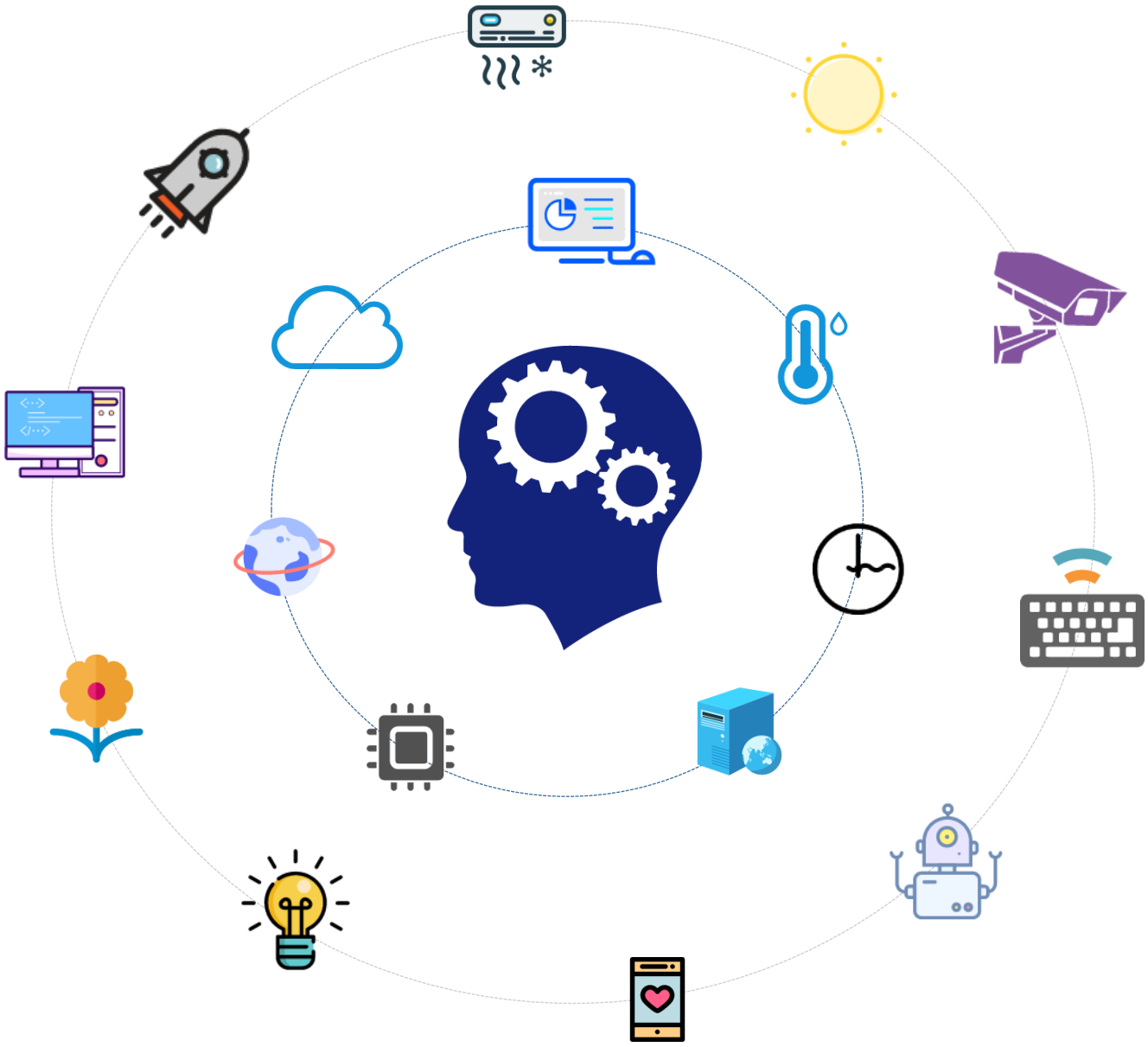
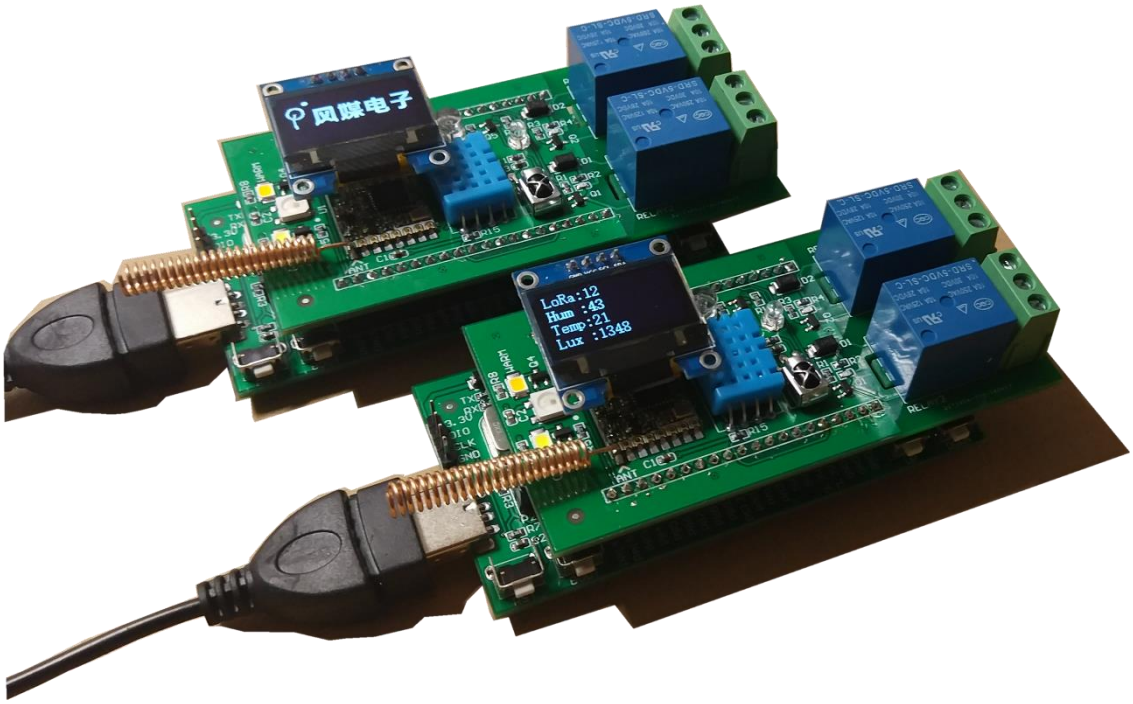


STM32 物联网实战教程

（STM32+计算机网络+项目实战）

版本	内容	作者	说明
Ver1.0	编著	ZX	编著最初版本/2018 年 2 月 1 日
Ver1.1	修正	ZX	修正格式/2018 年 4 月 18 日
Ver1.2	修正	ZX	删除冗余、修正语法及错别字/2018 年 4 月 20 日
Ver1.3	修正	ZX	完善细节/2018 年 4 月 20 日
Ver1.4	修正	ZX	完善出厂检验步骤、修改 WIFI 固件烧录说明/2018 年 5 月 15 日
*	*	*	*



前言

关于教程

《STM32 物联网实战教程》集成了单片机教学、计算机网络以及物联网实战这三部分。配合着风媒电子出品的青柚 ZERO 物联网开发板以及配套的丰富的例程和资料，使得该教程非常适合各大高校信息专业学生以及电子爱好者入门单片机和物联网，并快速开发出自己的物联网项目。

风媒电子的教学特点是以服务为核心，硬件为基础，着重打造适合初学者入门的一整套教学服务。这些服务包括可进行 LoRa 组网的易于扩展的青柚 ZERO 物联网开发板、数十个轻松有趣的实验例程、12 万字 400 余页的教程，以及风媒提供的在线售后支持（官方博客留言、微信公众号、QQ 群等）。

1. 硬件

目前市面上的单片机开发板种类繁多，但适合初学者学习的却很少，这些开发板普遍存在的问题是：通过销售硬件盈利，对软件和教学资源支持较差，一般只会提供原理图和官方的一些英文资料，情况好一些的可能会提供在网上下载的或者使用代码生成工具生成的 STM32 范例，这些代码往往存在着编程风格不规范、无注释或全英文注释等问题，而且这些资料还必须购买开发板才可以获取，这么做无疑是让消费者承担购买风险，除此之外对初学者来说还会造成学习上的额外负担，更不用说售后的服务了，往往这类开发板只适合有经验的开发者购买，其目的是以更低的成本快速搭建自己的项目而并非用于学习。

另外物联网也是近几年的热点，很多高校也都开设了物联网相关的专业，但是市面上却没有一款真正用于物联网教学的提供完整且丰富教学资源和物联网实战项目的开发板，市面上现存的所谓物联网开发板都是只提供几个案例但对于细节和原理却没有详细讲解，正因为初学者存在这样的学习痛点，所以风媒电子推出了一整套的用于系统学习物联网的教学资源，而我们的青柚 ZERO 物联网开发板则是这些资源的硬件载体。

青柚 ZERO 物联网开发板是一套可拆分、可进行 LoRa 组网、可以发送邮件和微信的物联网硬件开发平台，它集成了 WIFI 联网模组、LoRa 模组以及常用的传感器（环境温湿度、光照强度）和输出外设（RGB、高亮 LED、继电器、红外发射等），使其可以应用到各种物联网项目当中，快速构建自己的产品 Demo，比如：智能灯泡，远程灌溉，远程监控等等。关于青柚 ZERO 的详细介绍可阅读第三章。

2. 例程

风媒电子出品的实验例程是我们整套产品的亮点之一。在编写文档的过程中按照难易交替的规律向前推进，并且保证安排在前面的章节不会引用后面还没有学到的知识，而后面的章节则尽可能的调用前面讲到的内容，目的是复习前面学过的知识。同时为了增加学习的趣味性，我们大多数的实验例程都是围绕一个有趣实用的小项目来展开的，比如使用

ADC 实现模拟示波器、使用 RTC 实现带闹钟功能的万年历、使用 USB 控制 windows 经典游戏——3 维弹球、使用 PWM 驱动蜂鸣器弹奏两只老虎、通过 ESP8266 发送电子邮件以及灯光的色温亮度和颜色调节等。

同时该例程提供了非常规范的代码编码风格以及大量的中文注释，使读者无需教程文档配合也可以读懂程序，这么做的目的只有一个，就是希望让初学者养成良好的代码编程风格。

3. 教程

教程是例程的扩充，在教程中主要讲解单片机各个外设的工作流程以及例程中各个函数的运行原理。本套教程在截稿之时累计字数已经达到 12 万，400 余页，共计 38 章教程，这 38 章教程按照内容将其分为三大部分：

第一部分是 STM32 的学习，其内容为各个外设及其驱动程序的讲解，在第三十一章结束，该部分就是一套完整的 STM32 教程，如果对物联网不感兴趣，则只学习该部分即可。

第二部分讲解的是物联网的基础，即计算机网络相关的基础知识，比如：IP，DNS，域名，TCP、UDP 等等，该部分并没有过多深入讲解 TCP/IP 协议族是如何实现的，而是完全以应用为出发点进行的系统的教学。该部分在第三十五章结束，其中 TCP/IP 的知识主要集中在第三十三章。

第三部分是实战部分，有了前面单片机和网络的知识作为技术依托，我们便可以很轻松的实现一些物联网或非物联网的项目，在该部分带领大家将开发板接入第三方物联网云平台，并通过云平台实现几个非常经典的物联网案例，这些案例即可以作为学习用途也完全可以将其单独的作为一个产品来对待（只需要一个公模外壳即可），比如：智能灯泡或者植宠精灵。

但这些并不是该教程的全部，在后续我们还会加入更多的实战项目和单片机例程来丰富本套教程内容，为了保证不对现有的章节造成影响，风媒电子决定将后续新填入的章节进行统一的特殊命名，可参考格式为：【补充教程 x_y】，x 为教程编号，y 为教程标题。

风媒电子希望本套教程能够成为学习单片机和物联网的百科全书，这就要保证文档的开放性，所以本套教程完全开源，可任意在网络上传播、下载，但不可用于盈利或者商业目的，如果在博客或者期刊、论文中引用了本套教程内容请标明出处，最终解释权归风媒电子所有。同时也欢迎大家踊跃投稿或者将自己有创意的想法告诉我们，您的来搞或者想法一经采纳将会登载在本教程中。另外本教程由于体积庞大，难免会出现疏漏或者表达错误，希望大家发现后及时指正并反馈给我们，您的建议是我么前进的动力。

4. 售后

为了提供更加完善的用户体验，我们开设了官方博客（www.fengmeitech.club）、微信公众号以及 QQ 群方便大家交流讨论，如果在学习过程中遇到问题，可以 CTRL+鼠标左键点击页眉链接部分进入到风媒官方博客并在相应章节下留言即可，我们会在第一时间回复您的问题。

如遇到问题建议大家到博客中留言，QQ 群只是用于我们日常交流，如果将问题反馈到 QQ 群中将有可能会被其他消息淹没。

联系我们

淘宝店铺

风媒官网

微信公众号

QQ 群：632055240（#1 群）
资料下载地址：http://fengmeitech.club/download/
官方博客：http://fengmeitech.club/
业务联系：z132269836（微信）
淘宝店铺：https://shop149414327.taobao.com

***最新资料以及动态请关注官方博客和微信公众号**

目录

单片机入门篇

第一章 STM32 介绍

1.1 什么是单片机

学习单片机之前我们当然要知道什么是单片机。对单片机通俗的定义是：“它是一台功能被阉割的低成本专用型计算机，我们可以通过程序来人为的控制它并代替人类重复工作”。

计算机大家都很熟悉，它有输入输出设备（键鼠，显示器等其他接口外设），CPU，内存，硬盘。映射到单片机，输入输出设备就是 IO 口，各种通讯接口（USB，UART,SPI 等），计算机中的 CPU 对应的就是单片机中的 CPU，用来执行指令和对数据进行处理，内存对应单片机的 RAM，它是掉电数据丢失的随机存储器，用来存储 CPU 计算的结果和需要执行的程序指令，是硬盘和 CPU 之间的缓冲桥梁，由于 RAM 的读取速度比硬盘快很多，因此它的存在是用来解决处理速度瓶颈的难题，因为我们可以事先把执行的指令从硬盘读到内存，然后 CPU 读取内存中的指令并执行（对于 PC 机的 CPU 来说还要经过速度更快的 cache）。计算机中的硬盘对应单片机的 ROM，它是掉电数据不丢失的只读存储器，通常用来储存我们写给单片机的程序固件，即单片机的“To Do List”。

我们平时使用计算机的某个应用软件来实现特定功能，比如播放音乐，其本质还是 CPU 去执行存储在硬盘当中的这个软件的代码，只不过现代计算机向上封装了操作系统，驱动层，GUI 等，使得计算机开发者能够完全与硬件隔离，缩短开发周期，也给用户提供了友好的操作界面，并且有足够高的处理速度和足够大存储容量来安装或者运行各种应用，比如即时通讯软件，浏览器等，因此可以将其称之为通用计算机。但是对于单片机开发则不同，我们通过编程来使单片机实现特定的功能，应用于特定的场合，比如每隔一段时间采集一次传感器数据然后发送到服务器，这就是一种专用计算机，因此通常也称这种设备为嵌入式设备。这是两者用途上的区别，但是都属于计算机体系架构，两者同根同源。

经过这样一番解释相信大家已经很直观的了解到单片机的概念了。

1.2 STM32 系列单片机介绍

STM32 单片机是意法半导体（ST）公司的生产的基于 ARM Cortex M3 架构的 32 位精简指令集单片机。下面我们简单了解一下 ST 公司旗下的所有单片机分类。

下图是 STM32 系列单片机的选型表：

图 1.1　STM32 系列单片机选型表

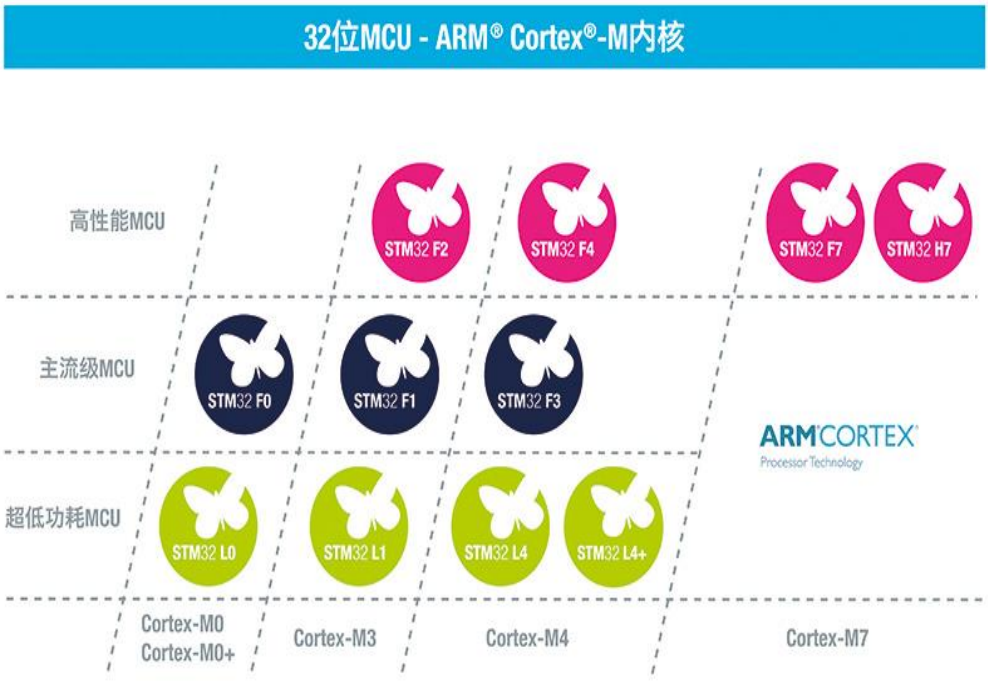


图 1.2　STM32 系列单片机架构分类

图 1.1 列出了 STM32 系列的所有单片机，ST 为开发者提供尽可能多的单片机选择，无论你是开发高性能比如图像处理或者音视频解码的项目还是开发需要极低功耗的项目，都可以在 STM32 家族中找到适合你的一款。当然，ST 还提供了其他系列的单片机，比如 8 位的 STM8 单片机和车用的 32 位 SPC56 单片机。

下面这幅图是 ST 所有的单片机分类。

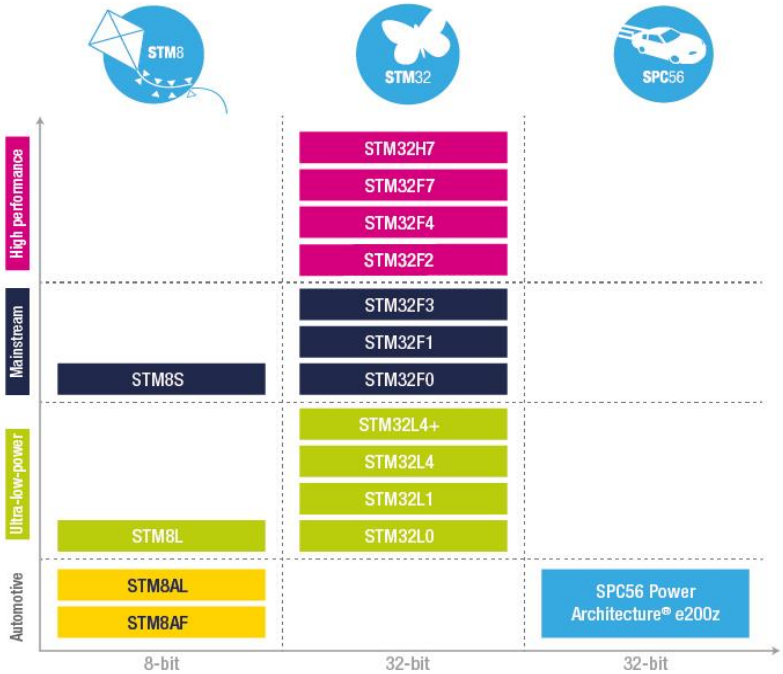


图 1.3　ST 单片机型号汇总

这几年国内也出现了很多 ARM 公司授权的基于 ARM Cortex M3 架构的单片机，而且无论是引脚还是程序都完全兼容 STM32，而且价格更便宜。比如纳瓦特的 NV32F100x 单片机、兆易创新的 GD32F1/F2/F4 系列单片机，对于这些型号单片机，大家有兴趣可以查阅官方提供的资料，这里就不过多描述。

1.3 STM32F103 系列单片机介绍

通过上面的章节，我们了解了 ST 旗下的单片机分类，这一章节将详细讲解一下青柚 ZERO 开发板的主角——STM32F103 系列单片机。

在了解 STM32F103 之前先看一下 ST 单片机的命名规则

图 1.4 ST 系列单片机命名规则

从上图就可以很容易看出 STM32 单片机是如何命名的了。

目前按照 flash 大小和外设种类数量的多少把 STM32F103 系列单片机分为三类：

- 小容量型

  STM32F103x46 这一类为小容量单片机

- 中等容量型

  STM32F103x8B 这一类为中等容量单片机

- 大容量型

  STM32F103xCDE 这里类为大容量单片机

做这些分类的原因是 ST 可以为开发者提供尽可能多的选择余地。小容量和大容量单片机是中等容量的延伸，主要体现在封装、价格、片上硬件资源上。这三类唯一的区别就是 flash，RAM 的大小和外设多少，比如大容量的自然就比小容量和中等容量的单片机有更大

的 flash 和 RAM 空间以及更多的外设数量和种类，比如 FSMC。但在引脚和功能上是完全兼容的，也就是说在小容量单片机上跑的程序不需要更改任何代码即可烧录到中等容量或者大容量单片机上面跑(但是要通过宏定义告诉编译器此时开发者用的哪种容量的，这个后面章节会讲到)。反之亦然（前提是小容量有对应的外设）。

在介绍完 STM32F103 系列单片机之后，我们再来具体讲解一下青柚 ZERO 开发板上使用的单片机——STM32F103C8T6 的硬件资源：

- 最高 72MHz 的工作频率，其运算速度远远高于 51，AVR 和 MSP430 等单片机
- 高达 64KB 闪存，20KB SRAM
- 3 个通用定时计数器（TIM2/3/4/5）
- 1 个高级定时计数器（TIM1）
- 1 个滴答定时计数器
- 1 个 RTC 实时时钟
- 1 个独立看门狗定时计数器
- 1 个窗口看门狗定时计数器
- 2 个 SPI 通讯接口
- 2 个 I2C 通讯接口
- 3 个 UART 通讯接口
- 1 个全速 USB2.0 接口（仅支持作为 SLAVE 设备）
- 1 个 CAN 接口（2.0B 主动）
- 37 个 GPIO 端口
- 2*10 路 12 位 ADC
- 7 通道 DMA 控制器
- 支持 SWD & JTAG 仿真接口

接下来通过更加直观的方式向大家展示 STM32 整体的架构框图：

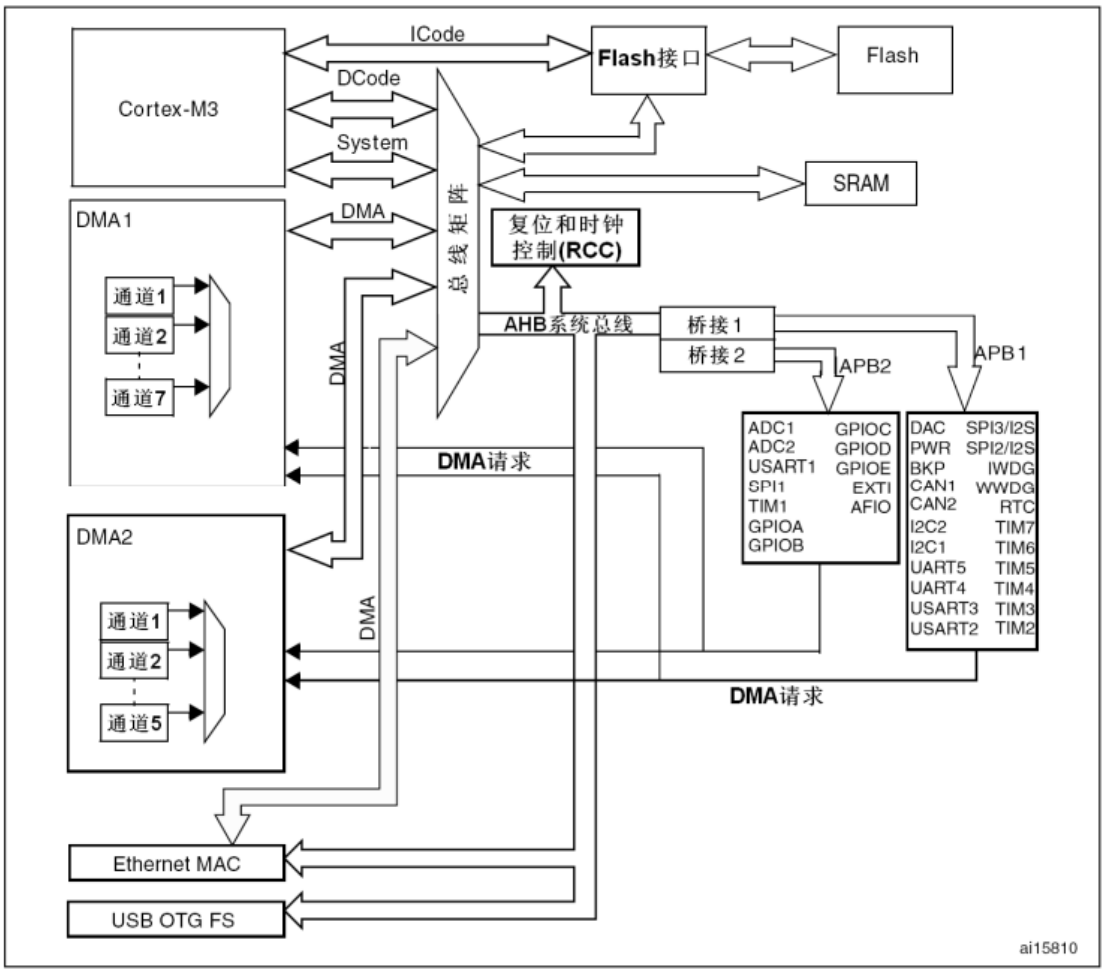


图 1.5　STM32 架构框图

通过这张图就可以很清晰的看到 STM32103F103 整体的框图结构，需要大家关注的就是 AHB 总线下的 APB1 和 APB2 上挂载的各个外设，并且 STM32 单片机在开发时要在使用相关外设之前使能该外设的时钟才可以工作，这和平时用到的 51 单片机操作方式不同，这样做的好处是用户可以自行控制设备功耗，因为时钟增多直接导致功耗增大，如果我们只开启用到外设的时钟就可以降低功耗，如果大家想在继续了解可查阅 STM32F10x 的参考手册的第二章——存储器和总线架构。

1.4 STM32F103 单片机时钟系统

这一小节带领大家了解一下 STM32 的时钟系统，下图是 STM32F10x 时钟树框图：

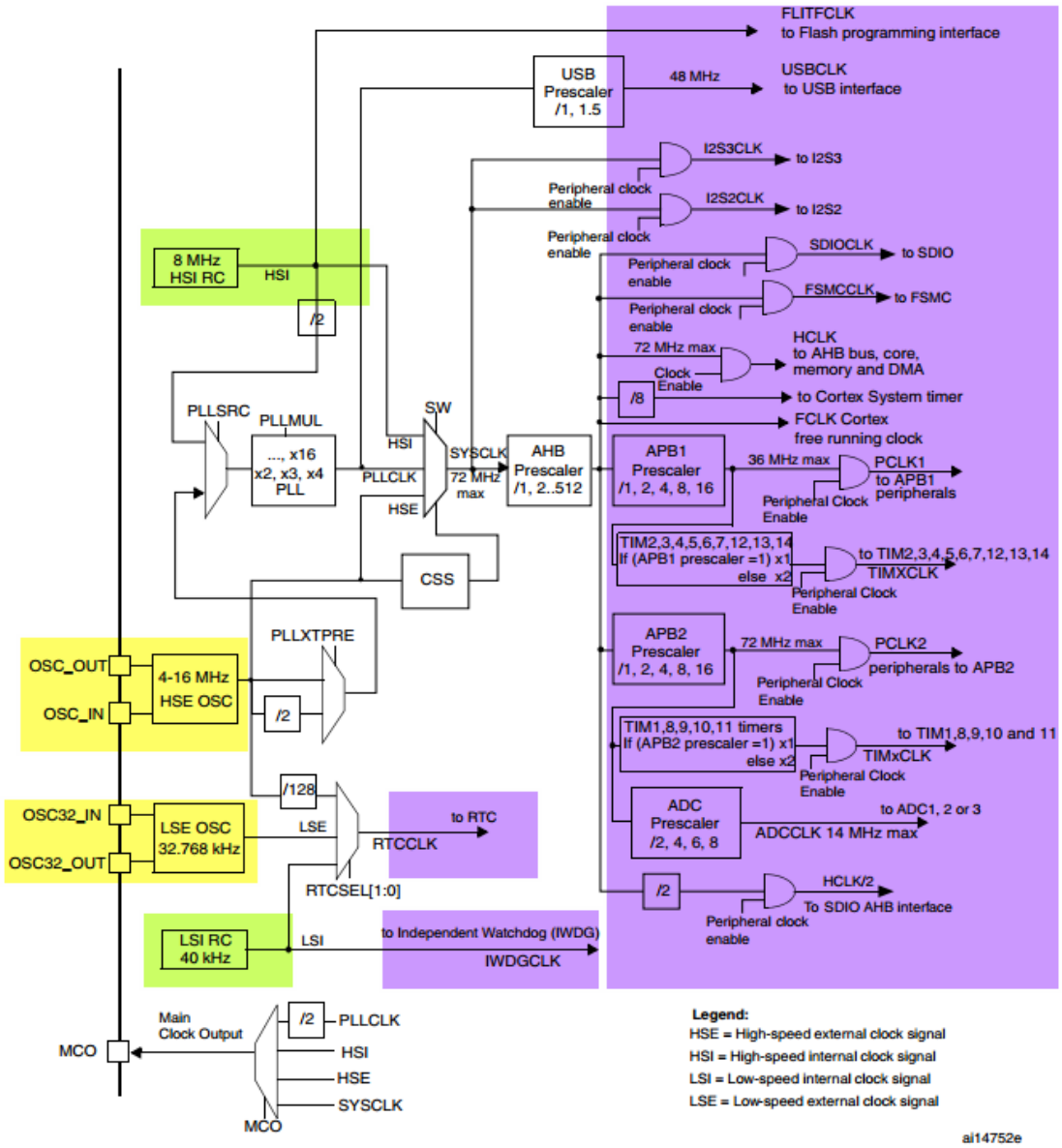


图 1.6　STM32 时钟树

作者对整个时钟系统进行了简单的分类。图 1.6 从左到右左侧依次是时钟输入，即产生和调制（倍频/分频）时钟的作用。右侧则是时钟信号应用的部分，比如左侧时钟信号经过一系列的分频、倍频最终把时钟信号输出给对应的外设使用。

先说一下左半边时钟输入部分，即黄色部分。STM32 有 4 个时钟源分别是高速外部时钟源（HSE），高速内部时钟源（HSI），低速外部时钟源（LSE），低速内部时钟源（LSI），下面通过一个框图直观感受一下：

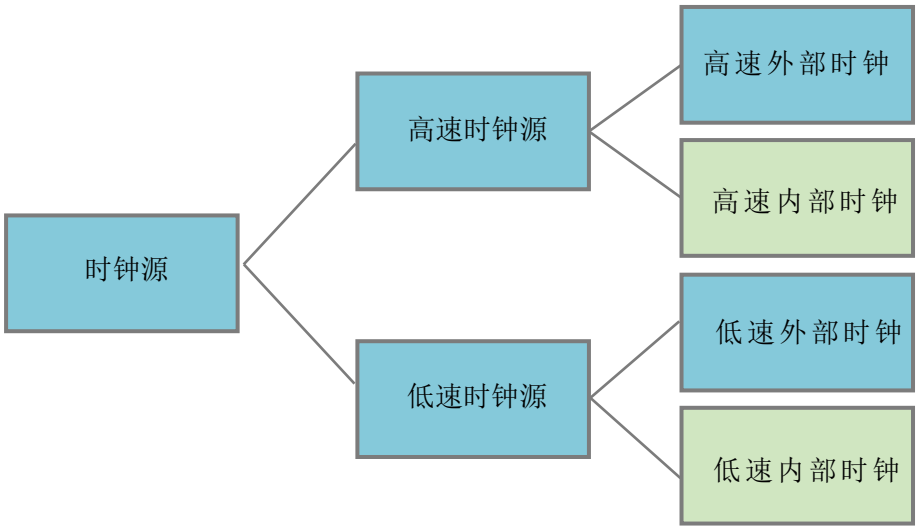


图 1.7　STM32 时钟源分类

这里先和大家解释时钟的概念。人类生活的世界可以用 4 维坐标表示，分别是决定物体位置的横纵坐标和高度，另外一个就是时间维度，没有了时间，所有物体都将静止。所以不难理解，单片机中的时钟就是单片机世界里的时间，只有时钟一刻不停的重复变化，才会使得程序得以执行。如果去掉时钟（假设硬件不会报错），那么单片机即使上电也不会执行任何指令，因为没有时间“推动”它。

下面再来普及一下何为内部时钟和外部时钟。顾名思义，内部时钟就是单片机内自带的 RC 震荡电路产生的时钟信号，对于不需要严格时序通信的情况下，使用内部时钟可以降低设备成本。但是内部时钟的缺点也很明显，就是温度的变化对它的时钟频率有很大影响。另外，如果使用低速内部时钟为 RTC 提供时钟源，那么不同温度下时间的误差就变得很大了。因此内部时钟一般应用在时序要求不高的场合。

外部时钟与内部时钟相反，外部时钟采用有源或者无源晶振产生稳定的时钟信号，相比于内部 RC 时钟，外部时钟源温漂不明显，稳定性更好。

1.5 STM32F103 单片机存储映像

STM32 除了拥有丰富的外设和灵活的时钟系统之外，还拥有 4G 的存储映射空间，见下图：

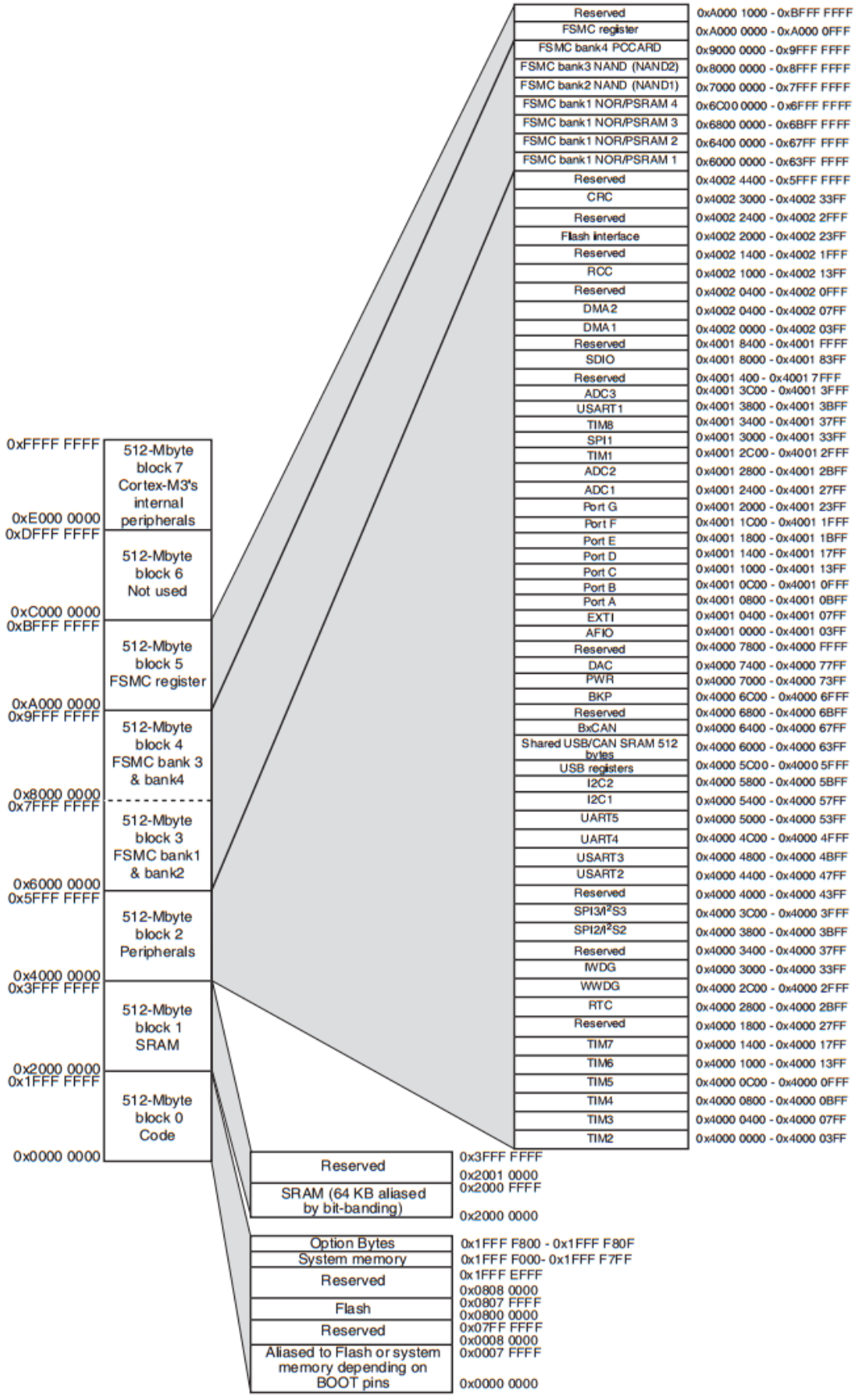

0xFFFF FFFF
512-Mbyte block 7 Cortex-M3's internal peripherals
0xE000 0000
0xDFFF FFFF
512-Mbyte block 6 Not used
0xC000 0000
0xBFFF FFFF
512-Mbyte block 5 FSMC register
0xA000 0000
0x9FFF FFFF
512-Mbyte block 4 FSMC bank 3 & bank4
0x8000 0000
0x7FFF FFFF
512-Mbyte block 3 FSMC bank1 & bank2
0x6000 0000
0x5FFF FFFF
512-Mbyte block 2 Peripherals
0x4000 0000
0x3FFF FFFF
512-Mbyte block 1 SRAM
0x2000 0000
0x1FFF FFFF
512-Mbyte block 0 Code
0x0000 0000
Reserved
0xA000 1000 - 0xBFFF FFFF
FSMC register
0xA000 0000 - 0xA000 0FFF
FSMC bank4 PCCARD
0x9000 0000 - 0x9FFF FFFF
FSMC bank3 NAND (NAND2)
0x8000 0000 - 0x8FFF FFFF
FSMC bank2 NAND (NAND1)
0x7000 0000 - 0x7FFF FFFF
FSMC bank1 NOR/PSRAM 4
0x6C00 0000 - 0x6FFF FFFF
FSMC bank1 NOR/PSRAM 3
0x6800 0000 - 0x6BFF FFFF
FSMC bank1 NOR/PSRAM 2
0x6400 0000 - 0x67FF FFFF
FSMC bank1 NOR/PSRAM 1
0x6000 0000 - 0x63FF FFFF
Reserved
0x4002 4400 - 0x5FFF FFFF
CRC
0x4002 3000 - 0x4002 33FF
Reserved
0x4002 2400 - 0x4002 2FFF
Flash interface
0x4002 2000 - 0x4002 23FF
Reserved
0x4002 1400 - 0x4002 1FFF
RCC
0x4002 1000 - 0x4002 13FF
Reserved
0x4002 0400 - 0x4002 0FFF
DMA2
0x4002 0400 - 0x4002 07FF
DMA1
0x4002 0000 - 0x4002 03FF
Reserved
0x4001 8400 - 0x4001 FFFF
SDIO
0x4001 8000 - 0x4001 83FF
Reserved
0x4001 400 - 0x4001 7FFF
ADC3
0x4001 3C00 - 0x4001 3FFF
USART1
0x4001 3800 - 0x4001 3BFF
TIM8
0x4001 3400 - 0x4001 37FF
SPI1
0x4001 3000 - 0x4001 33FF
TIM1
0x4001 2C00 - 0x4001 2FFF
ADC2
0x4001 2800 - 0x4001 2BFF
ADC1
0x4001 2400 - 0x4001 27FF
Port G
0x4001 2000 - 0x4001 23FF
Port F
0x4001 1C00 - 0x4001 1FFF
Port E
0x4001 1800 - 0x4001 1BFF
Port D
0x4001 1400 - 0x4001 17FF
Port C
0x4001 1000 - 0x4001 13FF
Port B
0x4001 0C00 - 0x4001 0FFF
Port A
0x4001 0800 - 0x4001 0BFF
EXTI
0x4001 0400 - 0x4001 07FF
AFIO
0x4001 0000 - 0x4001 03FF
Reserved
0x4000 7800 - 0x4000 FFFF
DAC
0x4000 7400 - 0x4000 77FF
PWR
0x4000 7000 - 0x4000 73FF
BKP
0x4000 6C00 - 0x4000 6FFF
Reserved
0x4000 6800 - 0x4000 6BFF
BxCAN
0x4000 6400 - 0x4000 67FF
Shared USB/CAN SRAM 512 bytes
0x4000 6000 - 0x4000 63FF
USB registers
0x4000 5C00 - 0x4000 5FFF
I2C2
0x4000 5800 - 0x4000 5BFF
I2C1
0x4000 5400 - 0x4000 57FF
UART5
0x4000 5000 - 0x4000 53FF
UART4
0x4000 4C00 - 0x4000 4FFF
USART3
0x4000 4800 - 0x4000 4BFF
USART2
0x4000 4400 - 0x4000 47FF
Reserved
0x4000 4000 - 0x4000 43FF
SPI3/I²S3
0x4000 3C00 - 0x4000 3FFF
SPI2/I²S2
0x4000 3800 - 0x4000 3BFF
Reserved
0x4000 3400 - 0x4000 37FF
IWDG
0x4000 3000 - 0x4000 33FF
WWDG
0x4000 2C00 - 0x4000 2FFF
RTC
0x4000 2800 - 0x4000 2BFF
Reserved
0x4000 1800 - 0x4000 27FF
TIM7
0x4000 1400 - 0x4000 17FF
TIM6
0x4000 1000 - 0x4000 13FF
TIM5
0x4000 0C00 - 0x4000 0FFF
TIM4
0x4000 0800 - 0x4000 0BFF
TIM3
0x4000 0400 - 0x4000 07FF
TIM2
0x4000 0000 - 0x4000 03FF
Reserved
0x3FFF FFFF
0x2001 0000
SRAM (64 KB aliased by bit-banding)
0x2000 FFFF
0x2000 0000
Option Bytes
0x1FFF F800 - 0x1FFF F80F
System memory
0x1FFF F000- 0x1FFF F7FF
Reserved
0x1FFF EFFF
0x0808 0000
Flash
0x0807 FFFF
0x0800 0000
Reserved
0x07FF FFFF
0x0008 0000
Aliased to Flash or system memory depending on BOOT pins
0x0007 FFFF
0x0000 0000

图 1.8　STM32 存储映射

由上图可以看出，STM32 存储器分为了 8 个 512MB 的存储块，加在一起一共 4GB 的存储空间，这 4GB 存储空间包括了程序存储器，数据存储器，各个外设寄存器以及很大一部分的预留空间，方便用户扩展。这里需要关心的是外设寄存器的存储地址，知道了这些地址才能对寄存器进行操作。

下面是各个外设的地址，方便大家开发时查阅用：

Boundary address	Peripheral	Bus	Register map
0xA000 0000 - 0xA000 0FFF	FSMC	AHB	*Section 21.6.9 on page 563*
0x5000 0000 - 0x5003 FFFF	USB OTG FS		*Section 28.16.6 on page 912*
0x4003 0000 - 0x4FFF FFFF	Reserved		-
0x4002 8000 - 0x4002 9FFF	Ethernet		*Section 29.8.5 on page 1069*
0x4002 3400 - 0x4002 7FFF	Reserved		-
0x4002 3000 - 0x4002 33FF	CRC		*Section 4.4.4 on page 65*
0x4002 2000 - 0x4002 23FF	Flash memory interface		-
0x4002 1400 - 0x4002 1FFF	Reserved		-
0x4002 1000 - 0x4002 13FF	Reset and clock control RCC		*Section 7.3.11 on page 120*
0x4002 0800 - 0x4002 0FFF	Reserved		-
0x4002 0400 - 0x4002 07FF	DMA2		*Section 13.4.7 on page 288*
0x4002 0000 - 0x4002 03FF	DMA1		
0x4001 8400 - 0x4001 FFFF	Reserved		-
0x4001 8000 - 0x4001 83FF	SDIO		*Section 22.9.16 on page 620*

图 1.9　STM32 外设寄存器地址_1

Boundary address	Peripheral	Bus	Register map
0x4001 5800 - 0x4001 7FFF	Reserved	APB2	-
0x4001 5400 - 0x4001 57FF	TIM11 timer		*Section 16.5.11 on page 467*
0x4001 5000 - 0x4001 53FF	TIM10 timer		*Section 16.5.11 on page 467*
0x4001 4C00 - 0x4001 4FFF	TIM9 timer		*Section 16.4.13 on page 457*
0x4001 4000 - 0x4001 4BFF	Reserved		-
0x4001 3C00 - 0x4001 3FFF	ADC3		*Section 11.12.15 on page 251*
0x4001 3800 - 0x4001 3BFF	USART1		*Section 27.6.8 on page 828*
0x4001 3400 - 0x4001 37FF	TIM8 timer		*Section 14.4.21 on page 362*
0x4001 3000 - 0x4001 33FF	SPI1		*Section 25.5 on page 742*
0x4001 2C00 - 0x4001 2FFF	TIM1 timer		*Section 14.4.21 on page 362*
0x4001 2800 - 0x4001 2BFF	ADC2		*Section 11.12.15 on page 251*
0x4001 2400 - 0x4001 27FF	ADC1		
0x4001 2000 - 0x4001 23FF	GPIO Port G		*Section 9.5 on page 193*
0x4001 1C00 - 0x4001 1FFF	GPIO Port F		
0x4001 1800 - 0x4001 1BFF	GPIO Port E		
0x4001 1400 - 0x4001 17FF	GPIO Port D		
0x4001 1000 - 0x4001 13FF	GPIO Port C		
0x4001 0C00 - 0x4001 0FFF	GPIO Port B		
0x4001 0800 - 0x4001 0BFF	GPIO Port A		
0x4001 0400 - 0x4001 07FF	EXTI		*Section 10.3.7 on page 213*
0x4001 0000 - 0x4001 03FF	AFIO		*Section 9.5 on page 193*

图 1.10　STM32 外设寄存器地址_2

Boundary address	Peripheral	Bus	Register map
0x4000 7800 - 0x4000 FFFF	Reserved		-
0x4000 7400 - 0x4000 77FF	DAC		*Section 12.5.14 on page 272*
0x4000 7000 - 0x4000 73FF	Power control PWR		*Section 5.4.3 on page 79*
0x4000 6C00 - 0x4000 6FFF	Backup registers (BKP)		*Section 6.4.5 on page 84*
0x4000 6400 - 0x4000 67FF	bxCAN1		*Section 24.9.5 on page 695*
0x4000 6800 - 0x4000 6BFF	bxCAN2		
0x4000 6000(1) - 0x4000 63FF	Shared USB/CAN SRAM 512 bytes		-
0x4000 5C00 - 0x4000 5FFF	USB device FS registers		*Section 23.5.4 on page 652*
0x4000 5800 - 0x4000 5BFF	I2C2		*Section 26.6.10 on page 785*
0x4000 5400 - 0x4000 57FF	I2C1		
0x4000 5000 - 0x4000 53FF	UART5		*Section 27.6.8 on page 828*
0x4000 4C00 - 0x4000 4FFF	UART4		
0x4000 4800 - 0x4000 4BFF	USART3		
0x4000 4400 - 0x4000 47FF	USART2		
0x4000 4000 - 0x4000 43FF	Reserved		-
0x4000 3C00 - 0x4000 3FFF	SPI3/I2S	APB1	*Section 25.5 on page 742*
0x4000 3800 - 0x4000 3BFF	SPI2/I2S		*Section 25.5 on page 742*
0x4000 3400 - 0x4000 37FF	Reserved		-
0x4000 3000 - 0x4000 33FF	Independent watchdog (IWDG)		*Section 19.4.5 on page 498*
0x4000 2C00 - 0x4000 2FFF	Window watchdog (WWDG)		*Section 20.6.4 on page 505*
0x4000 2800 - 0x4000 2BFF	RTC		*Section 18.4.7 on page 492*
0x4000 2400 - 0x4000 27FF	Reserved		-
0x4000 2000 - 0x4000 23FF	TIM14 timer		*Section 16.5.11 on page 467*
0x4000 1C00 - 0x4000 1FFF	TIM13 timer		
0x4000 1800 - 0x4000 1BFF	TIM12 timer		*Section 16.4.13 on page 457*
0x4000 1400 - 0x4000 17FF	TIM7 timer		*Section 17.4.9 on page 480*
0x4000 1000 - 0x4000 13FF	TIM6 timer		
0x4000 0C00 - 0x4000 0FFF	TIM5 timer		*Section 15.4.19 on page 422*
0x4000 0800 - 0x4000 0BFF	TIM4 timer		
0x4000 0400 - 0x4000 07FF	TIM3 timer		
0x4000 0000 - 0x4000 03FF	TIM2 timer		

图 1.11　STM32 外设寄存器地址_3

关于 STM32F103 单片机的中断部分，我们会在中断部分详细讲解，这里不做过多说明。

第二章 初识物联网

2.1 物联网存在的意义

青柚 ZERO 是一款基于 STM32 单片机的物联网开发板，所以在认识了 STM32 单片机之后，我们再来了解一下另一个主角——物联网（Internet of things，缩写为 IoT）。

物联网，顾名思义就是让“物体”连接网络，用户可以远程查看和控制这些物体。这里的物体泛指一切我们能够看到的东西，比如灯泡，家用电器，衣物，交通工具等等，一旦这些物体连接网络，用户对它的控制范围和使用权限就大大增加，用户体验也就会更好。这里举几个经典的案例，这些案例同时也是青柚 ZERO 实战的项目。

第一个案例，如果用户长期外出，比如出差或者度假，那么家中的植物，宠物就会无人照料，这时物联网设备就会派上用场。只要用户的设备连接到了外网就可以在任何时间任何地点通过手机来远程控制家中物联网设备进行定时定量的食物投放，浇花施肥的动作，而且通过设备上的传感器还可以采集家中温湿度、光照，然后用户依据这些参数来控制家中的空调或者自动窗帘，使得植宠生活环境达到最优状态。

第二个案例，用户外出或者上班途中突然想起家里的电暖气和照明灯没有关，这种情况下只好匆忙的返回家中查看，也因此导致上班迟到。但是家中有了物联网硬件之后一切就会变得美好。用户需要做的就是优雅的拿出手机，打开客户端，查看家中各个家用电器的状态，如果有异常就动动手指把它们关掉，另外也可以查看家中各个传感器状态，比如可燃气体传感器，门窗传感器，人体红外传感器，一旦传感器抛异常，用户就可以根据情况迅速做出处理。另外在下班的途中依然可以控制家中空调提前加热或制冷，控制电饭煲熬制自己喜欢喝的粥，控制榨汁机提前榨出果汁。因此有了物联网设备就真正的实现了“Life in my hand!”。

第三个案例，在智能农业领域，比如温室大棚的用户普遍存在的痛点就是：耕种设备经常丢失，早晚都要走很远去卷帘关帘浇水施肥。当然现在市面是也推出了 433MHz 或者 315MHz 的远程控制遥控器，但是它的控制范围有限，并不适用与所有用户，另外传输距离也会因为电量或者环境差的原因失灵，在出现意外时往往不能及时到达现场，造成严重损失。如果是使用物联网设备，情况就大不一样。在国内几乎不会存在运营商网络没有覆盖的地方。因此我们可以使用 GPRS 联网（使用 SIM 卡的流量业务），来远程对设备状态进行跟踪和控制。如果人体红外传感器异常，则可以判定有人进入用户的检测范围，此时可以报警或者静悄悄的抓个现行，另外通过网关我们可以远程控制十几公里范围内的多个温室大棚，这种优势是所有非物联网设备所不能达到的。

像这几年很火的共享经济，比如共享充电桩，共享单车（摩拜单车,ofo 等），共享汽车等，其实都是物联网应用的一个场景，其原理就是采集 GPS 地理位置然后报告给服务器，服务器再把这些位置推送到 APP 上，然后单车根据服务器发送过来的支付结果对车锁进行开或关。所以从共享单车大面积普及可以预见：想打造一款被用户广泛接受的物联网产品，重要的并不仅仅是技术本身，更是一个更加超前的创意！

2.2 图说物联网模型

目前物联网模型根据设备联网距离的不同可分成两类，一类就是在图 2.1 展示的远距离物联网模型，也是本套教程学习的模型，另外一类是图 2.4 展示的模型，即近距离物联网模型，该模型主要应用在可穿戴设备上，硬件设备一般会长时间贴身使用，比如智能手环或者智能跑鞋，下面我们分别进行详细讲解。

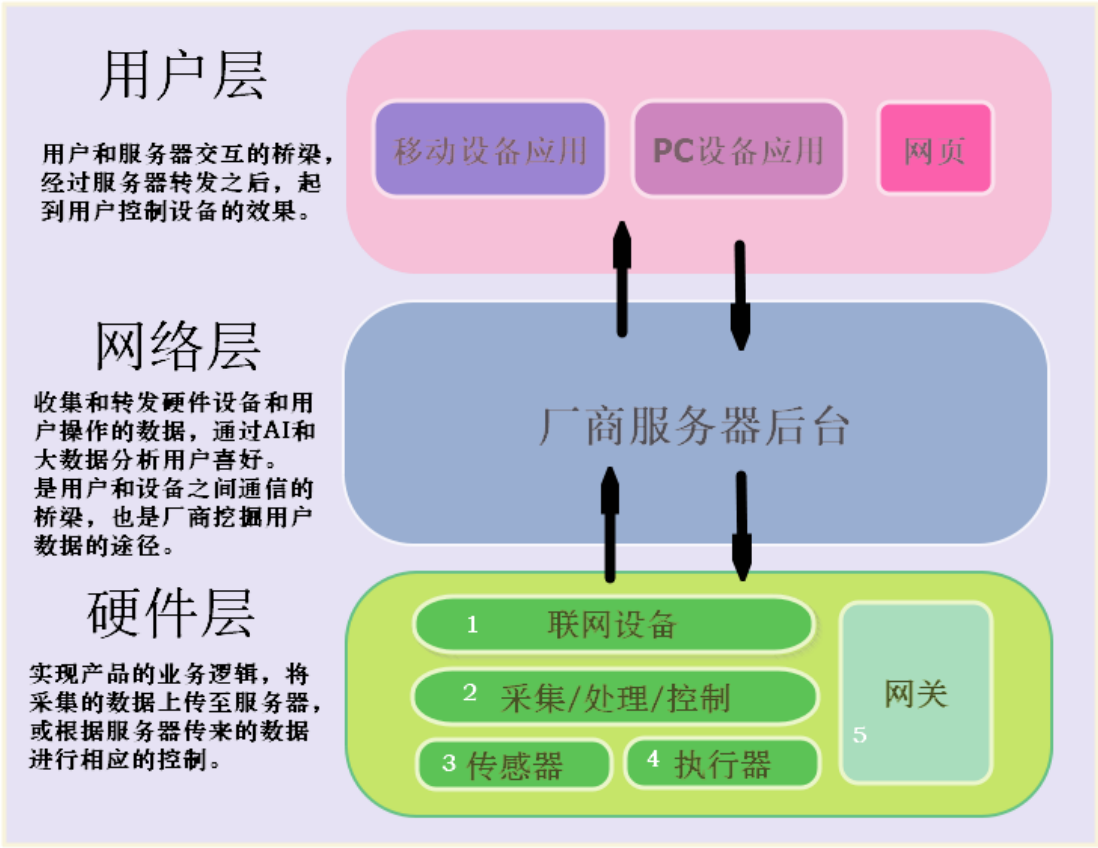


图 2.1　远距离通信物联网模型

从图中可以很清晰的看出一个完成的物联网产品都由哪些部分组成。

我们先从**硬件层**说起，硬件层指的是用户买到家中的物联网硬件产品，它用来实现产品的业务逻辑。我们看到硬件层被细分为更多的子类，**当然根据开发方式的不同分类方式也有区别**，这里举例说明。假设我们开发远程浇花设备，那么采集环境温湿度，土壤温湿度，光照度等这些参数就属于硬件层当中的传感器部分（标号‘3’）；设备根据我们发送的浇花指令吸合继电器使水泵通电，根据我们补光指令控制补光灯亮暗，这部分就是执行器部分（标号‘4’）；对传感器数据的采集、处理计算的部分就是单片机来实现的也就是标号‘2’的部分，此时设备并没有打通本地硬件设备到服务器的通道，因为还需联网，因此单片机还要控制联网设备将采集到的数据发送至服务器或者通过联网设备将服务器发来的控制数据获取到并经过处理后控制执行器动作。当然标号‘1’和‘2’部分并不一定是分开存在的，比如有的联网模块本身就是单片机，开发者可以直接对其进行编程，例如 ESP8266，它本身是集成了 WIFI 功能的单片机，如果直接使用 ESP8266 进行开发，那么标号‘1’‘2’两部

分就合并到了一起，成本也大幅度降低。但是使用 MCU+联网设备的主要原因是工程师更加熟悉这个 MCU 的平台，比如 STM32，51 等，我们只需要通过简单的 AT 指令就可以控制联网设备接入网络，这样做就可以大大缩短开发周期，所以最终使用哪种方式开发，是生产成本和时间成本的问题，要看开发者如何取舍。

现在我们切换到另外一个项目，假设用户是蔬菜大棚的农户，每个农户有多个大棚，每个棚之间的距离都比较远，而且又是野外，因此只能使用 GPRS 联网。此时用户的需求又是希望能控制所有的大棚，最简单的办法自然是每个大棚都安装一个物联网设备，这也意味着每个大棚都要配一张 SIM 卡，如果大棚有几十个，几百个呢？每个月维护 SIM 卡的成本将会非常可观，因为需要人为检查话费，充费，显然这种方式能实现但是无意义，因此我们思考能不能通过免费的传输方式建立一主多从的星型网络拓扑结构来管理多个大棚，主机负责采集和控制多个从机，他们之间通过免费传输方式传输数据，最后主机将这些数据统一打包经过通过 GPRS 发送到服务器或者根据服务器下发的针对某个从机的控制指令来控制对应的从机。这样无论大棚数量多少对于整套设备而言只是增加删除一个从机而已，而且只需要一张 SIM 卡。上面说的就是网关的概念，图 2.2 是温室大棚网关解决方案模型图（主从之间并不仅仅只有 485/LoRa 这两种，这里只是举例）。

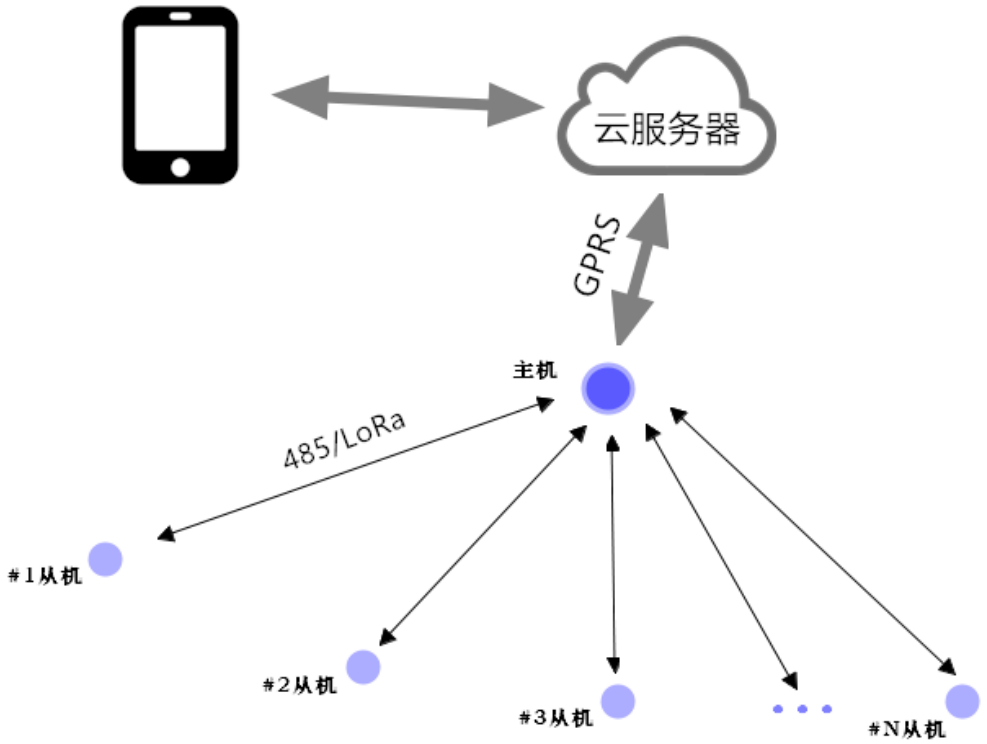


图 2.2　温室大棚网关模型

另外对于智能家居的应用场景来说，如果家中存在多个设备，并且都通过 WIFI 连接到家里的路由器，那么将会导致路由器负担加重，造成网络延时，甚至设备掉线，严重影响用户的上网体验，所以引入网关将从根本上解决这种问题，关于网关更多的内容，我们将会在相应章节讲解。

到这里硬件层就已经讲解完毕，我们的开发板就属于硬件层，在今后的开发也围绕硬件

层来展开，但是我们有必要知道物联网的其他环节，这与我们开发息息相关。

现在我们讲解物联网模型的第二层——**网络层**，它是一台在后台运行着管理、转发用户端和设备端消息的服务器。硬件层的设备通过网络模块（WIFI,以太网口，GPRS）连接到该服务器，说到这里有人可能会感到疑惑，为什么不让我们的硬件设备直接连到我们的手机上呢,这样就可以省去中间服务器开发的环节了啊？这是因为硬件设备要连接到指定设备首先就是要知道它的 IP 地址，目前普遍使用的是 IPv4 协议，使用 32 位来表示地址，它有 2^32-1 个 IP 地址（约为四十二亿个 IP）供人类使用，从这个数据就可以看出这些 IP 地址虽然很多但是对于人类而言资源还是太少了，并且会持续消耗，因此不可能为每个人都分配一个固定 IP，当然我们可以购买一个静态的 IP，但是一年的租金不菲，起码都要 1 万以上，相信这个价格大多数用户是无法接受的。顺便提一下，目前为了解决 IPv4 资源不足的问题，在很早就已经推出了 IPv6 协议，他可以表示的地址资源约为：2^128-1 个 IP 地址（约为 3.4*10^38 个），这样就可以为每个人甚至宠物来分配一个固定的 IP 地址了，到时我们的公民身份证多出个人 IP 一栏也不一定:P，当然 IPv6 普及缓慢，等到大规模普及还需要时间。

回归正题，经过上面的解释，大家应该就已经意识到服务器的重要性了，因为服务器的 IP 是固定的，我们通过将服务器作为一个数据中转就可以在效果上实现用户终端控制物联网设备了，即使用户终端和硬件设备不是固定 IP 也没关系，因为硬件设备和用户终端都是主动连接服务器的，服务器会记住这些连接。

另外服务器更重要的一点是整合用户信息，通过大数据和 AI 计算出用户的喜好、使用习惯等，这样就使得设备更加的懂自己，用户对设备的依赖性就会更强。

接下来再说一下**用户层**，它是用户和硬件之间“交流”的桥梁，我们通过简洁友好的用户操作界面来把硬件设备采集的数据图形化的展示给用户,或者用户点击按钮或者拖动滑动条等控件来对硬件进行设置，比如开关灯、调亮暗，如图 2.3：

图 2.3　控制面板示例

我们可以使用手机，平板电脑或个人 PC 来控制我们的硬件，他们是用户软件运行的平台。用户软件可以是移动设备上的应用，PC 机上的软件，或者是一个网页。另外个人认为网页是物联网设备与人交互的的未来，原因有二：

首先，对于用户来说，他不希望自己的手机上单独为一个硬件设备安装一个应用，如果用户使用的设备很多并且来自于不同厂家的产品，此时就需要安装更多的应用。但是用户的使用心理是拒绝为一个硬件设备横向安装一个应用的，在控制的时候还要翻来覆去的去找应用，他会认为很麻烦，还不如自己动手控制。

另外，对于厂商而言，如果去为产品开发一个软件的话，为了适应所有用户，他必须单独开发一个安卓平台的应用，一个 ios 平台的应用，使得开发队伍和后期维护成本（包括要对主流安卓手机进行适配）加大，根本原因在于缺少一种可以跨平台的手段来兼容不同平台的适配。由于网页是不受平台限制的，换句话说它可以跨平台，只需一个浏览器即可。同时，使用网页也可以将原本横向安装应用的方式变为纵向安装，比如所有设备都通过浏览器查看控制或者使用我们最常用的微信，然后利用微信内置的浏览器经过厂商公众号跳转到厂商服务器上并获取设备控制页面，微信在很早就已经进行这方面支持了，详情请点击查看：http://iot.weixin.qq.com/。

到这里，第一个远距离物联网模型就已经讲完了，这种模型也是本教程学习的重点。接着再来讨论下一个模型——近距离物联网模型。

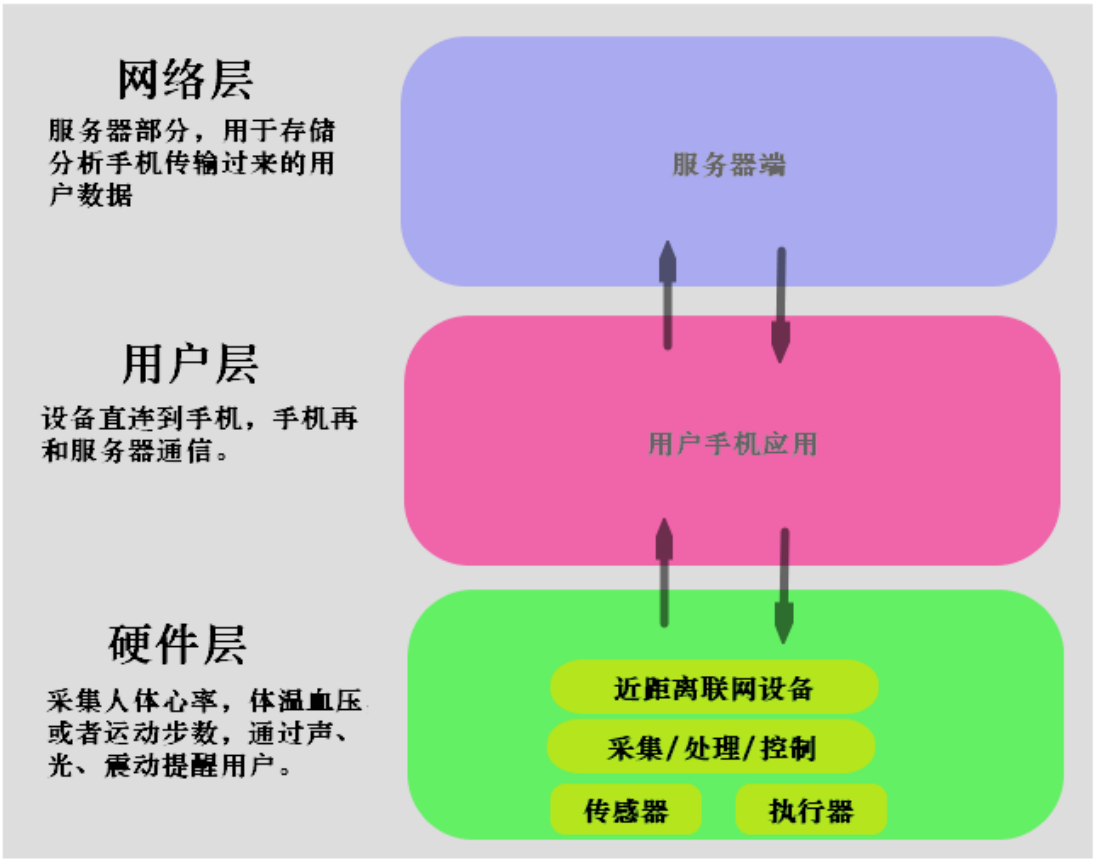


图 2.4　近距离物联网模型

这一类模型应用的场景是设备要么会被用户随身携带（手环，跑鞋，运动衣），要么在使用时必须近距离接触（体重秤，血压仪），如果使用远距离物联网模型的话需要解决的问题是联网问题，这个问题在于用户可能在 WIFI 信号覆盖的家中也能在没有 WIFI 信号的户外，这种情况下，厂商不可能为各种应用场景做联网设备兼容，成本太高，那么另外一个解决办法是将设备只做成 WIFI 联网，然后用户使用手机开热点，但是这种使用习惯用户不放

心而且在电话欠费的情况下硬件设备就不能用了，这样对网络依赖性太强，也不好，所以唯一的解决途径就是利用手机上现有的免费连接方式和硬件设备通信，然后通过应用软件查看和控制硬件设备，并通过手机将这些数据传输到服务器。目前手机支持的免费近距离通信有三个：WIFI，蓝牙，NFC，但是大多数还是使用的前两者，我们就拿小米物联网生态链为例，像电动牙刷，手环，体重秤这些都是通过蓝牙进行连接，用户（手机）离开这些设备就掉线，只能当做单机设备使用，走近之后又会自动配对连接，更新数据。再比如像小米相机就是通过 WIFI 的形式和手机进行连接，因为图像数据的传输需要很高的速度，所以会首选 WIFI，但是在设备和手机连接通信的过程中，手机是不能通过 WIFI 联网的，因此这种方式在某种场景下也存在不足。

其他就不过多讲解，对于近距离物联网通信的教程，风媒电子依然会借助青柚 ZERO 开发板陆续推出，敬请关注！

第三章 青柚 ZERO 开发板介绍

3.1 青柚 ZERO 起源

我们将青柚 ZERO 硬件介绍部分放在了介绍 STM32 和物联网的后面，目的是在讲解硬件的时候，不会对大家的理解造成障碍，尽量以循序渐进的教学形式向大家渗透。

开发青柚 ZERO 的目的是完善国内物联网教学套件的空白，作者将多年物联网软硬件开发经验总结为例程、文档以及硬件并完全开源给大家，希望能给予初学者帮助，方便开发者快速构建项目。

3.2 青柚 ZERO 硬件资源汇总

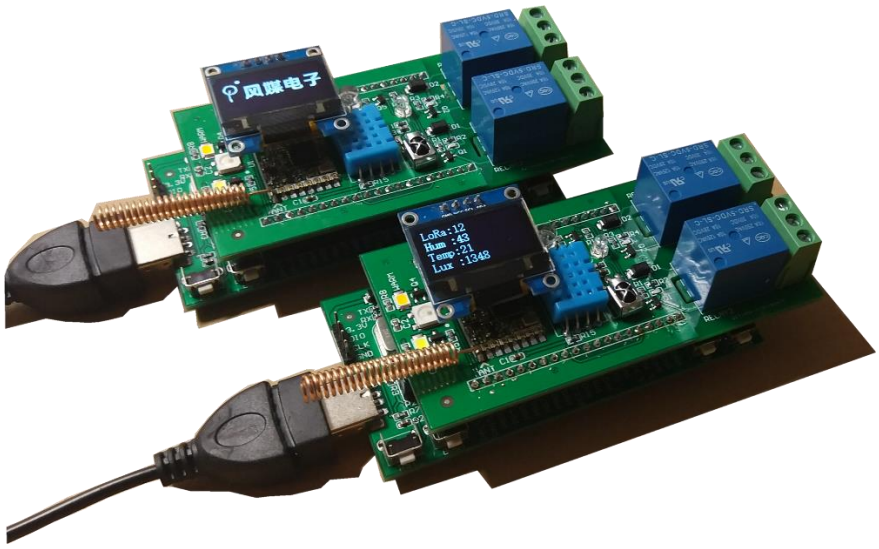


图 3.1　青柚 ZERO 物联网开发板效果图

青柚 ZERO 是一款可以弹奏音乐，发送电子邮件和微信的物联网开发板，每套开发板有两部分组成，第一部分是核心板，它是整个开发板的控制和联网核心，第二部分是扩展板，扩展板提供了常用传感器，双路继电器，冷/暖 LED+RGB 灯组，红外编解码，OLED 显示等功能，在平时使用时核心板和扩展板可以通过插接进行使用，这么做的好处是方便用户将核心板应用到自己的项目当中，使其简单连接后即可快速接入网络。下面分别详细介绍一下核心板和扩展板的硬件资源。

3.2.1 核心板硬件资源

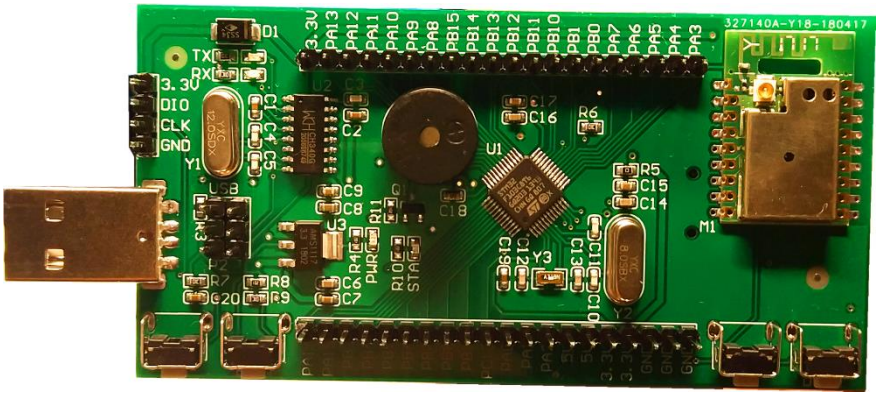


图 3.2　核心板正面

图 3.3　核心板背面

1. MCU 采用 STM32F103C8T6，性价比之王
2. 集成 Air640 WIFI 模组，使开发板轻松联网
3. 板载无源蜂鸣器，可以验证 PWM，并弹奏音乐
4. UART/USB 切换电路，方便进行 USB 相关实验
5. 内置 CH340 USB 转 UART 芯片，带状态指示灯，通信过程一目了然
6. IO 口全部引出，方便用户扩展
7. 独创启动模式切换电路，只需两颗按键即可轻松切换
8. 引出 SWD 调试下载接口，方便用户在线调试和固件烧录
9. 板载电源防反接电路，即使仿真器接反也不会烧毁开发板
10. A 型公头 USB，方便插接在笔记本电脑，移动电源以及手机充电器上
11. 两颗独立微动开关，和 USB 配合使用实现对计算机的控制，如玩游戏、控制 PPT 翻页等

3.2.2 扩展板硬件资源

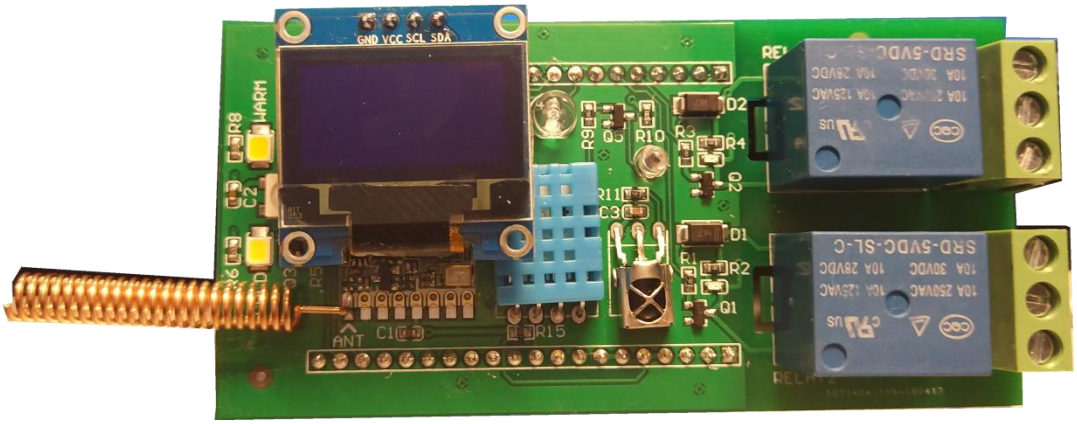


图 3.4　扩展板正面

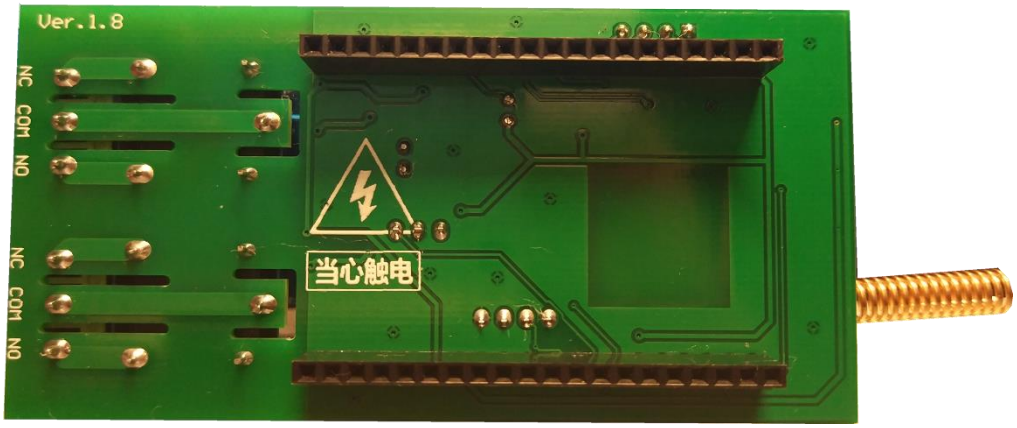


图 3.5　扩展板背面

1. 板载 SX1278 LoRa 模块，可以实现 LoRa 组网
2. 集成环境温湿度传感器，环境光强度传感器，适配大多数物联网项目
3. 支持 IIC 接口的 0.96 寸 OLED，可显示过渡动画，图片以及中英文
4. 高亮冷/暖 LED+RGB 灯组，可以实现对灯光亮度、色温以及颜色的调节
5. 双路继电器，带状态指示灯，可以控制家用电器以及电磁阀等高压器件
6. 集成红外编/解码，可以学习任意协议的红外遥控器并通过红外发射管对家电进行控制

3.3 出厂程序检验

为了保证产品质量，风媒电子的每套开发板都要经过出厂检验合格后才会发放到用户手中，用户收到的开发板默认刷入的是出厂检测程序的固件，如果用户想要再次对开发板进行硬件检验，则可以参考下面列出的方法，如果不需要测试可以直接跳到下一章。

有一点要建议大家:将青柚 ZERO 开发板应用在功耗较大的项目中时（如灯组和继电器以及屏幕等多个外设同时工作或用户使用开发板上的电源为外部大功率模块供电），建议大家直接将开发板插接到计算机的 USB 接口或者手机充电器上，不要使用 USB 数据线，因为任何的数据线都存在电阻，这部分电阻相当于是电源的内阻，它和板子上的负载是串联关系，所以当整机电流增大时数据线上分得的电压也就升高,这就有可能导致开发板由于供电不足而重启，或者某些模块不能正常工作，如 DHT11，但是在大多数应用中，比如我们提供的例程中，这种因为内阻问题而导致设备重启的情况是不会发生的，大家可放心使用。

另外在插拔核心板和扩展板时，应缓慢晃动拔下，防止掰弯引脚。

1. 上电开机（接入 5V），若 OLED 显示“Air640 OK！”，则说明 OLED 和 WIFI 模组没有问题。
2. 开机后 OLED 显示接收到的 LoRa 值以及采集到的传感器数据，同时核心板 LED 闪烁，如下图：

图 3.6　OLED 显示效果

此时说明单片机、温湿度传感器、环境光传感器、核心板状态指示 LED 无问题。LoRa 接收会一直显示一个固定值，原因是没有其他 LoRa 模块发送数据给它，对于 LoRa 模块我们已经进行过出厂检验，所以 LoRa 模块也是没有问题的，当然用户也可以向另外一块开发板烧录例程 20 中对应子设备固件（LoRa 发射）来检测 LoRa 模块是否正常。

3. 打开串口调试助手，按照下面设置操作，观察接收数据：

图 3.7　串口接收结果

如果收到如上数据，则说明 CH340 芯片无问题，另外我们可以观察到串口发送状态指示灯闪烁，说明状态指示灯无问题，此时使用串口调试助手发送数据给单片机，如果串口接收

状态指示灯闪烁，说明接收 LED 也无问题。

4. 用手遮挡环境光传感器，如果 OLED 显示的环境光数据有明显的变化，则说明环境光传感器也无问题。
5. 打开相机，观察红外发射管，如果出现粉红色，则说明红外发射管也无问题，如下：

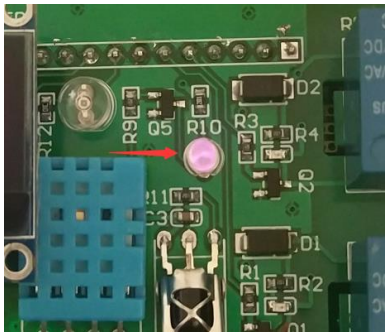


图 3.8　检测红外发射管

另外一种检测发射管的方法是：在比较暗的情况下，用手遮挡整个扩展板，如果采集的环境光传感器的结果明显变大，则也可以说明红外发射管正常。

6. 按下开发板配套的红外遥控器（拔下红外遥控器底部的绝缘塑料片）的任意按键，如果听到蜂鸣器无规则鸣叫，则说明红外遥控器、红外接收头以及蜂鸣器无问题。
7. 按下 UP 键，如果 RGB 颜色随机以不同颜色闪烁、冷光灯闪烁、蜂鸣器鸣叫、继电器 1 以及继电器状态指示灯状态改变，则说明 RGB、冷光 LED、蜂鸣器、继电器 1 及其对应的状态指示灯无问题。
8. 按下 DOWN 键，如果 RGB 颜色随机以不同颜色闪烁、暖光灯闪烁、蜂鸣器鸣叫、继电器 2 以及继电器状态指示灯状态改变、OLED 显示日期值，则说明 RGB、暖光 LED、蜂鸣器、RTC、继电器 2 及其对应的状态指示灯无问题。
9. 按下 BOOT 键后再按下 RST 键，随后先松开 RST 键再松开 BOOT 键，如观察到程序卡死，则说明成功进入系统存储器启动模式，即 UART 烧录模式，此时说明 RST 和 BOOT 键无问题。

如果蓝色 LED 亮起且扫描到该热点则说明 WIFI 模组工作正常。
到此开发板检测结束。

第四章 开发环境搭建

4.1 安装破解 MDK5.24

在开始开发之前必须要先要搭建好集成开发环境（IDE），支持 STM32 编程的 IDE 有很多，目前国内使用人数最多的就是 KEIL 公司出品的 KEIL MDK，我们本套教程就是基于 MDK 进行开发的。在开发之前要先安装 MDK。

下面是安装步骤：

步骤一：双击 MDK5.24 安装包

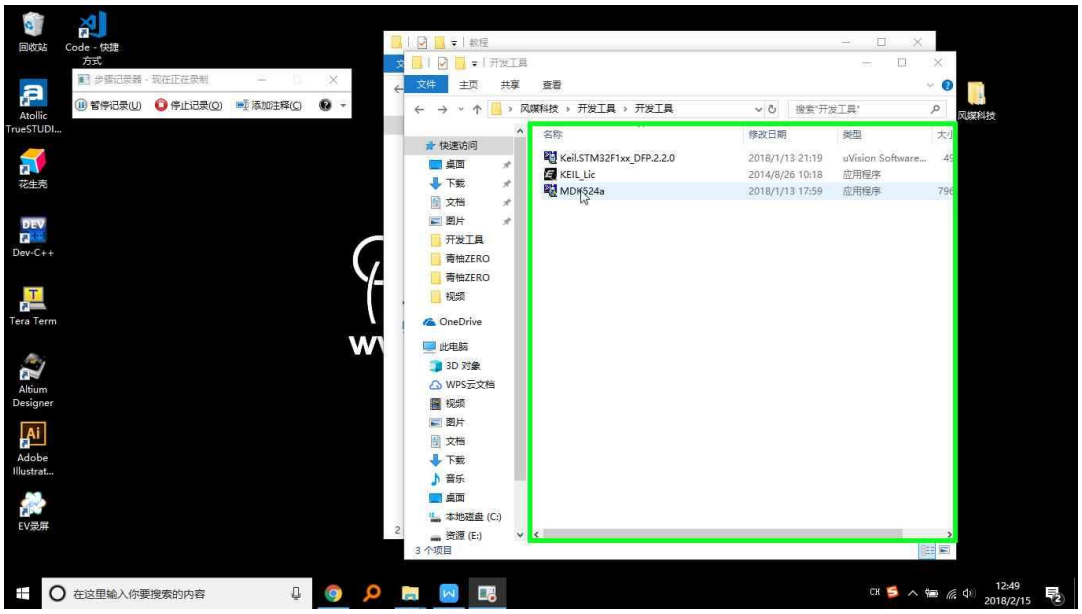


图 4.1　安装流程

步骤二：下一步

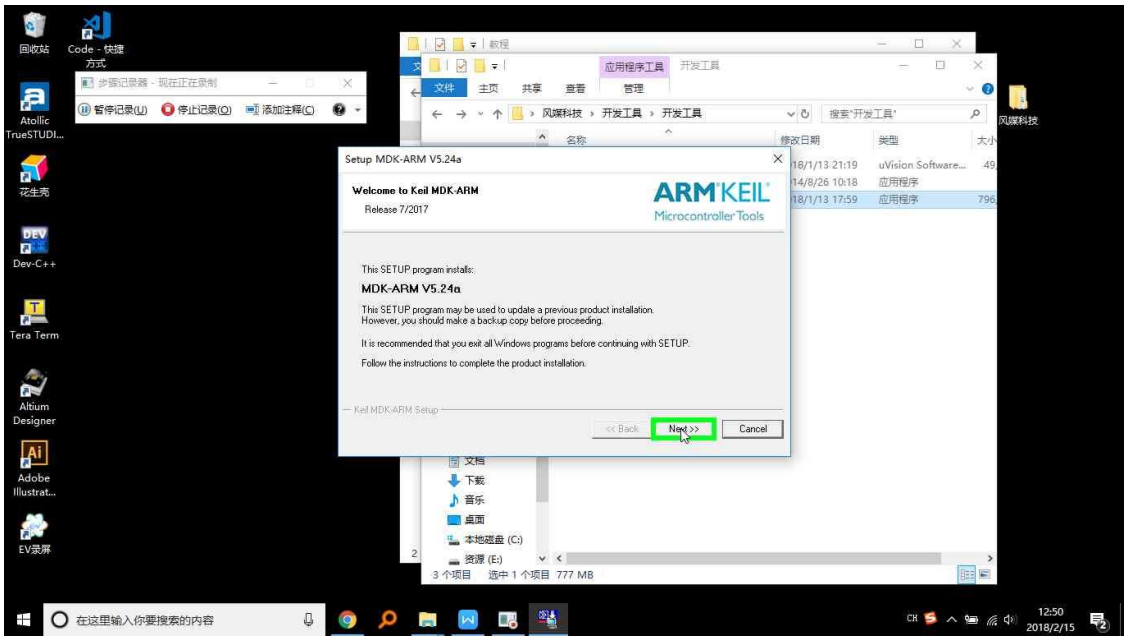


图 4.2　安装流程

步骤三：勾选我同意

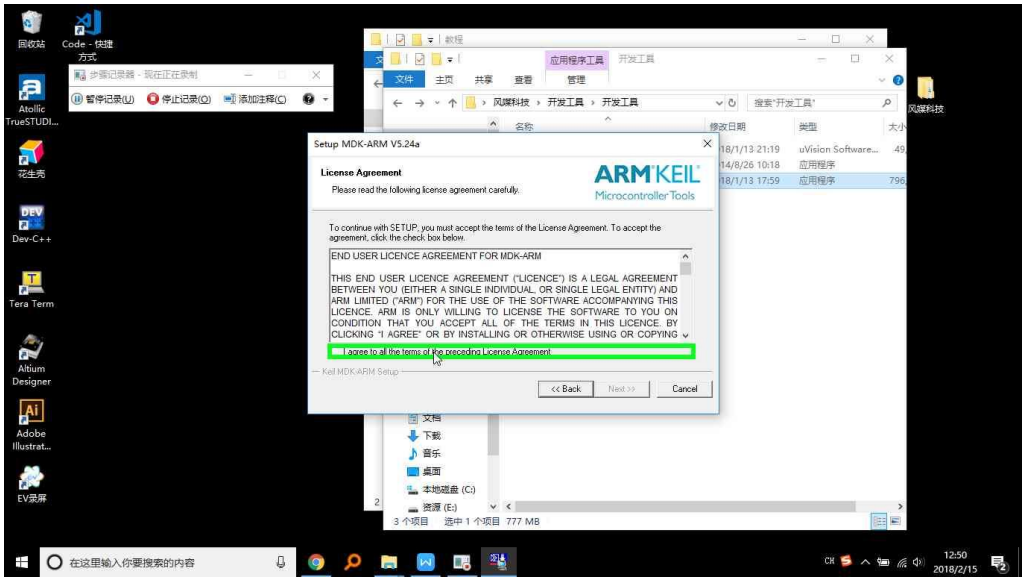


图 4.3　安装流程

步骤四：下一步

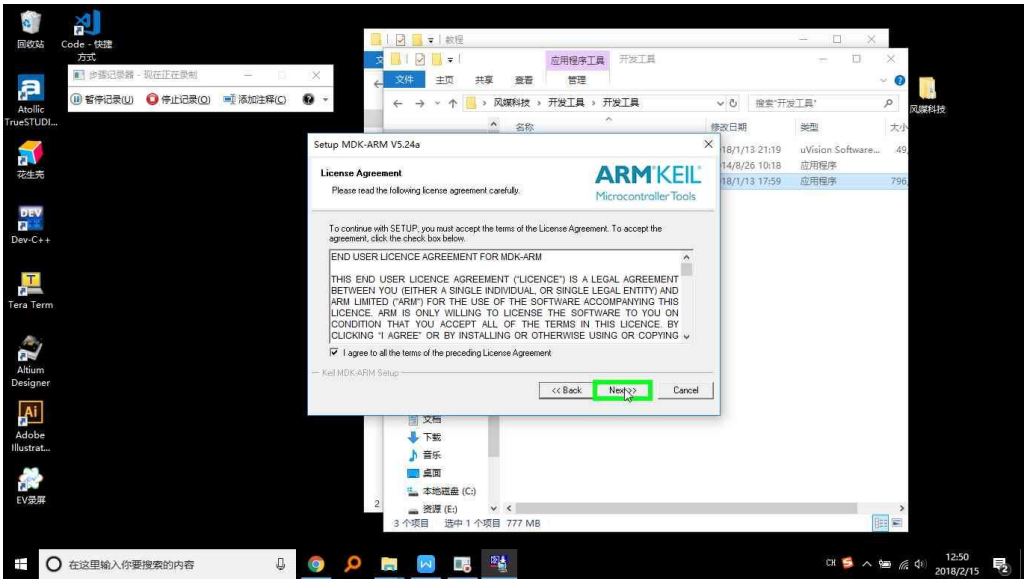


图 4.4　安装流程

步骤五：默认安装路径即可

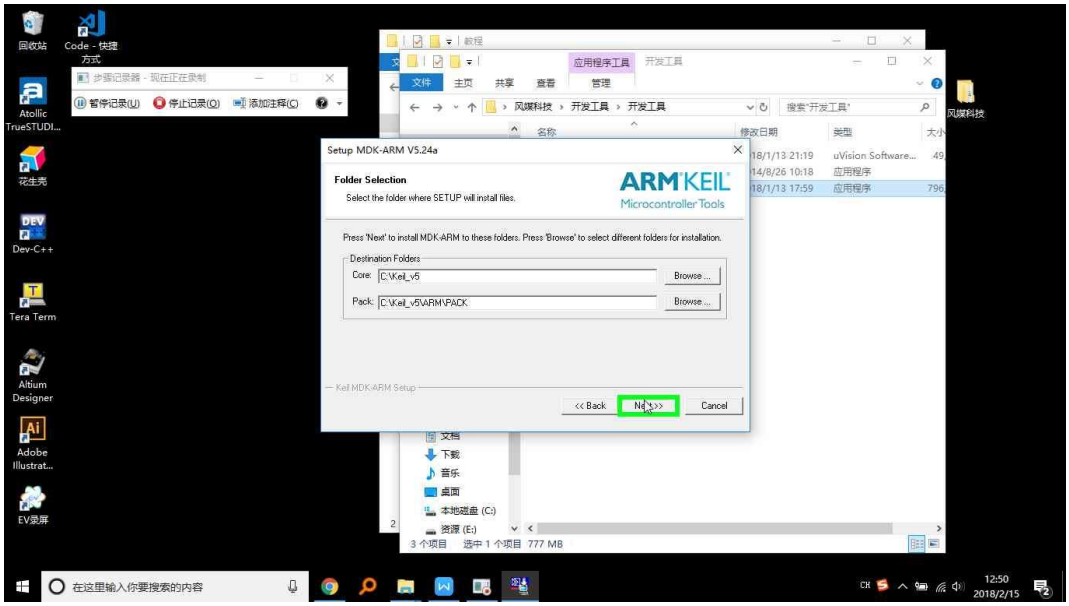


图 4.5　安装流程

步骤六：填写个人信息（随便填）

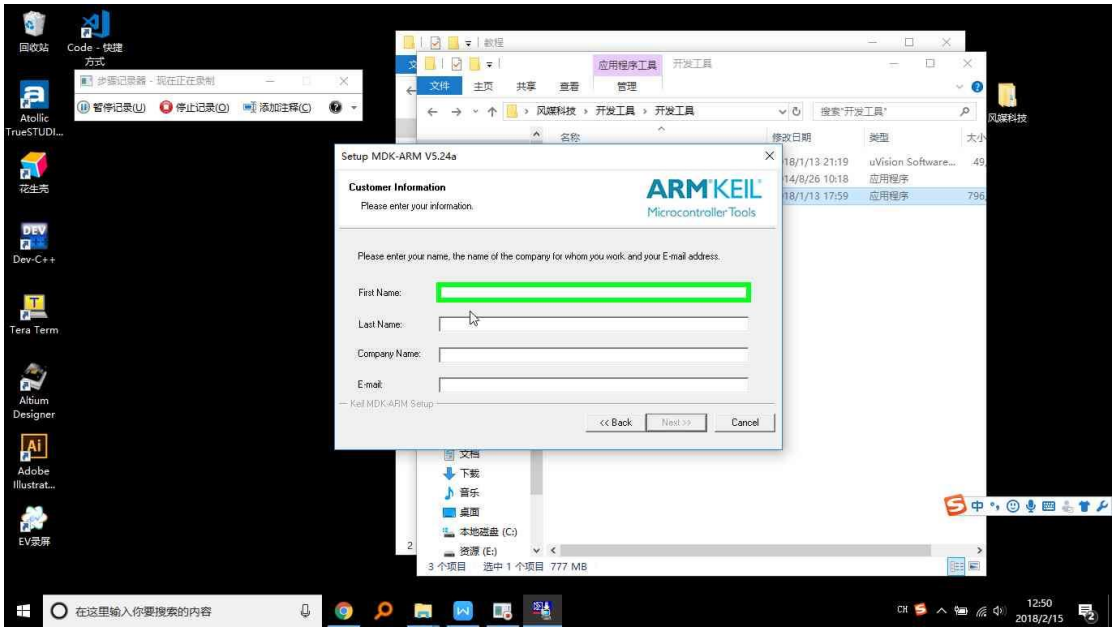


图 4.6　安装流程

步骤七：点击下一步

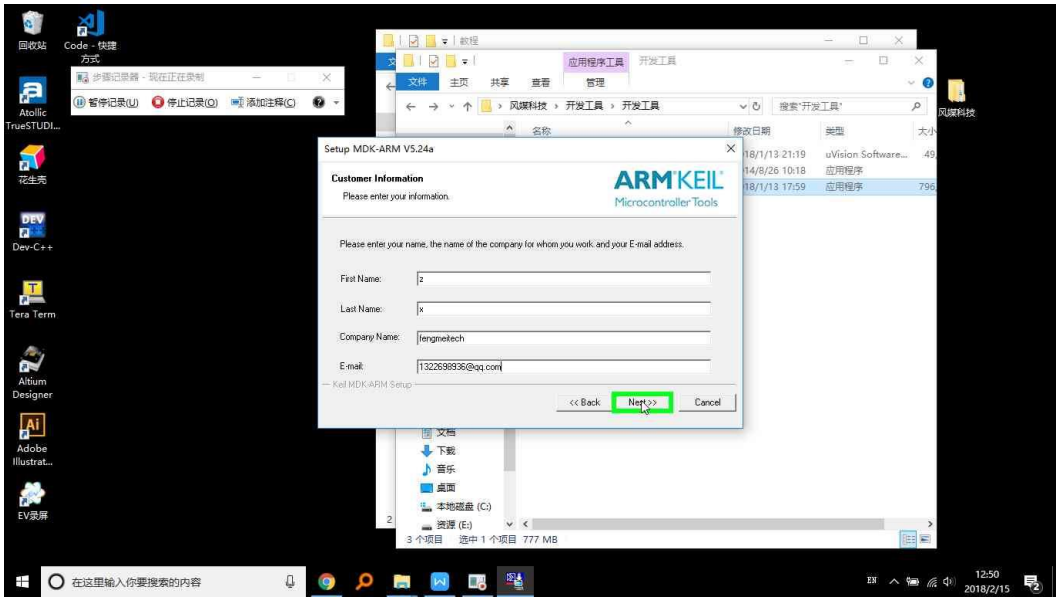


图 4.7　安装流程

步骤八：不勾选“show release Notes”，完成安装

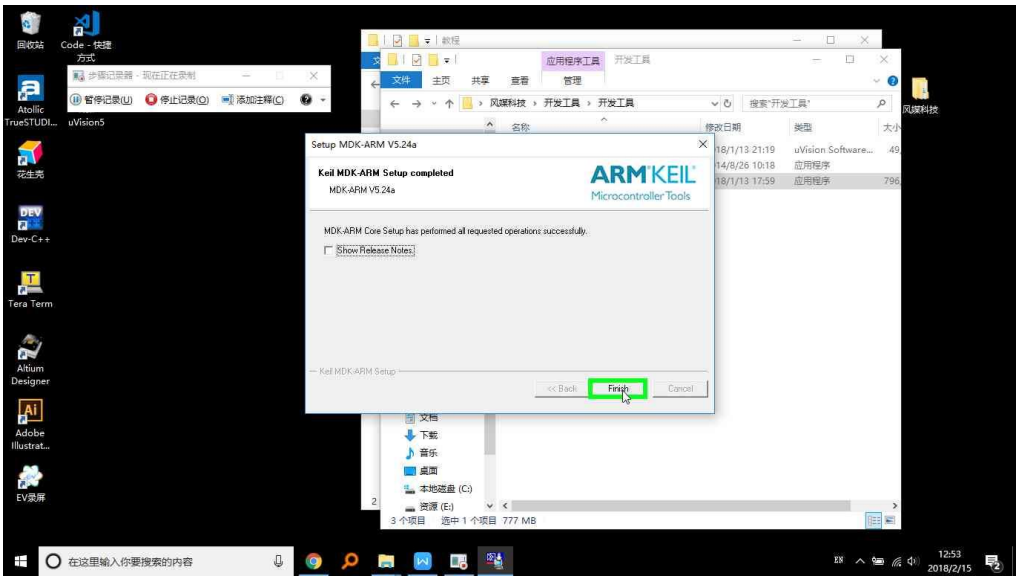


图 4.8　安装流程

接下来弹出引导你安装硬件支持包界面，不用管，关掉。

步骤九：破解

以管理员身份运行注册机（安静场合建议关闭音量，注册机运行时声音很大）

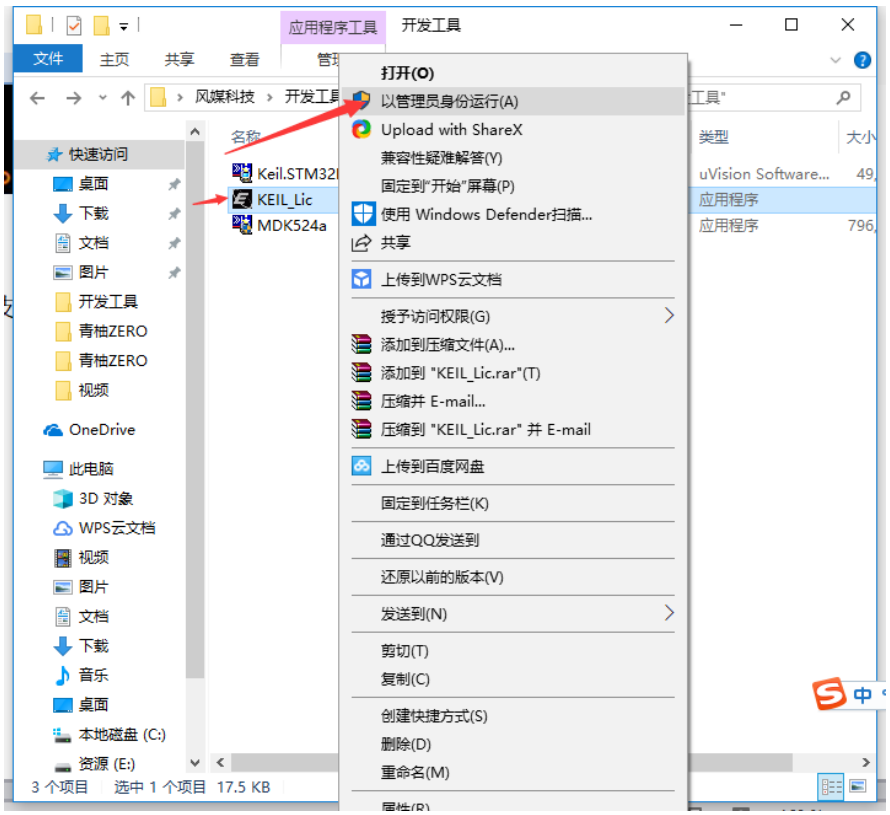


图 4.9　安装流程

步骤十：以管理员身份运行 MDK

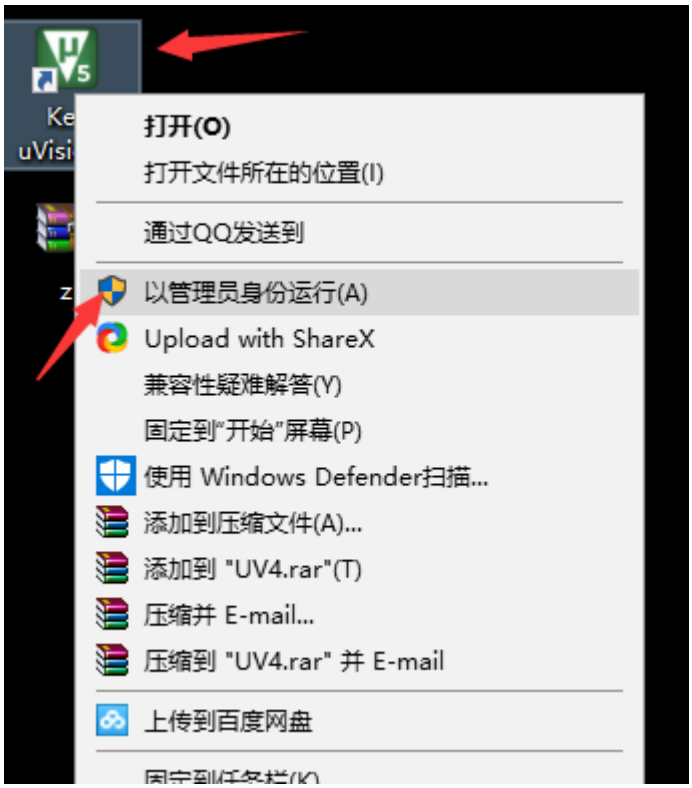


图 4.10　安装流程

点击“File”-“License Management”，复制 CID 码

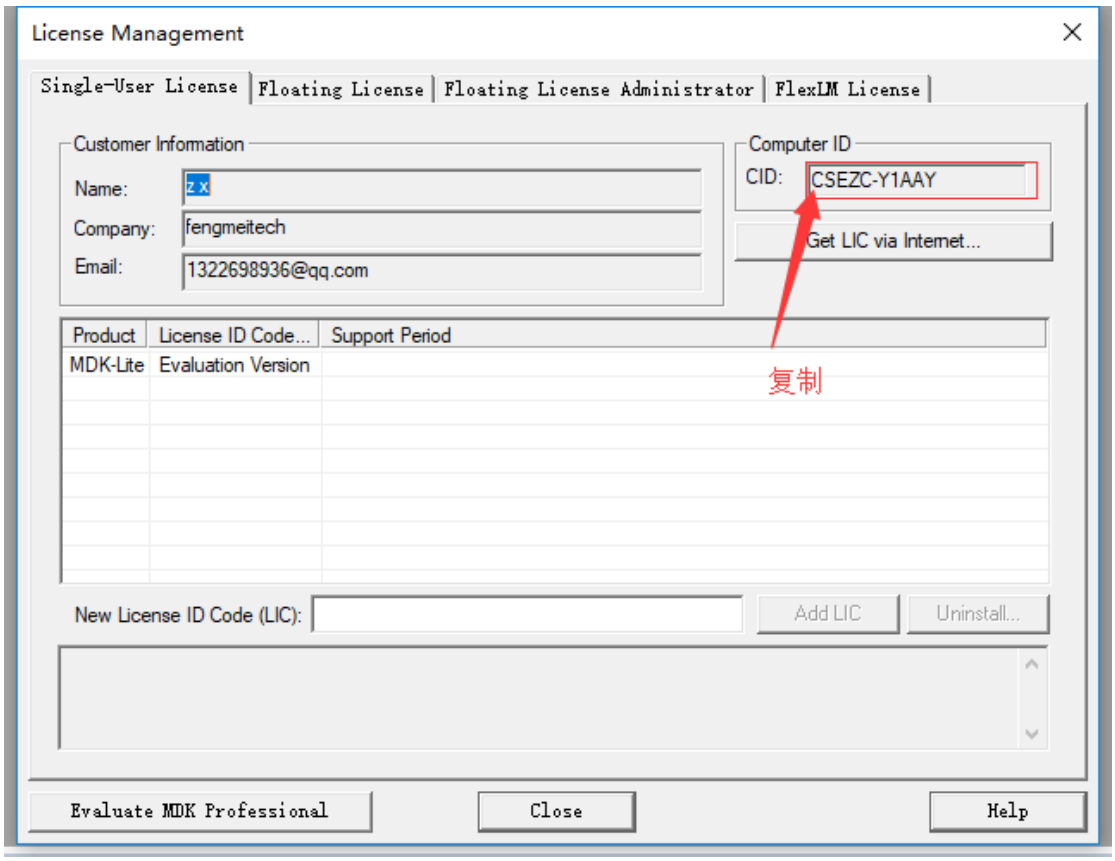


图 4.11　安装流程

步骤十一：使用注册机破解

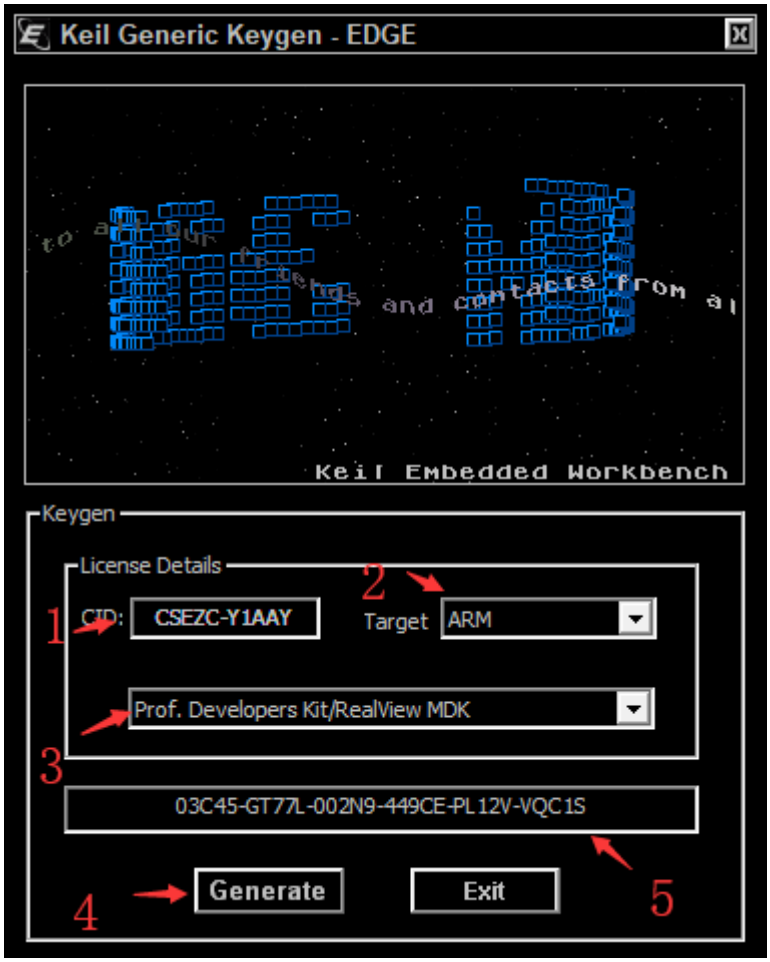


图 4.12　安装流程

复制注册码，添加到 MDK 中，大功告成！

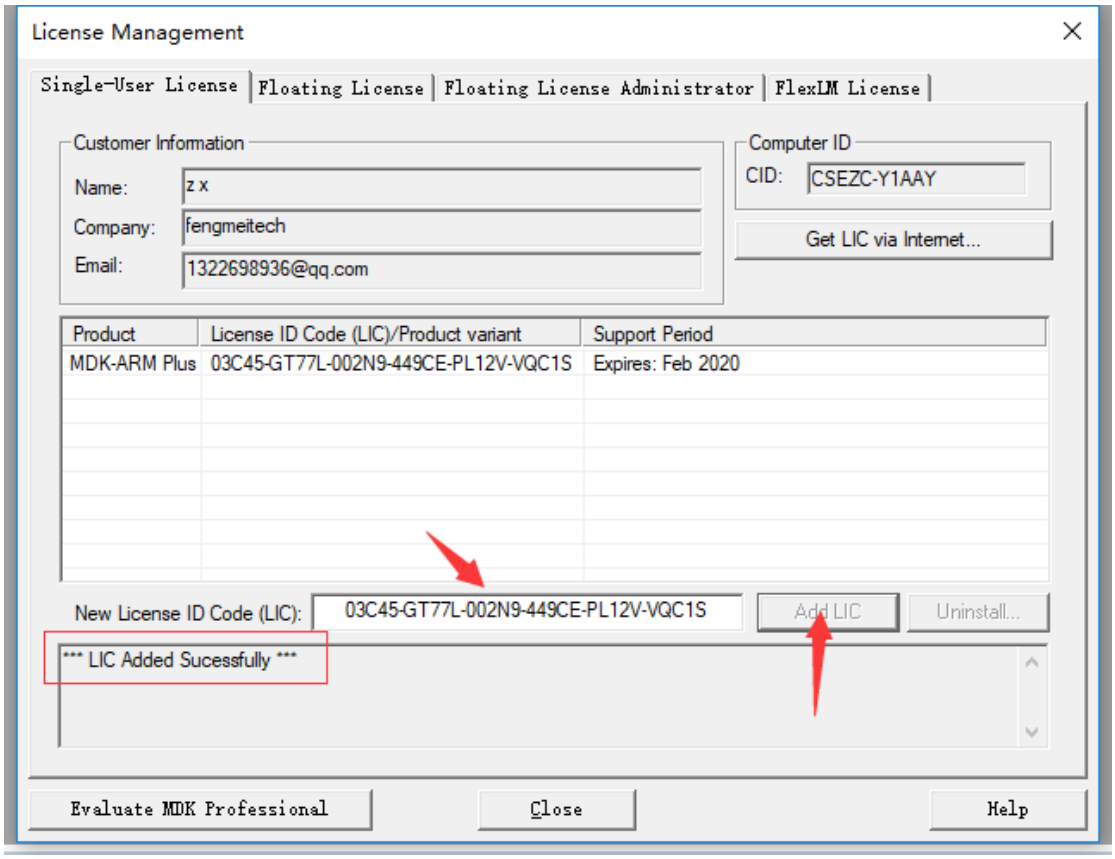


图 4.13　安装流程

步骤十二：安装芯片支持包

由于刚刚安装过 MDK，所以芯片支持包默认的打开方式就是 MDK，因此直接双击支持包即可：

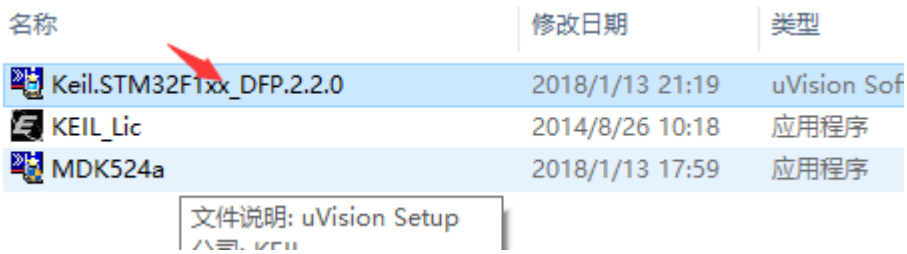


图 4.14　安装流程

这里跟大家解释一下，为什么要以管理员身份运行 MDK 和注册机，作者之前在其他电脑上对 MDK 进行破解的时候就没有勾选管理员选项，导致最后注册失败，所以建议大家无论在哪种计算机上面都要以管理员身份运行，避免浪费时间。

另外，如果你是从 51 过渡到 STM32，那你的桌面上一定还有 KEIL C51，有些人反映 KEIL C51 和 KEIL MDK 不能共存，存在打开 STM32 的工程打开的却是 KEIL C51，其实原因是这样：你在安装 MDK 的时候路径默认的是之前你安装 KEIL C51 的路径，所以两个软件混到了一起，而且两个软件的代码编辑器都是一样的只不过编译器不同，因此使用 51 单片机的编译器当

然就不能编译通过 STM32 的了。

所以解决办法如下：

在安装 MDK 的时候重新新建一个安装路径，比如之前 KEIL C51 安装路径是“C:\Keil_C51”，那么安装 MDK 的时候你可以新建一个文件夹“C:\Keil_MDK”，然后安装到里面，最后把这两个软件的快捷方式发送到桌面，重新命名，比如一个叫“KEIL C51”，一个叫“KEIL MDK”，然后你在打开工程时候，直接将工程文件拖动到对应快捷方式即可。

4.2 工欲善其事必先利其器

现在大家已经安装完开发环境，可以进行开发了，但是开发之后经过编译链接的二进制文件要如何烧录到单片机当中呢，我们在开发过程中要如何调试我们编写的代码呢？这就要用到对应的开发工具。

1.2.1 程序的烧录（下载）工具

通过一个平台为另一个平台开发、编译应用的过程叫做交叉编译，那么就存在将该平台编译的二进制文件烧录到目标平台的操作，对于大家很熟悉的 51 单片机的开发者来说，每片单片机内部都在出厂的时候内置了一段 **ISP**（在系统编程）启动代码，在单片机启动时，会先运行该段代码，来检测此时是否有下载任务进行，如果有就通过该段代码将 PC 机传过来的二进制文件存储到 ROM 中去，在今后单片机重启并且没有下载任务的时候，会自动执行之前烧录的程序。ISP 烧录的特点是无需将芯片拆下，直接可通过 PCB 上预留的下载接口进行下载，但是如果用户想要更新程序，就必须重新拿到板子，然后手动下载，如果批量升级的话，势必会带来很大的人力开销。因此和 ISP 对应的另外一种升级方式 **IAP**（在应用升级），也就是说单片机在正常运行的状态下可以自己为自己更新程序，比如，物联网应用中，可以通过服务器远程升级应用（联网模块的 UART 和单片机的 UART 相连），就无需人为干预了，节约了成本，但是 IAP 这种形式，用户要自己编写和 ISP 类似的启动程序也叫 BootLoader，从本质上来说 ISP 和 IAP 的原理都是一样的，即通过内置程序对 ROM 编程。

STM32 支持 ISP 和 IAP 烧录，但是平时用 ISP 会比较多，因为无需编写启动代码。STM32 在使用 ISP 编程的时候需要设置 BOOT0 和 BOOT1 引脚来决定芯片复位后在哪个位置启动并执行程序，如下图：

启动模式选择引脚		启动模式	说明
BOOT1	BOOT0		
X	0	主闪存存储器	主闪存存储器被选为启动区域
0	1	系统存储器	系统存储器被选为启动区域
1	1	内置SRAM	内置SRAM被选为启动区域

图 4.15 STM32 启动配置

我们要通过 ISP 升级，首先要运行 ISP 程序，因此就要配置 BOOT0、BOOT1 从系统存储器启动，因为 STM32 出厂内置的 ISP 程序位于此处，当进入 ISP 烧写完毕之后，我们再次设置 BOOT 引脚，把系统存储器启动改为主闪存存储器启动，复位之后既可运行。

下面通过串口烧录一个固件测试下效果：

这里用到的烧录软件是 STM32CubeProgrammer（安装过程一直点下一步），它将 Flash Loader、ST Visual Programmer、DFUse Device Firmware Update 整合到了一起，使得不同的烧录方式通过一个软件即可搞定。在烧录之前需要安装 CH340 的驱动，CH340 是一款由江苏沁恒研发生产的 USB 转 UART 的芯片，我们通过它来和单片机 UART 建立通信连接。安装步骤很简单：双击安装包-点击安装按钮-稍等片刻，会弹出下面对话框，表示安装成功。

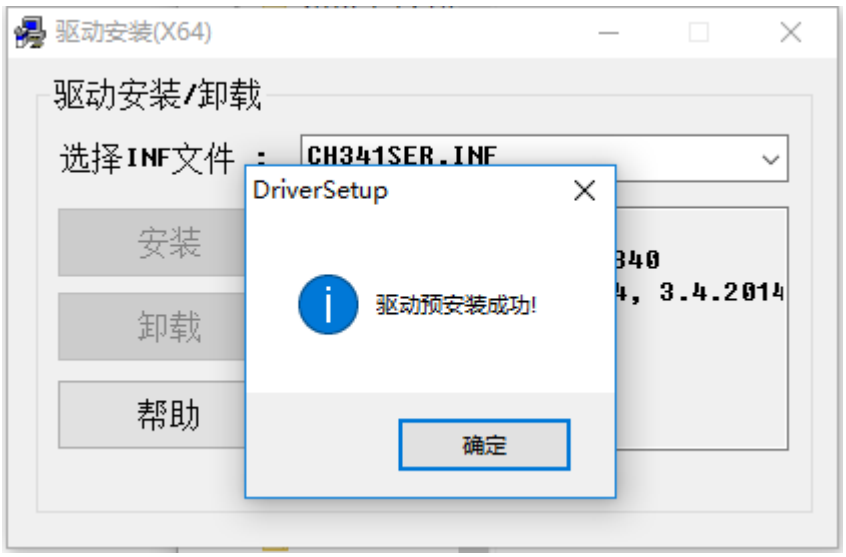


图 4.15　CH340 驱动安装完成

接下来将开发板插到 USB 口，打开 STM32CubeProgrammer，然后先按下开发板 BOOT 按键不松开，再按下 RST 按键，接着依次松开 RST 和 BOOT 键，此时开发板就进入了 ISP 下载模式（可以看到两颗串口指示灯在不停的闪烁）。不用保持 BOOT 按键一直按下的原因是，在复位后的 SYSCLK 的第四个时钟沿 BOOT 两个引脚状态会被锁存，之后 BOOT 引脚处于任何状态都不会受影响。

按照下图设置：

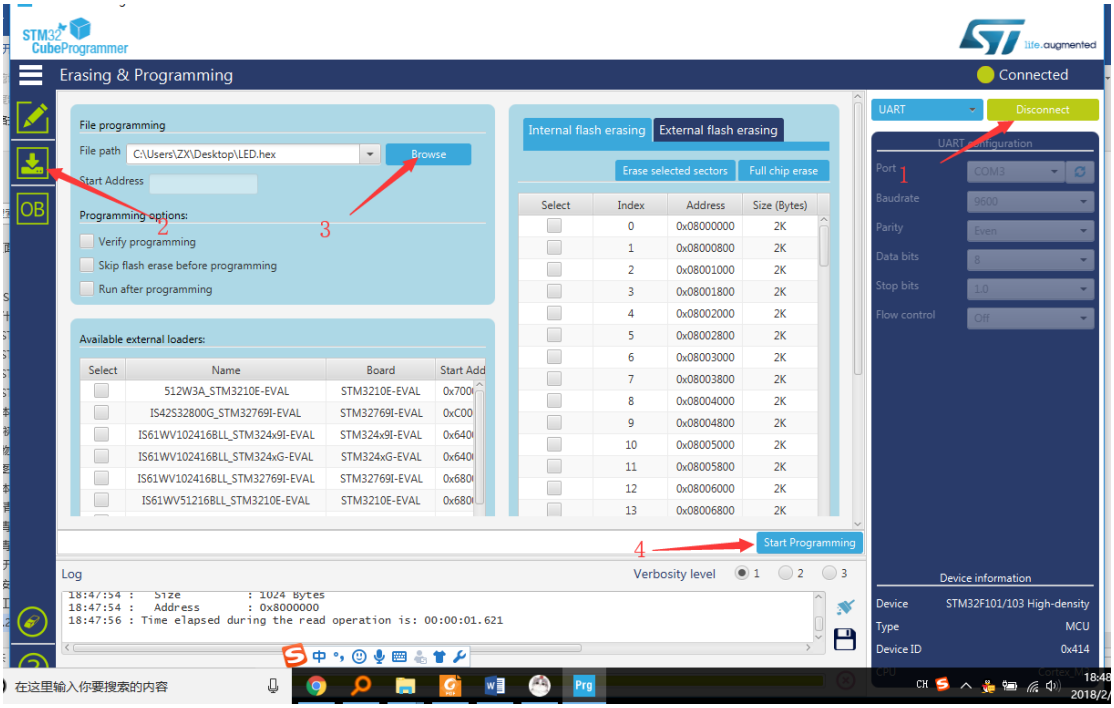


图 4.16　软件配置

右侧选择 UART 烧录方式，波特率任选，但是必须设置为**偶校验**，然后刷新串口并连接，此时再点击左侧烧录图标进入烧录界面，接着点击“Browse”选择对应的二进制文件（文件路径要保证是英文），二进制文件支持.bin,/.elf/.axf/.out/.hex/.Srec。这里我们以 hex 文件为例。最后点击开始编程等待烧录结束后复位设备。

到这里，我们已经可以让单片机运行我们写的程序了，但是如果我们想要在线调试怎么办呢，比如查看某个寄存器或者变量的值，所以还要引出另外一种烧录方式——通过仿真器下载，STM32 支持 JTAG 和 SWD 下载，二者都可以对单片机烧录和在线调试，但是 JTAG（5pin）占用引脚比 SWD（2pin）多很多，所以在平时开发中更倾向于 SWD 烧录调试。通过使用仿真器除了可以烧录程序外还可以在 IDE 环境下在线仿真。

市面上仿真工具有很多，比如 SEGGER 公司的 JLink，ARM 公司的 ULink 和 ST 官方提供的 STLink，我们所使用的是精简版的 STLink V2，只有 U 盘大小更方便携带，下面我们使用 STLink V2 来下载程序（先要安装我们提供的驱动），软件还是 STM32CubeProgrammer：

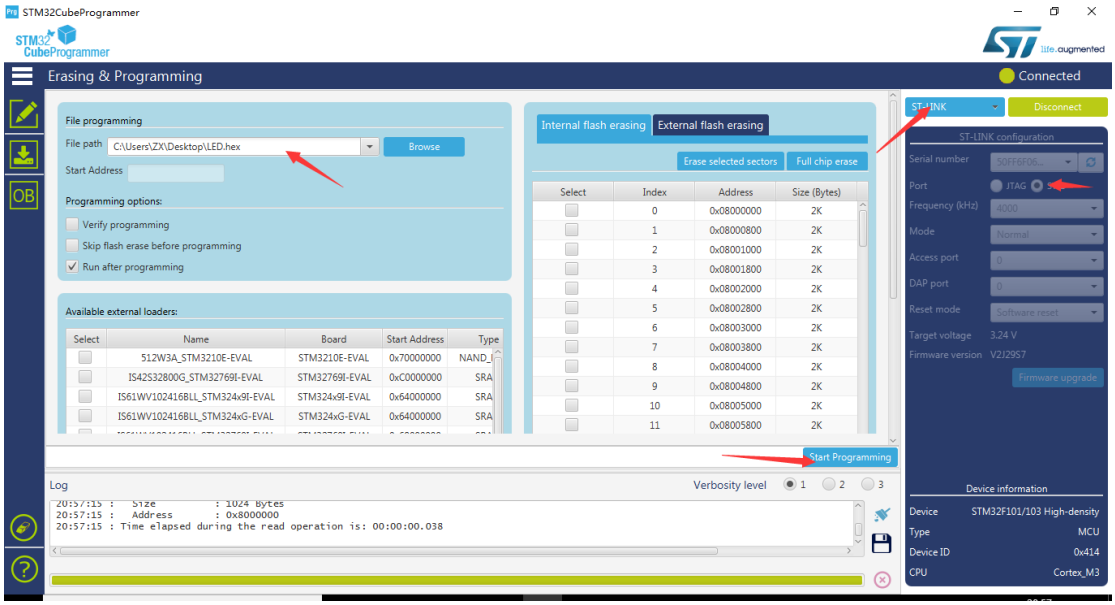


图 4.17　软件配置

使用 STLink V2 烧录程序的步骤和之前配置差不多，只是把下载方式改成使用 STLink 仿真器，通过 SWD 下载即可。

当然，我们也可以在 MDK 环境下通过 STLink 对单片机进行烧录和调试，这样一来就无需来回切换软件了，另外 MDK 的在线仿真工具也是非常强大的，下面就以一个简单的实例来通过 MDK 对开发板进行烧录。看图：

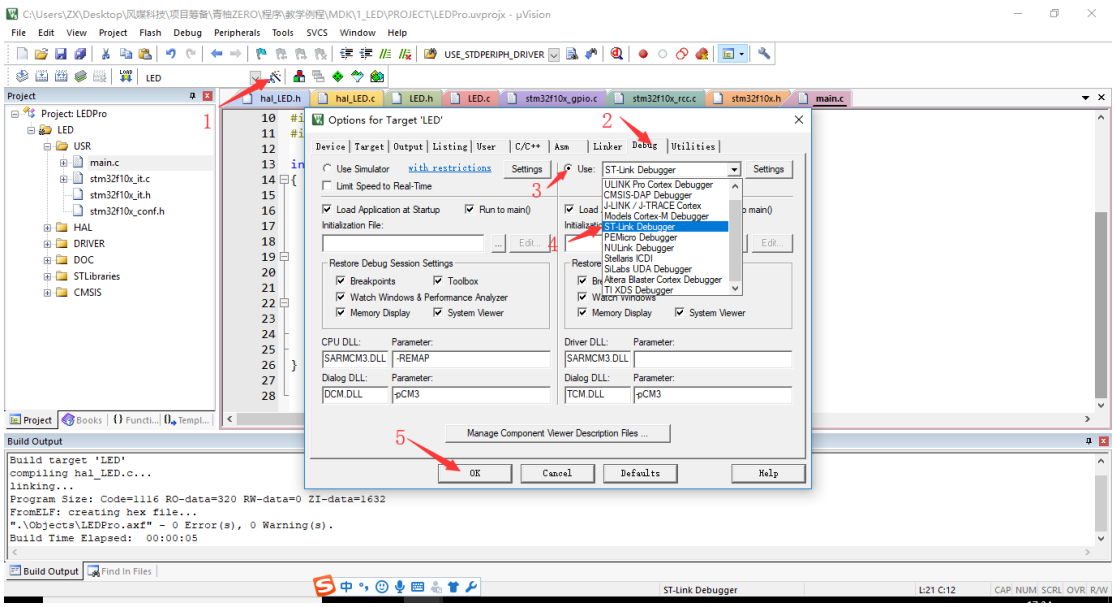


图 4.18　仿真配置流程

这个项目是一个很简单的 LED 闪烁实验，如何实现暂时不用管。我们在编译链接生成目标文件之后，打开魔术棒-选择 Debug-选择右侧使用硬件仿真-硬件使用 STLink-点击 OK 即可。

