力的武器，我们给大家配备了丰富的开发板资源，不仅适合学习，也可以给大家二次开发用。第二篇我们给大家带来软件篇的介绍，我们使用开源的 STM32CubeMX，通过图形化的配置，简化步骤，配置更加动态和直观，可以自动生成一些底层的代码，减少了大家学习和开发的难度。第三篇就是我们的应用实战篇，带领大家从最基础的实验做起，一步一步提高，后面的实验也会综合前面实验的内容。让大家温故而知新，学习后面知识的同时，也不断应用前面学习的知识。同时我们配套的相应的文字解释，同时我们为了使大家更好的理解每一个例程的步骤以及详细的过程，我们配套更加详细的视频讲解，尽最大限度地做到手把手教大家做每一个实验。最后配套了一个综合的例程。我们一整套的资料，最终目的是让大家知其然更要知其所以然。

我们竭尽全给大家最全面的支持和帮助，希望大家我们的脚步，砥砺前行，勇往直前，在学习物联网上面，收获更多的知识。同样，我们在创新的道路上也从未停止过脚步。

目录

第一篇 硬件篇.....	1
1.1 STM32L1 开发板资源.....	2
1.1.1 STM32 开发板资源.....	2
第二篇 软件篇.....	4
1.1 手把手教你入门 STM32CubeMX 图形配置工具.....	5
1.1.1 STM32CubeMX 简介.....	5
1.1.2 STM32CubeMX 运行环境搭建.....	5
1.2.3 使用 STM32CubeMX 工具配置工程模板.....	9
第三篇 实战篇.....	15
第一章 LED 灯实验.....	16
1.1 STM32L1 I/O 简介.....	16
1.2 硬件设计.....	21
1.3 STM32CubeMX 配置 I/O 口输出和软件设计.....	22
1.4 下载验证.....	26
第二章 按键输入实验.....	27
2.1 STM32L1 I/O 口简介.....	27
2.2 硬件设计.....	27
2.3 STM32CubeMX 配置 I/O 口输出和软件设计.....	28
2.4 下载验证.....	30
第三章 串口通信实验.....	31
3.1 STM32L151 串口简介.....	31
3.2 硬件设计.....	36
3.3 STM32CubeMX 配置串口和软件设计.....	37
3.4 下载验证.....	45
第四章 外部中断实验.....	46
4.1 STM32L1 外部中断简介.....	46
4.2 硬件设计.....	49
4.3 STM32CubeMX 配置外部中断和软件设计.....	49
4.4 下载验证.....	54
第五章独立看门狗（IWDG）实验.....	55
5.1 STM32L1 独立看门狗简介.....	55
5.2 硬件设计.....	58
5.3 STM32CubeMX 配置 IWDG 和软件设计.....	58
5.4 下载验证.....	60
第六章定时器中断实验.....	61
6.1 STM32L151 通用定时器简介.....	61
6.2 硬件设计.....	66

6.3	STM32CubeMX 配置定时器更新中断功能和软件设计.....	66
6.4	下载验证.....	70
第七章	PWM 输出实验.....	71
7.1	PWM 简介.....	71
7.2	硬件设计.....	76
7.3	STM32CubeMX 配置定时器 PWM 输出功能和软件设计.....	76
7.4	下载验证.....	81
第八章	TFT_LCD 实验.....	82
8.1	TFTLCD&SPI 简介.....	82
8.2	硬件设计.....	89
8.3	STM32CubeMX 配置 SPI 和软件设计.....	89
8.4	下载验证.....	93
第九章	RTC 实时时钟实验.....	94
9.1	STM32L151 RTC 时钟简介.....	94
9.2	硬件设计.....	105
9.3	STM32CubeMX 配置 RTC 和软件设计.....	105
9.4	下载验证.....	110
第十章	待机唤醒实验.....	111
10.1	STM32L151 待机模式简介.....	111
10.2	硬件设计.....	114
10.3	STM32CubeMX 配置 WKUP 和软件设计.....	114
10.4	下载与测试.....	117
第十一章	ADC 实验.....	118
11.1	STM32L1 ADC 简介.....	118
11.2	硬件设计.....	125
11.3	STM32CubeMX 配置 ADC 和软件设计.....	125
11.4	下载验证.....	129
第十二章	IIC 实验.....	130
12.1	IIC 简介.....	130
12.2	硬件设计.....	130
12.3	STM32CubeMX 配置 IIC 和软件设计.....	131
12.4	下载验证.....	139
第十三章	温湿度传感器实验.....	140
13.1	IIC 简介.....	140
13.2	硬件设计.....	140
13.3	软件设计.....	141
13.4	下载验证.....	148
第十四章	SPI 实验.....	149
14.1	SPI 简介.....	149
14.2	硬件设计.....	149
14.3	STM32CubeMX 配置 SPI 和软件设计.....	151
14.4	下载验证.....	159

第十五章 485 实验.....	160
15.1 485 简介.....	160
15.2 硬件设计.....	161
15.3 STM32CubeMX 配置串口和软件设计.....	162
15.4 下载验证.....	164
第十六章 无源蜂鸣器实验.....	165
16.1 无源蜂鸣器简介.....	165
16.2 硬件设计.....	165
16.3 STM32CubeMX 配置定时器 PWM 输出功能和软件设计.....	166
16.4 下载验证.....	170
第十七章 汉字显示实验.....	171
17.1 汉字显示原理简介.....	171
17.2 硬件设计.....	172
17.3 软件设计.....	173
17.4 下载验证.....	175
第十八章 窄带物联网实验.....	176
18.1 窄带物联网简介.....	176
18.2 硬件设计.....	187
18.3 STM32CubeMX 配置和软件设计.....	189
18.4 下载验证.....	201

第一篇 硬件篇

实践出真知，要想学好 STM32L151，实验平台必不可少！本篇将详细介绍我们用来学习 STM32L151 的硬件平台，通过该篇的介绍，你将了解到我们的学习平台 STM32L151 开发板的功能及特点。

本章主要向大家简要介绍我们的实验平台：STM32L1 开发板。通过本章的学习，你将 对我们后面使用的实验平台有个大概了解，为后面的学习做铺垫。

1.1 STM32L1 开发板资源

1.1.1 STM32 开发板资源

首先，我们来看STM32 开发板的资源图，如图 1.1.1.1 所示：

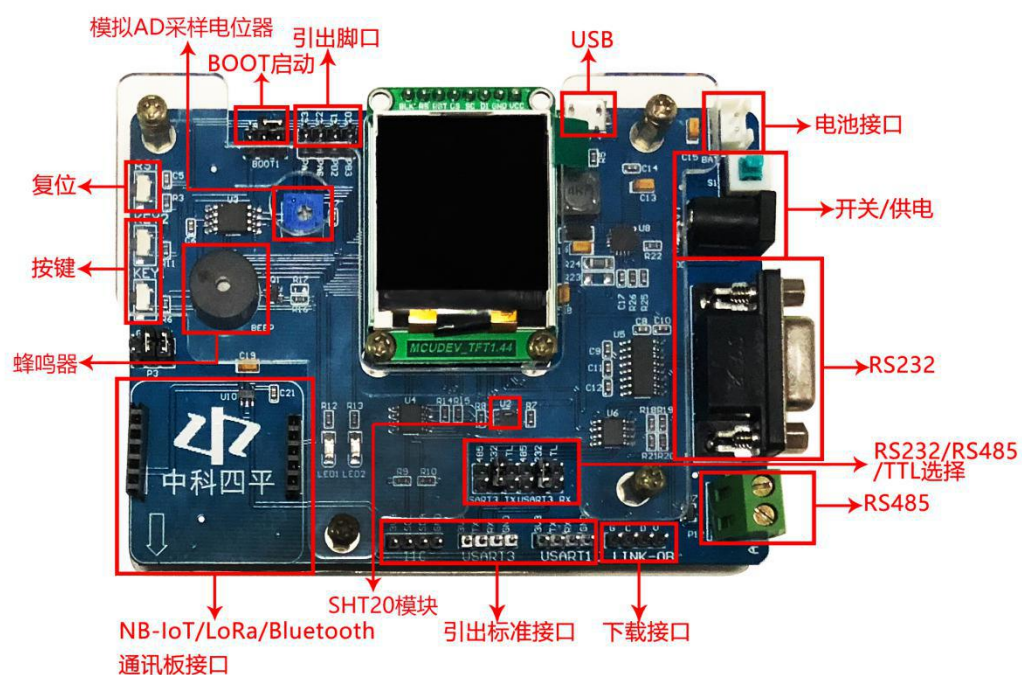


图 1.1.1.1 STM32 开发板

从图 1.1.1.1 可以看出，STM32L151 开发板底板，板载资源十分丰富，把 STM32L151 的内部资源发挥到了极致，基本所有STM32L151 的内部资源，都可以在此开发板上验证，同时扩充丰富的接口和功能模块，使之更加完善。

开发板的外形尺寸为 110mm*80mm 大小，板子的设计充分考虑了人性化设计，并结合多年的 STM32 开发板设计经验，经过多次改进，最终确定了这样的设计。

STM32L151 开发板板载资源如下：

- ◆ CPU：STM32L151R8T6，LQFP64，FLASH：64KB，SRAM：10KB
- ◆ 1 个复位按钮，可用于复位 MCU 和LCD
- ◆ 2 个功能按钮，其中 KEY_UP(即WK_UP)兼具唤醒功能
- ◆ 1 个双色电源充电指示灯（红绿色）
- ◆ 2 个 LED 指示灯（LED1：红色，LED2：蓝色）
- ◆ 1 路485 接口，采用SP3485 芯片
- ◆ 1 路RS232 串口接口，采用SP3232 芯片
- ◆ 1 个 OLED 模块接口
- ◆ 1 个无源蜂鸣器
- ◆ 1 个 SD 卡接口
- ◆ 一个备用电池接口
- ◆ 一个模拟 AD 采样电位器
- ◆ 1 组 5V 电源接口
- ◆ 1 个 IIC 接口温湿度传感器（SHT20）
- ◆ 引出 12 个 IO 口
- ◆ 外扩SPI FLASH：W25Q64，8M 字节
- ◆ 外扩 EEPROM：AT24C256，256 字节
- ◆ 1 个Micro USB 接口
- ◆ 1 个NB_IOT/WIFI/LORA 模块接口

STM32L151 开发板的特点包括：

- 1) 接口丰富。
- 2) 设计灵活。
- 3) 资源丰富。
- 4) 人性化设计。
- 5) 增加目前流行的几种物联网模块。

第二篇 软件篇

有了硬件部分的支持，我们也需要开发软件来进行学习和开发，下面这一章，你将了解到我们使用开源开发软件的操作过程，还有它的特性。大大降低学习和开发的难度。也是现在主流的一款软件开发工具。

1.1 手把手教你入门 STM32CubeMX 图形配置工具

本小节我们将给大家讲解 STM32CubeMX 相关知识，带领大家入门 STM32CubeMX 图形化配置工具。之所以我们要把 STM32CubeMX 讲解放在本小节，是因为 STM32CubeMX 最基本也是最重要的用途是配置时钟系统，所以我们要先讲解 STM32F151 的时钟系统之后，才能教大家学习 STM32CubeMX。这部分内容我们分 3 个小节来讲解：

- 1.1.1 STM32CubeMX 简介
- 1.1.2 STM32CubeMX 运行环境搭建
- 1.1.3 使用 STM32CubeMX 工具配置工程模板

1.1.1 STM32CubeMX 简介

STM32CubeMX 是 ST 意法半导体近几年来大力推荐的 STM32 芯片图形化配置工具，允许用户使用图形化向导生成 C 初始化代码，可以大大减轻开发工作，时间和费用。STM32CubeMX 几乎覆盖了 STM32 全系列芯片。它具有如下特性：

- ① 直观的选择 MCU 型号，可指定系列、封装、外设数量等条件
- ② 微控制器图形化配置
- ③ 自动处理引脚冲突
- ④ 动态设置时钟树，生成系统时钟配置代码
- ⑤ 可以动态设置外围和中间件模式和初始化
- ⑥ 功耗预测
- ⑦ C 代码工程生成器覆盖了 STM32 微控制器初始化编译软件，如 IAR，KEIL，GCC。
- ⑧ 可以独立使用或者作为 Eclipse 插件使用

对于 STM32CubeMX 和 STM32Cube 的关系这里我们还需要特别说明一下，STM32Cube 包含 STM32CubeMX 图形工具和 STM32Cube 库两个部分，使用 STM32CubeMX 配置生成的代码，是基于 STM32Cube 库的。也就是说，我们使用 STM32CubeMX 配置出来的初始化代码，和 STM32Cube 库兼容，例如硬件抽象层代码就是使用的 STM32 的 HAL 库。不同的 STM32 系列芯片，会有不同的 STM32Cube 库支持，而 STM32CubeMX 图形工具只有一种。所以我们配置不同的 STM32 系列芯片，选择不同的 STM32Cube 库即可。

它们之间的关系如下图 1.1.1.1：



图1.1.1.1 STM32CubeMX 和STM32Cube 关系

1.1.2 STM32CubeMX 运行环境搭建

STM32CubeMX 运行环境搭建包含两个部分。首先是 Java 运行环境安装，其次是 STM32CubeMX 软件安装。

对于Java 运行环境，大家可以到Java 官网www.java.com 下载最新的 Java 软件。这里大家需要注意，STM32CubeMX 的 Java 运行环境版本必须是 V1.7 及以上，如果你的电脑安装过V1.7 以下版本，请先删掉后重新安装最新版本。

对于 Java 运行环境安装，我们这里就不做过多讲解，大家直接双击安装包，根据提示安装即可。安装完成之后提示界面如下图1.1.2.1：

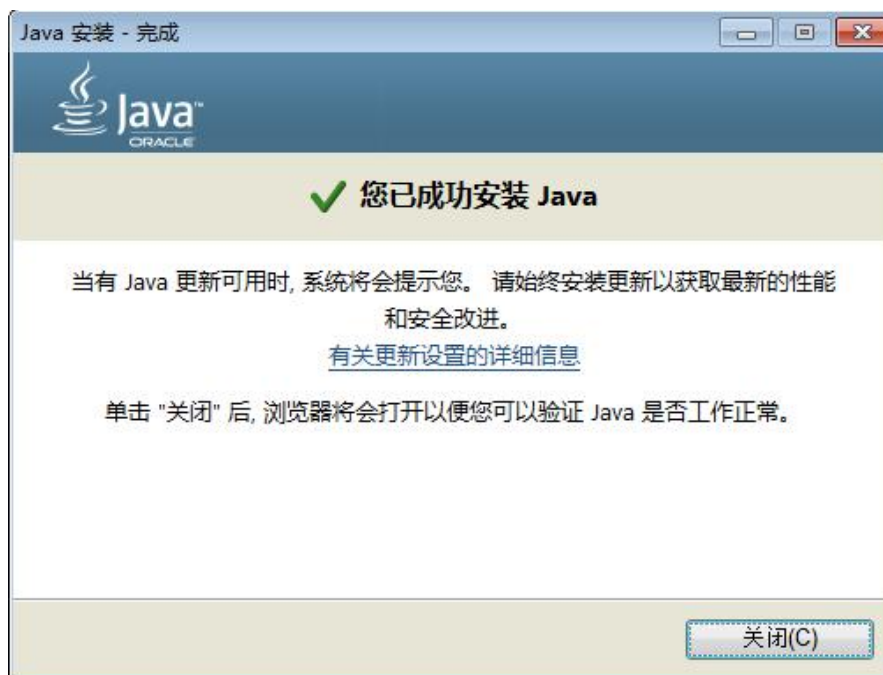


图1.1.2.1 Java 安装成功提示界面

安装完 Java 运行环境之后，为了检测是否正常安装，我们可以打开 Windows 的命令输入框，输入：java -version 命令，如果显示 Java 版本信息，则安装成功。提示信息如下图 1.1.2.2：

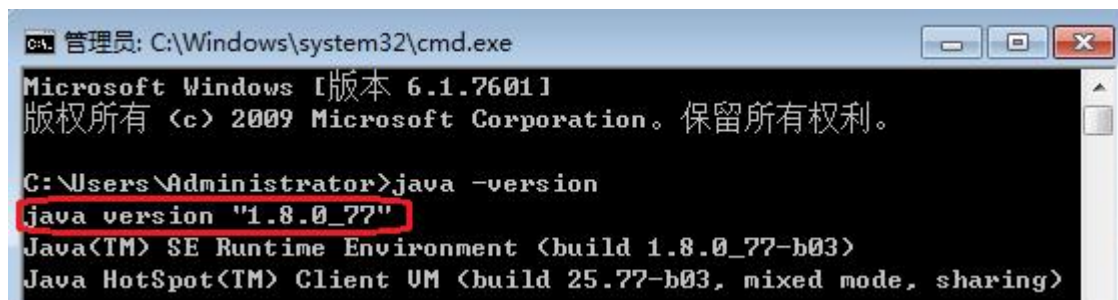


图1.1.2.2 查看Java 版本

在安装完 Java 运行环境之后，接下来我们安装 STM32CubeMX 图形化工具。可以直接从 ST 官方下载，下载地址为：www.st.com/stm32cube。

接下来我们直接双击 STM32CubeMX 安装包，根据提示信息安装即可。安装完成之后提示信息如下图 1.1.2.3 所示：

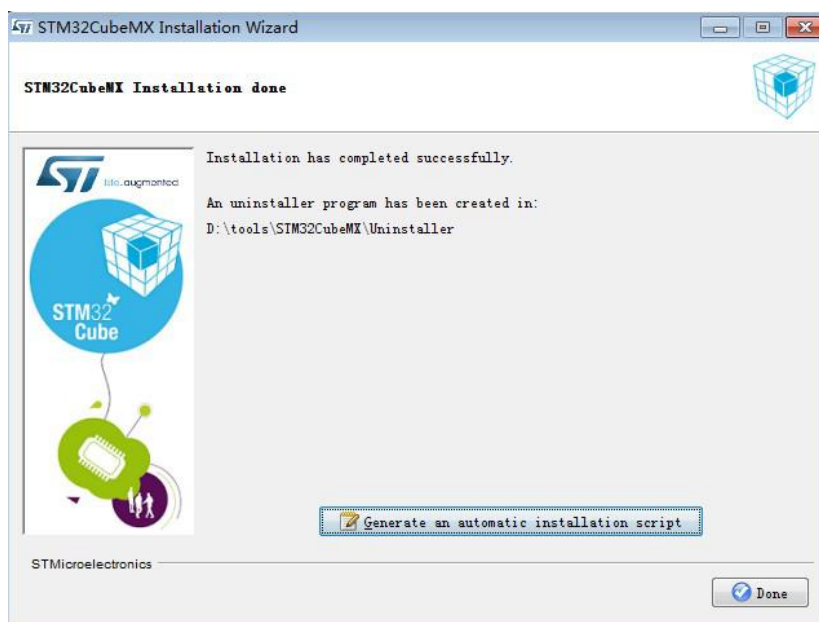


图 1.1.2.3 STM32CubeMX 安装完成的界面

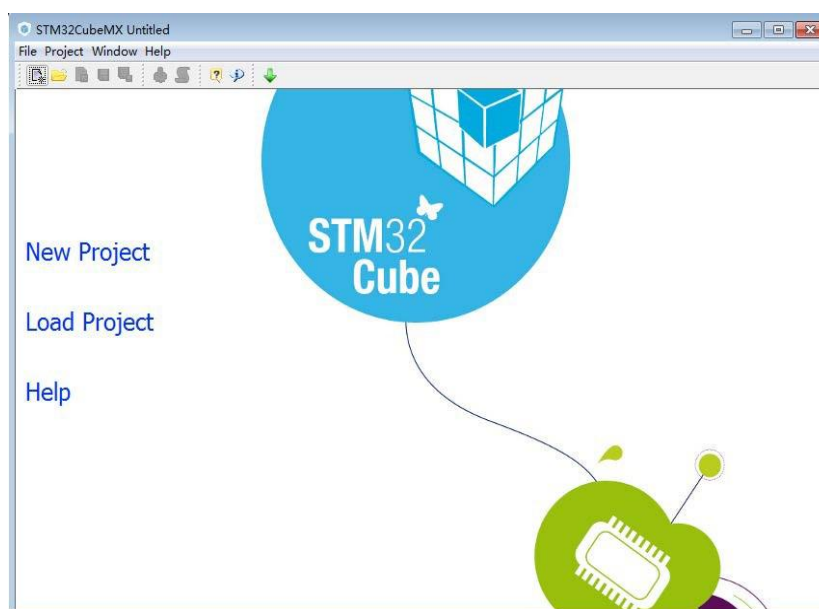


图1.1.2.4 STM32CubeMX 打开后的显示界面

在安装好 STM32CubeMX 之后，接下来我们要在软件中指定 STM32Cube 软件包。在 STM32CubeMX 操作界面，依次点击 Help->Updater Settings，弹出界面如下图 1.1.2.5 所示：

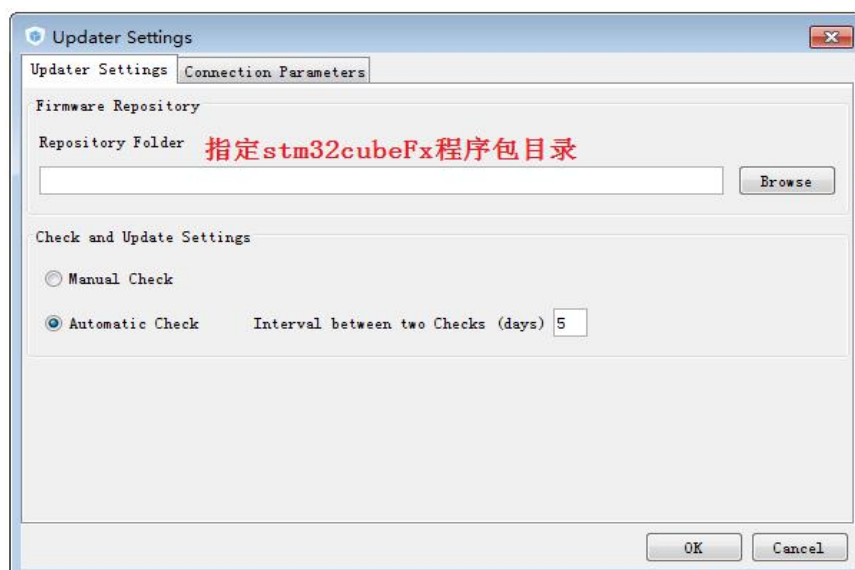


图1.1.2.5 Updater Settings 操作界面

在上图 1.1.2.5 中，我们只需要点击 Browse 按钮，定位到 stm32cubefx 存放目录即可。这里大家注意，stm32cubefx 文件夹名字遵循 STM32Cube_FW_Fx_Vm.n.z 格式，我们指的“Repository Folder”下面必须存在一个或者多个 STM32Cube_FW_Fx_Vm.n.z 格式程包，在 STM32CubeMX 生成工程的时候，会根据我们选择的芯片型号，去这个目录加载必要的库文件。一般情况下，我们会新建一个目录，然后把我们需要使用的各种 stm3cubefx 支持包解压放到该目录之下，然后把该目录指定为“Repository Folder”即可。操作方法如下图所示：

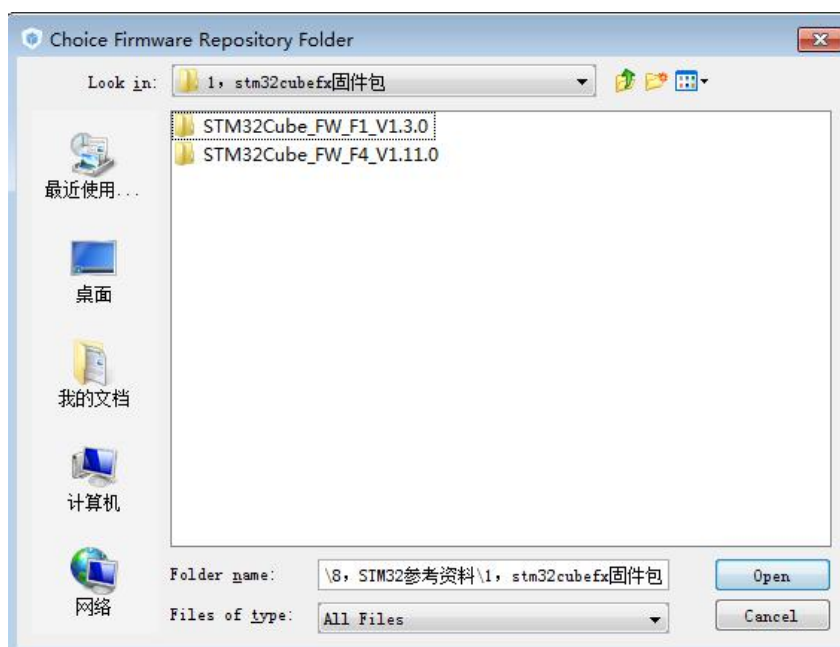


图 1.1.2.6 指定程序库目录

实际上，我们也可以直接在 STM32CubeMX 中点击 Help->Install New Libraries 下载需要的程序库，但是由于速度比较慢，而且在下载过程中很容易中断，所以我们不推荐直接在 CubeMX 中下载。

接下来我们将讲解怎么使用 STM32CubeMX 新建一个完整的 STM32L151 工程。

1.2.3 使用STM32CubeMX 工具配置工程模板

大多数情况下，我们都只使用 STM32CubeMX 来生成工程的时钟系统初始化代码以及外设的初始化代码，因为用户控制逻辑代码是无法在 STM32CubeMX 中完成的，需要用户自己根据需求来实现。使用 STM32CubeMX 配置工程的一般步骤为：

- 1) 工程初步建立和保存
- 2) RCC 设置
- 3) 时钟系统（时钟树）配置
- 4) 生成工程源码
- 5) 编写用户代码

接下来我们将按照上面 5 个步骤，依次教大家使用 STM32CubeMX 工具生成一个完整的工程。

1.2.3.1 工程初步建立和保存

工程建立的方法有两种方法，第一种方法是打开 STM32CubeMX 之后在主界面点击 New Project 按钮，第二种方法是在菜单栏依次点击File->New Project。操作方法如下图1.2.3.1.1所示：

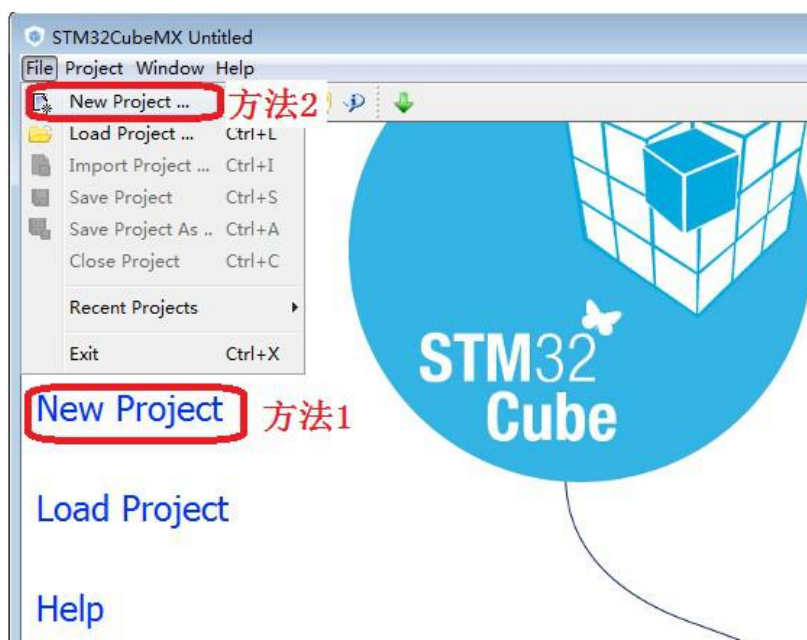
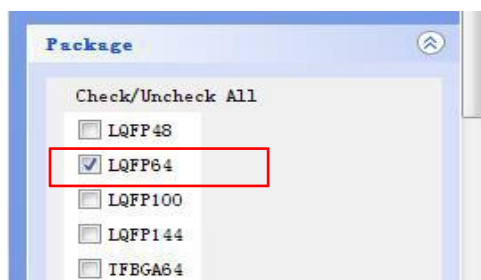
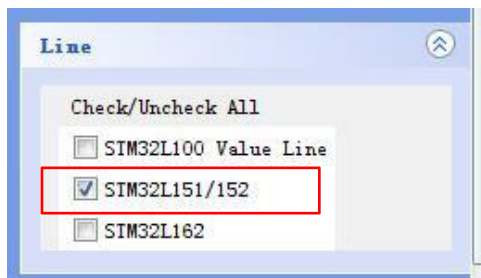
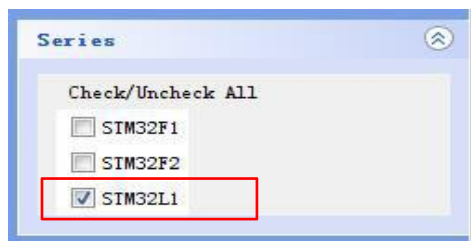
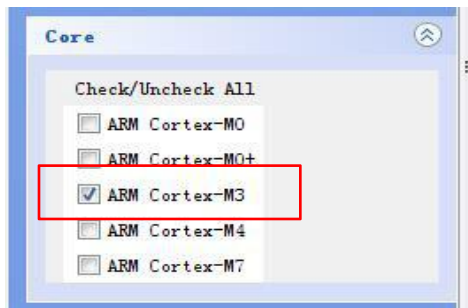


图 1.2.3.1.1 新建工程

点击新建工程按钮之后,会弹出 MCU 选择窗口。我们依次在选项卡 Series,Lines 和 Package 之下选择与我们使用的芯片 STM32L151 对应的参数,然后选择对应的芯片型号,最后点击 OK 按钮。操作方法如下图 1.2.3.1.2 所示:



MCUs List: 20 items + Display similar ...

Part No	Market	Unit Price f...	Package	Flash	RAM	IO	Freq.	DE...	HM...	M...	S...	TR...
SIM32L151R6	NRND	1.742	LQFP64	32 k...	10 k...	51	32 MHz	0	0	0	0	0
SIM32L151R6-A	Active	1.742	LQFP64	32 k...	16 k...	51	32 MHz	0	0	0	0	0
SIM32L151R8	NRND	1.958	LQFP64	64 k...	10 k...	51	32 MHz	0	0	0	0	0
SIM32L151R8-A	Active	1.958	LQFP64	64 k...	32 k...	51	32 MHz	0	0	0	0	0
SIM32L151RB	Active	2.195	LQFP64	128 ...	16 k...	51	32 MHz	0	0	0	0	0
SIM32L151RB-A	Active	2.195	LQFP64	128 ...	32 k...	51	32 MHz	0	0	0	0	0

为了避免在软件使用过程中出现意外导致工程没有保存，所以我们选择好芯片型号之后，先对工程进行保存。依次点击菜单栏File→Save Project，然后保存工程到某个文件夹下面即可。操作过程如下图1.2.3.1.3 所示：

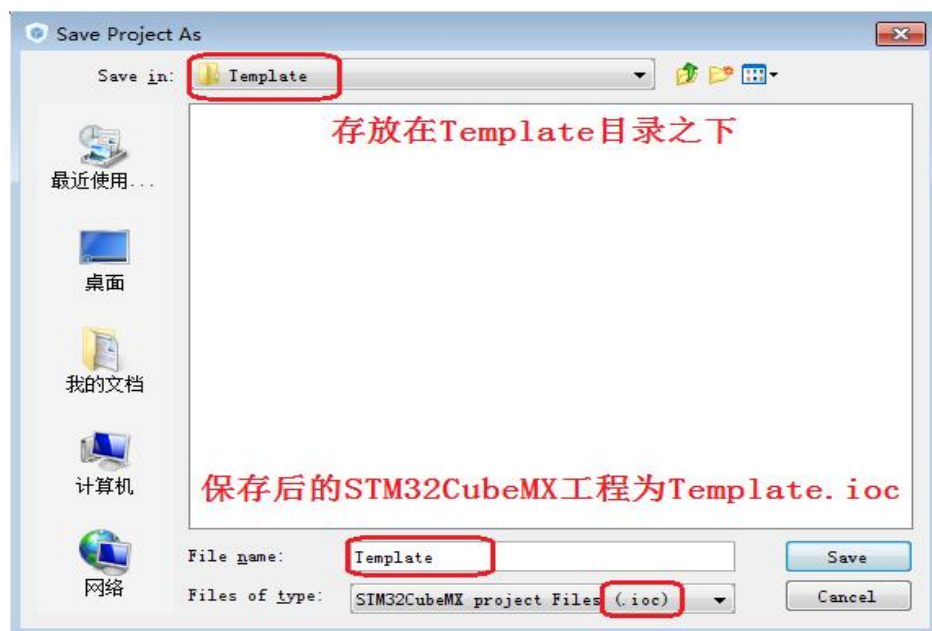


图 1.2.3.1.3 保存工程

保存完成之后，大家进入 Template 目录后发现目录中多了一个 Template.ioc 文件，下次我们点击这个文件就可以直接打开这个工程。

工程新建好之后会直接进入 Pinout 选项卡，这个时候界面会展示芯片完整引脚图，如下图 1.2.3.1.4 所示：

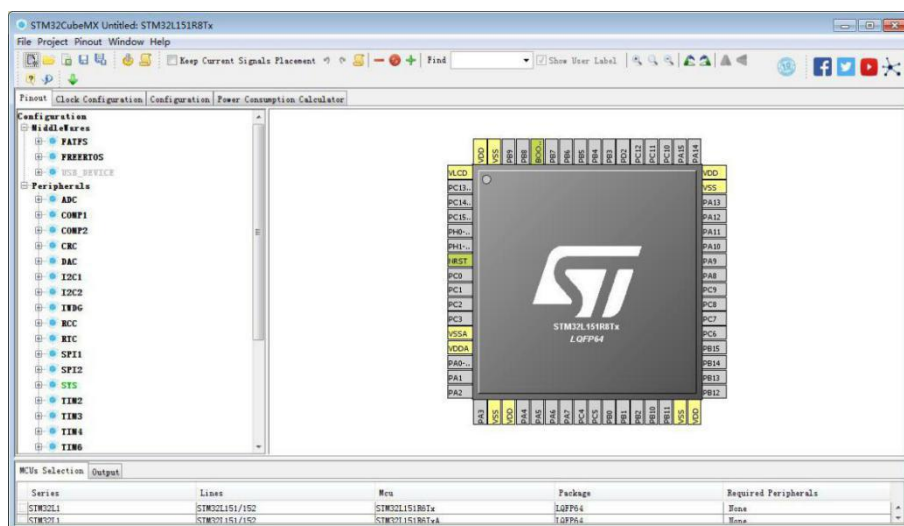


图1.2.3.1.4 STM32CubeMX 中芯片引脚图

在引脚图中，我们可以对引脚功能进行配置。图中黄色的引脚主要是一些电源和 GND 引脚，如果某个引脚被使用，那么会显示为绿色。

1.2.3.2 RCC 设置

对STM32芯片而言，RCC配置的重要性不言而喻。在STM32CubeMX中，RCC相关设置却非常简单，因为它把时钟系统独立出来配置。在操作界面，依次点击选项卡Pinout->Peripherals->RCC便可进入RCC配置栏，操作步骤如下图1.2.3.2.1所示：

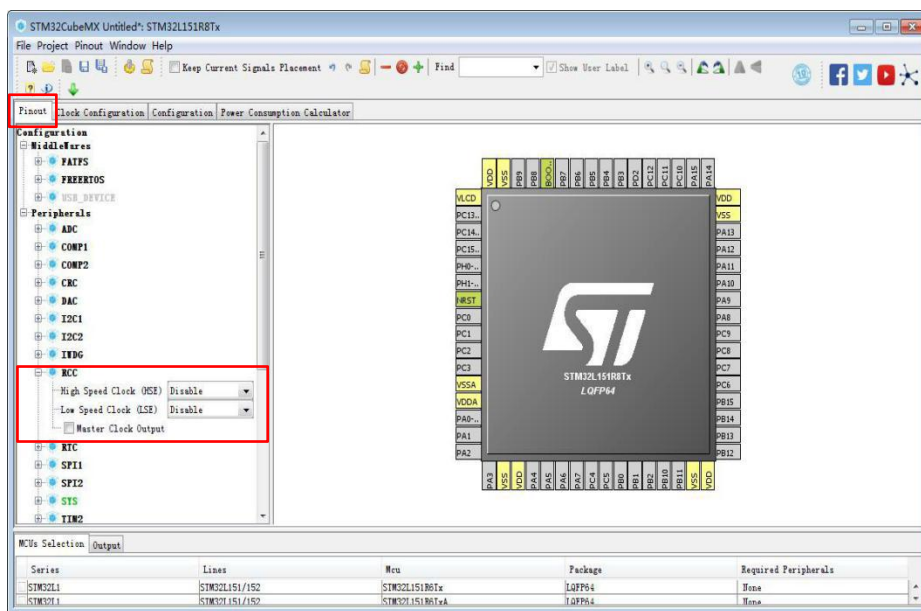


图 1.2.3.2.1 进入 RCC 配置栏

从上图可以看出，RCC配置栏实际上只有3个配置项。选项High Speed Clock（HSE）用来配置HSE，第二个选项Low Speed Clock（LSE）用来配置LSE，选项Master Clock Output用来选择是否使能MCO引脚时钟输出。

HSE：可接石英/陶瓷振荡器，或者接外部时钟源。

LSE：接频率为32.768kHz的石英晶体。

本小节我们只使用到HSE，所以我们设置选项High Speed Clock（HSE）的值为Crystal/Ceramic Resonator（使用晶振/陶瓷振荡器）即可。这里还需要说明一下，值Bypass Clock Source的意思是旁路时钟源，也就是不使用使用晶振/陶瓷振荡器，直接通过外部提供一个可靠的4-26MHz时钟作为HSE。配置好的RCC配置选项如下图1.2.3.2.2所示：

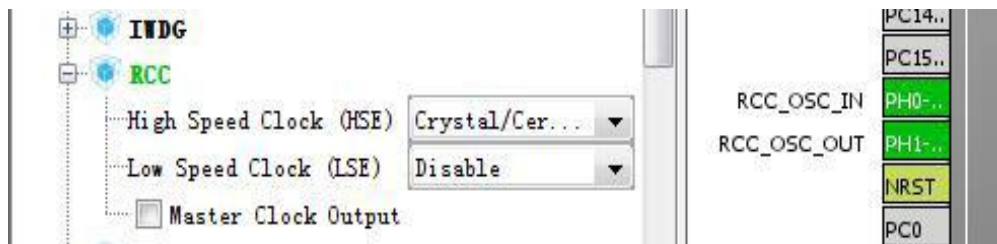


图 1.2.3.2.2 RCC 选项配置

从上图还可以看出，在我们打开了HSE之后，右边的引脚图中，相应的引脚会由灰色变为绿色，表示该引脚已经被使用。配置完RCC之后，接下来我们来看看配置时钟系统树的方法。

1.2.3.3 时钟系统（时钟树）配置

在使用 STM32CubeMX 配置时钟树之前，大家需要充分理解 STM32 时钟系统，这在我们前面 4.3 小节有非常详细的讲解，只有熟练掌握了 STM32 时钟系统，那么在软件中配置时钟树才会得心应手。

点击 Clock Configuration 选项卡即可进入时钟系统配置栏，如下图 1.2.3.3.1 所示：

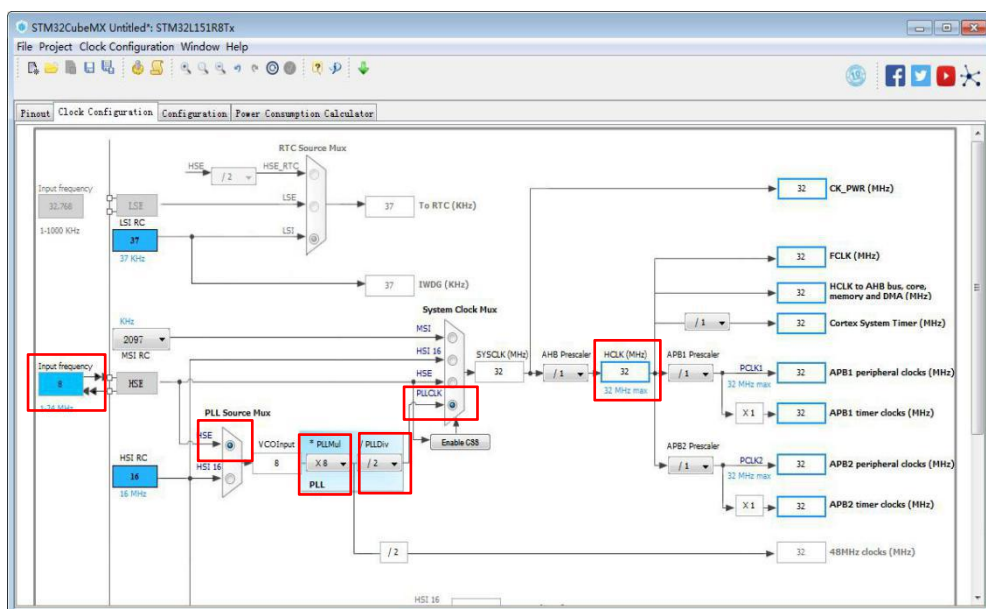


图 1.2.3.3.2 系统时钟配置图

我们把系统时钟配置分为 5 个步骤，分别用标号①~⑤表示，详细过程为：

- ① 时钟源参数设置：HSE 或者 HSI 配置。这里我们选择 HSE 为时钟源，所以我们之前必须在 RCC 配置中我们开启 HSE。
- ② 时钟源选择：HSE 还是 HSI。这里我们配置选择器选择 HSE 即可。
- ③ 主 PLL 倍频系数 N 配置。倍频系数 N 我们设置为 8。
- ④ 主 PLL 分频系数 P 配置。分频系数 P 我们配置为 2。
- ⑤ 系统时钟时钟源选择：PLL, HSI 还是 HSE。这里毫无疑问，我们选择 PLL，选择器选择 PLLCLK 即可。

经过上面的 5 个步骤，就会生成标准的 32MHz 系统时钟。

注：PLL：为锁相环倍频输出，其时钟输入源可选择为 HSI/2、HS 或者 HSE/2。倍频可选择为 2~16 倍，但是其输出频率最大不得超过 72MHz。

1.2.3.5 生成工程源码

接下来，我们将使用 STM32CubeMX 生成我们需要的工程源码。在 STM32CubeMX 操作界面，依次点击菜单 Project->Generate Code 即可生成源码，操作方法如下图 1.2.3.5.1 所示：

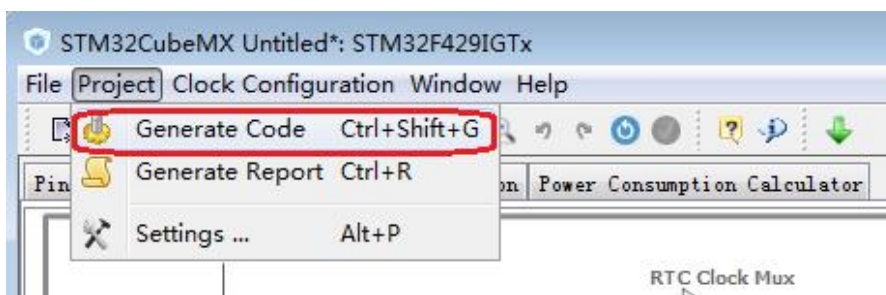


图1. 2. 3. 5. 1

点击Generate Code 选项点击之后，弹出的界面会要求配置生成的工程名称，保存目录以及使用的编译软件类型。我们依次填写工程名称和保存目录即可，对于编译软件我们选择 MDK5 即可。操作过程如下图 1. 2. 3. 5. 2 所示：

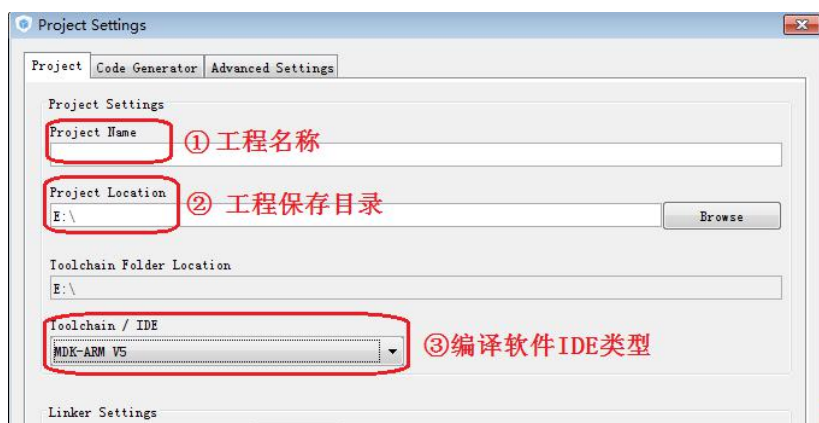


图 1. 2. 3. 5. 2 工程参数配置

配置完成后，点击OK 开始生产源码。源码生产完成之后，就保存在我们Project Location 选项配置的目录中，同时弹出生成成功提示界面，我们可以点击界面的“Open Folder” 按钮打开工程保存目录，也可以点击界面的“Open Project” 按钮直接使用 IDE 打开工程。提示界面如下图 1. 2. 3. 5. 3 所示：

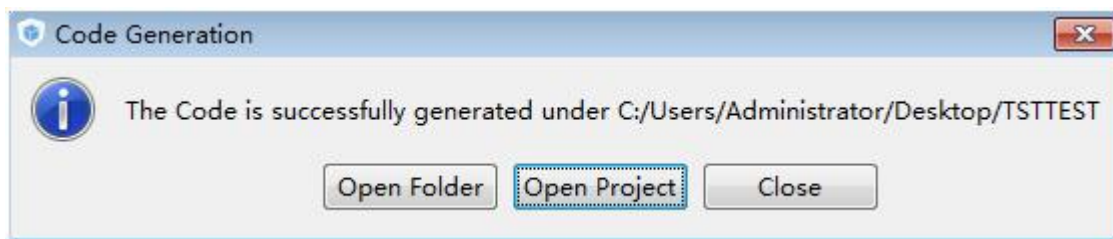


图 1. 2. 3. 5. 3 代码生成后提示界面

至此，一个完整的STM32L151 工程就已经生成完

第三篇 实战篇

经过前两篇的学习，我们对 STM32L1 开发的软件和硬件平台都有了个比较深入的了解了，接下来我们将通过实例，由浅入深，带大家一步步的学习 NB 开发板。

STM32L1 的内部资源非常丰富，对于初学者来说，一般不知道从何开始。本篇将从 STM32L1 最简单的外设说起，然后一步步深入。每一个实例都配有详细的操作步骤，以及图形和代码的解释，手把手教你如何入手 STM32L1 的各种外设，通过本篇的学习，希望大家能学会 STM32L1 绝大部分外设的使用。本篇总共分为 18 章，每一章即一个例程，下面就让我们开始精彩的 STM32L1 之旅。

第一章 LED 灯实验

任何一个单片机，最简单的操作莫过于 I/O 口的高低电平控制了，本章将通过一个经典的点亮 LED 程序，带大家开启 STM32L151 之旅，通过本章的学习，你将了解到 STM32L151 I/O 口作为输出使用的方法。在本章中，我们将通过代码控制 STM32 开发板上的两个 LED 灯 LED1 和 LED2 交替闪烁，实现类似跑马灯的效果。本章分为如下五个小节：

- 1.1 STM32L1 I/O 口简介
- 1.2 硬件设计
- 1.3 STM32CubeMX 配置 I/O 口输出和软件设计
- 1.4 下载验证

1.1 STM32L1 I/O 简介

本章将要实现的是控制 STM32 开发板上的两个 LED 实现一个类似跑马灯的效果，该实验的关键在于如何控制 STM32L151 的 I/O 口输出。了解了 STM32L151 的 I/O 口如何输出的，就可以实现跑马灯了。通过这一章的学习，你将初步掌握 STM32L151 基本 I/O 口的使用，而这是迈向 STM32L151 的第一步。

我们在讲解 HAL 库之前会首先对重要寄存器进行讲解，这样是为了大家对寄存器有个初步的了解，大家学习 HAL 库，并不需要记住每个寄存器的作用，而只是通过了解寄存器来对外设一些功能有基本的了解，这样对以后的学习也很有帮助。

STM32L1 每组通用 I/O 端口包括 4 个 32 位配置寄存器 (MODER、OTYPER、OSPEEDR 和 PUPDR)、2 个 32 位数据寄存器 (IDR 和 ODR)、1 个 32 位位置位/复位寄存器 (BSRR)、1 个 32 位锁定寄存器 (LCKR) 和 2 个 32 位复用功能选择寄存器 (AFRH 和 AFRL) 等。

这样，STM32L1 每组 I/O 有 10 个 32 位寄存器控制，其中常用的有 4 个配置寄存器+2 个数据寄存器+2 个复用功能选择寄存器，共 8 个，如果在使用的时候，每次都直接操作寄存器配置 I/O，代码会比较多，也不容易记住，所以我们在讲解寄存器的同时会讲解是用库函数配置 I/O 的方法。

STM32L1 的 I/O 可以由软件配置成如下 8 种模式中的任何一种：

- 1、输入浮空：线路两端连接着发送端和接收端，他们都需要准确获取对方的信号电平，不需要外界的干预。（主要用于来做 I2C，USART 的实验）
- 2、输入上拉：I/O 内部上拉电阻输入（输入高电平）
- 3、输入下拉：I/O 内部下拉电阻输入（输入低电平）
- 4、模拟输入：不经过输入数据寄存器，所以我们无法通过读取输入数据寄存器来获取模拟输入的值，我觉得这一点也是很好理解的，因为输入数据寄存器中存放的不是 0 就是 1，而模拟输入信号不符合这一要求，所以自然不能放进输入数据寄存器（主要用于 ADC 外设的时候）
- 5、开漏输出：I/O 输出 0 接 GND，I/O 输出 1，悬空，需要外接上拉电阻，才能实现输出高电平。当输出为 1 时，I/O 口的状态由上拉电阻拉高电平，但由于是开漏输出模式，这样 I/O 口也就可以由外部电路改变为低电平或不变。
- 6、推挽输出：推挽输出使用了推挽电路，结合推挽电路的特性，它是由两个 MOSFET（三极管）组成，一个导通的同时，另外一个截至，两个 MOSFET 分别连接高低电平，所以哪一个导通就会输出相应的电平。推挽电路速度快，输出能力强，直接输出高电平或者低电平。
- 7、推挽式复用功能：复用模式

8、开漏式复用功能：复用模式

接下来我们详细介绍 IO 配置常用的 8 个寄存器：MODER、OTYPER、OSPEEDR、PUPDR、ODR、IDR、AFRH 和 AFRL。同时讲解对应的 HAL 库配置方法。

首先看 MODER 寄存器，该寄存器是 GPIO 端口模式控制寄存器，用于控制 GPIOx（STM32F4 最多有 9 组 IO，分别用大写字母表示，即 x=A/B/C/D/E/F/G/H/I，下同）的工作模式，该寄存器各位描述如表 1.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MODER15[1:0]		MODER14[1:0]		MODER13[1:0]		MODER12[1:0]		MODER11[1:0]		MODER10[1:0]		MODER9[1:0]		MODER8[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MODER7[1:0]		MODER6[1:0]		MODER5[1:0]		MODER4[1:0]		MODER3[1:0]		MODER2[1:0]		MODER1[1:0]		MODER0[1:0]	
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

MODERy[1:0]: 端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 方向模式。

00: 输入（复位状态）

01: 通用输出模式

10: 复用功能模式

11: 模拟模式

表1.1.1 GPIOx MODER 寄存器各位描述

该寄存器各位在复位后，一般都是 0（个别不是 0，比如 JTAG 占用的几个 IO 口），也就是默认条件下一般是输入状态的。每组 IO 下有 16 个 IO 口，该寄存器共 32 位，每 2 个位控制 1 个 IO。

然后看 OTYPER 寄存器，该寄存器用于控制 GPIOx 的输出类型，该寄存器各位描述见表 1.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

位 31:16 保留，必须保持复位值。

位 15:0 **OTy[1:0]:** 端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 端口的输出类型。

0: 输出推挽（复位状态）

1: 输出开漏

表1.1.2 GPIOx OTYPER 寄存器各位描述

该寄存器仅用于输出模式，在输入模式（MODER[1:0]=00/11 时）下不起作用。该寄存器低 16 位有效，每一个位控制一个 IO 口。设置为 0 是推挽输出，设置为 1 是开漏输出。复位后，该寄存器值均为 0，也就是在输出模式下 IO 口默认为推挽输出。

然后看 OSPEEDR 寄存器，该寄存器用于控制 GPIOx 的输出速度，该寄存器各位描述见表 1.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
OSPEEDR15[1:0]	OSPEEDR14[1:0]	OSPEEDR13[1:0]	OSPEEDR12[1:0]	OSPEEDR11[1:0]	OSPEEDR10[1:0]	OSPEEDR9[1:0]	OSPEEDR8[1:0]	OSPEEDR7[1:0]	OSPEEDR6[1:0]	OSPEEDR5[1:0]	OSPEEDR4[1:0]	OSPEEDR3[1:0]	OSPEEDR2[1:0]	OSPEEDR1[1:0]	OSPEEDR0[1:0]
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OSPEEDR7[1:0]	OSPEEDR6[1:0]	OSPEEDR5[1:0]	OSPEEDR4[1:0]	OSPEEDR3[1:0]	OSPEEDR2[1:0]	OSPEEDR1[1:0]	OSPEEDR0[1:0]								
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

OSPEEDRy[1:0]: 端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 输出速度。

00: 2 MHz (低速)

01: 25 MHz (中速)

10: 50 MHz (快速)

11: 30 pF 时为 100 MHz (高速) (15 pF 时为 80 MHz 输出 (最大速度))

表1.1.3 GPIOx OSPEEDR 寄存器各位描述

该寄存器也仅用于输出模式，在输入模式 (MODER[1:0]=00/11 时) 下不起作用。该寄存器每2个位控制一个IO 口，复位后该寄存器值一般为0。

然后看PUPDR 寄存器，该寄存器用于控制GPIOx 的上拉/下拉，该寄存器各位描述见表

1.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
PUPDR15[1:0]	PUPDR14[1:0]	PUPDR13[1:0]	PUPDR12[1:0]	PUPDR11[1:0]	PUPDR10[1:0]	PUPDR9[1:0]	PUPDR8[1:0]	PUPDR7[1:0]	PUPDR6[1:0]	PUPDR5[1:0]	PUPDR4[1:0]	PUPDR3[1:0]	PUPDR2[1:0]	PUPDR1[1:0]	PUPDR0[1:0]
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PUPDR7[1:0]	PUPDR6[1:0]	PUPDR5[1:0]	PUPDR4[1:0]	PUPDR3[1:0]	PUPDR2[1:0]	PUPDR1[1:0]	PUPDR0[1:0]								
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

PUPDRy[1:0]: 端口 x 配置位 (Port x configuration bits) (y = 0..15)

这些位通过软件写入，用于配置 I/O 上拉或下拉。

00: 无上拉或下拉

01: 上拉

10: 下拉

11: 保留

表1.1.4 GPIOx PUPDR 寄存器各位描述

该寄存器每 2 个位控制一个 IO 口，用于设置上下拉，这里提醒大家，STM32L1 是通过 ODR 寄存器控制上下拉的，而 STM32F4 则由单独的寄存器 PUPDR 控制上下拉，使用起来更加灵活。复位后，该寄存器值一般为 0。

前面，我们讲解了 4 个重要的配置寄存器。顾名思义，配置寄存器就是用来配置 GPIO 的相关模式和状态，接下来我们讲解怎么在 HAL 库中初始化 GPIO 配置。

GPIO 相关的函数和定义分布在 HAL 库文件 stm32l1xx_hal_gpio.c 和头文件 stm32l1xx_hal_gpio.h 文件中。

在 HAL 库中，操作四个配置寄存器初始化 GPIO 是通过 HAL_GPIO_Init 函数完成：

```
void HAL_GPIO_Init(GPIO_TypeDef *GPIOx, GPIO_InitTypeDef *GPIO_Init);
```

该函数有两个参数，第一个参数是用来指定需要初始化的 GPIO 对应的 GPIO 组，取值范围为 GPIOA~GPIOK。第二个参数为初始化参数结构体指针，结构体类型为 GPIO_InitTypeDef。下面我们看看这个结构体的定义。首先我们打开我们的跑马灯实验，然后找到 HALLIB 组下面的 stm32f4xx_hal_gpio.c 文件，定位到 HAL_GPIO_Init 函数体处，双击入口参数类型 GPIO_InitTypeDef 后右键选择“Go to definition of ...”可以查看结构体的定义如下：


```
typedef struct
{
    uint32_t Pin;      //指定IO
    uint32_t Mode;     //模式设置
    uint32_t Pull;     //上下拉设置
    uint32_t Speed;    //速度设置
    uint32_t Alternate; //复用映射配置
}GPIO_InitTypeDef;
```

下面我们通过一个 GPIO 初始化实例来讲解这个结构体的成员变量的含义。初始化 GPIO 的常用格式是：

```
GPIO_InitTypeDef GPIO_InitStructure
GPIO_InitStructure.Pin=GPIO_PIN_0;      //PB0
```

```
GPIO_InitStructure.Mode=GPIO_MODE_OUTPUT_PP; //推挽
GPIO_InitStructure.Pull=GPIO_PULLUP;         //上拉
GPIO_InitStructure.Speed=GPIO_SPEED_HIGH;    //高速
HAL_GPIO_Init(GPIOB,&GPIO_InitStructure);
```

上面代码的意思是设置 PB0 端口为推挽输出模式，输出速度为高速，上拉。

从上面初始化代码可以看出，结构体 GPIO_InitStructure 的第一个成员变量 Pin 用来设置是要初始化哪个或者哪些 IO 口。第二个成员变量 Mode 是用来设置对应 IO 端口的输出输入端口模式，这个变量实际配置的是我们前面讲解的 GPIOx 的 MODER 寄存器。第三个成员变量 Pull 是用来设置上拉还是下拉，配置的是 GPIOx PUPDR 寄存器。第四个成员变量 Speed 用来设置输出速度，配置的是 GPIOxOSPEEDR 寄存器。第五个成员变量 Alternate 是用来设置引脚的复用映射的。

看完了 GPIO 的参数配置寄存器，接下来我们看看 GPIO 输入输出电平控制相关的寄存器。首先我们看 ODR 寄存器，该寄存器用于控制 GPIOx 的输出电平，该寄存器各位描述见表 1.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW	rW

位 31:16 保留，必须保持复位值。

位 15:0 **ODRy[15:0]**: 端口输出数据 (Port output data) (y = 0..15)

