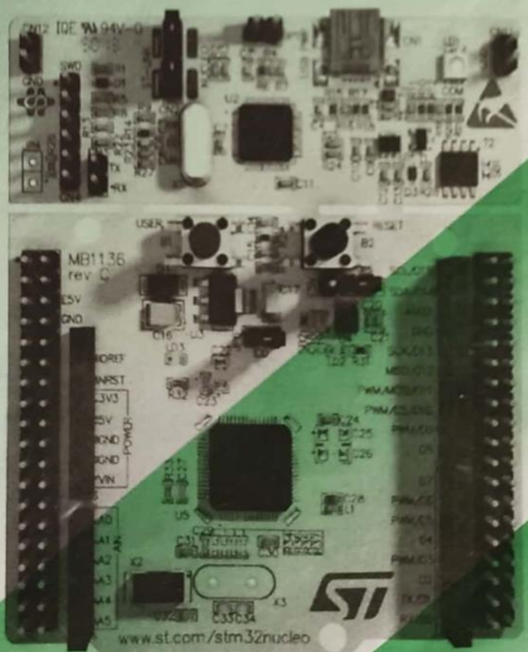
挑战性学习课程

嵌入式系统设计

STM32CubeMX与HAL库



主编：漆强　电子科技大学

目录

第一章　嵌入式系统概要

本章导读

本章重点介绍嵌入式系统的基本概念及组成，从硬件和软件两个方面详细分析嵌入式系统的组成。硬件方面介绍了嵌入式处理器的种类，软件方面介绍了嵌入式软件的组成，并引入了工程应用中的硬件抽象层的设计思想。在编程架构上分析了传统的前后台编程方法和基于嵌入式实时操作系统编程方法的优缺点。

本章知识点

嵌入式系统基本概念

嵌入式系统的组成

常见的嵌入式微处理器及嵌入式实时操作系统

硬件抽象层的设计思想

1.1　嵌入式系统的概念及特点

嵌入式系统是电子信息产业的基础，是智能系统的核心，广泛应用于工业控制、汽车电子、智能家居、医疗器械和智能穿戴设备等众多领域。伴随物联网和人工智能的快速发展，嵌入式系统在智能系统中发挥着越来越大的作用。

嵌入式系统的发展离不开通用计算机技术的发展：在微型计算机时代，人们开始尝试将通用计算机系统用于航海，机械控制等领域。但是在使用过程中发现，由于通用计算机体积较大，外围电路众多，会占据较大的物理空间。而对于控制领域而言，往往只需要进行简单的输入/输出，不需要海量的存储和高分辨率的显示。

由于上述领域的需求，向计算机专家提出了新的设计目标：设计一种单芯片的处理器，将存储器，输入/输出接口、定时器和中断系统等都集成在上面，并减少芯片体积和功耗。于是，各个半导体厂家不断进行研发，最终以 1976 年英特尔公司设计的 MCS-48 系列处理器为代表，诞生了真正意义上的嵌入式系统，进而形成了现代计算机技术的两大发展分支：通用计算机按照高速运算和海量存储的技术方向发展；嵌入式系统则向低功耗、高可靠性和便携性等方向发展。

根据 IEEE（电气和电子工程师协会）的定义，嵌入式系统是“用于控制、监视或者辅助~~操作机器或设备的装置~~”(devices used to control,monitor,or assist the operation of equipment,machinery or plants)。从中可以看出嵌入式系统是软件和硬件的综合体，还可以涵盖机械等附属装备。

综合国内外文献，普遍认同的嵌入式系统定义为：嵌入到对象体系中，以应用为中心，以计算机技术为基础，软硬件可裁剪，适应应用系统对功能、可靠性、成本、体积、功耗等严格要求的专用计算机系统。

从上述的定义可以看出，嵌入式系统具备以下三个特点：

- 嵌入性

嵌入式系统必须嵌入到对象系统中，满足对象系统的要求，如物理环境（小型）、电气环境（可靠）、成本（价廉）。

- 专用性

嵌入式系统特别强调“量身定做”的原则，开发人员往往需要针对不同的用途开发出一

个截然不同的嵌入式系统，即使功能相近的嵌入式系统，也需要对硬件和软件进行相应的修改才能够使用。同时嵌入式系统的开发也需要专用的开发环境和开发工具。开发环境一般是基于通用计算机上的集成开发环境，用于嵌入式程序的开发；开发工具则包括了仿真器、逻辑分析仪和数字示波器等硬件设备。

- 计算机系统

嵌入式系统以计算机技术为基础，融合光学、机械学和电子学等多个学科的知识。以为嵌入式处理器为系统核心，并配置与对象系统相适应的外围电路。

广义的嵌入式系统将凡是带有嵌入式处理器的专用系统都称为嵌入式系统，而狭义上的嵌入式系统则要求在嵌入式处理器上运行嵌入式操作系统，完成特定功能，用于特定场合。

我们日常生活中经常使用的智能手环就是一个典型的嵌入式系统。智能手环的外观及内部结构如图 1-1 所示。

图 1-1　智能手环外观图及内部结构图

在智能手环的电路设计中，采用了柔性电路板，将显示电路和控制电路对折在一起。显示电路使用行列排布的发光二极管（Light Emitting Diode，LED）阵列来显示运动时间和运动步数；步数计量采用加速度传感器检测运动状态；数据通信采用蓝牙芯片与手机进行信息传递；主控芯片则使用意~~体~~法半导体公司的低功耗系列微控制器 STM32L 完成对整个系统的控制、显示和信息处理。

1.2　嵌入式系统组成

嵌入式系统主要由硬件和软件两个部分组成，其结构层次如图 1-2 所示。

硬件部分主要包括由嵌入式处理器、电源电路、时钟电路、复位电路、调试电路组成的最小系统电路，以及用于人机交互的显示电路和键盘电路，用于数据采集的采集电路，用于数据传输的通信电路等外围电路。针对不同的工程应用，具体的嵌入式系统所需的外围电路各不相同。

软件部分由驱动程序、操作系统（可选）和应用程序三个部分组成。驱动程序用于实现硬件和软件之间的接口；操作系统实现进程调度、任务处理；应用程序则完成具体的业务逻辑，需要根据具体的应用环境由开发人员自行编写。

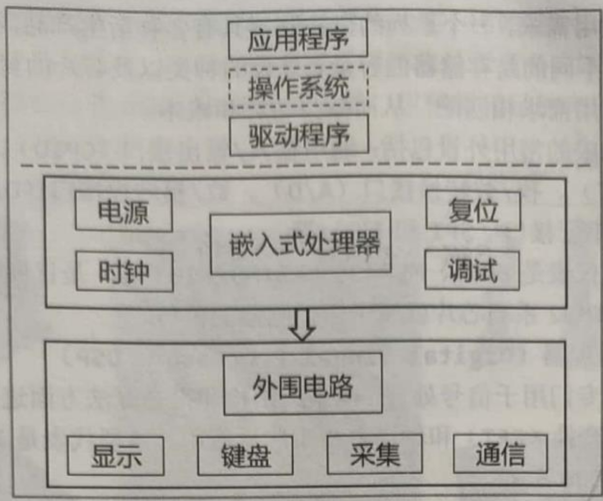


图 1-2　嵌入式系统层次结构图

1.2.1　嵌入式系统硬件

嵌入式系统硬件主要由嵌入式处理器和外围电路两大部分组成。

1. 嵌入式处理器

嵌入式处理器是嵌入式系统的核心，常用的嵌入式处理器有以下几种类型：

(1) 微处理器（Micro Processor Unit，MPU）

微处理器由通用计算机中的 CPU 演变而来，一般具有内置的内存管理单元（MMU），提供内存保护和多任务能力的虚拟存储器等功能，具有较高的性能和丰富的外围部件接口。典型的代表是 386EX、PowerPC 和 MIPS 等。

(2) 微控制器（Micro Controller Unit，MCU）

微控制器是一种将中央处理器（Central Processing Unit，CPU）、存储器、中断系统和外设集成在一块单芯片上的集成电路，我们常说的单片机就属于微控制器。微控制器的结构框图如图 1-3 所示。

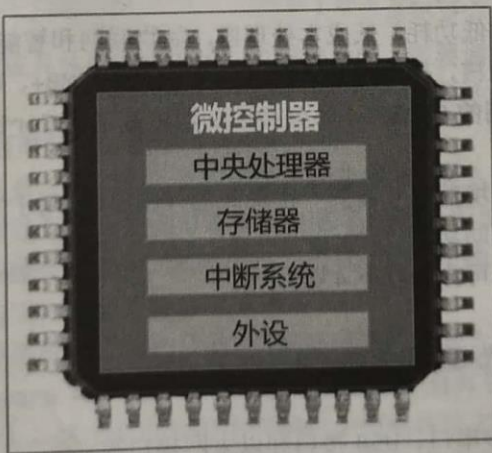


图 1-3　微控制器结构框图

和通用计算机系统相比，它将通用计算机系统中许多由板卡完成的功能集成在芯片内部，从而有利于嵌入式系统的小型化，同时还具有很高的效率和可靠性。

为适应不同的应用需求，一个系列的微控制器具有多种衍生产品，每种衍生产品的处理器内核都是一样的，不同的是存储器的容量、外设的种类以及芯片的封装。这样可以使微控制器最大限度的和应用需求相匹配，从而减少功耗和成本。

微控制器内部集成的常用外设包括：通用输入/输出接口（GPIO）、定时器（Timer）、串行通信接口（UART）、模/数转换接口（A/D）、数/模转换接口（D/A）、通用串行总线接口（USB）和同步串行接口（SPI 和 I^2C）等。

微控制器是典型代表是 8051、MSP430 和 STM32F103 等，是目前最常用的嵌入式处理器，本书所介绍的 STM32 系列芯片就属于微控制器的范畴。

(3) 数字信号处理器（Digital Singal Processor，DSP）

数字信号处理器专门用于信号处理，在系统结构和指令算法方面进行了特殊设计。在数字滤波、快速傅里叶变换（FFT）和频谱分析中广泛应用。典型代表是 TI（德州仪器）公司的 TMS320C2000/C5000 系列。

(4) 片上系统（System On Chip，SOC）

近年来，嵌入式系统正在被设计到单个硅片上，成为片上系统。SOC 将微处理器、IP（知识产权）核和存储器集成在单一芯片上，通常是客户定制或者面向特定用途的标准产品。

(5) ARM 处理器（Advanced RISC Machines，ARM）

ARM 系列处理器是由英国 ARM 公司设计的低功耗处理器。ARM 公司不生产芯片，只负责设计处理器的内核，并授权给半导体厂商，各大半导体厂商在此基础上加上各具特色的外设后，生产出的芯片统称为 ARM 处理器。

常用的 ARM 微处理器有 ARM7 系列、ARM9 系列、ARM11 系列和 Cortex 系列等。其中，Cortex 系列处理器采用了全新的 ARMv7 架构，用于取代 ARM7、ARM9 和 ARM11 等系列，根据适用的领域不同，Cortex 系列处理器分成了 3 大类：

- Cortex-A ——
  面向高端应用的高性能处理器，如智能手机等。
- Cortex-R ——
  面向实时应用的高可靠性处理器，如汽车电子和自动控制等。
- Cortex-M ——
  面向微控制器应用的低功耗、低成本处理器，如物联网和智能硬件等。Cortex-M 处理器目前已经发展出 5 代内核，分别是 Cortex-M0、Cortex-M0+、Cortex-M3、Cortex-M4 和 Cortex-M7。本书介绍的 STM32F4 系列微控制器就是基于 Cortex-M4 内核。

2. 外围电路

嵌入式系统和外界环境的交互需要一定形式的外围电路，用于完成环境感知、信息传输、数据存储和人机交互等辅助功能。

根据外围设备的功能可分为以下 4 类：

(1) 环境感知类

温度/湿度传感器、光电类传感器、压力传感器等。

(2) 接口类

A/D 接口、同步/异步串口、USB 接口和以太网接口等。

(3) 存储类

SRAM、DRAM、NAND Flash 和 NOR Flash 等。

(4) 人机交互类

指示灯、数码管、液晶显示屏和键盘等。

选择什么样的外围电路要根据具体的应用场景来决定。比如显示系统的输出信息，如果是应用在便携式低功耗的产品中，则应该选择液晶屏作为显示设备。如果要求在户外能够清晰的显示，则应该选择数码管或者 LED 矩阵显示屏，以确保显示的亮度。

常见的外围电路实物图如图 1-4 所示。

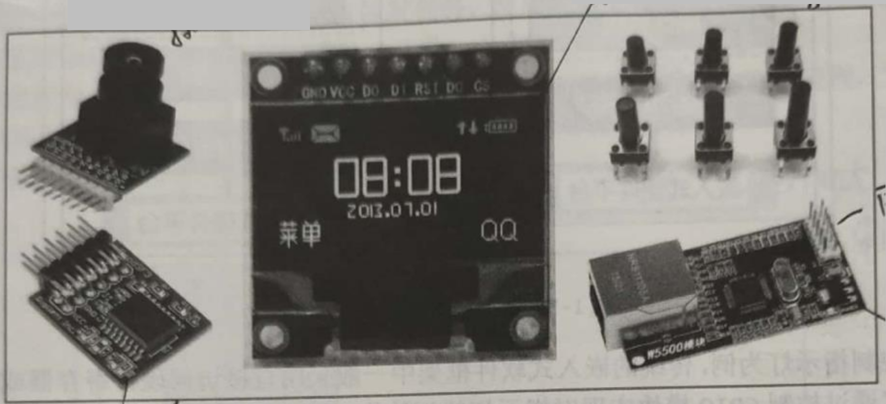


图 1-4 嵌入式系统常见外围电路

1.2.2 嵌入式系统软件

随着微电子技术的发展，微控制器片内集成的外设越来越多，而外围电路的功能则相对比较固定，因此在嵌入式系统开发中，软件设计所占的工作量越来越大。

传统嵌入式软件框架由驱动程序、操作系统（可选）和应用程序三个层次组成。

- 驱动程序

主要控制硬件模块的工作，包括嵌入式处理器的片内外设及外围电路，为操作系统和应用程序提供接口函数。

- 操作系统

负责嵌入式系统的进程调度和任务管理，为应用程序提供接口函数。用户可以根据系统的复杂程度选择是否加入操作系统。

- 应用程序

根据实际的需求，完成具体的业务逻辑。访问硬件时，需要调用驱动程序提供的接口函数。

传统的嵌入式软件框架结构如图 1-5 所示。

不同于通用的计算机系统，其硬件平台是标准且兼容的。嵌入式硬件平台往往针对不同的应用场合，选用的处理器类型及外围电路的设计各不相同，即使是功能相同的应用程序，如果运行在不同的嵌入式硬件平台上，都需要重新设计。相对于通用计算机系统的软件开发而言，嵌入式软件开发的工作量和难度都要大很多。

为了减少嵌入式软件移植的工作量和难度，需要在原来的嵌入式软件框架中增加一个中间层：硬件抽象层（Hardware Abstraction Layer，HAL），有时也称为板级支持包（Board Support Package，BSP）。这个新增加的中间层位于驱动程序之上，它能够通过特定的上层接口与应用程序和操作系统进行交互，向它们提供底层硬件信息，并根据它们的要求完成对硬件的直接操作。此时，传统的嵌入式软件框架结构演变为四个层次，如图 1-5 所示。