此时，可以点击 Setting 按钮，查看 STLink 是否识别：

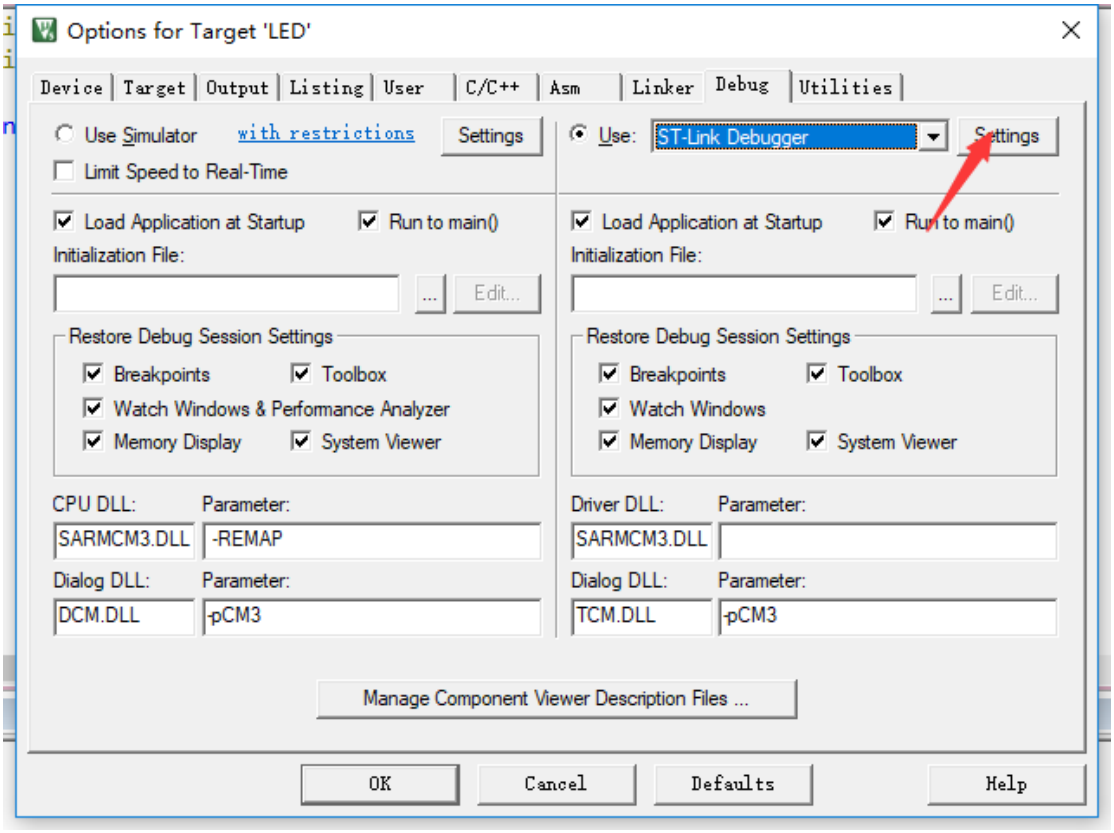


图 4.19　检测仿真设备

如果出现如下标识，标识已经成功连接：

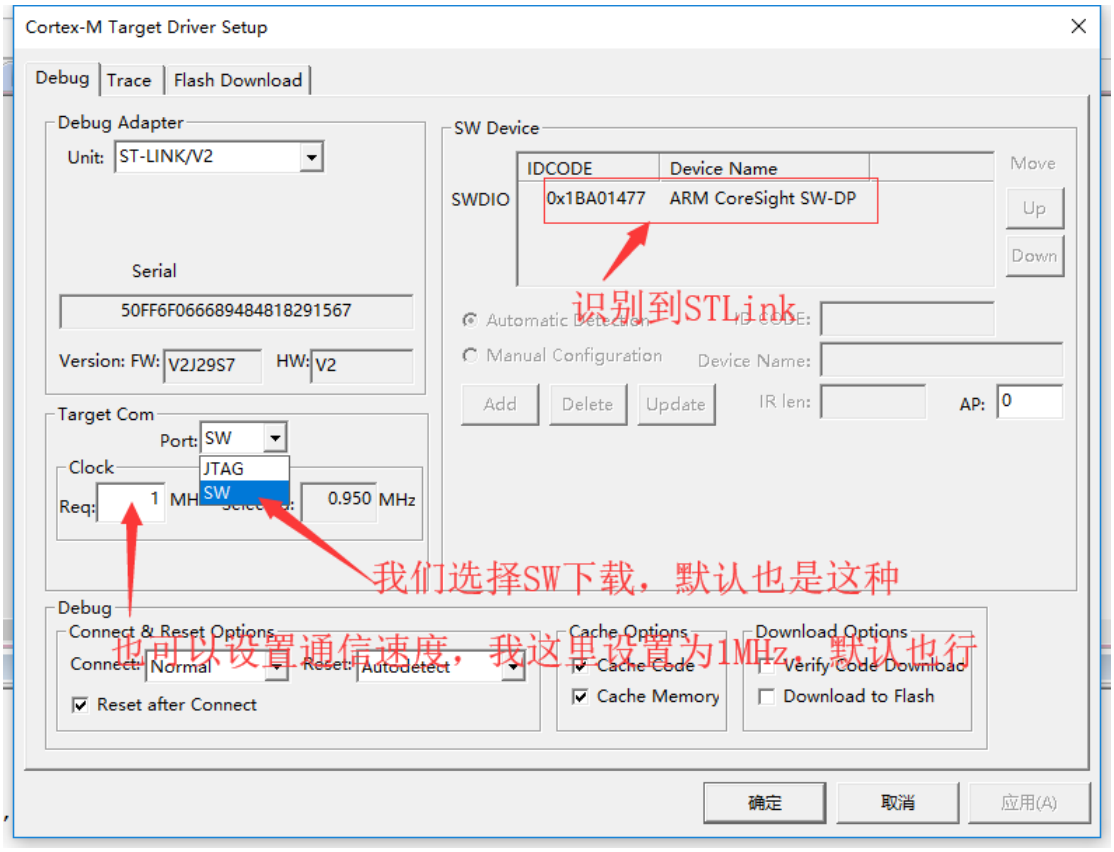


图 4.20 检测仿真设备

然后，我们点击下载按钮，即可烧录：

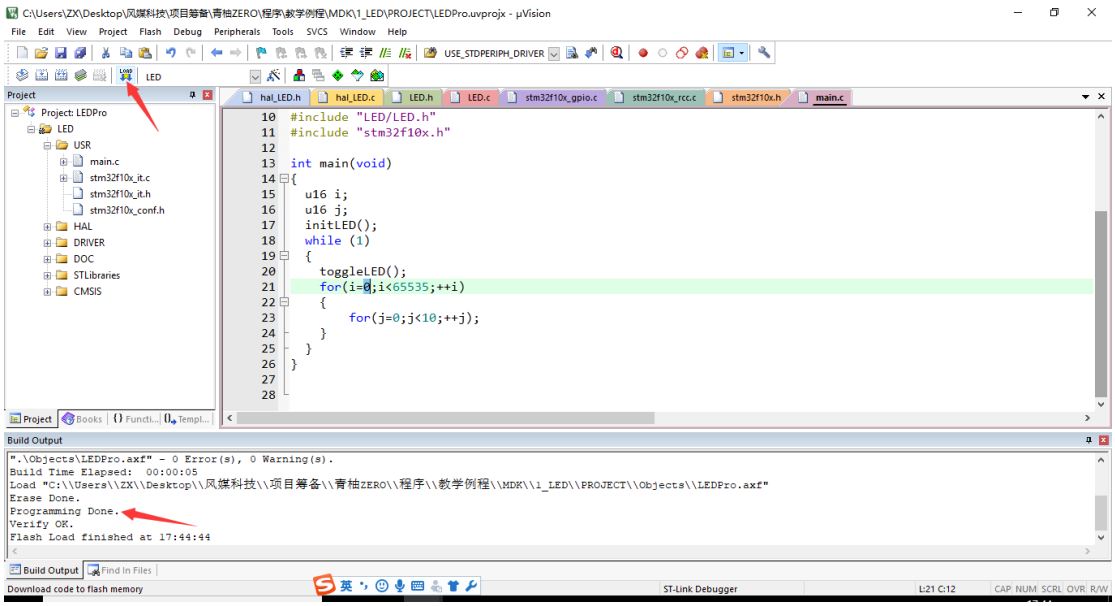


图 4.21 烧录流程

然后复位设备即可查看效果。

到这里我们已经可以熟练的通过 MDK 烧录程序了，那么如何进入仿真呢？在工具栏点

击放大镜按钮，即可进入或者退出仿真模式，如下图。

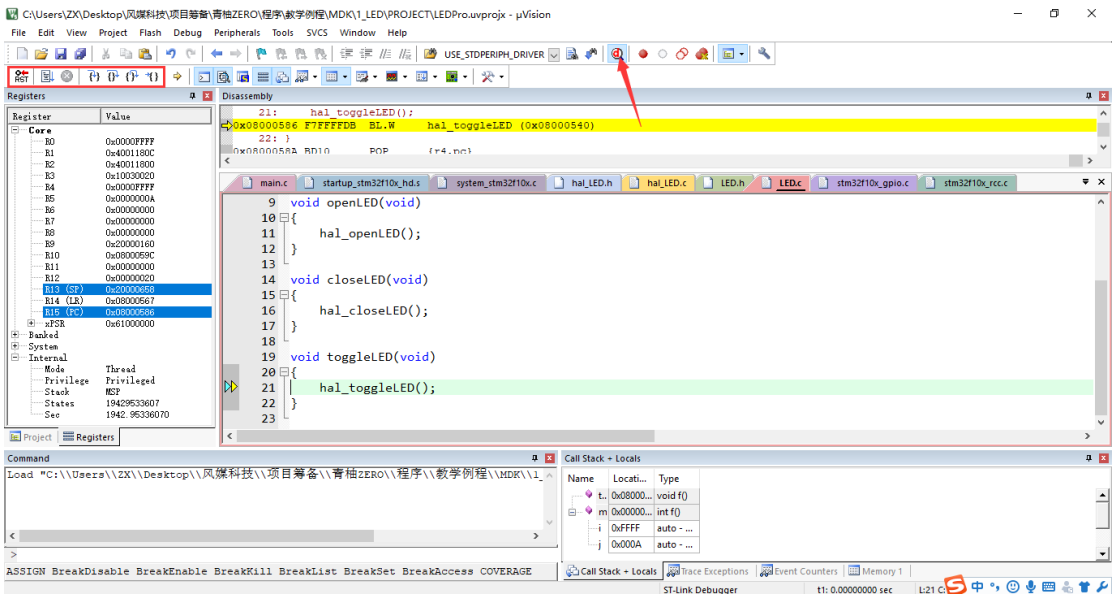


图 4.22 仿真界面

图中框起来的是平时开发时最常用的功能，我们把鼠标停在按钮下方，他会显示该按钮的功能，比如，从左至右依次是：

复位单片机，等同于按下复位按钮，只不过这种复位之后单片机不执行代码，等待用户点击执行按钮；

执行按钮，让单片机开始执行代码，碰到断点会自动停下；

停止正在执行的程序，但并不是复位，而是停在当前执行的代码上，此时再次按下执行按钮继续执行后面的程序；

跳到函数内部，比如函数 funA()包含函数 funB(),函数 funB 又包含 funC()，那么使用此按钮会在执行到 funA()的时候跳进 funB,再按下会跳到 funC(),然后执行完 funC()自动返回调用它的上层函数；

从当前函数跳出到调用它的地方，比如上面举的例子，在 funC()中按下该按钮，就会在执行完 funC()函数之后跳到 funB()；

跳过执行，也就是说不管这行函数里面是否调用其他函数，只要按下此按钮就会把该函数一次性执行完然后跳到下一行。

跳到当前光标处，看图说明：

图 4.23 跳到当前光标处举例

只要按下该按钮，程序便会执行到该行后停止（21 行位置），再按下按钮，则会再次执行该行下面和上面代码并再次停在 21 行，有点类似断点，但是断点可以存在多个而光标只有一个。

再说一下如何查看变量，我们选中一个变量，比如 i，然后鼠标右键按图操作即可。

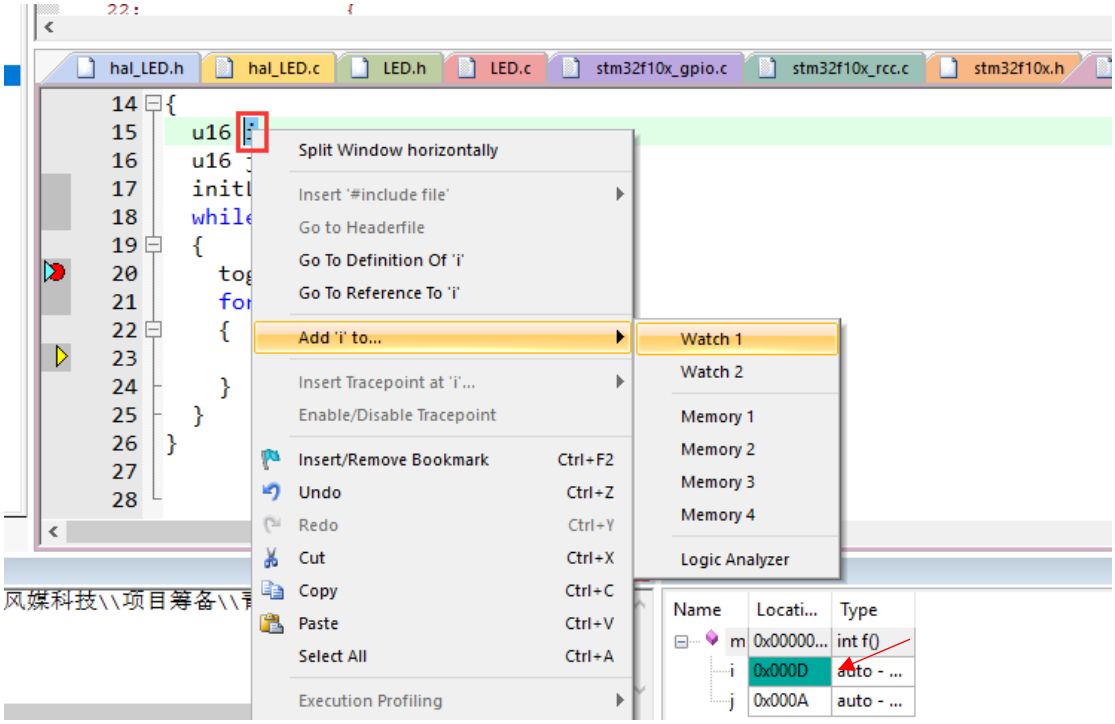


图 4.24 查看变量

接下来在学习一下如何查看外设寄存器，这里以 GPIO 为例，看图操作：

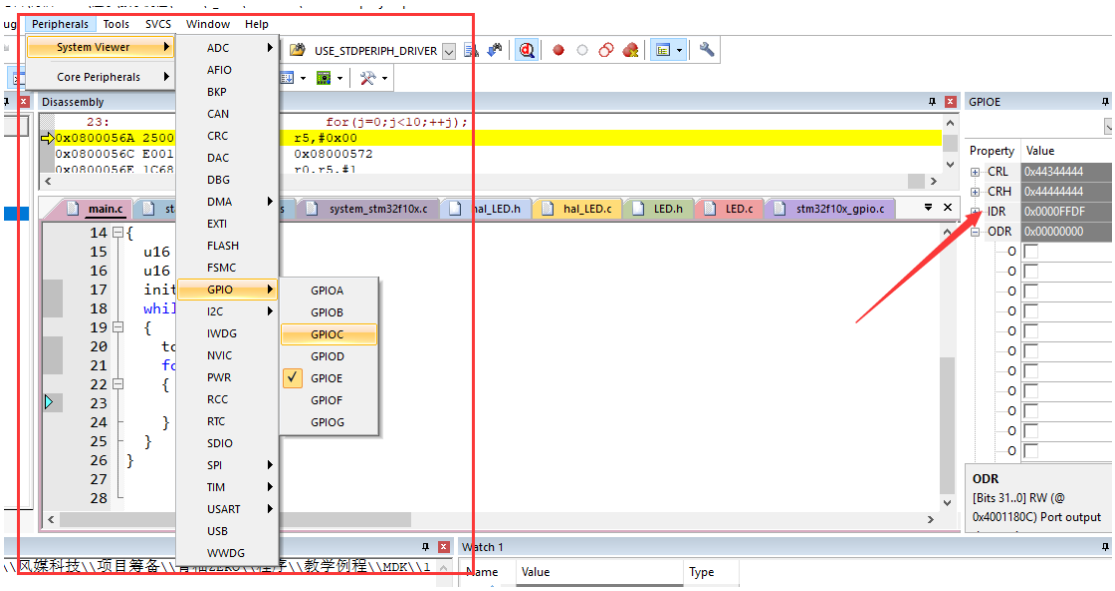


图 4.25 查看外设寄存器

同时左侧可以方便查看 ARM 内核寄存器：

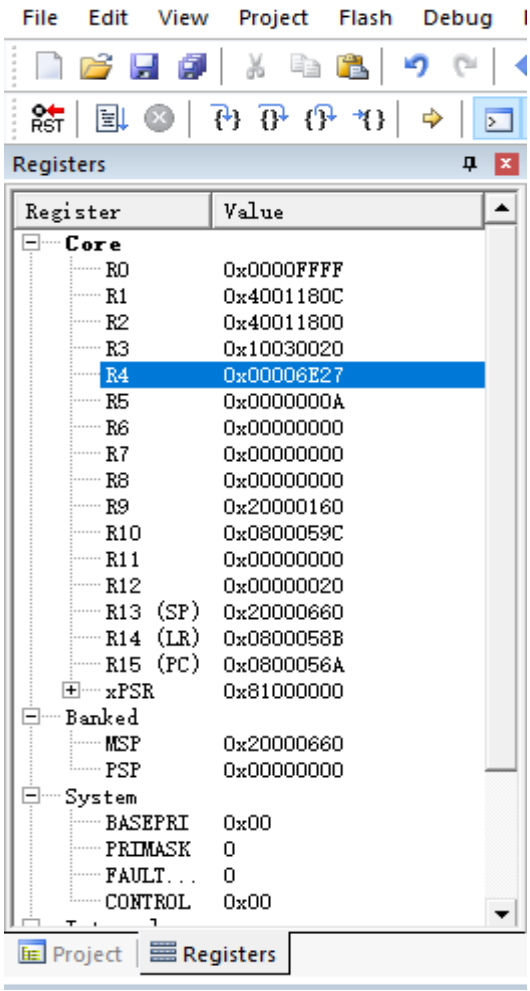


图 4.26 查看 ARM 内核寄存器

1.2.2 开发辅助工具

集成开发环境（IDE）它是集成了代码编辑器、编译器、链接器、调试器于一身的开发工具合集，但是我们所使用的 IDE（MDK）也并非完美，让人吐槽最多的可能就是它的代码编辑功能，比如：如果不编译工程就不会有代码提示功能，函数参数不会提醒，中文支持不完善等。因此我们可以使用其他代码编辑器来替代 MDK 的代码编辑器，我们只使用它的编译和调试功能即可。这里我想大家推荐的编辑器是微软的开源代码编辑器——Visual Studio Code。

下载网址：https://code.visualstudio.com/Download

下载完成之后，先安装 C/C++插件，该插件可以提供代码补全，查看定义位置，参数提醒等功能。如下图：

图 4.27 安装 C/C++插件

接下来对编辑器进行配置，配置的目的是避免在导入 KEIL 工程文件时遇到中文注释会出现乱码的情况，很简单，看图：

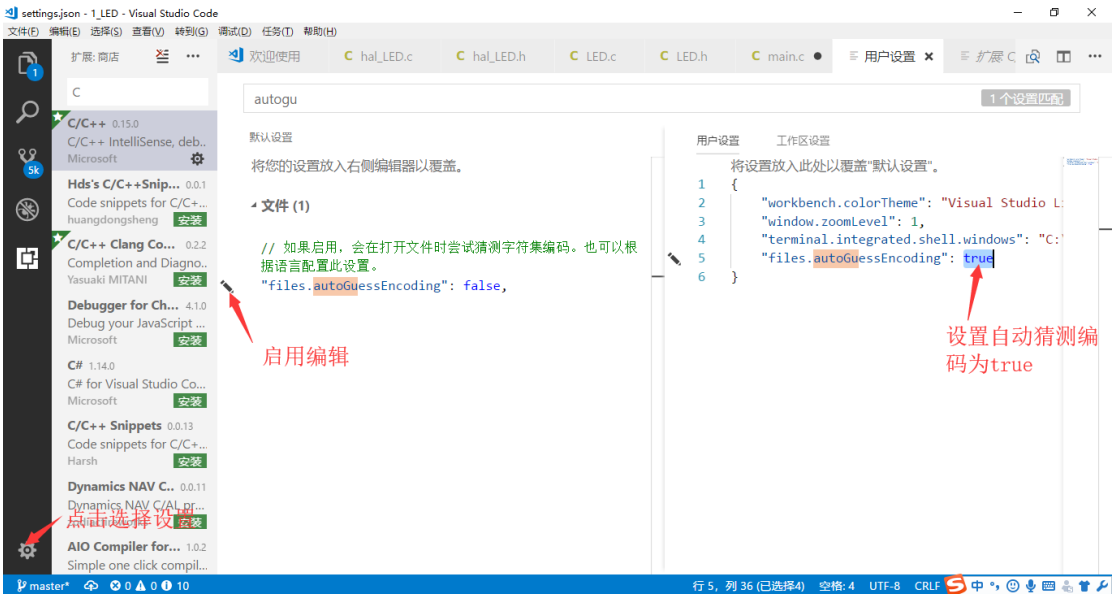


图 4.28 设置编码格式

这里设置的是查看代码时的编码格式，大家切记不要以某种编码格式保存，尤其是在驱动 LCD 或者 OLED 显示中文的情况下，因为不同的编码规则会导致中文字库的中文索引变为乱码，使其显示异常，这种问题往往很难发现。

现在点击“文件”-“打开文件夹”，选择目标工程目录：

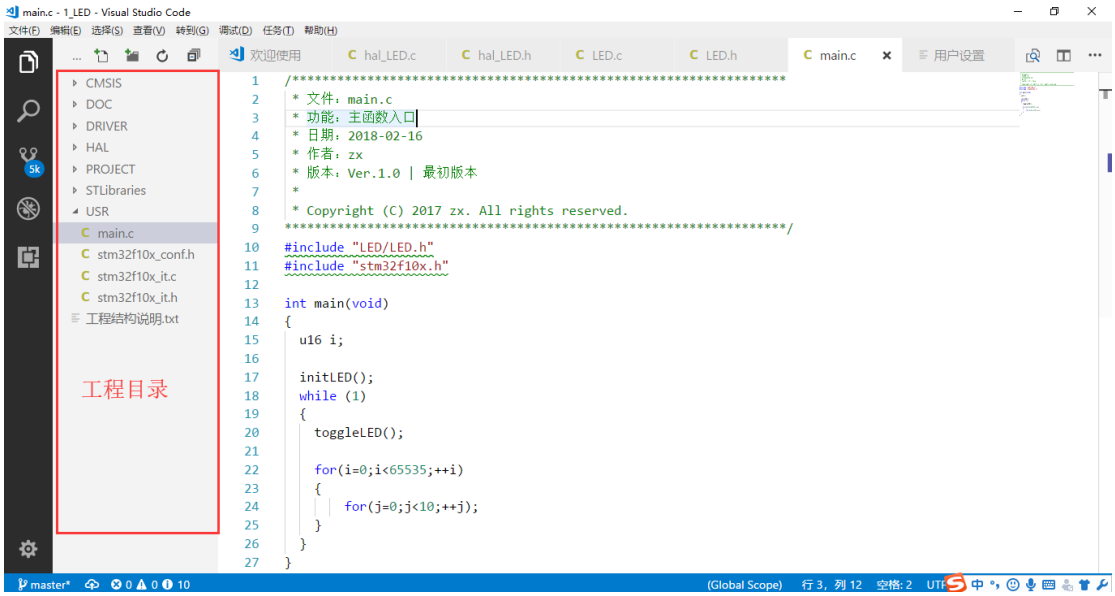
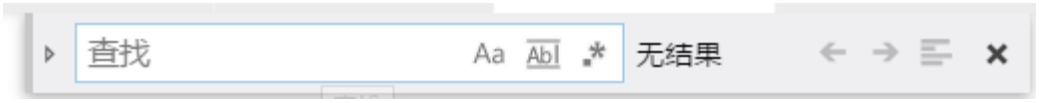


图 4.29 工程目录展示

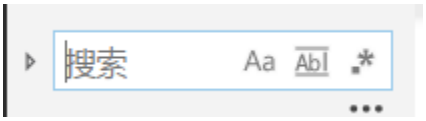
VS Code 的功能十分强大，这里作者总结几个用到最多的操作：

1. 文件内查找替换



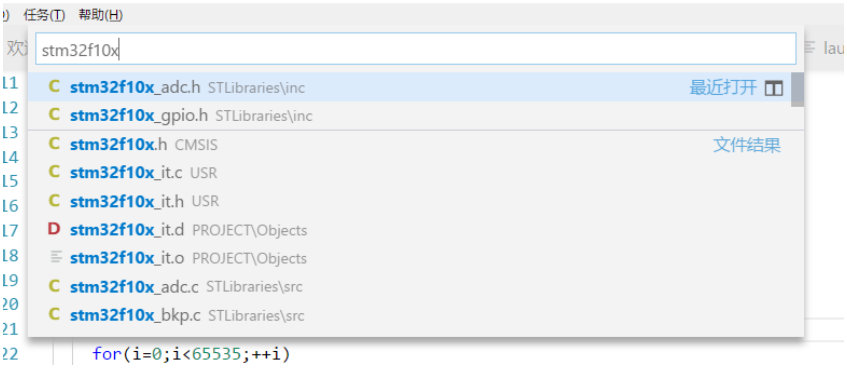
CTRL+F 呼出文件内查找替换，右侧按钮为查找筛选条件，如全词匹配，区分大小写和通过正则表达式筛选。

2. 全局查找替换



SHIFT+CTRL+F 呼出全局查找替换，功能同上。

3. 快速查找文件



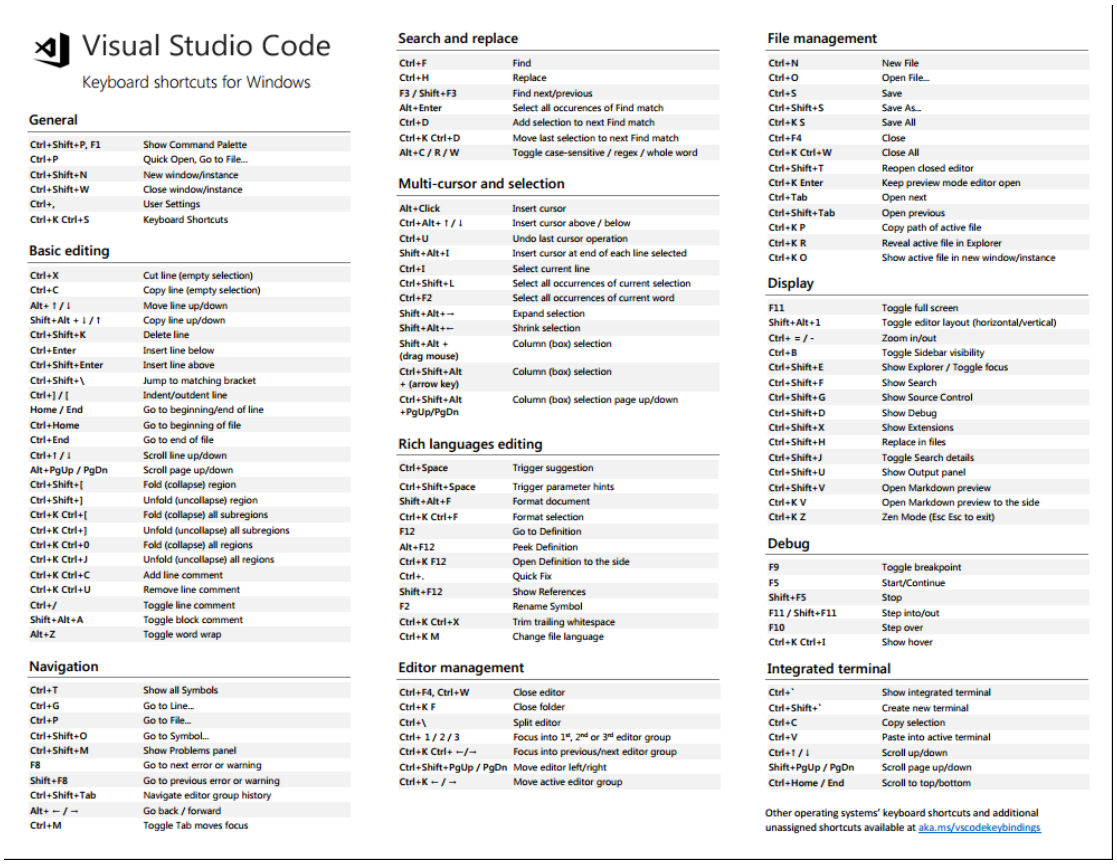
CTRL + P 呼出文件跳转，此时键入要查找的文件即可。另外也可以在文件跳转查询框内输入“: + 数字”来跳转到该文件的某一行代码。

4. 快速跳转到变量或者函数定义处

两种方式：第一种，在待查找函数或者变量上按下 CTRL+鼠标左键即可跳转到定义处。第二种，选中待查找对象然后鼠标右键选择转到定义即可，另外鼠标右键呼出菜单中的速览定义则无需跳转，直接在本行代码下方显示定义内容，非常方便。

下面作者列出了 VS Code 的全部快捷键，方便大家查阅。如果大家想看高清版本，点击“全部快捷键”链接进入。

对于代码托管的功能后续会推出教程，这和我们学习无关，不做讲解。



Visual Studio Code

Keyboard shortcuts for Windows

General

Ctrl+Shift+P, F1	Show Command Palette
Ctrl+P	Quick Open, Go to File...
Ctrl+Shift+N	New window/instance
Ctrl+Shift+W	Close window/instance
Ctrl+,	User Settings
Ctrl+K Ctrl+S	Keyboard Shortcuts

Basic editing

Ctrl+X	Cut line (empty selection)
Ctrl+C	Copy line (empty selection)
Alt+ ↑ / ↓	Move line up/down
Shift+Alt + ↓ / ↑	Copy line up/down
Ctrl+Shift+K	Delete line
Ctrl+Enter	Insert line below
Ctrl+Shift+Enter	Insert line above
Ctrl+Shift+\	Jump to matching bracket
Ctrl+] / [	Indent/outdent line
Home / End	Go to beginning/end of line
Ctrl+Home	Go to beginning of file
Ctrl+End	Go to end of file
Ctrl+↑ / ↓	Scroll line up/down
Alt+PgUp / PgDn	Scroll page up/down
Ctrl+Shift+[	Fold (collapse) region
Ctrl+Shift+]	Unfold (uncollapse) region
Ctrl+K Ctrl+[	Fold (collapse) all subregions
Ctrl+K Ctrl+]	Unfold (uncollapse) all subregions
Ctrl+K Ctrl+0	Fold (collapse) all regions
Ctrl+K Ctrl+J	Unfold (uncollapse) all regions
Ctrl+K Ctrl+C	Add line comment
Ctrl+K Ctrl+U	Remove line comment
Ctrl+/	Toggle line comment
Shift+Alt+A	Toggle block comment
Alt+Z	Toggle word wrap

Navigation

Ctrl+T	Show all Symbols
Ctrl+G	Go to Line...
Ctrl+P	Go to File...
Ctrl+Shift+O	Go to Symbol...
Ctrl+Shift+M	Show Problems panel
F8	Go to next error or warning
Shift+F8	Go to previous error or warning
Ctrl+Shift+Tab	Navigate editor group history
Alt+ ← / →	Go back / forward
Ctrl+M	Toggle Tab moves focus

Search and replace

Ctrl+F	Find
Ctrl+H	Replace
F3 / Shift+F3	Find next/previous
Alt+Enter	Select all occurences of Find match
Ctrl+D	Add selection to next Find match
Ctrl+K Ctrl+D	Move last selection to next Find match
Alt+C / R / W	Toggle case-sensitive / regex / whole word

Multi-cursor and selection

Alt+Click	Insert cursor
Ctrl+Alt+ ↑ / ↓	Insert cursor above / below
Ctrl+U	Undo last cursor operation
Shift+Alt+I	Insert cursor at end of each line selected
Ctrl+I	Select current line
Ctrl+Shift+L	Select all occurrences of current selection
Ctrl+F2	Select all occurrences of current word
Shift+Alt+→	Expand selection
Shift+Alt+←	Shrink selection
Shift+Alt + (drag mouse)	Column (box) selection
Ctrl+Shift+Alt + (arrow key)	Column (box) selection
Ctrl+Shift+Alt +PgUp/PgDn	Column (box) selection page up/down

Rich languages editing

Ctrl+Space	Trigger suggestion
Ctrl+Shift+Space	Trigger parameter hints
Shift+Alt+F	Format document
Ctrl+K Ctrl+F	Format selection
F12	Go to Definition
Alt+F12	Peek Definition
Ctrl+K F12	Open Definition to the side
Ctrl+.	Quick Fix
Shift+F12	Show References
F2	Rename Symbol
Ctrl+K Ctrl+X	Trim trailing whitespace
Ctrl+K M	Change file language

Editor management

Ctrl+F4, Ctrl+W	Close editor
Ctrl+K F	Close folder
Ctrl+\	Split editor
Ctrl+ 1 / 2 / 3	Focus into 1st, 2nd or 3rd editor group
Ctrl+K Ctrl+ ←/→	Focus into previous/next editor group
Ctrl+Shift+PgUp / PgDn	Move editor left/right
Ctrl+K ← / →	Move active editor group

File management

Ctrl+N	New File
Ctrl+O	Open File...
Ctrl+S	Save
Ctrl+Shift+S	Save As...
Ctrl+K S	Save All
Ctrl+F4	Close
Ctrl+K Ctrl+W	Close All
Ctrl+Shift+T	Reopen closed editor
Ctrl+K Enter	Keep preview mode editor open
Ctrl+Tab	Open next
Ctrl+Shift+Tab	Open previous
Ctrl+K P	Copy path of active file
Ctrl+K R	Reveal active file in Explorer
Ctrl+K O	Show active file in new window/instance

Display

F11	Toggle full screen
Shift+Alt+1	Toggle editor layout (horizontal/vertical)
Ctrl+ = / -	Zoom in/out
Ctrl+B	Toggle Sidebar visibility
Ctrl+Shift+E	Show Explorer / Toggle focus
Ctrl+Shift+F	Show Search
Ctrl+Shift+G	Show Source Control
Ctrl+Shift+D	Show Debug
Ctrl+Shift+X	Show Extensions
Ctrl+Shift+H	Replace in files
Ctrl+Shift+J	Toggle Search details
Ctrl+Shift+U	Show Output panel
Ctrl+Shift+V	Open Markdown preview
Ctrl+K V	Open Markdown preview to the side
Ctrl+K Z	Zen Mode (Esc Esc to exit)

Debug

F9	Toggle breakpoint
F5	Start/Continue
Shift+F5	Stop
F11 / Shift+F11	Step into/out
F10	Step over
Ctrl+K Ctrl+I	Show hover

Integrated terminal

Ctrl+`	Show integrated terminal
Ctrl+Shift+`	Create new terminal
Ctrl+C	Copy selection
Ctrl+V	Paste into active terminal
Ctrl+↑ / ↓	Scroll up/down
Shift+PgUp / PgDn	Scroll page up/down
Ctrl+Home / End	Scroll to top/bottom

Other operating systems' keyboard shortcuts and additional unassigned shortcuts available at aka.ms/vscodekeybindings

下面再介绍另外一款开发神器——有人二合一网络串口调试工具。

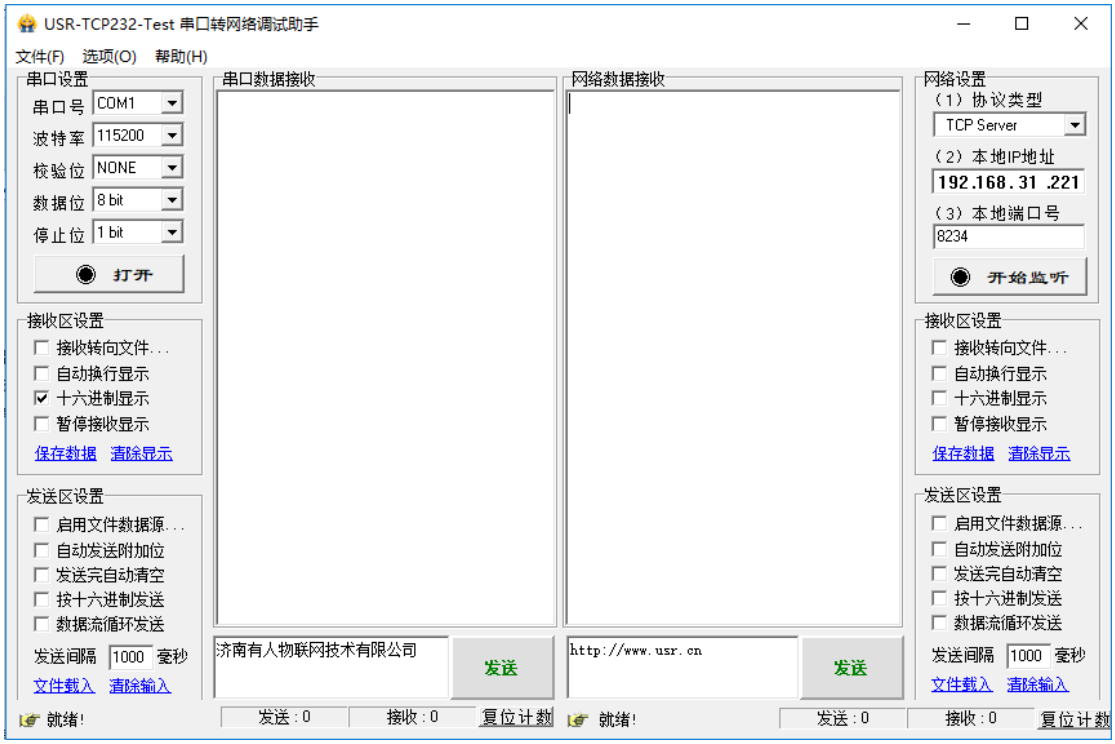


图 4.31　软件界面展示

我们在平时开发时会经常通过串口打印一些日志来查看程序运行状态，比如开机初始化过程，是否执行某条判断语句等等。另外，学习物联网网络调试是必不可少的，比如 TCP、UDP 的测试等等。针对这些开发需求，通常会使用两个独立的软件来回切换，很不方便，但通过有人的串口网络二合一的调试工具使得我们可以在同一个软件界面上就可以查看两种打印信息，提高开发效率。对于如何操作我们在对应章节再讲。

第五章 STM32 库函数讲解

5.1 什么是库

函数库的意义在于它能够为开发者屏蔽一些底层的操作，一般这些底层的操作都是很麻烦且重复的，对于 STM32 的库来说它就是底层操作和用户逻辑之间的一个桥梁，我们看一下这个图加深理解。

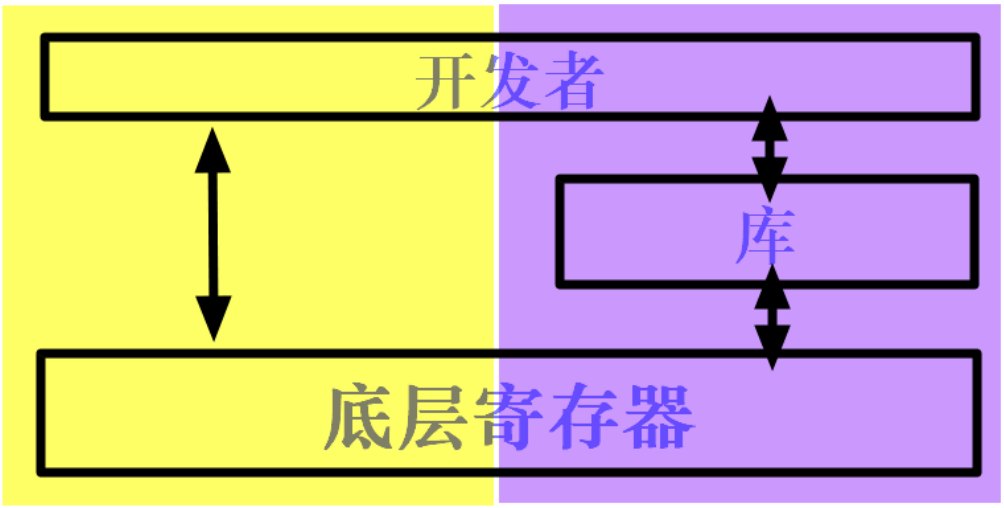


图 5.1 寄存器和库开发图示

其中黄色部分是我们直接对寄存器进行操作的方式，紫色是使用库进行开发的方式，通过该图就可以看出：使用库可以屏蔽底层操作，开发者开发起来更加方便。

5.2 为什么使用库

相信很多初学者都学过 51 单片机，我们对它的控制也极为简单粗暴，直接通过对寄存器的相应位进行位的操作即可，也熟悉了这种开发模式，一旦从 51 转到 STM32，在开发方式上很不适应，主要顾虑如下：

1. 不学习寄存器的操作总觉得学习不深入，没有完全掌握，心里不踏实
2. 担心库函数的执行效率低

对于 51 单片机来说，它的寄存器并不是很多，配置起来也不复杂，所以也就没有开发库函数的必要（这里顺便提一下，STC 官方在几年前已经推出了 STC15 单片机的库函数，代码风格和 STM32 库函数如出一辙，大家感兴趣可以去官网下载）。而对于 STM32 这样的单片机来说寄存器的数量和寄存器的长度都是 51 的好几倍，非常复杂，如果使用寄存器开发，那么大部分时间都会浪费在查阅手册上，得不偿失。ST 官方也意识到这一点，所以就开发

了一套标准外设库函数使得开发者不用直接操作寄存器，只需要学会使用提供给我们的 API 接口即可。

另外，在开发中最忌讳的就是重复造轮子。项目开发的目的是尽快完成功能并保证功能可靠，而不是把时间浪费在不必要的细节上，所以能用现成的就用现成的，因为它更成熟，有更多人用，如果你自己实现一套函数库未必会有官方提供的可靠稳定，所以时间成本大于一切。

其次，这也是初学者在接触新事物的过程中对旧观点否定的矛盾心理，其实不同的开发方式之间并不是对立存在的，只要大家用上一段时间的库函数，你就会认为使用库函数开发理所当然。但是作者并不是否定寄存器的开发方式，只是各有优缺点而已。

另外对于代码执行效率的顾虑也是多余的，现在单片机的速度已经非常快了，就拿 STM32F103 来说，它 1 秒内可以执行七千两百万次单周期指令，所以只是几个函数的跳转和参数类型是否正确的判断对于 STM32 的执行时间的消耗来说微乎其微。

同时使用库函数可以使得你的代码更易维护，就像是为什么单片机开发会更青睐于 C 语言而不是汇编的原因。

5.2 STM32 标准库函数讲解

既然要使用库函数开发，那么就必须知道库函数怎么用，每个库文件都是干嘛的。在讲解之前，我们先下载 STM32 的标准库函数。下载并解压之后，进入目录，看到如下文件结构：

图 5.2　库函数目录结构

我们用到的就是框选的 Libraries 文件夹。进入该文件夹：

图 5.3　Libraries 目录结构

CMSIS 文件夹内存放的文件如下：

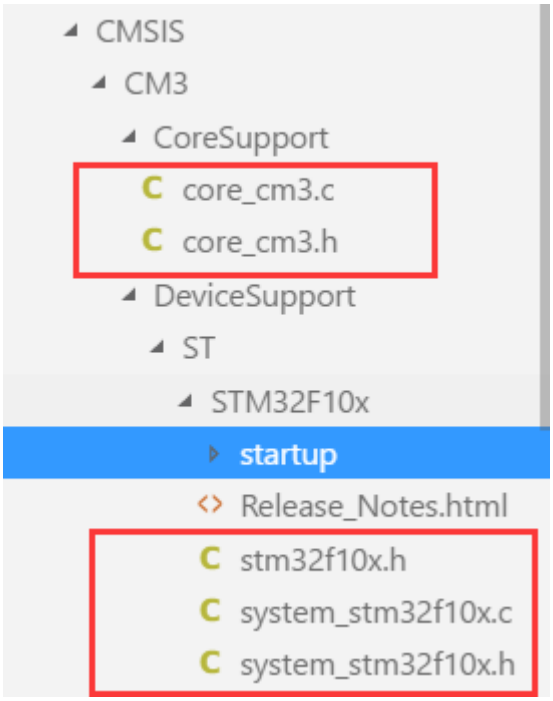


图 5.4　CMSIS 目录结构

- core_cm3.h/.c

该文件定义了 ARM Cortex M3 内核外设的寄存器地址，比如中断，时钟控制等，以及设置寄存器对应位的宏定义。

- stm32f10x.h

该文件定义了 STM32 各个外设的寄存器地址（GPIO，ADC 等）以及设置寄存器对应位的宏定义，这也是 ST 标准库的精髓，ST 将各个外设的寄存器都定义为一个结构体，然后将该外设地址强制转换为结构体指针。看图：

```
1001    typedef struct
1002    {
1003      __IO uint32_t CRL;
1004      __IO uint32_t CRH;
1005      __IO uint32_t IDR;
1006      __IO uint32_t ODR;
1007      __IO uint32_t BSRR;
1008      __IO uint32_t BRR;
1009      __IO uint32_t LCKR;
1010    } GPIO_TypeDef;
```

图 5.5 寄存器结构体定义

由于结构体和数组的存储方式相同，都是连续存储，因此非常适合定义连续空间。再看下图：

```
1408  #define GPIOA              ((GPIO_TypeDef *) GPIOA_BASE)
1409  #define GPIOB              ((GPIO_TypeDef *) GPIOB_BASE)
1410  #define GPIOC              ((GPIO_TypeDef *) GPIOC_BASE)
1411  #define GPIOD              ((GPIO_TypeDef *) GPIOD_BASE)
1412  #define GPIOE              ((GPIO_TypeDef *) GPIOE_BASE)
1413  #define GPIOF              ((GPIO_TypeDef *) GPIOF_BASE)
1414  #define GPIOG              ((GPIO_TypeDef *) GPIOG_BASE)
```

图 5.6 寄存器地址强制转换

以 GPIOA 为例，后面 GPIOA_BASE 就是一个整数，我们把一个整数强制转换为 GPIO_TypeDef 类型指针，并通过使用 GPIOA->的结构体指针操作来设置对应的寄存器。如下图：

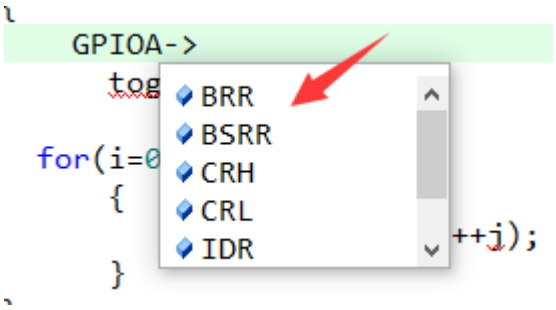


图 5.7 编程范例

另外在该文件中，通过 typedef 关键字为开发者声明了常见类型的别名定义，比如:unsigned char 被定义为 u8，很明显后者更易理解也更方便键盘输入。

- system_stm32f10x.h/.c

该文件实现了对系统时钟进行初始化，在单片机上电之后，会自动执行该文件中的系统时钟初始化函数，默认系统时钟频率为 72MHz。

```
109   #else
110   /* #define SYSCLK_FREQ_HSE    HSE_VALUE */
111   /* #define SYSCLK_FREQ_24MHz  24000000 */
112   /* #define SYSCLK_FREQ_36MHz  36000000 */
113   /* #define SYSCLK_FREQ_48MHz  48000000 */
114   /* #define SYSCLK_FREQ_56MHz  56000000 */
115   #define SYSCLK_FREQ_72MHz  72000000
```

图 5.8 设置系统时钟

箭头所指宏定义表示 STM32 工作的时钟频率。这里设置为最大，即 72Mhz。

```
419   static void SetSysClock(void)
420   {
421   #ifdef SYSCLK_FREQ_HSE
422     SetSysClockToHSE();
423   #elif defined SYSCLK_FREQ_24MHz
424     SetSysClockTo24();
425   #elif defined SYSCLK_FREQ_36MHz
426     SetSysClockTo36();
427   #elif defined SYSCLK_FREQ_48MHz
428     SetSysClockTo48();
429   #elif defined SYSCLK_FREQ_56MHz
430     SetSysClockTo56();
431   #elif defined SYSCLK_FREQ_72MHz
432     SetSysClockTo72();
433   #endif
```

图 5.9 生效时钟设置

最后系统时钟初始化函数 SystemInit 会调用 SetSysClock 函数，从而完成时钟初始化动作，这里顺便提一下，我们无需再 main 函数里面调用时钟初始化函数，在设备启动后，该初始化过程会在 main 函数之前被调用，大家感兴趣可以进入调试模式看一下启动时的汇编程序，也可以参考下面的一篇博客，加深理解。

- startup 文件夹下的启动文件

针对不同编译器 ST 把启动文件进行了分类，我们使用 ARM 文件夹下的启动文件，前面说过 STM32F103 分为高中低三种密度，对应的启动文件也分为三种，针对 STM32F103 可能用到的有 3 个，大家根据自己使用的芯片进行选择：

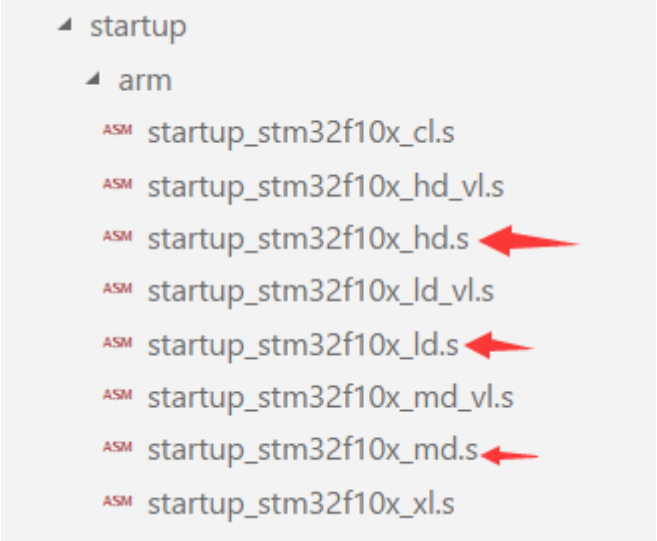


图 5.10 启动文件选择

其他的启动文件分别是互联型和 STM32F100 系列单片机启动文件。

启动文件是汇编程序，它用于设置单片机堆栈大小并控制单片机启动过程，另外我们可以在启动文件中找到 SystemInit,这就是说此处调用了该函数。

再来看一下 Libraries\STM32F10x_StdPeriph_Driver 路径下的文件该路径下的文件是 STM32 各个外设函数实现和声明的地方，具体的我们在各个例程中进一步学习。

另外在平时开发的时候一般还要使用下面文件：

图 5.11 启动文件选择

第一个文件用于设置在编译的时候使用哪些外设的头文件。

```
#include "stm32f10x_adc.h"
#include "stm32f10x_bkp.h"
#include "stm32f10x_can.h"
#include "stm32f10x_cec.h"
#include "stm32f10x_crc.h"
#include "stm32f10x_dac.h"
#include "stm32f10x_dbgmcu.h"
#include "stm32f10x_dma.h"
#include "stm32f10x_exti.h"
#include "stm32f10x_flash.h"
#include "stm32f10x_fsmc.h"
#include "stm32f10x_gpio.h"
#include "stm32f10x_i2c.h"
#include "stm32f10x_iwdg.h"
#include "stm32f10x_pwr.h"
#include "stm32f10x_rcc.h"
#include "stm32f10x_rtc.h"
#include "stm32f10x_sdio.h"
#include "stm32f10x_spi.h"
#include "stm32f10x_tim.h"
#include "stm32f10x_usart.h"
#include "stm32f10x_wwdg.h"
#include "misc.h" /* High level func
```

图 5.12　外设配置选项

另外 stm32f10x_it.h/.c 文件是 ST 为了统一管理中断函数，所以把各个中断函数都集中到了一起存放，当然大家也可以在任何一个.c 文件中放置中断函数，但是要保证整个工程中的某个中断函数不重复。那么有人会问，我去哪里查看所有中断函数呢？你可以通过两种途径，第一个就是把 stm32f10x_it.h/.c 的中断函数剪切过来，另外一个方法：你可以查阅启动文件中对中断函数的定义，如下图：

```
DCD     WWDG_IRQHandler             ; Window Watchdog
DCD     PVD_IRQHandler              ; PVD through EXTI Line detect
DCD     TAMPER_IRQHandler           ; Tamper
DCD     RTC_IRQHandler              ; RTC
DCD     FLASH_IRQHandler            ; Flash
DCD     RCC_IRQHandler              ; RCC
DCD     EXTI0_IRQHandler            ; EXTI Line 0
```

只截取一部分

图 5.13　中断函数参考

图中带_IRQHandler 的都是中断函数名，大家复制过来用即可。

到这里大家对 ST 的库有了一些认识，但是还会有人多疑惑，所以最好的办法是大家亲自去浏览一下这些文件的内容，同时也可以学习一下人家的编程规范。

另外跟着我们后面的例程走，相信大家会慢慢熟悉标准库函数的使用的。

第六章 建立一个 MDK 工程

6.1 万事开头难

作者单独拿出一章来带领大家熟悉并建立一个 MDK 工程的原因是因为很多初学者被工程和文件夹的概念弄得焦头烂额，他不清楚工程以及工程中各个文件夹的作用、为什么这么添加。因此有必要详细的向初学这讲解，对于这部分很熟悉的读者可以跳过。

工程的概念用于将与该工程相关的文件管理起来，这样做的好处是我们能宏观的管理、修改、查看工程内的文件以及内容。

在新建工程之前，我要先讲解一下风媒电子的工程目录结构，让大家先知道为什么这么做然后再去学习如何做。

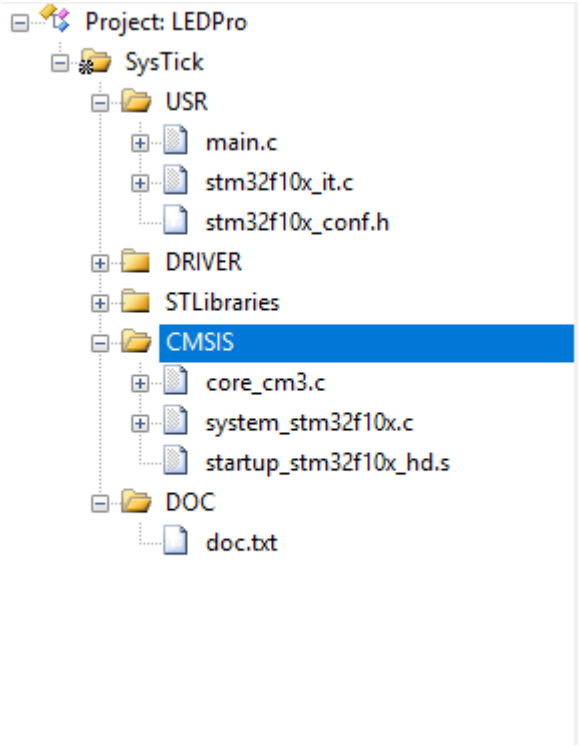


图 6.1 工程结构示例

上图可以看到 LED 工程分为五个目录管理。

- USR

  该目录下存放用于实现用户逻辑的代码以及 main 函数,它是实现整个项目逻辑功能的地方。

- DRIVER

  该目录存放供 USR 目录中的业务逻辑函数调用的各个硬件功能模块的接口函数，例如 LED 驱动、按键等。

- STLibraries 目录

  该目录下存放的就是第五章讲到 **STM32 外设**的标准库文件

- CMSIS 目录

  该目录存放的是第五章讲到的 **ARM Cortex M3 内核外设**的库文件

- DOC 目录

  该目录存放一个 txt 文本文件，不参与工程编译，只是用于记录开发过程中的日志，比如版本号，或者在何时做了哪种更改。

看到这里相信大家对整个工程目录目录的作用还不是很清楚，下面作者用一张图来加深理解。

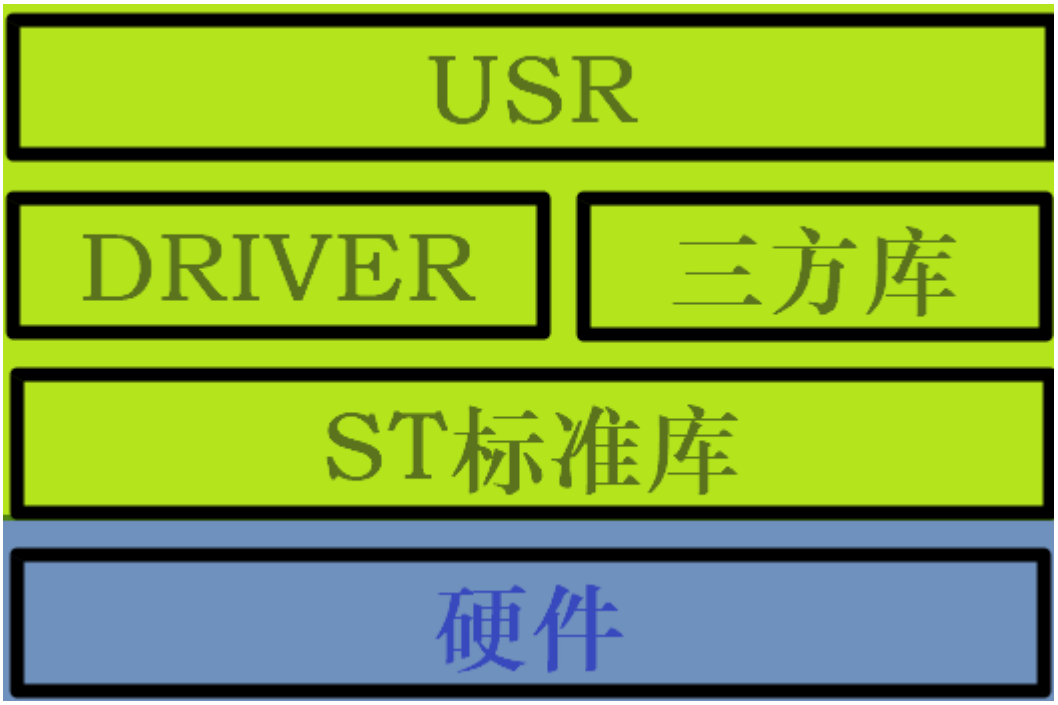


图 6.2　工程分层图

上图由下向上依次是调用关系，从图中可以看出在 DRIVER 层实现的函数都是通过调用 ST 标准库提供的函数接口来操作硬件的。另外三方库是指我们要实现一些功能时可以使用现有的封装好的库来实现，比如 USB 库、LCD 显示二维码、虚拟示波器等等，但是在大多数例程中是没有用到的。最顶层的用户层（USR），用来调用比它层级低的层级提供的接口来实现工程的业务逻辑代码，换句话说 USR 层是用于实现项目要求的地方。

6.2 新建工程

第一步：新建工程文件并保存

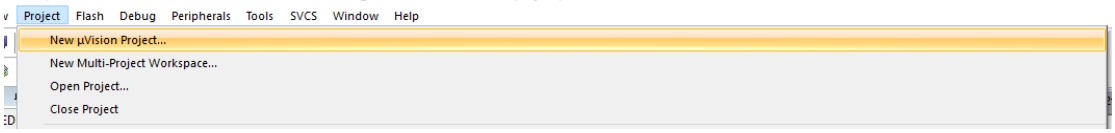


图 6.3　新建工程示例

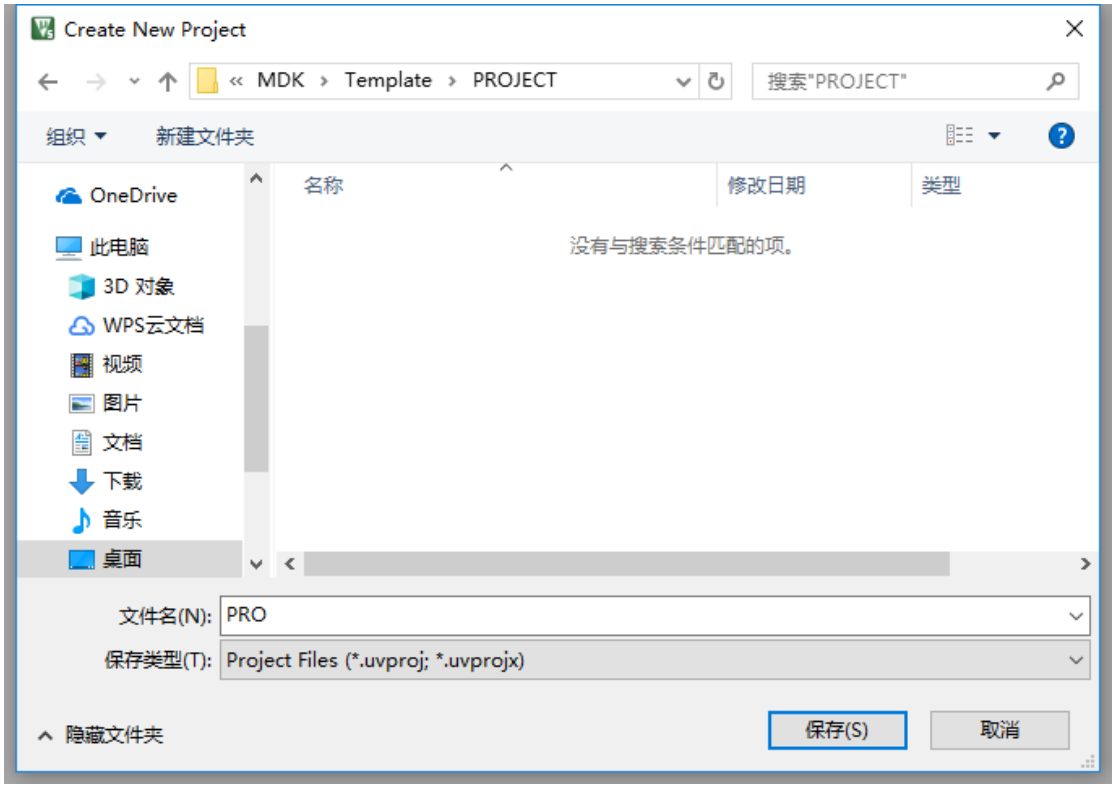


图 6.4　保存工程

第二步：选择单片机

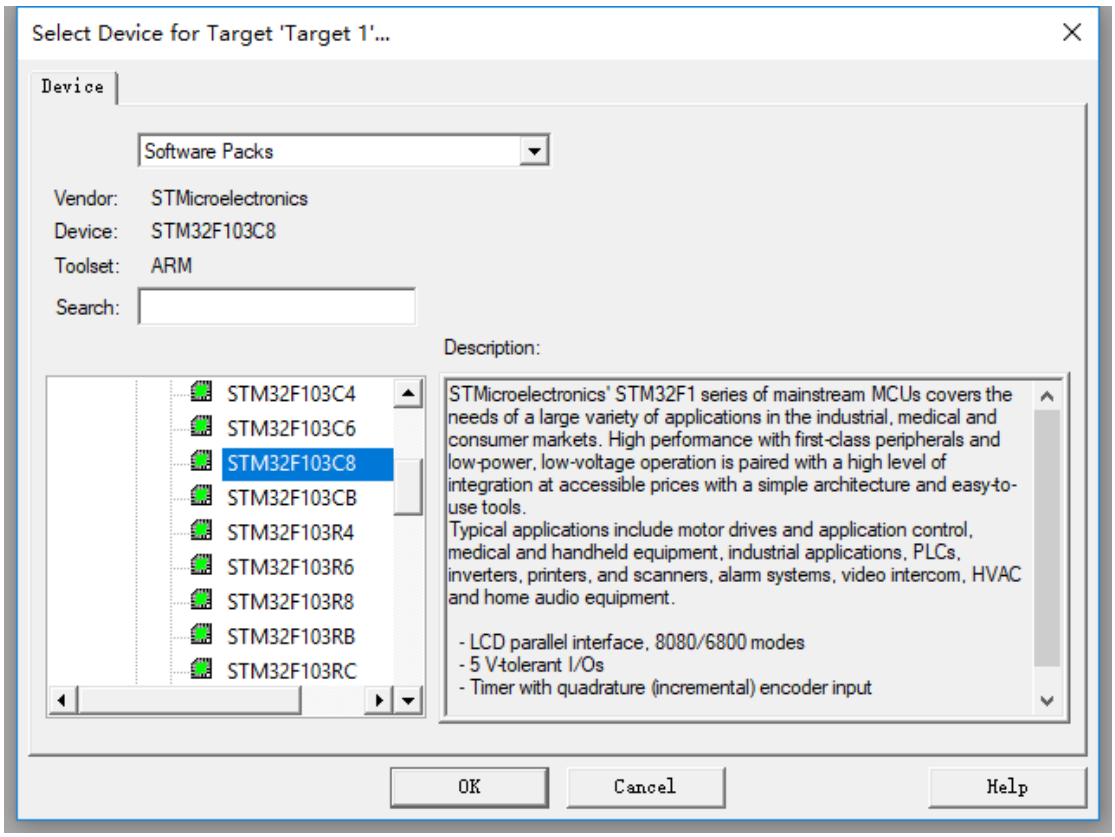


图 6.5　选择单片机

选中并确认之后，会弹出包管理界面，此时可以选择对应外设库添加到工程，这也是新版本 MDK 的特色，但是我们这里不采用这种方法添加外设库文件，目的是想兼容以前 4.x 版本的操作方式，因为大多数院校还在采用 4.x 的版本，这一步我们点击取消。

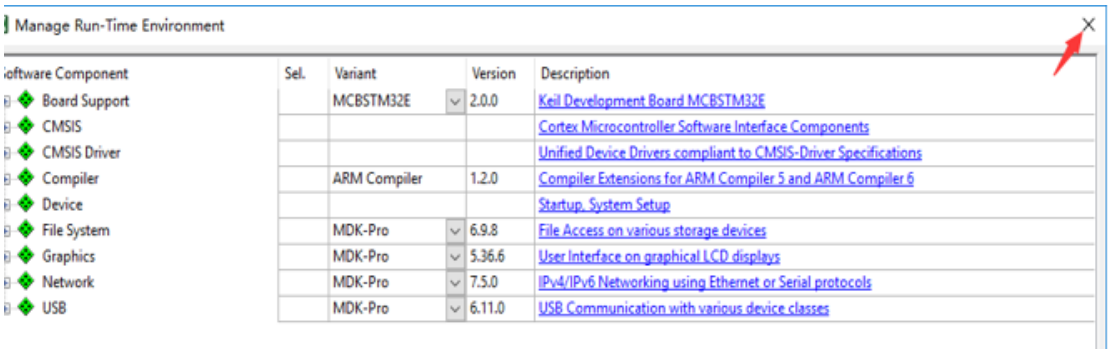


图 6.6　软件包

最后得到的默认工程如下：

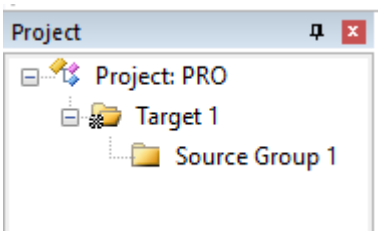


图 6.7　默认空工程

第三步：建立工程目录并添加文件

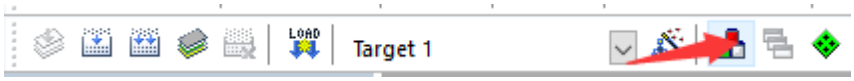


图 6.8　工程管理设置

点击三个小箱子图标，它的作用是用于管理工程当中的各个成员，得到结果如下：

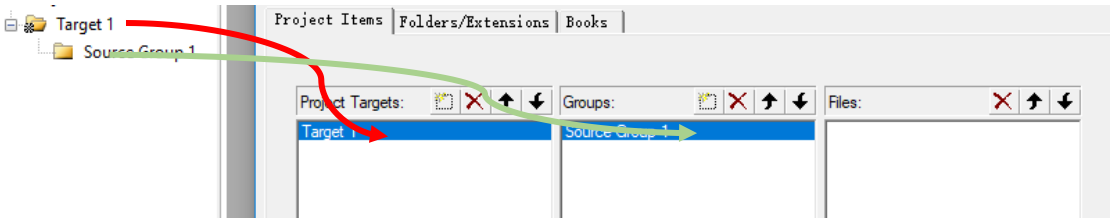


图 6.9　设置工程成员

从左到右依次是工程名，工程目录，以及对应目录下的文件。现在按照 6.11 的方式新建各个目录并添加文件，如下图：

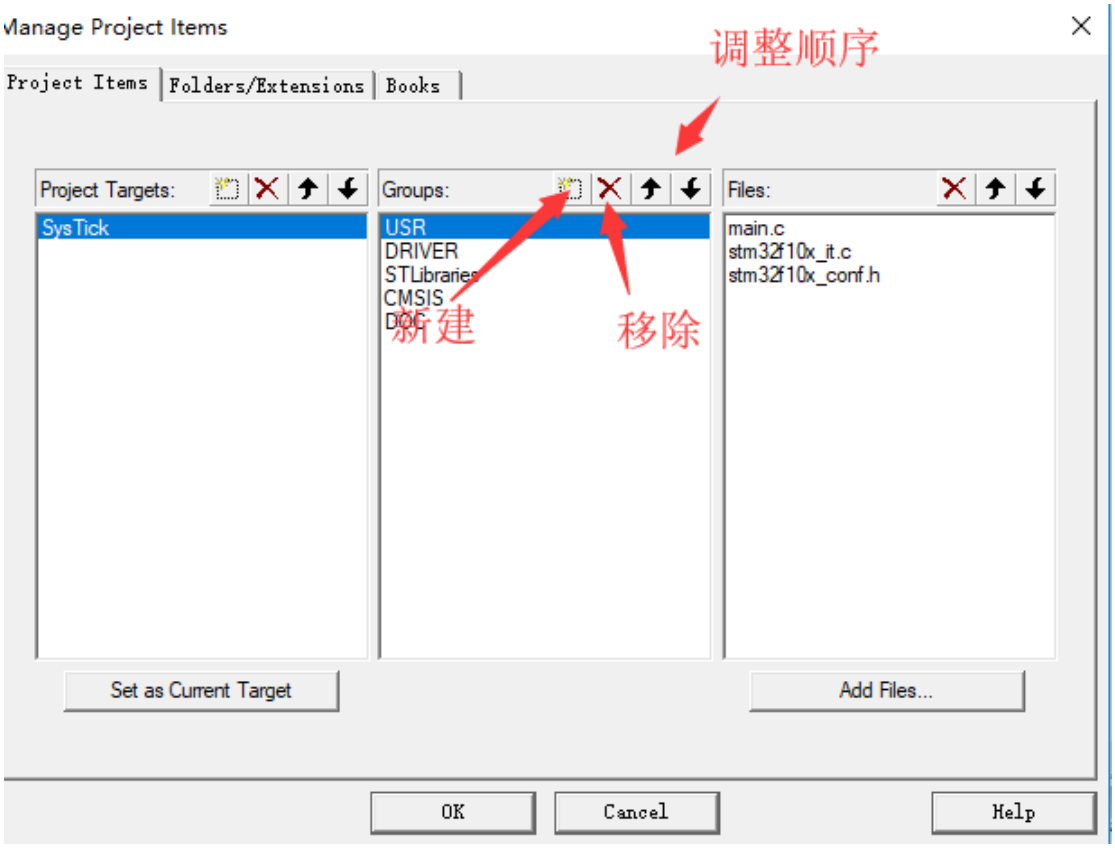


图 6.10 向工程中添加项目目录

此时得到没有文件的工程目录，接下来我们新建文件，然后添加到对应目录。

使用快捷键 CRL+N 新建一个文件然后，CTRL+S 保存到对应文件夹内。但是此时文件并没有添加到工程目录当中去，我们可以双击某个目录选择添加的文件，或者再次点击三个小

箱子按钮来添加文件，最后文件添加结果如下：

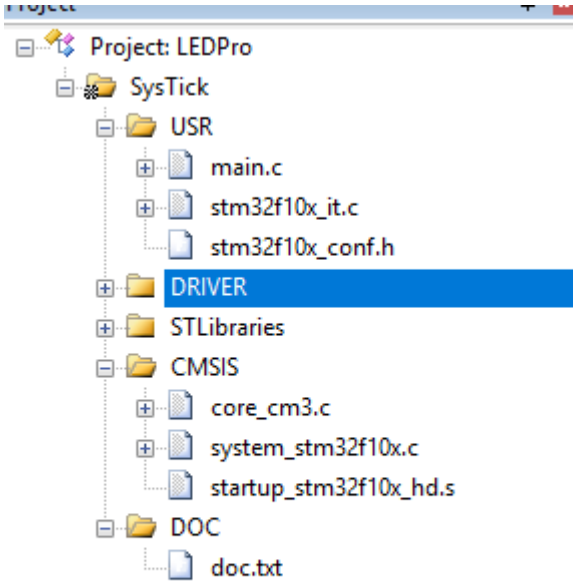


图 6.11　向工程目录中添加文件

第四步：设置工程参数

打开（魔术棒）  ，进行如下设置：

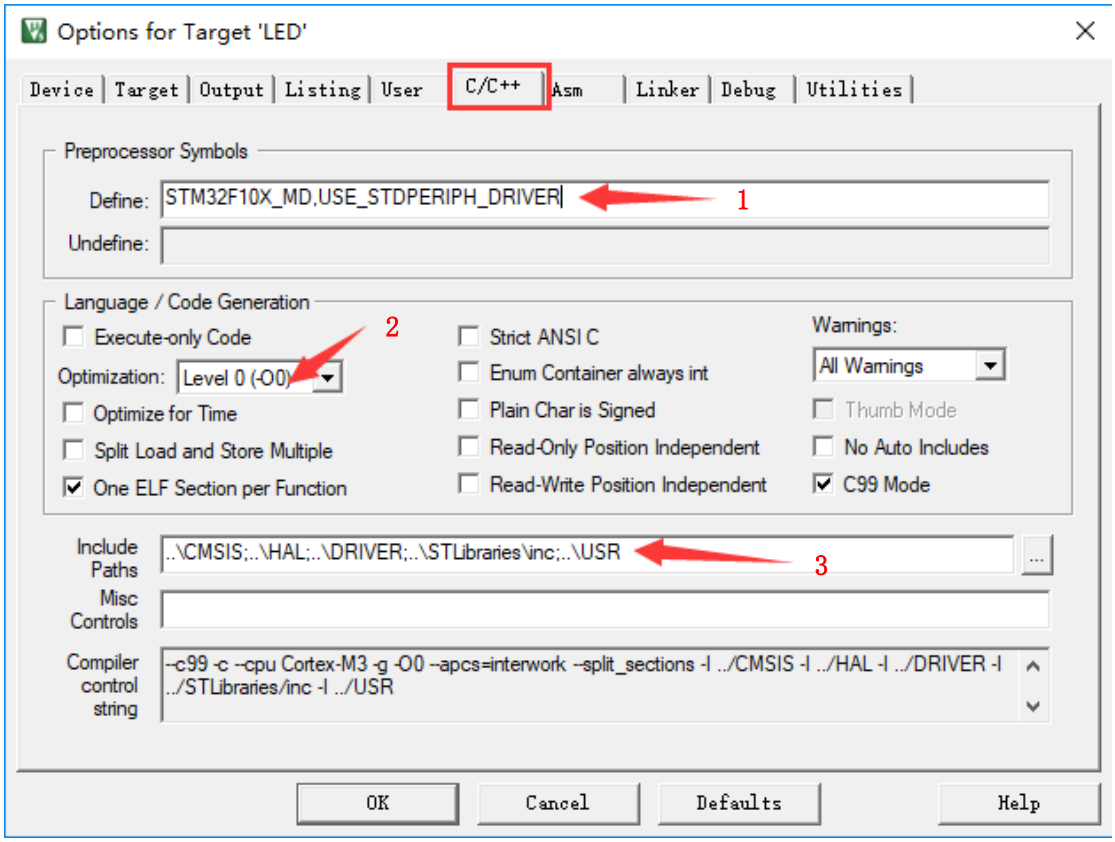


图 6.12　配置工程

1. 这个是全局的预处理宏定义，它等效于在文件中进行#define 宏定义，那么我们来看一下这两个宏定义都干什么的：

STM32F10X_MD:因为ST提供这一套标准库他并不是针对某一具体芯片型号开发的而是适用于所有 STM32F10x 系列的单片机，由于 STM32F10x 系列单片机不同型号之间的外设种类和数量不同，所以在具体应用中必须要开发者指定用的是哪一款具体型号的单片机，这也是这个宏定义的意义。当然也可以在文件中定义这个宏，根据你使用的单片机类型来参考如下操作（位于 stm32f10x.h 文件中）：

```
65 #if !defined (STM32F10X_LD) && !defined (STM32F10X_LD_VL) && !defined (STM32F10X_MD) && !defined (STM32F10X_MD_VL)
66   /* #define STM32F10X_LD */     /*!< STM32F10X_LD: STM32 Low density devices */
67   /* #define STM32F10X_LD_VL */  /*!< STM32F10X_LD_VL: STM32 Low density Value Line devices */
68    #define STM32F10X_MD          /*!< STM32F10X_MD: STM32 Medium density devices */
69   /* #define STM32F10X_MD_VL */  /*!< STM32F10X_MD_VL: STM32 Medium density Value Line devices */
70   /* #define STM32F10X_HD */     /*!< STM32F10X_HD: STM32 High density devices */
71   /* #define STM32F10X_HD_VL */  /*!< STM32F10X_HD_VL: STM32 High density value line devices */
72   /* #define STM32F10X_XL */     /*!< STM32F10X_XL: STM32 XL-density devices */
73   /* #define STM32F10X_CL */     /*!< STM32F10X_CL: STM32 Connectivity line devices */
```

图 6.13 对应单片机选择的宏定义

想用哪个就取消掉哪个的注释即可。

USE_STDPERIPH_DRIVER:它的作用是可以让开发者配置使用的标准库外设，看下图（位于 stm32f10x.h 文件中）：

```
#ifdef USE_STDPERIPH_DRIVER
  #include "stm32f10x_conf.h"
#endif
```

图 6.14 预定义讲解

stm32f10x_conf.h 在之前讲过，它用于选择此时项目依赖的外设，这样做的好处是，我们在各个外设驱动文件中直接包含 stm32f10x.h 即可，无需再包含对应外设头文件，比如:stm32f10x_gpio.h,stm32f10x_rcc.h 等。同样我们也可以在文件中自行实现：

```
 99 #if !defined  USE_STDPERIPH_DRIVER
100 /**
101   * @brief Comment the line below if you will not use the
102     In this case, these drivers will not be included and
103     be based on direct access to peripherals registers
104     */
105   #define USE_STDPERIPH_DRIVER
106 #endif
```

图 6.15 预定义讲解

只要把这一行注释取消即可。

2. 我们再来看一下 2 号标记处的作用，这个一般情况下保持默认即可，它用于设置程序编译时的优化程度，程序优化程度越高，代码量越小，执行速度也更快。但是过度优化会带来一些难以察觉的错误，比如作者之前用 IAR for STM8 给 STM8 单片机编程，要使用硬件定时器实现延时并通过 while 条件判断延时结束,演示代码如下：

```
void Delay_ms(uint16_t ms)
{
    uint8_t counts = 0;
    wihle(counts<ms)
    {
        Delay1ms();
    }
}
```

图 6.16 过度优化讲解

因为我进行了优化设置，就导致框选部分的表达式被当做了一个不为 0 的常量处理了，所以程序会陷入无休止的死循环当中，程序就卡死了。当时排查很久才发现，当然在 MDK 中进行优化还没有发现异常，但不代表不存在，这里要表达的就是如果你的程序如果不工作并且你认为你的程序和硬件没问题，那么你就检查一下优化程度吧。

3. 标号 3 用于设置程序编译时头文件的路径，这么做的好处是我们无需在包含头文件时把很长的头文件路径也包含进去了，换句话说你这么做会告诉编译器编译时需要的头文件要到你指定的这些目录中去查找（本质就是更改 Makefile 文件）。比如我们这里添加的这些目录：

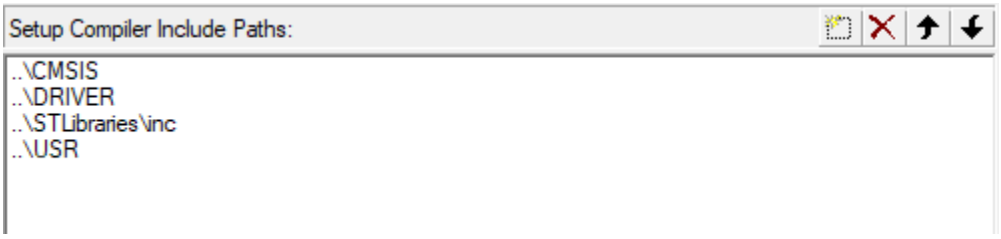


图 6.17 添加头文件路径

还要注意一点：我们只包含了 DRIVER 目录，但是在实际开发中 DRIVER 文件夹下面还会新建的其他文件夹来存放对应的外设文件，比如：

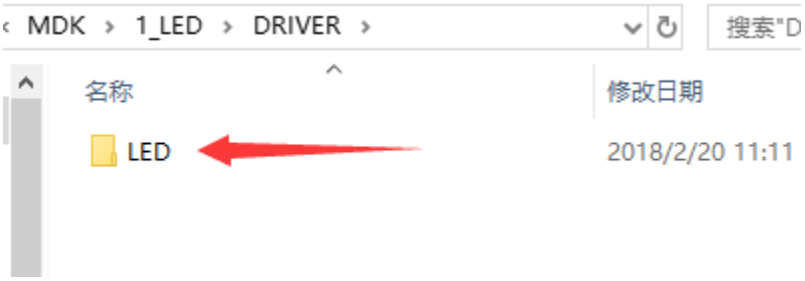


图 6.18　DRIVER 路径下存放对应外设文件夹

那么，在调用 LED 文件的时候可以这么包含 LED 头文件：

```
#include "LED/LED.h"
```

当然大家可以继续把 LED 目录也添加到头文件路径里，这只不过是习惯问题。

第五步：编写程序然后编译生成目标文件。

先看最常用的工具栏按钮：

图 6.19　编译功能按钮

从左到右依次是编译，构建工程（链接），重新构建工程（重新链接），对不同目标工程进行编译链接操作（不常用），最右面的是将二进制文件烧录到单片机的烧录按钮。这里需要注意的是我们没有必要先点编译，再点链接，在点击链接时会判断在此之前是否进行过编译，如果没有就先编译再链接，如果有，则直接链接。另外重新链接和链接的区别在于：如果我们对程序做了改动并且之前编译链接过，那么此时点击链接按钮会链接的会快一些，因为它只编译链接了改动部分，而重新链接则是将所有部分都编译链接一遍，时间自然就会长一点。在平时开发时不会分的这个细，一般点一下链接按钮就行了，没有错误再点烧录按钮。另外想要编译后生成 HEX 文件，可以点击魔术棒然后按照如下设置：

图 6.20　设置生成 HEX 文件

如果希望 MDK 有代码补全功能，则必须要先编译一下工程才行，所以还是推荐大家用 VS CODE 进行代码编辑，界面友好效率更高。

到这里新建工程的任务就结束了，大家可以动手操作，最好是参照我们的例程目录结构来练习，先模仿，后创新。

第七章 GPIO 输出实验_点亮 LED

从这章开始，我们正式步入了单片机教学部分。学会使用单片机是开发项目的前提，作者希望大家除了学习 STM32 单片机之外，更重要的是要学会单片机开发的的通用编程方法，虽然不同单片机的开发方式有异，但是学会一种单片机之后再学习其他单片机的时候就会简单很多。

另外作者在讲解如何对单片机编程的同时还会讲解开发板的原理图，让大家在学习单片机的同时再学习一些电路知识。

注：本章只用到核心板，对应例程 1。

7.1 项目要求

通过驱动 STM32 的 GPIO 让开发板上的 LED 灯实现定时闪烁。

7.2 原理讲解

LED 灯也叫发光二极管，广泛应用在家居照明，其特点是亮度高、功耗低、寿命长且价格便宜，LED 灯有不同类型的封装，外观有很大的差异，但是它们的原理都是相同的，另外由于它是二极管的一种，因此也具有单向导电的特性，在电压反接时断路。下面是直插 LED 和贴片 LED 的示意图（封装种类远比展示要多）：

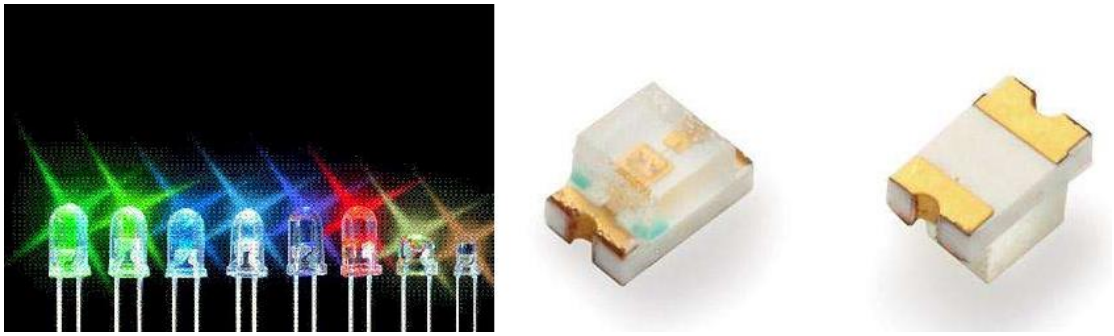


图 7.1 LED 灯展示

开发板上 LED 的原理图如下：

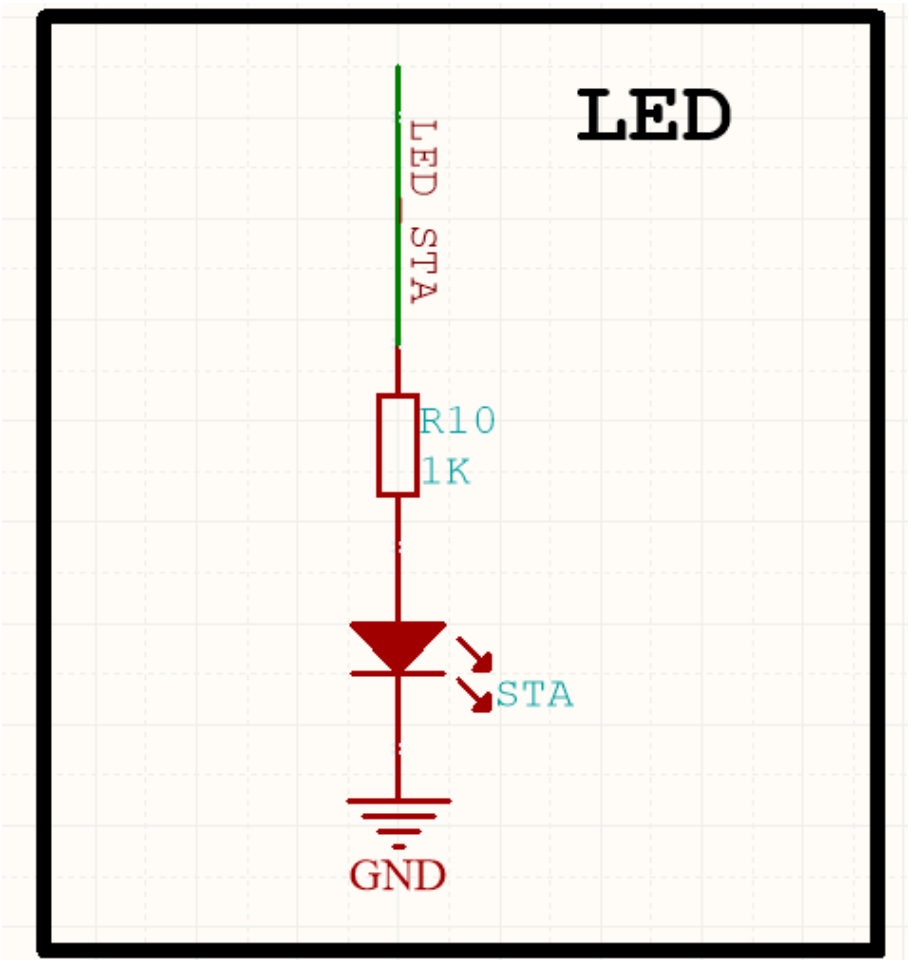


图 7.2　LED 原理图

众所周知，二极管一个很重要的特性就是它的正向压降，一般的二极管的正向压降都会小于等于 0.7V，而发光二极管的压降会比一般的二极管高出很多，而且不同颜色压降也不同，如下图：

注：1.无特殊标注，尺寸单位为mm　2. 公差范围在+0.1mm

产品 Product	型号 Pay numbei	胶体颜色 Leds color	发光颜色 Source	色温 CCT (K)	亮度 Min	亮度 Typ	正向电压 Typ	正向电压 Max	发光角度 Angle
0805	0805UWC	黄色透明	白色	6000-8000K	207	249	2.8V	2.9V	120°
0805	0805URC	无色透明	红色	620-625nm	100	150	2.0V	2.3V	120°
0805	0805UYC	无色透明	黄色	588-590nm	95	115	2.0V	2.1V	120°
0805	0805UBC	无色透明	蓝色	463-466nm	50	60	2.8V	2.9V	120°
0805	0805UGC	无色透明	翠绿	517.5-520nm	230	276	2.6V	2.8V	120°
0805	0805UGC	无色透明	橙色	602-603nm	110	130	2.0V	2.1V	120°
0805	0805UGC	无色透明	黄绿	572-573nm	38	45	2.0V	2.1V	120°

图 7.3　发光二极管压降（数据来自优信电子）

另外发光二极管的电流一般要保证小于 20ma（大功率照明 LED 除外），否则寿命会有很大衰减。因此我们不可能直接将 LED 接到电源，而是通过串联限流电阻来保证流过 LED 的电流大小，计算公式为：

$$I_{LED} = (VCC-V_{LED})/R$$

公式中的 V_{LED} 是 LED 的正向压降，通常取 1.7V。阻值 R 在 300Ω 到 1KΩ 之间选取。

现在我们开始驱动 LED，想要 LED 发光，那么就必须保证阳极和阴极之间存在一个正向压差且这个正向压差要大于 LED 的压降，这里采用 IO 口（3.3V）直接驱动 LED，当 IO 口输出高电平的时候 LED 点亮，输出低电平则熄灭，因此我们只需要操作 IO 为输出，并控制它的输出电平状态即可。

在开始学习 GPIO 控制 LED 之前先了解两个概念：端口复用和重映射。

之前介绍过 STM32 的外设功能十分的丰富，有各类通信的接口（UART，SPI，IIC 等），也有 ADC，PWM 输出等，这些都必须通过 IO 口来实现功能，因此这里就涉及到端口复用概念，我们可以通过软件来设置此时使用哪种外设，**复用的概念解决了外设数量与单片机引脚数量不一致所导致的问题。**

STM32 的一个 IO 口会有很多外设复用，如下图：

PB13	I/O	FT	通用IO PB13	SPI2_SCK/I2S2_CK 复用功能 USART3_CTS(9)/ TIM1_CH1N

图 7.4　PB13 端口复用情况

上图可以看到，一个 IO 口可以被很多外设复用，那么如果此时想要使用 PCB13 作为 IO 用，同时又想使用 UART3_CTS 功能怎么办，这里就又引出了端口重映射的概念，即我们通过软件可以设置某个具有重映射功能的外设由 A 引脚映射到 B 引脚（假设此时 B 引脚没有用到），**通过这种机制就解决了多种外设同时使用同一引脚时所引起的冲突问题。**

STM32 每个 GPIO 端口有两个 32 位配置寄存器(GPIOx_CRL， GPIOx_CRH)，两个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR)，一个 32 位置位/复位寄存器(GPIOx_BSRR)，一个 16 位复位寄存器(GPIOx_BRR)和一个 32 位锁定寄存器(GPIOx_LCKR)。他们可以通过软件配置寄存器来指定 GPIO 的输入输出功能，常用 GPIO 功能如下：

- 输入浮空
- 输入上拉
- 输入下拉
- 模拟输入
- 开漏输出
- 推挽式输出
- 推挽式复用功能
- 开漏复用功能

STM32 GPIO 结构如下：

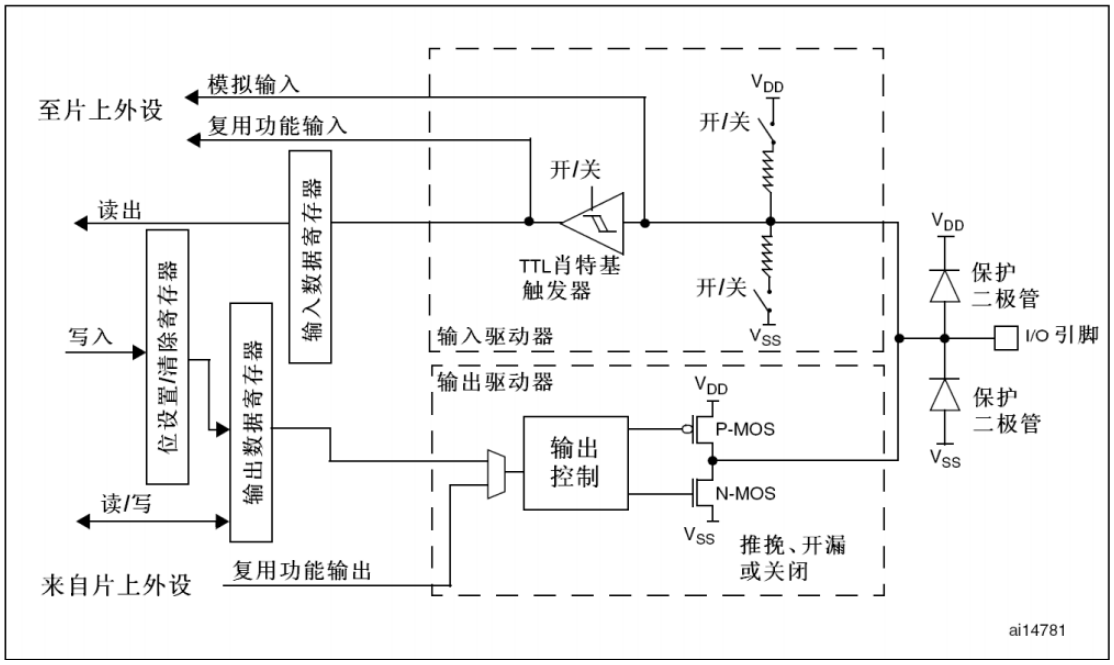


图 7.5　STM32 GPIO 结构图

接下来根据这张图讲解一下各个输入输出配置的信号流向。

- 浮空输入

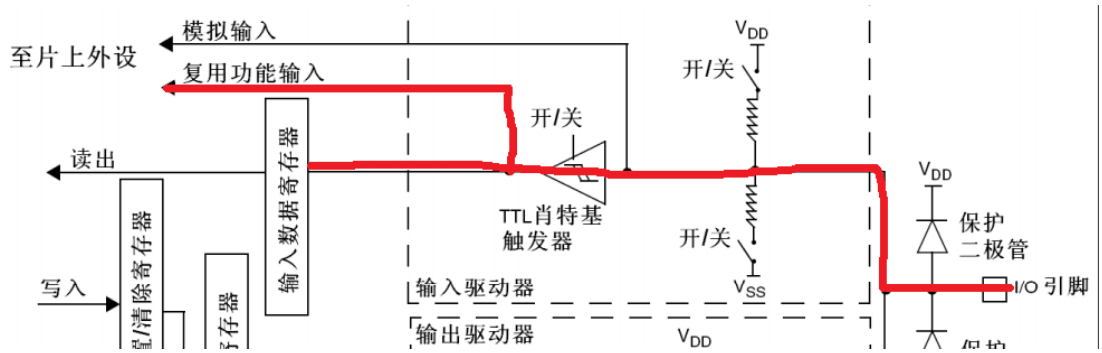


图 7.6　STM32 浮空输入

浮空输入是指即不上拉到 VDD 也不上拉到 VSS，此时它是“浮空”的，因此状态是不确定的，这就需要外部电路来给它一个状态，比如外部高电平或者低电平，我们在使用定时器的输入捕获功能时就需要将 IO 设置为浮空输入，那么可以保证单片机收到的电平状态一定就是来自于外部的电平信号。

- 模拟输入

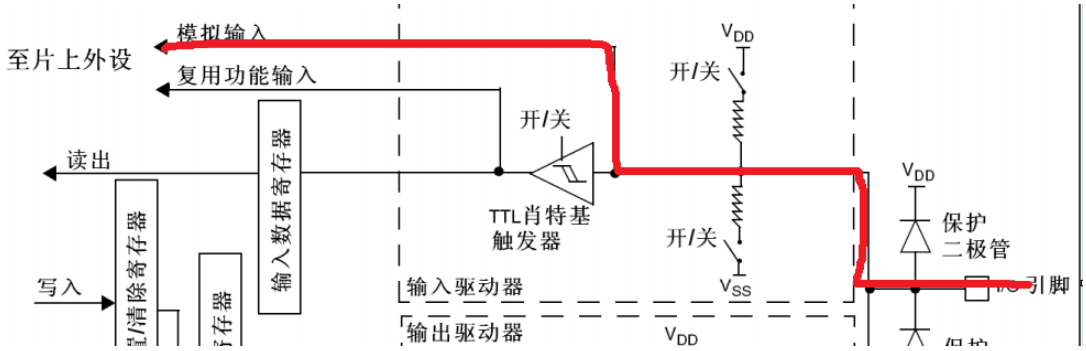


图 7.7　STM32 模拟输入

模拟输入的其实是也是浮空输入，只不过没有走 TTL 施密特触发器（译文错误），因为我们希望得到流入引脚的是模拟信号而不是两种状态的 0 或 1 的数字信号。

- 上下拉输入

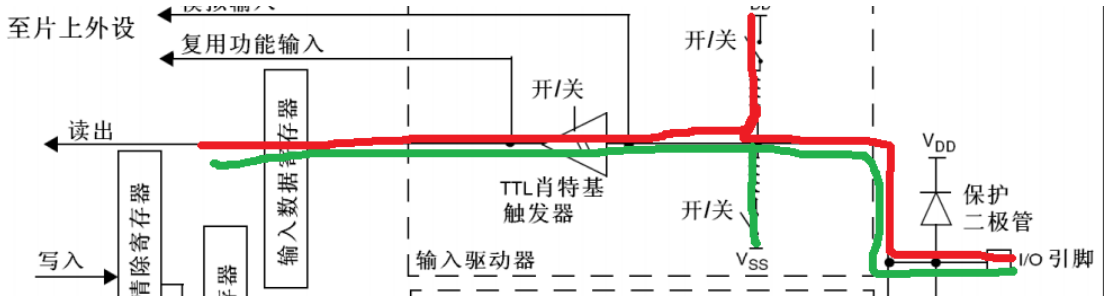


图 7.8　STM32 上下拉输入

上下拉输入我合并到一起讲解，上拉就是通过电阻将引脚拉到 VCC，空闲状态一直是高电平，程序读取到的数据为 1，反之下拉读取到的为 0。

- 推挽输出

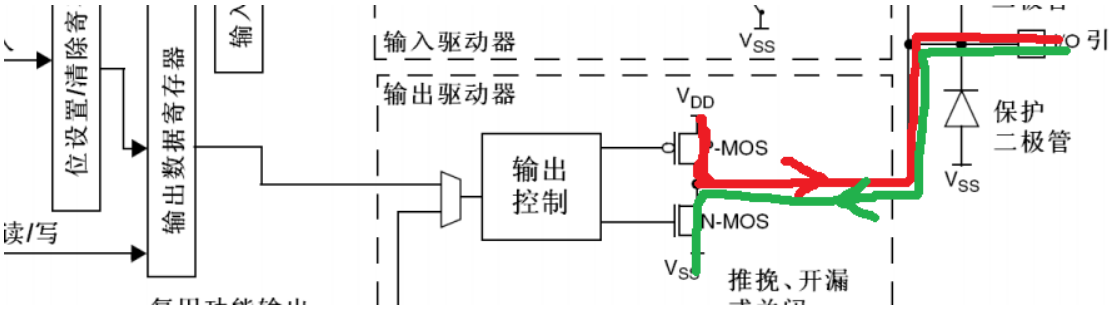


图 7.9　STM32 推挽输出

当某个引脚的输出寄存器设置位为 1 是，对应 PMOS 导通 NMOS 截止，IO 口输出高电平，此时引脚向外输出电流（拉电流），当输出寄存器位 0，则对用 PMOS 截止，NMOS 导通，此

时引脚等效于接到单片机的地，电流由外部电路流入单片机（灌电流），但是大家在电路设计的时候要保证每个 IO 口拉/灌电流不要超过 25ma 且流过单片机的电流不要大于 150ma，否则会减损单片机寿命。

- 开漏输出

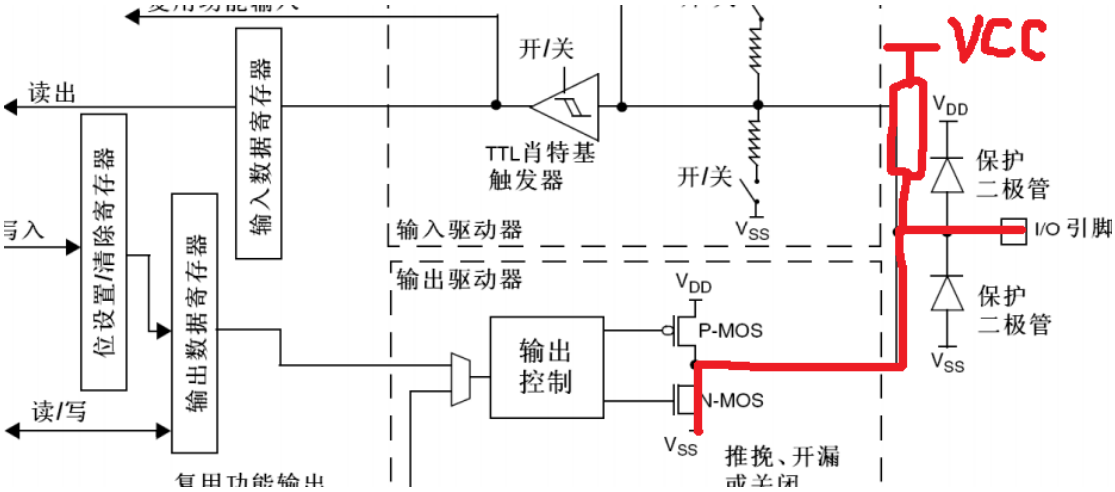


图 7.10　STM32 开漏输出

开漏指的是集电极开漏，也就是说集电极悬空，那么此时需要外部的上拉电阻来连接 IO 口，学过 51 单片机的都知道 P0 端口需要连接上拉电阻，原因就是 P0 的 8 个 IO 口都是集电极开漏输出的，集电极开漏的好处在于，用户可以通过更改外部上拉电阻来自定义拉电流，提高 IO 口驱动能力（此时要考虑当 NMOS 导通时灌电流是否超过最大值），另外也可以通过集电极开漏输出做电平匹配，比如现在想通过 STM32 去和一个 2.8V 的器件通信，那么我们完全可以将电阻上拉到 2.8V，除此之外，我们还可以将多个集电极开漏输出的 IO 通过共用一个上拉电阻的方式来实现线与的功能。

对于复用推挽或者复用开漏和上面的结构一致，只不过数据接通的是对应的复用外设。

接下来介绍本例程用到的 GPIO 寄存器。

- GPIO 的端口配置寄存器（GPIOx_CRL/ GPIOx_CRH）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[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
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 0…7) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位29:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 0…7) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度50MHz

图 7.11　端口配置寄存器低 32 位

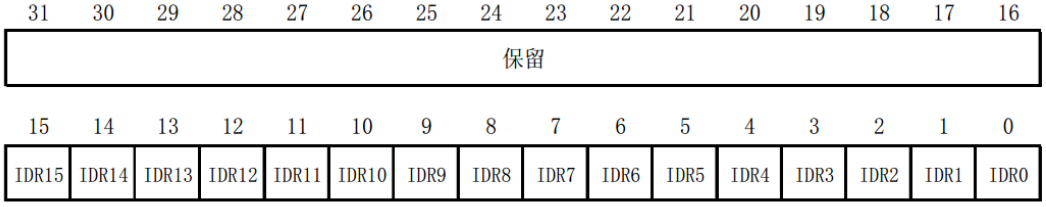
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF15[1:0]		MODE15[1:0]		CNF14[1:0]		MODE14[1:0]		CNF13[1:0]		MODE13[1:0]		CNF12[1:0]		MODE12[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF11[1:0]		MODE11[1:0]		CNF10[1:0]		MODE10[1:0]		CNF9[1:0]		MODE9[1:0]		CNF8[1:0]		MODE8[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:30 27:26 23:22 19:18 15:14 11:10 7:6 3:2	**CNFy[1:0]**：端口x配置位(y = 8…15) (Port x configuration bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 在输入模式(MODE[1:0]=00)： 00：模拟输入模式 01：浮空输入模式(复位后的状态) 10：上拉/下拉输入模式 11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位9:28 25:24 21:20 17:16 13:12 9:8, 5:4 1:0	**MODEy[1:0]**：端口x的模式位(y = 8…15) (Port x mode bits) 软件通过这些位配置相应的I/O端口，请参考表17端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度10MHz 10：输出模式，最大速度2MHz 11：输出模式，最大速度50MHz

图 7.12　端口配置寄存器高 32 位

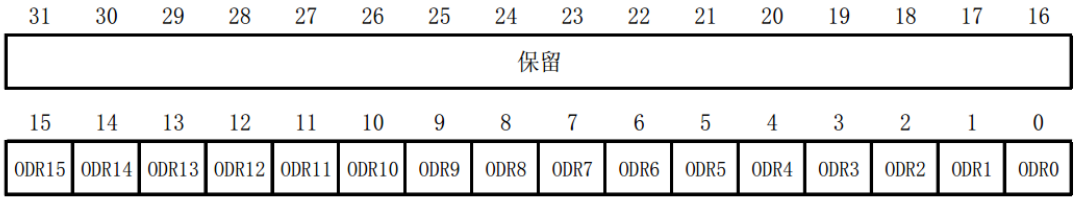
STM32 的每个 IO 口对应 4 位寄存器，低 2 位用于设置 IO 口输入还是输出以及输出的速度，高 2 位用于设置对应输入或输出模式下的更具体的工作模式。

- GPIO 端口数据寄存器（GPIOx_IDR/GPIOx_ODR）



位31:16	保留，始终读为0。
位15:0	**IDRy[15:0]**：端口输入数据(y = 0…15) (Port input data) 这些位为只读并只能以字(16位)的形式读出。读出的值为对应I/O口的状态。

图 7.13　端口数据输入寄存器



位31:16	保留，始终读为0。
位15:0	**ODRy[15:0]**：端口输出数据(y = 0…15) (Port output data) 这些位可读可写并只能以字(16位)的形式操作。 注：对GPIOx_BSRR(x = A…E)，可以分别地对各个ODR位进行独立的设置/清除。

图 7.13　端口数据输出寄存器

输入寄存器上对应位的值代表此时该位对应的 IO 引脚的电平状态，输出寄存器对应位的值决定了此时该位对应的 IO 引脚输出的电平状态。这里可以发现输出寄存器都是以字的形式读写的，不能单独对某一个位进行设置，当然，我们可以先读取此时寄存器的值然后对某一位或者某几位进行位的操作，来达到只设置某一位或某几位的目的。例如：我想设置 PA2 输出高电平，有两种方法可以实现（他们等效只不过写法不同）：

```
1.  GPIOA->ODR = GPIOA->ODR | GPIO_Pin_2;
2.  GPIOA->ODR |= GPIO_Pin_2;
```

或者设置 PA2 输出低电平，有两种方法可以实现（他们等效只不过写法不同）：

```
1.  GPIOA->ODR = GPIOA->ODR & ~GPIO_Pin_2;
2.  GPIOA->ODR &= ~GPIO_Pin_2;
```

上面的不管是设置输出高电平还是低电平，他们的过程都是一样的：

1. 读取 GPIOA 的 ODR 寄存器 16 位的值到临时变量
2. 修改临时变量
3. 将临时变量设置的值以字（16 位）为单位一次性写入到 GPIOA 的 ODR

这就涉及到读-改-写三个过程，假设此时设置 PA2 为 1，当执行到流程 1 或者 2 的时候，中断发生了，在中断函数中把 PA2 设置为 0，但是退出中断后，程序继续执行流程 3，也就是它又把 PA2 设置为 1，那么中断就不起作用了。这个问题就出在了对 ODR 设置的时间过长，因此 ST 为我们提供了端口位设置/清除寄存器来解决这个问题。（当然上面所说的这种情况很少发生，但不代表不存在）

- 端口位设置/清除寄存器（GPIO_BSRR/GPIO_BRR）

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**: 清除端口x的位y (y = 0…15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0 注：如果同时设置了BSy和BRy的对应位，BSy位起作用。
位15:0	**BSy**: 设置端口x的位y (y = 0…15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：设置对应的ODRy位为1

图 7.14　BSRR 寄存器

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

位31:16	保留。
位15:0	**BRy**: 清除端口x的位y (y = 0…15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响 1：清除对应的ODRy位为0

图 7.15　BRR 寄存器

有了位设置/清除寄存器，我们对 IO 引脚输出状态的设置就会在单写入周期完成，没有读-改的过程，因此操作就更加安全。一般对 BSSR 或者 BRR 进行设置的时候直接一条语句搞定：

GPIOA->BSSR = 0x02;

另外一个寄存器就是端口配置锁定寄存器(GPIOx_LCKR)，这里就不做过多讲解了，因为用的不多，这个寄存器的作用就是把当前 GPIO 的设置给锁定，使得在下次单片机复位或者重新上电之前我们都不能对 GPIO 再进行设置。

7.3 程序讲解

根据上面所讲的这些来开始我们点亮 LED 的学习。大家直接看代码：

```
17  void initLED(void)
18  {
19      GPIO_InitTypeDef GPIO_InitStructure; //定义GPIO初始化结构体
20
21      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //使能GPIO时钟
22
23      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;          //设置对应引脚
24      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;   //设置推挽输出
25      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz; //工作速度50MHz
26      GPIO_Init(GPIOB, &GPIO_InitStructure);             //设置生效
27      GPIO_ResetBits(GPIOB, GPIO_Pin_5);                 //默认低电平，LED不亮
28  }
```

图 7.16　LED 初始化函数

在控制外设之前我们都会设置外设的功能来满足我们的需求，这个过程就称之为初始化。我们要驱动 LED 当然要设置 GPIO 为输出，并指定输出引脚，以及输出状态。我们在前几章说过，要让外设工作就必须开启的时钟（这和 51 以及其他一些单片机不同，大家慢慢适应），因此我们第一件事儿就是开启外设 GPIOB 的时钟，接着对定义的功能结构体进行设置，大家要区分 GPIO_InitTypeDef 和 GPIO_TypeDef 结构体的区别，前者是设置 GPIO 属性的集合，比如使用哪个引脚，工作的模式，输出的速度，而后者则是 GPIO 各个寄存器的集合。

如果大家对 GPIO 设置属性的参数感兴趣的话可以跳转到定义处查看，这里我们以 GPIO 工作模式为例向大家介绍一下：

```
typedef enum
{ GPIO_Mode_AIN = 0x0,
  GPIO_Mode_IN_FLOATING = 0x04,
  GPIO_Mode_IPD = 0x28,
  GPIO_Mode_IPU = 0x48,
  GPIO_Mode_Out_OD = 0x14,
  GPIO_Mode_Out_PP = 0x10,
  GPIO_Mode_AF_OD = 0x1C,
  GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;
```

图 7.17　GPIO 模式参数

可以看到除了推挽输出还有其他我们之前介绍过的模式。

在配置完 GPIO 的工作属性之后（引脚，模式，速度）之后通过 GPIO_Init(GPIOB, &GPIO_InitStructure);初始化 GPIO，GPIO 配置生效。该函数其实就是对我们之前讲解的寄存器进行操作，只不过在操作之前还做了很多诸如参数是否正确的判断等。

另外还用到了如下两个函数来点亮或者熄灭 LED：

GPIO_SetBits(GPIOB, GPIO_Pin_5);

GPIO_ResetBits(GPIOB, GPIO_Pin_5);

我们打开定义，可以看到：

```
void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  GPIOx->BSRR = GPIO_Pin;
}
```

图 7.18　GPIO 置位函数

```
void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
{
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));

  GPIOx->BRR = GPIO_Pin;
}
```

图 7.19　GPIO 复位函数

与他们类似的函数还有 void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal)函数，只不过该内部是通过设置 ODR 寄存器实现的：

```
void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal)
{
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));

  GPIOx->ODR = PortVal;
}
```

图 7.20　GPIO_Write 函数

在平时开发时还是前两者用的较多。

读到这有人会说：我怎么知道库函数有哪些呢？没关系，我们打开标准库函数文件夹，会发现一个 CHM 文件：

图 7.21　库函数查看方法

打开这个文件，然后用到哪个外设就打开哪个外设的函数，然后点击那个数字就是这个函数在对应文件中的行号。

此时 LED 的驱动就写完了，接着我们在 main 函数中调用 LED 驱动函数并实现用户业务逻辑——LED 闪烁。

```
int main(void)
{
    u16 i,j;

    initLED(); //初始化LED
    while (1)
    {
        toggleLED();//翻转LED

        for(i=0;i<10000;++i) //阻塞单片机产生延时
        {
            for(j=0;j<100;++j);
        }
    }
}
```

图 7.22　业务逻辑实现

主函数中先对 LED 进行初始化，然后进入死循环，重复执行我们的业务代码。这里顺便说一下可以使用 for(;;)代替 while(1)，他们的效果是等同的。

因为现在我们还没有讲解精准的定时器延时，因此只是简单通过循环语句阻塞单片机重

复的做无意义的事情来延时，我们将空循环执行了 100 万次换来的是 300ms 左右的延时。此时我们编译链接程序然后烧录到开发板上并复位，可以看到 LED 在不停的闪烁。

本章 LED 就讲解到这里了，本章需要大家理解的知识很多，希望大家多动手慢慢消化。本章的知识点主要是，GPIO 的通用功能（只作为 GPIO 用），GPIO 复用功能，GPIO 重映射，GPIO 的几种工作模式(开漏，推挽，浮空，上拉，下拉等)以及库函数如果使用。

第八章 SysTick 定时器实现精准延时

8.1 项目要求

将第七章的 LED 实验例程中的粗略延时更改为使用 SysTick 定时器实现的精准定时。

注：本章只用到核心板，对应例程 2。

8.2 原理讲解

第七章我们使用两个 for 循环嵌套来实现的粗略延时，在时序要求不高的场合不会出现什么问题，但是依然有很多缺点，第一个就是不能精准延时，另外程序的优化程度也会影响延时的时间，除此之外，人工计算延时时间会花费很多时间和精力，因此有必要使用硬件定时器来实现精准延时。

延时是单片机开发当中一个非常重要的概念，差不多每一个项目当中都要用到延时，比如通过 LED 的闪烁来判断程序是否跑飞或卡死，这就要用到延时函数，另外通过 GPIO 来模拟时序来读取外部传感器比如开发板上的 DHT11，这时就必须严格按照传感器给出的时序进行操作，这就需要很精准的延时。

要想在单片机上实现很精准的延时就必须使用定时计数器，定时计数器顾名思义就是用来计数的单片机外设硬件，而它的每次计数都需要一段固定时间，那么我们就可以通过让它计一定数量的数来实现精准的延时。STM32 的定时器有很多，有高级定时器，基础定时器，RTC 定时器，SysTick 定时器，在这里我们使用 SysTick 定时器来实现延时，它是 ARM 内核自带的一个定时计数器。一般我们都用它来实现延时。

SysTick 是 24 位递减定时计数器，最多可计数 16777216 个数，当计数值从 1 变为 0 时，重装载寄存器会自动把自身的值赋给 SysTick 计数器，无需人为干预。SysTick 的时钟源来自于 AHB 时钟的 8 分频，如下图：

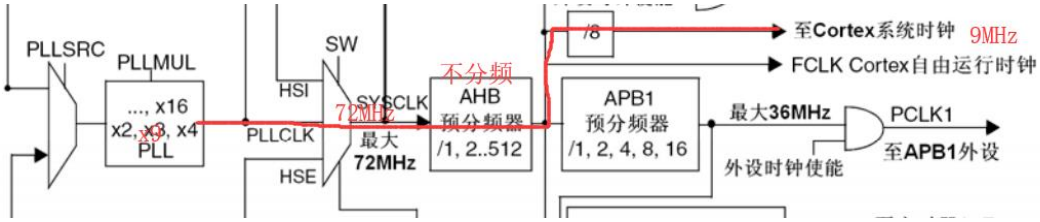


图 8.1 SysTick 时钟源

我们在前面讲解过 SystemInit()；将系统时钟 SYSCLK 设置 72MHz，而后在经过 AHB 的时候并没有对其分频（RCC_CFGR 寄存器设置 AHB 分频的对应设置位复位后的默认值为 0，即不分频），因此此时 AHB 的时钟也是 72MHz，在经过对 AHB 时钟 8 分频后最终提供给 SysTick 定时器的时钟频率为 9MHz，也就是说 SysTick 一次计数会消耗 1/9us，那么记完 16777216 个数需要大约 1.8 秒，对于我们平时的延时间隔足够用了，下面看一下 SysTick 的寄存器：

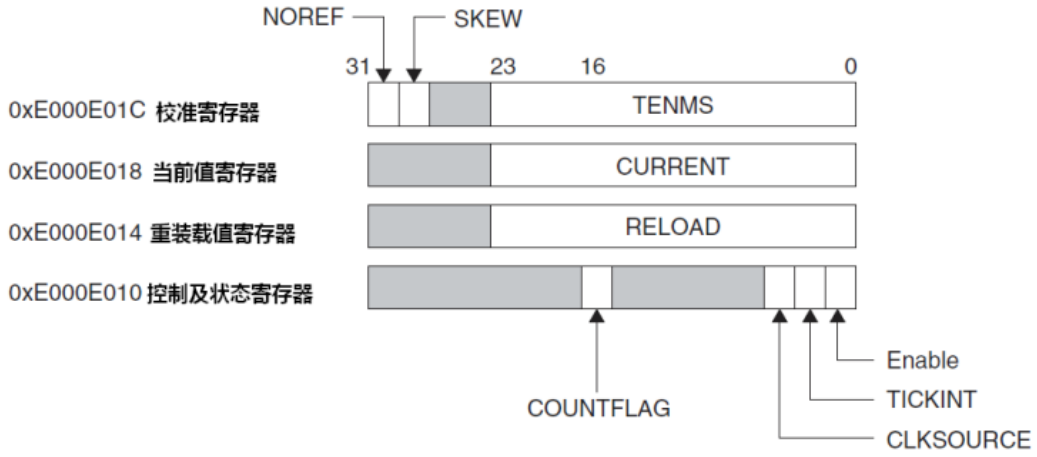


图 8.2 SysTick 寄存器

- 校准寄存器用于在不同 CM3 芯片上产生恒定的 SysTick 中断频率，ST 设置的校准值为 9000，当 SysTick 的频率为 9MHz 的时候可以产生 1ms 的中断频率。但是我们一般不会用到这个寄存器，所以大家知道就可以了。

位段	名称	类型	复位值	描述
31	NOREF	R	-	1=没有外部参考时钟（STCLK 不可用） 0=外部参考时钟可用
30	SKEW	R	-	1=校准值不是准确的 10ms 0=校准值是准确的 10ms
23:0	TENMS	R/W	0	10ms 的时间内倒计数的格数。芯片设计者应该通过 Cortex-M3 的输入信号提供该数值。若该值读回零，则表示无法使用校准功能

图 8.3 SysTick 校准寄存器位域

- 当前值寄存器可以实时的更新此时计数器记到了那里，我们在用时钟摘取法实现延时的时候会用得到。

位段	名称	类型	复位值	描述
23:0	CURRENT	R/Wc	0	读取时返回当前倒计数的值，写它则使之清零，同时还会清除在 SysTick 控制及状态寄存器中的 COUNTFLAG 标志

图 8.4 SysTick 当前值寄存器位域

- 重装载寄存器里面放置的是 24 位的重装载值，当计数器记到 0 后自动装载到计数器。

位段	名称	类型	复位值	描述
23:0	RELOAD	R/W	0	当倒数至零时，将被重装载的值

图 8.5 SysTick 重装载寄存器位域

- 控制及状态寄存器：用于设置定时器时钟源、中断使能，定时器使能（开关定时器，可用于对输入电平进行计时）以及重装载标志位。

位段	名称	类型	复位值	描述
16	COUNTFLAG	R	0	如果在上次读取本寄存器后，SysTick 已经数到了 0，则该位为 1。如果读取该位，该位将自动清零
2	CLKSOURCE	R/W	0	0=外部时钟源(STCLK) 1=内核时钟(FCLK)
1	TICKINT	R/W	0	1=SysTick 倒数到 0 时产生 SysTick 异常请求 0=数到 0 时无动作
0	ENABLE	R/W	0	SysTick 定时器的使能位

图 8.6 SysTick 控制及状态寄存器位域

这里说一下，想要清除重装载标志位（COUNTFLAG）可以读控制及状态寄存器或者对当前值寄存器写操作。

8.3 程序讲解

在实现延时之前我们要先初始化 SysTick 定时器：

```
/**
 * 功能：初始化Systick定时器
 * 参数：None
 * 返回值：None
 */
void initSysTick(void)
{
    SysTick_CLKSourceConfig(SysTick_CLKSource_HCLK_Div8); //设置时钟源8分频
    SysTick->CTRL |= SysTick_CTRL_TICKINT_Msk;            //使能中断
    SysTick->CTRL |= SysTick_CTRL_ENABLE_Msk;             //开定时器
    SysTick->LOAD = 9;                                    //随意设置一个重装载值
}
```

图 8.7 SysTick 控制及状态寄存器位域

这里设置了时钟源 8 分频，开启定时器更新中断（计到 0 产生中断），开启定时器并给定时器赋了一个计数值，让其有“数”可计。

下面我们来讲一下 us 和 ms 级别的延时函数：

```
/**
 * 功能：us级别延时
 * 参数：xus：要延时的时间
 * 返回值：None
 */
void Delay_us(u32 xus)
{
    SysTick->LOAD = 9 * xus; //计9次为1us，xus则重装载值要*9
    SysTick->VAL = 0;        //计数器归零
    while (!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk)); //等待计数完成
}
```

图 8.8 us 延时函数

72MHz 经过 8 分频后得到 9MHz 频率，也就是 1/9us，那么要达到 1us 就需要计数 9 次，因此计数 xus 需要 x*9。我们设置完成重装载值之后，对计数器清零，然后一直等待这些 us 计数完成。

```
/**
 * 功能：ms级别延时
 * 参数：xms：要延时的时间
 * 返回值：None
 */
void Delay_ms(u32 xms)
{
    SysTick->LOAD = 9000; //计9次为1us，1000次为1ms
    SysTick->VAL = 0;     //计数器归零
    while (xms--)
    {
        while (!(SysTick->CTRL & SysTick_CTRL_COUNTFLAG_Msk)); //等待单次计数完成
    }
}
```

图 8.9 ms 延时函数

SysTick 记完一个周期（16777216 个计数值）需要 1.8s 左右，那么如果我们还像 us 延时函数一样操作的话（为重装载寄存器赋值）会带来一个问题：ms 延时函数的延时时间不能超过 1.8s，否则就超出了定时器的最大计数值，因此我们采用的策略是：我只设置 1ms 延时，然后通过 while 递减判断来进行任意时间延时，如上图所看到那样。

接下来更改 main 函数延时部分：

```
    while (1)
    {
        toggleLED();//翻转LED
        Delay_ms(500);
    }
```

图 8.10　main 函数部分

编译链接-烧录-复位，看到了 LED 每隔 500ms 翻转一次。

当然 Systick 的玩法绝不只是实现延时而已，我们可以通过 Systick 来计算外部信号的频率、占空比等（软件输入捕获）以及生成 PWM（软件输出比较），但是我们的延时函数会和这些功能发生冲突，因为定时计数器的重装载值一直在改变，从而导致定时器中断周期改变，因此导致捕获的数据不准，PWM 输出的频率不同，当然，解决办法是可以在初始化

Systick 的时候指定重装载值必须为 9，也就是周期为 1us，而后延时函数可以这样写：

```
25  void Delay_us(u32 us)
26  {
27      SysTick->VAL = 0;
28      while(us--)
29      {
30         while(!(SysTick->CTRL&SysTick_CTRL_COUNTFLAG_Msk));
31
32      }
33  }
34
35  void Delay_ms(u32 ms)
36  {
37      while(ms--)
38      {
39        Delay_us(1000);
40      }
41  }
42
```

图 8.11 延时函数的另外一种实现方法

这就保证的了中断周期固定为 1us 了，而且 us 和 ms 的延时都是在 1us 的基础上实现的。但是作者不推荐这么做，因为中断频率太快了，程序执行的很大一部分时间会浪费在中断函数跳转上，即每隔 1us 进行一次堆栈操作以及函数的跳转，当然你也可以增加中断的间隔，但是这将会导致延时精度的降低。

到这里本章讲解完成，希望大家慢慢消化延时函数的原理，当然也不要太较真其他延时的实现方法，我们掌握一种即可。

第九章 GPIO 输入实验_按键采集

9.1 项目要求

通过两个按键来改变 LED 的闪烁频率，单独按下 UP 键，LED 以 50ms 的频率闪烁，单独按下 DOWN 键 LED 以 2s 的频率闪烁。同时按下 UP 和 DOWN 键 LED 以亮 50ms 灭 500ms 的频率闪烁 10 次。另外在这章对我们之前的 LED 程序进行一次升级，使其实用性更强。

注：本章只用到核心板，对应例程 3。

9.2 原理讲解

按键也叫微动开关，当你按下按键后簧片会和两端引脚的触点接触使其导通，松开则断开连接。我们开发板上的按键原理图如下：

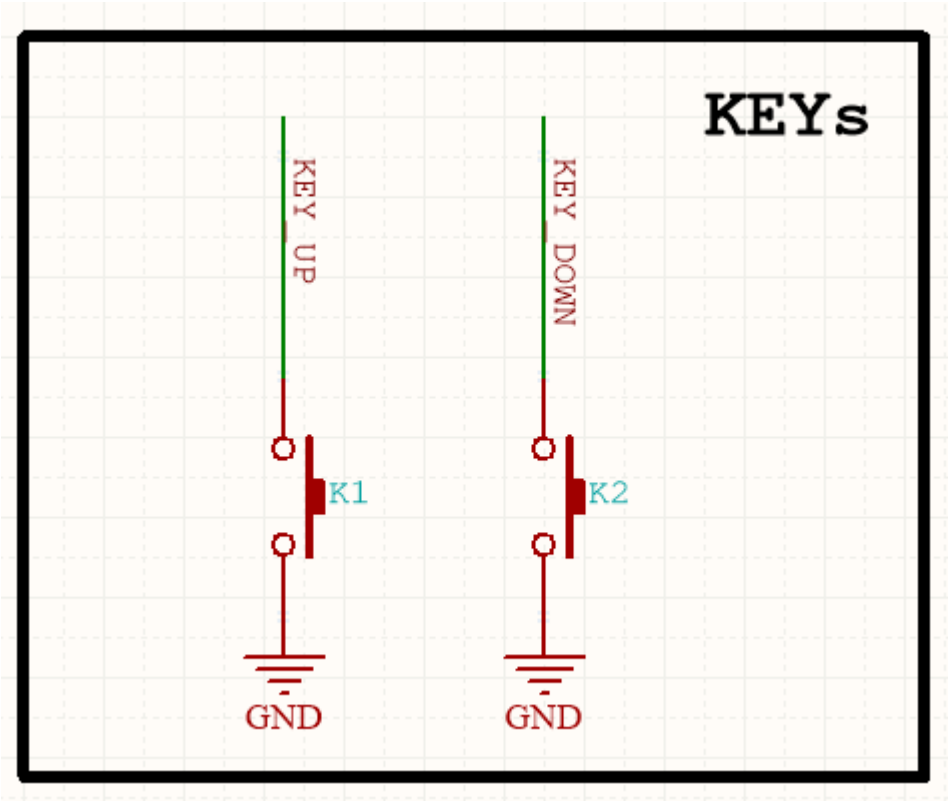


图 9.1 按键原理图

本例程中我们将 GPIO 设置为上拉输入，当没有按下按键时，GPIO 检测到的是高电平，当按键按下后，由于 GPIO 被拉到地，此时 GPIO 检测到的低电平，我们根据这个来判断按键是否按下。

由于我们使用的是机械按键，所以在按键按下以及松开后会有机械抖动，这种抖动一般会持续几 ms，抖动波形示意如下：

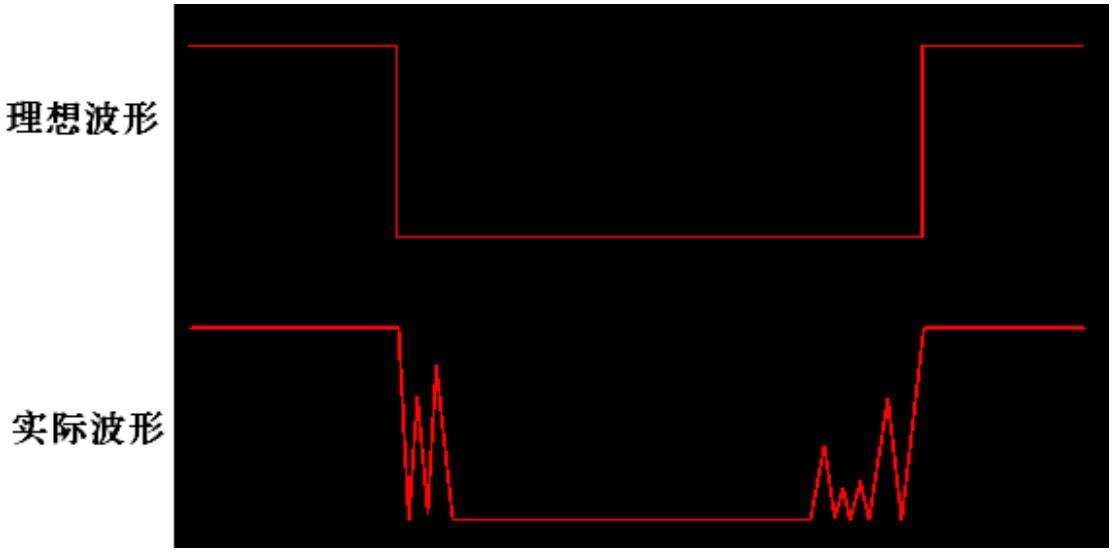


图 9.2 按键抖动波形图

这种干扰对我们是不利的，假如按键采集恰好在抖动期间，那么我们采集的数据将是不准确的，因此就必须使用一种手段来屏蔽掉这种干扰，第一个方法可以使用 RC 滤波电路，前提是得到按键抖动的频率并根据频率计算 RC 的值，但是这样做很麻烦，同时也会增加硬件成本，我们能用软件解决的尽量不要用硬件解决，那么第二种方法就是使用软件去抖，即在检测按键的时候如果检测到 IO 口是低电平则延时一段时间来跳过抖动，一般选取的延时时间为 10ms，在延时之后我们再判断此时 GPIO 的状态，如果是低电平，那么就能肯定按键是按下的状态了。

9.3 程序讲解

- 按键初始化函数

```
12  /**
13   * 功能：初始化按键
14   * 参数：None
15   * 返回值：None
16   */
17  void initKey(void)
18  {
19      GPIO_InitTypeDef GPIO_InitStructure; //定义GPIO初始化结构体
20
21      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA  ENABLE); //使能GPIO时钟
22                                  #define KEY_UP_PIN GPIO_Pin_0
23      GPIO_InitStructure.GPIO_Pin = KEY_UP_PIN|KEY_DOWN_PIN;          //设置按键对应引脚 UP->PA0, DOWN->PA1
24      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;  //设置上拉输入
25      GPIO_Init(GPIOA, &GPIO_InitStructure);              //设置生效
26  }
```

图 9.3 按键初始化

这一段代码的目的是将按键连接的两个引脚初始化为上拉输入。当按键按下时上拉电阻被拉低，引脚采集到的数据为 0，释放按键引脚又拉回到 VCC，此时采集到的数据时 1.