这些位可通过软件读取和写入。

表1.1.5 GPIOx ODR 寄存器各位描述

该寄存器用于设置某个 IO 输出低电平 (ODRy=0) 还是高电平 (ODRy=1)，该寄存器也仅在输出模式下有效，在输入模式 (MODER[1:0]=00/11 时) 下不起作用。该寄存器在HAL 库中使用不多，操作这个寄存器的库函数主要是HAL_GPIO_TogglePin 函数：

```
void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

该函数是通过操作 ODR 寄存器，达到取反 IO 口输出电平的功能。接下来我们看看另一个非常重要的寄存器 BSRR，它叫置位/复位寄存器。该寄存器和 ODR 寄存器具有类似的作用，都可以用来设置GPIO 端口的输出位是1 还是0。寄存器描述如下：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位 31:16 **BRy**：端口 x 复位位 y (Port x reset bit y) (y = 0..15)

这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。

0：不会对相应的 ODRx 位执行任何操作

1：对相应的 ODRx 位进行复位

注意：如果同时对 BSx 和 BRx 置位，则 BSx 的优先级更高。

位 15:0 **BSy**：端口 x 置位位 y (Port x set bit y) (y= 0..15)

这些位为只写形式，只能在字、半字或字节模式下访问。读取这些位可返回值 0x0000。

0：不会对相应的 ODRx 位执行任何操作

1：对相应的 ODRx 位进行置位

表1.1.6 BSRR 寄存器各位描述

对于低 16 位 (0-15)，我们往相应的位写 1，那么对应的 IO 口会输出高电平，往相应的位写 0，对 IO 口没有任何影响。高 16 位 (16-31) 作用刚好相反，对相应的位写 1 会输出低电平，写 0 没有任何影响。也就是说，对于 BSRR 寄存器，你写 0 的话，对 IO 口电平是没有任何影响的。我们要设置某个 IO 口电平，只需要相关位设置为 1 即可。而 ODR 寄存器，我们要设置某个 IO 口电平，我们首先需要读出来 ODR 寄存器的值，然后对整个 ODR 寄存器重新赋值来达到设置某个或者某些 IO 口的目的，而 BSRR 寄存器，我们就不需要先读，而是直接设置即可，这在多任务实时操作系统中作用很大。

BSRR 寄存器使用方法如下：

```
GPIOA->BSRR=1<<1; //设置GPIOA.1 为高电平
```

```
GPIOA->BSRR=1<<(16+1) //设置GPIOA.1 为低电平;
```

库函数操作BSRR 寄存器来设置IO 电平的函数为：

```
void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin,
                        GPIO_PinState PinState);
```

该函数用来设置一组IO 口中的一个或者多个IO 口的电平状态。比如我们要设置GPIOB. 5 输出高，方法为：

```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET); //GPIOB. 5 输出高
```

设置GPIOB. 5 输出低电平，方法为：

```
HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_RESET); //GPIOB. 5 输出低
```

接下来我们看看 IDR 寄存器，该寄存器用于读取 GPIOx 的输入数据，该寄存器各位描述见表 1.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留，必须保持复位值。

位 15:0 **IDRy[15:0]**：端口输入数据 (Port input data) ($y = 0..15$)

这些位为只读形式，只能在字模式下访问。它们包含相应 I/O 端口的输入值。

表1.1.7 GPIOx IDR 寄存器各位描述

该寄存器用于读取某个 IO 的电平，如果对应的位为 0 ($IDRy=0$)，则说明该 IO 输入的是低电平，如果是 1 ($IDRy=1$)，则表示输入的是高电平。HAL 库操作该寄存器读取 IO 输入数据相关函数：

```
GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
```

该函数用来读取一组 IO 下一个或者多个 IO 口电平状态。比如我们要读取 GPIOF.5 的输入电平，方法为：

```
HAL_GPIO_ReadPin(GPIOF, GPIO_PIN_5); //读取PF5 的输入电平
```

该函数返回值就是 IO 口电平状态。

最后我们来看看 2 个 32 位复用功能选择寄存器 (AFRH 和 AFRL)，这两个寄存器是用来设置 IO 口的复用功能的。实际上，在我们调用函数 HAL_GPIO_Init 的时候，如果我们设置了初始化结构体成员变量 Mode 为复用模式，同时设置了 Alternate 的值，那么会在该函数内部自动设置这两个寄存器的值，达到设置端口复用映射的目的。

GPIO 相关的函数我们先讲解到这里。虽然 IO 操作步骤很简单，这里我们还是做个概括性的总结，操作步骤为：

- 1) 使能 IO 口时钟，调用函数为 HAL_RCC_GPIOX_CLK_ENABLE(其中 $X=A\sim K$)。
- 2) 初始化 IO 参数。调用函数 HAL_GPIO_Init();
- 3) 操作 IO 输入输出。操作 IO 的方法就是上面我们讲解的方法。

上面我们讲解了 STM32L1IO 口的基本知识以及 HAL 库操作 GPIO 的一些函数方法，下面我们来讲解我们的跑马灯实验的硬件和软件设计。

1.2 硬件设计

本章用到的硬件只有 LED (LED1 和 LED2)。电路在 STM32 开发板上默认是已经连接好了的。LED1 接 PB0，LED2 接 PB1。所以在硬件上不需要动任何东西。如图 1.2.1

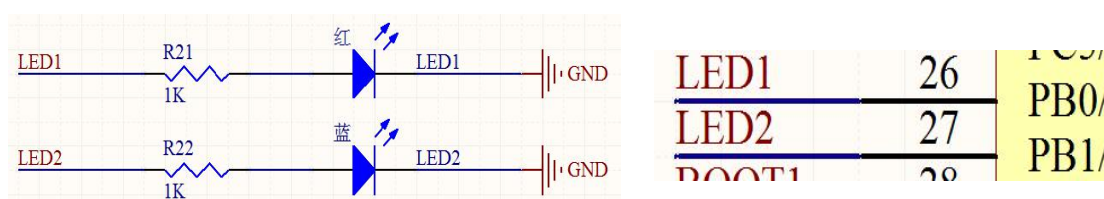


图1.2.1 LED 与STM32L151 连接原理图

1.3 STM32CubeMX 配置 IO 口输出和软件设计

本小节我们教大家怎么使用 STM32CubeMX 图形化配置工具配置 GPIO 初始化过程。使用 STM32CubeMX 配置 GPIO 口的步骤如下：

第一步，打开 STM32CubeMX 工具，在引脚图中选择要配置的 IO 口。这里我们选择 PB0 为例，在弹出的下拉菜单中选择要配置的 IO 口模式，如下图 1.3.1 所示：

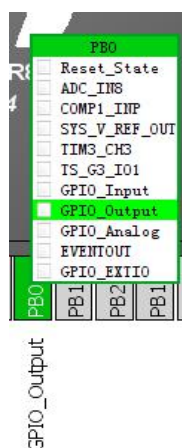


图1.3.1 选择IO 口模式

从上图可以看出，这里我们除了配置 IO 口为输入输出之外，还可以选择 IO 口的复用功能或者作为外部中断引脚功能。对于本章跑马灯实验，PB0 是作为输出，所以我们选 GPIO_Output 即可。我们也可以右键输入该引脚的 User Label，分别设置为 LED1 和 LED2，为防止后面管脚开启太多的时候，出现错误。

第二部，进入 Configuration->GPIO，在弹出的界面配置 IO 口的详细参数。如下图 1.3.2 和图 1.3.3 所示：

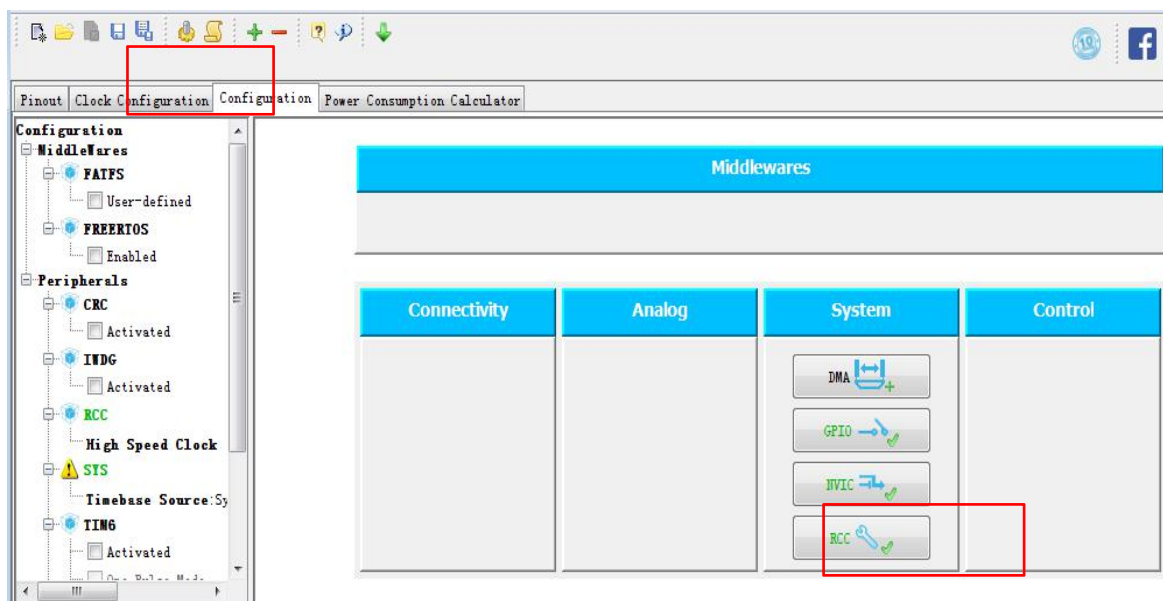


图1.3.2 进入GPIO 详细参数配置界

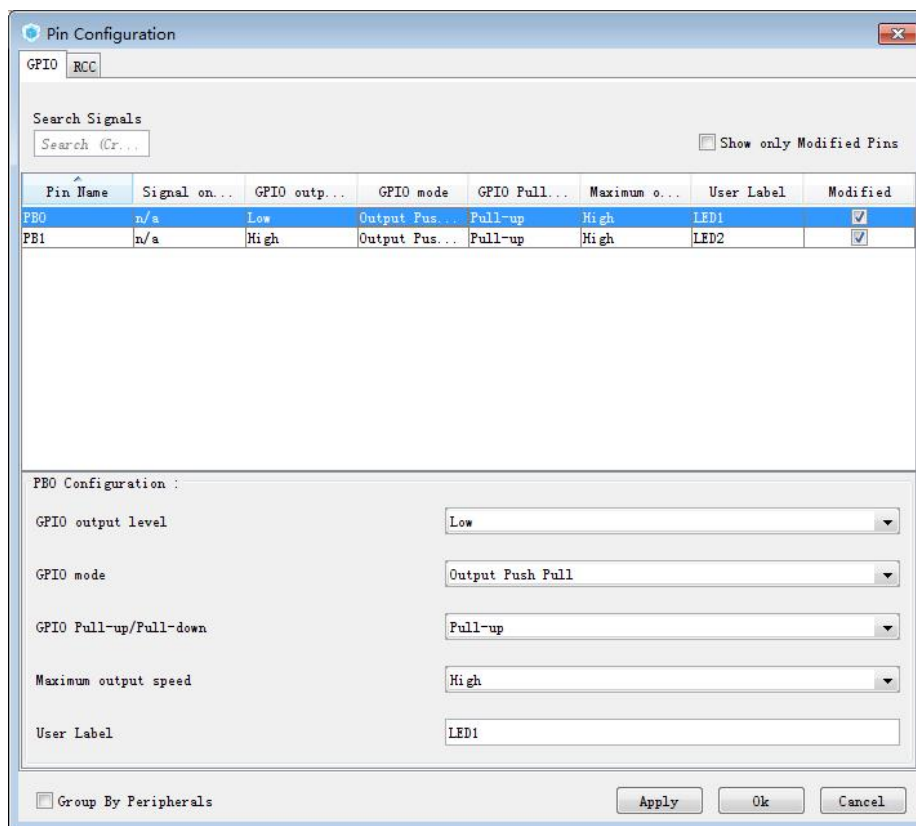


图1.3.3 配置I/O口详细参数

在 I/O 口详细参数配置界面，点击我们要配置的 I/O 口，会在窗口下方显示该 I/O 口配置の詳細参数表下面我们依次来解释这些配置项的含义：

① 选项 GPIO output level 用来设置 I/O 口初始化电平状态为 High(高电平)还是 Low(低电平)。本实验我们设置为默认输出低 LOW。

② 选项 GPIO mode 用来设置输出模式为 Output Push Pull(推挽)还是 Output Open Drain(开漏)。本实验我们设置为推挽输出 Output Push Pull。

③ 选项 GPIO Pull-up/Pull-down 用来设置 I/O 口是上拉/下拉/没有上下拉。本实验我们设置为上拉 (Pull-up)。

④ 选项 Maximum output speed 用来设置输出速度为高速 (High)/快速 (Fast)/中速 (Medium)/低速 (Low)。本实验我们设置为高速 High。

⑤ 选项 User Label 是用来设置初始化的 I/O 口 Pin 值为我们自定义的宏，一般情况我们可以不用设置，有兴趣的同学可以自由设置后查看生成后的代码就很容易明白其含义。

配置完 PB0 后，PB1 配置方法和参数都一模一样，这里我们就不重复配置。然后我们生成工程源码。接下来打开工程的主文件 main.c 文件可以看到，该文件内部由 STM32CubeMX 生成了函数 MX_GPIO_Init，内容如下：


```
void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LED1_GPIO_Port, LED1_Pin, GPIO_PIN_RESET);

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(LED2_GPIO_Port, LED2_Pin, GPIO_PIN_SET);

    /*Configure GPIO pins : PBPIn PBPIn */
    GPIO_InitStruct.Pin = LED1_Pin|LED2_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
}
```

上面是我们对 GPIO 口进行的一个初始化，回到我们的工程, 新建一个文件，然后保存在 Src 文件夹下面，保存为 led.c，然后工程中添加 led.c 文件，如图 1.3.4 所示：

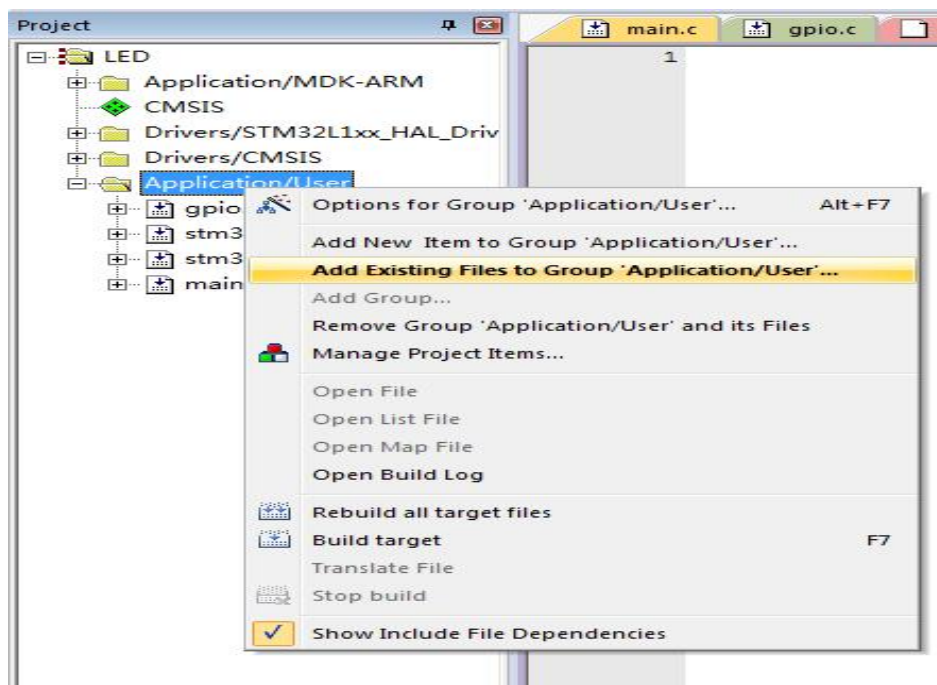


图 1.3.4 添加 led.c 文件

然后在 led.c 文件中输入如下代码，输入后保存即可：

```
#include "led.h"
```

由于 LED 的 GPIO 的初始化已经在 gpio.c 文件里，这里我们的 led.c 里面不再添加代码。保存 led.c 代码，然后我们按同样的方法，新建一个 led.h 文件，保存在 Inc 文件夹下面。

在 led.h 中输入如下代码：

```
#ifndef      __LED_H__
#define      __LED_H__

#include "main.h"

#define OP_LED1()
HAL_GPIO_WritePin(LED1_GPIO_Port,LED1_Pin,GPIO_PIN_SET)
#define CL_LED1()
HAL_GPIO_WritePin(LED1_GPIO_Port,LED1_Pin,GPIO_PIN_RESET)

#define OP_LED2()
HAL_GPIO_WritePin(LED2_GPIO_Port,LED2_Pin,GPIO_PIN_SET)
#define CL_LED2()
HAL_GPIO_WritePin(LED2_GPIO_Port,LED2_Pin,GPIO_PIN_RESET)
```

这里需要注意的是我们声明头文件#ifndef 的意思是 if not define，就是如果没有定义 led.h 的头文件，下面开始定义，如果定义了，编辑的时候不会出现重复定义的错误。

我们也可以不用宏定义，直接去写管脚的高低电平。修改 main.c 文件内容如下：

```
#include "main.h"
#include "stm32l1xx_hal.h"
#include "gpio.h"
void
SystemClock_Config(void);
int main(void)
{

    HAL_Init();
    SystemClock_Config()
    ; ; MX_GPIO_Init();
```


```
while(1)
{
    OP_LED1();
    CL_LED2();
    HAL_Delay(500);
    CL_LED1();
    OP_LED2();
    HAL_Delay(500);
}

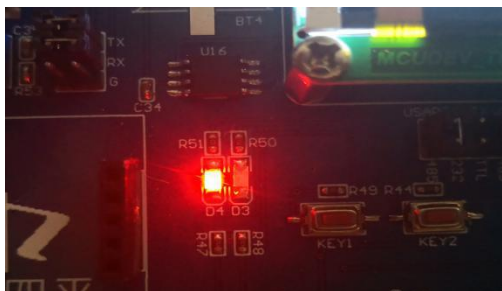
}
```

代码包含了#include "led.h"这句，使得 LED1、LED2 等能在 main() 函数里被调用。main() 函数非常简单，先调用 HAL_Init 函数初始化 HAL 库，然后调用 SystemClock_Config() 进行

时钟系统配置，然后调用 `delay_init()` 函数进行延时 初始化。接着就是调用 `MX_GPIO_Init()` 来初始化 PB0 和 PB1 为推挽输出模式，最后在 `while` 死循环里面实现 LED1 和 LED2 交替闪烁，间隔为 500ms。接下来，大家就可以使用 Jlink-OB 编译下载验证了。

1.4 下载验证

我们使用 Keil 自带的下载功能烧写程序，将 Jlink-OB 连接试验平台点击下载按钮 , 烧写完成后我们可以看到 LED1 和 LED2 交替闪烁。



至此，我们第一章关于使用 HAL 操作 GPIO 口的知识就给大家讲解到这里，本章作为 STM32L1 的入门第一个例子，介绍了 STM32L1 的 IO 口的使用及注意事项，希望大家好好理解一下。

第二章 按键输入实验

上一章，我们介绍了 STM32L1 的 I/O 口作为输出的使用，这一章，我们将向大家介绍如何使用 STM32L1 的 I/O 口作为输入用。在本章中，我们将利用板载的 2 个按键，来控制板载的两个 LED 的亮灭。通过本章的学习，你将了解到 STM32L1 的 I/O 口作为输入口的使用方法。本章分为如下几个小节：

- 2.1 STM32L1 I/O 口简介
- 2.2 硬件设计
- 2.3 STM32CubeMX 配置 I/O 口输出和软件设计
- 2.4 下载验证

2.1 STM32L1 I/O 口简介

STM32L1 的 I/O 口在上一章已经有了比较详细的介绍，这里我们不再多说。STM32L1 的 I/O 口做输入使用的时候，是通过调用函数 HAL_GPIO_ReadPin () 来读取 I/O 口的状态的。了解了这点，就可以开始我们的代码编写了。

这一章，我们将通过 STM32 开发板上载有的 2 个按钮（KEY1、KEY2），来控制板上的 2 个 LED（LED1 和 LED2），其中 KEY1 控制 LED1, KEY2 控制 LED2，按下按键点亮 LED, 松开按键熄灭 LED。

2.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 LED1、LED2。
- 2) 2 个按键：KEY1、KEY2。

LED1、LED2 和 STM32L151 的连接在上一章已经介绍过了，在 STM32 开发板上的按键如图 2.2.1 所示：

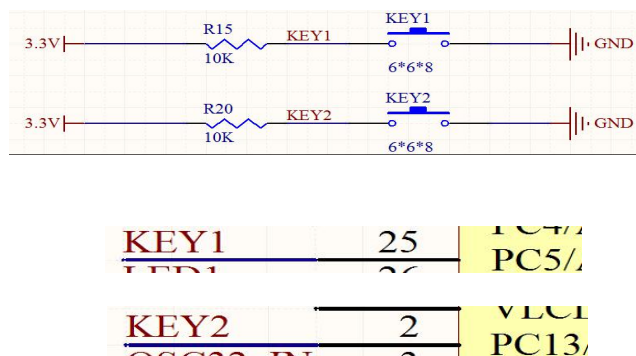


图 2.2.1 按键与 STM32L151 连接原理图

这里需要注意的是：KEY1 是低电平有效，KEY2 是高电平有效。

原理：按键未按下时，10k 电阻连接到 3.3v 电源所以为高电平，当按键按下的时候，3.3v 电源与地形成回路。10k 电阻被拉成低电平。

2.3 STM32CubeMX 配置 I/O 口输出和软件设计

上一章我们讲解了使用 STM32CubeMX 工具配置 GPIO 的一般方法。本章我们主要教大家配置 I/O 口为输入模式，操作方法和配置 I/O 口为输出模式基本一致。这里我们就直接列出 I/O 口配置截图，具体方法请参考上一章跑马灯实验。

根据 2.2 小节讲解，开发板上有 2 个按键，分别连接 2 个 I/O 口 PC5，PC13。操作方法如下图所示 2.3.1 所示：

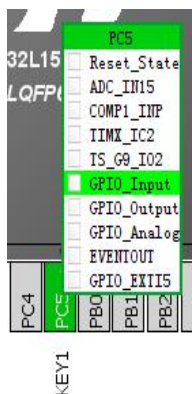


图 2.3.1 配置 PC5 为输入模式

同样的方法，我们配置 PC13 为输入模式。然后我们进入 Configuration->GPIO 配置界面，配置 I/O 口详细参数。在配置界面点击 PC5 可以发现，当我们在前面设置 I/O 口为输入 GPIO_Input 之后，其配置参数只剩下模式 GPIO Mode 和上下拉 GPIO Pull-up/ Pull-down，并且模式值中只有输入模式 Input Mode 可选。这里我们配置 PC5 为无上下拉即可。配置方法如下图所示 2.3.2 所示

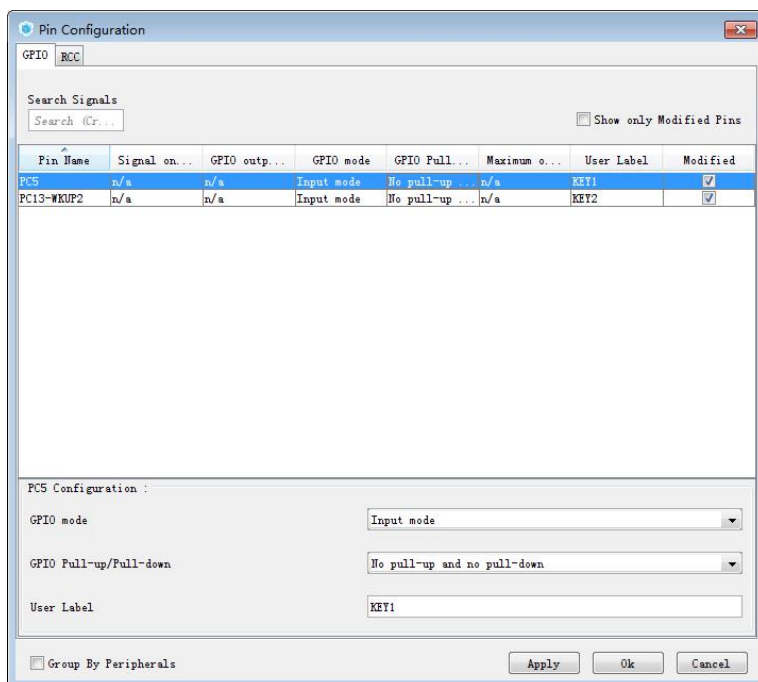


图 2.3.2 配置 I/O 口详细参数

配置完成 IO 口参数之后，接下来我们生成工程。打开生成的工程会发现，main.c 文件中添加了函数 MX_GPIO_Init 函数，内容如下：

```
void MX_GPIO_Init(void)
{

    GPIO_InitTypeDef GPIO_InitStruct;

    HAL_RCC_GPIOC_CLK_ENABLE();
    HAL_RCC_GPIOH_CLK_ENABLE();
    HAL_RCC_GPIOB_CLK_ENABLE();

    HAL_GPIO_WritePin(GPIOB, LED1_Pin|LED2_Pin, GPIO_PIN_RESET);

    GPIO_InitStruct.Pin = KEY2_Pin|KEY1_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    GPIO_InitStruct.Pin = LED1_Pin|LED2_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

}
```

工程我们需要加入 key.c 文件以及头文件 key.h。下面我们首先建立 key.c 件，关键代码如下：

```
#include
"key.h"
#include
"led.h"
void
KEY(void)
{
    if(key1())
    {
        OP_LED1()
        ;
    }
    else
    {
        CL_LED1()
        ;
    }

    if(key2())
    {
        OP_LED2()
        ;
    }

}
```

接下来我们看看头文件 key.h 里面的代码：

```
#ifndef __KEY_H__
#define __KEY_H__

#include "main.h"
#include "gpio.h"

#define key1()    HAL_GPIO_ReadPin(KEY1_GPIO_Port,KEY1_Pin)?0:1
#define key2()    HAL_GPIO_ReadPin(KEY2_GPIO_Port,KEY2_Pin)?1:0

void KEY(void);

#endif
```

这里我们运用了一个运算符，这里使用的是调用HAL库函数HAL_GPIO_ReadPin来实现读取某个IO口的1个位，以key1键为例，如果引脚的状态为真则写为0，否则为1。最后，我们看看mainc里面编写的主函数代码如下：

主函数代码比较简单，先进行一系列的初始化操作，然后在死循环中调用按键扫描函数KEY()扫描按键控制LED的状态。接下来，大家就可以使用Jlink-OB编译下载验证了。

2.4 下载验证

我们使用Keil自带的下载功能烧写程序，将Jlink-OB连接试验平台点击下载按钮，烧写完成后我们可以看到按下KEY1后LED1点亮，松开后LED1熄灭。

```
#include "main.h"
#include "stm32l1xx_hal.h"
#include "gpio.h"
#include "key.h"
void SystemClock_Config(void);
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    while (1)
    {
        KEY();
    }
}
```

第三章 串口通信实验

前面两章介绍了 STM32L151 的 I/O 口操作。这一章我们将学习 STM32L151 的串口，教大家如何使用 STM32L151 的串口来发送和接收数据。本章将实现如下功能：STM32L151 通过串口和上位机的对话，STM32L151 在收到上位机发过来的字符串后，原原本本的返回给上位机。本章分为如下几个小节：

- 3.1 STM32L15 串口简介
- 3.2 硬件设计
- 3.3 STM32CubeMX 配置串口和软件设计
- 3.4 下载验证

3.1 STM32L151 串口简介

串口作为 MCU 的重要外部接口，同时也是软件开发重要的调试手段，其重要性不言而喻。现在基本上所有的 MCU 都会带有串口，STM32 自然也不例外。

STM32L151 的串口资源相当丰富的，功能也相当强劲。STM32L151 开发板所使用的 STM32L151R8T6 最多可提供 3 路串口。

接下来我们先从寄存器层面，告诉你如何设置串口，以达到我们最基本的通信功能。本章我们将实现利用串口 3 不停的打印信息到电脑上，同时接收从串口发过来的数据，把发送过来的数据直接送回给电脑。STM32L151 开发板板载了 1 个 RS232 串口，我们本章介绍的是过 RS232 串口和电脑通信。

串口最基本的设置，就是波特率的设置。STM32L151 的串口使用起来还是蛮简单的，只要你开启了串口时钟，并设置相应 I/O 口的模式，然后配置一下波特率，数据位长度，奇偶位等信息，就可以使用了。下面我们就简单介绍下这几个与串口基本配置直接相关的寄存器。

1. 串口时钟使能。串口作为 STM32L151 的一个外设，其时钟由外设时钟使能寄存器控制，这里我们使用的串口 3 是在 APB2ENR 寄存器的第 4 位。APB2ENR 寄存器在之前已经介绍过了，这里不再介绍。只是说明一点，就是除了串口 1 和串口 6 的时钟使能在 APB2ENR 寄存器，其他串口的时钟使能位都在 APB1ENR 寄存器。

2. 串口波特率设置。每个串口都有一个自己独立的波特率寄存器 USART_BRR，通过设置该寄存器就可以达到配置不同波特率的目的。

3. 串口控制。STM32L151 的每个串口都有 3 个控制寄存器 USART_CR1~3，串口的很多配置都是通过这 3 个寄存器来设置的。这里我们只要用到 USART_CR3 就可以实现我们的功能了，该寄存器的各位描述如图 8.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OVER8	Reserved	UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
r/w	Res.	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

图 8.1.1 USART_CR1 寄存器各位描述

该寄存器的高 16 位没有用到，低 16 位用于串口的功能设置。OVER8 为过采样模式设置位，我们一般设置位 0，即 16 倍过采样已获得更好的容错性；UE 为串口使能位，通过该位置 1，以使能串口；M 为字长选择位，当该位为 0 的时候设置串口为 8 个字长外加 n 个停止位，停止位的个数（n）是根据 USART_CR2 的[13:12]位设置来决定的，默认为0；PCE 为校验使能位，设置为 0，则禁止校验，否则使能校验；PS 为校验位选择位，设置为 0 则为偶校验，否则为奇校验；TXIE 为发送缓冲区空中断使能位，设置该位为 1，当 USART_SR 中的 TXE 位为 1 时，将产生串口中断；TCIE 为发送完成中断使能位，设置该位为1，当USART_SR 中的TC 位为1 时，将产生串口中断；RXNEIE 为接收缓冲区非空中断使能，设置该位为1，当 USART_SR 中的ORE 或者RXNE 位为1 时，将产生串口中断；TE 为发送使能位，设置为1 将开启串口的发送功能；RE 为接收使能位，用法同TE。

4. 数据发送与接收。STM321151 的发送与接收是通过数据寄存器USART_DR 来实现的，这是一个双寄存器，包含了TDR 和RDR。当向DR 寄存器写数据的时候，实际是写入TDR，串口就会自动发送数据；当收到数据，读DR 寄存器的时候，实际读取的是RDR。TDR 和RDR对外是不可见的，所以我们操作的就只有DR 寄存器，该寄存器的各位描述如图8. 1. 2 所示：



图8. 1. 2 USART_DR 寄存器各位描述

可以看出，虽然是一个32 位寄存器，但是只用了低9 位（DR[8:0]），其他都是保留。

DR[8:0]为串口数据，包含了发送或接收的数据。由于它是由两个寄存器(TDR 和 RDR)组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。

当使能校验位(USART_CR1 中 PCE 位被置位)进行发送时，写到 MSB 的值(根据数据的长度不同，MSB 是第 7 位或者第8 位)会被后来的校验位取代。

当使能校验位进行接收时，读到的MSB 位是接收到的校验位。

5, 串口状态。串口的状态可以通过状态寄存器 USART_SR 读取。USART_SR 的各位描述如图 8. 1. 3 所示：

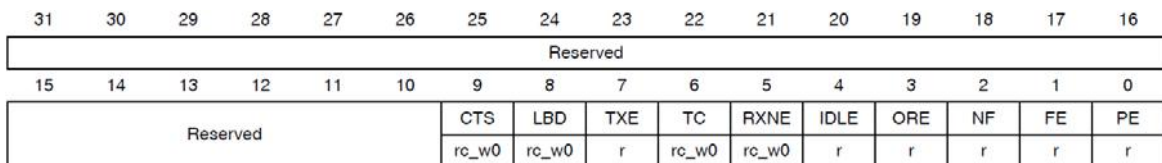


图8. 1. 3 USART_SR 寄存器各位描述

这里我们关注一下两个位，第 5、6 位 RXNE 和 TC。RXNE（读数据寄存器非空），当该位被置 1 的时候，就是提示已经有数据被接收到了，并且可以读出来了。这时候我们要做的就是尽快去读取 USART_DR，通过读 USART_DR 可以将该位清零，也可以向该位写 0，直接清除。TC（发送完成），当该位被置位的时候，表示 USART_DR 内的数据已经被发送完成了。如果设置了这个位的中断，则会产生中断。该位也有两种清零方式：1）读 USART_SR，写 USART_DR。

2) 直接向该位写 0。通过以上一些寄存器的操作外加一下 IO 口的配置，我们就可以达到串口最基本的配置了，

对于怎么直接使用寄存器配置串口收发，请参考我们寄存器版本教程和源码。接下来我们将着重讲解使用 HAL 库实现串口配置和使用的方法。在 HAL 库中，串口相关的函数和定义主要在文件 stm32f4xx_hal_uart.c 和 stm32f4xx_hal_uart.h 中。接下来我们看看 HAL 库提供的串口相关操作函数。

1) 串口参数初始化（波特率/停止位等）并使能串口。

串口作为STM32 的一个外设，HAL 库为其配置了串口初始化函数。接下来我们看看串口初始化函数HAL_UART_Init 相关知识，定义如下：

```
HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart);
```

该函数只有一个入口参数huart，为UART_HandleTypeDef 结构体指针类型，我们俗称其为串口句柄，它的使用会贯穿整个串口程序。一般情况下，我们会定义一个 UART_HandleTypeDef 结构体类型全局变量，然后初始化各个成员变量。接下来我们看看结构体 UART_HandleTypeDef 的定义：

```
typedef struct
{
    USART_TypeDef          *Instance;
    UART_InitTypeDef       Init;
    uint8_t                *pTxBuffPtr;
    uint16_t               TxXferSize;
    uint16_t               TxXferCount;
    uint8_t                *pRxBuffPtr;
    uint16_t               RxXferSize;
    uint16_t               RxXferCount;
    DMA_HandleTypeDef      *hdmatx;
    DMA_HandleTypeDef      *hdmarx;
    HAL_LockTypeDef        Lock;
    __IO HAL_UART_StateTypeDef State;
    __IO uint32_t           ErrorCode;
}UART_HandleTypeDef;
```

该结构体成员变量非常多，一般情况下载调用函数 HAL_UART_Init 对串口进行初始化的时候，我们只需要先设置 Instance 和 Init 两个成员变量的值。接下来我们依次解释一下各个成员变量的含义。

Instance 是 USART_TypeDef 结构体指针类型变量，它是执行寄存器基地址，实际上这个基地址 HAL 库已经定义好了，如果是串口 1，取值为 USART1 即可。

Init 是 UART_InitTypeDef 结构体类型变量，它是用来设置串口的各个参数，包括波特率，停止位等，它的使用方法非常简单。UART_InitTypeDef 结构体定义如下：

```
typedef struct
{
    uint32_t BaudRate;        //波特率
    uint32_t WordLength;      //字长
    uint32_t StopBits;        //停止位
    uint32_t Parity;          //奇偶校验
    uint32_t Mode;            //收/发模式设
    置uint32_t HwFlowCtl;     //硬件流设置
    uint32_t OverSampling;    //过采样设置
}UART_InitTypeDef
```

该结构体第一个参数 BaudRate 为串口波特率，波特率可以说是串口最重要的参数了，用来确定串口通信的速率。第二个参数 WordLength 为字长，可以设置为 8 字长或者 9 位字长，这里我们设置为 8 位字长数据格式 UART_WORDLENGTH_8B。第三个参数 StopBits 为停止位设置，可以设置为 1 个停止位或者 2 个停止位，这里我们设置为 1 位停止位 UART_STOPBITS_1。第四个参数 Parity 设定是否需要奇偶校验，我们设定为无奇偶校验位。第五个参数 Mode 为串口模式，可以设置为只收模式，只发模式，或者收发模式。这里我们设置为全双工收发模式。第六个参数 HwFlowCtl 为是否支持硬件流控制，我们设置为无硬件流控制。第七个参数 OverSampling 用来设置过采样为 16 倍还是 8 倍。

pTxBuffPtr, TxXferSize 和 TxXferCount 三个变量分别用来设置串口发送的数据缓存指针，发送的数据量和还剩余的要发送的数据量。而接下来的三个变量 pRxBuffPtr, RxXferSize 和 RxXferCount 则是用来设置接收的数据缓存指针，接收的最大数据量以及还剩余的要接收的数据量。这六个变量是 HAL 库处理中间变量，详细使用方法在我们讲解中断服务函数的時候给大家讲解。

hdmatx 和 hdmarx 是串口 DMA 相关的变量，指向 DMA 句柄，这里我们先不讲解。其他的三个变量就是一些 HAL 库处理过程状态标志位和串口通信的错误码。

函数 HAL_UART_Init 使用的一般格式为：

```
UART_HandleTypeDef huart3; //UART 句柄
UART3_Handler.Instance=USART3;
//USART3
UART3_Handler.Init.BaudRate=9600; //波特率
UART3_Handler.Init.WordLength=UART_WORDLENGTH_8B; //字长为8 位格式
UART3_Handler.Init.StopBits=UART_STOPBITS_1; //一个停止
位UART3_Handler.Init.Parity=UART_PARITY_NONE; //无奇偶校
验位UART3_Handler.Init.HwFlowCtl=UART_HWCONTROL_NONE; //无硬件流控
UART3_Handler.Init.Mode=UART_MODE_TX_RX; //收发模式
HAL_UART_Init(&huart3); //HAL_UART_Init() 会使能UART3
```


这里我们需要说明的是，函数 HAL_UART_Init 内部会调用串口使能函数使能相应串口，所以调用了该函数之后我们就不需要重复使能串口了。当然，HAL 库也提供了具体的串口使能和关闭方法，具体使用方法如下：

<code>__HAL_UART_ENABLE(huart);</code>	//使能句柄handler 指定的串口
<code>__HAL_UART_DISABLE(huart);</code>	//关闭句柄handler 指定的串口

这里还需要提醒大家，串口作为一个重要外设，在调用的初始化函数 HAL_UART_Init 内部，会先调用 MSP 初始化回调函数进行 MCU 相关的初始化，函数为：

```
void HAL_UART_MspInit(UART_HandleTypeDef *huart);
```

我们在程序中，只需要重写该函数即可。一般情况下，该函数内部用来编写 IO 口初始化，时钟使能以及NVIC 配置。

2) 使能串口和GPIO 口时钟

我们要使用串口，所以必须使能串口时钟和使用到的 GPIO 口时钟。例如我们要使用串口 3，所以必须使能串口 3 时钟和GPIOC 时钟（串口3 使用的是PC11 和PC10）。具体方法如下：

```
__HAL_RCC_USART3_CLK_ENABLE(); //使能USART3 时钟
__HAL_RCC_GPIOC_CLK_ENABLE();
```

3) GPIO 口初始化设置（速度，上下拉等）以及复用映射配置

我们在跑马灯实验中讲解过，在 HAL 库中 IO 口初始化参数设置和复用映射配置是在函数 HAL_GPIO_Init 中一次性完成的。这里大家只需要注意，我们要复用 PC11 和 PC10 为串口发送接收相关引脚，我们需要配置 IO 口为复用，同时复用映射到串口 3。配置源码如下：

```
GPIO_InitTypeDef GPIO_InitStructure;

GPIO_InitStructure.Pin=GPIO_PIN_11|GPIO_PIN_10; //PC11/PC10
GPIO_InitStructure.Mode=GPIO_MODE_AF_PP; //复用推挽输出
GPIO_InitStructure.Pull=GPIO_PULLUP; //上拉
GPIO_InitStructure.Speed=GPIO_SPEED_FAST; //高速
GPIO_InitStructure.Alternate=GPIO_AF7_USART3; //复用为USART3
HAL_GPIO_Init(GPIOA,&GPIO_InitStructure); //初始化
PC11/PC10
```

3) 开启串口相关中断，配置串口中断优先级

HAL 库中定义了一个使能串口中断的标识符 __HAL_UART_ENABLE_IT，大家可以把它当作一个函数来使用，具体定义请参考 HAL 库文件 stm32l1xx_hal_uart.h 中该标识符定义。例如我们要使能接收完成中断，方法如下：

```
__HAL_UART_ENABLE_IT(huart, UART_IT_RXNE); //开启接收完成中断
```

第一个参数为我们步骤 1 讲解的串口句柄，类型为 UART_HandleTypeDef 结构体类型。
第二个参数为我们开启的中断类型值，可选值在头文件 stm32l1xx_hal_uart.h 中有宏定义。有开启中断就有关闭中断，操作方法为：

```
__HAL_UART_DISABLE_IT(huart,UART_IT_RXNE); //关闭接收完成中断
```

对于中断优先级配置，方法就非常简单参考方法为：

```
HAL_NVIC_EnableIRQ(USART1_IRQn);           //使能USART1 中断通道  
HAL_NVIC_SetPriority(USART1_IRQn,3,3);       //抢占优先级3，子优先级3
```

4) 编写中断服务函数

串口3 中断服务函为：

```
void USART3_IRQHandler(void) ;
```

当发生中断的时候，程序就会执行中断服务函数。然后我们在中断服务函数中编写们相应的逻辑代码即可。HAL 库实际上对中断处理过程进行了完整的封装，具体内容我们通过结合实验源码给大家详细讲解。

5) 串口数据接收和发送

STM32L1 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到数据的时候，也是存在该寄存器内。HAL 库操作USART_DR 寄存器发送数据的函数是：

```
HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart,  
                                     uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

通过该函数向串口寄存器USART_DR 写入一个数据。HAL 库操作 USART_DR 寄存器读取串口接收到的数据的函数是：

```
HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart,  
                                   uint8_t *pData, uint16_t Size, uint32_t Timeout);
```

通过该函数可以读取串口接受到的数据。

3.2 硬件设计

本实验需要用到的硬件资源有：

1) 串口3

串口3 之前还没有介绍过，本实验用到的串口3 与 RS232 并没有在 PCB 上连接在一起，需要通过跳线帽来连接一下。这里我们把 P22 的 USART3_TX 和 USART3_RX 用跳线帽与 232 连接起来。如图 3.2.1 所示：

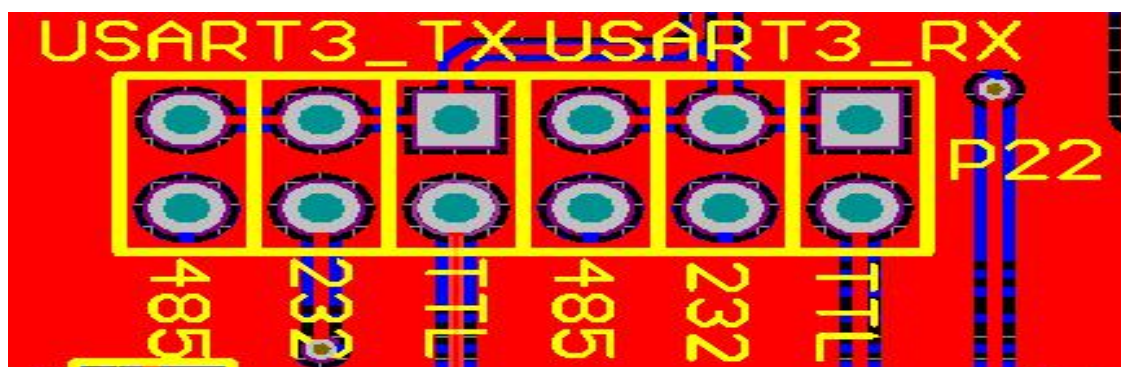


图 3.2.1 硬件连接图示意图

连接上这里之后，我们在硬件上就设置完成了，可以开始软件设计了。

3.3 STM32CubeMX 配置串口和软件设计

前面章节我们详细讲解了使用 STM32CubeMX 配置 I/O 口输入输出，本小节我们将讲解使用 STM32CubeMX 配置串口方法。这里我们要配置串口 3，所以首先我们要使能串口 3 然后设置相应通信模式。打开 Pinout 选项卡界面，左侧依次进入 Configuration->Peripherals->USART1 配置栏，如下图 3.3.1 所示：

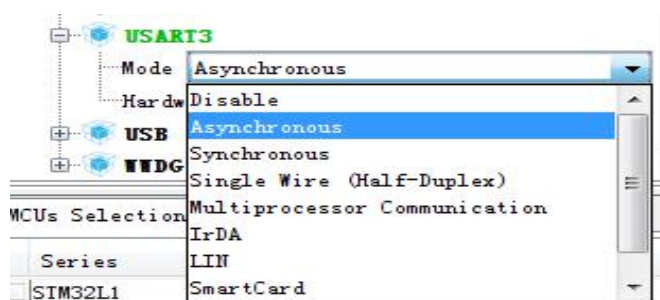


图3.3.1 USART3 配置

USART1 配置栏有 2 个选项。第一个选项 Mode 用来设置串口 1 的模式或者关闭串口 1。第二个选项 Hardware Flow Control (RS232) 用来开启/关闭串口 1 的硬件流控制，该选项只有在 Mode 选项值为 Asynchronous (异步通信) 模式的前提下才有效。下面 Synchronous (同步通信) 这里我们要开启串口 1 的异步模式，并且不使用硬件流控制，所以这里我们直接选择 Mode 值 Asynchronous 即可。

配置好串口 1 为异步通信模式后，那么在硬件上会使用默认引脚作为串口 3 的发送接收脚。如果默认引脚和我们实际使用的引脚不一致可以点击使用的引脚修改。修改的方式可以参考硬件原理图上面的引脚设置。在 STM32CubeMX 中，当我们选择好外设的工作模式之后，软件会自动配置 GPIO 口的相关模式和参数。在 Pinout 界面我们看看芯片引脚图会发现默认引脚与使用引脚不一致，所以我们点击使用 PC10 和 PC11 引脚修改，如下图 3.3.2 所示：

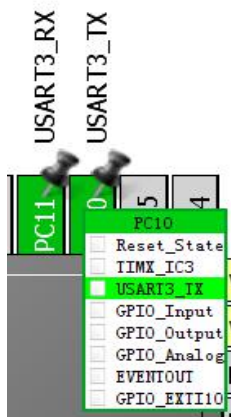


图3.3.2 PC10/PC11 配置

同时，进入 GPIO 配置详细界面会发现，IO 口的模式等参数都做了相应的修改。依次进入 Configuration->GPIO 界面会发现，Pin Configuration 界面多了一个 USART3 选项卡，该选项卡界面便是用来配置和查看串口引脚 PC10 和 PC11 配置参数的。如下图 3.3.3 所示：

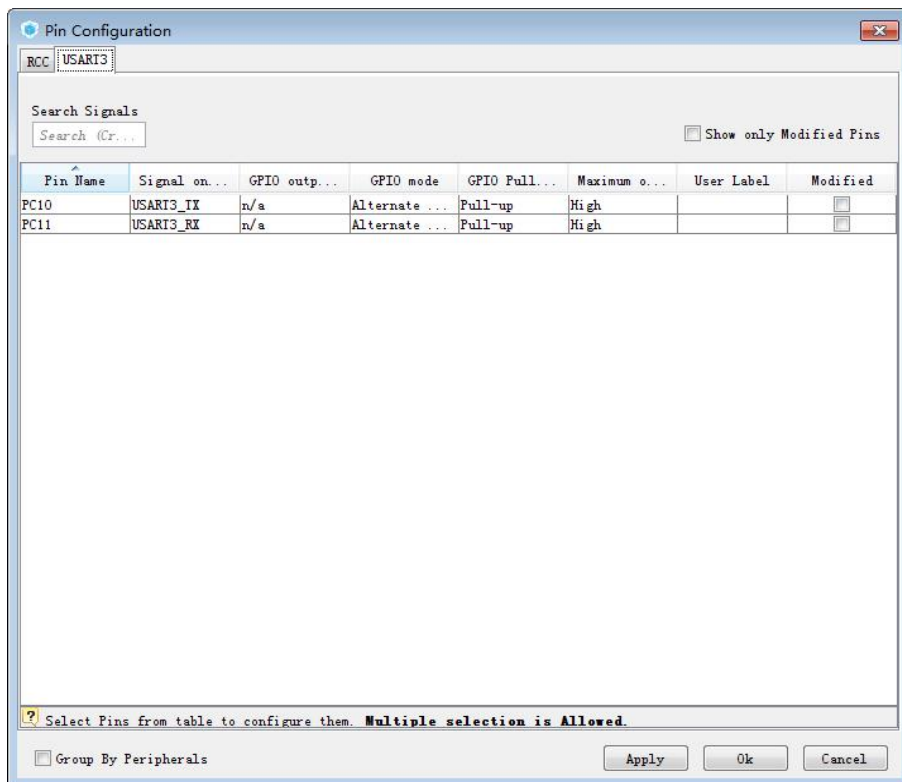


图 3.3.3 USART3 引脚详细配置界面

对于外设的功能引脚，在我们使能相应的外设（比如 USART3）之后，STM32CubeMX 会自动设置 GPIO 相关配置，一般情况下用户不再需要去修改。所以这里，对于 PC10 和 PC11 的配置我们就保留软件配置即可。接下来我们需要配置 USART3 外设相关的参数，包括波特率，停止位等。我们直接进入 Configuration 选项卡，如果我们之前使能了 USART3，那么在 Connectivity 栏会出现 USART3 配置按钮。如下图 3.3.4 所示：

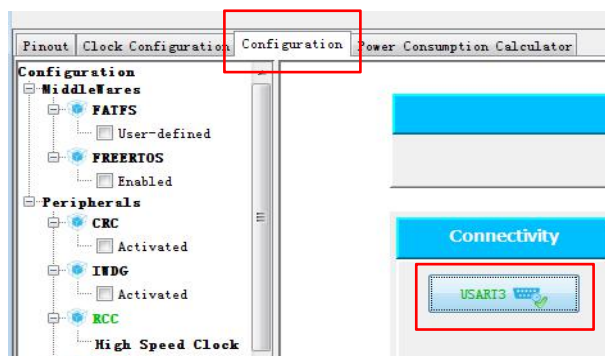
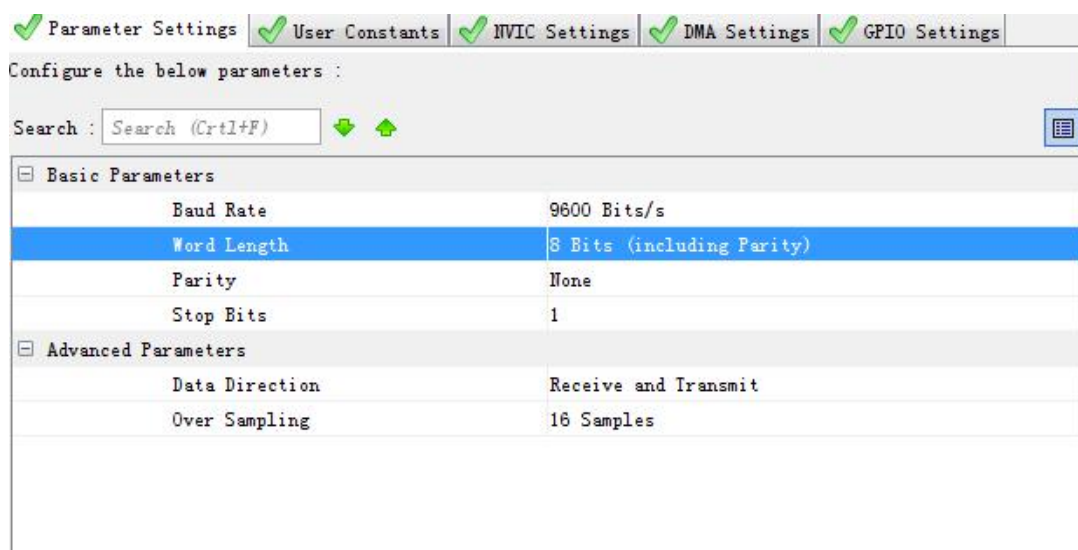


图 3.3.4 Configuration 选项卡

接下来我们点击 USART3 配置按钮，进入 USART3 详细参数配置界面。USART3Configuration 界面会出现 5 个配置选项卡。

Parameter Settings 选项卡用来配置 USART3 的初始化参数，包括波特率停止位等等。这里我们将 USART3 配置为：波特率 9600，8 位字长模式，无奇偶校验位，1 个停止位，发送/接收均开启。



User Constants 是用来配置用户常量。

NVIC 选项卡用来使能 USART3 中断。这里我们勾上 Enabled 选项。

DMA Setting 是在使用 USART3DMA 的情况才需要配置，这里我们不配置。

GPIO Setting 便是查看和配置 USART3 相关的 IO 口。

配置完 USART1 相关 IO 口和 USART3 参数之后，如果我们使用到串口中断，那么我们还需要设置中断优先级分组。接下来便是配置 NVIC 相关参数。同样的方法，进入 Conguration 选项卡，点击 NVIC 按钮，如下图 3.3.5 所示：

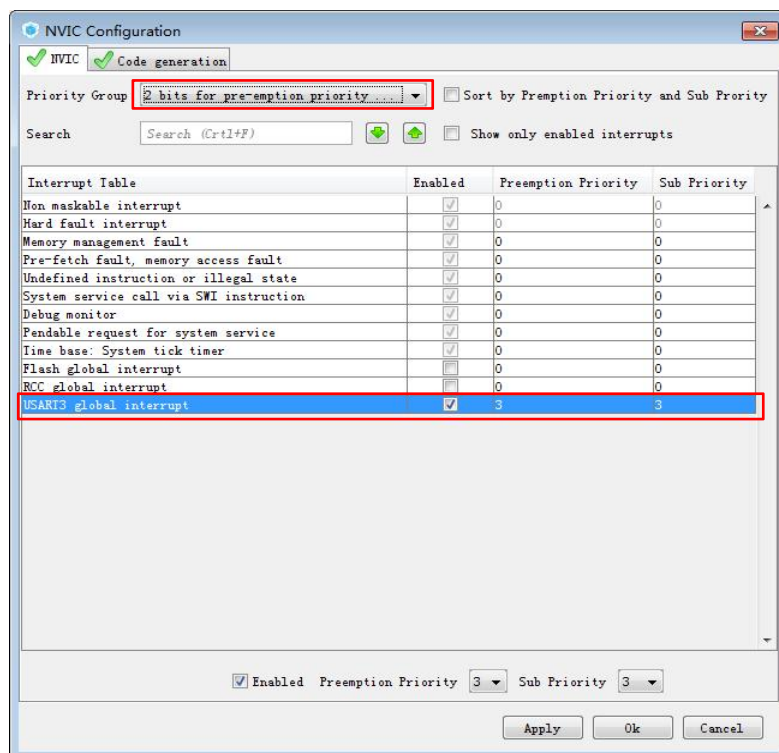


图 3.3.5 NVIC Configuration 配置界面

在弹出的 NVIC Configuration 界面,我们首先设置中断优先级分组级别,我们系统初始化设置为分组 2,那么就是 2 为抢占优先级和 2 位响应优先级。所以这里的参数我们选择“2 bits for pre-emption priority”,也就是 2 位抢占优先级。配置完中断优先级分组之后,接下来我们要配置的是 USART1 的抢占优先级和响应优先级值,这里我们设置抢占和响应优先级均为 3 即可。进行完上面的操作之后,接下来我们便是生成工程代码。打开生成的工程可以看到,在 uart.c 文件中生成了如下串口初始化关键代码:

```
void MX_USART3_UART_Init(void)
{
    huart3.Instance = USART3;
    huart3.Init.BaudRate = 9600;
    huart3.Init.WordLength = UART_WORDLENGTH_8B;
    huart3.Init.StopBits = UART_STOPBITS_1;
    huart3.Init.Parity = UART_PARITY_NONE;
    huart3.Init.Mode = UART_MODE_TX_RX;
    huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart3.Init.OverSampling = UART_OVERSAMPLING_16;
```


串口 MSP 函数 HAL_UART_MspInit 函数我们自定义了其内容，代码如下：

```
void HAL_UART_MspInit(UART_HandleTypeDef* uartHandle)
{

    GPIO_InitTypeDef
    GPIO_InitStruct;
    if(uartHandle->Instance==USART3)
    {
        __HAL_RCC_USART3_CLK_ENABLE();

        GPIO_InitStruct.Pin = GPIO_PIN_10|GPIO_PIN_11;
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
        GPIO_InitStruct.Pull = GPIO_PULLUP;
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
        GPIO_InitStruct.Alternate = GPIO_AF7_USART3;
        HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);
    }
}
```

该函数代码实现的是我们 3.1 小节讲解的步骤 2 到 4 的内容。通过上面两个函数，我们就配置了串口相关设置。接下来就是编写中断服务函数 USART1_IRQHandler。

中断函数的位置在 stm32l1xx_it.c 里面而 HAL 库中，对中断服务函数的编写有非常严格的讲究。首先 HAL 库定义了一个串口中断处理通用函数 HAL_UART_IRQHandler，该函数声明如下：

```
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart);
```

该函数只有一个入口参数就是 UART_HandleTypeDef 结构体指针类型的串口句柄 huart，使用我们在调用 HAL_UART_Init 函数时设置的同一个变量即可。该函数一般在中断服务函数中调用，作为串口中断处理的通用入口。一般调用方法为：

```
void USART1_IRQHandler(void)
{
    HAL_UART_IRQHandler(&UART1_Handler); //调用 HAL 库中断处理公用函数
    ...//中断处理完成后的结束工作
}
```

也就是说真正的串口中断处理逻辑我们会最终在函数 HAL_UART_IRQHandler 内部执行。

而该函数是 HAL 库已经定义好，而且用户一般不能随意修改。这个时候大家会问，那么我们的中断控制逻辑编写在哪里呢？为了把这个问题讲解清楚，我们要来看看函数

HAL_UART_IRQHandler 内部具体实现过程因为本章实验我们主要实现的是串口中断接收，也就是每次接收到一个字符后进入中断服务函数来处理。所以我们就以中断接收为例给大家讲解。这里为了篇幅考虑，我们仅仅列出串口中断执行流程中与接收相关的源码。

函数HAL_UART_IRQHandler 关于串口接收相关源码如下：

```
void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
{
    uint32_t tmp1 = 0, tmp2 = 0;
    ...//此处省略部分代码

    tmp1 = _HAL_UART_GET_FLAG(huart, UART_FLAG_RXNE); tmp2 =
    HAL_UART_GET_IT_SOURCE(huart, UART_IT_RXNE);

    if((tmp1 != RESET) && (tmp2 != RESET))
    {
        UART_Receive_IT(huart);

        ...//此处省略部分代码
```

从代码逻辑可以看出，在函数HAL_UART_IRQHandler 内部通过判断中断类型是否为接收完成中断，确定是否调用HAL 另外一个函数UART_Receive_IT()。函数UART_Receive_IT()的作用是把每次中断接收到的字符保存在串口句柄的缓存指针 pRxBuffPtr 中，同时每次接收一个字符，其计数器RxXferCount 减1，直到接收完成RxXferSize 个字符之后RxXferCount 设置为 0，同时调用接收完成回调函数HAL_UART_RxCpltCallback 进行处理。这里我们仅列出UART_Receive_IT()函数调用回调函数HAL_UART_RxCpltCallback 的处理逻辑，代码如下：

```
static HAL_StatusTypeDef UART_Receive_IT(UART_HandleTypeDef *huart)
{
    ...//此处省略部分代码
    if(--huart->RxXferCount == 0)
    {
        HAL_UART_RxCpltCallback(huart);
    }
    ...//此处省略部分代码
}
```