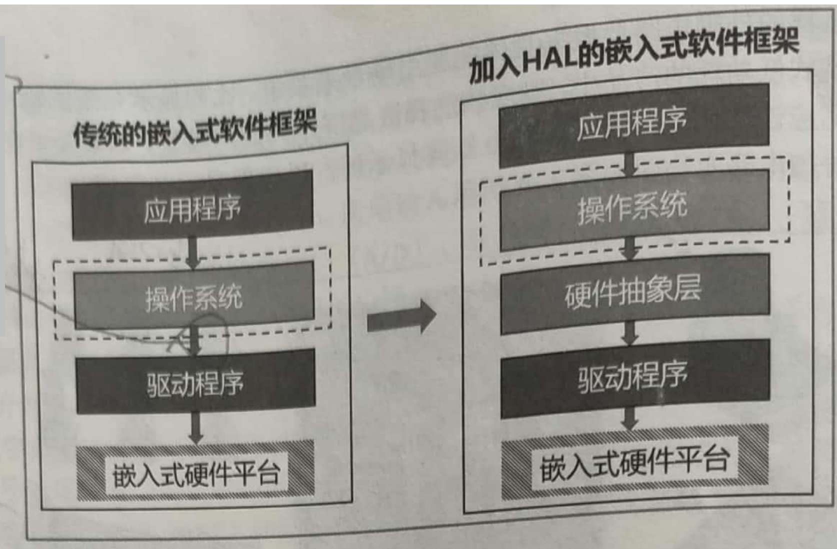


图 1-5　两种嵌入式系统软件框架

以控制指示灯为例，传统的嵌入式软件框架中一般采用直接访问硬件寄存器或调用库函数的形式通过控制 GPIO 模块实现对指示灯的初始化、开启和关闭等操作。当嵌入式硬件平台发生变化时，如更换微控制器芯片或者修改指示灯控制引脚，则应用程序和操作系统中与指示灯控制相关的代码都必须修改。

当应用程序和操作系统基于 HAL 进行设计时，我们只需要设计几个通用的指示灯控制接口函数：

- BSP_LED_Init ：用于初始化指示灯控制引脚
- BSP_LED_On ：用于开启指示灯
- BSP_LED_Off ：用于关闭指示灯

应用程序和操作系统控制指示灯时，只需要调用 HAL 提供的三个接口函数，而不考虑接口函数的具体实现。当嵌入式硬件平台发生变化时，只需要修改硬件抽象层所提供的接口函数本身，而不必修改应用程序。

通过引入硬件抽象层，屏蔽了底层硬件的多样性，应用程序和操作系统不再面对具体的硬件环境，而是面对由这个中间层次所代表的、逻辑上的硬件环境。这种软件框架减少了嵌入式软件的开发难度和工作量，推动了嵌入式系统软件的通用化。

1.3　嵌入式操作系统

1.3.1　编程模式

嵌入式系统的软件开发一般有两种编程模式：前后台系统和嵌入式操作系统。开发人员可以根据系统的复杂程度来选择对应的编程模式。

1. 前后台系统

对于控制流程简单，任务固定的嵌入式系统而言，可以采用前后台系统。整个程序框架由一个死循环和若干个中断服务程序组成。

- 后台系统

一个无限循环，循环中调用相关函数完成所需的操作。

- 前台系统

若干个中断服务程序，用于处理系统的异步事件和实时性要求较高的任务。

在顺序执行后台程序的时候，如果有中断来临，那么中断会打断后台程序的正常执行流

程，转而执行中断服务程序。在中断服务程序中进行事件的处理：如果该事件的处理时间较短，则可以直接在中断服务程序里面处理；如果该事件较为复杂，处理时间较长，则设置事件发生的标志，具体的处理过程交给后台程序完成，以确保中断的快速响应。

以日常使用的豆浆机为例，只需要加热、预打浆、打浆、煮浆、断电报警等五个流程，各个流程之间是线性的先后关系，没有并发任务，使用简单的循环程序就可以了。这样的系统就可以采用前后台系统的编程模式进行开发。

采用前后台系统时，程序结构简洁，设计人员能够全盘掌控程序的运行流程，方便程序的调试。因此对大多数的中小型项目而言，一般建议使用前后台系统。

前后台系统的程序运行流程如图 1-6 所示。

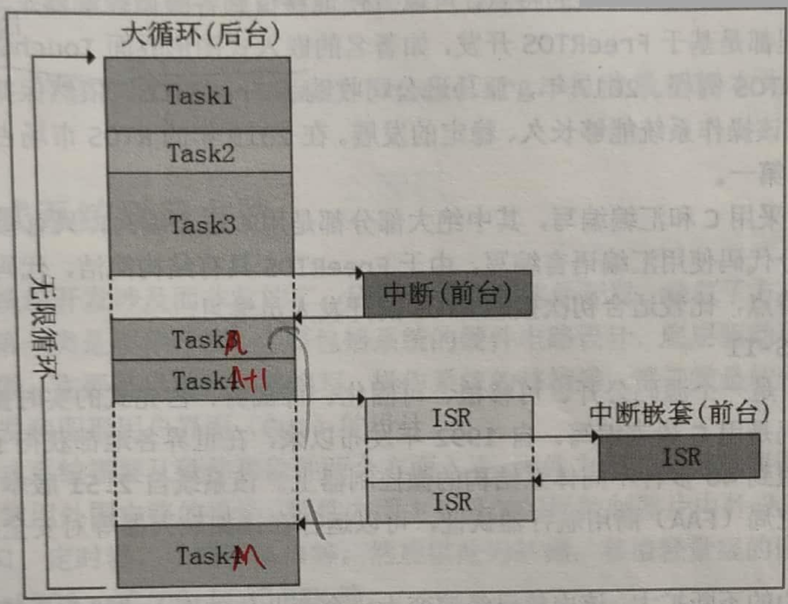


图 1-6　前后台系统运行流程示意图

2.　嵌入式操作系统

对于功能复杂、外围电路较多，各个任务之间交互频繁的嵌入式系统而言，则一般基于嵌入式操作系统进行开发。

对于一个复杂的应用程序，通常可以按照软件工程中的任务划分原则将整个程序分解为多个独立的任务模块，每个任务模块单独编写和调试，任务之间的调度和通信由嵌入式操作系统完成，这样的编程模式可以简化应用程序的设计过程，缩短开发时间，更好地保证系统的实时性和可靠性。

除此以外，嵌入式操作系统还提供任务管理、任务通信（事件标志、信号量和邮箱）、时间管理，内存管理等接口函数，进一步方便了程序的开发。

1.3.2　常用的实时操作系统

嵌入式操作系统历史悠久、门类繁多、用途广泛，不仅包括实时操作系统（Real Time Operation System，RTOS）、开源的 Linux 系统、机器人和路由器操作系统，还包括新型的物联网操作系统，以及边缘计算操作系统平台。这里主要介绍一下适合微控制器使用的实时操作系统。

实时操作系统在实际使用中有实时内核和实时操作系统的区分：

- 实时内核

RTOS: Real-Time Operating

管理微控制器全部资源的软件，提供任务管理、时间管理和任务通信等基本的系统服务。

● 实时操作系统

包含实时内核和其他高级服务，如文件系统，协议栈，图形用户界面等组件。

1. FreeRTOS

FreeRTOS 是一个小型的实时操作系统内核，提供了实时操作系统所需的所有功能：任务管理、时间管理、信号量、消息队列、内存管理等，可基本满足中小规模的嵌入式系统需求。FreeRTOS 的官方网站为：www.freertos.org，目前最新版本为 10.2.10 版（截止 2019/11）。

相对 μC/OS-II 等商业操作系统，FreeRTOS 是完全免费的操作系统，具有源码公开、可移植、可裁减和调度策略灵活的特点，可以方便地移植到各类微控制器上运行。很多厂商提供的软件例程都是基于 FreeRTOS 开发，如著名的嵌入式图形界面 TouchGFX 和意法半导体公司提供的 RTOS 例程。2017 年，亚马逊公司收购了 FreeRTOS，依然保持了免费和开源的特点，也确保该操作系统能够长久、稳定的发展。在 2019 年的 RTOS 市场占用率统计中，FreeRTOS 排名第一。

FreeRTOS 采用 C 和汇编编写，其中绝大部分都是用 C 语言编写，只有极少数与处理器密切相关的部分代码使用汇编语言编写。由于 FreeRTOS 具有结构简洁，代码可读性强，开发文档完善等特点，比较适合初次接触 RTOS 的开发人员学习。

2. μC/OS-II

μC/OS-II 是一个源码公开、可移植、可固化、可裁剪、占先式的实时操作系统内核。其绝大部分源码是用 C 语言编写。自 1992 年发布以来，在世界各地都获得了广泛的应用，目前已经被移植到 40 多种不同体系结构的微控制器上。该系统自 2.51 版本之后，就通过了美国联邦航空局（FAA）商用航行器认证，可以运行在诸如航天器等对安全要求极为苛刻的系统之上。

随着影响力的不断扩大，该内核已经演变为一个商业化的嵌入式操作系统，并提供一整套的应用解决方案：包括图形用户界面（μC/GUI），文件系统（μC/FS），网络协议（μC/NET）和 USB 协议（μC/USB）等。μC/OS-II 目前已经发展到 μC/OS-III，从 μC/OS-III 开始，该内核不再公开源码，而是以库的形式提供，并增加了很多新的特性，例如在同一优先级下支持任务的轮转调度以及没有任务数量的限制等。

μC/OS 的官方网站为：www.micrium.com，目前最新版本为 3.07.03（截止 2019/11）。

3. RTX（CMSIS-RTOS）

RTX 是一个源码公开的实时内核，它由德国 Keil 公司（现已被 ARM 公司收购）开发，配套于业界主流的嵌入式开发软件 RealView Microcontroller Development Kit（以下简称 MDK）使用，可用于 ARM7、ARM9 和 Cortex-M 等内核的微控制器。它提供时间片轮转和优先级抢占两种任务调度方式，支持信号量、互斥量、定时器、事件标志和邮箱等系统服务。

由于嵌入式操作系统数量众多，各个嵌入式操作系统提供的应用程序接口（API）各不相同。如果用户更换了嵌入式操作系统，往往需要对代码进行大量的修改。为此，ARM 公司设计了一个标准化的实时操作系统编程接口 CMSIS-RTOS，原有的实时操作系统经过 CMSIS-RTOS 的二次封装后，就可以为用户提供统一的接口。RTX 经过封装后，以 CMSIS-RTOS V2 的形式提供，其源代码在安装了 MDK 开发软件后，可在安装目录下获得：\Pack\ARM\CMSIS\5.5.1\CMSIS\RTOS2。

CMSIS-RTOS 的相关资料可以在 Keil 公司的官方网站上获取，网址为：www.keil.com，

目前最新版本为 2.1.3（截止 2019/11）。

4. RT-Thread

RT-Thread 是一款来自国内的开源实时操作系统，由国内 RT-Thread 开发团队进行开发与维护。它不仅是一款高效、稳定的实时操作系统内核，也是一个组件完整丰富、简易开发、超低功耗、高安全性的物联网操作系统，具备物联网平台所需的所有关键组件，例如图形用户界面、网络协议栈、安全传输、低功耗组件等。

RT-Thread 起初只有一个实时内核，但在发展过程中，得到了来自全国嵌入式开发工程师的鼎力支持，为 RT-Thread 添砖加瓦，现在它已慢慢演变成一个完善的、全功能的操作系统。经过多年的累积发展，RT-Thread 被广泛应用于能源、车载、医疗、消费电子等多个行业，累积装机量超过两千万台，已成为国人自主开发、国内最成熟稳定和装机量最大的开源 RTOS。

RT-Thread 的官方网站为：www.rt-thread.org，目前最新版本为 3.1.3（截止 2019/11）。

1.4 嵌入式系统学习方法

嵌入式系统的开发涉及面非常的广，从硬件设计到软件开发，涵盖了方方面面。一般可以分成三类：第一类是硬件开发，主要包括系统的硬件电路设计，底层驱动程序的设计；第二类是系统开发，主要是启动代码的编写，操作系统的移植等；第三类是软件开发，主要是应用程序的开发和图形用户界面（GUI）的设计。

学习嵌入式系统需要从硬件和软件两个方面入手。硬件方面主要是掌握微控制器最小系统的设计以及常用外围电路的设计。软件方面主要是学习微控制器片内外设的应用，如通用输入/输出接口、定时器、中断和串口等，然后以此为基础，移植轻量级的嵌入式操作系统内核，如 FreeRTOS、μC/OS-II 和 RTX 等。

在学习微控制器的编程方法时，也有两种方式：

- 寄存器方式

从微控制器底层开始学习，学习微控制器的内部结构，各个功能模块，掌握每一个寄存器的应用，完全自己编写底层驱动程序。这种方法可以对微控制器有全面的了解，编写出执行效率高的程序。但是学习难度较大，开发效率较低，更换新的微控制器时，需要完全重新学习。

- 库函数方式

使用芯片厂家提供的固件库来进行编程。用户不用去掌握处理器的内部结构以及片内外设的相关寄存器，只需要了解库函数的接口参数即可，并且可以和第三方软件如嵌入式操作系统和图形用户界面无缝衔接。这种方式可以减轻用户的开发难度，提高工作效率。但是这种开发方式要求用户有较高的 C 语言编程能力，能熟练掌握指针、结构体和位运算等相关操作。

从目前产业界发展来看，越来越多的公司推出了跨平台的开发方式。ARM 公司主推的 Mbed 平台，抽象了微控制器底层硬件的差异，设计了一套可以适用于各种微控制器的开发平台，并提供在线编译器。Keil 公司的 RTE Driver 也力图提供一套适用于所有微控制器的接口函数。因此库函数的开发方式已经成为产业界的主流开发方式。

从学习的角度出发，我们建议初学者从寄存器入手，用寄存器开发的方式去学习微控制器内部的一个功能模块，再以此为基础来学习库函数的开发方式。

本书在介绍 GPIO 模块时，将采用寄存器的开发方式，介绍 GPIO 模块各个寄存器的含义和使用方法，然后再引入官方的库函数，并引导大家尝试编写自己的库函数，从而掌握从寄存器开发到库函数开发的全过程。

1.5 本章小结

本章的主要目的是建立嵌入式系统的基本概念。从嵌入式、专用性和计算机系统这三个特点去理解嵌入式系统，特别注意它和通用计算机系统的差异，可以结合生活中的一些实例看看嵌入式系统在我们生活中的应用。

从硬件和软件两个方面掌握嵌入式系统的组成。硬件上掌握 MCU 和 CPU 的区别。软件上了解前后台编程模式的基本流程，这也是本书后续章节中采用的基本编程模式。同时深入体会硬件抽象层的设计思想，这是成为一个优秀嵌入式开发人员应具备的设计理念。

习题 1

1. 嵌入式系统的概念是什么？
2. 嵌入式系统有哪些特点？
3. 举例说明嵌入式的应用领域。
4. 嵌入式系统硬件由哪些部分组成？
5. 嵌入式系统软件由哪些部分组成？
6. 请从组成、用途和开发方式等方面比较通用计算机系统和嵌入式系统的区别。
7. 硬件抽象层接口的定义和代码设计有哪些特点？
8. 分析寄存器开发方式和库函数开发方式的优缺点。