另外 KEY_UP_PIN 和 KEY_DOWN_PIN 是对于按键引脚的宏定义，增加代码的可读性，如图中提示的内容（在使用 VS CODE 的时候，我们可以鼠标停靠在变量或者宏定义上面，这是会自动显示其定义的内容）。

- 按键采集函数

```
/**
 * 功能：获取按键值，支持同时按下，按键释放后生效
 * 参数：mode:用于指定按键触发方式，参数可选：KEY_PRESS,KEY_RELEASE
 * 返回值：返回按键值，当同时按下时，返回值是 KEY_UP|KEY_DOWN 即0x03
 */
u8 getKeyValue(u8 mode)
{
    u8 keyval = 0;

    if(mode == KEY_PRESS) //按下按键即可生效
    {
        /**************检测UP键函数段*******************/
        if(GPIO_ReadInputDataBit(GPIOA,KEY_UP_PIN)==0) //第一次检测时按下
        {
            Delay_ms(10); //延时去抖
            if(GPIO_ReadInputDataBit(GPIOA,KEY_UP_PIN)==0) //第二次检测还为零则可以确定按键按下
            {
                keyval |= KEY_UP;
            }else
            {
                keyval &= ~KEY_UP;
            }
        }else
        {
            keyval &= ~KEY_UP;  //没有按下 对应位清零
        }

        /**************检测DOWN键函数段*******************/
        if(GPIO_ReadInputDataBit(GPIOA,KEY_DOWN_PIN)==0) //第一次检测时按下
        {
            Delay_ms(10); //延时去抖
            if(GPIO_ReadInputDataBit(GPIOA,KEY_DOWN_PIN)==0) //第二次检测还为零则可以确定按键按下
            {
                keyval |= KEY_DOWN;
            }else
            {
                keyval &= ~KEY_DOWN;
            }
        }else
        {
            keyval &= ~KEY_DOWN;  //没有按下 对应位清零
        }
    }else if(mode == KEY_RELEASE) //松开按键生效
    {
        /**************检测UP键函数段*******************/
        if(GPIO_ReadInputDataBit(GPIOA,KEY_UP_PIN)==0) //第一次检测时按下
        {
            Delay_ms(10); //延时去抖
            if(GPIO_ReadInputDataBit(GPIOA,KEY_UP_PIN)==0) //第二次检测还为零则可以确定按键按下
            {
                keyval |= KEY_UP;
            }else
            {
```

```
81              keyval &= ~KEY_UP;
82          }
83      }else
84      {
85          keyval &= ~KEY_UP;  //没有按下 对应位清零
86      }
87
88      /*************检测DOWN键函数段******************/
89      if(GPIO_ReadInputDataBit(GPIOA,KEY_DOWN_PIN)==0) //第一次检测时按下
90      {
91          Delay_ms(10); //延时去抖
92          if(GPIO_ReadInputDataBit(GPIOA,KEY_DOWN_PIN)==0) //第二次检测还为零则可以确定按键按下
93          {
94              keyval |= KEY_DOWN;
95          }else
96          {
97              keyval &= ~KEY_DOWN;
98          }
99      }else
100     {
101         keyval &= ~KEY_DOWN;  //没有按下 对应位清零
102     }
103
104     while(GPIO_ReadInputDataBit(GPIOA,KEY_UP_PIN)==0 || GPIO_ReadInputDataBit(GPIOA,KEY_DOWN_PIN)==0); //等待按键释放
105 }
106
107 return keyval;
108 }
```

图 9.4 按键采集

先说说该函数的参数，这个参数指定按键触发的方式，按键触发方式有两种，第一种是按下触发，第二种是释放触发。**按下触发方式**就是说只要按键按下就可以立即返回此时采集的键值，而**释放触发方式**指的是当按键被按下后，要一直等到按键松开才返回按键值。这两种触发模式可以应用在不同的场合，这里要引出一个概念——**用户代码循环周期**。其实就是 while 死循环中的代码执行一次所需要的时间。那么问题出现了：如果该周期较短（比如 5ms 执行一次 while 内的代码），用户通过按下触发的方式来获取键值，那么就会造成在按键按下的这段时间内按键采集函数被调用多次（因为人为按下的时间会持续几百 ms），该键值对应的的处理函数也被调用多次，如果此时按键用于设置参数，你会发现你根本设置不准，因为它太“灵敏”。因此在这种情况下，我们可以加入一些机制，例如，当检测到按键按下后，并不立即返回结果，而是要等到按键彻底释放之后（IO 口检测到 1）才返回键值，这种方式就是**释放触发方式**。**按下触发方式**主要应用在长按可以连续递增或者递减的场合，比如长按按键，音量递增或递减，只要我们在两次按键采集之间加入合理的延时即可。

该函数内需要介绍的就是 uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)函数，它用于返回某个引脚的电平状态，另外和它很像的是：uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx)函数，该函数返回的并不是某个引脚的电平状态，而是某组端口的 16 个 IO 引脚的的所有状态，即它返回的是输入数据寄存器 IDR 中的值。

另外我们对之前的 LED 例程进行的升级，添加 LED 闪烁函数：

- LED 闪烁函数

```
/**
 * 功能：LED闪烁
 * 参数：times：闪烁次数 open_ms:高电平持续时间 close_ms：低电平持续时间
 * 返回值：None
 */
void blinkLED(u8 times,u32 open_ms,u32 close_ms)
{
    u8 t; //用于翻转计数
    GPIO_ResetBits(GPIOB, GPIO_Pin_5); //设置引脚为低电平
    for(t=0;t<times;++t)
    {
        toggleLED();       //翻转为高电平
        Delay_ms(open_ms);
        toggleLED();       //翻转为低电平
        Delay_ms(close_ms);
    }
}
```

图 9.5　LED 闪烁函数

在 LED 多次以某种频率闪烁的应用场景中，一个函数就可以搞定，无需用户自己实现闪烁次数和间隔。

- 主函数

```
int main(void)
{
    u16 time = 500; //LED闪烁时间 开机默认500ms
    u8 keyvalue;

    initSysTick(); //初始化Systick
    initLED();     //初始化LED
    initKey();     //初始化按键
    while (1)
    {
        keyvalue= getKeyValue(KEY_PRESS); //获取键值
        switch(keyvalue)
        {
            case KEY_UP:    time = 50;          break;//50ms频率闪烁
            case KEY_DOWN:  time = 2000;        break;//2s频率闪烁
            case KEY_ALL: blinkLED(10,50,500); break;//同时按下时，亮50ms灭500ms闪烁10次

            default:                            break;//没按下不做任何处理
        }
        blinkLED(1,time,time);
    }
}
```

图 9.6　主函数

在经过初始化之后，我们先获取键值，然后在是 switch-case 语句中执行对应键值的操作。最后将更改的参数传给 LED 翻闪烁函数。

编译-烧录-复位，观察结果，刚开机时没有按键按下，LED 会以 500ms 的默认频率闪烁，当按下 UP 键，LED 会以 50ms 的频率飞快闪烁，当按下 DOWN 键，LED 会以 2S 的周期缓慢闪烁，当同时按下两个按键时，LED 会以亮 50ms 灭 500ms 的频率闪烁 10 次，而后退出到正常的翻转状态。

现在我猜你那边又有新的疑惑了：为什么在 LED 闪烁很快的情况下（UP 模式）按下 DOWN 键会很灵敏，可以立即进入 DOWN 模式，而在 DOWN 模式的缓慢闪烁的情况下，按下 UP 键会很迟钝甚至失灵呢？这是因为用户代码循环周期太长（4S），我们在按下按键的时候还没有执行到按键采集函数，自然就会感到迟钝了。这也是我们平时会出现的错误，解决办法有几种，第一个就是使用外部中断（后续会讲），第二个就是在定时器溢出中断中定期执行按键

采集函数（后续会讲），第三个就是使用操作系统，因为系统可以自动进行任务调度。第四个办法是减小代码循环周期来解决这种问题，比如此时我们将 DOWN 模式的 2000 改为 100，这种迟钝的现象就会好很多。第五个办法是我们可以像方法四那样让用户代码循环周期变得很短，比如 1ms，然后根据不同的代码对延时的要求，通过变量的递增来实现更长时间的延时，这里我使用伪代码来举个例子：在用户循环中，传感器需要 1S 钟采集一次数据，同时也要满足按键采集不能出现失灵：

```
void main()
{
    u32 i = 0;
    u8 keyvalue;
    u8 sensordata;
    initALL();
    while(1)
    {
        if(++i==1000)
        {
            i = 0;
            sensordata = getSensorData();
        }
        keyvalue = getKeyValue();
        Delay_ms(1);
    }
}
```

图 9.7 伪代码展示

没想到如此简单的按键采集会有这么多坑吧，在今后大家开发的时候如果遇到了类似的问题，可以尝试用我们提供的方案解决。另外建议大家多修改一下主函数中的参数，观察一下不同参数对效果有什么影响。

第十章 中断

10.1 什么是中断

中断是单片机当中非常重要的一种机制，它可以提高单片机处理异步事件的实时响应速度。中断，顾名思义就是打断正在执行的任务，去执行优先级更高的任务。例如：你此时正在家中看电视，突然接到快递员电话，让你马上去楼下取快递，此时取快递就是中断事件，它的优先级更高，因此就必须打断看电视的任务，当取到快递之后（中断任务完成），你可以继续看电视（用户任务继续）。

这里先介绍一个概念——前后台任务：

在设计程序的时候 while 死循环中的代码称为**后台任务**，中断则称为**前台任务**。在大部分时间后台任务得以执行，只有在需要紧急处理的中断发生时才会切换到前台任务即中断服务函数，前台任务执行完之后会自动切换到后台任务。另外也可以把后台任务叫做任务级，把前台任务叫做中断级。 我们看一幅图加深理解：

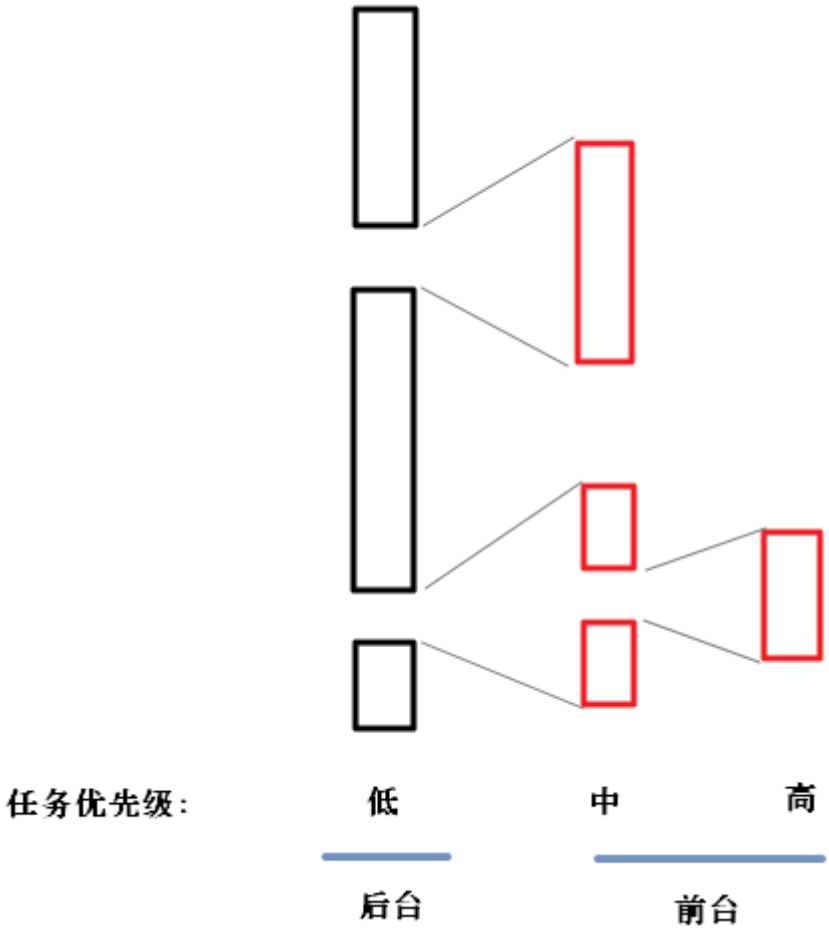


图 10.1 前后台模型

图中黑色部分就是前面讲的后台任务，红色部分就是前台任务。另外可以看到，红色部分又进行了一次嵌套，也就是说高优先级的中断同样也会打断低优先级的中断。

10.2 STM32F103 中断讲解

由于 STM32F103 是基于 ARM Cortex M3 内核设计的，因此我们先了解一下 ARM Cortex M3 的中断，M3 内核的中断分为两大部分，第一部分是系统中断，一共有 16 个，第二部分是留给芯片厂家的 240 的外设中断（这两部分加起来一共有 256 个中断）。但是芯片厂家可能用不上 240 个外设中断，因此会对其进行裁剪，我们的 STM32F103 只是用了 60 个外设中断。下面我们看一下 STM32F103 这 76 个中断（16 个系统中断有些是预留的，有意义的系统中断为 10 个。另外**中断优先级数越小优先级别越高**）：

位 置	优先级	优先级类型	名称	说明	地址
-	-	-	保留	0x0000_0000	
-3	固定	Reset	复位	0x0000_0004	
-2	固定	NMI	不可屏蔽中断 RCC 时钟安全系统(CSS)联接到 NMI 向量	0x0000_0008	
-1	固定	硬件失效(HardFault)	所有类型的失效	0x0000_000C	
0	可设置	存储管理(MemManage)	存储器管理	0x0000_0010	
1	可设置	总线错误(BusFault)	预取指失败，存储器访问失败	0x0000_0014	
2	可设置	错误应用(UsageFault)	未定义的指令或非法状态	0x0000_0018	
-	-	-	保留	0x0000_001C ~0x0000_002B	
3	可设置	SVCall	通过 SWI 指令的系统服务调用	0x0000_002C	
4	可设置	调试监控(DebugMonitor)	调试监控器	0x0000_0030	
-	-	-	保留	0x0000_0034	
5	可设置	PendSV	可挂起的系统服务	0x0000_0038	
6	可设置	SysTick	系统嘀嗒定时器	0x0000_003C	
0	7	可设置	WWDG	窗口定时器中断	0x0000_0040

1	8	可设置	PVD	连到 EXTI 的电源电压检测(PVD)中断	0x0000_0044
2	9	可设置	TAMPER	侵入检测中断	0x0000_0048
3	10	可设置	RTC	实时时钟(RTC)全局中断	0x0000_004C
4	11	可设置	FLASH	闪存全局中断	0x0000_0050

5	12	可设置	RCC	复位和时钟控制(RCC)中断	0x0000_0054
6	13	可设置	EXTI0	EXTI 线 0 中断	0x0000_0058
7	14	可设置	EXTI1	EXTI 线 1 中断	0x0000_005C
8	15	可设置	EXTI2	EXTI 线 2 中断	0x0000_0060
9	16	可设置	EXTI3	EXTI 线 3 中断	0x0000_0064
10	17	可设置	EXTI4	EXTI 线 4 中断	0x0000_0068
11	18	可设置	DMA1 通道 1	DMA1 通道 1 全局中断	0x0000_006C
12	19	可设置	DMA1 通道 2	DMA1 通道 2 全局中断	0x0000_0070
13	20	可设置	DMA1 通道 3	DMA1 通道 3 全局中断	0x0000_0074
14	21	可设置	DMA1 通道 4	DMA1 通道 4 全局中断	0x0000_0078
15	22	可设置	DMA1 通道 5	DMA1 通道 5 全局中断	0x0000_007C
16	23	可设置	DMA1 通道 6	DMA1 通道 6 全局中断	0x0000_0080
17	24	可设置	DMA1 通道 7	DMA1 通道 7 全局中断	0x0000_0084
18	25	可设置	ADC1_2	ADC1 和 ADC2 的全局中断	0x0000_0088
19	26	可设置	USB_HP_CAN_TX	USB 高优先级或 CAN 发送中断	0x0000_008C
20	27	可设置	USB_LP_CAN_RX0	USB 低优先级或 CAN 接收 0 中断	0x0000_0090
21	28	可设置	CAN_RX1	CAN 接收 1 中断	0x0000_0094
22	29	可设置	CAN_SCE	CAN SCE 中断	0x0000_0098
23	30	可设置	EXTI9_5	EXTI 线[9:5]中断	0x0000_009C
24	31	可设置	TIM1_BRK	TIM1 刹车中断	0x0000_00A0
25	32	可设置	TIM1_UP	TIM1 更新中断	0x0000_00A4
26	33	可设置	TIM1_TRG_COM	TIM1 触发和通信中断	0x0000_00A8
27	34	可设置	TIM1_CC	TIM1 捕获比较中断	0x0000_00AC
28	35	可设置	TIM2	TIM2 全局中断	0x0000_00B0
29	36	可设置	TIM3	TIM3 全局中断	0x0000_00B4
30	37	可设置	TIM4	TIM4 全局中断	0x0000_00B8
31	38	可设置	I2C1_EV	I2C1 事件中断	0x0000_00BC
32	39	可设置	I2C1_ER	I2C1 错误中断	0x0000_00C0
33	40	可设置	I2C2_EV	I2C2 事件中断	0x0000_00C4
34	41	可设置	I2C2_ER	I2C2 错误中断	0x0000_00C8
35	42	可设置	SPI1	SPI1 全局中断	0x0000_00CC

36	43	可设置	SPI2	SPI2 全局中断	0x0000_00D0
37	44	可设置	USART1	USART1 全局中断	0x0000_00D4
38	45	可设置	USART2	USART2 全局中断	0x0000_00D8
39	46	可设置	USART3	USART3 全局中断	0x0000_00DC
40	47	可设置	EXTI15_10	EXTI 线[15:10]中断	0x0000_00E0
41	48	可设置	RTCAlarm	连到 EXTI 的 RTC 闹钟中断	0x0000_00E4
42	49	可设置	USB 唤醒	连到 EXTI 的从 USB 待机唤醒中断	0x0000_00E8
43	50	可设置	TIM8_BRK	TIM8 刹车中断	0x0000_00EC
44	51	可设置	TIM8_UP	TIM8 更新中断	0x0000_00F0
45	52	可设置	TIM8_TRG_COM	TIM8 触发和通信中断	0x0000_00F4
46	53	可设置	TIM8_CC	TIM8 捕获比较中断	0x0000_00F8
47	54	可设置	ADC3	ADC3 全局中断	0x0000_00FC
48	55	可设置	FSMC	FSMC 全局中断	0x0000_0100
49	56	可设置	SDIO	SDIO 全局中断	0x0000_0104
50	57	可设置	TIM5	TIM5 全局中断	0x0000_0108
51	58	可设置	SPI3	SPI3 全局中断	0x0000_010C
52	59	可设置	UART4	UART4 全局中断	0x0000_0110
53	60	可设置	UART5	UART5 全局中断	0x0000_0114
54	61	可设置	TIM6	TIM6 全局中断	0x0000_0118
55	62	可设置	TIM7	TIM7 全局中断	0x0000_011C
56	63	可设置	DMA2 通道 1	DMA2 通道 1 全局中断	0x0000_0120
57	64	可设置	DMA2 通道 2	DMA2 通道 2 全局中断	0x0000_0124
58	65	可设置	DMA2 通道 3	DMA2 通道 3 全局中断	0x0000_0128
59	66	可设置	DMA2 通道 4_5	DMA2 通道 4 和 DMA2 通道 5 全局中断	0x0000_012C

表 10.1　STM32 中断系统

其中蓝色部分是系统的 16 个中断，其余 60 个为外设中断。另外除了三个中断优先级固定外，剩余的中断优先级都是可以通过程序设置的，STM32 支持 16 个可编程优先级。

这里说一下和中断设置有关的名词：

中断使能/失能：中断使能的意思是让中断有效或者说是开中断，反之，失能则是关闭中断。

中断挂起/解挂：当一个中断发生时，此时单片机正在执行和该中断同优先级或者更高优先级的另外一个中断时，该中断将被挂起，直到另一个中断执行完才可解挂并执行，其实就是根据中断优先级排队。

中断嵌套：正如前面图片表述那样，一个高优先级的中断打断了低优先级的中断的过程就是中断嵌套，这样做使得不同优先级的中断能够按照优先级的大小依次执行，可以理解为 VIP 插队。

接下来我们看一下 STM32F103 中断的相关寄存器（这些寄存器是用于设置 STM32F103 的 60 个外设中断的）：

```
132  typedef struct
133  {
134    __IO uint32_t ISER[8];                 /*!< Offset: 0x000  Interrupt Set Enable Register           */
135         uint32_t RESERVED0[24];
136    __IO uint32_t ICER[8];                 /*!< Offset: 0x080  Interrupt Clear Enable Register         */
137         uint32_t RSERVED1[24];
138    __IO uint32_t ISPR[8];                 /*!< Offset: 0x100  Interrupt Set Pending Register          */
139         uint32_t RESERVED2[24];
140    __IO uint32_t ICPR[8];                 /*!< Offset: 0x180  Interrupt Clear Pending Register        */
141         uint32_t RESERVED3[24];
142    __IO uint32_t IABR[8];                 /*!< Offset: 0x200  Interrupt Active bit Register           */
143         uint32_t RESERVED4[56];
144    __IO uint8_t  IP[240];                 /*!< Offset: 0x300  Interrupt Priority Register (8Bit wide) */
145         uint32_t RESERVED5[644];
146    __O  uint32_t STIR;                    /*!< Offset: 0xE00  Software Trigger Interrupt Register     */
147  } NVIC_Type;
```

图 10.2 NVIC 寄存器

该段代码位于 core_cm3.h 文件中，它定义的是就是 STM32 的中断相关寄存器。我们依次讲解一下。

- 中断使/失能寄存器（ISER/ICER）

ISER 用于使能某个中断，寄存器中的一个位对应着一个中断，写 1 就使能，写 0 被忽略。由于 CM3 内核最多支持 240 个外设中断，因此 ISER 是由 8 个 32 位的寄存器组成（只不过剩余的最高 16 位没有用到），而 STM32F103 只有 60 个中断，因此只用到了 ISER[0]和 ISER[1]的低 28 位。

同理，ICER 用于设置 STM32 的中断失能，写 1 失能，写 0 忽略。

- 中断挂起/解挂寄存器(ISPR/ICPR)

这两组寄存器用于设置中断挂起和解挂，操作和中断使/失能寄存器一致。

- 活动中断标志寄存器

字面理解就是用于记录此时正在执行中断的寄存器，如果一个中断被执行，那么该中断对应位就会被置 1，即使该中断被其他更高优先级的中断打断，只要该中断没有执行完就一直保持 1 的状态，直到中断服务函数执行完才会被自动清零。

- 中断优先级寄存器(IP)

中断优先级寄存器用于设置每个中断的优先级，CM3 内核最大支持 240 个外设中断并为每个中断分配了 8 位可编程优先级，但是 STM32F103 只用到了 60 个中断优先级寄存器，而且每个中断只用到了高四位来设置中断优先级。这高四位又被分组为两部分优先级：抢占优先级和子优先级（也可以叫响应优先级或者亚优先级）。一个中断是否需能够抢占其他中断由三个因素决定：抢占优先级，子优先级，中断编号。

先说抢占优先级，如果中断 A 的抢占优先级比中断 B 的抢占优先级高，那么不管这两个中断的子优先级和中断编号是什么，A 都会抢占 B，或者说发生 B 的时候 A 在执行，那么 B 会被挂起。

当 A、B 两个中断的抢占优先级相同的情况下，且 A 的子优先级比 B 高，那么当他们同时发生或者同时解挂的时候会优先执行 A,如果 A 中断发生的时候 B 中断正在执行，那么 A

也不会抢占 B，而是要挂起，等待正在执行的 B 中断执行完成才执行 A 中断。

当两个中断的抢占优先级和子优先级都相同的情况下且两个中断一起发生时，则根据中断编号进行响应（号小的先响应）。

- 软件触发中断寄存器(STIR)

用户可以通过写该寄存器来手动的触发对应中断，比如写入 0x08，则触发中断编号 8 的中断。

下面说一下中断分组，STM32 的中断分组由 AIRCR 寄存器的 10-8 位控制：

位段	名称	类型	复位值	描述
31:16	VECTKEY	RW	-	访问钥匙：任何对该寄存器的写操作，都必须同时把 0x05FA 写入此段，否则写操作被忽略。若读取此半字，则 0xFA05
15	ENDIANESS	R	-	指示端设置。1＝大端(BE8)，0＝小端。此值是在复位时确定的，不能更改。
10:8	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位
1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息。通常只在调试时用，或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位 CM3 处理器内核（调试逻辑除外），但是此复位不影响芯片上在内核以外的电路

图 10.3　AIRCR 寄存器

该中断分组设置是全局的，例如此时将中断优先级分组设置为第三分组，那么对于所有的外设中断来说，都将使用这种分组方式。STM32F103 的中断共分为 5 组：

中断分组	Bit7	Bit6	Bit5	Bit4	Bit3	Bit2	Bit1	Bit0
0 组								
1 组								
2 组								
3 组								
4 组								

抢占优先级　子优先级　未使用

图 10.4 优先级分组示意图

就拿 2 组为例，此时抢占优先级的取值范围是 0~3，子优先级取值范围也是 0~3，分组的好处是可以扩展优先级的功能。我们在配置中断的时候，一般要做两个工作：

1. 在一开机就设置中断分组，且在下次重启之前最好不要再进行分组设置。我们看一下设置中断分组的函数（位于 misc.c）：

```
80  /**
81    * @brief  Configures the priority grouping: pre-emption priority and subpriority.
82    * @param  NVIC_PriorityGroup: specifies the priority grouping bits length.
83    *   This parameter can be one of the following values:
84    *     @arg NVIC_PriorityGroup_0: 0 bits for pre-emption priority
85    *                                4 bits for subpriority
86    *     @arg NVIC_PriorityGroup_1: 1 bits for pre-emption priority
87    *                                3 bits for subpriority
88    *     @arg NVIC_PriorityGroup_2: 2 bits for pre-emption priority
89    *                                2 bits for subpriority
90    *     @arg NVIC_PriorityGroup_3: 3 bits for pre-emption priority
91    *                                1 bits for subpriority
92    *     @arg NVIC_PriorityGroup_4: 4 bits for pre-emption priority
93    *                                0 bits for subpriority
94    * @retval None
95    */
96  void NVIC_PriorityGroupConfig(uint32_t NVIC_PriorityGroup)
97  {
98    /* Check the parameters */
99    assert_param(IS_NVIC_PRIORITY_GROUP(NVIC_PriorityGroup));
100
101   /* Set the PRIGROUP[10:8] bits according to NVIC_PriorityGroup value */
102   SCB->AIRCR = AIRCR_VECTKEY_MASK | NVIC_PriorityGroup;
103 }
```

图 10.5 中断分组函数

该函数先检查参数是否合理，然后将该分组号赋给 AIRCR 寄存器，其中的 AIRCR_VECTKEY_MASK 是该寄存器的访问秘钥，对该寄存器写操作就必须先填入秘钥。我们此时将其设置为分组 2：NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2)；

2. 设置相应外设的中断，设置内容选择某个外设中断并设置它的抢占优先级和子优先级。代码如下（以外部中断 0 为例）：

```
NVIC_InitTypeDef NVIC_InitStructure;                          //定义中断结构体

NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;              //设置中断为外部中断0
NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;     //设置抢占优先级为1
NVIC_InitStructure.NVIC_IRQChannelSubPriority = 3;            //设置子优先级为3
NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;               //设置使能该中断

NVIC_Init(&NVIC_InitStructure);                               //生效中断设置
```

图 10.5 中断配置

本章需要大家掌握的是 STM32 的分组概念以及根据分组设置中断优先级，在程序上其实只需要两步即可，第一步是开机分组，第二步是设置对应中断优先级并使能。

第十一章 外部中断实验

11.1 项目要求

使用外部中断实现第九章的按键采集功能。

注：本章只用到核心板，对应例程 4。

11.2 原理讲解

当外部的一个特定信号在单片机引脚上出现时，会引起单片机产生中断动作，这种中断方式就称之为外部中断。所谓的特定信号可以是某种电平状态，也可以是信号上升沿和下降沿的变化。电平状态就是高电平或者低电平，上升沿是指信号由低电平跳转为高电平的过程，下降沿反之类似。

STM32F103 有 19 个外部中断线，其中 16 个分配给了 GPIO（因为每组 GPIO 都有 16 个引脚），剩下的三个外部中断线分配给了：

- EXTI 线 16 连接到 PVD 输出
- EXTI 线 17 连接到 RTC 闹钟事件
- EXTI 线 18 连接到 USB 唤醒事件

我们来看一下外部中断的内部结构框图：

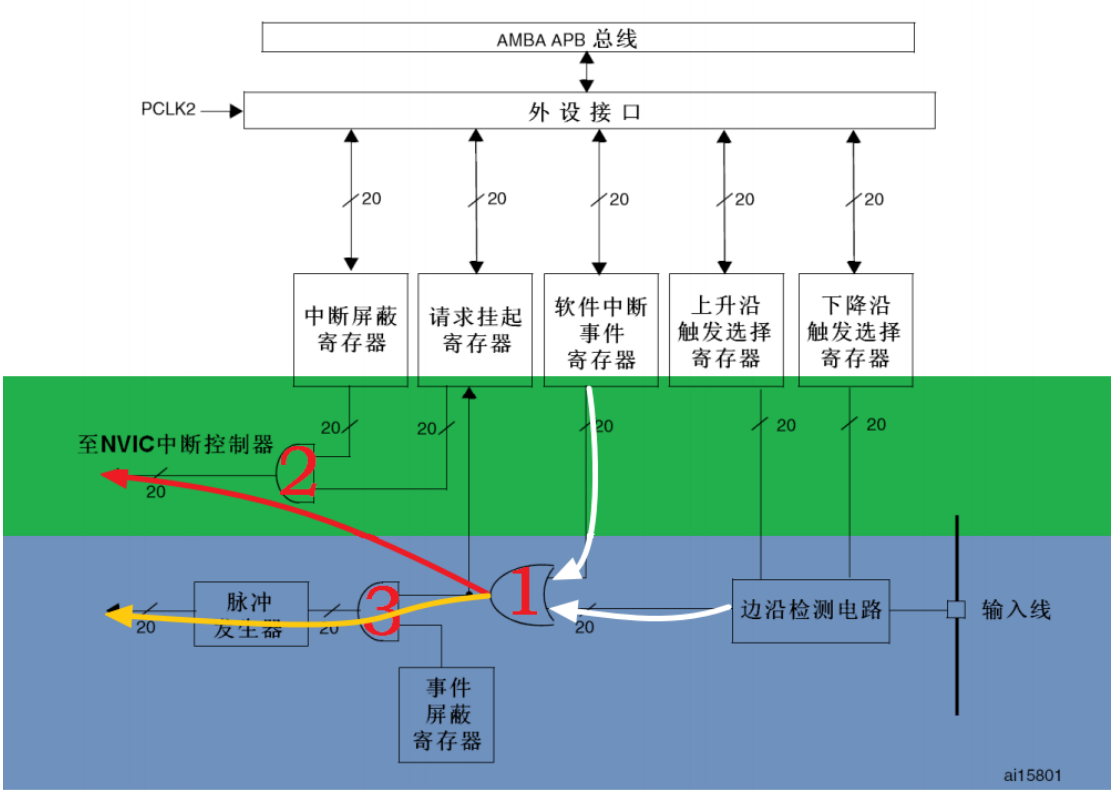


图 11.1 外部中断结构图

STM32F103 的触发来源有两种（标号 1 的或门输入）：

1. 软件触发中断
2. 外部边沿触发中断

STM32F103 的输出结果有两种（绿色和蓝色区域）：

1. 触发外部中断
2. 产生事件脉冲

结合上图作者来讲解一下：

当边沿检测电路检测到引脚上产生用户设置的触发沿时边沿检测电路会输出高电平给标号 1 部分，由于 1 是或门所以输入有一个是 1 输出就是 1，此时 1 号或门的输出给请求中断挂起寄存器，并让该寄存器对应中断线挂起位为 1，如果此时中断屏蔽寄存器使能，即该中断线中断使能位为 1 时，2 号与门输出 1，并传给 NVIC，如果此时 NVIC 使能了该中断线中断，则外部中断发生，外部中断服务函数得以执行。同时，如果事件屏蔽寄存器中的对应位设置为 1，即开放来自该中断线的事件时，3 号与门输出 1，进而使得脉冲发生器产生脉冲给对应外设，对应外设在收到脉冲后执行相应的联动触发动作，如触发 ADC 采集。

同理如果设置软件中断寄存器相应位为 1，即通过软件产生外部中断时也会触发上面所述动作。

那么已经有了外部中断服务函数了，为什么还要存在事件呢，是否多此一举呢？其实不

然，当外部信号触发外部中断时，程序会跳转到外部中断服务函数并执行，这一过程是需要 CPU 参与的，而事件触发则与之不同，事件通过脉冲触发外设动作的过程是由硬件自动完成的，不需要 CPU 干预，减轻了 CPU 处理负担，而且触发速度更快（硬件级），这也是为什么存在事件触发的原因了。当然在平时开发时事件触发用的不多。

我们再说一下分配给 GPIO 的着 16 个中断线，STM32F103 对这 16 个外部中断线的 GPIO 分配使用了一个很巧妙的机制——中断线共享。

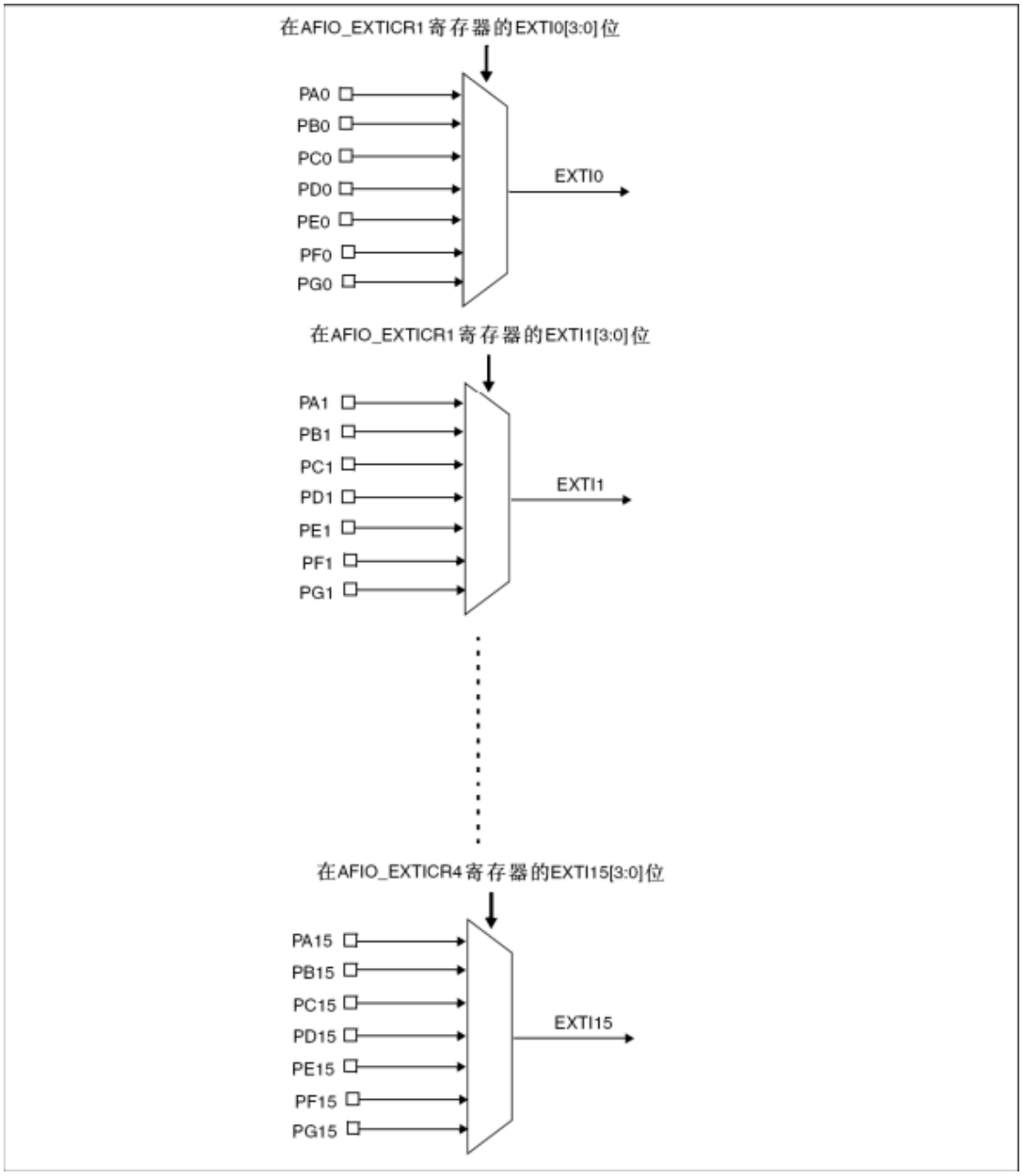


图 11.2 中断线和引脚对应关系

以图中的 EXTI0 为例，我们可以通过程序将 EXTI0 中断线分配到最多 7 个 IO 端口上，这样做的好处是可以灵活配置中断线，当某一个 IO 口（比如 PA0）被占用的时候可以将该中断线分配到其他 IO 口（比如 PB0/PC0/PD0…），因此避免了冲突，另外在绘制 PCB 的时

候也有帮助，我们可以将中断线分配到一个方便布线的引脚，但是要注意的是外部中断在同一时刻只允许分配到一个引脚。

下面来看相关寄存器：

- 中断屏蔽寄存器(EXTI_IMR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												MR19	MR18	MR17	MR16
												rw	rw	rw	rw

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR4	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:20	保留，必须始终保持为复位状态(0)。
位19:0	**MRx**: 线x上的中断屏蔽 (Interrupt Mask on line x) 0：屏蔽来自线x上的中断请求； 1：开放来自线x上的中断请求。 注：位19只适用于互联型产品，对于其它产品为保留位。

图 11.3 中断屏蔽寄存器

当对应中断屏蔽位为 1 时，如果此时挂起寄存器对应位也为 1，则产生 NVIC 中断。

- 事件屏蔽寄存器(EXTI_EMR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												MR19	MR18	MR17	MR16
												rw	rw	rw	rw

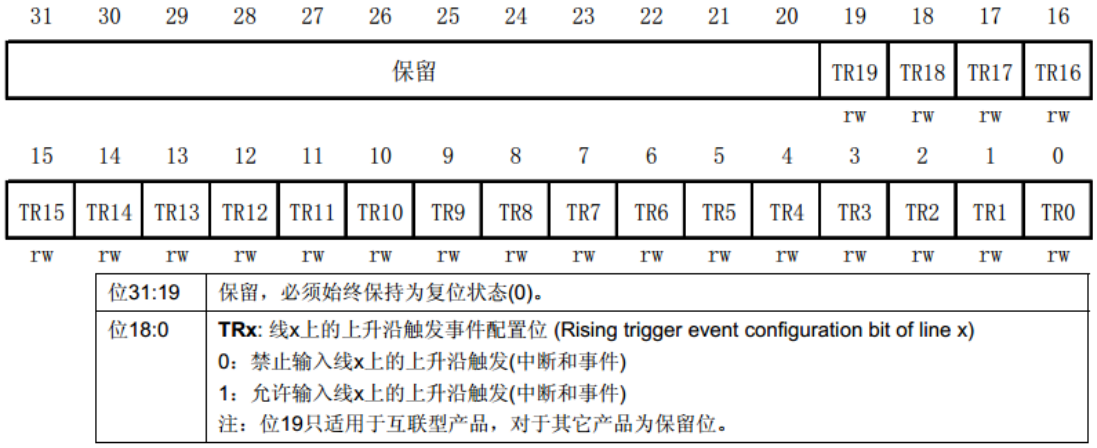
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
MR15	MR14	MR13	MR12	MR11	MR10	MR9	MR8	MR7	MR6	MR4	MR4	MR3	MR2	MR1	MR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:20	保留，必须始终保持为复位状态(0)。
位19:0	**MRx**: 线x上的事件屏蔽 (Event Mask on line x) 0：屏蔽来自线x上的事件请求； 1：开放来自线x上的事件请求。 注：位19只适用于互联型产品，对于其它产品为保留位。

图 11.4 事件屏蔽寄存器

当事件屏蔽寄存器对应位为 1 时，一旦有触发信号产生，就会生成事件脉冲给其他外设。

- 上升沿触发选择寄存器(EXTI_RTSR)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												TR19	TR18	TR17	TR16
												rw	rw	rw	rw

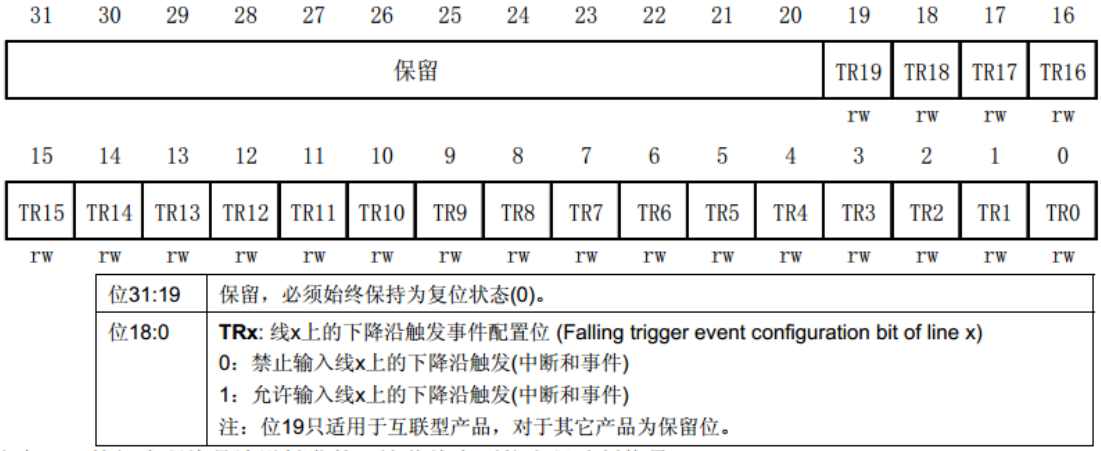
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**TRx**: 线x上的上升沿触发事件配置位 (Rising trigger event configuration bit of line x) 0：禁止输入线x上的上升沿触发(中断和事件) 1：允许输入线x上的上升沿触发(中断和事件) 注：位19只适用于互联型产品，对于其它产品为保留位。

图 11.5 上升沿触发选择寄存器

设置触发外部中断/事件的信号为上升沿触发。上升下降沿寄存器可以同时配合使用来产生上升下降沿触发。

- 下降沿触发选择寄存器(EXTI_FTSR)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												TR19	TR18	TR17	TR16
												rw	rw	rw	rw

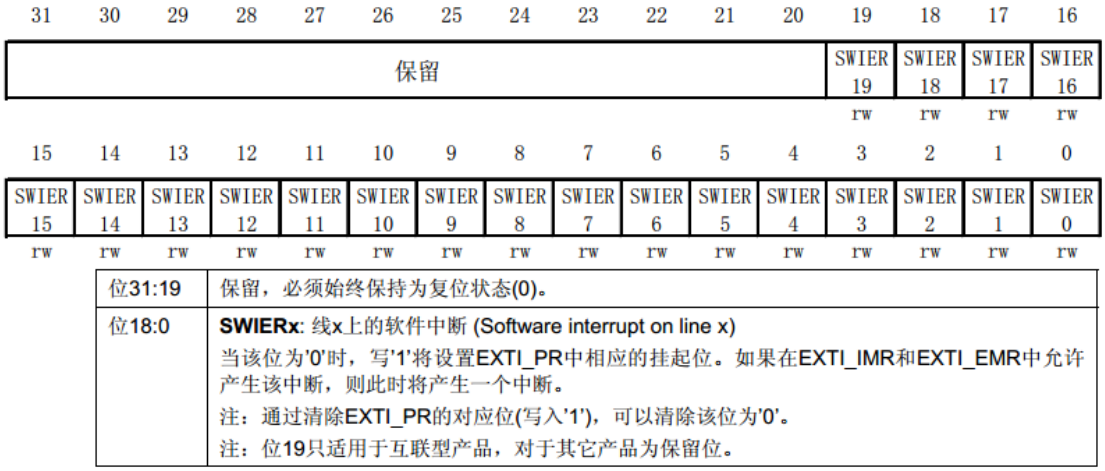
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
TR15	TR14	TR13	TR12	TR11	TR10	TR9	TR8	TR7	TR6	TR5	TR4	TR3	TR2	TR1	TR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**TRx**: 线x上的下降沿触发事件配置位 (Falling trigger event configuration bit of line x) 0：禁止输入线x上的下降沿触发(中断和事件) 1：允许输入线x上的下降沿触发(中断和事件) 注：位19只适用于互联型产品，对于其它产品为保留位。

图 11.6 下沿触发选择寄存器

设置触发外部中断/事件的信号为下降沿触发。上升下降沿寄存器可以同时配合使用来产生上升下降沿触发。

- 软件中断事件寄存器(EXTI_SWIER)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												SWIER19	SWIER18	SWIER17	SWIER16
												rw	rw	rw	rw

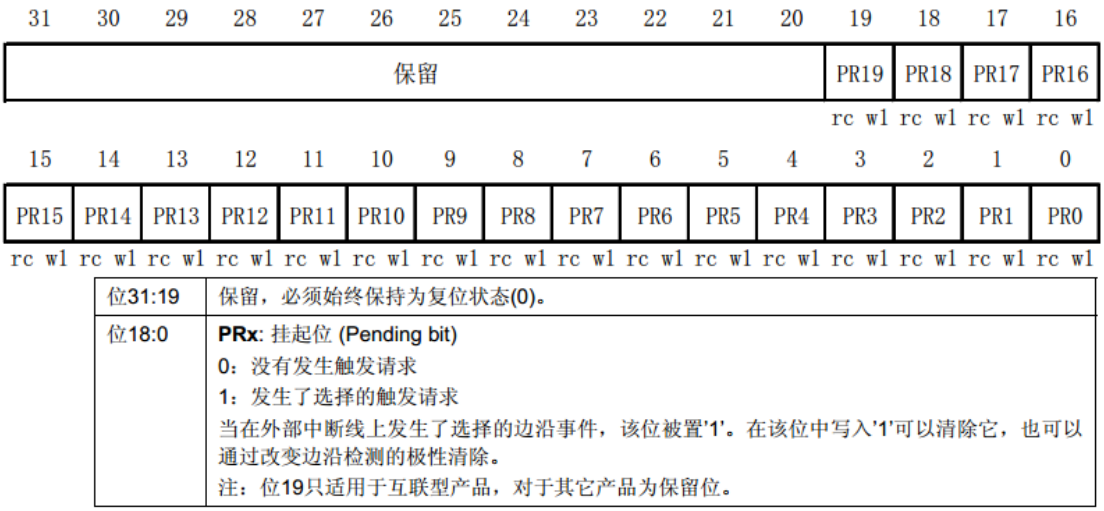
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
SWIER15	SWIER14	SWIER13	SWIER12	SWIER11	SWIER10	SWIER9	SWIER8	SWIER7	SWIER6	SWIER5	SWIER4	SWIER3	SWIER2	SWIER1	SWIER0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**SWIERx**: 线x上的软件中断 (Software interrupt on line x) 当该位为'0'时，写'1'将设置EXTI_PR中相应的挂起位。如果在EXTI_IMR和EXTI_EMR中允许产生该中断，则此时将产生一个中断。 注：通过清除EXTI_PR的对应位(写入'1')，可以清除该位为'0'。 注：位19只适用于互联型产品，对于其它产品为保留位。

图 11.7 软件中断寄存器

通过对某位置 1，可以人为的产生外部中断/事件。

- 挂起寄存器(EXTI_PR)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留												PR19	PR18	PR17	PR16
												rc w1	rc w1	rc w1	rc w1

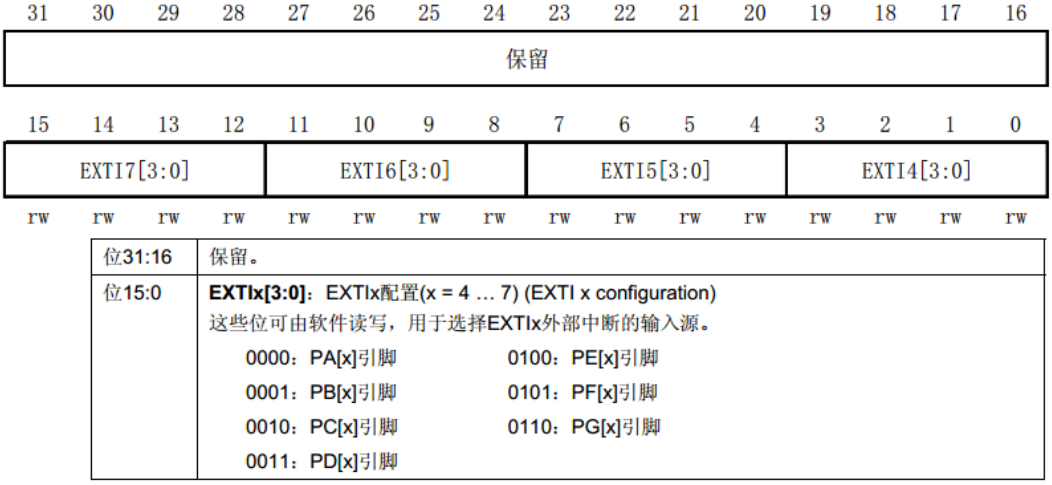
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
PR15	PR14	PR13	PR12	PR11	PR10	PR9	PR8	PR7	PR6	PR5	PR4	PR3	PR2	PR1	PR0
rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1	rc w1

位31:19	保留，必须始终保持为复位状态(0)。
位18:0	**PRx**: 挂起位 (Pending bit) 0：没有发生触发请求 1：发生了选择的触发请求 当在外部中断线上发生了选择的边沿事件，该位被置'1'。在该位中写入'1'可以清除它，也可以通过改变边沿检测的极性清除。 注：位19只适用于互联型产品，对于其它产品为保留位。

图 11.8 挂起寄存器

当外部信号/或软件产生了触发请求（即 1 号或门的输出为 1），则挂起寄存器对应位被设置为 1，通过写 1 清零。

- 外部中断配置寄存器 1-4(AFIO_EXTICR1-4)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
EXTI7[3:0]				EXTI6[3:0]				EXTI5[3:0]				EXTI4[3:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:16	保留。
位15:0	**EXTIx[3:0]**：EXTIx配置(x = 4 … 7) (EXTI x configuration) 这些位可由软件读写，用于选择EXTIx外部中断的输入源。 0000：PA[x]引脚　0100：PE[x]引脚 0001：PB[x]引脚　0101：PF[x]引脚 0010：PC[x]引脚　0110：PG[x]引脚 0011：PD[x]引脚

图 11.9 外部中断配置寄存器

这里只列出 AFIO_EXTICR2，这样的 32 位寄存器一种有 4 组，每组用于设置 4 个外部中断，该寄存器用于指定外部中断的输入信号来自于那组 GPIO 的引脚，目的是将外部中断线和某个对应引脚连接在一起。

对于本章项目来说，我们需要将外部中断设置为下降沿触发，因为平时 GPIO 都是上拉输入的。

11.3 程序讲解

要想实现外部中断需要进行如下设置：

1. 设置 NVIC 中断分组以及对应外部中断的优先级
2. 设置外部中断，设置的内容包括：使用哪个中断线，该中断线使用哪个引脚作为触发信号输入，触发信号的跳变沿，是否使能外部中断屏蔽位。
3. 设置对应的引脚为输入（上/拉，浮空）。

这三步，分别在三个文件中：
第一步——NVIC.c
第二步——EXTI.c
第三步——key.c
我们分别看一下：

```
18  void initNVIC(u32 NVIC_PriorityGroup)
19  {
20      NVIC_PriorityGroupConfig(NVIC_PriorityGroup); //中断分组
21
22      setNVIC(EXTI0_IRQn,1,1,ENABLE); //配置外部中断0优先级
23      setNVIC(EXTI1_IRQn,1,2,ENABLE); //配置外部中断1优先级
24  }
```

```
35  void setNVIC(u8 InterruptNum, u8 PreemptionPriority, u8 SubPriority, FunctionalState NVIC_Sta)
36  {
37      NVIC_InitTypeDef NVIC_InitStructure;                                    //定义NVIC初始化结构体
38
39      NVIC_InitStructure.NVIC_IRQChannel = InterruptNum;                      //指定配置的中断
40      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = PreemptionPriority; //设置抢占优先级
41      NVIC_InitStructure.NVIC_IRQChannelSubPriority = SubPriority;            //设置子优先级
42      NVIC_InitStructure.NVIC_IRQChannelCmd = NVIC_Sta;                       //使/失能中断
43      NVIC_Init(&NVIC_InitStructure);                                         //设置生效
44  }
45
```

图 11.10 NVIC 配置

初始化 NVIC，并设置对应外部中断的优先级：

NVIC_PriorityGroupConfig(NVIC_PriorityGroup);用于设置中断优先级的分组，其内部就是对 AIRCR 寄存器进行设置。设置完分组之后，开始为待设置的中断分配优先级，以及是否中断使能。我们封装的 setNVIC()函数的目的是方便大家在今后使用其他外设中断时能够节约时间。

接下来就是对 EXTI0 和 EXTI1 进行设置：

```
18  void initEXTI_0_1(void)
19  {
20      EXTI_InitTypeDef EXTI_InitStructure;                        //定义外部中断初始化结构体
21
22      RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);        //开启外设复用时钟
23
24      GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource0); //使用PA0作为EXTI0信号输入源
25      EXTI_InitStructure.EXTI_Line = EXTI_Line0;                  //设置外部中断线0
26      EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;         //外部中断模式
27      EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;     //下降沿触发中断
28      EXTI_InitStructure.EXTI_LineCmd = ENABLE;                   //使能外部中断屏蔽寄存器
29      EXTI_Init(&EXTI_InitStructure);                             //设置生效
30
31      GPIO_EXTILineConfig(GPIO_PortSourceGPIOA, GPIO_PinSource1); //使用PA1作为EXTI1信号输入源
32      EXTI_InitStructure.EXTI_Line = EXTI_Line1;                  //设置外部中断线1
33      EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;         //外部中断模式
34      EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;     //下降沿触发中断
35      EXTI_InitStructure.EXTI_LineCmd = ENABLE;                   //使能外部中断屏蔽寄存器
36      EXTI_Init(&EXTI_InitStructure);                             //设置生效
37  }
```

图 11.11 EXTI 配置

由于 EXTI 是复用功能，因此需要开启复用功能外设的时钟，如第 22 行代码处所示，接着要通过 GPIO_EXTILineConfig()函数来制定外部中断线和单片机的哪个引脚连接起来，前面说过，EXTI0 被 PA0,PB0,PC0…共享，因此要根据实际使用情况来确定要使用哪个引脚为 EXTI0 或 EXTI1 提供触发信号。紧接着，我们要对 EXTI 进行设置，分别是要指定我们用哪条中断线，这里使用的中断线 0 和中断线 1，然后要设置是外部中断模式（触发后跳转到中断服务函数）还是外部事件模式（产生事件脉冲给外设），我们这里使用的是外部中断模式，然后设置触发沿的方式，我们使用下降沿触发，最后设置 EXTI 是否使能，其实这一步设置的就是是否使能外部中断屏蔽寄存器或事件屏蔽寄存器。大家感兴趣可以去看一下 EXTI_Init()函数内部是如何实现的，它的实现方法，充分的利用了指针的优点，程序开始它先将 EXTI 寄存器起始地址以整数的形式赋给了变量 temp，然后根据寄存器的偏移地址对 temp 进行自加，最后再把 temp 转换成地址，并修改该地址对应的寄存器内的位。

对于 GPIO 的设置在第九章已经讲过，大家可以参考之前的程序和教程。

接下来看一下外部中断服务函数（以外部中断 0 的中断服务函数为例）：

```
45  void EXTI0_IRQHandler(void)
46  {
47      EXTI_GetFlagStatus(EXTI_Line0);     //获取外部中断挂起位
48      keyvalue = getKeyValue(KEY_PRESS); //获取键值
49      EXTI_ClearFlag(EXTI_Line0);         //清除外部中断挂起位
50
51  }
```

图 11.12 外部中断服务函数

当触发外部中断后，会自动跳转到该函数并执行，但是要注意，该函数名是固定的，如果你想修改中断名字,那么你可以去启动文件中修改或者宏定义函数名即可。

EXTI_GetFlagStatus(EXTI_Line0);函数用于获取对应中断线的挂起位，挂起位为 1 就说明检测到了外部信号的下降沿。但是在这里没有用到该函数，STM32 的外部中断服务函数并不是和外部中断线一一对应的，如果你使用的是外部中断 5 到外部中断 9，这些外部中断共用一个外部中断服务函数 void EXTI9_5_IRQHandler(void),同理外部中断 10 到外部中断 15 对应的中断服务函数为：void EXTI15_10_IRQHandler(void),在这种情况就需要在外部中断服务函数中判断此时是哪个外部中断发生了。另外还有一个和它数类似的函数 EXTI_GetITStatus(uint32_t EXTI_Line),他们之间的不同之处在于，该函数要同时满足此时是外部中断模式且该中断挂起位为 1 时才返回 1，它只用于判断外部中断模式下的挂起位。实现函数如下：

```
224 ITStatus EXTI_GetITStatus(uint32_t EXTI_Line)
225 {
226   ITStatus bitstatus = RESET;
227   uint32_t enablestatus = 0;
228   /* Check the parameters */
229   assert_param(IS_GET_EXTI_LINE(EXTI_Line));
230
231   enablestatus =  EXTI->IMR & EXTI_Line;
232   if (((EXTI->PR & EXTI_Line) != (uint32_t)RESET) && (enablestatus != (uint32_t)RESET))
233   {
234     bitstatus = SET;
235   }
236   else
237   {
238     bitstatus = RESET;
239   }
240   return bitstatus;
241 }
```

图 11.13 函数实现

而 EXTI_GetFlagStatus(EXTI_Line0)实现过程则没有这个判断：

```
186  FlagStatus EXTI_GetFlagStatus(uint32_t EXTI_Line)
187  {
188    FlagStatus bitstatus = RESET;
189    /* Check the parameters */
190    assert_param(IS_GET_EXTI_LINE(EXTI_Line));
191
192    if ((EXTI->PR & EXTI_Line) != (uint32_t)RESET)
193    {
194      bitstatus = SET;
195    }
196    else
197    {
198      bitstatus = RESET;
199    }
200    return bitstatus;
201  }
```

图 11.14 函数实现

中断服务函数的最后不要忘记使用 EXTI_ClearFlag(uint32_t EXTI_Line)或 EXTI_ClearITPendingBit(uint32_t EXTI_Line)函数对挂起位清零，如果不对挂起位清零的话，那么该中断服务函数执行完之后又会立即重新执行，导致的结果就是程序卡死。对于这两个清除挂起位的函数来说其内部的代码完全一样，这么做的目的我觉得是要和 EXTI_ClearITPendingBit(uint32_t EXTI_Line)函数成对出现。

另外，该中断服务函数中的 keyvalue 变量是全局变量，可以在不同文件之间访问。要想达到这种效果，首先要在某个文件内设置为全局变量，其他文件在用到这个变量时，使用 extern 关键字即可。

最后就是主函数，主函数除了增加了 EXTI 和 NVIC 的初始化函数之外其他的和第九章实现的功能相同，这里就不做过多赘述。我们编译链接-烧录-重启后，使用按键 UP 和 DOWN 设置 LED 闪烁频率，此时你可能还会发现在 DOWN 模式下（2s 周期闪烁）按 UP 键后，并不会立即切换到 UP 模式，存在一个不固定的延时，其原因并不是 UP 键失灵，在执行 DOWN 模式闪烁期间，的确发生了 UP 键对应的外部中断并执行了外部中断服务函数，也更新了全局变量 keyvalue 的值，只不过是此时正在执行耗时的 LED 翻转函数，还没有执行到 switch-case 语句来根据键值设置 time 变量的代码位置。

到这里本章就讲解完成了，需要大家了解的就是配置 EXTI 的三步曲：

1. 设置 NVIC
2. 设置 EXTI
3. 设置 GPIO 为输入

另外，事件和外部中断要分清，它是外部触发信号发生后导致的两种结果。外部中断被触发则会执行中断服务函数，事件被触发则会产生脉冲给对应的硬件，这一过程是硬件自动完成，无需人为编程。

通过第九章和本章的例程，我希望能让大家意识到在嵌入式开发时很重要的一点：**要极力避免长时间阻塞单片机，因为这会对整个系统的实时性产生很糟糕的影响，同时也会浪费单片机的硬件资源。**

这里给大家留一个作业：利用我在第九章最后讲解的方法五，来提高整个程序的实时性。

提示：在 main 函数中将 LED 翻转的周期都固定为 1ms。

第十二章 UART 串口通信

12.1 项目要求

本章为大家准备两个实验：

1. 单片机每隔 1s 打印一次：www.fengmeitech.club
2. 上位机发送十六进制的 0 和十六进制的非 0 整数来控制 LED 灯状态，并且当按键按下会向上位机发送此时的键值。

注：本章只用到核心板，对应例程 5、6。

12.2 原理讲解

UART 也就是我们常说的串口，但是严格意义上讲，UART 是串口通信的一种，只不过大家习惯称 UART 通信为串口通信。

我们先了解一下有关通信的概念：

- 串口通信

串口通信是指数据通过一条数据线（或者两条差分线）一位接着一位的传输出去，类似于单车道。串口通信的优点是占用硬件资源少，且传输距离较远，缺点是传输速度慢（这些优缺点都是相对于并口通信而言，其实在平时应用中串口的通信速度完全能够满足绝大部分通信场景的要求）。

- 并口通信

并口通信是说通过多条数据线一次性的将数据传输出去，类似于多车道，其优点当然就是传输速度快，但是缺点也很明显，第一它占用更多的硬件资源，第二是它的传输距离短，对 PCB 布线有更高的要求，因为每条数据线如果不等长会导致寄生电容也不同，最终会在数据线之间产生传输延时，导致数据传输失败。并口一般应用在 LCD 显示和快速存储设备上面。

- 单工通信

单工通信是指无论在何时数据的流向是单向的，接收端只负责接收不具备发送功能，发送端只负责发送，不具备接收功能。

- 半双工通信

半双工通信是指数据传输方向是双向的，但是在同一时刻只允许数据单向传输，比如我

们即将学习的 UART 和 IIC

- 全双工通信

全双工通信则支持在同一时刻数据可以双向传输，比如 SPI

- 同步通信方式

同步通信方式是指收发双方之间通过一条时钟信号线来同步数据的传输，在发生传输时必须在时钟线的同步下将数据一位一位的传输出去，这种方式传输速度较快，但是因为时钟和数据线是单独分开的，因此它的传输距离较近，典型的应用就是 IIC 和 SPI，他们都有一条时钟线来同步数据线上的位传输。

- 异步通信方式

异步通信方式是指，收发双方没有数据线同步，要通过特定的起始以及停止标志位来判断此时数据是否开始传输以及是否传输完成。另外异步通信的收发双发要保证通信速度的统一，典型的就是 UART，RS485 等。

本章学习的 UART（Universal Asynchronous Receiver/Transmitter，通用异步收发传输器）属于异步串行通信，UART 是我们平时使用最多的一个串行通信方式，主要应用场景如下：

1. 用于打印程序调试信息
2. 和上位机配套软件通信，使得操作人员可以通过上位机来查看或者设置下位机
3. 用于 ISP 或者 IAP 程序下载

我们使用的 STM32F103C8T6 单片机有三个 USART，USART 和 UART 的区别在于 USART 支持同步模式，在同步模式下有一个时钟信号线用于同步数据，但是我们平时使用的都是 UART 的异步通信，可以说 UART 是 USART 的一个子集，除此之外 STM32 的 USART 还支持 LIN(局部互连网)，智能卡协议和 IrDA SIR ENDEC 规范，但是平时用的最多就是 UART，因此本章着重讲解 UART。

要想使用 UART，就必须确保收发双方的通信约定要一致，这些约定包括：波特率（通信速度），数据位（8 位或 9 位），停止位个数（1 位，1.5 位，2 位），校验方式(奇校验，偶校验，无校验)。波特率就是要制定双方通信的速度，我们经常使用的波特率有：（1200,2400,4800，9600,115200），数据位就是我们想要传输的有效数据的部分，因为一个字节是 8 位，因此经常使用的也是 8 位数据位，一般 9 位数据位的应用场合有以下几种：

1. 8 位数据的扩展（8 位可以表示 256 种可能，9 位可以表示 512 种可能）
2. 使用第 9 位来表示此时数据传输的模式，比如第九位为 1 代表此时是读模式，为 0 表示是写模式。
3. 使用第 9 位作为负数，例如第 9 位为 1 表示-，第 9 位为 0 表示+

另外就是停止位，停止位是双发通信的结束标志，常用的是 1 位停止位，最后是校验方式，校验可以保证数据传输的正确性、完整性，但是会增加数据帧的长度，通常我们使用无

奇偶校验。

下面看一下 STM32 的结构框图：

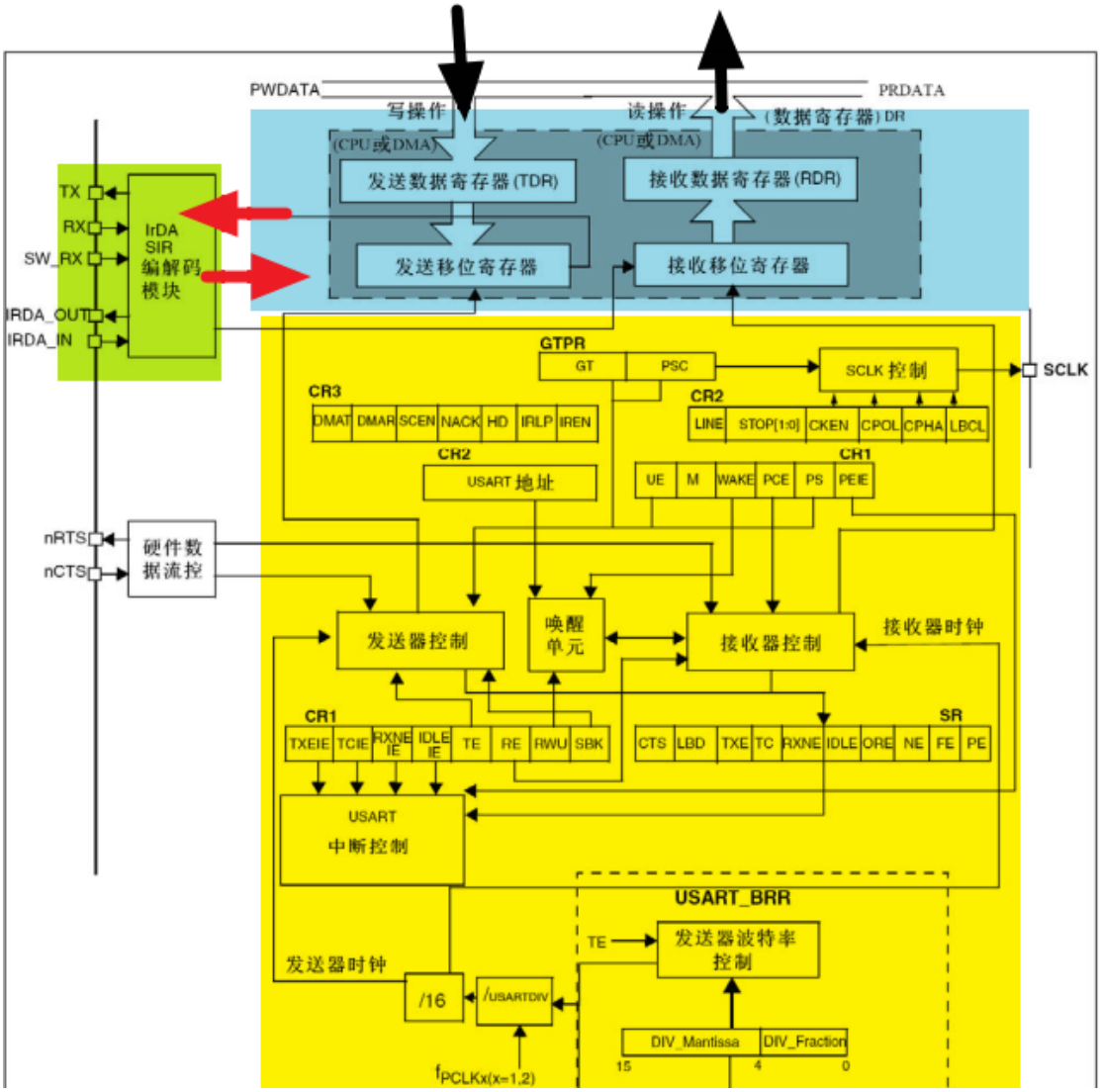


图 12.1 USART 结构框图

UART 由三大部分组成：第一部分就是用于设置和标记传输模式的控制寄存器和标志寄存器部分（黄色高亮区域），第二部分是引脚通信（绿色高亮），该部分和发送/接收移位寄存器进行连接，第三部分是数据转换部分（蓝色高亮），该部分将发送数据寄存器内的数据通过移位寄存器传输到 TXD 引脚，接收同理。

关于寄存器部分，我会结合程序对个别需要注意的寄存器进行讲解，大家感兴趣可以到 540 页查看 USART 所有寄存器功能。

要想使用 UART，需要先进行配置：

1. 配置 GPIO，TXD 引脚要设置为复用推挽输出，RXD 要设置为浮空输入
2. 配置 UART，包括波特率、数据位、停止位、奇偶校验位以及是否使用硬件流控，硬件流控通过 RTS 和 CTS 作为流控引脚，我们一般应用是不需要的。
3. 如果想实现串口的发送或接受中断，那么还要设置 NVIC

12.3 程序讲解

- 串口初始化函数

```
18  void initUART(void)
19  {
20      GPIO_InitTypeDef GPIO_InitStructure;
21      USART_InitTypeDef USART_InitStructure;
22
23      RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1 | RCC_APB2Periph_GPIOA, ENABLE);  //使能USART1时钟
24      /*******************GPIO Config*************************/
25      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;                                      //发送管脚
26      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;                                //复用推挽
27      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
28      GPIO_Init(GPIOA, &GPIO_InitStructure);
29
30      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;                                     //接收管脚
31      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;                          //浮空输入
32      GPIO_Init(GPIOA, &GPIO_InitStructure);
33
34      /*******************UART Config*************************/
35      USART_InitStructure.USART_BaudRate = 9600;                                     //设置波特率
36      USART_InitStructure.USART_WordLength = USART_WordLength_8b;                    //8bits数据位
37      USART_InitStructure.USART_StopBits = USART_StopBits_1;                         //1bit停止位
38      USART_InitStructure.USART_Parity = USART_Parity_No;                            //无奇偶校验位
39      USART_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;//不使用硬件流控
40      USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;                //发送接收均使能
41      USART_Init(USART1, &USART_InitStructure);                                      //设置生效
42      USART_Cmd(USART1, ENABLE);                                                     //使能串口外设
43      USART_ITConfig(USART1, USART_IT_RXNE, ENABLE);                                 //使能串口接收中断
44
45  }
```

图 12.2 UART 初始化函数

在初始化函数中，依次初始化了用于通信的 TXD 和 RXD 引脚，然后设置了串口的波特率为 9600,8 位数据位，1 位停止位，无奇偶校验位，无硬件流控，USART_Init()函数用于将各个设置写到对应寄存器当中，生效设置，对于 USART_Cmd()大家可能会有疑问，它是用来干什么的，下面是该函数定义：

- UART 使能函数

```
351  void USART_Cmd(USART_TypeDef* USARTx, FunctionalState NewState)
352  {
353    /* Check the parameters */
354    assert_param(IS_USART_ALL_PERIPH(USARTx));
355    assert_param(IS_FUNCTIONAL_STATE(NewState));
356
357    if (NewState != DISABLE)
358    {
359      /* Enable the selected USART by setting the UE bit in the CR1 register */
360      USARTx->CR1 |= CR1_UE_Set;
361    }
362    else
363    {
364      /* Disable the selected USART by clearing the UE bit in the CR1 register */
365      USARTx->CR1 &= CR1_UE_Reset;
366    }
367  }
```

图 12.2 USART_CR1 寄存器 UE 位使能函数

它用于设置 USART 控制寄存器 1 中的 UE 位，UE 位可以理解为是 USART 外设的开关，关闭它可以降低功耗，在使用的时候在开启它而不需要重新设置 USART 参数，下面是寄存器中的描述：

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

位31:14	保留位，硬件强制为0。
位13	**UE：USART使能 (USART enable)** 当该位被清零，在当前字节传输完成后USART的分频器和输出停止工作，以减少功耗。该位由软件设置和清零。 0：USART分频器和输出被禁止； 1：USART模块使能。

图 12.3 USART_CR1 UE 位说明

大家感兴趣可以把条语句注释掉，看看一下结果。

另外就是第 43 行 USART_ITConfig(USART1, USART_IT_RXNE, ENABLE)函数用于开启 USART 中的某个中断，USART 的中断非常多，有发送数据寄存器空中断，发送完成中断，接收寄存器非空中断等等，这里选择接收寄存器非空中断，它的意思是说当接收数据寄存器中接收到数据后它会产生中断，跳转到对应串口的中断服务函数中，添加这条语句是为下一个串口实验准备的，本实验用不到，而且我们在 NVIC 中关了 USART 的总中断，所以 USART 任何中断都不生效。

然后看一下通过 UART1 发送单字节数据的函数：

- 发送字节函数

```
47  /**
48   * 功能：指定某个UART发送一个字节
49   * 参数：USARTx：使用的目标串口x为1-3
50   *       byte：待发送字节
51   * 返回值：None
52   */
53  void sendByte(USART_TypeDef *USARTx, u16 byte)
54  {
55      USART_SendData(USARTx, byte);                       //发送一个字节
56      while (!USART_GetFlagStatus(USARTx, USART_FLAG_TC));//等待发送完成
57      USART_ClearFlag(USARTx, USART_FLAG_TC);             //软件清除发送完成标志位
58  }
```

图 12.4 UART 发送单字节函数

USART_SendData()函数用于将参数 byte 填入到发送数据寄存器当中，byte 是 16 位的原因是 byte 要支持 9 位数据位的情况。中间的 while 循环用于等待这一字节发送完成才结束该函数。USART_FLAG_TC 就是发送完成的标志，如果把该条语句注释掉，就会导致发送不出数据，因为在本次数据还没发送完成时第二次发送任务就又来了（假设是连续发送的场合），导致数据不停的被刷新，大家可以注释掉看看效果。最后我们获取到发送成功标志位之后可以选择手动清除，当然不清除也没关系，它会被自动清除，清除的条件是：

位6	**TC**：发送完成 (Transmission complete) 当包含有数据的一帧发送完成后，并且TXE=1时，由硬件将该位置'1'。如果USART_CR1中的TCIE为'1'，则产生中断。由软件序列清除该位(先读USART_SR，然后写入USART_DR)。TC位也可以通过写入'0'来清除，只有在多缓存通讯中才推荐这种清除程序。 0：发送还未完成； 1：发送完成。

图 12.5 传输完成标志位

很明显我们在 56 行已经读取了 USART_SR，并且在下一次发送的时候在第 55 行将数据写入到了 USART_DR 中，但是为了兼容所应用场合还是建议加上清除操作。

下面是根据发送一个字节函数衍生出发送字符串函数：

- 发送字符串函数

```
60  /**
61   * 功能：指定某个串口发送字符串
62   * 参数：USARTx：使用的目标串口x为1-3
63   *       str：字符串指针
64   * 返回值：None
65   */
66  void sendString(USART_TypeDef *USARTx, char *str)
67  {
68      while (*str)
69      {
70          sendByte(USARTx,*str++);
71      }
72  }
```

图 12.6 发送字符串函数

在讲解之前要明确几个概念：

1. C 语言中字符串都是以数字 0 结尾的
2. 字符串地址等于字符串中首个字符的地址
3. Str 为地址，*Str 就是地址指向的数据内容

68 行 while(*Str)用于判断的是此时是否到达字符串末尾，即：如果此时*Str 等于 0 就说明字符串结束，停止发送数据。

70 行是用于将*Str 代表的数据发送出去，并对 Str 加 1 供 while 判断。

- 重定向 fputc()函数

```
109  int fputc(int ch, FILE *f)
110  {
111      /* 将Printf内容发往串口 */
112      USART_SendData(USART1, (unsigned char)ch);
113      while (!USART_GetFlagStatus(USART1, USART_FLAG_TC))
114          ;
115      USART_ClearFlag(USART1, USART_FLAG_TC);
116
117      return (ch);
```

图 12.7 fputc 函数重定向

重定向该函数后就可以在 STM32 上面使用 printf 函数了，另外还要进行一处设置，点击魔术棒按钮，勾选是用微库：

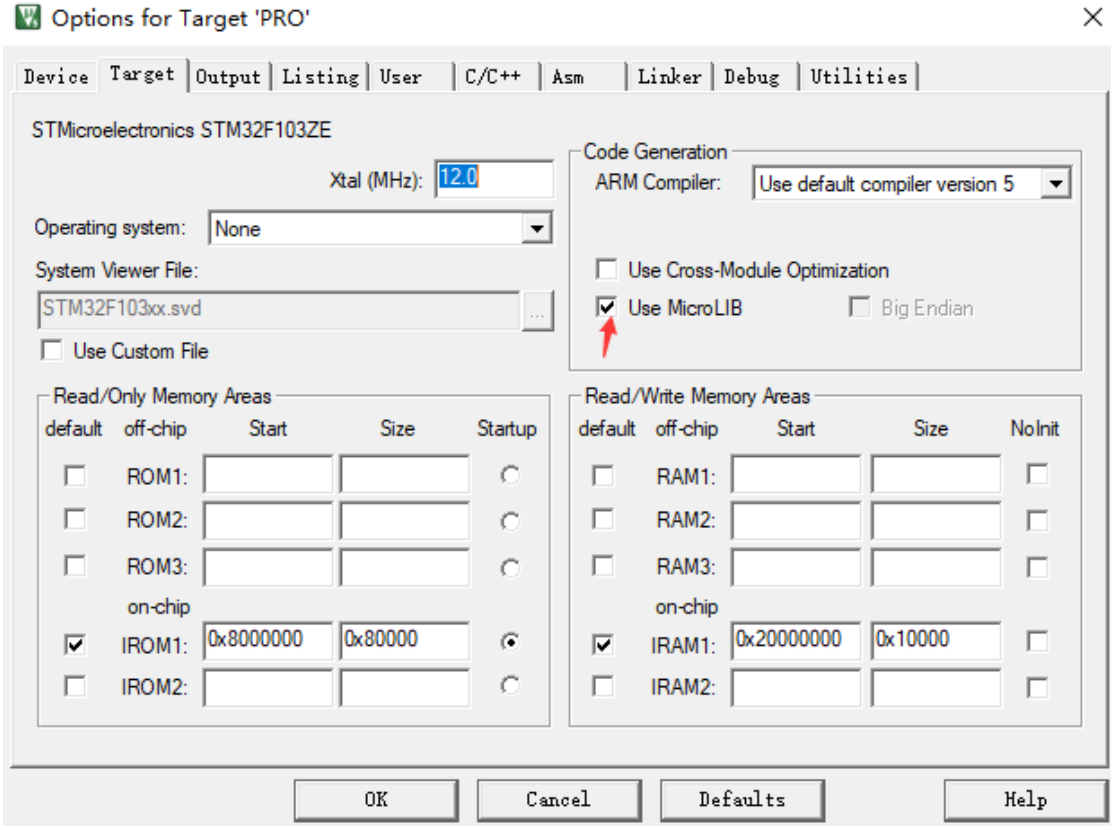


图 12.8 勾选微库

- NVIC 初始化函数

```
18  void initNVIC(u32 NVIC_PriorityGroup)
19  {
20      NVIC_PriorityGroupConfig(NVIC_PriorityGroup); //中断分组
21
22      setNVIC(USART1_IRQn,1,1,DISABLE);
23  }
```

图 12.9 初始化 NVIC

在这里设置了中断分组，指定了 USART1 的中断优先级，但是失能了 USART 的总中断，因为本第一个实验没有用到 USART 的中断，如果你使能了它，那么，你在上位机上发送数据给单片机会导致单片机卡死，因为此时我们的中断服务函数内什么也没有，因此中断标志位没有被清除，中断服务函数就是一直重复执行，感兴趣可以试一试。

- 主函数

```
15  int main(void)
16  {
17      initLED();
18      initNVIC(NVIC_PriorityGroup_2);
19      initUART();
20      initSysTick();
21      while (1)
22      {
23  //      printf("www.fengmeitech.club\n");
24          sendString(USART1,"www.fengmeitech.club\n");
25          toggleLED();
26          Delay_ms(1000);
27      }
28  }
```

图 12.10 主函数

printf()函数我注释掉了，大家可以去掉注释试一试效果，另外使用 printf 函数的好处是可以打印任意指定格式的整形，字符，浮点数等数据。

配置有人串口网络调试二合一工具：

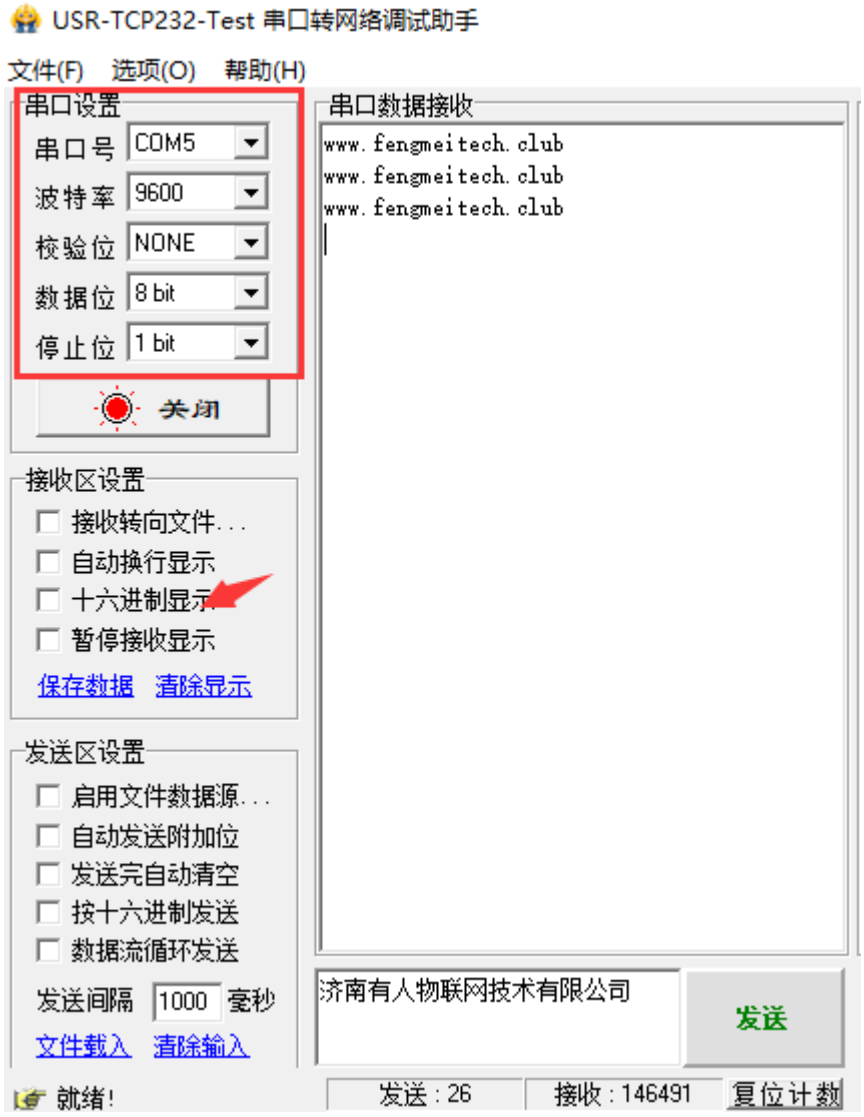


图 12.11 串口配置

编译链接-烧录-重启，可以看到每隔一秒打印一次风媒电子的博客地址 www.fengmeitech.club，同时 LED 灯 1s 周期闪烁，到此串口的第一个实验完成了。

接下来看一下第二个实验，在该实验中要向大家介绍一下面向对象的编程方法。

万物皆对象，对于每一个物体而言都有它的属性和动作，比如人类的属性包括：姓名，性别，年龄，身高，体重，动作包括讲话，行走等等。因此我们可以将这些参数和动作集中到一起管理，这就是面向对象的概念，它的好处是可以让一个复杂的项目经过对象的封装之后结构变得清晰，使程序员不必去管理庞杂混乱的变量和函数，当然 C 语言不属于面向对象的编程语言，但是却可以以面向对象的方式编程，最常用的就是使用结构体来模拟面向对象语言当中的类概念。如下图：

```
typedef struct
{
/*********属性*************/
    char * Name;
    float Height;
    float Weight;
    u8 Age;
    char * Sex;
/*********动作*************/
    void (*speak)(char * message);
    void (*run)(u8 speed);
}HUMAN;
```

图 12.12 面向对象演示伪代码

对于一个项目而言，它也会有很多参数，比如一个物联网产品，它会有温度，湿度，光照强度，继电器状态等，动作包括采集传感器，控制继电器，联网等。这些特征共同描述了这个产品，本实验就试着带领大家使用这种编程方式，我们在 USR 文件中新建了 devtype.h 文件，并在里面声明了设备的抽象描述：

```
typedef struct
{
    u8 LEDStatus; //此时LED状态 ON为亮 OFF为灭
    u8 KeyValue;//当前键值
}DevStatusStructure;
```

图 12.13 设置参数结构体

包括 LED 状态和键值，我们在主函数中定义了这个变量实体：

DevStatusStructure DevStatus = {OFF,KEY_NO}; 并赋予初始值。今后对于所有的操作都是针对该结构体中的这两个参数的，我们通过上位机发来的数据改变 LED 状态，通过采集到的按键键值来向上位机报告。对该结构体内容的修改代码位于 uart.c 和 EXTI.c 中，主函数如下：

```
    while (1)
    {
        if(DevStatus.LEDStatus!=0)
        {
            openLED();
        }else
        {
            closeLED();
        }

        switch(DevStatus.KeyValue)
        {
            case KEY_UP:
                        sendString(USART1,"KEY UP pressed!\n");
                        DevStatus.KeyValue = KEY_NO;
                                                                    break;
            case KEY_DOWN:
                        sendString(USART1,"KEY DOWN pressed!\n");
                        DevStatus.KeyValue = KEY_NO;
                                                                    break;

            default : DevStatus.KeyValue = KEY_NO;                  break;
        }
    }
}
```

图 12.14 主函数

主函数中分为两大部分，第一部分是和 LED 有关代码，第二部分是和按键有关代码。编译链接-烧录-重启，在串口软件发送配置栏勾选“按十六进制发送”：

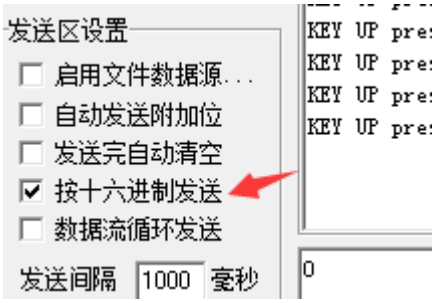


图 12.15 串口软件配置

此时按下按键可以看到串口软件接收栏收到单片机发来的键值数据，我们在发送栏输入 0 并发送，输入 1 再发送，观察效果：

图 12.16 运行结果

本实验展示的面向对象编程方式，只是一个演示作用，在实际工程中还会有其他的参数后者动作加进来，面向对象的编程方式我们在开发项目的时候用的很多，它的好处在大型复杂项目上尤为凸显，另外大家也可以根据我前面所述，对结构体进行升级，比如将 LED，按键等外设同时使用结构体封装起来，然后将这些结构体变量添加到 DevStatus 中，使 DevStatus 变量的实用性更强。

第十三章 模数转换_ADC

12.1 项目要求

使用 ADC 采集光照强度，并通过串口将其打印出来。

注：本章用到核心板+扩展板，对应例程 7。

12.2 原理讲解

学习 ADC 之前先了解两个概念：

- 模拟信号

模拟信号是指时间上和数值上均连续的信号，其特点就是连续，比如温度传感器，压力传感器，光强传感器的输出值等都是模拟信号，模拟信号的优点的是它可以精确的表示事物，如自然界的音频信号，到时缺点也同样明显，它容易受到噪声干扰，且有可能造成噪声积累。

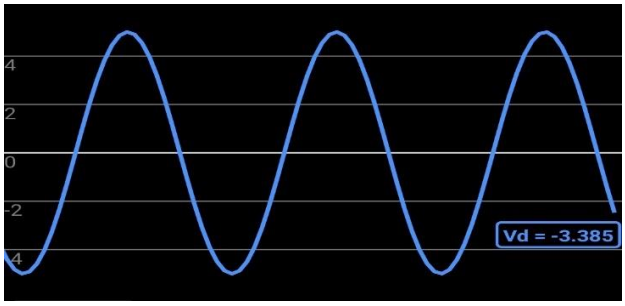


图 13.1 模拟信号

- 数字信号

数字信号是指高电平和低电平两个二进制数字量的信号，因此数字信号是一种矩形波信号。因为只有两种电平状态，因此对噪声有很强的抗干扰能力，且不会产生噪声积累。通过逻辑门电路对数字信号进行运算就组成了我们现在用到的数字器件。

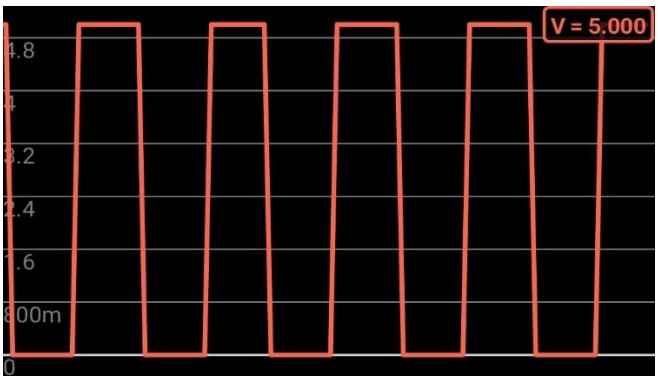


图 13.2 数字信号

但是现实情况是很多传感器的输出都是模拟信号，但是单片机只能处理数字信号，因此就需要一种执行机构，能够把模拟信号通过采样转换成数字信号，拥有这种细腻感知能力的外设就是本章的主角——ADC（Analog-to-Digital Converter）。

ADC 有两个参数，一个是转换周期，一个是分辨率。

1. 采样周期

   采样周期也可以叫转换率，它指的是 ADC 采集模拟信号的速度，试想一个模拟信号的周期是 1KHz，而采样周期是 500Hz，那么就会造成采集到的数据不能完全表示输入的模拟信号，因此，可以将转换周期认为是数字信号还原模拟信号的程度。

2. 分辨率

   分辨率是指单位数字变化对应的输入模拟信号电压的差值，比如 STM32 单片机的 ADC 模拟输入电压范围是 0 到 3.3V，分辨率是 12bit，那么单位数字对应的模拟信号的差值就应该是：

$$V\Delta = 3.3V/(2\hat{}12) \approx 0.0008V$$

接下来开始讲解 STM32 单片机的 ADC 外设。

STM32 的 ADC 是一个 12 位分辨率的逐次逼近型模拟数字转换器。它有多达 18 个通道，可测量 16 个外部和 2 个内部信号源（内部温度传感器和内部参考电压）。各通道的 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/低阀值。

下面是它的内部结构框图：

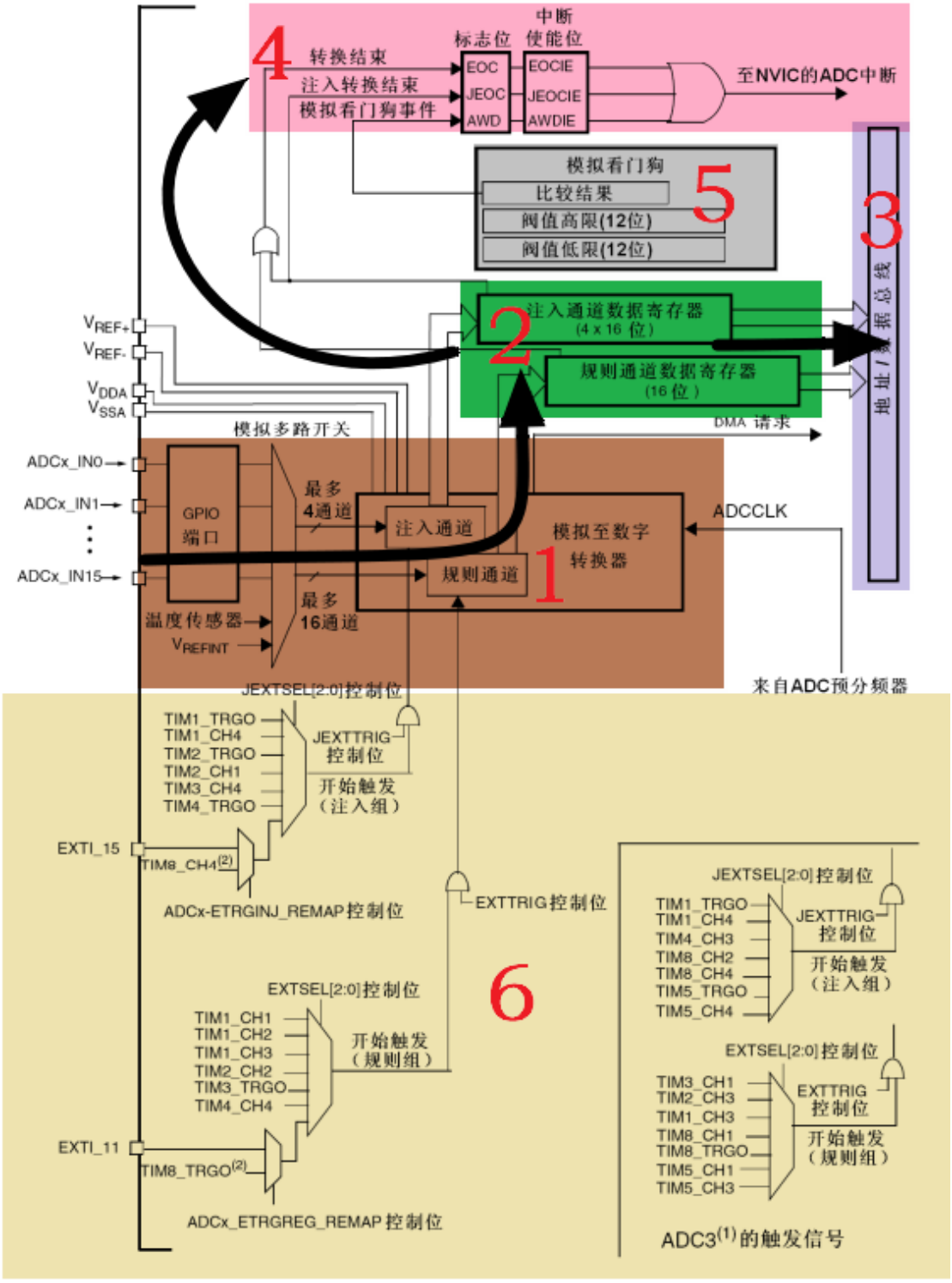


图 13.3 ADC 内部结构图

1 区是 ADC 最核心的部分，可以把它比喻为 ADC 的数据引擎，负责把模拟信号燃料转换成数字信号并泵到对应数据寄存器，它包含了 ADC 转换部分，各个控制和标志位寄存器，该区完成的就是模拟到数字的转换过程。

2 区用于存放转换过后的结果，一共有 5 个数据寄存器（4 个注入通道对应 4 个，16 路

规则通道共用一个），然后 2 区的数据寄存器和 3 区数据总线相连用于读取转换结果。

在转换过程中会产生各种事件，包括转换结束，注入转换结束，模拟看门狗事件等，如果置位这些事件的中断使能位，将产生 NVIC 中断，这些发生在 4 区。

6 区用于触发 ADC 转换，这些触发可以自动的进行 ADC 采集转换，例如前面说过的外部中断事件产生的脉冲信号就是 ADC 触发源的其中一个，但是在一般情况下，我们还是选择手动的进行 ADC 采集，这样灵活性高，也好配置。

5 区的模拟看门狗可用于检测外部电压变化，如果外部电压没有在上下阈值之间则会产生模拟看门狗事件。

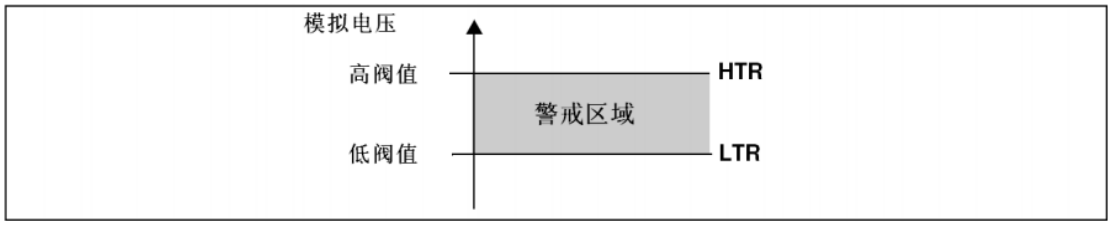
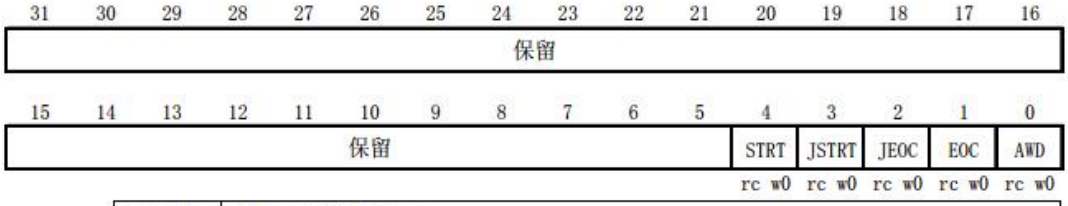


图 13.4 模拟看门狗阈值

内部结构框图上转换通道分为规则转换通道和注入转换通道，平时使用最多的还是规则转换通道，注入通道类似于 ADC 转换中的 VIP，它可以强行插队规则通道的 ADC 转换，当注入转换开始时会打断当前规则转换，注入转换可以应用在一些需要紧急转换的场合，可以理解为注入转换是一种高优先级的 ADC 转换。

接下来看一下要用到的寄存器。

- ADC 状态寄存器(ADC_SR)



位31:15	保留。必须保持为0。
位4	**STRT**：规则通道开始位 (Regular channel Start flag) 该位由硬件在规则通道转换开始时设置，由软件清除。 0：规则通道转换未开始； 1：规则通道转换已开始。
位3	**JSTRT**：注入通道开始位 (Injected channel Start flag) 该位由硬件在注入通道组转换开始时设置，由软件清除。 0：注入通道组转换未开始； 1：注入通道组转换已开始。
位2	**JEOC**：注入通道转换结束位 (Injected channel end of conversion) 该位由硬件在所有注入通道组转换结束时设置，由软件清除 0：转换未完成； 1：转换完成。
位1	**EOC**：转换结束位 (End of conversion) 该位由硬件在(规则或注入)通道组转换结束时设置，由软件清除或由读取ADC_DR时清除 0：转换未完成； 1：转换完成。
位0	**AWD**：模拟看门狗标志位 (Analog watchdog flag) 该位由硬件在转换的电压值超出了ADC_LTR和ADC_HTR寄存器定义的范围时设置，由软件清除 0：没有发生模拟看门狗事件； 1：发生模拟看门狗事件。

图 13.5 ADC 状态寄存器

该寄存器用于表示此时 ADC 的转换状态，最常用的就是 EOC 位，即转换结束，如果该位为 1，则说明此时 ADC 转换完成，可以进行读取了。该位可以通过软件清除，也可以通过读 ADC 数据寄存器进行自动清零（类似于前面讲到的 USART 接收数据寄存器），此时如果对应的中断允许位为 1，则会触发对应中断给 NVIC。

- ADC 控制寄存器 1(ADC_CR1)

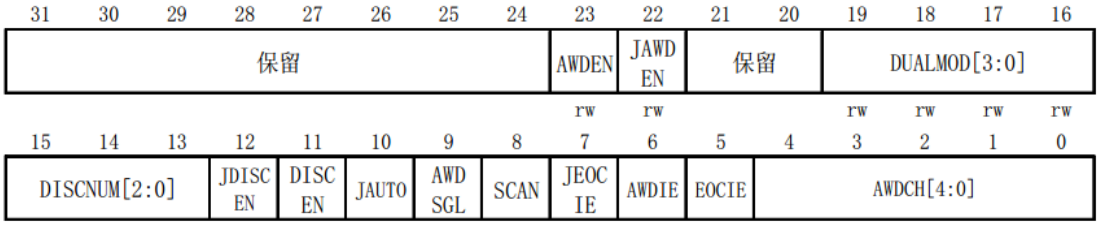


图 13.6 ADC 控制寄存器 1

这里说一下 DUALMOD 位域，该位域用于设置 ADC 的操作模式，STM32 ADC 的操作模式非常丰富，可以满足很多特殊使用场景，但是我们一般情况使用独立模式比较多：

位19:16	**DUALMOD[3:0]**：双模式选择 (Dual mode selection) 软件使用这些位选择操作模式。 0000：独立模式 0001：混合的同步规则+注入同步模式 0010：混合的同步规则+交替触发模式 0011：混合同步注入+快速交叉模式 0100：混合同步注入+慢速交叉模式 0101：注入同步模式 0110：规则同步模式 0111：快速交叉模式 1000：慢速交叉模式 1001：交替触发模式 注： 在ADC2和ADC3中这些位为保留位 在双模式中，改变通道的配置会产生一个重新开始的条件，这将导致同步丢失。建议在进行任何配置改变前关闭双模式。

图 13.7 ADC 工作模式

另外该寄存器第 5 位 EOCIE 用于使能转换完成中断：

位5	**EOCIE**：允许产生EOC中断 (Interrupt enable for EOC) 该位由软件设置和清除，用于禁止或允许转换结束后产生中断。 0：禁止EOC中断； 1：允许EOC中断。当硬件设置EOC位时产生中断。

图 13.8 转换完成中断允许位

如果使用查询法（等待 EOC 标志位）采集 ADC 的话需要把 EOCIE 设置为 0。

- ADC 控制寄存器 2（ADC_CR2）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								TS VREFE	SW START	JSW START	EXT TRIG	EXTSEL[2:0]			保留
								rw	rw	rw	rw	rw	rw	rw	
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
JEXT TRIG	JEXTSEL[2:0]			ALIGN	保留		DMA	保留				RST CAL	CAL	CONT	ADON
rw	rw	rw	rw	rw			rw					rw	rw	rw	rw

图 13.9 ADC 控制寄存器 2

因为我们想通过软件随时启动一次 ADC 转换，因此需要设置为软件转换，其对应的位域如下：

位22	**SWSTART**：开始转换规则通道 (Start conversion of regular channels) 由软件设置该位以启动转换，转换开始后硬件马上清除此位。如果在EXTSEL[2:0]位中选择了SWSTART为触发事件，该位用于启动一组规则通道的转换， 0：复位状态； 1：开始转换规则通道。

图 13.10 ADC 软件触发转换设置位

位19:17	**EXTSEL[2:0]**：选择启动规则通道组转换的外部事件 (External event select for regular group) 这些位选择用于启动规则通道组转换的外部事件 ADC1和ADC2的触发配置如下 000：定时器1的CC1事件　　100：定时器3的TRGO事件 001：定时器1的CC2事件　　101：定时器4的CC4事件 010：定时器1的CC3事件　　110：EXTI线11/ TIM8_TRGO事件，仅大容量产品具有TIM8_TRGO功能 011：定时器2的CC2事件　　111：SWSTART ADC3的触发配置如下 000：定时器3的CC1事件　　100：定时器8的TRGO事件 001：定时器2的CC3事件　　101：定时器5的CC1事件 010：定时器1的CC3事件　　110：定时器5的CC3事件 011：定时器8的CC1事件　　111：SWSTART

图 13.11 ADC 触发事件选择

我们先通过 EXTSEL 位域来选择 ADC 转换的触发方式，这里选择软件转换，然后就可以通过设置 SWSTART 来进行一次转换了。

ADC 采集完成之后会把数据存放到转换寄存器，转换寄存器是 32 位的，但是只使用了低 16 位用来存放 12 位转换值，那么还剩下 4 位空缺，因此还要设置 ADC 读取结果的数据对齐形式，分为左对齐和右对齐，设置对应的位域为：

位11	**ALIGN**：数据对齐 (Data alignment) 该位由软件设置和清除。参考图29和图30。 0：右对齐； 1：左对齐。

图 13.12 ADC 数据寄存器数据对齐设置

左右对齐的示意图如下：

右对齐：					11	10	9	8	7	6	5	4	3	2	1	0
左对齐：	11	10	9	8	7	6	5	4	3	2	1	0				

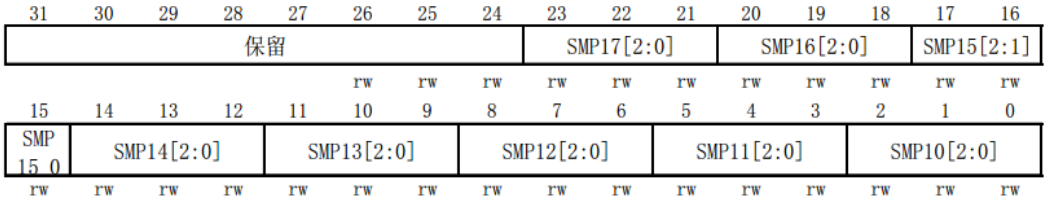
图 13.13 ADC 数据寄存器对齐方式示意图

如果在平时不使用 ADC 的时候可以让其断电，此时的功耗只有几个微安，我们可以通过该寄存器内的 ADON 位来切换：

位0	**ADON**：开/关A/D转换器 (A/D converter ON / OFF) 该位由软件设置和清除。当该位为'0'时，写入'1'将把ADC从断电模式下唤醒。 当该位为'1'时，写入'1'将启动转换。应用程序需注意，在转换器上电至转换开始有一个延迟t_{STAB}，参见图25。 0：关闭ADC转换/校准，并进入断电模式； 1：开启ADC并启动转换。 注：如果在这个寄存器中与ADON一起还有其他位被改变，则转换不被触发。这是为了防止触发错误的转换。

图 13.14 ADC 开关设置

- ADC 采样时间寄存器 1/2(ADC_SMPR1/2)



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留								SMP17[2:0]			SMP16[2:0]			SMP15[2:1]	
					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
SMP 15_0	SMP14[2:0]			SMP13[2:0]			SMP12[2:0]			SMP11[2:0]			SMP10[2:0]		
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:24	保留。必须保持为0。
位23:0	**SMPx[2:0]**：选择通道x的采样时间 (Channel x Sample time selection) 这些位用于独立地选择每个通道的采样时间。在采样周期中通道选择位必须保持不变。 000：1.5周期　　100：41.5周期 001：7.5周期　　101：55.5周期 010：13.5周期　　110：71.5周期 011：28.5周期　　111：239.5周期 注：ADC1的模拟输入通道16和通道17在芯片内部分别连到了温度传感器和V_{REFINT}。 ADC2的模拟输入通道16和通道17在芯片内部连到了Vss。 ADC3模拟输入通道14、15、16、17与Vss相连

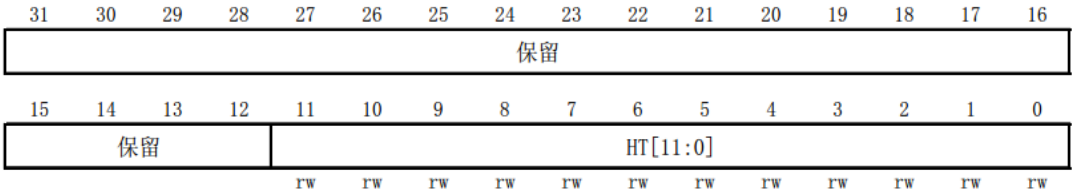
图 13.15 ADC 采样周期设置位域

这两个寄存器用于设置各个 ADC 采集通道的转换时间，STM32 的转换时间计算公式为：

$$T_{CONV} = 采样时间 + 12.5\ 个周期$$

如果此时 ADC 时钟为 14MHz，ADC 采样时间为 1.5 周期的话，整个的转换周期为 14 个周期即 1us。采样周期长采集结果就会更精确一些，但是误差差别不大，因此我们都选最短的 1.5 周期。

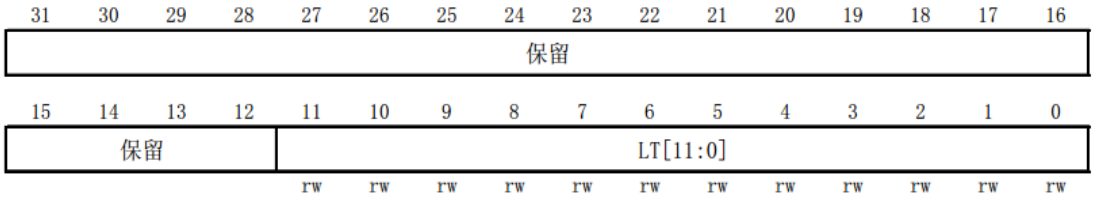
- 模拟看门狗高/低阈值寄存器（ADC_HTR/ADC_LRT）



31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

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

位31:12	保留。必须保持为0。
位11:0	**HT[11:0]**：模拟看门狗高阀值 (Analog watchdog high threshold) 这些位定义了模拟看门狗的阀值高限。

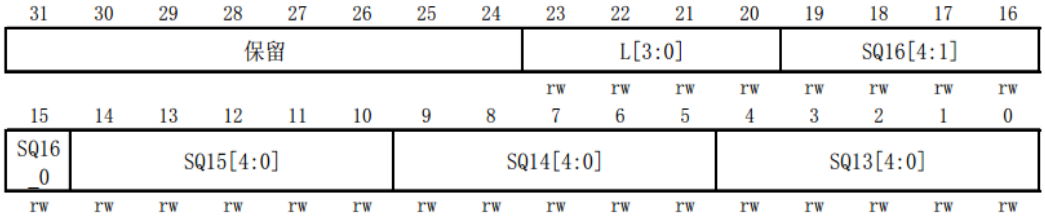


位31:12	保留。必须保持为0。
位11:0	**LT[11:0]**：模拟看门狗低阈值 (Analog watchdog low threshold) 这些位定义了模拟看门狗的阈值低限。

图 13.16 模拟看门狗高低阈值设置

该寄存器在使用模拟看门狗的时候会用到，高低阈值都存放到低 12 位中。

- ADC 规则序列寄存器 1/2/3(ADC_SQR1/2/3)
  这里我们只拿 ADC_SQR1 寄存器为例：



位31:24	保留。必须保持为0。
位23:20	**L[3:0]**：规则通道序列长度 (Regular channel sequence length) 这些位由软件定义在规则通道转换序列中的通道数目。 0000：1个转换 0001：2个转换 ..... 1111：16个转换
位19:15	**SQ16[4:0]**：规则序列中的第16个转换 (16th conversion in regular sequence) 这些位由软件定义转换序列中的第16个转换通道的编号(0~17)。
位14:10	**SQ15[4:0]**：规则序列中的第15个转换 (15th conversion in regular sequence)
位9:5	**SQ14[4:0]**：规则序列中的第14个转换 (14th conversion in regular sequence)
位4:0	**SQ13[4:0]**：规则序列中的第13个转换 (13th conversion in regular sequence)

图 13.17 ADC 规则序列寄存器设置位

在这里，要向大家介绍一下 STM32 ADC 的工作模式，它的工作模式可分为：单次转换和连续转换（ADC_CR2 中的 CON 位设置）。每次转换都会按照在 ADC_SQR1/2/3 寄存器定义的转换顺序进行转换，举个例子加深理解。

1. **单次转换**就是转换完成设置的序列后之后马上停止，类似于音乐播放器的顺序播放模式，当播放完最后一首音乐就停止播放。

图 13.18 ADC 单次转换示意

而这些转换序列就是通过 ADC_SQR1/2/3 寄存器进行设置的，一次转换有多少转换序列（多少首歌曲）通过设置 ADC_SQR1 中的 L 位域来实现（比如播放列表有 3 首歌曲，则 L 设置的就是 0010），转换序列对应的通道是由 SQx 位域进行设置的，比如我可以设置第一个转换序列是通道 12，第二个转换序列是通道 1，第三个转换序列是通道 6。

2. 连续转换

连续转换对应的就是播放设置里的列表循环，当播放完最后一首歌曲后，会继续自动播放第一首歌曲。

图 13.19 ADC 连续转换示意

上面举的例子就形象的说明了**单次/连续转换**和**转换序列数量以及通道**的关系。

另一个重要的概念是扫描模式，扫描模式控制位位于 ADC_CR1 的 SCAN 位，当扫描模式使能后，ADC 会根据规则序列寄存器中的规则序列通道长度（L[3:0]）和各通道设置的转换序列(SQx[4:0])进行轮流转换,如果是单次转换那么该组规则转换序列转换完成后即停止，

如果是连续转换，则在最后一个通道转换完成后自动切换第一个通道，以此往复，其实单次/连续转换就是扫描模式下的不同工作方式。

另外还有注入通道的转换序列寄存器（ADC_JSQR），用法和规则通道的一样。

- ADC 规则数据寄存器(ADC_DR)

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
ADC2DATA[15:0]															
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
DATA[15:0]															
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:16	**ADC2DATA[15:0]**：ADC2转换的数据 (ADC2 data) - 在ADC1中：双模式下，这些位包含了ADC2转换的规则通道数据。见11.9：双ADC模式 - 在ADC2和ADC3中：不使用这些位。
位15:0	**DATA[15:0]**：规则转换的数据 (Regular data) 这些位为只读，包含了规则通道的转换结果。数据是左对齐或右对齐，如图29和图30所示。

图 13.20 ADC 数据寄存器

虽然规则转换最多有 18 个通道，但是他们都共用一个 ADC 数据寄存器，该寄存器用于存储转换后的 12 位结果，与其类似的还有注入转换的数据寄存器（ADC_JDRx，x=1,2,3,4），只不过注入转换的寄存器是四个，即：每个注入通道分配一个数据寄存器。

接下来看一下开发板上的原理图：

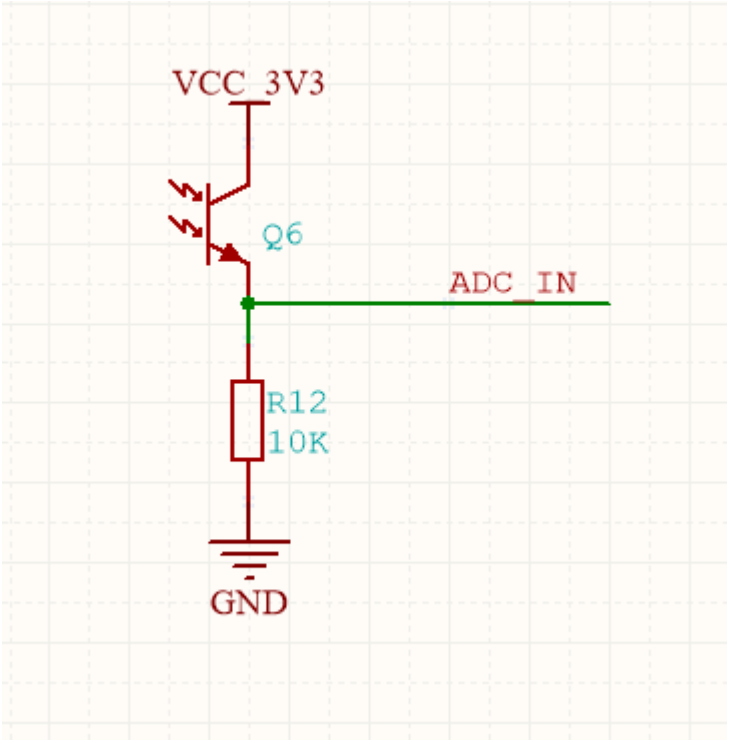


图 13.21 光线强度采集电路

Q6 是一个光电二极管，它的基极是感光材料，当光线增强时，流过 Q6 的电流就会变大，反之变小，我们可以把它当成一个恒流源，根据欧姆定律可知，电流不同作用到电阻 R12 上的电压也不同，我们通过 ADC 采集这个电压就可以得光线强度和电压的关系，在本电路中，光线越强电压越高。输出电压连接到 STM32 的 PB1 引脚：

PB1	ADC12_IN9/TIM3_CH4(9) TIM8_CH3N

图 13.22 ADC 引脚定义

从数据手册的引脚定义可知，ADC1 和 ADC2 的通道 9 共同作为 PB1 的复用功能存在，我们使用的是 ADC1_IN9 作为 ADC 的输入。

12.3 程序讲解

ADC 的初始化配置可以参考下面的步骤：

1. 初始化对应通道的 GPIO 为 ADC 输入模式
2. 配置 ADC，配置内容包括：ADC 时钟分频（官方建议不要超过 14MHz），ADC 工作模式，是否连续转换，扫描模式，数据寄存器对齐模式转换序列等。
3. 校准 ADC 并开启 ADC，建议在第一次启动 ADC 都进行校准，这样采集的数据也会更加精确（经过测试不进行校准和加校准所采集的结果误差还是很大的）。

因为青柚 ZERO 的扩展板上控制冷光灯的引脚是 PA15，PA15 在上电之后默认是 JTAG 模式下的 JTDI 引脚，该引脚为上拉，因此会导通 NPN 三极管使其点亮 LED，会对本实验的光照强度采集造成影响，因此在初始化 ADC 的时候还要将 PA15 复用功能重映射到该引脚上，可能大家会有疑问，PA15 不应该是主功能吗，对于 STM32 来说，其 JTAG 上电后和其他 IO 功能不同，如下图：

A8	38	A6	50	77	29	PA15	I/O	FT	JTDI		TIM2_CH1_ETR PA15/SPI1_NSS
B9	-	B7	51	78	-	PC10	I/O	FT	PC10		USART3_TX

图 13.23 GPIO 复用

PC10 默认主功能就是 GPIO，但是 PA15 默认主功能是 JTDI，如果想要使用 PA15 的 GPIO 功能，就必须重映射 PA15。如下图：

复位后，JTAG引脚被置于输入上拉或下拉模式：
- PA15：JTDI置于上拉模式
- PA14：JTCK置于下拉模式
- PA13：JTMS置于上拉模式
- PB4： JNTRST置于上拉模式

图 13.24 JTAG 模式引脚复位后默认状态

重映射代码如下：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_ADC1 | RCC_APB2Periph_GPIOA | RCC_APB2Periph_AFIO, ENABLE);
GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);                    //重映射JTAG引脚,只使用SWD模式
```

图 13.25 重映射设置函数

红色框选部分是我们要开启 GPIOA 时钟以及复用功能时钟，第二行就是引脚重映射配置函数，该配置函数参数为：禁止 JTAG 使能 SWD（因为 STM32 是支持 JTAG+SWD 下载仿真的），如果哪位调皮的小伙伴把参数改为 **GPIO_Remap_SWJ_Disable（JTAG+SWD 全部禁止）**，那就意味着你只能通过 ISP 的下载方式把该设置改回来，这里再顺便说一下我自己的一个经历，如果你使用 STLink（其他仿真器也一样）给一个在睡眠模式和运行模式频繁切换的 STM32 单片机下载程序时，下载成功完全取决于你的运气，因为在睡眠模式下 STLink 下载不了程序，所以只能赶在运行模式给 STM32 下载才行，这时候要么继续期待美好的事情即将发生要么只能无奈的使用 ISP 下载了。

然后我们设置 GPIO 初始化属性并生效设置：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;                                  //设置对应引脚
GPIO_Init(GPIOA, &GPIO_InitStructure);                                      //设置生效
GPIO_ResetBits(GPIOA, GPIO_Pin_15);                                         //默认低电平
```

图 13.26 PA15GPIO 设置

接下来回到主题，我们先设置 ADC 使用的 GPIO 引脚——PB1。

```
/*******************配置PB1*************************/
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_1;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AIN;                               //模拟输入引脚
GPIO_Init(GPIOB, &GPIO_InitStructure);
```

图 13.27 设置 PB1 为 ADC 输入模式

这和设置按键时差不多，只不过此时是 ADC 输入，因此不需要指定输出工作频率。然后，初始化 ADC：

1. 设置 ADC 工作频率

```
RCC_ADCCLKConfig(RCC_PCLK2_Div8);                                           //设置ADC时钟分频，此时ADC时钟频率为72/8=9MHz
```

图 13.28 配置 ADC 输入时钟为 8 分频

该函数就是将系统时钟分频后供 ADC 使用，分频值可以设置（2/4/6/8 分频），但是 STM32 参考手册建议，ADC 时钟频率不要超过 14MHz，我们这里设置的是最大分频，分频后 ADC 时钟频率为 9MHz，满足要求。

2. 配置 ADC 初始化结构体变量并使能初始化设置

```
/*******************配置ADC*************************/
ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;                    //配置为独立工作模式
ADC_InitStructure.ADC_ScanConvMode = DISABLE;                         //设置ADC工作在单通道
ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;                   //设置ADC为单次转换
ADC_InitStructure.ADC_ExternalTrigConv = ADC_ExternalTrigConv_None;   //失能外部触发，使用软件人为触发
ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;                //设置ADC数据寄存器右对齐
ADC_InitStructure.ADC_NbrOfChannel = 1;                               //规则转换通道数，我们只使用一个
ADC_Init(ADC1, &ADC_InitStructure);                                   //生效ADC设置

ADC_Cmd(ADC1, ENABLE);                                                //设置ADC_CR2中的ADON位，使其给ADC1上电
```

图 13.29 ADC 初始化结构设置

本例程只使用了一个 ADC 通道，并通过软件来启动 ADC 转换，因此就要设置 ADC 为独立模式，禁止扫描模式，禁止连续转换模式。最后通过 ADC_Cmd(ADC1, ENABLE)函数给 ADC 外设上电，接下来要使用 ADC_RegularChannelConfig()函数来设置对应 ADC 的对应通道参数，这些参数包括转换序列和采样周期：

```
ADC_RegularChannelConfig(ADC1, ADC_Channel_9, 1, ADC_SampleTime_239Cycles5);
```

图 13.30 ADC 规则通道配置

这里只用到了 ADC1_IN9，因此将该通道设置为第一转换序列且采样周期为 239.5 个 ADC 时钟周期。到这里并没有结束，虽然现在 ADC 跑起来了，但是 ADC 还没有进行使用前的校准，我们经过测试，如果不进行校准，误差会在几十到一百不等。因此需要校准，校准配置函数如下：

```
ADC_ResetCalibration(ADC1);                                   //复位ADC1校准
while (ADC_GetResetCalibrationStatus(ADC1));                  //等待复位校准完成
ADC_StartCalibration(ADC1);                                   //开始ADC1的校准
while (ADC_GetCalibrationStatus(ADC1));                       //等待校准完成
```

图 13.31 ADC 校准

接下来是 ADC 采集部分：

```
u16 getConvValue(void)
{
    ADC_SoftwareStartConvCmd(ADC1, ENABLE);          //开启软件转换，置位ADC_CR2的SWSTART位
    while (!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC)); //等待转换完成
    return ADC_GetConversionValue(ADC1);             //获取ADC转换结果
}
```

图 13.32 单次 ADC 采集

步骤很简单，先是使用 ADC_SoftwareStartConvCmd()函数触发 ADC 对应通道转换，紧接着等待 ADC 转换完成，最后将 ADC 的值通过 ADC_GetConversionValue()函数获取并返回。

对于取平均值的 ADC 采集同理该函数：

```
u16 getConvValueAve(u8 counts, u32 xus)
{
    u8 i;
    u32 sum = 0;                //累加值临时存放变量

    for (i = 0; i < counts; ++i)
    {
        sum += getConvValue(); //采样结果累加
        Delay_us(xus);
    }

    return sum / counts;        //取平均值并返回
}
```

图 13.33 多次 ADC 采集取平均值

ADC 取平均值函数无外乎就是加了循环次数和相邻两次采集的时间间隔。
最后主函数初始化各外设，并执行用户业务代码：

```
int main(void)
{
    /*各外设初始化*/
    initADC();
    initUART();
    initLED();
    initSysTick();

    while (1)
    {
        printf("Lux is %d\n",getConvValueAve(5,10000)); //采集周期50ms
        toggleLED();
        Delay_ms(800);//800ms刷新一次
    }
}
```

图 13.34 主函数

本实验例程实现的是每隔 800ms 采集一次平均值，并打印到串口调试软件（波特率 9600），串口调试结果如下：

- 黑暗条件下（使用取平均值函数）

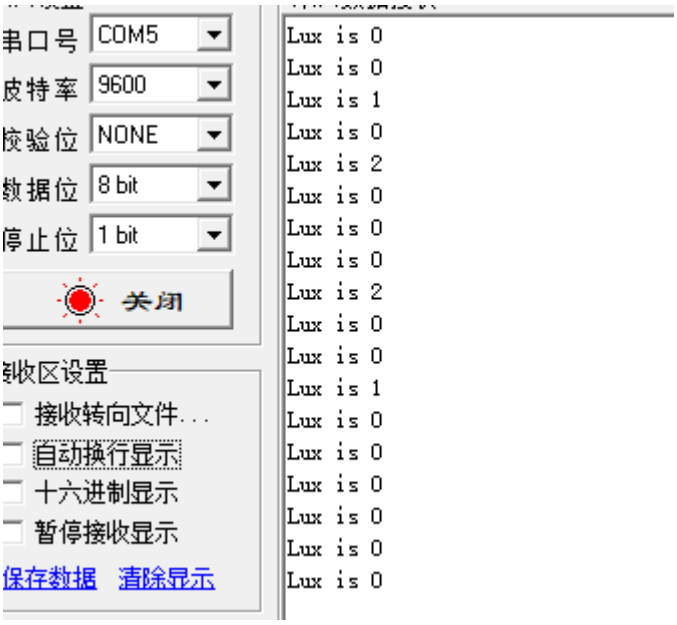


图 13.35 黑暗测试结果_1

- 黑暗条件下（不使用取平均值函数）

图 13.36 黑暗测试结果_2

- 强光条件下（使用取平均值函数）

图 13.37 强光测试结果_1

- 强光条件下（不使用取平均值函数）

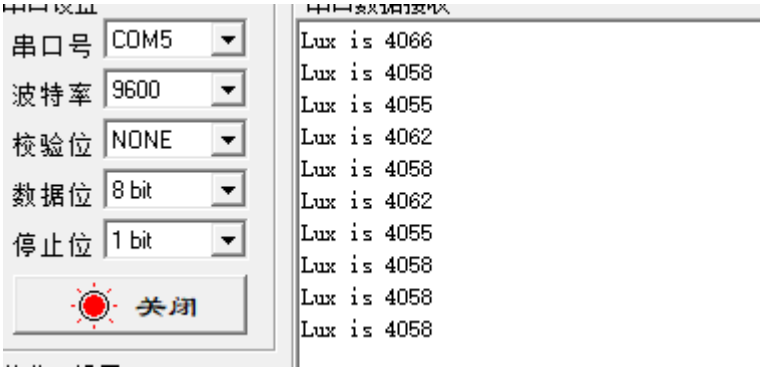


图 13.38 强光测试结果_2

所以这就引出是否使用取平均值方式的两个应用场景：

1. 用户对采集频率要求不高，但是要保证准确稳定，比如采集电池电压，此时使用取平均值法最合适；
2. 如果用户想要处理的是实时响应数据，并用于数字信号处理算法上，那么此时不取平均值最合适。