最后我们列出串口接收中断的一般流程，如图3.3.6 所示：

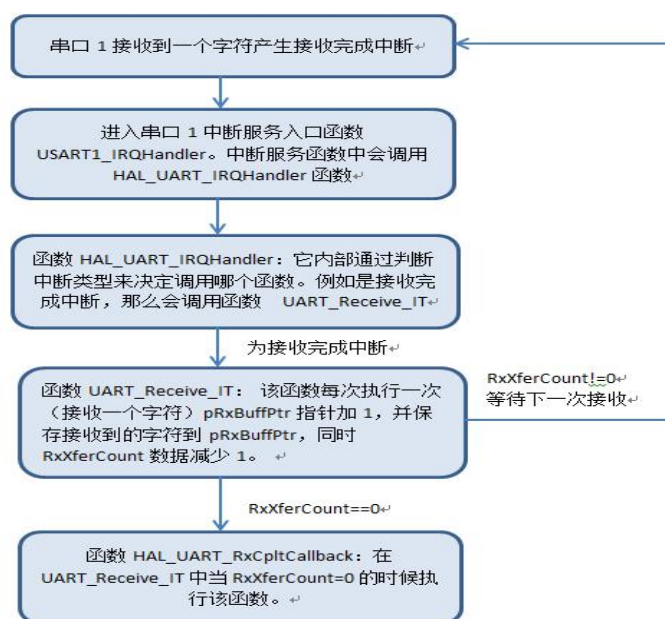


图 3.3.6 串口接收中断执行流程图

这里, 我们再把串口接收中断的一般流程进行概括: 当接收到一个字符之后, 在函数 `UART_Receive_IT` 中会把数据保存在串口句柄的成员变量 `pRxBuffPtr` 缓存中, 同时 `RxXferCount` 计数器减 1。如果我们设置 `RxXferSize=10`, 那么当接收到 10 个字符之后, `RxXferCount` 会由 10 减到 0 (`RxXferCount` 初始值等于 `RxXferSize`), 这个时候再调用接收完成回调函数 `HAL_UART_RxCpltCallback` 进行处理。接下来我们看看我们的配置。

首先, 我们回到用户函数 `MX_USART3_UART_Init` 定义可以看到, 调用 `HAL_UART_Init` 后我们还调用了 `HAL_UART_Receive_IT` 开启接收中断, 并且初始化串口句柄的缓存相关参数。代码如下:

```
HAL_UART_Receive_IT(&huart3, aRxBuffer, USART3_size);
```

而 `aRxBuffer` 是我们定义的一个全局数组变量, `USART3_size` 是我们定义的一个标识符:

```
#define USART3_size 1
unsigned char aRxBuffer[USART3_size]={0};
```

所以, 调用 `HAL_UART_Receive_IT` 函数后, 除了开启接收中断外还确定了每次接收 `USART3_size` 个字符后标示接收结束从而进入回调函数 `HAL_UART_RxCpltCallback` 进行相应处理。最后我们看看 `HAL_UART_RxCpltCallback` 函数定义:

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance==USART3)
    {
        switch(USART3_Count)
        {
            case 0:
                if(aRxBuffer[0]!=0x0d)           //如果收到的不是 0xAA
                {
                    USART3_Receviebuf[USART3_len]=aRxBuffer[0];
                    USART3_len++;
                }
                else
                {
                    USART3_Count=1;
                }
                break;
            case 1:
                if(aRxBuffer[0]==0x0a)
                {
                    USART3_Recevie_flag=1;
                }
                USART3_Count=0;
                break;
            default:
                break;
        }
        HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
    }
}
```

因为我们设置了串口句柄成员变量 `RxXferSize` 为 1，也就是每当串口 1 发生了接收完成中断后（接收到一个字符），就会跳到该函数执行。当串口接受到一个字符后，它会保存在缓存 `aRxBuffer` 中，由于我们设置了缓存大小为 1，而且 `RxXferSize=1`，所以每次接受一个字符，会直接保存到 `RxXferSize[0]` 中，我们直接通过读取 `RxXferSize[0]` 的值就是本次接收到的字符。这里我们设计了一个小小的接收协议：通过这个函数，配合一个数组 `USART3_Receviebuf[]`，一个接收状态 `USART3_Recevie_flag` 实现对串口数据的接收管理。设计思路如下：当接收到从电脑发过来的数据，把接收到的数据保存在 `USART3_Receviebuf` 中，当收到回车（回车的表示由 2 个字节组成：0x0D 和 0x0A）的第一个字节 0x0D 时，这样完成一次接收，然后将接收到的数据发送给电脑。

在函数 `USART1_IRQHandler` 的结尾还有一行代码：

```
HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
```

这行代码是继续调用 HAL_UART_Receive_IT 函数来开启中断和重新设置 RxXferSize 和 RxXferCount 的初始值为1，也就是开启新的接收中断。

最后我们来看看主函数：

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_USART3_UART_Init();
    MX_NVIC_Init();
    while (1)
    {
        if(USART3_Recevie_flag)
        {
            USART3_Recevie_flag=0;
            HAL_UART_Transmit_IT(&huart3,USART3_Receviebuf,USART3_len);
            while( _HAL_UART_GET_FLAG(&huart3,UART_FLAG_TC)!=SET);
            USART3_len=0;
        }
    }
}
```

这段代码逻辑比较简单，首先判断全局变量 USART3_Recevie_flag 是否为 1，如果为 1 的话，那么代表前一次数据接收已经完成，接下来就是把我们自定义接收缓冲的数据发送到串口。接下来我们重点以下两句：

```
HAL_UART_Transmit_IT(&huart3,USART3_Receviebuf,USART3_len);
while( _HAL_UART_GET_FLAG(&huart3,UART_FLAG_TC)!=SET);
```

第一句，其实就是调用 HAL 串口发送函数 HAL_UART_Transmit 来发送一个字符到串口。

第二句呢，就是我们发送一个字节之后之后，要检测这个数据是否已经被发送完成了。

3.4 下载验证

我们把程序下载到 STM32L151 开发板，接着我们打开串口助手，设置串口为（得根据你自己的电脑选择，我的电脑是 COM14，另外请注意：波特率是 9600），可以看到如图 3.4.1 因为我们在程序上面设置了必须输入回车，串口才认可接收到的数据，所以必须在发送数据后再发送一个回车符，我们再次在发送区输入你想要发送的文字，然后单击发送，可以得到如图 3.4.1 所示结果。



图 3.4.1 串口调试助手收到的信息

第四章 外部中断实验

这一章，我们将向大家介绍如何使用 STM32L151 的外部输入中断。在前面几章的学习中，我们掌握了 STM32L151 的 IO 口最基本的操作。本章我们将介绍如何将 STM32L151 的 IO 口 作为外部中断输入，在本章中我们将以中断的方式，实现我们在第二章所实现的功能。本章分为如下几个部分：

- 4.1 STM32L1 外部中断简介
- 4.2 硬件设计
- 4.3 STM32CubeMX 配置外部中断和软件设计
- 4.4 下载验证

4.1 STM32L1 外部中断简介

STM32L1 的 IO 口在前面有详细介绍，而中断优先级分组管理在前面也有详细的阐述。这里我们将介绍 STM32L1 外部 IO 口的中断功能，通过中断的功能，达到第二章实验的效果，即：通过板载的 2 个按键，控制板载的两个 LED 的亮灭。

这里我们首先讲解 STM32L1 IO 口中中断的一些基础概念。STM32L1 的每个 IO 都可以作为外部中断的中断输入口，这点也是 STM32L1 的强大之处。STM32L151 的中断控制器支持 23 个外部中断/事件请求。每个中断设有状态位，每个中断/事件都有独立的触发和屏蔽设置。STM32L151 的 23 个外部中断为：

EXTI 线 0~15：对应外部 IO 口的输入中断。

EXTI 线 16：连接到 PVD 输出。

EXTI 线 17：连接到 RTC 警报事件。EXTI 线 18：连接到 USB 唤醒事件。

EXTI 线 19：连接到 RTC 入侵和时间戳事件。

EXTI 线 20：连接到 RTC 唤醒事件。

EXTI 线 21：连接到 COMP1 唤醒事件。EXTI 线 22：连接到 COMP2 唤醒事件。

从上面可以看出，STM32L1 供 IO 口使用的中断线只有 16 个，但是 STM32L1 的 IO 口却远远不止 16 个，那么 STM32L1 是怎么把 16 个中断线和 IO 口一一对应起来的呢？于是 STM32 就这样设计，GPIO 的引脚 GPIOx.0~GPIOx.15(x=A,B,C,D,E,F,G,H,I) 分别对应中断线 0~15。这样每个中断线对应了最多 9 个 IO 口，以线 0 为例：它对应了 GPIOA.0、GPIOB.0、GPIOC.0、GPIOD.0、GPIOE.0、GPIOF.0、GPIOG.0、GPIOH.0、GPIOI.0。而中断线每次只能连接到 1 个 IO 口上，这样就需要通过配置来决定对应的中断线配置到哪个 GPIO 上了。下面我们看看 GPIO 跟中断线的映射关系图：

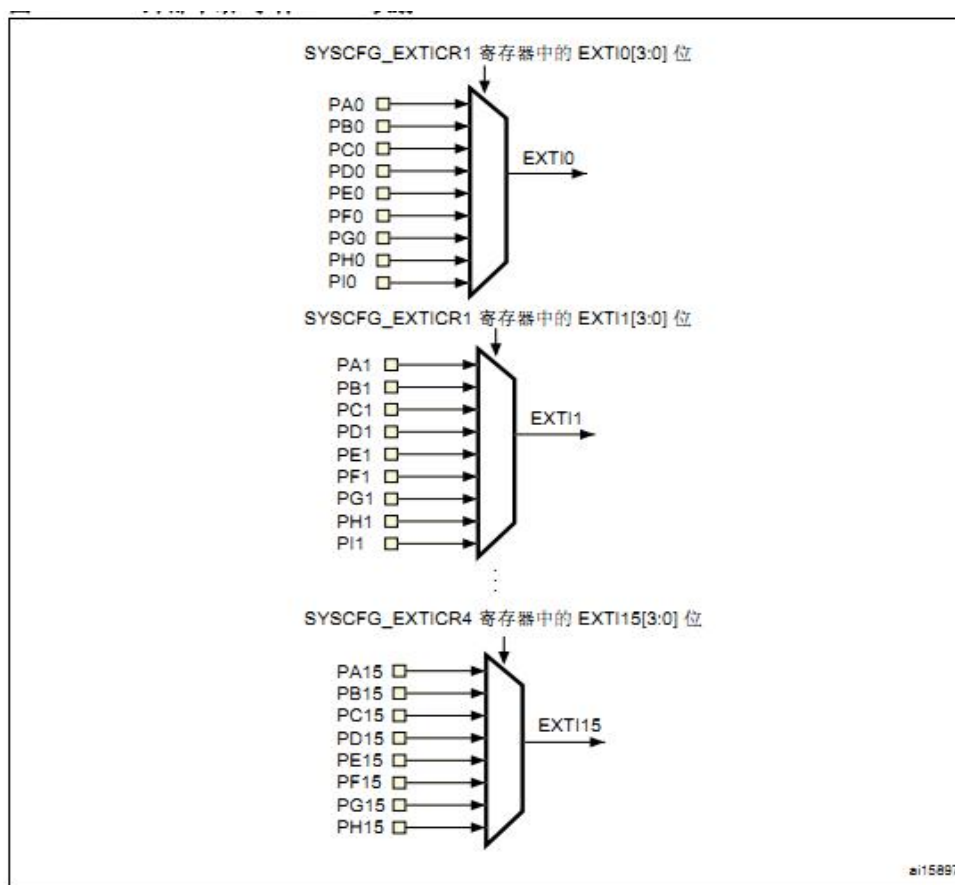


图4. 1. 1 GPIO 和中断线的映射关系图

GPIO 和中断线映射关系是在寄存器 SYSCFG_EXTICR1~SYSCFG_EXTICR4 中配置的。所以要配置外部中断，还需要打开SYSCFG 时钟。

接下来我们来看看使用 HAL 库配置外部中断的一般步骤。

1) 使能 IO 口时钟。

首先，我们要使用IO 口作为中断输入，所以我们要使能相应的IO 口时钟，具体的操作方法跟我们按键实验是一致的，这里就不做过多讲解。

2) 设置IO 口模式，触发条件，开启SYSCFG 时钟，设置IO 口与中断线的映射关系。

该步骤如果我们使用标准库那么需要多个函数分部实现。而当我们使用 HAL 库的时候，则都是在函数 HAL_GPIO_Init 中一次性完成的。例如我们要设置 PA0 链接中断线 0，并且为上 升沿触发，代码为：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.Pin=GPIO_PIN_0; //PA0
GPIO_InitStructure.Mode=GPIO_MODE_IT_RISING; //外部中断，上升沿触发
GPIO_InitStructure.Pull=GPIO_PULLDOWN; //默认下拉
HAL_GPIO_Init(GPIOA,&GPIO_InitStructure);
```

当我们调用 HAL_GPIO_Init 设置 IO 的 Mode 值为 GPIO_MODE_IT_RISING（外部中断上升沿触发），GPIO_MODE_IT_FALLING（外部中断降沿触发）或者 GPIO_MODE_IT_RISING_FALLING（外

部中断双边沿触发)的时候,该函数内部会通过判断 Mode 的值来开启 SYSCFG 时钟,并且设置 IO 口和中断线的映射关系。

因为我们这里初始化的是 PA0,根据图 9.1.1 可知,调用该函数后中断线 0 会自动连接到 PA0。如果某个时间,我们用同样的方式初始化了 PB0,那么 PA0 与中断线的链接将被清除,而直接链接 PB0 到中断线 0。

3) 配置中断优先级(NVIC)并使能中断。

我们设置好中断线和 GPIO 映射关系,然后又设置好了中断的触发模式等初始化参数。既然是外部中断,涉及到中断我们当然还要设置 NVIC 中断优先级。这个在前面已经讲解过,这里我们就接着上面的范例,设置中断线 0 的中断优先级并使能外部中断 0 的方法为:

```
HAL_NVIC_SetPriority(EXTI0_IRQn,2,1);    //抢占优先级为2,子优先级为1
HAL_NVIC_EnableIRQ(EXTI0_IRQn);          //使能中断线2
```

上面这段代码相信大家都不陌生,我们在前面的串口实验的时候讲解过,这里不再讲解。

4) 编写中断服务函数。

我们配置完中断优先级之后,接着要做的就是编写中断服务函数。中断服务函数的名字是在 HAL 库中事先有定义的。这里需要说明一下,STM32L1 的 IO 口外部中断函数只有 7 个,分别为:

```
void EXTI0_IRQHandler();
void EXTI1_IRQHandler();
void EXTI2_IRQHandler();
void EXTI3_IRQHandler();
void EXTI4_IRQHandler();
void EXTI9_5_IRQHandler();
void EXTI15_10_IRQHandler();
```

中断线 0-4 每个中断线对应一个中断函数,中断线 5-9 共用中断函数 EXTI9_5_IRQHandler,中断线 10-15 共用中断函数 EXTI15_10_IRQHandler。一般情况下,我们可以把中断控制逻辑直接编写在中断服务函数中,但是 HAL 库把中断处理过程进行了简单封装,请看下面步骤 5 讲解。

5) 编写中断处理回调函数 HAL_GPIO_EXTI_Callback

在使用 HAL 库的时候,我们也可以跟使用标准库一样,在中断服务函数中编写控制逻辑。但是 HAL 库为了用户使用方便,它提供了一个中断通用入口函数 HAL_GPIO_EXTI_IRQHandler,在该函数内部直接调用回调函数 HAL_GPIO_EXTI_Callback。

我们可以看看 HAL_GPIO_EXTI_IRQHandler 函数定义:

```
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
    if( __HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)
    {
        HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);
        HAL_GPIO_EXTI_Callback(GPIO_Pin);
    }
}
```


该函数实现的作用非常简单，就是清除中断标志位，然后调用回调函数 HAL_GPIO_EXTI_Callback() 实现控制逻辑。所以我们编写中断控制逻辑将跟串口实验类似，在中断服务函数中直接调用外部中断共用处理函数 HAL_GPIO_EXTI_IRQHandler，然后在回调函数 HAL_GPIO_EXTI_Callback 中通过判断中断是来自哪个IO 口编写相应的中断服务控制逻辑。

讲到这里，相信大家对STM32 的IO 口外部中断已经有了一定的了解。下面我们再总结一下配置IO 口外部中断的一般步骤：

- 1) 使能IO 口时钟。
- 2) 调用函数HAL_GPIO_Init 设置IO 口模式，触发条件，使能SYSCFG 时钟以及设置IO 口与中断线的映射关系。
- 3) 配置中断优先级（NVIC）并使能中断。
- 4) 在中断服务函数中调用外部中断共用入口函数 HAL_GPIO_EXTI_IRQHandler。
- 5) 编写外部中断回调函数HAL_GPIO_EXTI_Callback。通过以上几个步骤的设置，我们就可以正常使用外部中断了。本章，我们要实现同第二章差不多的功能，但是这里我们使用的是中断来检测按键，还是 KEY1 控制LED1；KEY2 控制LED2。

4.2 硬件设计

本实验用到的硬件资源和第二章实验的完全一样，这里就不重复介绍了，大家可自行翻阅。

4.3 STM32CubeMX 配置外部中断和软件设计

本小节我们将教会大家用 STM32CubeMX 配置外部中断相关的初始化代码。关于 STM32CubeMX 的配置，从本小节开始，我们对于每个实验只讲解实验相关关键配置部分，如果大家不知道具体怎么进入相关界面，请仔细看看前面几个章节实验。

对于外部中断的配置，首先在 MCU 引脚配置图界面选择相应的 GPIO 设置其模式为外部 中断模式。这里以PC5 为例，如下图4.3.1：

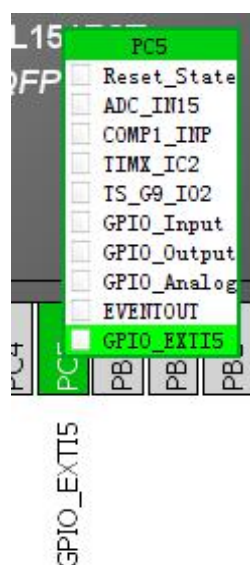


图 4.3.1 PC5 引脚设置

同样的方法依次配置 PC13，然后我们打开依次点进入GPIO 详细配置界
PinConfiguration，界面会列出2 个IO 口的配置信息。我们选中PC5，看看此时IO 口配置信息详
情如下图4. 3. 2 所示：

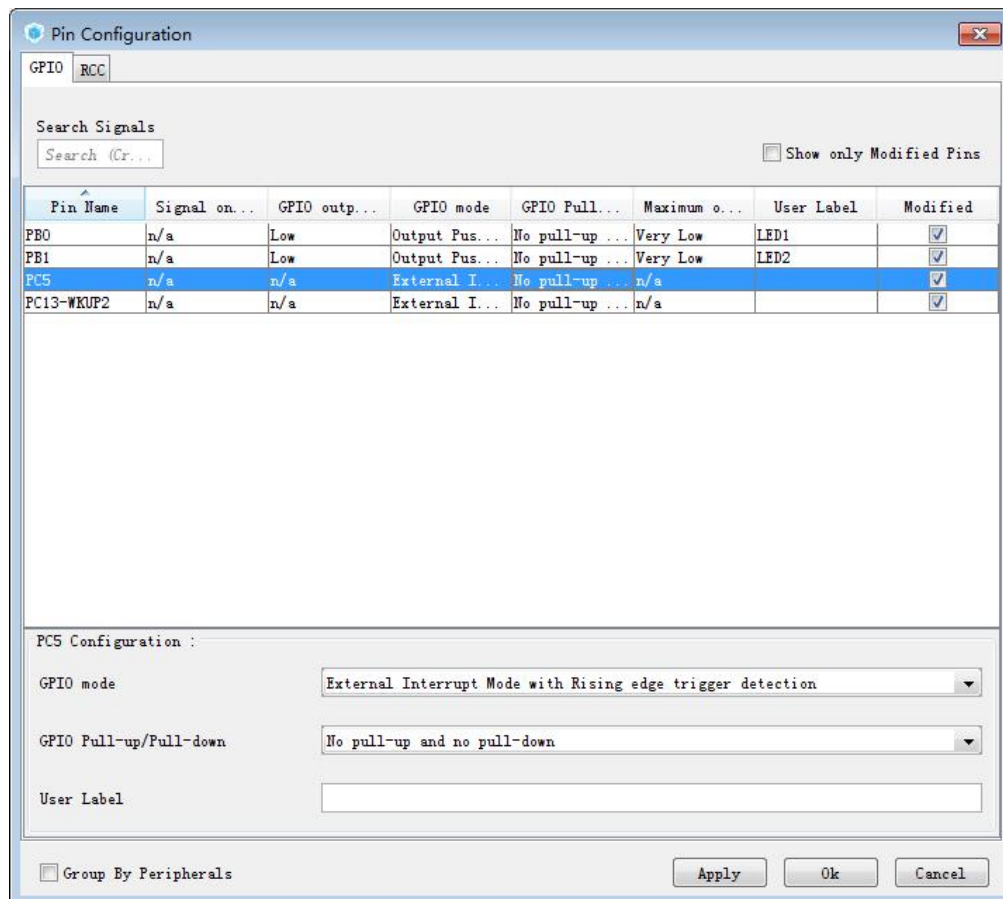


图4. 3. 2 GPIO 配置详情界面

从上图界面可以看出，当我们配置 IO 口作为外部中断触发引脚之后，其详细配置界面便只有三个选项。第一个选项 GPIO mode 用来设置外部中断触发方法，上升沿触发还是下降沿触发还是双边沿触发。第二个选项 GPIOPull-up/Pull-down 用来设置是默认上拉还是下拉。这两个参数根据我们前面讲解的外部中断知识就很好理解了。配置好 IO 口信息之后，接下来就需要配置 NVIC 中断优先级设置。依次点击 Configuration->NVIC，进入 NVIC 配置界面。在界面可以看到有四个外部中断线可配置，这是因为我们前面开启了 2 个 IO 口的外部中断（对应 2 个外部中断线。我们按照实验讲解，依次配置 2 个中断线的 NVIC 即可，配置完成后如下图 4. 3. 3 所示

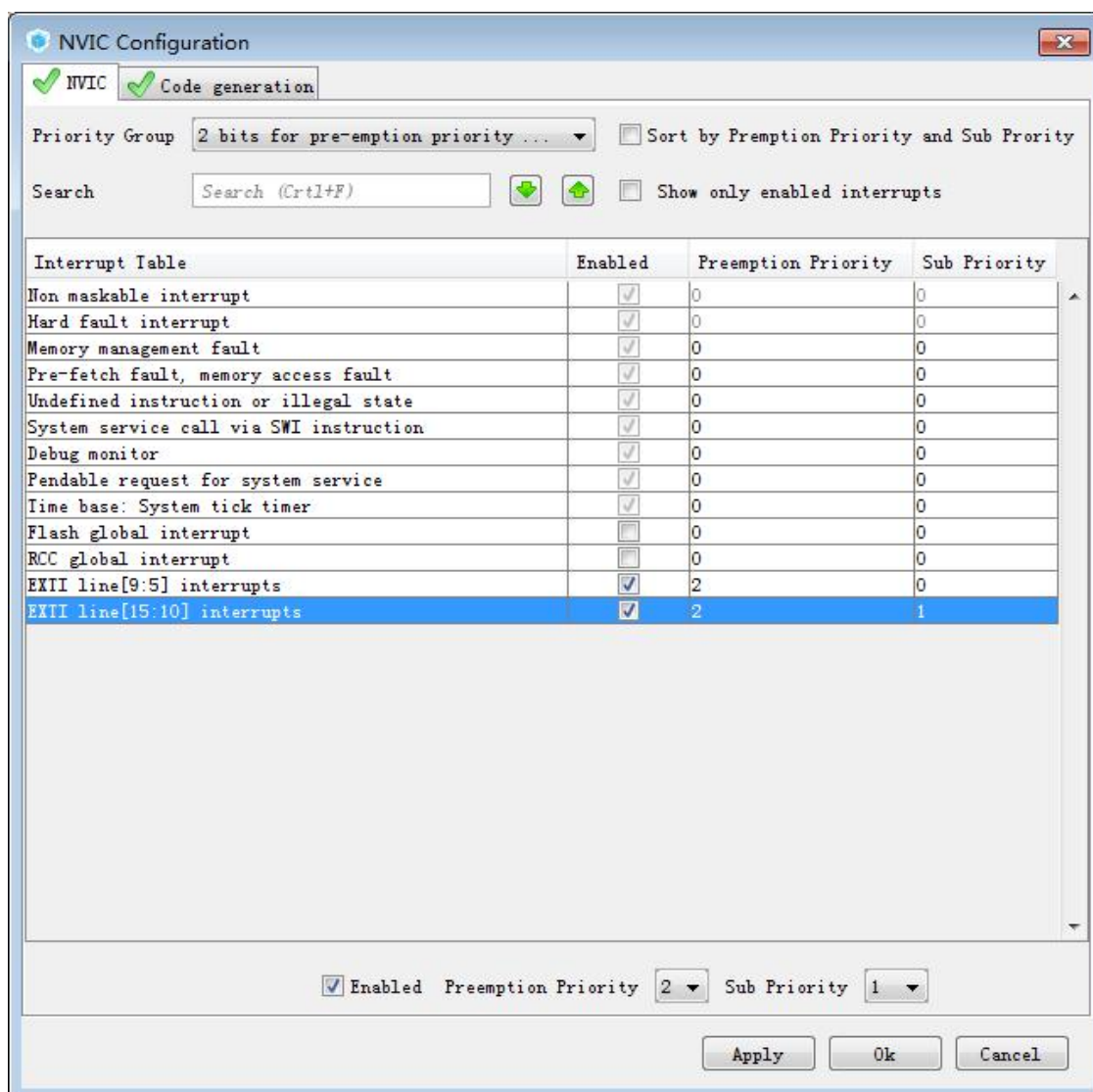


图4.3.3 NVIC 配置界面

最后生成工程，为了篇幅考虑，这里我们就不再列出生成的关键代码。回调函数 HAL_GPIO_EXTI_Callback 的内容软件是无法自动生成的，需要我们自己编写。

我们直接打开中断实验工程，可以看到文件代码如下：

```
void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct;

    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOC_CLK_ENABLE();
    __HAL_RCC_GPIOH_CLK_ENABLE();
    __HAL_RCC_GPIOB_CLK_ENABLE();

    /*Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOB, LED1_Pin|LED2_Pin, GPIO_PIN_RESET);
    /*Configure GPIO pins : PC13 PC5 */
    GPIO_InitStruct.Pin = GPIO_PIN_13|GPIO_PIN_5;
    GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    HAL_GPIO_Init(GPIOC, &GPIO_InitStruct);

    /*Configure GPIO pins : PBPin PBPin */
    GPIO_InitStruct.Pin = LED1_Pin|LED2_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_NOPULL;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
    HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);

    /* EXTI interrupt init*/
    HAL_NVIC_SetPriority(EXTI9_5_IRQn, 2, 0);
    HAL_NVIC_EnableIRQ(EXTI9_5_IRQn);
    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 2, 1);
    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
}
```

外部中断初始化函数 MX_GPIO_Init 用来配置 I/O 口外部中断相关步骤并使能中断，另一个函数 HAL_GPIO_EXTI_Callback 是外部中断共用回调函数，用来处理所有外部中断真正的控制逻辑。其他 2 个都是中断服务函数。

void EXTI9_5_IRQHandler(void) 是外部中断 9 的服务函数；

void EXTI15_10_IRQHandler(void) 是外部中断 2 的服务函数；

首先是外部中断初始化函数 MX_GPIO_Init(void)，该函数内部主要做了两件事情。首先是调用 I/O 口初始化函数 HAL_GPIO_Init 来初始化 I/O 口其次是设置中断优先级并使能中断线。接下来我们看看外部中断服务函数，一共 2 个。所有的中断服务函数内部都只调用了同样一个函数 HAL_GPIO_EXTI_IRQHandler，该函数是外部中断共用入口函数，函数内部会进行中断标志位清零，并且调用中断处理共用回调函数 HAL_GPIO_EXTI_Callback。

```
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    HAL_Delay(100);
    switch(GPIO_Pin)
    {
        case GPIO_PIN_5:
            if(HAL_GPIO_ReadPin(LED2_GPIO_Port, LED2_Pin))
            {
                CL_LED2();
            }
            else
            {
                OP_LED2();
            }
            break;
        case GPIO_PIN_13:
            if(HAL_GPIO_ReadPin(LED1_GPIO_Port, LED1_Pin))
            {
                CL_LED1();
            }
            else
            {
                OP_LED1();
            }
            break;
        default:
            break;
    }
}
```

最后是外部中断回调函数 `HAL_GPIO_EXTI_Callback`，该函数用来编写真正的外部中断控制逻辑。该函数有一个入口参数就是 IO 口序号。所以我们在该函数内部，一般通过判断 IO 口序号值来确定中断是来自哪个 IO 口，也就是哪个中断线，然后编写相应的控制逻辑。所以在该函数内部，我们通过 `switch` 语句判断 IO 口来源，例如是来自 `GPIO_PIN_5`，那么一定是来自 PC5，因为中断线一次只能连接一个 IO 口，而 2 个 IO 口中序号为 5 的 IO 口只有 PC5，所以中断线 5 一定是连接 PC5，也就是外部中断由 PC5 触发。

接下来我们看看主函数，`main` 函数代码如下：

```
int main(void)
{
    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();

    while (1)
    {

    }
}
```

该部分代码很简单，先进行各项初始化之后，在 while 死循环中不需要处理任何任务。当有某个外部按键按下之后，会触发中断服务函数做出相应的反应。

4.4 下载验证

在编译成功之后，我们就可以下载代码到 STM32 开发板上，实际验证一下我们的程序是否正确。下载代码后我们按下 KEY1 后可以看到 LED1 会点亮，再按一次后 LED1 熄灭。

第五章独立看门狗（IWDG）实验

这一章，我们将向大家介绍如何使用 STM32L151 的独立看门狗（以下简称 IWDG）。STM32L151 内部自带了 2 个看门狗：独立看门狗（IWDG）和窗口看门狗（WWDG）。这一章我们只介绍独立看门狗，窗口看门狗将在下一章介绍。在本章中，我们将通过按键 KEY1 来喂狗，然后通过 LED1 提示复位状态。本章分为如下几个部分：

- 5.1 STM32L1 独立看门狗简介
- 5.2 硬件设计
- 5.3 STM32CubeMX 配置 IWDG 和软件计
- 5.4 下载验证

5.1 STM32L1 独立看门狗简介

STM32L1 的独立看门狗由内部专门的 37Khz 低速时钟（LSI）驱动，即使主时钟发生故障，它也仍然有效。这里需要注意独立看门狗的时钟是一个内部 RC 时钟，所以并不是准确的 37Khz，而是在 15~47Khz 之间的一个可变化的时钟，只是我们在估算的时候，以 37Khz 的频率来计算，看门狗对时间的要求不是很精确，所以时钟有一些偏差，都是可以接受的。

单片机系统在外界的干扰下会出现程序跑飞的现象导致出现死循环，看门狗电路就是为了避免这种情况的发生。看门狗的作用就是在一定时间内（通过定时计数器实现）没有接收喂狗信号（表示 MCU 已经挂了），便实现处理器的自动复位重启（发送复位信号）。

独立看门狗有几个寄存器与我们这节相关，我们分别介绍这几个寄存器，首先是关键字寄存器 IWDG_KR，该寄存器的各位描述如图 5.1.1 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																KEY[15:0]															
																w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位 31:16 保留，必须保持复位值。

位 15:0 **KEY[15:0]**：键值 (Key value)（只写位，读为 0000h）

必须每隔一段时间便通过软件对这些位写入键值 AAAAh，否则当计数器计数到 0 时，看门狗会产生复位。

写入键值 5555h 可使能对 IWDG_PR 和 IWDG_RLR 寄存器的访问

写入键值 CCCCh 可启动看门狗（选中硬件看门狗选项的情况除外）

图 5.1.1 IWDG_KR 寄存器各位描述

在关键字寄存器 (IWDG_KR) 中写入 0xCCCC，开始启用独立看门狗；此时计数器开始从其复位值 0xFFFF 递减计数。当计数器计数到末尾 0x000 时，会产生一个复位信号 (IWDG_RESET)。无论何时，只要关键字寄存器 IWDG_KR 中被写入 0xAAAA，IWDG_RLR 中的值就会被重新加载到计数器中从而避免产生看门狗复位。

IWDG_PR 和 IWDG_RLR 寄存器具有写保护功能。要修改这两个寄存器的值，必须先向 IWDG_KR 寄存器中写入 0x5555。将其他值写入这个寄存器将会打乱操作顺序，寄存器将重新被保护。重装载操作（即写入 0xAAAA）也会启动写保护功能。

接下来，我们介绍预分频寄存器（IWDG_PR），该寄存器用来设置看门狗时钟的分频系数，最低为 4，最高位 256，该寄存器是一个 32 位的寄存器，但是我们只用了最低 3 位，其他都是保留位。预分频寄存器各位定义如图 5.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																												PR[2:0]			
																												rw	rw	rw	

位 31:3 保留，必须保持复位值。

位 2:0 **PR[2:0]**: 预分频器 (Prescaler divider)

这些位受写访问保护，通过软件设置这些位来选择计数器时钟的预分频因子。若要更改预分频器的分频系数，IWDG_SR 的 PVU 位必须为 0。

000: 4 分频 100: 64 分频
001: 8 分频 101: 128 分频
010: 16 分频 110: 256 分频
011: 32 分频 111: 256 分频

注意：读取该寄存器会返回 VDD 电压域的预分频器值。如果正在对该寄存器执行写操作，则读取的值可能不是最新的/有效的。因此，只有在 IWDG_SR 寄存器中的 PVU 位为 0 时，从寄存器读取的值才有效。

图 5.1.2 IWDG_PR 寄存器各位描述

在介绍完 IWDG_PR 之后，我们介绍一下重装载寄存器 IWDG_RLR。该寄存器用来保存重装载到计数器中的值。该寄存器也是一个 32 位寄存器，但是只有低 12 位是有效的，该寄存器的各位描述如图 5.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved																				RL[11:0]											
																				rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:12 保留，必须保持复位值。

位 11:0 **RL[11:0]**: 看门狗计数器重载值 (Watchdog counter reload value)

这些位受写访问保护，请参考之前介绍。这个值由软件设置，每次对 IWDG_CR 寄存器写入值 AAAAh 时，这个值就会重装载到看门狗计数器中。之后，看门狗计数器便从该装载的值开始递减计数。超时周期由该值和时钟预分频器共同决定。

若要更改重载值，IWDG_SR 中的 RVU 位必须为 0。

注意：读取该寄存器会返回 VDD 电压域的重载值。如果正在对该寄存器执行写操作，则读取的值可能不是最新的/有效的。因此，只有在 IWDG_SR 寄存器中的 RVU 位为 0 时，从寄存器读取的值才有效。

图 5.1.3 IWDG_RLR 重装载寄存器各位描述

只要对以上三个寄存器进行相应的设置，我们就可以启动 STM32L1 的独立看门狗。独立看门狗相关的 HAL 库操作函数在文件 stm32l1xx_hal_iwdg.c 和头文件 stm32l1xx_hal_iwdg.h 中。接下来我们讲解一下通过 HAL 库配置独立看门狗的步骤：

(1) 取消寄存器写保护，设置看门狗预分频系数和重装载值

首先我们必须取消 IWDG_PR 和 IWDG_RLR 寄存器的写保护，这样才可以设置寄存器 IWDG_PR 和 IWDG_RLR 的值。取消写保护和设置预分频系数以及重装载值在 HAL 库中是通过函数 HAL_IWDG_Init 实现的。该函数声明为：

```
HAL_StatusTypeDef HAL_IWDG_Init(IWDG_HandleTypeDef*hiwdg);
```

该函数只有一个入口参数 hiwdg，该参数是 IWDG_HandleTypeDef 结构体指针类型。接下来我们看看结构体 IWDG_HandleTypeDef 定义：

```
typedef struct
{
    IWDG_TypeDef                *Instance;

    IWDG_InitTypeDef            Init;

    HAL_LockTypeDef              Lock;

    IOHAL_IWDG_StateTypeDef     State;
} IWDG_HandleTypeDef;
```

成员变量 Instance 用来设置看门狗寄存器基地址，实际上在 HAL 库中已经通过标识符定义了，这里对于独立看门狗直接设置为标识符 IWDG 即可。

成员变量 Init 是一个 IWDG_InitTypeDef 结构体类型，该结构体只有 2 个成员变量，分别用来设置独立看门狗的预分频系数和重装载值，定义如下：

```
typedef struct
{
    uint32_t Prescaler;

    uint32_t Reload;
} IWDG_InitTypeDef;
```

成员变量 Lock 是一个锁存变量，该变量在 HAL 库中当操作配置 IWDG 之前设置为锁住 LOCK，当配置操作完成之后设置为 UNLOCK，实际上是一个操作状态标识符。

成员变量 State 也是 HAL 定义的一个过程标识符，用来记录 IWDG 处理状态。

HAL_IWDG_Init 函数使用的一般方法为：

```
IWDG_HandleTypeDef IWDG_Handler;    //独立看门狗句柄
IWDG_Handler.Instance=IWDG;         //独立看门狗
IWDG_Handler.Init.Prescaler=IWDG_PRESCALER_64; //设置 IWDG 分频系数
IWDG_Handler.Init.Reload=1000;      //重装载值
HAL_IWDG_Init(&IWDG_Handler);
```

上面程序的作用是初始化 IWDG，设置分频系数为 64，重装载值为 1000。设置完预分频系数和重装载值后，我们就可以知道看门狗的喂狗时间（也就是看门狗溢出时间），该时间的计算方式为： $Tout = ((4 \times 2^{\text{prer}}) \times \text{rlr}) / 37$ 其中 Tout 为看门狗溢出时间（单位为 ms），prer 为看门狗时钟预分频值（IWDG_PR 值）范围为 0~7；rlr 为看门狗的重装载值（IWDG_RLR 的值），比如我们设定 prer 值为 4（4 代表的是 64 分频，HAL 库中可以使用宏定义标识符 IWDG_PRESCALER_64），rlr 值为 1000，那么就可以得到 $Tout = 64 \times 1000 / 37 = 1800\text{ms}$ ，这样，看门狗的溢出时间就是 2s，只要你在 1 秒钟之内，有一次写入 0xAAAA 到 IWDG_KR，就不会导致看门狗复位（当然写入多次也是可以的）。这里需要提醒大家的是，看门狗的时钟不是准确的 37Khz，所以在喂狗的时候，最好不要太晚了，否则，有可能发生看门狗复位。

(2) 重载计数值喂狗（向 IWDG_KR 写入 0XAAAA）

在 HAL 中重载计数值的函数是 HAL_IWDG_Refresh，该函数声明为：

```
HAL_StatusTypeDef HAL_IWDG_Refresh(IWDG_HandleTypeDef*hiwdg);
```

该函数有一个入口参数为前面讲解的 IWDG_HandleTypeDef 结构体类型指针，它的作用是把值 0xAAAA 写入到 IWDG_KR 寄存器，从而触发计数器重载，即实现独立看门狗的喂狗操作。

通过上面 2 个步骤，我们就可以启动 STM32L1 的独立看门狗了，使能了看门狗，在程序里就必须间隔一定时间喂狗，否则将导致程序复位。利用这一点，我们本章将通过一个 LED 灯来指示程序是否重启，来验证 STM32L1 的独立看门狗。在配置看门狗后，LED1 将常亮，如果 KEY1 按键按下，就喂狗，只要 KEY1 不停的按，看门狗就一直不会产生复位，保持 LED1 的常亮，一旦超过看门狗定溢出时间（Tout）还没按，那么将会导致程序重启，这将导致 LED1 熄灭一次。

5.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 LED1
- 2) KEY1 按键
- 3) 独立看门狗

前面两个在之前都有介绍，而独立看门狗实验的核心是在 STM32L151 内部进行，并不需要外部电路。但是考虑到指示当前状态和喂狗等操作，我们需要 2 个 IO 口，一个用来输入喂狗信号，另外一个用来指示程序是否重启。喂狗我们采用板上的 KEY1 键来操作，而程序重启，则是通过 LED1 来指示的。

5.3 STM32CubeMX 配置 IWDG 和软件设计

使用 STM32CubeMX 工具配置 IWDG 生成初始化代码的步骤非常简单，我们只需要使能 IWDG，同时配置 IWDG 的预分频系数和自动装载值即可。

首先我们看看使能 IWDG 的方法，在 Pinout 界面的 Peripherals 一栏选择 IWDG，然后勾选上 Activated 选项即可使能 IWDG。操作方法如下图 5.3.1：

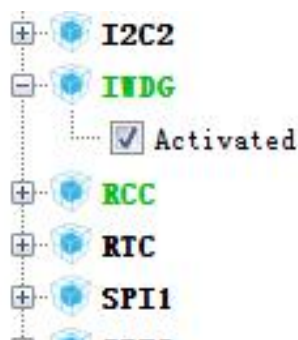


图 5.3.1 IWDG 配置选项

接下来依次点击 Configuration→IWDG，进入 IWDG 参数配置界面。进入界面后，我们依次配置 IWDG 的预分频系数和自动装载值，这两个参数的含义我们在前面已经讲解。IWDG Configuration 配置界面如下图 5.3.2 所示：

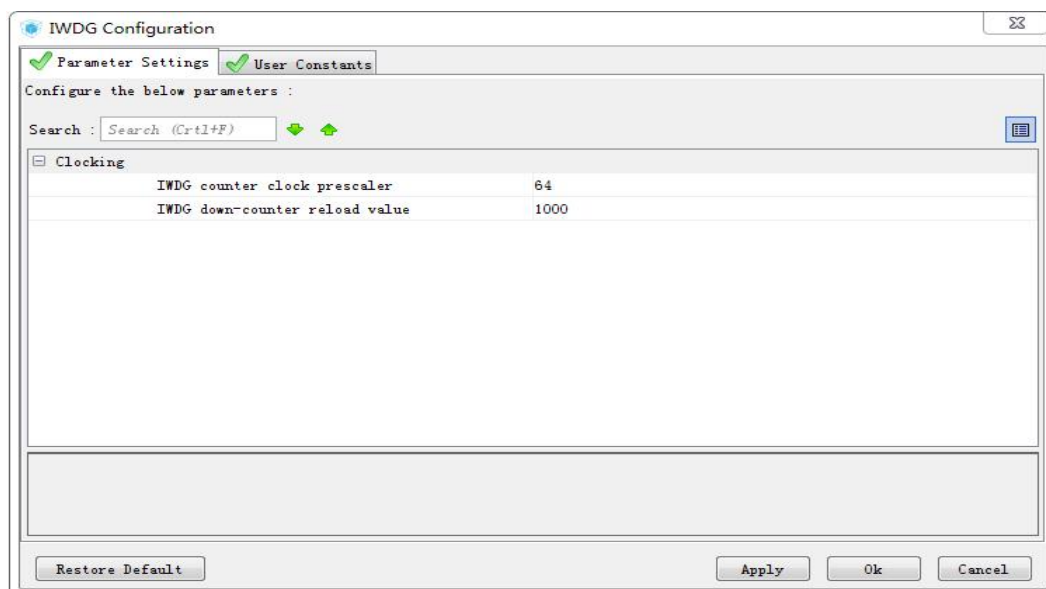


图 5.3.2 IWDG 参数配置界面

这里我们配置预分频系数为 64，同时自动装载值为 500 即可。配置完成后生成实验工程。在生成的工程中，打开 main.c 文件可以看到生成的函数 MX_IWDG_Init。用户根据自己需求在合适的程序段中编写开启看门狗和喂狗操作。代码如下：

```
void MX_IWDG_Init(void)
{
    hiwdg.Instance = IWDG;
    hiwdg.Init.Prescaler = IWDG_PRESCALER_64;
    hiwdg.Init.Reload = 1000;
    if (HAL_IWDG_Init(&hiwdg) != HAL_OK)
    {
        Error_Handler(__FILE__, __LINE__);
    }
}
//喂独立看门狗

void IWDG_Feed(void)
{
    HAL_IWDG_Refresh(&IWDG_Handler); //重装载
}
```

该代码就 2 个函数，void MX_IWDG_Init(void) 是独立看门狗初始化函数，就是按照上面介绍的步骤 1 来初始化独立看门狗。该函数有 2 个参数，分别用来设置预分频数与重装载寄存器的值。通过这两个参数，就可以大概知道看门狗复位的时间周期为多少了。其计算方式上面有详细的介绍，这里不再多说了。

void IWDG_Feed(void)函数，该函数用来喂狗，因为 STM32 的喂狗只需要向关键字寄存器写入 0XAAAA 即可，也就是调用库函数 HAL_IWDG_Refresh，所以这个函数也是很简单的。

接下来我们看看主函数，主程序里面我们先初始化一下系统代码，然后启动按键输入和看门狗，在看门狗开启后马上点亮 LED1，并进入死循环等待按键的输入，一旦 KEY1 有按键，则喂狗，否则等待 IWDG 复位的到来。该部分代码如下：

```
int main(void)
{
    HAL_Init();SystemClock_Config();MX_GPIO_Init();MX_IWDG_Init();HAL_Delay(1000);OP_LED1();
    while(1)
    {
        if(key1())
        {
            HAL_Delay(1000);if(key1())
            {
                IWDG_Feed();
            }
        }
    }
}
```

5.4 下载验证

在编译成功之后，我们就可以下载代码到 STM32 开发板上，实际验证一下，我们的程序是否正确。下载代码后可以看到 LED1 不停的闪烁，这时我们试试不停的按 KEY1 按键（喂狗），可以看到 LED1 就常亮了不会再闪烁。说明我们的实验是成功的。

第六章定时器中断实验

这一章，我们将向大家介绍如何使用 STM32L151 的通用定时器。在本章中，我们将使用 TIM3 的定时器中断来控制 LED1 的翻转，在主函数用 LED1 的翻转来提示程序正在运行。本章，我们选择难度适中的通用定时器来介绍，本章将分为如下几个部分：

- 6.1 STM32L151 通用定时器简介
- 6.2 硬件设计
- 6.3 STM32CubeMX 配置定时器更新中断功能和软件设计
- 6.4 下载验证

6.1 STM32L151 通用定时器简介

STM32L151 的通用定时器包含一个 16 位或 32 位自动重载计数器 (CNT)，该计数器由可编程预分频器 (PSC) 驱动。STM32L151 的通用定时器可以被用于：测量输入信号的脉冲长度 (输入捕获) 或者产生输出波形 (输出比较和 PWM) 等。使用定时器预分频器和 RCC 时钟控制器预分频器，脉冲长度和波形周期可以在几个微秒到几个毫秒间调整。STM32L151 的每个通用定时器都是完全独立的，没有互相共享的任何资源。

STM3 的通用定时器功能包括：

- 1) 16 位/32 位向上、向下、向上/向下自动装载计数器 (TIMx_CNT)
- 2) 16 位可编程 (可以实时修改) 预分频器 (TIMx_PSC)，计数器时钟频率的分频系数为 1~65535 之间的任意数值。
- 3) 4 个独立通道 (TIMx_CH1~4)，这些通道可以用来作为：
 - A. 输入捕获
 - B. 输出比较
 - C. PWM 生成 (边缘或中间对齐模式)，注意：TIM9~TIM14 不支持中间对齐模式
 - D. 单脉冲模式输出
- 4) 可使用外部信号 (TIMx_ETR) 控制定时器和定时器互连 (可以用 1 个定时器控制另外一个定时器) 的同步电路。
- 5) 如下事件发生时产生中断/DMA：
 - A. 更新：计数器向上溢出/向下溢出，计数器初始化 (通过软件或者内部/外部触发)
 - B. 触发事件 (计数器启动、停止、初始化或者由内部/外部触发计数)
 - C. 输入捕获
 - D. 输出比较
 - E. 支持针对定位的增量 (正交) 编码器和霍尔传感器电路
 - F. 触发输入作为外部时钟或者按周期的电流管理

下面我们介绍一下与我们这章的实验密切相关的几个通用定时器的寄存器
首先是控制寄存器 1 (TIMx_CR1) 该寄存器的各位描述如图 6.1.1 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved						CKD[1:0]		ARPE	CMS		DIR	OPM	URS	UDIS	CEN
						rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 0 **CEN**: 计数器使能 (Counter enable)

0: 禁止计数器

1: 使能计数器

注意: 只有事先通过软件将 **CEN** 位置 1, 才可以使用外部时钟、门控模式和编码器模式。而触发模式可通过硬件自动将 **CEN** 位置 1。

在单脉冲模式下, 当发生更新事件时会自动将 **CEN** 位清零。

图 6. 1. 1TIMx_CR1 寄存器各位描述

在本实验中, 我们只用到了 TIMx_CR1 的最低位, 也就是计数器使能位, 该位必须置 1, 才能让定时器开始计数。接下来介绍第二个与我们这章密切相关的寄存器: DMA/中断使能寄存器 (TIMx_DIER)。该寄存器是一个 16 位的寄存器, 其各位描述如图 6. 1. 2 所示:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TDE	Res.	CC4DE	CC3DE	CC2DE	CC1DE	UDE	Res.	TIE	Res.	CC4IE	CC3IE	CC2IE	CC1IE	UIE
rw	rw		rw	rw	rw	rw	rw		rw		rw	rw	rw	rw	rw

位 0 **UIE**: 更新中断使能 (Update interrupt enable)

0: 禁止更新中断

1: 使能更新中断

图 6. 1. 2TIMx_DIER 寄存器各位描述

这里我们同样仅关心它的第 0 位, 该位是更新中断允许位, 本章用到的是定时器的更新中断所以该位要设置为 1, 来允许由于更新事件所产生的中断。

接下来我们看第三个与我们这章有关的寄存器: 预分频寄存器 (TIMx_PSC)。该寄存器用设置对时钟进行分频, 然后提供给计数器, 作为计数器的时钟。该寄存器的各位描述如图 6. 1. 3 所示:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PSC[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **PSC[15:0]**: 预分频器值 (Prescaler value)

计数器时钟频率 CK_CNT 等于 $f_{CK_PSC} / (PSC[15:0] + 1)$ 。

PSC 包含在每次发生更新事件时要装载到实际预分频器寄存器的值。

图 6. 1. 3TIMx_PSC 寄存器各位描述

这里, 定时器的时钟来源有 4 个:

1) 内部时钟 (CK_INT)

2) 外部时钟模式 1: 外部输入脚 (TIx)

3) 外部时钟模式 2: 外部触发输入 (ETR) 仅适用于 TIM2、TIM3、TIM4

4) 内部触发输入 (ITRx): 使用 A 定时器作为 B 定时器的预分频器 (A 为 B 提供时钟)

这些时钟, 具体选择哪个可以通过 TIMx_SMCR 寄存器的相关位来设置。这里的 CK_INT 时钟是从 APB1 倍频的来的, 除非 APB1 的时钟分频数设置为 1 (一般都不会是 1), 否则通用定时器 TIMx 的时钟是 APB1 时钟的 2 倍, 当 APB1 的时钟不分频的时候, 通用定时器 TIMx 的时钟就

等于 APB1 的时钟。这里还要注意的就是高级定时器以及 TIM9~TIM11 的时钟不是来自 APB1，而是来自 APB2 的。

这里顺带介绍一下 TIMx_CNT 寄存器，该寄存器是定时器的计数器，该寄存器存储了当前定时器的计数值。

接着我们介绍自动重装载寄存器 (TIMx_ARR)，该寄存器在物理上实际对应着 2 个寄存器。一个是程序员可以直接操作的，另外一个程序员看不到的，这个看不到的寄存器被叫做影子寄存器。事实上真正起作用的是影子寄存器。根据 TIMx_CR1 寄存器中 APRE 位的设置：APRE=0 时，预装载寄存器的内容可以随时传送到影子寄存器，此时 2 者是连通的；而 APRE=1 时，在每一次更新事件 (UEV) 时，才把预装载寄存器 (ARR) 的内容传送到影子寄存器。自动重装载寄存器的各位描述如图 6.1.4 所示

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ARR[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 15:0 **ARR[15:0]: 自动重载值 (Auto-reload value)**
 ARR 为要装载到实际自动重载寄存器的值。
 当自动重载值为空时，计数器不工作。

图 6.1.4 TIMx_ARR 寄存器各位描述

最后，我们要介绍的寄存器是：状态寄存器 (TIMx_SR)。该寄存器用来标记当前与定时器相关的各种事件/中断是否发生。该寄存器的各位描述如图 6.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
Reserved			CC4OF	CC3OF	CC2OF	CC1OF	Reserved			TIF	Res	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc_w0	rc_w0	rc_w0	rc_w0				rc_w0		rc_w0	rc_w0	rc_w0	rc_w0	

位 0 **UIF: 更新中断标志 (Update interrupt flag)**

- 该位在发生更新事件时通过硬件置 1。但需要通过软件清零。
0: 未发生更新。
1: 更新中断挂起。该位在以下情况下更新寄存器时由硬件置 1:
 - 上溢或下溢 (对于 TIM2 到 TIM5) 以及当 TIMx_CR1 寄存器中 UDIS = 0 时。
 - TIMx_CR1 寄存器中的 URS = 0 且 UDIS = 0，并且由软件使用 TIMx_EGR 寄存器中的 UG 位重新初始化 CNT 时。
- TIMx_CR1 寄存器中的 URS=0 且 UDIS=0，并且 CNT 由触发事件重新初始化

图 6.1.5 TIMx_SR 寄存器各位描述

只要对以上几个寄存器进行简单的设置，我们就可以使用通用定时器了，并且可以产生中断这一章，我们将使用定时器产生中断，然后在中断服务函数里面翻转 LED1 上的电平，来指示定时器中断的产生。接下来我们以通用定时器 TIM3 为实例，来说明要经过哪些步骤，才能达到这个要求，并产生中断。这里我们就对每个步骤通过库函数的实现方式来描述。首先要提到的是，定时器相关的库函数主要集中在 HAL 库文件 stm32l1xx_hal_tim.h 和 stm32l1xx_hal_tim.c 文件中。定时器配置步骤如下：

1) TIM3 时钟使能。

HAL 中定时器使能是通过宏定义标识符来实现对相关寄存器操作的，方法如下：

```
HAL_RCC_TIM3_CLK_ENABLE(); //使能 TIM3 时钟
```

2) 初始化定时器参数, 设置自动重装值, 分频系数, 计数方式等。

在 HAL 库中定时器的初始化参数是通过定时器初始化函数 HAL_TIM_Base_Init 实现的：

```
HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef*htim);
```

该函数只有一个入口参数，就是 TIM_HandleTypeDef 类型结构体指针，结构体类型为下面我们看看这个结构体的定义：

```
typedef struct
{
    TIM_TypeDef          *Instance;
    TIM_Base_InitTypeDef Init;
    HAL_TIM_ActiveChannel Channel;
    DMA_HandleTypeDef    *hdma[7];
    TIM_DMA_Handle_index /*
    HAL_LockTypeDef       Lock;
    __IO HAL_TIM_StateTypeDef State;
}TIM_HandleTypeDef;
```

第一个参数 Instance 是寄存器基地址。和串口、看门狗等外设一样，一般外设的初始化结构体定义的第一个成员变量都是寄存器基地址。这在 HAL 中都定义好了，比如要初始化串口 1，那么 Instance 的值设置为 TIM1 即可。

第二个参数 Init 为真正的初始化结构体 TIM_Base_InitTypeDef 类型。该结构体定义如下：

```
Typedef struct
{
    uint32_t Prescaler;          //预分频系数
    uint32_t CounterMode;        //计数方式
    uint32_t Period;             //自动装载值 ARR
    uint32_t ClockDivision;      //时钟分频因子
    uint32_t RepetitionCounter;}
```

该初始化结构体中，参数 Prescaler 是用来设置分频系数的，刚才上面有讲解。参数 CounterMode 是用来设置计数方式，可以设置为向上计数，向下计数方式还有中央对齐计数方式，比较常用的是向上计数模式 TIM_CounterMode_Up 和向下计数模式 TIM_CounterMode_Down。参数 Period 是设置自动重载计数周期值。参数 ClockDivision 是用来设置时钟分频因子，也就是定时器时钟频率 CK_INT 与数字滤波器所使用的采样时钟之间的分频比。参数 RepetitionCounter 用来设置重复计数器寄存器的值，用在高级定时器中。

第三个参数 Channel 用来设置活跃通道。前面我们讲解过，每个定时器最多有四个通道可以用来做输出比较，输入捕获等功能之用。这里的 Channel 就是用来设置活跃通道的，取值范围为：HAL_TIM_ACTIVE_CHANNEL_1~HAL_TIM_ACTIVE_CHANNEL_4。

第四个 hdma 是定时器的 DMA 功能时用到，为了简单起见，我们暂时不讲解太复杂。

第五个参数 Lock 和 State，是状态过程标识符，是 HAL 库用来记录和标志定时器处理过程。定时器初始化范例如下：

```
TIM_HandleTypeDef htim3;  
htim3.Instance = TIM3;  
htim3.Init.Prescaler = 6400-1;  
htim3.Init.CounterMode = TIM_COUNTERMODE_UP;  
htim3.Init.Period = 10000-1;  
htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;  
HAL_TIM_Base_Init(&htim3)
```

3) 使能定时器更新中断，使能定时器

HAL 库中，使能定时器更新中断和使能定时器两个操作可以在函数 HAL_TIM_Base_Start_IT() 中一次完成的，该函数声明如下：

```
HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef*htim);
```

该函数非常好理解，只有一个入口参数。调用该定时器之后，会首先调用 HAL_TIM_ENABLE_IT 宏定义使能更新中断，然后调用宏定义 HAL_TIM_ENABLE 使能相应的定时器。这里我们分别列出单独使能/关闭定时器中断和使能/关闭定时器方法：

```
HAL_TIM_ENABLE_IT(htim,TIM_IT_UPDATE);//使能句柄指定的定时器更新中断  
HAL_TIM_DISABLE_IT (htim,TIM_IT_UPDATE);//关闭句柄指定的定时器更新  
__中断 HAL_TIM_ENABLE(htim);//使能句柄 htim 指定的定时器  
__ HAL_TIM_DISABLE(htim);//关闭句柄 htim 指定的定时器
```

4) TIM3 中断优先级设置。

在定时器中断使能之后，因为要产生中断，必不可少的要设置 NVIC 相关寄存器，设置中断优先级。之前多次讲解到中断优先级的设置，这里就不重复讲解。

和串口等其他外设一样，HAL 库为定时器初始化定义了回调函数 HAL_TIM_Base_MspInit。一般情况下，与 MCU 有关的时钟使能，以及中断优先级配置我们都会放在该回调函数内部。函数声明如下：

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef*htim);
```

对于回调函数，这里我们就不做过多讲解，大家只需要重写这个函数即可。

5) 编写中断服务函数。

在最后，还是要编写定时器中断服务函数，通过该函数来处理定时器产生的相关中断。通常情况下，在中断产生后，通过状态寄存器的值来判断此次产生的中断属于什么类型。然后执行相关的操作，我们这里使用的是更新（溢出）中断，所以在状态寄存器 SR 的最低位。在处理完中断之后应该向 TIM3_SR 的最低位写 0，来清除该中断标志。

跟串口一样，对于定时器中断，HAL 库同样为我们封装了处理过程。这里我们以定时器 3 的更新中断为例来讲解。

首先，中断服务函数是不变的，定时器 3 的中断服务函数为：

```
TIM3_IRQHandler();
```

一般情况下我们是在中断服务函数内部编写中断控制逻辑。但是 HAL 库为我们定义了新的定时器中断共用处理函数 HAL_TIM_IRQHandler，在每个定时器的中断服务函数内部，我们会调用该函数。该函数声明如下：

```
void HAL_TIM_IRQHandler(TIM_HandleTypeDef*htim);
```

而函数 HAL_TIM_IRQHandler 内部，会对相应的中断标志位进行详细判断，判断确定中断来源后，会自动清掉该中断标志位，同时调用不同类型中断的回调函数。所以我们的中断控制逻辑只用编写在中断回调函数中，并且中断回调函数中不需要清中断标志位。