第二章　STM32F4 系列微控制器的硬件电路设计

本章导读

本章重点介绍 STM32F4 系列微控制器的硬件电路设计。首先概要介绍了 STM32 微控制器的产品线、STM32F4 系列微控制器的特点以及 STM32F411 芯片的引脚功能。然后以官方开发板 Nucleo 为例，从最小系统电路、外围电路和仿真器电路等三个方面详细介绍开发板的电路设计原理。

本章知识点

STM32 微控制器基本特点

微控制器最小系统的组成及设计方法

常见的外围电路

2.1　STM32 微控制器概述

STM32 由意法半导体公司（以下简称 ST）生产的 32 位微控制器，基于 ARM 公司的 Cortex-M 系列内核。主要面向高性能、低成本和低功耗的嵌入式设计，广泛应用于工业控制、消费电子、物联网、通讯设备、医疗服务和安防监控等领域。

STM32 微控制器拥有丰富的产品系列，按照不同的用途可以分成四类：高性能、主流、低功耗和无线，各系列的基本特点如表 2-1 所示。

表 2-1　STM32 微控制器产品

分类	型号	内核	特点
主流型	STM32F0	Cortex-M0	低成本、入门级微控制器，高性价比，最高主频 48MHz
	STM32F1	Cortex-M3	通用型微控制器，最高主频 72MHz
	STM32F3	Cortex-M4	侧重混合信号应用，最高主频 72MHz
	STM32G0	Cortex-M0+	F0 系列升级产品，最高主频 64MHz
	STM32G4	Cortex-M4	F3 系列升级产品，最高主频 170MHz
高性能	STM32F2	Cortex-M3	通用型微控制器，最高主频 120MHz
	STM32F4	Cortex-M4	通用型微控制器，最高主频 180MHz
	STM32F7	Cortex-M7	通用型微控制器，最高主频 216MHz
	STM32H7	Cortex-M7	通用型微控制器，最高主频 400MHz
低功耗	STM32L0	Cortex-M0+	低功耗应用，最高主频 32MHz
	STM32L1	Cortex-M3	低功耗应用，最高主频 32MHz
	STM32L4	Cortex-M4	低功耗应用，最高主频 80MHz
	STM32L4+	Cortex-M4	低功耗应用，最高主频 120MHz
	STM32L5	Cortex-M33	低功耗应用，提高了安全性，最高主频 110MHz
无线	STM32WB	Cortex-M4 Cortex-M0+	双核微控制器，侧重无线应用，支持 2.4G 无线通信、蓝牙 5.0 和 IEEE 802.15.4 无线标准，最高主频 64MHz

STM32 微控制器的引脚具备高度的兼容性。对于相同封装的芯片，除少数电源与新增功能引脚有区别外，大部分引脚的功能都是基本相同的。用户可以在不修改电路的条件下，根

据需要更换不同片内资源和不同主频的芯片，方便系统的升级换代。

STM32 微控制器包含了不同的产品系列，对于初学者来讲，一般建议选择 STM32F1 和 STM32F4 两个系列来学习。因为这两个系列是目前产业界应用的主流芯片，上市时间长，相关技术文档、书籍和网络资源都比较丰富。而 STM32F4 系列相对于 F1 系列而言，增加了浮点运算单元（FPU）和数字信号处理（DSP）指令，并且内核性能、工作频率、存储器容量和片内外设数量等方面都有了大幅度的提升，因此本书将以 STM32F4 系列微控制器作为讲授重点。

2.2 STM32F4 系列微控制器

2.2.1 概述

STM32F4 系列芯片基于 ARM Cortex-M4 内核，片内集成单精度浮点运算单元（FPU），具备增强的数字信号处理（DSP）指令集，工作频率高达 180MHz，属于 STM32 产品系列中的高性能 MCU。

按照性能的不同，STM32F4 系列 MCU 分成了三种类型：

- 入门型

特点是卓越的动态效果，高性价比，包括 STM32F401、STM32F410、STM32F411、STM32F412、STM32F413/423 等型号。

- 基本型

特点是高主频、全功能及加密功能，包括 STM32F405/415、STM32F407/417、STM32F446 等型号。

- 高级型

特点是增强的图像处理性能、高速显示接口和大容量存储空间，包括 STM32F427/437、STM32F429/439、STM32F469/479 等型号。

本书将以 STM32F4 系列的入门型芯片为主来介绍嵌入式系统的开发和应用，其他系列芯片的用法与之相仿。

2.2.2 STM32F4 入门型芯片

STM32F4 系列入门型芯片，侧重于功耗和成本敏感的领域，具有动态功耗小，性能和集成度高，提供丰富的连接外设等特点。与其他产品系列相比，加入了很多新的特性：

- 采用 STM32 动态效率技术（Dynamic Efficiency），能够实现更低的动态功耗和更高性能的完美结合：运行模式（89uA/MHz），停止模式（10uA）
- 支持数据的批量数据模式（BAM），优化了传感器数据批处理的功耗
- 存储器保护单元（MPU），可以实施对存储器（主要是内存和外设寄存器）的保护，避免程序意外修改存储器内容，以使软件更加稳定可靠
- 真随机数发生器，增强芯片安全性能

STM32F4 系列入门型芯片，片内最大集成 1.5MB 的 Flash 存储器和 320KB 的 SRAM，提供从 36～144 引脚的多种封装。

片内集成外设包括：

- 10 路 USART，速度高达 12.5Mbit/s
- 5 路 SPI（I^2S 复用），速度高达 50Mbit/s
- 4 路 I^2C，速度高达 1Mbit/s
- 1 个 SDIO（SD 卡接口），频率高达 48MHz

- 1 个 USB 2.0 OTG（全速）
- 2 路全双工 I^2S 和 3 路单工 I^2S，速度高达 32 位/192KHz
- 多达 3 路 CAN
- 转换速率高达 2.4MHz 的 12 位 ADC
- 频率高达 100MHz 的 16 和 32 位定时器
- STM32F412/413 具备 16 位数据总线（FSMC），可扩展外部存储器

STM32F4 系列入门型芯片又分成了 5 类，各类芯片的片内资源如表 2-2 所示。

表 2-2　STM32F4 入门型芯片选型列表

器件	工作频率 (MHz)	Flash (KB)	SRAM (KB)	FSMC 总线	QSPI	CAN 2.0B	DAC	USB2.0 OTG FS
STM32F401	84	128～512	up to 96	–	–	–	–	√
STM32F410	100	64～128	32	–	–	–	√	–
STM32F411	100	256～512	128	–	–	–	–	√
STM32F412	100	512～1024	256	√	√	√	–	√
STM32F413/423	100	1024～1536	320	√	√	√	√	√

2.3　STM32 Nucleo 开发板

ST 公司为了方便用户进行芯片功能的验证和产品原型的快速开发，设计了官方开发板，主要分为 3 类，如图 2-1 所示。

图 2-1　意法半导体官方开发板

- 评估板（Evaluation board）

评估板是功能最齐全的一类开发板，主要用来对微控制器的功能做全面的评估，其特点是集成了丰富的外设资源，如外扩的存储器（SRAM 和 Flash）、TFT 液晶屏、音频接口、CAN 总线和网络接口等。

- 探索套件板（Discovery kits）

探索套件板在评估板的基础上进行了功能的裁剪，注重于对芯片重点功能的测试，如图形处理功能等。

● **Nucleco 开发板（Nucleo board）**

Nucleo 开发板是在探索套件板的基础上进行了改进，提供了兼容 Arduino 外设的接口，使用户能够方便地使用 Arduino 生态圈的数百种外设。Nucleo 开发板涵盖了 ST 微控制器的全部产品线，用户可以快速地验证芯片性能，并且价格便宜，购买渠道多，是目前 ST 公司主推的官方开发板。

2.3.1 Nucleo 开发板概述

STM32 Nucleo 开发板是 ST 公司于 2014 年推出的低成本开发平台，按照板载主控芯片的引脚数量分成了 Nucleo32、Nucleo64 和 Nucleo144 三个子系列。各个型号的 Nucleo 开发板在电路结构上有高度的兼容性，掌握了其中一种型号的 Nucleo 开发板之后，能够快速入门其他型号的开发板，真正做到一板多用。

Nucleo 开发板的基本特点如下：

● 模块化设计

将 ST-Link 仿真器（版本 V2-1）和应用电路集成在一块电路板上，仿真器同时支持对外部 MCU 的调试。

● 统一的 MCU 核心板设计

各系列开发板的 MCU 核心板采用统一的电路设计，除 MCU 型号不同外，外围电路和接口均一致，各开发板之间的程序也可以无缝移植。

● 丰富的接口扩展

提供 Arduino UNO 扩展板接口，方便用户接入 Arduino 生态圈的各类扩展板，并将微控制器所有 I/O 口通过排针引出，方便用户进行二次开发。

● 简洁的外设模块

一般只提供一个用户指示灯和一个用户按键。

● 多功能的 USB 接口

USB 接口实现 3 个功能：虚拟串口、大容量存储（Mass Storage）和调试接口（SWD）。

● 灵活的供电方式

可以通过两种方式供电： USB 接口供电和外部接口供电。

● 支持多种开发环境

支持 Keil MDK-ARM、IAR EWARM、Mbed 等多种集成开发环境。

2.3.2 Nucleo-F411RE 开发板

本书所采用的 Nucleo 开发板型号为：Nucleo-F411RE，开发板上搭载的主控芯片是 STM32F411RET6，芯片引脚数量为 64，属于 Nucleo64 系列。

【小贴士】STM32 芯片命名规则

STM32 的每一类芯片，按照芯片的封装形式以及片内存储器的大小分为了不同的子型号，以后缀作为区分，如 STM32F411xxyy。

第一个 x 代表引脚数：T—36pin，C—48pin，R—64pin，V—100pin，Z—144pin；

第二个 x 代表 Flash 容量：6—32K，8—64K，B—128K，C—256K，D—384K，E—512K；

第一个 y 代表封装：H—BGA 封装，T—LQFP 封装，U—QFN 封装；

第二个 y 代表工作稳度范围：6 代表—-40 到 85 摄氏度，7 代表—-40 到 105 摄氏度。

芯片型号 STM32F411RET6 就代表一款引脚数量为 64、片内 Flash 为 512KB、封装为 LQFP、工作温度为-40～85 摄氏度的芯片。

Nucleo 开发板分为上、下两个部分：上半部分是 ST-Link 仿真器，下半部分是 MCU 的最小系统及外围电路。两者之间采用邮票孔形式连接。这种连接形式的好处是：ST-Link 仿真器可以裁剪下来作为独立的仿真器，去仿真外部 MCU。

具体的实物电路及板上资源分布如图 2-2 所示。

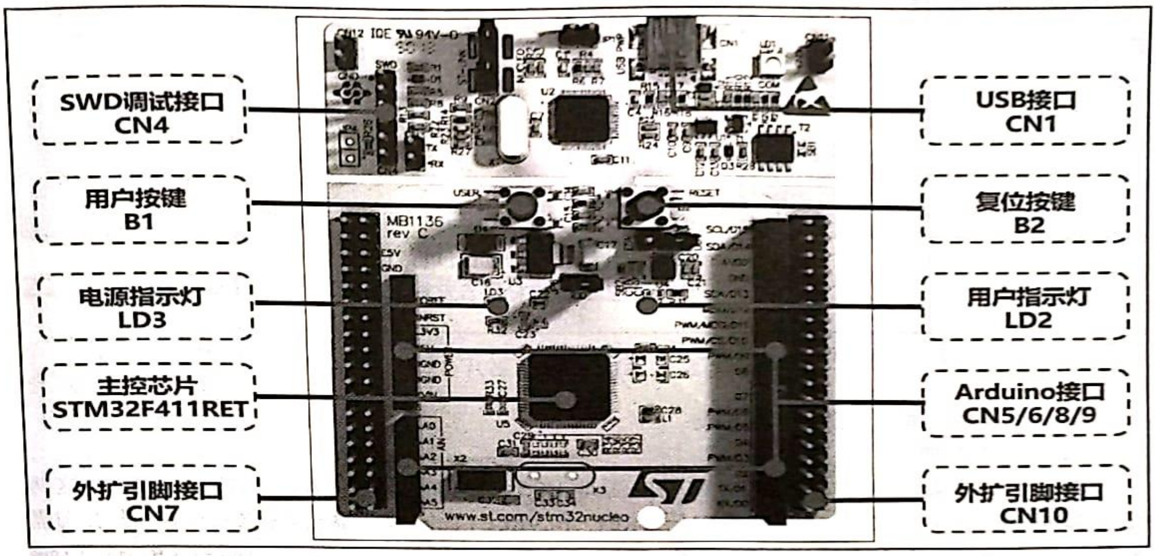


图 2-2　Nucleo-F411RE 开发板及板上资源分布图

Nucleco-F411RE 开发板包含的硬件资源如下：

- CPU：STM32F411RET6，LQFP64，Flash：512K，SRAM：128K，主频：100MHz
- 1 个 USB 接口 CN1，集成 ST-Link 仿真器，可用于程序下载、仿真和串口通信
- 1 个 SWD 调试接口 CN4，用于调试外部 MCU
- 1 个电源指示灯 LD3（红色）；1 个用户指示灯 LD2（黄色）
- 1 个复位按键 B2（黑色键帽）；1 个用户按键 B1（蓝色键帽）
- Arduino UNO 扩展接口 CN5、CN6、CN8 和 CN9
- ST Morpho 扩展接口 CN7 和 CN10，引出主控芯片的全部 I/O 引脚

2.3.3　STM32F411RET 芯片引脚介绍

Nucleco-F411RE 开发板上搭载的主控芯片为 STM32F411RET6，该芯片为 32 位的 ARM Cortex-M4 内核，属于 STM32F4 系列的入门级芯片。片内集成单精度浮点运算单元（FPU），具备增强的数字信号处理（DSP）指令集，工作频率为 100MHz，片内主要资源如下：

- 512KB Flash 和 128KB SRAM
- 1 个 16 位高级定时器，2 个 32 位通用计时器，5 个 16 位通用定时器
- 3 路 USART；5 路 SPI/I^{2}S；3 路 I^{2}C；1 路 SDIO（SD 卡接口）
- 1 路 12 位 16 通道 ADC
- 1 个全速 USB 2.0 OTG
- 50 个通用数字 I/O 口