到这里 ADC 部分就讲解完成了，大家可以配合我们的例程，试着去改一些参数来验证自己的想法。

附：下图是设备运行一夜，每隔 30 分钟采集一次环境亮度的实验结果：

```
Lux is 3
Lux is 0
Lux is 0
Lux is 1
Lux is 0
Lux is 0
Lux is 1
Lux is 1
Lux is 1
Lux is 0
Lux is 1
Lux is 21
Lux is 24
Lux is 131
Lux is 179
Lux is 232
Lux is 287
```

图 13.39 长时间测试结果

第十四章 使用 ADC 实现虚拟示波器

14.1 项目要求

通过使用 ADC 并配合 Data Scope 虚拟示波器软件实现上位机实时显示此时环境亮度的功能。

注：本章用到核心板+扩展板，对应例程 8。

14.2 原理讲解

示波器是电子工程师最常用的调试仪器之一，它可以实时的显示信号波形，频率，幅值等参数，通过示波器可以让看不见的电信号“现形”。

示波器目前分为两种，一种是采集显示一体的（以泰克示波器为例）：

图 14.1 泰克示波器示意

另外一种就是采集和显示分离的，一般显示是通过上位机软件来完成，这就是我们今天的主角，虚拟示波器，当然 STM32 的 ADC 采集速度还远远不能达到专业示波器的水平，但是对于平时开发来说已经足够了。目前市面上的虚拟示波器软件有很多，我们使用的是 Data Scope，它支持最多 10 通道浮点数信号显示，同时还有很多扩展功能，比如：3D 立体显示，地平面显示等，这些功能可以用到航模调试上面，极大地方便了我们的开发。

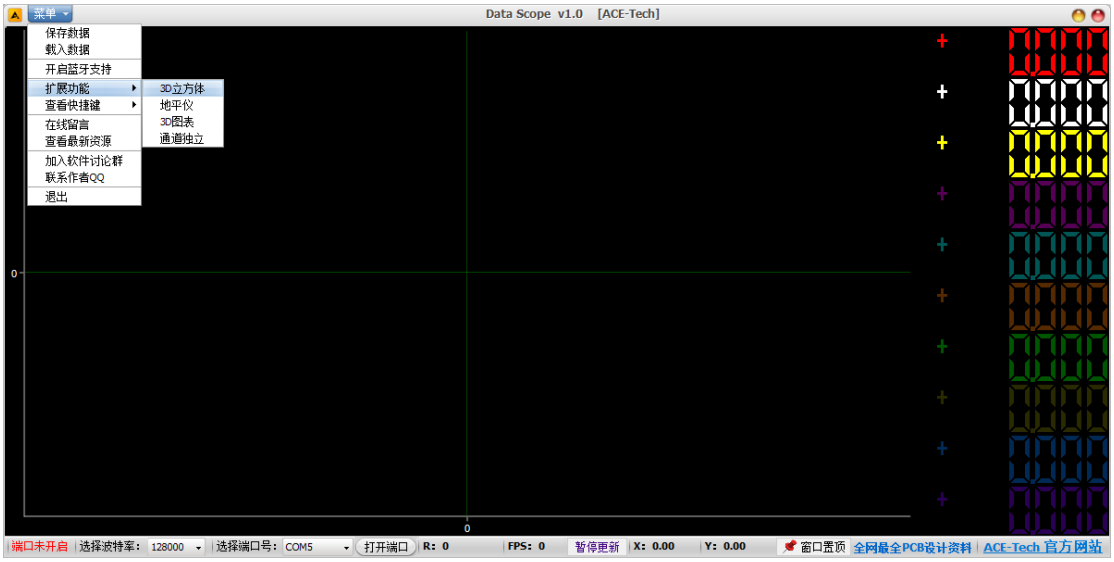


图 14.2　Data Scope 操作界面

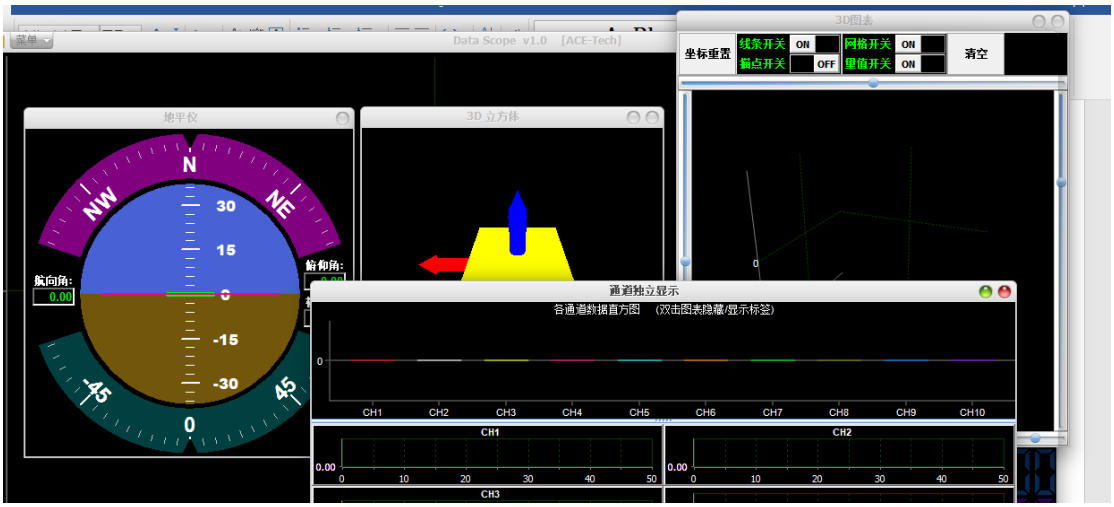


图 14.3　Data Scope 扩展功能

虚拟示波器的原理是上位机软件通过 UART 端口来监听下位机发送过来的事先约定好的数据帧格式，一般一帧数据要有一个数据帧头，它可以是任何一个有意义的符号，比如字母、数字、标点符号等，而且为了保证数据的有效性，通常还会在一帧数据的后面加上校验和。

Data Scope 已经为我们提供了数据通信函数和帧格式，我们来学习一下。

- 帧格式

  通信的一帧数据由一个包含 42 个成员的无符号 8 位整形数组组成，该数组用于存放 10 通道的数据以及一些标识参数。

```
unsigned char DataScope_OutPut_Buffer[42];     //串口发送缓冲区
```

图 14.4　发送 buffer

数据帧具体内容如下：

图 14.5　虚拟示波器通信协议帧

蓝色块帧数据头，内容固定为$，占用一字节

通道 1-10 的数据，由于每个通道都支持浮点型，所以为每个通道分配了 4 个字节用于存储一个浮点型，该部分最多占用 40 字节（10 通道全用），最少占用 4 字节（只用通道 1）。

最后数据长度紧跟最后一个通道数据的后面，它用于告诉我们要发送多少字节给上位机。

举个例子：如果此时只用到了通道 1，那么数据就应该是：

$ + 4 字节通道一数据 + 数据长度，此时 42 个元素的数组我们只用到了 6 个，但是发送给上位机的有效数据只有前 5 个字节。

14.3 程序讲解

```
extern void DataScope_Get_Channel_Data(float Data,unsigned char Channel)
{
    if ( (Channel > 10) || (Channel == 0) )
    {
        return;  //通道个数大于10或等于0，直接跳出，不执行函数
    }else
    {
        switch (Channel)
        {
            case 1:  Float2Byte(&Data,DataScope_OutPut_Buffer,1); break;
            case 2:  Float2Byte(&Data,DataScope_OutPut_Buffer,5); break;
            case 3:  Float2Byte(&Data,DataScope_OutPut_Buffer,9); break;
            case 4:  Float2Byte(&Data,DataScope_OutPut_Buffer,13); break;
            case 5:  Float2Byte(&Data,DataScope_OutPut_Buffer,17); break;
            case 6:  Float2Byte(&Data,DataScope_OutPut_Buffer,21); break;
            case 7:  Float2Byte(&Data,DataScope_OutPut_Buffer,25); break;
            case 8:  Float2Byte(&Data,DataScope_OutPut_Buffer,29); break;
            case 9:  Float2Byte(&Data,DataScope_OutPut_Buffer,33); break;
            case 10: Float2Byte(&Data,DataScope_OutPut_Buffer,37); break;
        }
    }
}
```

图 14.6　格式化发送 buffer

作用是把一个浮点型的数添加到对应通道的数据区域，对于通道 1 就是 buffer 的 1-4，对于通道 2 就是 buffer 的 5-8。Float2Byte()函数用于把一个浮点型的数切成 4 份发给对应 buffer 的位置：

```
extern void Float2Byte(float *target,unsigned char *buf,unsigned char beg)
{
    unsigned char *point;
    point = (unsigned char*)target;    //得到float的地址
    buf[beg]   = point[0];
    buf[beg+1] = point[1];
    buf[beg+2] = point[2];
    buf[beg+3] = point[3];
}
```

图 14.7　浮点型转整型函数

最后一个函数是用于获取有效部分的 buffer 数据并给 buffer 添加帧数据头：

```
extern unsigned char DataScope_Data_Generate(unsigned char Channel_Number)
{
    if ( (Channel_Number > 10) || (Channel_Number == 0) ) //通道个数大于10或等于0，直接跳出，不执行函数
    {
        return 0;
    }else
   {
        DataScope_OutPut_Buffer[0] = '$';  //帧头

        switch(Channel_Number)
        {
            case 1:   DataScope_OutPut_Buffer[5]  =  5; return  6; break;
            case 2:   DataScope_OutPut_Buffer[9]  =  9; return 10; break;
            case 3:   DataScope_OutPut_Buffer[13] = 13; return 14; break;
            case 4:   DataScope_OutPut_Buffer[17] = 17; return 18; break;
            case 5:   DataScope_OutPut_Buffer[21] = 21; return 22; break;
            case 6:   DataScope_OutPut_Buffer[25] = 25; return 26; break;
            case 7:   DataScope_OutPut_Buffer[29] = 29; return 30; break;
            case 8:   DataScope_OutPut_Buffer[33] = 33; return 34; break;
            case 9:   DataScope_OutPut_Buffer[37] = 37; return 38; break;
            case 10:  DataScope_OutPut_Buffer[41] = 41; return 42; break;
        }
  }
    return 0;
}
```

图 14.8　获取有效数据长度

我们来看一下主函数如何调用：

```
DataScope_Get_Channel_Data(getConvValue()/10.0, 1 ); //将采集到的数据赋值给发送buffer
counts = DataScope_Data_Generate(1);                 //获取有效buffer长度

for(i=0;i<counts;++i)                                //将buffer中的数据发送到Data Scope
{
    sendByte(USART1,DataScope_OutPut_Buffer[i]);
}
toggleLED();
Delay_ms(20);                                        //50帧更新速度
```

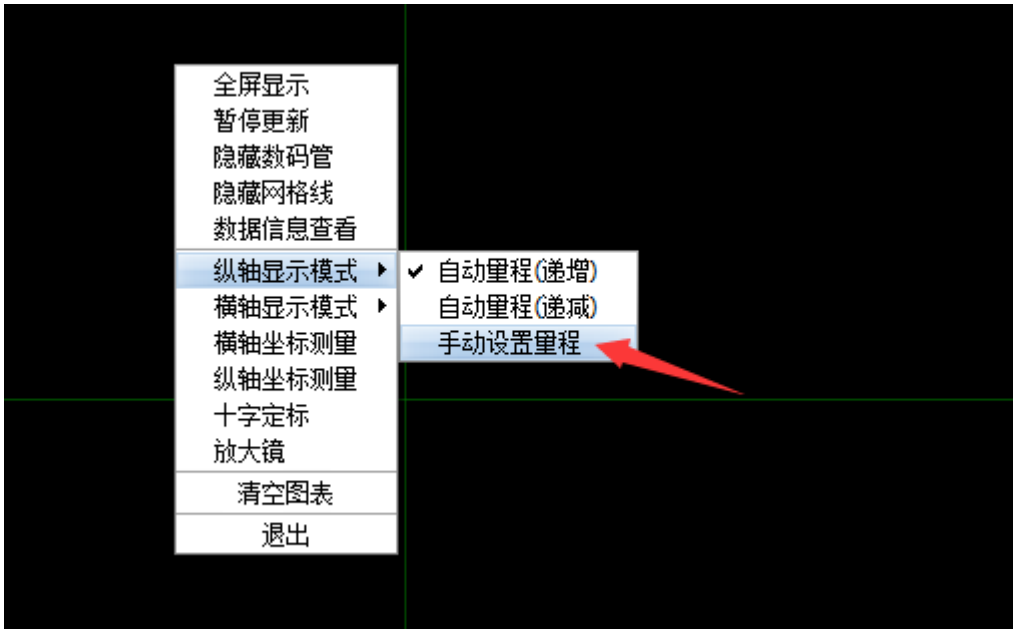
图 14.9　主函数

从程序上可以看出，我们每隔 20ms 就对 ADC 进行一次采集，并将采集结果除 10.0（减小 ADC 表示范围，更适合显示）转换为浮点数赋给通道 1 对应的 buffer，然后调用：DataScope_Data_Generate(1)将数据头加到发送 buffer 并获取有效的发送数据大小。最后我们一字节一字节的把 buffer 中的数据发送出去。这里顺便提一下数据类型转换，因为

getConvValue()返回的是一个整形的数据，对于计算机而言，一个整形的数据除一个整形的数据其结果是整形，它会截掉小数部分，如果想要得到精确的小数部分，则需要保证除数或者被除数二者至少有一个是浮点型，因此这就是为什么除 10.0 的原因。

接下来打开软件（以管理员身份运行）设置虚拟示波器的显示参数（主界面鼠标右键）：

- 设置纵坐标



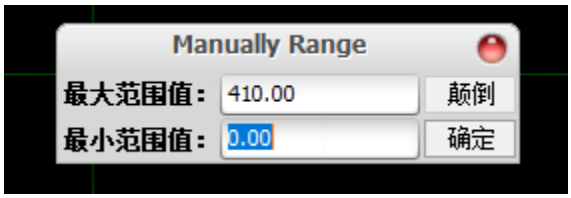


图 14.10 纵坐标设置

设置该范围的原因是，12 位 ADC 它的量程是 0-4096，除 10 之后就是 0.0-409.6，这里我们取整。

- 设置横坐标

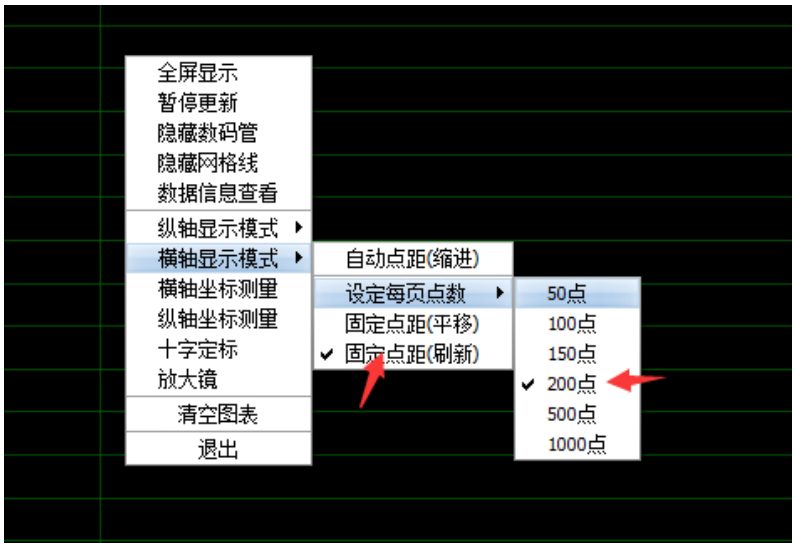


图 14.11 横坐标设置

横坐标还有很多显示模式，大家可以尝试一下，我这里设置的是最适合观看数据的选项。

- 设置 UART

图 14.11 横坐标设置

端口号会自动识别，每台计算机可能不同，我这里是 COM5，最后打开串口。我通过调光台灯来改变此时环境亮度，可以看到波形随着亮度发生了变化：

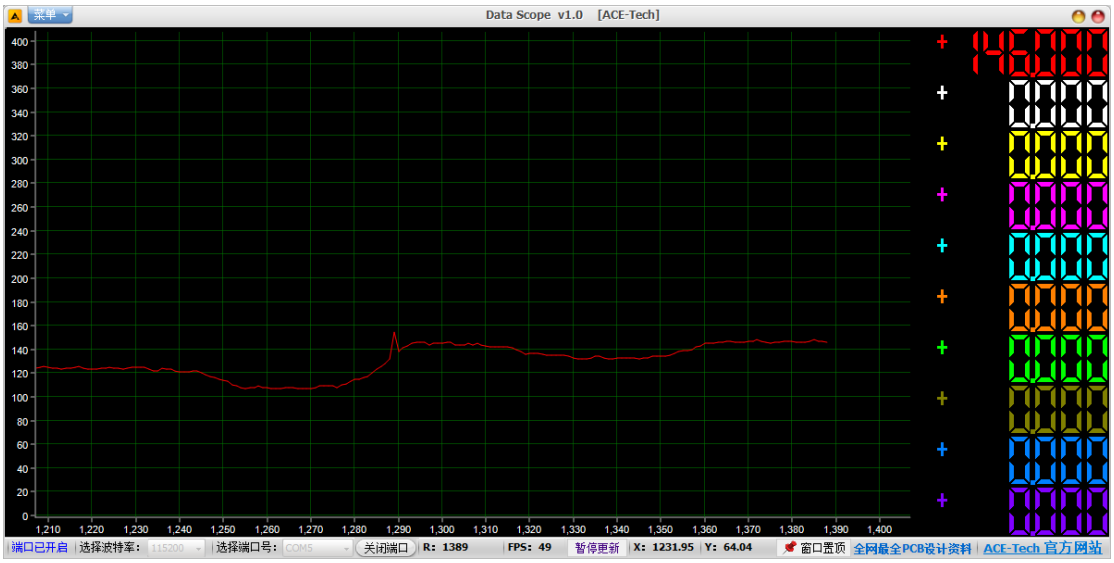


图 14.12 测试效果_1

使用手机手电筒功能中的 SOS 闪烁，得到下面的波形：

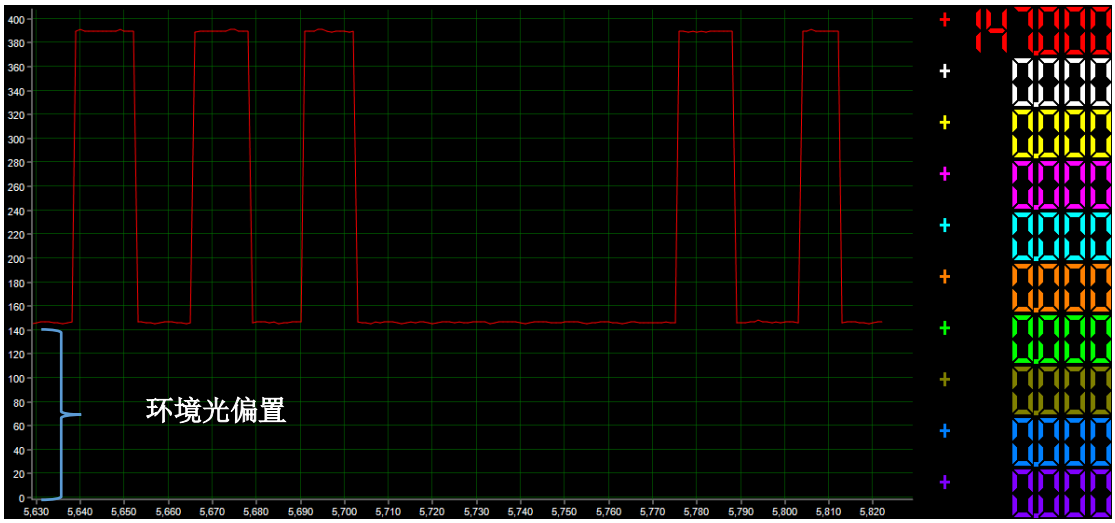


图 14.13　测试效果_2

之所以存在环境光偏置的原因是，在有稳定光源的环境下使用手电筒进行的实验。该偏置就是稳定光源的亮度。

对于本实验的 50Hz 刷新速度来说可以查看一些实时性要求不高的波形，比如电池电压，环境亮度等等，另外建议大家不要把刷新速度调节太快，这会造成上位机还未处理完上一次数据的绘制，下一次数据就又发过来了，导致上位机无波形。

可能有人会问了，如果我要测量的信号最大值大于 3.3V 怎么办，比如 5V、10V 等等，最简单的办法就是利用电阻分压，除此之外还应该在输入信号和 ADC 引脚之间加一个跟随器，如下图：

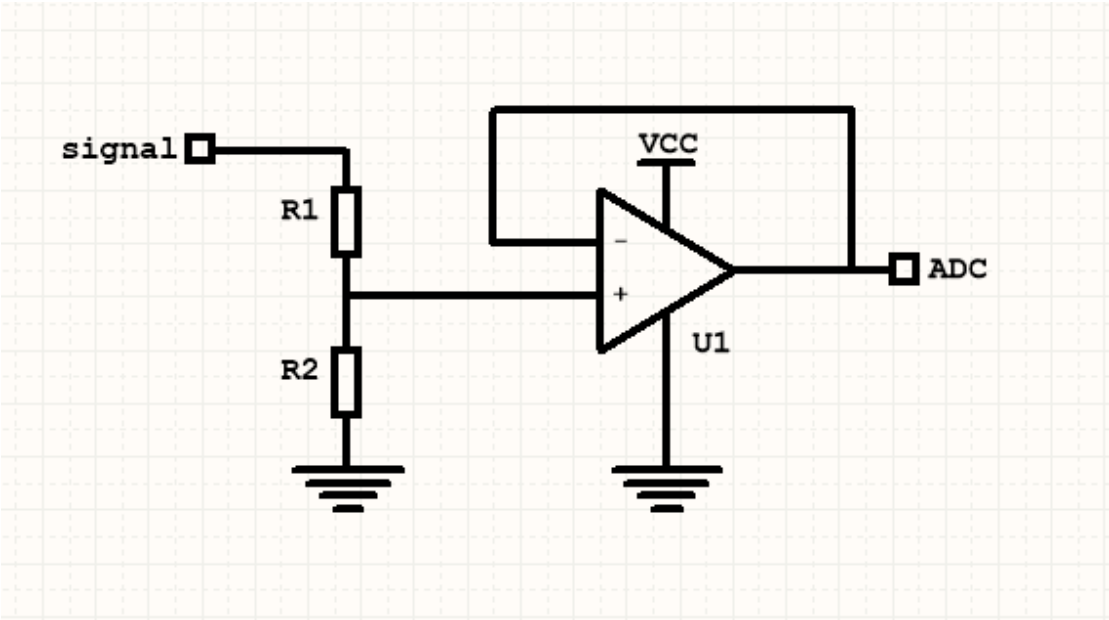


图 14.14　推荐输入处理电路

电阻 R1 和 R2 可以对输入的信号进行分压，使其输出给单片机的最大值等于 3.3V，分压公式为：

$$V_{ADC} = V_{in} * R2/(R1+R2)$$

其中 V_{ADC} 为传给单片机的输入信号，V_{in} 为外部输入信号。

我们单片机的 ADC 在采集信号时或多或少都会吸收一些来自信号的电流，这将会导致所测量的结果出现误差，为了排除这种误差，就引入了跟随器。跟随器是同相放大器的一种特殊应用，它的电压增益为 1，即输入输出电压之比为 1，它的特点是输入输出信号一致，输入阻抗非常大，输出阻抗非常小，因此常用于作为信号隔离。如果感兴趣，大家可以利用手边的资源搭建一个如上所述带档位和隔离的输入信号处理电路（运放可以可以使用 LM358 或者 NE5532）。

现在，我们来介绍一下虚拟示波器的另外一个用法——显示数据的变化趋势。有时，我们希望能够直观的查看某一个变量的变化趋势，当然我们可以通过串口打印出来或者在调试界面查看变量，但还是很难察觉它的变化过程，因此可以使用虚拟示波器来观察该变量。

这里来验证一下对一个无符号 8 位整型的变量自加，观察当超过 255 时它会发生什么。下面是主函数：

```
while (1)
{
    DataScope_Get_Channel_Data(j++, 1 );
    counts = DataScope_Data_Generate(1);              //获取有效buffer长度

    for(i=0;i<counts;++i)                             //将buffer中的数据发送到Data S
    {
        sendByte(USART1,DataScope_OutPut_Buffer[i]);
    }
    toggleLED();
    Delay_ms(20);                                     //50帧更新速度
}
```

图 14.15 整型溢出测试程序

j 是初始值为 0 的无符号 8 位整型变量，我们对其自加，并把其值发送到虚拟示波器显示，效果如下：

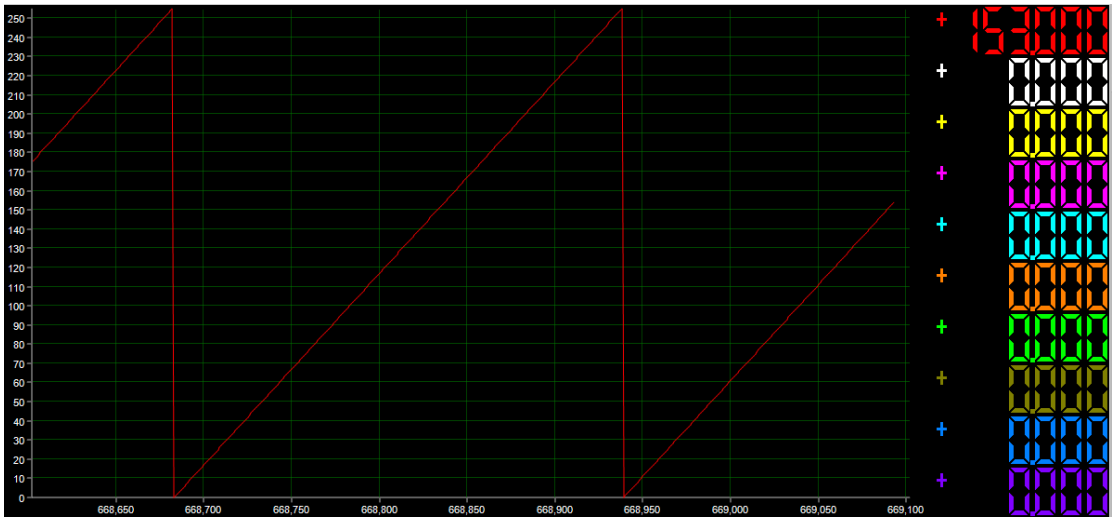


图 14.16 测试效果_3

这说明无符号整型变量溢出后会自动清零。下面是有符号整型变量的递加结果：

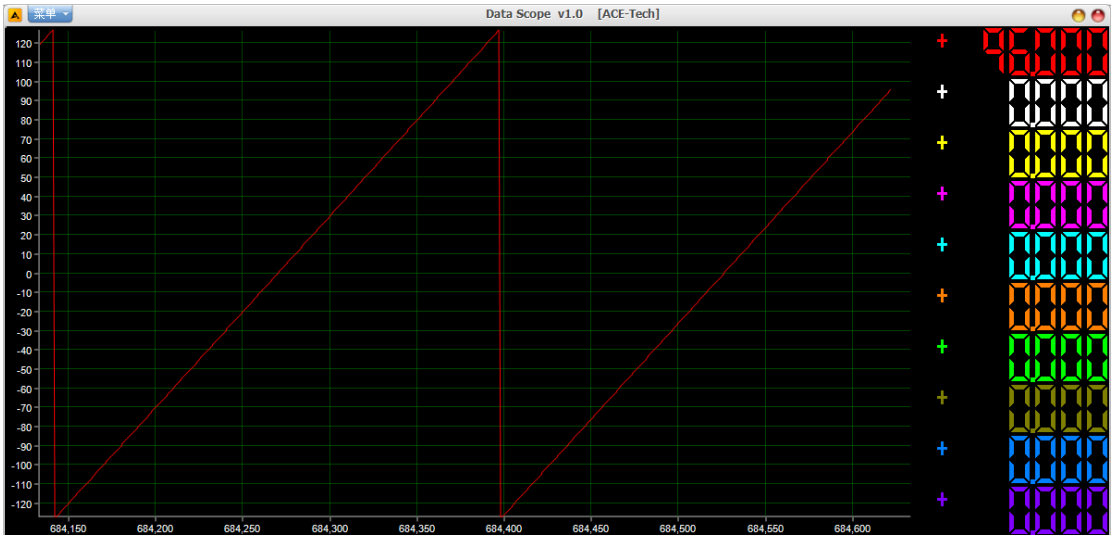


图 14.17 测试效果_4

到这里本章的内容就结束了，大家可以继续发挥想象，发掘虚拟示波器的其他用法。

第十五章 IIC 实验_驱动 OLED 屏幕

15.1 项目要求

通过 IIC 驱动 0.96 寸 OLED 屏幕。在开机时显示风媒 LOGO，在工作状态下显示按下的键值、当前光线强度、上位机发送过来的串口数据。

注：本章用到核心板+扩展板，对应例程 9。

15.2 原理讲解

15.2.1 详解 IIC 协议

IIC 是飞利浦公司在 80 年代发明的一种二线制半双工同步串行通信协议，我们之前讲的 UART 是半双工异步串行通信协议，因此 IIC 是带有一条时钟线的，用于同步数据线上的数据，时钟线类似于一个勺子，它将待搬运的东西一勺一勺（一个时钟周期）的搬到目的地。

IIC 通信协议和其他通信协议的功能一样（如 UART），他们都是以某种事先约定好的时序进行数据传输的通信协议，只不过不同通信协议之间的约定不同而已。

IIC 一般应用于芯片之间的通信，它的传输距离短，但其好处是 IIC 支持一主多从的挂载方式，因此主机和多个从机之间的通信线只要两条就够了，示意图如下：

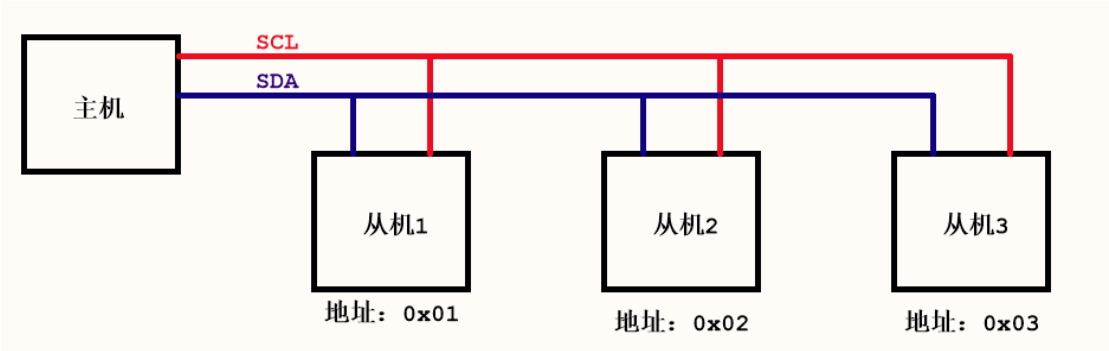


图 15.1 IIC 主从结构

其中 SCL 为时钟线，SDA 为数据线，IIC 的主机通过各个从机的地址选通对应从机。总线上从机的数量受总线最大电容 400pf 的限制，当然大多数情况下这种限制我们是遇不到的。下面作者带领大家揭开 IIC 通信协议的神秘面纱。

IIC 主从机之间通讯步骤如下：

1. 主机发送一个起始信号通知各从机就位
2. 主机发送从机地址和读写标志位
   从机地址和读写标志位一共占用 8 位，地址占用高 7 位，读写标志位占用最低位，

如下：

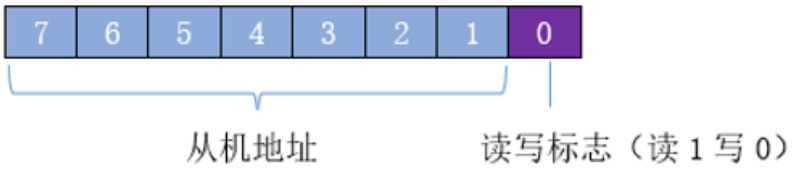


图 15.2　IIC 地址字节划分

3. 从机给主机回复响应
4. 如果是写模式，主机发送一字节数据等待从机响应，主机收到响应之后如果还有数据要发就继续发送第二段数据等待响应…直到发送完成；
   如果是读模式，此时主机读取从机发来的数据，并给从机响应，如果从机还有数据要发送（接着汇报第二段），主机接着读取然后发送响应给从机…
5. 主机给从机一个停止信号
   *注：以上所说的响应，包括 ACK 和 NACK

下面是写时序：

图 15.3　IIC 写操作字节流

下面是读时序：

图 15.4　IIC 读操作字节流

图中的灰色部分为从机的动作，彩色部分为主机动作。可以看到在写时序上，主从之间保持着一发一答的规律，主机发从机答，主机对整个发送过程有完全的控制权限。在读时序上，主从之间也同样保持一发一答的规律，在主机选择完对应从机后，从机发主机答，最后主机发送 NACK 告诉从机就发到这里，然后主机发起停止对话操作。在很多时候我们还可以对上面的时序进行拼接，即两个或者多个读/写时序拼在一起，这种情况下两个相邻时序之间没有停止信号，如下图：

图 15.5　IIC 复合操作字节流

再来看一下具体的时序：

- 起始信号

  在时钟线 SCL 高电平期间数据线 SDA 发生下降沿跳变产生起始信号

- 结束信号

  在时钟线 SCL 高电平期间数据线 SDA 发生上升沿跳变产生停止信号
  如下图高亮部分：

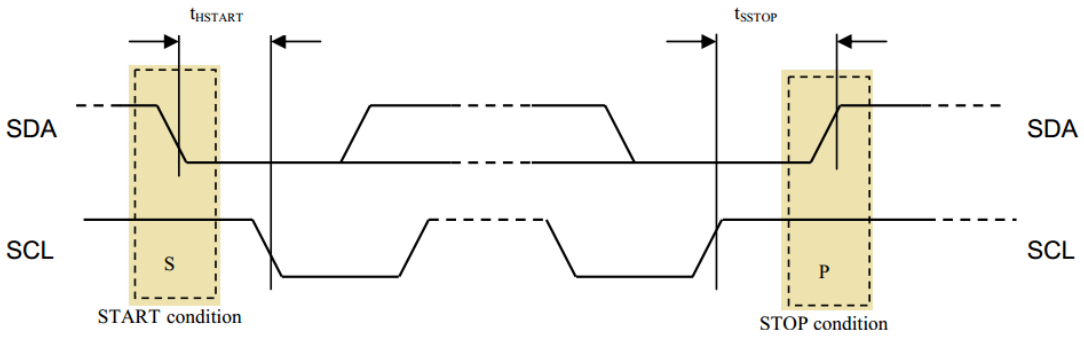


图 15.6　IIC 起始/结束时序图

- 应答信号

  在时钟线 SCL 为高电平期间数据线 SDA 保持低电平为应答信号

- 非应答信号

  在时钟线 SCL 为高电平期间数据线 SDA 保持高电平为非应答信号

  如下图高亮部分：

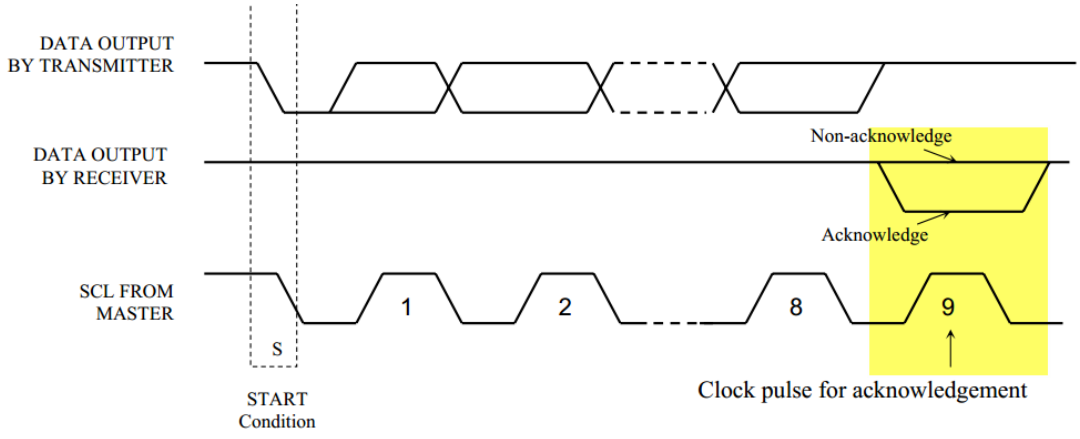


图 15.7　IIC 响应时序图

- 数据信号

  在数据传输期间，时钟线 SCL 为高电平期间，如果数据线 SDA 为高电平则代表二进制 1，同理，时钟线 SCL 为高电平期间，如果数据线 SDA 为低电平则代表二进制 0。

另外数据线 SDA 上的二进制数据要在时钟线 SCL 为低电平期间发生改变。

传输时序如下图：

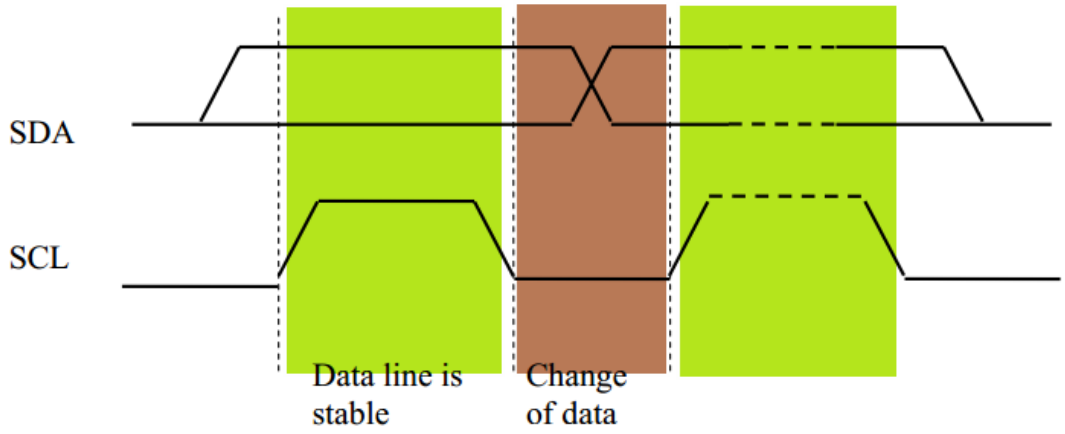


图 15.8　IIC 数据传输时序图

绿色高亮部分是 SDA 数据有效期，褐色部分是数据改变期。
其实 IIC 就是按照人类交流逻辑来定义的，建议大家多看几遍加深理解。

15.2.2 浅谈 OLED 屏幕

在嵌入式开发时，最常用的显示屏有五种：
第一种是 TFTLCD，如下图，它的特点是屏幕可以做到很大，性价比高，而且色彩丰富，适合显示一些视觉方面的内容，比如手机屏幕，笔记本屏幕等。

图 15.9　TFTLCD 示意图（图片来源网络）

第二种是字符液晶屏，如 LCD1206（12*6 的像素），LCD12864（128*64 像素），如下图，其特点是单色，像素粗糙，但是价格低廉，体积小，适合显示一些数据用，常用于仪器

仪表。

图 15.10 LCD1602/LCD12864 示意图（图片来源优信电子）

第三种是段码液晶，其特点是价格便宜，功耗低，可以作为数码管的替代品，在实际应用中一般需要定制来满足项目要求，像家中的冰箱或者空调遥控器等都会看到它的身影，段码液晶如下图：

图 15.11 段码液晶示意图（图片来源网络）

第四种是数码管，相信这个大家一定很熟悉，在很多电影里主角身上绑的定时炸弹一般都用数码管来显示倒计时（记得成龙有一部电影里面就有定时炸弹的一个片段，放大后发现是一个 STC89C52 单片机核心板+数码管做的道具），它的特点是价格非常低，几毛钱一片，但是驱动电路复杂，需要的引脚很多，而且如果用单片机来驱动很占用单片机的 CPU 资源，通常使用专门的 LED 驱动芯片如 TM1628 或者使用移位寄存器进行驱动。

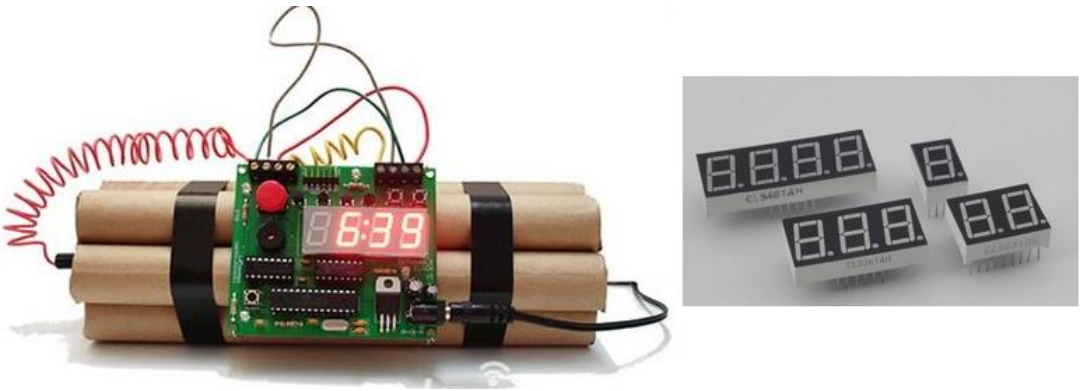


图 15.12 数码管示意图

第五种也是我们今天的主角，OLED 显示屏，现在流行的高端显示器和手机屏幕都是用的 OLED 屏幕，其实它的内部是由非常多的小 LED 灯组成的，因此它是自发光屏幕，它的优点是像素高，色彩还原度好，但是缺点也很明显，就是制造困难，所以它的价格会比其他同尺寸的屏幕要高。如下图：

图 15.13 OLED 示意图（图片来源优信电子）

当然还有墨水屏，但是其价格更高，这里不过多讲解，墨水屏如下图：

图 15.14 三色墨水屏示意图（图片来源优信电子）

我们开发板上使用的是 0.96 寸 OLED 屏幕，分辨率为 128*64，可显示图片，字符，中西方文字等。其内部使用 SSD1306 驱动屏幕，SSD1306 支持 6800/8000 并口协议也支持 3 线或 4 线 SPI 以及两线的 IIC，其中使用并口的效果最好，它可以达到最大的刷新速度，适合显示一些低帧率的动画，本例程使用的是它的 IIC 通信。SSD1306 通过 BS[2:0]引脚的电平状态来选择通信协议，如下：

SSD1306 Pin Name	I^2C Interface	6800-parallel interface (8 bit)	8080-parallel interface (8 bit)	4-wire Serial interface	3-wire Serial interface
BS0	0	0	0	0	1
BS1	1	0	1	0	0
BS2	0	1	1	0	0

图 15.15 SSD1306 通信接口配置选项

SSD1306 为我们提供了 128*8 字节的显存空间，即 128*64 个位，每个位对应着屏幕上的一个像素点。内存映射分布如下所示：

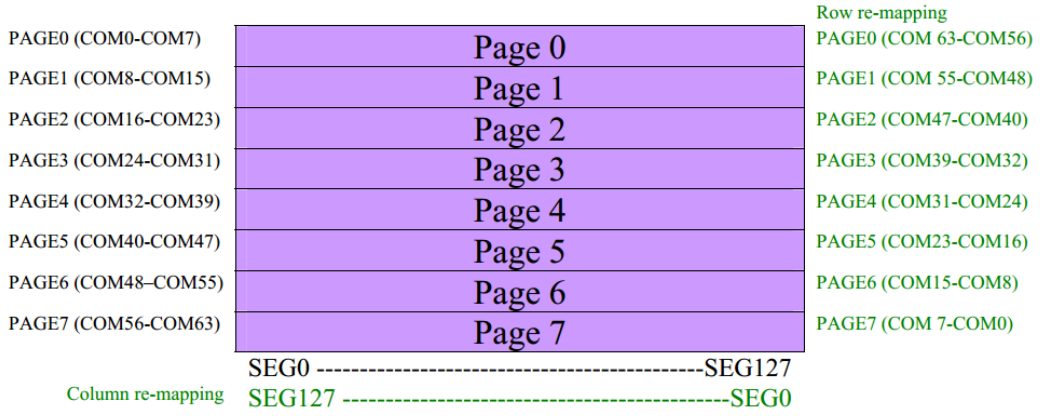


图 15.16　SSD1306 位段映射图

其中 128 个 SEGx 对应的就是横向的 128 列，64 个 COMx 对应就是纵向的 64 行，只要提供具体的行、列数值可以精确控制屏幕上的某一点。另外 SSD1306 又将 64 个 COM 每 8 个 COM 分为一组，称之为一页，即我们向某一页写入的 8 位数据会分布在某一个列（SEGx）的 COMx 到 COMx+8 这 8 位，如下图高亮部分：

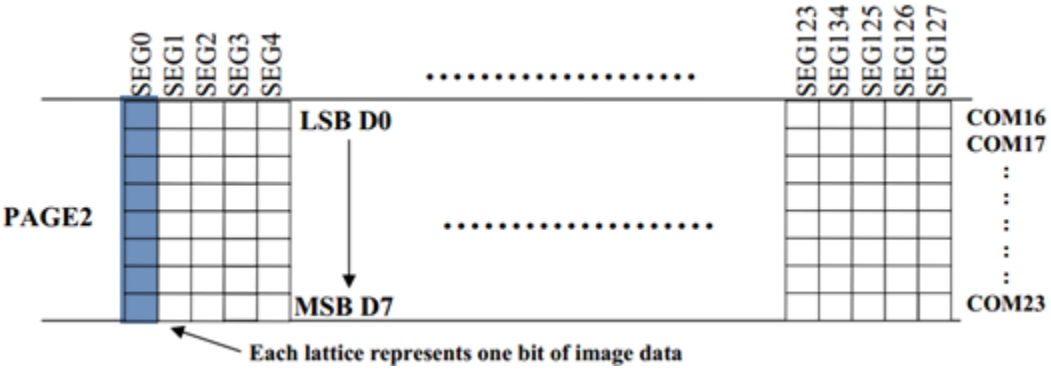


图 15.17　SSD1306 位段写入顺序

至于写入数据的低位写在该页的 COM 低位还是 COM 高位要取决于程序中设置的 COM 扫描方向。如果此时我们继续向该页写入数据，列指针（SEG）会自动加 1，直到 SEG127 才会停止，此时根据我们事先设置好的内存地址模式进行下一步操作，内存地址模式一共有 3 种：

1. 页地址模式（本例程使用该模式）

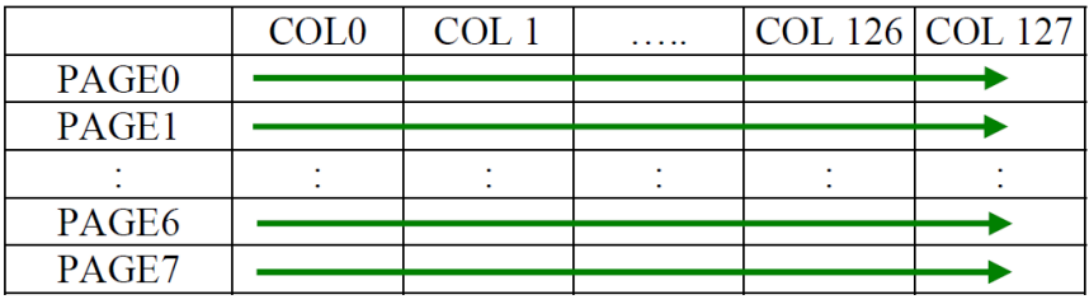


图 15.18　页地址模式

写入数据后，列地址自动加 1，当列地址加到最大值时，会自动回到列开始地址，但是页地址不改变，此时要人为刷新下一个页地址。

2. 水平地址模式

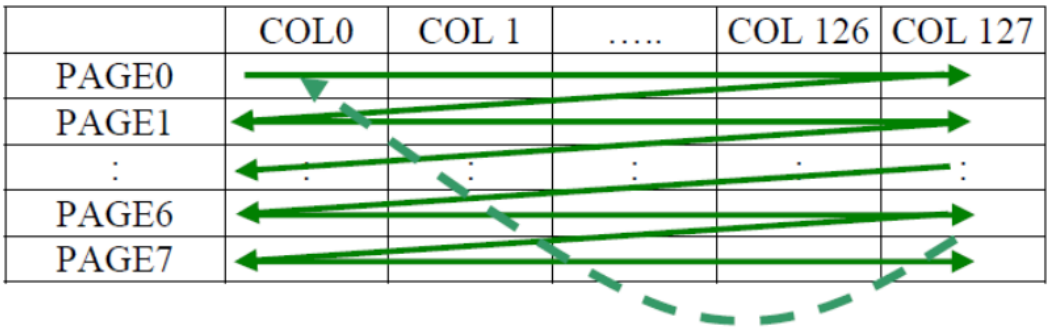


图 15.19 水平地址模式

写入数据后列地址自动加 1，列地址加到最大值后，列地址回到列起始地址，同时页地址加 1。

3. 垂直地址模式

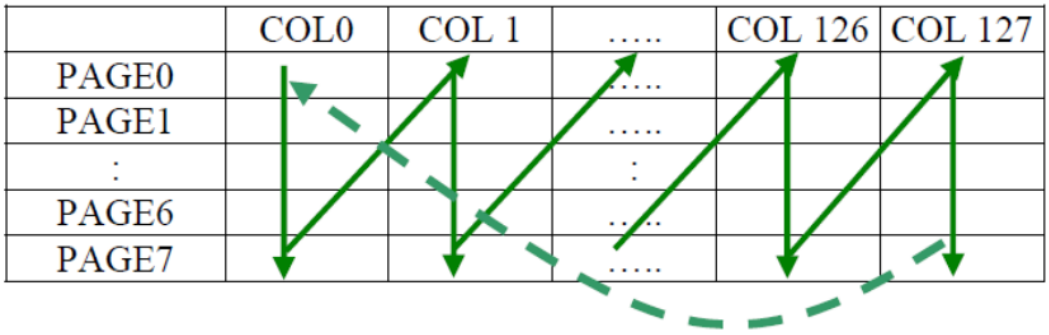


图 15.20 垂直地址模式

写入数据后，页地址加 1，当页地址达到最大后返回页起始地址，同时列地址加 1。

接下里看一下 IIC 控制 SSD1306 的数据格式：

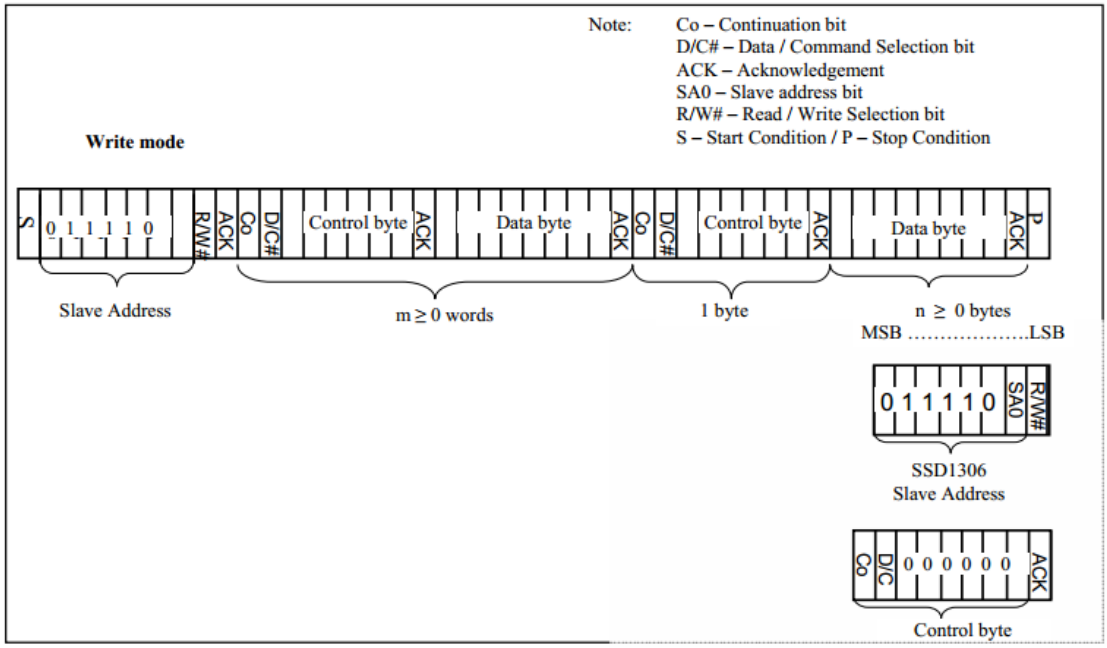


图 15.21 IIC 控制 SSD1306 字节流

先发送写模式从机地址“0x78”，从机地址可以通过设置 SA0 来提供两种可选地址，如下图：

b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
0	1	1	1	1	0	SA0	R/W#

图 15.22 SSD1306 地址配置

本例程中 SA0 设置为 0。接下来发送 Control Byte，起始的 Co 位我们设置为 0，意思是接下来传输的信息只包含数据字节，D/C#位用来指定接下来发送的信息是 SSD1306 的命令还是数据，0 代表接下来发送命令，1 代表发送数据。

SSD1306 的命令有很多，比如开启显示，COM 扫描方向，调节亮度，颜色翻转等，具体的大家可以参考 SSD1306 的 128 页查看，也可以通过程序来查看，建议大家不要刻意去记忆这些指令，知道有哪些功能即可，使用时再去查找。

15.3 程序讲解

15.3.1 IIC 程序讲解

STM32 自带了硬件 IIC，但是作者出于使用习惯和程序可移植性考虑，选择了用 IO 口去模拟 IIC 协议。网上有很多开发者对 STM32 的硬件 IIC 褒贬不一，有的说好用，有的说有 bug，作者这里保留意见，我们不能完全否定 STM32 硬件 IIC 的功能，因为说它有问题的开发者可能是自身没摸透 IIC 协议的配置再加上别人说它有问题，潜意识里就否定了自己。即

使 STM32 的 IIC 存在 bug，也无法阻挡 STM32 的魅力。在后续我会仔细研究一下 STM32 的硬件 IIC 然后和大家分享。

使用 IO 口模拟 IIC 的好处有三点：

1. 使用 IO 模拟 IIC 协议可以让大家把之前学过的 GPIO 知识再进行深度的理解和扩展
2. 加深对 IIC 时序流程的认识
3. 方便移植到 STM32 的任何一个引脚，如再做修改可以移植到其他 MCU 平台

当然这种方法也存在一些缺点，比如考虑这样一种极端情况，在程序运行时会有执行时间很长的中断服务函数打断 IIC 时序，造成 IIC 写失败或者读失败。如果存在这种情况，建议大家在进行 IIC 操作之前关闭全局中断，使用后再打开。

在模拟 IIC 之前，我们定义了一些宏定义，这些宏定义重命名了 SDA 和 SCL 对应引脚，以及时钟，目的是：在今后移植到其他引脚时，可以只改动对应宏定义一处即可产生全局更改。

```
15  /*************************以下宏定义用于移植*****************************/
16  #define IIC_SPEED 1                                          //IIC通信延时时间，单位us
17
18  #define IIC_SDA_RCC RCC_APB2Periph_GPIOB                     //SDA时钟
19  #define IIC_SCL_RCC RCC_APB2Periph_GPIOB                     //SCL时钟
20
21  #define IIC_SDA_PORT GPIOB                                   //SDA引脚所在GPIO组
22  #define IIC_SCL_PORT GPIOB                                   //SCL引脚所在GPIO组
23
24  #define IIC_SDA_PIN GPIO_Pin_10                              //SDA引脚
25  #define IIC_SCL_PIN GPIO_Pin_11                              //SCL引脚
26
27  #define IIC_SDA_IN() setSDA_IN()                             //设置SDA为输入
28  #define IIC_SDA_OUT() setSDA_OUT()                           //设置SDA为输出
29
30  #define IIC_SDA_H() GPIO_SetBits(IIC_SDA_PORT,IIC_SDA_PIN)   //拉高SDA
31  #define IIC_SDA_L() GPIO_ResetBits(IIC_SDA_PORT,IIC_SDA_PIN) //拉低SDA
32
33  #define IIC_SCL_H() GPIO_SetBits(IIC_SCL_PORT,IIC_SCL_PIN)   //拉高SCL
34  #define IIC_SCL_L() GPIO_ResetBits(IIC_SCL_PORT,IIC_SCL_PIN) //拉低SCL
35
36  /*******************************~END~**********************************/
```

图 15.23 IIC 宏定义

这里的 IIC_SPEED 是 IIC 通信时时钟线的延时时间，单位为 us，如果你是 DIY，IIC 的导线很长，那么建议增大延时速度，之前作者对一个 IIC 协议的红外传感器编写驱动时一直调不通，最后加了延时时间就好了，所以，这点要注意，对于 OLED 来说 1us 就可以了，手册上面给出的时序时间都是 ns 级别的，如果延时时间太长，会导致 OLED 刷新速度变慢。

第 27,28 行代码目的是可以人为的在程序运行时动态更改 SDA 的输入输出模式，比如在给从机传输数据时就是输出状态，等待 ACK/NACK 信号以及接受数据时就是输入。对应的函数实现如下：

- SDA 输入/输出模式设置

```
18  void setSDA_IN(void)
19  {
20      GPIO_InitTypeDef GPIO_InitStructure;                    //定义GPIO初始化结构体
21
22      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE); //使能GPIO时钟
23
24      GPIO_InitStructure.GPIO_Pin = IIC_SDA_PIN;              //SDA引脚
25      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;           //设置上拉输入
26
27      GPIO_Init(IIC_SDA_PORT, &GPIO_InitStructure);           //设置生效
28  }

35  void setSDA_OUT(void)
36  {
37      GPIO_InitTypeDef GPIO_InitStructure;                    //定义GPIO初始化结构体
38
39      RCC_APB2PeriphClockCmd(IIC_SDA_RCC, ENABLE);            //使能GPIO时钟
40
41      GPIO_InitStructure.GPIO_Pin = IIC_SDA_PIN;              //SDA引脚
42      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD;        //开漏输出
43      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
44
45      GPIO_Init(IIC_SDA_PORT, &GPIO_InitStructure);           //设置生效
46      GPIO_SetBits(IIC_SDA_PORT,IIC_SDA_PIN);                 //SDA拉高
47  }
```

图 15.24 SDA 输入输出函数

设置输出时也可以是推挽输出，只不过此时 IIC 总线上的上拉电阻就起不到作用了，这种在多从机的情况下可能会产生问题，所以建议大家设置开漏输出。

- 接着是 IIC 引脚的上电初始化函数

```
54  void initIIC(void)
55  {
56      GPIO_InitTypeDef GPIO_InitStructure;                    //定义GPIO初始化结构体
57
58      RCC_APB2PeriphClockCmd(IIC_SDA_RCC, ENABLE);            //使能SDA时钟
59
60      GPIO_InitStructure.GPIO_Pin = IIC_SDA_PIN;              //设置SDA引脚
61      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_OD; //开漏输出
62      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//设置输出速度
63      GPIO_Init(IIC_SDA_PORT, &GPIO_InitStructure);           //设置生效
64      GPIO_SetBits(IIC_SDA_PORT,IIC_SDA_PIN);                 //拉高SDA
65
66      RCC_APB2PeriphClockCmd(IIC_SCL_RCC, ENABLE);            //使能SCL时钟
67      GPIO_InitStructure.GPIO_Pin = IIC_SCL_PIN;              //设置SCL引脚
68      GPIO_Init(IIC_SCL_PORT, &GPIO_InitStructure);           //设置生效
69      GPIO_SetBits(IIC_SCL_PORT,IIC_SCL_PIN);                 //拉高SCL
70  }
```

图 15.25 初始化 IIC

默认都是输出，且空闲状态为高电平。

下面来模拟 IIC 时序：

- 起始信号

```
77   void startIIC(void)
78   {
79       IIC_SDA_OUT();          //设置SDA引脚为开漏输出
80
81       IIC_SCL_H();            //拉高SCL
82       IIC_SDA_H();            //拉高SDA
83       Delay_us(IIC_SPEED);    //延时一段时间
84       IIC_SDA_L();            //拉低SDA，在SCK高电平器件产生下降沿
85       Delay_us(IIC_SPEED);    //延时一段时间
86       IIC_SCL_L();            //拉低时钟，完成一个时钟周期
87   }
```

图 15.26 IIC 起始信号函数

目的是在 SCL 为高电平期间 SDA 产生下降沿

- 停止信号

```
94   void stopIIC(void)
95   {
96       IIC_SDA_OUT();
97
98       IIC_SDA_L();
99       IIC_SCL_H();
100      Delay_us(IIC_SPEED);
101      IIC_SDA_H();            //SCL高电平期间，产生SDA下降沿
102      Delay_us(IIC_SPEED);
103      IIC_SCL_L();
104  }
```

图 15.27 IIC 停止信号函数

- 等待从机响应

```
201  IIC_ACK waitAck(void)
202  {
203      u8 i = 0;
204
205      IIC_SDA_IN();
206      IIC_SCL_H();
207      while(GPIO_ReadInputDataBit(IIC_SDA_PORT,IIC_SDA_PIN))
208      {
209          if(++i>50)
210          {
211              IIC_SCL_L();
212              return  NACK;
213          }
214          Delay_us(1);
215      }
216
217      IIC_SCL_L();
218      return ACK;
219  }
```

图 15.28 IIC 等待从机响应函数

该部分主要是对 ACK 和 NACK 信号的采集，所以用了 while 来获取 SDA 电平状态，其实在大多数应用当中，无需做这种判断，直接控制一个时钟周期过去就可以了。

- 发送一个字节

```
111  void sendIICByte(u8 byte)
112  {
113      u8 i;
114
115      IIC_SDA_OUT();
116
117      for(i=0;i<8;++i)
118      {
119          if(byte & 0x80)
120          {
121              IIC_SDA_H();
122          }else
123          {
124              IIC_SDA_L();
125          }
126          byte <<= 1;
127
128          IIC_SCL_H();
129          Delay_us(IIC_SPEED);
130          IIC_SCL_L();
131          Delay_us(IIC_SPEED);
132      }
133  }
```

图 15.29 IIC 发送字节函数

IIC 发送数据时先发送的高位数据，因此我们使用数据和 0x80 做位与，在执行完输出之后，通过移位操作把下一位提到最高位，本质上就是软件实现移位寄存器的操作。

还有其他的接收数据和发送给主机响应的函数这里就不做讲解，大家可以参照程序理解，这里要说明一下，我们 OLED 只用到了 IIC 写数据，因此 IIC 读数据和给从机发送响应的函数我暂时没有做测试，后期我会验证函数的正确性，大家以最新版本的例程为准。

到这里 IIC 就介绍完了，我们模拟的 IIC 协议其实就是一个工具，单纯的拿出来也用不到，它是供 OLED 函数调用的，所以可以定义模拟 IIC 协议的函数为通用函数。

15.3.2 OLED 程序讲解

对于屏幕来说，主要的功能无外乎就是显示 ANSICII 字符，显示整数，显示汉字，显示图片，这些函数在我们的例程中都已经实现，另外我们还编写了其他的非常实用灵活性更高的函数，比如设置屏幕显示方向（和手机屏幕翻转类似），息屏显屏操作（和手机唤醒屏类似）以及屏幕反色（类似负片，可用于视力有障碍的人观看的高对比度显示模式）。

这里先介绍一下比较常用的 OLED 控制指令：

1. 设置页坐标（范围 0-7）
   设置页坐标其实就是设置 OLED 屏幕的 y 轴坐标，指令为 0xB0+y,y 是要设置的坐标
2. 设置段坐标（0-127）
   即设置 OLED 的 x 轴坐标，对应指令有两条，每条记录段坐标的 4 位，加起来一共 8 位。对应指令为设置低四位的 0x00 和设置高四位的 0x10，即 0x10 的低四位作为 x 轴坐标的高四位，0x00 的低四位作为 x 轴坐标的低四位。
3. 开启/关闭 OLED 显示，0xAF 为开启屏幕，0xAE 为关闭屏幕。
4. 设置 OLED 反色，正常显示为 0xA6,反色显示为 0xA7
5. 设置段重映射，即屏幕水平翻转，指令为 A0/A1
6. 设置 COM 扫描方向，即垂直翻转，指令为 C0/C8

通常 5,6 一起使用用于设置屏幕翻转。

在讲程序之前，我们要先准备字库和和图片库供程序调用，有两种方式提供这些调用资源，先说字库，第一种就是使用字库芯片提供常用字体大小的中英文字库，例如高通的 GT20L16S1Y，第二种是使用软件来转换我们常用的字库，然后存储在单片机的 ROM 当中，对于图片来说，因为我们使用的是单色 OLED 屏幕所以显示不了色彩丰富的图片，因此可直接通过软件生成，这个软件就 “PCtoLCD 2002”，我们先对其进行设置，来和我们屏幕设置保持一致（其实就是扫描模式）：

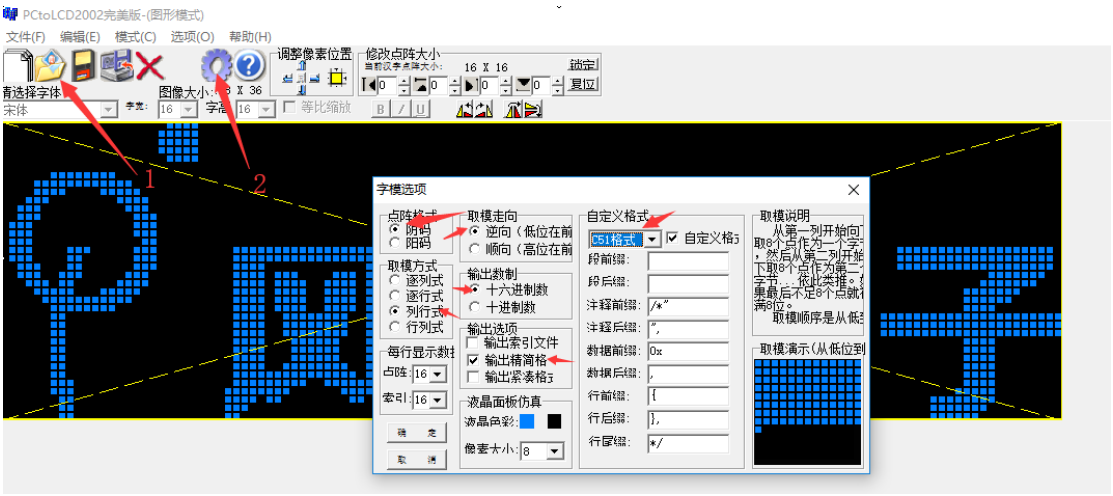


图 15.30　设置取模软件“PCtoLCD”

按照步骤设置即可。

接下里说一说如何制作图片库：

去网上随便下载一个图标，这里推荐阿里矢量图标库，比如我们下载一个 PNG 格式的微信图标，然后使用 windows 自带的画图软件打开图片并另存为 BMP 单色位图，此时变为黑白图片。接着对其进行如下设置：

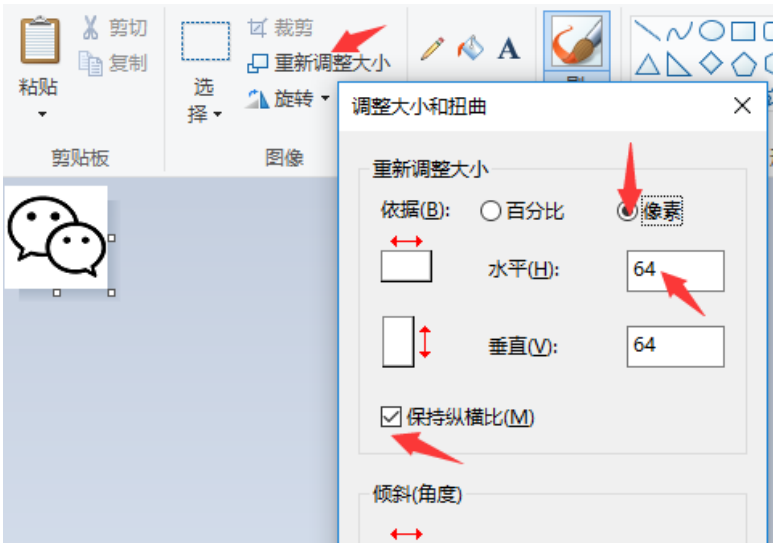


图 15.31　设置 LOGO 大小

然后调整图片大小为 128*64，并使用选择工具将图标选择并拖动到中心位置并保存。

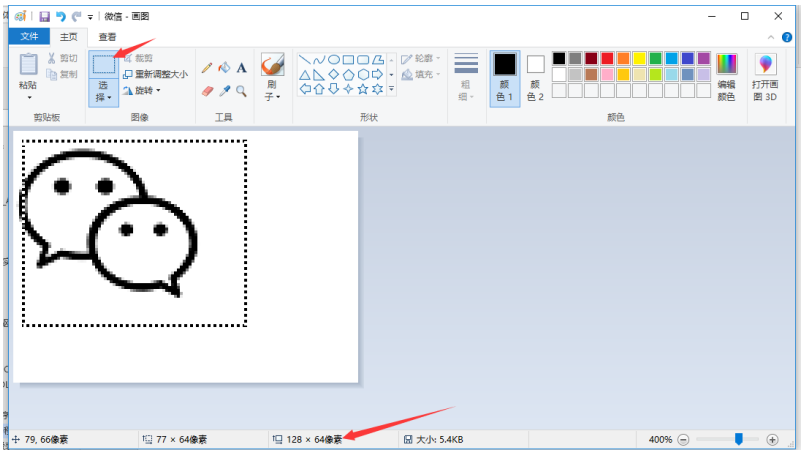


图 15.31　调整 LOGO 位置和画布尺寸

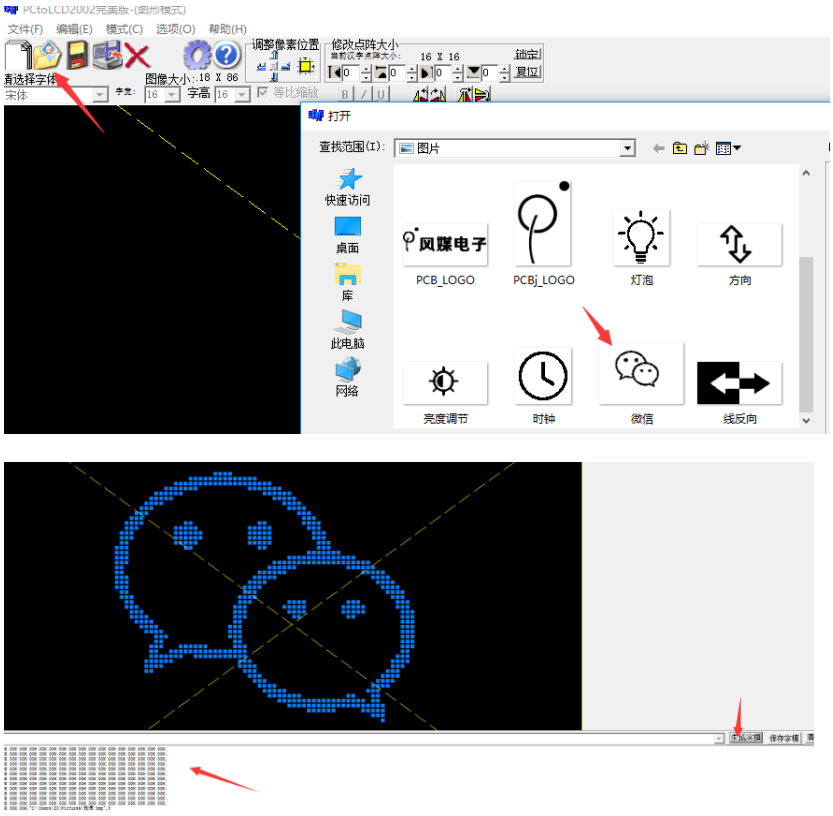


图 15.32　生成 LOGO 数据

复制数据到 BMP.h 文件中的 WX_LOGO 数组中即可（由于例程没有用到该图标，因此没有加入到该项目当中，大家可以自己生成）。

字库制作办法如下：

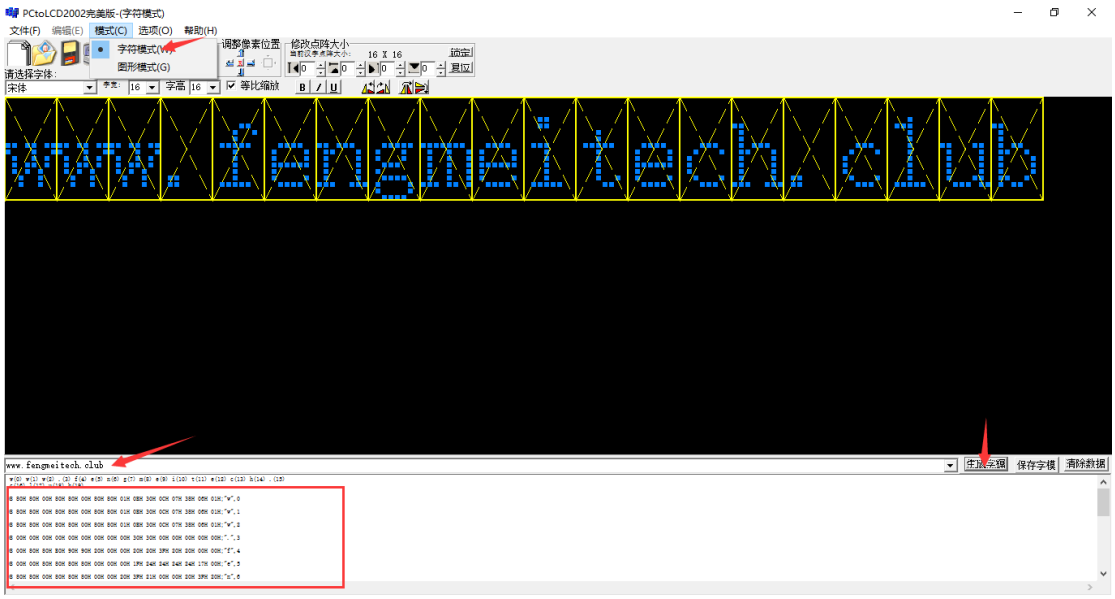


图 15.33 生成文字数据

然后将生成的字库放到 FONT.h 文件中对应的数组内即可。这里说明一下，由于汉字的特殊性 16*8 的点阵是放不下的，因此要用 16*16 或者其他横纵比为 1 的点阵。

下面开始讲解 OLED 驱动代码：

设置坐标

```
25  static void setPos(unsigned char x, unsigned char y)
26  {
27      writeCommand(0xb0+y);
28      writeCommand(((x&0xf0)>>4)|0x10);
29      writeCommand(x&0x0f);
30  }
```

图 15.34 设置显示坐标起点

该函数被 static 关键字修饰，其意义是只供本源文件内的函数使用，外部文件不能访问，除此之外它也起到提示程序员的作用，告诉程序员，这个函数不能被外部调用。为什么要用 static 修饰函数呢？主要是因为我们希望只向调用该文件的上层函数提供有用的或者是我们希望他们了解的接口，屏蔽不想让上层知道的接口，因此就要用 static 来修饰。

该函数用于设置 OLED 绘制的起始坐标，它是最底层的最基础的函数，因此被调用的最多。

- 汉字索引查找函数

```
39  static u8 findCNIndex(u8* str,u8* cnfont_index)
40  {
41      u16 cnfont_size = strlen(cnfont_index);
42
43      u8 index = 0;
44      for(index=0;index<cnfont_size/3;++index)
45      {
46          if(((str[0]^cnfont_index[index*3+0])||(str[1]^cnfont_index[index*3+1])||(str[2]^cnfont_index[index*3+2]))==0)
47          {
48              return index;
49          }
50      }
51
52      return 0; //没有匹配到直接返回字库第一个索引，这里是"风"
53  }
```

图 15.35 汉字索引查找函数

该函数用于汉字索引，我们打开 FONT.h 头文件，在汉字字库的后面又多出来一个汉字字符串（箭头所指）：

```
422  const unsigned char CN1616[4][32] =
423  {
424  风  {0x00,0x00,0xFE,0x02,0x12,0x22,0xC2,0x02,0xC2,0x32,0x02,0xFE,0x00,0x00,0x00,
425
426  媒  {0x10,0x10,0xF0,0x1F,0x10,0xF0,0x04,0x04,0xFF,0x24,0x24,0x24,0xFF,0x04,0x04,
427
428  电  {0x00,0x00,0xF8,0x88,0x88,0x88,0x88,0xFF,0x88,0x88,0x88,0x88,0xF8,0x00,0x00,
429
430  子  {0x80,0x82,0x82,0x82,0x82,0x82,0x82,0xE2,0xA2,0x92,0x8A,0x86,0x82,0x80,0x80,
431  };
432
433  unsigned char CN1616_Index[] = "风媒电子"; //16*16中文字库索引表
```

图 15.36 汉字索引和汉字字库对应关系

假设没有该字符串，那么在调用汉字显示的话就必须人为指定 CN1616 数组的汉字位置，比如我们想显示“风媒电子”这四个字，就要依次调用 CN1616[0]，CN1616[1]，CN1616[2]，CN1616[3]来实现，不方便程序员记忆，灵活性差。因此我们希望有一种类似目录的东西存在，让我们直接在显示汉字函数的参数中传入“风媒电子”四个字即可自动匹配这些汉字在 CN1616 数组中的位置，然后显示即可。因为 KEIL 环境下，汉字使用 UTF-8 编码，因此一个汉字占用 3 个字节空间，那么我们就要比较传入的该汉字的这 3 个字节和我们 CN1616_Index 数组中的某三个字节是否存在一致，如果一致就返回该汉字的位置，该位置对应的就是 CN1616 数组中对应该函数的位置。我们比较数据是否相同，采用的是位的异或操作，即两个数据相同结果为 0，具体看上面函数。

- 屏幕设置函数

```
void setScreenReverse(SCREEN_SHOW set);
void setScreenDir(SCREEN_DIR set);
void setScreenSwtich(SCREEN_SWITCH set);
```

图 15.37 OLED 设置函数

这三个函数比较简单，大家知道即可，第一个用于设置反色，第二个设置显示方向，第三个设置 OLED 是否开启。

- OLED 初始化

```
115  void initOLED(void)
116  {
117      writeCommand(0x81);      //设置亮度
118      writeCommand(0xFF);      //亮度值最大 复位默认0x7F
119      writeCommand(0xA1);      //设置段映射方式即设置是否水平翻转 A0表示翻转 通常和C0一起使用
120      writeCommand(0xC8);      //设置COM扫描模式即设置是否垂直翻转 C0表示翻转 通常和A0一起使用
121      writeCommand(0x8D);      //电荷泵使能
122      writeCommand(0x14);
123
124      writeCommand(0xAF);      //开屏幕，默认是关闭的就和没上电一样，所以要手动开启
125  }
```

图 15.38 OLED 初始化函数

主要就是对 OLED 进行一些基础的配置，**需要注意的就是要在 IIC 初始化完之后再调用 OLED 初始化。**

- 通信接口函数

```
61  static void writeCommand(unsigned char cmd)
62  {
63      startIIC();
64      sendIICByte(0x78); //发送从机地址及写指令位('0')
65      waitAck();
66      sendIICByte(0x00); //写入控制字节
67      waitAck();
68      sendIICByte(cmd);
69      waitAck();
70      stopIIC();
71  }
72
73  /**
74   * 功能：写入数据给OLED
75   * 参数：
76   *      data:数据
77   * 返回值：None
78   */
79  static void writeData(unsigned char data)
80  {
81      startIIC();
82      sendIICByte(0x78);  //发送从机地址及写指令位('0')
83      waitAck();
84      sendIICByte(0x40);  //写入控制字节
85      waitAck();
86      sendIICByte(data);
87      waitAck();
88      stopIIC();
89  }
```

图 15.39 OLED 写数据/写命令函数

这两个函数也是用的最多的底层函数，也同样被 static 修饰，供内部使用。一个函数用于发送命令，一个函数用于发送显示数据。OLED 是没有 IIC 读操作的，这点要注意。

- 屏幕格式化函数

```
/**
 * 功能：格式化屏幕，常使用0x00或者0xFF清屏，使用不同数据可以产生不同的条纹
 * 参数：
 *      format_data：格式化内容，一般清屏会用到0x00或者0xFF
 * 返回值：None
 */
void formatScreen(u8 format_data)
{
    u8 x,y;
    for(y=0;y<8;++y)
    {
        writeCommand(0xb0+y);     //设置页地址（0~7）
        writeCommand(0x00);       //设置显示位置—列低地址
        writeCommand(0x10);       //设置显示位置—列高地址
        for(x=0;x<128;++x)
        {
            writeData(format_data);
        }
    }
}
```

图 15.40 屏幕格式化函数

屏幕格式化函数用于将指定的 format_data 数据写入都屏幕上，一般用于清屏操作，0x00 屏幕全灭，0xFF 屏幕全亮，也就是白屏，当然大家可以写入其他数据，让屏幕产生美丽的花纹或者雪花状。

- 在屏幕指定位置显示一个 ANSICII 字符

```
void showChar(u8 x,u8 y,u8 ch,FONT_SIZE f_size)
{
    u8 index = ch-' ';
    u8 i;

    if(x > 127 || y > 7)            //参数异常处理
    {
        x = 0;
        y = 0;
    }
    if(f_size == FONT_16_EN)        //如果是16*8点阵
    {
        setPos(x,y);
        for(i=0;i<8;++i)            //由于是8*16的点阵，因此占用两页，要分成写入，此时写入第一页
        {
            writeData(ANSIC0816[index][i]);
        }
        setPos(x,y+1);              //人为指定下一页地址
        for(i=8;i<16;++i)           //由于是8*16的点阵，因此占用两页，要分成写入，此时写入第二页
        {
            writeData(ANSIC0816[index][i]);
        }

    }else if(f_size == FONT_8_EN)   //6*8点阵
```

```
216         {
217             setPos(x,y);
218             for(i=0;i<6;i++)            //6*8点阵，写入一页即可
219             {
220                 writeData(ANSIC0608[index][i]);
221             }
222         }else
223         {
224             /*其他字体敬请期待:)*/
225         }
226     }
```

图 15.41 显示字符函数

它就类似于组成句子的单词，和我们串口发送单个字节函数的功能同理，显示字符串、数字都是以该函数为基础编写的。

- 显示 ANSICII 字符串

```
237 void showString(u8 x,u8 y,u8* str,FONT_SIZE f_size)
238 {
239     while(*str)
240     {
241         showChar(x,y,*str++,f_size);
242         x += f_size;          //增加横坐标，移到下一个汉字位置
243     }
244 }
```

图 15.42 显示字符串函数

原理就是显示完一个字符之后，跳到下一个字符位置显示下一个字符。

- 显示整数函数

```
258  void showNumber(u8 x,u8 y,s32 number,RADIX radix,u8 ndigit,FONT_SIZE f_size)
259  {
260      u8 i = 0;
261      u8 str[25] = {0};               //定义数字转字符串的存储buffer
262
263      if(radix==DEC)                  //按十进制存储
264      {
265          sprintf(str,"%d",number);
266      }else if(radix==HEX)            //按十六进制存储
267      {
268          sprintf(str,"%X",number);
269      }else if(radix==OCT)            //按八进制存储
270      {
271          sprintf(str,"%o",number);
272      }else
273      {
274          sprintf(str,"%d",number);   //参数错误，按十进制处理
275      }
276
277
278      for(i=strlen(str);i<ndigit;++i)
279      {
280          str[i] = ' ';
281      }
282
283      i = 0;
284      while(str[i])
285      {
286          showChar(x,y,str[i++],f_size);
287          x += f_size;
288      }
289  }
```

图 15.43 显示数字函数

该函数异常强大，用户可以设置显示位数，显示进制，支持负数，其核心就是使用格式化字符串函数 sprintf(str,"%d",number),它的作用类似于胶水，它把 number 格式化后和中间字符串“粘”到一起，然后将合并后的结果存放到 str 指向的缓冲区中。比如 sprintf(str,"number is %d",123);,str 是 char *类型指针,指向一个足够大的数组空间，其结果是字符串“number is 123”被存到了 str 指向的空间。第 278 行的 for 语句用于清除上次残留的显示数据，比如我要显示的是 4 位的整数 0-4095，假如此时数据由 4025 减小到 23 时，屏幕显示的是 2325，因为新数据只刷新了两位，因此我们要将空白区域的上次残留数据给抹掉，就要进行这样的操作。

- 显示汉字函数

```
301  void showCNString(u8 x,u8 y,u8* str,FONT_SIZE f_size)
302  {
303      u8 i;
304      u8 cn_index;
305      u8 count;
306      if(x > 127 || y > 7) //参数异常处理
307      {
308          x = 0;
309          y = 0;
310      }
311
312      for(count=0;count<strlen(str)/3;++count)
313      {
314          cn_index = findCNIndex(str+count*3,CN1616_Index);
315          setPos(x+16*count,y);
316          for(i=0;i<16;++i)
317          {
318              writeData(CN1616[cn_index][i]);
319          }
320          setPos(x+16*count,y+1);
321          for(i=16;i<32;++i)
322          {
323              writeData(CN1616[cn_index][i]);
324          }
325      }
326
327  }
```

图 15.44 显示汉字函数

其过程就是不断的在汉字字库索引数组中遍历和待查找汉字相同的汉字索引位置，然后返回该位置进行显示，如果传入参数不正确，则返回的是该字库第一个汉字位置，这里是“风”对应的索引为 0。

大家如果感兴趣可以使用 VSCODE 将文件编码方式改变一下，这时你会发现查找一直失败，因为同一字符在不同编码情况下的内容是不同的。如下图：

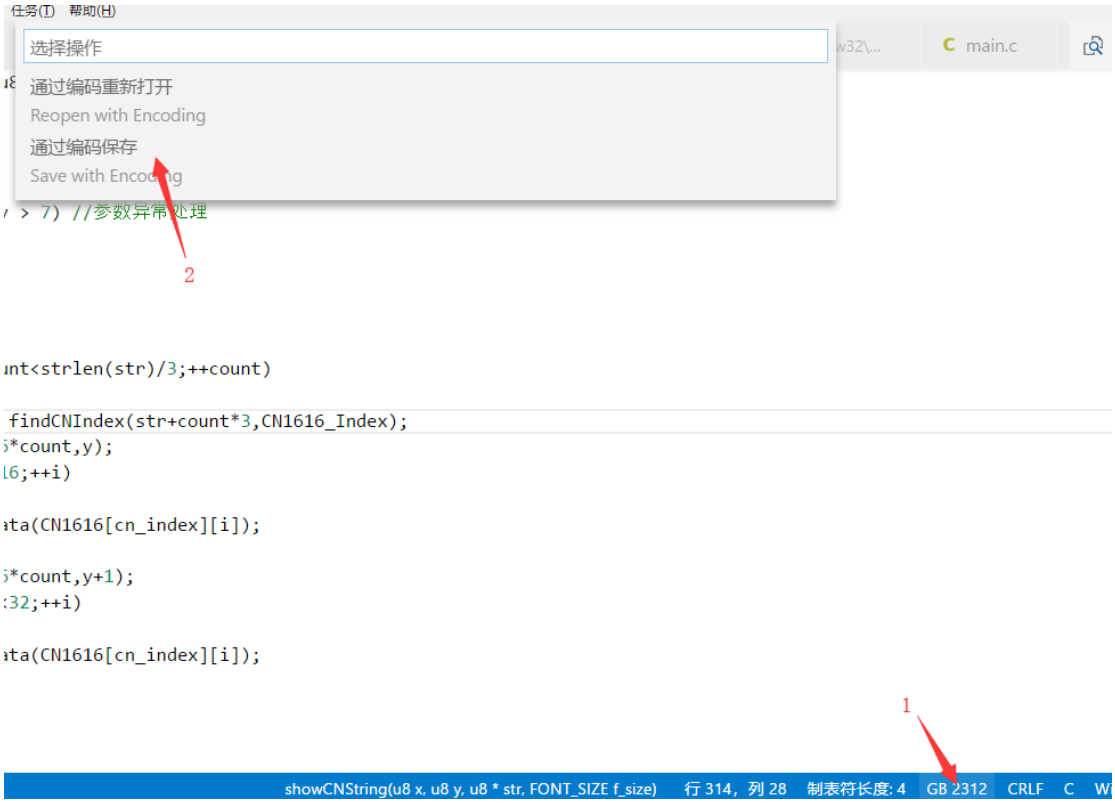


图 15.45 设置文件编码

这是一种很难察觉的错误，编译器本身并不会发现，这种错误甚至比内存溢出更加诡异，因为我们本能上无法察觉这种错误，因此大家今后要注意，要保证定义汉字的字库索引文件和调用显示汉字函数的文件字符编码要相同。如下图所示：

图 15.46 汉字调用位置

- 图像绘制函数

```
342  void showImage(u8 xpos, u8 ypos,u8 x_len, u8 y_len,IMAGE_INDEX  image_index)
343  {
344      u16 i,j;
345
346      for(i=0;i<y_len;++i)                    //页地址控制
347      {
348          setPos(xpos,ypos++);
349
350          for(j=i*128+xpos;j<i*128+x_len;++j) //列地址控制
351          {
352              switch(image_index)
353              {
354                  case FM_LOGO_ENUM        :writeData(FM_LOGO[j]);        break;
355                  case BRIGHTNESS_LOGO_ENUM:writeData(BRIGHTNESS_LOGO[j]);break;
356                  case DIRECT_LOGO_ENUM    :writeData(DIRECT_LOGO[j]);    break;
357                  case REVERSAL_LOGO_ENUM  :writeData(REVERSAL_LOGO[j]);  break;
358
359                  default                  :                              break;
360              }
361
362          }
363      }
364  }
```

图 15.47 图像绘制函数

用于在指定位置，指定区域绘制图形，一般我们拿该函数来绘制全屏 LOGO，但是它的用法不止这一种，比如例程中的开机 LOGO 从左侧切入的动画就是利用该函数完成的。

接下来是主函数，主函数用于显示开机界面和 ADC 转换值，键值以及 UART 接收到的数据，效果图如下：

图 15.48 程序运行效果

Lux 是 ADC 采集的光照值，Key 对应键值，COM 对应上位机通过 UART 传过来的数据。下面是主函数代码：

```
int main(void)
{
    u8 i = 0;
    /*初始化各外设*/
    initSysTick();
    initADC();
    initUART();           //波特率9600
    initLED();
    initKey();
    initNVIC(NVIC_PriorityGroup_2);
    initIIC();
    initOLED();
    formatScreen(0x00);
    for(i=0;i<=128;i+=2)  //风媒电子LOGO从左向右动态显示
    {
        showImage(0,0,i,8,FM_LOGO_ENUM);
    }
    formatScreen(0x00);
    showCNString(32,0,"风媒电子",FONT_16_CN);
    showString(0,2,"Lux:",FONT_16_EN);
    showString(0,4,"Key:",FONT_16_EN);
    showString(0,6,"COM:",FONT_16_EN);

    while (1)
    {
        lux_value = getConvValueAve(10,100); //1ms采10次取均值
        key_value = getKeyValue(KEY_PRESS);//按键松开生效
        /*com_value被UART1中断服务函数赋值*/

        showNumber(33,2,lux_value,DEC,4,FONT_16_EN);
        switch(key_value)
        {
            case KEY_UP   :  showString(33,4,"up key  ",FONT_16_EN); break;
            case KEY_DOWN:  showString(33,4,"down key",FONT_16_EN); break;
            case KEY_ALL :  showString(33,4,"all key ",FONT_16_EN); break;

            default :       showString(33,4,"no key  ",FONT_16_EN); break;
        }
        showNumber(33,6,com_value,HEX,2,FONT_16_EN);
        toggleLED();

    }
}
```

图 15.49 主函数

其过程就是采集和显示，这里就不多说了，这些函数之前都讲过。

最后说点其他的，可能有人会诟病我们提供的图片绘制函数太麻烦，即如果在 BMP.h 文件中添加一个图片，还要在 OLED.h 文件中的 IMAGE_INDEX 枚举添加新的枚举值成员，同样还要在 OLED.c 中对绘制图形函数相关位置添加新图片的绘制部分，我们这做的目的是让图片数组的数据存储在 ROM 而不是 RAM，想把数据存储到 ROM 就必须使用 const 来修饰数组，但是 const 修饰的数组类型不允许以其指针的形式传入函数，因为编译器认为你正在企图破坏一个受保护的常量数据，因此会报错，你不得不将 const 去掉，这带来的结果是图像数据存到了 RAM 中，RAM 占用急剧加升，RAM 只有 20KB 空间（ROM 有 64KB）存储程序运行结果、各个变量以及图形数据，很可能会造成内存溢出，因此我们牺牲了可维护性换来的是健壮的程序。

本章结束，最后带大家看一些你懂得的图片放松下：

方向

图 15.50 屏幕倒转效果

反色

图 15.51 屏幕反色效果

第十六章 定时器_实现定时任务

16.1 项目要求

通过两个按键控制定时器发生中断的时间，KEY_UP 设置中断时间为 100ms，KEY_DOWN 设置中断事件为 800ms，在中断服务函数中对 LED 进行翻转，并将中断次数通过串口发送给上位机。

注：本章用到核心板，对应例程 10。

16.2 原理讲解

16.2.1 STM32 定时器简述

我们使用的 STM32F103C8T6 含有 4 个 16 位定时器，分别是一个高级定时器 TIM1 和 3 个通用定时器 TIM2-4，高级定时器除了拥有通用定时器的功能之外，还支持死区控制和紧急刹车，用于产生控制电机的 PWM。但在大多数应用场合我们拿 TIM1 当通用定时器来用。

通用定时器是一个通过可编程预分频器驱动的 16 位自动装载计数器构成。它适用于多种场合，包括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)。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。每个定时器都是完全独立的，没有互相共享任何资源。

通用定时器功能包括：

- 16 位向上、向下、向上/向下自动装载计数器
- 16 位可编程(可以实时修改)预分频器，计数器时钟频率的分频系数为 1～65536 之间的任意数值
- 4 个独立通道：
  - 输入捕获（用于采集波形周期和占空比）
  - 输出比较
  - PWM 生成(边缘或中间对齐模式)
  - 单脉冲模式输出
- 使用外部信号控制定时器和定时器互连的同步电路
- 如下事件发生时产生中断/DMA：
  - 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)
  - 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)
  - 输入捕获
  - 输出比较
- 支持针对定位的增量(正交)编码器和霍尔传感器电路
- 触发输入作为外部时钟或者按周期的电流管理

STM32 的定时器非常强大，功能也异常丰富，但是我们平时用到的无外乎三种：

1. 一个是定时器溢出中断用于执行定时任务，比如每隔固定时间采集一次传感器数据；
2. 第二个是输入捕获，它可以获取外部输入信号的高低电平时间，进而计算出该波形的周期（相邻下降沿或者相邻上升沿的间隔）和占空比（相邻上升沿和下降沿或者相邻下降沿和上升沿时间）；
3. 第三个就是通过输出比较产生 PWM，PWM 应用非常广泛，比如调节灯光亮度，调节电机转速，调节声音等等，对于其他的功能来说用的不是很多，因此大家把主要的这三个功能掌握即可。

16.2.2 图说定时器

本章学习的是第一个功能——通过定时器溢出中断实现定时任务，在前面第八章我们讲解过的 Systick 定时器和本章讲解的 TIMx 定时器一样，最基本的单元都是定时计数器，即通过计数来产生等间隔时间，这些时间累积得到我们想要的周期。在讲解 STM32 的定时器操作之前我们先理解一下什么是定时计数器。

现在通过一幅图来加深对定时器计数器的理解：

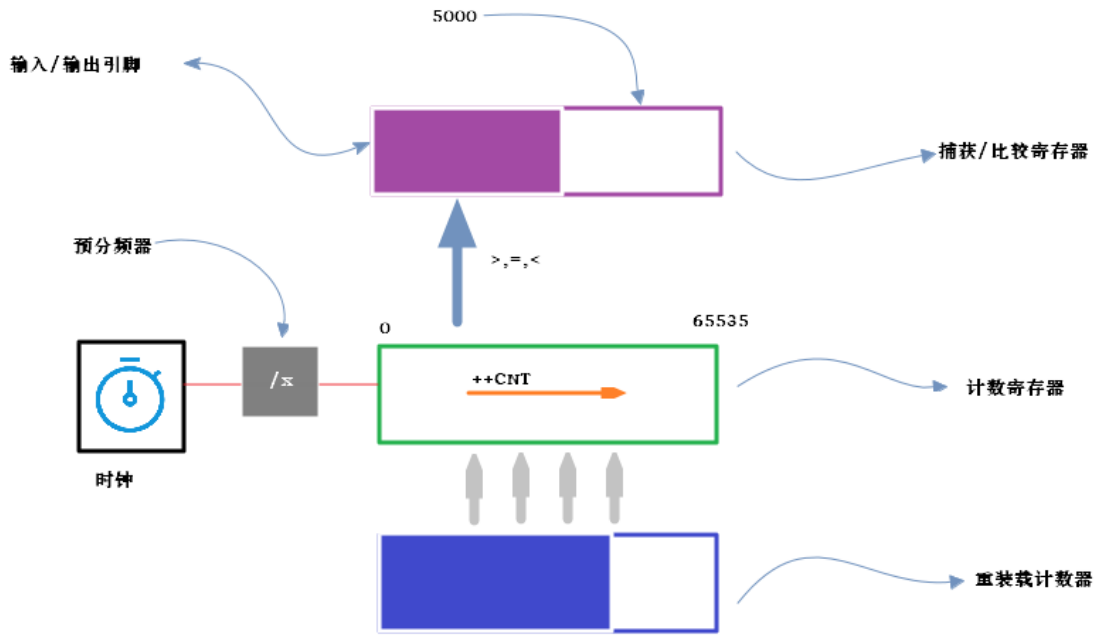


图 16.1　定时计数器基本结构

1. 时钟源

这是任何一个定时计数器的基础部件，要计数就要有时钟，时钟来源根据单片机而异，我们使用的 STM32 的时钟源有四种：

- 内部时钟(CK_INT)
- 外部时钟模式 1：外部输入脚(TIx)

- 外部时钟模式 2：外部触发输入(ETR)
- 内部触发输入(ITRx)：使用一个定时器作为另一个定时器的预分频器，如可以配置一个定时器 Timer1 作为另一个定时器 Timer2 的预分频器。

我们大多数情况下使用的是内部时钟，内部时钟结构框图如下：

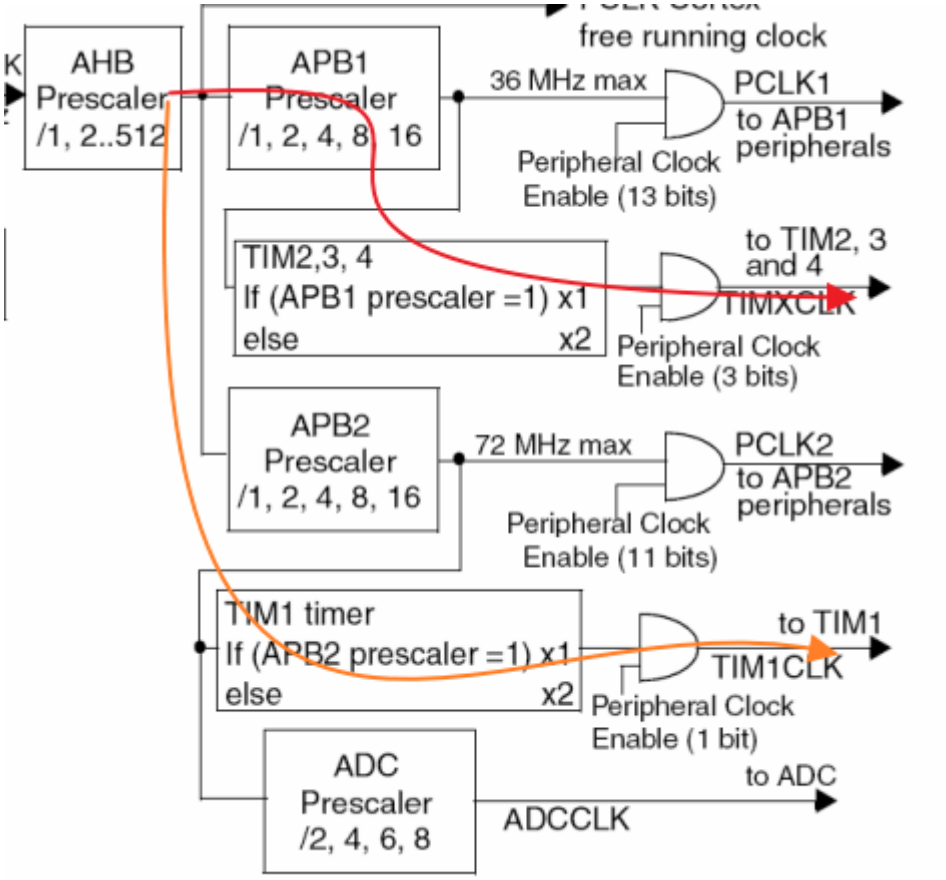


图 16.2　STM32 定时器内部时钟源

可以看到通用定时计数器和高级定时计数器的内部时钟来源是不同的，这点大家要注意。另外提供给定时器的时钟是经过倍频的，只不过倍频的情况要依据 APB1 和 APB2 的分频系数决定，对于 APB2 来说，它的时钟分频系数为 1，因此 TIM1 的时钟最大就是 72MHz（1 倍频），而对于 APB1 来说它的时钟分频系数为 2，因此 TIM2-4 的时钟频率为 2 倍 APB1 时钟，即：72MHz，这么做的目的是保证当 APB1 为二分频和 APB2 为一分频的情况下，各个定时器的时钟频率一致。

2. 计数器（计数寄存器）

接下来是时基部分（计数寄存器，绿色框部分），这部分的任务就是一刻不停的像某个方向计数，图中展示的是向上计数（++），STM32 还支持向下计数（--）和双向计数，双向计数也叫向上/向下计数，它的计数方式类似于往返跑，即从 0 记到设置值后产生一个向上溢出中断，然后自动从设置值向下计数到 0，再产生一个向下溢出中断，以此往复。

现假设此时给计数器的时钟周期是 1ms，计数最大值设置为 1000，则定时计数器的定时中断周期就是 1S。

3. 重装载寄存器

那么问题来了，计数器周期设置的值来自哪里，这就是重装载寄存器的作用（蓝色框部分），它是这个设置值的一个备份，每当计数器溢出后，该寄存器会自动将其内部的数据一次性的“倒入”计数器，告诉计数器：“按照这个工作量来计，计完通知我。”

4. 捕获/比较寄存器

前面讲了时钟-计数器-重装载寄存器，他们三个只能实现定时中断的功能，对于输入捕获或者输出比较我们还需使用捕获/比较寄存器（紫色框部分）。

捕获对应的是输入，即通过 IO 口捕获外部的数字信号，来计算外部信号的电平长度，周期，占空比等，一般捕获的过程如下（假如此时设置的是下降沿捕获）：

当第一个下降沿来临时，触发捕获比较寄存器将此时计数器中的计数值拷贝过来。记做 c1；

当第二个下降沿来临时，再次触发捕获比较寄存器将此时计数器中的计数值拷贝过来。记做 c2；

则此时该电平的长度为(c2-c1)*计数器时钟周期(假设该过程计数器没有发生溢出，即：两次触发在一个定时中断周期内）。

比较对应的是输出，通常我们通过输出比较来在引脚的产生 PWM（Pulse Width Modulation)，这是一种周期和占空比可调的方波，根据占空比不同即等效电压不同的原理，我们可以给电机调速，给灯光调节亮度。要想使用输出比较功能就要给捕获比较寄存器提供一个比较的值，比如设置 1000，那么在计数器小于 1000 时为高电平（也可以通过寄存器设置为高电平），大于 1000 小于溢出值时为低电平，这就实现了 PWM 的生成。

讲完了标准定时计数器结构之后，相信大家在学习 STM32 定时器计数器的结构时就很轻松了：

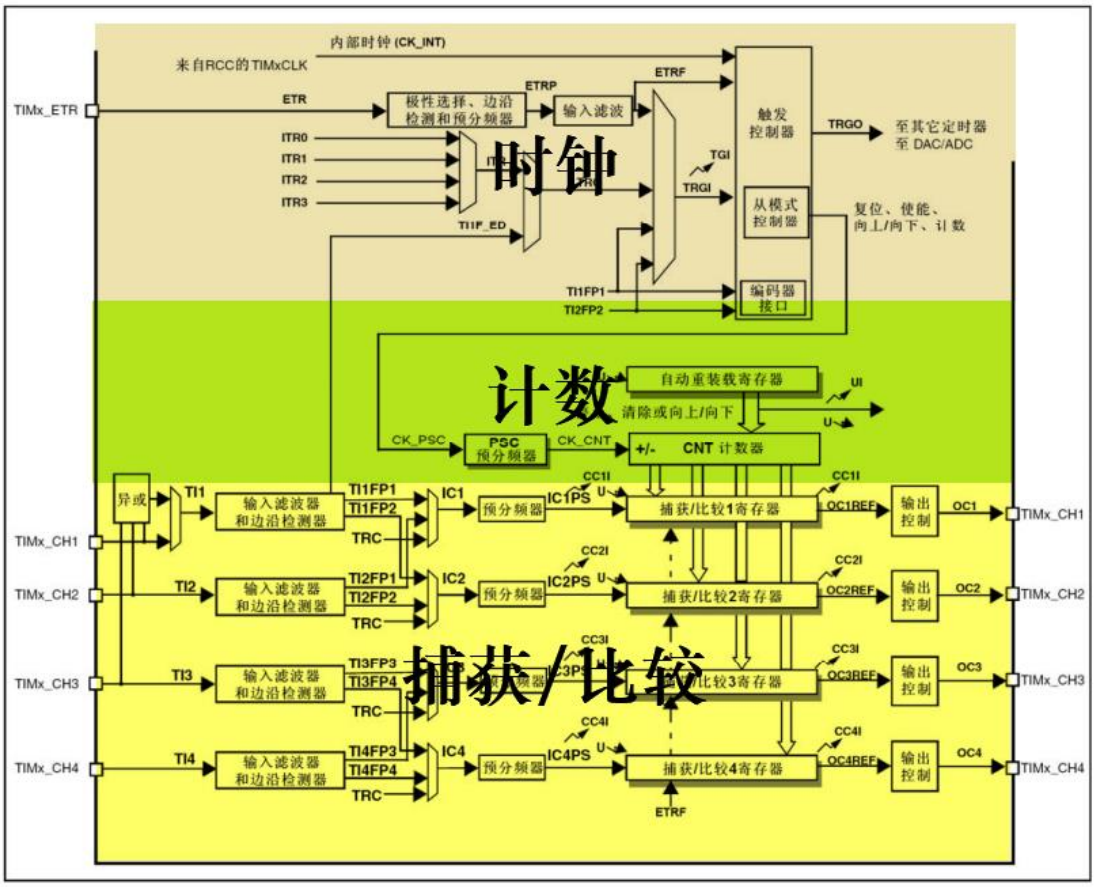


图 16.3　SM32 定时器内部结构框图

16.2.3 寄存器讲解

本章之讲解和定时溢出中断有关的寄存器，其他的寄存器会在下两章单独讲解，目的是减轻大家的学习负担。

- 控制寄存器 1（TIMx_CR1）

需要关注的有 CMS 位域，DIR 位和 CEN 位。

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

位7	**ARPE**：自动重装载预装载允许位 (Auto-reload preload enable) 0：TIMx_ARR寄存器没有缓冲； 1：TIMx_ARR寄存器被装入缓冲器。

位6:5	**CMS[1:0]**：选择中央对齐模式 (Center-aligned mode selection) 00：边沿对齐模式。计数器依据方向位(DIR)向上或向下计数。 01：中央对齐模式1。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位，只在计数器向下计数时被设置。 10：中央对齐模式2。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位，只在计数器向上计数时被设置。 11：中央对齐模式3。计数器交替地向上和向下计数。配置为输出的通道(TIMx_CCMRx寄存器中CCxS=00)的输出比较中断标志位，在计数器向上和向下计数时均被设置。 注：在计数器开启时(CEN=1)，不允许从边沿对齐模式转换到中央对齐模式。
位4	**DIR**：方向 (Direction) 0：计数器向上计数； 1：计数器向下计数。 注：当计数器配置为中央对齐模式或编码器模式时，该位为只读。
位0	**CEN**：使能计数器 0：禁止计数器； 1：使能计数器。 注：在软件设置了CEN位后，外部时钟、门控模式和编码器模式才能工作。触发模式可以自动地通过硬件设置CEN位。 在单脉冲模式下，当发生更新事件时，CEN被自动清除。

图 16.4　控制寄存器

其中 4 到 6 位用于设置计数器的计数方向，0 位用于开启计数器，即生效预分频器的时钟输出 CK_CNT，ARPE 位用于设置是否使用重装载寄存器的预装载功能，我们在后面结合程序进行讲解。

- DMA/中断使能寄存器(TIMx_DIER)

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

位0	**UIE**：允许更新中断 (Update interrupt enable) 0：禁止更新中断； 1：允许更新中断。

图 16.5　DMA/中断使能寄存器

该寄存器用于使能各个触发事件的中断，这里只用到 0 位的 UIE，即使能更新中断。

- 状态寄存器（TIMx_SR）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留			CC4OF	CC3OF	CC2OF	CC1OF	保留		TIF	保留	CC4IF	CC3IF	CC2IF	CC1IF	UIF
			rc w0	rc w0	rc w0	rc w0			rc w0		rc w0	rc w0	rc w0	rc w0	rc w0

图 16.6　状态寄存器

该寄存器记录着各个触发事件志位，我们如果不使用中断，可以通过查询法来获取此时

定时器的各个标志位的状态。

- 计数器（TIMx_CNT）

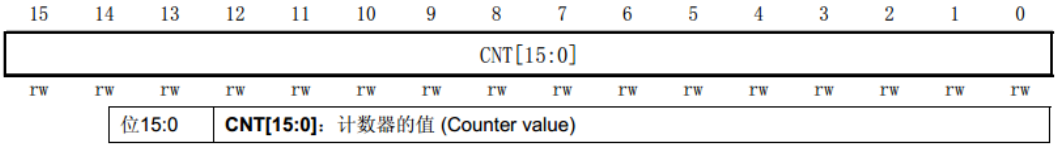


图 16.7　计数器寄存器

- 预分频器(TIMx_PSC)

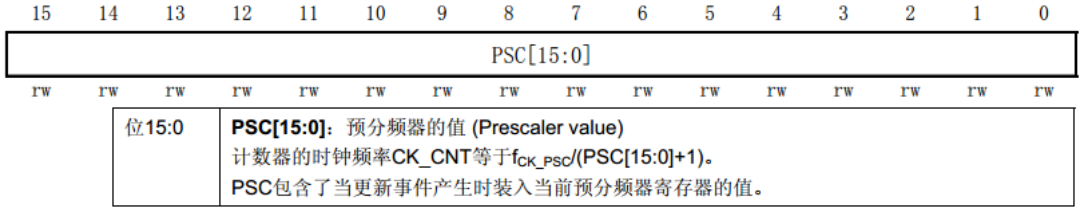


图 16.8　预分频寄存器

用于设置预分频值。

- 自动重装载寄存器(TIMx_ARR)

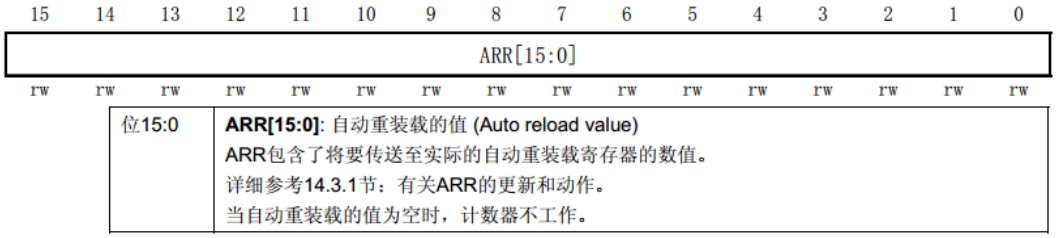


图 16.9　自动重装载寄存器

下面说一下软件配置步骤：

1. 配置 NVIC，开启 TIM3 中断，并为其分配抢占优先级和子优先级；
2. 配置定时器初始化结构，主要内容包括：计数方向，分频值，重装载值；
3. 初始化定时器配置；
4. 使能定时器更新中断，即溢出时产生中断

16.3 程序讲解

- 定时器初始化函数

```
16  /**
17   * 功能：初始化定时器
18   * 参数：
19   *      TIMx：指定待设置的定时器，TIM1-TIM4
20   *      prescaler：设置预分频值 0-65535
21   *      period：设置中断周期，即设置重装载寄存器的值 0-65535
22   *      IT_Source:中断源，比如更新中断，四个通道的输入比较中断,取值查看：TIM_interrupt_sources
23   *      NewState：是否使能IT_Source参数指定的中断，ENABLE,DISENABLE
24   * 返回值：None
25   */
26  void initTIMx(TIM_TypeDef* TIMx,u16 prescaler,u16 period,u16 IT_Source,FunctionalState NewState)
27  {
28      TIM_TimeBaseInitTypeDef  TIM_TimeBaseStructure;
29
30      switch((u32)TIMx)
31      {
32          case (u32)TIM1: RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1, ENABLE);break; //开启定时器1时钟
33          case (u32)TIM2: RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM2, ENABLE);break; //开启定时器2时钟
34          case (u32)TIM3: RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM3, ENABLE);break; //开启定时器3时钟
35          case (u32)TIM4: RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4, ENABLE);break; //开启定时器4时钟
36          /*其他密度的单片机对case进行删减即可兼容，本程序针对中密度单片机*/
37          default  :                                                  break;
38      }
39
40      TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;          //向上计数模式
41      TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;              //设置时钟分频因子，该分频不是预分频值，要区分
42      TIM_TimeBaseStructure.TIM_Period = period;                           //设置定时器周期
43      TIM_TimeBaseStructure.TIM_Prescaler =prescaler;                      //设置预分频值
44      TIM_TimeBaseInit(TIMx, &TIM_TimeBaseStructure);                      //生效更改
45
46      TIM_Cmd(TIMx, ENABLE);                                               //开启计数器
47
48      TIM_ITConfig(TIMx,IT_Source,NewState);                              //使能定时器更新中断
49  }
```

图 16.10 定时器初始化函数

为了达到更好的兼容性，我们对定时器的初始化做了兼容封装，用户只需要指定哪个定时器要设置以及设置参数即可。该部分程序的重点是配置定时器初始化结构体，从注释可以看到程序中设置了向上计数值，预分频器的值，定时器周期，时钟分频因子是指定义在定时器时钟(CK_INT)频率与数字滤波器(ETR， TIx)使用的采样频率之间的分频比例，大家无需了解，直接默认即可，然后生效的更改，接着我们还要做两个工作，一个是打开定时器（第 42 行），另一个是使能定时器更新中断。

- 设置定时器预分频值

```
54  void setPeriod(TIM_TypeDef* TIMx,u16 period)
55  {
56      TIM_SetAutoreload(TIMx, period);  //设置重装载值
57  }
```

图 16.11 设置定时器周期

该函数用于设置中断周期，即计数器内要计数的值，我们在函数中调用了 ST 标准库提供的设置重装载寄存器函数。

- 设置定时器的分频倍数

```
66  void setPrescaler(TIM_TypeDef* TIMx,u16 prescaler)
67  {
68      /*设置预分频值等于fCK_PSC/(PSC[15:0]+1)
69       *fCK_PSC为内部输入时钟，默认72MHz，假如我们想要72000倍分频，即时钟周期1ms
70       *则我们传入的值应该是71999*/
71      TIM_PrescalerConfig(TIMx, prescaler,TIM_PSCReloadMode_Update); //设置定时器周期
72  }
```

图 16.12 设置定时器分频倍数

该函数直接影响定时器时钟的频率，内部时钟通过定时器的预分频器为定时器提供时钟，默认内部时钟是 72MHz，但是 72 并不适合整数运算，因此我们把分频值设置为 7200，即 100us，另外要注意设置 7200 分频值时，我们传入的参数是 7199（根据公式计算得到），其实差 1 影响不大。

- 定时器 3 中断服务函数

```
75  /*************************************定时器中断服务函数*********************************************/
76  void TIM3_IRQHandler(void)
77  {
78      static u32 times = 0;
79      if (TIM_GetITStatus(TIM3, TIM_IT_Update) == SET)  //由于定时器中断源很多，因此要判断是哪个中断源触发的中断
80      {
81          TIM_ClearITPendingBit(TIM3, TIM_IT_Update  );  //软件清除中断挂起位，否则会一直卡死在中断服务函数
82          toggleLED();
83          printf("Curren time is %d\n",times++);
84      }
85  }
```

图 16.13 定时器中断服务函数

当定时器更新中断产生后，会跳到该函数执行，在该函数中，我们翻转了 LED，打印了中断的次数，注意 static 修饰的局部变量的存活周期和全局变量相同。

- 开启 NVIC 定时器 3 的全局中断

```
18  void initNVIC(u32 NVIC_PriorityGroup)
19  {
20      NVIC_PriorityGroupConfig(NVIC_PriorityGroup); //中断分组
21      setNVIC(TIM3_IRQn,0,1,ENABLE);                //使能定时器3总中断
22  }
```

图 16.14 定时器中断使能

在这里我们定义了 TIM3 的抢占优先级和子优先级。

- 主函数

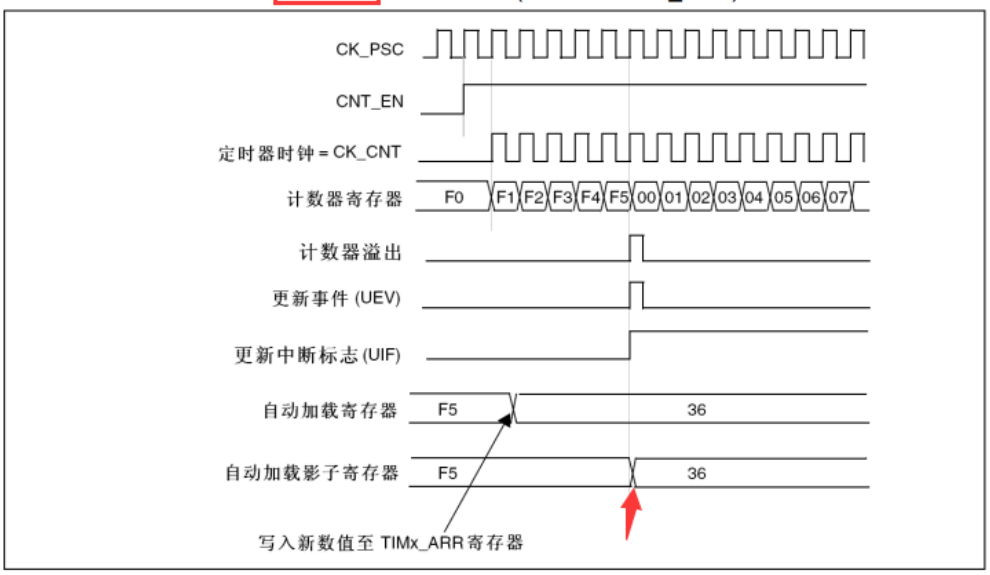
```
25      initNVIC(NVIC_PriorityGroup_2);
26      initTIMx(TIM3,7199,999,TIM_IT_Update,ENABLE);
27      TIM_ARRPreloadConfig(TIM3,ENABLE);
28      while (1)
29      {
30          key_value = getKeyValue(KEY_PRESS);
31
32          if(key_value==KEY_UP)               //UP键按下
33          {
34              setPeriod(TIM3,1000);
35          }else if(key_value==KEY_DOWN)       //DOWN键按下
36          {
37              setPeriod(TIM3,8000);
38          }else                               //没有按键按下或全按时不进行任何操作
39          {
40
41          }
42      }
```

图 16.15 主函数

定时器被初始化为 7200 倍分频（100us），周期计数值为 999（100ms）。接着就是按键采集，不同的键值对应不同的中断周期，分别是 100ms 和 800ms。

大家可能注意到了箭头所指的函数，它用于设置控制寄存器 1 中的第 7 位 ARPE，STM32 定时器的重装载寄存器其实有两个，我们平时操作的是重装载寄存器 TIMx_ARR，另外一个是被 ST 隐藏的重装载影子寄存器，真正和计数器打交道的是重装载影子寄存器，当计数器记满，其实是重装载影子寄存器把自身的值赋给了计数器，让其重新计数的，TIMx_ARR 相当于是影子寄存器的一个缓冲，是否使用这个缓冲要通过 ARPE 位来设置。当 ARPE 为 1 则用户写入的数据存储在 TIMx_ARR，此时要等到定时器产生更新事件时才把 TIMx_ARR 中的值写入到影子寄存器，当 ARPE 为 0 时，我们对 TIMx_ARR 寄存器写入的值直接存储到影子寄存器，结合下图理解一下这个过程：

图106 计数器时序图，当ARPE=1时的更新事件(预装入了TIMx_ARR)



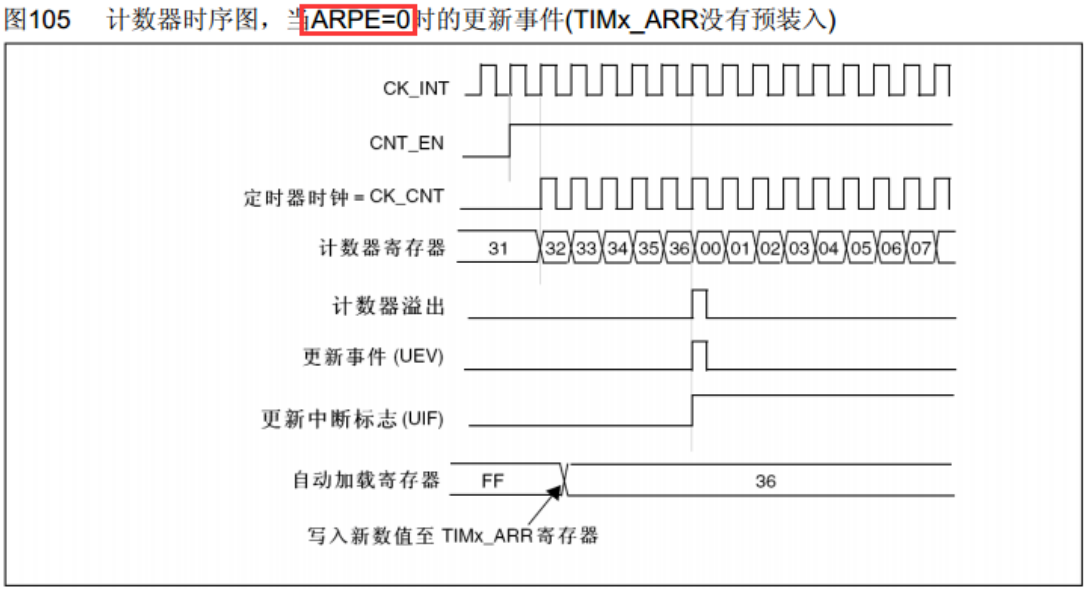


图 16.16 ARPE 示意图

由上图可以看出，直接操作影子寄存器（ARPE=0）的实时性更好，但是这种操作有一定危险。当 ARPE=0 这种情况下，我们在定时器 800ms 的间隔下更改重装载值改为 100ms 时，计数器要计到 65535 才会更新重装载值！其原因是将影子寄存器中的值改小时，恰巧计数器此时的值大于影子寄存器中的值，因此计数器要计到最大值，然后从 0 开始，所以我们要通 TIM_ARRPreloadConfig()函数将 ARPE 设置为 1，即使用预装载寄存器并在定时器发生更新时更改重装载值。最后编译链接-烧录-重启观察效果。

第十七章 定时器_实现定时采集按键

17.1 项目要求

使用定时器定时采集按键值，UP 键被按下主函数的 LED 翻转周期为 100ms，DOWN 键被按下，LED 翻转周期为 800ms。

注：本章用到核心板，对应例程 11。

17.2 程序讲解

我们直接看程序：

- 主函数

```
    while (1)
    {
        if(key_value==KEY_UP)               //UP键按下
        {
            xms = 100;
        }else if(key_value==KEY_DOWN)       //DOWN键按下
        {
            xms = 800;
        }else                               //没有按键按下或全按时不进行任何操作
        {

        }
        toggleLED();
        Delay_ms(xms);
    }
}
```

图 17.1 主函数

该项目中定时器初始化的中断周期为 30ms（按键采集最长时间为 20ms）。

- 定时计数器中断服务函数

```
void TIM3_IRQHandler(void)
{
    static u32 times = 0;
    u8 key_cache = 0;

    if (TIM_GetITStatus(TIM3, TIM_IT_Update) == SET)
    {
        TIM_ClearITPendingBit(TIM3, TIM_IT_Update  );
        key_cache = getKeyValue(KEY_RELEASE);
        if(key_cache != KEY_NO)
        {
            key_value = key_cache;
        }

    }
}
```

图 17.2　定时器中断服务函数

这里要说两点：

第一个红色圈中部分，该部分的作用就是避免采集到的按键键值被覆盖，因为主函数最短的运行周期也要 100ms，假设定时器中断服务函数中采集到的键值此时是 DOWN 键，DOWN 键赋给全局变量 key_value，但是假设主函数此时才执行到 20ms 处，也就是说还要等到 80ms 后才能更新 LED 的延时变量 xms，那么在此期间定时器中断还会发生（30ms 一次），而如果此时按键没有按下，则返回键值将是 KEY_NO，因此到更新 xms 变量的时候就导致 LED 没有任何变化。看下图加深理解：

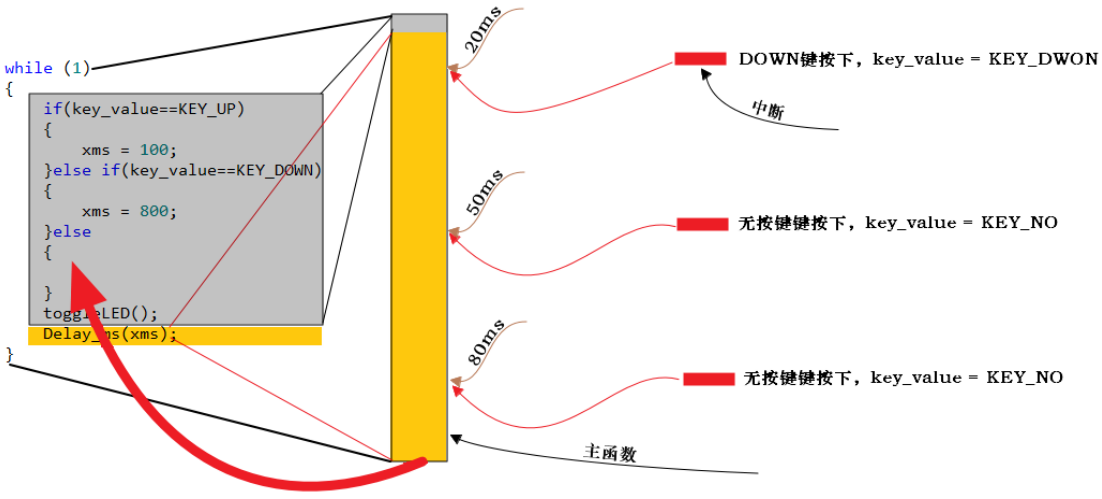


图 17.3　程序执行流程

因此加上该段代码，可以过滤掉按键没有按下的情况。

第二个要说的就是此时使用的采集按键函数的模式时按下生效，因此不存在 while 等待，如果此时将 KEY_PRESS 改为 KEY_RELASE，运行之后会发现，如果按键不松开就会一直

卡死在按键采集上，因此这里要告诫大家：“定时器中断服务函数的执行时间要小于定时器中断周期。”

关于定时器的定时任务就讲解到这里，下一章我们来破解家用遥控器的奥秘。

第十八章 定时器_输入捕获

18.1 项目要求

本实验例程要实现如下功能：

1. 通过输入捕获来解码 NEC 红外遥控器键值
2. 使用 OLED 和串口来显示和打印解码后的键值
3. 通过 EQ 键（键值 0x09）控制 LED 的翻转

注：本章用到核心板+扩展板，对应例程 12。

18.2 原理讲解

18.2.1 NEC 红外遥控器

红外遥控器广泛应用于家用电器的控制，比如电视机，空调都是使用的红外编码来进行信息传输的，红外遥控器由两部分组成，一个是发射端（遥控器），一个是接收端（家用电器），发射端是通过人眼不可见的红外光来对信息进行编码的，一般使用 38KHz 1/3 占空比的载波进行信息的调制，如下图：