比如定时器更新中断回调函数为：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef*htim);
```

跟串口中断回调函数一样，我们只需要重写该函数即可。对于其他类型中断，HAL 库同样提供了几个不同的回调函数，这里我们列出常用的几个回调函数：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef*htim);//更新中断
void HAL_TIM_OC_DelayElapsedCallback(TIM_HandleTypeDef*htim);//输出比较
void HAL_TIM_IC_CaptureCallback(TIM_HandleTypeDef*htim);//输入捕获
void HAL_TIM_TriggerCallback(TIM_HandleTypeDef*htim);//触发中断
```

对于这些回调函数的使用方法我们在后面用到的时候会给大家详细讲解。通过以上几个步骤，我们就可以达到我们的目的了，使用通用定时器的更新中断，来控制 LED1 的亮灭。

6.2 硬件设计

本实验用到的硬件资源有：

1) 指示灯 LED1 2) 定时器 TIM3

本章将通过 TIM3 的中断来控制 LED1 的亮灭，TIM3 属于 STM32L151 的内部资源，只需要软件设置即可正常工作。

6.3 STM32CubeMX 配置定时器更新中断功能和软件设计

经过前面多个章节的学习，大家对 STM32CubeMX 配置已经非常熟悉。从本章开始，出于篇幅考虑，我们将不再像之前章节一样讲解那么详细，我们将只会列出配置的关键点，然后生成工程，大家自行与光盘中提供的实验代码对照学习。

定时器 3 中断配置非常简单。配置步骤如下：

① Pinout->TIM3 配置项中，配置 ClockSource 为 InternalClock，如下图 6.3.1 所示：

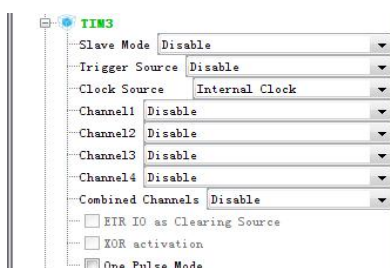


图 6.3.1TIM3 配置

② 进入 Configuration->TIM3 配置页，在弹出的界面中点击 Parameter Settings 选项卡，Counter Settings 配置栏下面的四个选项就是用来配置定时器的预分频系数，自动装载值，计数模式以及时钟分频因子。操作方法如下图 6.3.2 所示：

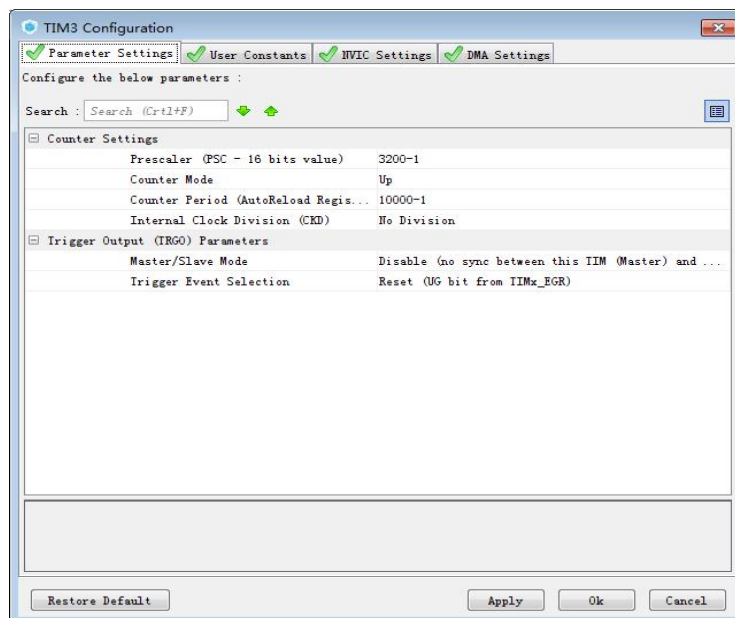


图 6.3.2TIM3 参数设置界面

注意：TIM3 的时钟位 32M，预分频系数为了方便计算，选择 3200。经过溢出时间的计算公式，可以得到重装载值位 10000。预分频系数和重装载值都减 1 是因为从 0 开始计数。

③ 进入 Configuration->NVIC 配置页，在弹出的界面中点击 NVIC 选项卡，配置 Interrupt Table 中的 TIM3globalinterrupt，使能中断，配置抢占优先级和响应优先级。

经过上面三个步骤，生成代码，我们打开工程可以看到 TIME3 初始化程序，该文件一共有 4 个函数：

```
void MX_TIM3_Init(void)
{
    TIM_ClockConfigTypeDef sClockSourceConfig;
    TIM_MasterConfigTypeDef sMasterConfig;

    htim3.Instance = TIM3;
    htim3.Init.Prescaler = 6400-1;
    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim3.Init.Period = 10000-1;
    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    if (HAL_TIM_Base_Init(&htim3) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
}
```

```
sClockSourceConfig.ClockSource = TIM_CLOCKSOURCE_INTERNAL;
if (HAL_TIM_ConfigClockSource(&htim3, &sClockSourceConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}
HAL_TIM_Base_Start_IT(&htim3);
}
```

第一个函数 TIM3_Init() 用来初始化定时器 3，使能定时器 3 更新中断以及使能定时器，实现的是 6.1 小节讲解的步骤 2 和步骤 3 配置功能。该函数的 2 个参数用来设置 TIM3 的溢出时间。因为我们在 Stm32_Clock_Init 函数里面已经初始化 APB1 的时钟为 1 分频，所以 APB1 的时钟为 32M，而从 STM32L151 的内部时钟树图（图 4.3.1.1）得知：当 APB1 的时钟分频数为 1 的时候，TIM2~7 的时钟为 APB1 的时钟，而如果 APB1 的时钟分频数不为 1，那么 TIM2~7 的时钟频率将为 APB1 时钟的两倍。因此，TIM3 的时钟为 32M，再根据我们设计的 arr 和 psc 的值，就可以计算中断时间了。计算公式如下：

$$T_{out} = ((arr + 1) * (psc + 1)) / T_{clk};$$

其中：

Tclk：TIM3 的输入时钟频率（单位为 Mhz）。

Tout：TIM3 溢出时间（单位为 us）

```
void HAL_TIM_Base_MspInit(TIM_HandleTypeDef* tim_baseHandle)
{
    if(tim_baseHandle->Instance==TIM3)
    {
        /* TIM3 clock enable */
        __HAL_RCC_TIM3_CLK_ENABLE();

        /* TIM3 interrupt Init */
        HAL_NVIC_SetPriority(TIM3_IRQn, 2, 3);
        HAL_NVIC_EnableIRQ(TIM3_IRQn);
    }
}
```

第二个函数 HAL_TIM_Base_MspInit 是定时器初始化回调函数，主要是使能定时器 3 时钟以及定时器 3 的 NVIC 配置，实现的是 6.1 小节讲解的步骤 1 和步骤 4 功能。


```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&htim3))
    {
        if(HAL_GPIO_ReadPin(LED1_GPIO_Port,LED1_Pin))
        {
            CL_LED1();
        }
        else
        {
            OP_LED1();
        }
    }
}
```

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(htim==(&htim3))
    {
        if(HAL_GPIO_ReadPin(LED1_GPIO_Port,LED1_Pin))
        {
            CL_LED1();
        }
        else
        {
            OP_LED1();
        }
    }
}
```

第三个函数 TIM3_IRQHandler 是中断服务入口函数，该函数内部只有一行代码就是调用定时器中断共用处理函数 HAL_TIM_IRQHandler。根据前面的讲解，函数 HAL_TIM_IRQHandler 内部会判断中断来源，根据中断来源调用不同的中断处理回调函数。这里我们开启的是定时器 3 的更新中断，所以我们需要重定义更新中断回调函数 HAL_TIM_PeriodElapsedCallback。第四个函数 HAL_TIM_PeriodElapsedCallback 就是更新中断回调函数，也就是真正的中断处理函数，该函数内部通过判断中断是定时器 3 之后，然后控制 LED1 翻转。timer.h 头文件内容比较简单，这里我们就不做讲解。

最后，我们看看主函数代码如下：

```
Int main(void)
{
    HAL_Init();

    SystemClock_Config

();

    MX_GPIO_Init
();MX_TIM3_I
nit();

    while(1)
    {
```

这里的代码和之前大同小异，此段代码对 TIM3 进行初始化之后，进入死循环等待 TIM3 溢出中断，当 TIM3_CNT 的值等于 TIM3_ARR 的值的时候，就会产生 TIM3 的更新中断，然后在中断里面取反 LED1，TIM3_CNT 再从 0 开始计数。

这里定时器定时时长 1s 是这样计算出来的，定时器的时钟为 32Mhz，分频系数为 3199，所以分频后的计数频率为 $32\text{Mhz} / (3199 + 1) = 10\text{KHz}$ ，然后计数到 9999，所以时长为 $(9999 + 1) / 10000 = 1\text{s}$ ，所以溢出时间为 1s。

6.4 下载验证

在完成软件设计之后，我们将编译好的文件下载到 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误，我们将看 LED1 不停闪烁（每 1s 闪烁一次）。

第七章 PWM 输出实验

上一章，我们介绍了 STM32L151 的通用定时器 TIM3，用该定时器的中断来控制 LED2 的闪烁，这一章，我们将向大家介绍如何使用 STM32L151 的 TIM3 来产生 PWM 输出。在本章中，我们将使用 TIM3 的通道 4 来产生 PWM 来控制 LED2 的亮度。本章分为如下几个部分：

- 7.1 PWM 简介
- 7.2 硬件设计
- 7.3 STM32CubeMX 配置定时器 PWM 输出功能和软件设计
- 7.4 下载验证

7.1 PWM 简介

脉冲宽度调制(PWM)，是英文“Pulse WidthModulation”的缩写，简称脉宽调制，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。简单一点，就是对脉冲宽度的控制，PWM 原理如图 7.1.1 所示：

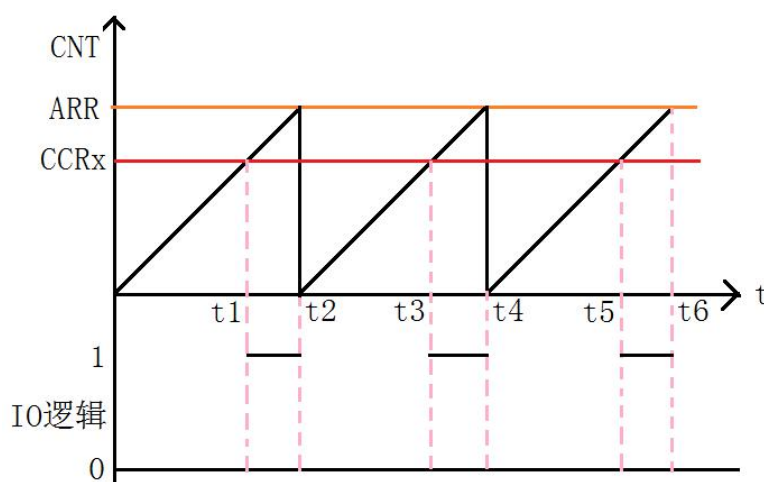


图 7.1.1 PWM 原理示意图

图 7.1.1 就是一个简单的 PWM 原理示意图。我们看一下 PWM 的一个周期：

- (1) 定时器从 0 开始向上计数
- (2) 当 0~t1 段，定时器计数器 TIMx_CNT 值小于 CCRx 值，输出低电平
- (3) t1~t2 段，定时器计数器 TIMx_CNT 值大于 CCRx 值，输出高电平
- (4) 当 TIMx_CNT 值达到 ARR 时，定时器溢出，重新向上计数... 循环此过程至此一个 PWM 周期完成。

要使 STM32L151 的通用定时器 TIMx 产生 PWM 输出，除了上一章介绍的寄存器外，我们还会用到 3 个寄存器，来控制 PWM 的。这三个寄存器分别是：捕获/比较模式寄存器

(TIMx_CCMR1/2)、捕获/比较使能寄存器 (TIMx_CCER)、捕获/比较寄存器 (TIMx_CCR1~4)。接下来我们简单介绍一下这三个寄存器。

首先是捕获/比较模式寄存器（TIMx_CCMR1/2），该寄存器一般有 2 个：TIMx_CCMR1 和 TIMx_CCMR2。TIMx_CCMR1 控制 CH1 和 2，而 TIMx_CCMR2 控制 CH3 和 4。以下我们将以 TIM3 为例进行介绍。TIM3_CCMR2 寄存器各位描述如图 7.1.2 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC4CE	OC4M[2:0]			OC4PE	OC4FE	CC4S[1:0]		OC3CE	OC3M[2:0]			OC3PE	OC3FE	CC3S[1:0]	
IC4F[3:0]				IC4PSC[1:0]				IC3F[3:0]			IC3PSC[1:0]				
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

图 7.1.2TIM3_CCMR2 寄存器各位描述

该寄存器的有些位在不同模式下，功能不一样，所以在图 7.1.2 中，我们把寄存器分了 2 层，上面一层对应输出而下面的则对应输入。这里我们需要说明的是模式设置位 OC4M，此部分由 3 位组成。总共可以配置成 7 种模式，我们使用的是 PWM 模式，所以这 3 位必须设置为 110/111。这两种 PWM 模式的差别就是输出电平的极性相反。另外 CC4S 用于设置通道的方向（输入/输出）默认设置为 0，就是设置通道作为输出使用。

接下来，我们介绍 TIM3 的捕获/比较使能寄存器（TIM3_CCER），该寄存器控制着各个输入输出通道的开关。该寄存器的各位描述如图 7.1.3 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CC4NP	Res.	CC4P	CC4E	CC3NP	Res.	CC3P	CC3E	CC2NP	Res.	CC2P	CC2E	CC1NP	Res.	CC1P	CC1E
r/w		r/w	r/w	r/w		r/w	r/w	r/w		r/w	r/w	r/w		r/w	r/w

图 7.1.3TIM3_CCER 寄存器各位描述

该寄存器比较简单，我们这里只用到了 CC4E 位，该位是输入/捕获 4 输出使能位，要想 PWM 从 IO 口输出，这个位必须设置为 1，所以我们需要设置该位为 1。

最后，我们介绍一下捕获/比较寄存器（TIMx_CCR1~4），该寄存器总共有 4 个，对应 4 个通道 CH1~4。我们使用的是通道 4，TIM3_CCR4 寄存器的各位描述如图 6.1.4 所示

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CCR4[31:16] (depending on timers)															
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR4[15:0]															
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

位 31:16 **CCR4[31:16]**: 捕获/比较 4 的高 16 位（对于 TIM2 和 TIM5）。

位 15:0 **CCR4[15:0]**: 捕获/比较 4 的低 16 位 (Low Capture/Compare value)

- 如果 CC4 通道配置为输出（CC4S 位）：
CCR4 为要装载到实际捕获/比较 4 寄存器的值（预装载值）。
如果没有通过 TIMx_CCMR 寄存器中的 OC4PE 位来使能预装载功能，写入的数值会被直接传输至当前寄存器中。否则只有发生更新事件时，预装载值才会复制到活动捕获/比较 4 寄存器中。
实际捕获/比较寄存器中包含要与计数器 TIMx_CNT 进行比较并在 OC4 输出上发出信号的值。
- 如果 CC4 通道配置为输入（TIMx_CCMR4 寄存器中的 CC4S 位）：
CCR4 为上一步输入捕获 4 事件 (IC4) 发生时的计数器值。

图 6.1.4 寄存器 TIM3_CCR4 各位描述

在输出模式下，该寄存器的值与 CNT 的值比较，根据比较结果产生相应动作。利用这点，我们通过修改这个寄存器的值，就可以控制 PWM 的输出脉宽了。如果是通用定时器，则配置以上三个寄存器就够了，但是如果是高级定时器，则还需要配置：刹车和死区寄存器（TIMx_BDTR）该寄存器各位描述如图 6.1.5 所示：

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MOE	AOE	BKP	BKE	OSSR	OSSI	LOCK[1:0]		DTG[7:0]							
r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w	r/w

位 15 **MOE**: 主输出使能 (Main output enable)

只要断路输入变为有效状态, 此位便由硬件异步清零。此位由软件置 1, 也可根据 AOE 位状态自动置 1。此位仅对配置为输出的通道有效。

0: OC 和 OCN 输出禁止或被强制为空闲状态。

1: 如果 OC 和 OCN 输出的相应使能位 (TIMx_CCER 寄存器中的 CCxE 和 CCxNE 位) 均置 1, 则使能 OC 和 OCN 输出。

图 6.1.5 寄存器 TIMx_BDTR 各位描述

该寄存器, 我们只需要关注最高位: MOE 位, 要想高级定时器的 PWM 正常输出, 则必须设置 MOE 位为 1, 否则不会有输出。注意: 通用定时器不需要配置这个。

本章, 我们使用的是 TIM3 的通道 4, 所以我们需要修改 TIM3_CCR4 以实现脉宽控制 LED2 的亮度。至此, 我们把本章要用的几个相关寄存器都介绍完了, 下面我们介绍通过 HAL 库来配置该功能的步骤。

首先要提到的是, PWM 实际跟上一章节一样使用的是定时器的功能, 所以相关的函数设置同样在库函数文件 stm32l1xx_tim.h 和 stm32l1xx_tim.c 文件中。

1) 开启 TIM3 和 GPIO 时钟, 配置 PB1 选择复用功能 AF1 (TIM3) 输出。

要使用 TIM3, 我们必须先开启通道 4 的时钟, 这点相信大家看了这么多代码, 应该明白了。这里我们还要配置 PB1 为复用 (AF1) 输出, 才可以实现 TIM13_CH4 的 PWM 经过 PB1 输出。HAL 库使能 TIM3 时钟和 GPIO 时钟方法是:

```
__HAL_RCC_TIM3_CLK_ENABLE(); //使能定时器 3
__HAL_RCC_GPIOB_CLK_ENABLE(); //开启 GPIOB 时钟
```

接下来便是要配置 PB1 复用映射为 TIM3 的 PWM 输出引脚。关于 IO 口复用映射, 在串口通信实验中有详细讲解, 主要是通过函数 HAL_GPIO_Init 来实现的:

```
GPIO_InitStruct.Pin = GPIO_PIN_1; //PB1
GPIO_InitStruct.Mode = GPIO_MODE_AF_PP; //复用推挽输出
GPIO_InitStruct.Pull = GPIO_NOPULL; //上拉-
GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW; //高速
GPIO_InitStruct.Alternate = GPIO_AF2_TIM3; //PB1 复用为 TIM3_CH4
HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
```

在 IO 口初始化配置中, 我们只需要将成员变量 Mode 配置为复用推挽输出, 同时成员变量 Alternate 配置为 GPIO_AF2_TIM3, 即可实现 PB1 映射为定时器 3 通道 4 的 PWM 输出引脚。这里还需要说明一下, 对于定时器通道的引脚关系, 大家可以查看 STM32L1 对应的数据手册, 比如我们 PWM 实验, 我们使用的是定时器 3 的通道 4, 对应的引脚 PB1 可以从数据手册表中查看:

PB1	I/O	FT	(4)	TIM3_CH4 / TIM8_CH3N/ OTG_HS_ULPI_D2/ ETH_MII_RXD3 / OTG_HS_INTN / TIM1_CH3N/ EVENTOUT	ADC12_IN9
-----	-----	----	-----	--	-----------

2) 初始化 TIM3, 设置 TIM3 的 ARR 和 PSC 等参数

根据前面的讲解, 初始化定时器的 ARR 和 PSC 等参数是通过函数 HAL_TIM_Base_Init 来实现的, 但是这里大家要注意, 对于我们使用定时器的 PWM 输出功能时, HAL 库为我们提供了一个独立的定时器初始化函数 HAL_TIM_PWM_Init, 该函数声明为:

```
HAL_StatusTypeDef HAL_TIM_PWM_Init(TIM_HandleTypeDef*htim);
```

该函数实现的功能以及使用方法和 HAL_TIM_Base_Init 都是类似的, 作用都是初始化定时器的 ARR 和 PSC 等参数。为什么 HAL 库要提供这个函数而不直接让我们使用 HAL_TIM_Base_Init 函数呢?

这是因为 HAL 库为定时器的 PWM 输出定义了单独的 MSP 回调函数 HAL_TIM_PWM_MspInit, 也就是说, 当我们调用 HAL_TIM_PWM_Init 进行 PWM 初始化之后, 该函数内部会调用 MSP 回调函数 HAL_TIM_PWM_MspInit。而当我们使用 HAL_TIM_Base_Init 初始化定时器参数的时候, 它内部调用的回调函数为 HAL_TIM_Base_MspInit, 这里大家注意区分。所以大家一定要注意, 使用 HAL_TIM_PWM_Init 初始化定时器时, 回调函数为 HAL_TIM_PWM_MspInit, 该函数声明为:

```
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef*htim);
```

一般情况下, 上面步骤 1 的时钟使能和 IO 口初始化映射都编写在回调函数内部。

3) 设置 TIM3_CH4 的 PWM 模式, 输出比较极性, 比较值等参数。

接下来, 我们要设置 TIM3_CH4 为 PWM 模式 (默认是冻结的), 因为我们的 DS0 是低电平亮, 而我们希望当 CCR4 的值小的时候, DS0 就暗, CCR4 值大的时候, LED2 就亮, 所以我们要通过配置 TIM3_CCMR2 的相关位来控制 TIM3_CH4 的模式。在 HAL 库中, PWM 通道设置是通过函数 HAL_TIM_PWM_ConfigChannel 来设置的:

```
HAL_StatusTypeDef HAL_TIM_PWM_ConfigChannel(TIM_HandleTypeDef*htim,  
                                              TIM_OC_InitTypeDef* sConfig,  
                                              uint32_tChannel);
```

第一个参数 htim 是定时器初始化句柄, 也就是 TIM_HandleTypeDef 结构体指针类型, 这和 HAL_TIM_PWM_Init 函数调用时候参数保存一致即可。

第二个参数 sConfig 是 TIM_OC_InitTypeDef 结构体指针类型, 这也是该函数最重要的参数。该参数用来设置 PWM 输出模式, 极性, 比较值等重要参数。首先我们来看看结构体定义:

```
typedef struct  
{  
    uint32_t OCMODE;  
    uint32_t Pulse;  
    uint32_t OCPolarity;  
    uint32_t OCNPolarity;  
    uint32_t OCFastMode;  
    uint32_t OCIdleState;  
    uint32_t OCNIdleState;  
} TIM_OC_InitTypeDef;
```

该结构体成员我们重点关注前三个。成员变量 OCMODE 用来设置模式，也就是我们前面讲解的 7 种模式，这里我们设置为 PWM 模式 1。成员变量 Pulse 用来设置捕获比较值。成员变量 TIM_OCpolarity 用来设置输出极性是高还是低。其他的参数 TIM_OutputNState, TIM_OCNPolarity, TIM_OCIdleState 和 TIM_OCNIIdleState 是高级定时器才用到的。

第三个参数 Channel 用来选择定时器的通道，取值范围为 TIM_CHANNEL_1~TIM_CHANNEL_4。这里我们使用的是定时器 3 的通道 4，所以取值为TIM_CHANNEL_4 即可。

例如我们要初始化定时器 3 的通道 4 为 PWM 模式, 输出极性为低, 那么实例代码为:

```
TIM_OC_InitTypeDef sConfigOC;
sConfigOC.OCMode = TIM_OCMODE_PWM1;
sConfigOC.Pulse = 0;
sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_4) !=
HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}

HAL_TIM_MspPostInit(&htim3);
HAL_TIM_PWM_Start(&htim3,TIM_CHANNEL_4);
```

4) 使能 TIM3, 使能 TIM3 的 CH4 输出。

在完成以上设置了之后, 我们需要使能 TIM3 并且使能 TIM3_CH4 输出。在 HAL 库中, 函数 HAL_TIM_PWM_Start 可以用来实现这两个功能, 函数声明如下:

```
HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_tChannel);
```

该函数第二个入口参数 Channel 是用来设置要使能输出的通道号。对于单独使能定时器的方法, 在上一章定时器实验我们已经讲解。实际上, HAL 库也同样提供了单独使能定时器的输出通道函数, 函数为:

```
void TIM_CCxChannelCmd(TIM_TypeDef* TIMx, uint32_t Channel,
uint32_tChannelState);
```

5) 修改 TIM3_CCR4 来控制占空比。

最后, 在经过以上设置之后, PWM 其实已经开始输出了, 只是其占空比和频率都是固定的, 而我们通过修改比较值 TIM3_CCR4 则可以控制 CH4 的输出占空比。继而控制 LED2 的亮度。HAL 库中并没有提供独立的修改占空比函数, 这里我们可以编写这样一个函数如下:

```
//设置 TIM3_CCR4 来控制占空比
//compare:比较值
void TIM_SetTIM3Compare4(unsigned int compare)
{
    TIM3->CCR4=compare;
}
```


实际上，因为调用函数 HAL_TIM_PWM_ConfigChannel 进行 PWM 配置的时候可以设置比较值，所以我们也可以直接使用该函数来达到修改占空比的目的：

```
void TIM_SetCompare4(TIM_TypeDef *TIMx,u32compare)
{
    TIM3_CH4Handler.Pulse=compare;
    HAL_TIM_PWM_ConfigChannel(&TIM3_Handler,&TIM3_CH4Handler,TIM_CHANNEL_4);
}
```

这种方法因为要调用 HAL_TIM_PWM_ConfigChannel 函数对各种初始化参数进行重新设置，所以大家在使用中一定要注意，例如在实时系统中如果多个线程同时修改初始化结构体相关参数，可能导致结果混乱。

7.2 硬件设计

本实验用到的硬件资源有：

- 1) 指示灯 LED2
- 2) 定时器 TIM3

这两个我们前面都已经介绍了，因为 TIM3_CH4 可以通过 PB1 输出 PWM，而 LED2 就是直接接在 PB1 上面的，所以电路上并没有任何变化。

7.3 STM32CubeMX 配置定时器 PWM 输出功能和软件设计

使用 STM32CubeMX 配置 PWM 输出的配置步骤和配置定时器中断的配置步骤非常接近，步骤如下：

- ① 在 Pinout->TIM3 配置项中，配置 Channel4 的值为 PWMgenerationCH4，然后 Clock Source 为 InternalClock。操作过程如下图 7.3.1 所示：

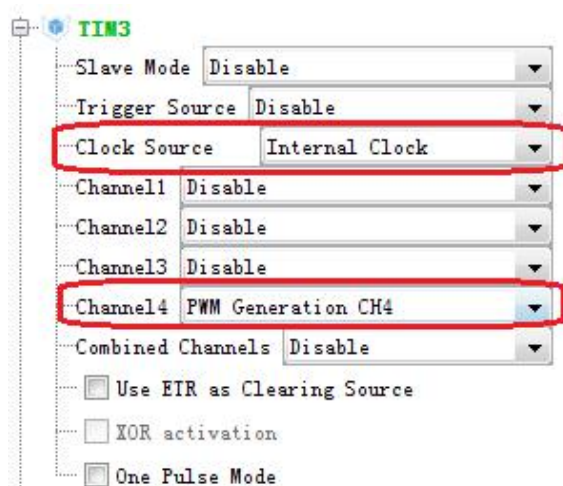
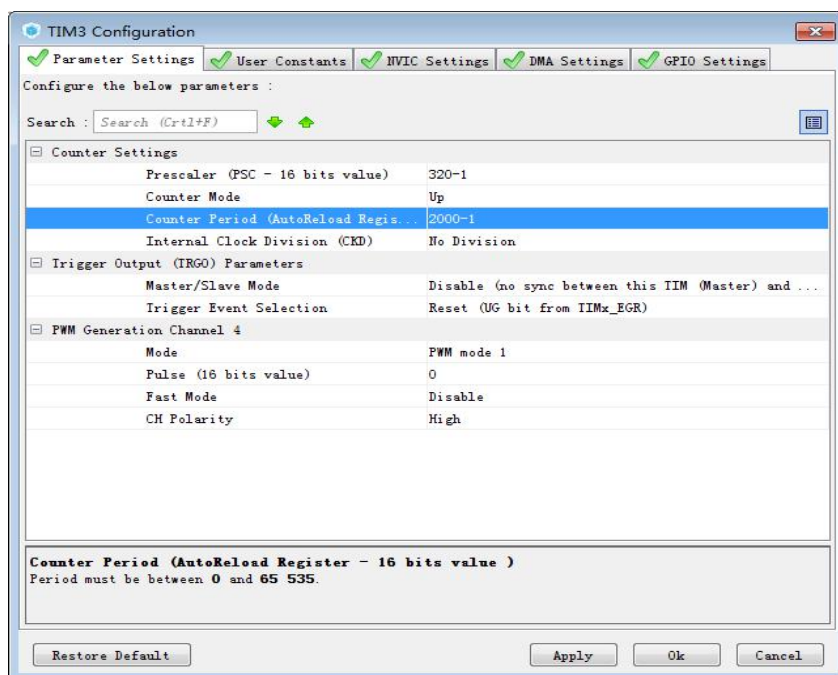


图 7.3.1TIM3 配置

- ② 进入 Configuration->TIM3 配置页，在弹出的界面中点击 Parameter Settings 选项卡，Counter Settings 配置栏下面的四个选项就是用来配置定时器的预分频系数，自动装载值，计数模式以及时钟分频因子。在界面的PWMSGenerationChannel4 配置栏配置 PWM 模式，比较值，极性参数，操作方法如下图 7.3.2 所示：



本章 PWM 输出实验，我们并没有使用到中断，所以我们不需要使能中断和配置 NVIC。经过上面的配置就可以生成工程源码，我们打开工程可以看到 tim.c 文件如下，此部分代码包含三个函数，完全实现了前面 7.1 小节讲解的 5 个配置步骤：

```
void MX_TIM3_Init(void)
{
    TIM_MasterConfigTypeDef sMasterConfig;
    TIM_OC_InitTypeDef sConfigOC;

    htim3.Instance = TIM3;
    htim3.Init.Prescaler = 320-1;
    htim3.Init.CounterMode = TIM_COUNTERMODE_UP;
    htim3.Init.Period = 2000-1;
    htim3.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
    if (HAL_TIM_PWM_Init(&htim3) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    sMasterConfig.MasterOutputTrigger = TIM_TRGO_RESET;
    sMasterConfig.MasterSlaveMode = TIM_MASTERSLAVEMODE_DISABLE;
    if (HAL_TIMEx_MasterConfigSynchronization(&htim3, &sMasterConfig) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    sConfigOC.OCMode = TIM_OCMODE_PWM1;
    sConfigOC.Pulse = 0;
    sConfigOC.OCpolarity = TIM_OCPOLARITY_HIGH;
    sConfigOC.OCFastMode = TIM_OCFAST_DISABLE;
    if (HAL_TIM_PWM_ConfigChannel(&htim3, &sConfigOC, TIM_CHANNEL_4) !=
    HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }

    HAL_TIM_MspPostInit(&htim3);
    HAL_TIM_PWM_Start(&htim3,TIM_CHANNEL_4);
}
```

第一个函数 TIM3_PWM_Init 实现的是 7.1 小节讲解的步骤 2-4，首先通过调用定时器 HAL 库函数 HAL_TIM_PWM_Init 初始化 TIM3 并设置 TIM3 的 ARR 和 PSC 等参数，其次通过调用函数 HAL_TIM_PWM_ConfigChannel 设置 TIM3_CH4 的 PWM 模式以及比较值等参数，最后通过调用函数 HAL_TIM_PWM_Start 来使能 TIM3 以及使能 PWM 通道 TIM3_CH4 输出。

```
void HAL_TIM_PWM_MspInit(TIM_HandleTypeDef* tim_pwmHandle)
{

    if(tim_pwmHandle->Instance==TIM3)
    {
        /* USER CODE BEGIN TIM3_MspInit 0 */

        /* USER CODE END TIM3_MspInit 0 */
        /* TIM3 clock enable */
        __HAL_RCC_TIM3_CLK_ENABLE();
        /* USER CODE BEGIN TIM3_MspInit 1 */

        /* USER CODE END TIM3_MspInit 1 */
    }
}
```

第二个函数 HAL_TIM_PWM_MspInit 是 PWM 的 MSP 初始化回调函数，该函数实现的是 7.1 小节步骤 1，主要是使能相应时钟以及初始化定时器通道 TIM3_CH4 对应的 IO 口模式，同时设置复用映射关系。

```
void HAL_TIM_MspPostInit(TIM_HandleTypeDef* timHandle)
{

    GPIO_InitTypeDef GPIO_InitStruct;
    if(timHandle->Instance==TIM3)
    {
        /* USER CODE BEGIN TIM3_MspPostInit 0 */

        /* USER CODE END TIM3_MspPostInit 0 */

        /**TIM3 GPIO Configuration
        PB1      -----> TIM3_CH4
        */
        GPIO_InitStruct.Pin = GPIO_PIN_1; //PB1
        GPIO_InitStruct.Mode = GPIO_MODE_AF_PP; //, 'ÓÄÍÆíÊä³ ö
        GPIO_InitStruct.Pull = GPIO_NOPULL; //ÉĬÀ-
        GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW; //μÍËÙ
        GPIO_InitStruct.Alternate = GPIO_AF2_TIM3; //PB1, 'ÓÄÎª TIM3_CH4
        HAL_GPIO_Init(GPIOB, &GPIO_InitStruct);
    }
}
```

```
}
```

第三个函数 HAL_TIM_PWM_MspInit 主要是初始化定时器通道 TIM3_CH4 对应的 IO 口模式，同时设置复用映射关系。

```
void TIM_SetTIM3Compare4(unsigned int compare)
{
    TIM3->CCR4=compare;
}
```

第四个函数 TIM_SetTIM3Compare4 是用户自定义的设置比较值函数，这在我们 7.1 小节步骤 5 有详细讲解。

接下来，我们看看 main 函数内容如下：

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM3_Init();
    unsigned char flag=1;
    unsigned int step_num=0;
    while (1)
    {
        if(flag)
        {
            step_num++;
            if(step_num>300)
                flag=0;
        }
        else
        {
            step_num--;
            if(step_num==0)
            {
                flag=1;
            }
        }
        TIM_SetTIM3Compare4(step_num);
        HAL_Delay(20);
    }
}
```

这里，我们从死循环函数可以看出，我们控制 `step_num` 的值从 0 变到 300，然后又从 300 变到 0，如此循环，因此 LED2 的亮度也会跟着从暗变到亮，然后又从亮变到暗。至于这里值，我们为什么取 300，是因为 PWM 的输出占空比（占空比指在一个脉冲循环内，通电时间相对于总时间所占的比例，呼吸灯的效果与占空比）达到这个值的时候，我们的 LED 亮度变化就不大了（虽然最大值可以设置到 499），因此设计过大的值在这里是没必要的。至此，我们的软件设计就完成了。

7.4 下载验证

在完成软件设计之后，将我们将编译好的文件下载到 STM32 开发板上，观看其运行结果是否与我们编写的一致。如果没有错误我们将看 LED2 不停的由暗变到亮然后又从亮变暗。

第八章 TFT_LCD 实验

前面几章的实例，均没涉及到液晶显示，本章我们将介绍 TFT_LCD 模块，该模块可以显示 16 位色的真彩图片。在本章中，我们将使用 STM32L151 开发板底板上的 TFT_LCD 接口，来点亮 TFT_LCD，并实现 ASCII 字符和彩色的显示等功能。本章分为如下几个部分：

- 8.1 TFTLCD&SPI 简介
- 8.2 硬件设计
- 8.3 STM32CubeMX 配置 SPI 和软件计
- 8.4 下载验证

8.1 TFTLCD&SPI 简介

本章我们将通过 STM32L151 的 SPI 接口来控制 TFT_LCD 的显示，所以本节分为两个部分，分别介绍 TFT_LCD 和 SPI。

TFT_LCD 简介

TFT_LCD 即薄膜晶体管液晶显示器。其英文全称为:Thin Film Transistor-Liquid Crystal Display。TFT_LCD 与无源 TN-LCD、STN-LCD 的简单矩阵不同，它在液晶显示屏的每一个像素上都设置有一个薄膜晶体管（TFT），可有效地克服非选通时的串扰，使显示液晶屏的静态特性与扫描线数无关，因此大大提高了图像质量。TFT_LCD 也被叫做真彩液晶显示器。

该模块有如下特点：

- 1，128*128 的分辨率。
- 2，16 位真彩显示。该模块的外观图如图 8.1.1.1 所示：

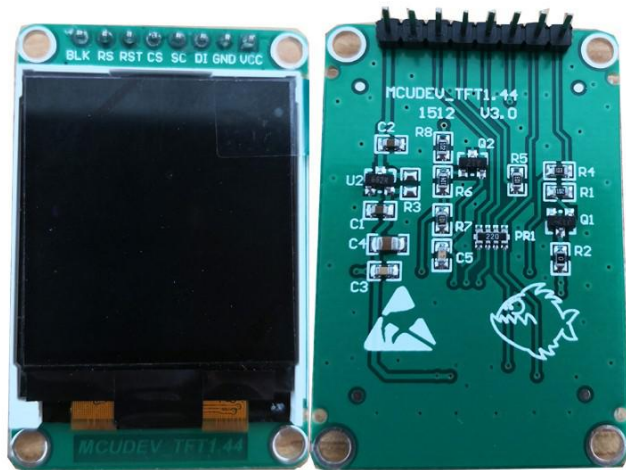


图 8.1.1.1. 1.44 寸 TFTLCD 外观图

模块原理图如图 8.1.1.2 所示：

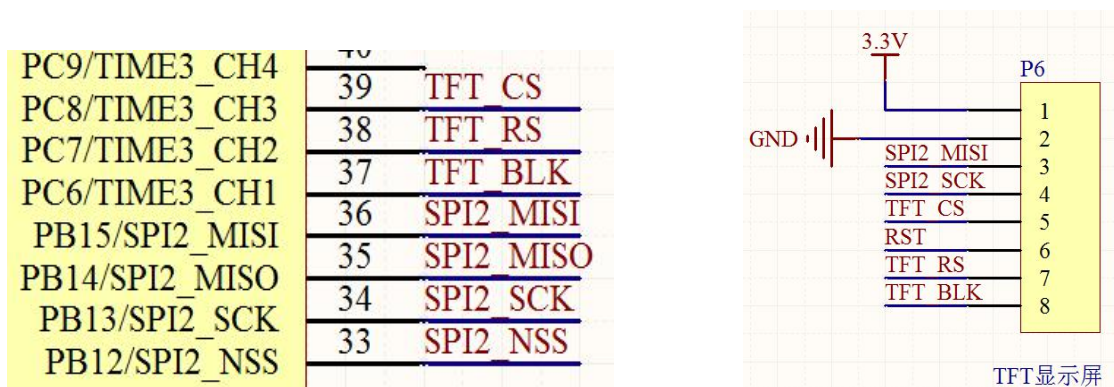


图 8.1.1.2 1.44 寸 TFT_LCD 模块原理图

从图 8.1.1.2 可以看出，TFT_LCD 模块采用 SPI 方式与外部连接。该模块的接口还有如下一些信号线：

TFT_CS：TFTLCD 片选信号。

SPI2_MISI：SPI 数据线。

SPI2_SCK：SPI 时钟线。

RST：硬复位 TFTLCD。

TFT_RS：命令/数据标志（0，读写命令；1，读写数据）。

需要说明的是，TFTLCD 模块的 RST 信号线是直接接到 STM32L151 的复位脚上，并不由软件控制，这样可以省下来一个 I/O 口。另外我们还需要一个背光控制线来控制 TFT_LCD 的背光。所以，我们总共需要的 I/O 口数目为 5 个。其中硬复位和初始化序列，只需要执行一次即可。而画点流程就是：设置坐标写 GRAM 指令写入颜色数据，然后在 LCD 上面，我们就可以看到对应的点显示我们写入的颜色了。读点流程为：设置坐标读 GRAM 指令读取颜色数据，这样就可以获取到对应点的颜色数据了。

以上只是最简单的操作，也是最常用的操作，有了这些操作，一般就可以正常使用 TFTLCD 了。接下来我们将该模块用来显示字符和数字，通过以上介绍，我们可以得出 TFTLCD 显示需要的相关设置步骤如下：

1) 设置 STM32L151 与 TFTLCD 模块相连接的 I/O。

这一步，先将我们与 TFTLCD 模块相连的 I/O 口进行初始化，以便驱动 LCD。这里我们用到的是 SPI，SPI 将在 8.1.2 节向大家详细介绍。

2) 初始化 TFTLCD 模块。

这里我们没有硬复位 LCD，因为 STM32L151 开发板的 LCD 接口，将 TFTLCD 的 RST 同 STM32L151 的 RESET 连接在一起了，只要按下开发板的 RESET 键，就会对 LCD 进行硬复位。初始化序列，就是向 LCD 控制器写入一系列的设置值，这些初始化序列一般 LCD 供应商会提供给客户，我们直接使用这些序列即可，不需要深入研究。在初始化之后，LCD 才可以正常使用。

3) 通过函数将字符和数字显示到 TFTLCD 模块上。

设置坐标写 GRAM 指令写 GRAM 来实现，但是这个步骤，只是一个点的处理，我们要显示字符/数字，就必须多次使用这个步骤，从而达到显示字符/数字的目的，所以需要设计一个函数来实现数字/字符的显示，之后调用该函数，就可以实现数字/字符的显示了。

SPI 简介

SPI 是英语 Serial Peripheral interface 的缩写，顾名思义就是串行外围设备接口。是 Motorola 首先在其 MC68HCXX 系列处理器上定义的。SPI 接口主要应用在 EEPROM，FLASH，实时时钟，AD 转换器，还有数字信号处理器和数字信号解码器之间。SPI，是一种高速的，全双工，同步的通信总线，并且在芯片的管脚上只占用四根线，节约了芯片的管脚，同时为 PCB 的布局上节省空间，提供方便，正是出于这种简单易用的特性，现在越来越多的芯片集成了这种通信协议，STM32L1 也有 SPI 接口。下面我们看看 SPI 的内部简明图（图 8.1.2.1）

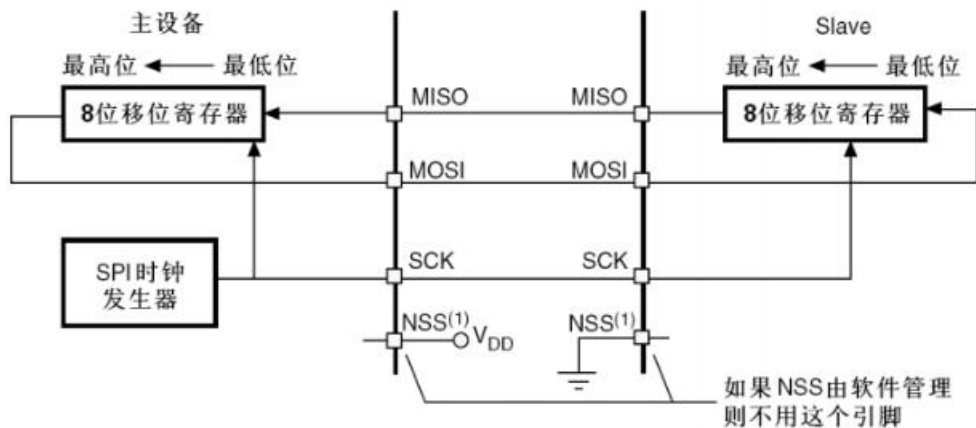


图 8.1.2.1 SPI 的内部简明图

SPI 接口一般使用 4 条线通信：

MISO 主设备数据输入，从设备数据输出。

MOSI 主设备数据输出，从设备数据输入。

SCLK 时钟信号，由主设备产生。

CS 从设备片选信号，由主设备控制。

从图中可以看出，主机和从机都有一个串行移位寄存器，主机通过向它的 SPI 串行寄存器，写入一个字节来发起一次传输。寄存器通过 MOSI 信号线将字节传送给从机，从机也将自己的移位寄存器中的内容通过 MISO 信号线返回给主机。这样，两个移位寄存器中的内容就被交换。外设的写操作和读操作是同步完成的。如果只进行写操作，主机只需忽略接收到的字节；反之，若主机要读取从机的一个字节，就必须发送一个空字节来引发从机的传输。

SPI 主要特点有：可以同时发出和接收串行数据；可以当作主机或从机工作；提供频率可编程时钟；发送结束中断标志；写冲突保护；总线竞争保护等。

SPI 总线四种工作方式 SPI 模块为了和外设进行数据交换，根据外设工作要求，其输出串行同步时钟极性和相位可以进行配置，时钟极性（CPOL）对传输协议没有重大的影响。如果 CPOL=0，串行同步时钟的空闲状态为低电平；如果 CPOL=1，串行同步时钟的空闲状态为高电平。时钟相位（CPHA）能够配置用于选择两种不同的传输协议之一进行数据传输。如果 CPHA=0，在串行同步时钟的第一个跳变沿（上升或下降）数据被采样；如果 CPHA=1，在串行同步时钟的第二个跳变沿（上升或下降）数据被采样。SPI 主模块和与之通信的外设备 时钟相位和极性应该一致。

不同时钟相位下的总线数据传输时序如图 8.1.2.2 所示：

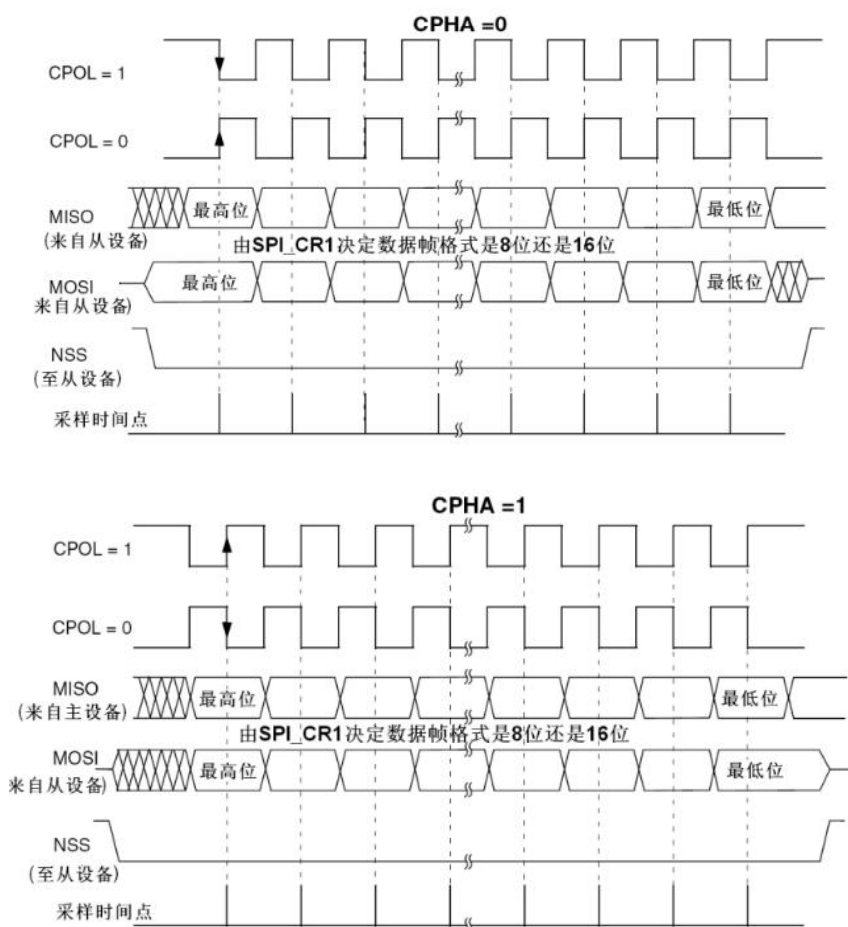


图 8.1.2.2 不同时钟相位下的总线传输时序 (CPHA=0/1)

STM32L151 的 SPI 功能很强大, SPI 时钟最高可以到 45Mhz, 支持 DMA, 可以配置为 SPI 协议或者 I2S 协议 (支持全双工 I2S)。

这节, 我们使用 STM32L151 的 SPI2 的主模式, 下面就来看看 SPI2 部分的设置步骤吧。SPI 相关的库函数和定义分布在文件 stm32l1xx_hal_spi.c 以及头文件 stm32l1xx_hal_spi.h 中。STM32 的主模式配置步骤如下:

1) 配置相关引脚的复用功能, 使能 SPI2 时钟。

我们要用 SPI2, 第一步就要使能 SPI2 时钟和响应引脚时钟。其次要设置 SPI2 的相关引脚为复用(AF5)输出, 这样才会连接到 SPI2 上。这里我们使用的是 PB13/PB14/PB15 这 3 个 (SCK、MISO、MOSI, CS 使用软件管理方式), 所以设置这三个为复用 IO, 复用功能为 AF5。使能 SPI2 时钟的方法为: HAL_RCC_SPI2_CLK_ENABLE(); //使能 SPI2 时钟复用 PB13/PB14/PB15 为 SPI2 引脚是通过 HAL_GPIO_Init 函数实现, 代码如下:

```
GPIO_InitStruct.Pin=GPIO_PIN_13|GPIO_PIN_15;GPIO_InitStruct.Mode=GPIO_MODE_AF_PP;
GPIO_InitStruct.Pull=GPIO_NOPULL;
GPIO_InitStruct.Speed=GPIO_SPEED_FREQ_VERY_HIGH;
GPIO_InitStruct.Alternate=GPIO_AF5_SPI2;
HAL_GPIO_Init(GPIOB,&GPIO_InitStruct);
```

2) 初始化 SPI2, 设置 SPI2 工作模式等。

这一步全部是通过 SPI2_CR1 来设置，我们设置 SPI2 为主机模式，设置数据格式为 8 位，然后通过 CPOL 和 CPHA 位来设置 SCK 时钟极性 & 采样方式。并设置 SPI2 的时钟频率（最大 45Mhz），以及数据的格式（MSB 在前还是 LSB 在前）。在 HAL 库中初始化 SPI 的函数为：

```
HAL_StatusTypeDef HAL_SPI_Init(SPI_HandleTypeDef* hspi);
```

下面我们来看看 SPI_HandleTypeDef 定义：

```
typedef struct __SPI_HandleTypeDef
{
    SPI_TypeDef                *Instance;    /*基地址 */
    SPI_InitTypeDef            Init;         /*初始化结构体 */
    uint8_t                    *pTxBuffPtr; /*发送缓存*/
    uint16_t                    TxXferSize; /* 发送数据大小 */
    __IO uint16_t               TxXferCount; /* 还剩下多少个数据要发送 */
    uint8_t                    *pRxBuffPtr; /* 接收缓存*/
    uint16_t                    RxXferSize; /*接收数据大小 */
    __IO uint16_t               RxXferCount; /* 还剩余多少个数据要接收 */
    DMA_HandleTypeDef          *hdmatx;     /*DMA 发送句柄 */
    DMA_HandleTypeDef          *hdmarx;     /* *DMA 接收句柄*/
    void                        (*RxISR)(struct __SPI_HandleTypeDef * hspi);
    void                        (*TxISR)(struct __SPI_HandleTypeDef * hspi);
    HAL_LockTypeDef             Lock;
    __IO HAL_SPI_StateTypeDef   State;
    __IO uint32_t               ErrorCode;
} SPI_HandleTypeDef;
```

该结构体和串口句柄结构体类似，同样有 6 个成员变量和 2 个 DMA_HandleTypeDef 指类型变量。这几个参数的作用这里我们就不做过多讲解，大家如果对 HAL 库串口通信理解了，那么这些就很好理解。这里我们主要讲解第二个成员变量 Init，它是 SPI_InitTypeDef 结构体类型，该结构体定义如下：

```
typedef struct
{
    uint32_t Mode; //模式：主（SPI_MODE_MASTER），从（SPI_MODE_SLAVE）
    uint32_t Direction; //方式：只接受模式，单线双向通信数据模式，全双工
    uint32_t DataSize; //8 位还是 16 位帧格式选择项
    uint32_t CLKPolarity; //时钟极性
    uint32_t CLKPhase; //时钟相位
    uint32_t NSS; //ss 信号由硬件（NSS 管脚）还是软件控制
    uint32_t BaudRatePrescaler; //设置 SPI 波特率预分频值
    uint32_t FirstBit; //起始位是 MSB 还是 LSB
    uint32_t TIMode; //帧格式 SPI motorola 模式还是 TI 模式
    uint32_t CRCCalculation; //硬件 CRC 是否使能
    uint32_t CRCPolynomial; //CRC 多项式
} SPI_InitTypeDef;
```

该结构体各个成员变量的含义我们已经在成员变量后面注释了，请大家参考学习。SPI 初始化实例代码如下：

```
void MX_SPI2_Init(void)
{
    hspi2.Instance = SPI2;
    hspi2.Init.Mode = SPI_MODE_MASTER;
    hspi2.Init.Direction = SPI_DIRECTION_2LINES;
    hspi2.Init.DataSize = SPI_DATASIZE_8BIT;
    hspi2.Init.CLKPolarity = SPI_POLARITY_HIGH;
    hspi2.Init.CLKPhase = SPI_PHASE_2EDGE;
    hspi2.Init.NSS = SPI_NSS_SOFT;
    hspi2.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
    hspi2.Init.FirstBit = SPI_FIRSTBIT_MSB;
    hspi2.Init.TIMode = SPI_TIMODE_DISABLE;
    hspi2.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    hspi2.Init.CRCPolynomial = 10;
    if (HAL_SPI_Init(&hspi2) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    __HAL_SPI_ENABLE(&hspi2);
}
```

```
void MX_SPI2_Init(void)
{

    hspi2.Instance = SPI2;
    hspi2.Init.Mode = SPI_MODE_MASTER;
    hspi2.Init.Direction = SPI_DIRECTION_2LINES;
    hspi2.Init.DataSize = SPI_DATASIZE_8BIT;
    hspi2.Init.CLKPolarity = SPI_POLARITY_HIGH;
    hspi2.Init.CLKPhase = SPI_PHASE_2EDGE;
    hspi2.Init.NSS = SPI_NSS_SOFT;
    hspi2.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
    hspi2.Init.FirstBit = SPI_FIRSTBIT_MSB;
    hspi2.Init.TIMode = SPI_TIMODE_DISABLE;
    hspi2.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    hspi2.Init.CRCPolynomial = 10;
    if (HAL_SPI_Init(&hspi2) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    __HAL_SPI_ENABLE(&hspi2);
}
```

同样，HAL 库也提供了 SPI 初始化 MSP 回调函数 HAL_SPI_MspInit，定义如下：

```
void
```

关于回调函数使用，这里我们就不做过多讲解。

3) 使能 SPI2。

这一步通过 SPI2_CR1 的 bit6 来设置，以启动 SPI2，在启动之后，我们就可以开始 SPI 通讯了。使能 SPI2 的方法为：

```
HAL_SPI_ENABLE(&hspi2); //使能 SPI2
```

4) SPI 传输数据

通信接口当然需要有发送数据和接受数据的函数，HAL 库提供的发送数据函数原型为：

```
HAL_StatusTypeDef HAL_SPI_Transmit(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                     uint16_t Size, uint32_t Timeout);
```

这个函数很好理解，往 SPIx 数据寄存器写入数据 Data，从而实现发送。

HAL 库提供的接受数据函数原型为：

```
HAL_StatusTypeDef HAL_SPI_Receive(SPI_HandleTypeDef *hspi, uint8_t *pData,
                                    uint16_t Size, uint32_t Timeout);
```

这个函数也不难理解，从 SPIx 数据寄存器读出接受到的数据。

前面我们讲解了 SPI 通信的原理，因为 SPI 是全双工，发送一个字节的的同时接受一个字节，发送和接收同时完成，所以 HAL 也提供了一个发送接收统一函数：

```
HAL_StatusTypeDef HAL_SPI_TransmitReceive(SPI_HandleTypeDef *hspi, uint8_t *pTxData, uint8_t *pRxData, uint16_t Size, uint32_t Timeout);
```

该函数发送一个字节的的同时负责接收一个字节。

5) 设置 SPI 传输速度

SPI 初始化结构体 SPI_InitTypeDef 有一个成员变量是 BaudRatePrescaler，该成员变量用来设置 SPI 的预分频系数，从而决定了 SPI 的传输速度。但是 HAL 库并没有提供单独的 SPI 分频系数修改函数，如果我们需要在程序中不时的修改速度，那么我们就要通过设置 SPI 的 CR1 寄存器来修改，具体实现方法请参考后面软件设计小节相关函数。

8.2 硬件设计

硬件设计我们前面已经讲过，这里不再讲解。

8.3 STM32CubeMX 配置 SPI 和软件设计

当大家了解了 SPI 的基本工作原理，那么使用 STM32CubeMX 配置 SPI 相关参数就会非常简单。如果大家对 SPI 没有理解，请仔细看教程学习。这里我们不再详细讲解每个配置项的含义。使用 STM32CubeMX 配置 SPI 的一般步骤为：

- ① 进入 Pinout->SPI2 配置栏，配置 SPI2 基本参数。

置参数如下图 8.3.1 所示：

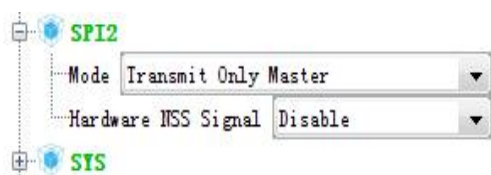


图 8.3.1 SPI 配置参数

因为这里我们只需要给 TFT_LCD 发送数据不需要接收数据，所以我们只需要选择 Transmit OnlyMaster 模式就可以，也可以选择全双工模式，NSS 片选引脚由硬件控制，不需要软件控制。其他引脚配置如下图 8.3.2 所示：

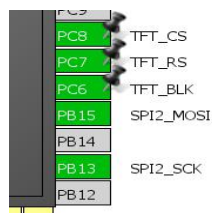


图 8.3.2 引脚配置

TFT_CS: TFTLCD 片选信号，输出模式。SPI2_MISI:

SPI 数据线。

SPI2_SCK: SPI 时钟线。

TFT_BLK: TFT_LCD 背光控制脚，输出模式。

TFT_RS: 命令/数据标志 (0, 读写命令; 1, 读写数据), 输出模式。

②点击 Configuration->SPI 进入 SPI 配置界面配置方法如下图 8.3.3 所示:

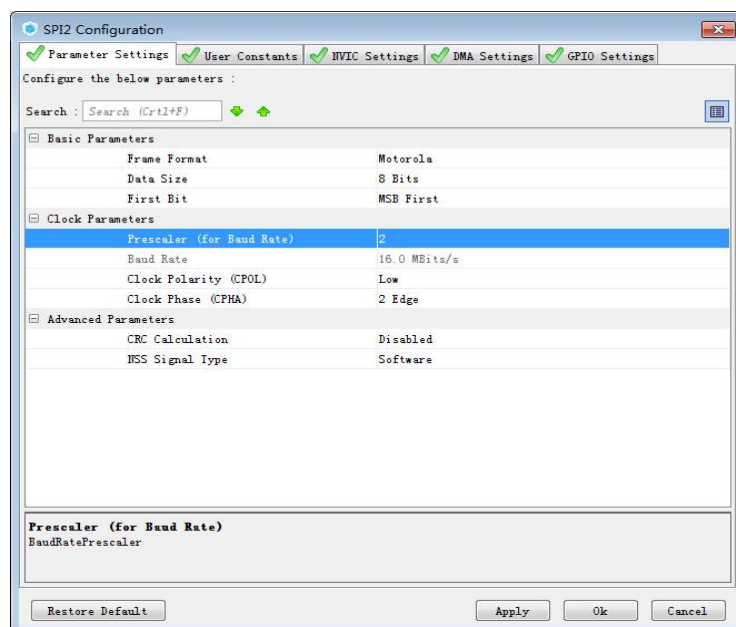


图 8.3.3 SPI 选项卡

点击 Configuration->GPIO 进入 GPIO 配置界面配置方法如下图 8.3.4 所示 (PC6、7、8 一样的设置):