该芯片一共 64 个引脚，大部分引脚都有复用功能，引脚布局如图 2-3 所示。

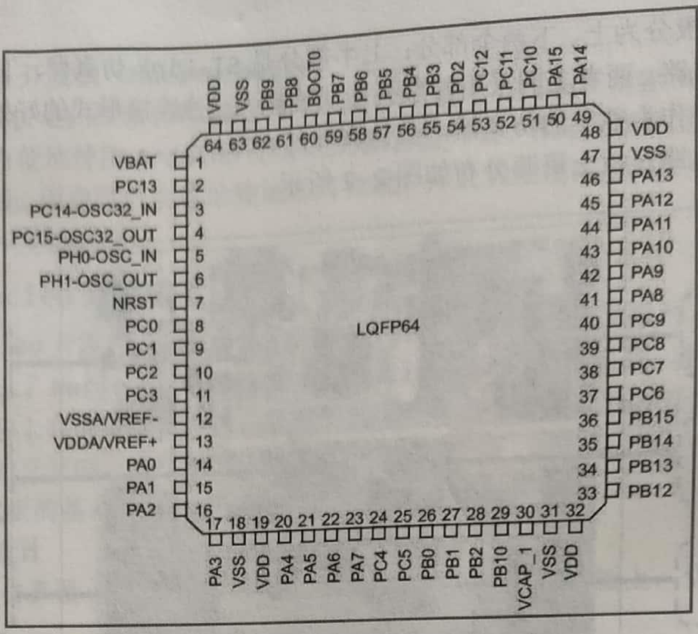


图 2-3　STM32F411 芯片引脚图

按照引脚的功能，可以分为五类：

1. 电源引脚

电源引脚主要分为主电源 VDD 和主电源地 VSS、模拟电源 VDDA 和模拟地 VSSA（这两个引脚同时也作为 ADC 模块的参考电压 VREF-和 VREF+）、调节器电源 VCAP_1（该引脚需要外接一个电容，用于内核电源的滤波，电容大小由芯片型号决定），实时时钟（Real Time Clock，RTC）供电引脚 VBAT（Pin1）。

2. 复位及启动模式引脚

复位引脚 NRST（Pin7），用于复位信号的输入，该引脚为低电平时将器件复位，并使 I/O 口和片内外设恢复至默认状态，MCU 将从存储器地址 0 开始执行。

启动模式引脚 BOOT0（Pin28）和 BOOT1（Pin60）用于设置 MCU 的启动模式。

3. 时钟引脚

时钟引脚分成了 2 类：一个是外部高速时钟引脚：OSC_IN（Pin5）和 OSC_OUT（Pin 6）；另一个是外部低速时钟引脚：OSC32_IN（Pin3）和 OSC32_OUT（Pin4）。

4. 仿真调试引脚

仿真调试引脚分成了 2 类：一类是常用的 JATG 接口，使用 JTMS（Pin46）、JTCK（Pin49）、JTDI（Pin50）、JTDO（Pin55）、JTRST（Pin56）这 5 个引脚；一类是串行调试接口（SWD），使用 SWDIO（Pin46）、SWCLK（Pin49）和 SWO（Pin55）。由于 SWD 接口只需要 2 个引脚（SWO 可选），因此在工程应用中多采用 SWD 接口。

注意：仿真调试引脚和部分通用 I/O 引脚复用，如果使用了仿真调试功能，则这些引脚就不能作为通用 I/O 引脚。

5. 通用数字 I/O 引脚

STM32F411RET6 一共有 50 个 I/O 引脚，通常和其他外设功能复用，分成了 PA0～PA15、PB0～PB15（PB11 引脚被 VCAP_1 引脚取代）、PC0～PC15 等 3 组引脚，以及 PD2、PH0 和 PH1 等 3 个独立引脚。每一个 I/O 引脚都具备外部中断功能。

50 个 I/O 引脚中，PH0 和 PH1 用于接入外部高速时钟，PC14 和 PC15 用于接入外部低速时钟，PA13 和 PA14 用于 SWD，实际留给用户使用的 I/O 引脚只有 44 个。

【小贴士】芯片手册

学习任何一款 STM32 芯片时，常常会用到以下 2 个官方文档：

参考手册（Reference Manual）

芯片每个模块的具体描述和功能介绍，比如我们要查询 GPIO、Timer、UART 等模块的相关寄存器和功能介绍就可以使用这个手册，一般用于软件开发。

数据手册（Data Sheet）

介绍芯片引脚定义、电气特性和机械封装等，一般用于硬件电路设计。

这些文档可以在以下几个官方网站查找：

ST 英文官网：www.st.com

ST 中文官网：www.stmcu.com.cn

ST 中文社区：www.stmcu.org.cn

2.4 Nucleo-F411RE 开发板电路原理

按照功能模块的划分，Nucleo-F411RE（以下简称 Nucleo）开发板的硬件电路可以分为 3 个模块：

1. 最小系统
2. 外围电路
3. ST-Link 仿真器

2.4.1 最小系统电路设计

微控制器的最小系统指的是使 MCU 具备基本运行条件所需的外围电路，一般由电源电路、时钟电路、复位电路和调试电路等四部分组成。所有的微控制器电路都需要这 4 个模块才能运行，其电路设计具备一定的通用性。

1. 电源电路

电源电路为整个电路系统提供能量，是整个系统工作的基础，具有极其重要的地位。

Nucleo 开发板可以使用 USB 接口供电，也可以采用外部接口供电，其电源电路如图 2-4 所示。

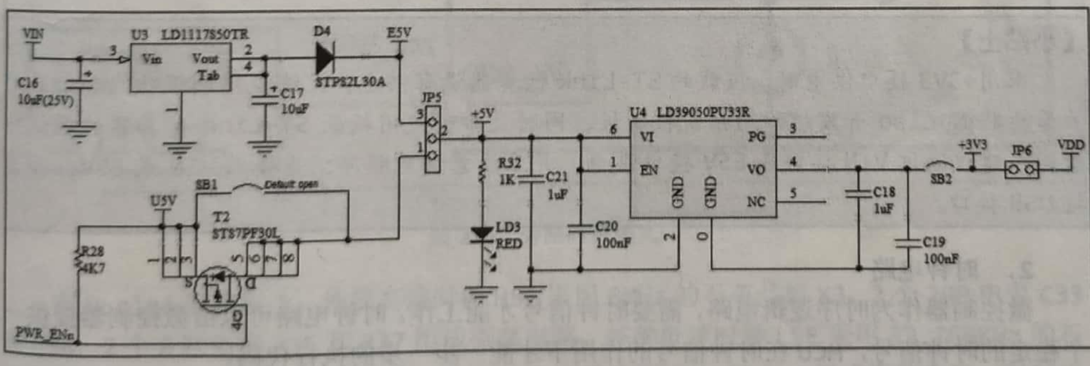


图 2-4 Nucleo-F411RE 开发板电源电路

(1) USB 接口供电（短接 JP5 的引脚 1 和引脚 2）

Nucleo 开发板可以通过 USB 接口 CN1（网络标号 U5V）供电。当开发板通过 USB 连接线和电脑的 USB 接口连接时，经过 USB 枚举过程，板载 ST-Link 仿真器将使能 PWR_ENn

信号（低电平有效），PWR_ENn 信号控制 P 沟道的 MOS 管 T2（STS7PF30L）导通。如果将 JP5 的引脚 1 与引脚 2 之间的跳线短接，U5V 将与+5V 连接，使电源指示灯 LD3 点亮。

+5V 电压通过低压差线性稳压芯片 U4（LD39050PU33R）转换成+3.3V，并通过 JP6 连接至 VDD，向 Nucleco 开发板的 MCU 及外围电路供电。

LD39050PU33R 是低压差（LDO）的线性稳压芯片，固定输出 3.3V 电源，最大输出电流为 500mA。和传统的 78 系列稳压芯片相比，它只需要输入电压和输出电压之差大于 0.4V 即可，这样可以减少稳压芯片上的功率消耗，不用增加散热片，减小了电路体积。在稳压芯片的输入端和输出端分别加入 C20、C21、C18 和 C19 四个电容进行滤波，滤除高频和低频信号的干扰，保证输出电压的稳定。

注意：使用电脑的 USB 接口供电时，才会有 USB 枚举过程。使用 USB 接口的充电器或者插座时，由于没有 USB 枚举过程，PWR_ENn 信号无效，MOS 管无法导通，开发板将无法正常供电。如果用户需要采用 USB 接口的充电器或者插座供电时，需要短接焊桥 SB1（焊接 0 欧电阻），以便跳过 MOS 管的导通环节。

(2) 外部接口供电（短接 JP5 的引脚 2 和引脚 3）

Nucleco 开发板可以通过三个外部接口供电：VIN、E5V 和+3V3，这三个接口均位于外扩接口的插针/插座上。

- VIN 接口供电

电源 VIN 位于外扩接口 CN6 的 8 脚和 CN7 的 24 脚，可加入 7～12V 的直流电压。经过低压差线性稳压芯片 U3（LD1117S50TR）后，输出恒定的 5V 电压。该芯片最大支持的输出电流为 800mA。

- E5V 接口供电

电源 EV5 位于外扩接口 CN7 的 6 脚，可加入 5V 的直流电压。EV5 与 U3 之间有一个二极管 D4，用于防止电流倒灌至 U3，引起 U3 的损坏。

当 JP5 的引脚 2 和引脚 3 短接后，上述两种外部接口的输出电压将与+5V 连接，使电源指示灯 LD3 点亮，并通过 U4 转换成 3.3V，向 Nucleco 开发板的 MCU 及外围电路供电。

- +3V3 接口供电

电源+3V3 位于外扩接口 CN6 的 4 脚和 CN7 的 16 脚，可加入 3.3V 的直流电压，直接向 Nucleco 开发板的 MCU 及外围电路供电。

【小贴士】

采用+3V3 接口供电时，板载的 ST-Link 仿真器没有供电，无法使用调试功能。如果用户需要将 Nucleo 开发板作为系统控制板，同时又希望使用板载 ST-Link 仿真器的调试功能时，建议通过 VIN 接口或 E5V 接口供电，并且注意供电顺序：先接入 VIN 或 E5V，再连接 USB 接口。

2. 时钟电路

微控制器作为时序逻辑电路，需要时钟信号才能工作。时钟电路可以给微控制器提供一个稳定的时钟信号，MCU 在时钟信号的作用下才能一步一步的执行代码。

STM32F4 系列微控制器包括 4 个时钟源：

- LSI（Low Speed Internal）

内部低速时钟。由芯片内部的 RC 振荡器提供，默认频率为 32KHz。

- HSI（High Speed Internal）

内部高速时钟。由芯片内部的 RC 振荡器提供，默认频率为 16MHz。

● HSE（High Speed External）

外部高速时钟。通过在 OSC_IN 和 OSC_OUT 引脚接入晶振实现，频率范围为 4MHz～26MHz。也可以直接接入外部时钟信号，频率范围为 1MHz～50MHz。

● LSE（Low Speed External）

外部低速时钟。通过在 OSC32_IN 和 OSC32_OUT 引脚接入 32.768KHz 的晶振实现。

在 MCU 复位后，默认使用内部 RC 振荡器提供时钟，由于该时钟精度较低，因此实际应用中通常通过软件切换至 HSE 时钟。

时钟电路的设计一般有两种方式：

● 振荡模式

在该模式下，在 MCU 外部连接一个晶振和两个电容，利用片内集成的反相器，就可以形成振荡电路，进而产生 MCU 所需的时钟信号，电路示意图如图 2-5 所示。

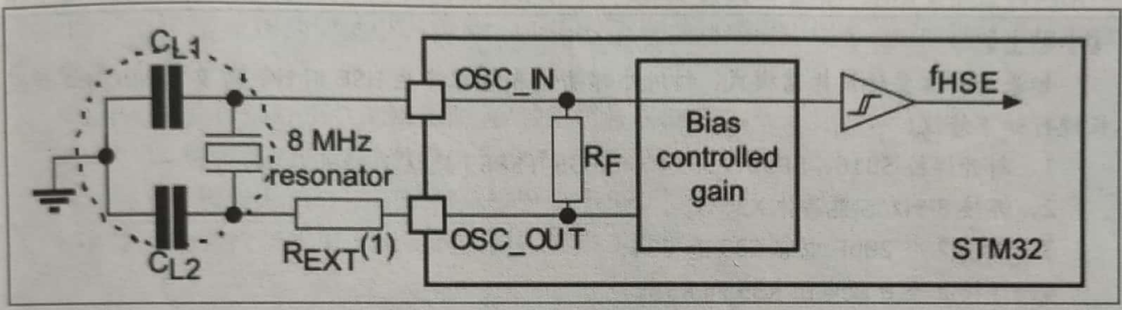


图 2-5　振荡模式

● 旁路时钟模式（Bypass）

在该模式下，直接由外部电路输入一个稳定的时钟信号（占空比约为 50%的正弦波、方波或三角波），频率范围为 1MHz～50MHz。时钟信号送入 OSC_IN 引脚，OSC_OUT 引脚保持高阻状态。如图 2-6 所示。

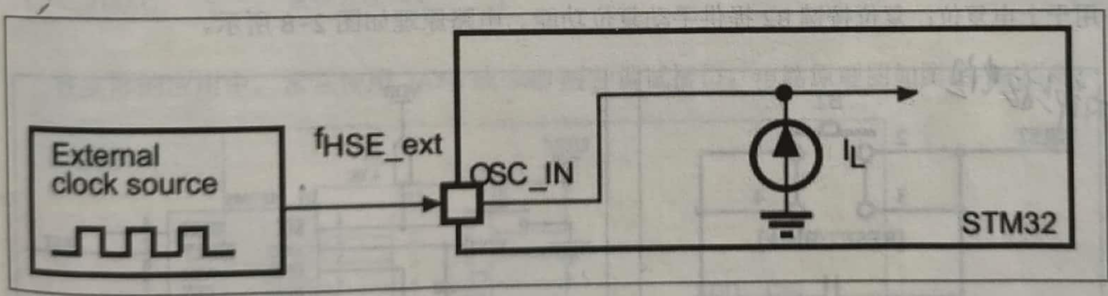


图 2-6　旁路时钟模式

在 Nucleo 开发板上，外部高速时钟 HSE 采用 8MHz 的石英晶振 X3，2 个 20P 电容 C33 和 C34，2 个 0 欧电阻 R35 和 R37 组成振荡电路。外部低速时钟 LSE 采用 32.768KHz 的石英晶振 X2，2 个 4.3P 电容 C31 和 C32，2 个 0 欧电阻 R34 和 R36 组成振荡电路。时钟电路原理图如图 2-7 所示。

Nucleo 开发板没有焊接 X3、C33、C34、R35 和 R37 等元件，而是采用旁路时钟模式，利用板载的 ST-Link 仿真器输出一个 8MHz 的时钟信号 MCO 送入 OSC_IN 引脚，作为 HSE 时钟。因此用户进行时钟配置时，可以选择使用 HSI 或者 HSE 来提供系统时钟。