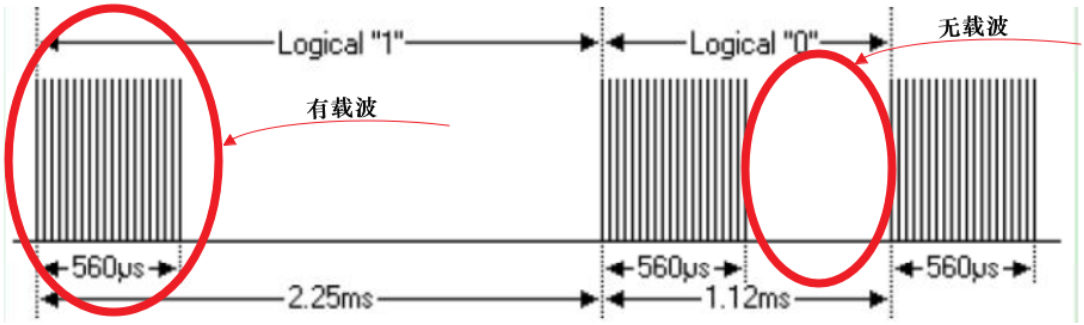


图 18.1 载波示意图

有载波时表示逻辑 1，无载波表示逻辑 0，然后通过逻辑 0 和 1 的不同组合就衍生出了不同的红外协议，比如上图举例的是 NEC 红外编码协议。在硬件上，我们使用下面的驱动电路来产生红外载波：

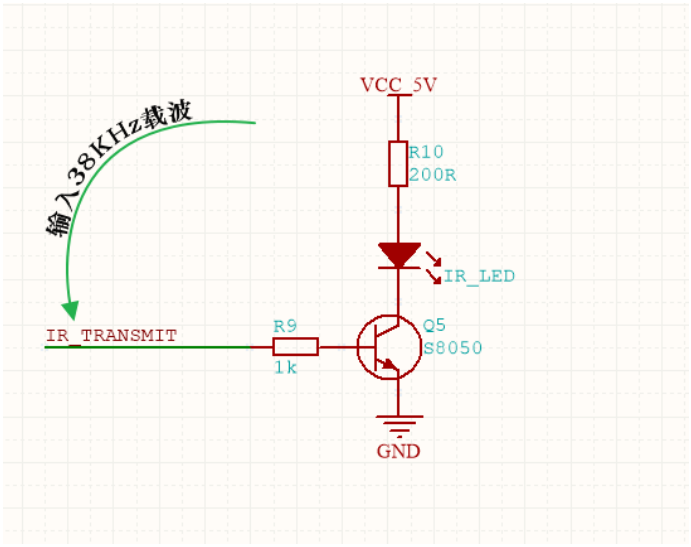


图 18.2　红外发射驱动电路原理图

这里大家可能会有两个疑问：

1. 为什么使用载波？

答：使用载波的目的有两个，第一个是通过载波可以降低外界干扰。如果我们直接通过 LED 通断来作为逻辑 0 和 1 的话，外界存在的其他红外干扰源会将传输的信息打乱，如下图：

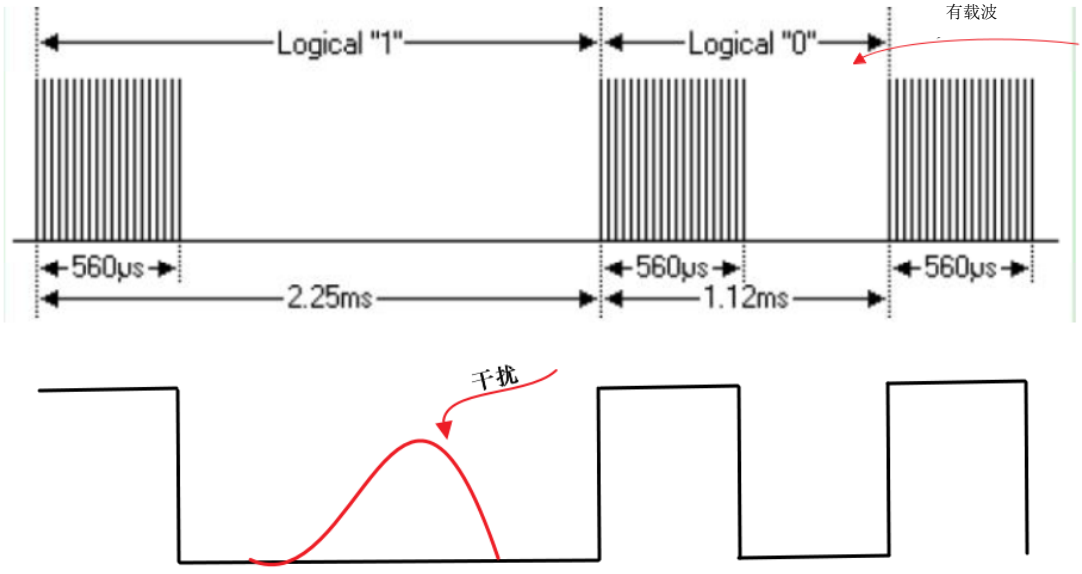


图 18.3　干扰对有无载波的影响

这将会传输一个无意义或者违背使用者控制意图的信息。而用载波调制之后传输就会变得相对安全，因为红外接收部分有带通滤波器，它只会让和载波频率相近的红外信号通过，因此其他频率的干扰源就被过滤掉了，而不加调制的话，接收端是无法做到 0Hz 的带通滤波器的，另外接收头其实就是一个使用运放搭建的反馈电路，无论载波的信号强弱，到最后输出都会把它变为恒定的电压，因此使用载波时，传输距离也会变远。第二个好处就是低功耗，

因为使用 1/3 占空比的载波和不是用载波在工作相同时间时，有载波的会更省电。

除此之外红外遥控器的好处是不同房间的同一个家用电器不会受到干扰。

2. 载波频率为什么是 38KHz？

答：红外遥控的载波不止有 38KHz 一种，只不过 38KHz 是最常用的，因为发射电路使用的晶振是 455KHz 的，经过 12 分频之后得到 37.9KHz 约等于 38KHz，另外红外发射频率和红外接收头频率要保持一致。

我们接下来讲解一下 NEC 红外协议的时序：

图 18.4 NEC 协议时序图

时序的开头由 9ms 的逻辑 1 和 4.5ms 的逻辑 0 组成，该开头不代表任何数据，知识告诉接收端我要开始传输了。

接下来的部分由“一字节地址码+一字节地址反码+一字节命令码+一字节命令反码”组成，反码的作用是用于校验数据是否正确，如果想要表达更多的信息，可以不使用反码。地址码也可以叫用户编码，它类似于设备配对的指纹，通过它可以单独控制某个家用电器，而其他电器不受影响。命令码就是遥控器的键值，受控设备通过键值来执行对应动作。

二进制的 0 和 1 是通过逻辑 0 和 1 的不同组合时间来区分的，如下：

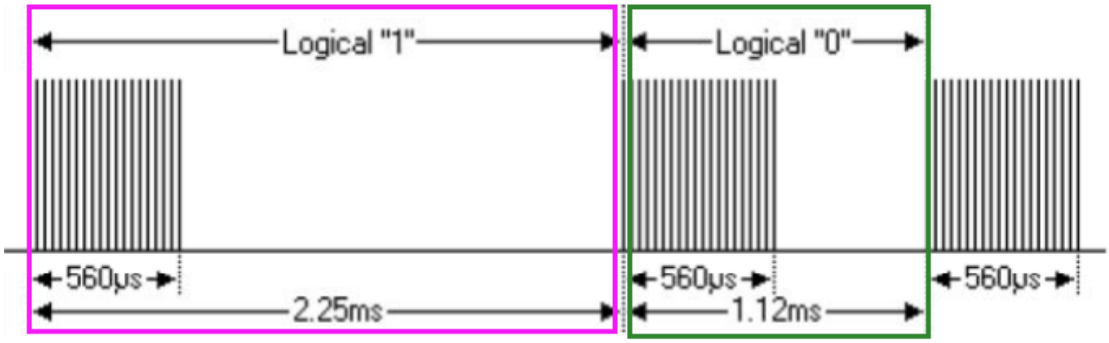


图 18.5 二进制 0 和 1 的组成

NEC 红外协议最先传输的是字节的低位，比如上面例子中的地址：10011010，其真实值时：01011001。

我们都有过这种经历，长时间按住遥控器的音量键不松开的话，音量会递增或者递减，这是因为在发送一个完整数据帧之后，如果按键还在按下，那么遥控器会每隔 110ms 发送一个重复码信号，如下：

图 18.6　重复码

重复码时序由 9ms 逻辑 1+2.25ms 逻辑 0+0.56ms 逻辑 1 组成：

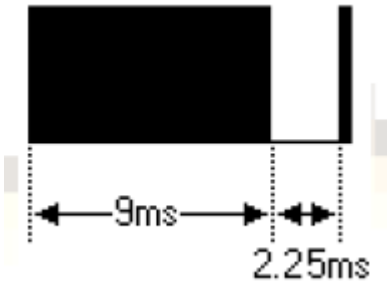


图 18.7　重复码标志信号

这里要注意，在接收端接收的电平和发送端是相反的，比如起始信号，发送端是 9ms 逻辑 1 加 4.5ms 逻辑 0，对应接收端的输出电压是 9ms 逻辑 0 加 4.5ms 逻辑 1，下面是红外接收电路：

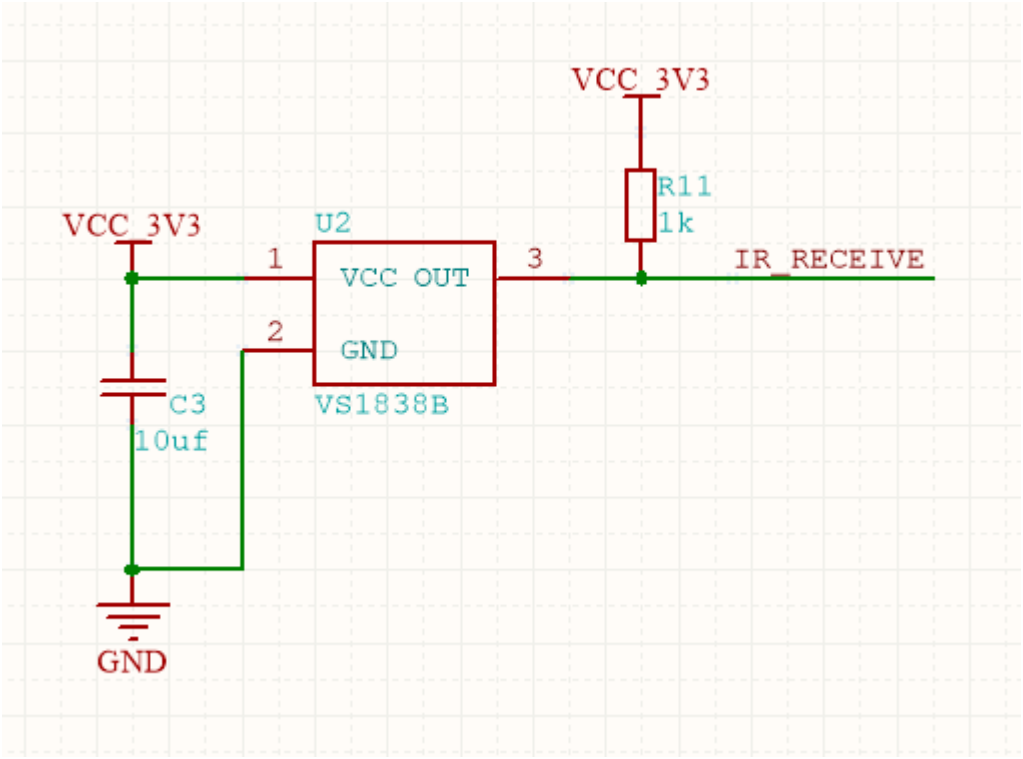


图 18.8　红外接收电路原理图

关于 NEC 红外协议就介绍到这里，红外的协议还有很多，比如 RC5，RC6，ITT，JVC 等，大家可以查看我们提供的红外相关资料进行理解。

18.2.2 定时器输入捕获

在上一章大家已经了解了定时器的三大法宝“定时中断”、“输入捕获”、“比较输出”。这章我们讲解一下输入捕获，使用输入捕获可以侦测外部数字信号的周期，占空比，如果再加上某些协议时序的判断，既可以作为一个逻辑分析仪来用，本章使用输入捕获来解析 NEC 红外协议逻辑，进而解码出遥控器键值。

按照习惯，我们先了解一下与输入捕获相关的寄存器。

作为定时器计数器基础单元的计数器，重装载寄存器，预分频寄存器，中断使能寄存器等，大家直接参考第十七章即可，这里不再重复。

- 捕获/比较模式寄存器 n(TIMx_CCMRn),n = 1,2

  我们以 TIMx_CCMR2 为例，因为我们用到的是定时器 4 的第 3 通道：

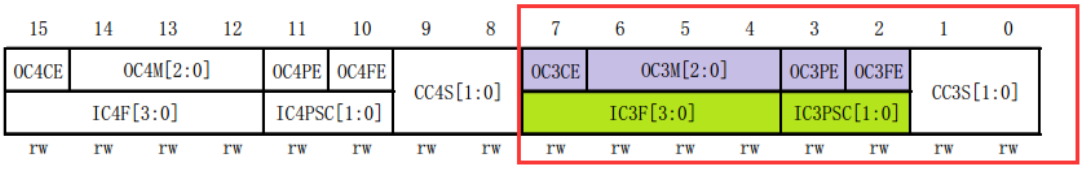


图 18.9 捕获/比较模式寄存器

STM32 的通用定时计数器（TIM2/3/4），每个都有 4 个通道，每个通道可单独配置为捕获输入或者比较输出，TIMx_CCMR1/2 用于配置这 4 个通道的工作模式，每个寄存器负责两个通道，我们以通道 3 为例：

位7:4	**IC3F[3:0]**：输入捕获3滤波器 (Input capture 3 filter)
位3:2	**IC3PSC[1:0]**：输入/捕获3预分频器 (Input capture 3 prescaler)
位1:0	**CC3S[1:0]**：捕获/比较3选择 (Capture/Compare 3 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00：CC3通道被配置为输出； 01：CC3通道被配置为输入，IC3映射在TI3上； 10：CC3通道被配置为输入，IC3映射在TI4上； 11：CC3通道被配置为输入，IC3映射在TRC上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 注：CC3S仅在通道关闭时(TIMx_CCER寄存器的CC3E='0')才是可写的。

图 18.10 捕获/比较模式寄存器位域

位域[1:0]用于设置工作模式，我们这里配置的是输入，映射在 TI3 上，位域[3:2]设置的是预分频器：

这2位定义了CC1输入(IC1)的预分频系数。
一旦CC1E='0'(TIMx_CCER寄存器中)，则预分频器复位。
00：无预分频器，捕获输入口上检测到的每一个边沿都触发一次捕获；
01：每2个事件触发一次捕获；
10：每4个事件触发一次捕获；
11：每8个事件触发一次捕获。

图 18.11 分频系数

我们选择每个边沿都会触发捕获。
位域[7:4]，用于设置输入捕获引脚采样频率和滤波次数：

0000：无滤波器，以f_{DTS}采样
0001：采样频率$f_{SAMPLING}=f_{CK_INT}$，N=2
0010：采样频率$f_{SAMPLING}=f_{CK_INT}$，N=4
0011：采样频率$f_{SAMPLING}=f_{CK_INT}$，N=8
0100：采样频率$f_{SAMPLING}=f_{DTS}/2$，N=6
0101：采样频率$f_{SAMPLING}=f_{DTS}/2$，N=8
0110：采样频率$f_{SAMPLING}=f_{DTS}/4$，N=6
0111：采样频率$f_{SAMPLING}=f_{DTS}/4$，N=8
1000：采样频率$f_{SAMPLING}=f_{DTS}/8$，N=6
1001：采样频率$f_{SAMPLING}=f_{DTS}/8$，N=8
1010：采样频率$f_{SAMPLING}=f_{DTS}/16$，N=5
1011：采样频率$f_{SAMPLING}=f_{DTS}/16$，N=6
1100：采样频率$f_{SAMPLING}=f_{DTS}/16$，N=8
1101：采样频率$f_{SAMPLING}=f_{DTS}/32$，N=5
1110：采样频率$f_{SAMPLING}=f_{DTS}/32$，N=6
1111：采样频率$f_{SAMPLING}=f_{DTS}/32$，N=8

图 18.12 滤波选项

比如滤波次数为 8，则表示检测到 8 个相同的信号沿时才触发一次捕获动作，其功能就是可以过滤掉外部的干扰信号，这里我们选择不滤波（0000）。

- 捕获/比较使能寄存器（TIMx_CCER）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		CC4P	CC4E	保留		CC3P	CC3E	保留		CC2P	CC2E	保留		CC1P	CC1E
		rw	rw			rw	rw			rw	rw			rw	rw

位1	**CC1P**：输入/捕获1输出极性 (Capture/Compare 1 output polarity) **CC1通道配置为输出：** 0：OC1高电平有效 1：OC1低电平有效 **CC1通道配置为输入：** 该位选择是IC1还是IC1的反相信号作为触发或捕获信号。 0：不反相：捕获发生在IC1的上升沿；当用作外部触发器时，IC1不反相。 1：反相：捕获发生在IC1的下降沿；当用作外部触发器时，IC1反相。
位0	**CC1E**：输入/捕获1输出使能 (Capture/Compare 1 output enable) **CC1通道配置为输出：** 0： 关闭－ OC1禁止输出。 1： 开启－ OC1信号输出到对应的输出引脚。 **CC1通道配置为输入：** 该位决定了计数器的值是否能捕获入TIMx_CCR1寄存器。 0：捕获禁止； 0：捕获使能。

图 18.13 捕获/比较使能寄存器

捕获/比较使能寄存器用于设置输入捕获的信号触发沿以及使能或者失能捕获功能。每个通道占用两位。

- 捕获/比较寄存器（TIMx_CCRn），n = 1,2,3,4

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR3[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**CCR3[15:0]**: 捕获/比较3的值 (Capture/Compare 3 value) **若CC3通道配置为输出：** CCR3包含了装入当前捕获/比较3寄存器的值(预装载值)。 如果在TIMx_CCMR3寄存器(OC3PE位)中未选择预装载特性，写入的数值会被立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较3寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC3端口上产生输出信号。 **若CC3通道配置为输入：** CCR3包含了由上一次输入捕获3事件(IC3)传输的计数器值。

图 18.14 捕获/比较寄存器

该寄存器存放的是捕获到的当前计数器中的值，或者和计数器进行比较的比较值。本章实验应该设置的触发信号沿时下降沿（接收端与发送端信号反向），我们通过检测相邻下降沿之间的时间长度来判断起始信号，地址码，命令码，重复码，各个码长如下（单位 ms）：

起始信号：9+4.5=13.5ms（注：接收到的并不是精确的计算值，存在一些误差）

二进制 1：0.56+1.69=2.25ms

二进制 0：0.56+0.56=1.12ms

重复码：110ms

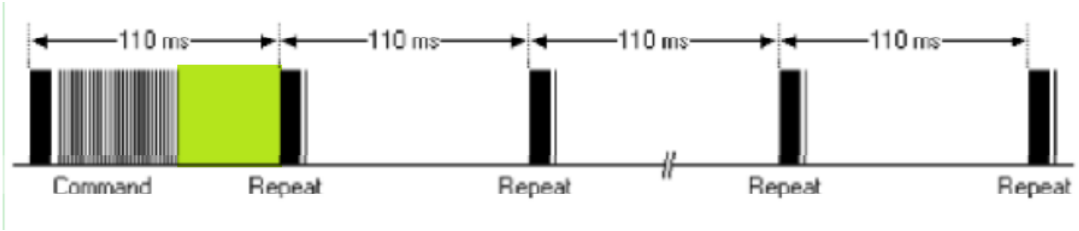


图 18.15 空闲时长示意图

配置输入捕获的步骤：

1. 配置定时计数器初始化结构体（与上一章相同）
2. 配置对应捕获引脚为浮空输入
3. 配置输入捕获初始化结构体，包括使用的通道，触发信号沿的类型，分频值，是否滤波等
4. 使能捕获比较中断和定时器更新中断（更新非必须，在本项目中需要更新中断）

18.3 程序讲解

- 定时器 4 输入捕获 3 通道初始化函数

```
51  /**
52   * 功能：初始化定时器4输入捕获
53   * 参数：
54   *       ICPolarity：信号触发沿极性，TIM_ICPolarity_Rising，TIM_ICPolarity_Falling
55   * 返回值：None
56   */
57  void initTIM4IC3(u16 ICPolarity)
58  {
59      TIM_ICInitTypeDef  TIM_ICInitStructure;                         //定义输入捕获初始化结构体
60
61      TIM_ICInitStructure.TIM_Channel = TIM_Channel_3;                //配置输入捕获通道3为输入通道
62      TIM_ICInitStructure.TIM_ICPolarity = ICPolarity;                //设置触发极性
63      TIM_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI; //将通道3连接到TI3
64      TIM_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;           //无预分频
65      TIM_ICInitStructure.TIM_ICFilter = 0x00;                        //无滤波
66      TIM_ICInit(TIM4, &TIM_ICInitStructure);                         //生效输入捕获设置
67  }
```

图 18.16 输入捕获初始化

该函数分为两部分初始化：第一部分是对应输入捕获 IO 口的初始化，我们配置为浮空输入，红外接收头数据输出端为上拉电阻，所以，空闲状态下时高电平。第二部分是输入捕获的配置，其中包括输入捕获的通道，触发的信号沿的选择，这里注意输入捕获触发的信号沿只能是上升沿和下降沿二选一（因为寄存器是通过 1 位来设置的），接着就是连接方式，连接方式设置为 TIM_ICSelection_DirectTI 时，对应的通道连接如下图箭头所示：

位1:0	**CC3S[1:0]**：捕获/比较3选择 (Capture/Compare 3 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00：CC3通道被配置为输出； 01：CC3通道被配置为输入，IC3映射在TI3上； 10：CC3通道被配置为输入，IC3映射在TI4上； 11：CC3通道被配置为输入，IC3映射在TRGI上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 注：CC3S仅在通道关闭时(TIMx_CCER寄存器的CC3E='0')才是可写的。

图 18.17 选通通道设置位域

最后设置为无滤波即可。

- NEC 初始化函数

```
15  /**
16   * 功能：初始化NEC
17   * 参数：   None
18   * 返回值：None
19   */
20  void initNEC(void)
21  {
22      GPIO_InitTypeDef GPIO_InitStructure;
23
24      /***************************红外接收IO初始化***************************/
25      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);           //开启GPIOB时钟
26
27      GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_8;                      //配置PB8
28      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;           //配置为浮空输入
29      GPIO_Init(GPIOB, &GPIO_InitStructure);
30  }
```

图 18.18 NEC 初始化函数

该函数用于初始化红外接收头的 IO 口，将 IO 口设置为浮空输入。

- NEC 码长采集以及解码

该部分处理代码位于定时器的中断服务函数中

```
121 void TIM4_IRQHandler(void)
122 {
123     static u8 times = 0;                                //捕获次数，用于标记是否是第一次捕获，0是第一次，非0不是
124     static u8 rep_times = 0;                            //用于第一次接收重复码时，将上一次重复码次数清除掉
125     static u8 i = 0;                                    //数组NEC_Buffer的索引值
126     u16 cache = 0;                                      //用于临时存放输入捕获时定时器的值
127     if(TIM_GetITStatus(TIM4, TIM_IT_CC3) == SET)        //发生捕获
128     {
129         TIM_ClearITPendingBit(TIM4, TIM_IT_CC3);        //清除捕获挂起位(标志位)
130
131         if(times == 0)                                  //第一次捕获
132         {
133             ++times;                                    //给times赋一个非0值即可
134             TIM_SetCounter(TIM4,0);                     //清零计数器，让其从0开始计数
135         }else                                           //非第一次捕获
136         {
137             cache = TIM_GetCapture3(TIM4);              //核算NEC码的时长
138
139             if(cache > 1230 && cache < 1500)            //起始码 在12.3ms和15ms之间
140             {
141                 i= 0;
142                 NEC_DataStructure.NEC_Buffer[i++] = cache;
143                 TIM_SetCounter(TIM4,0);
144             }else if(cache > 70 && cache < 270)         //二进制1或0 在0.7ms和2.7ms之间
145             {
146                 if(NEC_DataStructure.NEC_Buffer[0] != 0) //说明成功接收到起始码
```

```
147                 {
148                     NEC_DataStructure.NEC_Buffer[i++] = cache;
149                     TIM_SetCounter(TIM4,0);
150                 }else                                   //没有接收到起始码接收到了数据码，说明是干扰
151                 {
152                     times = 0;
153                     i = 0;
154                 }
155 
156             }else if(cache > 9000 && cache < 13000)      //重复码 在100ms和120ms之间
157             {
158                 if(NEC_DataStructure.NEC_Buffer[0] != 0) //接收到起始码重复 码才有效
159                 {
160                     if(rep_times==0)
161                     {
162                         NEC_DataStructure.NEC_Buffer[34] = 0;
163                         ++rep_times;
164                     }
165                     ++NEC_DataStructure.NEC_Buffer[34];  //重复码数+1
166                     TIM_SetCounter(TIM4,0);
167                 }else
168                 {
169                     times = 0;
170                     i = 0;
171                 }
172             }else                                       //无意义的码长，从零重新接
173             {
174                 times = 0;
175                 i = 0;
176             }
177 
178         }
179     }
180 
181     if(TIM_GetITStatus(TIM4, TIM_IT_Update) == SET)     //定时器溢出中断,说明已经停止接收红外码(溢出需要200ms)
182     {
183         TIM_ClearITPendingBit(TIM4, TIM_IT_Update);     //软件清除溢出挂起位
184         decodeNEC(&NEC_DataStructure,0x00);             //解码
185         rep_times = 0;
186         times = 0;
187         i = 0;
188     }
189 }
```

图 18.19 定时器 4 中断服务函数

输入捕获中断部分的代码用于获取各个 NEC 红外编码的长度，然后统一将这些时间长度存放于数组当中（等下介绍），由于每个码长（起始码，数据 0,1，重复码）虽然给出了明确的时间长度，但是实际传输时，是存在误差的（遥控器内部和外界传输原因），因此我们不能将捕获到的码长和理想值比较是否相等，如果这样，只有极小的几率才会解码成功。

定时器更新中断部分的代码用于进行 NEC 的解码，也就是说，定时器如果出现更新中断就说明此时红外接收完成，因为我们在主函数中，将定时器的更新时间设置为 200ms，而 NEC 红外编码最长的码长就是重复码的间隔，大约在 100ms 左右，我们每次捕获采集一个码长都会立即对计数器清零，也就是说，正常的捕获期间定时器不可能溢出，只有全部传输完成，才存在定时器的溢出，因此，我们选择在这个时候进行解码，当然，在平时没有捕获时，定时器也会每个 200ms 解码一次，不过对我们的结果无影响。

- 红外解码函数

```
/*********************NEC数据码结构体************************/
typedef struct
{
    u8 AddrCode_H;       //地址码
    u8 AddrCode_L;       //地址反码
    u8 CmdCode_H;        //命令码
    u8 CmdCode_L;        //命令反码

    u16 RepeatCode;      //重复码个数，该变量成员记录的是重复码的个数
    /**
     * 存放输入捕获采集到的NEC码对应时长，单位为10us
     * 1个起始码+32个二进制位码占用数组NEC_Buffer[0]-NEC_Buffer[32]
     * 重复码次数存放于数组最后一个元素中
     */
    u16 NEC_Buffer[35]; //存放捕获到的时间长度

    u8 DecodeOver;       //暂时没有用到，预留
}NEC_DataTypeDef;
```

图 18.20 NEC 红外结构体

```
/**
 * 功能：NEC解码函数
 * 参数：
 *      pNECData:存放NEC相关数据，比如捕获时长的数组、码值和重复值个数
*       Addr    :遥控器地址码（用户码），用于校验数据
 * 返回值：None
 */
void decodeNEC(NEC_DataTypeDef* pNECData,u8 Addr)
{
    u8 i,j;
    u8* pdata = (u8*)pNECData;                    //将结构体指针强制转换为u8指针，方便存放码值

    for(i=0;i<4;++i)                              //循环读取4个码值
    {
        for(j=0;j<8;++j)                          //读取单个码值
        {
            if(pNECData->NEC_Buffer[j+i*8+1] <270 && pNECData->NEC_Buffer[j+i*8+1] > 170)       //二进制1
            {
                *(pdata+i) |= 0x80>>(7-j);
            }else if(pNECData->NEC_Buffer[j+i*8+1] <170 && pNECData->NEC_Buffer[j+i*8+1] > 70)  //二进制0
            {
                *(pdata+i) &= ~(0x80>>(7-j));
            }else                                                                               //无效值
            {
                memset(NEC_DataStructure.NEC_Buffer,0,35);
                return;
            }

            if(*pdata != Addr)                                                                  //防止数据错误
            {
                *pdata = 0;
                memset(NEC_DataStructure.NEC_Buffer,0,35);
                return ;
            }
        }
    }

    memset(NEC_DataStructure.NEC_Buffer,0,35);
    pNECData->RepeatCode = pNECData->NEC_Buffer[34];                                            //重复码次数赋值
}
```

图 18.21 NEC 红外解码函数

前面介绍到的所有函数都是围绕 NEC 红外结构体来展开的，我们把 NEC 红外相关的数据封装到了结构体当中，目的是方便管理，也是前面提到的面向对象的编程思想，在该结构体中，有用于临时存放捕获到的码长的数组和解码后存放地址码和命令码的结构体成员。

接下来的函数是将码长转换为有实际意义的 NEC 数据的解码部分，该部分将结构体指针强制转换为无符号 8 位整形的指针类型(u8*)，目的是为了更方便的存放解码后的数据。结构体其实就是一个复合数据类型的数组，他们在内存空间上的存储方式是连续的，这点和数组相同，因此我们可以使用操作数组的方式去操作结构体，大家好好理解一下。然后我们分四步分别对地址码、地址反码、指令码、指令反码进行赋值，每一步又分为 8 步，这 8 步用于将数据位一个一个的解析出来然后凑成一个字节。

最后就是检验数据是否正确，然后是将重复码赋值。

- 主函数

```
41      /*使用ADC初始化函数只是为了将连接PA15的LED灯关掉*/
42      initADC();
43      formatScreen(0x00);
44      showCNString(32,0,"风媒电子",FONT_16_CN);
45      showString(0,2,"ADDR:",FONT_16_EN);
46      showString(0,4,"CMD :",FONT_16_EN);
47      while (1)
48      {
49          showNumber(40,2,NEC_DataStructure.AddrCode_H,HEX,2,FONT_16_EN);
50          showNumber(40,4,NEC_DataStructure.CmdCode_H,HEX,2,FONT_16_EN);
51          printf("AddrCode is :%02X\n",NEC_DataStructure.AddrCode_H);
52          printf("CmdCode is :%02X\n",NEC_DataStructure.CmdCode_H);
53          if(NEC_DataStructure.CmdCode_H == 0x09)  //如果是EQ键按下
54          {
55              NEC_DataStructure.CmdCode_H = 0;      //清除EQ键值，防止LED重复闪烁
56              toggleLED();
57          }
58          Delay_ms(100);
59      }
60  }
```

图 18.22 主函数

这里需要注意的是，我们因为使用了扩展板，所以 PA15 连接的 LED 就会亮起，影响我们观察 OLED，因此我们调用 ADC 初始化函数将其 PA15 关掉，原因查看 ADC 章节。

运行效果如下：

图 18.23 实验现象

按 EQ 键 LED 会闪烁。下面是遥控器的码值表：

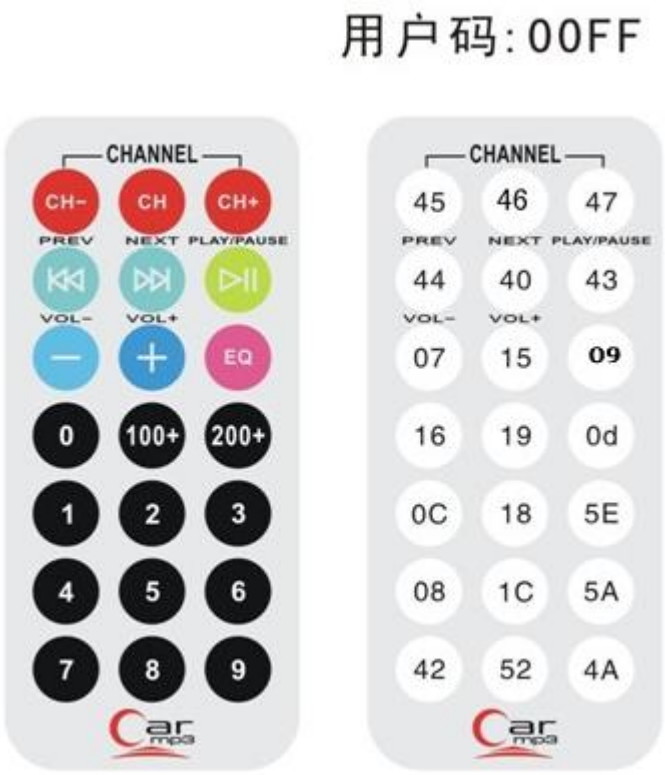


图 18.24 红外遥控器码值表

第十九章 红外遥控继电器

19.1 项目要求

使用红外遥控器的数字‘1’和数字‘2’键分别对扩展板上的继电器 1 和 2 进行开关控制。例如：按下数字 1 继电器 1 吸合，继电器 1 工作指示灯亮起，再按下数字 1 继电器 1 断开，继电器 1 工作指示灯熄灭。

注：本章用到核心板+扩展板，对应例程 13。

在实验中如果将继电器连接至市电，请注意用电安全，切勿触碰扩展板背面继电器触点，防止发生触电！

19.2 原理讲解

继电器可以说是电子工程师控制强电的一个桥梁，在物联网方面应用也极其广泛，因为大多数电器都是 220V 的市电，所以只能通过继电器控制通断。继电器目前在原理上分为两种：第一种是机械继电器，通俗一点就是工作时能听到声音的继电器，它是利用电磁原理制成的，当通电会后，电磁线圈产生磁场，将衔铁吸引，进而使常开触点接触闭合，使常闭触点断开，如下图：

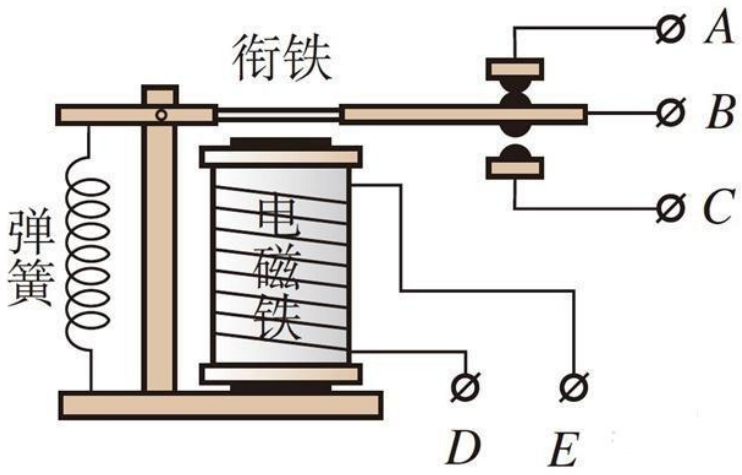


图 19.1 机械继电器原理图

另外一种是固态继电器，它是由开关三极管、双向可控硅等半导体器件组成，和机械继电器相比，固态继电器工作寿命长，安静，无触点火花，但是价格会贵一些，我们扩展板采用的是松乐的机械继电器，原理图如下：

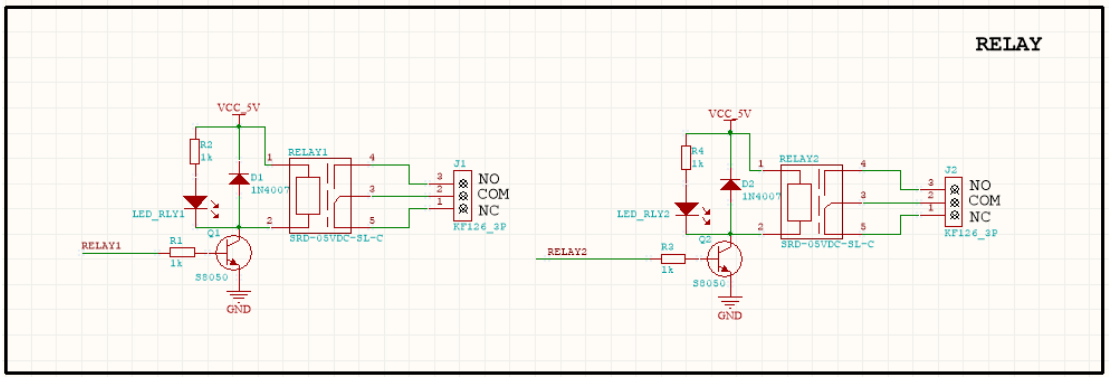


图 19.2　继电器驱动电路原理图

这里和大家说一下 D1，D2 二极管的作用，在继电器应用中该二极管称为续流二极管，用于衰减继电器断电后产生的高压脉冲。继电器线圈是感性器件，感性器件有着抵抗电流变化的特性，因此当电流突然消失时，电感线圈会在之前电流方向产生一个高压脉冲，如下图所示，该高压脉冲通常是高于几倍的电源电压，因此长时间频繁吸合所产生的高压脉冲会对三极管反复造成冲击，缩短三极管寿命，甚至击穿。

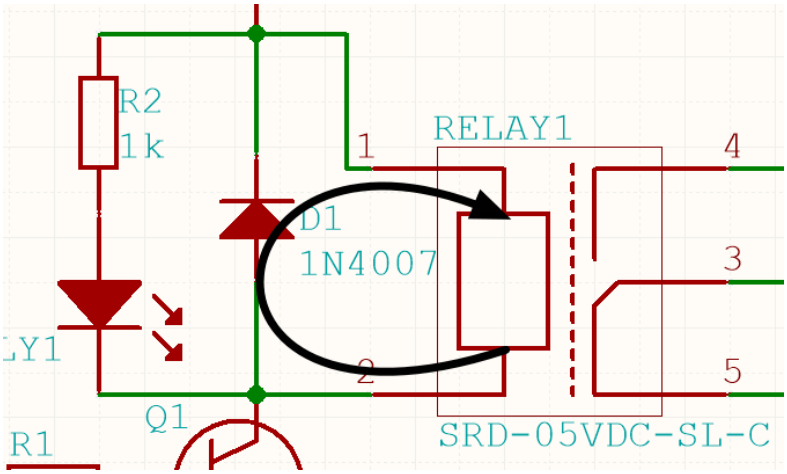


图 19.3　高压脉冲流动方向

因此我们加入了二极管，在正常工作时，二极管处于截止状态，无作用，当继电器突然断电，所产生的高压脉冲会使二极管导通，从而产生一个电流回路，将电流消耗掉，在应用中，也可以再和二极管串联一个电阻，起到消耗电流的作用。

在工业应用中，会对上面的电路进行升级，即在加上光耦隔离，使强弱电彻底分开。

控制继电器和控制 LED 一样简单，只要给三极管基极一个高电平，继电器常开就会吸合，反之低电平断开。下面直接看程序。

19.3 程序讲解

- 继电器初始化函数

```
17  void initRelay(void)
18  {
19      GPIO_InitTypeDef GPIO_InitStructure; //定义GPIO初始化结构体
20
21      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB, ENABLE); //使能GPIO和复用外设时钟
22
23      GPIO_InitStructure.GPIO_Pin = RELAY1;               //设置继电器1对应引脚
24      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;        //设置推挽输出
25      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;       //工作速度50MHz
26      GPIO_Init(GPIOA, &GPIO_InitStructure);                  //设置生效
27      GPIO_ResetBits(GPIOA, RELAY1);                      //默认低电平，继电器1不工作
28
29      GPIO_InitStructure.GPIO_Pin = RELAY2;               //设置对应引脚
30      GPIO_Init(GPIOB, &GPIO_InitStructure);                  //设置生效
31      GPIO_ResetBits(GPIOB, RELAY2);                      //默认低电平，继电器2不工作
32  }
```

图 19.4 继电器初始化函数

和 LED 初始化一样，默认关闭继电器。

- 翻转继电器函数

```
39  void toggleRelay(u16 xRelay)
40  {
41      if(xRelay == RELAY1)
42      {
43          GPIOA->ODR ^= RELAY1;
44      }else if(xRelay == RELAY2)
45      {
46          GPIOB->ODR ^= RELAY2;
47      }else
48      {
49
50      }
51  }
```

图 19.5 继电器翻转函数

继电器状态翻转。

- 设置继电器状态

```
60  void setRelay(u16 xRelay,RELAY_STATUS sta)
61  {
62      if(xRelay == RELAY1)
63      {
64          if(sta == RELAY_OPEN)
65          {
66              GPIO_ResetBits(GPIOA, RELAY1);
67          }else if(sta==RELAY_CLOSE)
68          {
69              GPIO_SetBits(GPIOA, RELAY1);
70          }else
71          {
72
73          }
74
75      }else if(xRelay == RELAY2)
76      {
77          if(sta == RELAY_OPEN)
78          {
79              GPIO_ResetBits(GPIOB, RELAY2);
80          }else if(sta==RELAY_CLOSE)
81          {
82              GPIO_SetBits(GPIOB, RELAY2);
83          }else
84          {
85
86          }
87
88      }else
89      {
90
91      }
92  }
```

图 19.6 设置继电器状态

设置指定继电器状态。

- 主函数

```
36      while (1)
37      {
38          if(NEC_DataStructure.CmdCode_H == 0x0C)        //'1'键按下
39          {
40              toggleRelay(RELAY1);
41              NEC_DataStructure.CmdCode_H = 0;           //清除键值，防止继电器重复动作
42
43          }else if(NEC_DataStructure.CmdCode_H == 0x18) //'2'键按下
44          {
45              toggleRelay(RELAY2);
46              NEC_DataStructure.CmdCode_H = 0;           //清除键值，防止继电器重复动作
47          }else                                          //其他键值，啥也不干
48          {
49
50          }
51
52          toggleLED();
53          Delay_ms(100);
54      }
55  }
```

图 19.7 主函数业务逻辑循环

主函数部分和上一章通过 EQ 键控制 LED 的原理是一样的，这里不做过多讲解。以下是测试结果（**再次提醒大家，连接市电请小心！**）：

图 19.8 运行效果

第二十章 定时器_输出比较

20.1 项目要求

使用定时器的输出比较功能产生 PWM 来驱动无源蜂鸣器，使用红外遥控器的 CH+/-键来设置蜂鸣器的频率的升高或者降低，使用外红遥控器的+/-键来设置蜂鸣器的占空比的大小，同时串口打印此时的蜂鸣器的频率(音调)和音量 。

注：本章用到核心板+扩展板，对应例程 14。

20.2 原理讲解

20.2.1 认识 PWM

物联网的硬件设备，无外乎就是将采集到的外界数据上传服务器或者根据服务器下发的指令控制设备，因此，采集、控制和联网是物联网硬件设备的三大特性。

对于控制有两种，第一种就是二进制信号量的控制，即控制设备的开或者关，比如第十九章的继电器，另外一种就是线性控制，比如控制灯光的亮暗，电机的转速，本章就来讲解第二种控制。在线性控制中采用最多的方式就是使用 PWM 波进行控制。

PWM（Pulse Width Modulation，脉冲宽度调制）是一种周期和占空比可变的方波，因此在 PWM 的调制上也分为两种：

周期调制：通过改变周期即频率可以控制蜂鸣器发出不同音调的声音，也可以调制不同频率的信号（比如用 PWM 来模拟 NEC 红外协议），另外现在很多照明厂商宣传自己的 LED 照明产品无频闪（打开手机照相功能，如果发现光源闪烁就是有频闪，比如传统的白炽灯），其原理就是将 LED 的驱动信号频率设置的高一些而已，比如普遍在 1KHz。

占空比调制：占空比就是同一周期的高电平持续时间和周期的比值，比如周期 100ms，高电平持续时间为 75ms，低电平持续时间为 25ms，则占空比为 75%。如下图所示：

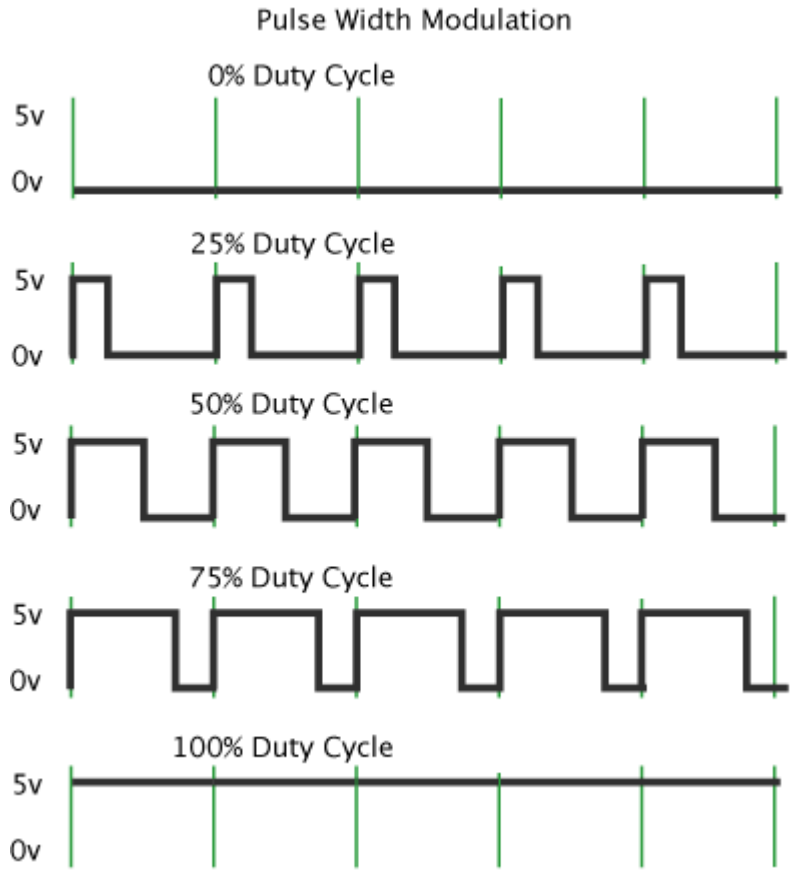


图 20.1 PWM 占空比示意图

占空比越高对应的等效电压就越高，反之占空比越低，对应的等效电压越低。因此通过调节占空比的大小可以控制灯光的亮度和电机的转速等。平时我们拿 PWM 作为一种数模转换器来用（DAC）。

20.2.2 输出比较寄存器讲解

通常使用定时器的输出比较功能来产生 PWM，原理如下图：

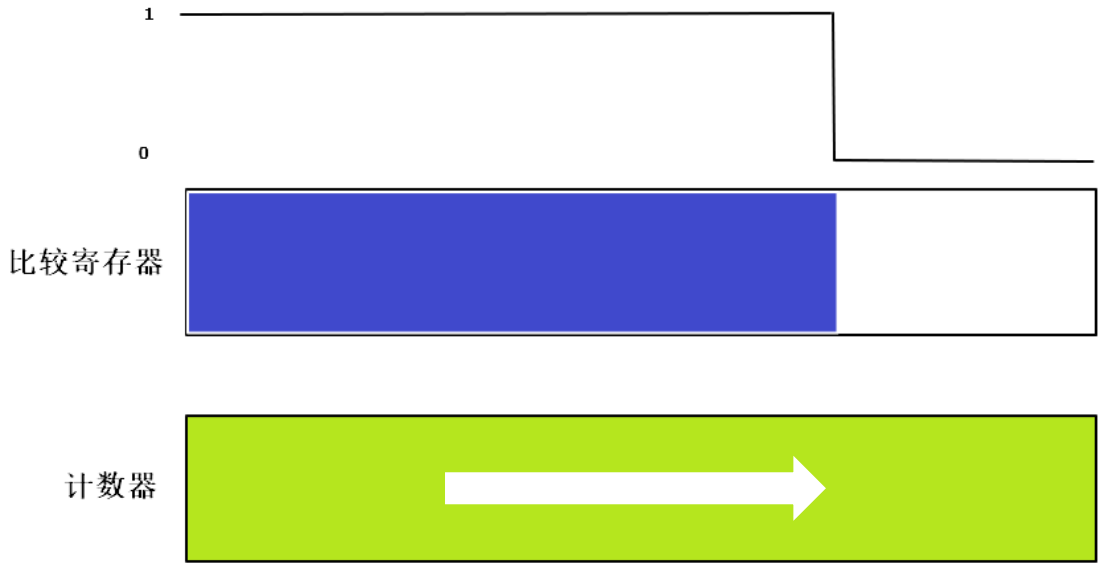


图 20.2　PWM 产生原理

这里最核心的就是两个寄存器，一个是计数器，一个是比较寄存器，计数器的计数周期值对应的就是 PWM 的周期，比较寄存器当中的数值就是我们设置的占空比，当计数器的值小于比较寄存器当中的值时输出高电平（也可输出低电平要看寄存器设置），反之，当计数器的值大于比较寄存器中的值时输出就为低电平（也可输出高电平要看寄存器设置），比如计数器的周期是 100，比较寄存器中存放的值时 75，那么当计数值在 0-75 时输出高电平，在 76-100 期间输出低电平。

接下来看一下和输出比较相关的寄存器：

计数器、重装载寄存器，控制寄存器等在第十六章已经讲解过，这里不做赘述。

- 捕获/比较模式寄存器 n（TIMx_CCMRn），n = 1,2

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OC2CE	OC2M[2:0]			OC2PE	OC2FE	CC2S[1:0]		OC1CE	OC1M[2:0]			OC1PE	OC1FE	CC1S[1:0]	
IC2F[3:0]				IC2PSC[1:0]				IC1F[3:0]				IC1PSC[1:0]			
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位7	**OC1CE**：输出比较1清0使能 (Output compare 1 clear enable) 0：OC1REF 不受ETRF输入的影响； 1：一旦检测到ETRF输入高电平，清除OC1REF=0。
位6:4	**OC1M[2:0]**：输出比较1模式 (Output compare 1 enable) 该3位定义了输出参考信号OC1REF的动作，而OC1REF决定了OC1的值。OC1REF是高电平有效，而OC1的有效电平取决于CC1P位。 000：冻结。输出比较寄存器TIMx_CCR1与计数器TIMx_CNT间的比较对OC1REF不起作用； 001：匹配时设置通道1为有效电平。当计数器TIMx_CNT的值与捕获/比较寄存器1(TIMx_CCR1)相同时，强制OC1REF为高。 010：匹配时设置通道1为无效电平。当计数器TIMx_CNT的值与捕获/比较寄存器1(TIMx_CCR1)相同时，强制OC1REF为低。 011：翻转。当TIMx_CCR1=TIMx_CNT时，翻转OC1REF的电平。 100：强制为无效电平。强制OC1REF为低。 101：强制为有效电平。强制OC1REF为高。 110：PWM模式1－ 在向上计数时，一旦TIMx_CNT<TIMx_CCR1时通道1为有效电平，否则为无效电平；在向下计数时，一旦TIMx_CNT>TIMx_CCR1时通道1为无效电平(OC1REF=0)，否则为有效电平(OC1REF=1)。 111：PWM模式2－ 在向上计数时，一旦TIMx_CNT<TIMx_CCR1时通道1为无效电平，否则为有效电平；在向下计数时，一旦TIMx_CNT>TIMx_CCR1时通道1为有效电平，否则为无效电平。 注1：一旦LOCK级别设为3(TIMx_BDTR寄存器中的LOCK位)并且CC1S='00'(该通道配置成输出)则该位不能被修改。 注2：在PWM模式1或PWM模式2中，只有当比较结果改变了或在输出比较模式中从冻结模式切换到PWM模式时，OC1REF电平才改变。
位3	**OC1PE**：输出比较1预装载使能 (Output compare 1 preload enable) 0：禁止TIMx_CCR1寄存器的预装载功能，可随时写入TIMx_CCR1寄存器，并且新写入的数值立即起作用。 1：开启TIMx_CCR1寄存器的预装载功能，读写操作仅对预装载寄存器操作，TIMx_CCR1的预装载值在更新事件到来时被传送至当前寄存器中。 注1：一旦LOCK级别设为3(TIMx_BDTR寄存器中的LOCK位)并且CC1S='00'(该通道配置成输出)则该位不能被修改。 注2：仅在单脉冲模式下(TIMx_CR1寄存器的OPM='1')，可以在未确认预装载寄存器情况下使用PWM模式，否则其动作不确定。
位2	**OC1FE**：输出比较1 快速使能 (Output compare 1 fast enable) 该位用于加快CC输出对触发器输入事件的响应。 0：根据计数器与CCR1的值，CC1正常操作，即使触发器是打开的。当触发器的输入出现一个有效沿时，激活CC1输出的最小延时为5个时钟周期。 1：输入到触发器的有效沿的作用就象发生了一次比较匹配。因此，OC被设置为比较电平而与比较结果无关。采样触发器的有效沿和CC1输出间的延时被缩短为3个时钟周期。 该位只在通道被配置成PWM1或PWM2模式时起作用。
位1:0	**CC1S[1:0]**：捕获/比较1 选择 (Capture/Compare 1 selection) 这2位定义通道的方向(输入/输出)，及输入脚的选择： 00：CC1通道被配置为输出； 01：CC1通道被配置为输入，IC1映射在TI1上； 10：CC1通道被配置为输入，IC1映射在TI2上； 11：CC1通道被配置为输入，IC1映射在TRC上。此模式仅工作在内部触发器输入被选中时(由TIMx_SMCR寄存器的TS位选择)。 注：CC1S仅在通道关闭时(TIMx_CCER寄存器的CC1E='0')才是可写的。

图 20.3　捕获/比较模式寄存器

捕获/比较寄存器在输入捕获章节已经介绍过了，不过我们介绍的是该寄存器和捕获相

关的第二行，现在来看一下输出比较相关的第一行。

我们以输出捕获 1 为例来讲解，CC1S 设置通道方向，这里设置为 00，表示配置为输出比较模式。OC1FE 用于设置比较发生时的响应速度，这里保持默认设置即可。OC1PE 是输出比较 1 预装载使能，这和我们在第十六章中定时器控制寄存器 1 的 ARPE 位功能相似，通过设置或者清除该位会直接影响 PWM 占空比改变的时间，即当改变捕获比较寄存器中的值后，是在更新事件发生时将更改生效还是立即生效。OC1M 位域用于设置输出比较的模式，本例程应用的是 PWM 模式。

- 捕获/比较使能寄存器(TIMx_CCER)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		CC4P	CC4E	保留		CC3P	CC3E	保留		CC2P	CC2E	保留		CC1P	CC1E

位1	**CC1P**：输入/捕获1输出极性 (Capture/Compare 1 output polarity) **CC1通道配置为输出：** 0：OC1高电平有效 1：OC1低电平有效 **CC1通道配置为输入：** 该位选择是IC1还是IC1的反相信号作为触发或捕获信号。 0：不反相：捕获发生在IC1的上升沿；当用作外部触发器时，IC1不反相。 1：反相：捕获发生在IC1的下降沿；当用作外部触发器时，IC1反相。
位0	**CC1E**：输入/捕获1输出使能 (Capture/Compare 1 output enable) **CC1通道配置为输出：** 0： 关闭－ OC1禁止输出。 1： 开启－ OC1信号输出到对应的输出引脚。 **CC1通道配置为输入：** 该位决定了计数器的值是否能捕获入TIMx_CCR1寄存器。 0：捕获禁止； 0：捕获使能。

图 20.4 捕获/比较使能寄存器

这里说一下 CC1P 位，它和上面讲解的 PWM1 和 PWM2 模式共同决定了 PWM 的输出极性，即计数值小于或大于比较寄存器中的值时，对应的电平状态。

- 捕获/比较寄存器(TIMx_CCR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CCR1[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位15:0	**CCR1[15:0]**: 捕获/比较1的值 (Capture/Compare 1 value) **若CC1通道配置为输出：** CCR1包含了装入当前捕获/比较1寄存器的值(预装载值)。 如果在TIMx_CCMR1寄存器(OC1PE位)中未选择预装载特性，写入的数值会被立即传输至当前寄存器中。否则只有当更新事件发生时，此预装载值才传输至当前捕获/比较1寄存器中。 当前捕获/比较寄存器参与同计数器TIMx_CNT的比较，并在OC1端口上产生输出信号。 **若CC1通道配置为输入：** CCR1包含了由上一次输入捕获1事件(IC1)传输的计数器值。

图 20.5　捕获/比较寄存器

20.2.3　蜂鸣器介绍

目前大多数开发板都是使用 LED 来验证 PWM，但是使用 LED 只能验证 PWM 的占空比，却不能验证频率的线性变化，因此我们选择使用了无源蜂鸣器，通过蜂鸣器的音调来实验 PWM 的频率，通过蜂鸣器的音量来实验 PWM 的占空比。

这里要顺带讲解一下蜂鸣器的知识。蜂鸣器其实就是塑封的小体积的喇叭，在产品中作为提醒音来用，蜂鸣器分为有源蜂鸣器和无源蜂鸣器，有源蜂鸣器内部含有振荡电路，因此只要给蜂鸣器供电就可以发出声音，而无源蜂鸣器内部不含振荡器，我们需要人为的提供驱动信号给它，使其发出声音，无源蜂鸣器的好处是它可以提供更加灵活的声音定制，能够产生不同音调的声音（取决于驱动信号的频率），但是缺点就是需要人为提供驱动信号，驱动起来相对于有源蜂鸣器更麻烦。下图是开发板上无源蜂鸣器的驱动电路原理图：

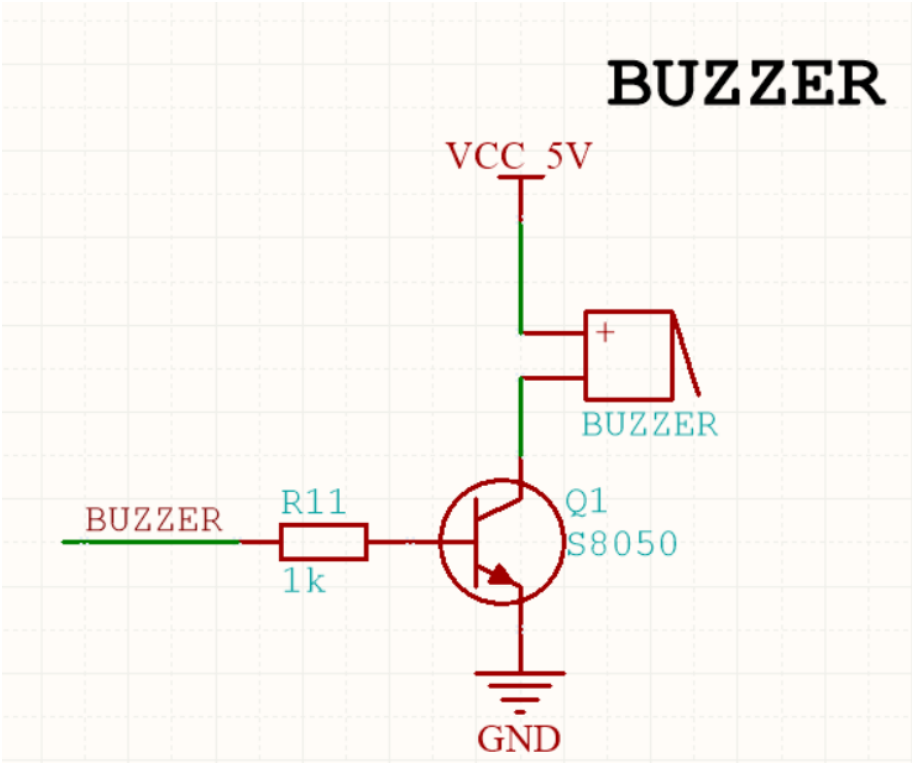


图 20.6　蜂鸣器驱动电路原理图

因为驱动蜂鸣器需要较大电流，所以不适合直接使用 IO 口驱动，因此最常用的驱动方式就是使用三极管对其驱动。如上图。

开发板上的蜂鸣器连接到了 PB4：

A6	A6	B4	56	90	134	PB4	I/O	FT	NJTRST	SPI3_MISO	PB4/TIM3_CH1/ SPI1_MISO

图 20.7　PB4 重映射

从上图可以看到 PB4 默认功能是 JTAG 的复位引脚，因此要将 TIM3_CH1 重映射到该引脚，这一点需要注意，映射方式在第十三章已经讲解过了，所以不做赘述。

20.3 程序讲解

输出比较生成 PWM 的软件配置过程：

1. 设置定时计数器初始化结构体，包括定时器时钟，周期，是否使能各个中断等；
2. 设置输出比较初始化结构体，包括输出比较模式（PWM1 和 PWM2 二选一），输出的极性；
3. 设置 PWM 输出对应的 GPIO 为复用推挽输出；
   最后可以在程序运行时通过更改重装载寄存器 ARR 来改变 PWM 频率，通过更改捕获比较寄存器 CCRx 来改变 PWM 占空比。

- 蜂鸣器初始化函数

```
24  void initBuzzer(u8 vol,u16 tone)
25  {
26      GPIO_InitTypeDef GPIO_InitStructure;                                    //定义GPIO初始化结构体变量
27
28      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO, ENABLE); //使能GPIOB和复用功能时钟
29
30      GPIO_PinRemapConfig(GPIO_PartialRemap_TIM3, ENABLE);                    //将TIM3部分重映射到PB4
31
32      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;                         //设置为复用推挽输出
33      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_4;                               //设置引脚4
34      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;                       //工作速度50MHz
35      GPIO_Init(GPIOB, &GPIO_InitStructure);                                  //初始化设置生效
36
37      setBuzzerVol(vol);                                                      //设置蜂鸣器默认音量
38      setBuzzerTone(tone);                                                    //设置蜂鸣器默认音调
39  }
```

图 20.8　蜂鸣器初始化

初始化函数分为两部分设置，第一部分是重映射 TIM3 到 PB4 引脚，并初始化 PB4 引脚为复用推挽输出，第二部分是设置蜂鸣器的音量和音调。

- 蜂鸣器音调设置函数

```
52  /**
53   * 功能：设置蜂鸣器的音调
54   * 参数：
55   *          tone:音调大小单位Hz，建议100-10000
56   *               另外我们封装了七音阶音调，方便大家使用
57   * 注意：    这里是以定时器时钟频率10us进行计算的
58   * 返回值：None
59   */
60  void setBuzzerTone(u16 tone)
61  {
62      setPeriod(TIM3,100000/tone);
63  }
```

图 20.9　设置蜂鸣器音调

其原理就是设置定时器的重装载寄存器 ARR 中的值，这里说一下频率转换成周期的公式，我们定时器时钟周期是 10us，也就是说记一次数要 10us，那么记 100000 次就是 1s 即 1Hz，因此可知，nHz 对应的周期值就应该是 100000/n。

- 蜂鸣器音量设置函数

```
41  /**
42   * 功能：设置蜂鸣器的音量
43   * 参数：
44   *          vol:音量大小 0-99 0就是静音 99就是最大音量
45   * 返回值：None
46   */
47  void setBuzzerVol(u8 vol)
48  {
49      TIM_SetCompare1(TIM3,getPeriod(TIM3)*vol/100);
50  }
```

图 20.10　设置蜂鸣器音量

通过设置捕获比较寄存器中的值来设置音量，传入的音量值其实就是一个百分比，即 PWM 的占空比等于这个百分比乘上 PWM 的周期。另外 getPeroid()函数是我们封装的一个用于获取重装载寄存器 ARR 中的值函数，因为 ST 标准库并未向开发者提供这个功能的函数，不知道是不是官方疏漏，该函数内部就一条语句:return TIMx->ARR。

- PWM 初始化函数

```
78  /**
79   * 功能：初始化定时器3输出比较通道1来生成PWM
80   * 参数：
81   *      duty：设置PWM输出的占空比
82   * 返回值：None
83   */
84  void initTIM3OC1(u16 duty)
85  {
86      TIM_OCInitTypeDef  TIM_OCInitStructure;                 //定义输出比较初始化结构体
87
88      TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;       //设置PWM1模式
89      TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //输出使能
90      TIM_OCInitStructure.TIM_Pulse = duty;                   //设置捕获比较寄存器的值
91      TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;   //设置有效电平为高电平
92
93      TIM_OC1Init(TIM3, &TIM_OCInitStructure);                //生效初始化设置
94
95      TIM_OC1PreloadConfig(TIM3, TIM_OCPreload_Enable);       //使能输出比较预装载，即更改完输入捕获寄存器的值要等到更新事件发生才生效,该行加不加无所谓
96  }
```

图 20.11　PWM 初始化函数

主要是初始化 PWM 的模式，极性和使能比较输出。在使用 PWM1 模式，有效电平为高电平时，当 CCR1 中的值小于 CNT 中的值时输出高电平，反之输出低电平。最后一句是否使用捕获比较寄存器的预装载功能，这里怎么设置都可以。

- 主函数

```c
int main(void)
{
    u16 tone = 1000;
    u8 vol = 50;

    /*初始化各外设*/
    initSysTick();
    initLED();
    initUART();
    initNVIC(NVIC_PriorityGroup_2);
    /*定时器4时钟周期10us,中断周期200ms,使能更新中断和捕获通道3中断*/
    initTIMx(TIM4,719,20000,TIM_IT_Update|TIM_IT_CC3,ENABLE);
    /*设置下降沿触发捕获*/
    initTIM4IC3(TIM_ICPolarity_Falling);

    /*定时器3时钟周期10us,中断周期1ms,即1KHz，禁止更新中断和捕获通道1中断*/
    initTIMx(TIM3,719,100,TIM_IT_Update|TIM_IT_CC1,DISABLE);
    TIM_ARRPreloadConfig(TIM3,ENABLE);
    initTIM3OC1(90);

    /*蜂鸣器默认最大音量，频率1000Hz*/
    initBuzzer(MAXVOL,1000);
    /*使用ADC初始化函数只是为了将连接PA15的LED灯关掉*/
    initADC();
    initBuzzer(50,1000);
    while (1)
    {
        if(NEC_DataStructure.CmdCode_H == 0x45)        //CH-键按下
        {
            NEC_DataStructure.CmdCode_H = 0;           //清除键值，防止继电器重复动作
            setBuzzerTone(tone);
            if((tone -= 100) <=100)
            {
                tone = 100;
            }

        }else if(NEC_DataStructure.CmdCode_H == 0x47) //CH+键按下
        {
            NEC_DataStructure.CmdCode_H = 0;           //清除键值，防止继电器重复动作
            if((tone += 100)>=10000)
            {
                tone = 1000;
            }


        }else if(NEC_DataStructure.CmdCode_H == 0x07) //-键按下
        {
            NEC_DataStructure.CmdCode_H = 0;           //清除键值，防止继电器重复动作
            if((vol -= 5)<=10)
            {
                vol = 10;
            }

        }else if(NEC_DataStructure.CmdCode_H == 0x15) //+键按下
        {
            NEC_DataStructure.CmdCode_H = 0;           //清除键值，防止继电器重复动作
            if((vol += 5)>=95)
            {
                vol = 95;
            }

        }else
        {

        }

        setBuzzerTone(tone);
        setBuzzerVol(vol);


        printf("tone is :%d\n",tone);
        printf("vol is :%d\n",vol);
        Delay_ms(100);
        toggleLED();

    }
}
```

图 20.12　主函数

执行流程就是先进行外设的初始化，这里先是初始化了 TIM3 的输出比较功能，然后设置了一个初始频率和占空比（随便设置一个即可，因为后面还有蜂鸣器的初始化），接着就是蜂鸣器的初始化，该函数将蜂鸣器音量设置为最大，频率设置为 1000Hz。在用户逻辑循环代码段中判断了对应红外键值的动作。最后下载运行，串口调试助手结果如下：

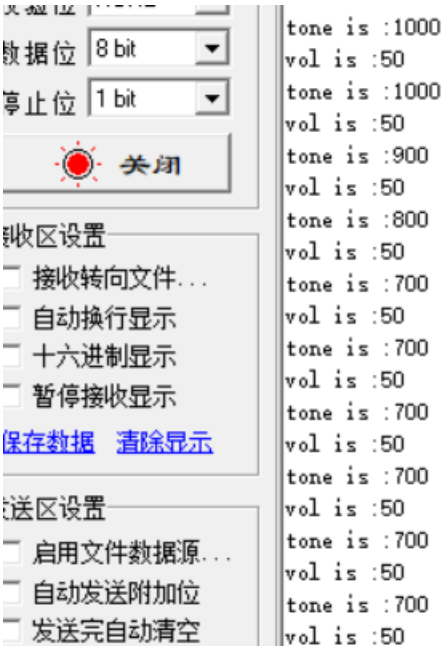


图 20.13　运行结果

在调试时我们发现占空比对蜂鸣器音量的影响其实很小，在音量小于 20 时效果是比较明显的，在其他音量条件下效果几乎没变化。另外在测试的时候也只测试了最大 10KHz 的情况。对于平时用于提醒的话，还是在 500Hz 到 1500Hz 为宜，因为其他频段，尤其是频率上到了 6000Hz 时会有耳鸣感，身体会感到不适。

第二十一章 使用蜂鸣器弹奏两只老虎

21.1 项目要求

通过蜂鸣器来弹奏两只老虎，并每隔 2S 循环播放。

注：本章用到核心板，对应例程 15。

21.2 原理讲解

我们先将七音阶对应的频率找到，然后根据简谱上音阶的标注来制作弹奏文件，其本质就是一个频率的集合，当然有了频率还不能弹奏音乐，还要在每个音阶之间加上一段延时，让其产生节奏，为此，在本例程中封装了这样一个数组：

```
14  /********七音阶对应频率**************/
15  typedef enum
16  {
17      DO = 261,
18      RE = 294,
19      MI = 330,
20      FA = 350,
21      SO = 392,
22      LA = 440,
23      SI = 494
24  }BUZZER_TONE;
```

图 21.1 七音阶频率枚举

```
5   /*两只老虎简谱，数字表示音调持续时长，单位ms*/
6   u16 TwoTiger[] =
7   {
8       DO,400,RE,600,MI,400,DO,600,DO,400,RE,600,MI,400,DO,600,   //两只老虎，两只老虎
9       MI,400,FA,400,SO,800,MI,400,FA,400,SO,800,                //跑得快，跑得快
10      SO,200,LA,200,SO,200,FA,200,MI,600,DO,200,                //一只没有眼睛
11      SO,200,LA,200,SO,200,FA,200,MI,600,DO,200,                //一只没有耳朵
12      RE,400,SO,600,DO,800,RE,400,SO,600,DO,800,               //真奇怪，真奇怪
13      0,0                                                       //用于结尾标识
14  };
15
16  #endif
```

图 21.2 乐谱数组

第一个枚举用于定义各个音阶对应的频率值，这样封装的目的是可以让简谱数组更易维护，接下来就是定义的两只老虎的简谱数组，只要大家按照该简谱结构编写数组，就可以让单片机弹奏你的音乐。该数组有三部分组成，第一部分就是音调，第二部分就是数字表示的

延时节奏，最后是两个 0，用于告诉单片机，此时演奏完毕，退出演奏循环。用两个 0 的目的是，如果简谱是一个节奏对应一个延时的话，那么它是偶数个演奏单元，所以会用到最后一个零，如果是奇数个演奏单元，就用到第一个 0 来标识结束。

21.3 程序讲解

我们为蜂鸣器封装了一个播放函数，如下：

```
63  /**
64   * 功能：根据乐谱弹奏音乐
65   * 参数：
66   *          vol:音量大小 0-99 0就是静音 99就是最大音量
67   *          musicscore:乐谱指针
68   * 注意：    这里是以定时器时钟频率10us进行计算的
69   * 返回值：None
70   */
71  void playMusic(u8 vol,u16* musicscore)
72  {
73      u16 i = 0;
74
75      setBuzzerVol(vol);
76      while(*(musicscore+i))       //判断乐谱结束标识
77      {
78          setBuzzerTone(*(musicscore+i));
79          setBuzzerVol(vol);
80          Delay_ms(*(musicscore+i+1));
81
82          i += 2;
83      }
84      setBuzzerVol(MUTE);
85  }
```

图 21.3　乐谱播放函数

该函数流程是播放音乐开始时设置音量，然后播放音乐，直到遇到播放结束标志，然后静音，等待下一下次被调用。**需要注意的是，在每次更改完频率之后，必须同时更新音量，因为不同频率下相同音量所对应的捕获比较寄存器中的值是不同的。**

下面是主函数：

```
46      while (1)
47      {
48          playMusic(50,TwoTiger);
49          toggleLED();
50          Delay_ms(2000);
51      }
52  }
```

图 21.4　业务逻辑循环

音量设置为 50，2S 循环播放。

因为作者不懂音乐，所以如果音调不对请见谅，我这里只是想抛砖引玉，如果有懂音乐的朋友，欢迎大家一起完善乐谱。

另外开发板配套的红外遥控器就是用于 MP3 控制的，因此大家可以自行扩展音乐的静音，播放，上/下一曲（切换乐谱数组），快进/快退（乐谱索引加偏置）。

第二十二章 使用 PWM 模拟 NEC

22.1 项目要求

此项目需要用到两块青柚 ZERO 的开发板，A 开发板通过 UP 键和 DOWN 键来控制产生并发送 NEC 遥控器的 1 键键值和 2 键键值对应的红外编码，B 开发板运行的是第十九章的红外控制继电器程序。该项目实现的效果是 A 开发板可以控制 B 开发板的继电器状态。

注：本章用到核心板+扩展板，对应例程 16。

22.2 原理讲解

在第十八章和第十九章分别讲解了 NEC 红外编码协议和继电器控制，本章就不做过多讲解。我们开发板板载红外发射管的原因是物联网在智能家居上的应用场合很多都是需要红外进行控制的，比如远程控制空调温度、风量、模式等，这就存在一个问题：每个厂商的电器设备所用到的红外编码协议和用户码、命令码都不同，因此想要做到完全兼容，第一个想到的办法就是获取到市面上绝大多数的设备的红外码库，但是这个码库信息量很大，并且需要花钱购买，同时又不能保证数据是最新的（因为会不断有新的电器推出），因此我们可以另辟蹊径，把红外编码的采集工作交给用户，比如用户想要控制家中空调的开/关和风量，那就需要用户事先将遥控器对准家中的智能硬件上的红外接收头，让硬件先采集这些功能对应的红外编码并记录存储，然后用户就可以远程控制智能硬件了，让其通过红外发射管将之前采集到的红外编码时序发出即可。要实现这种红外控制功能，就必须保证红外发射足够灵活，因此最适合的方式就是使用 PWM 来根据采集到的时序来模拟出红外控制信号。

本章依旧是使用 NEC 红外编码，我们要做的工作先是调制出 38KHz 的 1/3 占空比的红外载波，然后根据 NEC 协议将传入的待发送数据编码发送出去即可。

22.3 程序讲解

我们完善了 NEC 的初始化函数，在 NEC 初始化函数中添加了红外发射管 IO 引脚的初始化，如下图：

- NEC 初始化函数

```c
/**
 * 功能：初始化NEC
 * 参数：  None
 * 返回值：None
 */
void initNEC(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /*************************红外接收IO初始化*************************/
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);           //开启GPIOB时钟

    GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_8;                      //配置PB8
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;           //配置为浮空输入
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    /*************************红外发射IO初始化*************************/
    GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_7;                      //配置PB7
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;                 //配置为复用输出
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    NO_CARRIER();
}
```

图 22.1　NEC 初始化函数

因为该引脚用于 PWM 输出，所以要使用复用推挽模式。

- PWM 初始化函数

```c
/**
 * 功能：初始化定时器4输出比较通道2来生成PWM
 * 参数：
 *      duty：设置PWM输出的占空比
 * 返回值：None
 */
void initTIM4OC2(u16 duty)
{
    TIM_OCInitTypeDef  TIM_OCInitStructure;                  //定义输出比较初始化结构体

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;        //设置PWM1模式
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //输出使能
    TIM_OCInitStructure.TIM_Pulse = duty;                    //设置捕获比较寄存器的值
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;  //设置有效电平为高电平

    TIM_OC2Init(TIM4, &TIM_OCInitStructure);                 //生效初始化设置

    TIM_OC2PreloadConfig(TIM4, TIM_OCPreload_Enable);        //使能输出比较预装载，即更改完输入捕获寄存器的值要等到更新事件发生才生效,该行加不加无所谓
```

图 22.2　PWM 初始化函数

该函数和第二十章讲解的 PWM 初始化过程相同，这里不做赘述。

- NEC 编码函数

```c
/**
 * 功能：发送NEC起始码
 * 参数：  None
 * 返回值：None
 */
static void codeStart(void)
{
    /*起始码*/
    HAVE_CARRIER();
    Delay_us(9000);
    NO_CARRIER();
    Delay_us(4500);
}
```

```
29
30  /**
31   * 功能：发送一个字节
32   * 参数：
33   *         data：待发送数据
34   * 返回值：None
35   */
36  static void codeByte(u8 data)
37  {
38      u8 i;
39
40      /*发送数据*/
41      for(i=0;i<8;++i)
42      {
43          if((data>>i)&0x01)      //先发送低位
44          {
45              HAVE_CARRIER();
46              Delay_us(560);
47              NO_CARRIER();
48              Delay_us(1690);
49          }else
50          {
51              HAVE_CARRIER();
52              Delay_us(560);
53              NO_CARRIER();
54              Delay_us(560);
55          }
56      }
57  }

59  /**
60   * 功能：发送一个字节
61   * 参数：
62   *         repeatcnt：待发送重复码个数
63   * 返回值：None
64   */
65  static void codeRepeat(u8 repeatcnt)
66  {
67      u8 i;
68
69      if(repeatcnt==0)          //如果没有重复码就直接设置无载波，发射管进行空闲状态
70      {
71          HAVE_CARRIER();
72          Delay_us(560);
73          NO_CARRIER();
74      }else
75      {
76          for(i=0;i<repeatcnt;++i)
77          {
78              HAVE_CARRIER();
79              Delay_us(560);
80              NO_CARRIER();
81              Delay_ms(98);
82              HAVE_CARRIER();
83              Delay_us(9000);
84              NO_CARRIER();
85              Delay_us(2250);
86          }
87
88          HAVE_CARRIER();
89          Delay_us(560);
90          NO_CARRIER();
91      }
92  }
```

图 22.3　NEC 编码函数

上面这三个函数是封装的 NEC 编码的基础函数，供 NEC 编码函数调用。图中的 HAVE_CARRIER 和 NO_CARRIER 是在 NEC.h 中定义的宏，如下图：

```
15  /**********************是否产生载波宏定义**********************/
16  #define HAVE_CARRIER()  TIM_SetCompare2(TIM4,9)
17  #define NO_CARRIER()    TIM_SetCompare2(TIM4,0)
```

图 22.4　红外载波控制宏

即设置是否产生 1/3 占空比载波，当占空比为 0 则无载波，为 9 则为 1/3 载波。另外大家还需要注意的就是 NEC 发送的红外数据是从一个字节的低位开始发送的。

```
158 /**
159  * 功能：NEC编码函数
160  * 参数：
161  *       addr:NEC地址码
162  *       cmd:NEC命令码
163  *       repeatcnt:重复码个数
164  * 返回值：None
165  */
166 void codeNEC(u8 addr,u8 cmd,u8 repeatcnt)
167 {
168     codeStart();                //发送起始码
169 
170     codeByte(addr);             //发送地址码
171     codeByte(~addr);            //发送地址反码
172     codeByte(cmd);              //发送命令码
173     codeByte(~cmd);             //发送命令反码
174 
175     codeRepeat(repeatcnt);      //发送重复码
176 }
```

图 22.5　NEC 编码函数

该函数是 NEC 编码函数，从结构上能够一目了然的看到整个发码的过程。另外一般情况下是不发送重复码的，因为重复码是用在持续累加的场合，而大多数应用都是直接控制状态的，而且发射重复码会升高 NEC 发码时间，对于实时性要求高的场合要慎用。

- 部分主函数

```
    initNEC();
    /*定时器4时钟周期1us,周期值26,对应频率为38.4KHz,失能更新中断和捕获通道3中断*/
    initTIMx(TIM4,71,26,TIM_IT_Update|TIM_IT_CC3,DISABLE);
    /*设置占空比为定时器周期的1/3*/
    initTIM4OC2(0);
    /*使能预分频*/
    TIM_ARRPreloadConfig(TIM4,ENABLE);

    /*使用ADC初始化函数只是为了将连接PA15的LED灯关掉*/
    initADC();
    while (1)
    {
        key_value = getKeyValue(KEY_RELEASE);   //获取键值，松开生效

        if(key_value==KEY_UP)
        {
            codeNEC(0x00,0x0C,0);               //发送1键键值
        }else if(key_value==KEY_DOWN)
        {
            codeNEC(0x00,0x18,0);               //发送2键键值
        }else
        {

        }

        toggleLED();
        Delay_ms(100);
    }
}
```

图 22.5　主函数

在函数初始化时，设置了定时器 4 的时钟周期是 1us，则 38KHz 的对应时钟数是：1000 000/38 000 ≈ 26，占空比开机后应该设置为 0，即不发码。我们在主函数循环中不断的查询按键键值，然后根据键值做出相应发射动作。

最后将程序烧录到板子当中，实验的效果是：当按下 A 开发板的 UP 键，B 开发板的继电器 1 吸合，DOWN 键功能同理。

由这个项目可以作进一步的扩展，就是在本章原理讲解中提到的，我们可以通过按键来切换红外接收或发送模式，在红外接收模式下，设备捕获遥控器发射过来的红外信号时长并保存，在发射模式下，再将保存的后外时长还原出来并控制设备，此时需要更改的就是输入捕获的中断服务函数，因为我们不能保证收到的红外信号是 NEC 协议，因此直接做采集即可，更加的简单，另外还需要在 NEC.h/.c 中添加还原编码函数。

大家在做这个实验时要注意，现在有些家用电器的遥控器都是 2.4G 遥控的了，但是他们都留有红外窗（可以和红外产品共用模具），所以在做实验时要先确定它是哪种遥控器。

第二十三章 PWM 实现调光

23.1 项目要求

通过 PWM 驱动扩展板上的冷光 LED 和暖光 LED，并通过红外遥控器调节占空比来实现调节色温和调节亮度的功能。具体要求如下：

1. 使用 CH+/-键设置灯光的色温
2. 使用+/-键设置灯光亮度
3. 使用 EQ 键控制灯光开启或者关闭
4. 通过串口打印此时的亮度和色温百分比

注：本章用到核心板+扩展板，对应例程 17。

23.2 原理讲解

学习物联网最经典的案例就是实现一款智能灯，用户可以对其进行色温、亮度、颜色（下一章讲解）的控制，色温就是指灯光的颜色是偏白（冷光源）还是偏黄（暖光源），单位是 K（开尔文），色温值越高光源颜色就越白反之色温越低，光源颜色就越黄。通常使用冷光灯和暖光灯配合的方式，通过调节各自的亮度来实现色温的任意调节。

调节灯光亮度的原理就是通过 PWM 的占空比来设置灯光亮暗，调节灯光色温则是在当前亮度值（占空比）基础上再次进行比例的分配，如下图：

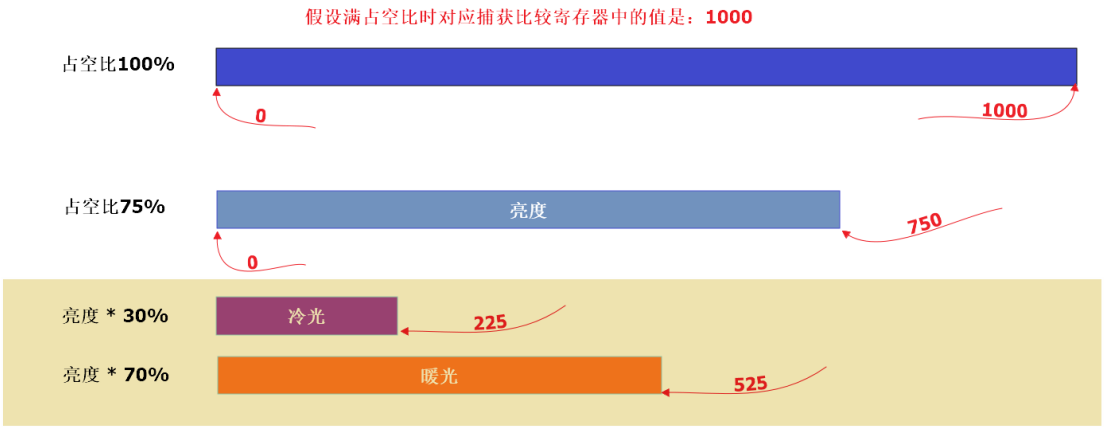


图 23.1 调光原理

开发板上使用的冷光灯色温为 6000K，暖光灯为 3000K，上图这种情况下对应的色温值为 3900K，另外为了保证色温恒定在固定值，通常亮度不能设置为 0，而是有一个保底的最小值，比如 10%。还需要注意的是，为了大家方便理解今后不使用 K 作为色温单位，而是像亮度那样使用百分比值，范围 0-100，值越大，灯光越白。

开发板上的两颗高亮 LED 分别是暖光 LED 和冷光 LED，他们的原理图如下：

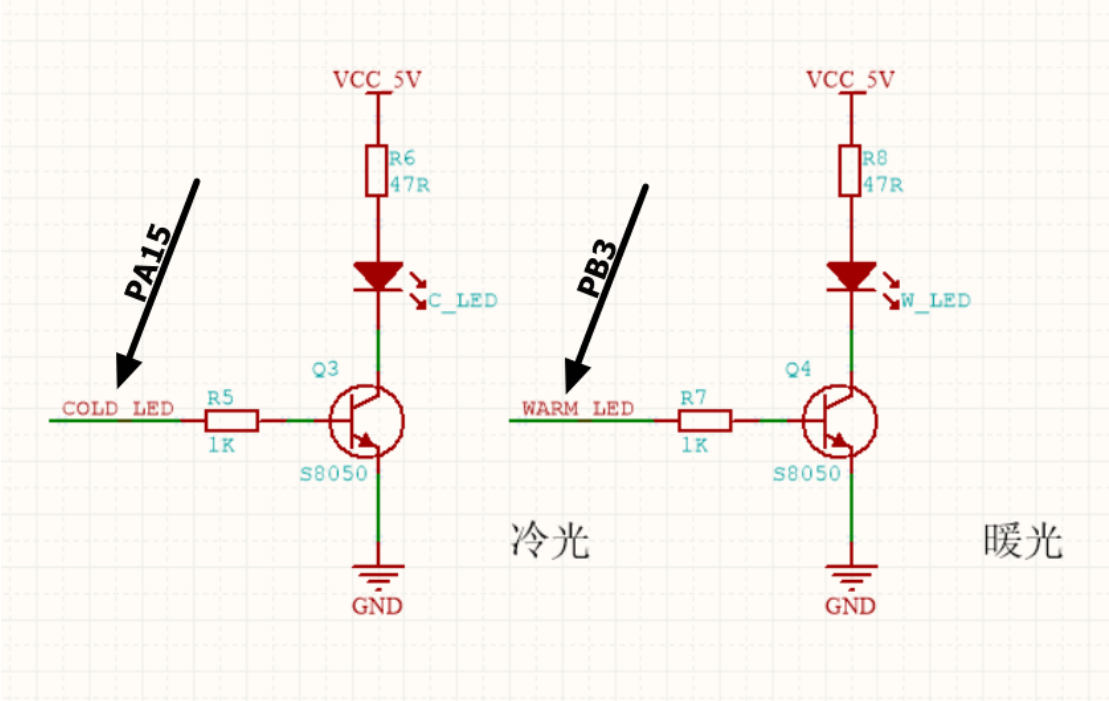


图 23.2　光源驱动电路原理图

PA15 在 ADC 的章节说过，上电后，该引脚默认是 JTAG 的 JTDI，因此需要将其引脚重映射为 TIM2 的通道 1，PB3 上电默认功能是 JTAG 的 JTDO，所以也需要 PB3 重映射为 TIM2 的通道 1，这两个引脚的功能映射图如下：

A7	A7	A4	55	89	133	PB3	I/O	FT	JTDO	SPI3_SCK / I2S3_CK	PB3/TRACESWO TIM2_CH2/ SPI1_SCK
A10	A8	C3	50	77	110	PA15	I/O	FT	JTDI	SPI3_NSS/I2S3_WS	TIM2_CH1_ETR PA15/SPI1_NSS

图 23.3　光源驱动引脚映射情况

配置定时器 2 的通道 1/2 作为 PWM 输出的步骤如下：

1. 重映射 TIM2 的 1/2 通道到 PA15 和 PB3 引脚，并初始化为复用输出
2. 初始化定时器，主要是设置定时器周期和重装载寄存器
3. 分别初始化定时器 2 的输出比较通道 1/2

本章难点就在于将亮度和色温值最终转换为占空比的计算公式上。我们结合程序来学习这两个公式。

23.3 程序讲解

- 灯光初始化函数

```
/**
 * 功能：初始灯光
 * 参数：
 *          brightness:初始化亮度 10-100
 *          colortemp:初始化色温 0-100
 * 返回值：None
 */
void initLight(u8 brightness,u8 colortemp)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO,ENABLE);

    GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);     //禁止JTAG保留SWD
    GPIO_PinRemapConfig(GPIO_PartialRemap1_TIM2, ENABLE);        //设置JTAG为定时器2部分映射,只使用SWD模式

    /*设置冷光灯*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOA, &GPIO_InitStructure);

    /*设置暖光灯*/
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    setLight(brightness,colortemp);
}
```

图 23.4　灯光初始化

灯光初始化函数实现了复用功能重映射，并初始化冷光灯和暖光灯的控制引脚为复用输出。

- 设置灯光函数

```
/**
 * 功能：设置灯光亮度和色温
 * 参数：
 *          brightness:亮度 10-100
 *          colortemp:色温 0-100
 * 返回值：None
 */
void setLight(u8 brightness,u8 colortemp)
{
    /**
     * 无论是亮度还是色温，最终都体现在LED的亮度上
     * LED最终的亮度计算公式为：满占空比(重装载值) * 亮度百分比 * 色温百分比
     * 并且要保证冷光和暖光的色温比值之和为100%
     * */
    TIM_SetCompare1(TIM2,getPeriod(TIM2)*brightness/100*colortemp/100);         //设置冷光 对应PA15 TIM2_CH1
    TIM_SetCompare2(TIM2,getPeriod(TIM2)*brightness/100*(100-colortemp)/100);   //设置暖光 对应PB3  TIM2_CH2
}
```

图 23.5　调光函数

为了方便讲解，作者并没有把占空比计算公式进行合并。我们通过 TIM_setCompare1()和 TIM_setCompare2()函数对定时器 2 的捕获/比较寄存器 1/2 赋值来改变 PWM 占空比，传入函数的第二个参数就是亮度-色温的计算公式，在公式中先获取了重装载寄存器中的值（满占空比对应的计数值），然后乘上亮度的百分比得到我们想要的亮度，最后在此基础上又将亮度值乘上色温百分比，这里要注意，冷/暖光的色温比值之和是 100%，这就保证了，无论亮度值是多少，最终呈现出来的色温是不受影响的。

- PWM 初始化函数

```c
/**
 * 功能：初始化定时器2输出比较通道1来生成PWM 对应冷光
 * 参数：
 *      duty：设置PWM输出的占空比
 * 返回值：None
 */
void initTIM2OC1(u16 duty)
{
    TIM_OCInitTypeDef  TIM_OCInitStructure;                  //定义输出比较初始化结构体

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;        //设置PWM1模式
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //输出使能
    TIM_OCInitStructure.TIM_Pulse = duty;                    //设置捕获比较寄存器的值
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;//设置有效电平为高电平

    TIM_OC1Init(TIM2, &TIM_OCInitStructure);                 //生效初始化设置

    TIM_OC1PreloadConfig(TIM2, TIM_OCPreload_Enable);        //使能输出比较预装载，即更改完输入捕获寄存器的值要等到更新事件发生才生效,该行加不加无所谓
}

/**
 * 功能：初始化定时器2输出比较通道2来生成PWM 对应冷光
 * 参数：
 *      duty：设置PWM输出的占空比
 * 返回值：None
 */
void initTIM2OC2(u16 duty)
{
    TIM_OCInitTypeDef  TIM_OCInitStructure;                  //定义输出比较初始化结构体

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;        //设置PWM1模式
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //输出使能
    TIM_OCInitStructure.TIM_Pulse = duty;                    //设置捕获比较寄存器的值
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;//设置有效电平为高电平

    TIM_OC2Init(TIM2, &TIM_OCInitStructure);                 //生效初始化设置

    TIM_OC2PreloadConfig(TIM2, TIM_OCPreload_Enable);        //使能输出比较预装载，即更改完输入捕获寄存器的值要等到更新事件发生才生效,该行加不加无所谓
}
```

图 23.6 PWM 初始化函数

这一部分初始化和前面讲到的相同，这里不做赘述。

- 主函数

```c
int main(void)
{
    u8 light_power = 1;     //默认开机
    s8 brightness = 100;    //亮度
    s8 colortemp = 50;      //色温

    /*初始化各外设*/
    initSysTick();
    initLED();
    initUART();
    initNVIC(NVIC_PriorityGroup_2);

    /*初始化NEC*/
    initNEC();
    /*定时器4时钟周期10us,中断周期200ms,使能更新中断和捕获通道3中断*/
    initTIMx(TIM4,719,20000,TIM_IT_Update|TIM_IT_CC3,ENABLE);
    /*设置下降沿触发捕获*/
    initTIM4IC3(TIM_ICPolarity_Falling);

    /*设置定时器2时钟为10us,1KHz*/
    initTIMx(TIM2,719,100,TIM_IT_Update,DISABLE);
    initTIM2OC1(50);
    initTIM2OC2(50);
    /*亮度10%,色温50%*/
    initLight(100,50);
```

```
    while(1)
    {
        if(NEC_DataStructure.CmdCode_H == 0x45)       //CH-键按下
        {
            NEC_DataStructure.CmdCode_H = 0;          //清除键值，防止继电器重复动作
            colortemp -= 10;
            if(colortemp<0)
            {
                colortemp = 0;
            }
            setLight(brightness,colortemp);
        }else if(NEC_DataStructure.CmdCode_H == 0x47) //CH+键按下
        {
            NEC_DataStructure.CmdCode_H = 0;          //清除键值，防止继电器重复动作
            colortemp += 10;
            if(colortemp>100)
            {
                colortemp = 100;
            }
            setLight(brightness,colortemp);
        }else if(NEC_DataStructure.CmdCode_H == 0x07) //-键按下
        {
            NEC_DataStructure.CmdCode_H = 0;          //清除键值，防止继电器重复动作
            brightness -= 10;
            if(brightness<10)
            {
                brightness = 10;                       //保底值
            }
            setLight(brightness,colortemp);
        }else if(NEC_DataStructure.CmdCode_H == 0x15) //+键按下
        {
            NEC_DataStructure.CmdCode_H = 0;          //清除键值，防止继电器重复动作
            brightness += 10;
            if(brightness>100)
            {
                brightness = 100;
            }
            setLight(brightness,colortemp);
        }else if(NEC_DataStructure.CmdCode_H == 0x09) //EQ键按下
        {
            NEC_DataStructure.CmdCode_H = 0;          //清除键值，防止继电器重复动作
            light_power = !light_power;               //开关机状态取反
            if(light_power == 0)
            {
                setLight(0,0);                        //关灯
            }else
            {
                setLight(brightness,colortemp);       //开灯
            }
        }else
        {

        }

        printf("brightness is :%d%%\n",brightness);
        printf("color temperature is :%d%%\n",colortemp);
        toggleLED();
        Delay_ms(100);
    }
}
```

图 23.7　主函数

定时器 2 设置的时钟周期为 10us，则 1KHz（1ms）需要计数 100 个。然后初始化 PWM 并设置灯光开机默认最大亮度，色温 50%。

在用户逻辑循环中不断的判断此时红外键值，并根据键值做出相应的灯光控制动作。编译烧录到板子，观察效果如下：

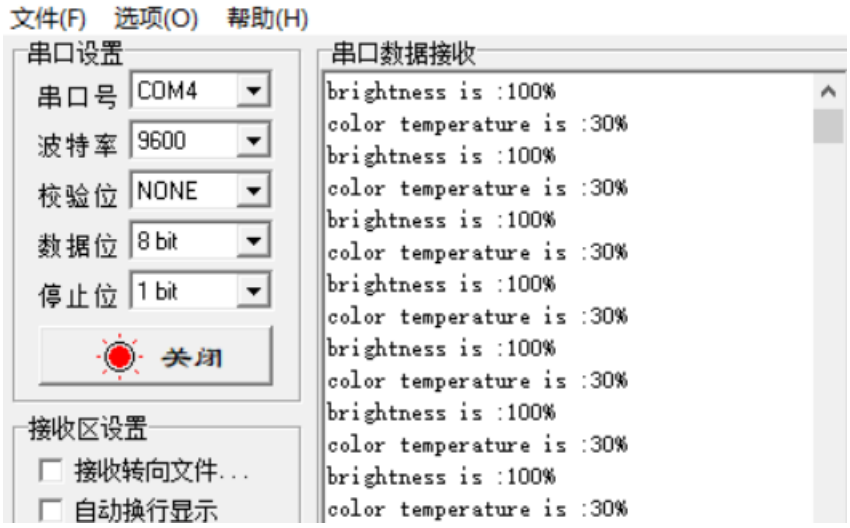


图 23.8　运行效果

提醒大家：不要长时间注视强光，调试的时候可以盖上一张白纸或者对着白色墙壁。

第二十四章 PWM+DMA 驱动 WS2812

24.1 项目要求

使用 PWM+DMA 的方式驱动 WS2812B，并每隔 800ms 随机显示不同颜色。
另外，我们也将在本章探索一下 STM32 的极限性能，内容包括：

1. 使用库函数翻转电平和使用寄存器翻转电平的代码执行效率的差距
2. while(1)和 for(;;)的执行效率的对比

注：本章用到核心板+扩展板，对应例程 18。

24.2 原理讲解

24.2.1 帮 CPU 跑腿的 DMA

在程序中，如果要实现将一个数据块复制到另外一个数据块时，常规做法就是将源数据放在 for 循环当中，然后一个一个的复制到目标位置。这里 CPU 就起到了一个“搬运工”的作用，但是，单纯的搬运工作将会阻塞 CPU，使得 CPU 重复执行耗时的复制任务，降低了整个系统的实时性，因此 DMA（Direct Memory Access，直接内存存取）诞生了。DMA 就是 CPU 的贴身小秘书，CPU 只需要告诉 DMA，你要去哪取货（源地址），要送到哪（目标地址），要求的每次运货的货物重量（数据长度，8/16/32 位），要运几次等等。CPU 给 DMA 分配完任务之后，CPU 就去执行它自己的业务逻辑去了，而 DMA 也开始进行数据的搬运，这样一来就大大降低了 CPU 的工作负担。

STM32F103 上面最多包含两个 DMA 外设（DMA1 和 DMA2）和 12 个 DMA 通道（DMA1 包含 7 个通道，DMA2 包含 5 个通道），其中 DMA2 仅存在于大容量单片机中。这里的每个通道就代表着一个数据搬运工，并且为了防止多个搬运工同时进行数据搬运时产生冲突（多通道共用一个数据总线），就又规定了 4 个传输优先级，分别是很高，高，中等和低，当优先级相同时通道号小的传输优先级比通道号大的传输优先级要高，另外，STM32 的 DMA 除了可以在内存中来回搬运数据外，还可以在外设和内存以及内存和外设之间进行数据的传输，比如我们学习 ADC 章节的时候是通过 CPU 进行数据采集的，如果使用 DMA 的话，则由 DMA 负责采集 ADC 的数据寄存器并将其值存放到我们指定的内存变量当中。DMA1/2 各个通道支持的外设如下：

外设	通道1	通道2	通道3	通道4	通道5	通道6	通道7
ADC1	ADC1						
SPI/I²S		SPI1_RX	SPI1_TX	SPI/I2S2_RX	SPI/I2S2_TX		
USART		USART3_TX	USART3_RX	USART1_TX	USART1_RX	USART2_RX	USART2_TX
I²C				I2C2_TX	I2C2_RX	I2C1_TX	I2C1_RX
TIM1		TIM1_CH1	TIM1_CH2	TIM1_TX4 TIM1_TRIG TIM1_COM	TIM1_UP	TIM1_CH3	
TIM2	TIM2_CH3	TIM2_UP			TIM2_CH1		TIM2_CH2 TIM2_CH4
TIM3	PB6	TIM3_CH3	TIM3_CH4 TIM3_UP			TIM3_CH1 TIM3_TRIG	
TIM4	TIM4_CH1			TIM4_CH2	TIM4_CH3		TIM4_UP

外设	通道1	通道2	通道3	通道4	通道5
ADC3(1)					ADC3
SPI/I2S3	SPI/I2S3_RX	SPI/I2S3_TX			
UART4			UART4_RX		UART4_TX
SDIO(1)				SDIO	
TIM5	TIM5_CH4 TIM5_TRIG	TIM5_CH3 TIM5_UP		TIM5_CH2	TIM5_CH1
TIM6/ DAC通道1			TIM6_UP/ DAC通道1		
TIM7/ DAC通道2				TIM7_UP/ DAC通道2	
TIM8(1)	TIM8_CH3 TIM8_UP	TIM8_CH4 TIM8_TRIG TIM8_COM	TIM8_CH1		TIM8_CH2

1. ADC3、SDIO和TIM8的DMA请求只在大容量的产品中存在。

图 24.1　DMA 通道外设对照表

本章使用的是定时器 4 的通道 1（PB6 引脚），所以要使用 DMA1 的通道 1。

接下来看一下和 DMA 相关的寄存器：

- DMA 中断状态寄存器（DMA_ISR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				TEIF7	HTIF7	TCIF7	GIF7	TEIF6	HTIF6	TCIF6	GIF6	TEIF5	HTIF5	TCIF5	GIF5
				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
TEIF4	HTIF4	TCIF4	GIF4	TEIF3	HTIF3	TCIF3	GIF3	TEIF2	HTIF2	TCIF2	GIF2	TEIF1	HTIF1	TCIF1	GIF1
r	r	r	r	r	r	r	r	r	r	r	r	r	r	r	r

位31:28	保留，始终读为0。
位27，23，19，15，11，7，3	**TEIFx**：通道x的传输错误标志(x = 1 … 7) (Channel x transfer error flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有传输错误(TE)； 1：在通道x发生了传输错误(TE)。
位26，22，18，14，10，6，2	**HTIFx**：通道x的半传输标志(x = 1 … 7) (Channel x half transfer flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有半传输事件(HT)； 1：在通道x产生了半传输事件(HT)。
位25，21，17，13，9，5，1	**TCIFx**：通道x的传输完成标志(x = 1 … 7) (Channel x transfer complete flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有传输完成事件(TC)； 1：在通道x产生了传输完成事件(TC)。

位24，20，16，12，8，4，0	**GIFx**：通道x的全局中断标志(x = 1 … 7) (Channel x global interrupt flag) 硬件设置这些位。在DMA_IFCR寄存器的相应位写入'1'可以清除这里对应的标志位。 0：在通道x没有TE、HT或TC事件； 1：在通道x产生了TE、HT或TC事件。

图 24.2　DMA 中断状态寄存器

这些标志位用于判断此时 DMA 的运行状态，比如传输到一半或者完全传输标志位，其中用到最多的就是传输完成标志，即 TCIFx。

- DMA 中断标志清除寄存器（DMA_IFCR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留				CTEIF7	CHTIF7	CTCIF7	CGIF7	CTEIF6	CHTIF6	CTCIF6	CGIF6	CTEIF5	CHTIF5	CTCIF5	CGIF5
				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
CTEIF4	CHTIF4	CTCIF4	CGIF4	CTEIF3	CHTIF3	CTCIF3	CGIF3	CTEIF2	CHTIF2	CTCIF2	CGIF2	CTEIF1	CHTIF1	CTCIF1	CGIF1
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:28	保留，始终读为0。
位27，23，19，15，11，7，3	**CTEIFx**：清除通道x的传输错误标志(x = 1 … 7) (Channel x transfer error clear) 这些位由软件设置和清除。 0：不起作用 1：清除DMA_ISR寄存器中的对应TEIF标志。
位26，22，18，14，10，6，2	**CHTIFx**：清除通道x的半传输标志(x = 1 … 7) (Channel x half transfer clear) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应HTIF标志。
位25，21，17，13，9，5，1	**CTCIFx**：清除通道x的传输完成标志(x = 1 … 7) (Channel x transfer complete clear) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应TCIF标志。
位24，20，16，12，8，4，0	**CGIFx**：清除通道x的全局中断标志(x = 1 … 7) (Channel x global interrupt clear) 这些位由软件设置和清除。 0：不起作用 0：清除DMA_ISR寄存器中的对应的GIF、TEIF、HTIF和TCIF标志。

图 24.3　DMA 中断清除寄存器

和其他外设清除标志位的方式不同，DMA 要使用单独的标志清除寄存器来清除掉 DMA_ISR 寄存器中的对应标志位。

- DMA 通道 x 配置寄存器(DMA_CCRn) n = 1-7

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留	MEM2MEM	PL[1:0]		MSIZE[1:0]		PSIZE[1:0]		MINC	PINC	CIRC	DIR	TEIE	HTIE	TCIE	EN
	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位13:12	**PL[1:0]**：通道优先级 (Channel priority level) 这些位由软件设置和清除。 00：低 01：中 10：高 11：最高
位11:10	**MSIZE[1:0]**：存储器数据宽度 (Memory size) 这些位由软件设置和清除。 00：8位 01：16位 10：32位 11：保留
位9:8	**PSIZE[1:0]**：外设数据宽度 (Peripheral size) 这些位由软件设置和清除。 00：8位 01：16位 10：32位 11：保留
位7	**MINC**：存储器地址增量模式 (Memory increment mode) 该位由软件设置和清除。 0：不执行存储器地址增量操作 1：执行存储器地址增量操作
位6	**PINC**：外设地址增量模式 (Peripheral increment mode) 该位由软件设置和清除。 0：不执行外设地址增量操作 1：执行外设地址增量操作
位5	**CIRC**：循环模式 (Circular mode) 该位由软件设置和清除。 0：不执行循环操作 1：执行循环操作
位4	**DIR**：数据传输方向 (Data transfer direction) 该位由软件设置和清除。 0：从外设读 1：从存储器读
位3	**TEIE**：允许传输错误中断 (Transfer error interrupt enable) 该位由软件设置和清除。 0：禁止TE中断 0：允许TE中断
位2	**HTIE**：允许半传输中断 (Half transfer interrupt enable) 该位由软件设置和清除。 0：禁止HT中断 0：允许HT中断
位1	**TCIE**：允许传输完成中断 (Transfer complete interrupt enable) 该位由软件设置和清除。 0：禁止TC中断 0：允许TC中断

图 24.4　DMA 通道配置寄存器

位域[13:12]用于设置各个通道的传输优先级；

位域[11:8]用于设置源/目标的数据宽度，即每次传输数据的最小单位，通常源/目标数据宽度要设置为一样；

位域[6:7]用于设置传输完本次数据后是否对其数据指针加 1，比如通过 DMA 将内存中的一个数组传输给 UART 的数据寄存器，要就需要开启存储器地址增量，关闭外设地址增量。

位 5 用于设置 DMA 的传输模式，不使用循环则在传输完本次数据之后就停止传输，想要进行下一次传输，就必须重新设置传输数量，然后再开启 DMA，如果使用了循环传输则在本次数据传输完成之后，传输数量寄存器会自动重装载设置的值，并重复上次的传输过程，并以此不断循环下去。

位 4 用来设置源/目标的传输方向，如果使用 DMA 对 ADC 采集则是从外设读，如果是将内存当中的数据传输到 UART 数据寄存器则是从存储器读。

其余的位用于设置是否开启相关 DMA 中断，如半传输中断，传输完成中断等，本章中并没有使用 DMA 中断，因此全部禁止即可。

- DMA 通道 x 传输数量寄存器(DMA_CNDTRx) n = 1-7

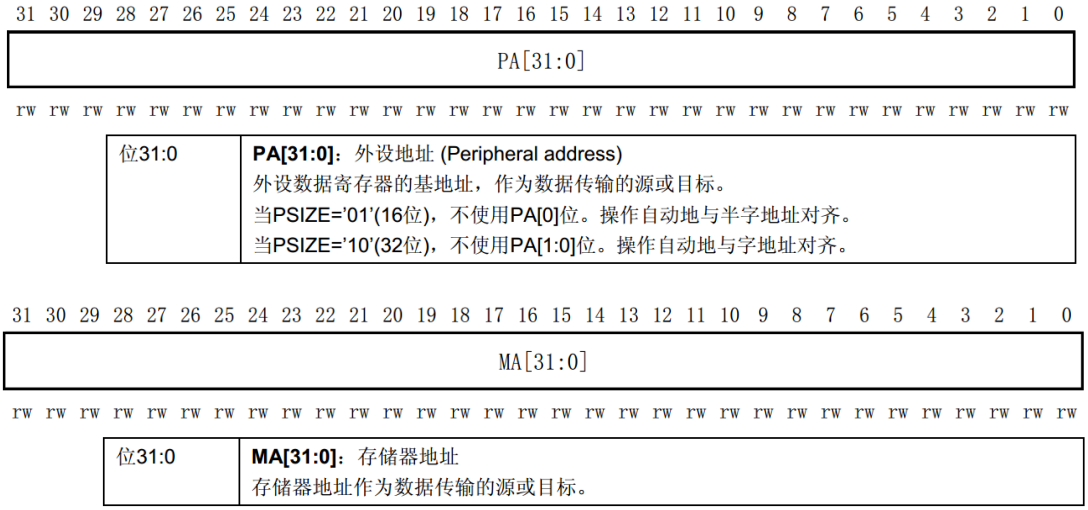
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
NDT[15:0]															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:16	保留，始终读为0。
位15:0	**NDT[15:0]**：数据传输数量 (Number of data to transfer) 数据传输数量为0至65535。这个寄存器只能在通道不工作(DMA_CCRx的EN=0)时写入。通道开启后该寄存器变为只读，指示剩余的待传输字节数目。寄存器内容在每次DMA传输后递减。 数据传输结束后，寄存器的内容或者变为0；或者当该通道配置为自动重加载模式时，寄存器的内容将被自动重新加载为之前配置时的数值。 当寄存器的内容为0时，无论通道是否开启，都不会发生任何数据传输。

图 24.5　DMA 通道传输数量寄存器

该寄存器用于配置传输的数据数量，比如此时源/目标数据宽度都是 16 位（半字），我们希望将内存当中的 100 个 16 位无符号整形的数组(u16 buffer[100])传输到外设，则该寄存器中的值应为 100 * 2 字节，即 200。作者在编写程序时以为它代表的是传输的数据个数(100)而非字节数(200)，就导致输出的波形总是丢帧，最后将数量设置为 sizeof(buffer)即可，buffer 就是待传输的数组。

- DMA 通道 x 外设地址寄存器(DMA_CPARx)/ DMA 通道 x 存储器地址寄存器(DMA_CMARx) n = 1-7



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
PA[31:0]																															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:0	**PA[31:0]**：外设地址 (Peripheral address) 外设数据寄存器的基地址，作为数据传输的源或目标。 当PSIZE='01'(16位)，不使用PA[0]位。操作自动地与半字地址对齐。 当PSIZE='10'(32位)，不使用PA[1:0]位。操作自动地与字地址对齐。

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
MA[31:0]																															
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:0	**MA[31:0]**：存储器地址 存储器地址作为数据传输的源或目标。

图 24.6　DMA 外设/存储器地址寄存器

这两个寄存器用于设置存储器和外设地址，即数据传输的源/目标地址。

综上所述可以总结配置 DMA 的步骤：

1. 设置 DMA 的通道，传输数据数量，传输方向，源/目标地址，源/目标的数据宽度，是否循环传输，并开启 DMA；
2. 在相应外设中，设置触发 DMA 传输的事件，比如本章是通过定时器发生比较事件时触发 DMA 传输；

24.2.2 WS2812B 讲解

WS2812B 是 WorldSemi 公司推出的外控集成 RGB LED 光源。传统的 RGB LED 是将红绿蓝三个 LED 集成到一个封装中，然后通过 3 路 PWM 对其进行调光，使用这种方式的好处是成本低，但是也存在不足之处：

第一个就是它占用的外设引脚多；

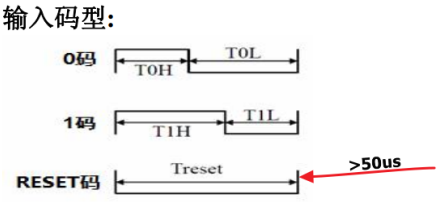
第二个不支持级联 LED 的单独控制，就是说串接在一起的 RGB 灯组不能单独的控制某一个 RGB 的颜色，这些串接的 RGB 在同一时刻都是相同的颜色状态；

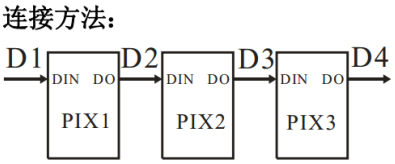
第三个就是传入的 RGB 预期颜色和他们显示出来的颜色存在差异，原因是不同颜色的 LED 的压降不同，如果采用相同的限流电阻，势必会导致表现出来的颜色存在误差。

而 WS2812B 则是将控制 IC 和传统 RGB 结合到了一起，我们使用一条数据线即可驱动所有 RGB，另外也使得颜色控制的精度更高，同时也支持 RGB 灯组的单个 RGB 颜色控制，因此 WS2812B 除了可以用于照明外，还能应用到办公楼外墙来作为屏幕进行广告宣传，此时每个 WS2812B 就对应的是一个像素点。

WS2812B 数据协议采用单线归零码的通讯方式，像素点在上电复位以后，DIN 端接受从控制器传输过来的数据，首先送过来的 24bit 数据（红绿蓝各 8 位）被第一个像素点提取后，送到像素点内部的数据锁存器，剩余的数据经过内部整形处理电路整形放大后通过 DO 端口开始转发输出给下一个级联的像素点，每经过一个像素点的传输，信号减少 24bit。像素点采用自动整形转发技术，使得该像素点的级联个数不受信号传送的限制，仅仅受限信号

传输速度要求。WS2812B 传输协议时序图如下：





数据传输时间(TH+TL=1.25μs±600ns)

800KHz

T0H	0码， 高电平时间	0.4μs	±150ns
T1H	1码， 高电平时间	0.85 μs	±150ns
T0L	0码， 低电平时间	0.85μs	±150ns
T1L	1码， 低电平时间	0.4 μs	±150ns
RES	帧单位，低电平时间	50μs以上	

图 24.7　WS2812B 通信时序

从时序图可以看到，二进制的 0 和 1 用周期相同的不同占空比的方波来表示（1 对应 68%占空比，0 对应 32%占空比），因此我们可以通过改变 PWM 的占空比来模拟出要传输的数据，要实现实时改变 PWM 的占空比首先想到的应该就是利用比较中断，在比较中断发生时更改比较寄存器中的值，但是 WS2812B 协议对传输速度要求非常高，如果使用中断的话时序就会被打乱，因为中断发生时，单片机要先保存现场数据然后进行中断函数的跳转，接着还要判断中断类型并清除中断标志位，完成这些步骤的过程中，就会多产生几次 PWM 波形，最终导致驱动 WS2812B 失败或者显示的颜色是随机的。因此，为了保证实时性，最好的办法是使用硬件级别得响应速度来更改比较寄存器中的值，所以我们使用了 DMA+PWM 这种方法，在比较事件发生时，DMA 立即响应并将对应数据传输到比较寄存器中。也许会有人想到使用最原始的也是最简单粗暴的 IO 口模拟，IO 口模拟就要涉及延时，但是小于 1us 的延时很难做到，可能在判断延时函数是否结束的时候（while 循环判断），延时就已经过去了，即判断过程所耗时间远大于延时时间，对于 IO 口的极限测试，我们后面会讲解。

WS2812 的传输过程如下图：

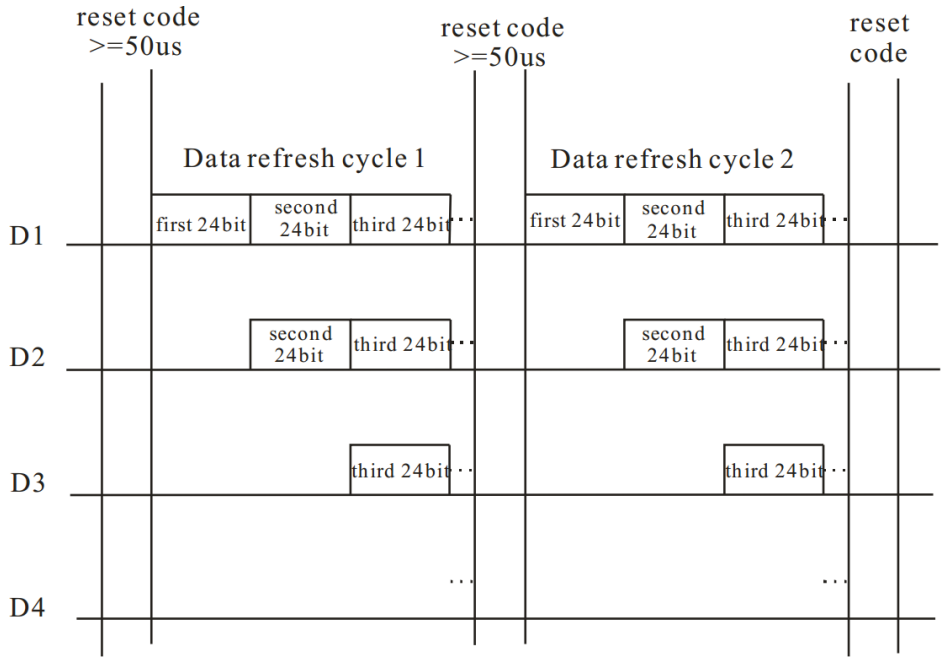


图 24.8　WS2812B 级联传输示意图

每经过一个 WS2812B，数据就被截走一部分，该过程类似于老师为排着队的小朋友发糖果，每个小朋友拿到的糖果颜色取决于老师如何安排。等到最后一个小朋友收到糖果后整个过程结束。

WS2812B 的 24 位数据如下：

24bit 数据结构：

G7	G6	G5	G4	G3	G2	G1	G0	R7	R6	R5	R4	R3	R2	R1	R0	B7	B6	B5	B4	B3	B2	B1	B0

图 24.9　WS2812B 数据帧结构

数据按照高位在前的顺序分别输出绿色，红色和蓝色控制数据。在这里我们顺便说一下 RGB 的取色原理。RGB 由三种基本色构成，分别是红,绿,蓝，也叫加法三原色，通过这三种颜色的不同比例可以组合出各种颜色，而不同比例可以通过 PWM 的占空比来实现。如果想要特定颜色，可以使用调色板取色，如下图：

图 24.10　调色板

24.3 程序讲解

- WS2812B 初始化函数

```
24  void initLight(u8 brightness,u8 colortemp,u8 red,u8 green,u8 blue)
25  {
26      u8 i;
27      GPIO_InitTypeDef GPIO_InitStructure;
28
29      RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA |RCC_APB2Periph_GPIOB | RCC_APB2Periph_AFIO,ENABLE);
30
31      GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);     //禁止JTAG保留SWD
32      GPIO_PinRemapConfig(GPIO_PartialRemap1_TIM2, ENABLE);        //设置JTAG为定时器2部分映射,只使用SWD模式
33
34      /*设置冷光灯*/
35      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15;
36      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
37      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
38      GPIO_Init(GPIOA, &GPIO_InitStructure);
39
40      /*设置暖光灯*/
41      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_3;
42      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
43      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
44      GPIO_Init(GPIOB, &GPIO_InitStructure);
45
46      /*设置RGB*/
47      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
48      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
49      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
50      GPIO_Init(GPIOB, &GPIO_InitStructure);
51
52      setLight(brightness,colortemp);
53      setRGB(red,green,blue);
54  }
```

图 24.11　WS2812B 初始化函数

在上一章的灯光初始化函数的基础上做了升级，添加了 RGB 驱动引脚的初始化，因为使

用的是 PWM 输出，所以要设置为复用推挽输出。并在最后初始化 RGB 颜色。

- RGB 颜色设置函数

```
13  u16 RGB_buffer[69] = {0};

75  /**
76   * 功能：设置RGB颜色
77   * 参数：
78   *          red:RGB红色比例 0-255
79   *          green:RGB绿色比例 0-255
80   *          blue: RGB蓝色比例 0-255
81   * 返回值：None
82   */
83  void setRGB(u8 red,u8 green,u8 blue)
84  {
85      u8 i = 0;
86
87      u32 rgb_value = green<<16 | red<<8 | blue;
88
89      while(DMA_GetFlagStatus(DMA1_FLAG_TC1)==RESET);
90      DMA_ClearFlag(DMA1_FLAG_TC1);
91
92      DMA_Cmd(DMA1_Channel1, DISABLE);
93      DMA_SetCurrDataCounter(DMA1_Channel1,sizeof(RGB_buffer));
94      for(i=0;i<24;++i)
95      {
96          if((rgb_value<<i)&0x800000)     //高位先发，此时高位为1时
97          {
98              RGB_buffer[i+45] = 61;       //68%占空比
99          }else
100         {
101             RGB_buffer[i+45] = 28;       //32%占空比
102         }
103     }
104     DMA_Cmd(DMA1_Channel1, ENABLE);
105 }
```

图 24.12　颜色设置函数

我们将 RGB 的每一位对应的 PWM 占空比存放到 RGB_buffer 数组中，然后通过 DMA 依次将这写数据传输到比较寄存器中，到这里可能会有人感到疑惑：存放占空比数据的数组只需要 24 个数组元素就够用了，为什么要增加到 69 个呢？这是因为每次传输都要进行一次复位，即拉低电平超过 50us，那么我们可以将占空比设置为 0，RGB_buffer 中的前 45 个元素存放的都是 0,因此可以拉低电平的时间为:1.25us * 45 = 56.25us,低电平时间大于 50us，可以满足复位时序要求，然后使用剩下的 24 个数组元素存放对应的 24 位 RGB 颜色值。

如果此时为数组赋值和 DMA 同时发生（几率很小），则有可能会导致 RGB 显示的颜色不正常,所以我们要等到 DMA 传输完成并关掉 DMA 后再对数组进行操作。操作完成后打开 DMA，传输开始，RGB 颜色被更新。

- DMA 初始化函数

```
/**
 * 功能：初始化DMA
 * 参数：
 *          DMA_Chx:设置使用的DMA通道 DMA1_Channel1-DMA1_Channel7
 *          periph_addr:外设地址
 *          memory_addr: 存储器地址
 * 返回值：None
 */
void initDMA(DMA_Channel_TypeDef* DMA_Chx, u32 periph_addr, u32 memory_addr, u16 trans_size)
{
    DMA_InitTypeDef DMA_InitStructure;

    RCC_AHBPeriphClockCmd(RCC_AHBPeriph_DMA1, ENABLE);

    DMA_DeInit(DMA_Chx);

    DMA_InitStructure.DMA_PeripheralBaseAddr = periph_addr;                     //外设地址
    DMA_InitStructure.DMA_MemoryBaseAddr = memory_addr;                         //存储器地址
    DMA_InitStructure.DMA_DIR = DMA_DIR_PeripheralDST;                          //数据方向为存储器到外设
    DMA_InitStructure.DMA_BufferSize = trans_size;                              //单次发送数据个数
    DMA_InitStructure.DMA_PeripheralInc = DMA_PeripheralInc_Disable;            //外设地址不自增
    DMA_InitStructure.DMA_MemoryInc = DMA_MemoryInc_Enable;                     //内存地址自增
    DMA_InitStructure.DMA_PeripheralDataSize = DMA_PeripheralDataSize_HalfWord; //设置外设数据位宽为半字(16位)
    DMA_InitStructure.DMA_MemoryDataSize = DMA_PeripheralDataSize_HalfWord;     //设置存储器数据位宽为半字(16位)
    DMA_InitStructure.DMA_Mode = DMA_Mode_Normal;                               //正常在工作模式，另外一个是循环模式
    DMA_InitStructure.DMA_Priority = DMA_Priority_High;                       //设置最高优先级
    DMA_InitStructure.DMA_M2M = DMA_M2M_Disable;                                //失能存储器到存储器传输
    DMA_Init(DMA_Chx, &DMA_InitStructure);                                      //初始化配置生效

    DMA_Cmd(DMA_Chx, ENABLE);
}
```

图 24.13　DMA 初始化函数

该函数设置了内存变量和外设地址，传输方向为内存到外设，16 位数据宽度，不使用循环模式。

- 定时器 4 输出比较 1 通道初始化函数

```
/**
 * 功能：初始化定时器4输出比较通道1来生成PWM驱动WS2812B
 * 参数：
 *      duty：设置PWM输出的占空比
 * 返回值：None
 */
void initTIM4OC1(u16 duty)
{
    TIM_OCInitTypeDef  TIM_OCInitStructure;                     //定义输出比较初始化结构体

    TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;           //设置PWM1模式
    TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable; //输出使能
    TIM_OCInitStructure.TIM_Pulse = duty;                       //设置捕获比较寄存器的值
    TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;   //设置有效电平为高电平

    TIM_OC1Init(TIM4, &TIM_OCInitStructure);                    //生效初始化设置

    TIM_OC1PreloadConfig(TIM4, ENABLE);                         //使能输出比较预装载功能

    TIM_DMACmd(TIM4,TIM_DMA_CC1,ENABLE);                        //使能捕获比较DMA请求
}
```

图 24.14　定时器输出比较初始化函数

该初始化函数在前面 PWM 章节已经讲过，这里只添加了比较触发 DMA 的设置。当计数器和比较寄存器中的值相同时，DMA 会发送一次数据给比较寄存器。注意：DMA 触发事件发生时，DMA 并不是一次性将多个数据同时发送出去，而是只发送一个数据单元，这里是一个十六位的无符号整形数据。

- 主函数

```
23  int main(void)
24  {
25      /*初始化各外设*/
26      initSysTick();
27      initLED();
28
29      initDMA(DMA1_Channel1, (u32)(&TIM4->CCR1), (u32)RGB_buffer,sizeof(RGB_buffer));
30      initTIMx(TIM4,0,89,TIM_IT_Update,DISABLE);
31      initTIM4OC1(0);
32      /*亮度10%,色温50%,RGB红光*/
33      initLight(0,50,255,0,0);
34
35      while(1)
36      {
37          setRGB(rand()%256,rand()%256,rand()%256);
38          toggleLED();
39          Delay_ms(800);
40      }
41  }
```

图 24.15　主函数

在用户逻辑循环之前先对 DMA 和定时器进行了初始化，在定时器不分频的情况下时钟频率为 72MHz，要实现周期 800KHz（1.25us）则需要 90 个计数值。

在用户逻辑循环中，我们使用随机数生成函数来产生 0-255 范围内的随机值，并将其作为 setRGB 的参数，使 WS2812B 产生随机的颜色。下图是运行效果以及采集到的波形：