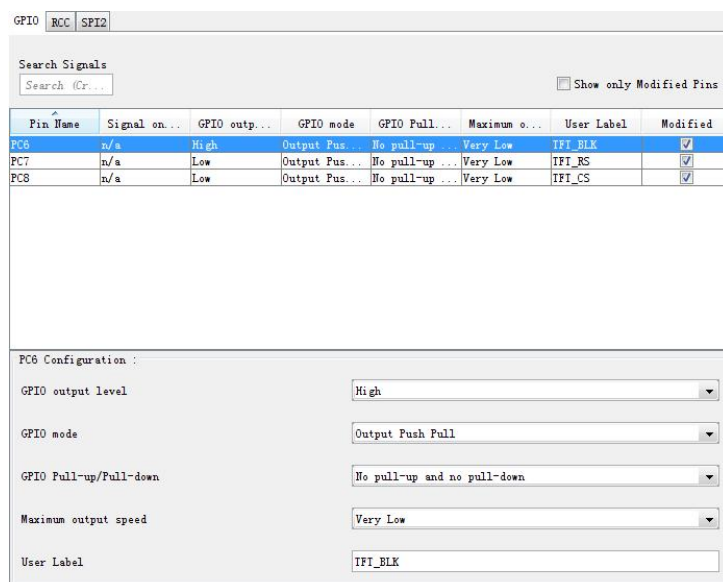


图 8.3.4 GPIO 选项卡

经过上面配置步骤, 我们就可以生成相应的初始化代码。打开工程我们可以添加 lcd.c 文件, 写入关键代码如下:


```
void LCD_WR_REG(unsigned short int data)
{
    TFT_CS_ENABLE();
    TFT_RS_DISABLE();
    SPI_WriteByte(data);
    TFT_CS_DISABLE();
}
void LCD_WR_DATA(unsigned char data)
{
    TFT_CS_ENABLE();
    TFT_RS_ENABLE();
    SPI_WriteByte(data);
    TFT_CS_DISABLE();
}
void LCD_WR_DATA_16Bit(unsigned short int data)
{
    TFT_CS_ENABLE();
    TFT_RS_ENABLE();
    SPI_WriteByte(data>>8);
    SPI_WriteByte(data);
    TFT_CS_DISABLE();
}
```

上面三个函数是写显示屏寄存器和发送 8/16bit 数据，其他函数我们不在这里一一讲解，有兴趣的同学可以自行了解。

```
//显示一个字符串，包括中英文显示
void Show_Str(unsigned short int x, unsigned short int y, unsigned short int fc,
unsigned short int bc, unsigned char *str,unsigned char size,unsigned char mode)
{
    unsigned short int x0=x;
    unsigned char bHz=0;    //字符或者中文
    while(*str!=0)    //数据未结束
    {
        if(!bHz)
        {
            if(x>(lcddev.width-size/2) || y>(lcddev.height-size))
                return;
            if(*str>0x80)bHz=1;    //中文
            else{    //字符
                if(*str==0x0D)//换行字符
                {
                    y+=size;
                    x=x0;
                    str++;
                } else{
                    if(size==12 || size==16)
                    {
                        LCD_ShowChar(x,y,fc,bc,*str,size,mode);
                        x+=size/2;    //字符为全字的一半
                    }
                    else//字符中没有集成 16X32 的英文字体,用 8X16 代替
                    {
                        LCD_ShowChar(x,y,fc,bc,*str,16,mode);
                        x+=8; //字符为全字的一半
                    }
                }
            }
            str++;
        }
        }else    //中文
        {
            if(x>(lcddev.width-size) || y>(lcddev.height-size))
                return;
            bHz=0;//有汉字库
            if(size==32)
                GUI_DrawFont32(x,y,fc,bc,str,mode);
            else if(size==24)
                GUI_DrawFont24(x,y,fc,bc,str,mode);
        }
    }
}
```

```
        else
            GUI_DrawFont16(x,y,fc,bc,str,mode);
            str+=2;
            x+=size;    //下一个字偏移
        }
    }
}
```

上面这个函数是显示字符串或者汉字的。

接下来我们看一下主函数里设计：

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI2_Init();
    LCD_Init();           //液晶屏初始化
    LCD_Clear(BLACK);     //清屏
    Show_Str(0,10,BLUE,BLACK," this is ademo ",16,1);
    while (1)
    {

    }
}
```

我们在主函数里可以看到，我们进行了 GPIO 和 SPI2 的初始化，接着就是 TFT_LCD 初始化，然后将 LCD 刷屏，显示一句 this is a demo。

8.4 下载验证

我们将编译好的文件下载到 STM32 开发板，我们可以看到和我们程序设计的是一样的，黑色的界面上用蓝色的字体显示了一句 this is a demo.

第九章 RTC 实时时钟实验

前面我们介绍了 TFT_LCD 模块，这一章我们将介绍 STM32L151 的内部实时时钟 (RTC)。在本章中，我们将使用 LCD 模块来显示日期和时间，实现一个简单的实时时钟，并可以设置闹铃。另外，本章将顺带向大家介绍 BKP 的使用。本章分为如下几个部分：

9.1 STM32L151 RTC 时钟简介

9.2 硬件设计

9.3 STM32CubeMX 配置和软件设计

9.4 下载验证

9.1 STM32L151 RTC 时钟简介

STM32L151 的 RTC，是一个独立的 BCD 定时器/计数器。RTC 提供一个日历时钟（包含年月日时分秒信息）、两个可编程闹钟（ALARM A 和 ALARM B）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。

两个 32 位寄存器 (TR 和 DR) 包含二进制十进制格式 (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。

STM32L151 的 RTC 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。

RTC 模块和时钟配置是在后备区域，即在系统复位或从待机模式唤醒后 RTC 的设置和时间维持不变，只要后备区域供电正常，那么 RTC 将可以一直运行。但是在系统复位后，会自动禁止访问后备寄存器和 RTC，以防止对后备区域(BKP)的意外写操作。所以在要设置时间之前，先要取消备份区域 (BKP) 写保护。RTC 的简化框图，如图 9.1.1 所示：

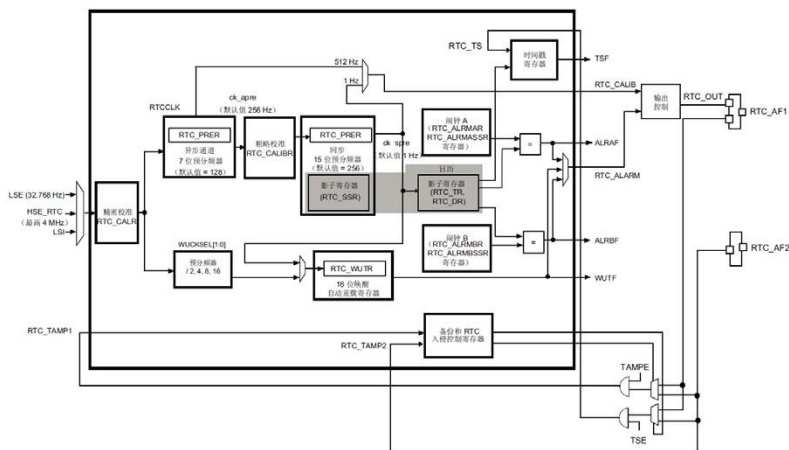


图 9.1.1RTC 框图

知识点:

- (1) 时钟源：对于 RTC 时钟来说，我们一般选择 LSE (低速外部时钟) 作为时钟来源。
- (2) 预分频器：异步预分频器时钟 CK_APER 用于为二进制 RTC_SSR 亚秒递减计数器提供时钟，同步预分频器 CK_SPER 用于更新日历。
- (3) 实时时钟和日历：时/分/秒可以直接从 RTC 时间寄存器 (RTC_TR) 中读取。

亚秒由 RTC_SSR 寄存器的值计算得到。

日期包含的年月日可以直接从 RTC 日期寄存器 (RTC_DR) 中读取。

本章我们用到 RTC 时钟和日历，并且用到闹钟功能。接下来简单介绍下 STM32L151RTC 时钟的使用。

1, 时钟和分频

首先，我们看 STM32L151 的 RTC 时钟分频。STM32L151 的 RTC 时钟源 (RTCCLK) 通过时钟控制器，可以从 LSE 时钟、LSI 时钟以及 HSE 时钟三者中选择 (通过 RCC_BDCR 寄存器选择)。一般我们选择 LSE，即外部 32.768KHz 晶振作为时钟源 (RTCCLK)，而 RTC 时钟核心，要求提供 1Hz 的时钟，所以，我们要设置 RTC 的可编程预分频器。STM32L151 的可编程预分频分配器 (RTC_PRER) 分为 2 个部分：

1, 一个通过 RTC_PRER 寄存器的 PREDIV_A 位配置的 7 位异步预分频器。

2, 一个通过 RTC_PRER 寄存器的 PREDIV_S 位配置的 15 位同步预分频器。图 9.1.1

RTC 框图中，ck_spre 的时钟可由如下计算公式计算：

$$Fck_spre = Frtcclk / [(PREDIV_S + 1) * (PREDIV_A + 1)]$$

其中，Fck_spre 即可用于更新日历时间等信息。PREDIV_A 和 PREDIV_S 为 RTC 的异步和同步分频器。且推荐设置 7 位异步预分频器 (PREDIV_A) 的值较大，以最大程度降低功耗。要设置为 32768 分频，我们只需要设置：PREDIV_A=0X7F，即 128 分频；PREDIV_S=0XFF，即 256 分频，即可得到 1Hz 的 Fck_spre。

另外，图 9.1.1 中，ck_apre 可作为 RTC 亚秒递减计数器 (RTC_SSR) 的时钟，Fck_apre 的计算公式如下：

$$Fck_apre = Frtcclk / (PREDIV_A + 1)$$

当 RTC_SSR 寄存器递减到 0 的时候，会使用 PREDIV_S 的值重新装载 PREDIV_S。而 PREDIV_S 一般为 255，这样，我们得到亚秒时间的精度是：1/256 秒，即 3.9ms 左右，有了这个亚秒寄存器 RTC_SSR，就可以得到更加精确的时间数据。

2, 日历时间 (RTC_TR) 和日期 (RTC_DR) 寄存器

STM32L151 的 RTC 日历时间 (RTC_TR) 和日期 (RTC_DR) 寄存器，用于存储时间和日期 (也可以用于设置时间和日期)，可以通过与 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。

每隔 2 个 RTCCLK 周期，当前日历值便会复制到影子寄存器，并置位 RTC_ISR 寄存器的 RSF 位。我们可以读取 RTC_TR 和 RTC_DR 来得到当前时间和日期信息，不过需要注意的是：时间和日期都是以 BCD 码的格式存储的，读出来要转换一下，才可以得到十进制的数字。

3, 可编程闹钟

STM32L151 提供两个可编程闹钟：闹钟 A (ALARM_A) 和闹钟 B (ALARM_B)。通过 RTC_CR 寄存器的 ALRAE 和 ALRBE 位置 1 来使能可编程闹钟功能。当日历的亚秒、秒、分、小时、日期分别与闹钟寄存器 RTC_ALRMASR/RTC_ALRMAR 和 RTC_ALRMBSSR/RTC_ALRMBR 中的值匹配时，则可以产生闹钟 (需要适当配置)。

本章我们将利用闹钟 A 产生闹铃，即设置 RTC_ALRMASR 和 RTC_ALRMAR 即可。

4, 周期性自动唤醒

STM32L151 的 RTC 不带秒钟中断了，但是多了一个周期性自动唤醒功能。周期性唤醒功能，由一个 16 位可编程自动重载递减计数器 (RTC_WUTR) 生成，可用于周期性中断/唤醒。

我们可以通过 RTC_CR 寄存器中的 WUTE 位设置使能此唤醒功能。唤醒定时器的时钟输入可以是：2、4、8 或 16 分频的 RTC 时钟 (RTCCLK)，也可以是 ck_spre 时钟（一般为 1Hz）

当选择 RTCCLK (假定 LSE 是：32.768kHz) 作为输入时钟时，可配置的唤醒中断周期介于 122us（因为 RTCCLK/2 时，RTC_WUTR 不能设置为 0）和 32 s 之间，分辨率最低为：61us

当选择 ck_spre (1Hz) 作为输入时钟时，可得到的唤醒时间为 1s 到 36h 左右，分辨率为 1 秒。并且这个 1s~36h 的可编程时间范围分为两部分：

当 WUCKSEL[2:1]=10 时为：1s 到 18h。

WUCKSEL[2:1]=11 时约为：18h 到 36h。

在后一种情况下，会将 2^{16} 添加到 16 位计数器当前值（即扩展到 17 位，相当于最高位用 WUCKSEL[1]代替）

初始化完成后，定时器开始递减计数。在低功耗模式下使能唤醒功能时，递减计数保持有效。此外，当计数器计数到 0 时，RTC_ISR 寄存器的 WUTF 标志会置 1，并且唤醒寄存器会使用其重载值（RTC_WUTR 寄存器值）动重载，之后必须用软件清零 WUTF 标志。

通过将 RTC_CR 寄存器中的 WUTIE 位置 1 来使能周期性唤醒中断时，可以使 STM32L151 退出低功耗模式。系统复位以及低功耗模式（睡眠、停机和待机）对唤醒定时器没有任何影响，它仍然可以正常工作，故唤醒定时器，可以用于周期性唤醒 STM32L151。

接下来，我们看看本章我们要用到的 RTC 部分寄存器，首先是 RTC 时间寄存器：RTC_TR，该寄存器各位描述如图 9.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PM	HT[1:0]		HU[3:0]			
									rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved	MNT[2:0]			MNU[3:0]				Reserved	ST[2:0]			SU[3:0]			
	rw	rw	rw	rw	rw	rw	rw		rw	rw	rw	rw	rw	rw	rw

位 31-24 保留

位 23 保留，必须保持复位值。

位 22 **PM**: AM/PM 符号 (AM/PM notation)

0: AM 或 24 小时制

1: PM

位 21:20 **HT[1:0]**: 小时的十位 (BCD 格式) (Hour tens in BCD format)

位 16:16 **HU[3:0]**: 小时的个位 (BCD 格式) (Hour units in BCD format)

位 15 保留，必须保持复位值。

位 14:12 **MNT[2:0]**: 分钟的十位 (BCD 格式) (Minute tens in BCD format)

位 11:8 **MNU[3:0]**: 分钟的个位 (BCD 格式) (Minute units in BCD format)

位 7 保留，必须保持复位值。

位 6:4 **ST[2:0]**: 秒的十位 (BCD 格式) (Second tens in BCD format)

位 3:0 **SU[3:0]**: 秒的个位 (BCD 格式) (Second units in BCD format)

图 9.1.2 RTC_TR 寄存器各位描述

这个寄存器比较简单，注意数据保存时 BCD 格式的，读取之后需要稍加转换，才是十进制的时分秒等数据，在初始化模式下，对该寄存器进行写操作，可以设置时间。

然后 RTC 日期寄存器：RTC_DR，该寄存器各位描述如图 9.1.3 所示

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								YT[3:0]				YU[3:0]			
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WDU[2:0]				MT		MU[3:0]			Reserved		DT[1:0]		DU[3:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31-24 保留

位 23:20 **YT[3:0]**: 年份的十位 (BCD 格式) (Year tens in BCD format)

位 19:16 **YU[3:0]**: 年份的个位 (BCD 格式) (Year units in BCD format)

位 15:13 **WDU[2:0]**: 星期几的个位 (Week day units)

000: 禁止

001: 星期一

...

111: 星期日

位 12 **MT**: 月份的十位 (BCD 格式) (Month tens in BCD format)

位 11:8 **MU**: 月份的个位 (BCD 格式) (Month units in BCD format)

位 7:6 保留, 必须保持复位值。

位 5:4 **DT[1:0]**: 日期的十位 (BCD 格式) (Date tens in BCD format)

位 3:0 **DU[3:0]**: 日期的个位 (BCD 格式) (Date units in BCD format)

图 9.1.3 RTC_DR 寄存器各位描述

同样, 该寄存器的数据采用 BCD 码格式 (如不熟悉 BCD, 百度即可), 其他的就比较简单了。同样, 在初始化模式下, 对该寄存器进行写操作, 可以设置日期。

接下来, 看 RTC 亚秒寄存器: RTC_SSR, 该寄存器各位描述如图 9.1.4 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SS[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留

位 15:0 **SS**: 亚秒值 (Sub second value)

SS[15:0] 是同步预分频器计数器的值。此亚秒值可根据以下公式得出:

亚秒值 = (PREDIV_S - SS) / (PREDIV_S + 1)

注意: 仅当执行平移操作之后, SS 才能大于 PREDIV_S。在这种情况下, 正确的时间/日期比 RTC_TR/RTC_DR 所指示的时间/日期慢一秒钟。

图 9.1.4 RTC SSR 寄存器各位描述

该寄存器可用于获取更加精确的 RTC 时间。不过, 在本章没有用到, 如果需要精确时间的地方, 大家可以使用该寄存器。

接下来看 RTC 控制寄存器: RTC_CR, 该寄存器各位描述如图 9.1.5 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								COE	OSEL[1:0]		POL	COSEL	BKP	SUB1H	ADD1H
								rw	rw	rw	rw	rw	rw	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TSIE	WUTIE	ALRBE	ALRAIE	TSE	WUTE	ALRBE	ALRAE	DCE	FMT	BYPH HAD	REFCKON	TSEDGE	WUCKSEL[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图 9.1.5 RTC_CR 寄存器各位描述

该寄存器我们不详细介绍每个位了，重点介绍几个要用到的：WUTIE，ALRAIE 是唤醒定时器中断和闹钟 A 中断使能位，本章要用到，设置为 1 即可。WUTE 和 ALRAE，则是唤醒定时器和闹钟 A 定时器使能位，同样设置为 1，开启。FMT 为小时格式选择位，我们设置为 0，选择 24 小时制。最后 WUCKSEL[2:0]，用于唤醒时钟选择，这个前面已经有介绍了，我们这里就不多说了。

接下来看 RTC 初始化和状态寄存器：RTC_ISR，该寄存器各位描述如图 9.1.6 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															REC PF
															r
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	TAMP 2F	TAMP 1F	TSOVF	TSF	WUTF	ALRBF	ALRAF	INIT	INITF	RSF	INITS	SHPF	WUT WF	ALRB WF	ALRA WF
	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rw	r	rc_w0	r	rc_w0	r	r	r

图 9.1.6 RTC_ISR 寄存器各位描述

该寄存器中，WUTF、ALRBF 和 ALRAF，分别是唤醒定时器闹钟 B 和闹钟 A 的中断标志位，当对应事件产生时，这些标志位被置 1，如果设置了中断，则会进入中断服务函数，这些位通过软件写 0 清除；INIT 为初始化模式控制位，要初始化 RTC 时，必须先设置 INIT=1；INITF 为初始化标志位，当设置 INIT 为 1 以后，要等待 INITF 为 1，才可以更新时间、日期和预分频寄存器；RSF 位为寄存器同步标志，仅在该位为 1 时，表示日历影子寄存器已同步，可以正确读取 RTC_TR/RTC_DR 寄存器的值了；WUTWF、ALRBWF 和 ALRAWF 分别是唤醒定时器、闹钟 B 和闹钟 A 的写标志，只有在这些位为 1 的时候，才可以更新对应的内容，比如：要设置闹钟 A 的 ALRMAR 和 ALRMASR，则必须先等待 ALRAWF 为 1，才可以设置。

接下来看 RTC 预分频寄存器：RTC_PRER，该寄存器各位描述如图 9.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved									PREDIV_A[6:0]						
									rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	PREDIV_S[14:0]														
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:24 保留

位 23 保留，必须保持复位值。

位 22:16 **PREDIV_A[6:0]**: 异步预分频系数 (Asynchronous prescaler factor)

下面是异步分频系数的公式：

$ck_apre \text{ 频率} = RTCCLK \text{ 频率} / (PREDIV_A + 1)$

注意：PREDIV_A [6:0]= 000000 为禁用值。

位 15 保留，必须保持复位值。

位 14:0 **PREDIV_S[14:0]**: 同步预分频系数 (Synchronous prescaler factor)

下面是同步分频系数的公式：

$ck_spre \text{ 频率} = ck_apre \text{ 频率} / (PREDIV_S + 1)$

图 9.1.7 RTC_PRER 寄存器各位描述

该寄存器用于 RTC 的分频，我们在之前也有讲过，这里就不多说了。该寄存器的配置，必须在初始化模式（INITF=1）下，才可以进行。

接下来看 RTC 唤醒定时器寄存器：RTC_WUTR，该寄存器各位描述如图 9.1.8 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
WUT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:16 保留

位 15:0 **WUT[15:0]**: 唤醒自动重载值位 (Wakeup auto-reload value bit)

当使能唤醒定时器时 (WUTE 置 1), 每 (WUT[15:0] + 1) 个 ck_wut 周期将 WUTF 标志置 1 一次。 ck_wut 周期通过 RTC_CR 寄存器的 WUCKSEL[2:0] 位进行选择。

当 WUCKSEL[2] = 1 时, 唤醒定时器变为 17 位, WUCKSEL[1] 等效为 WUT[16], 即要重载到定时器的最高有效位。

注意: WUTF 第一次置 1 发生在 WUTE 置 1 之后 (WUT+1) 个 ck_wut 周期。禁止在 WUCKSEL[2:0]=011(RTCCLK/2) 时将 WUT[15:0] 设置为 0x0000。

图 9.1.8 RTC_WUTR 寄存器各位描述

该寄存器用于设置自动唤醒重装载值, 可用于设置唤醒周期。该寄存器的配置, 必须等待 RTC_ISR 的 WUTWF 为 1 才可以进行。

接下来看 RTC 闹钟 A 器寄存器: RTC_ALRMAR, 该寄存器各位描述如图 9.1.9 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
MSK4	WDSEL	DT[1:0]		DU[3:0]				MSK3	PM	HT[1:0]		HU[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MSK2	MNT[2:0]			MNU[3:0]				MSK1	ST[2:0]			SU[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31 **MSK4**: 闹钟 A 日期掩码 (Alarm A date mask)

0: 如果日期/日匹配, 则闹钟 A 置 1

1: 在闹钟 A 比较中, 日期/日无关

位 30 **WDSEL**: 星期几选择 (Week day selection)

0: DU[3:0] 代表日期的个位

1: DU[3:0] 代表星期几。DT[1:0] 为无关位。

位 29:28 **DT[1:0]**: 日期的十位 (BCD 格式) (Date tens in BCD format)。

位 27:24 **DU[3:0]**: 日期的个位或日 (BCD 格式) (Date units or day in BCD format)。

位 23 **MSK3**: 闹钟 A 小时掩码 (Alarm A hours mask)

0: 如果小时匹配, 则闹钟 A 置 1

1: 在闹钟 A 比较中, 小时无关

位 22 **PM**: AM/PM 符号 (AM/PM notation)

0: AM 或 24 小时制

1: PM

位 21:20 **HT[1:0]**: 小时的十位 (BCD 格式) (Hour tens in BCD format)。

位 19:16 **HU[3:0]**: 小时的个位 (BCD 格式) (Hour units in BCD format)。

位 15 **MSK2**: 闹钟 A 分钟掩码 (Alarm A minutes mask)

0: 如果分钟匹配, 则闹钟 A 置 1

1: 在闹钟 A 比较中, 分钟无关

位 14:12 **MNT[2:0]**: 分钟的十位 (BCD 格式) (Minute tens in BCD format)。

位 11:8 **MNU[3:0]**: 分钟的个位 (BCD 格式) (Minute units in BCD format)。

位 7 **MSK1**: 闹钟 A 秒掩码 (Alarm A seconds mask)

0: 如果秒匹配, 则闹钟 A 置 1

1: 在闹钟 A 比较中, 秒无关

位 6:4 **ST[2:0]**: 秒的十位 (BCD 格式) (Second tens in BCD format)。

位 3:0 **SU[3:0]**: 秒的个位 (BCD 格式) (Second units in BCD format)。

图 9.1.9 RTC_ALRMAR 寄存器各位描述

该寄存器用于设置闹铃 A，当 WDSEL 选择 1 时，使用星期制闹铃，本章我们选择星期制闹铃。该寄存器的配置，必须等待 RTC_ISR 的 ALRAWF 为 1 才可以进行。另外，还有 RTC_ALRMASR 寄存器。

接下来看 RTC 写保护寄存器：RTC_WPR，该寄存器比较简单，低八位有效。上电后，所有 RTC 寄存器都受到写保护（RTC_ISR[13:8]、RTC_TAFCR 和 RTC_BKPxR 除外），必须依次写入：0XCA、0X53 两关键字到 RTC_WPR 寄存器，才可以解锁。写一个错误的关键字将再次 激活 RTC 的寄存器写保护。接下来，我们介绍下 RTC 备份寄存器：RTC_BKPxR，该寄存器组总共有 20 个，每个寄存器是 32 位的，可以存储 80 个字节的用户数据，这些寄存器在备份域中实现，可在 VDD 电源关闭时通过 VBAT 保持上电状态。备份寄存器不会在系统复位或电源复位时复位，也不会 MCU 从待机模式唤醒时复位。

复位后，对 RTC 和 RTC 备份寄存器的写访问被禁止，执行以下操作可以使能对 RTC 及 RTC 备份寄存器的写访问：

- 1) 通过设置寄存器 RCC_APB1ENR 的 PWREN 位来打开电源接口时钟
- 2) 电源控制寄存器 (PWR_CR) 的 DBP 位来使能对 RTC 及 RTC 备份寄存器的访问。我们可以用 BKP 来存储一些重要的数据，相当于一个 EEPROM，不过这个 EEPROM 并不是真正的 EEPROM，而是需要电池来维持它的数据。最后，我们还要介绍一下备份区域控制寄存器 RCC_BDCR。该寄存器的各位描述如图 9.1.10 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															BDRST
															r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
RTCCEN	Reserved					RTCSEL[1:0]		Reserved					LSEBYP	LSERDY	LSEON
r/w						r/w							r/w	r	r/w

位 31:17 保留，必须保持复位值。

位 16 **BDRST**：备份域软件复位 (Backup domain software reset)

由软件置 1 和清零。

0：复位未激活

1：复位整个备份域

注意：BKPSRAM 不受此复位影响，只能在 Flash 保护级别从级别 1 更改为级别 0 时复位 BKPSRAM。

位 15 **RTCCEN**：RTC 时钟使能 (RTC clock enable)

由软件置 1 和清零。

0：RTC 时钟禁止

1：RTC 时钟使能

位 14:10 保留，必须保持复位值。

位 9:8 **RTCSEL[1:0]**：RTC 时钟源选择 (RTC clock source selection)

由软件置 1，用于选择 RTC 的时钟源。选择 RTC 时钟源后，除非备份域复位，否则其不可再更改。可使用 BDRST 位对其进行复位。

00：无时钟

01：LSE 振荡器时钟用作 RTC 时钟

10：LSI 振荡器时钟用作 RTC 时钟

11：由可编程预分频器分频的 HSE 振荡器时钟（通过 RCC 时钟配置寄存器 (RCC_CFGR) 中的 RTCPRE[4:0] 位选择）用作 RTC 时钟

位 7:3 保留，必须保持复位值。

位 2 **LSEBYP**：外部低速振荡器旁路 (External low-speed oscillator bypass)

由软件置 1 和清零，用于旁路调试模式下的振荡器。只有在禁止 LSE 时钟后才能写入该位。

0：不旁路 LSE 振荡器

1：旁路 LSE 振荡器

位 1 **LSERDY**：外部低速振荡器就绪 (External low-speed oscillator ready)

由硬件置 1 和清零，用于指示外部 32 kHz 振荡器已稳定。在 LSEON 位被清零后，LSERDY 将在 6 个外部低速振荡器时钟周期后转为低电平。

0：LSE 时钟未就绪

1：LSE 时钟就绪

位 0 **LSEON**：外部低速振荡器使能 (External low-speed oscillator enable)

由软件置 1 和清零。

0：LSE 时钟关闭

1：LSE 时钟开启

图 9.1.10 RCC_BDCR 寄存器各位描述

RTC 的时钟源选择及使能设置都是通过这个寄存器来实现的，所以我们在 RTC 操作之前先要通过这个寄存器选择 RTC 的时钟源，然后才能开始其他的操作。

RTC 寄存器介绍就给大家介绍到这里了，我们下面来看看要经过哪几个步骤的配置才能使 RTC 正常工作。接下来我们来看看通过 HAL 库配置 RTC 一般配置步骤。RTC 相关的 HAL 库文件为 stm32l1xx_hal_rtc.c 以及头文件 stm32l1xx_hal_rtc.h 中：

1) 使能电源时钟，并使能 RTC 及 RTC 后备寄存器写访问。

前面已经介绍了，我们要访问 RTC 和 RTC 备份区域就必须先使能电源时钟，然后使能 RTC 即后备区域访问。电源时钟使能，通过 RCC_APB1ENR 寄存器来设置；RTC 及 RTC 备份寄存器的写访问，通过 PWR_CR 寄存器的 DBP 位设置。HAL 库设置方法为：

```
HAL_RCC_PWR_CLK_ENABLE();//使能电源时钟
PWR_HAL_PWR_EnableBkUpAccess();//取消备份区
```

2) 开启外部低速振荡器 LSE，选择 RTC 时钟，并使能。

配置启 LSE 的函数为 HAL_RCC_OscConfig，使用方法为：

```
RCC_OscInitStruct.OscillatorType=RCC_OSCILLATORTYPE_LSE;//LSE 配置
RCC_OscInitStruct.PLL.PLLState=RCC_PLL_NONE;
RCC_OscInitStruct.LSEState=RCC_LSE_ON;
HAL_RCC_OscConfig(&RCC_OscInitStruct);
```

选择 RTC 时钟源为函数为 HAL_RCCEx_PeriphCLKConfig，使用方法为：

```
PeriphClkInitStruct.PeriphClockSelection=RCC_PERIPHCLK_RTC;//外设为
RTCPeriphClkInitStruct.RTCClockSelection=RCC_RTCCLKSOURCE_LSE;//RTC 时钟源 LSE
HAL_RCCEx_PeriphCLKConfig(&PeriphClkInitStruct);
```

使能 RTC 时钟方法为：

```
HAL_RCC_RTC_ENABLE();//RTC 时钟使能
```

3) 初始化 RTC，设置 RTC 的分频，以及配置 RTC 参数。

在 HAL 中，初始化 RTC 是通过函数 HAL_RTC_Init 实现的，该函数声明为：

```
HAL_StatusTypeDef HAL_RTC_Init(RTC_HandleTypeDef* hrtc);
```

同样按照以前的方式，我们来看看 RTC 初始化参数结构体 RTC_HandleTypeDef 定义：

```
typedef struct
{
    RTC_TypeDef                *Instance;
    RTC_InitTypeDef            Init;
    HAL_LockTypeDef            Lock;
    __IO HAL_RTCStateTypeDef   State;
}RTC_HandleTypeDef;
```

这里我们着重讲解成员变量 Init 含义，因为它是真正的 RTC 初始化变量，它是 RTC_InitTypeDef 结构体类型，结构体 RTC_InitTypeDef 定义为：


```
typedef struct
{
    uint32_t HourFormat;    //小时格式
    uint32_t AsynchPrediv;  //异步预分频系数
    uint32_t SynchPrediv;   //同步预分频系数
    uint32_t OutPut;        //选择连接到 RTC_ALARM 输出的标志
    uint32_t OutPutPolarity; //设置 RTC_ALARM 的输出极性
    uint32_t OutPutType;    //设置 RTC_ALARM 的输出类型为开漏输出还是推挽输出
}RTC_InitTypeDef;
```

该结构体有 6 个成员变量

成员变量 HourFormat 用来设置小时格式，为 12 小时制或者 24 小时制，取值为 RTC_HOURFORMAT_12 或者 RTC_HOURFORMAT_24。

AsynchPrediv 用来设置 RTC 的异步预分频系数，也就是设置 RTC_PRER 寄存器的 PREDIV_A 相关位，因为异步预分频系数是 7 位，所以最大值为 0x7F，不能超过这个值。

SynchPrediv 用来设置 RTC 的同步预分频系数，也就是设置 RTC_PRER 寄存器的 PREDIV_S 相关位，因为同步预分频系数也是 15 位，所以最大值为 0x7FFF，不能超过这个值。

OutPut 用来选择要连接到 RTC_ALARM 输出的标志，取值为：RTC_OUTPUT_DISABLE（禁止输出），RTC_OUTPUT_ALARMA（使能闹钟 A 输出），RTC_OUTPUT_ALARMB（使能闹钟 B 输出）和 RTC_OUTPUT_WAKEUP（使能唤醒输出）。

OutPutPolarity 用来设置 RTC_ALARM 的输出极性，与 Output 成员变量配合使用，取值为 RTC_OUTPUT_POLARITY_HIGH（高电平）或 RTC_OUTPUT_POLARITY_LOW（低电平）。

OutPutType 用来设置 RTC_ALARM 的输出类型为开漏（RTC_OUTPUT_TYPE_OPENDRAIN）还是推挽输出（RTC_OUTPUT_TYPE_PUSHPULL）。与成员变量 OutPut 和 OutPutPolarity 配合使用。

接下来我们看看 RTC 初始化的一般格式：

```
RTC_Handler.Instance=RTC;
RTC_Handler.Init.HourFormat=RTC_HOURFORMAT_24;//RTC 设置为 24 小时格式
RTC_Handler.Init.AsynchPrediv=0X7F; //RTC 异步分频系数(1~0X7F)
RTC_Handler.Init.SynchPrediv=0XFF; //RTC 同步分频系数（0~7FFF）
RTC_Handler.Init.OutPut=RTC_OUTPUT_DISABLE;
RTC_Handler.Init.OutPutPolarity=RTC_OUTPUT_POLARITY_HIGH;
RTC_Handler.Init.OutPutType=RTC_OUTPUT_TYPE_OPENDRAIN;
HAL_RTC_Init(&RTC_Handler);
```

接下来我们看看 RTC 初始化的一般格式：

```
RTC_Handler.Instance=RTC;
RTC_Handler.Init.HourFormat=RTC_HOURFORMAT_24;//RTC 设置为 24 小时格式
RTC_Handler.Init.AsynchPrediv=0X7F; //RTC 异步分频系数(1~0X7F)
RTC_Handler.Init.SynchPrediv=0XFF; //RTC 同步分频系数 (0~7FFF)
RTC_Handler.Init.OutPut=RTC_OUTPUT_DISABLE;
RTC_Handler.Init.OutPutPolarity=RTC_OUTPUT_POLARITY_HIGH;
RTC_Handler.Init.OutPutType=RTC_OUTPUT_TYPE_OPENDRAIN;
HAL_RTC_Init(&RTC_Handler);
```

同样，HAL 库也提供了 RTC 初始化 MSP 函数。函数声明为：

```
void HAL_RTC_MspInit(RTC_HandleTypeDef*hrtc);
```

该函数内部一般存放时钟使能，时钟源选择等操作程序。

4) 设置 RTC 的时间。

HAL 库中，设置 RTC 时间的函数为：

```
HAL_StatusTypeDef HAL_RTC_SetTime(RTC_HandleTypeDef*hrtc,
                                   RTC_TimeTypeDef *sTime, uint32_tFormat);
```

实际上，根据我们前面寄存器的讲解，RTC_SetTime 函数是用来设置时间寄存器 RTC_TR 的相关位的值。

RTC_SetTime 函数的第三个参数 Format，用来设置输入的时间格式为 BIN 格式还是 BCD 格式，可选值为 RTC_FORMAT_BIN 和 RTC_FORMAT_BCD。

我们接下来看看第二个初始化参数结构体 RTC_TimeTypeDef 的定义：

```
typedef struct
{
    uint8_tHours;
    uint8_tMinutes;
    uint8_tSeconds;
    uint8_tTimeFormat;
    uint32_tSubSeconds
    ;
    uint32_tSecondFraction;
    uint32_tDayLightSaving
    uint32_tStoreOperation;
```

前面四个成员变量就比较好理解了，分别用来设置 RTC 时间参数的小时，分钟，秒钟，以及 AM/PM 符号，大家参考前面讲解的 RTC_TR 的位描述即可。SubSeconds 用来读取保存亚秒寄存器 RTC_SSR 的值，SecondFraction 用来读取保存同步预分频系数的值，也就是 RTC_PRER 的位 0~14，DayLightSaving 用来设置日历时间增加 1 小时，减少 1 小时，还是不变。StoreOperation 用户可对此变量设置以记录是否已对夏令时进行更改。HAL_RTC_SetTime 函数参考实例如下：

```
RTC_TimeTypeDef RTC_TimeStructure;  
RTC_TimeStructure.Hours=1;  
RTC_TimeStructure.Minutes=1;  
RTC_TimeStructure.Seconds=1;  
RTC_TimeStructure.TimeFormat=RTC_HOURFORMAT12_PM;  
RTC_TimeStructure.DayLightSaving=RTC_DAYLIGHTSAVING_NONE;  
RTC_TimeStructure.StoreOperation=RTC_STOREOPERATION_RESET;  
HAL_RTC_SetTime(&RTC_Handler,&RTC_TimeStructure,RTC_FORMAT_BIN);
```

5) 设置 RTC 的日期。

设置 RTC 的日期函数为：

```
HAL_StatusTypeDef HAL_RTC_SetDate(RTC_HandleTypeDef* hrtc,  
                                   RTC_DateTypeDef* sDate, uint32_t Format);
```

实际上，根据我们前面寄存器的讲解，HAL_RTC_SetDate 设置日期函数是用来设置日期寄存器 RTC_DR 的相关位的值。

该函数有三个入口参数，我们着重讲解第二个入口参数 sDate，它是结构体 RTC_DateTypeDef 指针类型变量，结构体 RTC_DateTypeDef 定义如下：

```
typedef struct  
{  
    uint8_t WeekDay;    //星期几  
    uint8_t tMonth;    //月份  
    uint8_t tDate;    //日期  
    uint8_t tYear;    //年份  
} RTC_DateTypeDef;
```

结构体一共四个成员变量，这四个成员变量非常好理解，对应的是 RTC_DR 寄存器相关设置位，这里我们就不做过多讲解。

6) 获取 RTC 当前日期和时间。

获取当前 RTC 时间的函数为：

```
HAL_StatusTypeDef HAL_RTC_GetTime(RTC_HandleTypeDef* hrtc,  
                                   RTC_TimeTypeDef* sTime, uint32_t Format);
```

获取当前 RTC 日期的函数为：

```
HAL_StatusTypeDef HAL_RTC_GetDate(RTC_HandleTypeDef* hrtc,  
                                   RTC_DateTypeDef* sDate, uint32_t Format);
```


这两个函数非常简单，实际就是读取 RTC_TR 寄存器和 RTC_DR 寄存器的时间和日期的值，然后将值存放到相应的结构体中。

通过以上 6 个步骤，我们就完成了对 RTC 的配置，RTC 即可正常工作，而且这些操作不是每次上电都必须执行的，可以视情况而定。当然，我们还需要设置时间、日期、唤醒中断、闹钟等，这些将在后面介绍。

9.2 硬件设计

本实验用到的硬件资源有：

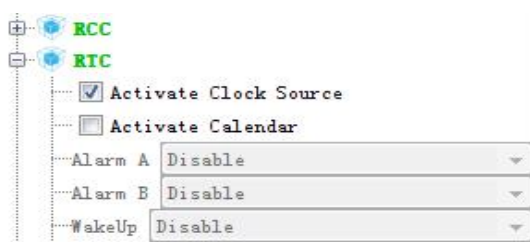
- 1) 指示灯 LED1
- 2) RTC
- 3) TFT_LCD 模块

前面 2 个都介绍过了，而 RTC 属于 STM32L1 内部资源，其配置也是通过软件设置好就可以了。不过 RTC 不能断电，否则数据就丢失了，我们如果想让时间在断电后还可以继续走，那么必须确保开发板有电，STM32L1 没有后备电池接口，所以每次开发板断电之后需要重新设置 RTC。

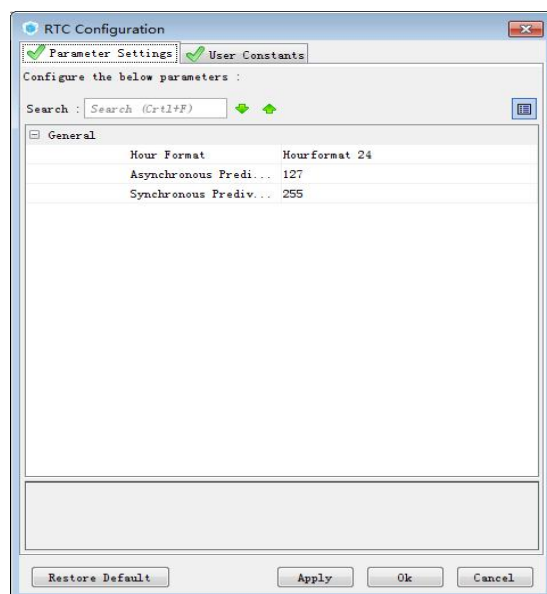
9.3 STM32CubeMX 配置 RTC 和软件设计

使用 STM32CubeMX 配置 RTC 的一般步骤为：

- ① 进入 Pinout->RTC 配置栏，启动 RTC 基本参数：



- ②进入 Configuration->RTC 配置栏，配置基本参数：



注：关于 LCD 显示屏的 SPI 配置，和上一章一样。详情可参见上一章，这里就不重复了。
打开生成工程可以看到，首先是 MX_RTC_Init，其代码如下：

```
void MX_RTC_Init(void)
{
    hrtc.Instance = RTC;
    hrtc.Init.HourFormat = RTC_HOURFORMAT_24;
    hrtc.Init.AsynchPrediv = 127;
    hrtc.Init.SynchPrediv = 255;
    hrtc.Init.OutPut = RTC_OUTPUT_DISABLE;
    hrtc.Init.OutPutPolarity = RTC_OUTPUT_POLARITY_HIGH;
    hrtc.Init.OutPutType = RTC_OUTPUT_TYPE_OPENDRAIN;
    if (HAL_RTC_Init(&hrtc) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    RTC_Set_Time(0X18,0X01,0X30,0X02,0X10,0X10,0X00);
}
```

该函数用来初始化 RTC 配置以及日期和时钟，需要注意的是 STM32L1 没有后备电池接口，所以我们不用处理后备寄存器，我们也不在这里讲解。

这里设置时间和日期，分别是通过 RTC_Set_Time 和 RTC_Set_Date 函数来实现的，这两个函数实际就是调用库函数里面的 HAL_RTC_SetTime 函数和 HAL_RTC_SetDate 函数来实现。自己编写 RTC_Set_Time()。

```
void RTC_Set_Time(unsigned char year,unsigned char month,unsigned char
date,unsigned char week,unsigned char hour,unsigned char min,unsigned char
sec)
{
    RTC_TimeTypeDef sTime;
    RTC_DateTypeDef DateToUpdate;
    DateToUpdate.WeekDay = week;
    DateToUpdate.Month = month;
    DateToUpdate.Date = date;
    DateToUpdate.Year = year;
    HAL_RTC_SetDate(&hrtc,&DateToUpdate,RTC_FORMAT_BIN);
    sTime.Hours = hour;
    sTime.Minutes = min;
    sTime.Seconds = sec;
    HAL_RTC_SetTime(&hrtc, &sTime, RTC_FORMAT_BIN);
}
```

这里得到时间和日期，分别是通过 `RTC_Get_Time` 和 `RTC_Get_Date` 函数来实现的，这两个函数实际就是调用库函数里面的 `HAL_RTC_GetTime` 函数和 `HAL_RTC_GetDate` 函数来实现。

```
void RTC_Get_NowTime(void)
{
    HAL_RTC_GetDate(&hrtc,&Get_Date,RTC_FORMAT_BIN);
    HAL_RTC_GetTime(&hrtc,&Get_Time,RTC_FORMAT_BIN);
}
```

接着，我们介绍一下 `RTC_Set_AlarmA` 函数，该函数代码如下：

```
//设置闹钟 A（设置 ALRMAR 和 ALRMASR 寄存器的值）
void RTC_Set_AlarmA(unsigned char week,unsigned char hour,unsigned char min,
                    unsigned char sec)
{
    RTC_AlarmTypeDef RTC_AlarmSturuct;
    RTC_AlarmSturuct.AlarmTime.Hours=hour; //小时
    RTC_AlarmSturuct.AlarmTime.Minutes=min; //分钟
    RTC_AlarmSturuct.AlarmTime.Seconds=sec; //秒
    RTC_AlarmSturuct.AlarmTime.TimeFormat=RTC_HOURFORMAT12_AM;

    RTC_AlarmSturuct.AlarmMask=RTC_ALARMMASK_NONE; //精确匹配星期，时分秒
    RTC_AlarmSturuct.AlarmDateWeekDaySel=
        RTC_ALARMDATEWEEKDAYSEL_WEEKDAY; //按星期
    RTC_AlarmSturuct.AlarmDateWeekDay=week; //星期
    RTC_AlarmSturuct.Alarm=RTC_ALARM_A; //闹钟
    HAL_RTC_SetAlarm_IT(&RTC_Handler,&RTC_AlarmSturuct,RTC_FORMAT_BIN);

    HAL_NVIC_SetPriority(RTC_Alarm_IRQn,0x01,0x02); //抢占优先级 1,子优先级
    HAL_NVIC_EnableIRQ(RTC_Alarm_IRQn);
}
```

该函数用于设置闹钟 A，也就是设置 `ALRMAR` 和 `ALRMASR` 寄存器的值，来设置闹钟时间，这里 HAL 库中用来设置闹钟并开启闹钟中断的函数为：

```
HAL_StatusTypeDef HAL_RTC_SetAlarm_IT(RTC_HandleTypeDef* hrtc,
                                       RTC_AlarmTypeDef* sAlarm, uint32_t Format);
```

第三个参数 `RTC_Format` 用来设置格式，这里前面我们讲解过，就不做过多讲解。接下来我们着重看看第二个参数 `sAlarm`，该入口参数是 `RTC_AlarmTypeDef` 结构体指针类型，结构体定义如下：

```
typedef struct
{
    RTC_TimeTypeDef AlarmTime;
    uint32_t AlarmMask;
    uint32_t AlarmSubSecondMask;
    uint32_t AlarmDateWeekDaySel;
    uint8_t AlarmDateWeekDay;
    uint32_t Alarm;
}RTC_AlarmTypeDef;
```

该结构体有6个成员变量,第一个成员变量AlarmTime用来设置闹钟时间,是RTC_TimeTypeDef结构体类型,该结构体前面我们已经讲解过各个成员变量含义,这里我们就不做过多讲解。

AlarmMask用来设置闹钟时间掩码,也就是在我们第一个参数设置的时间中(包括后面参数RTC_AlarmDateWeekDay设置的星期几/哪一天),哪些是无关的。比如我们设置闹钟时间为每天的10点10分10秒,那么我们可以选择值RTC_AlarmMask_DateWeekDay,也就是我们不关心是星期几/每月哪一天。这里我们选择为RTC_AlarmMask_None,也就是精确匹配时间,所有的时分秒以及星期几/(或者每月哪一天)都要精确匹配。

AlarmSubSecondMask和AlarmMask作用类似,只不过该变量是用来设置亚秒的。

AlarmDateWeekDaySel用来选择是闹钟是按日期还是按星期。比如我们选择RTC_AlarmDateWeekDaySel_WeekDay那么闹钟就是按星期。如果我们选择RTC_AlarmDateWeekDaySel_Date那么闹钟就是按日期。这与后面第四个参数是有关联的,我们在后面第四个参数讲解。

AlarmDateWeekDay用来设置闹钟的日期或者星期几。比如我们第三个参数RTC_AlarmDateWeekDaySel设置了值为RTC_AlarmDateWeekDaySel_WeekDay,也就是按星期,那么参数RTC_AlarmDateWeekDay的取值范围就为星期一~星期天,也就是RTC_Weekday_Monday~RTC_Weekday_Sunday。如果第三个参数RTC_AlarmDateWeekDaySel设置值为RTC_AlarmDateWeekDaySel_Date,那么它的取值范围就为日期值,0~31。

Alarm用来设置是闹钟A还是闹钟B,这个很好理解。

调用函数RTC_SetAlarm设置闹钟A的参数之后,最后,开启闹钟A中断(连接在外部中断线17),并设置中断分组。当RTC的时间和闹钟A设置的时间完全匹配时,将产生闹钟中断。

接着,我们介绍一下RTC_Set_WakeUp函数,该函数代码如下:

```
void RTC_Set_WakeUp(u32 wksel,u16cnt)
{
    HAL_RTC_WAKEUPTIMER_CLEAR_FLAG(&RTC_Handler,
                                     RTC_FLAG_WUTF); //清除 RTCWAKE UP 的标志
    HAL_RTCEx_SetWakeUpTimer_IT(&RTC_Handler,cnt,wksel); //设置重装载值和时钟
    HAL_NVIC_SetPriority(RTC_WKUP_IRQn,0x02,0x02); //抢占优先级 1,子优先级 2
    HAL_NVIC_EnableIRQ(RTC_WKUP_IRQn);
}
```

该函数用于设置 RTC 周期性唤醒定时器，实现周期性唤醒中断，连接在外部中断线 22。该函数调用的是 HAL 库函数 HAL_RTCEx_SetWakeUpTimer_IT 实现的，该函数使用方法比较简单，这里我们就不做过多讲解。

有了中断设置函数，就必定有中断服务函数，同时因为 HAL 库会开放中断处理回调函数，接下来看这两个中断的中断服务函数和中断处理回调函数，代码如下：

```
//RTC 闹钟中断服务函数
void RTC_Alarm_IRQHandler(void)
{
    HAL_RTC_AlarmIRQHandler(&RTC_Handler);
}

//RTC WAKE UP 中断服务函数
void RTC_WKUP_IRQHandler(void)
{
    HAL_RTCEx_WakeUpTimerIRQHandler(&RTC_Handler);
}

//RTC 闹钟 A 中断处理回调函数
void HAL_RTC_AlarmAEventCallback(RTC_HandleTypeDef* hrtc)
{
}

//RTC WAKE UP 中断处理回调函数
void HAL_RTCEx_WakeUpTimerEventCallback(RTC_HandleTypeDef* hrtc)
{
    if(HAL_GPIO_ReadPin(LED1_GPIO_Port, LED1_Pin))
    {
        CL_LED1();
    }
    else
    {
        OP_LED1();
    }
}
```

RTC_WKUP_IRQHandler 函数用于 RTC 自动唤醒定时器中断，其中断控制逻辑写在中断回调函数 HAL_RTCEx_WakeUpTimerEventCallback 中，可以通过观察 LED1 的状态来查看 RTC 自动唤醒中断的情况。

最后我们看看 main 函数源码如下：

```
Int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI2_Init();
    LCD_Init();
    LCD_Clear(BLACK);
    MX_RTC_Init();
    RTC_Set_WakeUp(RTC_WAKEUPCLOCK_CK_SPRE_16BITS,0);
    Show_Str(0,10,BLUE,BLACK,"Time:",16,1);
    while(1)
    {
        RTC_Get_NowTime();
        POINT_COLOR=BLUE;
        LCD_Fill(0,30,16,46,BLACK);
        LCD_ShowNum(0,30,Get_Date.Year,2,16);
        Show_Str(16,30,BLUE,BLACK,"/",16,1);
        LCD_Fill(24,30,40,46,BLACK);
        LCD_ShowNum(24,30,Get_Date.Month,2,16);
        Show_Str(40,30,BLUE,BLACK,"/",16,1);
        LCD_Fill(48,30,64,46,BLACK);
        LCD_ShowNum(48,30,Get_Date.Date,2,16);
        LCD_Fill(0,50,16,66,BLACK);
        LCD_ShowNum(0,50,Get_Time.Hours,2,16);
        Show_Str(16,50,BLUE,BLACK,":",16,1);
        LCD_Fill(24,50,40,66,BLACK);
        LCD_ShowNum(24,50,Get_Time.Minutes,2,16);
        Show_Str(40,50,BLUE,BLACK,":",16,1);
        LCD_Fill(48,50,64,66,BLACK);
        LCD_ShowNum(48,50,Get_Time.Seconds,2,16);
    }
}
```

main.c 前面是初始化，然后在死循环里读取时间值，然后显示。

9.4 下载验证

将程序下载到 STM32L151 后，可以看到 LED1 每隔一秒钟亮一次，说明周期性唤醒中断工作正常。然后，可以看到 LCD 模块开始显示时间，实际显示效果如图 9.4.1 所示：

第十章待机唤醒实验

本章我们将向大家介绍 STM32L151 的待机唤醒功能。在本章中，我们将使用 KEY_UP 按键来实现唤醒和进入待机模式的功能，然后使用 LED1 指示状态。本章将分为如下几个部分：

- 10.1 STM32L151 待机模式简介
- 10.2 硬件设计
- 10.3 STM32CubeMX 配置 WKUP 和软件设计
- 10.4 下载验证

10.1 STM32L151 待机模式简介

很多单片机都有低功耗模式，STM32L151 也不例外。在系统或电源复位以后，微控制器处于运行状态。运行状态下的 HCLK 为 CPU 提供时钟，内核执行程序代码。当 CPU 不需继续运行时，可以利用多个低功耗模式来节省功耗，例如等待某个外部事件时。用户需要根据最低电源消耗，最快速启动时间和可用的唤醒源等条件，选定一个最佳的低功耗模式。

STM32L151 提供了 3 种低功耗模式以达到不同层次的降低功耗的目的这三种模式如下：

- 1) 睡眠模式（CM3 内核停止工作，外设仍在运行）；
- 2) 停止模式（所有的时钟都停止）；
- 3) 待机模式；

在运行模式下，我们也可以通过降低系统时钟关闭 APB 和 AHB 总线上未被使用的外设的时钟来降低功耗。三种低功耗模式一览表见表 10.1.1 所示：

模式名称	进入	唤醒	对 1.2 V 域时钟的影响	对 V _{DD} 域时钟的影响	调压器
睡眠 (立即休眠或退出时休眠)	WFI	任意中断	CPU CLK 关闭 对其它时钟或模拟时钟源无影响	无	开启
	WFE	唤醒事件			
停止	PDDS 和 LPDS 位 + SLEEPDEEP 位 + WFI 或 WFE	任意 EXTI 线（在 EXTI 寄存器中配置，内部线和外部线）	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	开启或处于低功耗模式（取决于用于 STM32F405xx/07xx 和 STM32F415xx/17xx 的 PWR 电源控制寄存器 (PWR_CR) 和用于 STM32F42xxx 和 STM32F43xxx 的 PWR 电源控制寄存器 (PWR_CR)）
待机	PDDS 位 + SLEEPDEEP 位 + WFI 或 WFE	WKUP 引脚上升沿、RTC 闹钟（闹钟 A 或闹钟 B）、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位、IWDG 复位	所有 1.2 V 域时钟都关闭	HSI 和 HSE 振荡器关闭	关闭

表 10.1.1 STM32L151 低功耗一览表

在这三种低功耗模式中，最低功耗的是待机模式，在此模式下，最低只需要 2.2uA 左右的电流。停机模式是次低功耗的，其典型的电流消耗在 350uA 左右。最后就是睡眠模式了。用户可以根据自己的需求来决定使用哪种低功耗模式。

本章，我们仅对 STM32L151 的最低功耗模式-待机模式，来做介绍。待机模式可实现 STM32L151 的最低功耗。该模式是在 CM3 深睡眠模式时关闭电压调节器。整个 1.8V 供电区域被断电。PLL、HSI 和 HSE 振荡器也被断电。SRAM 和寄存器内容丢失。除备份域（RTC 寄存器、RTC 备份寄存器和备份 SRAM）和待机电路中的寄存器外，SRAM 和寄存器内容都将丢失。

那么我们如何进入待机模式呢？其实很简单，只要按图 10.1.1 所示的步骤执行就可以了：

待机模式	说明
进入模式	WFI（等待中断）或 WFE（等待事件），且： <ul style="list-style-type: none"> 将 Cortex™-M4F 系统控制寄存器中的 SLEEPDEEP 位置 1 将电源控制寄存器 (PWR_CR) 中的 PDDS 位置 1 将电源控制/状态寄存器 (PWR_CSR) 中的 WUF 位清零 将与所选唤醒源（RTC 闹钟 A、RTC 闹钟 B、RTC 唤醒、RTC 入侵或 RTC 时间戳标志）对应的 RTC 标志清零
退出模式	WKUP 引脚上升沿、RTC 闹钟（闹钟 A 和闹钟 B）、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、NRST 引脚外部复位 和 IWDG 复位。
唤醒延迟	复位阶段。

图 10.1.1 STM32L151 进入及退出待机模式的条件

图 10.1.1 还列出了退出待机模式的操作，从图 10.1.1 可知，我们有多种方式可以退出待机模式，包括：WKUP 引脚的上升沿、RTC 闹钟、RTC 唤醒事件、RTC 入侵事件、RTC 时间戳事件、外部复位 (NRST 引脚)、IWDG 复位等，微控制器从待机模式退出。

从待机模式唤醒后的代码执行等同于复位后的执行(采样启动模式引脚，读取复位向量等)。电源控制/状态寄存器 (PWR_CSR) 将会指示内核由待机状态退出。

在进入待机模式后，除了复位引脚、RTC_AF1 引脚 (PC13)（如果针对入侵、时间戳、RTC 闹钟输出或 RTC 时钟校准输出进行了配置）和 WK_UP (PA0)（如果使能了）等引脚外，其他所有 IO 引脚都将处于高阻态。

图 10.1.1 已经清楚的说明了进入待机模式的通用步骤，其中涉及到 2 个寄存器，即电源控制寄存器 (PWR_CR) 和电源控制/状态寄存器 (PWR_CSR)。下面我们介绍一下这两个寄存器：

电源控制寄存器 (PWR_CR)，该寄存器的各位描述如图 10.1.2 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res.	VOS	Reserved				FPDS	DBP	PLS[2:0]			PVDE	CSBF	CWUF	PDDS	LPDS
	rw					rw	rw	rw	rw	rw	rw	rw_w1	rw_w1	rw	rw

位 2 CWUF：将唤醒标志清零 (Clear wakeup flag)
 此位始终读为 0。
 0：无操作
 1：写 1 操作 2 个系统时钟周期后将 WUF 唤醒标志清零

位 1 PDDS：深度睡眠掉电 (Power-down deepsleep)
 此位由软件置 1 和清零。与 LPDS 位结合使用。
 0：器件在 CPU 进入深度睡眠时进入停止模式。调压器状态取决于 LPDS 位。
 1：器件在 CPU 进入深度睡眠时进入待机模式。

图 10.1.2 PWR_CR 寄存器各位描述

该寄存器我们只关心 bit1 和 bit2 这两个位，这里我们通过设置 PWR_CR 的 PDDS 位，使 CPU 进入深度睡眠时进入待机模式，同时我们通过 CWUF 位，清除之前的唤醒位。

电源控制/状态寄存器（PWR_CSR）的各位描述如图 10.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
Res.															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Res	VOS RDY	Reserved				BRE	EWUP	Reserved Res.				BRR	PVDO	SBF	WUF
	r					rw	rw					r	r	r	r

位 8 **EWUP**：使能 WKUP 引脚 (Enable WKUP pin)

此位由软件置 1 和清零。

0：WKUP 引脚用作通用 I/O。WKUP 引脚上的事件不会把器件从待机模式唤醒。

1：WKUP 用于从待机模式唤醒器件并被强制配置成输入下拉（WKUP 引脚出现上升沿时从待机模式唤醒系统）。

注意：此位通过系统复位进行复位。

位 0 **WUF**：唤醒标志 (Wakeup flag)

此位由硬件置 1，清零则只能通过 POR/PDR（上电复位/掉电复位）或将 PWR_CR 寄存器中的 CWUF 位置 1 来实现。

0：未发生唤醒事件

1：收到唤醒事件，可能来自 WKUP 引脚、RTC 闹钟（闹钟 A 和闹钟 B）、RTC 入侵事件、RTC 时间戳事件或 RTC 唤醒事件。

注意：如果使能 WKUP 引脚（将 EWUP 位置 1）时 WKUP 引脚已为高电平，系统将检测到另一唤醒事件。

图 10.1.3 PWR_CSR 寄存器各位描述

这里，我们通过设置 PWR_CSR 的 EWUP 位，来使能 WKUP 引脚用于待机模式唤醒。我们还可以从 WUF 来检查是否发生了唤醒事件，不过本章我们并没有用到。

对于使能了 RTC 闹钟中断或 RTC 周期性唤醒等中断的时候，进入待机模式前，必须按如下操作处理：

- 1，禁止 RTC 中断（ALRAIE、ALRBIE、WUTIE、TAMPIE 和 TSIE 等）
- 2，清零对应中断标志位。
- 3，清除 PWR 唤醒(WUF)标志（通过设置 PWR_CR 的 CWUF 位实现）
- 4，重新使能 RTC 对应中断。
- 5，进入低功耗模式。

在有用到 RTC 相关中断的时候，必须按以上步骤执行之后，才可以进入待机模式，这个大家一定要注意，否则可能无法唤醒。

通过以上介绍，我们了解了进入待机模式的方法，以及设置 KEY2 引脚用于把 STM32L1 从待机模式唤醒的方法。低功耗相关操作函数和定义在 HAL 库文件 stm32l1xx_hal_pwr.c 和头文件 stm32l1xx_hal_pwr.h 中。具体步骤如下：

1) 使能 PWR 时钟。

因为要配置 PWR 寄存器，所以必须先使能 PWR 时钟。

在 HAL 库中，使能 PWR 时钟的方法是：