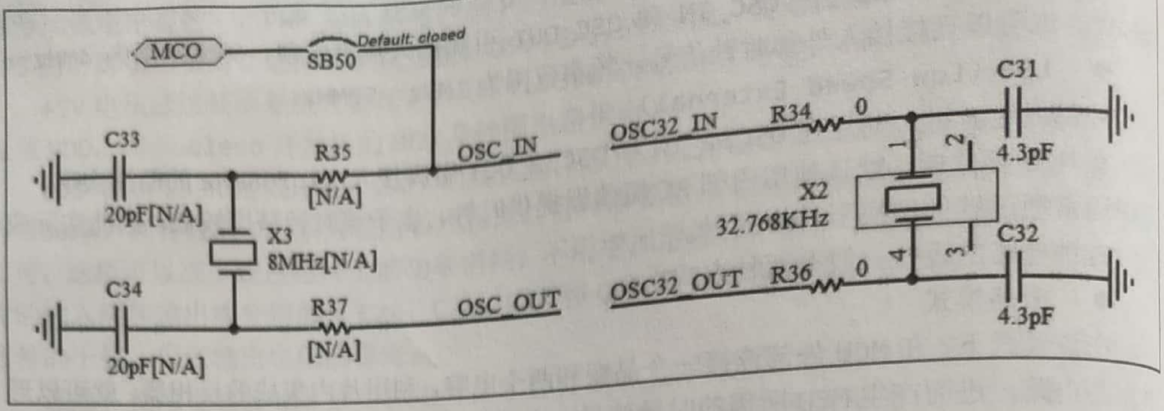


图 2-7　时钟电路

【小贴士】

如果用户需要使用振荡模式，利用外部晶体和电容产生 HSE 时钟，需要对 Nucleo 开发板进行如下修改：

1、断开焊桥 SB16、SB50、SB54 和 SB55（SB55 默认为断开状态）；

2、焊接 8MHz 石英晶体 X3；

3、焊接 2 个 20pF 电容 C33 和 C34；

4、焊接 2 个 0 欧电阻 R35 和 R37。

3. 复位电路

复位的目的是让系统每次启动时，都从一个确定的状态开始执行，以确保系统每次执行的结果一致。STM32 微控制器的复位有效电平为低电平，当复位管脚上的低电平持续一定的时间（大于 300ns）就会发生芯片复位。

在 Nucleo 开发板上，MCU 的复位电路采用了传统的阻容复位方式，具有上电复位和手动复位两种功能：由 MCU 的复位引脚 NRST 内部的上拉电阻和电容 C14 构成阻容复位电路，用于上电复位；复位按键 B2 提供手动复位功能。电路原理如图 2-8 所示。

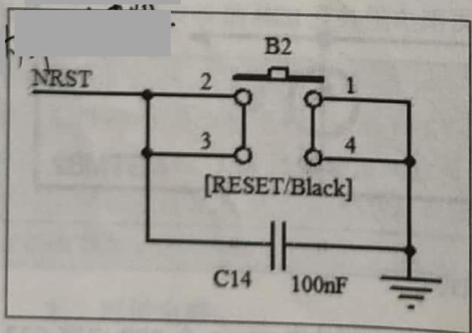


图 2-8　阻容复位电路

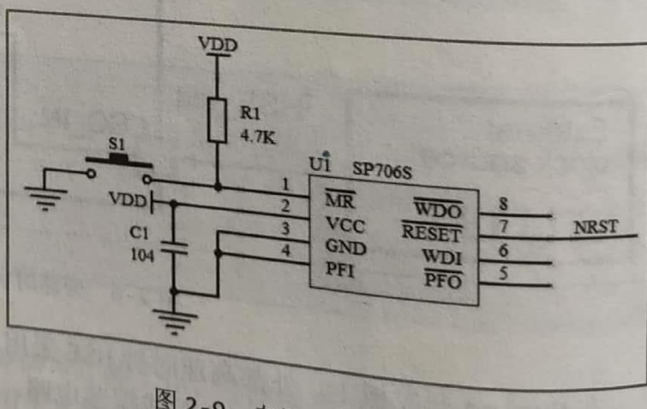


图 2-9　专用复位芯片电路

阻容复位电路的工作原理：当系统上电时，电容 C14 两端电压为 0V，MCU 处于复位状态，主电源电压 VDD 通过 NRST 引脚内部的上拉电阻给 C14 充电，当 C14 的电压上升到 MCU 的高电平门槛电压时，MCU 退出复位状态转入运行状态。在设计电路时，需要选择适当的电容大小，以保证 NRST 引脚上的低电平持续时间满足 MCU 复位时间的要求。

在系统运行过程中，如果出现运行错误，需要重启整个系统时，可以将复位按键 B2（黑

色键帽）按下，则电容 C14 被短路接地， NRST 引脚直接接地，输出低电平复位 MCU。释放按键后，结束手动复位操作，系统从头开始执行代码。

上电复位和手动复位的区别与个人电脑的冷启动和热启动的区别很类似：上电复位是从系统不带电时开始执行；而手动复位则是在系统带电时执行。

一般而言，这种阻容复位电路可以正常完成系统的复位要求，但是在比较严格的工程应用中，建议采用专用的复位芯片来设计复位电路，以增加系统的可靠性。这类专用的复位芯片可以提供长达 200ms 的复位脉冲，支持手动复位和上电复位，还带有电压监控和外部看门狗功能。

采用专用复位芯片 SP706S 设计的复位电路如图 2-9 所示。

4. 调试电路

调试电路主要用于仿真器和目标 MCU 之间的硬件连接，仿真器通过调试电路进行程序的下载和仿真。由于 Nucleo 开发板上设计了 ST-Link 仿真器电路，采用了串行调试接口（SWD），将目标 MCU 的调试引脚和 ST-Link 仿真器的对应引脚相连，因此没有设计单独调试电路。

- PA13 引脚作为 SWD 的数据输入/输出信号 SWDIO
- PA14 引脚作为 SWD 的时钟信号 SWCLK
- PB3 引脚作为 SWD 的调试信息输出信号 SWO（可选）

STM32 系列微控制器用于调试的引脚如表 2-3 所示。

表 2-3 STM32 微控制器调试引脚

引脚名称	JATG 调试端口	SWD 调试端口	引脚分配
JTMS/SWDIO	测试模式选择	串行数据输入/输出	PA13
JTCK/SWCLK	测试时钟	串行时钟	PA14
JTDI	测试数据输入	–	PA15
JTDO/SWO	测试数据输出	调试信息输出	PB3
JTRST	复位目标 MCU	–	PB4

在实际的应用中，常常使用 JATG 或 SWD 两种调试接口，电路原理图如图 2-10 所示。

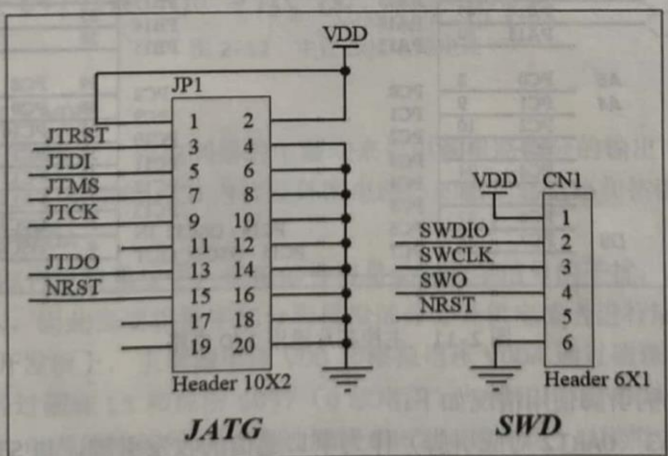


图 2-10 JATG/SWD 调试接口

JATG 调试接口采用标准的 20 针插座，需要占用 MCU 的 5 个引脚，并且在电路板上所占面积较大。而 SWD 调试接口只需要占用 MCU 的 2 个引脚 SWDIO 和 SWCLK（SWO 为可选引脚），并且可以利用简单的排针作为硬件连接，在电路板上所占面积较小。因此，在实际的工程应用中常使用 SWD 调试接口。

【小贴士】

1、SWO 引脚的作用主要是配合 Keil MDK 等集成开发环境输出调试信息，起到串口输出的作用。在工程应用中一般不连接 SWO 引脚，因此实际的调试引脚只有 SWDIO 和 SWCLK。

2、NRST 信号连接到目标 MCU 的复位引脚，以便仿真器能够控制 MCU 复位。

3、VDD 信号是目标 MCU 的供电电压，该信号不是由仿真器提供给目标 MCU，而是由目标 MCU 提供给仿真器，为仿真器内部的电平转换芯片供电。

5. 主控芯片

Nucleo 开发板的主控芯片采用 LQFP64 封装的 STM32F411RET6。由于 Nucleo 开发板在设计时考虑了主控芯片的兼容性，设计了大量的焊桥（0 欧电阻）作为跳线，通过焊接不同的焊桥来适配不同的主控芯片，如 STM32F0/F2/F3 系列的 MCU。这里我们将原始的电路图进行了简化，只考虑主控芯片为 STM32F411RET6 的情况。

主控芯片部分的电路分成两个模块，一个是通用 I/O 部分，一个是电源部分：

(1) 通用 I/O

通用 I/O 的电路连接及功能如图 2-11 所示。

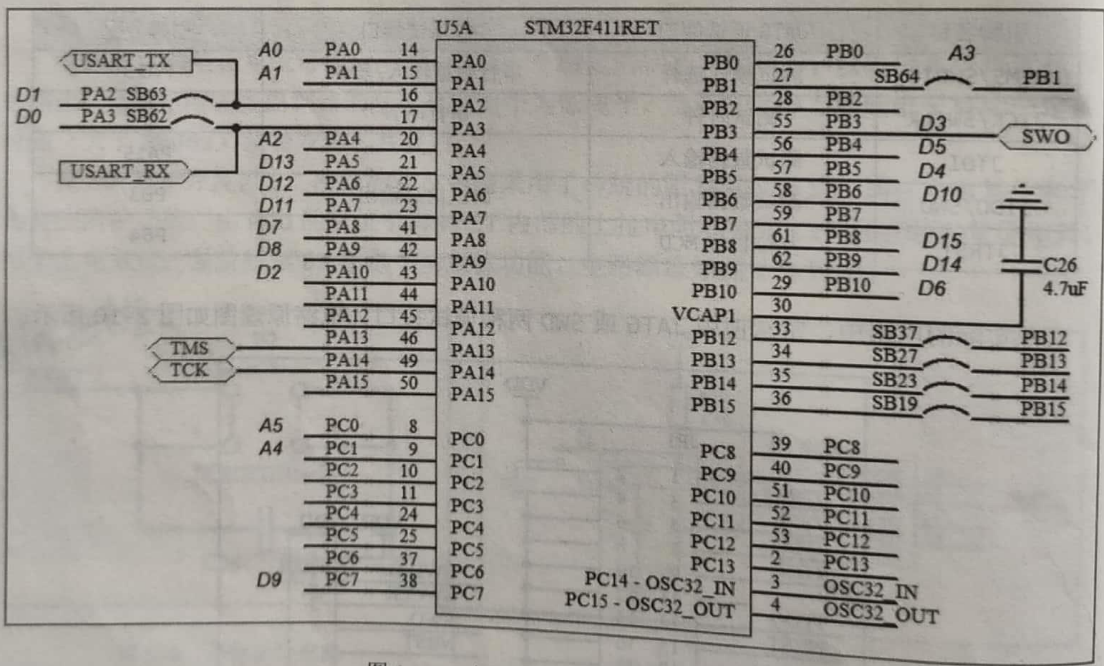


图 2-11 主控芯片通用 I/O 电路

通用 I/O 部分的引脚使用情况如下：

- PA2 和 PA3（UART2 对应引脚）作为串口通信的收发引脚，与 ST-Link 仿真器电路相连。
- PA5 控制指示灯 LD2。

- PC13 控制按键 B1。
- PA13、PA14、PB3 作为 SWD 调试接口。
- PC14 和 PC15 作为外部低速时钟 LSI。

同时，所用 I/O 引脚都扩展到两组外部接口：Arduino UNO 扩展接口和 ST Morpho 扩展接口，便于用户进行二次开发。

注意： STM32F411RET6 的 30 脚为 VCAP_1，该引脚用于内核电源的滤波，按照芯片参考手册的要求，需要外接一个 4.7uF 的滤波电容，以提供一个稳定的内核电压。

(2) 电源部分

STM32 微控制器的电源部分需要 3 种电压对其供电，分别是：

- 内核及外围电路所需的 3.3V 主电源电压 VDD
- 模拟部分（如片内 ADC、DAC 和 PLL）所需的 3.3V 模拟电压 VDDA
- 实时时钟（RTC）所需的 3.3V 电池电压 VBAT

电源部分的电路原理图如图 2-12 所示。

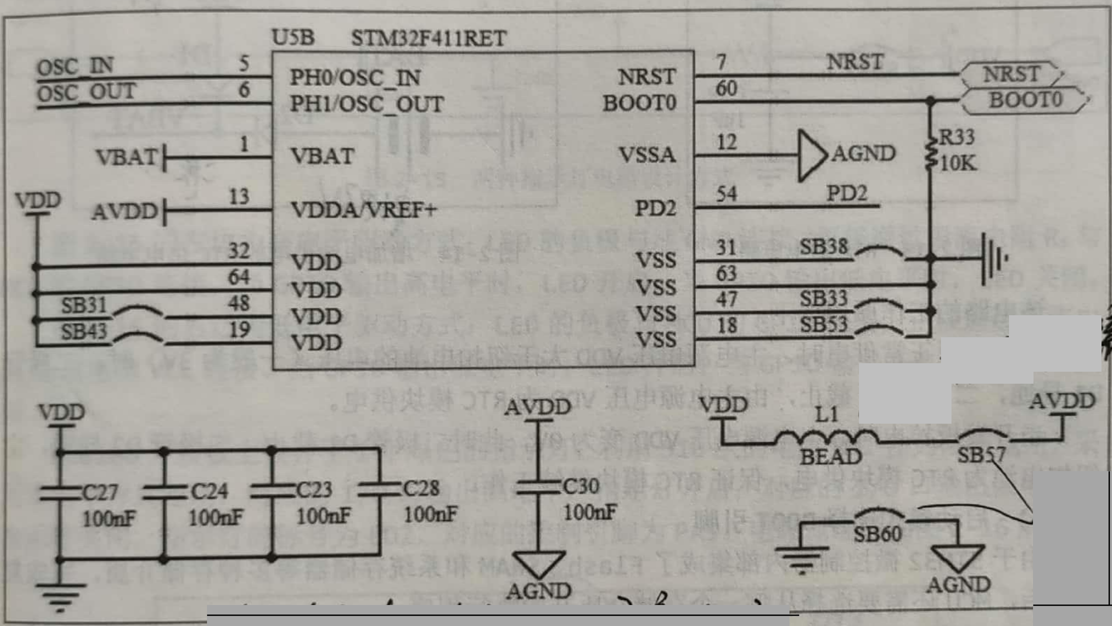


图 2-12 主控芯片电源电路

1) 主电源电压 VDD

在 Nucleco 开发板上，微控制器的电源均来自电源电路部分的输出电压 VDD，该电压直接作为主电源电压，提供给 MCU 内核及外围电路，如指示灯电路和按键电路。

2) 模拟电压 VDDA

在数字/模拟混合电路系统中，模拟信号容易受到数字信号的干扰，这些干扰信号一般通过电源电路传入，因此需要将数字部分和模拟部分分开供电或者进行隔离。

在 Nucleco 开发板上，主电源电压 VDD 和模拟电压 VDDA 通过磁珠和电阻进行隔离：主电源电压 VDD 经过磁珠 L1 和焊桥 SB57（0 欧电阻）与模拟电压 VDDA（电路网络标号为 AVDD）进行隔离；主电源的地信号 GND 通过焊桥 SB57（0 欧电阻）与模拟电压的地信号 VSSA（电路网络标号为 AGND）进行隔离，电路原理如图 2-12 所示。

在实际的工程应用中，为了提高电源的稳定性，在主控芯片每一对 VDD 和 VSS 之间需要设计 1 个 0.1uF 的去耦电容。在实际的硬件电路设计时，这些去耦电容的位置要尽量靠