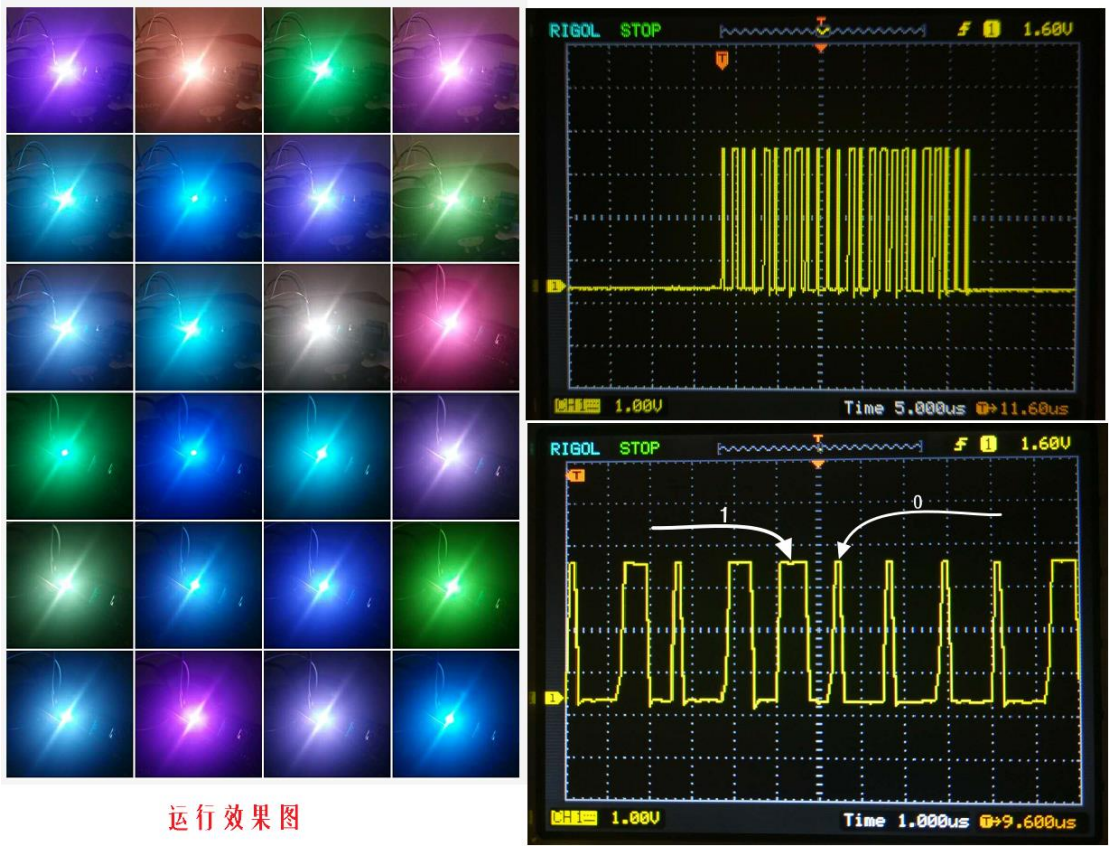


图 24.16　实验效果

如果想要实现驱动多个 WS2812B，就需要增加 DMA 传输数量以及 RGB_buffer 的大小，

buffer 大小和 WS2812 数量的关系为：45 + 24*WS2812B 级联数量。

24.4 时间旅行

现在让我们把时间减慢一千万倍，观察一下下面的代码在千万分之一秒究竟发生了什么。

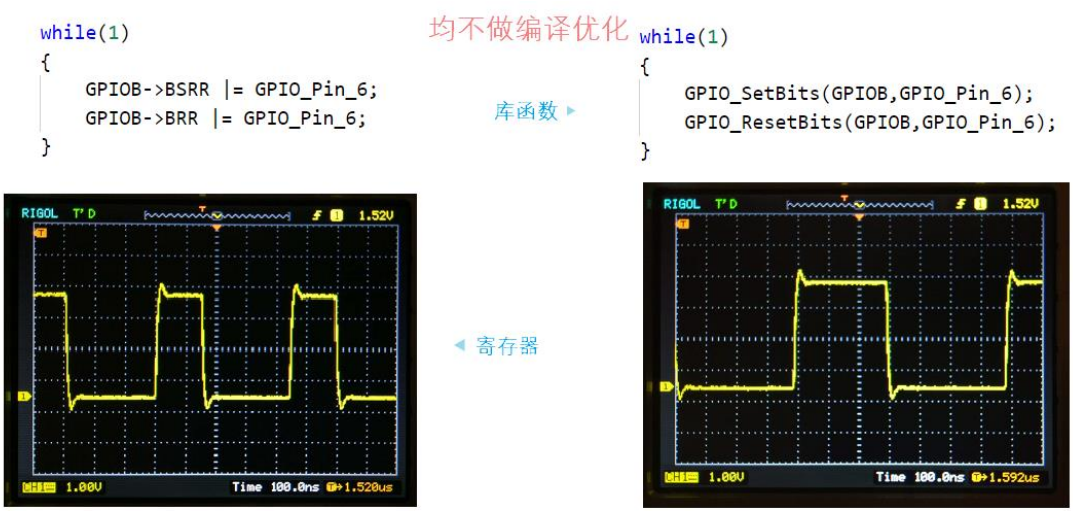


图 24.17　库函数/寄存器操作 GPIO 执行效率对比

高电平的这段时间完成了对 BRR 寄存器的赋值（设置低电平但是还没生效），低电平期间完成了 while 循环条件的真假判断以及对 BSRR 寄存器的赋值（设置高电平但是还没效），

从测试结果来看，while 循环的判断时间为 100ns 左右，使用寄存器操作比使用库函数快了将近一倍（只针对 GPIO 的操作），这种差距在速度和精度要求都非常高的时序上面体现的会更加明显，但是多数应用要求并没有这么高，因此使用库函数和寄存器所带来的时间开销的差距就没有那么重要了。

接下来将上述程序中的 while(1)更换为 for(;;)，经过测试发现，他们之间的时间消耗相同，因此不存在所谓的哪种循环执行起来更加高效。

回到 WS2812B，根据所测量的结果我们完全可以通过 IO 口来模拟时序，然后驱动 WS2812B，但是这样做有很大的局限性，尤其是在高速且精度要求很高的时序上这种局限性会更加明显，试想在本章实验中我们为了让电平保持一段时间，就必须使用延时，而这种延时必须使用示波器去配合，然后不断去测试，直到延时满足要求，即使现在已经调试出了能够精确延时 0.4us 和 0.85us 的延时函数了，但是数据发送还需要使用 for 循环，因此你还要在原有基础上再对延时做修改来满足要求，但是假如编译时使用了不同的优化选项或者使用不同的编译器，其结果又会有很大差别，所以说，**使用 IO 口去模拟高速高精度的时序会使程序兼容性变的很差。**

到这里，我们结束了定时计数器的学习，定时计数器在平时开发中经常会用到，所以我们也使用了很大的篇幅去讲解，其中最重要的也是需要大家掌握的就是输出比较生成 PWM 和输入捕获解码 NEC 红外遥控器协议，因为这些在物联网开发中是使用最频繁的外设。

现在 STM32 的知识我们已经掌握大半，使用之前学习过的知识已经可以做一些小的项目了。作者希望大家在学习完本章后稍作整顿，回顾一下之前学习过的内容，做到温故知新。

第二十五章 环境温湿度采集

25.1 项目要求

通过 DHT11 温湿度传感器采集现场温湿度，并显示在 OLED 屏幕上。

注：本章用到核心板+扩展板，对应例程 19。

25.2 原理讲解

物联网应用中，用到最多的就是环境的温湿度采集，本章我们通过 DHT11 温湿度传感器来采集环境温湿度。

DHT11 数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器。温湿度采集范围为：温度 0-50℃ ±2℃，20-90%RH ±5%RH，它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有很高的可靠性与卓越的长期稳定性。传感器包括一个电阻式感湿元件和一个 NTC 测温元件，并与一个高性能 8 位单片机相连接。DHT11 采用单总线双向串行通信协议，每次采集都要由单片机发起开始信号，然后 DHT11 会向单片机发送响应并开始传输 40 位数据帧，高位在前。这四十位数据由：8bit 湿度整数数据 + 8bit 湿度小数数据 + 8bit 温度整数数据 + 8bit 温度小数数据+8bit 校验位。温湿度的小数部分默认为零，也就是说单片机采集到的温湿度数据都是整数，最后为了数据的有效性，进行了简单的校验和，即：将前 4 个字节的数据相加取结果的低八位数据作为校验和。我们举例说明：

示例一：接收到的 40 位数据为：

0011 0101	0000 0000	0001 1000	0000 0000	0100 1101
湿度高 8 位	湿度低 8 位	温度高 8 位	温度低 8 位	校验位

计算：

0011 0101+0000 0000+0001 1000+0000 0000= 0100 1101

接收数据正确：

湿度：0011 0101=35H=53%RH

温度：0001 1000=18H=24℃

示例二：接收到的 40 位数据为：

0011 0101	0000 0000	0001 1000	0000 0000	0100 1001
湿度高 8 位	湿度低 8 位	温度高 8 位	温度低 8 位	校验位

计算：

0011 0101+0000 0000+0001 1000+0000 0000 = 0100 1101

01001101 不等于 0100 1001

本次接收的数据不正确，放弃，重新接收数据。

图 25.1 数据转换举例

下面来看一下 DHT11 的通信时序：

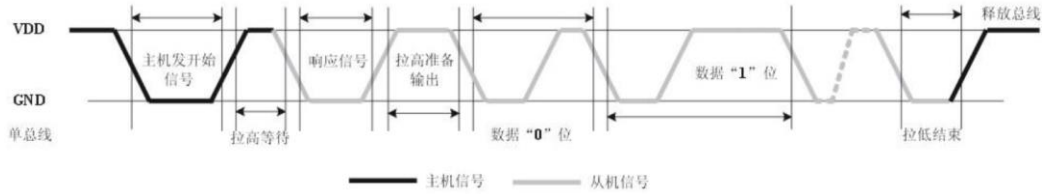


图 25.2　通信时序图

要完成一次采集，主机（单片机）要发送一个开始信号给传感器，步骤为主机先将驱动 IO 设置为输出，然后拉低总线大于 18ms 后设置为输入并释放总线，等待从机响应，主机开始信号时序如下：

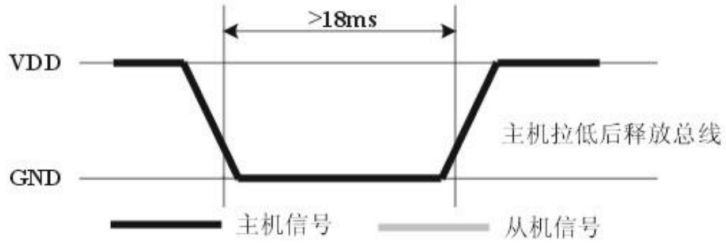


图 25.3　开始信号时序图

如果传感器存在且正常，则会在收到主机的开始信号后拉低总线并持续 80us 来通知主机此时传感器正常，然后拉高总线 80us，通知主机准备接收，传感器响应时序如下：

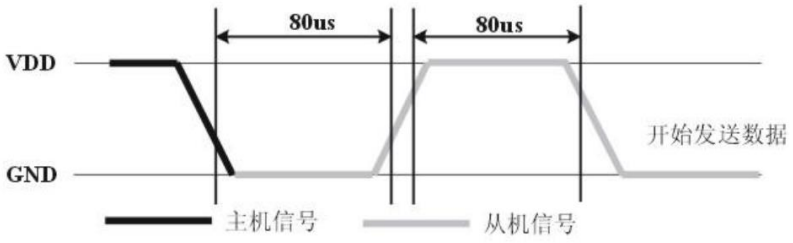


图 25.4　从机响应时序图

接着传感器开始按照高位在前的顺序将数据按照如下格式，一位一位的输出给主机：

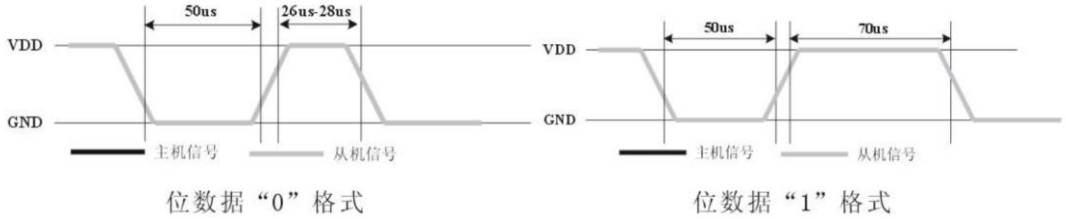


图 25.5　数据时序图

这里就要涉及到程序要如何区分数据 0 和数据 1 的格式，我们的做法是：先判断此时引脚的电平状态，如果是低电平就一直循环等待，直到高电平出现，高电平出现后再延时 40us，并读取延时后的电平状态，如果此时是高电平，则是数据 1，否则是数据 0。

最后传输完 40 位数据后，传感器再次输出一个 50us 的低电平后，将数据总线释放，采

集过程结束。

本章的采集程序就是通过 GPIO 去模拟 DHT11 的时序，这和 GPIO 模拟 IIC 的原理是相同的，本质就是将 GPIO 按照时序切换为输入或者输出。最后程序不要忘记计算校验和。

下面是 DHT11 电路原理图：

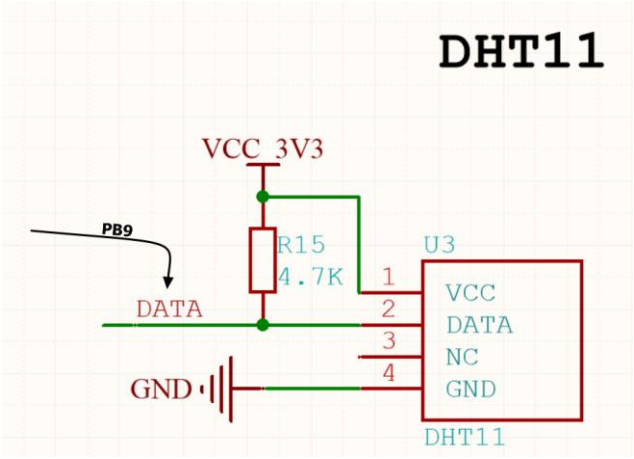


图 25.6　DHT11 电路原理图

25.3 程序讲解

- DHT11 初始化函数

```
/**
 * 功能：初始化DHT11 GPIO
 * 参数：None
 * 返回值：采集到的字节数据
 */
void initDHT11(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(DATA_LINE_CLOCK, ENABLE);

    GPIO_InitStructure.GPIO_Pin = DATA_LINE_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(DATA_LINE_PORT,&GPIO_InitStructure);

    GPIO_SetBits(DATA_LINE_PORT,DATA_LINE_PIN);

    Delay_ms(1500);                                    //开机稳定1.5S
}
```

图 25.7　DHT11 初始化函数

开机默认输出模式，保持高电平。DHT11 开机后会有 1S 左右的不稳定时间，因此在初始化时有必要加入延时，跳过这一阶段，这里设置延时为 1.5S。

- 设置数据线输入输出函数

```
/**
 * 功能：设置数据线为输入
 * 参数：None
 * 返回值：None
 */
static void setLineIn(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin = DATA_LINE_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;

    GPIO_Init(DATA_LINE_PORT,&GPIO_InitStructure);
}

/**
 * 功能：设置数据线为输出
 * 参数：None
 * 返回值：None
 */
static void setLineOut(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin = DATA_LINE_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;

    GPIO_Init(DATA_LINE_PORT,&GPIO_InitStructure);
    GPIO_SetBits(DATA_LINE_PORT,DATA_LINE_PIN);
}
```

图 25.8　数据线方向设置函数

数据线在发起采集开始信号时要设置为输出模式，采集过程要设置为输入，这点和 IIC 比较像，函数实现上也和 IIC 章节对应函数类似。

- 采集单字节函数

```c
/**
 * 功能：读取单个字节,整个读取流程需要调用该函数4次
 * 参数：None
 * 返回值：采集到的字节数据
 */
static u8 readByte(void)
{
    u8 i,byte = 0;
    u8 j = 0;

    for(i=0;i<8;++i)
    {
        while(!GPIO_ReadInputDataBit(DATA_LINE_PORT,DATA_LINE_PIN) && ++j<10)      //等待低电平结束
        {
            Delay_us(10);
        }
        j = 0;

        Delay_us(30);                                                              //延时 用于后续判断
        if(GPIO_ReadInputDataBit(DATA_LINE_PORT,DATA_LINE_PIN))                    //延时后为高电平则为二进制1
        {
            byte <<= 1;
            byte |= 0x01;
        }else
        {
            byte <<= 1;
        }

        while(GPIO_ReadInputDataBit(DATA_LINE_PORT,DATA_LINE_PIN) && ++j<10)       //等待剩余高电平
        {
            Delay_us(10);
        }
    }
    return byte;
}
```

图 25.9　读取字节函数

读取一个字节函数是整个读取过程的基础。其原理就是在 50us 低电平耗尽后，延时一段时间，这段延时时间（程序中设置为 40us）大于二进制 0 的高电平并小于二进制 1 的低电平，如下图：

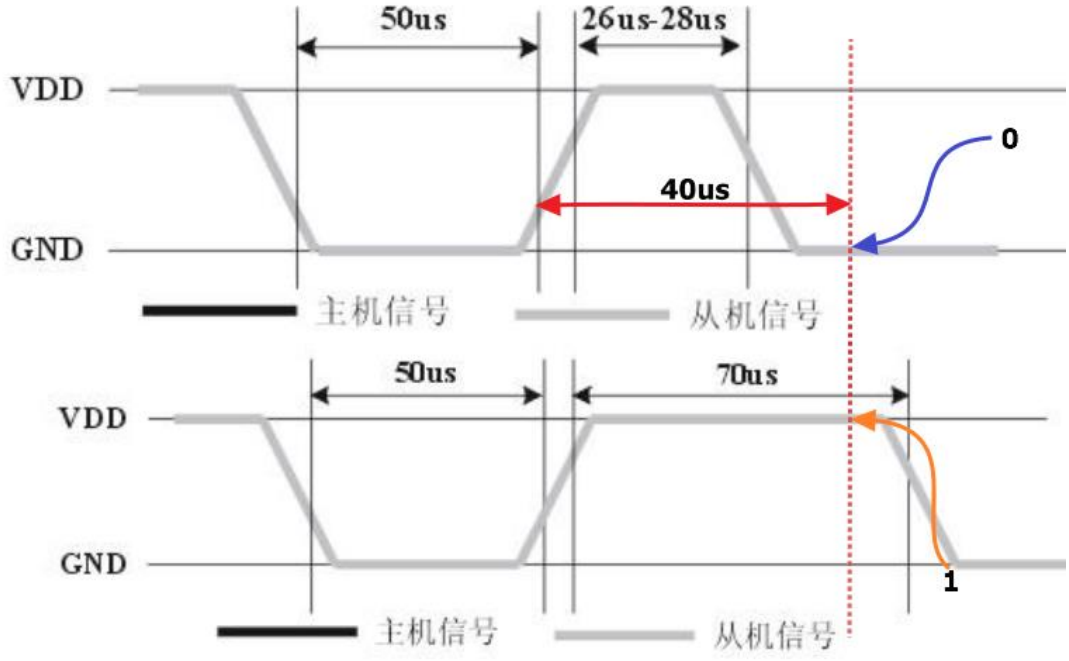


图 25.10　数据识别示意图

- 读取温湿度

```
/**
 * 功能：读取DHT11采集到的数据
 * 参数：
 *        mode：返回数据类型 用户可以指定返回温湿度当中的某一个或者所有
 *              返回所有数据时，湿度在高八位，温度在低八位
 * 返回值：采集到的字节数据
 *
 */
u16 readDHT11(void)
{
    u8 i = 0;

    u8 Hum_H,Hum_L,Temp_H,Temp_L,CheckByte;                                 //依次是湿度整数、小数部分、温度整数、小数部分、校验和字节

    setLineOut();                                                           //发起读取开始信号
    LINE_LOW();
    Delay_ms(25);
    LINE_HIGH();

    setLineIn();

    while(GPIO_ReadInputDataBit(DATA_LINE_PORT,DATA_LINE_PIN) && ++i<10)    //主机释放总线空闲阶段
    {
        Delay_us(10);
    }
    i = 0;
    while(!GPIO_ReadInputDataBit(DATA_LINE_PORT,DATA_LINE_PIN) && ++i<10)  //传感器响应阶段
    {
        Delay_us(10);
    }

    i = 0;
    while(GPIO_ReadInputDataBit(DATA_LINE_PORT,DATA_LINE_PIN) && ++i<10)   //传感器拉高通知主机准备阶段
    {
       Delay_us(10);
    }

    Hum_H = readByte();                                                     //读取有效湿度值
    Hum_L = readByte();                                                     //湿度小数部分为0
    Temp_H = readByte();                                                    //读取有效温度值
    Temp_L = readByte();                                                    //温度小数部分为0
    CheckByte = readByte();                                                 //校验和

    if(Hum_H+Hum_L+Temp_H+Temp_L==CheckByte)                                //计算校验和
    {
        return Hum_H<<8 | Temp_H;

    }else
    {
        return 0xFFFF;                                                      //返回无效值，标识采集有误
    }

}
```

图 25.11　DHT11 读取函数

如果采集失败则返回 0xFFFF，0xFFFF 是个无效数据，因为任何情况下温湿度都不能等于此值（255），所以可以依据此值进行错误数据判断。

- 主函数

```
39      while (1)
40      {
41          if(++i>4)       //每隔2S采集一次 建议采集间隔最小不要小于2S
42          {
43              i = 0;
44              data = readDHT11();
45          }
46          showNumber(40,2,data>>8,DEC,3,FONT_16_EN);
47          showNumber(40,4,data&0x00FF,DEC,3,FONT_16_EN);
48          showNumber(40,6,getConvValueAve(5,1000),DEC,4,FONT_16_EN);
49
50          toggleLED();
51          Delay_ms(500);
52
53      }
54  }
55
56
```

图 25.12　用户逻辑循环函数段

在采集温湿度的时候需要注意，官方手册建议采集间隔为 5S 以上，这里我们设置的采集间隔是 2S，小于 2S 时所采集的数据会间歇性的返回 0xFFFF。

另外需要注意的是，如果在你的项目中有大功率的器件工作，比如 DHT11 和抽水电机使用同一电源，这时当电机启动时或者堵转时，DHT11 的采集也会发生变化，其原因是当系统中负载加重后，电源电压会降低，而 DHT11 内部使用的是 ADC 的方式采集温湿度，势必会对其采集结果造成干扰（同样适用于通过 ADC 采集的光照值），比较简单的缓解办法是，在靠近 DHT11 电源的处并联一个大容量电容和一个小容量瓷片电容。大容量电容在正常工作时和电源电压值相同，当电源电压下降后，电容将会放电，缓解电源压力，另外电机启动时会产生高频干扰的杂波，我们可以使用瓷片电容滤掉，一般选择 100nf（104）即可。如果条件允许，建议大功率器件和控制部分分开供电。

下面是运行效果：

图 25.13　运行效果图

第二十六章 SPI 驱动 SX1278

26.1 项目要求

使用 SPI 通信协议来驱动扩展板上的 SX1278 无线模块，并使用两套开发板进行无线数据传输，一套开发板作为网关，负责发送数据，另外一套开发板作为子设备，负责接收数据并显示。

本章实现的功能比较简单，目的是学习 STM32 的 SPI 外设和 SX1278，如果想要实现 LoRa 组网，还需要额外添加组网通信协议，LoRa 组网见下一章。

注：本章用到核心板+扩展板，对应例程 20。

26.2 原理讲解

26.2.1 详解 SPI

单片机应用中，最常用的通信协议主要有三个，前面我们已经介绍了其中两个（UART 和 IIC），本章将介绍第三个通信协议——SPI。

SPI(Serial Peripheral Interface Bus)由摩托罗拉公司开发，它是高速全双工同步串行通信协议。SPI 支持一主多从，这点类似于 IIC，但是又与 IIC 选通从设备的方式不同，IIC 是通过发送从机地址来选通从机，而 SPI 则是通过拉低连接到从机的 NSS 引脚对从机进行选通的。SPI 一般应用由四个引脚组成（一主一从）：

- SCLK（Serial Clock）：串行时钟，由主机发出
- MOSI（Master Output,Slave Input）：主机输出从机输入信号，由主机发出
- MISO（Master Input,Slave Output）：主机输入从机输出信号，由从机发出
- NSS（Slave Selected）：选择信号，由主机发出，一般是低电位有效

下图是主从连接示意图：

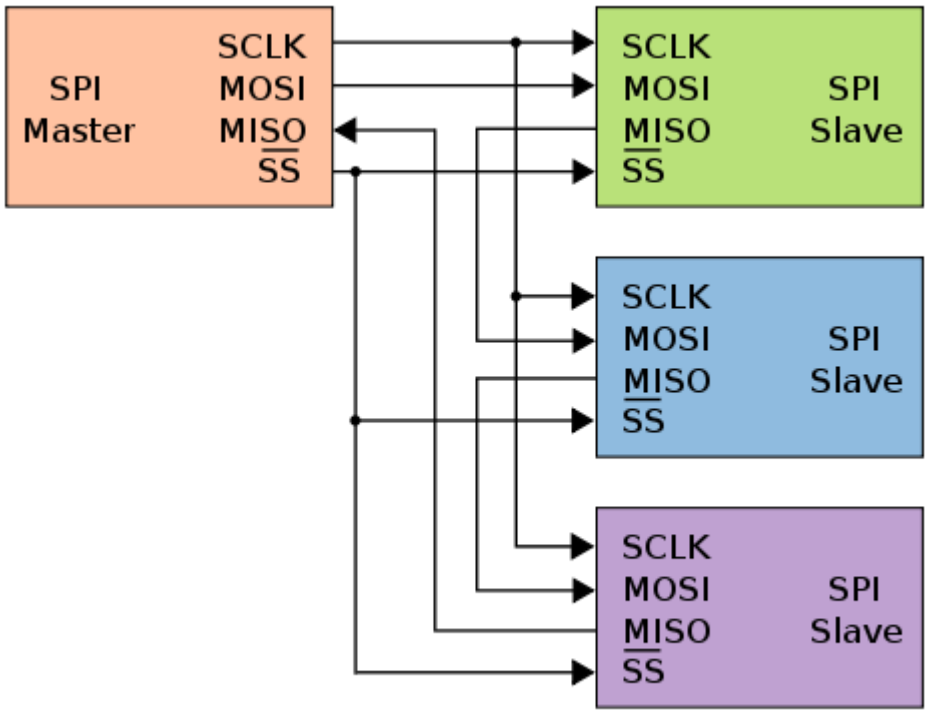


图 26.1　SPI 主从连接示意图

图中可以看出虽然 SPI 也是串行通信协议，但是主机所占用的引脚依然比 IIC 和 UART 的多，而且主机引脚数量会随着从机数量的增加而增加（增加对从机的选通部分）。

主机在通过 MOSI 数据线发送数据的同时，从机也会通过 MISO 将数据传输给主机（收发同时进行），它们以虚拟环形拓扑连接。数据通常先移出最高位，在时钟边沿，主机和从机均移出一位，然后在传输线上输出给对方（改变数据）。在下一个时钟沿，主从设备的接收器都从传输线接受该位，并设置为移位寄存器的新的最低有效位（采样数据）。在完成这样一个移出-移入的周期后，主机和从机就交换了寄存器中的一位，传输可能会持续任意数量的时钟周期。传输完成后，主设备会停止时钟信号，并拉高 NSS 选通线。下图是 SPI 通信时序：

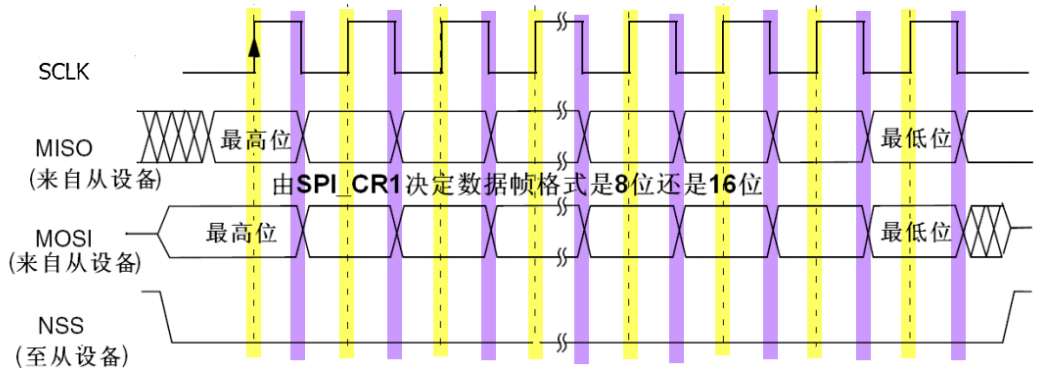


图 26.2　SPI 传输时序图

在上图中，黄色高亮部分是采样部分，紫色高亮部分是数据改变部分。在 NSS 拉低电平选通从设备的前提下，当黄色时钟信号沿发生时，数据线的状态被保存到移位寄存器，高电平对应二进制 1，低电平对应二进制 0。当紫色时钟信号沿发生时，主从设备的移位寄存器均移出一位，并将该位作用到数据线上，即：该时钟信号沿用于改变数据。

SPI 是一种非常灵活的通信协议，我们可以配置它的时钟极性、时钟相位、一次传输的数据位数等。我们依次来看一下：

- 时钟极性(CPOL)

时钟极性用于设置时钟在空闲时的电平状态，CPOL 为 1 则时钟空闲时为高电平，CPOL 为 0,时钟空闲时为低电平，如下图。这就直接导致了第一个信号沿是下降沿还是上升沿，通常 CPOL 和时钟相位(CPHA)配合使用。

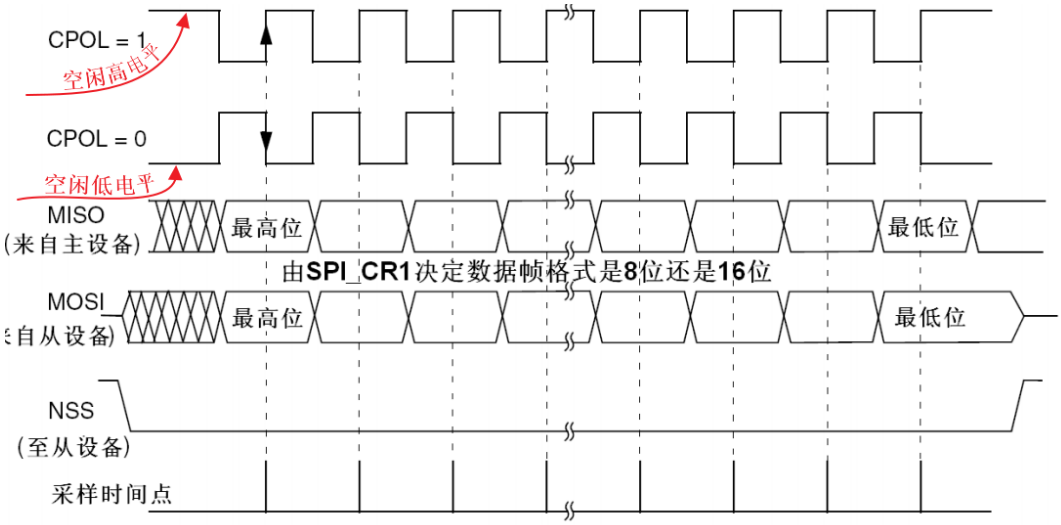


图 26.3　SPI 时钟极性对比

- 时钟相位(CPHA)

时钟相位用于设置在第几个时钟沿发生时对数据线进行采样，当 CPHA=1 时表示在第二个时钟沿对数据线进行采样，当 CPHA 为 0 时，在第一个时钟沿对数据进行采样，如下图：

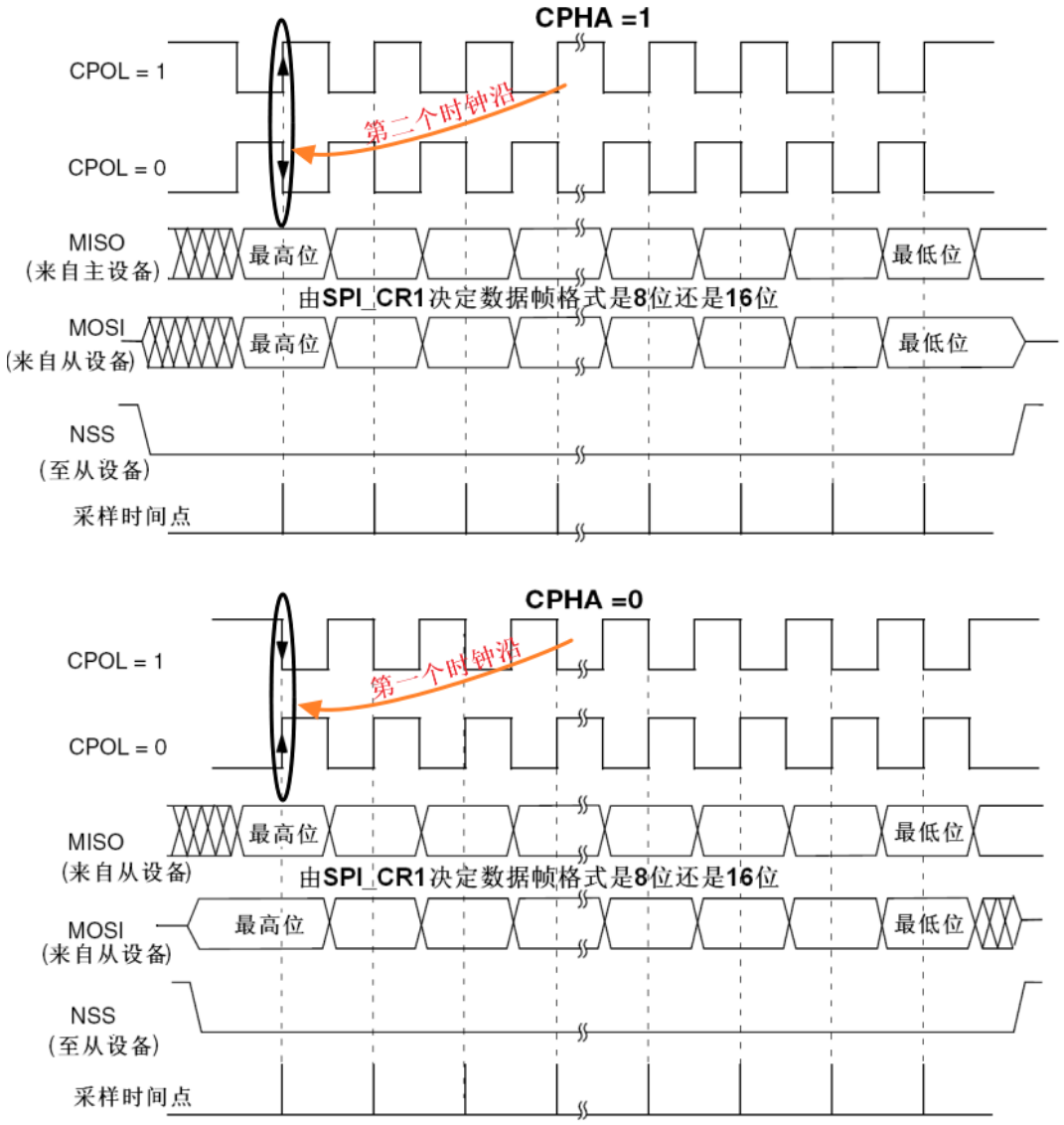


图 26.4　SPI 时钟相位对比

另外在图中可以看到，CPOL 和 CPHA 的组合能够产生 4 中不同的工作模式，以适应不同的使用场景。

我们开发板上的 STM32F103C8T6 含有 2 个 SPI 外设，下图是 SPI 内部结构框图：

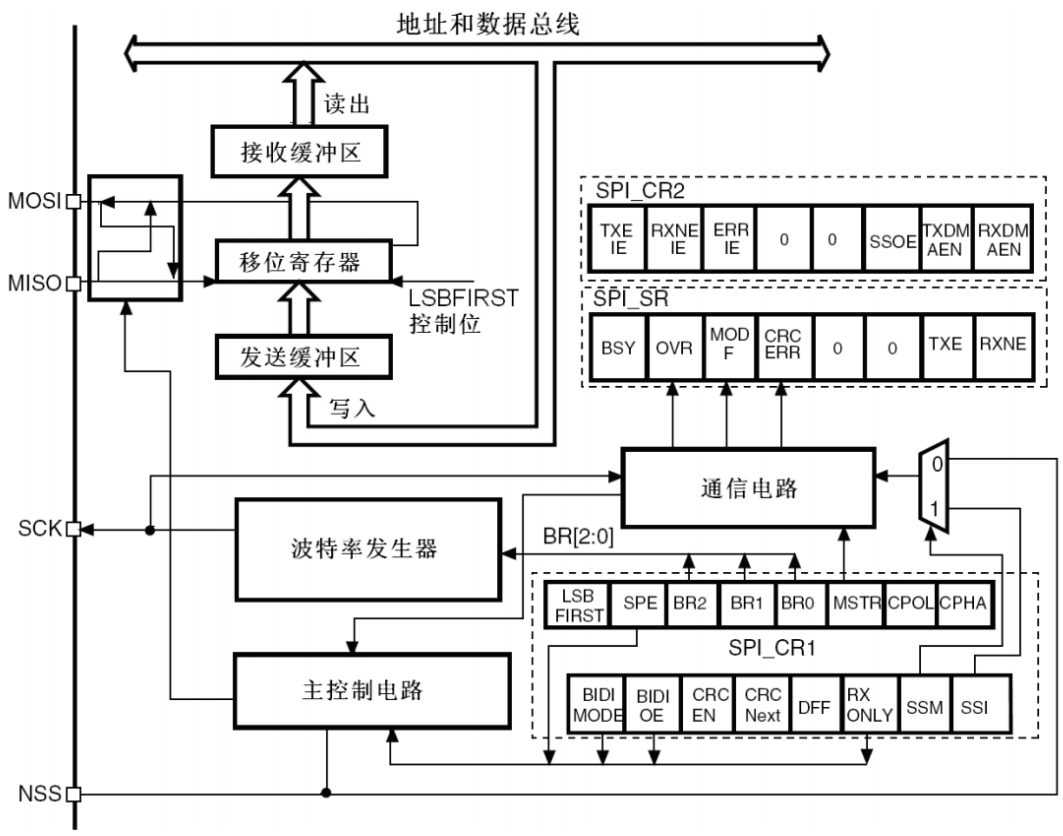


图 26.5　STM32 SPI 内部结构框图

STM32 的 SPI 外设包含两种通信协议，第一个是 SPI 协议，第二个是 I2S 协议，他们通过 SPI_I2S 配置寄存器(SPI_I2S_CFGR)的 11 位 I2SMOD 进行选择。接下来结合 SPI 内部结构框图来介绍一下本例程用到的 SPI 相关寄存器：

- SPI 控制寄存器 1(SPI_CR1)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BIDI MODE	BIDI OE	CRCEN	CRC NEXT	DFF	RX ONLY	SSM	SSI	LSB FIRST	SPE	BR[2:0]			MSTR	CPOL	CPHA

位15	**BIDIMODE**：双向数据模式使能 (Bidirectional data mode enable) 0：选择“双线双向”模式； 1：选择“单线双向”模式。 注：I^2S模式下不使用。

位13	**CRCEN**：硬件CRC校验使能 (Hardware CRC calculation enable) 0：禁止CRC计算； 1：启动CRC计算。 注：只有在禁止SPI时(SPE=0)，才能写该位，否则出错。 该位只能在全双工模式下使用。 注：I^2S模式下不使用。

位11	**DFF**：数据帧格式 (Data frame format) 0：使用8位数据帧格式进行发送/接收； 1：使用16位数据帧格式进行发送/接收。 注：只有当SPI禁止(SPE=0)时，才能写该位，否则出错。 注：I^2S模式下不使用。

位9	**SSM**：软件从设备管理 (Software slave management) 当SSM被置位时，NSS引脚上的电平由SSI位的值决定。 0：禁止软件从设备管理； 1：启用软件从设备管理。 注：I^2S模式下不使用。

位7	**LSBFIRST**：帧格式 (Frame format) 0：先发送MSB； 1：先发送LSB。 注：当通信在进行时不能改变该位的值。 注：I^2S模式下不使用。

位6	**SPE**：SPI使能 (SPI enable) 0：禁止SPI设备； 1：开启SPI设备。 注：I^2S模式下不使用。 注：当关闭SPI设备时，请按照第23.3.8节的过程操作。

位5:3	**BR[2:0]**：波特率控制 (Baud rate control) 000：$f_{PCLK}/2$　001：$f_{PCLK}/4$　010：$f_{PCLK}/8$　011：$f_{PCLK}/16$ 100：$f_{PCLK}/32$　101：$f_{PCLK}/64$　110：$f_{PCLK}/128$　111：$f_{PCLK}/256$ 当通信正在进行的时候，不能修改这些位。 注意：I^2S模式下不使用。

位2	**MSTR**：主设备选择 (Master selection) 0：配置为从设备； 1：配置为主设备。 注：当通信正在进行的时候，不能修改该位。 注：I^2S模式下不使用。
位1	**CPOL**：时钟极性 (Clock polarity) 0：空闲状态时，SCK保持低电平； 1：空闲状态时，SCK保持高电平。 注：当通信正在进行的时候，不能修改该位。 注：I^2S模式下不使用。
位0	**CPHA**：时钟相位 (Clock phase) 0：数据采样从第一个时钟边沿开始； 1：数据采样从第二个时钟边沿开始。 注：当通信正在进行的时候，不能修改该位。 注：I^2S模式下不使用。

图 26.5　SPI 控制寄存器 1

位 15 用于设置 SPI 的传输模式，通常选择双线双向模式，即同时使用 MOSI 和 MISO 数据线。

位 13 可以使能 SPI 的 CRC 校验功能，保证传输数据的有效性。

位 11 用于选择传输数据的单位长度，一般情况下都选择 8 位数据帧格式。

位 9 设置的对象是 NSS 引脚，我们可以通过该位来设置在给从机发送数据之前是通过程序员软件控制 NSS 拉低还是硬件自动拉低，平时都选择使用软件进行控制，这样控制会更灵活

位 7 用于设置发送数据位的顺序，通常选择先发 MSB，即先发高位。

位 6 使能 SPI。

位域[5:3]设置通信速度，这里要注意 SPI1 位于 APB2，SPI2 和 SPI3 位于 APB1，APB2 的 PCLK 最高为 72MHz，APB1 的 PCLK 最高为 36MHz，我们设置波特率的时候还要考虑从机的速度，所以通信系统中的 SPI 通信波特率取决于速度最低的那台设备。

位 2 用来设置该设备的角色，通常将 STM32 设置为主设备，即主机。

位 1 和位 0 用于设置前面介绍过的时钟极性和时钟相位。

- SPI 控制寄存器 2(SPI_CR2)

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								TXEIE	RXNEIE	ERRIE	保留		SSOE	TXDMAEN	RXDMAEN

位7	**TXEIE**：发送缓冲区空中断使能 (Tx buffer empty interrupt enable) 0：禁止TXE中断； 1：允许TXE中断，当TXE标志置位为'1'时产生中断请求。
位6	**RXNEIE**：接收缓冲区非空中断使能 (RX buffer not empty interrupt enable) 0：禁止RXNE中断； 1：允许RXNE中断，当RXNE标志置位时产生中断请求。

图 26.6　SPI 控制寄存器 2

SPI 控制寄存器 2 用于使能对应的 SPI 中断，这里常用的是**位 7** 和**位 6**，一个是发送缓冲区为空中断使能，一个是接收缓冲区非空中断使能，这两个中断其实就是发送完成和接收完成中断使能位。

- SPI 状态寄存器（SPI_SR）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								BSY	OVR	MODF	CRCERR	UDR	CHSIDE	TXE	RXNE

位7	**BSY**：忙标志 (Busy flag) 0：SPI不忙； 1：SPI正忙于通信，或者发送缓冲非空。 该位由硬件置位或者复位。 注：使用这个标志时需要特别注意，详见第23.3.7节和第23.3.8节。

位1	**TXE**：发送缓冲为空 (Transmit buffer empty) 0：发送缓冲非空； 1：发送缓冲为空。
位0	**RXNE**：接收缓冲非空 (Receive buffer not empty) 0：接收缓冲为空； 1：接收缓冲非空。

图 26.7　SPI 状态寄存器

当标志位为 1 时，如果开启了该标志位对应的中断使能，将会发生 SPI 中断。

- SPI 数据寄存器（SPI_DR）

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

位15:0	**DR[15:0]**：数据寄存器 (Data register) 待发送或者已经收到的数据 数据寄存器对应两个缓冲区：一个用于写(发送缓冲)；另外一个用于读(接收缓冲)。写操作将数据写到发送缓冲区；读操作将返回接收缓冲区里的数据。 **对SPI模式的注释：**根据SPI_CR1的DFF位对数据帧格式的选择，数据的发送和接收可以是8位或者16位的。为保证正确的操作，需要在启用SPI之前就确定好数据帧格式。 对于8位的数据，缓冲器是8位的，发送和接收时只会用到SPI_DR[7:0]。在接收时，SPI_DR[15:8]被强制为0。 对于16位的数据，缓冲器是16位的，发送和接收时会用到整个数据寄存器，即SPI_DR[15:0]。

图 26.8　SPI 数据寄存器

发送或者接收数据都是通过写入或读取该寄存器完成的。

- SPI CRC 多项式寄存器(SPI_CRCPR)和 SPI Rx CRC 寄存器(SPI_RXCRCR)

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

位15:0	**CRCPOLY[15:0]**：CRC多项式寄存器 (CRC polynomial register) 该寄存器包含了CRC计算时用到的多项式。 其复位值为0x0007，根据应用可以设置其他数值。 注：在I^2S模式下不使用。

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

位15:0	**RXCRC[15:0]**：接收CRC寄存器 在启用CRC计算时，RXCRC[15:0]中包含了依据收到的字节计算的CRC数值。当在SPI_CR1的CRCEN位写入'1'时，该寄存器被复位。CRC计算使用SPI_CRCPR中的多项式。 当数据帧格式被设置为8位时，仅低8位参与计算，并且按照CRC8的方法进行；当数据帧格式为16位时，寄存器中的所有16位都参与计算，并且按照CRC16的标准。 注：当BSY标志为'1'时读该寄存器，将可能读到不正确的数值。 注：在I^2S模式下不使用。

图 26.9　SPI CRC 校验相关寄存器

这两个寄存器用于计算和存储 CRC 校验值，其中 CRC 寄存器我们设置默认值即可。

- SPI_I2S 配置寄存器（SPI_I2S_CFGR）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留				I2S MOD	I2SE	I2SCFG		PCM SYNC	保留	I2SSTD		CKPOL	DATLEN		CHLEN
res				rw	rw	rw		rw	res	rw		rw	rw		rw

位15:12	保留位，硬件强制为0
位11	**I2SMOD**：I^2S模式选择 (I^2S mode selection) 0: 选择SPI模式； 1: 选择I^2S模式。 注：该位只有在关闭了SPI或者I^2S时才能设置。
位10	**I2SE**: I^2S使能 (I^2S enable) 0: 关闭I^2S； 1: I^2S使能。 注：在SPI模式下不使用。

图 26.10　SPI 模式配置寄存器

因为 STM32 将 SPI 和 I2S 合并到了一起，因此我们在使用 SPI 时要先设置该寄存器的位 11 来指定使用 SPI 功能。

配置 SPI 步骤如下：

1. 初始化 SCK、MOSI 引脚为复用推挽输出，初始化 MISO 引脚为浮空输入；
2. 初始化 NSS 引脚为推挽输出并默认拉高（使用软件 NSS 情况下）；
3. 配置 SPI 初始化结构体，内容包括：主/从模式，高/低位在前，数据帧位数（8 位/16 位），配置时钟极性（CPOL）、时钟相位（CPHA）、软件 NSS、时钟分频值等。

26.2.2 LoRa 介绍

介绍物联网模型的章节我们说过，网关结构是为了解决联网设备成本高，服务器负荷大的问题而出现的，试想智能城市中需要采集和控制的设备数量相当可观，如果全部以联网的形式（比如 GPRS）将设备接入服务器，将会对服务器造成很大的负担，同时设备的制造成本

和维护成本也十分惊人，因此最好的办法就是使用网关的形式，即一主多从的星型网络拓扑结构，比如设备连接服务器以小区为单位，将极大的降低成本和维护的开销，服务器的负担也将减轻。目前实现主从通信按照传输介质分为有线传输和无线传输，其中有线传输用的比较多的是 RS485 和 CAN，无线用的最多的是 ZigBee 和 LoRa 以及 2.4G 频段的无线通信协议，对于有线传输来说，其传输距离、抗干扰性以及应用场合均没有无线传输的方式灵活，但是有线传输的成本会更低一些，因此选择哪一种传输方式还要看具体应用，具体要求。

今天主要讲解一下 LoRa，在讲解 LoRa 之前先了解一个概念：低功耗广域网（Low Power Wide Area Network，LPWAN），LPWAN 泛指具有低功耗，远距离通信特点的 M2M 星型网络结构，即在前面一直提到的主从网关结构。LoRa 是 LPWAN 的一种，LoRa 技术是由 Semtech 公司提出并推广的一种超低功耗，远距离，低速率传输的无线应用技术。它工作在全球免费频段，如 433MHz，868MHz，915MHz，国内常用 433MHz 作为 LoRa 的运行频段。

我们开发板上使用的 LoRa 模块由凌承芯提供，模块上的射频芯片由 Semtech 公司研发，该模块通过 SPI 接口进行通信设置和数据收发，理想传输距离为 5000 米，作者在小区中进行距离测试试验。在扩频因子为 7，带宽为 9，编码率为 2 的情况下将发射模块放于 30 米高处，在 1.2KM 范围内，无丢包发生，在大于 1.2Km 时，偶尔发生丢包，最远测试距离为 1.7KM，如下图：

图 26.11　SX1278 通信距离测试

26.3 程序讲解

SX1278 的驱动程序分为两部分，第一部分是 SPI 的配置，第二部分是对 SX1278 的设置，以及数据收发函数的封装，另外 SX1278 是单双工的射频芯片，因此要想实现两板的通信，就必须保证一个开发板处于发射模式，另一个开发板处于接收模式。

26.3.1 SPI 程序讲解

- SPI 初始化函数

```
#include "SPIx/SPIx.h"

/**
 * 功能：初始化SPI相关引脚和SPI外设工作状态
 * 参数：
 *       SPIx：指定待初始化的SPI，SPI1-SPI3
 * 返回值：None
 */
void initSPIx(SPI_TypeDef* SPIx)
{
    GPIO_InitTypeDef GPIO_InitStructure;
    SPI_InitTypeDef  SPI_InitStructure;

    if(SPIx==SPI1)
    {
        /*按需添加*/
    }else if(SPIx==SPI2)
    {
        RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA|RCC_APB2Periph_GPIOB, ENABLE);
        RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);                //使能SPI2时钟
    }else if(SPIx==SPI3)
    {
        /*按需添加*/
    }else
    {
       /*按需添加*/
    }
    /**********************SPI GPIO配置**************************/
    if(SPIx==SPI1)
    {
       /*按需添加*/
    }else if(SPIx==SPI2)
    {
        GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_8;                      //配置PA8为NSS选通引脚
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_Init(GPIOA, &GPIO_InitStructure);
        GPIO_SetBits(GPIOA,GPIO_Pin_8);                                 //默认拉高

        GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_13|GPIO_Pin_15;         //配置MOSI(PB15)和SCK(PB13)
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
        GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
        GPIO_Init(GPIOB, &GPIO_InitStructure);

        GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_14;                     //配置MISO(PB14)
        GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
        GPIO_Init(GPIOB, &GPIO_InitStructure);
    }else if(SPIx==SPI3)
    {
        /*按需添加*/
    }else
    {
       /*按需添加*/
    }
    /**********************SPI 工作模式**************************/
    SPI_InitStructure.SPI_Direction = SPI_Direction_2Lines_FullDuplex;  //设置两线全双工
    SPI_InitStructure.SPI_Mode = SPI_Mode_Master;                       //设置为SPI主机
    SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;                   //8位数据帧格式
    SPI_InitStructure.SPI_CPOL = SPI_CPOL_Low;                          //时钟线空闲低电平
    SPI_InitStructure.SPI_CPHA = SPI_CPHA_1Edge;                        //第一个时钟沿(上升沿)采样，和SX1278一致
    SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;                           //软件控制NSS选通
    SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_128;//没有速度要求尽量设置的慢一点，这样可以兼容更多设备
    SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;                  //高位在前传输，和SX1278一致
    SPI_InitStructure.SPI_CRCPolynomial = 7;                            //设置默认值即可
    SPI_Init(SPIx, &SPI_InitStructure);                                 //配置生效

    SPI_Cmd(SPIx, ENABLE);                                                  //开启SPI
}
```

图 26.12　SPI 初始化函数

我们赋予了 SPI 初始化函数更大的灵活性，用户如果在今后使用其他 SPI 外设(SPI1 和 SPI3)，在相应判断位置添加代码即可。在初始化 SPI 之前初始化了 SPI 使用到的引脚，其中要将 MOSI，SCLK 配置为复用推挽输出，MISO 配置为浮空输入，NSS 配置为推挽输出并默认拉高。接下来是 SPI 的初始化，SPI 的初始化要根据 SX1278 的 SPI 模式而定，下图是 SX1278 的 SPI 操作时序图：

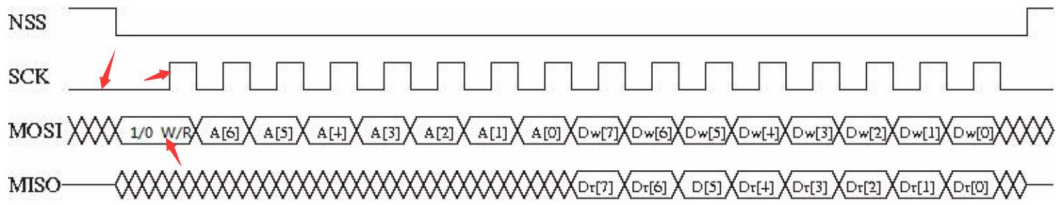


图 26.13　SX1278 SPI 时序

从图中可以看出，SX1278 使用的 SPI 模式为：时钟空闲低电平，第一个时钟沿采样，高位在前传输，8 位数据帧，因此在配置 STM32 的 SPI 时也要遵循该时序约定，在配置完 SPI 之后开启 SPI 即可。

- SPI 发送单字节函数

```
96  /**
97   * 功能：发送一字节
98   * 参数：
99   *       SPIx:指定SPI
100  *       byte:待发送数据
101  * 返回值：None
102  */
103 void sendSPIxByte(SPI_TypeDef* SPIx, u16 byte)
104 {
105     while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_TXE) == RESET);     //等待发送缓冲区为空
106     SPI_I2S_ClearFlag(SPIx,SPI_I2S_FLAG_TXE);
107
108     SPI_I2S_SendData(SPIx, byte);
109
110     while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_RXNE) == RESET);     //等待接收缓冲区非空
111     SPI_I2S_ClearFlag(SPIx,SPI_I2S_FLAG_RXNE);
112
113     SPI_I2S_ReceiveData(SPIx);
114 }
```

图 26.14　SPI 发送单字节函数

在发送数据之前先要判断发送缓冲区是否可用，如果可用再进行发送动作，由于 SPI 收发是同时进行的，因此接收缓冲区如果非空就说明数据接收结束，则数据发送也结束，另外最后一句用于读取接收到的数据，对于发送数据来说，接收数据无意义，**但如果注释掉该条语句，SPI 发送函数将会卡死，因此需要加上。**

- SPI 接收字节函数

```
115 /**
116  * 功能：接收一字节
117  * 参数：
118  *       SPIx:指定SPI
119  * 返回值：读取到的数据
120  */
121 u16 readSPIxByte(SPI_TypeDef* SPIx)
122 {
123     while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_TXE) == RESET);     //等待发送缓冲区为空
124     SPI_I2S_ClearFlag(SPIx,SPI_I2S_FLAG_TXE);
125 
126     SPI_I2S_SendData(SPIx, 0x31);
127 
128     while (SPI_I2S_GetFlagStatus(SPIx, SPI_I2S_FLAG_RXNE) == RESET);     //等待接收缓冲区非空
129     SPI_I2S_ClearFlag(SPIx,SPI_I2S_FLAG_RXNE);
130 
131     return SPI_I2S_ReceiveData(SPIx);
132 }
133 
```

图 26.15　SPI 接收单字节函数

SPI 读取函数和 SPI 发送函数的内容相同，如果想要接收到数据，主机就必须先发送一个字节出去，然后才会收到从机发来的数据（因为时钟由主机提供），对于发送的内容，随意写一个字节即可，我们这里写的是 0xFF。

26.3.2 SX1278 程序讲解

讲解程序之前，先熟悉几个和 SX1278 相关的概念。第一个是信道，每个信道对应不同的通信频率，各个信道之间的信号由于传输频率不同，因此不会造成干扰，这将会大大提高 LoRa 组网的从机个数。第二个是数据速率(DR)，即信号传输的速度，这一指标由带宽（BW），扩频因子（SF）和编码率（CR）来共同决定，其计算公式为：

$$DR= SF*(BW/2^SF)*CR$$

高速率意味着传输速度的提升，但是通信距离会降低，反之低速率虽然传输速度降低，但是传输距离会增加。

SX1278 的驱动函数有很多，大多数都是设置其工作参数的，这里我们只讲解经常被用到的函数。

- 读/写 SX1278 函数

```
/*************************读写SX1278的基础函数*******************************/
/**
 * 功能：向SX1278指定地址中写入数据，通常用作更改某寄存器中的值
 * 参数：
 *       addr:寄存器地址，寄存器地址参考头文件或者官方手册
 *            读或写由地址最高位决定
 *       byte:待发送数据
 * 返回值：None
 */
void writeSX1278(unsigned char addr, unsigned char buffer)
{
    setNSS(0);                      //拉低NSS
    sendSPIxByte(SPI2, addr|0x80);
    sendSPIxByte(SPI2, buffer);
    setNSS(1);                      //拉高NSS
}

/**
 * 功能：读取SX1278返回的数据
 * 参数：
 *       addr:寄存器地址，寄存器地址参考头文件或者官方手册
 *            读或写由地址最高位决定
 * 返回值：返回指定寄存器当中的值
 */
unsigned char readSX1278(unsigned char addr)
{
    unsigned char Value;

    setNSS(0);                      //拉低NSS
    sendSPIxByte(SPI2, addr&0x7f);
    Value = readSPIxByte(SPI2);
    setNSS(1);                     //拉高NSS

    return Value;
}
```

图 26.16　读写 SX1278 寄存器函数

这两个函数是操作 SX1278 最基础的函数，写函数用于在指定的寄存器地址中，写入要设置的参数，接收数据则是读取指定寄存器中的数值。

- SX1278 工作状态设置函数

```
/**
 * 功能：设置SX1278工作状态
 * 参数：
 *       opMode:操作模式 取值见RFMode_SET枚举
 * 返回值：None
 */
void setSX1278OpMode( RFMode_SET opMode )
{
    unsigned char opModePrev;
    opModePrev=readSX1278(REG_LR_OPMODE);      //读0x01模式寄存器
    opModePrev &=0xf8;                         //清零低三位
    opModePrev |= (unsigned char)opMode;       //或上形参
    writeSX1278(REG_LR_OPMODE, opModePrev);    //重新写回去
}
```

图 26.17　SX1278 工作状态设置函数

该函数用于设置 SX1278 的工作状态，主要用于设置是否处于休眠模式，接收模式或者发送模式等。

- SX1278 工作频率设置

```
/**
 * 功能：设置SX1278射频频率
 * 参数：None
 * 返回值：None
 * 补充：射频频率= (Frequency[0]<<16 | Frequency[1]<<8 |  Frequency[2])*61.035
 *       即射频频率 = 0x6C8000 * 61.035 = 433.99MHz
 */
void setSX1278RFFrequency(void)
{
    writeSX1278( REG_LR_FRFMSB, Frequency[0]);  //写0x06寄存器
    writeSX1278( REG_LR_FRFMID, Frequency[1]);  //写0x07寄存器
    writeSX1278( REG_LR_FRFLSB, Frequency[2]);  //写0x08寄存器
}
```

图 26.18　SX1278 工作频率设置函数

该函数就是用于设置 SX1278 工作的频率，即信道，频率值保存在 Frequency 数组中，如下：

```
unsigned char   Frequency[3]={0x6c,0x80,0x00};
```

图 26.19　SX1278 工作频率存储数组

如果想要更改工作频率只需按照注释中提供的公式计算即可，建议每个信道间隔 3MHz，频率设置范围建议在 410MHz-525MHz 之间。如果要实现多通道，则可以将各个通道频率设置为二维数组。

- SX1278 负载字节长度设置函数

```
/**
 * 功能：设置SX1278负载字节长度
 * 参数：
 *         value：负载字节长度，最大值256,调用时要注意
 * 返回值：None
 */
void setSX1278PayLoadLength(unsigned char value)
{
    writeSX1278( REG_LR_PAYLOADLENGTH, value );  //写0x22寄存器，RegPayloadLength
}
```

图 26.20　SX1278 负载字节长度设置函数

负载字节长度就是指一次发送的数据包中的字节数量，SX1278 支持最大的数据包字节数量为 256 个。

- SX1278 复位函数

```
/**
 * 功能：复位SX1278,只在初始化之前复位
 * 参数：None
 * 返回值：None
 */
void resetSX1278(void)
{
    GPIO_ResetBits(GPIOB,GPIO_Pin_12);
    Delay_ms(10);
    GPIO_SetBits(GPIOB,GPIO_Pin_12);
    Delay_ms(10);
}
```

图 26.21　SX1278 复位函数

官方规定：在初始化 SX1278 之前必须必须先对 SX1278 进行复位，复位流程图如下：

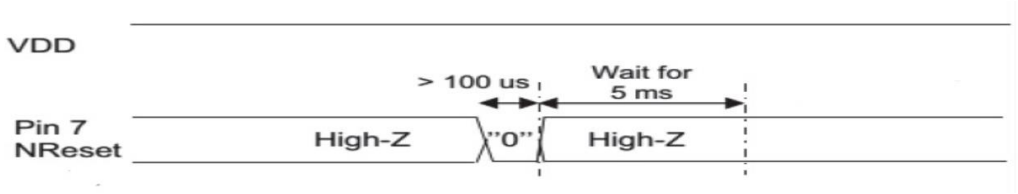


图 26.22　SX1278 复位时序图

平时工作时要将 RESET 复位线拉高，如果要复位，就必须拉低 RESET 复位线超过 100us。

- 初始化 SX1278

```
/**
 * 功能：初始化SX1278
 * 参数：None
 * 返回值：None
 */
void initSX1278(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);

    GPIO_InitStructure.GPIO_Pin  = GPIO_Pin_12;            //配置PB12为SX1278复位引脚
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    GPIO_SetBits(GPIOB,GPIO_Pin_12);

    resetSX1278();                                         //复位设备
    setSX1278OpMode(Sleep_mode);                           //设置睡眠模式
    setSX1278LoRaFSK(LORA_mode);                           // 设置扩频模式
    setSX1278OpMode(Stdby_mode);                           // 设置为普通模式
    writeSX1278( REG_LR_DIOMAPPING1,GPIO_VARE_1);          //写0x40寄存器,DIO引脚映射设置，设为00.
    writeSX1278( REG_LR_DIOMAPPING2,GPIO_VARE_2);          //写0x41寄存器
    setSX1278RFFrequency();                                //频率设置
    setSX1278RFPower(powerValue);                          //功率设置
    setSX1278SpreadingFactor(SpreadingFactor);             // 扩频因子设置
    setSX1278ErrCode(CodingRate);                          //纠错编码率设置
    setSX1278PacketCRC(true);                              //CRC 校验打开
    setSX1278BandWidth(Bw_Frequency);                      //设置扩频带宽，125khz
    setSX1278ImplicitHeader(false);                        //同步头是显性模式
    setSX1278PayLoadLength(0xff);                          //设置有效负载长度
    setSX1278SymbTimeout(0x3FF);                           //设置接收超时时间，TimeOut = SymbTimeout * Ts。
    setSX1278MobileNode(true);                             //低数据率优化，当有效数据速率较低时，必须使用LowDataRateOptimize位提高LoRa链路的鲁棒性。
    writeSX1278(REG_LR_IRQFLAGS, 0xff);
    setSX1278Receive();                                    //设置为连续接收模式
}
```

图 26.23　SX1278 初始化函数

该函数在设备上电或者复位后用于设置 SX1278 的工作参数，这些参数决定了 SX1278 的工作频率，工作速率，射频发射功率等等。这些函数我们都做了详细的注释，这里就不一一讲解。

- 数据包发送/接收函数

```
void transmitPackets(unsigned char *buffer,unsigned char len)
{
    unsigned char i;

    setSX1278OpMode( Stdby_mode );
    writeSX1278( REG_LR_HOPPERIOD, 0 );                //不做频率跳变
    writeSX1278(REG_LR_IRQFLAGSMASK,IRQN_TXD_Value); //打开发送中断
    writeSX1278( REG_LR_PAYLOADLENGTH, len);           //最大数据包设置
    writeSX1278( REG_LR_FIFOTXBASEADDR, 0);            //写Tx FIFO基址
    writeSX1278( REG_LR_FIFOADDRPTR, 0 );              //FIFO指针

    setNSS(0);                                       //拉低NSS
    sendSPIxByte(SPI2,0x80);                           //0X00 | 0X80，突发访问，写
    for(i = 0;i<len;i++)                               //将待发送数据发送到FIFO并等待发送
    {
        sendSPIxByte(SPI2,*buffer);
        buffer++;
    }
    setNSS(1);                                         //拉高NSS
    writeSX1278(REG_LR_DIOMAPPING1,0x40);              //设置0x40寄存器为0100 0000b，即设置发射完成指示映射到DIO0引脚
    writeSX1278(REG_LR_DIOMAPPING2,0x00);
    setSX1278OpMode(Transmitter_mode);                 //设置为传输模式

    while((readSX1278(REG_LR_IRQFLAGS)&0x08) != 0x08)//等待发送完成
    {
        Delay_ms(20);
    }

    writeSX1278(REG_LR_IRQFLAGS, 0xff);                //清零所有标志位，所有的DIOx口都会恢复低电平
}
```

```
352  void receivePackets(unsigned char* buffer)
353  {
354      unsigned char i;
355      unsigned char len;
356
357      setSX1278OpMode( Stdby_mode );
358      writeSX1278(REG_LR_IRQFLAGSMASK, IRQN_RXD_Value);    //打开接收中断
359      writeSX1278(REG_LR_HOPPERIOD,    PACKET_MIAX_Value);//0x24寄存器，设置频率跳变周期为最大
360      writeSX1278( REG_LR_DIOMAPPING1, 0X00);             //端口映射恢复为默认
361      writeSX1278( REG_LR_DIOMAPPING2, 0x00);
362      setSX1278OpMode(Receiver_mode);
363
364      if((readSX1278(REG_LR_IRQFLAGS)&0x40)==0x40)             //接收完成
365      {
366          if((readSX1278(REG_LR_MODEMCONFIG2)&0x04)==0x04)     //是否CRC校验完成
367          {
368              writeSX1278 (REG_LR_FIFOADDRPTR,0x00);           //设置SPI接口在FIFO缓冲区中的地址指针
369              len = readSX1278(REG_LR_NBRXBYTES);              //读取最后一个包的字节数
370              setNSS(0);
371              sendSPIxByte(SPI2,0x00);                         //0X00 | 0X00，突发访问，读。
372
373              for(i=0;i<len;++i)                               //读取接收到的数据到指定数组
374              {
375                  buffer[i]= readSPIxByte(SPI2);
376              }
377              setNSS(1);
378          }
379      }
380  }
```

图 26.24　SX1278 数据包收/发函数

在前面做的所有工作都是为了封装这两个函数，一个用于将 SX1278 设置为发射模式并向空中设置指定的数据包数据，另外一个用于接收发送端发送过来的数据包。

判断数据发送完成或者接收完成有两种方式：

1. 使用 SX1278 的 DIO0 引脚作为指示；
2. 通过 SPI 查询 SX1278 的中断寄存器对应标志位

在本章程序中使用的是第二种方式。另外在发送数据时，建议不要太过频繁，因为 LoRa 本身并不是为了适应高实时性、大吞吐量的应用场合而设计的，在平时使用时，建议以秒为单位，这样做的好处是可以降低整机功耗，延长电池使用寿命。

最后来看一下网关和子设备的主函数：

- 网关设备主函数

```
46      while (1)
47      {
48          transmitPackets(&tx,1);
49
50          ++tx;
51          printf("hello,now is %d\n",tx);
52          toggleLED();
53          Delay_ms(1000);
54
55      }
56  }
```

图 26.25　网关主函数

函数实现的业务逻辑为：每隔 1S 向外发射一次数据，数据内容是一个自加的八位无符号整形变量，在子设备端可以通过查看接收数据的连续性来获知是否出现丢包。

- 子设备主函数

```
47      while (1)
48      {
49          receivePackets(&rx);
50          printf("%d\n",rx);
51          showNumber(0,2,rx,DEC,3,FONT_16_EN);
52          toggleLED();
53          Delay_ms(500);
54      }
55  }
```

图 26.26　子设备主函数

子设备实现的业务逻辑为：每隔 500ms 查询一次接收数据，并将数据显示在 OLED 上。

最后下载程序到两个开发板中，观察实验现象，大家可以更改发射和接收的时间间隔，来观察对比不同发送间隔对数据传输的影响。

第二十七章 使用自有协议实现 LoRa 组网

27.1 项目要求

网关设备：

1. 每隔 3S 对从机传感器进行依次采集，并将采集结果显示在 OLED 屏幕上。
2. 根据按键键值对从机继电器进行控制，UP 键对应继电器 1，DOWN 键对应继电器 2

子设备：

1. 根据网关操作码类型做相应动作
2. 将采集到传感器结果显示在 OLED 上

注：本章用到核心板+扩展板，对应例程 21。

27.2 原理讲解

27.2.1 组网要求

上一章我们已经实现了使用 SX1278 进行数据传输，但是此时的传输是不安全的，无线通信不同于有线通信，有线通信的信号传输介质是数据线，对于不属于传输线上的设备来说其数据在硬件上是不能被访问的，但是无线传输的传输介质是整个空间，因此 SX1278 所发送的数据会被同处于相同工作参数（相同信道，相同速率）的设备接收，如果存在多个 LoRa 设备，那么数据的发送和接收过程将会变得十分混乱。另外，我们使用 LoRa 的目的就是进行主从组网，因此就更需要为整个 LoRa 网络提供一种“秩序”，让处在网内的设备都要遵循这种秩序并有条不紊的运行。

进行 LoRa 组网要实现两个最基本的功能：

1. 在本网络内，每台从机要有唯一设备标识（从机地址），使得主机能够单独的对某一个从机进行采集和控制；
2. 在不同网络之间，每个网络要有对应的网络标识（网络地址），即使多个网络出现叠加也不会对数据传输造成影响。

首先要实现第一点，就必须将设备地址添加进去，另外还要在地址后面添加和数据处理相关的帧结构，比如主机此时的意图是读还是写，以及读时的要求和写时的数据，最后，为了让传输数据更加安全，还需要加上校验。这样的数据帧结构可以参考下图：

图 27.1　内网数据帧参考模型

接着实现第二个组网要求。要想实现多网之间不发生冲突，有三种常用的解决方案：

第一种：在同信道的情况下，向内网数据帧中添加网络地址（用户 ID），要保证现存网络的网络地址具有唯一性，我们可以将主机的硬件 ID 作为网络地址，在向网络中添加子设备时，需要先和网关进行配对，比如长按网关和子设备的配对按键，使其进入配对模式，网关会自动为子设备分配地址，子设备将分配到的地址记录到 EEPROM 中，退出配对模式或重启后，网关就可以对之前分配过地址的子设备进行访问了。该方案类似于将公司中的所有部门人员都放到一间办公室内，和某个人沟通时要先加上 xx 部的张三或李四。此时数据帧结构可以参考下图：

图 27.2　外网数据帧参考模型 1

这种方案适用于网络数量和网络内子设备不多的情况，否则容易造成数据的阻塞，因为每个网关读取子设备的时间以及子设备上报的时间都是随机的，所以就有可能会出现两个不同网络的子设备同时发射信号，造成阻塞，导致网关读取子设备数据失败。

第二种：依然使用内网数据帧结构，只是要保证不同的 LoRa 网络工作在不同的信道，使得通信之间的数据传输不会受到干扰。该方案类似为公司为每个部门单独分配一间办公室，各个部门内的通信不会受到其他部门干扰。

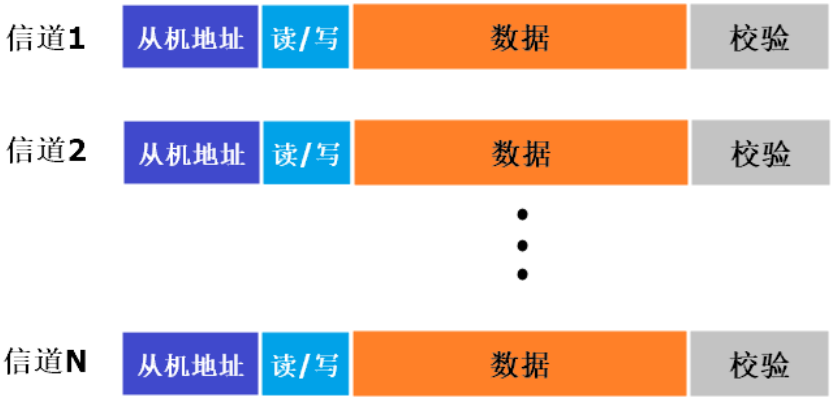


图 27.3　外网数据帧参考模型 2

但是当有一个小几率的事件发生时，会导致该方案失效：即两个用户都将各自的网络设置为相同信道且两个网络传输距离部分相交，子设备地址部分相同时，则会出现控制混乱。因此就衍生除了第三种组网方案，该方案就是将前两种方案合到一起，即：为不同网络分配唯一网络地址，并将每个网络设置为不同的信道。参考模型如下：

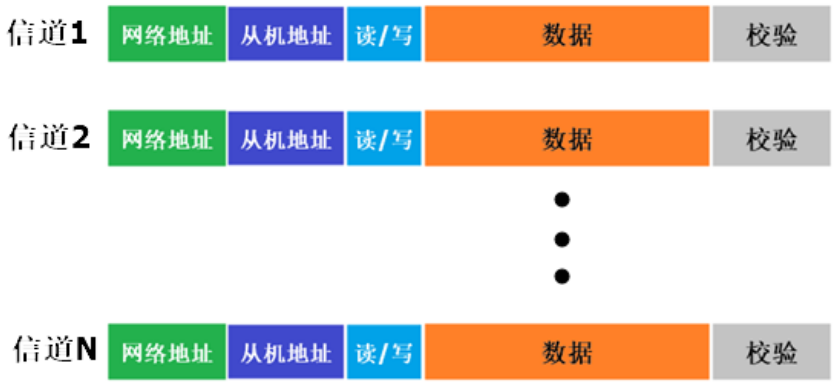


图 27.4　外网数据帧参考模型 3

27.2.2 ModBus 协议

在学习风媒自有组网协议之前先学习一下 ModBus 协议，因为风媒的组网协议就是以 ModBus 为模板进行设计的，大家把它作为知识扩展即可，无需深入。

ModBus 是由 Modicon 公司在 1979 年发明的，是全球第一个真正用于工业现场的总线协议。ModBus 被应用于多从机通信，通信模型如下（以 RS485 为例）：

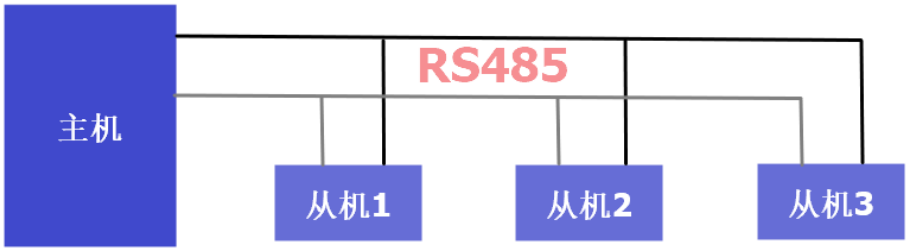


图 27.5　网关结构示意图

其数据帧结构为**从机地址+功能码+数据+校验**，如下图所示：

图 27.6　ModBus 结构图

目前 ModBus 有三个分支，分别是 ModBus RTU，ModBus ASCII，ModBus TCP。我们分别了解一下：

ModBus RTU：ModBus RTU 的数据部分按原值传输，比如从机要传输此时采集到的温度，该温度值为 25 摄氏度，则数据部分对应的二进制为：0001 1001。ModBus RTU 的校验方式使用的是 CRC（循环冗余校验）校验。

ModBus ASCII：ModBus ASCII 传输的是数据对应的字符，我们还以上面为例，假设要传输数值 25，对应的十六进制数值为：0x19,则 ModBus ASCII 会将 0x19 拆成字符‘1’和字符‘9’进行发送，其数据部分对应的二进制为：0011 0010,0011 0101。另外 ModBus ASCII 还增加了开始标记符号“：”和结束标记符号“CR，LF”，其校验方式使用的是 LRC（纵向冗余校验）校验，ModBus ASCII 比较适合高级编程语言调用，比如使用 JAVA 可以将收到的字符通过调用库来转换为其他进制数据并参与计算或者无需任何更改直接显示到应用界面上。但是这种方式的缺点是传输效率没有 ModBus RTU 高。

ModBus TCP：ModBus TCP 是在 TCP/IP 协议基础上使用传输层的 TCP 协议对数据帧进行传输的，ModBus TCP 的特点是省去了从机地址和校验，直接将功能码和数据部分嵌入到 TCP 报文中。如下图：

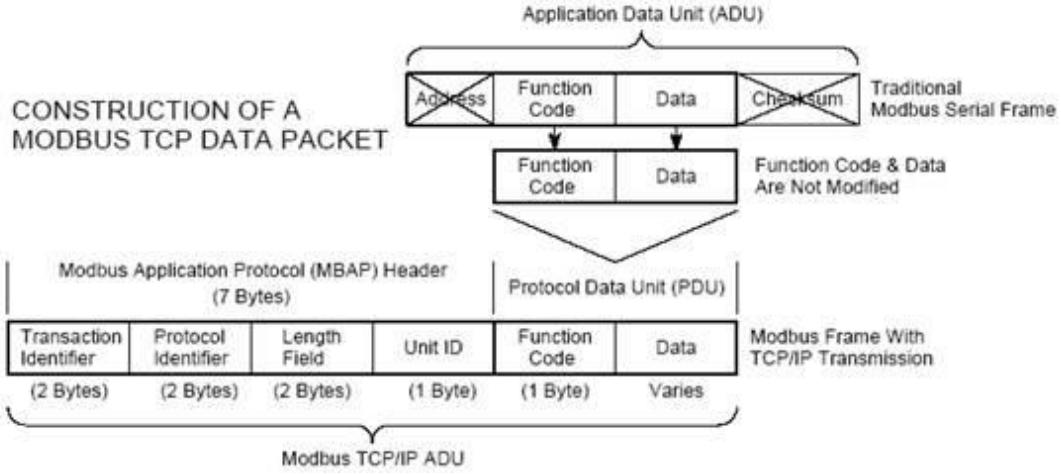


图 27.7　ModBus TCP 报文结构

我们平时使用最多的还是 ModBus RTU，因为它传输效率高，并且更适合嵌入式工程师对其进行位的操作。

现在，我们分别讲解一下 ModBus 中地址，功能码，数据，校验这四个概念。

首先地址就是该设备在本网络中的唯一标识，其范围是 0-255，通常 255 可作为广播地址。广播一般用于批量更改子设备运行参数。

功能码就是要告诉子设备主机此时要进行哪种操作，比如读取线圈状态、设置线圈状态、读写寄存器等，写入传感器状态等。下表是 ModBus RTU 常用功能码：

功能码	功能
0x01	读线圈状态，应用中常用于读取继电器吸合状态
0x02	读输入离散量，常用于读取子设备引脚状态。
0x03	读多个寄存器，常用于读取子设备传感器数据
0x05	写单个线圈，常用于控制子设备单个继电器状态。
0x06	写单个寄存器，常用于设置子设备参数，如波特率，信道频率等。
0x0F	写多个线圈，常用于批量设置子设备的继电器状态
0x10	写多个寄存器，常用于批量设置子设备参数

表 27.1　ModBus RTU 常用功能码

主从之间在通信时，采用一问一答的形式，下面我简单举几个例子，加深理解：

1. 读线圈

主机端：

地址	功能码	地址高字节	地址低字节	线圈数量高字节	线圈数量低字节	CRC 低字节	CRC 高字节
01	01	xx	xx	00	02	xx	xx

表 27.2 ModBus RTU 读线圈指令主机问询

线圈地址并不是固定的，大家可以自己定义，另外线圈数量是指要读取多少个继电器状态（我们这里是两个），CRC 要经过计算才能得到，我们后面会有说明。

从机端：

地址	功能码	字节数量	线圈状态	CRC 低字节	CRC 高字节
01	01	xx	xx	xx	xx

表 27.3 ModBus RTU 读线圈从机响应

字节数量代表后面的线圈状态占用多少字节，比如读取了 16 路继电器，那么字节数量就是 2，线圈状态就是 xxxx，一个 x 代表 4 位二进制。

2. 写单个线圈

主机端：

地址	功能码	地址高字节	地址低字节	线圈状态高字节	线圈状态低字节	CRC 低字节	CRC 高字节
01	05	xx	xx	00	xx	xx	xx

表 27.4 ModBus RTU 写线圈主机问询

将线圈地址对应的线圈设置为吸合或者断开，线圈低字节状态为 FF 该继电器吸合，0x00 该继电器断开。

从机端：

地址	功能码	地址高字节	地址低字节	线圈状态高字节	线圈状态低字节	CRC 低字节	CRC 高字节
01	05	xx	xx	00	xx	xx	xx

表 27.5 ModBus RTU 写线圈从机响应

从机返回和主机端相同的数据。

3. 读单个寄存器

主机端：

地址	功能码	地址高字节	地址低字节	寄存器数量高字节	寄存器数量低字节	CRC 低字节	CRC 高字节
01	03	xx	xx	00	xx	xx	xx

表 27.6　ModBus RTU 读单个寄存器主机问询

16 位地址代表要访问的寄存器地址，寄存器数量一般对应的是传感器的种类，例如本章中寄存器数量为 0x04 代表读取的传感器类型为环境温度（8 位）、湿度（8 位）、光照值（16 位）。

从机端：

地址	功能码	字节数量	寄存器数据	CRC 低字节	CRC 高字节
01	03	xx	xx	xx	xx

表 27.7　ModBus RTU 读单个寄存器从机响应

寄存器数量代表寄存器数据的字节数，如果读取的是 8 位的温度+8 位的湿度+16 位的光照值，则字节数量为 4。

最后讲解一下 ModBus RTU 中的 CRC 循环校验该如何计算。

1. 设定 16 位寄存器内容全为 1，即：CRC = 0xFFFF;

2. 将数据帧中第一个字节数据和 CRC 进行异或，即 CRC ^= byte_1;

3. 判断 CRC 最低位是否为 1

4. 如果为 1，则 CRC 右移一位之后异或 0xA001（固定值）；

5. 如果为 0，CRC 直接右移一位；

***注：一个字节是 8 位，因此要重复步骤 3-5 八次，即右移 8 次**

6. 重复 1-5 进行下一字节处理，直到最后一个字节为止

7. 最后将计算得到的 CRC 值高低字节调换即可

关于 ModBus 就介绍到这里，大家千万不要被 ModBus 的协议规定所迷惑，它只不过是为我们提供一个多机数据通信的模型，在自己的项目中，我们可以定制或者扩充 ModBus 协议来满足项目的要求，比如本章的项目，就要在 ModBus 的基础上进行扩展，我们最终保留的仅仅是数据帧的结构，其他都要根据组网要求做具体更改。

27.2.3 风媒 NodeBus 协议

风媒 NodeBus 协议是一种基于 ModBus 协议扩展而来的高效简单的适合组建多机通信网络的第三方自有组网协议。NodeBus 阉割了 ModBus 的大部分功能码，使用户操作起来更加简单，无需记忆或者查询指令，另外专门针对无线组网的应用场景，在数据头最前面增加了网络地址，使得各个无线网络即时发生重叠也不会导致数据传输出错。

NodeBus 协议数据帧结构如下：

主机端：

图 27.8　NodeBus 协议主机问询数据帧结构

从机端：

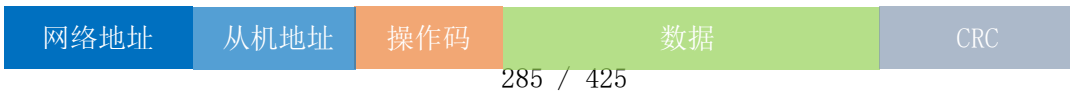


图 27.9　NodeBus 协议从机响应数据帧结构

网络地址

网络地址就是各个网络的唯一标识，使得不同网络之间不会产生干扰（网络地址类似于 TCP/IP 中的公网 IP），我们可以使用主机中单片机的 ID 号作为网络地址也可以自行指定，但是要保证该网络地址不会和其他网络地址发生冲突，网络地址长度不固定，用户可随意指定网络地址长度，本章为了方便起见使用了 8 位来表示网络地址。

从机地址

从机地址是该网络内用于区分各个从机的唯一标识（类似于局域网内的 IP），长度为 8 位，其中 0xFF 用于广播，广播状态下所有的从机都会收到主机发送过来的数据，广播用于批量更改从机参数，比如：波特率，信道频率，通信速率等。

操作码

操作码用于指定该条数据帧是读还是写以及读写对象的类型，我们通过操作码的最高位来判断读/写状态，0 为读 1 为写，剩下的低七位用于设置读写对象，常用操作码如下表：

读/写	读/写对象	功能
（读）0	0x01	读取传感器数据
（读）0	0x03	读取从机系统参数（如运行状态，波特率等）
（写）1	0x02	设置输出设备状态，比如继电器，电平状态等
（写）1	0x04	设置从机系统参数（如运行状态，波特率等）
预留		

表 27.8　NodeBus 操作码

在多数应用中用到最多就是读传感器数据（0x01）和设置输出设备状态（0x02）（采集-控制）这两个操作码。本章实验也只用到了这两个操作码。

参数

参数部分在读和写两种模式下的功能是不同的，在读模式（如：0x01），参数部分的每个位用于指定要读取的是哪个传感器，比如 0x01 对应的是温度传感器的值，0x02 是湿度传感器的值，0x04 是光照传感器的值，0x07 读取的是温湿度和光照强度。如果此时从机由 16 个传感器那么就需要两个字节的数据，即每一位对应一种传感器。在写模式（如：0x02），参数各个位对应的是不同的输出设备，比如参数为 0x01 对应的是控制继电器 1，参数 0x02

对应的继电器 2，参数 0x03 对应的是继电器 1 和继电器 2。

CRC 校验

CRC 校验在 ModBus 的 CRC 的计算方法做了更改。第一个改动是计算的结果无需高低字节调换，第二个改动是 CRC 低字节如果为零就抑或 0xA5 使其强制转换为非零数，这样做的目的是方便计算数据帧的长度，在计算数据帧长度时，我们将收到的数据帧最后一个不为零的数据认为是数据帧结尾，即 CRC 低字节。

上面说的这些主要是针对主机的读和写，对于主机发送过来的读数据，从机在返回时会将传感器或者设备参数数据放在操作码后面，对于主机发送过来的写数据，从机会在返回时将设置完后的输出设备状态数据放在操作码后面。

设备块

我们设计的任何协议最终的服务对象都是数据，对于 NodeBus 也是一样，在主机和从机上都定义个一个相同的设备块，设备块其实就是一块存储设备信息（传感器、继电器、系统参数等）的内存区域，该部分在后面配合程序进行讲解，设备块结构如下图：

图 27.10　NodeBus 协议设备块

27.3 程序讲解

- 计算 CRC 值

```
/**
* 功能：根据ModBus规则计算CRC16
* 参数：
*       _pBuf:待计算数据缓冲区,计算得到的结果存入_pBuf的最后两字节
*       _usLen:待计算数据长度(字节数)
* 返回值：16位校验值
*/
static unsigned short int getModbusCRC16(unsigned char *_pBuf, unsigned short int _usLen)
{
    unsigned short int CRCValue = 0xFFFF;           //初始化CRC变量各位为1
    unsigned char i,j;

    for(i=0;i<_usLen;++i)
    {
        CRCValue  ^= *(_pBuf+i);                    //当前数据异或CRC低字节
        for(j=0;j<8;++j)                            //一个字节重复右移8次
        {
            if((CRCValue & 0x01) == 0x01)           //判断右移前最低位是否为1
            {
                CRCValue = (CRCValue >> 1)^0xA001; //如果为1则右移并异或表达式
            }else
            {
                CRCValue >>= 1;                     //否则直接右移一位
            }
        }

    }

    if(CRCValue&0xFF==0)                            //保证CRC最后一个字节不为零
    {
        CRCValue |= 0xA5;
    }
    return CRCValue;
}
```

图 27.11　计算 CRC 值函数

保证最后一个字节不为零的原因是，我们将收到的数据放在了一个数组当中，为了计算 CRC 值就必须知道该数组中有效数据的个数，如果使用 sizeof 关键字，其计算结果是整个数据的长度，而使用 strlen 计算长度会有一定危险，即如果在数组中遇到有效数据值为 0 的数据会停止计算并返回此时的长度，这种因为错误判断引起的结果比真实值小。所以我们要封装一个计算数组中有效数据个数的函数，计算个数的策略是从数据的最后一个元素往前推，如果碰到非零值（CRC 低字节）就认为他是有效数据的结尾，然后返回长度。

- 计算有效数据的长度

```
/**
* 功能：计算数组中有效数据的长度
* 参数：
*       pbuffer:待计算数据数组
*       buffer_len:该数组的长度
* 返回值：数组buffer中有效数据个数
*/
static unsigned char getFrameLength(unsigned char* pbuffer,unsigned char buffer_len)
{
    unsigned char len = buffer_len-1;    //得到数组最后一个数据的索引

    while(len)                           //向前推算非零数据
    {
        if(*(pbuffer+len)!=0)
        {
            break;
        }
        --len;
    }
    return len+1;                        //有效数据长度
}
```

图 27.12　计算有效数据个数函数

函数说明同上。

- 主机发送指令函数

```
/**
* 功能：发送数据指令
* 参数：
*       slave_addr:从机地址
*       op_code:操作码
*       pram:操作码参数
* 返回值：None
*/
void sendMasterAsk(unsigned char slave_addr,unsigned char op_code,unsigned char pram)
{
    /*发送数据缓冲区，其大小根绝实际情况设置，本例程为6*/
    unsigned char sendbuffer[6] = {NET_ADDR,slave_addr,op_code,pram,0,0};
    unsigned short int CRC16 = getModbusCRC16(sendbuffer,4);              //计算CRC值

    sendbuffer[4] = CRC16>>8;                                             //赋值CRC高位
    sendbuffer[5] = CRC16;                                                //赋值CRC低位

    transmitPackets(sendbuffer,sizeof(sendbuffer));                       //发送数据缓冲区
}
```

图 27.13　主机发送指令函数

主机发送询问指令函数比较简单，就是将一个含有网络地址、从机地址、操作码、操作码参数的数据发送出去。

- 主机接收从机响应函数

```c
typedef struct
{
    /**
     * 对于主机端来说，还可以在最前面添加各个从机的运行状态
     * 如：unsigned char SlaveStatus[256];
     * 每个数组成员都对应一个从机的状态，比如电量不足、通信异常等等
     * 这里没有添加
     * */
    unsigned char Coils;        //线圈状态,Bit0对应继电器1,Bit1对应继电器2
    unsigned char Temeprature;  //环境温度
    unsigned char Humidity;     //环境湿度
    unsigned short int Lux;     //环境光照强度
    /*可继续添加其他传感器、被控单元和系统参数*/
}DeviceBlock;
```

图 27.14　设备块结构体

设备数据块结构体，该结构体包含设备的各种外设信息，我们只对本实验做了封装，如果有更复杂的设备属性直接添加结构体成员即可。

```c
typedef enum
{
    FRAME_OK = 0x00,                //数据帧正确
    FRAME_NETADDR_ERR = 0x01,       //网络地址错误
    FRAME_SLAVEADDR_ERR = 0x02,     //从机地址错误
    FRAME_CRC_ERR = 0x03,           //CRC校验错误
    FRAME_EMPTY = 0xFF              //数据为空，此时没接到数据
}FrameStatus;
```

图 27.15　帧状态结构体

帧状态枚举用于指定此时数据处理的状态。

```c
/**
* 功能：发送数据指令
* 参数：
*       slave_addr:从机地址
*       op_code:操作码
*       pram:操作码参数
*       pdevblock:设备数据块
* 返回值：处理状态
*/
FrameStatus receiveSlaveAck(unsigned char slave_addr,unsigned char op_code,unsigned char pram,DeviceBlock * pdevblock)
{
    /*接收数据缓冲区，其大小根绝实际情况设置，本例程最大值为9*/
    unsigned char receivebuffer[9];
    unsigned char len;
    unsigned char i;

    if(receivePackets(receivebuffer)==1) //收到数据
    {
        if(receivebuffer[0] != NET_ADDR) //不是本网络数据，扔掉
        {
            return FRAME_NETADDR_ERR;
        }

        if(receivebuffer[1] != slave_addr)//不是目标从机发送的数据，扔掉
        {
            return FRAME_SLAVEADDR_ERR;
        }

        len = getFrameLength(receivebuffer,sizeof(receivebuffer));
        if(getModbusCRC16(receivebuffer,len-2) != (receivebuffer[len-2]<<8 | receivebuffer[len-1]))//校验值不对数据无效扔掉
        {
```

```
113             return FRAME_CRC_ERR;
114         }
115 
116         /*一切正常开始处理数据*/
117         if(op_code==OP_W_COILS)           //此时是写线圈(继电器)操作
118         {
119             (pdevblock+slave_addr)->Coils = receivebuffer[3];//
120         }else if(op_code==OP_R_SENSOR)    //此时是读寄存器操作
121         {
122             /**
123              * 该函数传入的应该是一个DeviceBlock类型的数组，每个从机对应一个DeviceBlock类型的数组元素
124              */
125             for(i=0;i<8;++i)
126             {
127 
128                 switch(pram & (0x01<<i))
129                 {
130                     case PRAM_R_TEMPERATURE:(pdevblock+slave_addr)->Temeprature = receivebuffer[3];               break;
131                     case PRAM_R_HUMIDITY   :(pdevblock+slave_addr)->Humidity = receivebuffer[4];                  break;
132                     case PRAM_R_LUX        :(pdevblock+slave_addr)->Lux = receivebuffer[5]<<8 | receivebuffer[6];  break;
133 
134                     default                :                                                                      break;
135                 }
136             }
137         }else
138         {
139             /*其他操作码可在此扩充*/
140         }
141         return FRAME_OK;                  //接收数据成功
142     }else
143     {
144         /**
145          * 没收到从机发送过来的数据，此时的状态可能是处于等待消息阶段，
146          * 当长时间没有收到数据后，可以命令从机重发或者标注该从机出现问题
147          */
148         return FRAME_EMPTY;
149     }
```

图 27.16　主机接收从机响应

该函数进行的工作有两个，第一个是判断地址和校验码是否正确，第二个是根据操作码和操作码参数进行相应动作。

- 子设备处理函数

```c
/**
* 功能：子设备处理函数
* 参数：
*       pdevblock:设备数据块
* 返回值：处理状态
*/
FrameStatus processMasterAsk(DeviceBlock * pdevblock)
{
    unsigned char Askbuffer[6];
    unsigned char Ackbuffer[9] = {NET_ADDR,SLAVE1_ADDR};
    unsigned char len;
    unsigned short int CRC16;
    unsigned char i;

    if(receivePackets(Askbuffer)==1) //收到数据
    {
        if(Askbuffer[0] != NET_ADDR) //不是本网络数据，扔掉
        {
            return FRAME_NETADDR_ERR;
        }

        if(Askbuffer[1] != SLAVE1_ADDR)//不是目标从机发送的数据，扔掉
        {
            return FRAME_SLAVEADDR_ERR;
        }

        len = getFrameLength(Askbuffer,sizeof(Askbuffer));
        if(getModbusCRC16(Askbuffer,len-2) != (Askbuffer[len-2]<<8 | Askbuffer[len-1]))//校验值不对数据无效扔掉
        {
            return FRAME_CRC_ERR;
        }

        if(Askbuffer[2]==OP_W_COILS)
        {
            Ackbuffer[2] = OP_W_COILS;
            pdevblock->Coils = Askbuffer[3];
            Ackbuffer[3] = pdevblock->Coils;
            CRC16 = getModbusCRC16(Ackbuffer,4);
            Ackbuffer[4] = CRC16>>8;
            Ackbuffer[5] = CRC16;
        }else if(Askbuffer[2]==OP_R_SENSOR)
        {
            Ackbuffer[2] = OP_R_SENSOR;
            for(i=0;i<8;++i)
            {
                switch(Askbuffer[3] & (0x01<<i))
                {
                    case PRAM_R_TEMPERATURE:Ackbuffer[3] = pdevblock->Temeprature;break;
                    case PRAM_R_HUMIDITY   :Ackbuffer[4] = pdevblock->Humidity;   break;
                    case PRAM_R_LUX        :{
                                                Ackbuffer[5] = pdevblock->Lux>>8;
                                                Ackbuffer[6] = pdevblock->Lux;
                                             }                                    break;

                    default                :                                      break;
                }
            }

            CRC16 = getModbusCRC16(Ackbuffer,7);
            Ackbuffer[7] = CRC16>>8;
            Ackbuffer[8] = CRC16;
```

```
216         }else
217         {
218             /*其他操作码可在此扩充*/
219         }
220 
221         transmitPackets(Ackbuffer,sizeof(Ackbuffer));
222         return FRAME_OK;
223     }else
224     {
225         return FRAME_EMPTY;
226     }
227 
228     return  FRAME_EMPTY;    //不会执行到这里,添加该语句可以避免警告
229 }
```

图 27.17　子设备处理函数

该函数由子设备调用，用于解析主机发送过来的指令，并作出相应的回应，比如操作码为读取传感器指令则返回的是传感器的数据，如果此时是设置继电器的指令，则此时将设置自身继电器并将继电器状态返回给网关主机。

- 网关设备主函数

```
55  while (1)
56  {
57      if(++i==60) //3S采集一次从机数据
58      {
59          i=0;
60          sendMasterAsk(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL);//主机发送指令
61          receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray);//接收从机数据
62          Delay_ms(100);
63          while(receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray)!=FRAME_OK && ++i<10)//等待从机响应
64          {
65              Delay_ms(100);
66          }
67          i = 0;
68      }
69 
70      keyvalue = getKeyValue(KEY_RELEASE);//获取键值
71 
72      /*根据键值做出相应控制动作*/
73      if(keyvalue==KEY_UP)
74      {
75          j=0;
76          DeviceBlock_StructureArray[SLAVE1_ADDR].Coils ^= 0x01;
77          sendMasterAsk(SLAVE1_ADDR,OP_W_COILS,DeviceBlock_StructureArray[SLAVE1_ADDR].Coils);
78          receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray);
79          Delay_ms(100);
80          while(receiveSlaveAck(SLAVE1_ADDR,OP_W_COILS,0,DeviceBlock_StructureArray)!=FRAME_OK && ++j<30)
81          {
82              Delay_ms(100);
83          }
84      }else if(keyvalue==KEY_DOWN)
85      {
86          j=0;
```

```
87              DeviceBlock_StructureArray[SLAVE1_ADDR].Coils ^= 0x02;
88              sendMasterAsk(SLAVE1_ADDR,OP_W_COILS,DeviceBlock_StructureArray[SLAVE1_ADDR].Coils);
89              receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray);
90              Delay_ms(100);
91              while(receiveSlaveAck(SLAVE1_ADDR,OP_W_COILS,0,DeviceBlock_StructureArray)!=FRAME_OK && ++j<30)
92              {
93                  Delay_ms(100);
94              }
95          }else if(keyvalue==KEY_ALL)
96          {
97              j=0;
98              DeviceBlock_StructureArray[SLAVE1_ADDR].Coils ^= 0x03;
99              sendMasterAsk(SLAVE1_ADDR,OP_W_COILS,DeviceBlock_StructureArray[SLAVE1_ADDR].Coils);
100             receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray);
101             Delay_ms(100);
102             while(receiveSlaveAck(SLAVE1_ADDR,OP_W_COILS,0,DeviceBlock_StructureArray)!=FRAME_OK && ++j<30)
103             {
104                 Delay_ms(100);
105             }
106         }else
107         {
108
109         }
110
111         /*显示传感器数据*/
112         showNumber(40,2,DeviceBlock_StructureArray[SLAVE1_ADDR].Temeprature,DEC,3,FONT_16_EN);
113         showNumber(40,4,DeviceBlock_StructureArray[SLAVE1_ADDR].Humidity,DEC,3,FONT_16_EN);
114         showNumber(40,6,DeviceBlock_StructureArray[SLAVE1_ADDR].Lux,DEC,5,FONT_16_EN);
115
116         toggleLED();
117         Delay_ms(50);
118     }
119 }
```

图 27.18　网关设备主函数

主函数会每隔 3S 左右（大于 3S，因为 OLED 显示需要时间）对从机进行依次采集，并将采集结果显示在 OLED 上，同时我们按下按键可以对从机继电器进行控制，这里需要注意的是由于发送和接收需要时间，所以按键采集可能会存在延时，给人感觉是按键响应有时会比较迟钝。我们可以选择使用前面讲过的外部中断、定时器来采集按键来缓解这种延迟，当然最后还是使用操作系统，这样资源利用更加合理。

- 子设备主函数

```
58      while (1)
59      {
60          if(++j==10)
61          {
62              j = 0;
63              cache = readDHT11();  //获取温湿度
64          }
65
66          DeviceBlock_Structure.Temeprature = cache>>8;
67          DeviceBlock_Structure.Humidity = cache;
68          DeviceBlock_Structure.Lux = getConvValueAve(10,1000);
69
70          processMasterAsk(&DeviceBlock_Structure);//处理指令任务
71
72          /*根据设备块参数设置继电器状态*/
73          if(DeviceBlock_Structure.Coils&0x01)
74          {
75              setRelay(RELAY1,RELAY_CLOSE);
76          }else
77          {
78              setRelay(RELAY1,RELAY_OPEN);
79          }
80
81          if(DeviceBlock_Structure.Coils&0x02)
82          {
83              setRelay(RELAY2,RELAY_CLOSE);
84          }else
85          {
86              setRelay(RELAY2,RELAY_OPEN);
87          }
88
89          /*显示传感器数据*/
90          showNumber(40,2,DeviceBlock_Structure.Temeprature,DEC,3,FONT_16_EN);
91          showNumber(40,4,DeviceBlock_Structure.Humidity,DEC,3,FONT_16_EN);
92          showNumber(40,6,DeviceBlock_Structure.Lux,DEC,4,FONT_16_EN);
93
94          toggleLED();
95          Delay_ms(100);
96      }
97  }
```

图 27.19　子设备主函数

该函数用于处理主机发送过来的数据做处理。

我们对 SX1278 的发射和接收都是通过查询 SX1278 的中断标志位来实现的，为了能够达到更高的实时性，大家也可以使用外部中断来检测 SX1278 模组的 DIO0 引脚的电平状态，当发射、接收完成均会产生高电平。

关于 LoRa 就介绍到这里，在本章之后我们开始讲解 STM32 的第二梯队知识，这些知识安排到了最后是因为这些外设比较轻松有趣也非常实用，能够增加大家学习的积极性。

第二十八章 RTC 实现实时时钟

我看到了我的爱恋
我飞到她的身边
我捧出给她的礼物
那是一小块凝固的时间
时间上有美丽的条纹
摸起来像浅海的泥一样柔软
她把时间涂满全身
然后拉起我飞向存在的边缘
这是灵态的飞行
我们眼中的星星像幽灵
星星眼中的我们也像幽灵

——《三体》

28.1 项目要求

通过 STM32 的 RTC 实现日历和闹钟功能。具体要求如下：

1. 提供年月日时分秒和星期功能
2. 提供闹钟及开关闹钟功能
3. 长按 UP 键进行日期设置模式，此时需要用户键入新日期值
4. 长按 DOWN 键进入闹钟设置模式，此时需要用户键入新闹钟值
5. 同时长按 UP 键和 DOWN 键来开启或者关闭闹钟功能
6. 当闹钟发生时，蜂鸣器以“嘀-嘀----”频率鸣叫 20 次，在此期间可以按任意键退出蜂鸣器闹钟报警
7. 日期、时间、闹钟及闹钟状态显示在 OLED 上

注：本章用到核心板+扩展板，对应例程 22。

28.2 原理讲解

28.2.1 RTC

RTC（Real Time Clock）即实时时钟，常被应用于开发和日历相关的项目，比如市面上的万年历或者电子表都是使用实时时钟来实现的，STM32 也为我们提供了一个 RTC，使得我们不需要使用单独的 RTC 芯片（如 DS1302）就能实现一个日历。

在讲解 RTC 之前先说一下 STM32 的备份存储区（BKP）。我们看到开发板上有一个纽扣电池，它用于给备份存储区供电，使得当单片机断电后备份存储区的数据不会丢失，BKP 的

本质就是一组掉电数据不丢失的寄存器结合，低/中密度的单片机有 20 个寄存器，高密度的有 84 个寄存器。因此我们可以使用 BKP 来替代 EEPROM 存储一些设备的参数（前提是纽扣电池存在并有电），RTC 也位于备份区域，所以即使单片机断电 RTC 也能够正常的计数而不至于时间日期被清零，大家应该有过这种经历，用了好多年的电脑主机或者手机在每次上电开机后其时间都被设置为一个 N 年前的默认值（1970 年 1 月 1 日 0 时），其原因是主板上的纽扣电池没电或者损坏，导致设备一掉电时间就会被清零。

在访问备份存储区时存在一些限制，因为备份存储区在单片机上电或重启后默认是禁止访问数据的，因此需要如下操作，来使能访问 RTC 相关寄存器：

- **设置寄存器 RCC_APB1ENR 的 PWREN 和 BKPEN 位，使能电源和后备接口时钟**
- **设置寄存器 PWR_CR 的 DBP 位，使能对后备寄存器和 RTC 的访问。**

STM32 的 RTC 本质就是一个定时器计数器，只不过作为计算日期时会设置时钟周期为 1S，这样及产生一个时间基数单位，通常给 RTC 提供时钟的晶振选择 32.768KHz，因为 32.768KHz 经过 2^15 分频后周期为 1Hz，即 1S。STM32 的 RTC 时钟源有三个：HSE 的 128 分频，LSE（低速外部时钟），LSI（低速内部时钟），我们常用的是 LSE 作为 RTC 的时钟源，因为 LSE 这种方式获得时钟周期更加精确，不至于长期工作后出现较大的时间积累误差。和其他定时器的基本结构一样，STM32 的 RTC 有一个 32 位的计数器（如果周期为 1S，计数器从零开始计数到溢出，则需要 136 年），一个 32 位时钟预分频器（只用到其中 20 位），一个闹钟寄存器（本质就是输出比较，当时间值等于闹钟寄存器中的值时，相应标志位就会挂起，如果开启中断就会产生闹钟中断）。下图是 RTC 内部结构图：

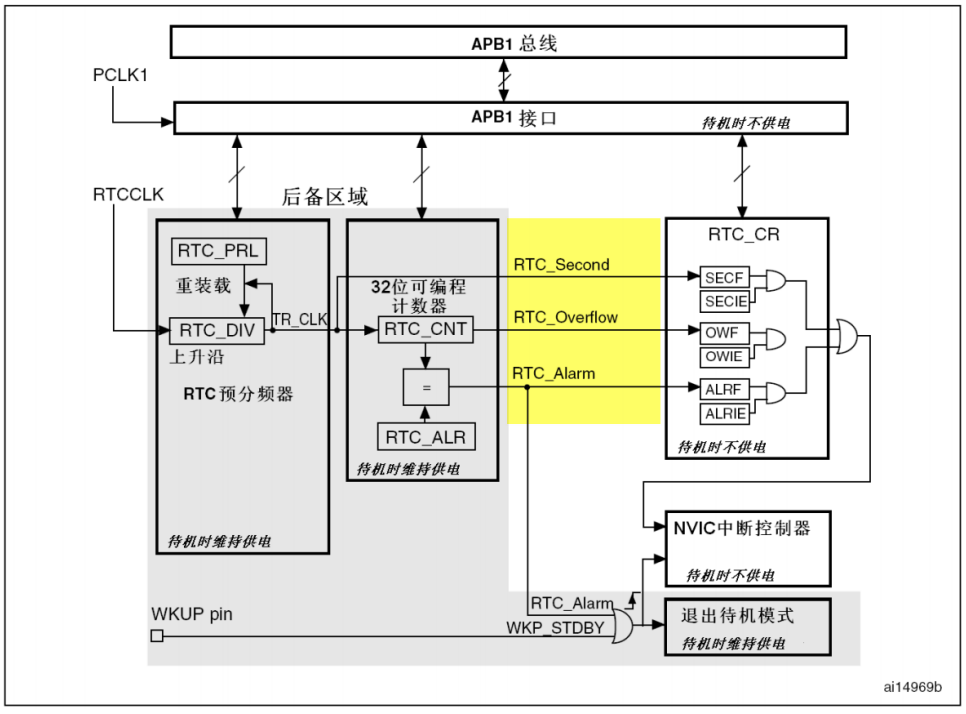


图 28.1　RTC 内部结构框图

图中灰色部分为备份区，我们可以看到除了闹钟中断之外还有秒中断和 RTC 计数器溢出中断，常用的就是闹钟中断和秒中断，前者可以用于执行闹钟动作，如果蜂鸣器滴答动作，

后者可以在每次秒中断产生后将最新的计数器中的值转换为日期并刷新 OLED 上的时间。

前面说到 RTC 位于备份区，想要访问备份区数据需要一些操作来解除访问约束，在解除完访问约束后，我们还不能直接对 RTC 进行读和写，还需要另外一些操作。

读取 RTC：

在读取 RTC 计数器，预分频器和闹钟寄存器之前必须等待 RTC_CRL（RTC 控制寄存器低位域）中的 RSF 被硬件置 1 后，才可对其进行读取。

写入 RTC：

写入 RTC 的 RTC_PRL、RTC_CNT、RTC_ALR 之前先判断 RTC_CRL 寄存器中的 RTOFF 位是否为 1，为 1 则说明上次的操作已经完成，接着还要将 CRL 寄存器中的 CNF 位设置为 1 进入配置模式，配置完成之后将 CNF 设置为 0 生效更改即可。对于写入其他寄存器如 RTC_CRH/L 则无需设置 CNF 位，但是要判断 RTOFF 位。

接下来学习 RTC 相关的寄存器，加深前面对 RTC 操作的理解：

- RTC 控制寄存器高位域（RTC_CRH）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留													OWIE	ALRIE	SECIE
													rw	rw	rw

位15:3	保留，被硬件强制为0。
位2	**OWIE**：允许溢出中断位 (Overflow interrupt enable) 0：屏蔽(不允许)溢出中断 1：允许溢出中断
位1	**ALRIE**：允许闹钟中断 (Alarm interrupt enable) 0：屏蔽(不允许)闹钟中断 1：允许闹钟中断
位0	**SECIE**：允许秒中断 (Second interrupt enable) 0：屏蔽(不允许)秒中断 1：允许秒中断

图 28.2　RTC 控制寄存器高位域

该寄存器用于使能或者失能 RTC 的三个中断，需要注意的是，NVIC 不在备份区域，因此掉电后是不能产生中断的。

- RTC 控制寄存器低位域（RTC_CRL）

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留										RTOFF	CNF	RSF	OWF	ALRF	SECF
										r	rw	rc_w0	rc_w0	rc_w0	rc_w0

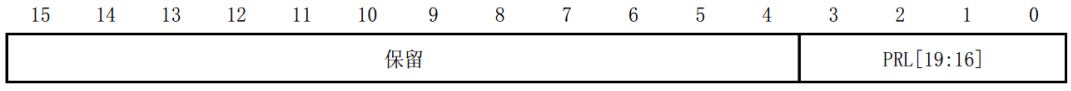
位15:6	保留，被硬件强制为0。
位5	**RTOFF**：RTC操作关闭 (RTC operation OFF) RTC模块利用这位来指示对其寄存器进行的最后一次操作的状态，指示操作是否完成。若此位为'0'，则表示无法对任何的RTC寄存器进行写操作。此位为只读位。 0：上一次对RTC寄存器的写操作仍在进行; 1：上一次对RTC寄存器的写操作已经完成。
位4	**CNF**：配置标志 (Configuration flag) 此位必须由软件置'1'以进入配置模式，从而允许向RTC_CNT、RTC_ALR或RTC_PRL寄存器写入数据。只有当此位在被置'1'并重新由软件清'0'后，才会执行写操作。 0：退出配置模式(开始更新RTC寄存器)； 1：进入配置模式。

位3	**RSF**：寄存器同步标志 (Registers synchronized flag) 每当RTC_CNT寄存器和RTC_DIV寄存器由软件更新或清'0'时，此位由硬件置'1'。在APB1复位后，或APB1时钟停止后，此位必须由软件清'0'。要进行任何的读操作之前，用户程序必须等待这位被硬件置'1'，以确保RTC_CNT、RTC_ALR或RTC_PRL已经被同步。 0：寄存器尚未被同步； 1：寄存器已经被同步。
位2	**OWF**：溢出标志 (Overflow flag) 当32位可编程计数器溢出时，此位由硬件置'1'。如果RTC_CRH寄存器中OWIE=1，则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0：无溢出； 1：32位可编程计数器溢出。
位1	**ALRF**：闹钟标志 (Alarm flag) 当32位可编程计数器达到RTC_ALR寄存器所设置的预定值，此位由硬件置'1'。如果RTC_CRH寄存器中ALRIE=1，则产生中断。此位只能由软件清'0'。对此位写'1'是无效的。 0：无闹钟； 1：有闹钟。
位0	**SECF**：秒标志 (Second flag) 当32位可编程预分频器溢出时，此位由硬件置'1'同时RTC计数器加1。因此，此标志为分辨率可编程的RTC计数器提供一个周期性的信号(通常为1秒)。如果RTC_CRH寄存器中SECIE=1，则产生中断。此位只能由软件清除。对此位写'1'是无效的。 0：秒标志条件不成立； 1：秒标志条件成立。

图 28.3　RTC 控制寄存器低位域

位[5:4]前面说过，主要起到 RTC 配置过程的标志作用；
位[3]前面也说过，用于起到 RTC 读过程的标志作用；
位[3:0]是 RTC 的三个中断标志位，这些标志位只能由硬件置 1 由软件清零。

- RTC 预分频装载寄存器(RTC_PRLH/RTC_PRLL)



位15:6	保留，被硬件强制为0。

位3:0	**PRL[19:16]**：RTC预分频装载值高位 (RTC prescaler reload value high) 根据以下公式，这些位用来定义计数器的时钟频率： $f_{TR\_CLK} = f_{RTCCLK}/(PRL[19:0]+1)$ 注：不推荐使用0值，否则无法正确的产生RTC中断和标志位。

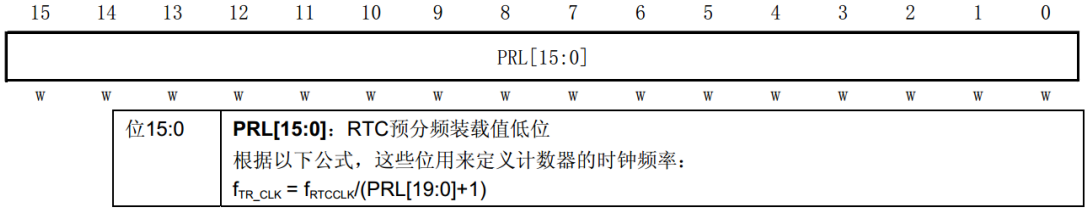


图 28.4　RTC 预分频寄存器

一般将预分频值设置为 32767（0x7FFF）就可产生 1Hz 的时钟周期。

- RTC 计数器寄存器（RTC_CNTH / RTC_CNTL）

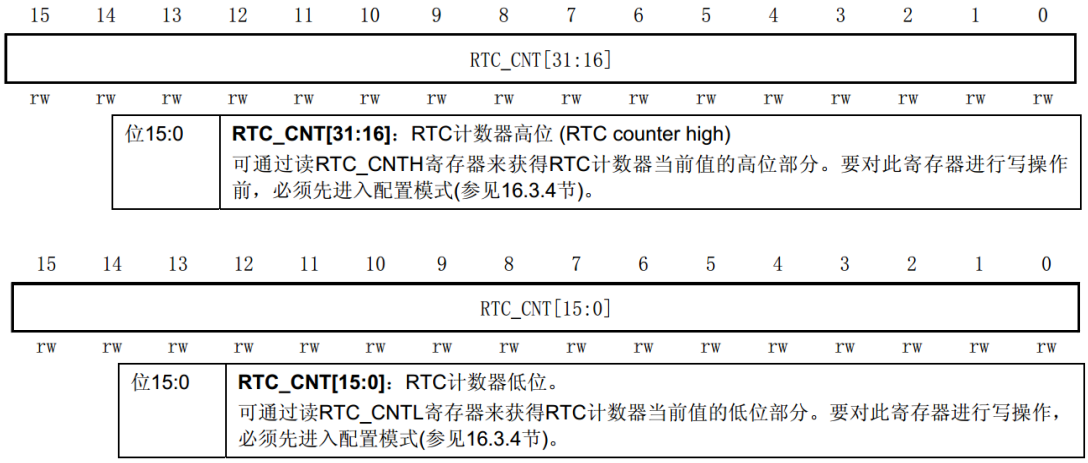


图 28.5　RTC 计数器寄存器

RTC 计数器用于存放 RTC 的计数值我们就是通过该寄存器来获取和设置时间，大家要注意，想要写入该寄存器需要判断 RTCOFF 位并设置 CNF 位。

- RTC 闹钟寄存器(RTC_ALRH/RTC_ALRL)

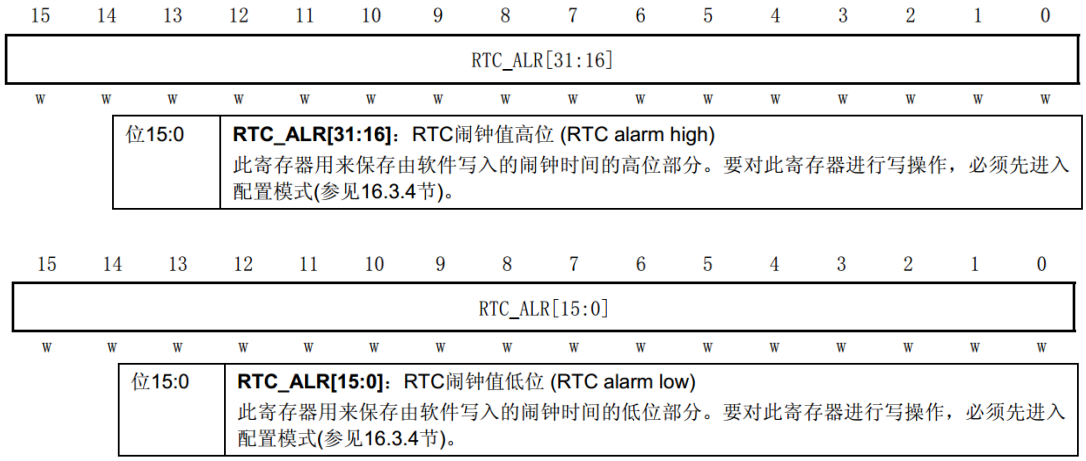


图 28.6　RTC 闹钟寄存器

该寄存器用于设置 RTC 闹钟值，可以理解为 RTC 的输出比较寄存器。同样的，配置该寄存器之前必须判断 RTOFF 位并设置 CNF 位。

- RTC 预分频器余数寄存器(RTC_DIVH / RTC_DIVL)

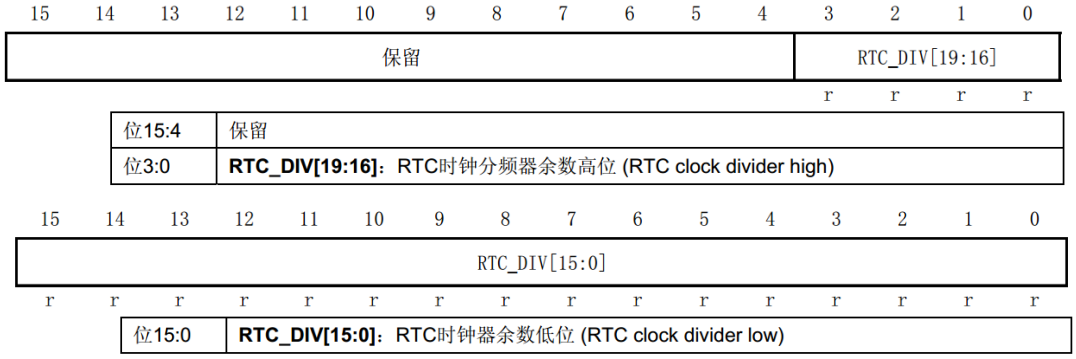


图 28.7　RTC 预分频器余数寄存器

该寄存器用于获取此时预分频寄存器中的值，即可以精确地获取还有多长时间到达 1S。当然平时几乎用不到这个寄存器，所以最后一个介绍它，大家了解即可。

28.2.2 时间

要想通过 RTC 实现电子日历，就需要在 RTC 计数器值和年月日时分秒星期之间通过特定的算法来建立联系，这里涉及比较关键的三个时间算法：

根据 RTC 计数值（秒数）计算年月日时分秒

我们将秒数比喻为以‘秒’为单位的一桶时间液体，现在分别由年、月、日、时、分、秒这 6 个不同容量的容器来盛装这些液体，要求是：盛装顺序为年-月-日-时-分-秒，并规定每种容器要么盛满要么不盛，且到最后桶内不能有剩余时间。

先由年来盛装，它是容量为 365*24*60*60S（暂不考虑闰年），当它发现所剩液体不够装满自身时，通知月来盛装；

月的容量为 30*24*60*60S（暂不考虑各月差异），当它发现所剩液体不够装满自身时，通知日来盛装；

日的容量为 24*60*60S，当它发现所剩液体不够装满自身时，通知时来盛装；

时的容量为 60*60S，当它发现所剩液体不够装满自身时，通知分来盛装；

分的容量为 60S，当它发现所剩液体不够装满自身时，通知秒来盛装；

秒的容量为时间液体的单位容量，它负责打扫桶内残余时间液体，直到桶内干净为止。

以上所述就是我们将 RTC 计数值转换为公历时间单位的过程，后面会结合程序说明。

根据给定年份计算平闰年

平闰年的区别是平年为 365 天而闰年比平年多一天，判断是闰年的方法为：当前年能被 4 整除但是不能被 100 整除或当前年能被 400 整除的都是闰年，这两种情况有一种发生就是闰年。

根据给定日期计算星期

我们根据蔡勒公式来计算当天是星期几，蔡勒公式适用于于公历，其公式如下：

$$w=y+[y/4]+[c/4]-2c+[26(m+1)/10]+d-1$$

其中 w 为星期，如果想要得到 0-6（星期日-星期六），则还需对 7 取余；

c 为世纪减 1，比如 2018 对应的 c 是 20,1990 年对应的 c 是 19；

y 是年份，如 2018 年 y 等于 18

m 是月份，其范围是 3 到 14，1 月和 2 月被安排到去年的 13 月和 14 月，如果月份是 1 月和 2 月那么年份就要减 1，比如 2000 年 1 月，此时 c 为 19，y 为 99，m 为 13。

d 就是几号

28.3 程序讲解

由于日历算法是不受硬件平台的限制的，因此我们将 RTC 功能函数和日历功能函数区分开来设计，这样，大家今后可以轻松将日历函数移植到其他平台。

28.3.1 与平台相关的函数接口

- RTC 初始化函数

```
/**
 * 功能：初始化RTC
 * 参数：None
 * 返回值：是否是第一次使用RTC,非0是第一次,0则不是
 *         程序员可以根据返回值来判断是否需要设置时间
 */
u8 initRTC(void)
{
    RCC_APB1PeriphClockCmd(RCC_APB1Periph_PWR | RCC_APB1Periph_BKP, ENABLE);    //使能PWR和BKP时钟

    PWR_BackupAccessCmd(ENABLE);                                                //BKP访问使能

    if(BKP_ReadBackupRegister(BKP_DR1) != 0x464D)                               //判断是否为第一次使用RTC
    {
        BKP_DeInit();                                                           //设置BKP寄存器回复默认初始值，该行可不加

        RCC_LSEConfig(RCC_LSE_ON);                                              //使能LSE
        while (RCC_GetFlagStatus(RCC_FLAG_LSERDY) == RESET);                    //等待LSE稳定

        RCC_RTCCLKConfig(RCC_RTCCLKSource_LSE);                                 //设置LSE作为RTC时钟源

        RCC_RTCCLKCmd(ENABLE);                                                  //开启RTC时钟

        RTC_WaitForSynchro();                                                   //等待RTC寄存器同步完成(RSF)

        RTC_WaitForLastTask();                                                  //等待上次写操作完成(RTOFF)

        RTC_ITConfig(RTC_IT_SEC, ENABLE);                                       //使能RTC秒中断，用于更新时间和刷新屏幕

        RTC_WaitForLastTask();                                                  //等待上次写操作完成(RTOFF)

        RTC_SetPrescaler(32767);                                                //设置预分频值为1S

        RTC_WaitForLastTask();                                                  //等待上次写操作完成(RTOFF)

        BKP_WriteBackupRegister(BKP_DR1, 0x464D);                               //写上非第一标识

        return 1;
    }else
    {
        RTC_WaitForLastTask();                                                  //等待上次写操作完成(RTOFF)

        RTC_ITConfig(RTC_IT_SEC, ENABLE);                                       //使能RTC秒中断，用于更新时间和刷新屏幕

        RTC_WaitForLastTask();                                                  //等待上次写操作完成(RTOFF)

        return 0;
    }
```

图 28.8　RTC 初始化函数

该函数中增加了一个是否是第一次使用时钟的判断，如果是第一次使用，则返回 1，否则返回 0，通过该函数的返回值我们可以决定是否在第一次使用时进入日期和闹钟配置模式。

在函数中我们先开启了对后备存储区的访问权限，这样才可以对 RTC 进行操作，接着配置 RTC 时钟源为低速外部时钟，在时钟稳定后，等待寄存器同步是否完成，同步完成我们才可以对 RTC 进行读操作，接着设置分频值并使能了秒中断，大家注意在对 RTC 寄存器进行操作一定要使用 RTC_WaitForLastTask()函数来等待上一次 RTC 操作完成后再进行其他读写操作，这里大家会疑惑，前面不是说还要设置 CNF 位进入配置模式吗？是的，ST 官方库为我们提供了这两个函数，如下图：

```
160  void RTC_SetPrescaler(uint32_t PrescalerValue)
161  {
162    /* Check the parameters */
163    assert_param(IS_RTC_PRESCALER(PrescalerValue));
164
165    RTC_EnterConfigMode();
166    /* Set RTC PRESCALER MSB word */
167    RTC->PRLH = (PrescalerValue & PRLH_MSB_MASK) >> 16;
168    /* Set RTC PRESCALER LSB word */
169    RTC->PRLL = (PrescalerValue & RTC_LSB_MASK);
170    RTC_ExitConfigMode();
171  }
```

图 28.9　程序举例 1

这里在重申一遍，只有 RTC 预分频寄存器、RTC 计数器寄存器和 RTC 闹钟寄存器才需要进出配置模式，其他寄存器不需要，但是要判断 RTOFF 位。

另外向备份寄存器 1 中写入的 16 位值是任意的非零值即可，这里的 0x464D 对应的是 FM 的 ASCII 码。

- 获取/设置 RTC 计数器中的值

```
76   /**
77    * 功能：获取RTC计数器中的值，通常用于转换成日期和时间
78    * 参数：None
79    * 返回值：RTC 32位计数器中的值
80    */
81   u32 getRTCConter(void)
82   {
83       return RTC_GetCounter();
84   }
85
86   /**
87    * 功能：设置RTC计数器中的值，通常用于设置日期
88    * 参数：
89    *       conter:待设置值
90    * 返回值：None
91    */
92   void setRTCConter(u32 conter)
93   {
94       RTC_WaitForLastTask();
95       RTC_SetCounter(conter);
96       RTC_WaitForLastTask();
97   }
```

图 28.10　获取/设置 RTC 计数器值函数

我们对官方提供的获取和设置 RTC 计数器的函数做了封装，这样，在移植到其他硬件平台，比如 51 单片机+DS1302，此时只需要将相关获取和设置计数器的代码填充到该函数接口中即可。

- 设置闹钟寄存器中的值

```
/**
 * 功能：设置闹钟寄存器中的值
 * 参数：
 *       conter:待设置值
 * 返回值：None
 */
void setRTCAlarmConter(u32 conter)
{
    RTC_WaitForLastTask();
    RTC_SetAlarm(conter);
    RTC_WaitForLastTask();
}
```

图 28.11　设置 RTC 闹钟寄存器值函数

这里要注意，RTC 闹钟寄存器是只写的，当读取该寄存器时是固定的 0x00FFFFFF，因此为了防止设备重启导致闹钟设置丢失，我们将闹钟时间和闹钟是否使能存储到了备份区数据寄存器。

28.3.2 与平台无关的日历函数

讲解日历函数之前先看一下头文件中定义的日历结构体和其他枚举等，大家在看后面的代码时可以回过头来参考相关枚举。

```c
/*日历年份范围，建议其差值小于135年*/
#define FIRST_YEAR 1990
#define LAST_YEAR  2099

/*平闰年枚举*/
typedef enum
{
    COMMON_YEAR = 0,  //平年
    LEAP_YEAR   = 1   //闰年
}YEARTYPE_ENUM;

/*星期枚举*/
typedef enum
{
    Sun  = 0,   //周日
    Mon  = 1,   //周一
    Tues = 2,   //周二
    Wed  = 3,   //周三
    Thur = 4,   //周四
    Fri  = 5,   //周五
    Sat  = 6    //周六
}WEEK_ENUM;

/*闹钟使能枚举*/
typedef enum
{
    ALARM_OFF = 0x00,   //关闭闹钟功能
    ALARM_ON = 0x01     //开启闹钟功能
}ALARM_ENUM;

/*配置模式枚举*/
typedef enum
{
    NORMAL_MODE = 0x00, //啥也不干
    ALARM_MODE = 0x01,  //闹钟配置模式
    TIME_MODE = 0x02    //日期/时间配置模式
}CONFIGMODE_ENUM;

/*日历结构体*/
typedef struct
{
    u16 Year;           //年
    u8  Month;          //月
    u8  Day;            //日
    u8  Hour;           //时
    u8  Minute;         //分
    u8  Second;         //秒
    u8  Week;           //星期

    u8 AlarmEN;         //闹钟使能
    u8 AlarmHour;       //闹钟小时部分
    u8 AlarmMinute;     //闹钟分钟部分
}Calendar_Structure;
```