```
HAL_RCC_PWR_CLK_ENABLE();           //使能 PWR 时钟
```

2) 设置 WK_UP 引脚作为唤醒源。

使能时钟之后再设置 PWR_CSR 的 EWUP 位，使能 WK_UP 用于将 CPU 从待机模式唤醒。在 HAL 库中，设置使能 WK_UP 用于唤醒 CPU 待机模式的函数是：

```
HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN1); //设置 WKUP 用于唤醒
```

3) 设置 SLEEPDEEP 位，设置 PDDS 位，执行 WFI 指令，进入待机模式。

进入待机模式，首先要设置 SLEEPDEEP 位，接着我们通过 PWR_CR 设置 PDDS 位，使得 CPU 进入深度睡眠时进入待机模式，最后执行 WFI 指令开始进入待机模式，并等待 WK_UP 中断的到来。在库函数中，进行上面三个功能进入待机模式是在函数 HAL_PWR_EnterSTANDBYMode 中实现的：

```
void HAL_PWR_EnterSTANDBYMode(void);
```

4) 最后编写 WK_UP 中断服务函数。

因为我们通过 WK_UP 中断（PC13 中断）来唤醒 CPU，所以我们有必要设置一下该中断函数，同时我们也通过该函数里面进入待机模式。关于外部中断服务函数以及中断服务回调函数的使用方法请参考外部中断实验，这里我们就不做过多讲解。

通过以上几个步骤的设置，我们就可以使用 STM32L1 的待机模式了，并且可以通过 KEY2 来唤醒 CPU，我们最终要实现这样一个功能：通过长按（3 秒）KEY2 按键开机，并且通过 LED1 的闪烁指示程序已经开始运行，再次长按该键，则进入待机模式，LED1 关闭，程序停止运行。类似于手机的开关机。

10.2 硬件设计

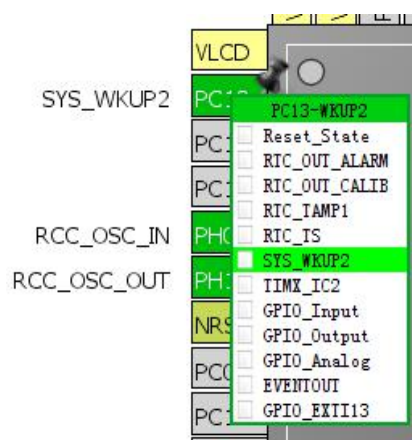
本实验用到的硬件资源有：

- 1) 指示灯 LED1
- 2) KEY_UP 按键

本章，我们使用了 KEY_UP 按键用于唤醒和进入待机模式。然后通过 LED1 来指示程序是否在运行。这几个硬件的连接前面均有介绍。

10.3 STM32CubeMX 配置 WKUP 和软件设计

在 STM32CubeMX 中，我们只需要设置 PC13 为 SYS_WKUP2 模式就可以：



然后我们点击生成工程，打开工程我们添加 wkup.c 文件，内容如下：

```
#include "wkup.h"

//系统进入待机模式
void Sys_Enter_Standby(void)
{
    __HAL_RCC_AHB_FORCE_RESET();           //复位所有 IO 口
    __HAL_RCC_PWR_CLK_ENABLE();           //使能 PWR 时钟

    __HAL_PWR_CLEAR_FLAG(PWR_FLAG_WU);     //清除 wake up 标志
    HAL_PWR_EnableWakeUpPin(PWR_WAKEUP_PIN2); //设置 wake up 标志
    HAL_PWR_EnterSTANDBYMode();             //进入待机模式
}

//检测 wake up 脚的信号
unsigned char Check_WKUP(void)
{
    unsigned char t=0; //记录按下的时间
    OP_LED1();
    HAL_Delay(500);
    while(1)
    {
        if(WKUP_KD)
        {
            t++;           //已经按下了
            HAL_Delay(30);
            if(t>=100)      //按下超过 3 秒钟
            {
                OP_LED1();
                return 1;   //按下 3 秒以上了
            }
        }
        else
        {
            CL_LED1();
            return 0; //按下不足 3 秒
        }
    }
}

//外部中断线 0 中断服务函数
void EXTI15_10_IRQHandler(void)
{
    HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);
}
```

```
//中断线 0 中断处理过程
//此函数会被 HAL_GPIO_EXTI_IRQHandler()调用
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
    if(GPIO_Pin==GPIO_PIN_13)//PA0
    {
        if(Check_WKUP())//关机
        {
            Sys_Enter_Standby();//进入待机模式
        }
    }
}

//PA0 WKUP 初始化
void WKUP_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    __HAL_RCC_GPIOC_CLK_ENABLE();           //开启 GPIOA 时钟

    GPIO_InitStructure.Pin=GPIO_PIN_13;      //PC0
    GPIO_InitStructure.Mode=GPIO_MODE_IT_RISING;
    GPIO_InitStructure.Pull=GPIO_NOPULL;
    GPIO_InitStructure.Speed=GPIO_SPEED_HIGH; //快速
    HAL_GPIO_Init(GPIOC,&GPIO_InitStructure);

    //检查是否开机正常
    if(Check_WKUP()==0)
    {
        Sys_Enter_Standby();//不是开机，进入待机模式
    }
    HAL_NVIC_SetPriority(EXTI15_10_IRQn, 2, 1);
    HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
}
```

该部分代码比较简单，我们在这里说明三点：

1，在 void Sys_Enter_Standby(void) 函数里面，我们要在进入待机模式前把所有开启的外设全部关闭，我们这里仅仅复位了所有的 IO 口，使得 IO 口全部为浮空输入。其他外设（比如 ADC 等），大家根据自己所开启的情况进行一一关闭就可，这样才能达到最低功耗！然后我们调用 HAL_RCC_PWR_CLK_ENABLE() 来使能 PWR 时钟，调用函数 HAL_PWR_EnableWakeUpPin() 用来设置 WK_UP 引脚作为唤醒源。最后调用 HAL_PWR_EnterSTANDBYMode() 函数进入待机模式。

2，在 void WKUP_Init(void) 函数里面，我们首先要使能 GPIOA 时钟，然后对 GPIOA 初始化为下拉输入，上升沿触发中断，同时初始化 NVIC 中断优先级。这上面的步骤实际上跟我们之前的外部中断实验知识是一样的，所以不理解的地方大家可以翻到外部中断实验章节看看。接下来程序通过判断 WK_UP 是否按下了 3 秒钟，来决定要不要开机，如果没有按下 3 秒钟，程序直接就进入了待机模式。所以在下载完代码的时候，是看不到任何反应的。我们**必须先按 WK_UP 按键 3 秒开机**，才能看到 LED1 闪烁。

3, 外部中断回调函数 HAL_GPIO_EXTI_Callback 内, 我们通过调用函数 Check_WKUP() 来判断 WK_UP 按下的时间长短, 来决定是否进入待机模式, 如果按下时间超过 3 秒, 则进入待机, 否则退出中断。

wkup.h 部分代码比较简单, 我们就不多说了。最后我们看看 main 函数内容如下:

```
Int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    WKUP_Init();
    while(1)
    {
        if(HAL_GPIO_ReadPin(LED1_GPIO_Port,LED1_Pin))
        {
            CL_LED1();
        }
        else
        {
            OP_LED1();
        }
        HAL_Delay(1000);
    }
}
```

这里我们先初始化 LED 和 WK_UP 按键 (通过 WKUP_Init() 函数初始化), 如果检测到有长按 WK_UP 按键 3 秒以上则开机, 如果没有长按, 则在 WKUP_Init 里面, 调用 Sys_Enter_Standby 函数, 直接进入待机模式了。

开机后, 在死循环里面等待 WK_UP 中断的到来, 在得到中断后, 在中断函数里面判断 WK_UP 按下的时间长短, 来决定是否进入待机模式, 如果按下时间超过 3 秒, 则进入待机, 否则退出中断, 继续执行 main 函数的死循环等待, 同时不停的取反 LED0, 让红灯闪烁。代码部分就介绍到这里, 大家记住**下载代码后, 一定要长按 WK_UP 按键, 来开机, 否则将直接进入待机模式, 无任何现象。**

10.4 下载与测试

在代码编译成功之后, 下载代码到 STM32 开发板上, 此时, 看到开发板 LED1 亮了一下就没有反应了。其实这是正常的, 在程序下载完之后, 开发板检测不到 WK_UP 的持续按下 (3 秒以上), 所以直接进入待机模式, 看起来和没有下载代码一样。此时, 我们长按 WK_UP 按键 3 秒钟左右, 可以看到 LED1 开始闪烁, 然后再长按 WK_UP, LED1 会灭掉, 程序再次进入待机模式。

第十一章 ADC 实验

本章我们将向大家介绍 STM32L151 的 ADC 功能。在本章中,我们将使用 STM32L151 的 ADC0 来采样外部电压值,并在 LCD 模块上显示出来。本章将分为如下几个部分:

- 11.1 STM32L151 ADC 简介
- 11.2 硬件设计
- 11.3 STM32CubeMX 配置 ADC 和软件设计
- 11.4 下载验证

11.1 STM32L1 ADC 简介

STM32L151 的 ADC 是 12 位逐次逼近型的模拟数字转换器。这些通道的 A/D 转换可以单次、连续、扫描或间断模式执行。ADC 的结果可以左对齐或右对齐方式存储在 16 位数据寄存器中。模拟看门狗特性允许应用程序检测输入电压是否超出用户定义的高/低阈值。

STM32L151 将 ADC 的转换分为 2 个通道组:规则通道组和注入通道组。规则通道相当于你正常运行的程序,而注入通道呢,就相当于中断。在你程序正常执行的时候,中断是可以打断你的执行的。同这个类似,注入通道的转换可以打断规则通道的转换,在注入通道被转换完成之后,规则通道才得以继续转换。

通过一个形象的例子可以说明:假如你在家里的院子内放了 5 个温度探头,室内放了 3 个温度探头;你需要时刻监视室外温度即可,但偶尔你想看看室内的温度;因此你可以使用规则通道组循环扫描室外的 5 个探头并显示 AD 转换结果,当你想看室内温度时,通过一个按钮启动注入转换组(3 个室内探头)并暂时显示室内温度,当你放开这个按钮后,系统又会回到规则通道组继续检测室外温度。从系统设计上,测量并显示室内温度的过程中断了测量并显示室外温度的过程,但程序设计上可以在初始化阶段分别设置好不同的转换组,系统运行中不必再变更循环转换的配置,从而达到两个任务互不干扰和快速切换的结果。可以设想一下,如果没有规则组和注入组的划分,当你按下按钮后,需要从新配置 AD 循环扫描的通道,然后在释放按钮后需再次配置 AD 循环扫描的通道。

上面的例子因为速度较慢,不能完全体现这样区分(规则通道组和注入通道组)的好处,但在工业应用领域中有很多检测和监视探头需要较快地处理,这样对 AD 转换的分组将简化事件处理的程序并提高事件处理的速度。STM32L151 其 ADC 的规则通道组最多包含 16 个转换,而注入通道组最多包含 4 个通道。

STM32L151 的 ADC 在单次转换模式下,只执行一次转换,该模式可以通过 ADC_CR2 寄存器的 ADON 位(只适用于规则通道)启动,也可以通过外部触发启动(适用于规则通道和注入通道),这时 CONT 位为 0。

以规则通道为例,一旦所选择的通道转换完成,转换结果将被存在 ADC_DR 寄存器中,EOC(转换结束)标志将被置位,如果设置了 EOCIE,则会产生中断。然后 ADC 将停止,直到下次启动。

接下来,我们介绍一下我们执行规则通道的单次转换,需要用到的 ADC 寄存器。第一个要介绍的是 ADC 控制寄存器(ADC_CR1 和 ADC_CR2)。ADC_CR1 的各位描述如图 11.1.1 所示:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	
Reserved					OVRIE	RES[1:0]		AWDEN	JAWDEN	Reserved					PDI	PDD
					rw	rw	rw	rw	rw						rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
DISCNUM[2:0]			JDISCEN	DISCEN	JAUTO	AWDSGL	SCAN	JEOCIE	AWDIE	EOCIE	AWDCH[4:0]					
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	

图 11.1.1 ADC_CR1 寄存器各位描述

这里我们不再详细介绍每个位，而是抽出几个我们本章要用到的位进行针对性的介绍。ADC_CR1 的 SCAN 位，该位用于设置扫描模式，由软件设置和清除，如果设置为 1，则使用扫描模式，如果为 0，则关闭扫描模式。在扫描模式下，由 ADC_SQRx 或 ADC_JSQRx 寄存器选中的通道被转换。如果设置了 EOCIE 或 JEOCIE，只在最后一个通道转换完毕后才会产生 EOC 或 JEOC 中断。

ADC_CR1[25:24]用于设置 ADC 的分辨率，详细的对应关系如图 11.1.2 所示：

Bits 25:24 RES[1:0]: Resolution
 These bits are written by software to select the resolution
 00: 12-bit ($T_{CONV} = 12 \text{ ADCCLK cycles}$)
 01: 10-bit ($T_{CONV} = 11 \text{ ADCCLK cycles}$)
 10: 8-bit ($T_{CONV} = 9 \text{ ADCCLK cycles}$)
 11: 6-bit ($T_{CONV} = 7 \text{ ADCCLK cycles}$)
 This bit must be written only when ADON=0.

图 11.1.2 ADC 分辨率选择

本章我们使用 12 位分辨率，所以设置这两个位为 0 就可以了。接着我们介绍 ADC_CR2，该寄存器的各位描述如图 11.1.3 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Res.	SWSTART	EXTEN		EXTSEL[3:0]				Res.	JSWSTART	JEXTEN		JEXTSEL[3:0]			
	r/w	r/w	r/w	r/w	r/w	r/w	r/w		r/w	r/w	r/w	r/w	r/w	r/w	r/w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved				ALIGN	EOCS	DDS	DMA	Res.	DELS			Res.	ADC_CFG	CONT	ADON
				r/w	r/w	r/w	r/w		r/w	r/w	r/w		r/w	r/w	r/w

图 11.1.3 ADC_CR2 寄存器各位描述

该寄存器我们也只针对性的介绍一些位：ADON 位用于开关 AD 转换器。而 CONT 位用于设置是否进行连续转换，我们使用单次转换，所以 CONT 位必须为 0。ALIGN 用于设置数据对齐，我们使用右对齐，该位设置为 0。

EXTEN[1:0]用于规则通道的外部触发使能设置，详细的设置关系如图 11.1.4 所示：

位 29:28 EXTEN: 规则通道的外部触发使能 (External trigger enable for regular channels)
 通过软件将这些位置 1 和清零可选择外部触发极性和使能规则组的触发。
 00: 禁止触发检测
 01: 上升沿上的触发检测
 10: 下降沿上的触发检测
 11: 上升沿和下降沿上的触发检测

图 11.1.4 ADC 规则通道外部触发使能设置

我们这里使用的是软件触发，即不使用外部触发，所以设置这 2 个位为 0 即可。ADC_CR2 的 SWSTART 位用于开始规则通道的转换，我们每次转换（单次转换模式下）都需要向该位写 1。

第二个要介绍的是 ADC 通用控制寄存器（ADC_CCR），该寄存器各位描述如图 11.1.5 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								TSVREFE	VBATE	Reserved				ADCPRE	
								rw	rw					rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DMA[1:0]		DDS	Res.	DELAY[3:0]				Reserved			MULTI[4:0]				
rw	rw	rw		rw	rw	rw	rw				rw	rw	rw	rw	rw

图 11.1.5 ADC_CCR 寄存器各位描述

该寄存器我们也只针对性的介绍一些位：TSVREFE 位是内部温度传感器和 Vrefint 通道使能位，内部温度传感器我们将在下一章介绍，这里我们直接设置为 0。ADCPRE[1:0]用于设置 ADC 输入时钟分频，00~11 分别对应 2/4/6/8 分频，STM32F429 的 ADC 最大工作频率是 36Mhz，而 ADC 时钟（ADCCLK）来自 APB2，APB2 频率一般是 90Mhz，我们设置 ADCPRE=01，即 4 分频，这样得到 ADCCLK 频率为 22.5Mhz。MULTI[4:0]用于多重 ADC 模式选择，详细的设置关系如图 11.1.6 所示：

位 4:0 MULTI[4:0]: 多重 ADC 模式选择 (Multi ADC mode selection)

通过软件写入这些位可选择操作模式。

所有 ADC 均独立：

00000：独立模式

00001 到 01001：双重模式，ADC1 和 ADC2 一起工作，ADC3 独立

00001：规则同时 + 注入同时组合模式

00010：规则同时 + 交替触发组合模式

00011：Reserved

00101：仅注入同时模式

00110：仅规则同时模式

仅交错模式

01001：仅交替触发模式

10001 到 11001：三重模式：ADC1、ADC2 和 ADC3 一起工作

10001：规则同时 + 注入同时组合模式

10010：规则同时 + 交替触发组合模式

10011：Reserved

10101：仅注入同时模式

10110：仅规则同时模式

仅交错模式

11001：仅交替触发模式

其它所有组合均需保留且不允许编程

图 11.1.6 多重 ADC 模式选择设置

本章我们仅用了 ADC1（独立模式）并没用到多重 ADC 模式，所以设置这 5 个位为 0 即可。

第三个要介绍的是 ADC 采样时间寄存器（ADC_SMPR0），这个寄存器用于设置通道 0~18 的采样时间，每个通道占用 3 个位。ADC_SMPR0 的各位描述如图 11.1.7 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										SMP31[2:0]			SMP30[2:0]		
										rw	rw	rw	rw	rw	rw

位 29:0 SMPx[2:0]:通道 X 采样时间选择

通过软件写入这些位可分别为各个通道选择采样时间。在采样周期期间，通道选择位必须保持不变

注意：000: 4 个周期 100: 48 个周期
 001: 9 个周期 101: 96 个周期
 010: 16 个周期 110: 192 个周期
 100: 24 个周期 111: 384 个周期

图 11.1.8 ADC_SMPR2 寄存器各位描述

对于每个要转换的通道，采样时间建议尽量长一点，以获得较高的准确度，但是这样会降低 ADC 的转换速率。ADC 的转换时间可以由以下公式计算：

$$T_{covn} = \text{采样时间} + 12 \text{ 个周期}$$

其中：Tcovn 为总转换时间，采样时间是根据每个通道的 SMP 位的设置来决定的。例如，当 ADCCLK=22.5Mhz 的时候，并设置 3 个周期的采样时间，则得到：Tcovn=3+12=15 个周期=0.67us。

第四个要介绍的是 ADC 规则序列寄存器 (ADC_SQR1~3)，该寄存器总共有 3 个，这几个寄存器的功能都差不多，这里我们仅介绍一下 ADC_SQR1，该寄存器的各位描述如图 11.1.9 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved								L[3:0]				SQ16[4:1]			
								rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SQ16_0		SQ15[4:0]				SQ14[4:0]				SQ13[4:0]					
rw	rw	rw	rw	rw	rw	rw	rw					rw	rw	rw	rw

位 31:24 保留，必须保持复位值。

位 23:20 **L[3:0]**: 规则通道序列长度 (Regular channel sequence length)

通过软件写入这些位可定义规则通道转换序列中的转换总数。

0000: 1 次转换
 0001: 2 次转换
 ...
 1111: 16 次转换

位 19:15 **SQ16[4:0]**: 规则序列中的第十六次转换 (16th conversion in regular sequence)

通过软件写入这些位，并将通道编号 (0..18) 分配为转换序列中的第十六次转换。

位 14:10 **SQ15[4:0]**: 规则序列中的第十五次转换 (15th conversion in regular sequence)

位 9:5 **SQ14[4:0]**: 规则序列中的第十四次转换 (14th conversion in regular sequence)

位 4:0 **SQ13[4:0]**: 规则序列中的第十三次转换 (13th conversion in regular sequence)

图 11.1.9 ADC_SQR1 寄存器各位描述

L[3: 0]用于存储规则序列的长度，我们这里只用了 1 个，所以设置这几个位的值为 0。其他的 SQ13~16 则存储了规则序列中第 13~16 个通道的编号（0~18）。另外两个规则序列寄存器同 ADC_SQR1 大同小异，我们这里就不再介绍了，要说明一点的是：我们选择的是单次转换，所以只有一个通道在规则序列里面，这个序列就是 SQ1，至于 SQ1 里面哪个通道，完全由用户自己设置，通过 ADC_SQR3 的最低 5 位（也就是 SQ1）设置。

第五个要介绍的是 ADC 规则数据寄存器(ADC_DR)。规则序列中的 AD 转化结果都将被存在这个寄存器里面，而注入通道的转换结果被保存在 ADC_JDRx 里面。ADC_DR 的各位描述如图 11.1.10:

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位 31:16 保留，必须保持复位值。

位 15:0 DATA[15:0]: 规则数据 (Regular data)

这些位为只读。它们包括来自规则通道的转换结果。数据有左对齐和右对齐两种方式。

图 11.1.10 ADC_ JDRx 寄存器各位描述

这里要提醒一点的是，该寄存器的数据可以通过 ADC_CR2 的 ALIGN 位设置左对齐还是右对齐。在读取数据的时候要注意。

最后一个要介绍的 ADC 寄存器为 ADC 状态寄存器 (ADC_SR)，该寄存器保存了 ADC 转换时的各种状态。该寄存器的各位描述如图 11.1.11 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
Reserved															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Reserved										OVR	STRT	JSTRT	JEOC	EOC	AWD
										rc_w0	rc_w0	rc_w0	rc_w0	rc_w0	rc_w0

图 11.1.11 ADC_ SR 寄存器各位描述

这里我们仅介绍将要用到的是 EOC 位，我们通过判断该位来决定是否此次规则通道的 AD 转换已经完成，如果该位为 1，则表示转换完成了，就可以从 ADC_DR 中读取转换结果，否则等待转换完成。

通过以上介绍，我们了解了 STM32L1 的单次转换模式下的相关设置，接下来我们介绍使用库函数来设置 ADC0 来进行 AD 转换的步骤，这里需要说明一下，使用到的库函数分布在 stm32l1xx_hal_adc.c/stm32l1xx_hal_adc_ex.c 文件和 stm32l1xx_hal_adc.h/stm32l1xx_hal_adc_ex.h 文件中。下面讲解其详细设置步骤：

1) 开启 PA 口时钟和 ADC 时钟，设置 PA0 为模拟输入。

STM32L151R8T6 的 ADC0 在 PA0 上，所以，我们先要使能 PORTA 的时钟，然后设置 PA0 为模拟输入。同时我们要把 PA0 复用为 ADC，所以我们要使能 ADC 时钟。

这里特别要提醒，对于 IO 口复用为 ADC 我们要设置模式为模拟输入，而不是复用功能。

使能 GPIOA 时钟和 ADC1 时钟都很简单，具体方法为：

```
__HAL_RCC_ADC1_CLK_ENABLE();
__HAL_RCC_GPIOA_CLK_ENABLE();
```

初始化 GPIOA0 为模拟输入，方法也多次讲解，关键代码为：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.Pin = GPIO_PIN_0;
GPIO_InitStructure.Mode = GPIO_MODE_ANALOG;
GPIO_InitStructure.Pull = GPIO_NOPULL;
HAL_GPIO_Init(GPIOA, &GPIO_InitStructure);
```

2) 初始化ADC，设置ADC时钟分频系数，分辨率，模式，扫描方式，对齐方式等信息。

在 HAL 库中，初始化 ADC 是通过函数 HAL_ADC_Init 来实现的，该函数声明为：

```
HAL_StatusTypeDef HAL_ADC_Init(ADC_HandleTypeDef*hadc);
```

该函数只有一个入口参数 hadc，为 ADC_HandleTypeDef 结构体指针类型，结构体定义为：

```
typedef struct
{
    ADC_TypeDef                *Instance;           //ADC1/ADC2/ADC3
    ADC_InitTypeDef            Init;               //初始化结构体变量
    __IO uint32_t              NbrOfConversionRank; //当前转换序列
    DMA_HandleTypeDef          *DMA_Handle;        //DMA 方式使用
    HAL_LockTypeDef            Lock;
    __IO uint32_t              State;
    __IO uint32_t              ErrorCode;
}ADC_HandleTypeDef;
```

该结构体定义和其他外设比较类似，我们着重看第二个成员变量 Init 含义，它是结构体 ADC_InitTypeDef 类型，结构体 ADC_InitTypeDef 定义为：

```
typedef struct
{
    uint32_t ClockPrescaler; //分频系数 2/4/6/8 分频 ADC_CLOCK_SYNC_PCLK_DIV4
    uint32_t Resolution;    //分辨率 12/10/8/6 位: ADC_RESOLUTION_12B
    uint32_t DataAlign;     //对齐方式: 左对齐还是右对齐: ADC_DATAALIGN_RIGHT
    uint32_t ScanConvMode;  //扫描模式 DISABLE
    uint32_t EOCSelection;  //EOC 标志是否设置 DISABLE
    uint32_t ContinuousConvMode; //开启连续转换模式或者单次转换模式 DISABLE
    uint32_t DMAContinuousRequests; //开启 DMA 请求连续模式或者单独模式 DISABLE
    uint32_t NbrOfConversion; //规则序列中有多少个转换
    uint32_t DiscontinuousConvMode; //不连续采样 DISABLE
    uint32_t NbrOfDiscConversion; //不连续采样通道数 0
    uint32_t ExternalTrigConv; //外部触发方式 ADC_SOFTWARE_START
    uint32_t ExternalTrigConvEdge; //外部触发边沿
}ADC_InitTypeDef;
```


我们直接把每个成员变量含义注释在结构体定义的后面，请大家仔细阅读上面注释。这里我们需要说明一下，和其他外设一样，HAL 库同样提供了 ADC 的 MSP 初始化函数，一般情况下，时钟使能和 GPIO 初始化都会放在 MSP 初始化函数中。函数声明为：

```
void HAL_ADC_MspInit(ADC_HandleTypeDef*hadc);
```

4) 开启 AD 转换器。

在设置完了以上信息后，我们就开启 AD 转换器了（通过 ADC_CR2 寄存器控制）

```
HAL_ADC_Start(&hadc); //开启 ADC
```

5) 配置通道，读取通道 ADC 值。

在上面的步骤完成后，ADC 就算准备好了。接下来我们要做的就是设置规则序列 1 里面的通道，然后启动 ADC 转换。在转换结束后，读取转换结果值就是了。

设置规则序列通道以及采样周期的函数是：

```
HAL_StatusTypeDef HAL_ADC_ConfigChannel(ADC_HandleTypeDef*hadc,  
                                          ADC_ChannelConfTypeDef*sConfig);
```

该函数有两个入口参数，第一个就不用多说了，接下来我们看第二个入口参数 sConfig，它是 ADC_ChannelConfTypeDef 结构体指针类型，结构体定义如下：

```
typedef struct  
{  
    uint32_t Channel; //ADC 通道  
    uint32_t Rank; //规则通道中的第几个转换  
    uint32_t SamplingTime; //采样时间  
    uint32_t Offset; //备用，暂未用到  
}ADC_ChannelConfTypeDef;
```

该结构体有四个成员变量，对于 STM32F4 只用到前面三个。Channel 用来设置 ADC 通道，Rank 用来设置要配置的通道是规则序列中的第几个转换，SamplingTime 用来设置采样时间。使用实例为：

```
sConfig.Channel=channel;  
  
sConfig.Rank =1;  
  
sConfig.SamplingTime =ADC_SAMPLETIME_48CYCLES;  
  
if (HAL_ADC_ConfigChannel(&hadc, &sConfig) !=HAL_OK)  
{  
    _Error_Handler(__FILE __, __LINE__);  
}
```


配置好通道并且使能 ADC 后接下来就是读取 ADC 值这里我们采取的是查询方式读取，所以我们还要等待上一次转换结束。此过程 HAL 库提供了专用函数 HAL_ADC_PollForConversion，函数定义为：

```
sConfig.Channe=channel;
sConfig.Rank =1;
sConfig.SamplingTime =ADC_SAMPLETIME_48CYCLES;
if (HAL_ADC_ConfigChannel(&hadc, &sConfig) !=HAL_OK)
{
    _Error_Handler(__FILE __, __LINE__);
}
```

配置好通道并且使能 ADC 后接下来就是读取 ADC 值这里我们采取的是查询方式读取，所以我们还要等待上一次转换结束。此过程 HAL 库提供了专用函数 HAL_ADC_PollForConversion，函数定义为：

```
HAL_StatusTypeDef HAL_ADC_PollForConversion(ADC_HandleTypeDef*hadc,uint32_tTimeout);
```

等待上一次转换结束之后，接下来就是读取 ADC 值，函数为：

```
uint32_tHAL_ADC_GetValue(ADC_HandleTypeDef*hadc);
```

通过以上几个步骤的设置，我们就能正常的使用 STM32L151 的 ADC0 来执行 AD 转换操作了。

11.2 硬件设计

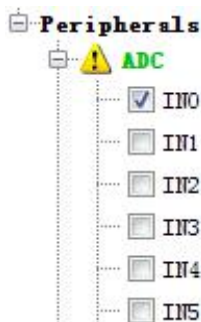
本实验用到的硬件资源有：

- 1) 电位器
- 2) LCD 模块

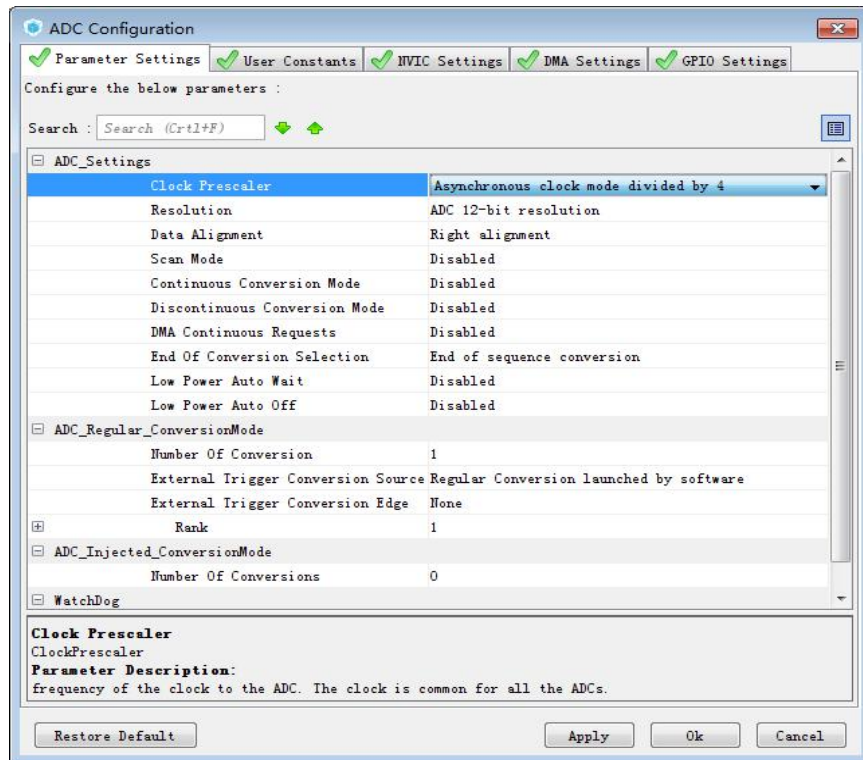
11.3 STM32CubeMX 配置 ADC 和软件设计

使用 STM32CubeMX 配置 ADC 的一般步骤为：

- ① 进入 Pinout->ADC 配置栏，启动 ADC0 基本参数：



②进入 Configuration->ADC 配置栏，配置基本参数：



注：关于 LCD 显示屏的 SPI 配置和之前一样，这里就不再重复了。

打开生成的工程打开 adc.c，代码如下：

```
#include "adc.h"
#include "gpio.h"

unsigned int ADC_Dat=0;
ADC_HandleTypeDef hadc;

void MX_ADC_Init(void)
{
    hadc.Instance = ADC1;
    hadc.Init.ClockPrescaler = ADC_CLOCK_ASYNC_DIV4;
    hadc.Init.Resolution = ADC_RESOLUTION_12B;
    hadc.Init.DataAlign = ADC_DATAALIGN_RIGHT;
    hadc.Init.ScanConvMode = ADC_SCAN_DISABLE;
    hadc.Init.EOCSelection = ADC_EOC_SEQ_CONV;
    hadc.Init.LowPowerAutoWait = ADC_AUTOWAIT_DISABLE;
    hadc.Init.LowPowerAutoPowerOff = ADC_AUTOPOWEROFF_DISABLE;
    hadc.Init.ChannelsBank = ADC_CHANNELS_BANK_A;
```

```
hadc.Init.ContinuousConvMode = DISABLE;
hadc.Init.NbrOfConversion = 1;
hadc.Init.DiscontinuousConvMode = DISABLE;
hadc.Init.ExternalTrigConv = ADC_SOFTWARE_START;
hadc.Init.ExternalTrigConvEdge = ADC_EXTERNALTRIGCONVEDGE_NONE;
hadc.Init.DMAContinuousRequests = DISABLE;
if (HAL_ADC_Init(&hadc) != HAL_OK)
{
    _Error_Handler(__FILE__, __LINE__);
}
}

void HAL_ADC_MspInit(ADC_HandleTypeDef* adcHandle)
{
    GPIO_InitTypeDef GPIO_InitStruct;
    if(adcHandle->Instance==ADC1)
    {
        __HAL_RCC_ADC1_CLK_ENABLE();

        GPIO_InitStruct.Pin = GPIO_PIN_0;
        GPIO_InitStruct.Mode = GPIO_MODE_ANALOG;
        GPIO_InitStruct.Pull = GPIO_NOPULL;
        HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
    }
}
```

```
unsigned int Get_ADC_data(unsigned int channel)
{
    ADC_ChannelConfTypeDef sConfig;

    sConfig.Channel = ADC_CHANNEL_0;
    sConfig.Rank = ADC_REGULAR_RANK_1;
    sConfig.SamplingTime = ADC_SAMPLETIME_4CYCLES;
    if (HAL_ADC_ConfigChannel(&hadc, &sConfig) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    HAL_ADC_Start(&hadc); //开启 ADC
    HAL_ADC_PollForConversion(&hadc,10); //查询方式读取 ADC 的值
    return (unsigned int)HAL_ADC_GetValue(&hadc);
}

unsigned int Get_ADC_Average(unsigned int ch,unsigned char times)
{
    unsigned int temp=0;
    unsigned char t;
    for(t=0;t<times;t++)
    {
        temp+=Get_ADC_data(ch);
        HAL_Delay(5);
    }
    return temp/times;
}
```

此部分代码就 4 个函数，MY_Adc_Init 函数调用函数 HAL_ADC_Init 初始化 ADC 相关参数。第二个函数 HAL_ADC_MspInit 是 MSP 初始化回调函数，用来使能时钟和初始化 IO 口。第三个函数 Get_ADC_data，用于读取某个通道的 ADC 值，最后一个函数 Get_Adc_Average，用于多次获取 ADC 值，取平均，用来提高准确度。

除此之外，还需要编写显示函数 display.c。

```
#include "display.h"
#include "lcd.h"
#include "gui.h"

extern unsigned int ADC_Dat;
void display_init(void)
{
    Show_Str(0,10,BLUE,BLACK,"this is a demo",16,1);
}
```

```
void display_AD(void)
{
    Show_Str(0,20,BLUE,BLACK,"ADC result:",16,1);
    POINT_COLOR=BLUE;
    LCD_Fill(90,20,128,40,BLACK);
    LCD_ShowNum(90,20,ADC_Dat,4,16);
}
```

头文件 adc.h 代码比较简单，主要是函数申明。接下来我们看看 main 函数内容：

```
Int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_ADC_Init();
    MX_SPI2_Init();
    LCD_Init();
    LCD_Clear(BLACK);
    while(1)
    {
        ADC_Dat=Get_ADC_data(ADC_CHANNEL_0);display_AD();
        HAL_Delay(3000);
    }
}
```

此部分代码，我们在 TFTLCD 模块上显示一些提示信息后，将每隔 3s 读取一次 ADC 通道 0 的值并显示读到的 ADC 值数字量。

11.4 下载验证

在代码编译成功之后，我们通过下载代码到 STM32 开发板上，可以看到 LCD 显示如图所示：



第十二章 IIC 实验

本章我们将向大家介绍如何使用 STM32L151 的 IIC 接口实现和 24C256 之间的双向通信，来实现 24C256 的读写，并将结果显示在 LCD 模块上。本章分为如下几个部分：

- 12.1 IIC 简介
- 12.2 硬件设计
- 12.3 STM32CubeMX 配置 IIC 和软件设计
- 12.4 下载验证

12.1 IIC 简介

IIC(Inter-IntegratedCircuit)总线是一种由 PHILIPS 公司开发的两线式串行总线，用于连接微控制器及其外围设备。它是由数据线 SDA 和时钟 SCL 构成的串行总线，可发送和接收数据。在 CPU 与被控 IC 之间、IC 与 IC 之间进行双向传送，高速 IIC 总线一般可达 400kbps 以上。

I2C 总线在传送数据过程中共有三种类型信号，它们分别是：开始信号、结束信号和应答信号。

开始信号：SCL 为高电平时，SDA 由高电平向低电平跳变，开始传送数据。

结束信号：SCL 为高电平时，SDA 由低电平向高电平跳变，结束传送数据。

应答信号：接收数据的 IC 在接收到 8bit 数据后，向发送数据的 IC 发出特定的低电平脉冲，表示已收到数据。CPU 向受控单元发出一个信号后，等待受控单元发出一个应答信号，CPU 接收到应答信号后，根据实际情况作出是否继续传递信号的判断。若未收到应答信号，由判断为受控单元出现故障。

这些信号中，起始信号是必需的，结束信号和应答信号，都可以不要。IIC 总线时序图如图 12.1.1 所示：

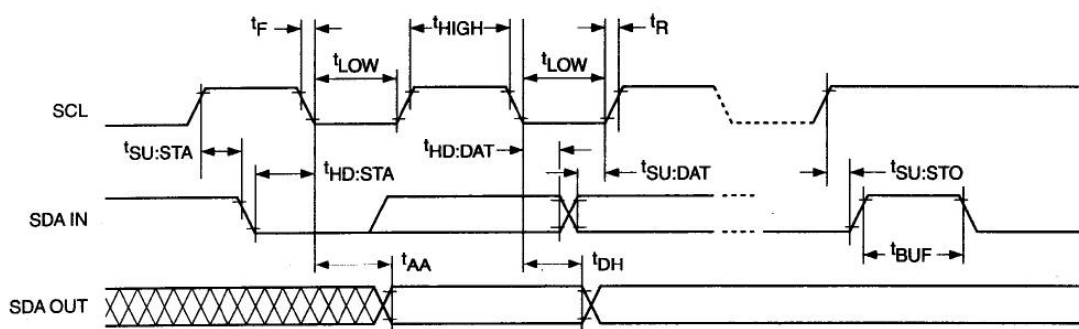


图 12.1.1 IIC 总线时序图

STM32L1 开发板板载的 EEPROM 芯片型号为 24C256。该芯片的总容量是 32768 个字节，该芯片通过 IIC 总线与外部连接，我们本章就通过 STM32L1 来实现 24C256 的读写。

本章实验功能简介：开机的时候先检测 24C256 是否存在，然后在主循环里面检测两个按键，其中 1 个按键(KEY1)用来执行写入 24C256 的操作，另外一个按键(KEY2)用来执行读出操作，在 TFTLCD 模块上显示相关信息。

12.2 硬件设计

本章需要用到的硬件资源有：

- 1) KEY1 和 KEY2 按键
- 2) 串口 3
- 3) LCD 模块
- 4) 24C256

前面 3 部分的资源，我们前面已经介绍了，请大家参考相关章节。这里只介绍 24C256 与 STM32L151 的连接，24C256 的 SCL 和 SDA 分别连在 STM32L151 的 PB6 和 PB7 上的，连接关系如图 12.2.1 所示：

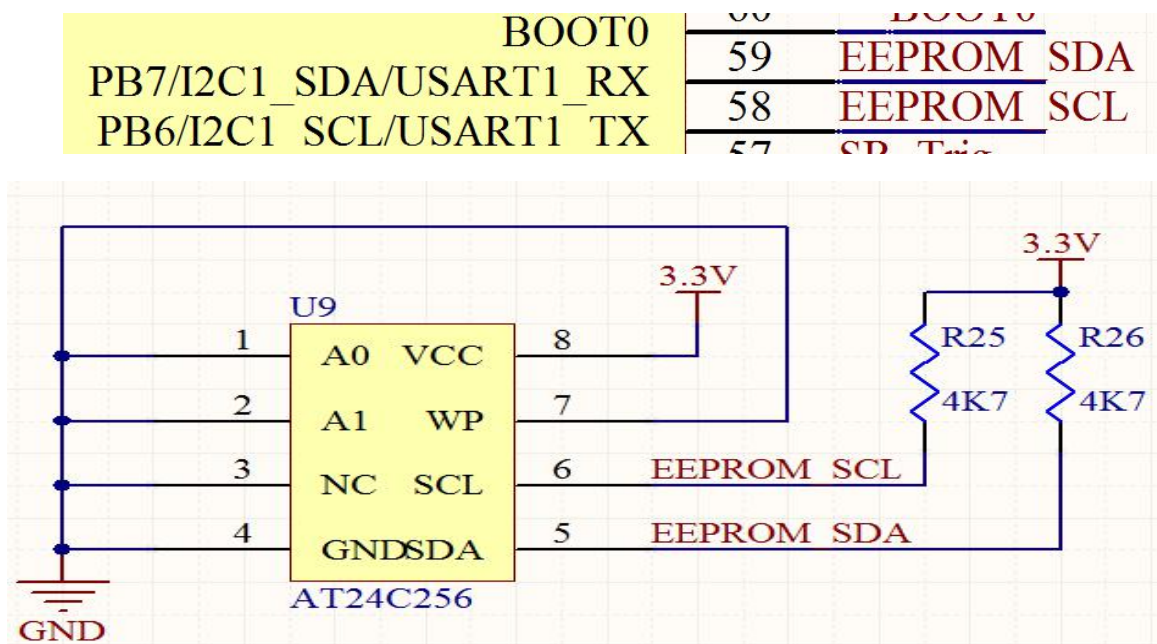


图 12.2.1 STM32L151 与 24C256 连接图

12.3 STM32CubeMX 配置 IIC 和软件设计

在配置 IIC 之前，配置串口 USART3 时直接开启时，默认使用的引脚时 PB10、PB11。我们要使用 PC10、PC11 引脚。在开启串口功能时，先设置引脚再开启。如下图：

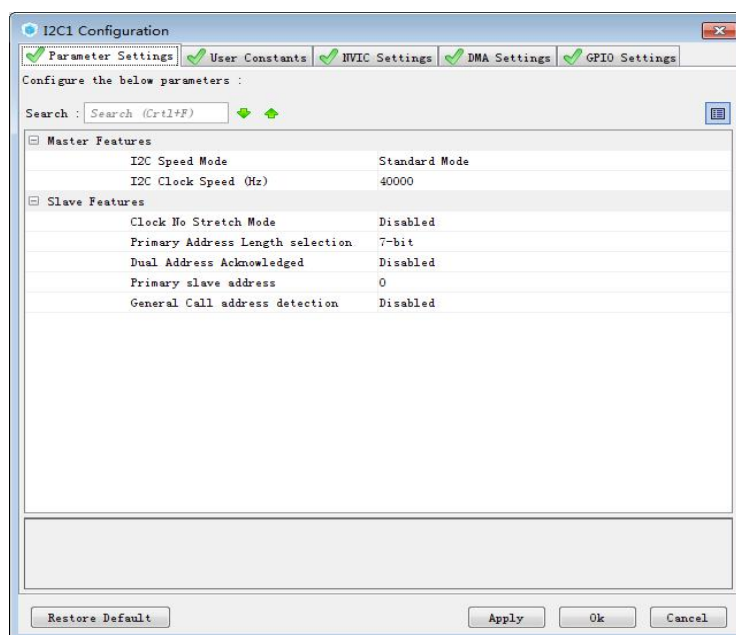


串口的参数配置和之前的串口实验一样。SPI 的参数配置也和之前一样。使用 CubeMX 配置 IIC 的一般步骤为：

① 进入 Pinout->IIC 配置栏，启动 IIC1 基本参数：



②进入 Configuration->IIC1 配置栏，配置基本参数：



生成工程后，我们首先打开 usart.c 文件，添加回调函数。变量的定义大家自己动手写一下。

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance==USART3)
    {
        switch(USART3_Count)
        {
            case 0:
                if(aRxBuffer[0]!=0x0D)
                {
                    USART3_Receviebuf[USART3_len]=aRxBuffer[0];
                    USART3_len++;
                }
                else
                {
                    USART3_Count=1;
                }
                break;
        }
    }
}
```

```
        case 1:
            if(aRxBuffer[0]!=0x0A)
            {
                EEPROM_len=0;
            }
            else
            {
                EEPROM_len=USART3_len;
                USART3_Recevie_flag=1;
            }
            USART3_len=0;
            USART3_Count=0;
            break;
        default:
            break;
    }
    HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
}

void USART3_SendData(unsigned char ch)
{
    HAL_UART_Transmit_IT(&huart3,&ch,1);
    while(__HAL_UART_GET_FLAG(&huart3,UART_FLAG_TC)!=SET);
}

void USART3_SendStr(unsigned char * str)
{
    while(*str)
        USART3_SendData(*str++);
}
```

不要忘记在 `MX_USART3_UART_Init()` 函数中添加代码

```
HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
```

我们打开文件 `i2c.c` 代码如下：

```
I2C_HandleTypeDef hi2c1;

void MX_I2C1_Init(void)
{
    hi2c1.Instance = I2C1;
    hi2c1.Init.ClockSpeed = 40000;
    hi2c1.Init.DutyCycle = I2C_DUTYCYCLE_2;
    hi2c1.Init.OwnAddress1 = 0;
    hi2c1.Init.AddressingMode = I2C_ADDRESSINGMODE_7BIT;
    hi2c1.Init.DualAddressMode = I2C_DUALADDRESS_DISABLE;
    hi2c1.Init.OwnAddress2 = 0;
    hi2c1.Init.GeneralCallMode = I2C_GENERALCALL_DISABLE;
    hi2c1.Init.NoStretchMode = I2C_NOSTRETCH_DISABLE;
    if (HAL_I2C_Init(&hi2c1) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
}

void HAL_I2C_MspInit(I2C_HandleTypeDef* i2cHandle)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    if(i2cHandle->Instance==I2C1)
    {
        GPIO_InitStructure.Pin = GPIO_PIN_6|GPIO_PIN_7;
        GPIO_InitStructure.Mode = GPIO_MODE_AF_OD;
        GPIO_InitStructure.Pull = GPIO_PULLUP;
        GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
        GPIO_InitStructure.Alternate = GPIO_AF4_I2C1;
        HAL_GPIO_Init(GPIOB, &GPIO_InitStructure);

        __HAL_RCC_I2C1_CLK_ENABLE();
    }
}
```

在 spi.c 文件中的 MX_SPI2_Init（）函数最后一行添加代码

```
__HAL_SPI_ENABLE(&hspi2);
```

同时我们添加 at24cxx.c 和 at24cxx.h 文件，打开 at24cxx.c 代码如下（用到的代码）：

```
#include "at24cxx.h"

#define I2C_TIMEOUT 100 //IIC 通信失败时的超时值

//读取多个数据
static HAL_StatusTypeDef I2Cx_ReadMultiple(uint8_t Addr,uint8_t Reg,uint16_t MemAddSize,uint8_t
                                           *Buffer,uint8_t Length)
{
    HAL_StatusTypeDef status=HAL_OK;
    status=HAL_I2C_Mem_Read(&hi2c1,Addr,(uint16_t)Reg,MemAddSize,Buffer,Length,I2C_TIMEOUT);
    if(status!=HAL_OK)
    {
        I2Cx_Error(Addr);
    }
    return status;
}

//使用 DMA 模式通过 BUS 在器件寄存器中写入值
Static HAL_StatusTypeDef I2Cx_WriteMultiple (uint8_t Addr,uint16_t Reg, uint16_t MemAddSize,
                                              uint8_t *Buffer,uint8_t Length)
{
    HAL_StatusTypeDef status=HAL_OK;
    status=HAL_I2C_Mem_Write(&hi2c1,Addr,(uint16_t)Reg,MemAddSize,Buffer,Length,I2C_TIMEOUT);
    if(status!=HAL_OK)
    {
        I2Cx_Error(Addr);
    }
    return status;
}

//通过重新初始化 IIC 来管理错误回调
static void I2Cx_Error(uint8_t Addr)
{
    HAL_I2C_DeInit(&hi2c1);
    MX_I2C1_Init();
}

//在 EEPROM 的一个写循环中写多个字节
HAL_StatusTypeDef AT24CXX_PageWrite(uint16_t MemAddress, uint8_t* pBuffer, uint32_t BufferSize)
{
    HAL_StatusTypeDef status=HAL_OK;
    status=I2Cx_WriteMultiple(AT24CXX_ADDRESS,MemAddress,AT24CXX_MEMADD_SIZE,pBuffer,BufferSize);
}
```

```
HAL_Delay(10);
return status;
}

//使用 DMA 通道将数据写入 EEPROM 中
HAL_StatusTypeDef AT24CXX_WriteData(uint16_t MemAddress,uint8_t* pBuffer,uint32_t BufferSize)
{
    uint8_t NumOfPage=0,NumOfSingle=0,Addr=0,count=0;
    HAL_StatusTypeDef err=HAL_OK;

    Addr=MemAddress%AT24CXX_PAGE_SIZE; //写入地址是每页的第几位
    count=AT24CXX_PAGE_SIZE-Addr;      //在开始的一页要写入的个数
    NumOfPage=BufferSize/AT24CXX_PAGE_SIZE; //要写入的页数
    NumOfSingle=BufferSize%AT24CXX_PAGE_SIZE; //不足一页的个数

    //写入地址是页的开始
    if(Addr==0)
    {
        //数据小于一页
        if(NumOfPage==0)
        {
            return (AT24CXX_PageWrite(MemAddress, pBuffer, BufferSize)); //写少于一页的个数
        }
        //数据大于等于一页
        else
        {
            while(NumOfPage--)
            {
                err=AT24CXX_PageWrite(MemAddress,pBuffer,AT24CXX_PAGE_SIZE); //写一页的数据
                if(err!=HAL_OK)
                    return err;
                MemAddress+=AT24CXX_PAGE_SIZE;
                pBuffer+=AT24CXX_PAGE_SIZE;
            }
            if(NumOfSingle!=0) //剩余数据小于一页
            {
                return (AT24CXX_PageWrite(MemAddress,pBuffer,NumOfSingle)); //写少于一页的数据
            }
        }
    }
    //写入的地址不是一页的开始
    else
    {
        //数据小于一页
```



```

if(BufferSize<=count)
{
    return (AT24CXX_PageWrite(MemAddress,pBuffer,NumOfSingle)); //写少于一页的数据
}
//数据大于等于一页
else
{
    BufferSize-=count;
    NumOfPage=BufferSize/AT24CXX_PAGE_SIZE; //重新计算要写入的页数
    NumOfSingle=BufferSize%AT24CXX_PAGE_SIZE; //重新计算不足一页的个数
    err=AT24CXX_PageWrite(MemAddress,pBuffer,count); //将开始的空间写满一页
    if(err!=HAL_OK)
        return err;
    MemAddress+=count;
    pBuffer+=count;
    while(NumOfPage--)
    {
        err=AT24CXX_PageWrite(MemAddress,pBuffer,AT24CXX_PAGE_SIZE); //写一页数据
        if(err!=HAL_OK)
            return err;
        MemAddress+=AT24CXX_PAGE_SIZE;
        pBuffer+=AT24CXX_PAGE_SIZE;
    }
    if(NumOfSingle!=0) //剩余数据小于一页
    {
        return (AT24CXX_PageWrite(MemAddress,pBuffer,NumOfSingle)); //写少于一页的
数据
    }
}
}
return err;
}

//使用 DMA 通道从 EEPROM 中读取数据
HAL_StatusTypeDef AT24CXX_ReadData(uint16_t MemAddress, uint8_t* pBuffer, uint32_t
                                   BufferSize)
{
    return (I2Cx_ReadMultiple(AT24CXX_ADDRESS, MemAddress, AT24CXX_MEMADD_SIZE,
                              pBuffer, BufferSize));
}

```

接下来我们看一下主函数设计，大家自己定义一下全局变量。代码如下：

```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_I2C1_Init();
    MX_SPI2_Init();
    MX_USART3_UART_Init();
    LCD_Init();
    LCD_Clear(BLACK);
    Show_Str(0,10,BLUE,BLACK,"IIC TEST",16,1);
    while (1)
    {
        if(USART3_Recevie_flag)
        {
            USART3_Recevie_flag=0;
            USART3_SendStr("OK!");
        }
        if(KEY1())
        {
            HAL_Delay(20);
            if(KEY1())
            {
                {
                    if(EEPROM_len)
                    {
                        AT24CXX_WriteData(0,USART3_Receviebuf,EEPROM_len);
                        LCD_Clear(BLACK);
                        LCD_Fill(0,30,128,50,BLACK);
                        Show_Str(0,30,BLUE,BLACK,"EEPROM Write OK!",16,1);
                    }
                }
                while(KEY1());
            }
        }
        if(KEY2())
        {
            HAL_Delay(20);
            if(KEY2())
            {
                {
                    if(EEPROM_len)
                    {
                        AT24CXX_ReadData(0,at24cxx_read,EEPROM_len);
                        LCD_Clear(BLACK);
                        LCD_Fill(0,30,128,50,BLACK);
                    }
                }
            }
        }
    }
}
```

```
        Show_Str(0,30,BLUE,BLACK,"EEPROM Read OK!",16,1);  
        POINT_COLOR=BLUE;  
        LCD_Fill(0,80,128,128,BLACK);  
        Show_Str(0,60,BLUE,BLACK,"EEPROM Read Data:",12,1);  
        LCD_ShowString(0,80,16,at24cxx_read,1);  
    }  
}  
}  
}
```

该段代码，我们通过 KEY1 按键来控制 24C256 的写入，通过另外一个按键 KEY2 来控制 24C256 的读取。并在 LCD 模块上面显示相关信息。至此，我们的软件设计部分就结束了。

12.4 下载验证

在代码编译成功之后，我们通过下载代码到 STM32L1 开发板上，先用串口写入数据，通过先按 KEY1 按键写入数据，然后按 KEY2 读取数据，得到如图 12.4.1 所示：

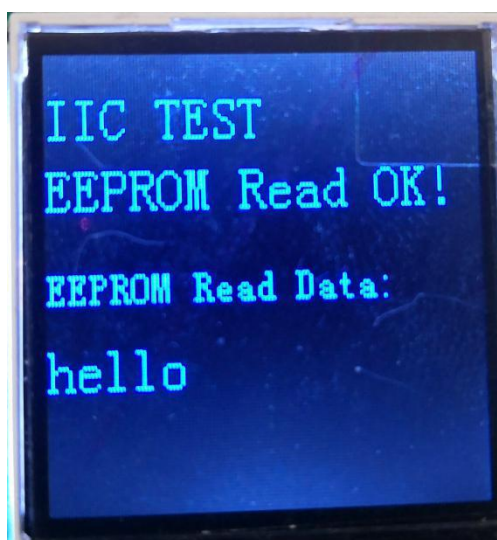


图 12.4.1 IIC 实验程序运行效果图

第十三章温湿度传感器实验

目前大部分MCU都带有IIC总线接口。但是这里我们不使用STM32L1的硬件IIC来读写SHT20, 而是通过软件模拟。ST为了规避飞利浦IIC专利问题, 将STM32的硬件IIC设计的比较复杂, 而且稳定性不怎么好, 所以这里我们不推荐使用。

用软件模拟IIC, 最大的好处就是方便移植, 同一个代码兼容所有MCU, 任何一个单片机只要有IO口, 就可以很快的移植过去, 而且不需要特定的IO口。而硬件IIC, 则换一款MCU, 基本上就得重新搞一次, 移植是比较麻烦的。

本章我们将向大家介绍如何使用STM32L151的IIC接口实现和温度湿度传感器SHT20之间的通信, 来读取SHT20的温湿度值, 并将结果显示在LCD模块上。本章分为如下几个部分:

- 13.1 IIC简介
- 13.2 硬件设计
- 13.3 软件设计
- 13.4 下载验证

13.1 IIC简介

IIC我们已经在前面介绍过了, 这里不再赘述。

13.2 硬件设计

本章需要用到的硬件资源有:

- 1) LCD模块
- 2) SHT20

前面2部分的资源, 我们前面已经介绍了, 请大家参考相关章节。这里只介绍SHT20与STM32L151的连接, SHT20的SCL和SDA分别连在STM32L151的PB8和PB9上的, 连接关系如图13.2.1所示:

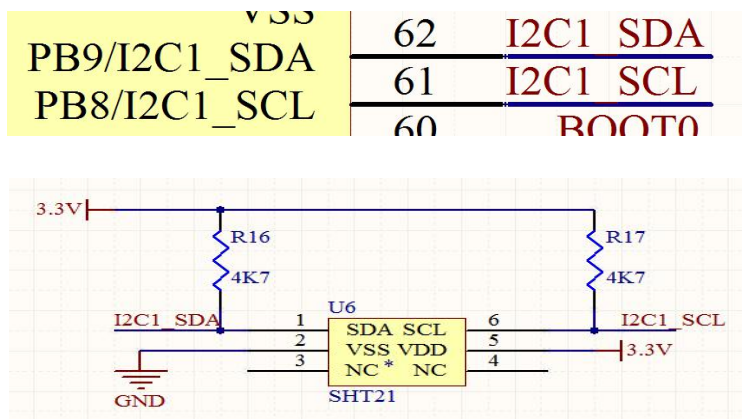
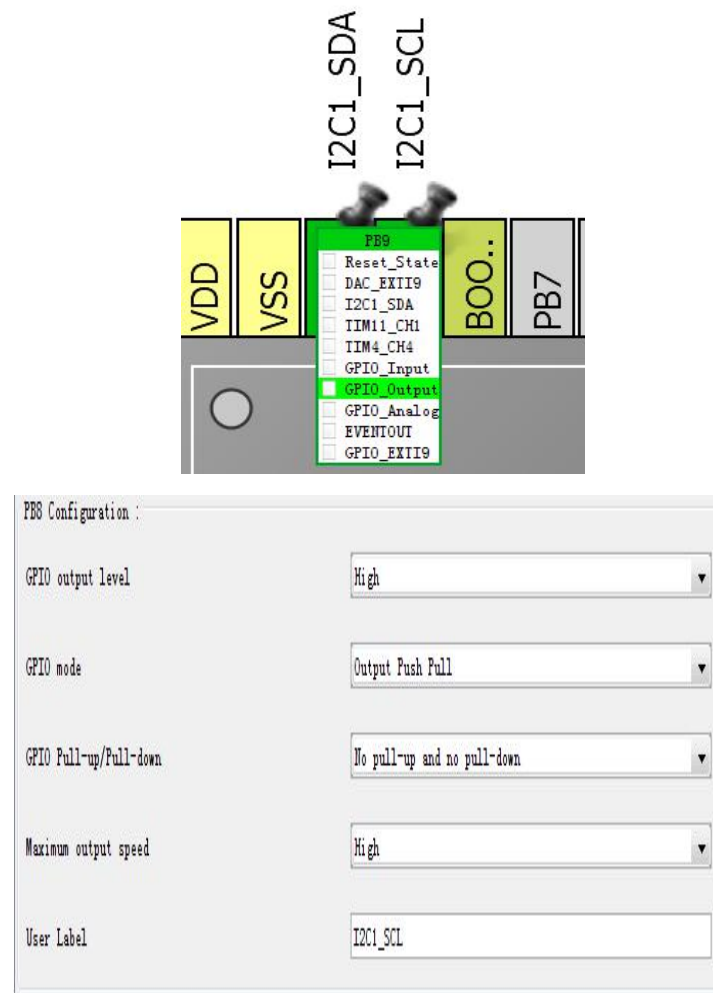


图 13.2.1 STM32L151 与 SHT20 连接图

13.3 软件设计

因为我们用的软件模拟 IIC，所以我们只需要配置 PB8 和 PB9 两个引脚为输出模式。



配置好生成工程之后，我们打开 sht20.c,代码如下：