近 VDD 和 VSS 引脚。

STM32F411RET6 有 4 对 VDD 和 VSS，1 对 VSSA 和 VDDA，因此设计了 5 个去耦电容 C27、C24、C23、C28 和 C30，如图 2-12 中所示。

3) RTC 电压 VBAT

STM32 微控制器片内都集成了 RTC 模块，可以提供时钟日历的功能，但是在系统掉电后，RTC 模块无法保存之前设置的时钟日历信息，每次上电后需要重新设置。为了保证 RTC 模块在系统掉电后能够继续工作，一般需要单独外接一个电池对 RTC 模块供电。

在 Nucleco 开发板上，简化了 RTC 的供电电路，由主电源电压 VDD 经过焊桥 SB45（0 欧电阻）产生 RTC 电压 VBAT，电容 C29 用于电源滤波，电路原理如图 2-13 所示。

图 2-13 所示的 RTC 供电电路在系统掉电后，无法保存设置的时钟日历信息。如果在实际应用中要求 RTC 模块在系统掉电后能够继续工作，则需要单独设计电池供电电路，如图 2-14 所示。

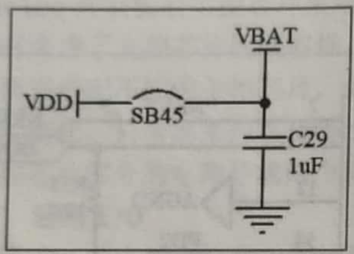


图 2-13 RTC 供电电路

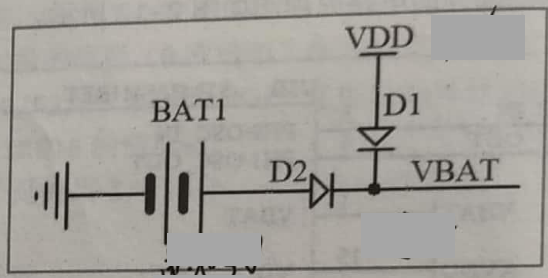


图 2-14 增加电池供电的 RTC 供电电路

该电路的工作原理如下：

当开发板正常供电时，主电源电压 VDD 大于纽扣电池的电压（一般为 3V）时，二极管 D1 导通，二极管 D2 截止，由主电源电压 VDD 为 RTC 模块供电。

当开发板掉电时，主电源电压 VDD 变为 0V，此时二极管 D1 截止，二极管 D2 导通，由纽扣电池为 RTC 模块供电，保证 RTC 模块继续工作。

4) 启动模式选择 BOOT 引脚

由于 STM32 微控制器内部集成了 Flash、SRAM 和系统存储器等多种存储介质，完成复位之后，MCU 还需要选择从哪一个存储介质开始执行程序。

STM32 系列微控制器可以通过 BOOT0 和 BOOT1 引脚来设置 MCU 复位之后的启动模式，具体的启动模式设置如表 2-4 所示。

表 2-4 启动模式选择

BOOT0	BOOT1	启动模式	说明
0	X	用户闪存存储器	从 MCU 片内 Flash 启动，常用启动方式
1	0	系统存储器	从系统存储器启动，主要用于串口下载（ISP 模式）
1	1	SRAM 启动	从 MCU 片内 SRAM 启动，主要用于代码调试

在 Nucleo 开发板上，将 BOOT0 引脚接入下拉电阻，并引出至外扩接口 CN7 的 7 脚，BOOT1 引脚默认作为通用 I/O 口 PB2 使用，并引出至外扩接口 CN10 的 22 脚，因此默认的启动模式为从用户闪存储存器启动。用户也可以修改外扩接口 CN7 和 CN10 对应引脚的电平来切换启动模式。

2.4.2 外围电路设计

在嵌入式系统设计中，微控制器只起到主控单元的作用，对于外界信息的采集、传输和处理等操作，还需要其他的外围电路。Nucleo 开发板上提供外围电路较少，只有一个指示灯和一个按键，但是设计了 2 类扩展接口：Arduino UNO 接口和 ST Morpho 接口。其中 Arduino Uno 接口用于接入 Arduino 生态圈的各类扩展板，而 ST Morpho 接口则可以访问主控芯片的全部 I/O 引脚，方便用户进行二次开发。

1. 指示灯电路

在进行微控制器外围电路设计时，一般都会设计指示灯，用于显示系统的状态。用户还可以通过编写指示灯闪烁的程序来检测 MCU 的最小系统是否正常工作。

指示灯一般使用发光二极管（Light-emitting diodes，LED），由于 LED 的单向导电性，指示灯电路一般有两种设计方式：高电平驱动和低电平驱动。电路原理如图 2-15 所示。

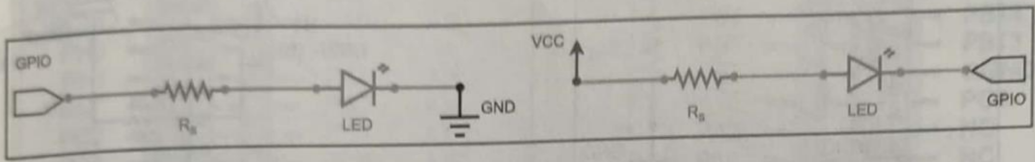


图 2-15　两种指示灯电路设计方式

图 2-15 的左边为高电平驱动方式：LED 的负极与地 GND 连接，正极通过限流电阻 R_s 与 MCU 的 GPIO 连接。当 GPIO 输出高电平时，LED 开启；当 GPIO 输出低电平时，LED 关闭。

图 2-15 的右边为低电平驱动方式：LED 的负极与 MCU 的 GPIO 连接，正极通过限流电阻 R_s 与电源 VCC 连接。当 GPIO 输出低电平时，LED 开启；当 GPIO 输出高电平时，LED 关闭。

Nucleo 开发板上设计了 1 个绿色的指示灯，利用 510 欧的电阻 R31 作为限流电阻，采用高电平方式驱动：对应的 I/O 口输出高电平，指示灯开启；对应的 I/O 口输出低电平，指示灯关闭。指示灯的标号为 LD2，对应的控制引脚为 PA5。电路原理图如图 2-16 所示。

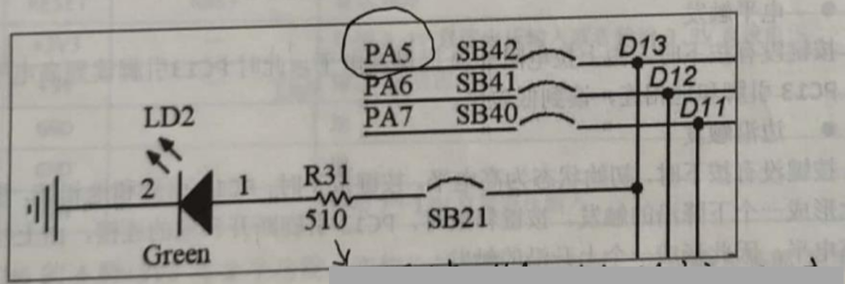


图 2-16　Nucleo 开发板指示灯电路

2. 按键

按键作为重要的人机接口设备，主要用于信息的输入。常见的按键形式一般有两种：

- 独立按键

采用微控制器的通用 I/O 引脚直接控制按键，一个 I/O 引脚控制一个按键，常用于按键数目较少的情况。

- 行列按键

采用矩阵方式进行按键的布局，利用扫描的方法完成按键译码，占用的 I/O 引脚较少，一般用于按键数目比较多的情况。比如 4 行 5 列的矩阵键盘，可以提供 20 个按键，只需要占用 9 个 I/O 引脚。

对于独立按键，在电路设计上一般有两种方式：上拉式按键和下拉式按键。电路原理如图 2-17 所示。

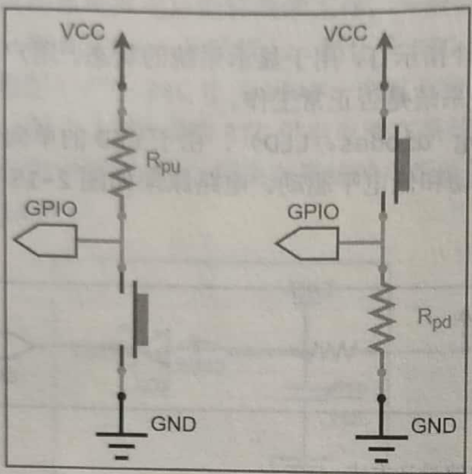


图 2-17　两种独立按键电路设计方式

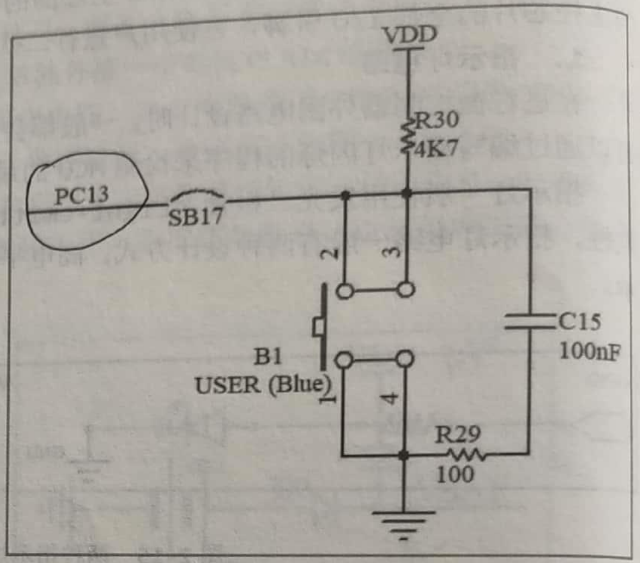


图 2-18　Nucleo 开发板按键电路

图 2-17 的左边为上拉式按键：按键的一端与地 GND 连接，另一端通过上拉电阻 Rpu 与电源 VDD 连接。当按键按下时，GPIO 输入低电平；按键释放时，GPIO 输入高电平。

图 2-17 的右边为下拉式按键：按键的一端与电源 VDD 连接，另一端通过下拉电阻 Rpd 与地 GND 连接。当按键按下时，GPIO 输入高电平；按键释放时，GPIO 输入低电平。

Nucleo 开发板上设计了 1 个独立按键（蓝色键帽），标号为 B1，对应的控制引脚为 PC13，采用了上拉式按键设计。电路原理图如图 2-18 所示。

按键 B1 的判断可以采用电平触发或者边沿触发方式：

- 电平触发

按键没有按下时，由上拉电阻 R30 提供高电平，此时 PC13引脚读到高电平；按键按下时，PC13 引脚和地相连，读到低电平。

- 边沿触发

按键没有按下时，初始状态为高电平。按键按下时，PC13 引脚和地相连，变为低电平，因此形成一个下降沿的触发。按键释放时，PC13 引脚断开和地的连接，由上拉电阻 R30 提供高电平，因此形成一个上升沿的触发。

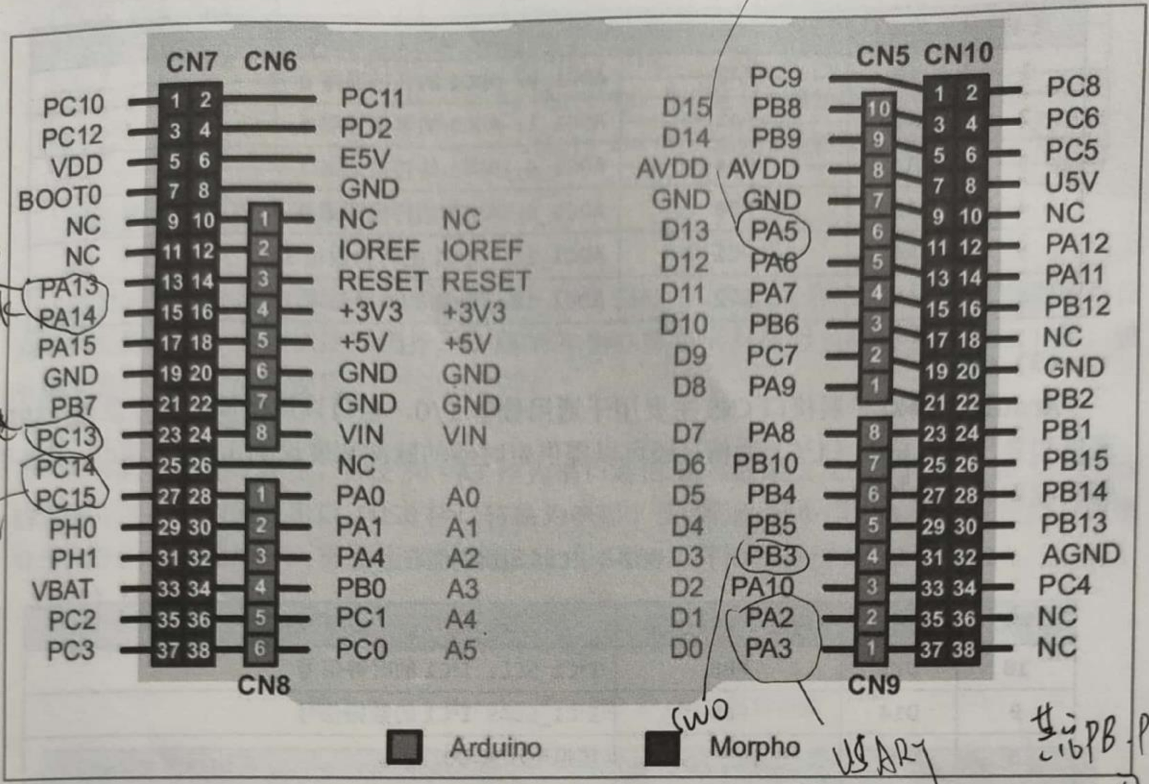
3. 扩展接口

在 Nucleo 开发板上设计了 2 类扩展接口：Arduino UNO 接口和 ST Morpho 接口。扩展接口的示意图如图 2-19 所示。

在图 2-19 中，CN5、CN6、CN8 和 CN9 是 Arduino UNO 扩展接口，以单排插座（间距 2.54mm）形式安装在 Nucleo 开发板上。CN7 和 CN10 是 ST Morpho 扩展接口，以 2×19 的双排插针（间距 2.54mm）形式安装在 Nucleo 开发板上。

(1) CN6

Arduino UNO 扩展接口 CN6 主要用于电源接口，各引脚功能如表 2-5 所示。



共同PB.PA PC口用户可用.