图 28.12　日历相关结构体、宏定义、枚举

- 设置模式相关函数

```
/**
* 功能：日历配置模式
* 参数：
*        mode：指定配置闹钟还是配置日期/时间
*        pcalendar：配置好后的日期存入该指针指向的日历结构体变量
* 返回值：None
*/
void configMode(CONFIGMODE_ENUM mode,Calendar_Structure* pcalendar)
{
    if(mode==NORMAL_MODE)       //正常工作模式,通常程序员不会无聊的指定该模式
    {
        return ;
    }else if(mode==TIME_MODE)   //日期/时间设置模式
    {
        printf("\r\n++++++++++++++++++++Time Settings++++++++++++++++++++++++++++++");
        printf("\r\n  Please input new time, e.g.:20180327130123");
        scanUserInput(TIME_MODE,pcalendar);
        setCalendar(pcalendar);
        printf("\r\n  config completed!");
    }else if(mode==ALARM_MODE)  //闹钟设置模式
    {
        printf("\r\n++++++++++++++++++++Alarm Settings++++++++++++++++++++++++++++++");
        printf("\r\n lease input new Alarm, e.g.:1256");
        scanUserInput(ALARM_MODE,pcalendar);
        setAlarm(pcalendar);
        printf("\r\n config completed!");
    }else                       //其他错误模式
    {
        return ;
    }
}
```

图 28.13　配置模式函数

编程时可以通过一些人为事件，如长按按键来触发日历配置函数，该函数分为两部分，第一部分是日期/时间的配置，第二部分是闹钟时间配置。该函数工作流程为进入相关配置模式并打印一些必要说明信息之后通过 scanUserInput()函数来采集用户的串口输入，然后将采集到的数据转换为 RTC 计数器中的值。scanUserInput()函数定义如下：

```
/**
* 功能：读取用户通过串口发送过来的数据并将其转换为时间相关单位
* 参数：
*        mode：需要程序员指定转换的对象类型:日期/时间或者闹钟
*        pcalendar：日历结构体指针
* 返回值：None
*/
static void scanUserInput(CONFIGMODE_ENUM mode,Calendar_Structure* pcalendar)
{
    u8 recbuffer[14];
    u8 i = 0;

    /*清除接收空闲标志位和接收非空标志位*/
    USART_GetFlagStatus(USART1,USART_FLAG_IDLE);
    USART_ReceiveData(USART1);

    if(mode==TIME_MODE)                                                 //设置日期/时间
    {
        while((USART_GetFlagStatus(USART1,USART_FLAG_IDLE)==RESET) && i<14)
        {
            while(USART_GetFlagStatus(USART1,USART_FLAG_RXNE)==RESET);
            recbuffer[i++] = USART_ReceiveData(USART1);
        }

        for(i=0;i<14;++i)                                               //将字符转换为对应数字,如:'1'-'0' = 1
        {
            recbuffer[i] -= '0';
        }
```

```
232         /*赋值给对象结构成员*/
233         pcalendar->Year = recbuffer[0]*1000 + recbuffer[1]*100 + recbuffer[2]*10 + recbuffer[3];
234         pcalendar->Month = recbuffer[4]*10 + recbuffer[5];
235         pcalendar->Day = recbuffer[6]*10 + recbuffer[7];
236         pcalendar->Hour = recbuffer[8]*10 + recbuffer[9];
237         pcalendar->Minute = recbuffer[10]*10 + recbuffer[11];
238         pcalendar->Second = recbuffer[12]*10 + recbuffer[13];
239     }else if(mode==ALARM_MODE)                                          //设置闹钟
240     {
241         while((USART_GetFlagStatus(USART1,USART_FLAG_IDLE)==RESET) && i<4)
242         {
243             while(USART_GetFlagStatus(USART1,USART_FLAG_RXNE)==RESET);
244             recbuffer[i++] = USART_ReceiveData(USART1);
245         }
246
247         for(i=0;i<4;++i)
248         {
249             recbuffer[i] -= '0';
250         }
251
252         /*赋值给对象结构成员*/
253         pcalendar->AlarmHour = recbuffer[0]*10 + recbuffer[1];
254         pcalendar->AlarmMinute = recbuffer[2]*10 + recbuffer[3];
255     }else
256     {
257         return ;
258     }
259 }
```

图 28.14　用户设置采集函数

函数中的 USART_FLAG_IDLE 为串口空闲标志位，即在串口接收完数据后的一段时间内如果没有再接收其他数据则由硬件设置为 1，通常用该标志位来判断一串数据是否全部接收完成。该函数中如果用户没有输入或者数组没有越界则一直等待用户输入。此时收到的数据是数字对应的字符而非数字本身，比如输入 2018 其实是字符‘2’‘0’‘1’‘8’的字符组合，如果想变成数字 2，0，1，8 则需要减去字符‘0’。最后将转换完成的日期/时间/闹钟参数赋值给日历结构体对应成员即可。

- 星期计算函数

```
/**
* 功能：根据给定日期计算星期
* 参数：
*       pcalendar 日历结构体指针
* 返回值：当前星期,星期日-星期六对应为 0-6
* 说明：该算法原理来自于Zeller(蔡勒)公式:w=y+[y/4]+[c/4]-2c+[26(m+1)/10]+d-1
*      w:星期值(最后还需对7取余) c:世纪-1 y:年份 m:月份 d:日期
* 补充:
*      1. []内表示取整数，不存在四舍五入，只要是小数就被舍掉
*      2. m取值为3到14，所以1月和2月按去年的13月和14月算，
*         因此当m为1或2时，对应的世纪和年份也要减1
*      3. 该函数计算的结构除了返回之外，也将星期存入到了日历结构中
*      4. 该函数适合计算1582年10月15日之后的星期值
*/
static WEEK_ENUM getWeek(Calendar_Structure* pcalendar)
{
    u8 week;
    u8 century;
    u8 year;
    u8 month;
    u8 day;

    century = pcalendar->Year/100;  //得到世纪
    year = pcalendar->Year%100;     //得到年份
    if(pcalendar->Month<3)          //得到月份
    {
        month = pcalendar->Month + 12;
        century -= 1;
        if(year==00)
        {
            year = 99;
        }
    }else
    {
        month = pcalendar->Month;
    }
    day = pcalendar->Day;           //得到日期

    /*蔡勒公式计算星期*/
    week = (u8)(year + year/4 + century/4 - 2*century + 2.6*(month+1)+day-1)%7;

    pcalendar->Week = week;

    return week;
}
```

图 28.15　星期计算函数

其原理已经在前面以及程序注释中讲解的很清楚了，这里不做赘述。另外该函数是静态函数，因此只在本文件域内有效，通常被 static 修饰的函数作为最基础的工具函数供本文件内其他函数调用。

- 平闰年计算函数

```
/**
* 功能：根据给定日年份计算年份类型(平年或闰年)
* 参数：
*        year 待计算的年份
* 返回值：平闰年类型
* 说明：能被4整除但是不能被100整除的或者能被400整除的都是闰年
*/
static YEARTYPE_ENUM getYearType(u16 year)
{
    if(((year%4==0) && (year%100!=0)) || (year%400==0))
    {
        return LEAP_YEAR;               //闰年
    }else
    {
        return COMMON_YEAR;             //平年
    }
}
```

图 28.16　平闰年计算函数

- 日历设置/获取函数

```
/**
* 功能：设置日期,根据日历结构中的内容转换成RTC计数器中的值
* 参数：
*        pcalendar: 日历结构体指针
* 返回值：None
*/
void setCalendar(Calendar_Structure* pcalendar)
{
    u32 rtc_conter = 0;
    u16 i;

    /*错误参数检测*/
    if(pcalendar->Year>LAST_YEAR || pcalendar->Year<FIRST_YEAR)
    {
        pcalendar->Year = 1990;
    }
    if(pcalendar->Month>12 || pcalendar->Month<1)
    {
        pcalendar->Month = 1;
    }
    if(pcalendar->Day>31 && pcalendar->Day<1)
    {
        pcalendar->Day = 1;
    }
    if(pcalendar->Hour>23)
    {
        pcalendar->Hour = 23;
    }
    if(pcalendar->Minute>59)
    {
        pcalendar->Minute = 59;
    }
```

```
293     if(pcalendar->Second>59)
294     {
295         pcalendar->Second = 59;
296     }
297
298     /*计算年份对应秒数*/
299     for(i=FIRST_YEAR;i<pcalendar->Year;++i)
300     {
301         if(getYearType(i)==COMMON_YEAR)                    //如果是平年,则一年有365天
302         {
303             rtc_conter += 365*24*60*60;
304         }else                                              //如果是闰年,则一年有366天
305         {
306             rtc_conter += 366*24*60*60;
307         }
308     }
309
310     /*计算月份对应秒数*/
311     for(i=0;i<pcalendar->Month-1;++i)
312     {
313         if(getYearType(pcalendar->Year)==COMMON_YEAR)      //平年月份天数表
314         {
315             rtc_conter += MonthTable[COMMON_YEAR][i]*24*60*60;
316         }else
317         {
318             rtc_conter += MonthTable[LEAP_YEAR][i]*24*60*60;   //闰年月份天数表
319         }
320     }

321
322     /*计算天数对应秒数*/
323     rtc_conter += (pcalendar->Day-1)*24*60*60;
324
325     /*计算小时对应秒数*/
326     rtc_conter += pcalendar->Hour*60*60;
327
328     /*计算分钟对应秒数*/
329     rtc_conter += pcalendar->Minute*60;
330
331     /*计算秒*/
332     rtc_conter += pcalendar->Second+1;
333
334     /*更新星期*/
335     getWeek(pcalendar);
336
337     /*设置生效*/
338     setRTCConter(rtc_conter);
339 }
```

图 28.17　设置日历函数

该函数为日历设置函数，即设置日期/时间函数，该函数是将给定的具体时间计算出 RTC 计数器中的值，类似于前面原理讲解中的反向操作，即年月日时分秒这 6 个容器将其中的时间液体倒入桶中（RTC 计数器寄存器）。

另外 MonthTable 数组是一个二维的 12 元素数组，分别用于存放平闰年各个月份的天数，数组定义如下：

```
16  const u8 MonthTable[2][12] =
17  {
18      {31,28,31,30,31,30,31,31,30,31,30,31}, //平年月份天数
19      {31,29,31,30,31,30,31,31,30,31,30,31}  //闰年月份天数
20  };
```

图 28.18　平闰年月份天数表

接下来的函数是获取时间函数，即根据 RTC 计数器中的值计算出具体的年月日时分秒：

```
/**
* 功能：获取最新时间,根据RTC计数器中的值计算出各个时间参数
* 参数：
*       pcalendar：转换后的数据存储到日历结构体指针指向的结构体
* 返回值：None
* 说明：建议在秒中断中使用,如果用户主动调用,建议最少不低于1/2的采集速度
*/
void getCalendar(Calendar_Structure* pcalendar)
{
    u32 rtc_conter = getRTCConter();
    u16 i;

    /*根据秒数计算年份*/
    for(i=FIRST_YEAR;i<LAST_YEAR;++i)                                   //减到不满一年,剩下的就是月份
    {
        if(getYearType(i)==COMMON_YEAR)
        {
            if(rtc_conter>365*24*60*60)
            {
                rtc_conter -= 365*24*60*60;
            }else
            {
                break;
            }

        }else
        {
            if(rtc_conter>366*24*60*60)
            {
                rtc_conter -= 366*24*60*60;
            }else
            {
                break;
            }
        }
    }
    pcalendar->Year = i;

    /*根据秒数计算月份*/
    for(i=0;i<12;++i)                                                   //减到不满一个月,剩下的就是天数
    {
        if(getYearType(pcalendar->Year)==COMMON_YEAR)
        {
            if(rtc_conter>MonthTable[COMMON_YEAR][i]*24*60*60)
            {
                rtc_conter -= MonthTable[COMMON_YEAR][i]*24*60*60;
            }else
            {
                break;
            }
        }else
        {
            if(rtc_conter>MonthTable[LEAP_YEAR][i]*24*60*60)
            {
                rtc_conter -= MonthTable[LEAP_YEAR][i]*24*60*60;
            }else
            {
                break;
            }
        }
    }
    pcalendar->Month = i+1;
```

```
403
404     /*根据秒数计算天数*/
405     for(i=0;i<31;++i)                                   //减到不满一天,剩下的就是小时
406     {
407         if(rtc_conter>24*60*60)
408         {
409             rtc_conter -= 24*60*60;
410         }else
411         {
412             break;
413         }
414     }
415     pcalendar->Day = i+1;
416
417     /*根据秒数计算小时*/
418     for(i=0;i<24;++i)                                   //减到不满一小时,剩下的就是分钟
419     {
420         if(rtc_conter>60*60)
421         {
422             rtc_conter -= 60*60;
423         }else
424         {
425             break;
426         }
427     }
428     pcalendar->Hour = i;
429
430     /*根据秒数计算分钟*/
431     for(i=0;i<60;++i)                                   //减到不满一分钟,剩下的就是秒数
432     {
433         if(rtc_conter>60)
434         {
435             rtc_conter -= 60;
436         }else
437         {
438             break;
439         }
440     }
441     pcalendar->Minute = i;
442
443     pcalendar->Second = rtc_conter-1;                   //剩余的是秒数
444
445     getWeek(pcalendar);                                 //更新星期
446 }
```

图 28.19　获取日历数据

- 闹钟设置函数

```
/**
* 功能：设置闹钟,将时间参数转换为闹钟寄存器中的值
* 参数：
*       pcalendar：日历结构体指针
* 返回值：None
*/
void setAlarm(Calendar_Structure* pcalendar)
{
    u32 alarm_conter = 0;
    u16 i;

    /*错误参数检测*/
    if(pcalendar->AlarmHour>23)
    {
        pcalendar->AlarmHour = 23;
    }
    if(pcalendar->AlarmMinute>59)
    {
        pcalendar->AlarmMinute = 59;
    }

    /*计算年份对应秒数*/
    for(i=FIRST_YEAR;i<pcalendar->Year;++i)
    {
        if(getYearType(i)==COMMON_YEAR)                         //如果是平年,则一年有365天
        {
            alarm_conter += 365*24*60*60;
        }else                                                   //如果是闰年,则一年有366天
        {
            alarm_conter += 366*24*60*60;
        }
    }

    /*计算月份对应秒数*/
    for(i=0;i<pcalendar->Month-1;++i)
    {
        if(getYearType(pcalendar->Year)==COMMON_YEAR)           //平年月份天数表
        {
            alarm_conter += MonthTable[COMMON_YEAR][i]*24*60*60;
        }else
        {
            alarm_conter += MonthTable[LEAP_YEAR][i]*24*60*60;      //闰年月份天数表
        }
    }

    /*计算天数对应秒数*/
    alarm_conter += (pcalendar->Day-1)*24*60*60;

    /*计算小时对应秒数*/
    alarm_conter += pcalendar->AlarmHour*60*60;

    /*计算分钟对应秒数*/
    alarm_conter += pcalendar->AlarmMinute*60;

    /*计算秒,闹钟只设置到整分*/
    alarm_conter += 1;

    /*设置生效*/
    setRTCAlarmConter(alarm_conter);

    BKP_WriteBackupRegister(BKP_DR2, pcalendar->AlarmHour);

    BKP_WriteBackupRegister(BKP_DR3, pcalendar->AlarmMinute);
}
```

图 28.20　闹钟设置函数

该函数和日历设置函数类似，都是将日期转换成计数值，只不过此时计数值是存放到闹钟寄存器中的。由于闹钟寄存器是只写的，为了在每次断电或者重启后不会导致闹钟数据丢失，我们在函数的最后将得到的闹钟时间保存到备份区数据寄存器。另外还需要注意的是，

本例程中闹钟时间精确到整分。

- 闹钟更新函数

```
448  /**
449  * 功能：更新最新闹钟值,为了每天都可以在相同时刻进行闹钟,因此我们需要在
450  *       新的一天到来时对闹钟寄存器中的值进行更新
451  * 参数：
452  *        pcalendar：日历结构体指针
453  * 返回值：None
454  * 说明：由于闹钟寄存器只能写不能读，因此我们使用了备份区寄存器存储我们的闹钟时间,
455  *       备份区寄存器2,3,4一次存储闹钟小时、闹钟分钟以及闹钟使能
456  */
457  void updateAlarm(Calendar_Structure* pcalendar)
458  {
459      static u8 day = 0;
460
461      if(pcalendar->Day != day)
462      {
463          day = pcalendar->Day;
464
465          pcalendar->AlarmHour = BKP_ReadBackupRegister(BKP_DR2);
466          pcalendar->AlarmMinute = BKP_ReadBackupRegister(BKP_DR3);
467          pcalendar->AlarmEN = BKP_ReadBackupRegister(BKP_DR4);
468
469          setAlarm(pcalendar);
470      }
471  }
```

图 28.21　闹钟更新函数

用户设置闹钟的意图自然是希望可以每天自动闹钟，因此这就需要一种操作：在今天的闹钟发生之后，就要为明天的闹钟做准备，即：将闹钟寄存器中的值增加 24*60*60，使其得到明天同时刻的闹钟值，因此我们的做法是，天数改变后，即时间变为第二天的 0 时 0 分 0 秒时更新最新的闹钟值给闹钟寄存器。

- 时间显示相关函数

```
179  /**
180  * 功能：拆分数字各位(个十百千万...)
181  *       例如：可以将数字1245拆分为1,2,4,5
182  * 参数：
183  *        buffer：拆分后的各位存储数组
184  *        number：待拆分数字
185  *        digits：想要拆分的位数，比如当number为12345时
186  *                该参数应为5，如果小于5会截去高位
187  *        offset：结果偏置,该参数主要用于设置得到的结果是数字还是字符
188  *                如果想要数字，该参数为0,如果想要得到字符该参数为'0'
189  * 返回值：None
190  * 说明：该函数提供了简单直接的求数字各位的功能,唯一需要注意的是：程序员提供的buffer数组大小要大于等于digit
191  *       否则会出现数组越界,导致跳到STM32的硬件错误中断服务函数并使单片机卡死
192  */
193  static void splitNumber(u8* buffer,u32 number,u8 digits,u8 offset)
194  {
195      u8 i = 0;
196
197      while(digits)
198      {
199          buffer[i++] = number%(u32)pow(10,digits)/(u32)pow(10,--digits)+offset;
200      }
201  }
```

图 28.22　数字拆分函数

要实现显示时间就需要得到日期/时间的每一位，比如 20180328230911（2018 年 3 月 28 日 23 时 9 分 11 秒），要将各个位拆分为字符。这里作者为大家编写了一个通用函数，支持任意数字长度，任意数字大小的各位拆分函数。原理如下（以 2018 为例）：

千位 2 = 2018%10000/1000----------2018%10^4/10^3
百位 0 = 2018%1000/100----------2018%10^3/10^2
十位 1 = 2018%100/10----------2018%10^2/10^1
个位 8 = 2018%10/1----------2018%10^1/10^0
根据上面的规律可以推算出其通用的计算公式：

```
X = number%10^digit/10^(digit-1);
```

X 为待计算的位结果，number 为待拆分数字，digit 为要拆分成几位，比如 number 为 2018 对应的 digit 就应该是 4，如果小于 4，比如 3，则结果为 0,1,8。

另外函数中的的 offset 偏置用于设置得到的位结果是数字还是字符，其原理前面和注释中已经说过，这里就不赘述。

```
549  /**
550  * 功能：显示日历各个时间参数
551  * 参数：
552  *        pcalendar：日历结构体指针
553  * 返回值：None
554  */
555  void showCalendar(Calendar_Structure* pcalendar)
556  {
557      u8 i;
558
559      /*日期各位拆分后的结果存放数组,一位对应一个数组*/
560      u8 year[4];
561      u8 month[2];
562      u8 day[2];
563      u8 hour[2];
564      u8 minute[2];
565      u8 second[2];
566
567
568      u8 alarm_hour[2];
569      u8 alarm_minute[2];
570
571      /*星期对应字符串*/
572      u8* weekstr[7] = {"Sun ","Mon ","Tues","Wed ","Thur","Fri ","Sat "};
573
574      /*数据拆分开始,拆分结果为字符,方便OLED显示*/
575      splitNumber(year,pcalendar->Year,sizeof(year),'0');
576      splitNumber(month,pcalendar->Month,sizeof(month),'0');
577      splitNumber(day,pcalendar->Day,sizeof(day),'0');
578      splitNumber(hour,pcalendar->Hour,sizeof(hour),'0');
579      splitNumber(minute,pcalendar->Minute,sizeof(minute),'0');
```

```
    splitNumber(second,pcalendar->Second,sizeof(second),'0');

    splitNumber(alarm_hour,pcalendar->AlarmHour,sizeof(alarm_hour),'0');
    splitNumber(alarm_minute,pcalendar->AlarmMinute,sizeof(alarm_minute),'0');

    /*第一行显示 年/月/日 e.g.:2018/03/27*/
    for(i=0;i<sizeof(year);++i)
    {
        showChar(i*FONT_16_EN,0,year[i],FONT_16_EN);
    }
    showChar(i*FONT_16_EN,0,'/',FONT_16_EN);
    for(i=0;i<sizeof(month);++i)
    {
        showChar(i*FONT_16_EN+40,0,month[i],FONT_16_EN);
    }
    showChar(i*FONT_16_EN+40,0,'/',FONT_16_EN);
    for(i=0;i<sizeof(day);++i)
    {
        showChar(i*FONT_16_EN+64,0,day[i],FONT_16_EN);
    }

    /*第二行显示 时:分:秒 e.g.:23:05:23*/
    for(i=0;i<sizeof(hour);++i)
    {
        showChar(i*FONT_16_EN,2,hour[i],FONT_16_EN);
    }
    showChar(i*FONT_16_EN,2,':',FONT_16_EN);
    for(i=0;i<sizeof(minute);++i)
    {
        showChar(i*FONT_16_EN+24,2,minute[i],FONT_16_EN);
    }
    showChar(i*FONT_16_EN+24,2,':',FONT_16_EN);
    for(i=0;i<sizeof(second);++i)
    {
        showChar(i*FONT_16_EN+48,2,second[i],FONT_16_EN);
    }

    /*第三行显示星期 e.g.:Tues*/
    showString(0,4,weekstr[pcalendar->Week],FONT_16_EN);

    /*第四行显示闹钟时间以及是否开启 e.g.:08:30   on*/
    for(i=0;i<sizeof(alarm_hour);++i)
    {
        showChar(i*FONT_16_EN,6,alarm_hour[i],FONT_16_EN);
    }
    showChar(i*FONT_16_EN,6,':',FONT_16_EN);
    for(i=0;i<sizeof(alarm_minute);++i)
    {
        showChar(i*FONT_16_EN+24,6,alarm_minute[i],FONT_16_EN);
    }
    if(pcalendar->AlarmEN==ALARM_ON)
    {
        showString(60,6,"on ",FONT_16_EN);
    }else
    {
        showString(60,6,"off",FONT_16_EN);
    }
}
```

图 28.23　显示日历函数

整个流程就是先拆分，后显示。

- RTC 中断服务函数

```
672  /**
673   * 功能：RTC中断服务函数,一般用于每秒更新一次时间以及显示数据
674   * 参数：None
675   * 返回值：None
676   */
677  void RTC_IRQHandler(void)
678  {
679      static u8 day = 0;
680      if (RTC_GetITStatus(RTC_IT_SEC) == SET)
681      {
682          getCalendar(&calendar);          //更新时间
683          updateAlarm(&calendar);
684          showCalendar(&calendar);         //显示最新时间
685      }
686
687      RTC_WaitForLastTask();
688      RTC_EnterConfigMode();
689      RTC_ClearITPendingBit(RTC_IT_SEC);  //软件清除秒中断标志位
690      RTC_ExitConfigMode();
691      RTC_WaitForLastTask();
692  }
```

图 28.24　RTC 中断服务函数

该函数功能用于实时获取最新时间数据并显示，另外也判断闹钟是否需要更新，即：第二天刚开始时设置。

- 主函数业务逻辑循环

```
52      if(initRTC()==1)
53      {
54          configMode(TIME_MODE,&calendar);
55          configMode(ALARM_MODE,&calendar);
56      }
```

```
56      while (1)
57      {
58          /*长按UP键两秒进入时间设置模式*/
59          if(getKeyValue(KEY_PRESS)==KEY_UP)
60          {
61              if(++time_cnt==30)
62              {
63                  time_cnt = 0;
64                  configMode(TIME_MODE,&calendar);
65              }
66          }else
67          {
68              time_cnt = 0;
69          }
70
71          /*长按DOWN键两面进入闹钟设置模式*/
72          if(getKeyValue(KEY_PRESS)==KEY_DOWN)
73          {
74              if(++alarm_cnt==30)
75              {
76                  alarm_cnt = 0;
77                  configMode(ALARM_MODE,&calendar);
78              }
79          }else
80          {
```

```
81              alarm_cnt = 0;
82          }
83
84          /*同时长按UP和DOWN键两秒关闭或开启闹钟*/
85          if(getKeyValue(KEY_PRESS)==KEY_ALL)
86          {
87              if(++alarm_en_cnt==30)
88              {
89                  alarm_en_cnt = 0;
90                  calendar->AlarmEN ^= 1;
91                  enableAlarm(calendar->AlarmEN,&calendar);
92              }
93          }else
94          {
95              alarm_en_cnt = 0;
96          }
97
98          /*闹钟提醒,按任意键停止闹钟*/
99          if(RTC_GetFlagStatus(RTC_FLAG_ALR)==SET)
100         {
101             alarm_en_cnt = 0;
102             /*人为关掉闹钟提醒后者提醒超过1分钟后退出*/
103             while(!getKeyValue(KEY_PRESS) && ++alarm_en_cnt<20)
104             {
105                 setBuzzerVol(MAXVOL);
106                 Delay_ms(100);
107                 setBuzzerVol(MUTE);
108                 Delay_ms(100);
109                 setBuzzerVol(MAXVOL);
110                 Delay_ms(100);
111                 Delay_ms(100);
112                 setBuzzerVol(MUTE);
113                 Delay_ms(1000);
114             }
115
116             alarm_en_cnt = 0;
117             setBuzzerVol(MUTE);
118             RTC_WaitForLastTask();
119             RTC_EnterConfigMode();
120             RTC_ClearITPendingBit(RTC_FLAG_ALR);  //软件清除秒中断标志位
121             RTC_ExitConfigMode();
122             RTC_WaitForLastTask();
123             updataAlarm(&calendar);
124         }
125
126         toggleLED();
127         Delay_ms(50);
128     }
129 }
```

图 28.25　主函数用户逻辑主循环

主循环上面的 RTC 初始化部分程序用于执行 RTC 是否是第一次使用的触发动作。如果是第一次使用，则自动依次进入日期/时间和闹钟配置模式。

主循环包含两大部分：第一部分是查询各种触发动作，第二个是检查闹钟是否到来。如果想要开启或关闭闹钟则同时按下 UP 键和 DOWN 超过两秒即可，此时 OLED 上会显示“on”或“off”，如果长按 UP 键或长按 DOWN 键则会进入时间设置后者闹钟设置，此时我们需要提前打开串口调试工具，然后进行设置，如下图：

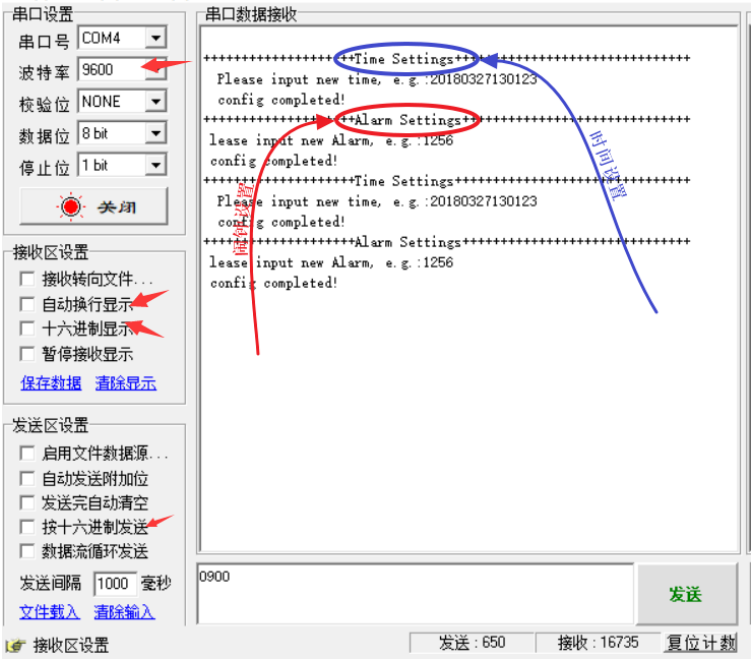


图 28.26　配置模式示例及其软件配置

按照图中配置即可，端口号根据实际情况选择。下图是运行结果：

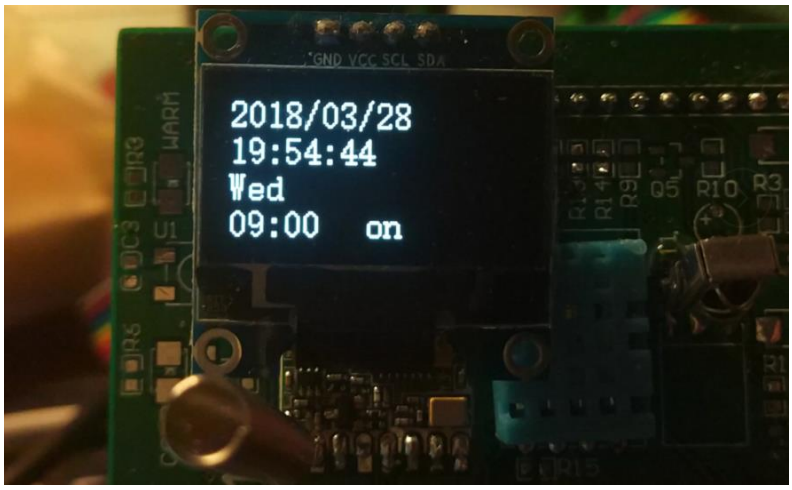


图 28.27　运行效果图

本章到此结束，虽然本章函数很多，但大多数都是根据时间换算算法衍生出来的，建议大家结合开发板亲自实践一下，加深理解。

第二十九章 使用 USB 制作键盘

29.1 项目要求

本章例程由两个项目组成：

第一个子项目：使用 STM32 的 USB 外设实现键盘功能，并通过来开发板上的按键来控制 Windows 经典游戏——3D 弹球。

第二个子项目：使用开发板上的 UP 键来实现计算机的一键关机。

注：本章用到核心板，对应例程 23、24，另需将核心板的跳线帽由 UART 改接到 USB

29.2 原理讲解

29.2.1 认识 USB

USB（Universal Serial Bus），即：通用串行总线，这应该算得上是我们最熟悉的计算机基本外设了，我们可以通过 USB 接口连接鼠标、键盘、大容量存储设备、甚至的声卡网卡等，USB 的特点是只需要很少的数据线（2 根,D+和 D-）就能够进行高速的数据传输，USB 发展至今经历了很多版本，每次版本的迭代其通信速度都会有质的提升，如下图：

USB版本	理论最大传输速率	速率称号	最大输出电流	推出时间
USB1.0	1.5Mbps(192KB/s)	低速(Low-Speed)	5V/500mA	1996年1月
USB1.1	12Mbps(1.5MB/s)	全速(Full-Speed)	5V/500mA	1998年9月
USB2.0	480Mbps(60MB/s)	高速(High-Speed)	5V/500mA	2000年4月
USB3.0	5Gbps(500MB/s)	超高速(Super-Speed)	5V/900mA	2008年11月 / 2013年12月
USB 3.1Gen 2	10Gbps(1280MB/s) [1]	超高速+(Super-speed+)	20V/5A	2013年12月

图 29.1 USB 版本汇总

虽然 USB3.0 超高速能达到 500MB/S，但是要受其硬件本身的限制，比如机械硬盘的读取速度，外接 USB 网卡的速度等。可能会有人产生疑惑，同样的是串行通信，为何 USB 可以达到这个高的速度，而同样是串行通信的 UART,IIC 等却只有几十 K 的传输速度？我们要知道如果一种传输协议想要达到很高的数据吞吐就意味着会有更高的几率导致传输数据出错，因此 USB 采用的差分数据的传输方式，这样能有效降低数据出错的几率，同时 USB 在数据传输上采用的 NRZI 编码方式，提高了 USB 的传输速度。

另外同样是串行通信协议，但是 USB 的功能却异常丰富，其原因是 USB 采用了完善健壮的通信协议，这里所说的通信协议代表是基于硬件之上的通信逻辑。比如主从之间的应答机制，数据封包的方式等，当然这也导致 USB 的通信协议变得异常复杂，所以本章并不打算延续前面章节的教学方式来介绍 STM32 的 USB 外设，大家可以阅读相关的参考资料或者书籍，建议大家阅读《圈圈教你玩 USB》，这本书讲解的很通俗易懂，但不是使用的 STM32 来实现的，大家可以阅读前面关于协议的章节（第一章和第三章）来快速了解 USB 通信协议。

接下来向大家介绍一些 USB 协议相关的知识，目的是为了大家能在开发 STM32 USB 项目时不会感到迷惑。

USB 是主从通信结构，主机（HOST）一般指的就是我们的电脑，从机（设备）是我们的键鼠或者 U 盘等，STM32 的 USB 只能作为从机来用，这点要注意。

STM32 的 USB 支持 USB2.0 协议，USB2.0 协议还分为低速、高速、全速模式，STM32 支持高速模式，当设备插入到 USB 主机中时主机是通过 D+和 D-数据线上电平状态来判断插入的设备属于低速还是高速设备的。在没有插入设备之前主机上的 D+和 D-差分数据线都通过 15K 的电阻下拉到地表示空闲状态，低速或高速是由设备（从机）来决定的，如果在设备的 D+数据线上上拉一个 1.5K 的电阻就表示该设备为高速模式，如果在 D-数据线上上拉一个 1.5K 电阻则表示是低速设备。其原理就是设备插入后，上拉的那条数据线由于电阻分压会导致主机读取该条数据线电平状态为高电平。另外根据这种机制，我们可以通过一个三极管来选通上拉电阻是否接入数据线，这样可以实现无需设备插拔就能自动断开连接的功能。

另外需要讲解的就是 USB 的描述符概念，描述符贯穿了我们整个 USB 的开发过程，如果描述符移植完成，那么整个项目也完成了大半。USB 描述符顾名思义就是告诉主机我这个设备是干什么的以及通信的方式是什么样的（设备的自我介绍）。这些描述符有：设备描述符、配置描述符、接口描述符、端点描述符。他们之间是互相嵌套的，即多个端点组成一个接口，多个接口组成一个配置，多个配置组成一个设备。反过来说，一个设备可以存在多个配置，每个配置可以定义多个接口，每个接口又包含多个端点，主机和设备之间的通信就是和端点的通信。

设备描述符用于描述所使用的 USB 协议版本号、设备类型、端点 0 的最大数据包大小，厂商 ID（VID），产品 ID（PID），设备版本号、厂商字符串索引、产品字符串索引、设备序列号索引，配置的数量等。对于厂商的 ID 和产品 ID 都是固定的，这样做的好处是一个设备插入之后，电脑可以根绝某个厂商的某个产品来安装对应的驱动。

配置描述符描述的信息有：该配置包含的接口数量，配置的编号、供电方式（总线供电还是自供电）、是否支持远程唤醒、电流需求量大小等。

接口描述符记录的内容包括：接口的编号、接口的端点数、接口所使用的类、子类、协议等。

端点描述符记录的信息有：端点号及方向、端点的传输类型、最大包长度、查询时间间隔等。

最后是 USB 的传输类型，一共有四种：批量传输、等时传输、中断传输和控制传输，我

们实现 HID 键盘用到的是中断传输，USB 的中断传输并不是指单片机中断传输，而是主机（计算机）每隔一段时间对设备进行一次采集，这个时间由描述符来决定，比如我们这里使用的是 32ms。

29.2.2 STM32 USB 讲解

STM32 的 USB 特点如下（了解即可）：

- 符合 USB2.0 全速设备的技术规范
- 可配置 1 到 8 个 USB 端点
- CRC(循环冗余校验)生成/校验，反向不归零(NRZI)编码/解码和位填充
- 支持同步传输
- 支持批量/同步端点的双缓冲区机制
- 支持 USB 挂起/恢复操作
- 帧锁定时钟脉冲生成

下图为 USB 内部结构框图：

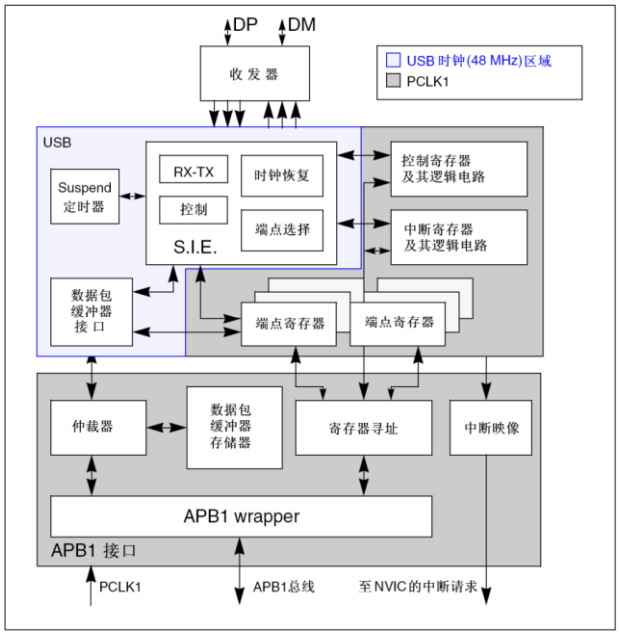


图 29.2　USB 内部结构框图

在前面的章节，用到的都是 STM32 标准库提供的库文件，比如：stm32f10x_gpio.h/.c 等，但是标准库却没有提供 USB 相关的库文件，原因是 USB 协议复杂，如果将驱动函数全部都封装到一个文件中，将会大大降低 USB 的可移植性和维护性，因此 ST 官方为我们开发者单独封装了一个 USB 库，作者已经为大家下载了这个 USB 库，我们今后所有的移植工作都是围绕这个库来展开的。让我们先看一下这个库长什么样子：

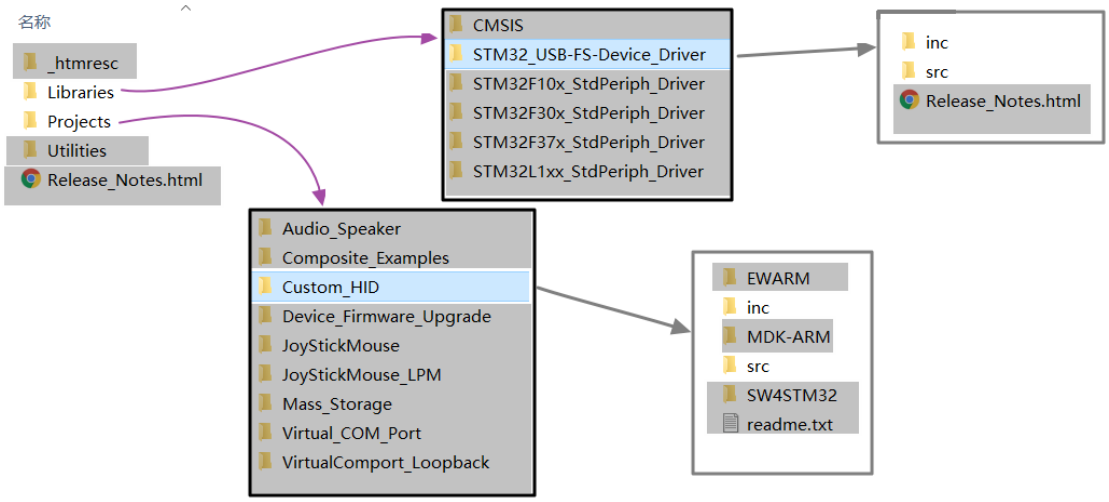


图 29.3　官方 USB 库目录结构

图中从左到右依次是嵌套关系，灰色部分与本项目无关，可以删掉的部分，这样，到最后我们需要的就是 **STM32_USB-FS-Device_Driver** 文件夹中的文件和 **Custom_HID** 文件夹中的文件，前者是 USB 的底层驱动库，后者是用于我们用于根据具体需求来进行定制的源文件。这两者的依存关系如下图（图片来自 USB 库使用手册）：

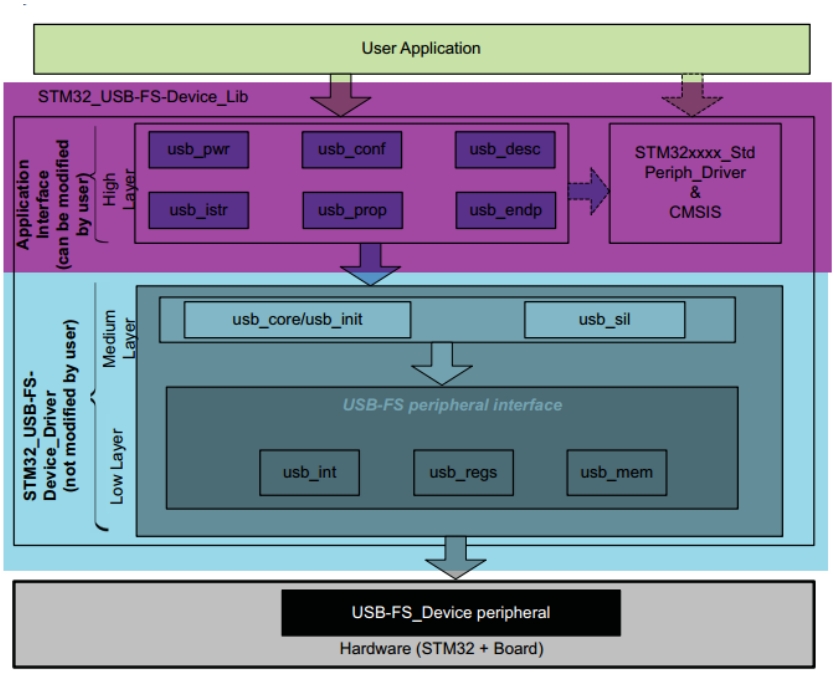


图 29.4　USB 库文件依赖关系

图中青色高亮部分对应 **STM32_USB-FS-Device_Driver**，紫色高亮部分对应 **Custom_HID**，因为我们本章用到的是 HID(Human Interface Device)键盘功能，所以包含的是 Custom_HID，其他项目使用 Projects 文件夹内的其他工程即可。ST 官方将这些文件分开管理恰恰体现了程序中的分层管理思想，这样做使得文件的耦合度，函数之间的耦合度都大大降低，程序的可移植性更好。现在我们再对这两个文件夹开刀，看看里面各个文件都是干什么的：

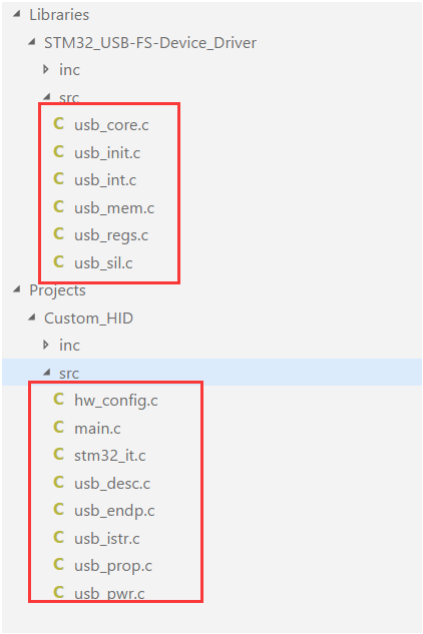


图 29.5　USB 库文件

这些库文件并不是全部都要移植修改，我们只对 usb_desc.c、usb_endp.c、usb_prop.c 文件开刀，其他无需关心。大家可以参考我们为大家准备好的 ST 的 USB 库用户手册进一步了解。

先看一下官方给出的移植流程：

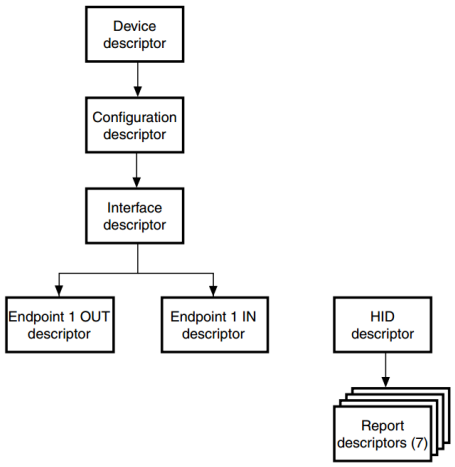


图 29.6　移植流程

可以看到，主要的工作任务就是对前面说过的各种描述符进行修改，另外在 HID 应用中使用的是端点 1 进行数据的收发，而端点 0 是主机和设备初次建立连接时使用的。下面联系程序来讲解一下移植的具体流程。

29.3程序讲解

修改 usb_desc.c

- 设备描述符

```
52  const uint8_t CustomHID_DeviceDescriptor[CUSTOMHID_SIZ_DEVICE_DESC] =
53    {
54      0x12,                       /*bLength */
55      USB_DEVICE_DESCRIPTOR_TYPE, /*bDescriptorType*/
56      0x00,                       /*bcdUSB */
57      0x02,
58      0x00,                       /*bDeviceClass*/
59      0x00,                       /*bDeviceSubClass*/
60      0x00,                       /*bDeviceProtocol*/
61      0x40,                       /*bMaxPacketSize40*/
62      0x83,                       /*idVendor (0x0483)*/
63      0x04,
64      0x50,                       /*idProduct = 0x5750*/
65      0x57,
66      0x00,                       /*bcdDevice rel. 2.00*/
67      0x02,
68      1,                          /*Index of string descriptor describing
69                                                  manufacturer */
70      2,                          /*Index of string descriptor describing
71                                                  product*/
72      3,                          /*Index of string descriptor describing the
73                                                  device serial number */
74      0x01                        /*bNumConfigurations*/
75    }
76    ; /* CustomHID_DeviceDescriptor */
```

图 29.7　设备描述符移植

设备描述符无需修改，保持默认即可，这里主要指定使用 USB 的版本，厂商和设备 ID。

- 配置描述符

```
81  const uint8_t CustomHID_ConfigDescriptor[CUSTOMHID_SIZ_CONFIG_DESC] =
82    {
83      0x09, /* bLength: Configuration Descriptor size */
84      USB_CONFIGURATION_DESCRIPTOR_TYPE, /* bDescriptorType: Configuration */
85      CUSTOMHID_SIZ_CONFIG_DESC,
86      /* wTotalLength: Bytes returned */
87      0x00,
88      0x01,         /* bNumInterfaces: 1 interface */
89      0x01,         /* bConfigurationValue: Configuration value */
90      0x00,         /* iConfiguration: Index of string descriptor describing
91                                                 the configuration*/
92      0xC0,         /* bmAttributes: Self powered */
93      0x32,         /* MaxPower 100 mA: this current is used for detecting Vbus */
94
95
96      /************** Descriptor of Custom HID interface ****************/
97      /* 09 */
98      0x09,         /* bLength: Interface Descriptor size */
99      USB_INTERFACE_DESCRIPTOR_TYPE,/* bDescriptorType: Interface descriptor type */
100     0x00,         /* bInterfaceNumber: Number of Interface */
101     0x00,         /* bAlternateSetting: Alternate setting */
102     0x02,         /* bNumEndpoints */
103     0x03,         /* bInterfaceClass: HID */
104     0x00,         /* bInterfaceSubClass : 1=BOOT, 0=no boot */
105     0x01,         /* nInterfaceProtocol : 0=none, 1=keyboard, 2=mouse */
106     0,            /* iInterface: Index of string descriptor */
107     /******************** Descriptor of Custom HID HID ********************/
108     /* 18 */
109     0x09,         /* bLength: HID Descriptor size */
110     HID_DESCRIPTOR_TYPE, /* bDescriptorType: HID */
111     0x10,        /* bcdHID: HID Class Spec release number */
112     0x01,
113     0x00,        /* bCountryCode: Hardware target country */
114     0x01,        /* bNumDescriptors: Number of HID class descriptors to follow */
115     0x22,        /* bDescriptorType */
116     CUSTOMHID_SIZ_REPORT_DESC,/* wItemLength: Total length of Report descriptor */
117     0x00,
118     /******************** Descriptor of Custom HID endpoints ******************/
119     /* 27 */
120     0x07,         /* bLength: Endpoint Descriptor size */
121     USB_ENDPOINT_DESCRIPTOR_TYPE, /* bDescriptorType: */
122
123     0x81,         /* bEndpointAddress: Endpoint Address (IN) */
124     0x03,         /* bmAttributes: Interrupt endpoint */
125     0x08,         /* wMaxPacketSize: 8 Bytes max */
126     0x00,
127     0x20,         /* bInterval: Polling Interval (32 ms) */
128     /* 34 */
129
130     0x07, /* bLength: Endpoint Descriptor size */
131     USB_ENDPOINT_DESCRIPTOR_TYPE, /* bDescriptorType: */
132       /*  Endpoint descriptor type */
133     0x01, /* bEndpointAddress: */
134       /*  Endpoint Address (OUT) */
135     0x03, /* bmAttributes: Interrupt endpoint */
136     0x08, /* wMaxPacketSize: 8 Bytes max  */
137     0x00,
138     0x20, /* bInterval: Polling Interval (32 ms) */
139     /* 41 */
140   }; /* CustomHID_ConfigDescriptor */
```

图 29.8　配置描述符移植

配置描述符大部分都可以保持默认，除了红色箭头部分，箭头所指就是我们需要修改或者需要注意的地方。

102 行设置端点为两个，分别是端点 0 和端点 1。

104 行和 105 行是告诉主机设备的用途，其中 BOOT 和 NO BOOT 是说是否支持 BIOS 启动，这里随便一个设置即可。105 行要指定是键盘功能。

125 行和 136 行要制定主机和设备通信的数据包大小，即从机将键值发送给主机的数据长度，这里设置为 8 而且必须为 8。现在假设这 8 个字节组成的数组名为 KeyBuffer[8]，我们平时使用的模式切换按键或者扩展按键包括左右 GUI（WINDOWS），左右 CTRL，左右 SHIFT，左右 ALT，这些功能键一共 8 种，这 8 种按键被封装在 KeyBuffer[0]中，每一个按键功能对应该字节中的一位，该位为 1 表示按下，为 0 为没有按下，具体的位映射如下表：

位	7	6	5	4	3	2	1	0
功能	右 GUI	右 ALT	右 SHIFT	右 CTRL	左 GUI	左 ALT	左 SHIFT	左 CTRL

图 29.9　组合键对应位图

KeyBuffer[1]为预留，保持全为 0 即可。

KeyBuffer[2]- KeyBuffer[7]这六个字节每个字节负责存储一个按键键值，因此可以最多支持 6 键同时按下，如果超出 6 键则这 6 个字节全为 0xFF。另外在红色框选处还设置了主机中断查询的时间为 32ms（主机并不会精准的每隔 32ms 查询一次，而是保证查询时间不超过 32ms），这个时间可以用户自行调整。

最后要说明一下，配置描述符包括了接口和端点描述符。

- 报告描述符

```
141  const uint8_t CustomHID_ReportDescriptor[CUSTOMHID_SIZ_REPORT_DESC] =
142  {
143      0x05, 0x01,                    // USAGE_PAGE (Generic Desktop)
144      0x09, 0x06,                    // USAGE (Keyboard)
145      0xa1, 0x01,                    // COLLECTION (Application)
146      0x05, 0x07,                    //   USAGE_PAGE (Keyboard)
147      0x19, 0xe0,                    //   USAGE_MINIMUM (Keyboard LeftControl)
148      0x29, 0xe7,                    //   USAGE_MAXIMUM (Keyboard Right GUI)
149      0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
150      0x25, 0x01,                    //   LOGICAL_MAXIMUM (1)
151      0x75, 0x01,                    //   REPORT_SIZE (1)
152      0x95, 0x08,                    //   REPORT_COUNT (8)
153      0x81, 0x02,                    //   INPUT (Data,Var,Abs)
154      0x95, 0x01,                    //   REPORT_COUNT (1)
155      0x75, 0x08,                    //   REPORT_SIZE (8)
156      0x81, 0x03,                    //   INPUT (Cnst,Var,Abs)
157      0x95, 0x05,                    //   REPORT_COUNT (5)
158      0x75, 0x01,                    //   REPORT_SIZE (1)
159      0x05, 0x08,                    //   USAGE_PAGE (LEDs)
160      0x19, 0x01,                    //   USAGE_MINIMUM (Num Lock)
161      0x29, 0x05,                    //   USAGE_MAXIMUM (Kana)
162      0x91, 0x02,                    //   OUTPUT (Data,Var,Abs)
163      0x95, 0x01,                    //   REPORT_COUNT (1)
164      0x75, 0x03,                    //   REPORT_SIZE (3)
165      0x91, 0x03,                    //   OUTPUT (Cnst,Var,Abs)
166      0x95, 0x06,                    //   REPORT_COUNT (6)
167      0x75, 0x08,                    //   REPORT_SIZE (8)
168      0x15, 0x00,                    //   LOGICAL_MINIMUM (0)
169      0x25, 0x65,                    //   LOGICAL_MAXIMUM (101)
170      0x05, 0x07,                    //   USAGE_PAGE (Keyboard)
171      0x19, 0x00,                    //   USAGE_MINIMUM (Reserved (no event indicated))
172      0x29, 0x65,                    //   USAGE_MAXIMUM (Keyboard Application)
173      0x81, 0x00,                    //   INPUT (Data,Ary,Abs)
174      0xc0                           // END_COLLECTION
175  }; /* CustomHID_ReportDescriptor */

#define CUSTOMHID_SIZ_REPORT_DESC 63
```

图 29.10　报告描述符移植

关于键盘应用对应的报告描述符大家无需知道如果编写，USB 官方已经为我们提供了生成 HID 描述符的工具——HID Descriptor Tool，我们在资料中已经为大家提供了。

打开软件，选择 File-Open，然后选择对应的键盘描述符，最后将描述符另存为文本即可。如下图：

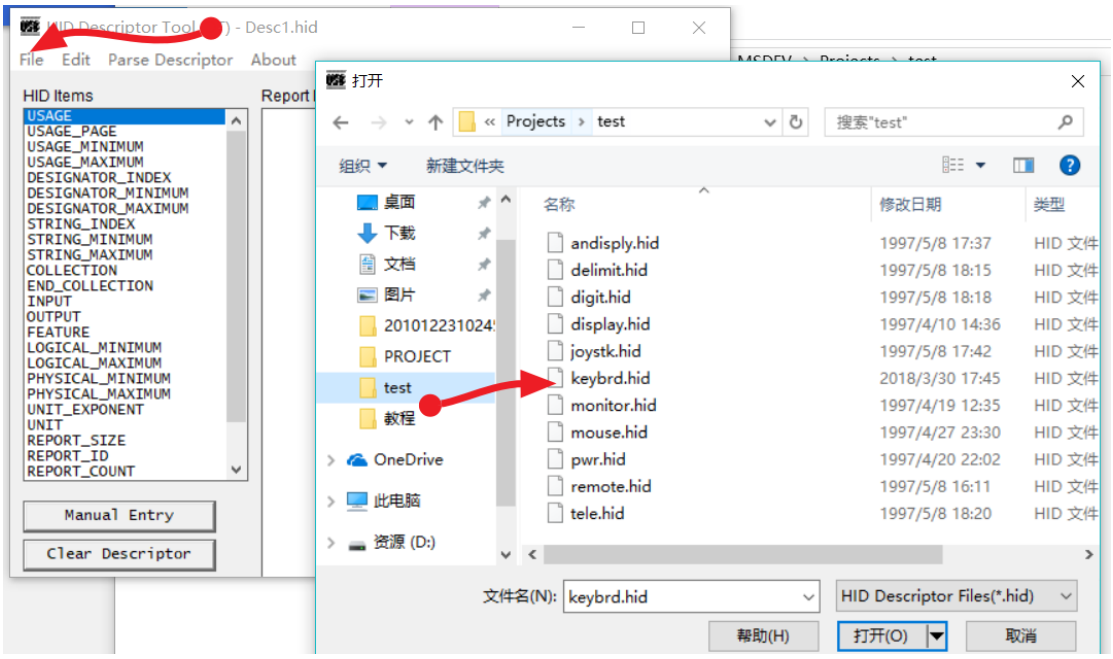


图 29.11　HID 描述符生成工具

这里要更改是将报告描述符数组大小改为 63，之前默认的是 163，这个宏定义位于 usb_desc.h 中。

修改 usb_prop.c

- CustomHID_Reset 函数

```
166  void CustomHID_Reset(void)
167  {
168    /* Set CustomHID_DEVICE as not configured */
169    pInformation->Current_Configuration = 0;
170    pInformation->Current_Interface = 0;/*the default Interface*/
171
172    /* Current Feature initialization */
173    pInformation->Current_Feature = CustomHID_ConfigDescriptor[7];
174
175    SetBTABLE(BTABLE_ADDRESS);
176
177    /* Initialize Endpoint 0 */
178    SetEPType(ENDP0, EP_CONTROL);
179    SetEPTxStatus(ENDP0, EP_TX_STALL);
180    SetEPRxAddr(ENDP0, ENDP0_RXADDR);
181    SetEPTxAddr(ENDP0, ENDP0_TXADDR);
182    Clear_Status_Out(ENDP0);
183    SetEPRxCount(ENDP0, Device_Property.MaxPacketSize);
184    SetEPRxValid(ENDP0);
185
186    /* Initialize Endpoint 1 */
187    SetEPType(ENDP1, EP_INTERRUPT);
188    SetEPTxAddr(ENDP1, ENDP1_TXADDR);
189    SetEPRxAddr(ENDP1, ENDP1_RXADDR);
190    SetEPTxCount(ENDP1, 8);
191    SetEPRxCount(ENDP1, 1);
192    SetEPRxStatus(ENDP1, EP_RX_VALID);
193    SetEPTxStatus(ENDP1, EP_TX_NAK);
194
195    /* Set this device to response on default address */
196    SetDeviceAddress(0);
197    bDeviceState = ATTACHED;
198  }
```

图 29.12　HID 复位函数移植

修改箭头处即可，目的是使能端点 1 的接收并且设置发送和接收的大小，这里所说的发送就是设备发送给主机的键值数组，接收是主机发给设备的键盘锁指示灯，大小写指示灯的控制结果，这里要明确一点，平时键盘上的各种指示灯是不归设备端管理的，这样做的原因是：当一台主机上挂接多个键盘时，在其中一个键盘上按下大写锁定后，会同步这种状态到所有键盘。其他状态指示也是如此。

最后还要将其他 USB 相关文件多余部分去掉，去掉的这些部分很简单，只要碰到 ADC 和 EXTI 相关的就把他们删除掉，因为这些部分是用来实现官方项目的。大家如果不想这么麻烦，可以直接使用作者移植好的 USB 库文件即可。我们还是希望大家能够用我们的工程文件和官方的对比一下，看一下究竟差别在哪里。

用户键盘驱动文件(KeyBoard.h/.c)

该文件是我们自己定义的，目的是用于将官方提供的函数结构进行统一封装，到最后呈现给使用者的就是一个很简单的发送键值到主机的函数。

```
/**
 * 功能：初始化USB
 * 参数：None
 * 返回值：None
 */
void initKeyBoard(void)
{

  USB_Interrupts_Config();//配置USB中断优先级
  Set_USBClock();         //设置USB时钟为72MHz的1.5倍分频,即：48MHz
  USB_Init();             //USB初始化
}

/**
 * 功能：将键值buffer(8字节)发送给主机
 * 参数：None
 * 返回值：None
 */
void sendKeyValue(u8* keybuffer)
{
    /*只有在上次发送完成并且USB成功连接至主机时才进行发送动作*/
    if(PrevXferComplete!=0 && bDeviceState == CONFIGURED)
    {
      USB_SIL_Write(EP1_IN, keybuffer, 8);  //将按键buffer写入到端点1输入(主机的输入)
      SetEPTxValid(ENDP1);                  //使能端点1发送
      PrevXferComplete = 0;                 //发送完成标志为清零,当发送完成后会自动由中断设置为1
    }
}
```

图 29.13　键盘初始化函数/发送键值函数

这两个函数一个用于初始化 USB，主要是时钟分频，中断配置，对于 USB 来说我们可以不用配置它的 GPIO 引脚，在开启 USB 功能后，GPIOA11 和 GPIOA12 自动分配给 USB 使用。当然官方提供了这个初始化函数的，只不过被我们删除了，官方提供的函数如下：

```
void Set_System(void)
{
  GPIO_InitTypeDef  GPIO_InitStructure;

  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA, ENABLE);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_11 | GPIO_Pin_12;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

图 29.14　USB 引脚初始化函数

在 KeyBoard.h 中，作者为大家定义了常用的按键键值，由于键值枚举篇幅较长，所以就不在这里展出。另外大家可以去附录 C 中查看 HID 的所有键值。

子项目 1 主函数

```
/************************************************************
 * HID键盘应用中，使用固定的8字节数组来存储键值
 * KeyBuffer[0]: 组合功能键
 *                 D7     D6      D5      D4      D3      D2      D1      D0
 *               右GUI   右ALT  右SHIFT  右CTRL   左GUI   左ALT  左SHIFT  左CTRL
 * KeyBuffer[1]: 预留,值为0
 * KeyBuffer[2]~KeyBuffer[7]:存储键值,每隔数据元素存储一个按键,最多存储6个
 *                          若大于6个,则这6个字节全部为0xFF
 ************************************************************/
u8 KeyBuffer[8] = {0};

int main(void)
{
    u8 keyvalue = KEY_NO;
    u8 i = 2;

    initSysTick(); //初始化Systick
    initLED();     //初始化LED
    initKey();     //初始化按键
    initKeyBoard();//初始化USB

    while(bDeviceState != CONFIGURED);

    while (1)
    {
        keyvalue= getKeyValue(KEY_PRESS); //获取键值
        switch(keyvalue)
        {
            case KEY_NO   :  memset(KeyBuffer,0,8);          break;//无按键按下要清零
            case KEY_UP   :  KeyBuffer[i++] = KEY_UPARROW;   break;//UP键

            case KEY_DOWN:  KeyBuffer[i++] = KEY_DOWNARROW; break;//DOWN键
            case KEY_ALL :  KeyBuffer[i++] = KEY_UPARROW;
                            KeyBuffer[i++] = KEY_DOWNARROW; break;//UP和DOWN键同时按下
            default      :                                  break;//其他值不作处理
        }

        sendKeyValue(KeyBuffer);                            //发送采集到的键值
        i = 2;                                              //恢复默认索引
        toggleLED();
        Delay_ms(20);
    }
}
```

图 29.15　主函数

这里需要说明的是，要想和主机通信就必须等待主机和设备连接成功，即 40 行代码处。另外在平时按键没有按下时要记得将按键缓冲清零并发给主机，否则即使你松开按键，主机此时还是会认为该按键没有被释放，最终的效果是按下一个按键之后，即使松开按键对应动作还会一直持续。

最后下载程序到开发板，并打开 3D 弹球，在游戏设置中将左右挡板控制按键更改为 UP 和 DOWN 键，下图是运行效果：

图 29.16　游戏界面

为了方便查看按键知否按下，我们也为大家下载了按键检测工具——KeyBoard Test Utility。需要注意：软件检测的键值和 HID 键值不是一回事。

图 29.17　按键检测工具

另外，大家还可以使用 UP 和 DOWN 键控制其他应用，比如文档翻页，幻灯片放映切换（如果再加上激光和外壳我们就已经可以做出一个幻灯片控制笔了）等。更多新奇好玩的使用场景大家可根据我们的程序进行进一步的扩展。

接下来我们开始第二个项目——一键控制计算机关机。

该部分没有单片机的知识需要讲解，唯一要说明的是关机的快捷键——WIN+D、ALT+F4,WIN+D 组合键作用是强制所有打开的界面使其最小化，更通俗一点的理解是按下该组合键之后会立即返回桌面，这么做的目的是只有在桌面按下 ALT+F4 才会弹出关机界面，

否则关掉的是当前激活的软件界面。接着在桌面按下 ALT+F4 会弹出关机界面，此时按回车即可关机，关机界面如下：

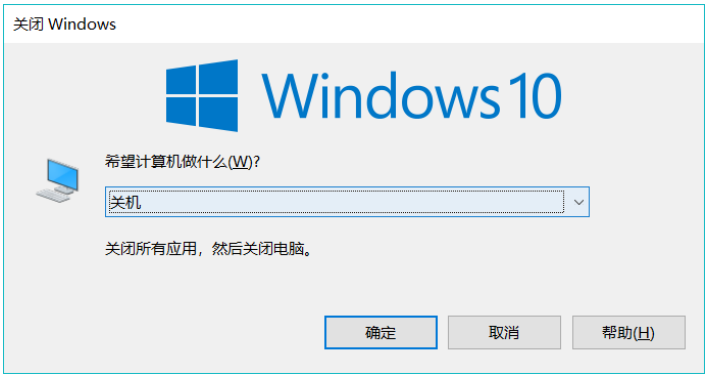


图 29.18　关机界面

因此按键顺序为 WIN+D—ALT+F4—Enter。

子项目 2 主函数

```
34  void ctrlComputer(void)
35  {
36      /*WIN + D退回桌面*/
37      KeyBuffer[0] = KEY_LGUI;
38      KeyBuffer[2] = KEY_D;
39      sendKeyValue(KeyBuffer);
40      memset(KeyBuffer,0,8);
41      sendKeyValue(KeyBuffer);
42      Delay_ms(1000);
43      /*ALT + F4弹出关机界面*/
44      KeyBuffer[0] = KEY_LALT;
45      KeyBuffer[2] = KEY_F4;
46      sendKeyValue(KeyBuffer);
47      memset(KeyBuffer,0,8);
48      sendKeyValue(KeyBuffer);
49      Delay_ms(1000);
50      /*Enter确认关机*/
51      KeyBuffer[0] = 0;
52      KeyBuffer[2] = KEY_ENTER;
53      sendKeyValue(KeyBuffer);
54      memset(KeyBuffer,0,8);
55      sendKeyValue(KeyBuffer);
56      Delay_ms(1000);
57  }
```

图 29.19　关机函数

这里要注意的是，为了保证计算机有足够的响应时间，因此在按下一组按键之后给了 1S 的延时，供计算机做出反应，另外为了防止同时发送多个按键，在发送完功能按键之后就发送了一组按键抬起操作。大家可以在保存正在处理的文件之后操作一下看看效果。也可以根

据我们提供的思路对其他快捷键做出扩展，甚至强制进入命令行（WIN+R-键入 cmd-Enter）并输入相关指令，比如定时（60s）关机，命令为：shutdown -s -t 60，如果想取消定时关机，则命令为：shutdown -a。

第三十章 独立看门狗_IWDG

30.1 项目要求

通过观察 LED 状态来验证独立看门狗是否产生复位。

注：本章用到核心板，对应例程 25。

30.2 原理讲解

30.2.1 认识看门狗

一套稳定健壮的嵌入式系统除了能够毫无差错的实现项目要求之外，还应具备一定的自我诊断和修复能力以应对复杂多变的运行环境，比如高温高湿高盐碱环境或者强电磁干扰环境等，工程师除了对硬件本身进行优化之外（比如防水外壳、金属屏蔽罩等）还应对软件增加一些抗干扰的处理，其中比较常用的就是独立看门狗。

独立看门狗其本质就是一个计数器，当其计数到临界值（比如递减到 0）则会产生系统复位，强制单片机重启，这样可以避免设备因某种软件或硬件的设计缺陷而导致的程序一直卡死。比如无人航天领域里如果因为强电磁干扰导致了某个子设备卡死，从而使整机瘫痪，此时就需要看门狗来强制重启，可见看门狗是挽救设备安全的一根救命稻草。

现在来讲解一下独立看门狗是如何实现只有在程序卡死时才会复位而正常情况不回复位的原因。这就要提到“喂狗”的操作，我们知道看门狗是一个不停计数的计数器（以递减为例），只要在其到达临界点 0 之前为计数器装载满值即可，这样就不会导致看门狗的复位动作，而当程序卡死时，喂狗动作随之也停止，因此一段时间后看门狗产生复位，要保证看门狗能够稳定工作的前提是：一个用户逻辑循环周期要小于看门狗复位周期，且循环周期中包含喂狗操作。其流程如下图：

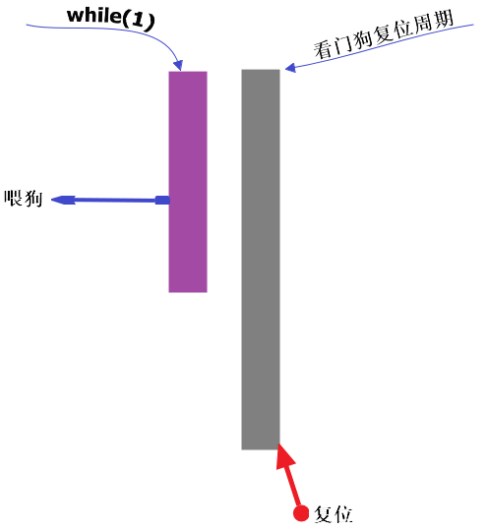


图 30.1 看门狗工作时序

看门狗很安全但也很危险，使用前有两种情况大家一定要注意：

1. 看门狗的复位周期不能略大于用户逻辑循环周期，因为大多数看门狗都是使用单片机内部不是很精准的 RC 时钟源提供的时钟，在不同温度甚至不同芯片之间其误差也是不同的，这就可能会导致部分设备不断的复位，从而失去了应有的功能。
2. 这里所说的用户逻辑循环周期并不仅仅代表 while(1){}内的代码段，而是还包含运行中断服务函数的时间，比如在外部中断中执行耗时任务，因此要将这些额外的时间开销也考虑进去。

建议在复位要求不是很高的场合，看门狗复位周期要高于用户逻辑循环周期的 30%以上。

30.2.2 STM32 独立看门狗讲解

先了解一下独立看门狗的内部结构：

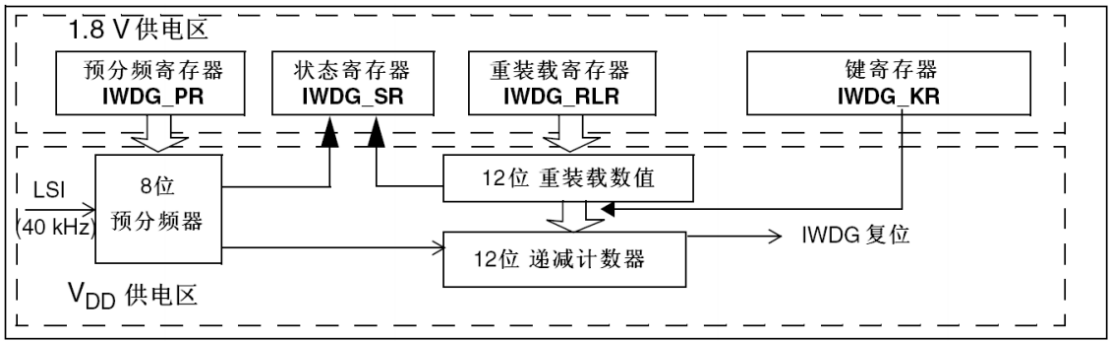


图 30.2　独立看门狗内结构框图

独立看门狗和其他定时器基本结构一样，都具有预分频器、重装载寄存器和计数器，STM32 的独立看门狗还有一个键寄存器，该寄存器根据不同的键入指令来执行不同的操作，比如当向键寄存器中写入 0xCCCC 时则开启看门狗，如果写入 0xAAAA 则将重装载寄存器中的值载入递减计数器中，即喂狗。对于看门狗的复位时间，官方参考手册中为我们提供了一个表格，我们可以在该表中选择合适的复位时间：

预分频系数	PR[2:0]位	最短时间(ms) RL[11:0] = 0x000	最长时间(ms) RL[11:0] = 0xFFF
/4	0	0.1	409.6
/8	1	0.2	819.2
/16	2	0.4	1638.4
/32	3	0.8	3276.8
/64	4	1.6	6553.6
/128	5	3.2	13107.2
/256	(6或7)	6.4	26214.4

图 30.3　独立看门狗复位之间参考表

如果想要自定义更多时间间隔，可以按照下面公式计算：

$$T = 4 * 2^{pr} * (RL+1)$$

T 为看门狗复位周期，pr 为预分频寄存器中的值，RL 为重装载值。

接下来继续了解 STM32 独立看门狗的各个寄存器：

- 键寄存器（IWDG_KR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

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

位31:16	保留，始终读为0。
位15:0	**KEY[15:0]**: 键值(只写寄存器，读出值为0x0000) (Key value) 软件必须以一定的间隔写入0xAAAA，否则，当计数器为0时，看门狗会产生复位。 写入0x5555表示允许访问IWDG_PR和IWDG_RLR寄存器。(见17.3.2节) 写入0xCCCC，启动看门狗工作(若选择了硬件看门狗则不受此命令字限制)。

图 30.4　键寄存器

该寄存器有 3 个键值：0xAAAA 用于重装载计数器，即产生喂狗动作，0x5555 用于在修改预分频寄存器和重装载寄存器之前取消寄存器保护。0xCCCC 用于启动看门狗工作。

- 预分频寄存器（IWDG_PR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															

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

位31:3	保留，始终读为0。
位2:0	**PR[2:0]**: 预分频因子 (Prescaler divider) 这些位具有写保护设置，参见17.3.2节。通过设置这些位来选择计数器时钟的预分频因子。要改变预分频因子，IWDG_SR寄存器的PVU位必须为0。 000: 预分频因子=4　　100: 预分频因子=64 001: 预分频因子=8　　101: 预分频因子=128 010: 预分频因子=16　　110: 预分频因子=256 011: 预分频因子=32　　111: 预分频因子=256 注意：对此寄存器进行读操作，将从VDD电压域返回预分频值。如果写操作正在进行，则读回的值可能是无效的。因此，只有当IWDG_SR寄存器的PVU位为0时，读出的值才有效。

图 30.5　预分频寄存器

在设置与分频值之前应先在键寄存器中写入 0x5555。对于该寄存器的读取我们一般用不到。

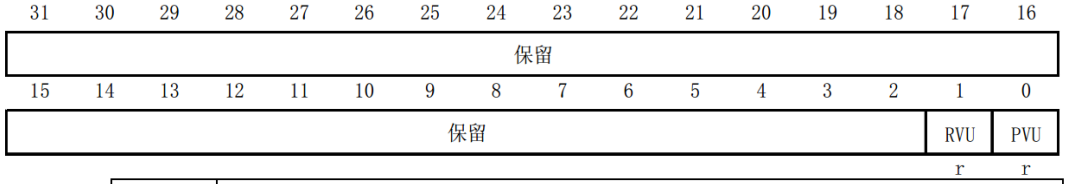
- 重装载寄存器（IWDG_RLR）

位31:12	保留，始终读为0。
位11:0	**RL[11:0]**: 看门狗计数器重装载值 (Watchdog counter reload value) 这些位具有写保护功能，参看17.3.2节。用于定义看门狗计数器的重装载值，每当向IWDG_KR寄存器写入0xAAAA时，重装载值会被传送到计数器中。随后计数器从这个值开始递减计数。看门狗超时周期可通过此重装载值和时钟预分频值来计算，参照表83。 只有当IWDG_SR寄存器中的RVU位为0时，才能对此寄存器进行修改。 注：对此寄存器进行读操作，将从VDD电压域返回预分频值。如果写操作正在进行，则读回的值可能是无效的。因此，只有当IWDG_SR寄存器的RVU位为0时，读出的值才有效。

图 30.6　重装载寄存器

当向键寄存器写入 0xAAAA 后将产生喂狗动作，即：将该寄存器中的值装载到计数器中。

- 状态寄存器（IWDG_SR）



位31:2	保留。
位1	**RVU**: 看门狗计数器重装载值更新 (Watchdog counter reload value update) 此位由硬件置'1'用来指示重装载值的更新正在进行中。当在VDD域中的重装载更新结束后，此位由硬件清'0'(最多需5个40kHz的RC周期)。重装载值只有在RVU位被清'0'后才可更新。
位0	**PVU**: 看门狗预分频值更新 (Watchdog prescaler value update) 此位由硬件置'1'用来指示预分频值的更新正在进行中。当在VDD域中的预分频值更新结束后，此位由硬件清'0'(最多需5个40kHz的RC周期)。预分频值只有在PVU位被清'0'后才可更新。

图 30.7　看门狗状态寄存器

RVU 和 PVU 用于指示此时预分频和重装载寄存器此时是否正在更新，如果处于更新中，则不可以对预分频和重装载寄存器赋值。

配置独立看门狗流程为：

1. 向键寄存器中写入 0x5555，允许对预分频寄存器和重装载寄存器的写入
2. 设置预分频值
3. 设置重装载值
4. 将设置的重装载值载入计数器
5. 开启看门狗

30.3 程序讲解

- 看门狗初始化函数

```
/**
 * 功能：初始化独立看门狗
 * 参数：
 *         IWDG_Prescaler：预分频值,取值范围如下：
 *                         IWDG_Prescaler_4：IWDG prescaler set to 4
 *                         IWDG_Prescaler_8：IWDG prescaler set to 8
 *                         IWDG_Prescaler_16：IWDG prescaler set to 16
 *                         IWDG_Prescaler_32：IWDG prescaler set to 32
 *                         IWDG_Prescaler_64：IWDG prescaler set to 64
 *                         IWDG_Prescaler_128：IWDG prescaler set to 128
 *                         IWDG_Prescaler_256：IWDG prescaler set to 256
 *         Reload：重装载值,0~0xFFF
 * 返回值：None
 */
void initIWDG(u8 IWDG_Prescaler, u16 Reload)
{
    IWDG_WriteAccessCmd(IWDG_WriteAccess_Enable);    //允许对预分频和重装载寄存器赋值

    IWDG_SetPrescaler(IWDG_Prescaler);               //设置预分频寄存器
    IWDG_SetReload(Reload);                          //设置重装载寄存器
    IWDG_ReloadCounter();                            //重装载计数器,即喂狗
    IWDG_Enable();                                   //开启独立看门狗
}
```

图 30.8　独立看门狗初始化函数

该函数用于初始化看门狗，主要是对预分频值和重装载寄存器进行设置并开启看门狗。

- 喂狗函数

```
/**
 * 功能：喂狗
 * 参数：None
 * 返回值：None
 */
void feedIWDG(void)
{
    IWDG_ReloadCounter();       //重装载计数器,即喂狗
}
```

图 30.9　独立看门狗喂狗函数

其原理就是向键寄存器中写入 0xAAAA。

- 主函数

```
14  int main(void)
15  {
16      initSysTick();
17      initLED();
18      Delay_ms(1000);
19
20      /*看门狗复位时间为410ms*/
21      initIWDG(IWDG_Prescaler_4,0xFFF);
22      while (1)
23      {
24          toggleLED();
25          Delay_ms(30);
26          feedIWDG();
27      }
```

注释掉观察现象

图 30.10　主函数

将主函数中的喂狗函数注释掉之后，会发现 LED 每隔一段时间停止闪烁，停止的时间为 1S，即此时处于 while(1){}前面的延时中，这就说明看门狗复位生效的，如果去掉注释，则 LED 会以 30ms 的周期闪烁，看门狗复位将不会发生。

第三十一章 窗口看门狗_WWDG

31.1 项目要求

通过观察 LED 状态来验证独立看门狗是否产生复位。

注：本章用到核心板，对应例程 26。

31.2 原理讲解

窗口看门狗是 STM32 的另一个用于监测、复位跑飞程序的看门狗，这里要理解窗口的概念，窗口指的是一个时间范围，只有在该范围内喂狗才不会产生复位。我们结合窗口看门狗的内部结构框图来说明其工作原理：

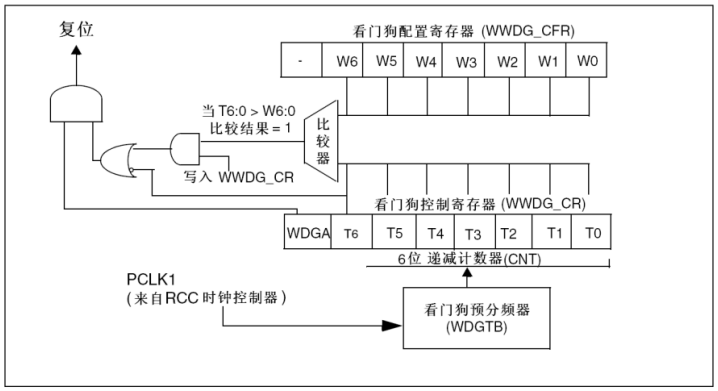


图 31.1 窗口看门狗内部结构框图

其中看门狗配置寄存器中存储的是一个我们设置好的固定值（W[6:0]），用于和看门狗控制寄存器中的递减计数器的值（T[6:0]）进行比较，如果在 T[6:0]大于 W[6:0]时喂狗，将产生一个单片机复位动作，如果在 T[6:0]小于 W[6:0]且大于 0x3F 时喂狗将执行正常的喂狗动作，单片机不会复位，此时的这个范围我们称之为喂狗窗口期，当 T[6:0]小于 W[6:0]且递减计数到 0x3F（不可设置的固定值）时，又将产生一个复位，其过程如下图：

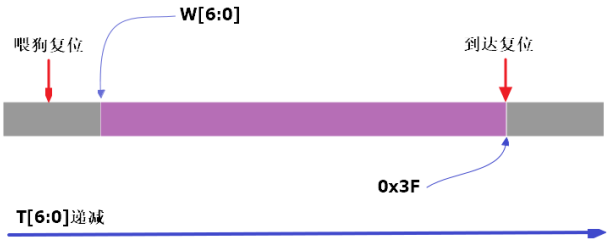


图 31.2 窗口看门狗复位过程

其中紫色区域为喂狗窗口期，只有在该时间段内喂狗才会生效不产生复位，计数值到达 0x3F 复位的真正原因是 T6 变为 0，由图中也可以看出，当 T6 为 0 后经过非门取反并连接到

或门，使或门输出为 1，随即产生一个复位。

另外当计数值计到 0x40 时（和复位只差一步）将会提前产生一个唤醒中断，这也是 STM32 的窗口看门狗和独立看门狗的最大区别，通常我们将喂狗的操作放在用户逻辑循环中，所以如果进入到该中断中就说明此时程序已经卡死，在该中断中可以对一些重要的数据进行保存，然后等待复位，但要注意该中断的执行时间不要太长，否则会导致还未存储完成就复位的危险，独立看门狗最长的计数器周期为 910us 左右，所以我们处理和保存的数据要小于该值。

下面介绍一下窗口看门狗的各个寄存器：

- 控制寄存器（WWDG_CR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留								WDGA	T6	T5	T4	T3	T2	T1	T0
								rs	rw	rw	rw	rw	rw	rw	rw

位31:8	保留。
位7	**WDGA**: 激活位 (Activation bit) 此位由软件置'1'，但仅能由硬件在复位后清'0'。当WDGA=1时，看门狗可以产生复位。 0：禁止看门狗 1：启用看门狗
位6:0	**T[6:0]**: 7位计数器(MSB至LSB) (7-bit counter) 这些位用来存储看门狗的计数器值。每(4096x2^{WDGTB})个PCLK1周期减1。当计数器值从40h变为3Fh时(T6变成0)，产生看门狗复位。

图 31.3　控制寄存器

控制寄存器包含了一个看门狗激活位以及一个 7 位的递减计数器。

- 配置寄存器（WWDG_CFR）

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留						EWI	WDG TB1	WDG TB0	W6	W5	W4	W3	W2	W1	W0
						rs	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:8	保留。
位9	**EWI**: 提前唤醒中断 (Early wakeup interrupt) 此位若置'1'，则当计数器值达到40h，即产生中断。 此中断只能由硬件在复位后清除。
位8:7	**WDGTB[1:0]**: 时基 (Timer base) 预分频器的时基可以设置如下： 00: CK计时器时钟(PCLK1除以4096)除以1 01: CK计时器时钟(PCLK1除以4096)除以2 10: CK计时器时钟(PCLK1除以4096)除以4 11: CK计时器时钟(PCLK1除以4096)除以8
位6:0	**W[6:0]**: 7位窗口值 (7-bit window value) 这些位包含了用来与递减计数器进行比较用的窗口值。

图 31.4　配置寄存器

该寄存器包括计数器的预分频、提前唤醒中断使能以及用于和 T[6:0]进行比较的窗口值，通常我们都会将窗口值设置为最大，即：0x7F。**此时只存在一种到达 0x3F 时的复位。**

窗口时间可以根据下面公式计算：

$$T_{WWDG} = 1/(f_{PCLK1}/4096/2^{WDGTB}) * (T[5:0] + 1))$$

其中 PCLK1 位 36MHz，WDGTB 为该寄存器中的值。

- 状态寄存器

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留															EWIF
															rc w0

位31:1	保留。
位0	**EWIF**: 提前唤醒中断标志 (Early wakeup interrupt flag) 当计数器值达到**40h**时，此位由硬件置'1'。它必须通过软件写'0'来清除。对此位写'1'无效。若中断未被使能，此位也会被置'1'。

图 31.5　状态寄存器

当计数器计到 0x40 是将由硬件置位该位，并由软件清零。

31.3 程序讲解

- 窗口看门狗初始化函数

```
12  /**
13   * 功能：初始化独立看门狗
14   * 参数：
15   *        WWDG_Prescaler: 预分频值,取值范围如下:
16   *          WWDG_Prescaler_1: WWDG counter clock = (PCLK1/4096)/1
17   *          WWDG_Prescaler_2: WWDG counter clock = (PCLK1/4096)/2
18   *          WWDG_Prescaler_4: WWDG counter clock = (PCLK1/4096)/4
19   *          WWDG_Prescaler_8: WWDG counter clock = (PCLK1/4096)/8
20   *       WindowValue: 窗口比较值,0x7F~0x40
21   *       Counter:计数值,0x7F~0x40
22   * 返回值：None
23   */
24  void initWWDG(uint32_t WWDG_Prescaler, uint8_t WindowValue, uint8_t Counter)
25  {
26      RCC_APB1PeriphClockCmd(RCC_APB1Periph_WWDG, ENABLE);     //使能WWDG时钟
27
28      WWDG_SetPrescaler(WWDG_Prescaler);                       //设置预分频值
29      WWDG_SetWindowValue(WindowValue);                        //设置窗口值
30      WWDG_Enable(Counter);                                    //设置计数值并开启WWDG
31  }
```

图 31.6　窗口看门狗初始化函数

在初始化函数中设置了看门狗的分频时基、窗口比较值、设置计数值并开启了独立看门狗，这里要注意的是大多数情况下都会将窗口比较值设置为最大，即 0x7F，此时就不存在由于不在窗口期喂狗导致的复位了。

- 喂狗函数

```
33  /**
34   * 功能：喂狗
35   * 参数：None
36   * 返回值：None
37   */
38  void feedWWDG(uint8_t Counter)
39  {
40      WWDG_SetCounter(Counter);       //重装载计数器,即喂狗
41  }
```

图 31.7　喂狗函数

在窗口期重装载计数值即可实现喂狗。

- 提前唤醒中断服务函数

```
45  void WWDG_IRQHandler(void)
46  {
47      WWDG_ClearFlag();             //清除标志位
48
49      /*快速做一些保存工作*/
50  }
```

图 31.8　提前唤醒中断服务函数

当预分频的时基设置为 8 时，独立看门狗的单个计数周期大约为 910us，所以要保证在该中断中的数据处理或者保存的时间要小于 910us。

- 主函数

```
15  int main(void)
16  {
17      initSysTick();
18      initLED();
19      initNVIC(NVIC_PriorityGroup_2);
20      Delay_ms(1000);
21
22      /*窗口时间大约为58ms*/
23      /*1/(36MHz/4096/8)*(127-64)*/
24      initWWDG(WWDG_Prescaler_8,0x7F,0x7F);
25      WWDG_EnableIT();
26
27      while (1)
28      {
29          toggleLED();
30          Delay_ms(30);
31          feedWWDG(0x7F);
32      }
33  }
```

图 31.9　主函数

主函数初始化独立看门狗的窗口时间和计数器时间均为 58ms，并在 while 循环中喂狗，此时程序可以正常执行，LED 以 30ms 周期闪烁，现在注释掉 31 行的喂狗操作，可以看到 LED 回每隔 1S 闪烁一次，说明已经重启。

到这里 STM32 外设部分的漫长学习之路就结束了，希望大家能够把这几十个实验部分的代码都再温习一遍，相信一定会有新的收获。接下来的网络学习和物联网实战就相对轻松一些了，这部分只占本教程的 10%左右，而且都是以我们学过的外设为基础实现的，可以说胜利就在眼前，希望大家怀着对电子的热情继续坚持下去。

网络篇

第三十二章 WIFI 联网

32.1 项目要求

了解 ESP8266，以及常用的 AT 指令。

32.2 原理讲解

32.2.1 ESP8266 讲解及固件烧录

介绍各种网络通信协议之前我们必须保证硬件设备先连接到网络才行，联网的方式有很多种，比如通过网线联网、GPRS 联网（SIM 流量业务）、或者家中 WIFI 路由器联网。这常用的三种方式只不过是硬件层不同而已，其内置的都是 TCP/IP 网络协议，其中最方便、成本最低的联网方式是 WIFI 联网，本章利用 AT 指令控制 ESP8266 WIFI 模块实现联网，为后面几个章节打下基础。

WIFI 射频芯片的供应商市面上有很多，常见的比如乐鑫，博通，瑞昱，德州仪器（TI）、庆科等，我们开发板上使用的就是合宙研发生产的 WIFI 模块——Air640,该模块是基于乐鑫的 ESP8266EX 芯片设计的，值得庆贺的是这款国产 WIFI 射频芯片在今年已经实现了出货 1 亿片，跻身于世界主流 WIFI 芯片供应商，ESP8266 的特点的就是价格便宜、资料齐全，支持 SDK 开发和 AT 开发。其中 SDK 开发可以省去外部的单片机，我们直接使用乐鑫提供的库函数来编写程序，使得成本降低，而 AT 开发是指使用外部单片机通过串口和 WIFI 模块通信，其通信的指令中包含 AT 前缀，这种开发方式虽然成本比 SDK 开发高，但是开发周期短。

另外除了合宙的 WIFI 模组之外，作者用的比较多的还有乐鑫官方推出的 ESP-WROOM-02 和安信可的 ESP8266 系列模组（常用的比如 ESP-12F）以及劢领的 MOM10xPO 系列（使用的是澜起的 WIFI 芯片）,MOM10xPO 模块的亮点是可以通过网页对其进行设置并支持 FTP 功能，这样可以很方便的对单片机进行在线升级。大家感兴趣的可以去了解一下，下图是这几家的模组预览（资源来自淘宝）：

图 32.1　产品展示

本章主要讲 AT 指令控制 ESP8266 模组。使用 AT 功能之前，模块内必须有 AT 的固件，每个模块的生产厂商都会对乐鑫官方的 AT 固件做一些修改和删减，但是其最核心的 AT 指令功能都是相同的，我们开发板上的 Air640 ~~默认刷入的是合宙官方的 AT 指令~~（目前开发板上的 WIFI 模块我们已经为大家下载好了乐鑫 AT 固件），为了方便大家扩展，我们预留出了 Air640 的下载接口，使得用户可以向开发板上的 Air640 刷入任何一家的 AT 固件。这里为了能让大家体验“原汁原味”的 AT 指令，我们以乐鑫官方最新的 AT 固件（Ver1.6.1）为例向大家讲解 AT 指令的使用方法，官方的 AT 指令参考资料齐全，且 AT 指令较多，使用起来也更加的灵活。在使用官方 AT 指令之前，需要按照如下步骤刷入官方 AT 固件：

1. 将开发板断电，并拔掉扩展板；
2. 使用我们提供的杜邦线将 **PA9 连接到 PA3，PA10 连接到 PA2**；
   由于开发板上使用的 CH340 是和 STM32 的 UART1 连接在一起的，而 Air640 是通过和 STM32 的 UART2 连接到一起的，所以想要使用 CH340 为 Air640 下载固件就必须将 UART1 和 UART2 连接到一起；
3. 使用 STLink 为 STM32 下载对应固件（**X_ESP8266 固件更新**）；
   为 STM32 下载固件的目的是设置 STM32 的各个引脚（主要是 UART1 和 UART2）为默认复位状态，即浮空输入，这样做可以避免使用 UART 为 Air640 下载固件时的产生干扰。我们打开程序后发现，其实什么也没有，只是让其开机并不断循环：

```c
/*****************************************************************
 * 文件：main.c
 * 功能：主函数入口
 * 日期：2018-02-16
 * 作者：zx
 * 版本：Ver.1.0 | 最初版本
 *
 * Copyright (C) 2017 zx. All rights reserved.
*****************************************************************/
int main(void)
{
    while (1);
}
```

图 32.2　固件更新时 STM32 默认程序

4. 按住 UP 键，并将开发板插入计算机的 USB 接口，等待 1S 后松开按键；
   ESP8266 和 STM32 一样，有多种启动模式，如正常运行模式，下载模式和测试模式等，各个模式通过 ESP8266 的 GPIO0,GPIO2 以及 GPIO15 在开机时的电平状态决定，如下图：

模式	CH_PD(EN)	RST	GPIO15	GPIO0	GPIO2	TXD0
UART 下载模式	高	高	低	低	高	高
Flash 运行模式	高	高	低	高	高	高
Chip 测试模式	-	-	-	-	-	低

图 32.3　ESP8266 下载模式

其中 EN、RST、GPIO2、GPIO0 内部都被上拉到了高电平，GPIO15 在模块内部也通过电阻下拉到了地，测试模式我们平时用不到，因此切换启动模式就只由 GPIO0 的电平状态来决定了，在开机后的几个时钟沿后会锁定这些引脚状态从而进入对应启动模式（在正常运行状

态下，改变 GPIO0 的电平状态无效，因此与 UP 键不发生冲突），这也就是为什么要按住 UP 键并上电的原因，成功进入下载模式后，模块上的红灯会亮起，如下图：

图 32.4　ESP8266 进入下载模式

5. 打开乐鑫官方烧录工具，进行如下设置，并烧录 AT 固件：

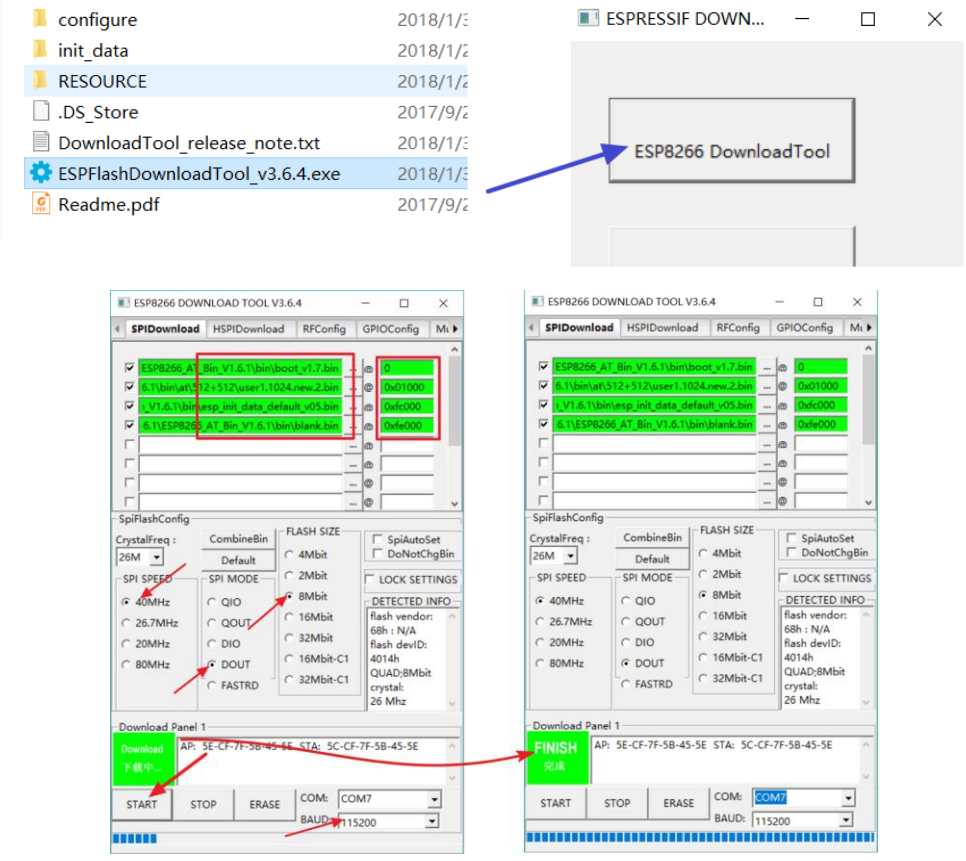


图 32.5　烧录软件操作流程

烧录的这四个固件，这四个固件位置如下（已为大家提供）：

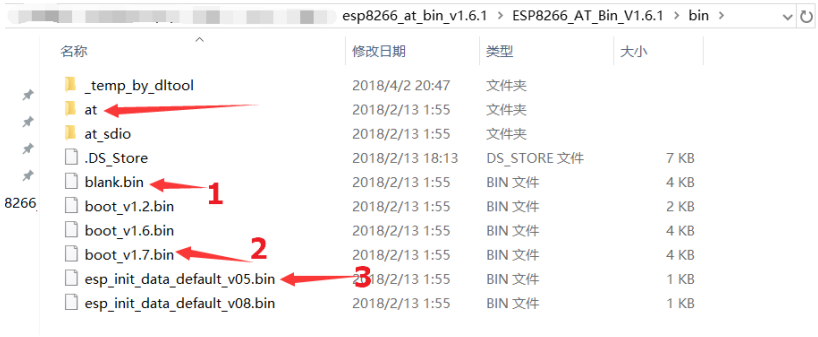


图 32.6　烧录固件文件位置

标号 1,3 用于初始化射频参数，标号 2 是 boot 文件，这些我们无需关心，在每次下载时记得添加到下载工具中即可，AT 固件则存放在 at/512+512 文件夹内的 user1.1024.new.2.bin 文件，该文件就是我们要下载的 AT 固件。

注：建议大家在刷不同版本的 AT 固件之前先点击 ERASE 按键擦除模块内部 flash，防止影响到新固件功能。另外乐鑫固件下载软件“脾气”不是很好，如果下载没有反应，就重新打开烧录工具，同时将开发板重新上电并进入下载模式。

6.　重新插拔开发板为开发板重新上电

经过这几步，ESP8266 的烧录工作就完成了，现在不要急着拔掉杜邦线，我们稍后会通过串口调试助手来给 ESP8266 发送指令，带领大家深入学习 ESP8266 和 WIFI 的一些概念。

32.2.2 AT 初探

乐鑫官方的 AT 指令有将近 100 条，但常用的就十几条，理解起来也非常简单，现在我们举例一些常用指令，并使用这些指令一步一步的通过 TCP 连接到远程的服务器实现收发数据，更多 AT 指令可以查阅我们为大家准备好的《ESP8266 AT 指令集手册》。

使用 AT 指令之前先说一说 AT 指令的构成（摘自乐鑫 AT 指令手册）：

类型	指令格式	描述
测试指令	AT+<x>=?	该命令用于该命令用于查询设置指令的参数以及取值范围。
查询指令	AT+<x>?	该命令用于返回参数的当前值。
设置指令	AT+<x>=<…>	该命令用于设置用户自定义的参数值。
执行指令	AT+<x>	该命令用于执行受模块内部程序控制的变参数不可变的功能。

⚠ 注意：

- 不是每条 AT 指令都具备上述 4 种类型的命令。
- [ ] 括号内为缺省值，不必填写或者可能不显示。
- 使用双引号表示字符串数据 "string"，例如：AT+CWSAP="ESP756290","21030826",1,4
- 默认波特率为 115200。
- AT 指令必须大写，并且以回车换行符结尾（CR LF）。

图 32.7　AT 指令格式定义

总结 AT 指令的构成就是，每条指令要以 AT 开始，后面跟要查询（读）或者要设置（写）的参数，例如查询 WIFI 模式对应的指令为 AT+CWMODE?，设置 WIFI 模式为 AT+CWMODE=3。令外要求的回车换行符结尾是说要在待发送的数据后面追加\r\n，即十六进制的 0x0D 0x0A。

注意：在使用串口调试助手发送 AT 指令时，只需要在待发送指令后面加回车即可，大家可以勾选“按十六进制发送”，观察数据后面是否追加了 0x0D 0x0A。如果在数据后面添加\r\n 则会将其作为字符发送而非换行符。

1. 测试模块是否正常（**AT**）

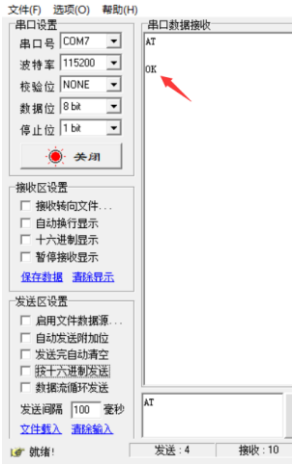


图 32.8　AT 测试

该指令通常在开机后查询模块是否正常启动，如果回复 OK 则表示为正常启动。

2. 开启/关闭回显（**ATE1/ATE0**）

从上图可以看出，我们发送了一个 AT，模块回复了 AT　OK，即模块将发送过来的指令原封不动的先复述了一遍后接着发送有效回复，我们称这种复述为回显，关闭回显（发送 ATE0）后如再次发送 AT，则只回复 OK。如下图：

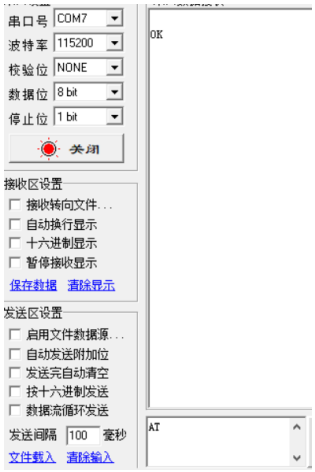


图 32.9　关闭回显测试

3.　设置 AP 模式及 AP 参数（**AT+CWMODE，AT+CWSAP_DEF**）

WIFI 模式有两种，一种叫 AP 模式，一种叫 Station 模式，AP 就是我们平时所说的热点，如 WIFI 路由器，开了热点的手机，或者是公共热点等，这些 AP 设备可以允许其他设备（如手机，笔记本电脑等）输入热点名和密码（也可不设置密码）后接入，Station 则是前面说的连接 AP 的设备，如：手机，笔记本电脑等，ESP8266 还有第三种模式：AP+Station，即：将 AP 和 Station 的功能合二为一，但是应用的场景不多，这里不做展示。

当 ESP8266 设置为 AP 模式时，其他设备可以接入该热点，最多支持 4 台 Station 设备接入。AP 模式也是 ESP8266 默认的模式。设置 ESP8266 流程如下：

- 设置 WIFI 模式为 AP 模式（**AT+CWMODE=2**）
  另：**AT+CWMODE=1** 为 Station 模式，**AT+CWMODE=3** 位 AP+Station 模式
- 设置 AP 热点属性（**AT+CWSAP= AT+CWSAP_DEF="fengmei","1234567890",5,3**）

其含义为：热点名为 fengmei，密码为 1234567890,使用通道 5，加密方式为 WPA2_PSK,这里的通道对应的就是不同的射频频率，如果同一空间内存在相同通道的 WIFI 信号，将会产生干扰，影响上网质量，因此可以设置通道来避免这种干扰，常用的通道有 1、6、11，因为这三个通道互不产生干扰。

另外还有两个可选参数，如下图：

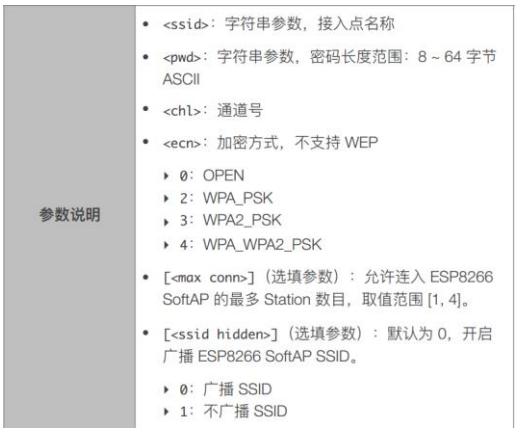


图 32.10　AP 设置参数

连接数量可以限制 Station 设备的接入数量，广播或者不广播 SSID 就是指是否隐藏热点名，使热点更加安全。另外 **AT+CWSAP=AT+CWSAP_DEF** 表示设置的参数会存储的 flash，还有另外一个类似指令 **AT+CWSAP=AT+CWSAP_CUR**，该条指令表示设置的参数重启后失效，即不保存到 flash 中。其他的 AT 指令也有类似的后缀。下图是作者的笔记本连接到热点后又断开的情况下，ESP8266 输出的信息：

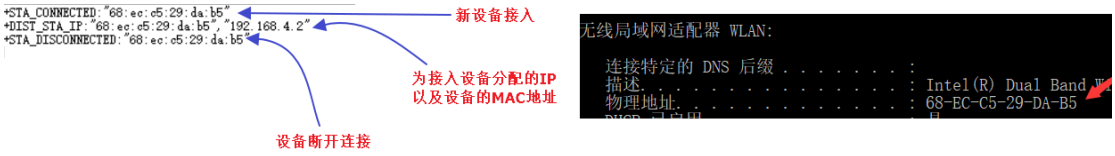


图 32.11　设备接入 AP 打印消息

这里说的 MAC 地址也是一个很重要的概念，MAC 地址也叫物理地址或者硬件地址，它是区分和指定具体某一个设备的唯一标识，可以理解为硬件指纹。假设现在存在两个同名的热点，则我们可以在连接 AP 的 AT 指令（下面介绍）后面指定待连接 AP 设备的 MAC 地址来选择性连接。补充：Windows 中查询 MAC 地址和 IP 地址的方法为命令窗口输入：ipconfig/all。

4.　设置为 Station 模式（**AT+CWMODE=1，AT+CWJAP**）

该模式是平时应用最多的模式，因为物联网设备需要连接到家中路由才可以接入外网，此时设备就作为 Station 连接到 AP 热点。设置 Station 并连接 AP 流程如下：

- 设置 WIFI 模式为 Station（**AT+CWMODE=1**）
- 连接到家中路由器（**AT+CWJAP_DEF="Xiaomi","fml2345"**）

  下图为连接过程的串口输出：

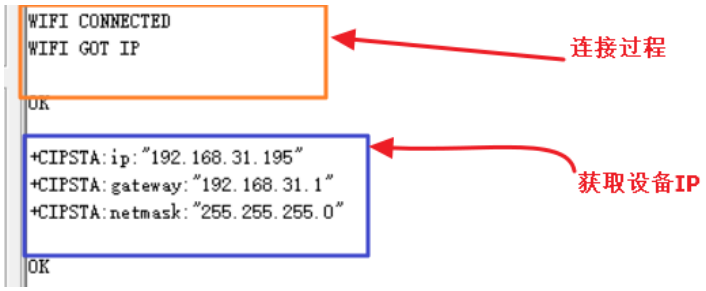


图 32.12　ESP8266 作为 Station 接入路由器

获取 IP 可以使用 AT+CIPSTA?指令，返回的是路由器分配给 ESP8266 的局域网 IP 以及网关地址和子网掩码。如果此时存在多个同名热点，可以使用如下指令：**AT+CWJAP="abc","0123456789","ca:d7:19:d8:a6:44"**，后面跟的就是对应路由器的 MAC 地址。另外可以使用开机自动连接热点指令：**AT+CWAUTOCONN=1**，这样 ESP8266 上电后就会自动连接之前存储的路由器。

在平时应用当中也会用到自动扫描当前可用 AP 热点指令：**AT+CWLAP**，ESP8266 经过扫描后会打印可用的 AP 以及每个 IP 的信息（热点名，热点 MAC 地址，加密方式，使用信道，信号强度等），如下图：

```
+CWLAP:(3,"beini搜?iPhone",-93,"a6:41:67:78:76:50",1,-17,0,4,4,7,0)
+CWLAP:(4,"M B L Q",-93,"a4:56:02:bf:3d:b6",1,-22,0,4,4,7,0)
+CWLAP:(4,"Xiaomi",-46,"50:64:2b:04:f7:49",3,-39,0,5,3,7,1)
+CWLAP:(4,"TP-LINK_1001",-83,"b8:f8:83:e6:82:45",6,-21,0,4,4,7,0)
+CWLAP:(4,"TP-LINK_C2DB",-91,"48:7d:2e:1e:c2:db",11,-26,0,4,4,7,0)
+CWLAP:(4,"ChinaNet-9Xk3",-81,"2c:63:73:78:30:81",11,-22,0,5,3,7,1)
```

图 32.13　扫描附近可用热点

如果想要断开连接，可以使用 **AT+CWQAP**。

5.　使用 TCP 实现局域网内的设备通信

- 建立 TCP 连接（**AT+CIPSTART**）

该条指令可以指定建立连接的协议类型，通常使用的有两种：TCP 和 UDP。我们先打开网络调试助手，并将其设置为 TCP Server 端，具体设置如下：

图 32.14　TCP 测试设置参数

接着发送 AT 指令建立 TCP 连接：**AT+CIPSTART="TCP","192.168.31.238",8234**，成功连接会提示 CONNECT，接着使用发送指令：AT+CIPSEND=15，其中 15 是发送数据的长度，该指令发送完成后，接收窗口会显示 > ，我们接着在发送窗口发送“I`m TCP Client ”,TCP Server 端会收到该信息，接着再通过 TCP Server 发送“I`m TCP Server”，串口接收端会打印 Client 收到的数据，结果如下图：

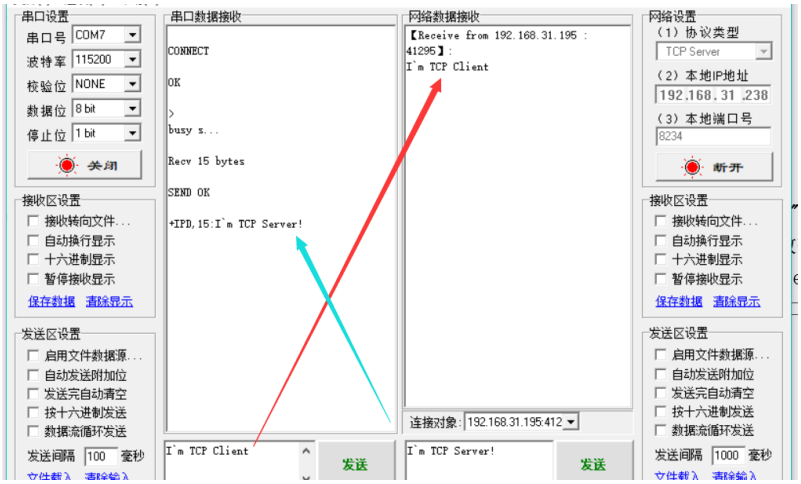


图 32.15　TCP 通信流程

收到的数据包含了一个固定的接收前缀“+IPD,”，后面跟的数字表示接收到数据字节数，后面则是数据。

- 开启透传传输（**AT+CIPMODE=1**）

前面在使用 TCP 进行数据发送时，在每次发送数据之前都要指定发送数据的长度，而且在接收到数据之后，还会有+IPD,<len>的前缀，这样很不方便进行数据的处理，因此我们可以使用 **AT+CIPMODE=1** 指令开启透传模式，开启透传模式后只需要在第一次发送数据时使用 **AT+CIPSEND** 指令来告诉 ESP8266 开始透传发送，随后我们直接发送想要的数据即可，在接收到数据时，也没有了+IPD,<len>前缀，如下图：

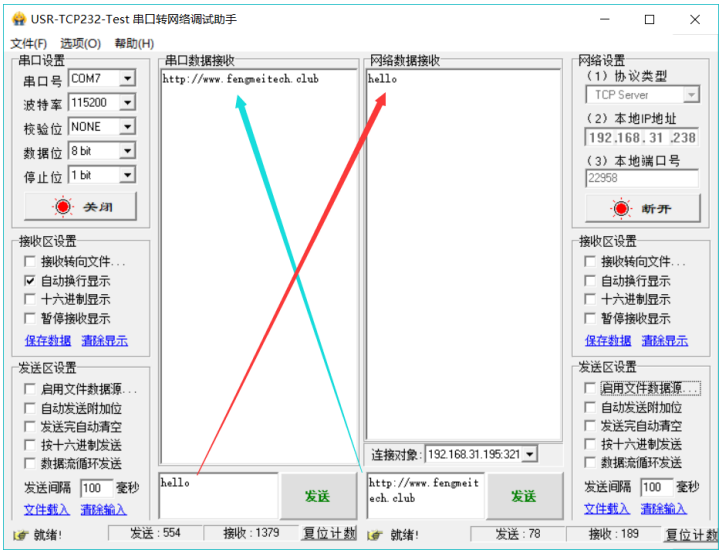


图 32.16　透传模式通信效果

如果想要退出透传发送模式，先发送+++（0x2B 0x2B 0x2B），注意没有换行符，接着使用 AT+CIPMODE=0 指令退出透传模式，恢复到默认传输模式。

6.　使用 SmartConfig 为设备配网（AT+CWSTARTSMART=<type>）

前面使用了 AT+CWJAP 指令来主动连接家中的 WIFI，但是在大多数的物联网产品中，缺少输入 WIFI 密码的输入设备，如：键盘，更不能将程序交给用户去修改家中的 WIFI 热点名和密码，在真正的项目开发中使用最多的方法就是通过一颗按键来使设备进入某种模式，并使用手机将当前 WIFI 的密码告知该设备来实现 WIFI 配网，这种模式就是 SmartConfig（不同厂商叫法不一，但功能相近），在该模式下 ESP8266 会监听指定端口的 UDP 广播包，如果收到符合规定格式的广播包后会对其进行解析并获得 WIFI 的 SSID 和 PWD，然后自动连接获取到的 WIFI 热点，从而实现 WIFI 配网。我们可以使用 AT+CWSTARTSMART=<type>指令来设置 ESP8266 使其进入 SmartConfig 模式，其中的 type 是指不同的配网协议，如下图：

参数说明	<type>： ▸ 1：ESP-TOUCH ▸ 2：AirKiss ▸ 3：ESP-TOUCH+AirKiss

图 32.17　配网参数

其中的 ESP-TOUCH 是乐鑫官方的配网协议，AirKiss 是微信推出的配网协议，由于微信拥有庞大的用户群体，所以很多厂商和产品都支持 AirKiss 协议配网。下面我们使用配网 AT 指令使 ESP8266 进入配网状态，此时大家扫描关注下方微信公众号，该公众号是后面例程要接入的云平台的厂商公众号，里面有 WIFI 配网功能，使用的是 AirKiss 协议：

图 32.18　TLink 微信公众号

在公众号中点击 WIFI 配网按钮，接着进入了设备配网界面（此时要保证手机连接到 WIFI 路由器），点击立即链接并输入 WIFI 密码，最后点击连接即可，效果如下：

图 32.19　配网结果

ESP8266 检测到了配网协议类型以及 WIFI 的 SSID 和 PWD 并自动连接到 WIFI 路由器，此时 ESP8266 会将这些 WIFI 信息存储到 flash 中，并且在设备每次重启时会自动联网。需要注意的是配网完成之后要记得停止配网模式（**AT+CWSTOPSMART**），释放 ESP8266 的内存。但是我们在后续的例程中并没有使用 SmartConfig 进行配网，而是使用手动连接，其目的是让大家理解 WIFI 联网的过程。

7.　获取网络时间（**AT+CIPSNTPTIME?**）

在前面的 RTC 章节实现了一个带闹钟功能的日历，但是随着时间的推移日历的时间会逐渐的积累误差，作者的测试结果是：在 7 天内时钟快了 18S，这是因为外部时钟信号或多或少的都会存在一些误差，这是我们无法改变的，当然我们可以在程序中每隔一段时间进行一次校准（增加或者减少偏置），但是更好的办法是能和网络时钟同步，幸运的是 ESP8266 为我们提供了获取网络时间的功能（SNTP），我们可以使用 SNTP 对 RTC 定期进行校准，甚至可以直接使用 SNTP 实现日历功能。在获取 SNTP 时间之前要先开启 SNTP 并设置时区，该设置指令为 **AT+CIPSNTPCFG=1,8**，接着使用 **AT+CIPSNTPTIME?**指令来查询日期和时间，如下图：

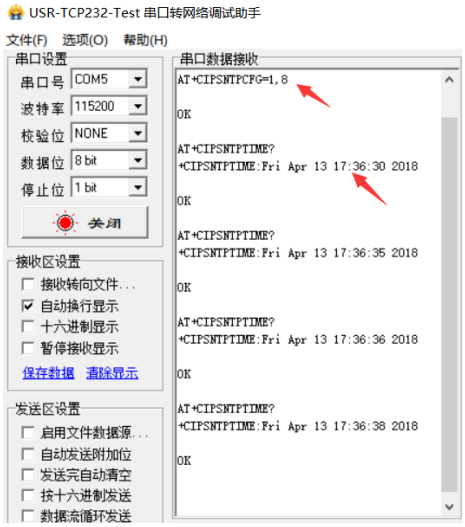


图 32.20　获取网络时间结果

关于更多的 AT 指令可以去翻阅我们为大家准备好的乐鑫官方的 ESP8266 AT 文档，也可以去乐鑫官网下载更多文档资源。

第三十三章 TCP/IP 网络协议

33.1 TCP/IP 模型概述

本章开始讲解计算机网络相关的协议。计算机网络在物联网应用中起到举足轻重的作用，它是硬件设备连接网络的基石。目前提到计算机网络协议指的就是著名的 TCP/IP 协议，我们身边只要是连接到网络的设备其内部都是使用的 TCP/IP 协议。TCP/IP 协议是众多协议的集合，根据 TCP/IP 的分层不同，对应的协议族也不同，下图是列举了 TCP/IP 包含的各个层的通信协议：

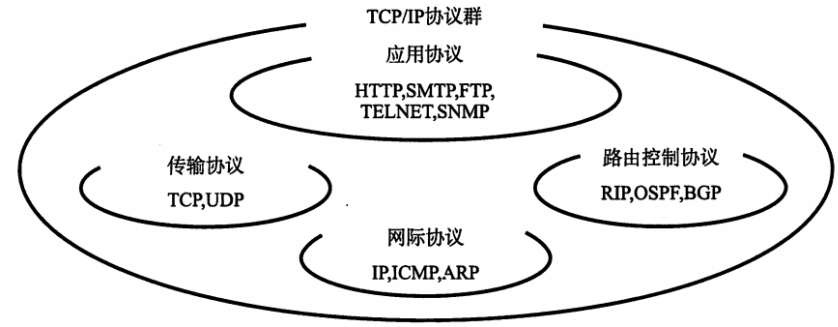


图 33.1 TCP/IP 协议族

在平时的物联网项目的开发时，并不会涉及到所有协议，常用的就是传输层的 TCP 协议和 UDP 协议，需要了解的是应用层的 HTTP、DHCP、DNS、FTP、SMTP 和 NTP 协议，以及一些网络基础知识，如 IP 的概念和分类，域名的划分和解析等。

由于 TCP/IP 网络协议非常复杂，介于本书篇幅有限，我们只讲解开发时用到的一些基础知识和基本的通信协议，能够保证的是，大家在学习了这些基本的通信协议之后再去深入学习计算机网络将会非常容易。

TCP/IP 网络最早起源于美国国防部的 ARPANET 项目，该项目的目的是研发一种可以实现多台计算机进行通信的网络，美国军方希望当计算机网络中的一个节点被敌人破坏后，不同计算机之间可以自动迂回的选用其他网路进行通信，目的是提高计算机网络通信的容灾性。ARPANET 在 1969 年正式投入使用，该年也称为计算机网络元年，在几年后，ARPANET 推出了最初的 TCP/IP 协议，TCP/IP 协议在几经修改后最终在 1983 年被指定为 ARPANET 的唯一指定协议，之后随着 UNIX 的发展，TCP/IP 协议也逐渐流行起来，最后由军用变为了民用，现在 TCP/IP 协议已经被各个操作系统作为默认组件嵌入到系统当中。现在我们每个人无时无刻的都在和 TCP/IP 协议打交道，只不过这种协议被友好的操作界面所隐藏了。

我们先从宏观上了解一下 TCP/IP 协议。TCP/IP 由于十分复杂，因此采用了分层设计的思想，共分为 4 层，从上往下依次是调用的关系，如下图：

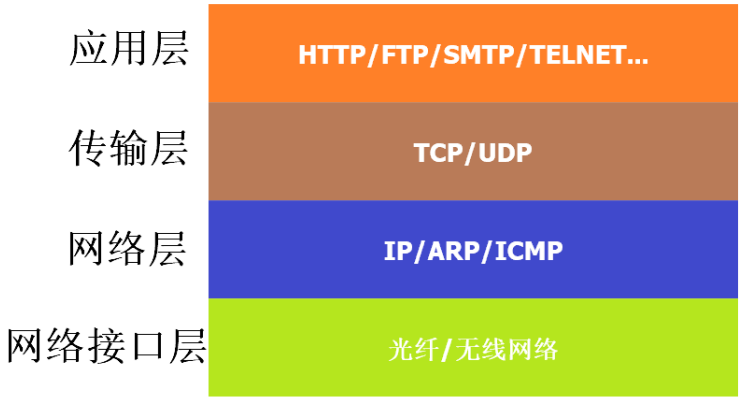


图 33.2　TCP/IP 协议四层结构

平时所说的 TCP/IP 协议位于应用层、传输层和网络层，网络接口层只是实现 TCP/IP 协议的物理设备和驱动（网卡）。网络层用于甄别数据传输介质中的某一数据包的是否属于本机，如果是则解析数据包并去掉数据的 IP 首部，接着将数据发给传输层处理，传输层在经解析去掉 TCP 首部，并将数据发送给对应的端口。也就是说数据的发送是从应用层开始封包，一层一层的套包装，接收数据则是拆包装的过程。如下图所示：

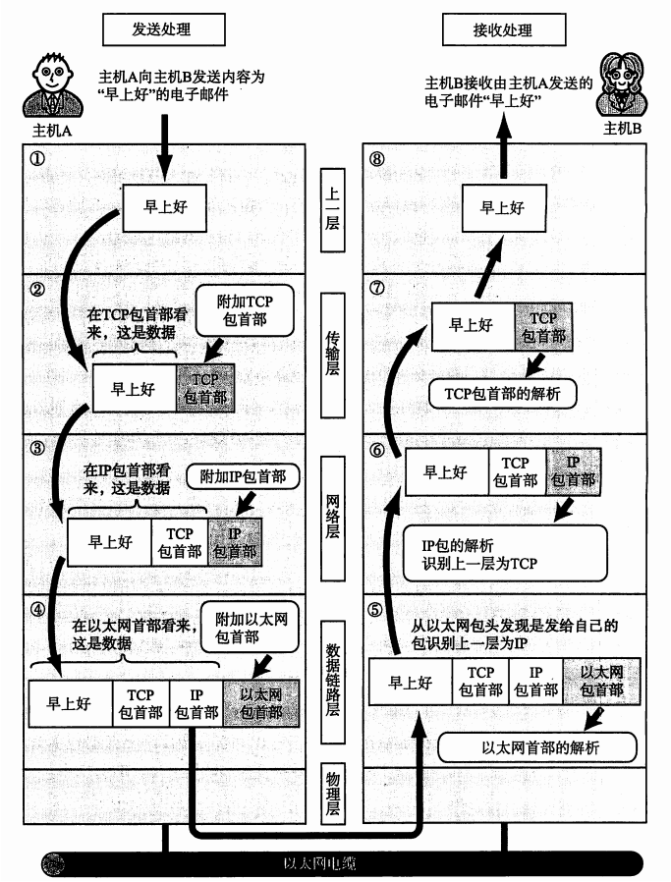


图 33.3　TCP/IP 消息发送流程

另外，TCP/IP 的这四层模型通常会和 OSI 的七层模型作对比，如下图：

图 33.4　TCP/IP 和 OSI 分层对比图

对于 OSI 模型就不做过多讲解，如果想了解更详细的内容，可以点击下载科来 TCP/IP 网络协议图。

33.2 TCP/IP 封包和共享传输介质

TCP/IP 采用共享传输介质的方式来进行多机的数据传输，比如同处于家庭无线局域网内的所有设备，或者使用同一根网线的多台计算机，如果不采取任何机制直接使用这种方式进行多台计算机的网络通信就会导致一个很严重的问题：一台计算机想要发送一个大文件，则会长时间占用硬件传输介质，迫使其他计算机处于等待状态，使得网络速度下降，传输实时性降低，为此 TCP/IP 针对共享传输介质采用了封包的概念，即：将用户的信息切割成多份并投放入传输介质中，但是这种封包进行传输不能保证各个数据包是按顺序传输的，因此 TCP/IP 为每个包都加入了顺序编号，接收端会将这些数据包按照编号重新整理组合成原始数据。在使用这种封包的传输机制之后，使得每台计算机的等待时间大大降低，在计算机想要发送数据之前，它会先在传输线上试探一下是否有数据包正在发送，如果有数据发送它会自己掷骰子来生成一个随机延时，等到延时结束接着试探，以此类推，所以我们会有这种经历，当一条网线上的接入设备较多后，网络会卡顿，其原因就是你的计算机检测到此时网线上有数据传输，自己在那里不停的掷骰子，导致传输延时。如下图举例，ABC 三个用户正在使用通讯软件传输消息，但这些消息在传输介质中却是以不可预知的顺序传输的。

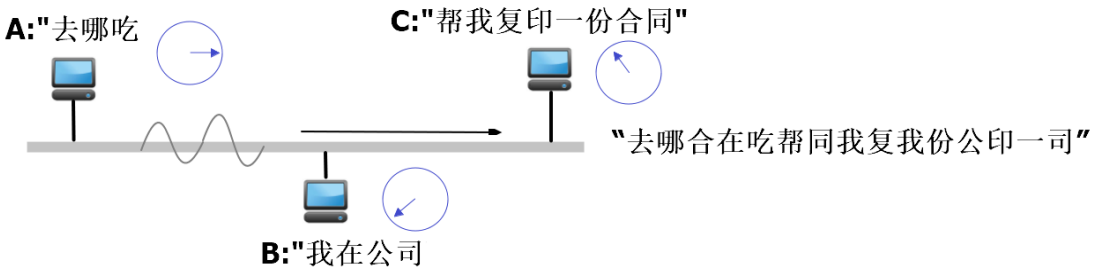


图 33.5　TCP/IP 数据传输示意图

33.3 IP 地址及端口号

双方计算机进行通信的前提是知道目标计算机的网络位置，即我们所说的 IP 地址，这是因为 IP 地址具有全网唯一性，大家最熟悉的 IP 地址大概就是 192.168.1.1 了。现在普遍使用的是 IPv4 协议，用 32 位二进制表示 IP 地址，但为了方便人类识别、记忆将其以 8 位为单位进行分割，共分成 4 份，每份之间用句点隔开，并使用十进制表示，即：点分十进制表示法，其取值范围为 0-255。这样看来 IP 地址有 2^{32}（4294967296）个，看似很难用完，但由于最初 IP 地址划分保留了一些 IP 地址导致这些地址不可用，另外随着更多用户接入互联网，最终在 2011 年 2 月 3 日 IP 地址被耗尽，现在供人类选择的就只有 IPv6 协议了，IPv6 使用 128 位表示 IP 长度，理论上可以产生 2^{128} 个 IP 地址，IPv6 使用 8 组 4 个十六进制数表示地址，如：1027:0cd6:7aa3:16f3:3329:4c6e:a360:4637。

IP 地址根据应用场景通过改变网络号和主机号的范围进行了分类，这些分类包有：A 类，B 类，C 类，D 类，E 类，我们常用的是前三者，下图是分类方法（以 B 类 IP 为例）：

图 33.6　IP 地址构成

网络号可以表示网络的数量，主机号则是每个网络下的主机数量，对于 A 类地址来说，其 IP 取值范围是：0.0.0.0-127.255.255.255,网络号为地址第一段号码，后面的三段为主机号，B 类地址取值范围是：128.0.0.0-191.255.255.255，前两段为网络号，后两段为主机号，如上图所示，C 类地址取值范围是：192.0.0.0-223.255.255.255，前三段为网络号，最后一段为主机号，根据这种分类可以看出由 A 到 C 网络数量逐渐变多，主机数量逐渐变少，对应的应用场景为大型网络、中型网络，小型网络。

一台计算上会同时运行多个不同的网络应用，比如 SMTP，FTP，HTTP 等，他们都处于一个 IP 地址下，因此就需要一种标识来告诉传输层接收到的数据要送到应用层的哪个具体应用中，这个标识就是端口号，端口号是一种虚拟的逻辑上的端口划分，这样做的好处是可以使各个应用合理的使用网络资源。端口号有 16 位，即可以表示 65536 个端口，这些端口其中一部分是被固定分配给指定的应用，比如 HTTP 占用 80 端口，SMTP 占用 25 端口，我们在做测试时可以在 Windows 命令号中输入“netstat -ano”命令来查看此时被占用的端口，避

免发生端口冲突。

33.4 ARP 协议

在前面已经知道了 IP 是识别一台计算机的唯一标识，但最终操作数据的还是网卡，所以要知道网卡的物理地址，即 MAC 地址，因此就需要一种能够将 IP 这一逻辑标识和 MAC 地址物理标识进行转换的协议，这种协议就是 APR 协议。大家了解其功能即可，无需深入。

33.5 DHCP（IP 户口登记）

之前说过 IPv4 提供的 IP 资源有限，不可能为每一个人提供一个固定的 IP，因此就会导致很多人因为没有 IP 地址而不能上网，想要解决这个问题最直接的办法就是使用更大 IP 容量的 IPv6 协议，但是想彻底从 IPv4 过渡到 IPv6 困难重重，因此人们想到了另外一个缓解办法：动态分配 IP。即为了每一个设备分配一个临时的 IP，当一个设备关机后这个空闲的 IP 地址就会分配给其他主机使用，一个主机开机就会请求路由器为其随机分配一个可用 IP，这样就使得原本短缺的 IP 资源动态的流动起来，使其使用效率大大提高。我们称这种方式为 DHCP（Dynamic Host Configuration Protocol）。

在一个主机刚开机时，由于没有 IP 地址，它会通过 DHCP 协议外广播,请求该网络内的 DHCP 主机（路由器）为其分配一个 IP，当 DHCP 主机收到这个请求后会在可用的 IP 中选择一个并广播出去，有 IP 地址的计算机会忽略这种广播，只有没有 IP 地址的主机才会将该广播接收，并将新 IP 地址为自己所用。在 Windows 操作系统中，TCP/IPv4 属性中的自动获得 IP 地址选项就用于选用或者关闭 DHCP 功能：

图 33.7　Windows 下 DHCP 设置

如果不开启 DHCP，则需要使用者自行指定 IP 地址，但前提是不能和该网络中的其他主机 IP 发生冲突，如果该网络下主机数量很多，将会使加剧这种冲突，所以一般采用 DHCP 来动态为我们主机分配 IP，避免不必要的麻烦。

33.6 NAT（IP 易容术）

NAT（Network Address Translation，网络地址转换），是另一种缓解 IP 资源短缺的方法，我们一定有过这种经历：无论在哪里连接 WIFI，大多数情况下上网设备分配的 IP 地址都是 192.168.x.x，该类地址称之为私有地址，在每个网络分类中都指定了某一 IP 地址范围为私有地址如 A 类 IP 的私有 IP 地址范围是：10.0.0.0～10.255.255.255，B 类 IP 的私有 IP 地址是：172.16.0.0～172.31.255.255，C 类 IP 私有 IP 地址是：192.168.0.0～192.168.255.255。这类私有地址是不会被外网承认的，因此就无法上网，通常将这类私有地址作为家庭或者公司的局域网来使用，然后通过路由器进行上网，而实现这一功能的就是由路由器中的 NAT 协议来实现的，NAT 会将局域网内的主机 IP 和端口号替换成路由器的公网 IP 和端口号，然后再将数据发送给目标主机或服务器，因此在效果上来看就好像是拥有公网 IP 的路由器发出的数据一样，从了实现了内网向外网的穿透，这么做的好处是可以使 IP 地址（私有地址）得到最大程度的复用，有效缓解 IP 资源的枯竭。另外 NAT 的一个好处是，它可以隐藏局域网内的联网设备，因为对于外网设备来说他们看到的只是路由器的公网 IP，其内部局域网内的主机他们是察觉不到的，因此可以提高网络的安全性。

大家不要将 NAT 和 DHCP 这两个概念混淆在一起，就以家庭网络为例，NAT 功能和 DHCP 功能都由路由器提供，路由器通过 DHCP 可以为每一个刚接入到该网络的设备按照某种策略随机分配一个局域网 IP，如：192.168.1.23，此时该设备想要向外网发送数据，路由器就需要通过 NAT 将数据包中的局域网 IP 和端口号转换成公网的 IP 和端口号实现外网访问。

如果想将路由器的某一个端口接收到的数据映射到指定的该局域网内的主机上，我们可以设置路由器的端口映射来满足要求，但是现在很多运营商为用户分配的动态 IP 也都是局域网，也就是说在我们路由器的上一层还有一台路由器，除非我们可以更改上一层路由器的端口映射，否则映射就会失败。另外外网 IP 是动态的，因此每次开机的 IP 地址都可能不同，如果将我们的计算机作为服务器（比如个人论坛或者网页）的话，那么在用户每次访问时就需要不停的更改 IP，显然不切实际。因此可以选择一些能否提供内网穿透的服务商来为我们提供内网穿透服务，比如作者常用的就是花生壳，大家可以免费注册和使用，对于开发调试来说已经足够了，而且会为用户分配一个域名（后面讲解），之后设置内网和外网的映射 IP 和端口号即可：

图 33.8　花生壳映射说明

例如我希望用自己的电脑作为物联网的服务器，来接收各个设备发送过来的数据，则根据上图配置，我们的物联网设备连接的服务器地址和端口号是 18h212253o.iok.la:22958，而作为服务器的（本台电脑）地址和端口号要设置为 192.168.31.238:5468，当设备向外网 IP 发送数据时就会将这些数据映射到服务器主机上了。

花生壳具体的操作流程如下：

1. 注册花生壳
2. 下载花生壳客户端
3. 点击客户端“内网穿透”，跳转到映射设置网页

4. 根据实际需求指定要映射的 IP 和端口号

33.7 DNS 和域名（IP 绰号）

虽然使用四段点分十进制来表达 IP 地址相对于二进制表示来说更加直观一些，但是依然不能表达特殊的人类可以理解的含义，因此就产生了域名（也叫网址），域名可以理解为一个 IP 地址的绰号，这个绰号更符合人类的记忆方式，这一点类似于电话本中的联系人，联系人的手机号码对应的就是 IP 地址，联系人的备注对应的就是域名，例如 www.fengmeitech.club（域名）就比 101.200.130.235（IP）更好记，我们可以在 Windows 命令行中输入 nslookup “域名”，如 nslookup fengmeitech.club 来解析域名对应 IP 地址。下图是测试结果：