```
//SDA 设置输入，读 SHT20 数据
static void ADX_SDA_IN(void) //sda 线输入
{
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.Pin = SHT20_SDA_pin;
    GPIO_InitStructure.Mode = GPIO_MODE_INPUT;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(SHT20_SDA_port, &GPIO_InitStructure);
    HAL_Delay(10);
}

//SDA 设置输出，向 SHT20 写入数据
static void ADX_SDA_OUT(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.Pin = SHT20_SDA_pin;
    GPIO_InitStructure.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStructure.Pull = GPIO_NOPULL;
    GPIO_InitStructure.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
    HAL_GPIO_Init(SHT20_SDA_port, &GPIO_InitStructure);
}

//产生 IIC 起始信号，SCL 为高电平时，SDA 由高电平向低电平跳变
static void ADX_IIC_Start(void)
{
    ADX_SDA_OUT();
    ADX_SDA_H();
}
```

```
    ADX_SCL_H();
    HAL_Delay(4);
    ADX_SDA_L();
    HAL_Delay(4);
    ADX_SCL_L();
    HAL_Delay(4);
}

//产生 IIC 停止信号，SCL 为高电平时，SDA 由低电平向高电平跳变
static void ADX_IIC_Stop(void)
{
    ADX_SDA_OUT();
    ADX_SCL_L();
    ADX_SDA_L();
    HAL_Delay(4);
    ADX_SCL_H();
    HAL_Delay(4);
    ADX_SDA_H();
    HAL_Delay(4);
}

//等待应答信号到来
static unsigned char IIC_Wait_Ack(void)
{
    unsigned char ucErrTime=0;
    ADX_SDA_IN();
    ADX_SDA_H();
    HAL_Delay(1);
    ADX_SCL_H();
    HAL_Delay(1);
    while(ADX_SDA_IN_Bit)
    {
        ucErrTime++;
        if(ucErrTime>250)
        {
            ADX_IIC_Stop();
            return 1;
        }
    }
    ADX_SCL_L();
    return 0;
}
```



```
static void IIC_Ack(void)
{
    ADX_SCL_L();
    ADX_SDA_OUT();
    ADX_SDA_L();
    HAL_Delay(20);
    ADX_SCL_H();
    HAL_Delay(2);
    ADX_SCL_L();
}

static void IIC_NAck(void)
{
    ADX_SCL_L();
    ADX_SDA_OUT();
    ADX_SDA_H();
    HAL_Delay(5);
    ADX_SCL_H();
    HAL_Delay(2);
    ADX_SCL_L();
}

static void IIC_Send_Byte(unsigned char txd)
{
    unsigned char t;
    ADX_SDA_OUT();
    ADX_SCL_L();
    for(t=0;t<8;t++)
    {
        if(txd&0x80) //txd 的最高位非 0
        {
            ADX_SDA_H();
        }
        else
        {
            ADX_SDA_L();
        }
        txd<<=1;    //t 左移一位
        HAL_Delay(5);
        ADX_SCL_H();
        HAL_Delay(5);
        ADX_SCL_L();
        HAL_Delay(5);
    }
}
```

```
static unsigned char ADX_IIC_Read_Byte(unsigned char ack)
{
    unsigned char i, receive=0;
    ADX_SDA_IN();
    for(i=0; i<8; i++)
    {
        ADX_SCL_L();
        HAL_Delay(5);
        ADX_SCL_H();
        receive<<=1;
        if(ADX_SDA_IN_Bit)
            receive++;
        HAL_Delay(5);
    }
    if(!ack)
        IIC_NAck();
    else
        IIC_Ack();
    return receive;
}
```

```
unsigned char SHT20_SoftReset(void)
{
    unsigned char err=0;
    ADX_IIC_Start();
    IIC_Send_Byte(0x80);
    err=IIC_Wait_Ack();
    IIC_Send_Byte(0xFE);
    err=IIC_Wait_Ack();
    ADX_IIC_Stop();
    return err;
}
```

```
unsigned char SHT20_Init(void)
{
    unsigned char err;
    err=SHT20_SoftReset();
    return err;
}
```

```
float SHT20_GetTempPoll(void)
{
    float TEMP;
    unsigned char ack, temp1, temp2;
```

```
int ST;
int i=0;
ADX_IIC_Start();
IIC_Send_Byte(I2C_ADR_W);
ack=IIC_Wait_Ack();
IIC_Send_Byte(TRIG_T_MEASUREMENT_POLL);
ack=IIC_Wait_Ack();
do{
    HAL_Delay(100);
    ADX_IIC_Start();
    IIC_Send_Byte(I2C_ADR_R);
    i++;
    ack=IIC_Wait_Ack();
    if(i==10)break;
}while(ack!=0);
temp1=ADX_IIC_Read_Byte(1);
temp2=ADX_IIC_Read_Byte(1);
ADX_IIC_Read_Byte(0);
ADX_IIC_Stop();
ST=(temp1<<8)|(temp2<<0);
ST&=~0X0003;
TEMP = ((float)ST * 0.00268127) - 46.85;
return(TEMP);
}

float SHT20_GetHumiPoll(void)
{
    float HUMI;
    unsigned char ack,temp1,temp2;
    int SRH;
    int i=0;
    ADX_IIC_Start();
    IIC_Send_Byte(I2C_ADR_W);
    ack=IIC_Wait_Ack();
    IIC_Send_Byte(TRIG_RH_MEASUREMENT_POLL);
    ack=IIC_Wait_Ack();
    do{
        HAL_Delay(100);
        ADX_IIC_Start();
        IIC_Send_Byte(I2C_ADR_R);
        i++;
        ack=IIC_Wait_Ack();
        if(i==10)break;
    }while(ack!=0);
```

```
temp1=ADX_IIC_Read_Byte(1);
temp2=ADX_IIC_Read_Byte(1);
ADX_IIC_Read_Byte(0);
ADX_IIC_Stop();

SRH=(temp1<<8)|(temp2<<0);
SRH&=~0X0003;
HUMI=((float)SRH*0.00190735)-6;
return(HUMI);
}

void Read_SHT20(void)
{
    SHT20_Sample.temperature=SHT20_GetTempPoll();
    SHT20_Sample.humidity=SHT20_GetHumiPoll();
}
```

接下来我们看一下主函数设计，代码如下：

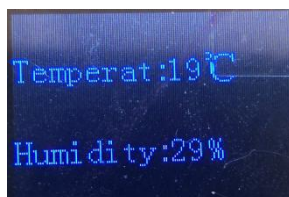
```
int main(void)
{
    unsigned char ge=0;
    unsigned char shi=0;
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI2_Init();
    LCD_Init();
    LCD_Clear(BLACK);
    Show_Str(0,20,BLUE,BLACK,"Temperat:   iæ",16,1);
    Show_Str(0,60,BLUE,BLACK,"Humidity:   %",16,1);
    SHT20_Init();
    while (1)
    {
        if(!SHT20_Read_Time)
        {
            Read_SHT20();
            ge=(unsigned int)(SHT20_Sample.temperature)/10%10;
            shi=(unsigned int)(SHT20_Sample.temperature)%10;
            LCD_Fill(70,20,86,40,BLACK);
            LCD_ShowNum(70,20,shi,1,16);
            LCD_ShowNum(78,20,ge,1,16);
            shi=(unsigned int)(SHT20_Sample.humidity)/10%10;
            ge=(unsigned int)(SHT20_Sample.humidity)%10;
```

```
LCD_Fill(70,60,86,80,BLACK);  
LCD_ShowNum(70,60,shi,1,16);  
LCD_ShowNum(78,60,ge,1,16);  
}  
SHT20_Read_Time++;  
if(SHT20_Read_Time>1000)  
{  
    SHT20_Read_Time=0;  
}  
}  
}
```

我们可以看到主函数里前面是 LCD 初始化和 SHT20 初始化，然后在死循环里读取 SHT20 的温湿度值，然后显示。这里需要说明一点，不能连续读取 SHT20 的值否则会造成数据错误，所以我们设计了一个变量 SHT20_Read_Time 来控制读取 SHT20 的时间。

13.4 下载验证

在代码编译成功之后，我们通过下载代码到 STM32L1 开发板上，得到如图 13.4.1 所示：



第十四章 SPI 实验

在本章中，我们将使用 STM32L151 自带的 SPI 来实现对外部 FLASH（W25Q64）的读写，并将结果显示在 LCD 模块上。本章分为如下几个部分：

- 14.1 SPI 简介
- 14.2 硬件设计
- 14.3 STM32CubeMX 配置 SPI 和软件设计
- 14.4 下载验证

14.1 SPI 简介

我们已经在前面介绍过 SPI 接口，这里我们不再赘述，下面我们来看一下硬件设计。

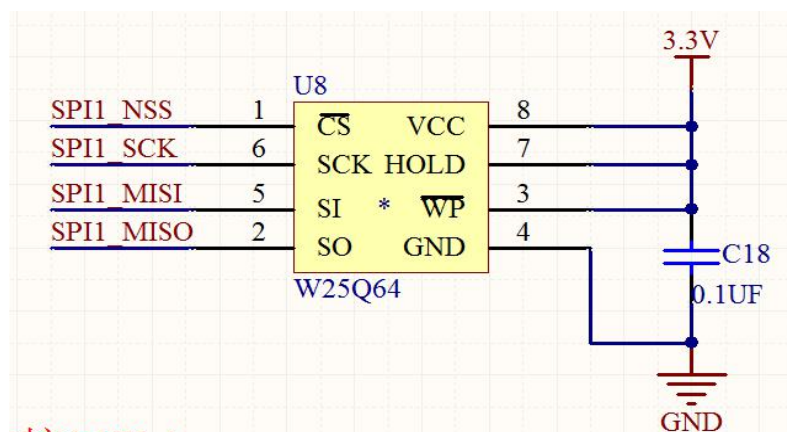
14.2 硬件设计

本章实验功能简介：在主循环里面检测两个按键，其中 1 个按键（KEY1）用来执行写入 W25Q64 的操作，另外一个按键（KEY2）用来执行读出操作，在 LCD 模块上显示相关信息。所用到的硬件资源如下：

- 2) KEY1 和 KEY2 按键
- 3) LCD 模块
- 4) SPI
- 5) W25Q64

这里只介绍 W25Q64 与 STM32L151 的连接，板上的 W25Q64 是直接连在 STM32L151 的 SPI1 上的，连接关系如图 14.2.1 所示：

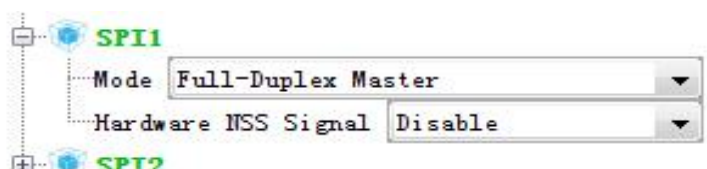
3.3V				19	VDD
	SPI1_NSS			20	PA4/SPI1_NSS
	SPI1_SCK			21	PA5/SPI1_SCK
	SPI1_MISO			22	PA6/SPI1_MISO
	SPI1_MOSI			23	PA7/SPI1_MOSI
				24	



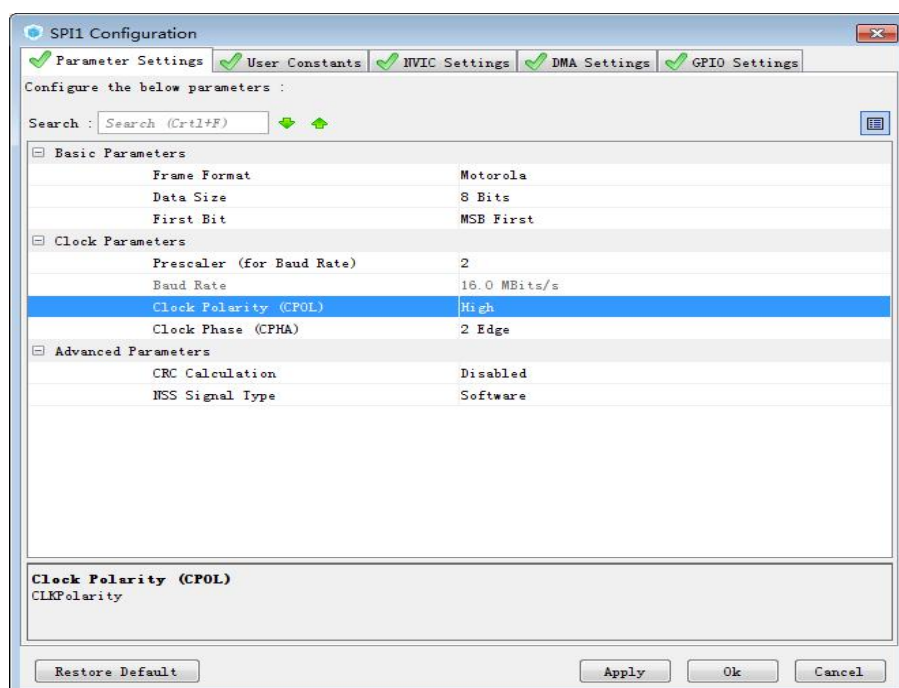
14.3 STM32CubeMX 配置 SPI 和软件设计

使用 STM32CubeMX 配置 SPI 的一般步骤为：

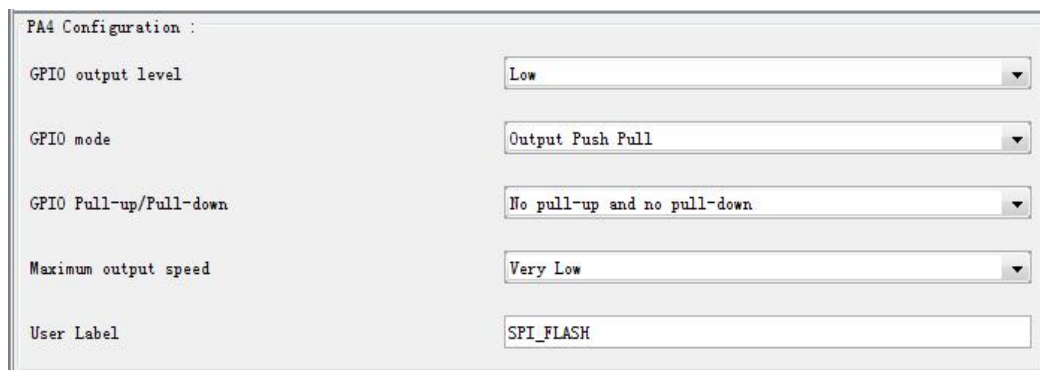
- ① 进入 Pinout->SPI 配置栏，启动 SPI1 基本参数：



- ② 进入 Configuration->SPI1 配置栏，配置基本参数：



- ③ 配置 PA4 引脚，选择 GPIO_Output 功能。串口和 SPI 的配置和之前的实验一样，这里不再重复。



生成工程后，我们首先在 usart.c 中添加回调函数，代码如下（大家自己定义变量）：

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance==USART3)
    {
        switch(USART3_Count)
        {
            case 0:
                if(aRxBuffer[0]!=0x0D)
                {
                    USART3_Receviebuf[USART3_len]=aRxBuffer[0];
                    USART3_len++;
                }
                else
                {
                    USART3_Count=1;
                }
                break;
            case 1:
                if(aRxBuffer[0]!=0x0A)
                {
                    flash_len=0;
                }
                else
                {
                    flash_len=USART3_len;
                    USART3_Recevie_flag=1;
                }
                USART3_len=0;
            USART3_Count=0;
            break;
            default:
                break;
        }
        HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
    }
}

void USART3_SendData(unsigned char ch)
{
    HAL_UART_Transmit_IT(&huart3,&ch,1);
    while(__HAL_UART_GET_FLAG(&huart3,UART_FLAG_TC)!=SET);
}
```

```
void USART3_SendStr(unsigned char *str)
{
    while(*str)
        USART3_SendData(*str++);
}
```

我们在 spi.c 文件中添加代码：

```
void MX_SPI1_Init(void)
{
    hspi1.Instance = SPI1;
    hspi1.Init.Mode = SPI_MODE_MASTER;
    hspi1.Init.Direction = SPI_DIRECTION_2LINES;
    hspi1.Init.DataSize = SPI_DATASIZE_8BIT;
    hspi1.Init.CLKPolarity = SPI_POLARITY_HIGH;
    hspi1.Init.CLKPhase = SPI_PHASE_2EDGE;
    hspi1.Init.NSS = SPI_NSS_SOFT;
    hspi1.Init.BaudRatePrescaler = SPI_BAUDRATEPRESCALER_2;
    hspi1.Init.FirstBit = SPI_FIRSTBIT_MSB;
    hspi1.Init.TIMode = SPI_TIMODE_DISABLE;
    hspi1.Init.CRCCalculation = SPI_CRCCALCULATION_DISABLE;
    hspi1.Init.CRCPolynomial = 10;
    if (HAL_SPI_Init(&hspi1) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    __HAL_SPI_ENABLE(&hspi1);
    SPI_Flash_WAKEUP();
}
```

同时我们添加 flash.c 和 flash.h 文件，打开 flash.c 代码如下：

```
//各种指令
#define SPI_FLASH_PageSize  256
#define SPI_FLASH_PerWritePageSize  256

#define W25X_ReleasePowerDown  0xAB
#define W25X_WriteEnable  0x06
#define W25X_SectorErase  0x20
#define W25X_ReadStatusReg  0x05
#define W25X_PageProgram  0x02
#define W25X_ReadData  0x03

#define WIP_Flag  0x01
#define Dummy_Byte  0xFF
```

```
//唤醒 FLASH
void SPI_Flash_WAKEUP(void)
{
    FLASH_SPI_CS_ENABLE();
    SPI_FLASH_SendByte(W25X_ReleasePowerDown); //发送 PowerDown 指令
    FLASH_SPI_CS_DISABLE();
    HAL_Delay(3);
}

//擦除某一页的数据
void SPI_FLASH_SectorErase(unsigned int SectorAddr)
{
    SPI_FLASH_WriteEnable();
    FLASH_SPI_CS_ENABLE();
    SectorAddr*=4096; //擦除扇区 4K=4*1024=4096
    SPI_FLASH_SendByte(W25X_SectorErase); //页编程指令
    SPI_FLASH_SendByte((SectorAddr&0xFF0000)>>16); //发送 SectorAddr 高半字节地址字节
    SPI_FLASH_SendByte((SectorAddr&0xFF00)>>8); //发送 SectorAddr 中等半字节地址字节
    SPI_FLASH_SendByte(SectorAddr&0xFF); //发送 SectorAddr 低半字节地址字节
    FLASH_SPI_CS_DISABLE();
    SPI_FLASH_WaitForWriteEnd();
}

//写使能
void SPI_FLASH_WriteEnable(void)
{
    FLASH_SPI_CS_ENABLE();
    SPI_FLASH_SendByte(W25X_WriteEnable); //·čĚĭ°Đ'Ê'ÄÜĭ±Ö_Áĥ
    FLASH_SPI_CS_DISABLE();
}

//SPI 向 FLASH 发送一个字节数据
unsigned char SPI_FLASH_SendByte(unsigned char byte)
{
    unsigned char d_read,d_send=byte;
    if(HAL_SPI_TransmitReceive(&hspi1,&d_send,&d_read,1,0xFFFFFFFF)!=HAL_OK)
        d_read=Dummy_Byte;
    return d_read;
}

//等待写完成
void SPI_FLASH_WaitForWriteEnd(void)
{
    unsigned char FLASH_Status=0;
    FLASH_SPI_CS_ENABLE();
```

```

SPI_FLASH_SendByte(W25X_ReadStatusReg); //发送“读取寄存器”指令
do //只要内存忙于写周期就循环
{
    FLASH_Status=SPI_FLASH_SendByte(Dummy_Byte);
}while((FLASH_Status&WIP_Flag)==SET);
FLASH_SPI_CS_DISABLE();
}

//写数据
void SPI_FLASH_BufferWrite(unsigned char* pBuffer, unsigned int WriteAddr, unsigned int
                           NumByteToWrite)
{
    unsigned char NumOfPage = 0, NumOfSingle = 0, Addr = 0, count = 0, temp = 0;
    Addr=WriteAddr%SPI_FLASH_PageSize; //写入地址是每页的第几位
    count=SPI_FLASH_PageSize-Addr; //在开始的一页要写入的个数
    NumOfPage=NumByteToWrite/SPI_FLASH_PageSize; //要写入的页数
    NumOfSingle=NumByteToWrite%SPI_FLASH_PageSize; //不足一页的个数

    if(Addr==0) //写入地址是页的开始
    {
        if(NumOfPage==0) //数据小于一页
        {
            SPI_FLASH_PageWrite(pBuffer,WriteAddr,NumByteToWrite); //写少于一页的个数
        }
        else
        {
            while(NumOfPage--)
            {
                SPI_FLASH_PageWrite(pBuffer,WriteAddr,SPI_FLASH_PageSize); //写一页的数
                WriteAddr+=SPI_FLASH_PageSize;
                pBuffer+=SPI_FLASH_PageSize;
            }
            SPI_FLASH_PageWrite(pBuffer,WriteAddr,NumOfSingle); //写少于一页的数据
        }
    }
    else //写入地址不是页的开始
    {
        if(NumOfPage==0)
        {
            if(NumOfSingle>count)
            {
                temp=NumOfSingle-count;
                SPI_FLASH_PageWrite(pBuffer,WriteAddr,count);
                WriteAddr+=count;
                pBuffer+=count;
            }
        }
    }
}

```

```
        SPI_FLASH_PageWrite(pBuffer,WriteAddr,temp);
    }
    else
    {
        SPI_FLASH_PageWrite(pBuffer,WriteAddr,NumByteToWrite);
    }
}
else
{
    NumByteToWrite-=count;
    NumOfPage=NumByteToWrite/SPI_FLASH_PageSize;
    NumOfSingle=NumByteToWrite%SPI_FLASH_PageSize;
    SPI_FLASH_PageWrite(pBuffer,WriteAddr,count);
    WriteAddr+=count;
    pBuffer+=count;
    while(NumOfPage--)
    {
        SPI_FLASH_PageWrite(pBuffer,WriteAddr,SPI_FLASH_PageSize);
        WriteAddr+=SPI_FLASH_PageSize;
        pBuffer+=SPI_FLASH_PageSize;
    }
    if(NumOfSingle!=0)
    {
        SPI_FLASH_PageWrite(pBuffer,WriteAddr,NumOfSingle);
    }
}
}
```

//写一页的数据

```
void SPI_FLASH_PageWrite(unsigned char* pBuffer, unsigned int WriteAddr, unsigned int
                        NumByteToWrite)
{
    SPI_FLASH_WriteEnable();
    FLASH_SPI_CS_ENABLE();
    SPI_FLASH_SendByte(W25X_PageProgram);
    SPI_FLASH_SendByte((WriteAddr&0xFF0000)>>16);
    SPI_FLASH_SendByte((WriteAddr&0xFF00)>>8);
    SPI_FLASH_SendByte(WriteAddr&0xFF);
    if(NumByteToWrite>SPI_FLASH_PerWritePageSize)
    {
        NumByteToWrite=SPI_FLASH_PerWritePageSize;
    }
}
```

```
while(NumByteToWrite--)  
{  
    SPI_FLASH_SendByte(*pBuffer);  
    pBuffer++;  
}  
FLASH_SPI_CS_DISABLE();  
SPI_Flash_Wait_Busy();  
}  
  
//等待写完成  
void SPI_Flash_Wait_Busy(void)  
{  
    while((SPI_Flash_ReadSR() & 0x01) != 0x01);  
}  
  
//读取 FLASH 状态  
unsigned char SPI_Flash_ReadSR(void)  
{  
    unsigned char byte=0;  
    FLASH_SPI_CS_ENABLE();  
    SPI_FLASH_SendByte(W25X_ReadStatusReg);  
    byte=SPI_FLASH_SendByte(0xFF);  
    FLASH_SPI_CS_DISABLE();  
    return byte;  
}  
  
//读数据（不限个数）  
void SPI_FLASH_BufferRead(unsigned char * pBuffer, unsigned int ReadAddr, unsigned int  
                           NumByteToRead)  
{  
    unsigned int i;  
    FLASH_SPI_CS_ENABLE();  
    SPI_FLASH_SendByte(W25X_ReadData);  
    SPI_FLASH_SendByte((unsigned char)((ReadAddr)>>16));  
    SPI_FLASH_SendByte((unsigned char)((ReadAddr)>>8));  
    SPI_FLASH_SendByte((unsigned char)ReadAddr);  
    for(i=0;i<NumByteToRead;i++)  
    {  
        pBuffer[i]=SPI_FLASH_SendByte(0xFF);  
    }  
    FLASH_SPI_CS_DISABLE();  
}
```

最后，我们看看 main 函数，代码如下：


```
int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_SPI1_Init();
    MX_SPI2_Init();
    MX_USART3_UART_Init();
    LCD_Init();
    LCD_Clear(BLACK);
    Show_Str(0,10,BLUE,BLACK,"SPI TEST",16,1);
    while (1)
    {
        if(USART3_Recevie_flag)
        {
            USART3_Recevie_flag=0;
            USART3_SendStr("OK!");
        }
        if(KEY1())
        {
            HAL_Delay(20);
            if(KEY1())
            {
                if(flash_len)
                {
                    SPI_FLASH_SectorErase(0);
                    SPI_FLASH_BufferWrite(USART3_Receviebuf,0,flash_len);
                    LCD_Clear(BLACK);
                    LCD_Fill(0,30,128,50,BLACK);
                    Show_Str(0,30,BLUE,BLACK,"FLASH Write OK!",16,1);
                }
            }
            while(KEY1());
        }
        if(KEY2())
        {
            HAL_Delay(20);
            if(KEY2())
            {
                if(flash_len)
                {
                    SPI_FLASH_BufferRead(flash_read,0,flash_len);
                    LCD_Clear(BLACK);
                    LCD_Fill(0,30,128,50,BLACK);
                }
            }
        }
    }
}
```

```
        Show_Str(0,30,BLUE,BLACK,"FLASH Read OK!",16,1);  
        POINT_COLOR=BLUE;  
        LCD_Fill(0,80,128,128,BLACK);  
        LCD_ShowString(0,80,16,flash_read,1);  
    }  
}  
while(KEY2());  
}  
}  
}
```

这部分代码和 IIC 实验那部分代码大同小异，我们就不多说了，实现的功能就和 IIC 差不多，不过此次写入和读出的是 SPIFLASH，而不是 EEPROM。

14.4 下载验证

在代码编译成功之后，我们通过下载代码到 STM32 开发板上，通过先按 KEY1 按键写入数据，然后按 KEY2 读取数据，得到如图 13.4.1 所示：



图 14.4.1 SPI 实验程序运行效果图

第十五章 485 实验

本章我们将向大家介绍如何使用 STM32L151 的串口实现 485 通信（半双工）。在本章中，我们将使用 STM32L151 的串口 3 来实现开发板和电脑之间的 485 通讯，并将结果显示在 LCD 模块上。本章分为如下几个部分：

- 15.1 485 简介
- 15.2 硬件设计
- 15.3 STM32CubeMX 配置串口 3 和软件设计
- 15.4 下载验证

15.1 485 简介

485（一般称作 RS485/EIA-485）是隶属于 OSI 模型物理层的电气特性规定为 2 线，半双工，多点通信的标准。它的电气特性和 RS-232 大不一样。用缆线两端的电压差值来表示传递信号。RS485 仅仅规定了接受端和发送端的电气特性。它没有规定或推荐任何数据协议。

RS485 的特点包括：

- ① 接口电平低，不易损坏芯片。RS485 的电气特性：逻辑“1”以两线间的电压差为 + (2~6)V 表示；逻辑“0”以两线间的电压差为 - (2~6)V 表示。接口信号电平比 RS232 降低了，不易损坏接口电路的芯片，且该电平与 TTL 电平兼容，可方便与 TTL 电路连接。
- ② 传输速率高。10 米时，RS485 的数据最高传输速率可达 35Mbps，在 1200m 时，传输速度可达 100Kbps。
- ③ 抗干扰能力强。RS485 接口是采用平衡驱动器和差分接收器的组合，抗共模干扰能力增强，即抗噪声干扰性好。传输距离远，支持节点多。RS485 总线最长可以传输 1200m 以上（速率 ≤ 100Kbps）
- ④ 一般最大支持 32 个节点如果使用特制的 485 芯片可以达到 128 个或者 256 个节点，最大的可以支持到 400 个节点。

RS485 推荐使用在点对点网络中，线型，总线型，不能是星型，环型网络。理想情况下 RS485 需要 2 个终端匹配电阻，其阻值要求等于传输电缆的特性阻抗（一般为 120Ω）。没有特性阻抗的话，当所有的设备都静止或者没有能量的时候就会产生噪声，而且线移需要双端的电压差。没有终接电阻的话，会使得较快速的发送端产生多个数据信号的边缘，导致数据传输出错。485 推荐的连接方式如图 15.1.2 所示：

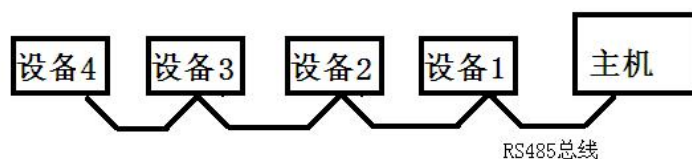


图15.1.2 RS485 连接

在上面的连接中，如果需要添加匹配电阻，我们一般在总线的起止端加入，也就是主机和设备4上面各加一个 120Ω 的匹配电阻。

由于RS485具有传输距离远、传输速度快、支持节点多和抗干扰能力更强等特点，所以RS485有很广泛的应用。

STM32L151开发板采用SP3485作为收发器，该芯片支持3.3V供电，最大传输速度可达10Mbps，支持多达32个节点，并且有输出短路保护。该芯片的框图如图15.1.2所示：

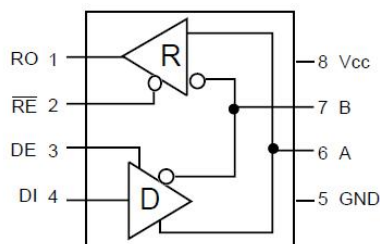


图15.1.2 SP3485 框图

图中A、B总线接口，用于连接485总线。RO是接收输出端，DI是发送数据收入端，RE是接收使能信号（低电平有效），DE是发送使能信号（高电平有效）。

本章，我们通过该芯片连接STM32L151的串口3，实现开发板和电脑之间的485通信。本章将实现这样的功能：通过连接STM32L151开发板的RS485接口和电脑，接收到电脑发送过来的数据重新发送给电脑显示。

15.2 硬件设计

本章要用到的硬件资源如下：

- 1) 串口3
- 2) RS485收发芯片SP3485

串口3之前都已经详细介绍过了这里我们介绍SP3485和串口3的连接关系如图15.2.1所示：

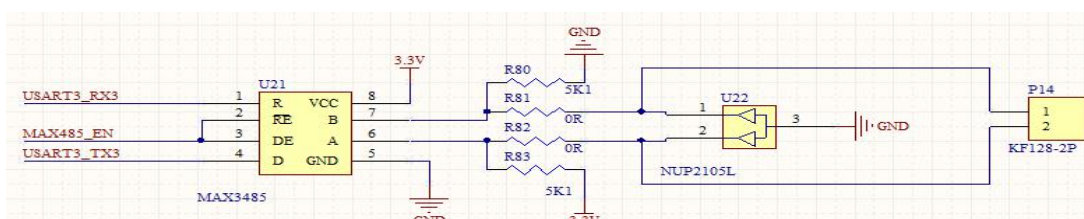


图 15.2.1STM32L151与SP3485连接电路图

从上图可以看出：STM32L151 的串口3 通过P8 端口设置，连接到SP3485。注意：RS485_EN 信号，是连接到 MCU。RS485_RE 控制 SP3485 的收发，当 RS485_EN=0 的时候，为接收模式；当RS485_EN=1 的时候，为发送模式。然后，我们要设置好开发板上P8 排针的连接，通过跳线帽将USART3_RX/USART3_TX 分别和 USART3_RX3/USART3_TX3 连接。

最后，我们用 USB-485 将开发板的 485 和电脑连接（A 接 A,B 接 B），接反了会导致通讯异常！！

15.3 STM32CubeMX 配置串口和软件设计

STM32CubeMX 配置串口的过程我们已经在前面讲过，配置完工程后生成代码，接下来我们看一下串口初始化函数，代码如下：

```
void MX_USART3_UART_Init(void)
{
    huart3.Instance = USART3;
    huart3.Init.BaudRate = 9600;
    huart3.Init.WordLength = UART_WORDLENGTH_8B;
    huart3.Init.StopBits = UART_STOPBITS_1;
    huart3.Init.Parity = UART_PARITY_NONE;
    huart3.Init.Mode = UART_MODE_TX_RX;
    huart3.Init.HwFlowCtl = UART_HWCONTROL_NONE;
    huart3.Init.OverSampling = UART_OVERSAMPLING_16;
    if (HAL_UART_Init(&huart3) != HAL_OK)
    {
        _Error_Handler(__FILE__, __LINE__);
    }
    MAX485_RX_EN();
    HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
}
void HAL_UART_MspInit(UART_HandleTypeDef* uartHandle)
{

```

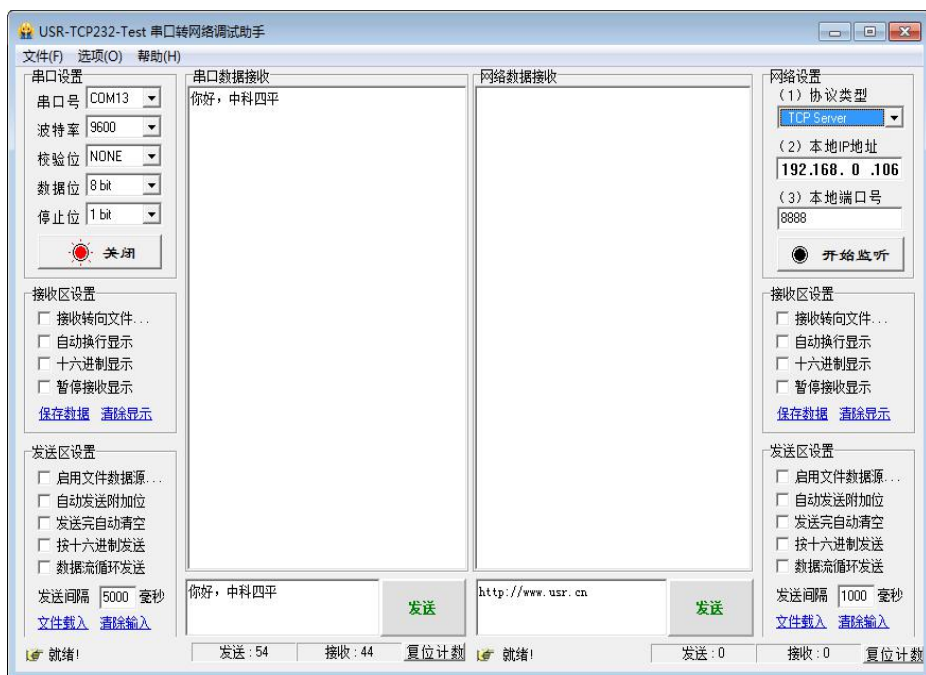
对比 232 课程我们可以发现，增加了一句 MAX485_RX_EN() 来控制 485 方向。接下来我们看一下回调函数和发送函数：

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    if(huart->Instance==USART3)
    {
        switch(USART3_Count)
        {
            case 0:
                if(aRxBuffer[0]!=0x0d)           //如果收到不是 0xAA
                {
                    USART3_Receviebuf[USART3_len]=aRxBuffer[0];
                    USART3_len++;
                }
                else
                {
                    USART3_Count=1;
                }
                break;
            case 1:
                if(aRxBuffer[0]==0x0a)
                {
                    USART3_Recevie_flag=1;
                }
                USART3_Count=0;
                break;
            default:
                break;
        }
        HAL_UART_Receive_IT(&huart3,aRxBuffer,USART3_size);
    }
}

void USART3_Sendbuf(unsigned char *buf,unsigned char len)
{
    MAX485_TX_EN();
    HAL_UART_Transmit_IT(&huart3,buf,len);
    while(__HAL_UART_GET_FLAG(&huart3,UART_FLAG_TC)!=SET);
    MAX485_RX_EN();
}
```

15.4 下载验证

在代码编译成功之后，我们通过下载代码到 STM32 开发板上，打开串口后用串口助手发送数据，需要注意的是在数据末尾需要加一个回车，得到如图 15.4.1 所示：



第十六章 无源蜂鸣器实验

本章，我们将利用STM32L15 的PWM 来控制蜂鸣器的声调，达到类似唱歌的效果。本章分为如下几个部分：

- 16.1 无源蜂鸣器简介
- 16.2 硬件设计
- 16.3 STM32CubeMX 配置 PWM 和软件设计
- 16.4 下载验证

16.1 无源蜂鸣器简介

我们现在用到的传感器一共是两种，一种是有源蜂鸣器（自带振荡电路，一通电就会发声），另一种是无源蜂鸣器（没有自带振荡电路，必须提供 2.5kHz 左右的方波振动，才能发声）。

无源蜂鸣器利用电磁感应现象，为音圈接入交变电流后形成的电磁铁与永磁铁相吸或相斥而推动振膜发声，接入直流电只能持续推动振膜而无法产生声音，只能在接通或断开时产生声音。无源蜂鸣器的工作原理与扬声器相同。

在使用方波信号源驱动的应反向并联一个二极管，防止突然断电时产生的高压反向电动势击穿其他元件以及使用寿命缩短。有源蜂鸣器往往比无源的贵，就是因为里面多个震荡电路，只需接入额定电压的直流电即可 发出指定频率的声音，频率由内部振荡电路决定，无法改变。

无源蜂鸣器的优点是：

1. 制作成本低
2. 声音频率范围宽，可高分贝的发出某些频率的超声波以及可以做出“多来米发索拉西”的效果

16.2 硬件设计

无源蜂鸣器不能直接接到 STM32 的 GPIO 口上，所以我们用 IO 口控制三极管通断达到控制蜂鸣器的目的，硬件设计如图 16.2.1 所示

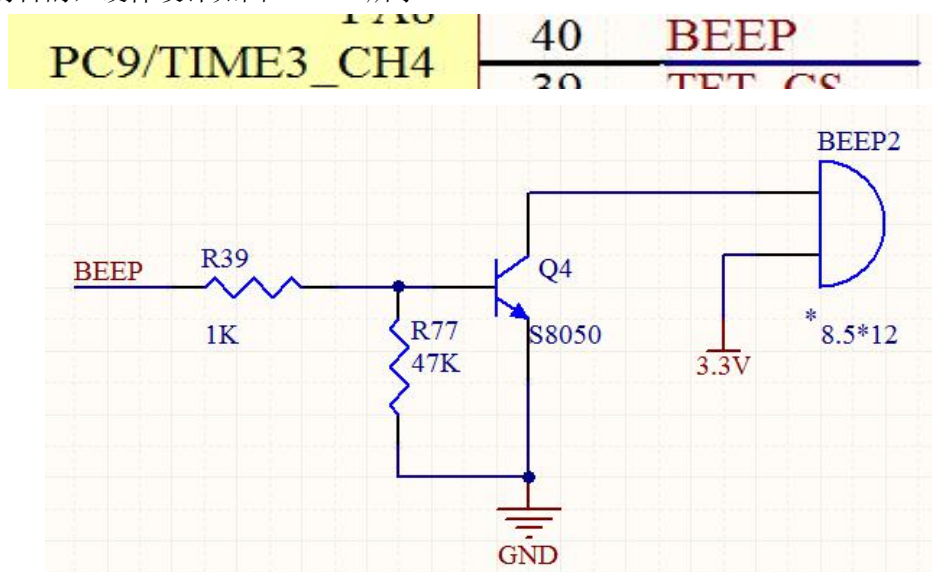


图 16.2.1 无源蜂鸣器硬件电路

16.3 STM32CubeMX 配置定时器PWM 输出功能和软件设计

使用 STM32CubeMX 配置 PWM 输出的配置步骤我们已经在前面 PWM 课程介绍过了，这里不再赘述。

我们配好工程后生成代码，我们打开工程后引入 music.c 和 music.h，打开 music.c 我们可以看到代码如下：

```
void buzzerQuiet(void) 蜂鸣器静音
{
    HAL_TIM_PWM_Stop(&htim3,TIM_CHANNEL_4); 关闭定时器和 CH4 的使能
    HAL_GPIO_WritePin(BEEP_GPIO_Port,BEEP_Pin,GPIO_PIN_RESET);
}
void buzzerSound(unsigned short usFreq) 蜂鸣器发出特定频率的声音
{
    unsigned long ulVal;
    if((usFreq<=3555555/65536UL)|| (usFreq>20000)) 发声频率取值为（系统时钟/65536）+1~20000
    {
        buzzerQuiet();
    }
    else
    {
        ulVal=3555555/usFreq;
        TIM3->ARR=ulVal; 初值
        BEEP_setvalue(ulVal/20); 比较值
        HAL_TIM_PWM_Start(&htim3,TIM_CHANNEL_4); 定时器以及 CH4 的使能
    }
}
void musicPlay(void)演奏乐曲
{
    unsigned char i=0;
    while(1)
    {
        if(MyScore[i].mTime==0)break;
        buzzerSound(MyScore[i].mName); 取出 MyScore[i].mName
        HAL_Delay(MyScore[i].mTime); 延时这个音调的时长
        i++; 取下一个音，循环
        buzzerQuiet();
        HAL_Delay(10);
    }
}
```

音 符	名 称	相 对 时 值
5—	全音符	T
5-	二分音符	T/2
5	四分音符	T/4
5	八分音符	T/8
5	十六分音符	T/16
5—	符点二分音符	T/2+T/4
5.	符点四分音符	T/4+T/8
5.	符点八分音符	T/8+T/16

表 16.3.1

在头文件“music.h”里定义有一个音符结构体 tNote，有两个数据成员：音名 mName 和时值 mTime。在 C 文件“music.c”里定义有一个 tNote 型常量数表 MyScore[]，用来保存实际乐谱转换成 tNote 格式的数据。有了上述一点点乐谱基础知识，我们就可以很方便地编辑这个数表了。比如音符“3”转换为“{M3, T/4}”，音符“3.”转换为“{M3, T/4+T/8}”，等等。接下来我们看一下 music.h 文件，代码如下：

```

#ifndef _MUSIC_H_
#define _MUSIC_H_

//定义低音音色
#define L1 262
#define L2 294
#define L3 330
#define L4 349
#define L5 392
#define L6 440
#define L7 494

//定义中音音色
#define M1 523
#define M2 587
#define M3 659
#define M4 698
#define M5 784
#define M6 880
#define M7 988

//定义高音音色
#define H1 1047
#define H2 1175
#define H3 1319
#define H4 1397
#define H5 1568
#define H6 1760
#define H7 1976
  
```

```
#define TT 2000
typedef struct
{
    short mName;
    short mTime;
}tNode;
const tNode MyScore[]=
{
    {L3,TT/8},{M6,TT/4},{M5,TT/4},{M6,TT/4},{M5,TT/8},{M3,TT/8},{M3,TT/4},{L3,TT/8},{M6,TT/4},{
    M5,TT/4},{M6,TT/4},{M5,TT/8},{M6,TT/8},{M6,TT/2},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2
    ,TT/8},{M1,TT/8},{L6,TT/4},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,TT/8},{M1,TT/8},{M2,TT/
    4},{M2,TT/8+TT/16},{M1,TT/8},{M1,TT/4},{M2,TT/4},{M3,TT/4},{M5,TT/4},{M6,TT},{M6,TT/8+TT/
    16},{M5,TT/16},{M3,TT/8},{M5,TT/8},{M6,TT/4},
    {L3,TT/8},{L6,TT/4},{L6,TT/8},{L5,TT/8},{L6,TT/8},{L3,TT/8},{L3,TT/8},{L5,TT/8},{L6,TT/8},{M1,TT/8},
    {L6,TT/8},{L5,TT/8},{L6,TT/4},{L3,TT/8},{L5,TT/8},{M1,TT/4},{M1,TT/8},{M1,TT/8},{M2,TT/8},
    {M2,TT/8},{M1,TT/8},{M2,TT/8},{M3,TT/2},{L3,TT/8},{M2,TT/4},{M2,TT/8},{M1,TT/8},{M2,TT/8},{L
    6,TT/8},{L6,TT/8},{M1,TT/8},{M2,TT/8},{M3,TT/8},{M2,TT/8},{M1,TT/8},{M2,TT/8},{M1,TT/8},{L6,T
    T/8},{M1,TT/8},{M3,TT/8},{M3,TT/8},{M2,TT/8},{M1,TT/8},{M3,TT/4},{L5,TT/4},{L6,TT/2},
    {L3,TT/8},{L6,TT/4},{L6,TT/8},{L5,TT/8},{L6,TT/8},{L3,TT/8},{L3,TT/8},{L5,TT/8},{L6,TT/8},{M1,TT/8},
    {L6,TT/8},{L5,TT/8},{L6,TT/4},{L3,TT/8},{L5,TT/8},{M1,TT/4},{M1,TT/8},{M1,TT/8},{M2,TT/8},
    {M2,TT/8},{M1,TT/8},{M2,TT/8},{M3,TT/2},{L3,TT/8},{M2,TT/4},{M2,TT/8},{M1,TT/8},{M2,TT/8},{L
    6,TT/8},{L6,TT/8},{M1,TT/8},{M2,TT/8},{M3,TT/8},{M2,TT/8},{M1,TT/8},{M2,TT/8},{M1,TT/8},{L6,T
    T/8},{M1,TT/8},{M3,TT/8},{M3,TT/8},{M2,TT/8},{M1,TT/8},{M3,TT/4},{L5,TT/4},{L6,TT/2},
    {L5,TT/4},{M1,TT/2+TT/4},{M1,TT/8},{M2,TT/8},{M3,TT/2+TT/4},{M3,TT/8},{M5,TT/8},{M5,TT/4+T
    T/8},{M3,TT/8},{M2,TT/4},{M1,TT/4},{M2,TT/2},{M2,TT/4+TT/8},{L6,TT/8},{M2,TT/4},{M3,TT/4},
    {M4,TT/8+TT/16},{M5,TT/16},{M4,TT/8},{M3,TT/8},{M2,TT/2},{M5,TT/8},{M5,TT/8},{M3,TT/8},{M
    2,TT/8},{M1,TT/4},{L5,TT/8},{L6,TT/2},
    {L3,TT/8},{M6,TT/4},{M5,TT/4},{M6,TT/4},{M5,TT/8},{M3,TT/8},{M3,TT/4},{L3,TT/8},{M6,TT/4},{M
    5,TT/4},{M6,TT/4},{M5,TT/8},{M6,TT/8},{M6,TT/2},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,T
    T/8},{M1,TT/8},{L6,TT/4},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,TT/8},{M1,TT/8},{M2,TT/4},
    {M2,TT/8+TT/16},{M1,TT/8},{M1,TT/4},{M2,TT/4},{M3,TT/4},{M5,TT/4},{M6,TT},{M6,TT/8+TT/16}
    ,{M5,TT/16},{M3,TT/8},{M5,TT/8},{M6,TT/4}
    {L3,TT/8},{L6,TT/4},{L6,TT/8},{L5,TT/8},{L6,TT/8},{L3,TT/8},{L3,TT/8},{L5,TT/8},{L6,TT/8},{M1,TT/8},
    {L6,TT/8},{L5,TT/8},{L6,TT/4},{L3,TT/8},{L5,TT/8},{M1,TT/4},{M1,TT/8},{M1,TT/8},{M2,TT/8},
    {M2,TT/8},{M1,TT/8},{M2,TT/8},{M3,TT/2},{L3,TT/8},{M2,TT/4},{M2,TT/8},{M1,TT/8},{M2,TT/8},{L
    6,TT/8},{L6,TT/8},{M1,TT/8},{M2,TT/8},{M3,TT/8},{M2,TT/8},{M1,TT/8},{M2,TT/8},{M1,TT/8},{L6,T
    T/8},
    {M1,TT/8},{M3,TT/8},{M3,TT/8},{M2,TT/8},{M1,TT/8},{M3,TT/4},{L5,TT/4},{L6,TT/2},//?????????
    {L5,TT/4},{M1,TT/2+TT/4},{M1,TT/8},{M2,TT/8},{M3,TT/2+TT/4},{M3,TT/8},{M5,TT/8},{M5,TT/4+T
    T/8},{M3,TT/8},{M2,TT/4},{M1,TT/4},{M2,TT/2},{M2,TT/4+TT/8},{L6,TT/8},{M2,TT/4},{M3,TT/4},
    {M4,TT/8+TT/16},{M5,TT/16},{M4,TT/8},{M3,TT/8},{M2,TT/2},{M5,TT/8},{M5,TT/8},{M3,TT/8},{M
    2,TT/8},{M1,TT/4},{L5,TT/8},{L6,TT/2},
```

```

{L3,TT/8},{M6,TT/4},{M5,TT/4},{M6,TT/4},{M5,TT/8},{M3,TT/8},{M3,TT/4},{L3,TT/8},{M6,TT/4},{M5,T
T/4},{M6,TT/4},{M5,TT/8},{M6,TT/8},{M6,TT/2},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,TT/8},
{M1,TT/8},{L6,TT/4},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,TT/8},{M1,TT/8},{M2,TT/4},{M2,TT/
8+TT/16},{M1,TT/8},{M1,TT/4},{M2,TT/4},{M3,TT/4},{M5,TT/4},{M6,TT},{M6,TT/8+TT/16},{M5,TT/16}
,
{M3,TT/8},{M5,TT/8},{M6,TT/4},
{L3,TT/8},{M6,TT/4},{M5,TT/4},{M6,TT/4},{M5,TT/8},{M3,TT/8},{M3,TT/4},{L3,TT/8},{M6,TT/4},{M5,T
T/4},{M6,TT/4},{M5,TT/8},{M6,TT/8},{M6,TT/2},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,TT/8},
{M1,TT/8},{L6,TT/4},{M3,TT/8},{M2,TT/8+TT/16},{M3,TT/16},{M2,TT/8},{M1,TT/8},{M2,TT/4},{M2,TT/
8+TT/16},{M1,TT/8},{M1,TT/4},{M2,TT/4},{M3,TT/4},{M5,TT/4},{M6,TT},{M6,TT/8+TT/16},{M5,TT/16}
,
{M3,TT/8},{M5,TT/8},{M6,TT/4}, //????????????
{0,0},
};
void buzzerQuiet(void);
void buzzerSound(unsigned short usFreq);
void musicPlay(void);
#endif

```

接下来我们看一下主函数设计，代码如下：

```

int main(void)
{
    HAL_Init();
    SystemClock_Config();
    MX_GPIO_Init();
    MX_TIM3_Init();

    while (1)
    {
        musicPlay();
    }
}

```

主函数里很简单，先初始化，然后在死循环里执行 musicPlay() 函数，我们重点看一下这个函数，代码如下：

```
void musicPlay(void)
{
    unsigned char
    i=0; while(1)
    {
        if (MyScore[i].mTime == 0) break;
        buzzerSound(MyScore[i].mName);
        HAL_Delay(MyScore[i].mTime);
        i++;
        buzzerQuiet();
        HAL_Delay(10);
    }
}
```

首先取出 MyScore[i].mName（音调），然后延时这个音调的时长，结束后取下一个音调重复，就达到播放音乐的效果。

16.4 下载验证

我们将编译好的文件下载到 STM32 开发板上，可以听到蜂鸣器播放出的歌曲，在 music.h 里定义了三首歌曲，有兴趣的同学可以试一下另外两首歌曲。

第十七章 汉字显示实验

汉字显示在很多单片机系统都需要用到，少则几个字，多则整个汉字库的支持，更有甚者还要支持多国字库，那就更麻烦了。本章我们将向大家介绍如何用 STM32L151 控制 LCD 显示汉字，然后将汉字显示在 LCD 上面。本章分为如下几个部分：

- 17.1 汉字显示原理简介
- 17.2 硬件设计
- 17.3 软件设计
- 17.4 下载验证

17.1 汉字显示原理简介

常用的汉字内码系统有 GB2312, GB13000, GBK, BIG5（繁体）等几种，其中 GB2312 支持的汉字仅有几千个，完全能满足我们一般应用的要求。

本实例我们将制作三个GB2312 字库，汉字在液晶上的显示其实就是一些点的显示与不显示，这就相当于我们的笔一样，有笔经过的地方就画出来，没经过的地方就不画，以12*12的汉字为例，假设其取模方向为从上到下，从左到右的方向取模，且高位在前，那么其取模原理如图17.1.1 所示：

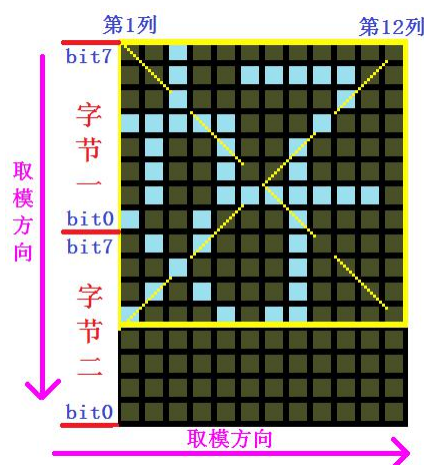


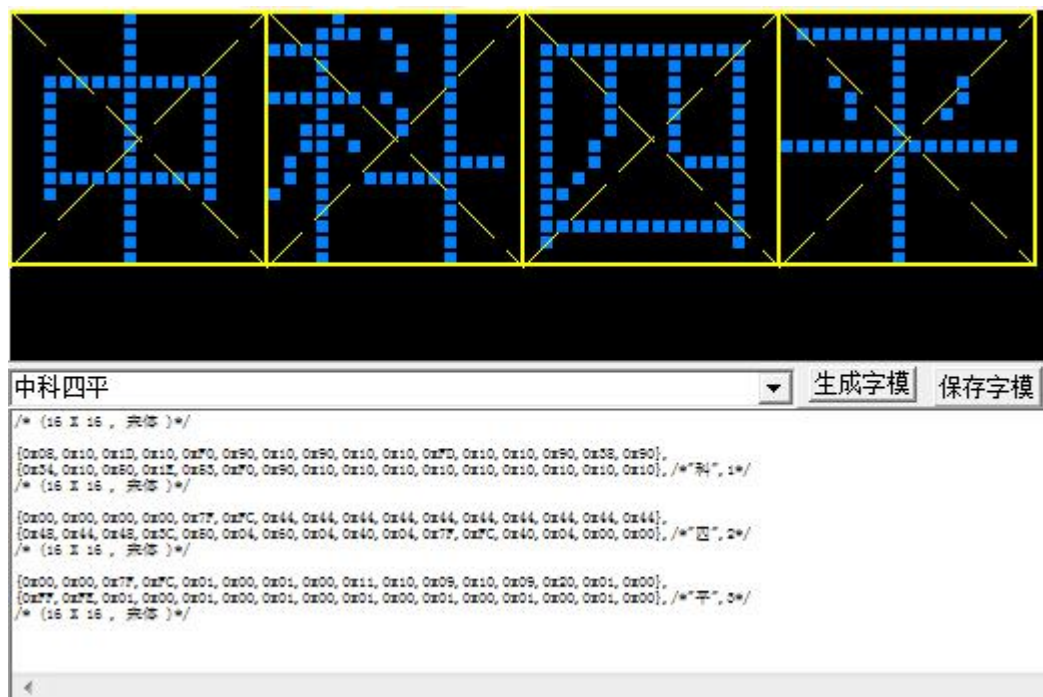
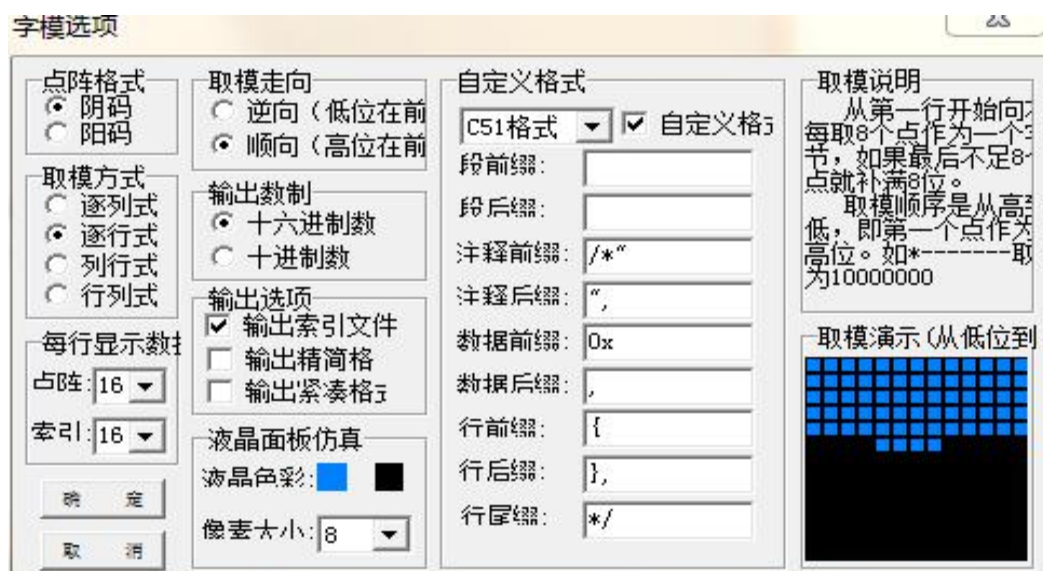
图 17.1.1 从上到下，从左到右取模原理

图中，我们取模的时候，从最左上方的点开始取（从上到下，从左到右），且高位在前（bit7 在表示第一个位），那么第一个字节就是：0X11（1，表示浅蓝色的点，即要画出来的点，0 则表示不要画出来），第二个字节是：0X10，第三个字节（到第二列了，每列 2 个字节）是：0X1E……，依次类推，一个12*12 的汉字，总共有12 列，每列2 个字节，总共需要 24 个字节来表示。

在显示的时候，我们只需要读取这个汉字的点阵数据（12*12 字体，一个汉字的点阵数据为24个字节），然后将这些数据，按取模方式，反向解析出来（坐标要处理好），每个字节，是1的位，就画出来，不是1的位，就忽略，这样，就可以显示出这个汉字了。

所以要显示汉字，我们首先要知道汉字的点阵数据，这些数据可以由专门的软件来生成。

取模软件的生成过程如一下图所示：



17.2 硬件设计

本章实验功能简介：用 16*16, 24*24, 32*32 三种字体显示制作好的汉字。所要用到的硬件资源如下：

1) LCD 模块 在之前的实例中都介绍过了，我们在此就不介绍了。

17.3 软件设计

打开本章实验目录可以看到，文件里有 font.h 代码如下