图 2-19　扩展接口示意图

表 2-5　CN6 引脚功能

序号	名称	对应 STM32 引脚	功能说明
1	NC	—	—
2	IOREF	—	输出 3.3V 直流电压
3	RESET	NRST	复位 MCU
4	+3V3	—	外接 3.3V 直流电压输入或者输出 3.3V 直流电压
5	+5V	—	输出 5V 直流电压
6	GND	—	地
7	GND	—	地
8	VIN	—	外接 7～12V 直流电压输入

注意：CN6 的 4 脚+3V3 有 2 个功能：在作为外部接口供电时，可以输入外部 3.3V 直流电压；在作为电压输出时，可以输出 3.3V 直流电压，用于对扩展板电路的供电。

(2) CN8

Arduino UNO 扩展接口 CN8 主要用于 A/D 转换。由于 ADC 模块的参考电压的 VREF+和 VREF-没有单独引出，而是在芯片内部和模拟电源 VDDA 以及模拟地 VSSA 相连，因此输入的模拟电压需要限制在 0～3.3V 之间，各引脚功能如表 2-6 所示。

表 2-6　CN8 引脚功能

序号	名称	对应 STM32 引脚	功能说明
1	A0	PA0	ADC1_0：ADC1 的转换通道 0
2	A1	PA1	ADC1_1：ADC1 的转换通道 1
3	A2	PA4	ADC1_4：ADC1 的转换通道 4
4	A3	PB0	ADC1_8：ADC1 的转换通道 8
5	A4	PC1	ADC1_11：ADC1 的转换通道 11
6	A5	PC2	ADC1_10：ADC1 的转换通道 10

(3) CN5

Arduino UNO 扩展接口 CN5 主要用于通用数字 I/O，也可以用于同步串行总线（SPI）通信和异步串行总线（I^2C）通信，还可以提供定时器的脉冲宽度调制（PWM）输出，其各引脚功能如表 2-7 所示。

表 2-7　CN5 引脚功能

序号	名称	对应 STM32 引脚	功能说明
10	D15	PB8	I^2C1_SCL：I^2C1 的时钟信号
9	D14	PB9	I^2C1_SDA：I^2C1 的数据信号
8	AREF	—	模拟电压 AVDD
7	GND	—	地
6	D13	PA5(1)	SPI1_SCK：SPI1 的时钟信号
5	D12	PA6	SPI1_MISO：SPI1 的主机输入/从机输出信号
4	D11	PA7	TIM1_CH1N：定时器 1 的通道 1 的互补输出（PWM） SPI1_MOSI：SPI1 的主机输出/从机输入信号
3	D10	PB6	TIM4_CH1：定时器 4 的通道 1（PWM） SPI1_CS：SPI1 的从机选择信号
2	D9	PC7	TIM3_CH2：定时器 2 的通道 2（PWM）
1	D8	PA9	通用数字 I/O

CN9 接口使用时要特别注意以下引脚：

1) PA5

引脚默认用于控制指示灯 LD2，如果将 PA5 作为外扩引脚使用时，需要将焊桥 SB21 断开。

(4) CN9

Arduino UNO 扩展接口 CN9 主要用于通用数字 I/O，也可以用于串口通信和定时器的 PWM 输出，各引脚功能如表 2-8 所示。

表 2-8　CN9 引脚功能

序号	名称	对应 STM32 引脚	功能说明
8	D7	PA8	通用数字 I/O
7	D6	PB10	TIM2_CH3：定时器 2 的通道 3（PWM）
6	D5	PB4	TIM3_CH1：定时器 3 的通道 1（PWM）

5	D4	PB5	通用数字 I/O
4	D4	PB3(1)	TIM2_CH2：定时器 2 的通道 2（PWM）
3	D2	PA10	通用数字 I/O
2	D1	PA2(2)	USART2_TX：串口通信 2 的数据发送引脚
1	D0	PA3(2)	USART2_RX：串口通信 2 的数据接收引脚

CN9 接口使用时要特别注意以下引脚：

1）PB3

当调试接口选择为 3 线制的 SWD 接口（使用 SWDIO、SWCLK 和 SWO）时，该引脚不能作为外扩引脚使用；当调试接口选择为 2 线制的 SWD 接口（只使用 SWDIO 和 SWCLK）时，该引脚才能作为外扩引脚使用。

2）PA2 和 PA3

在 Nucleo 开发板上，PA2 和 PA3 作为串口通信 2 的数据收发引脚，默认和 ST-Link 仿真器连接，作为虚拟串口（USB 接口转换为串口）使用，一般不作为外扩引脚使用。如果要作为外扩引脚使用时，需要连接焊桥 SB62 和 SB63，并断开焊桥 SB13 和 SB14。

(5) CN7 和 CN10

Morpho 扩展接口 CN7 和 CN10 的引脚功能如表 2-9 所示。

表 2-9 ST Morpho 接口 CN7 和 CN10 引脚功能

CN7 奇数引脚		CN7 偶数引脚		CN10 奇数引脚		CN10 偶数引脚	
序号	名称	序号	名称	序号	名称	序号	名称
1	PC10	2	PC11	1	PC9	2	PC8
3	PC12	4	PD2	3	PB8	4	PC6
5	VDD	6	E5V(7)	5	PB9	6	PC5
7	BOOT0(1)	8	GND	7	AVDD	8	U5V(16)
9	–	10	–	9	GND	10	–
11	–	12	IORET(8)	11	PA5(13)	12	PA12
13	PA13(2)	14	RESET(9)	13	PA6	14	PA11
15	PA14(2)	16	+3.3V(10)	15	PA7	16	PB12
17	PA15	18	+5V(11)	17	PB6	18	–
19	GND	20	GND	19	PC7	20	GND
21	PB7	22	GND	21	PA9	22	PB2
23	PC13(3)	24	VIN(12)	23	PA8	24	PB1
25	PC14(4)	26	–	25	PB10	26	PB15
27	PC15(4)	28	PA0	27	PB4	28	PB14
29	PH0(5)	30	PA1	29	PB5	30	PB13
31	PH1(5)	32	PA4	31	PB3(14)	32	AGND
33	VBAT(6)	34	PB0	33	PA10	34	PC4
35	PC2	36	PC1	35	PA2(15)	36	–
37	PC3	38	PC0	37	PA3(15)	38	–

CN7 和 CN10 引出了主控芯片 STM32F411RET 的全部 I/O 引脚，便于用户进行二次开

发。一般情况下，所有的 I/O 引脚均可使用，但需要注意以下引脚：

1） BOOT0

BOOT0 默认的状态为 0，当 CN7 的 5 脚和 7 脚用跳线短接后，可以设置为 1。

2） PA13 和 PA14

这两个引脚一般作为 SWD 串行调试接口使用，不推荐作为数字 I/O 使用。

3） PC13

引脚默认用于控制按键 B1，如果将 PC13 作为外扩引脚使用时，需要将焊桥 SB17 断开。

4） PC14 和 PC15

这两个引脚一般作为外部低速时钟 LSI，不推荐作为数字 I/O 使用。只有不使用外部低速时钟 LSI 时，才可以作为数字 I/O 使用。

5） PH0 和 PH1

这两个引脚一般作为外部高速时钟 HSE，不推荐作为数字 I/O 使用。

6） VBAT

RTC 模块供电电压，如果系统中需要使用 RTC 模块，可以接入一个电池，以保证 RTC 模块在系统掉电后仍然能够工作。

7） E5V

外接 5V 直流电压输入或者输出 5V 直流电压。

8） IORET

输出 3.3V 直流电压。

9） RESET

复位 MCU。

10）+3.3V

外接 3.3V 直流电压输入或者输出 3.3V 直流电压。

11）+5V

输出 5V 直流电压。

12）VIN

外接 7～12V 直流电压输入。

13）PA5

引脚默认用于控制指示灯 LD2，如果将 PA5 作为外扩引脚使用时，需要将焊桥 SB21 断开。

14）PB3

当调试接口选择为 3 线制的 SWD 接口（使用 SWDIO、SWCLK 和 SWO）时，该引脚不能作为外扩引脚使用；当调试接口选择为 2 线制的 SWD 接口（只使用 SWDIO 和 SWCLK）时，该引脚才能作为外扩引脚使用。

15）PA2 和 PA3

在 Nucleo 开发板上，PA2 和 PA3 作为串口通信 2 的数据收发引脚，默认和 ST-Link 仿真器连接，作为虚拟串口（USB 接口转换为串口）使用，一般不作为外扩引脚使用。如果要作为外扩引脚使用时，需要连接焊桥 SB62 和 SB63，并断开焊桥 SB13 和 SB14。

16）U5V

输出 5V 直流电压。U5V 直接来自 USB 接口，该电压先于+5V 电压出现。

在实际应用中，Arduino UNO 扩展接口和 ST Morpho 扩展接口的扩展板可以通过层叠的形式安装在 Nucleo 开发板的上方或下方，如图 2-20 所示。

低

模

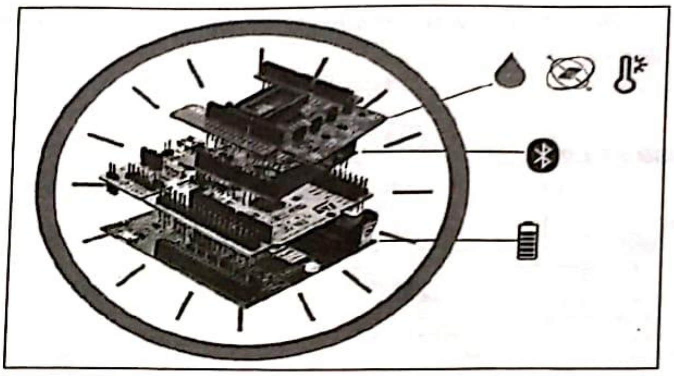


图 2-20　Nucleo 开发板和扩展板的连接方式

除了使用专用的扩展板以外，也可以利用杜邦线将独立的电路模块与 Nucleo 开发板的扩展接口相连，完成用户需要的设计。图 2-21 就是利用超声波传感器和字符型液晶模块 LCD1602 搭建的测距系统。

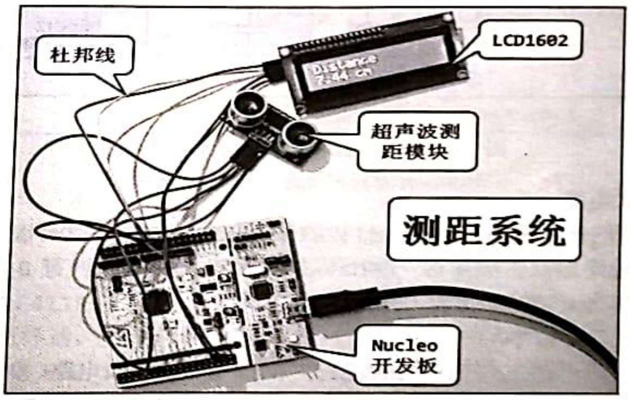


图 2-21　利用 Nucleo 开发板搭建的测距系统

2.4.3　ST-Link 仿真电路设计

Nucleo 板上设计了 ST-Link 仿真器，具有虚拟串口、大容量存储和仿真调试等功能，其中仿真调试功能仅支持 SWD 串行调试接口。

ST-Link 仿真器电路可以分为电源电路、仿真器 MCU 电路和 SWD 调试接口等 3 个模块：

1.　电源电路

ST-Link 仿真器有两种供电方式：

- USB 接口供电

通过 USB 接口 CN1 供电，USB 接口类型为 Mini-USB，输出+5V 直流电压（网络标号 U5V）。+5V 电压通过低压差线性稳压芯片 U1（LD39050PU33R）转换成 3.3V 电压（网络标号+3V3_ST_LINK）向 ST-Link 仿真器的 MCU 及外围电路供电。

- 外部供电

通过外部电源输入接口 E5V 或 VIN（VIN 可以产生 E5V 所需的电压）供电。

电路原理图如图 2-22 所示。

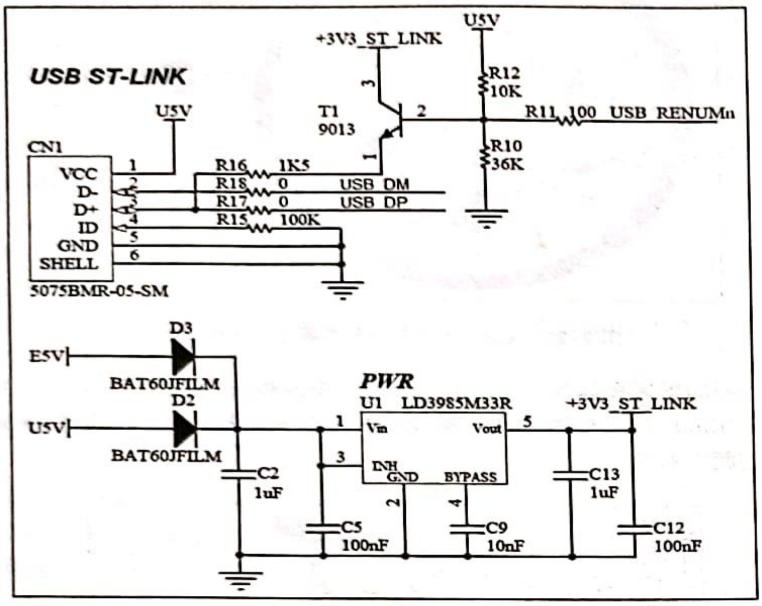


图 2-22　ST-Link 仿真器电源电路

2. 仿真器 MCU 电路

ST-Link 仿真器的 MCU 采用 LQFP48 封装的 STM32F103CBT6，具有 128KB 的内部 Flash 和 20KB 的 SRAM，最高运行主频可达 72MHz。片内外设比较丰富，有 2 个 12 位的 ADC、2 个 I²C、2 个 SPI、3 个 USART、1 个 USB 接口和 1 个 CAN 接口。

电路原理图如图 2-23 所示。

仿真器 MCU 部分的电路主要包括时钟电路、复位电路、串口电路、USB 接口以及调试电路，并结合内置的固件来实现虚拟串口、大容量存储和仿真调试等功能。

- 时钟电路

外接 8MHz 的晶振 X1，采用外部高速时钟 HSE 作为主时钟。

- 复位电路

利用电阻 R14 和电容 C6 构成的阻容复位完成上电复位功能。

- 串口电路

仿真器 MCU 的 PA2 和 PA3 作为串口通信使用，通过焊桥 SB14 和 SB13 与板载主控芯片 STM32F411RET 的 PA2 和 PA3（UART2）进行通讯，作为虚拟串口的物理实现。