```
C:\Users\ZX>nslookup www.fengmeitech.club
服务器:  XiaoQiang
Address:  192.168.31.1

非权威应答:
名称:    www.fengmeitech.club
Address:  101.200.130.235
```

图 33.9 DNS 命令

当一台计算机要访问指定域名的主机或者服务器，如访问 www.fengmeitech.club 时，就需要解析该域名对应的 IP 地址，因为最终的访问对象就是 IP 地址，这时就要用到 DNS（Domain Name System，域名解析系统），当需要进行域名解析时，计算机会向 DNS 服务器发送请求解析指令，该指令是通过 UDP 协议实现，这样的速度更快，DNS 服务器上存储的是 IP 和域名的对应关系表，如果查到域名后将返回域名对应 IP 地址给我们的计算机，此时计算机开始访问目标主机或者服务器。

更详细的流程是：当计算机要访问一个域名，此时它先从本地缓存查找是否存在这个域名，如果有，就把本地保存的域名对应 IP 提取出来然后直接访问，如果没有则会向你的网络提供商（ISP，如：网通，电信等）的 DNS 服务器发出查找指令，如果 ISP 的 DNS 服务器也没有这个域名，则 ISP DNS 服务器会向其他 DNS 服务器（根服务器，顶级服务器等）发送解析指令，当这些 DNS 服务器解析之后找到了这个域名对应的 IP，就会将这个 IP 传给 ISP 的 DNS 服务器，ISP 的 DNS 服务器再将找到了这个 IP 返回给你的电脑，同时 ISP 的 DNS 服务器会将这个域名 IP 对应关系缓存起来，不用下次再进行查找，到此整个的域名解析就完成了。

乐鑫 ESP8266 官方 AT 指令为我们提供了域名解析指令（AT+CIPDOMAIN="域名"），效果如下：

图 33.10　ESP8266 域名解析 AT 指令

接下来讲解一下域名的分类，域名分类可以很好的将域名进行管理，通过查看域名可以快速获悉域名性质，所属上级域名等信息。域名分类采用了树状图的形式进行分类，下面就以作者母校——沈阳化工大学官网（http://www.syuct.edu.cn/）为例进行讲解。

在最顶层的根域名服务器（无名称，可以用.表示，我们平时会忽略这个.）下面的是顶级域名，接着是二级域名，三级域名……主机名，顶级域名按照性质分为两种，一个是国家顶级域名（如：中国（.cn），美国（.us），英国（.uk）等等），另一个是国际顶级域名（如：营利组织性质的.com，非盈利组织的.org），二级域名由国家划分，按照性质分为两种，一种是按照地理位置划分的各省域名，如 ln.（辽宁）bj.（北京）以及按照职能性质分的 gov.（政府），edu.（教育）等。下图是以沈阳化工大学官网为例绘制的树状图：

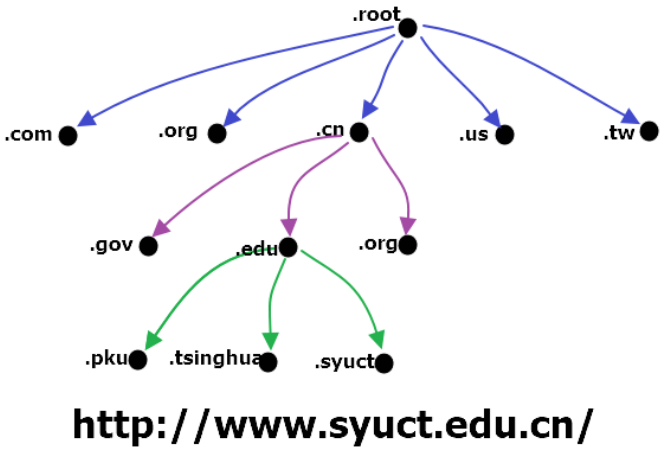


图 33.11　树形域名分类

可以看到域名从左到右依次为三级域名-二级域名-顶级域名。

33.8 TCP 和 UDP

前面做了这么多的铺垫主要是为了讲解 TCP 和 UDP 并对 TCP/IP 协议有一个清晰的认识。在物联网应用中，绝大部分的硬件设备和服务器的通信方式都是基于 TCP 或者 UDP 的（少数使用 HTTP），一般情况下，TCP 应用的比 UDP 要多一些。下面来讲解一下 TCP 和 UDP 的原理、特点以及区别，我们目的并不是学会这两个协议是如何实现的，而是要知道他们的大致机理并学会应用。

33.8.1 TCP 协议

TCP（Transmission Control Protocol，传输控制协议）是一种面向连接的、安全可靠的、基于字节流的传输层通信协议，在进行 TCP 传输时，要有一方作为服务器端（Server），一方作为客户端（Client），服务端用于监听来自其他客户端发来数据，一个服务端可以同时监听多个客户端，因此物联网应用中，硬件设备作客户端，服务器作为服务端。

一个 TCP 的客户端要和服务器端进行通信，客户端就必须先指定服务器端的 IP 地址和端口号，然后通过 3 次握手建立连接，如下图：

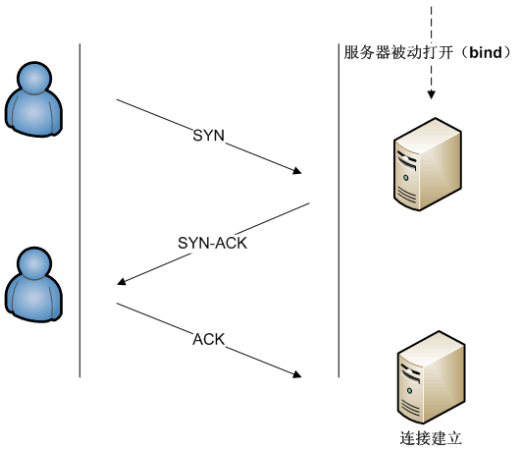


图 33.12　TCP 握手流程

当握手成功后，客户端和服务器端就正式建立了连接，此时就可以互相传输数据了，通常我们还会人为的对这些数据进行标识和加密，目的是保证数据传输安全，通过这些标识可以让服务器来区分该数据消息来自于哪个用户的哪台设备，加密则是防止物联网设备被恶意攻击，对于能够产生物理影响的物联网设备来说，安全是最重要的，否则设备被攻击后，攻击者会恶意控制家中设备，比如打开煤气一段时间后点燃，其后果不堪设想。

前面介绍了 TCP 的两个阶段：握手建立连接-数据收发，最后一个是断开连接，通常由客户端发起，断开连接要经过 4 次确认，如下图所示：

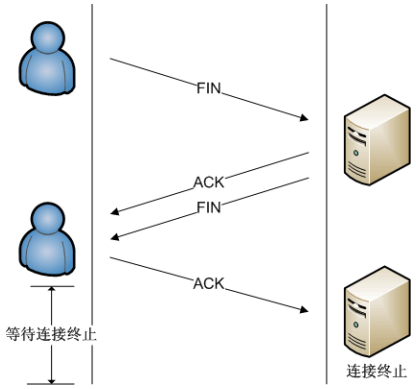


图 33.13　TCP 断开流程

从上面来看，TCP 类似于平时拨打电话，如果想要正常通信，就必须先向指定的目标拨号，拨通并建立连接后开始通话，通话结束后由一方断开连接。

到这里就已经介绍完了 TCP 的三个通信过程，另外我们可能都听说过 TCP 长连接和 TCP 短连接，TCP 长连接是指客户端和服务端长时间保持连接，这种情况在客户端连接数量很多时会增加服务器的负担，但是可以保证数据能够实时的传输。另外一种称为 TCP 短连接，即通信完之后立即断开，比如常见的 HTTP 协议，因为当客户端请求完数据之后，就不在需要继续保持 TCP 连接了，因此可以断开，降低服务器的工作负担。

在物联网应用中通常使用的都是 TCP 长连接，对于大多数物联网应用来说对传感器数据采集的实时性要求不高，因此间隔都比较长，比如快则几十秒，慢则几小时，同时这样也可以减轻服务器的处理负担，但是为了保证双方连接有效，通常物联网设备会每隔几秒或者更长的时间发送一个简短的数据包给服务器，证明自己还在，我们称这种数据包为心跳包，如果服务器长时间没有收到某一个设备发来的心跳包，则认为其断开连接，就会将和它的 TCP 连接断开，并释放资源，同时在用户界面显示该设备异常。

33.8.2 UDP 协议

UDP（User Datagram Protocol，用户数据报协议）是一种简单的面向数据报的传输层协议，它与 TCP 的传输不同，双方不存在所谓的客户端和服务端，只要一方知道另一方的 IP 地址和端口号，就可以直接建立连接并发送数据，并且也没有 TCP 一系列的传输安全措施，比如丢包重传和拥塞控制等，因此相对于 TCP 来说，UDP 的传输是不安全的，可能会出现丢包的现象，但正因为没有了握手，数据传输安全等处理，使得 UDP 的传输速度更快，UDP 通常应用在允许丢包但要保证通信速度的场合，比如流媒体。而像文件传输就需要使用 TCP，因为文件传输的前提是保证数据的完整。但是 TCP 和 UDP 通信速度的区别我们其实是察觉不到的，只不过都是相对而言，同样，UDP 丢包率也没有想象中的那么严重，像大家平时用的 QQ 都是通过 UDP 来进行传输的，所以完全可以放心的将 UDP 应用到物联网项目中，只不过使用 UDP 传输缺少一种连接上的控制。

到这里 TCP/IP 的基础知识就讲完了，关于这部分只要求大家了解它的用途就可以了，如果想要继续深入学习 TCP/IP 的原理，推荐大家阅读《图解 TCP/IP》。

第三十四章 单片机实现 TCP/UDP 通信

34.1 项目要求

1. 使用 STM32 的 UART2 和 ESP8266 模块进行 AT 通信，实现连接无线路由器功能以及 TCP、UDP 通信；
2. STM32 采集传感器数据并定时上传至服务器；
3. 根据服务器下发的指令控制两个继电器的工作状态。

注：本章用到核心板+扩展板，对应例程 27。

34.2 原理讲解

在第三十二章已经将常用的 AT 指令向大家做了详细的介绍，使用这些常用的指令就可以实现单片机的联网功能了。

本章主要实现 TCP 和 UDP 的通信，并对其进行数据传输调试来满足项目要求。在调试 TCP 时，我们使用合宙提供的在线 TCP 调试，其调试界面如下：

图 34.1 合宙在线 TCP 调试工具

注意：服务器的端口号是随机的，在每次刷新网页的时候都会变化。像这样的在线 TCP 调试界面还有安信可的在线透传调试，如下图：

图 34.2　安信可在线 TCP 调试工具

调试 UDP 时，则直接使用网络调试助手的 UDP 功能进行局域网通信。

在打通 TCP 和 UDP 通信之后，连接云平台就非常简单了，只要按照云平台的数据格式的要求通信即可。

34.3 程序讲解

实现 TCP 和 UDP 通信主要围绕连接热点，连接服务器，向服务器发送数据和接收服务器传回来的信息这四个大的方面来展开。

实现设备连接远程服务器的流程为：

1. 设置 ESP8266 连接 AP 热点使其正常上网
2. 设置 ESP8266 连接远程/局域网 TCP 或 UDP 服务器
3. 将采集到的传感器数据发送到服务器
4. 根据服务器传输过来的控制指令做出相应动作，如控制 LED 和继电器

- 查找指定字符串函数

```
/**
 * 功能：查找字符串中是否包含另一个字符串
 * 参数：
 *       dest：待查找目标字符串
 *       src：待查找内容
 *       retry_cn:查询超时时间
 * 返回值：查找结果，非0为查找成功,0为失败
 * 说明：
 *       当发出一个AT指令后，需要一段时间等待ESP8266回复,因此就需要等待一段时间,
 *       这个时间通常是几百ms(除了连接服务器和AP指令),本程序一般指令通常等待
 *       2S,耗时的连接AP和服务器的设置等待为8S,其实花不了那么多时间,但如果发生超时
 *       就一定可以判断是通信问题
 */
static u8 findStr(u8* dest,u8* src,u16 retry_cn)
{
    u16 retry = retry_cn;                    //超时时间
    u8 result_flag = 0;                      //查找结果

    while(strstr(dest,src)==0 && --retry!=0)//等待串口接收完毕或超时退出
    {
        Delay_ms(10);
    }

   if(retry==0)                              //如果超时则有问题,此时返回0
   {
      return 0;
   }
   result_flag = 1;                          //执行到这里说明一切正常，表示查找成功

    if(result_flag)
    {

        return 1;
    }else
    {
        return 0;
    }
}
```

图 34.3　查找指定字符串函数

该函数的核心是使用在标准 C 库文件 string.h 中提供的查找字符串函数——strstr(char* str1,char* str2);其含义是在字符串 str1 中查找是否含有 str2，如果有返回 str2 首个元素的地址，如果没有返回 NULL(0)。我们通常根据函数返回值是否为非 0 来判断两个字符串是否是包含关系，下面程序是使用 Dev C++编译的功能展示函数：

```
#include<stdio.h>
#include<string.h>

void fun(char* str)
{
    printf("%d\n",strlen(str));
}
int main(void)
{
    char str1[] = "www.fengmeitech.club";
    char str2[] = "fengmei";


    printf("%s\n",strstr(str1,str2));
    printf("%d\n",strstr(str1,str2));
    printf("%d\n",strstr(str1,"hello"));
    return 0;
}
```

```
C:\Users\ZX\Documents\C测试\未命名2.exe
fengmeitech.club
6487604
0

--------------------------------
Process exited after 0.01176 seconds with return va
请按任意键继续. . .
```

图 34.4　strstr 函数用法

本章的 findStr 函数根据 strstr 库函数做了扩展，添加了查询超时时间。当一条 AT 指令发送给 ESP8266 后，ESP8266 大约会在几百 ms 左右打印响应结果，因此在发送完 AT 指令就判断 UART 接收的数据是无意义的，起码要等上一段时间，如果等待的这段时间足够长但没有收到响应，则可以说明设置无效或者 ESP8266 有问题。

- ESP8266 握手函数

```
/**
 * 功能：检查ESP8266是否正常
 * 参数：None
 * 返回值：ESP8266返回状态
 *         非0 ESP8266正常
 *         0 ESP8266有问题
 */
u8 checkESP8266(void)
{
    memset(RXBuffer,0,RXBUFFER_LEN); //清空接收缓冲

    sendString(USART2,"AT\r\n");      //发送AT握手指令

    if(findStr(RXBuffer,"OK",200)!=0)//ESP8266正常
    {
        return 1;
    }else                            //ESP8266不正常
    {
        return 0;
    }
}
```

图 34.5　ESP8266 握手函数

通常会在程序中使用 AT\r\n 来检查此时 ESP8266 的状态，如果回复 OK，则说明 ESP8266 存在且正常，如果无回复则表示有问题，该函数通常用在 ESP8266 初始化函数中。

- ESP8266 初始化函数

```
/**
 * 功能：初始化ESP8266
 * 参数：None
 * 返回值：初始化结果，非0为初始化成功,0为失败
 */
u8 initESP8266(void)
{
    sendString(USART2,"+++");           //退出透传
    Delay_ms(500);
    sendString(USART2,"AT+RST\r\n");    //重启ESP8266
    Delay_ms(500);
    if(checkESP8266()==0)               //使用AT指令检查ESP8266是否存在
    {
        return 0;
    }

    memset(RXBuffer,0,RXBUFFER_LEN);    //清空接收缓冲
    sendString(USART2,"ATE0\r\n");      //关闭回显
    if(findStr(RXBuffer,"OK",200)==0)   //设置不成功
    {
        return 0;
    }

    return 1;                           //设置成功

}
```

图 34.6　ESP8266 初始化函数

退出透传模式并重启，之后主要是检查 ESP8266 是否正常，如果正常则关闭回显。在程序开始时退出透传的原因是，有时我们复位了 STM32 但是此时 ESP8266 仍处于 STM32 复位之前的状态，如果此时正是透传模式，则任何的 AT 指令都被认为是要发送给服务器的数据，所以为了保证函数设置的有效性就必须在使用 AT 指令之前退出透传模式，并重启 ESP8266，重启可以断开所有的服务器连接，让单片机和 ESP8266 状态同步。关闭回显后，我们发送的 AT 指令会不会再重复发过来，有助于判断接收数据并节省接收缓冲区。

- 恢复出厂设置

```
/**
 * 功能：恢复出厂设置
 * 参数：None
 * 返回值：None
 * 说明:此时ESP8266中的用户设置将全部丢失回复成出厂状态
 */
void restoreESP8266(void)
{
    sendString(USART2,"+++");           //退出透传
    Delay_ms(500);
    sendString(USART2,"AT+RESTORE\r\n");//恢复出厂
    NVIC_SystemReset();                 //同时重启单片机
}
```

图 34.7　ESP8266 恢复出厂函数

调用该函数后 ESP8266 会删除所有用户设置并重启，这时的 ESP8266 状态和刚刷入固

件时的状态一致，另外为了保证和STM32运行状态同步，STM32也需要使用NVIC_SystemReset函数进行重启。

- 连接 AP（无线路由器）

```
122  * 功能：连接热点
123  * 参数：
124  *         ssid:热点名
125  *         pwd:热点密码
126  * 返回值：
127  *         连接结果，非0连接成功，0连接失败
128  * 说明：
129  *         失败的原因有以下几种(UART通信和ESP8266正常情况下)
130  *         1. WIFI名和密码不正确
131  *         2. 路由器连接设备太多，未能给ESP8266分配IP
132  */
133 u8 connectAP(u8* ssid,u8* pwd)
134 {
135     memset(RXBuffer,0,RXBUFFER_LEN);
136     sendString(USART2,"AT+CWMODE?\r\n");                    //查询此时WIFI工作模式
137     if(findStr(RXBuffer,"CWMODE:1",200)==0)                 //如果此时不是MODE1模式，即不是STATION模式
138     {
139         memset(RXBuffer,0,RXBUFFER_LEN);
140         sendString(USART2,"AT+CWMODE_CUR=1\r\n");           //设置为STATION模式
141         if(findStr(RXBuffer,"OK",200)==0)
142         {
143             return 0;
144         }
145     }
146 
147     memset(TXBuffer,0,RXBUFFER_LEN);                        //清空发送缓冲
148     memset(RXBuffer,0,RXBUFFER_LEN);                        //清空接收缓冲
149     sprintf(TXBuffer,"AT+CWJAP_CUR=\"%s\",\"%s\"\r\n",ssid,pwd);//连接目标AP
150     sendString(USART2,TXBuffer);
151     if(findStr(RXBuffer,"OK",800)!=0)                       //连接成功且分配到IP
152     {
153         return 1;
154     }
155 }
```

图 34.8　连接热点函数

ESP8266 开机后默认状态是 AP 模式，所以需要将其设置为 STATION 模式才可以接入 AP 热点，随后使用连接 AP 指令连接到指定路由器。如果连接成功会返回 1，否则返回 0。

- 使用指定协议连接到指定服务器

```c
/**
 * 功能：使用指定协议(TCP/UDP)连接到服务器
 * 参数：
 *         mode:协议类型 "TCP","UDP"
 *         ip:目标服务器IP
 *         port:目标是服务器端口号
 * 返回值：
 *         连接结果,非0连接成功,0连接失败
 * 说明：
 *         失败的原因有以下几种(UART通信和ESP8266正常情况下)
 *         1. 远程服务器IP和端口号有误
 *         2. 未连接AP
 *         3. 服务器端禁止添加(一般不会发生)
 */
u8 connectServer(u8* mode,u8* ip,u16 port)
{
    memset(RXBuffer,0,RXBUFFER_LEN);
    memset(TXBuffer,0,RXBUFFER_LEN);

    sendString(USART2,"+++");                    //多次连接需退出透传
    Delay_ms(500);
    /*格式化待发送AT指令*/
    sprintf(TXBuffer,"AT+CIPSTART=\"%s\",\"%s\",%d\r\n",mode,ip,port);
    sendString(USART2,TXBuffer);
    if(findStr(RXBuffer,"CONNECT",800)!=0)
    {
        memset(RXBuffer,0,RXBUFFER_LEN);
        sendString(USART2,"AT+CIPMODE=1\r\n");  //设置为透传模式
        if(findStr(RXBuffer,"OK",200)!=0)
        {
            memset(RXBuffer,0,RXBUFFER_LEN);
            sendString(USART2,"AT+CIPSEND\r\n");//开始处于透传发送状态
            if(findStr(RXBuffer,">",200)!=0)
            {
                return 1;
            }else
            {
                return 0;
            }
        }else
        {
            return 0;
        }

    }else
    {
        return 0;
    }
}
```

图 34.9　连接服务器函数

使用 AT+CIPSTART 指令建立一个 TCP 或者 UDP 链接，使用 AT+CIPMODE=1 开启透传模式并进入透传发送状态，此后串口发送的任何数据（除了+++）都会原封不动的送到服务器。

这里再说一下头文件内容：

```c
/*连接AP宏定义*/
#define SSID "Xiaomi"
#define PWD  "fengmei_123"

/*连接服务器宏定义*/
#define TCP "TCP"
#define UDP "UDP"
#define IP  "192.168.31.238"
#define PORT 22958

/*发送接收缓冲区长度宏定义*/
#define TXBUFFER_LEN 50
#define RXBUFFER_LEN 30
```

图 34.10 相关宏定义

开发板连接的热点名、密码以及连接服务器的协议和服务器 IP 端口号都可以在此处更改，如果使用本套程序，可以根据数据量的大小来修改发送和接收缓冲区的大小。

注意：大家要根据自己的实际情况来更改热点和服务器参数。

- 断开与服务器的连接

```
/**
 * 功能：主动和服务器断开连接
 * 参数：None
 * 返回值：
 *          连接结果,非0断开成功,0断开失败
 */
u8 disconnectServer(void)
{
    sendString(USART2,"+++");              //退出透传
    Delay_ms(500);
    memset(RXBuffer,0,RXBUFFER_LEN);
    sendString(USART2,"AT+CIPCLOSE\r\n");//关闭链接

    if(findStr(RXBuffer,"CLOSED",200)!=0)//操作成功,和服务器成功断开
    {
        return 1;
    }else
    {
        return 0;
    }
}
```

图 34.11 断开服务器连接函数

调用该函数可以使设备主动和服务器断开。

- 发送数据到服务器

```
/**
 * 功能：透传模式下的数据发送函数
 * 参数：
 *       buffer:待发送数据
 * 返回值：None
 */
void sendBuffertoServer(u8* buffer)
{
    memset(RXBuffer,0,RXBUFFER_LEN);
    sendString(USART2,buffer);
}
```

图 34.12 发送函数

设备和服务器的通信都是通过该函数来完成的，UART2 发送的任何数据都会发送给服务器。

- 处理接收数据

```
238  /**
239   * 功能：处理服务器发回来的控制指令
240   * 参数：None
241   * 返回值：None
242   */
243  void processServerBuffer(void)
244  {
245      u8 i = 0;
246
247      /*LED状态控制*/
248      if(strstr(RXBuffer,"LED_ON"))
249      {
250          ++i;
251          openLED();
252      }else if(strstr(RXBuffer,"LED_OFF"))
253      {
254          ++i;
255          closeLED();
256      }else
257      {
258
259      }
260
261      /*继电器1状态控制*/
262      if(strstr(RXBuffer,"RELAY1_CLOSE"))
263      {
264          ++i;
265          setRelay(RELAY1,RELAY_CLOSE);
266      }else if(strstr(RXBuffer,"RELAY1_OPEN"))
267      {
```

```
268             ++i;
269             setRelay(RELAY1,RELAY_OPEN);
270         }else
271         {
272 
273         }
274 
275         /*继电器2状态控制*/
276         if(strstr(RXBuffer,"RELAY2_CLOSE"))
277         {
278             ++i;
279             setRelay(RELAY2,RELAY_CLOSE);
280         }else if(strstr(RXBuffer,"RELAY2_OPEN"))
281         {
282             ++i;
283             setRelay(RELAY2,RELAY_OPEN);
284         }else
285         {
286 
287         }
288 
289         /*只在接收控制指令时才清空,这样可避免接收AT指令时导致失败*/
290         if(i!=0)
291         {
292             memset(RXBuffer,0,RXBUFFER_LEN);
293         }
294     }
```

图 34.13 接收数据处理函数

该函数使用前面讲到的 strstr 函数来判断服务器发送过来的数据是否包含控制信息，这些信息包括:LED_ON、LED_OFF、RELAY1_CLOSE、RELAY1_OPEN、RELAY2_CLOSE、RELAY2_OPEN，另外为了保证实时性，该函数应该在串口接收空闲中断中被调用，下图是串口 2 的中断服务函数：

```
125 /**
126  * 功能：串口2中断服务函数
127  * 参数：None
128  * 返回值：None
129  */
130 void USART2_IRQHandler(void)
131 {
132     static u8 i = 0;
133 
134     if(USART_GetITStatus(USART2, USART_IT_RXNE)) //判断接收数据寄存器是否有数据
135     {
136         RXBuffer[i++] = USART_ReceiveData(USART2);
137         if(i==RXBUFFER_LEN)                     //超出接收缓冲范围,可能的情形是ESP8266复位,为防止溢出必须设置索引
138         {
139             i = RXBUFFER_LEN-1;
140         }
141     }
142 
143     if(USART_GetITStatus(USART2, USART_IT_IDLE))
144     {
145         USART_ReceiveData(USART2);              //读一次UART可以清除空闲标志位
146         i = 0;
147         processServerBuffer();
148     }
149 }
```

图 34.14 串口 2 中断服务函数

接收非空中断用于将接收的单字节数据放到接收缓冲区，接收空闲中断用于在接收完成之后，对接收的所有数据进行处理，并复位接收缓冲区数组的索引为 0。

- 主函数

```
26  int main(void)
27  {
28      u8 i = 0;
29      u8 key_value;
30
31      u16 DHT11_data; //存储DHT11传感器采集数据
32      u16 ADC_data;   //存储ADC数据
33
34      /*初始化各外设*/
35      initSysTick();
36      initADC();
37      initUART();
38      initUART2();
39      initLED();
40      initKey();
41      initRelay();
42      initDHT11();
43      initIIC();
44      initOLED();
45      initNVIC(NVIC_PriorityGroup_2);//开启UART2的接收和空闲中断
46
47      formatScreen(0x00);//清屏
48
49      /*打印ESP8266启动信息到OLED*/
50      if(initESP8266()!=0)
51      {
52          showString(0,0,"init ok!",FONT_16_EN);
53      }else
54      {
55          showString(0,0,"init error!",FONT_16_EN);
```

```
    }
    if(connectAP(SSID,PWD)!=0)
    {
        showString(0,2,"conn ap ok!",FONT_16_EN);
    }else
    {
        showString(0,2,"con ap error!",FONT_16_EN);
    }
    if(connectServer(TCP,IP,PORT)!=0)
    {
        showString(0,4,"con server ok!",FONT_16_EN);
    }else
    {
        showString(0,4,"con server error!",FONT_16_EN);
    }

    Delay_ms(1000);//让数据保持一段时间
    formatScreen(0x00);
    showCNString(32,0,"风媒电子",FONT_16_CN);
    showString(0,2,"Hum :",FONT_16_EN);
    showString(0,4,"Temp:",FONT_16_EN);
    showString(0,6,"Lux :",FONT_16_EN);
    while (1)
    {
        if(++i>=200)        //每隔2S左右采集一次 建议采集间隔最小不要小于2S
        {
            i = 0;

            /*采集传感器数据*/
            DHT11_data = readDHT11();
            ADC_data = getConvValueAve(5,1000);

            /*输出传感器数据到服务器*/
            memset(TXBuffer,0,TXBUFFER_LEN);
            sprintf(TXBuffer,"Hum:%d,Temp:%d,Lux:%d\n",DHT11_data>>8,DHT11_data&0x00FF,ADC_data);
            sendBuffertoServer(TXBuffer);

            /*同时显示在OLED上*/
            showNumber(40,2,DHT11_data>>8,DEC,3,FONT_16_EN);
            showNumber(40,4,DHT11_data&0x00FF,DEC,3,FONT_16_EN);
            showNumber(40,6,ADC_data,DEC,4,FONT_16_EN);
        }

        key_value = getKeyValue(KEY_RELEASE);
        /*UP键按下恢复出厂*/
        if(key_value==KEY_UP)
        {
            restoreESP8266();
        }

        /*DOWN键按下断开TCP连接*/
        if(key_value==KEY_DOWN)
        {
            disconnectServer();
        }

        Delay_ms(10);//基础延时
    }
}
```

图 34.15 主函数

主函数实现的功能是：在初始化了各种外设之后，打印 ESP8266 的启动信息，这些信息来源包括初始化 ESP8266 的结果、连接 AP 的结果、连接服务器的结果，接着在主循环中每隔 2S 采集一次传感器数据并上传服务器同时更新 OLED，当 UP 键按下后恢复出厂，DOWN 键按下后断开 TCP 连接。此时在上位机输入任何一个控制指令可以观察结果，下面是运行效果：

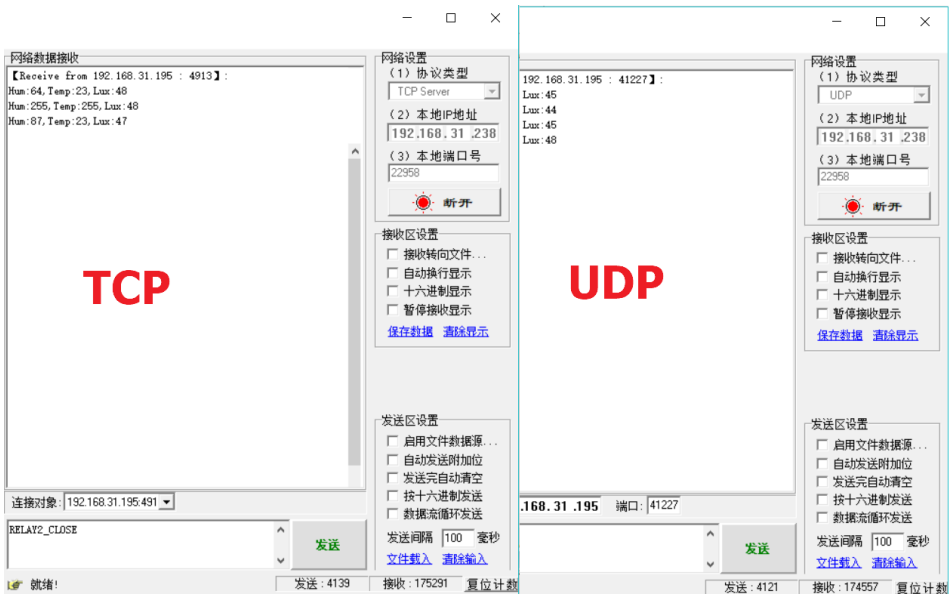


图 34.16 TCP/UDP 调试结果

这两种协议的切换只需要更改连接服务器函数的网络协议模式参数即可，在运行效果上 TCP 和 UDP 两者无区别。下图是使用 TCP 协议连接远程服务器的运行效果，这样我们可以摆脱局域网限制，进行设备的远程操控，此时我们的设备已经是一台物联网设备了：

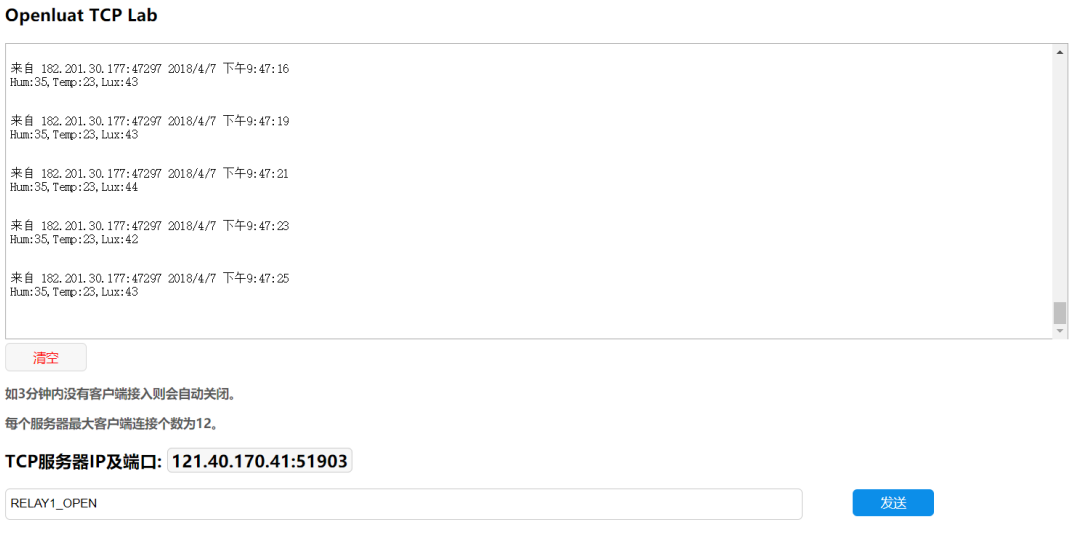


图 34.17 远程 TCP 服务器调试结果

第三十五章 单片机发送电子邮件

35.1 项目要求

使用单片机实现基于 TCP 协议的 SMTP（Simple Mail Transfer Protocol，简单邮件传输协议），并根据按下按键的键值向不同收件人发送不同内容的邮件，具体要求如下：

1. OLED 显示传感器采集到的值
2. UP 键按下时向邮箱 1 发送固定内容，邮件标题为：“HELLO EARTH MAN”,邮件正文为：“THE MESSAGE COMES FROM OUTSIDE THE EARTH,DON`T ANSWER!DON`T ANSWER!DON`T ANSWER!”
3. DOWN 键按下时向邮箱 2 发送可变内容，邮件标题为：“SENSOR”，邮件正文为：“Hum:xxx,Temp:xxx,Lux:xxxx.”，其中 x 代表传感器采集到的实时环境数据。

注：本章用到核心板+扩展板，对应例程 28。

35.2 原理讲解

电子邮箱是我们平时工作和生活经常用到的一种网络服务，大家对电子邮箱都不会感到陌生，在发送一封电子邮件时，需要做的是填写收件人的邮箱地址，邮件的标题，邮件的内容，然后点击发送即可。其中填写的邮箱地址类似于信件的信封，邮件的标题以及内容则对应的是信件本身的标题和正文，通常称邮件标题为邮件头，邮件内容为邮件体。以作者经常使用的 126 邮箱为例和传统纸质信件作对比，如下图：

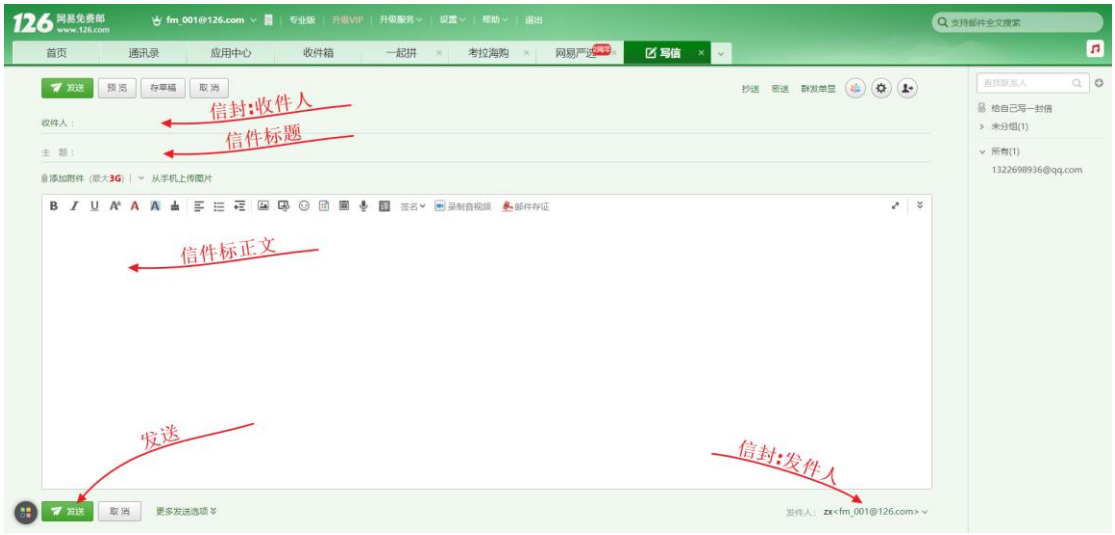


图 35.1 126 邮箱对比图

客户端的邮件编辑可以支持富文本操作，并插入图片音乐等，但是这些功能对于单片机紧张的硬件资源来说是很难实现的，因此我们只用单片机发送纯文本邮件。另外需要注意的是：如果使用第三方工具来发送邮件，则需要先设置开启 SMTP 功能，并使用生成的授权码

在第三方工具上登录邮箱，具体操作流程如下：

1. 开启 SMTP 服务（其他邮箱类似）

图 35.2 设置开启 SMTP 功能

2. 重新设置授权码

在点击开启 SMTP 服务后，页面会自动让用户设置用于第三方登录的授权码，授权码类似于邮箱的第二个密码，但是授权码和密码不能相同，我们使用单片机发送邮件自然属于第三方工具，因此需要授权码，否则使用原密码登录时，会提示：“您无权登录”。

在前面的章节已经提到过 SMTP，它位于 TCP/IP 协议的应用层，基于 TCP 协议实现，监听端口为 25，通常使用 SMTP 协议来实现邮件的发送（接收邮件使用 POP3 或 IMAP，二者区别点击链接），其实该协议非常简单，只要打通了设备到邮件服务器的 TCP 连接并按照 SMTP 的数据传输协议将内容发送给服务器即可。

本章就以网易的 126 邮箱为例，使用 126 邮箱向其他邮箱发送电子邮件。在发送邮件之前必须先要知道设备和邮箱服务器的通信流程：

1. ->使用 TCP Client 模式连接 126 邮箱服务器（smtp.126.com:25）；
2. <-服务器回复 220
3. ->发送 helo xx（xx 任意字符）和服务器打招呼标识身份
4. <-服务器回复 250
5. ->发送 auth login 请求登录邮箱
6. <-服务器回复 334 dXNlcm5hbWU6（username:的 BASE64 码），提醒输入用户名
7. ->发送 126 邮箱地址的 BASE64 编码
8. <- 服务器回复 334 UGFzc3dvcmQ6（password:的 BASE64 码），提醒输入密码
9. ->发送密码（授权码）的 BASE64 编码
10. <-服务器回复 235 表示登录成功
11. ->发送发信人邮箱地址（fm_001@126.com）（信封）

12. <-服务器回复 250 表示设置成功
13. ->发送收件人邮箱地址（1322698936@qq.com）（信封）
14. <-服务器回复 250 表示设置成功
15. ->发送 data 告知服务器此时开始输入信件内容（信件）
16. <-服务器回复 354，表示可以开始
17. ->发送发信人地址，收信人地址，邮件主题以及邮件内容，并以<CR><LF>.<CR><LF>.结尾，此时邮件会自动发送
18. <-服务器返回 250 表示发送成功
19. ->如果继续发信则接着步骤 15 向下进行，不发信则发送 QUIT 指令和服务器断开连接，关闭 SMTP 服务即可。

为了方便演示，作者使用 telnet 登录 smtp.126.com 来模拟 SMTP 并发送一封电子邮件。

先打开命令行（WIN+R->cmd->Endter），输入“telnet smtp.126.com 25”，然后回车，等到登录界面后开始键入 SMTP 发送流程的各个指令（不区分大小写），如下图：

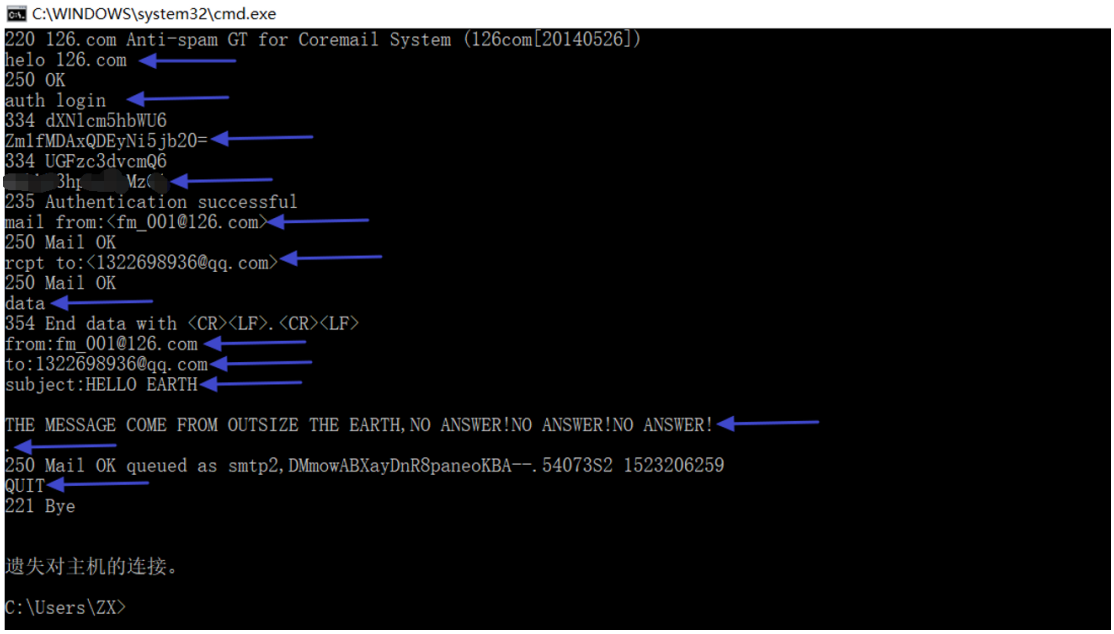


图 35.3 telnet 模拟 SMTP 结果

Windows 操作系统默认是没有开启 telnet 服务的，所以需要先开启，方法如下：

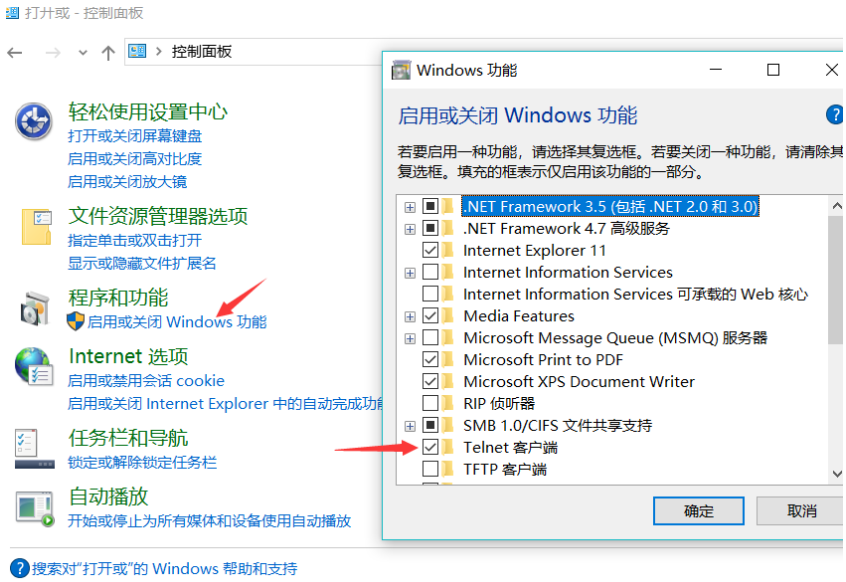


图 35.4　使能 telnet 功能

在命令行中蓝色箭头所指的是需要我们输入的地方，大家看不懂的可能就是第三和第四个箭头所指的那两串数字，其实它们分别是你的邮箱地址和邮箱登录授权码的 BASE64 编码结果，BASE64 编码就是将 3 个字节为一组的数据转换成 4 个字节的高两位为 0 即有效位数为 6 位的数据（3*8=4*6）。然后将所得的数据作为 BASE64 码表的索引，最后得到的码表对应值就是 BASE64 码，如果原数据不足 3 字节，则转换后的最后一个 6 位数据不够的位置用 0 填充，并使用=凑齐 4 个 BASE64 字节，示例如下。

原数据为： 10011010　　11010111　　10101010
转换后：　00100110 00101101 00011110 00101010

大家对 BASE64 感兴趣的话可以去网上查找 BASE64 具体的工作流程并且还有在线的 BASE64 转换工具。我们也向大家提供了高效的 BASE64 编解码函数供大家使用。

在邮件发送时有两点要注意：第一点是信件标题（subject，也可叫邮件头）和信件正文（content，也可叫邮件体）之间要留有一个空行，该空行用于区分邮件标题和正文；第二点就是输入完成正文时候要以\r\n.\r\n 结束，即在命令行中对应的就是：回车.回车，该句点不会包含在邮件中。

另外，在编程之前还要知道邮箱的各个响应代码是什么意思（如，250,334 等），下面是我们为大家整理的 SMTP 服务器相应代码（**错误响应码均以 4、5 开头，正确响应码均以 2、3 开头**）：

501 参数格式错误
502 命令不可实现
503 错误的命令序列，接收邮箱格式错误（例如 xx@dd,正确格式应该 xx@dd.es）
504 命令参数不可实现
211 系统状态或系统帮助响应
214 帮助信息
220 <domain>服务就绪
221 <domain>服务关闭

421 ＜domain＞服务未就绪，关闭传输信道
250 要求的邮件操作完成
251 用户非本地，将转发向＜forward-path＞
450 要求的邮件操作未完成，邮箱不可用
550 要求的邮件操作未完成，邮箱不可用
451 放弃要求的操作；处理过程中出错
551 用户非本地，请尝试＜forward-path＞
452 系统存储不足，要求的操作未执行
454 临时认证失败,可能账号被临时冻结
552 过量的存储分配，要求的操作未执行
553 邮箱名不可用，要求的操作未执行(有的服务器在邮箱账号被临时这样提示的)
354 开始邮件输入，以".″结束
554 操作失败

在调试时还要注意：如果用户频繁的向某个邮箱发送邮件，则发信端邮件服务器提供商和接收端邮件服务器提供商都会判定为非法或者垃圾邮件，此时会被禁封几十分钟，如果发送正常但没有接收到，则要去收件方垃圾箱中查找，如果发送不正常，则说明你发送的邮件被官方判定为垃圾邮件，此时会退信，关于退信代码可以参考网易声明。

通过上面的讲解，可以总结使用单片机实现 SMTP 的三个步骤：

1. 先让单片机联网
2. 使用 TCP Client 连接到邮箱服务器
3. 按照 SMTP 发送规则来发送数据。

35.3 程序讲解

- BASE64 编码函数

```
/**
 * 功能：对字符串进行BASE64编码
 * 参数：
 *      basestr:编码后存储缓冲指针
 *      str:待编码缓冲区指针
 * 返回值：None
 * 说明：
 *       对于传址处理的函数，一定要保证所传地址对应的
 *       空间足够大，才可以避免内存溢出。
 */
static void codeBase64(char* basestr,char* str)
{
    /*BASE64编码表*/
    const char Base64_table[] = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789+/";

    unsigned char len = strlen(str);
    unsigned char remain = len%3;
    unsigned char i,j=0;

    /*清空残留*/
    memset(basestr,0,strlen(basestr));
    /*每3个字节为一组进行切分*/
    for(i=0;i<len;i+=3)
    {
        /*不够3字节时进行补充*/
        if(len-i == remain)
        {
            /*剩一个字节(8bits)时,可以拆分成两个6bits的字节,
             *第二个6bits字节低4bits不够用0填充
             *剩两个字节时,可以拆分成3个6bits的字节,第三个6bits字节
             *低4bits不够用0填充*/
            if(remain==1)
            {
                basestr[j++] = Base64_table[str[i]>>2];
                basestr[j++] = Base64_table[(str[i]&0x03)<<4 | 0x00];
                basestr[j++] = '=';
                basestr[j] = '=';
            }else if(remain==2)
            {
                basestr[j++] = Base64_table[str[i]>>2];
                basestr[j++] = Base64_table[(str[i]&0x03)<<4 | (str[i+1]&0xF0)>>4];
                basestr[j++] = Base64_table[(str[i+1]&0x0F)<<2 | 0x00];
                basestr[j] = '=';
            }else
            {

            }
            break;
        }

        /*够3字节将3字节拆分成4个有效位数为6位的字节,并查表赋值*/
        basestr[j++] = Base64_table[str[i]>>2];
        basestr[j++] = Base64_table[(str[i]&0x03)<<4 | (str[i+1]&0xF0)>>4];
        basestr[j++] = Base64_table[(str[i+1]&0x0F)<<2 | (str[i+2]&0xC0)>>6];
        basestr[j++] = Base64_table[str[i+2]&0x3F];

    }
}
```

图 35.5　BASE64 编码

大家不必纠结于该函数如何实现，只要知道如何使用即可。

- 邮件发送函数

```
/**
 * 功能：给指定邮箱发送一封电子邮件
 * 参数：
 *      email_addr:收件人邮箱地址
 *      subject:邮件标题
 *      content：邮件正文
 * 返回值：非0表示发送成功,0表示发送失败
 */
unsigned char sendEmail(char* email_addr,char* subject,char* content)
{
    unsigned char smtp_sta = SERVER_NOLINK; //邮件发送状态
    unsigned char send_result = 0;          //发送结果

    /*格式化发送命令*/
    sprintf(Envelope_From,"mail from:<%s>\r\n",EMAIL_USER);
    sprintf(Envelope_To,"rcpt to:<%s>\r\n",email_addr);
    sprintf(Header_From,"from:%s\r\n",EMAIL_USER);
    sprintf(Header_To,"to:%s\r\n",email_addr);
    sprintf(Header_Subject,"subject:%s\r\n\r\n",subject);

    /*发送状态机*/
    switch(smtp_sta)
    {
        /*连接服务器*/
        case SERVER_NOLINK:
                                connectServer(TCP,SMTP_SERVER_IP,SMTP_SERVER_PORT);
                                if(findStr(RXBuffer,"220",200)==1)
                                {
                                    smtp_sta=SERVER_HELO;
                                }else
                                {
                                    smtp_sta = SERVER_NOLINK;
                                };
        /*和服务器打招呼,标明自己身份*/
        case SERVER_HELO  :
                                sendBuffertoServer("helo 126.com\r\n");
                                if(findStr(RXBuffer,"250",200)==1)
                                {
                                    smtp_sta=SERVER_LOGIN;
                                }else
                                {
                                    smtp_sta = SERVER_QUIT;
                                };

        /*请求登录*/
        case SERVER_LOGIN :
                                sendBuffertoServer("auth login\r\n");
                                if(findStr(RXBuffer,"334",200)==1)
                                {
                                    smtp_sta=SERVER_IN_NAME;
                                }else
                                {
                                    smtp_sta = SERVER_QUIT;
                                };

        /*输入用户名*/
        case SERVER_IN_NAME :
                                codeBase64(Base64,EMAIL_USER);
                                strcat(Base64,"\r\n");
                                sendBuffertoServer(Base64);
```

```
232                                 if(findStr(RXBuffer,"334",200)==1)
233                                 {
234                                     smtp_sta=SERVER_IN_PWD;
235                                 }else
236                                 {
237                                     smtp_sta = SERVER_QUIT;
238                                 };
239             /*输入密码*/
240             case SERVER_IN_PWD  :
241                                 codeBase64(Base64,EMAIL_PWD);
242                                 strcat(Base64,"\r\n");
243                                 sendBuffertoServer(Base64);
244                                 if(findStr(RXBuffer,"235",200)==1)
245                                 {
246                                     printf("\r\nLogin success!");
247                                     smtp_sta=SERVER_MIAL_FROM;
248                                 }else
249                                 {
250                                     printf("\r\nLogin fail!");
251                                     printf("\r\nLogin error message is:%s",RXBuffer);
252                                     smtp_sta = SERVER_QUIT;
253                                 };
254
255             /*输入信封发件人*/
256             case SERVER_MIAL_FROM  :
257                                 sendBuffertoServer(Envelope_From);
258                                 if(findStr(RXBuffer,"250",200)==1)
259                                 {
260                                     smtp_sta=SERVER_RCPT_TO;
261                                 }else
262                                 {
263                                     smtp_sta = SERVER_QUIT;
264                                 };
265             /*输入信封收件人*/
266             case SERVER_RCPT_TO   :
267                                 sendBuffertoServer(Envelope_To);
268                                 if(findStr(RXBuffer,"250",200)==1)
269                                 {
270                                     smtp_sta=SERVER_DATA;
271                                 }else
272                                 {
273                                     smtp_sta = SERVER_QUIT;
274                                 };
275             /*通知服务器开始输入邮件内容*/
276             case SERVER_DATA    :
277                                 sendBuffertoServer("data\r\n");
278                                 if(findStr(RXBuffer,"354",200)==1)
279                                 {
280                                     smtp_sta=SERVER_CONTENT;
281                                 }else
282                                 {
283                                     smtp_sta = SERVER_QUIT;
284                                 };
285             /*输入内容*/
286             case SERVER_CONTENT   :
287                                 sendBuffertoServer(Header_From);
288                                 sendBuffertoServer(Header_To);
289                                 sendBuffertoServer(Header_Subject);
290                                 sendBuffertoServer(content);
291                                 sendBuffertoServer("\r\n.\r\n");
292                                 if(findStr(RXBuffer,"250",200)==1)
293                                 {
```

```
294                                     send_result = 1;
295                                     smtp_sta=SERVER_QUIT;
296                                     printf("\r\nEmail send success!");
297                                 }else
298                                 {
299                                     printf("\r\nEmail send fail!");
300                                     printf("\r\nSend error message is:%s",RXBuffer);
301                                     smtp_sta = SERVER_QUIT;
302                                 };
303         /*结束和远程服务器的TCP连接*/
304         case SERVER_QUIT    :
305                                 sendBuffertoServer("QUIT\r\n");
306                                 if(findStr(RXBuffer,"221",200)==1)
307                                 {
308                                     smtp_sta = SERVER_NOLINK;
309                                 }else
310                                 {
311                                     disconnectServer();
312                                     smtp_sta = SERVER_NOLINK;
313                                 };
314
315         default:    break;
316     }
317
318     return send_result;//返回结果
319 }
```

图 35.6　邮件发送函数

该函数实现的就是 SMTP 的整个发送流程，只需要用户指定收件人邮箱地址和邮件标题及内容即可，非常方便，但是现在只支持英文邮件，这点大家要注意。

另外在 SMTP.c 文件中也实现了 BASE64 解码和在字符串的某个位置插入另一个字符串函数（本函数使用 sprintf 代替），只不过项目没有用到，但是我们提供给大家，方便在自己后续项目中能够用到，在使用字符串插入函数时，要注意该函数在程序中只能被调用 1 次，否则早晚会造成内存溢出，原因如下（在 str1 的第五个元素位置插入 str2）：

原始数据：

str1[100] = ”hello fengmei!”,str2[100]=”12345”

第一次插入：

str1:” hello12345 fengmei!”

第二次插入：

str1:” hello1234512345 fengmei!”

以此类推，内存必然溢出，最后导致单片机产生总线访问错误卡死在 HardFault_Handler 中断中。但是正是因为该函数“毒性”极强，所以在一些特殊场合有奇效，具体视项目要求而定。

- 主函数

```c
int main(void)
{
    u8 i = 0;

    u8 semail2_body[30];
    u8 key_value;

    u16 DHT11_data; //存储DHT11传感器采集数据
    u16 ADC_data;   //存储ADC数据

    /*初始化各外设*/
    initSysTick();
    initADC();
    initUART();
    initUART2();
    initLED();
    initKey();
    initDHT11();
    initIIC();
    initOLED();
    initNVIC(NVIC_PriorityGroup_2);//开启UART2的接收和空闲中断

    formatScreen(0x00);//清屏

    /*打印ESP8266启动信息到OLED*/
    if(initESP8266()!=0)
    {
        showString(0,0,"init ok!",FONT_16_EN);
    }else
    {
        showString(0,0,"init error!",FONT_16_EN);
    }
    if(connectAP(SSID,PWD)!=0)
    {
        showString(0,2,"conn ap ok!",FONT_16_EN);
    }else
    {
        showString(0,2,"con ap error!",FONT_16_EN);
    }

    Delay_ms(1000);//让数据保持一段时间
    formatScreen(0x00);
    showString(0,2,"Hum :",FONT_16_EN);
    showString(0,4,"Temp:",FONT_16_EN);
    showString(0,6,"Lux :",FONT_16_EN);
    while (1)
    {
        key_value = getKeyValue(KEY_RELEASE);

        /*数据采集显示*/
        if(++i>20)
        {
            i = 0;
            DHT11_data = readDHT11();
            ADC_data = getConvValueAve(10,1000);
            showNumber(40,2,DHT11_data>>8,DEC,3,FONT_16_EN);
            showNumber(40,4,DHT11_data&0x00FF,DEC,3,FONT_16_EN);
            showNumber(40,6,ADC_data,DEC,4,FONT_16_EN);
        }

```

```
84          /*按键状态处理*/
85          if(key_value==KEY_UP)
86          {
87              if(sendEmail("1322698936@qq.com",Header_Title,Body_Content)!=0)
88              {
89                  showString(0,0,"send email1 ok!",FONT_16_EN);
90              }else
91              {
92                  showString(0,0,"send email1 no!",FONT_16_EN);
93              }
94          }else if(key_value==KEY_DOWN)
95          {
96              sprintf(semail2_body,"Hum:%d,Temp:%d,Lux:%d.",DHT11_data>>8,DHT11_data&0x00FF,ADC_data);
97              if(sendEmail("2625724958@qq.com","SENSOR",semail2_body)!=0)
98              {
99                  showString(0,0,"send email2 ok!",FONT_16_EN);
100             }else
101             {
102                 showString(0,0,"send email2 no!",FONT_16_EN);
103             }
104         }else
105         {
106
107         }
108
109         toggleLED();
110         Delay_ms(100);
111     }
112 }
```

图 35.7　主函数

开机后 OLED 会显示此时传感器采集的数据，当按下 UP 键或 DOWN 键时会触发对应的邮件发送动作，且邮件的发送地址不同，内容也不同，当 UP 键按下后，会向其中一个邮箱发送固定内容，当 DOWN 键按下后，会向另外一个邮箱发送变化的传感器数据，但是大家要注意：在编写程序时要记得更改邮件接收方地址，如果都使用程序中的邮箱可能会被系统判定为垃圾邮件导致你的邮件被退回（但如果我收到了，会随机回信的:-)），另外发信方的地址和授权码也要换成自己的。如下图：

```
15  /*发信人邮箱登录名/密码*/
16  #define   EMAIL_USER  "xxx@126.com"
17  #define   EMAIL_PWD  "xxxxxxxxxx"
18
19  /*SMTP服务器IP及端口号*/
20  #define   SMTP_SERVER_IP  "smtp.126.com"
21  #define   SMTP_SERVER_PORT 25
22
23  /*邮件标题和正文*/
24  #define  Header_Title  "HELLO EARTH MAN\r\n"
25  #define  Body_Content  "\r\nTHE MESSAGE COMES FROM OUTSIDE THE EARTH,DON`T ANSWER!DON`T ANSWER!DON`T ANSWER!\r\n.\r\n"
```

图 35.8　用户名和授权码修改处

下面是运行效果：

图 35.8　邮件推送通知

图 35.9　DOWN 键对应邮件

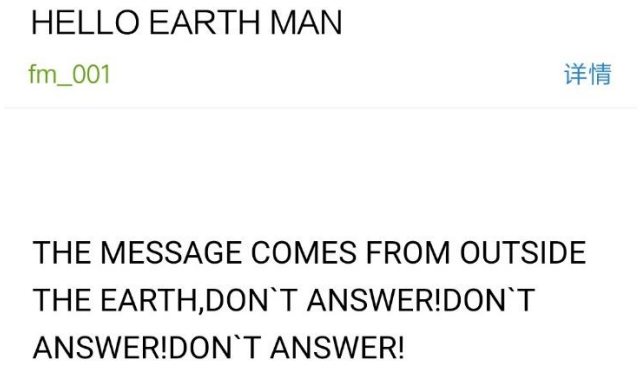


图 35.10　UP 键对应邮件

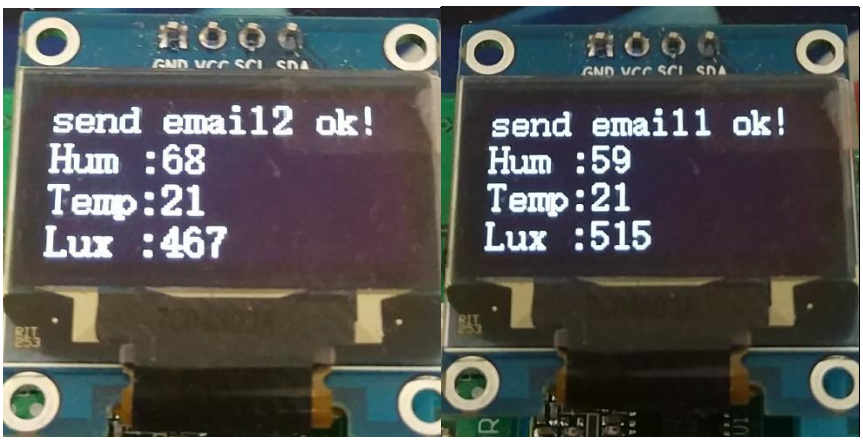


图 35.11　OELD 显示效果

物联网项目实战篇

第三十六章 连接 TLink 云平台

36.1 项目要求

该实战项目名为“植宠保姆”，主要针对长期外出（出差、旅游、探亲）家中植宠无人照料的痛点进行的设计。通过该设备，主人可以远程查看家中的环境数据，并对植物进行补光、灌溉或者对宠物进行饲喂投食（此时需要投食执行部分）等。具体要求如下：

1. 使用 TCP 连接方式将青柚 ZERO 开发板连接到 TLink 物联网云平台；
2. 开发板定时上传采集到的传感器数据；
3. 根据服务器下达的控制指令控制对应输出外设（继电器）；
4. OLED 显示采集到的传感器数据；
5. 使用 TLink 设备管理界面及 APP 对设备传过来的数据进行显示并控制设备。

注：本章用到核心板+扩展板，对应例程 29。

36.2 原理讲解

前几章介绍了网络相关的知识目的就是希望大家在接入云平台不会造成太多的困扰，其实云平台就是一个数据被以某些算法处理并被界面封装的服务器，我们同样也可以使用自己的电脑作为服务器（要保证存在静态的公网 IP，此时可以使用花生壳来映射），只要打通 TCP 或者 UDP 连接，然后将收集到的数据经过处理后显示到用户界面上并根据用户的点击操作对远程设备进行控制即可。前面用到的网络调试助手就充当了一个服务器后台程序的角色，只不过接收和发送数据的方式非常原始而已。因此想要实现更加智能、友好的用户体验就必须对设备传过来的数据进行可视化封装，目前市面上可以供使用的云平台有很多，作者用过的有：TLink，YeeLink（停止服务），机智云，除此之外比较有名的还有阿里云、氦氪云、深智云、百度云、青莲云、云智易、易微联、涂鸦云、传感云、中移物联、乐为物联等。

其中对开发者比较开放的要属 TLink 和机智云，下面分别来讲解一下这两个云平台的特点。

36.2.1 TLink 云平台特点

TLink 云平台对于初学者或者开发者来说最大的特点是提供了丰富的联网方式，比如基础的 TCP/UDP 连接，还有基于 TCP 协议的 ModBus 连接和 MQTT 连接，这里顺便说一下常见的两种物联网设备的开发方式。第一种自然就是设备厂商独立研发软硬件设备，即设备制造商，这种方式可以很好的控制产品成本，但开发周期会长一些；第二种则是使用现成的物联网模

块，比如负责采集环境数据的传感器模块，负责进行动作的被控单元模块如多路继电器模块，还有用于联网的 DTU 模块（可以使用客户端对其进行设置，指定联网协议（TCP/UDP）心跳包等，DTU 只是用于数据的透传），这些模块都通过 RS485 总线连接并使用 ModBus RTU 通信协议进行通信，其特点就是开发周期短，部署方便，但是成本极高，是第一种方案的几倍甚至十几倍（一个 DTU 200-300 元不等，多路继电器 200 元左右，传感器也要 100 多），并不适合小企业的大批量部署。

另外我们在前面说过 ModBus RTU 协议，它是一种主问从答的协议，所以设备的传感器数据的定期采集和输出控制都需要服务器来进行主动干预，这样服务器的开销相对于主动上传来说会高一些。TLink 中的 ModBus 连接就是针对前面提到的第二种方案来设计的，对于独立开发物联网设备来说一般会直接使用 TCP 或者 UDP 连接，这样会更加的灵活。

36.2.2 机智云云平台特点

作者最早使用的云平台就是机智云，当时公司正在研发智能吸顶灯，比较多个厂商之后决定和机智云合作，可以说机智云是这些云平台厂商当中软件服务最好的一家，他们除了提供云平台和数据分析等服务之外也简化了嵌入式工程师的工作内容，机智云可以根据用户的项目自动生成各个单片机平台的代码框架，此时嵌入式工程师需要做的就是在自己熟悉的单片机工程框架中实现产品的业务逻辑，更确切的说就是实现传感器的采集和输出设备的控制，如下图：

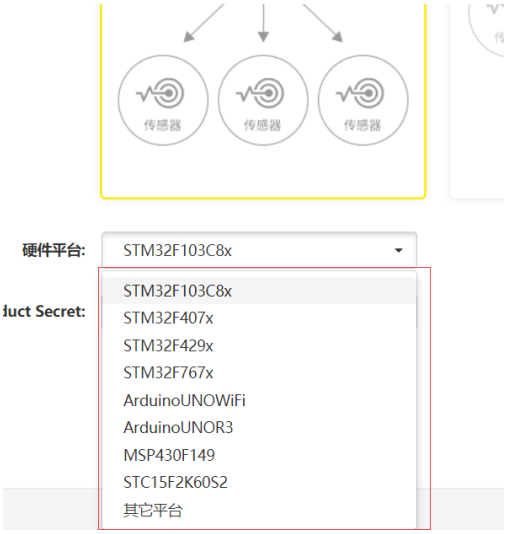


图 36.1 机智云代码生成设置

可以看到在除了支持主流单片机外还支持 Arduino 这样的开源编程套件，要说明的是：此时硬件是 MCU+联网模组（常用的 ESP8266）的方案，因此需要下载机智云提供的对应模组的固件。但也正是因为机智云为开发者做了太多的工作，所以就把一些细节屏蔽掉了，这对于初学者学习来说并不是件好事，因此在本教程中着重讲解使用 TLink 平台和开发板对接，这样可以让初学者接触更多设备联网的细节。

36.2.3 TLink 接入流程

步骤一：注册 TLink

注册后的界面如下：

图 36.2　TLink 主界面

看标题栏，常用的是**控制台**和**帮助中心**，控制台用于管理我们的物联网设备，帮助中心是各种开发文档，比如 TCP 连接的开发等，另外点击设备探索标题则会列出此时使用 TLink 的公开设备详情，如下图（植物保姆就是我们的设备）：

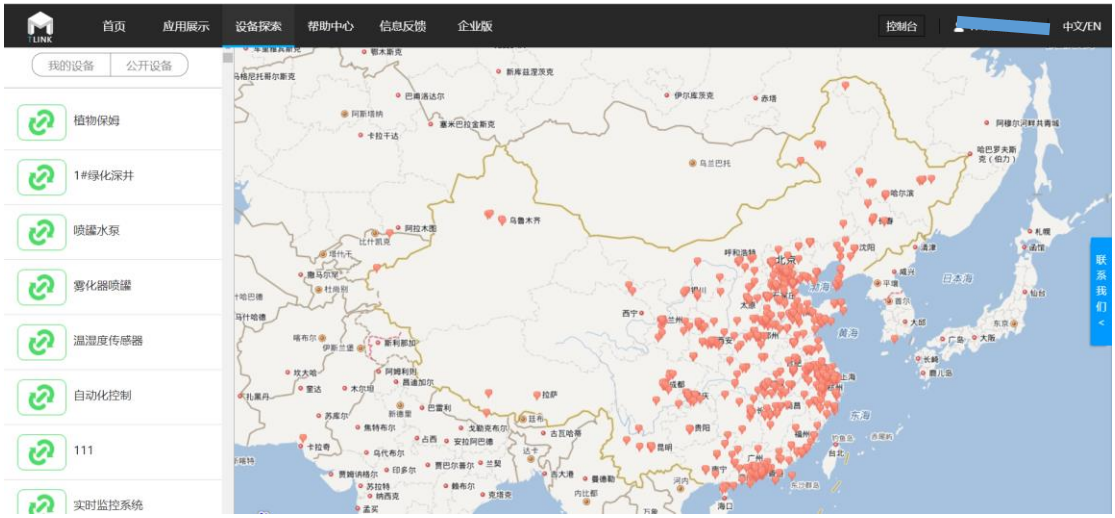


图 36.3　设备探索界面

如果想保护我们的设备数据，也可以在新建设备的时候勾选“不公开设备”。

步骤二：新建设备

在控制台中左侧-我的设备栏中点击添加设备，如下图：

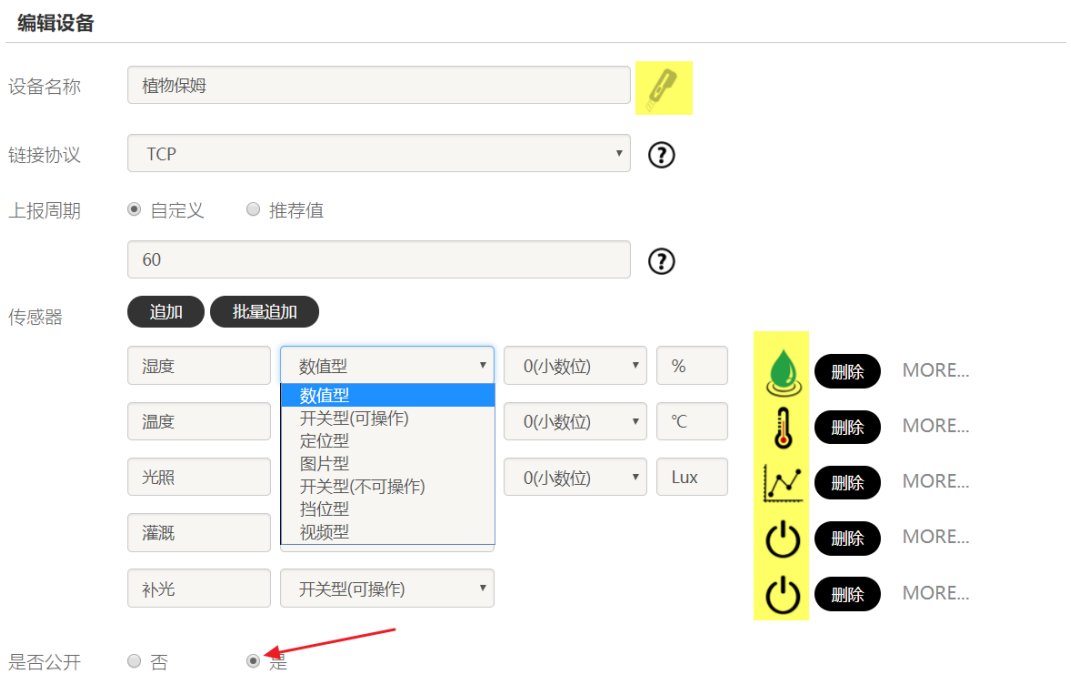


图 36.4　设备设置界面

高亮部分表示可更换图标。在链接协议中选择 TCP，上报周期是指如果在该周期内服务器没有收到该设备传来的数据则会认为设备掉线并在控制界面显示未连接，要注意的是此时服务器并没有断开和设备的 TCP 连接。因此我们要保证设备上报周期小于该周期。传感器部分则是用于描述设备功能属性的核心参数，比如本实验采集三种环境物理量并控制两个继电器，所以就要追加（新建）五个参数，其中三个传感器使用数值型，另外两个继电器使用可操作的开关型，这样就可以在客户端使用拨动开关组件对继电器进行控制了。

步骤三：设置连接协议

设置完设备基础属性之后，我们还需要规定设备通信的协议，这样数据传输会更加安全。

点击控制台下左侧设备标签栏，进入设置连接界面：

图 36.5　设备连接协议设置界面

图中的数据头标签和结束符标签用于告知服务器一帧数据的开始和结束的位置，在数据头标签和结束符标签之间放置的就是传输的有效数据（数据标签），并使用分隔符标签将各个部分区分开来。这些标签的具体含义如下：

	协议标签	标签含义
数据头标签	[H:数据]	"字符型"数据，数据包的包头
	[HE:数据]	"16进制"数据，不能用字符表示时使用16进制表示
分隔符标签	[S:数据]	"字符型"数据，一般为两个数据之间的特殊符号
	[SE:数据]	"16进制"数据，不能用字符表示时使用16进制表示
	[SN[长度]	固定长度的分隔符，只解析长度，不解析内容
数据标签	[D?]	"字符型"的十进制数，长度不固定。需与分隔符配合使用
	[D[长度]]	"字符型"的十进制数，固定字节长度
	[DE[长度]|ABCD]	"16进制整形"数据，ABCD表示字节顺序，最大4个字节
	[DEC[长度]|ABCD]	"16进制整形字符串"数据，ABCD表示字节顺序
	[DF[长度]|ABCD]	"16进制浮点型"数据，ABCD表示字节顺序
	[DSF[长度]|ABCD]	"16进制浮点型字符串"数据，ABCD表示字节顺序
[GPS]	GPS定位数据，格式: 纬度(格式ddmm.mmmm 即dd度 mm.mmmm分) 经度(格式dddmm.mmmm 即ddd度 mm.mmmm分) 2236.70368,N,11350.37840,E	
结束符标签	[T:数据]	"字符型"数据，数据包的结尾字符
	[TE:数据]	"16进制"数据，不能用字符表示时使用16进制表示
	[CRC16]	CRC16位校验值，"16进制"数据，低位在前，高位在后
	[CRC8]	CRC8位校验，"16进制"数据

三、TCP协议例举

例1："字符型"的十进制数

数据包：【ASCII】#AA+0023.1+0023.2+0023.2-0023.1+0023.1+0023.1+0023.1+0023.1(cr)(lf)

协议标签：[H:#AA] [D[7]] [D[7]] [D[7]] [D[7]] [D[7]] [D[7]] [D[7]] [D[7]] [TE:0D0A]

包头 "#AA" ,数据长度为7个字节，结束符回车换行(cr)(lf)十六进制表示是0x0D0A；

解析数据："+0023.1"——》23.1

例2："16进制整形字符串"数据

数据包：【ASCII】+NVn:3,01A4,011D(cr)(lf)

协议标签：[H:+NVn:3] [S:,] [DEC[2]AB] [S:,] [DEC[2]AB] [TE:0D0A]

解析数据："01A4"——》0x01A4——》420

例子3："16进制整形"数据

数据包：【HEX】55 00 01 19 02 BE 5D 35

协议标签：[HE:55 00] [DE[2]AB] [DE[2]AB] [CRC16]

解析数据：0x0119——》281

注：0x5D35为CRC16校验

例子4："字符型"的十进制数

数据包：【ASCII】$GPVTG，257.314，257.314，10.739，19.888，(cr)(lf)

协议标签：[H:$GPVTG] [S:,] [D?] [S:,] [D?] [S:,] [D?] [S:,] [D?] [S:,] [TE:0D0A]

四、协议标签使用规范

数据头标签：	必须得有数据头标签
分隔符标签：	"[S:数据]"和"[SN[长度]]"不能同时存在同一组标签协议 "[SE:数据]"和"[SN[长度]]"不能同时存在同一组标签协议 "[S:数据]"和"[SE[长度]]"不能同时存在同一组标签协议
数据标签：	"[D?]"+"[D[长度]]" 不能同时存在同一组标签协议 "[D?]"+"[DE[长度]|数据]" 不能同时存在同一组标签协议 "[D?]"+"[DEC[长度]|数据]" 不能同时存在同一组标签协议 "[D?]"+"[GPS]" 不能同时存在同一组标签协议 "[D?]"+"[DF[长度]|数据]" 不能同时存在同一组标签协议 "[D?]"+"[DSF[长度]|数据]" 不能同时存在同一组标签协议

图 36.6　设备连接协议使用说明

如果希望设备收到的控制指令是我们指定的内容，则可以如下设置：

图 36.7　设备控制指令

此时服务器下发的控制数据就变为了上面的设置值，测试结果如下：

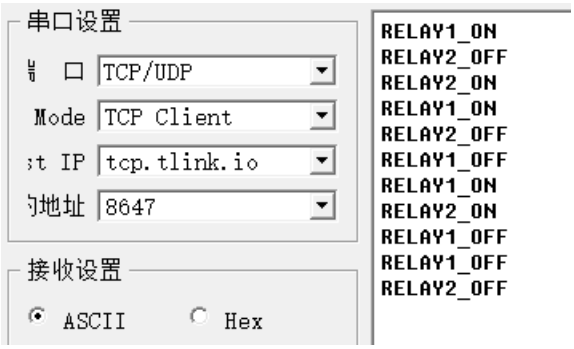


图 36.8　设备控制指令效果

步骤四：程序开发

在编写程序时使用 TCP Client 连接到 tcp.tlink.io:8647，并在建立 TCP 连接后发送一次注册包（产品序列号，是该设备的唯一标识），接着定期发送传感器数据即可，序列号位置如下：

图 36.9 复制设备注册序列号

36.3 程序讲解

- 连接 TLink 服务器函数

```
/**
 * 功能：连接TLink云平台
 * 参数：None
 * 返回值：非0表示连接成功,0表示失败
 */
u8 connectTlink(void)
{
    u8 sta = 0;

    sta = connectServer(TCP,TLINK_IP,TLINK_PORT);    //连接服务器

    if(sta!=0)
    {
        printf("\r\nLinked TLink!");
        sendBuffertoServer(APP_ID);                  //发送注册序号
    }else
    {
        printf("\r\nLink fail!");
    }


    return sta;
}
```

图 36.10 服务器连接函数

连接 TLink 服务器函数其实就是在连接服务器函数的基础上添加了设备序列号的发送，服务器根据设备发送的全服唯一序列号将该硬件设备和服务器存储的那个设备进行数据对接。

- 设备状态发送函数

```c
/**
 * 功能：发送设备状态(传感器数据以及继电器状态)
 * 参数：pdevsta 设备参数结构体指针
 * 返回值：None
 */
void sendDeviceStatus(DeviceSta_Strcture * pdevsta)
{
    char buffer[25] = {0};

    /*格式化数据帧*/
    sprintf(buffer,"FM:%d,%d,%d,%d,%d.#",pdevsta->Humidity,pdevsta->Temperature,pdevsta->Lux,pdevsta->Relay1,pdevsta->Relay2);

    /*发送数据到TLink服务器*/
    sendBuffertoServer(buffer);
}
```

图 36.11 设备状态发送函数

设备状态发送函数其实就是将设备的状态信息以前面事先约定好的数据帧格式进行传输。之后服务器解析并显示到该设备对应的页面中。

- 接收数据处理函数

```c
/**
 * 功能：处理从服务器接收的控制数据
 * 参数：pdevsta 设备参数结构体指针
 * 返回值：None
 * 说明：该函数应该被串口空闲中断调用,已达到最佳实时性能
 */
void processDeviceStatus(DeviceSta_Strcture * pdevsta)
{
    u8 i = 0;

    /*掉线重连*/
    if((u8)strstr(RXBuffer,"CLOSE"))
    {
        printf("\r\nDisconnected!");
        connectTlink();
        memset(RXBuffer,0,RXBUFFER_LEN);
        return;
    }

    /*对应状态控制*/
    if((u8)strstr(RXBuffer,"RELAY1_ON"))
    {
        i = 1;
        pdevsta->Relay1 = 1;
        setRelay(RELAY1,RELAY_CLOSE);

    }else if((u8)strstr(RXBuffer,"RELAY1_OFF"))
    {
        i = 1;
        pdevsta->Relay1 = 0;
        setRelay(RELAY1,RELAY_OPEN);
```

```
    }

    if((u8)strstr(RXBuffer,"RELAY2_ON"))
    {
        i = 1;
        pdevsta->Relay2 = 1;
        setRelay(RELAY2,RELAY_CLOSE);
    }else if((u8)strstr(RXBuffer,"RELAY2_OFF"))
    {
        i = 1;
        pdevsta->Relay2 = 0;
        setRelay(RELAY2,RELAY_OPEN);
    }

    /*清除接收缓冲并更新继电器状态到服务器*/
    if(i!=0)
    {
        printf("RXBuffer is %s.",RXBuffer);
        memset(RXBuffer,0,RXBUFFER_LEN);
        sendDeviceStatus(pdevsta);
    }
}
```

图 36.12 接收数据处理函数

接收数据处理函数用于处理服务器发送过来的控制指令（RELAYx_ON/OFF），根据对应指令控制对应继电器。该函数和第三十四章讲解的数据处理函数意义及功能相同，并且该函数应该被 UART2 空闲中断调用以达到最佳控制效果。

- 主函数

```
    /*打印ESP8266启动信息到OLED*/
    if(initESP8266()!=0)
    {
        showString(0,0,"init ok!",FONT_16_EN);
    }else
    {
        showString(0,0,"init error!",FONT_16_EN);
    }
    if(connectAP(SSID,PWD)!=0)
    {
        showString(0,2,"conn ap ok!",FONT_16_EN);
    }else
    {
        showString(0,2,"con ap error!",FONT_16_EN);
    }
    if(connectTlink()!=0)
    {
        showString(0,4,"conn TLink ok!",FONT_16_EN);
    }else
    {
        showString(0,4,"conn TLink no!",FONT_16_EN);
    }

    Delay_ms(1000);//让数据保持一段时间
    formatScreen(0x00);
    showCNString(32,0,"风媒电子",FONT_16_CN);
    showString(0,2,"Hum :",FONT_16_EN);
    showString(0,4,"Temp:",FONT_16_EN);
    showString(0,6,"Lux :",FONT_16_EN);
```

```
75      while (1)
76      {
77          /*数据采集显示*/
78          if(++i>30)
79          {
80              i = 0;
81              DHT11_data = readDHT11();
82
83              device.Humidity = DHT11_data>>8;
84              device.Temperature = DHT11_data&0x00FF;
85              device.Lux = getConvValueAve(10,1000);
86              showNumber(40,2,device.Humidity,DEC,3,FONT_16_EN);
87              showNumber(40,4,device.Temperature,DEC,3,FONT_16_EN);
88              showNumber(40,6,device.Lux,DEC,4,FONT_16_EN);
89              sendDeviceStatus(&device);
90          }
91
92          toggleLED();
93          Delay_ms(100);
94      }
95  }
```

图 36.13 主函数

主函数每隔 3S 采集一次传感器数据并更新数据到 TLink 服务器和 OLED。下图是运行效果（两幅图截取时间不同，因此部分传感器数值有差别）：

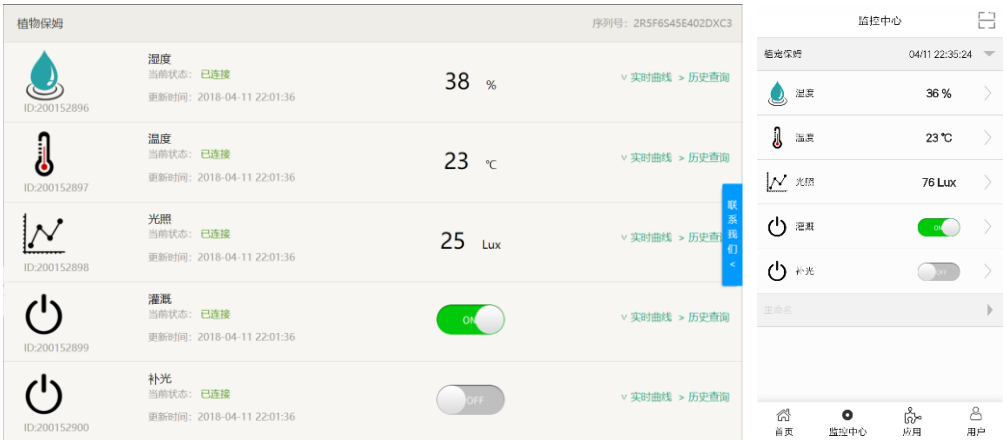


图 36.14 网页及 APP 控制界面

图 36.15 IOS 及 Android 应用下载

此时拨动灌溉或者补光开关对应继电器即可动作。需要注意的是数据的显示存在一定延

时，但不是数据传输的延时，原因是由于我们上传的数据的速度比界面刷新的速度快导致的。因此对于网页显示来说可以使用 F5 主动刷新页面，对于 APP 来说只能按照固有周期刷新，但是由于大多数物联网项目并不属于实时监控，因此显示的延时影响不大，而且很多物联网设备上传服务器的周期可以达到几十分钟甚至几个小时，比如监控一天中河堤的液位变化。

另外我们还可以通过客户端（网页或 APP）实时观测、下载数据波形，并可以下载指定时间段内的设备数据，如下图：

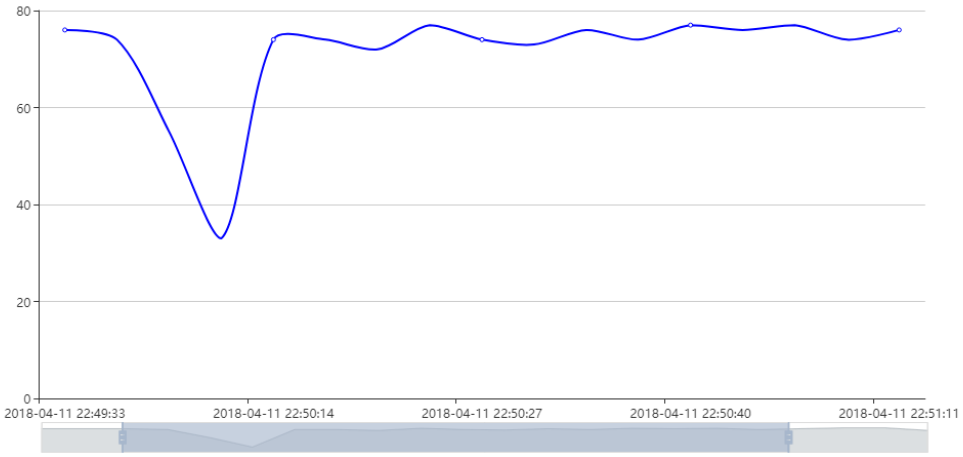


图 36.16 实时数据波形图

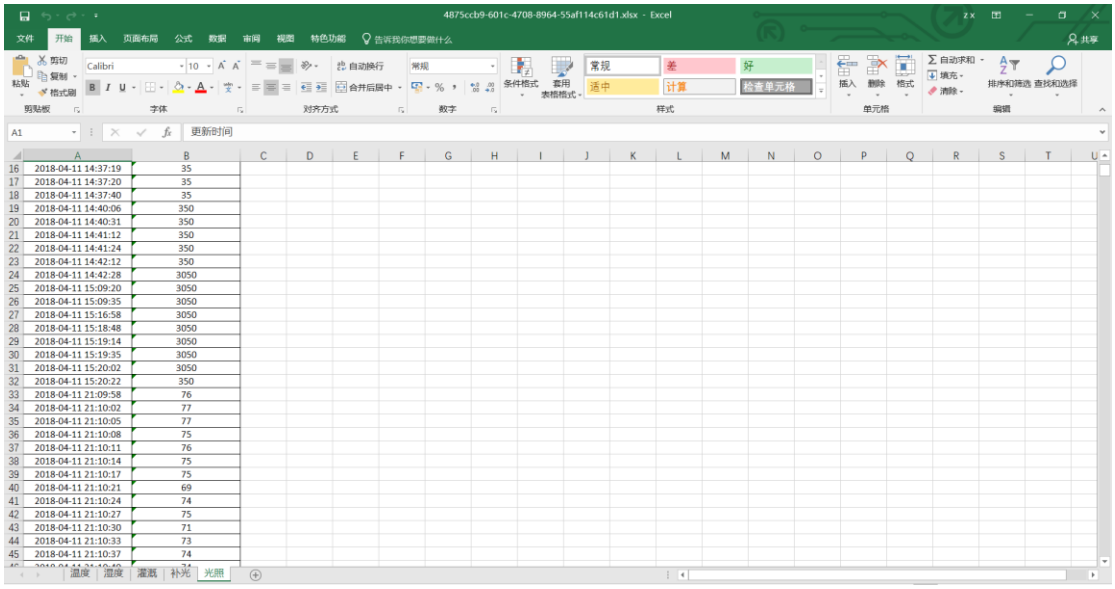


图 36.17 实时数据 EXCEL 表格

其中的 EXCEL 表格可以被用于第三方软件进行数据处理并以更加直观的图形界面显示设备此时的参数。

如果希望多台设备同时工作则可以在控制台的设备列表中点击复制设备，如下图，此时更改新设备程序的注册序列号即可，如下图：

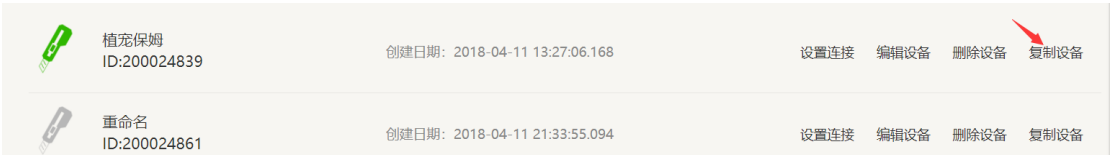


图 36.18 复制设备

这种方式非常适合一个用户多台设备的应用场景，比如温室大棚监控，油井监控等等。

36.3 发送微信

TLink 提供了触发器操作（位于控制台触发器便签下），触发器是指当设备的某个传感器数据触发了设置的阈值则会将此时设备的状态视为异常，并将异常推送到微信公众号。

具体操作步骤如下：

1. 关注 TLink 微信公众号并绑定用户

图 36.19 微信公众号及用户绑定

2. 点击控制台触发器，添加触发器
3. 输入触发器阈值以及是否同意数据转发

图 36.20　触发器设置

数据的转发其实就是当触发事件发生时，应该执行的动作，比如当光照高于设置的阈值时自动触发补光继电器断开。此时我们人为的触发触发器工作，微信公众号立即得到推送的信息，如下图：

图 36.21　微信接收到的数据

当然也可以随时在触发器界面关闭触发器操作，另外在平时开发时，转发功能用的比较少，但是报警用的比较多，比如开发板此时连接一个人体红外传感器，使用触发器后，可以在小偷没有察觉的情况下请他去局里喝茶（就将该项目命名为朝阳区机器人吧:-P）。除了微信报警之外我们还可以选择短信和邮件报警，具体视用户应用情况来定。

36.4 用户控制界面定制

完成开发之后，用户可能还希望高度定制属于自己的控制界面，针对此类需求 TLink 为用户提供了组态应用的编辑和发布功能，用户可以点击控制中心的组态应用，然后点击新增，即可进入编辑界面，如下图：

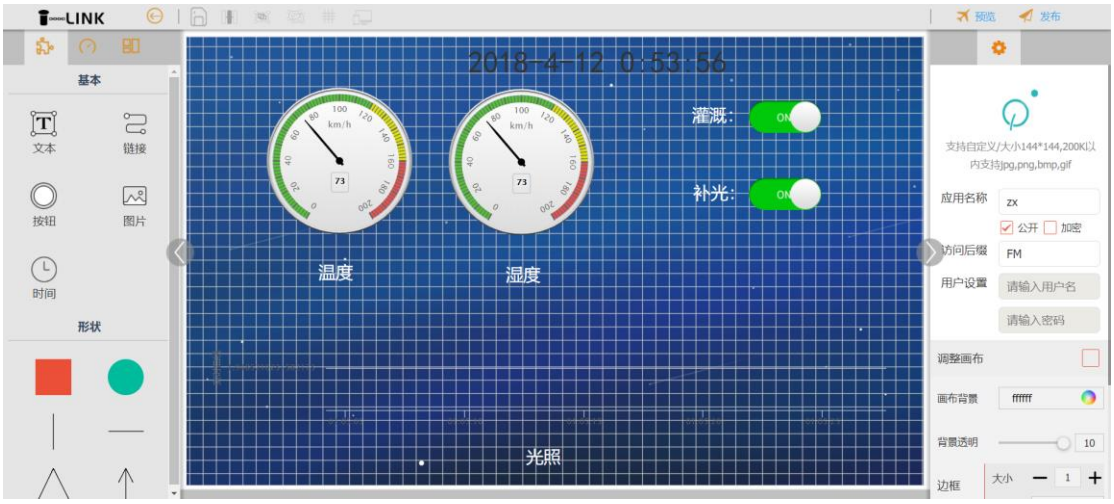


图 36.22 组态编辑界面

我们直接拖动对应的控件并指定控件显示或者控制的设备状态即可，另外也可以定制动态图标和背景图片，最后点击发布。本项目的组态应用界面如下：

图 36.23 组态控制界面效果

最后还有两点需要大家了解一下：

1. TLink 服务器后台如果监测到了一分钟内收到的设备上报数据大于 60 条则会显示上行过快并禁封该设备，此时只能重建或者复制之前的设备（记得修改继电器写入指令）然后在程序中更改序列号。
2. 当设备在上报周期内没有上报数据给服务器,此时服务器并不会主动断开 TCP 连接，只是会显示未连接，如果此时开发板发送数据给服务器则又会恢复正常。

关于 TLink 更多的使用方法可以参考官网的帮助中心。

第三十七章 智能灯泡

37.1 项目要求

使用 TLink 远程控制开发板灯光的亮度、色温以及 RGB 颜色。

注：本章用到核心板+扩展板，对应例程 30。

37.2 原理讲解

本章利用前面讲过的知识，来打造一款智能灯泡，但是由于 TLink 没有提供线性的控制数据类型，如滑动条，因此我们只好另辟蹊径，利用开关量来达到这种效果，其原理就是控制冷光 LED 和暖光 LED 的开关每次动作都会对亮度和色温值进行自加，自加单位为 10，到达 100 后清 0，控制 RGB 的开关每次动作会产生一种随机的颜色，但是要注意，开关的速度不要太快，否则会造成上一章所说的由于上行速度过快而导致 TLink 服务器禁封该设备。

我们先新建一个设备，然后追加传感器，最后效果图如下所示：

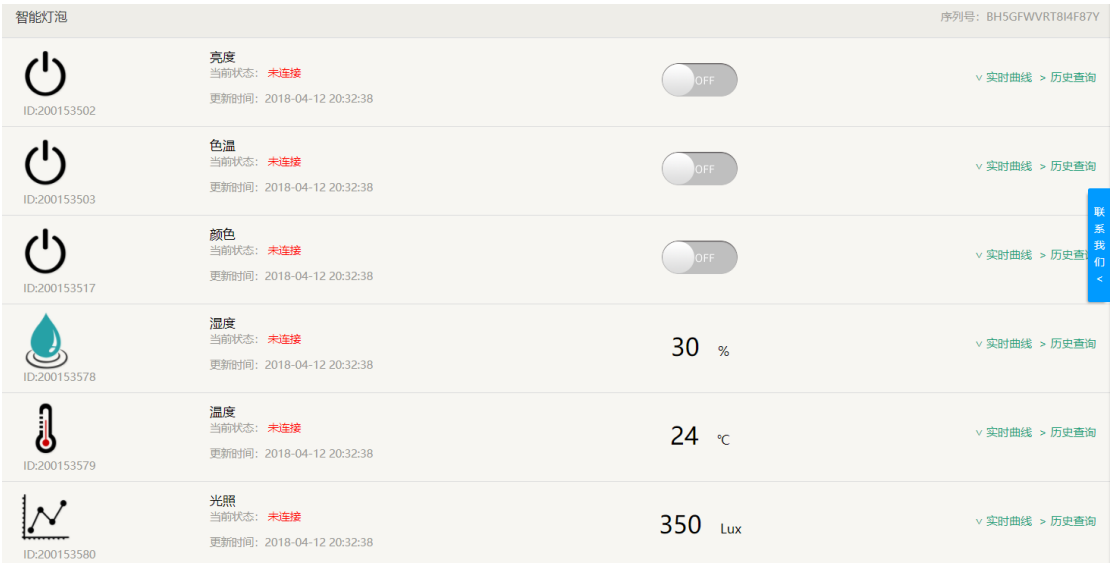


图 37.1 设备配置

接着新设置数据格式，如下图：

图 37.2 协议标签

协议标签中的前三个数据对应的是开关状态，这里设置为 0，其实我们不在乎开关的状态，只要开关动作时发送数据给开发板即可。后面三个数据分别是湿度、温度以及光照值。最后还要设置开关动作时下发的数据内容，如下图（以亮度为例）：

图 37.3　开关写入指令内容

另外两个是 COLORTEMP 以及 RGB，这里要注意 ON 和 OFF 状态的数据内容是一致的。接着复制序列号到程序中，根据上一章的程序稍作修改即可。

37.3 程序讲解

- 数据处理函数

```
57  /**
58   * 功能：处理从服务器接收的控制数据
59   * 参数：pdevsta 设备参数结构体指针
60   * 返回值：None
61   * 说明：该函数应该被串口空闲中断调用,已达到最佳实时性能
62   */
63  void processDeviceStatus(DeviceSta_Strcture * pdevsta)
64  {
65      u8 i = 0;
66
67      /*掉线重连*/
68      if((u8)strstr(RXBuffer,"CLOSE"))
69      {
70          printf("\r\nDisconnected!");
71          connectTlink();
72          memset(RXBuffer,0,RXBUFFER_LEN);
73          return;
74      }
75
76      /*对应状态控制*/
77      if((u8)strstr(RXBuffer,"BRIGHTNESS"))
78      {
79          i = 1;
80          pdevsta->Brightness += 10;
81          if(pdevsta->Brightness>100)
82          {
83              pdevsta->Brightness = 0;
84          }
85          setLight(pdevsta->Brightness,pdevsta->ColorTemp);
```

```
86      }
87
88      if((u8)strstr(RXBuffer,"COLORTEMP"))
89      {
90          i = 1;
91          pdevsta->ColorTemp += 10;
92          if(pdevsta->ColorTemp>100)
93          {
94              pdevsta->ColorTemp = 0;
95          }
96          setLight(pdevsta->Brightness,pdevsta->ColorTemp);
97      }
98
99      if((u8)strstr(RXBuffer,"RGB"))
100     {
101         i = 1;
102         setRGB(rand()%256,rand()%256,rand()%256);
103     }
104
105     /*清除接收缓冲并更新继电器状态到服务器*/
106     if(i!=0)
107     {
108         printf("RXBuffer is %s.",RXBuffer);
109         memset(RXBuffer,0,RXBUFFER_LEN);
110         sendDeviceStatus(pdevsta);
111     }
112 }
```

图 37.4　数据处理函数

该函数按照功能分为两部分，第一部分是亮度和色温值的自加，第二部分是 RGB 的随机动作。

- 发送设备状态函数

```
41  /**
42   * 功能：发送设备状态(传感器数据以及继电器状态)
43   * 参数：pdevsta 设备参数结构体指针
44   * 返回值：None
45   */
46  void sendDeviceStatus(DeviceSta_Strcture * pdevsta)
47  {
48      char buffer[25] = {0};
49
50      /*格式化数据帧*/
51      sprintf(buffer,"FM:%d,%d,%d,%d,%d,%d.#",0,0,0,pdevsta->Humidity,pdevsta->Temperature,pdevsta->Lux);
52
53      /*发送数据到TLink服务器*/
54      sendBuffertoServer(buffer);
55  }
```

图 37.5　发送设备状态函数

该函数上传的是设备采集的传感器数据，三个开关按键默认按 0 发送。最后的主函数除了初始化 RGB 和冷/暖灯光外，其他与上一章的主函数一样，都是开机连接 WIFI、服务器，然后定期（3S）上传设备数据。

另外风媒自己的服务器客户端正在开发，届时大家可以使用花生壳配合风媒服务器客户

端来将自己的电脑搭建成私人的物联网服务器，风媒服务器客户端谍照如下，大家敬请期待：

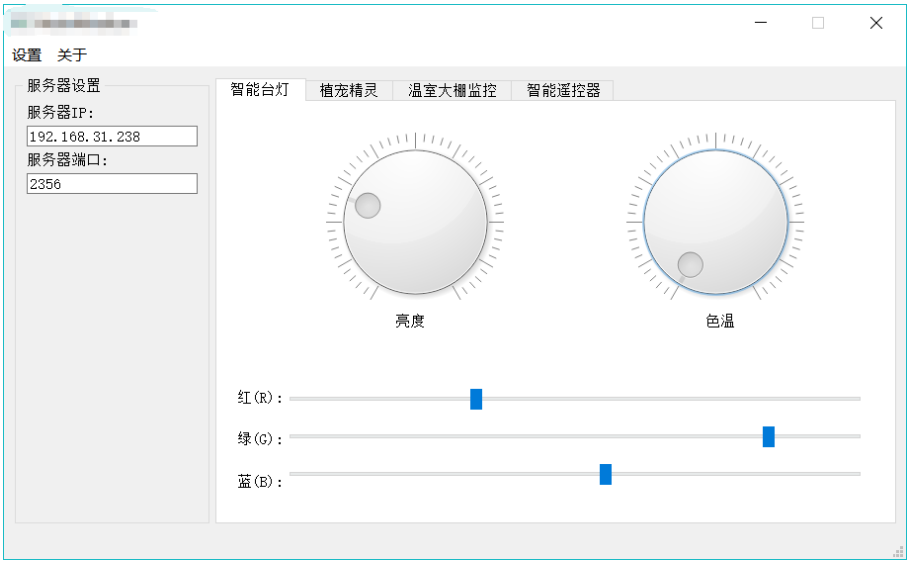


图 37.6　风媒服务器客户端

第三十八章 LoRa 抄表系统

38.1 项目要求

根据前面学习的 LoRa 知识来组建智能抄表系统，具体要求如下：

1. 主机负责定期读取从机的传感器数据，并将其更新到 TLink 和 OLED 上；
2. 从机负责定期采集传感器数据并显示在 OLED 上，同时可根据主机的请求指令来发送采集的传感器数据给主机。

注：该教程使用温湿度和光照来模拟电表/水表的数据。

注：本章用到核心板+扩展板，对应例程 31。

38.2 原理讲解

由于前面已经讲解了所有的外设原理以及网络和云平台接入的相关知识，所以第三十七章和本章就相对容易一些，只需要将用到的程序组合在一起即可。

本章就以第二十七章和第三十六章为基础，来组建一个可以连接云平台的 LoRa 抄表系统，采用一主一从的形式。同样的，需要大家新建一个设备然后追加传感器，最后设置通信协议，按下图设置即可：

图 38.1 设备设置

数据内容的顺序依次为：湿度-温度-光照强度。

38.3 程序讲解

程序的移植主要是针对网关设备（子设备无需更改），我们需要更改设备序列号、发送数据帧格式，以及在主函数中将采集传感器数据部分更换为读取从机数据。

- 发送设备状态函数

```
39  /**
40   * 功能：发送设备状态(传感器数据以及继电器状态)
41   * 参数：pdevsta 设备参数结构体指针
42   * 返回值：None
43   */
44  void sendDeviceStatus(DeviceSta_Strcture * pdevsta)
45  {
46      char buffer[25] = {0};
47
48      /*格式化数据帧*/
49      sprintf(buffer,"FM:%d,%d,%d.#",pdevsta->Humidity,pdevsta->Temperature,pdevsta->Lux);
50
51      /*发送数据到TLink服务器*/
52      sendBuffertoServer(buffer);
53  }
```

图 38.2　发送设备状态函数

程序员只需要更改格式化数据帧内容即可，另外如果上传的传感器数量和长度较多，还需要更改 buffer 数组的长度。

- 处理接受数据函数

```
55  /**
56   * 功能：处理从服务器接收的控制数据
57   * 参数：pdevsta 设备参数结构体指针
58   * 返回值：None
59   * 说明：该函数应该被串口空闲中断调用,已达到最佳实时性能
60   */
61  void processDeviceStatus(DeviceSta_Strcture * pdevsta)
62  {
63      u8 i = 0;
64
65      /*掉线重连*/
66      if((u8)strstr(RXBuffer,"CLOSE"))
67      {
68          printf("\r\nDisconnected!");
69          connectTlink();
70          memset(RXBuffer,0,RXBUFFER_LEN);
71          return;
72      }
73  }
```

图 38.3　处理接收数据函数

因为没有控制部分，所以该函数只需对 TCP 掉线进行重连操作即可。

- 主函数用户逻辑循环

```
76      while (1)
77      {
78          /*数据采集显示*/
79          if(++i>30)
80          {
81              i = 0;
82
83              sendMasterAsk(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL);//主机发送指令
84              receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray);//接收从机数据
85              Delay_ms(100);
86              while(receiveSlaveAck(SLAVE1_ADDR,OP_R_SENSOR,PRAM_R_ALL,DeviceBlock_StructureArray)!=FRAME_OK && ++i<10)//等待从机响应
87              {
88                  Delay_ms(100);
89
90              }
91              i = 0;
92
93
94              device.Humidity = DeviceBlock_StructureArray[SLAVE1_ADDR].Humidity;
95              device.Temperature = DeviceBlock_StructureArray[SLAVE1_ADDR].Temperature;
96              device.Lux = DeviceBlock_StructureArray[SLAVE1_ADDR].Lux;
97
98              showNumber(40,2,device.Humidity,DEC,3,FONT_16_EN);
99              showNumber(40,4,device.Temperature,DEC,3,FONT_16_EN);
100             showNumber(40,6,device.Lux,DEC,4,FONT_16_EN);
101             sendDeviceStatus(&device);
102         }
103
104         toggleLED();
105         Delay_ms(100);
106     }
107 }
```

图 38.4　主函数

可以看到和上一章不同之处就是将采集本机传感器数据部分替换为读取从机数据。接着将固件烧录到开发板中观察其运行效果如下图：

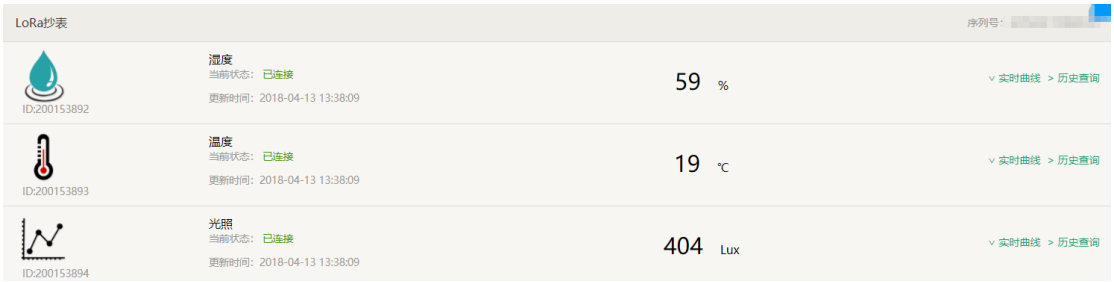


图 38.5　运行效果

到这里，漫长的物联网学习之路就结束了，大家可以继续发挥自己的想象，使用青柚 ZERO 开发板实现其他好玩有趣的项目，比如可以使用 LoRa+PWM 实现超远程遥控小车或者使用 USB+NEC 功能制作一个红外键盘，同时风媒电子也会陆续跟进新的教程和实战项目，不断的丰富和完善本套教程。

附录 A C 语言优先级表

C语言运算符优先级 rightrat

优先级	运算符	名称或含义	使用形式	结合方向	说明
1	[]	数组下标	数组名[常量表达式]	左到右	
	()	圆括号	(表达式)/函数名(形参表)		
	.	成员选择（对象）	对象.成员名		
	->	成员选择（指针）	对象指针->成员名		
2	-	负号运算符	-表达式	右到左	单目运算符
	(类型)	强制类型转换	(数据类型)表达式		单目运算符
	++	自增运算符	++变量名/变量名++		单目运算符
	--	自减运算符	--变量名/变量名--		单目运算符
	*	取值运算符	*指针变量		单目运算符
	&	取地址运算符	&变量名		单目运算符
	!	逻辑非运算符	!表达式		单目运算符
	~	按位取反运算符	~表达式		单目运算符
	sizeof	长度运算符	sizeof(表达式)		
3	/	除	表达式/表达式	左到右	双目运算符
	*	乘	表达式*表达式		双目运算符
	%	余数（取模）	表达式%表达式		双目运算符
4	+	加	表达式+表达式	左到右	双目运算符
	-	减	表达式-表达式		双目运算符
5	<<	左移	表达式>>表达式	左到右	双目运算符
	>>	右移	表达式<<表达式		双目运算符
6	>	大于	表达式>表达式	左到右	双目运算符
	>=	大于等于	表达式>=表达式		双目运算符
	<	小于	表达式<表达式		双目运算符
	<=	小于等于	表达式<=表达式		双目运算符
7	==	等于	表达式==表达式	左到右	双目运算符
	!=	不等于	表达式!=表达式		双目运算符
8	&	按位与	表达式&表达式	左到右	双目运算符
9	^	按位异或	表达式^表达式	左到右	双目运算符
10	|	按位或	表达式|表达式	左到右	双目运算符
11	&&	逻辑与	表达式&&表达式	左到右	双目运算符
12	||	逻辑或	表达式||表达式	左到右	双目运算符
13	?:	条件运算符	表达式1?表达式2:表达式3	右到左	三目运算符
14	=	赋值运算符	变量=表达式	右到左	
	/=	除后赋值	变量/=表达式		
	*=	乘后赋值	变量*=表达式		
	%=	取模后赋值	变量%=表达式		
	+=	加后赋值	变量+=表达式		
	-=	减后赋值	变量-=表达式		
	<<=	左移后赋值	变量<<=表达式		
	>>=	右移后赋值	变量>>=表达式		
	&=	按位与后赋值	变量&=表达式		
	^=	按位异或后赋值	变量^=表达式		
	|=	按位或后赋值	变量|=表达式		
15	,	逗号运算符	表达式1，表达式2...	左到右	从左向右运算

附录 B ASCII 码表

八进制	十六进制	十进制	字符	八进制	十六进制	十进制	字符	八进制	十六进制	十进制	字符	八进制	十六进制	十进制	字符
0	0	0	nul	40	20	32	sp	100	40	64	@	140	60	96	'
1	1	1	soh	41	21	33	!	101	41	65	A	141	61	97	a
2	2	2	stx	42	22	34	"	102	42	66	B	142	62	98	b
3	3	3	etx	43	23	35	#	103	43	67	C	143	63	99	c
4	4	4	eot	44	24	36	$	104	44	68	D	144	64	100	d
5	5	5	enq	45	25	37	%	105	45	69	E	145	65	101	e
6	6	6	ack	46	26	38	&	106	46	70	F	146	66	102	f
7	7	7	bel	47	27	39	`	107	47	71	G	147	67	103	g
10	8	8	bs	50	28	40	(	110	48	72	H	150	68	104	h
11	9	9	ht	51	29	41	)	111	49	73	I	151	69	105	i
12	0a	10	nl	52	2a	42	*	112	4a	74	J	152	6a	106	j
13	0b	11	vt	53	2b	43	+	113	4b	75	K	153	6b	107	k
14	0c	12	ff	54	2c	44	,	114	4c	76	L	154	6c	108	l
15	0d	13	er	55	2d	45	-	115	4d	77	M	155	6d	109	m
16	0e	14	so	56	2e	46	.	116	4e	78	N	156	6e	110	n
17	0f	15	si	57	2f	47	/	117	4f	79	O	157	6f	111	o
20	10	16	dle	60	30	48	0	120	50	80	P	160	70	112	p
21	11	17	dc1	61	31	49	1	121	51	81	Q	161	71	113	q
22	12	18	dc2	62	32	50	2	122	52	82	R	162	72	114	r
23	13	19	dc3	63	33	51	3	123	53	83	S	163	73	115	s
24	14	20	dc4	64	34	52	4	124	54	84	T	164	74	116	t
25	15	21	nak	65	35	53	5	125	55	85	U	165	75	117	u
26	16	22	syn	66	36	54	6	126	56	86	V	166	76	118	v
27	17	23	etb	67	37	55	7	127	57	87	W	167	77	119	w
30	18	24	can	70	38	56	8	130	58	88	X	170	78	120	x
31	19	25	em	71	39	57	9	131	59	89	Y	171	79	121	y
32	1a	26	sub	72	3a	58	:	132	5a	90	Z	172	7a	122	z
33	1b	27	esc	73	3b	59	;	133	5b	91	[	173	7b	123	{
34	1c	28	fs	74	3c	60	<	134	5c	92	\	174	7c	124	|
35	1d	29	gs	75	3d	61	=	135	5d	93	]	175	7d	125	}
36	1e	30	re	76	3e	62	>	136	5e	94	^	176	7e	126	~
37	1f	31	us	77	3f	63	?	137	5f	95	_	177	7f	127	del

附录 C HID 键值表

Usage ID (Dec)	Usage ID (Hex)	Usage Name	Ref: Typical AT-101 Position	PC-AT	Mac	UNIX	Boot
0	00	Reserved (no event indicated)[9]	N/A	√	√	√	4/101/104
1	01	Keyboard ErrorRollOver[9]	N/A	√	√	√	4/101/104
2	02	Keyboard POSTFail[9]	N/A	√	√	√	4/101/104
3	03	Keyboard ErrorUndefined[9]	N/A	√	√	√	4/101/104
4	04	Keyboard a and A[4]	31	√	√	√	4/101/104
5	05	Keyboard b and B	50	√	√	√	4/101/104
6	06	Keyboard c and C[4]	48	√	√	√	4/101/104
7	07	Keyboard d and D	33	√	√	√	4/101/104
8	08	Keyboard e and E	19	√	√	√	4/101/104
9	09	Keyboard f and F	34	√	√	√	4/101/104
10	0A	Keyboard g and G	35	√	√	√	4/101/104
11	0B	Keyboard h and H	36	√	√	√	4/101/104
12	0C	Keyboard i and I	24	√	√	√	4/101/104
13	0D	Keyboard j and J	37	√	√	√	4/101/104
14	0E	Keyboard k and K	38	√	√	√	4/101/104
15	0F	Keyboard l and L	39	√	√	√	4/101/104
16	10	Keyboard m and M[4]	52	√	√	√	4/101/104
17	11	Keyboard n and N	51	√	√	√	4/101/104
18	12	Keyboard o and O[4]	25	√	√	√	4/101/104
19	13	Keyboard p and P[4]	26	√	√	√	4/101/104
20	14	Keyboard q and Q[4]	17	√	√	√	4/101/104

21	15	Keyboard r and R	20	√	√	√	4/101/104
22	16	Keyboard s and S[4]	32	√	√	√	4/101/104
23	17	Keyboard t and T	21	√	√	√	4/101/104
24	18	Keyboard u and U	23	√	√	√	4/101/104
25	19	Keyboard v and V	49	√	√	√	4/101/104
26	1A	Keyboard w and W[4]	18	√	√	√	4/101/104
27	1B	Keyboard x and X[4]	47	√	√	√	4/101/104
28	1C	Keyboard y and Y[4]	22	√	√	√	4/101/104
29	1D	Keyboard z and Z[4]	46	√	√	√	4/101/104
30	1E	Keyboard 1 and !4	2	√	√	√	4/101/104
31	1F	Keyboard 2 and @[4]	3	√	√	√	4/101/104
32	20	Keyboard 3 and #[4]	4	√	√	√	4/101/104
33	21	Keyboard 4 and $[4]	5	√	√	√	4/101/104
34	22	Keyboard 5 and %[4]	6	√	√	√	4/101/104
35	23	Keyboard 6 and ^[4]	7	√	√	√	4/101/104
36	24	Keyboard 7 and &[4]	8	√	√	√	4/101/104
37	25	Keyboard 8 and *[4]	9	√	√	√	4/101/104
38	26	Keyboard 9 and ([4]	10	√	√	√	4/101/104
39	27	Keyboard 0 and)[4]	11	√	√	√	4/101/104
40	28	Keyboard Return (ENTER)[5]	43	√	√	√	4/101/104
41	29	Keyboard ESCAPE	110	√	√	√	4/101/104
42	2A	Keyboard DELETE (Backspace)[13]	15	√	√	√	4/101/104
43	2B	Keyboard Tab	16	√	√	√	4/101/104
44	2C	Keyboard Spacebar	61	√	√	√	4/101/104
45	2D	Keyboard - and (underscore)[4]	12	√	√	√	4/101/104
46	2E	Keyboard = and +[4]	13	√	√	√	4/101/104
47	2F	Keyboard [and {[4]	27	√	√	√	4/101/104
48	30	Keyboard] and }[4]	28	√	√	√	4/101/104
49	31	Keyboard \ and |	29	√	√	√	4/101/104
50	32	Keyboard Non-US # and ~[2]	42	√	√	√	4/101/104
51	33	Keyboard ; and :[4]	40	√	√	√	4/101/104
52	34	Keyboard ‘ and “[4]	41	√	√	√	4/101/104
53	35	Keyboard Grave Accent and Tilde[4]	1	√	√	√	4/101/104
54	36	Keyboard, and <[4]	53	√	√	√	4/101/104
55	37	Keyboard . and >[4]	54	√	√	√	4/101/104
56	38	Keyboard / and ?[4]	55	√	√	√	4/101/104
57	39	Keyboard Caps Lock[11]	30	√	√	√	4/101/104
58	3A	Keyboard F1	112	√	√	√	4/101/104

59	3B	Keyboard F2	113	√	√	√ 4/101/104
60	3C	Keyboard F3	114	√	√	√ 4/101/104
61	3D	Keyboard F4	115	√	√	√ 4/101/104
62	3E	Keyboard F5	116	√	√	√ 4/101/104
63	3F	Keyboard F6	117	√	√	√ 4/101/104
64	40	Keyboard F7	118	√	√	√ 4/101/104
65	41	Keyboard F8	119	√	√	√ 4/101/104
66	42	Keyboard F9	120	√	√	√ 4/101/104
67	43	Keyboard F10	121	√	√	√ 4/101/104
68	44	Keyboard F11	122	√	√	√ 101/104
69	45	Keyboard F12	123	√	√	√ 101/104
70	46	Keyboard PrintScreen[1]	124	√	√	√ 101/104
71	47	Keyboard Scroll Lock[11]	125	√	√	√ 4/101/104
72	48	Keyboard Pause[1]	126	√	√	√ 101/104
73	49	Keyboard Insert[1]	75	√	√	√ 101/104
74	4A	Keyboard Home[1]	80	√	√	√ 101/104
75	4B	Keyboard PageUp[1]	85	√	√	√ 101/104
76	4C	Keyboard Delete Forward[1;14]	76	√	√	√ 101/104
77	4D	Keyboard End[1]	81	√	√	√ 101/104
78	4E	Keyboard PageDown[1]	86	√	√	√ 101/104
79	4F	Keyboard RightArrow[1]	89	√	√	√ 101/104
80	50	Keyboard LeftArrow[1]	79	√	√	√ 101/104
81	51	Keyboard DownArrow[1]	84	√	√	√ 101/104
82	52	Keyboard UpArrow[1]	83	√	√	√ 101/104
83	53	Keypad Num Lock and Clear[11]	90	√	√	√ 101/104
84	54	Keypad /[1]	95	√	√	√ 101/104
85	55	Keypad *	100	√	√	√ 4/101/104
86	56	Keypad -	105	√	√	√ 4/101/104
87	57	Keypad +	106	√	√	√ 4/101/104
88	58	Keypad ENTER[5]	108	√	√	√ 101/104
89	59	Keypad 1 and End	93	√	√	√ 4/101/104
90	5A	Keypad 2 and Down Arrow	98	√	√	√ 4/101/104
91	5B	Keypad 3 and PageDn	103	√	√	√ 4/101/104
92	5C	Keypad 4 and Left Arrow	92	√	√	√ 4/101/104
93	5D	Keypad 5	97	√	√	√ 4/101/104
94	5E	Keypad 6 and Right Arrow	102	√	√	√ 4/101/104
95	5F	Keypad 7 and Home	91	√	√	√ 4/101/104
96	60	Keypad 8 and Up Arrow	96	√	√	√ 4/101/104

97	61	Keypad 9 and PageUp	101	√	√	√	4/101/104
98	62	Keypad 0 and Insert	99	√	√	√	4/101/104
99	63	Keypad . and Delete	104	√	√	√	4/101/104
100	64	Keyboard Non-US \ and |[3;6]	45	√	√	√	4/101/104
101	65	Keyboard Application[10]	129	√		√	104
102	66	Keyboard Power[9]			√	√	
103	67	Keypad =			√		
104	68	Keyboard F13			√		
105	69	Keyboard F14			√		
106	6A	Keyboard F15			√		
107	6B	Keyboard F16					
108	6C	Keyboard F17					
109	6D	Keyboard F18					
110	6E	Keyboard F19					
111	6F	Keyboard F20					
112	70	Keyboard F21					
113	71	Keyboard F22					
114	72	Keyboard F23					
115	73	Keyboard F24					
116	74	Keyboard Execute				√	
117	75	Keyboard Help				√	
118	76	Keyboard Menu				√	
119	77	Keyboard Select				√	
120	78	Keyboard Stop				√	
121	79	Keyboard Again				√	
122	7A	Keyboard Undo				√	
123	7B	Keyboard Cut				√	
124	7C	Keyboard Copy				√	
125	7D	Keyboard Paste				√	
126	7E	Keyboard Find				√	
127	7F	Keyboard Mute					√
128	80	Keyboard Volume Up					√
129	81	Keyboard Volume Down					√
130	82	Keyboard Locking Caps Lock[12]					√
131	83	Keyboard Locking Num Lock[12]					√
132	84	Keyboard Locking Scroll Lock[12]					√
133	85	Keypad Comma[27]	107				
134	86	Keypad Equal Sign[29]					

135	87	Keyboard International1[15,28]	56
136	88	Keyboard International2[16]	
137	89	Keyboard International3[17]	
138	8A	Keyboard International4[18]	
139	8B	Keyboard International5[19]	
140	8C	Keyboard International6[20]	
141	8D	Keyboard International7[21]	
142	8E	Keyboard International8[22]	
143	8F	Keyboard International9[22]	
144	90	Keyboard LANG1[25]	
145	91	Keyboard LANG2[26]	
146	92	Keyboard LANG3[30]	
147	93	Keyboard LANG4[31]	
148	94	Keyboard LANG5[32]	
149	95	Keyboard LANG6[8]	
150	96	Keyboard LANG7[8]	
151	97	Keyboard LANG8[8]	
152	98	Keyboard LANG9[8]	
153	99	Keyboard Alternate Erase[7]	
154	9A	Keyboard SysReq/Attention[1]	
155	9B	Keyboard Cancel	
156	9C	Keyboard Clear	
157	9D	Keyboard Prior	
158	9E	Keyboard Return	
159	9F	Keyboard Separator	
160	A0	Keyboard Out	
161	A1	Keyboard Oper	
162	A2	Keyboard Clear/Again	
163	A3	Keyboard CrSel/Props	
164	A4	Keyboard ExSel	
165-175	A5-AF	Reserved	
176	B0	Keypad 00	
177	B1	Keypad 000	
178	B2	Thousands Separator [33]	
179	B3	Decimal Separator [33]	
180	B4	Currency Unit [34]	
181	B5	Currency Sub-unit [34]	
182	B6	Keypad (	

183	B7	Keypad)					
184	B8	Keypad {					
185	B9	Keypad }					
186	BA	Keypad Tab					
187	BB	Keypad Backspace					
188	BC	Keypad A					
189	BD	Keypad B					
190	BE	Keypad C					
191	BF	Keypad D					
192	C0	Keypad E					
193	C1	Keypad F					
194	C2	Keypad XOR					
195	C3	Keypad ^					
196	C4	Keypad %					
197	C5	Keypad <					
198	C6	Keypad >					
199	C7	Keypad &					
200	C8	Keypad &&					
201	C9	Keypad |					
202	CA	Keypad ||					
203	CB	Keypad :					
204	CC	Keypad #					
205	CD	Keypad Space					
206	CE	Keypad @					
207	CF	Keypad !					
208	D0	Keypad Memory Store					
209	D1	Keypad Memory Recall					
210	D2	Keypad Memory Clear					
211	D3	Keypad Memory Add					
212	D4	Keypad Memory Subtract					
213	D5	Keypad Memory Multiply					
214	D6	Keypad Memory Divide					
215	D7	Keypad +/-					
216	D8	Keypad Clear					
217	D9	Keypad Clear Entry					
218	DA	Keypad Binary					
219	DB	Keypad Octal					
220	DC	Keypad Decimal					
221	DD	Keypad Hexadecimal					
222-223	DE-DF	Reserved					
224	E0	Keyboard LeftControl	58	√	√	√	4/101/104
225	E1	Keyboard LeftShift	44	√	√	√	4/101/104
226	E2	Keyboard LeftAlt	60	√	√	√	4/101/104
227	E3	Keyboard Left GUI[10;23]	127	√	√	√	104
228	E4	Keyboard RightControl	64	√	√	√	101/104
229	E5	Keyboard RightShift	57	√	√	√	4/101/104
230	E6	Keyboard RightAlt	62	√	√	√	101/104
231	E7	Keyboard Right GUI[10;24]	128	√	√	√	104
232-65535	E8-FFFF	Reserved					