```
typedef struct
{
    unsigned char Index[2];
    char Msk[32];
}typFNT_GB16;
const typFNT_GB16 tfont16[]=
{
    "中",0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x3F,0xF8,0x21,0x08,0x21,0x08,0x21,0x08,
    0x21,0x08,0x21,0x08,0x3F,0xF8,0x21,0x08,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,
    "科",0x08,0x10,0x1D,0x10,0xF0,0x90,0x10,0x90,0x10,0x10,0xFD,0x10,0x10,0x90,0x38,0x90,
    0x34,0x10,0x50,0x1E,0x53,0xF0,0x90,0x10,0x10,0x10,0x10,0x10,0x10,0x10,0x10,0x10,
    "四",0x00,0x00,0x00,0x00,0x7F,0xFC,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,0x44,
    0x48,0x44,0x48,0x3C,0x50,0x04,0x60,0x04,0x40,0x04,0x7F,0xFC,0x40,0x04,0x00,0x00,
    "平",0x00,0x00,0x7F,0xFC,0x01,0x00,0x01,0x00,0x11,0x10,0x09,0x10,0x09,0x20,0x01,0x00,
    0xFF,0xFE,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,0x01,0x00,
};

typedef struct
{
    unsigned char Index[2];
    char Msk[72];
}typFNT_GB24;
const typFNT_GB24 tfont24[]=
{
    "中",0x00,0x00,0x00,0x00,0x30,0x00,0x00,0x3C,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,
    0x38,0x00,0x38,0x38,0x1C,0x3F,0xFF,0xFC,0x18,0x38,0x38,0x18,0x38,0x38,0x18,0x38,
    0x38,0x18,0x38,0x38,0x18,0x38,0x38,0x18,0x38,0x38,0x3F,0xFF,0xF8,0x38,0x38,0x38,
    0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,
    0x38,0x00,0x00,0x38,0x00,0x00,0x00,0x00,
    "科",0x00,0x00,0x00,0x00,0x00,0x60,0x00,0xE0,0x78,0x07,0xF0,0x70,0x7F,0x00,0x70,0x03,
    0x0E,0x70,0x03,0x07,0x70,0x03,0x03,0x70,0x03,0x70,0x70,0x7F,0xF0,0x70,0x07,0x0C,
    0x70,0x07,0x0E,0x70,0x0F,0xE7,0x70,0x0F,0xE7,0x7E,0x1F,0x60,0x7F,0x3B,0x01,0xF8,
    0x33,0x7F,0x70,0x63,0x60,0x70,0xE3,0x00,0x70,0x07,0x00,0x70,0x07,0x00,0x70,0x07,
    0x00,0x70,0x07,0x00,0x70,0x00,0x00,0x60,
    "四",0x00,0x00,0x00,0x00,0x00,0x00,0x38,0x00,0x18,0x3F,0xFF,0xFC,0x38,0xE7,0x18,0x18,
    0xE7,0x18,0x18,0xE7,0x18,0x18,0xE7,0x18,0x18,0xE7,0x18,0x18,0xE7,0x18,0x18,0xE7,
    0x18,0x18,0xE7,0x18,0x18,0xC7,0x18,0x19,0xC7,0x18,0x19,0xC7,0x18,0x1B,0x87,0xF8,
    0x1F,0x00,0x18,0x1E,0x00,0x18,0x1C,0x00,0x18,0x18,0x00,0x18,0x3F,0xFF,0xFC,0x38,
    0x00,0x1C,0x38,0x00,0x1C,0x00,0x00,0x00,
    "平",0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x38,0x3F,0xFF,0xFC,0x00,0x38,0x0C,0x0C,
    0x38,0x70,0x0E,0x38,0xF0,0x07,0x38,0xE0,0x03,0xB9,0xC0,0x03,0xB9,0x80,0x03,0xBB,
    0x0C,0x00,0x3B,0x1E,0xFF,0xFF,0xF6,0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,
    0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,0x38,0x00,0x00,
    0x38,0x00,0x00,0x38,0x00,0x00,0x00,0x00,
};
```

```
typedef struct
{
    unsigned char Index[2];
    char Msk[132];
}typFNT_GB32;
const typFNT_GB32 tfont32[]=
{
    "中",0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0xC0,0x00,
        0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,
        0x0C,0x03,0x80,0x70,0x0F,0xFF,0xFF,0xF8,0x0E,0x03,0x80,0x70,0x0E,0x03,0x80,0x70,
        0x0E,0x03,0x80,0x70,0x0E,0x03,0x80,0x70,0x0E,0x03,0x80,0x70,0x0E,0x03,0x80,0x70,
        0x0E,0x03,0x80,0x70,0x0E,0x03,0x80,0x70,0x0F,0xFF,0xFF,0xF0,0x0E,0x03,0x80,0x70,
        0x0E,0x03,0x80,0x70,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,
        0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,
        0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x03,0x80,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
    "科",0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0C,0x01,0xC0,0x00,0x3E,0x01,0xE0,
        0x00,0xFE,0x01,0xC0,0x1F,0xE0,0x01,0xC0,0x38,0xE0,0xC1,0xC0,0x00,0xE0,0x71,0xC0,
        0x00,0xE0,0x39,0xC0,0x00,0xE0,0x3D,0xC0,0x00,0xE6,0x19,0xC0,0x00,0xEF,0x01,0xC0,
        0x7F,0xFF,0x81,0xC0,0x01,0xE0,0x01,0xC0,0x01,0xE0,0xE1,0xC0,0x03,0xE0,0x71,0xC0,
        0x03,0xF8,0x79,0xC0,0x07,0xFE,0x39,0xC0,0x07,0xEE,0x39,0xDC,0x0E,0xEE,0x01,0xFE,
        0x0E,0xE0,0x07,0xE0,0x1C,0xE1,0xFD,0xC0,0x38,0xEF,0x81,0xC0,0x70,0xE0,0x01,0xC0,
        0x60,0xE0,0x01,0xC0,0x00,0xE0,0x01,0xC0,0x00,0xE0,0x01,0xC0,0x00,0xE0,0x01,0xC0,
        0x00,0xE0,0x01,0xC0,0x00,0xE0,0x01,0xC0,0x00,0xC0,0x01,0xE0,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
    "四",0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x0C,0x00,0x00,0x70,
        0x0F,0xFF,0xFF,0xF8,0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,
        0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,
        0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,0x0E,0x1C,0x78,0x70,
        0x0E,0x3C,0x78,0x70,0x0E,0x38,0x78,0x70,0x0E,0x38,0x78,0x70,0x0E,0x70,0x78,0x70,
        0x0E,0x70,0x3F,0xF0,0x0E,0xE0,0x1F,0xF0,0x0F,0xC0,0x00,0x70,0x0F,0x80,0x00,0x70,
        0x0F,0x00,0x00,0x70,0x0E,0x00,0x00,0x70,0x0E,0x00,0x00,0x70,0x0F,0xFF,0xFF,0xF0,
        0x0E,0x00,0x00,0x70,0x0E,0x00,0x00,0x70,0x0C,0x00,0x00,0x60,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
    "平",0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0xF0,
        0x1F,0xFF,0xFF,0xF8,0x00,0x03,0xC0,0x18,0x00,0x03,0xC1,0x80,0x03,0x03,0xC1,0xC0,
        0x03,0x83,0xC3,0xE0,0x01,0xC3,0xC3,0x80,0x01,0xE3,0xC7,0x00,0x00,0xF3,0xC7,0x00,
        0x00,0xF3,0xCE,0x00,0x00,0x73,0xDC,0x00,0x00,0x03,0xD8,0x38,0x00,0x03,0xC0,0x7C,
        0x7F,0xFF,0xFF,0xE6,0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,
        0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,
        0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,
        0x00,0x03,0xC0,0x00,0x00,0x03,0xC0,0x00,0x00,0x03,0x00,0x00,0x00,0x00,0x00,0x00,
        0x00,0x00,0x00,0x00,
};
```

```
int main(void)
{

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_SPI2_Init();

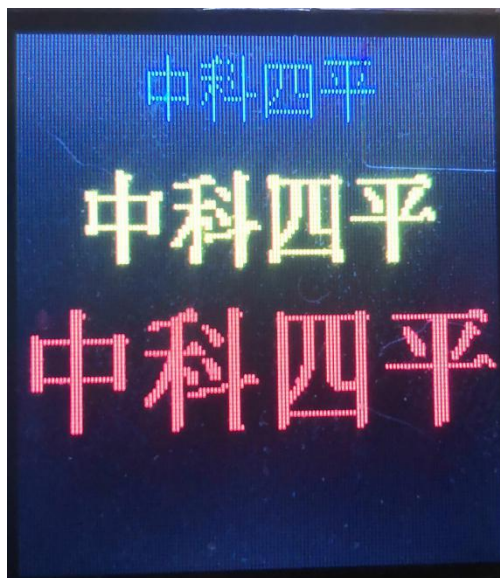
    LCD_Init();
    LCD_Clear(BLACK);

    Show_Str(32,5,BLUE,BLACK,"中科四平",16,1);
    Show_Str(16,32,YELLOW,BLACK,"中科四平",24,1);
    Show_Str(0,64,RED,BLACK,"中科四平",32,1);

    while (1)
    {
```

17.4 下载验证

我们将编译好的文件下载到 STM32 开发板上，可以看到显示如图所示：



第十八章 窄带物联网实验

本章，我们将向大家介绍 STM32L151 开发板的 NB_IOT 模块及其使用。本章分为如下几个部分：

- 18.1 窄带物联网简介
- 18.2 硬件设计
- 18.3 STM32CubeMX 配置和软件设计
- 18.4 下载验证

18.1 窄带物联网简介

18.1.1 窄带物联网简介

NB-IoT (Narrow Band Internet of Things, NB-IoT) 是 IoT 领域基于蜂窝的窄带物联网的一种新兴技术，支持低功耗设备在广域网的蜂窝数据连接，也被叫作低功耗广域网 (LPWA)。NB-IoT 只消耗大约 180KHz 的频段，可直接部署于 GSM 网络、UMTS 网络或 LTE 网络支持待机时间短、对网络连接要求较高设备的高效连接。其主要特点是覆盖广、连接多、速率低、成本低、功耗少、架构优等特点。NB-IOT 使用 License 频段，可采取带内、保护带或独立载波等三种部署方式。

LPWA (低功耗广域网) 又可分为两类：一类是工作于未授权频谱的 LoRa、SigFox 等技术；另一类是工作于授权频谱下，3GPP 支持的 2/3/4G 蜂窝通信技术，比如 EC-GSM、LTECat-m、NB-IoT 等。授权频谱下的代表技术是 NB-IOT，未授权频谱的代表技术是 LoRa (Long Range)。LoRa 是由升特公司 (Semtech) 发布的一种专用于无线电调制解调的技术，LoRa 融合了数字扩频、数字信号处理和前向纠错编码技术，据了解目前全球已经有数百万个物联网节点运用此技术进行相互连接。

NB-IoT 技术是一种 3GPP 标准定义的 LPWA (低功耗广域网) 解决方案，具有以下优势：

1. 海量连接：每小区可达 10 万连接；NB-IoT 比 2G/3G/4G 有 50-100 倍的上行容量提升，这也就意味着，在同一基站的情况下，NB-IoT 可以比现有无线技术提供 50~100 倍的接入数。
2. 超低功耗：电池寿命长达十年；低功耗特性是物联网应用一项重要指标，特别对于一些不能经常更换电池的设备和场合，如安置于高山荒野偏远地区中的各类传感监测设备，它们不可能像智能手机一天一充电，长达几年的电池使用寿命是最本质的需求。在电池技术无法取得突破的前提下，只能通过降低设备功耗以延长电池供电时间。通信设备消耗的能量往往与数据量或速率相关，即单位时间内发出数据包的大小决定了功耗的大小。数据量小，设备的调制解调器和功放就可以调到非常小的水平。NB-IoT 聚焦小数据量、小速率应用，因此 NB-IoT 设备功耗可以做到非常小，可以保障电池 5 年以上的使用寿命。
3. 深度覆盖：能实现比 GSM 高 20db 的覆盖增益；NB-IoT 比 LTE 提升 20dB 增益，相当于发射功率提升了 100 倍，即覆盖能力提升了 100 倍，就算在地下车库、地下室、地下管道等信号难以到达的地方也能覆盖到。
4. 安全性：继承 4G 网络安全能力，支持双向鉴权以及空口严格加密，确保用户数据的安全性；
5. 稳定可靠：能提供电信级的可靠性接入，有效支撑 IoT 应用和智慧城市解决方案。NB-IoT 直接部署于 GSM 网络、UMTS 网络或 LTE 网络，即可与现有网络基站复用以降低部署成本、实现平滑升级，但是使用单独的 180KHz 频段，不占用现有网络的语音和数据带宽，保证传统

业务和未来物联网业务可同时稳定、可靠的进行。以智能抄表应用为例，与采用有线 PLC 抄表数据回收成功率在 60%左右相比，NB-IoT 可以保证数据成功回收率达 99%，可靠性大大提高。

6. 低成本：低速率低功耗低带宽带来的是低成本优势。速率低就不需要大缓存，所以可以缓存小、DSP 配置低；低功耗，意味着 RF 设计要求低，小 PA 就能实现；因为低带宽，就不要复杂的均衡算法。这些因素使得 NB-IoT 芯片可以做得很小。芯片成本往往和芯片尺寸相关，尺寸越小，成本越低，模块的成本也随之变低。

18.1.2 NB_IOT 模块 BC95-B5

接下来我们介绍一下我们的 NB_IOT 模块 BC95-B5 (电信)



BC95 是一款高性能、低功耗的 NB-IoT 无线通信模块。其尺寸仅为 19.9×23.6×2.2mm，能最大限度地满足终端设备对小尺寸模块产品的需求，同时有效地帮助客户减小产品尺寸并优化产品成本。BC95 在设计上兼容移远通信 GSM/GPRS 系列的 M95 模块，方便客户快速、灵活的进行产品设计和升级。BC95 采用更易于焊接的 LCC 封装，可通过标准 SMT 设备实现模块的快速生产，为客户提供可靠的连接方式，特别适合自动化、大规模、低成本的现代化生产方式。SMT 贴片技术也使 BC95 具有高可靠性，以满足复杂环境下的应用需求。

凭借紧凑的尺寸、超低功耗和超宽工作温度范围，BC95 成为 M2M 应用领域的理想选择，常被用于无线抄表、智慧城市、安防、资产追踪、智能家居、农业和环境监测以及其它诸多行业，以提供完善的短信和数据传输服务。

主要特征

频段:BC95-B8: 900MHz

BC95-B5: 850MHz

BC95-B20:

800MHz

封装类型: LCC 脚管

数量:94

供电电压:3.1V~4.2V, 典型值 3.8V

工作温度:-40° C ~ +85° C

模块尺寸:19.9×23.6×2.2mm模块

重量:1.6g

AT 指令控制:3GPP Rel-13 以及增强型 AT 指令

下载方式:UART, Over the Air*SIM 应用工具包

规格参数

数据传输:100bps<bit rate<100kbps (TBC)

协议栈:IPV4/IPV6*UDP/COAP

短信*:点到点短信收发文/PDU 模式

电气特性

输出功率:23dBm

灵敏度:-129dBm

功耗:Sleep: 5uA

接口

SIM/USIM 卡 ×1

UART × 2

ADC* × 1

RESET × 1

天线 × 1

* 表示正在研发中

18.1.3 AT 指令介绍

AT 即 Attention, AT 指令集是从终端设备(Terminal Equipment, TE)或数据终端设备(Data Terminal Equipment, DTE)向终端适配器(Terminal Adapter, TA)或数据电路终端设备(Data Circuit Terminal Equipment, DCE)发送的。通过 TA, TE 发送 AT 指令来控制移动台(Mobile Station, MS)的功能,与 GSM 网络业务进行交互。用户可以通过 AT 指令进行呼叫、短信、电话本、数据业务、传真等方面的控制。

90 年代初, AT 指令仅被用于 Modem 操作。没有控制移动电话文本消息的先例, 只开发了一种叫 SMS BlockMode 的协议, 通过终端设备(TE)或电脑来完全控制 SMS。几年后, 主要的移动电话生产厂商诺基亚、爱立信、摩托罗拉和 HP 共同为 GSM 研制了一整套 AT 指令, 其中就包括对 SMS 的控制。AT 指令在此基础上演化并被加入 GSM07.05 标准以及 GSM07.07 标准, 完全标准化和比较健全的标准。如:对 SMS 的控制共有 3 种实现途径:最初的 BlockMode;基于 AT 指令的 TextMode;基于 AT 指令的 PDUMode。到 PDUMode 已经取代 BlockMode, 后者逐渐淡出。GSM 模块与计算机之间的通信协议是一些 AT 指令集, AT 指令是以 AT 作首, 字符结束的字符串, AT 指令的响应数据包在中。每个指令执行成功与否都有相应的返回。其他的一些非预期的信息(如有人拨号进来、线路无信号等), 模块将有对应的一些信息提示, 接收端可做相应的处理。

接下来我们看一下 BC95_B5 常用的几个 AT 指令:

在启动模块后, 将输出以下字符串:

```
<CR><LF>Neu1<CR><LF>OK<CR><LF>
```

在接收此字符串之后, AT 命令处理器就可以接收 AT 命令了。

AT 指令可以分为以下三类:

第一类, 查询指令使用方法: AT+XX=?

第二类，设置 AT 指令寄存器：AT+XX=XXXX, XXXX, XXXX

第三类，查询 AT 指令寄存器：AT+XX?

AT 命令	描述	实现的状态
3GPP Commands (27.007)		
AT+CGMI	查询制造商 ID	B350+
AT+CGMM	查询模块型号	B350+
AT+CGMR	查询固件版本	B350+
AT+CGSN	查询模块序列号	B350+
AT+CEREG	查询网络注册状态	B350+
AT+CSCON	查询信号连接状态	B350+
AT+CLAC	列出可用命令	B350+
AT+CSQ	获取信号强度	B350+
AT+CGPADDR	显示 PDP 地址	B350+
AT+COPS	选择接入的网络	B350+
AT+CGATT	PS 连接或分离	B350+
AT+CGACT	激活或停用 PDP 上下文	B657SP1+
AT+CIMI	查询国际移动设备身份码	B350+
AT+CGDCONT	定义一个 PDP 上下文	B350+
AT+CFUN	设置终端功能	B350+
AT+CMEE	报告移动终端错误	B600+
AT+CCLK	返回当前日期和时间	B656+

AT+CPSMS	省电模式设置	B657SP1+
AT+CEDRXS	eDRX 设置	B657SP1+
AT+CEER	扩展错误报告	B657SP1+
AT+CEDRXRDP	eDRX 阅读动态参数	B657SP1+
AT+CTZR	时区报告	B657SP1+

ETSI Commands (正在开发中)

AT+CSMS	选择短消息服务	B657SP1+
AT+CNMA	模块消息提醒	B657SP1+
AT+CSCA	服务中心地址	B657SP1+
AT+CMGS	发送短消息	B657SP1+
AT+CMGC	发送短信命令	B657SP1+
AT+CSODCP	通过控制层发送原始数据	B657SP1+
AT+CRTDCP	通过控制层传送终端数据	B657SP1+

General Commands通用命令

AT+NMGS	发送消息到 CDP 服务器	B350+
AT+NMGR	接收 CDP 服务器消息	B350+
AT+NNMI	接收消息标志	B350+
AT+NSMI	发送消息的标志	B350+
AT+NQMGR	查询接收到的消息量	B350+
AT+NQMGs	查询发送的消息量	B350+
AT+NMSTATUS	信息注册状态	B657SP1+
AT+NRB	模块重启	B350+

AT+NCDP	配置和查询 CDP 服务器设置	B350+
AT+NUESTATS	获取的操作统计	B350+
AT+NEARFCN	指定搜索频率	B350+
AT+NSOCR	创建 Socket	B350+
AT+NSOST	发送数据	B350+
AT+NSOSTF	发送有标记数据	B656+
AT+NSORF	接收命令	B350+
AT+NSOCL	关闭 Socket	B350+
+NSONMI	指示 Socket 消息到达(只响应)	B350+
AT+NPING	测试 IP 网络连接到远程主机	B350+
AT+NBAND	设置频段	B600+
AT+NLOGLEVEL	设置日志级别	B600+
AT+NCONFIG	配置模块的功能	B650+
AT+NATSPEED	配置 UART 端口波特率	B656+
AT+NCCID	卡片识别	B657SP1+
AT+NFWUPD	通过 UART 更新固件	B657SP1+
AT+NRDCTRL	控制无线配置	B657SP1+
AT+NCHIPINFO	读取系统信息	B657SP1+
Temporary Commands临时命令		
AT+NTSETID	设置 ID	B350+

以上 AT 指令详解请参考《Quectel_BC95_AT_Commands_Manual_V1.5》

18.1.4 联网过程

下面的示例展示了一个自动附加网络的简单示例。客户只需要查询模块是否通过以下命令连接了网络。

```
AT+NBAND? //确认模块型号
+NBAND:5
OK
AT+CFUN? //返回 1.
+CFUN:1
OK
AT+CIMI //确认 IMSI.
460012345678969
OK
AT+CSQ //确认信号质量
+CSQ:21,99
OK
AT+NUESTATS //确认模块状态
Signal power:-904 Total power:-874 TX power:23
TX time:4322
RX time:17847
Cell ID:256
DL MCS:0
UL MCS:0
DCI MCS:2
ECL:0
SNR:300 EARFCN:2525 PCI:0
OK
AT+CGATT? //确认是否附着网络，如果返回+CGARR:1 表示附着成功，否则等待 30s 继续查询
+CGATT:1
OK
AT+CEREG? //查询网络注册状态，1 表示在网络上注册，2 表示搜索网络。
+CEREG:0,1
OK
AT+CSCON? //查询信号连接状态，1 表示连接，0 表示空闲。
+CSCON:0,1
OK
```

下面展示了手动网络连接过程的两种方式。

1. 不指定 PLMN.

```
AT+CFUN=1           //配置 MT 的功能。
OK

AT+CIMI             //查询 IMSI，并在执行完 AT+CFUN=1 后等待 4
秒。如果 IMSI 返回说明已经识别卡；如未返回，请检查是否为 USIM 卡，
是否插入卡。460012345678966
OK

AT+NBAND?           //确认设备型号
+NBAND:5
OK

AT+CGDCONT=1, "IP", "APN" //APN 是一个本地访问点，它需要相应地配置。
OK

AT+CEREG=1          //设置自动报告网络注册状态，当模块是在网络上注册，将会报告一个
URC
OK

AT+CSCON=1          //设置自动报告网络注册状态，当模块是在网络上注册，将会报告一个
URC
OK

AT+CGATT=1          //激活网络。
OK
+CEREG:2            //报告 URC，MT 目前正在尝试连接或搜索注册。
+CSCON:1            //报告 URC，MT 是连接的。
+CEREG:1            //报告 URC，MT 已注册。
AT+CSQ              //查询当前信号质量。

+CSQ:31,99

OK
AT+NUESTATS         //查询模块的状态。
```

```
Signal power:-904
Total power:-874
TX power:23
TX time:4322
RX time:17847
Cell ID:256
DL MCS:0
UL MCS:0
DCI MCS:2
ECL:0
SNR:300
EARFCN:2525
PCI:0
OK
```

AT+CGATT? //确认是否附着网络，如果返回+CGATT:1 表示附着成功，否则等待 30s 继续查询

```
+CGATT:1
OK
```

AT+CEREG? //查询当前 EPS 网络注册状态：注册。 +CEREG:1,1
OK

AT+CSCON? //查询信号连接状态，1 表示连接，0 表示空闲。+CSCON:1,1
OK

2. 指定 PLMN.

AT+CFUN=1 //配置 MT 的功能。
OK

AT+CIMI //查询 IMSI，并在执行完 AT+CFUN=1 后等待 4 秒。如果 IMSI 返回说明已经识别卡;如未返回，请检查是否为 USIM 卡，是否插入卡。
460012345678966
OK

AT+NBAND? //确认设备型号
+NBAND:5

```

OK
AT+CGDCONT=1,"IP","APN" //APN 是一个本地访问点，它需要相应地配置。
OK
AT+CEREG=1 //设置自动报告网络注册状态，当模块在网络上注册，将会报告 URC。
OK
AT+CSCON=1 //设置自动报告网络注册状态，当模块在网络上注册，将会报告 URC。
OK
AT+COPS=1,2,"46000" //指定 PLMN 搜索或自动搜索，PLMN 需要被客户配置
OK
AT+CSQ //查询当前信号质量。
+CSQ:31,99
OK
AT+NUESTATS //查询模块的状态。
Signal power:-904
Total power:-874
TX power:23
TX time:4322
RX time:17847
CellID:256
DL MCS:0
UL MCS:0
DCI MCS:2
ECL:0
SNR:300
EARFCN:2525
PCI:0
OK
AT+CGATT? //确认是否附着网络,如果返回+CGARR:1 表示附着成功,否则等待
30s 继续查询
+CGATT:1
OK
AT+CEREG? //查询当前 EPS 网络注册状态: 注册。+CEREG:1,1
OK

AT+CSCON //查询信号连接状态, 1 表示连接, 0 表示空闲。+CSCON:1,1

OK
  
```

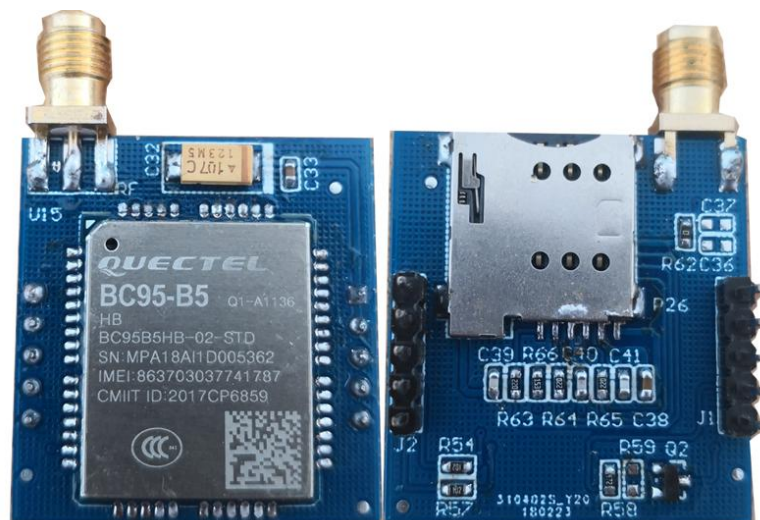

18. 1.5 发送/接收/读取 CoAP 信息

以下是发送、接收和读取 CoAP 消息的一个简单示例。

```
AT+CGSN=1 //查询 IMEI 号码。如果“错误”返回时，客户需要编写 IMEI。
+CGSN: 863703030104298
OK
AT+NCDP=192.53.100.53 //配置 CDP 服务器。此命令必须在执行完+cfun=0 后执行。（服务器可以在附加网络之前配置）
OK
AT+NCDP? //确认 CDP 服务器
+NCDP:192.53.100.53,5683
OK
AT+NSMI=1 //使能发送。
OK
AT+NNMI=2 //使能接收。
OK
AT+NMGS=10, AA7232088D0320623399 //发送数据。
OK
+NSMI:SENT //数据已经发送
AT+NQMGS //确认数据是否发送成功
PENDING=0,SENT=1,ERROR=0
OK
+NNMI //接收到新消息
AT+NQMGR //确认接收到新消息。
BUFFERED=1,RECEIVED=1,DROPPED=0
OK
AT+NMGR //读取消息。
2,AABB
OK
AT+NQMGR //确认消息是否读取成功
BUFFERED=0,RECEIVED=1,DROPPED=0
OK
```

18.2 硬件设计

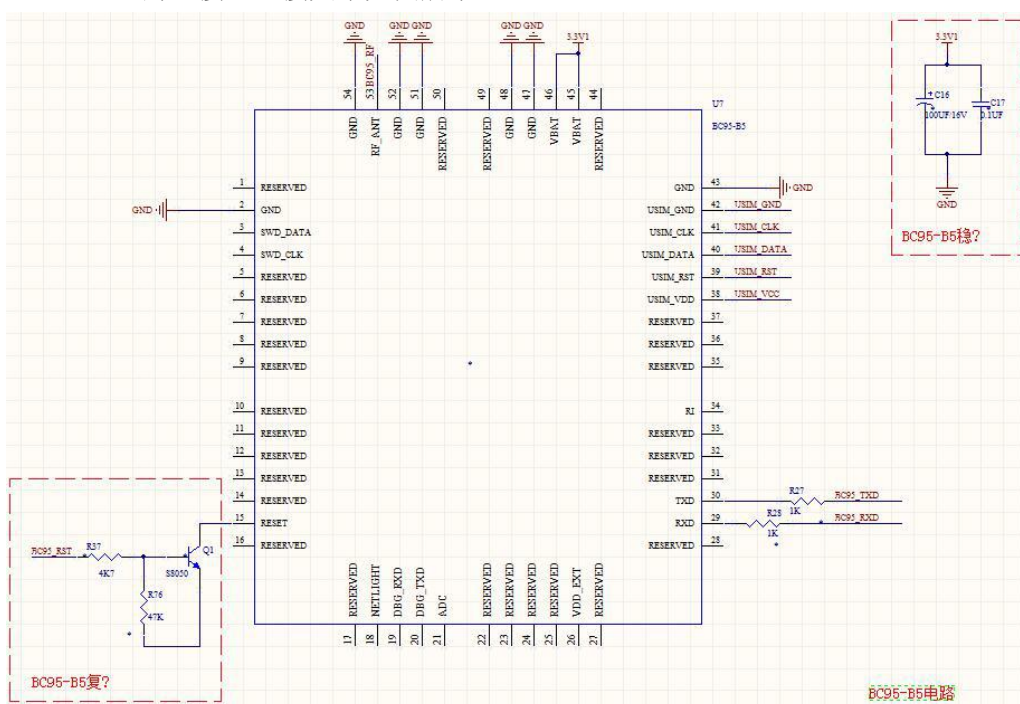
我们先来看一下 NB_IOT 模块，如图所示：

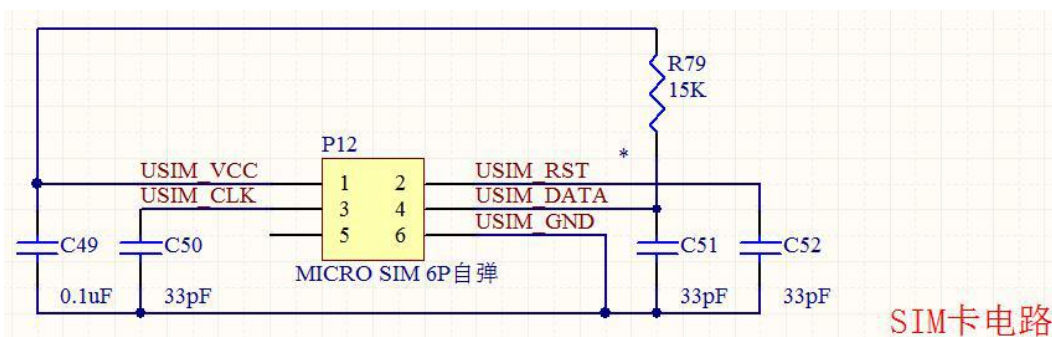
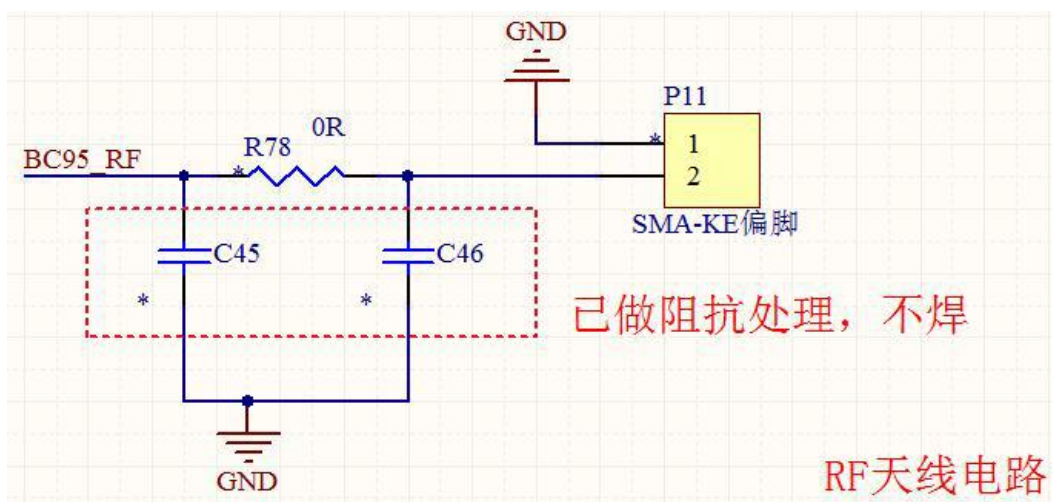


本章需要用到的硬件资源有：

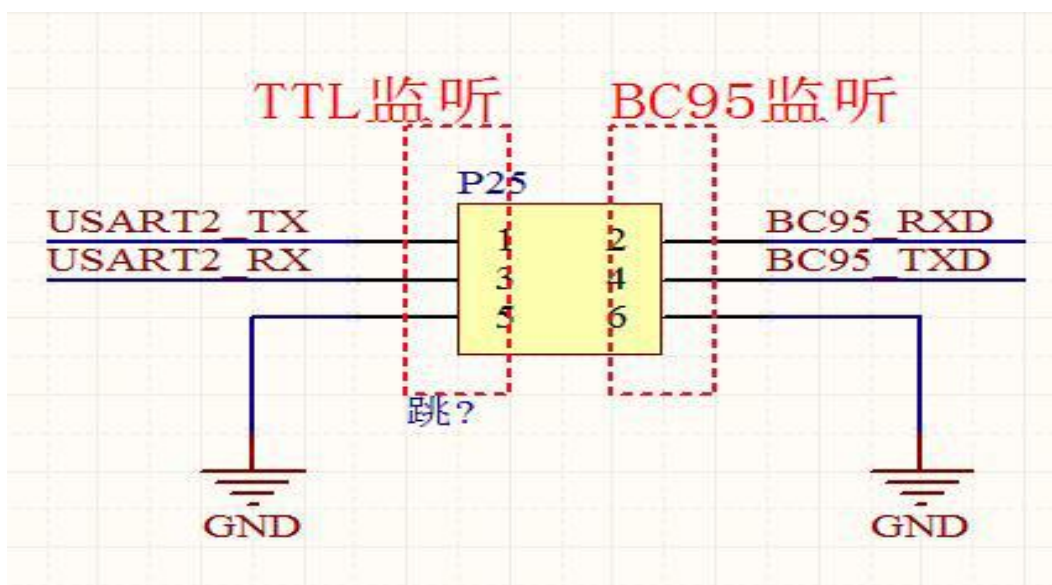
- 1) LCD 模块
- 2) SHT20
- 3) 串口 2
- 4) BC95_B5 模块
- 5) 定时器 2

前面 3 部分的资源，我们前面已经介绍了，请大家参考相关章节。这里只介绍 BC95_B5 与 STM32L151 的连接，连接关系如图所示：





BC95 RST	15	PA0/ADC0
USART2 TX	16	PA1/ADC1/TIME2_CH2
USART2 RX	17	PA2/USART2_TX
	18	PA3/USART2_RX



从上面的原理图可以看出，我们要控制 IOT 模块需要用到一个 IO (BC95_RST) 和串口 2，所以我们只需要在 STM32CubeMX 中配置串口 2 和 PA1 就可以。

18.3 STM32CubeMX 配置和软件设计

本章我们将向大家介绍如何使用 STM32L151 的串口 2 接口实现和 NB_IOT 模块 BC95-B5 之间的通信，读取 SHT20 的温湿度值，将结果显示在 LCD 模块上，并通过 BC95-B5 将数据上传到 IOT 平台上展示。

STM32CubeMX 配置串口，温湿度传感器，LCD 我们已经在前面章节讲过，这里不再赘述，生成工程后我们打开工程，引入 bc95.c 和 bc95.h，代码如下：

```

void BC95_init(void)
{
    while(!BC95_init_OK)
    {
        switch(BC95_init_step)
        {
            case 0:
                if(!BC95_init_SendFlag)
                {
                    USART2_SendStr("AT+CGATT?\r\n"); //查询是否附着网络成功
                    BC95_init_SendFlag=1;
                    BC95_init_delay=30;                //如果查询不到附着网络成功，等待 30S
                    OK_back_flag=0;

                    BC95_init_num++;
                    if(BC95_init_num>2)                //如果查询三次不能自动联网，则转去手动联
网
                {
                    BC95_init_num=0;
                    BC95_init_step=9;
                    BC95_init_SendFlag=0;
                    BC95_init_delay=0;
                    BC95_init_OK=1;
                    BC95_attach_net();
                }
            }
            else
            {
                if(OK_back_flag)                //如果返回了 OK
                {
                    BC95_init_step=1;
                    BC95_init_SendFlag=0;
                    BC95_init_delay=0;
                }
            }
            break;
        case 1:
            if(!BC95_init_SendFlag)
            {
                USART2_SendStr("AT+CSQ\r\n");
                BC95_init_SendFlag=1;
                BC95_init_delay=5;                //如果 5s 内不能返回 ok 则重新发送
                OK_back_flag=0;
            }
        }
    }
}

```

```
else
{
    if(OK_back_flag)
    {
        BC95_init_step=2;
        BC95_init_SendFlag=0;
        BC95_init_delay=0;
    }
}
break;
case 2:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+CGSN=1\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=3;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 3:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NCDP=180.101.147.115,5683\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=4;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
```

```
case 4:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NCDP?\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=5;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 5:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NSMI=1\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=6;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 6:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NNMI=2\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
}
```



```
else
{
    if(OK_back_flag)
    {
        BC95_init_step=7;
        BC95_init_SendFlag=0;
        BC95_init_delay=0;
    }
}
break;
case 7:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+CCLK?\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=8;
            BC95_init_OK=1;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
default:
    break;
}
}
HAL_Delay(500);    //控制每一步的时间，不要太快
}
```

```
void BC95_attach_net(void) 手动联网
{
    while(!BC95_attach_net_OK)
    {
        switch (BC95_init_step)
        {
            case 9:
                if(!BC95_init_SendFlag)
                {
                    USART2_SendStr("AT+CFUN=1\r\n");
                    BC95_init_SendFlag=1;
                    BC95_init_delay=5;
                    OK_back_flag=0;
                }
                else
                {
                    if(OK_back_flag)
                    {
                        BC95_init_step=10;
                        BC95_init_SendFlag=0;
                        BC95_init_delay=0;
                    }
                }
                break;
            case 10:
                if(!BC95_init_SendFlag)
                {
                    USART2_SendStr("AT+NCONFIG=AUTOCONNECT,TRUE\r\n");
                    BC95_init_SendFlag=1;
                    BC95_init_delay=5;
                    OK_back_flag=0;
                }
                else
                {
                    if(OK_back_flag)
                    {
                        BC95_init_step=11;
                        BC95_init_SendFlag=0;
                        BC95_init_delay=0;
                    }
                }
                break;
        }
    }
}
```

```
case 11:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NCONFIG=CR_0859_SI_AVOID,TRUE\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=12;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 12:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NCONFIG=CR_0859_SI_AVOID,TRUE\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=13;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 13:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+NRB\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=30;
    }
    break;
```

```
case 14:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+CIMI\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=15;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 15:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+CGDCONT=1,\"IP\", \"APN\"\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    else
    {
        if(OK_back_flag)
        {
            BC95_init_step=16;
            BC95_init_SendFlag=0;
            BC95_init_delay=0;
        }
    }
    break;
case 16:
    if(!BC95_init_SendFlag)
    {
        USART2_SendStr("AT+CGATT=1\r\n");
        BC95_init_SendFlag=1;
        BC95_init_delay=5;
        OK_back_flag=0;
    }
    }
```

```
else
{
    if(OK_back_flag)
    {
        BC95_init_step=0;
        BC95_attach_net_OK=1;
        BC95_init_SendFlag=0;
        BC95_init_delay=0;

        BC95_init_OK=0;

    }
}
break;
default:
break;
}
HAL_Delay(500);
}
}

//字符串转为 16 进制，发送数据的时候会用到
unsigned char CharToHex(unsigned char bHex)
{
    if((bHex>=0)&&(bHex<=9))
    {
        bHex += 0x30;
    }
    else if((bHex>=10)&&(bHex<=15))//Capital
    {
        bHex += 0x37;
    }
    else
    {
        bHex = 0xff;
    }
    return bHex;
}
```

下面我们讲解一下这段代码：首先我们在 while 循环里等待模块初始化成功，我们定义了一个变量 BC95_init_OK，初始化完成后 BC95_init_OK=1，跳出 while 循环。

```
while(!BC95_init_OK)
{
    省略代码...
}
```

然后在代码中用 switch 控制发送 AT 指令每一步，在 STM32 和 BC95-B5 通讯中 STM32 发送一条 AT 指令，BC95-B5 返回一条参数，所以用 switch 控制每一步指令发送和接收。

```
switch(BC95_init_step)
{
    省略代码...
}
```

接下来我们看一下 AT 指令发送：

```
USART2_SendStr("AT+CGATT?\r\n");
```

我们可以看到发送的 AT 指令其实是一个字符串 AT+CGATT?是 AT 指令查询是否附着网络，\r\n 是回车换行，因为 AT 指令必须使用\r\n 结尾表示命令发送完成。我们看一下一个完成的 AT 指令发送过程：

```
case 2:
if(!BC95_init_SendFlag) //变量控制循环的时候 AT 指令只发送一次
{
    USART2_SendStr("AT+CGSN=1\r\n"); //发送 AT 指
    BC95_init_SendFlag=1; //发送完成
    BC95_init_delay=5; //定时 5s,如果 5s 内收不到 BC95 返回来参数,则重新发
    送OK_back_flag=0;
}
else //AT 命令发送完成，等待解析 BC95 返回的参数
{
    if(OK_back_flag) //AT 指令返回成功并且解析正确
    {
        BC95_init_step=3; //进入下一步发送 AT 指
        BC95_init_SendFlag=0;
        BC95_init_delay=0;
    }
}
break;
```

我们可以看到有一个变量 OK_back_flag，这个变量在串口 2 回调函数中，代码如下：

```
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
    switch(BC95_init_step)
    {
        case 0:
            switch (USART2_Count)
            {
                case 0:
```

```
if(BC95_init_SendFlag)
{
    if(USART2_Data==':')
    {
        USART2_Count=1;
    }
    省略部分代码
case 1:
    if(USART2_Data=='K')
    {
        OK_back_flag=1;
    }
    else
    {
        OK_back_flag=0;
    }
    USART2_Count=0;
    break;
default:
    break;
}
break;
default:
    break;
}
HAL_UART_Receive_IT(&huart2,&USART2_Data,1);
}
void USART2_SendData(unsigned char ch)
{
    HAL_UART_Transmit_IT(&huart2,&ch,1);
    while(__HAL_UART_GET_FLAG(&huart2,UART_FLAG_TC)!=SET);    //等待发送结束
}
void USART2_SendStr(unsigned char * str)
{
    while(*str)
        USART2_SendData(*str++);
}
void USART2_Sendbuf(unsigned char *buf,unsigned char len)
{
    HAL_UART_Transmit_IT(&huart2,buf,len);
    while(__HAL_UART_GET_FLAG(&huart2,UART_FLAG_TC)!=SET);    //等待发送结束
}
```


这样一个完成的 AT 指令发送接收流程就结束了，接下来我们看一下主函数设计，代码如下：

```
int main(void)
{

    HAL_Init();

    SystemClock_Config();

    MX_GPIO_Init();
    MX_RTC_Init();
    MX_SPI2_Init();
    MX_TIM2_Init();
    MX_USART2_UART_Init();

    SHT20_Init();
    HAL_Delay(5000); //延时 5s 等待 BC95 上电稳定
    LCD_Init();
    LCD_Clear(LIGHTBLUE);
    BC95_init();

    while (1)
    {
        Read_SHT20();
        display_firstpage();

        if(!DATA_Send_flag)
        {
            ActiveHeart();
            DATA_Send_flag=1;
            DATA_Send_time=60;
        }
        COAP_data_PRO();
    }
}
```

主函数里设计很简单，在前面是外设初始化，比较重要的是由于 STM32 运行速度很快，为了等待 BC95-B5 稳定，需要一个至少 5s 的延时等待。接着在死循环里读取温湿度值显示，我们定义一个时间 DATA_Send_time=60;就是没 60s 发送一次数据。我们在定时器里控制时间，定时器代码：

```
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if(DATA_Send_time>0)
    {
        DATA_Send_time--;
        if(!DATA_Send_time)
        {
            DATA_Send_flag=0;
        }
    }

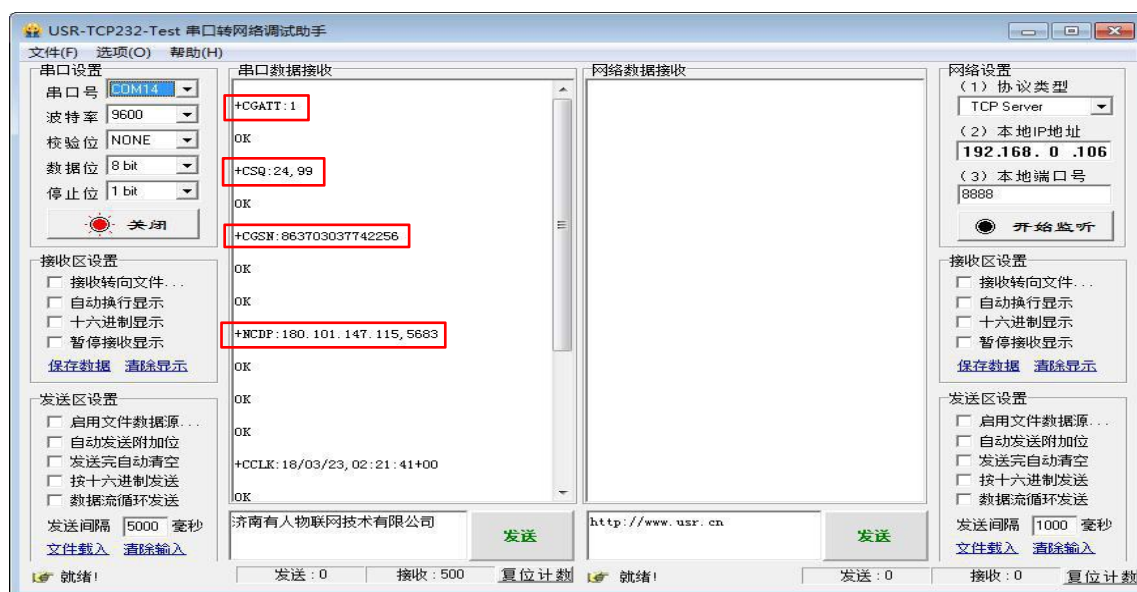
    if(BC95_init_delay)
    {
        BC95_init_delay--;
        if(!BC95_init_delay)
        {
            if(BC95_init_step==13)
            {
                BC95_init_step=14;
            }
            BC95_init_SendFlag=0;
        }
    }
}
```

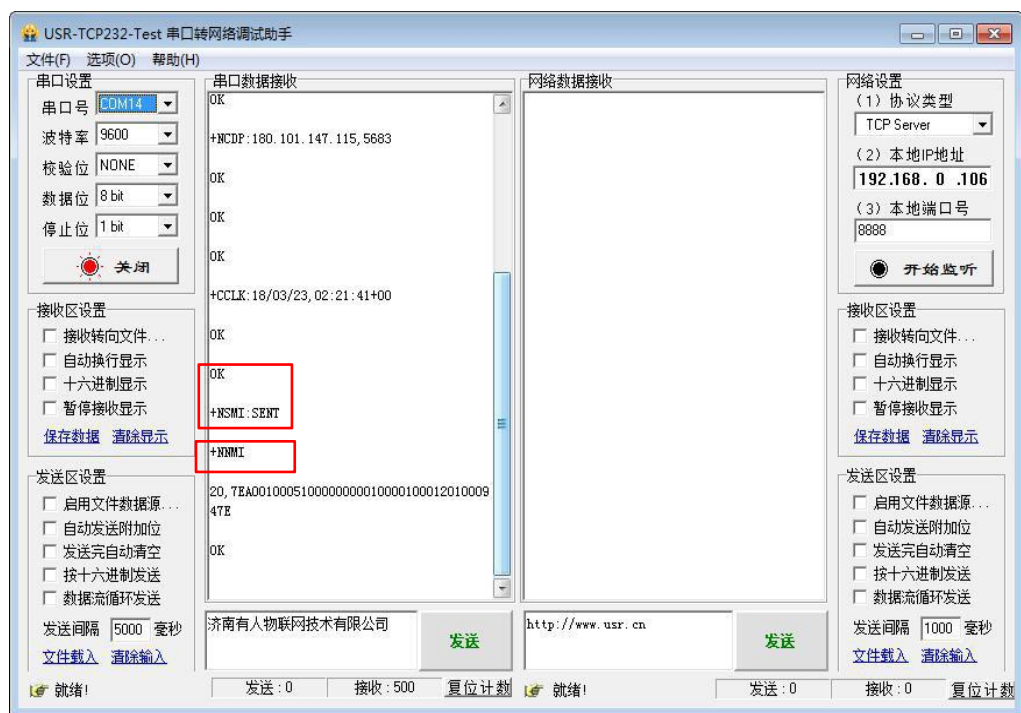
主函数里每 60s 发送一次数据

BC95-B5 初始化控制每一步超时时间

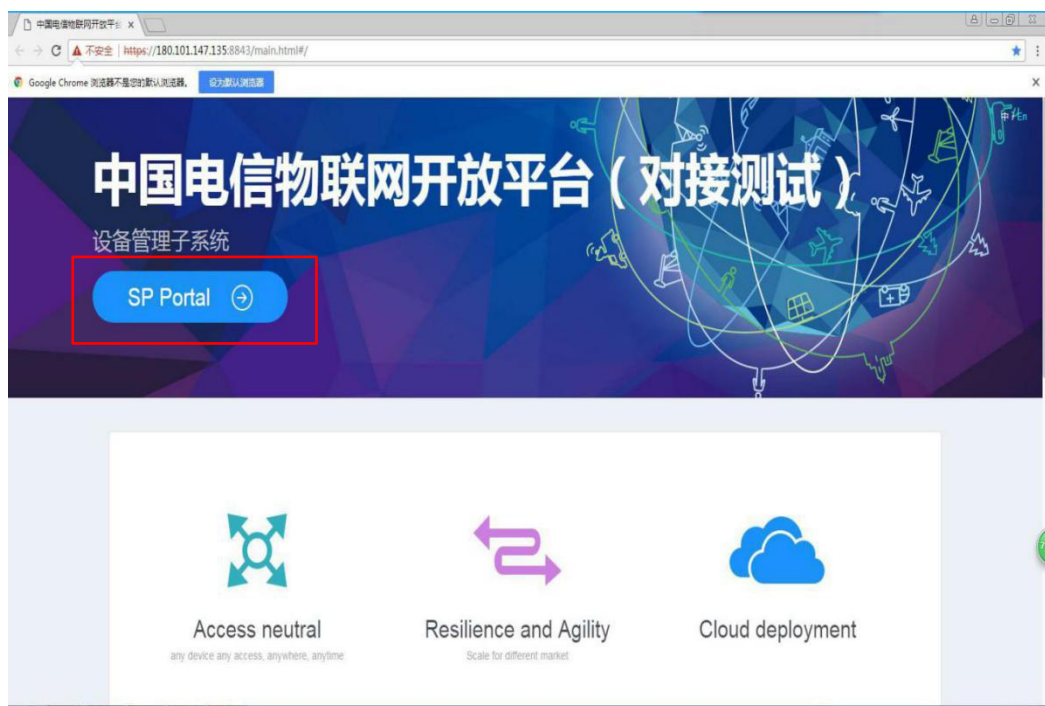
18.4 下载验证

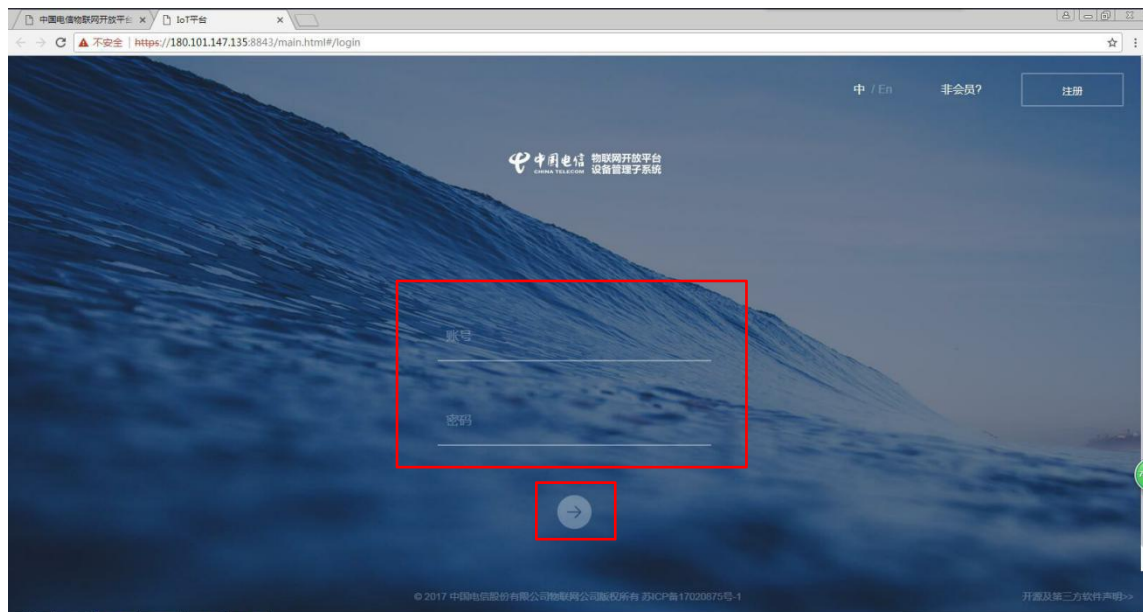
我们将编译好的文件烧写到 STM32 开发板上，连接好 USB-TTL 监听 BC95 返回的数据，我们可以看到 BC95 返回的数据，如图所示：

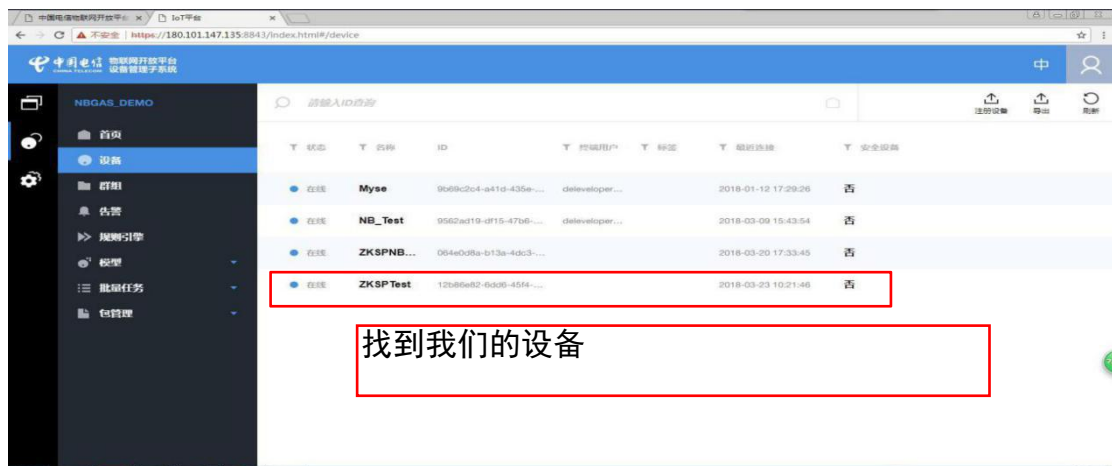
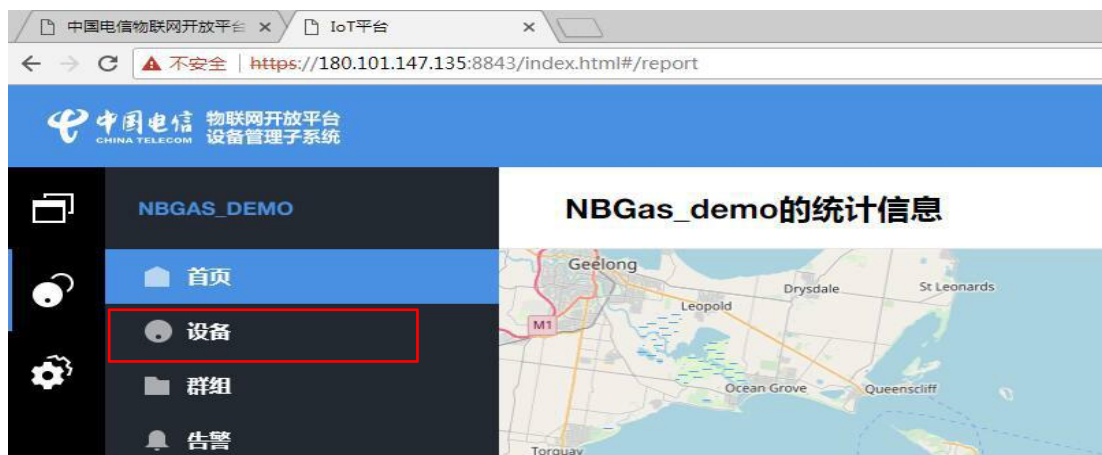




我们可以看到红框内的主要参数已经返回，并且每一步都配置成功，说明我们的初始化过程很成功，接下来我们再去 IOT 平台看一下数据显示是否正常，我们登陆电信 IOT 平台网站 <http://180.101.147.135:8843>，我们推荐用火狐浏览器登陆，否则有可能会被浏览器拦截。







这里牵扯到一些平台对接问题，我们不在这里讲解，有兴趣的同学可以去看《IOT 平台对接手册》。到这里为止，我们已经可以把 STM32 采集到的数据通过 BC95-B5 上传到 IOT 平台，这就完成了物联网的最基本的数据上传。