- USB 接口

仿真器 MCU 的 PA11 和 PA12 作为 USB 通信使用，与 USB 接口 CN1 的 D+和 D−连接，作为大容量存储功能的物理实现。

- 调试接口

通过仿真器 MCU 内置的固件，模拟出 SWD 接口的时序，实现 SWD 调试功能。

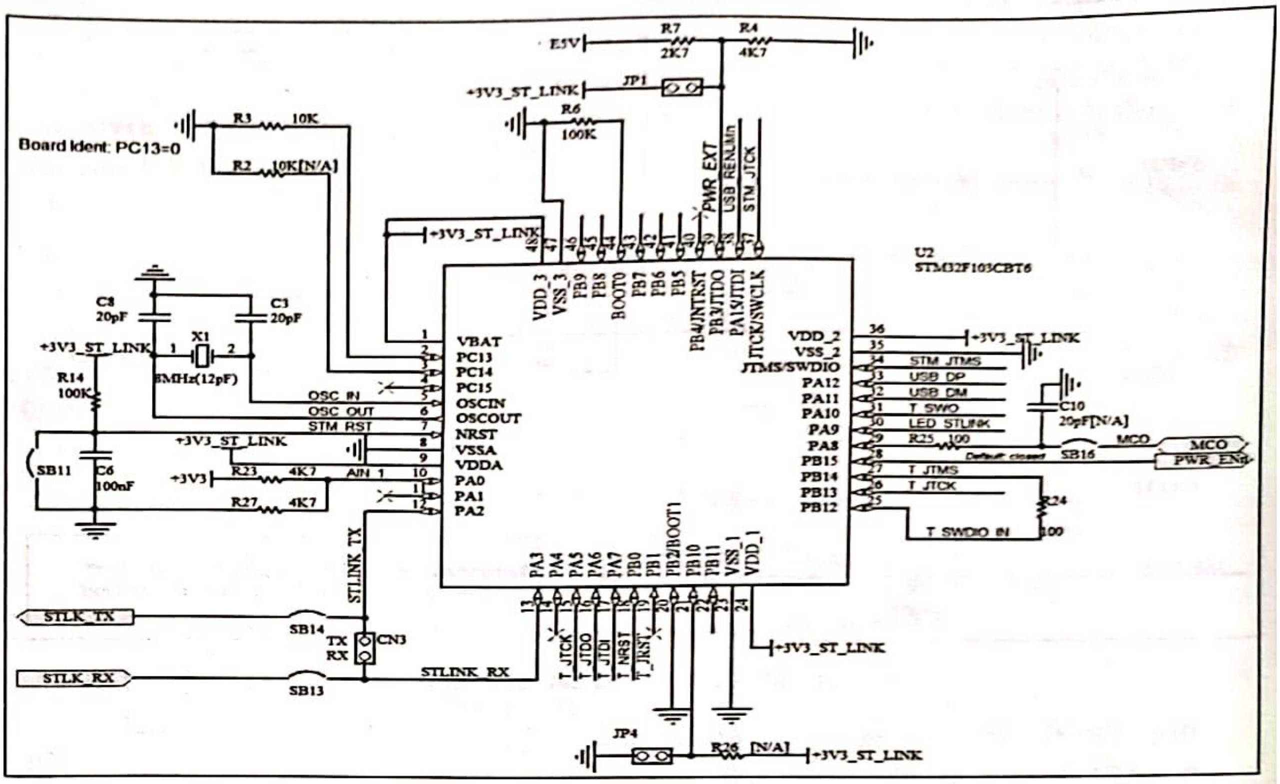


图 2-23　ST-Link 仿真器 MCU 电路

除此以外，仿真器 MCU 的 PC13 和 PC14 作为 Nucleo 板的识别码使用：当 PC13=0 时，认为该板属于 Nucleo 系列；仿真器 MCU 的 PA8（网络标号 MCO）输出 8MHz 的方波，作为板载主控芯片 STM32F411RET 的时钟；仿真器 MCU 的 PB15（网络标号 PWR_ENn）输出低电平，控制 MOS 管 T1 导通，确保板载主控芯片 STM32F411RET 及外围电路的供电；仿真器 MCU 的 PA9（网络标号 LED_STLINK）作为通信状态指示灯的控制引脚，通过改变指示灯的颜色来反映仿真器的通信状态。

3.　SWD 调试接口

ST-Link 仿真器调试接口的电路原理图如图 2-24 所示。

当 CN2 的跳线短接时，ST-Link 仿真器用于板载 MCU 的编程调试，使用了 4 个信号：SWCLK、SWDIO、T_NRST 和 T_SWO。具体的电路连接如下：

- T_JTCK 通过焊桥 SB3、CN2 的 1 脚和 2 脚、焊桥 SB5 与板载 MCU 的 SWCLK 相连。
- T_JTMS 通过焊桥 SB7、CN2 的 3 脚和 4 脚、焊桥 SB9 与板载 MCU 的 SWDIO 相连。
- T_NRST 通过焊桥 SB12 和板载 MCU 的复位引脚 NRST 相连，可控制 MCU 复位。
- T_SWO 通过焊桥 SB15 和板载 MCU 的 SWO 相连。（可选信号）

在 SWD 调试接口部分还设计了通信指示灯 LD1,用于显示 ST-Link 仿真器的通讯状态，可以显示红色、绿色和橙色等三种颜色。电路原理图如图 2-24 所示。

LD1 由仿真器 MCU 的 PA9 控制，显示原理如下：当 PA9 输出高电平时，LD1 发出红光；当 PA9 输出低电平时，LD1 发出绿光；当 PA9 输出一个 1KHz 左右的方波时，高、低电平的变化使得 LD1 指示灯交替发出红光和绿光，但由于颜色变化频率较快，人眼所见光为橙色。

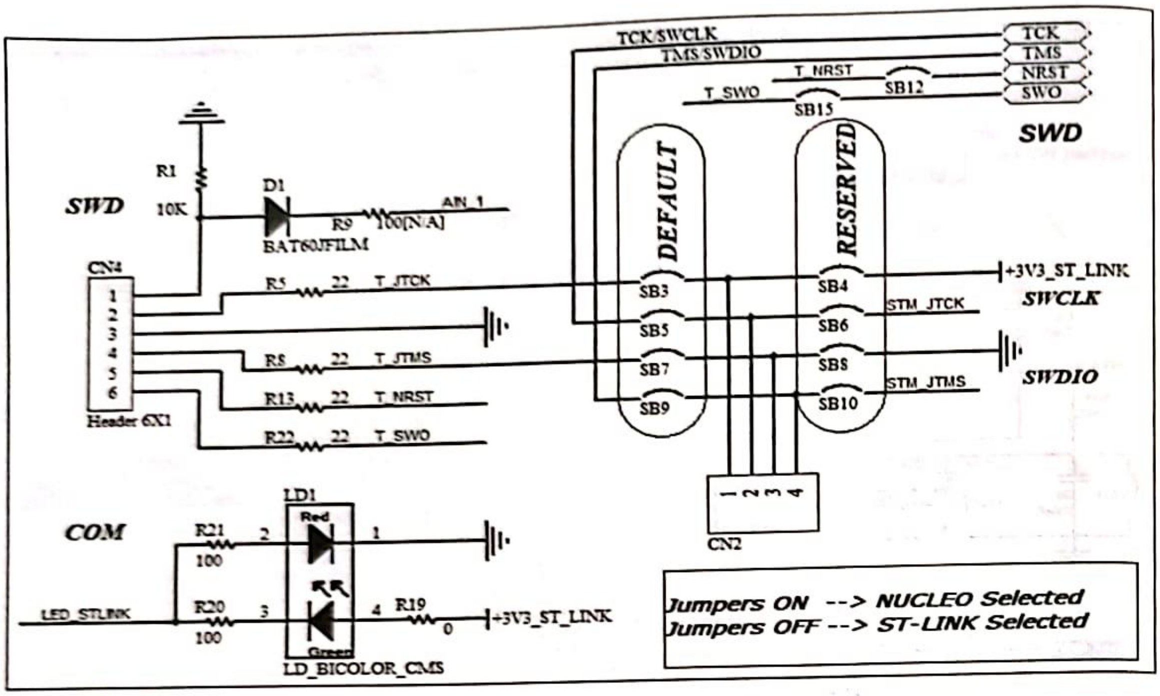


图 2-24 ST-Link 仿真器 SWD 调试接口电路

指示灯的不同状态，代表不同的通信状态：

- LD1 闪烁红色：USB 与 PC 正在枚举
- LD1 为红色：建立 PC 与 ST-Link 仿真器之间的通信（枚举过程结束）
- LD1 交替闪烁绿色和红色： PC 和目标芯片正在通信
- LD1 为绿色：通信完成且通信成功
- LD1 为橙色：通信故障

由于 SWD 调试接口电路，引入了比较多的焊桥，并且利用网络标号进行连接，查看电路图不太方便。这里我们将 SWD 调试接口的电路做一个简化，便于读者了解具体的电路连接，如图 2-25 所示。

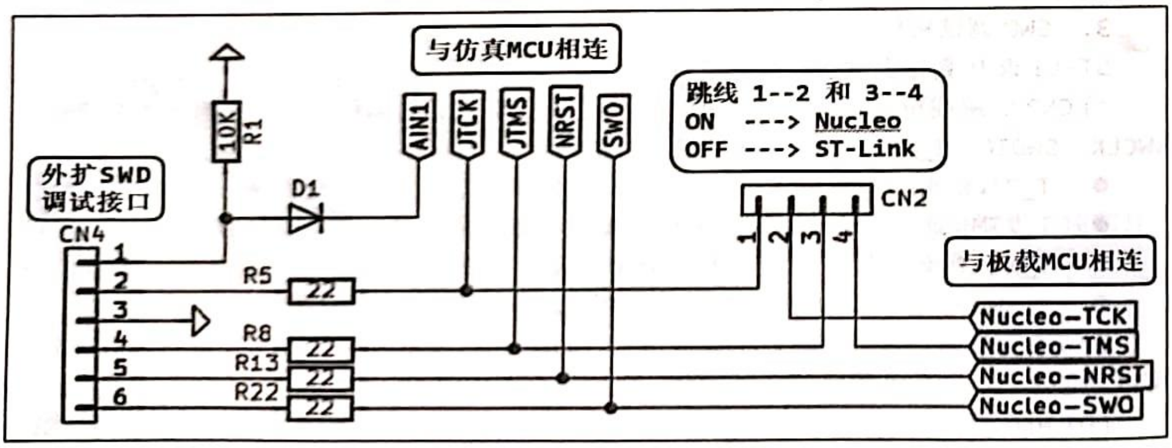


图 2-25 简化后的 SWD 调试接口电路

从图 2-25 就可以比较清晰的看到：由仿真 MCU 引出 4 个调试信号：JTCK、JTMS、NRST 和 SWO。其中 NRST 和 SWO 信号同时接入到外扩 SWD 调试接口和板载 MCU。而 JTCK(SWCLK) 通过插座 CN2 的 1 脚和 2 脚，JTMS（SWCDIO）通过插座 CN2 的 3 脚和 4 脚来控制连接：

- 当安装跳线帽时，JTCK（SWCLK）和 JTMS（SWCDIO）信号与板载 MCU 相连，可以对 Nucleo 开发板进行调试。
- 当去掉跳线帽时，JTCK（SWCLK）和 JTMS（SWCDIO）信号与外扩 SWD 调试接口相连，可以作为独立的 ST-Link 仿真器对外部目标板进行调试。

2.4.4 利用板载 ST-Link 仿真器调试外部 MCU

ST-Link 仿真器的调试接口除了可以对板载的 MCU 进行仿真外，还可以调试外部 MCU。当 CN2 的跳线断开时，ST-Link 仿真器就作为独立的仿真器，通过 CN4 接口对外部 MCU 进行仿真调试。

对外的 SWD 调试接口 CN4 的各引脚定义及连接方式如表 2-10 所示。

表 2-10 SWD 调试接口 CN4 引脚定义

引脚	名称	功能说明
1	VDD_Target	电源，连接外部 MCU 的供电电压
2	SWCLK	串行时钟，连接外部 MCU 的 SWCLK 引脚
3	GND	地，连接外部 MCU 的电源地
4	SWDIO	串行数据输入/输出，连接外部 MCU 的 SWDIO 引脚
5	NRST	复位信号，连接外部 MCU 的复位引脚
6	SWO	调试信号输出，可选信号，连接外部 MCU 的 SWO 引脚

CN4 在 Nucleo 开发板上的位置如图 2-26 所示。

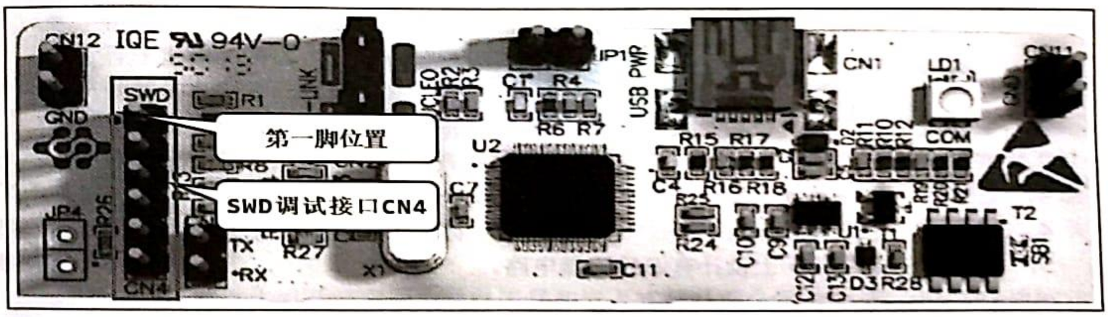


图 2-26 SWD 调试接口 CN4 在 Nucleo 开发板上的位置示意图

一般情况下，利用 ST-Link 仿真器调试外部 MCU 时，只需要连接 CN4 的 1～4 脚即可：VDD_Target、SWCLK、GND 和 SWDIO。如果需要利用仿真器控制外部 MCU 的复位，则需要连接 CN4 的 5 脚 NRST，同时将焊桥 SB12 断开。硬件连接示意图如图 2-27 所示。

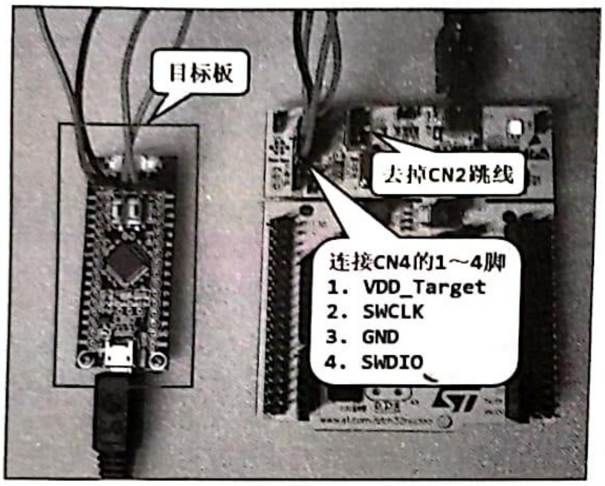


图 2-27　ST-Link 仿真器调试外部 MCU 的硬件连接示意图

关于 Nucleo 开发板更深入的介绍，可以在 ST 公司的官方网站 www.st.com 上下载 Nucleo 开发板的用户手册，文档标号为 UM1724。

2.5　本章小结

本章重点分析了 STM32F4 微控制器的硬件电路设计，其中特别重要的是最小系统的设计，包括了电源电路、时钟电路、复位电路和调试电路等四个功能模块。这些模块电路的设计具备通用性，所有的微控制器都采用类似的电路进行设计，读者需要好好掌握这些电路的设计原理，并灵活应用到自己的设计中。

习题 2

1. 尝试使用其他的复位芯片设计复位电路。
2. 在目前的电路基础上设计 8 个指示灯电路。
3. 在目前的电路基础上设计 4×4 的矩阵键盘电路。
4. 在目前的电路基础上设计 4 位 8 段数码管，采用动态扫描方式设计电路。
5. 在目前的电路基础上设计有源蜂鸣器电路。

第三章　STM32 微控制器开发环境的搭建

本章导读

本章重点介绍 STM32 微控制器开发环境的搭建。主要介绍了 STM32CubeMX 软件的安装、MDK-ARM 集成开发环境的安装以及 ST-Link 仿真器驱动程序的安装。

本章知识点

STM32Cube 生态系统

STM32CubeMX 软件及 MCU 软件包的安装

MDK-ARM 集成开发环境及器件支持包的安装

ST-Link 仿真器驱动程序的安装

3.1 STM32Cube 生态系统

在进行 STM32 微控制器的程序开发之前，我们需要搭建相应的开发环境。一般而言，用于 MCU 开发的完整工具链（ToolChain）包括以下组件：

- 编辑器

完成源代码的编辑

- 交叉编译器

将源文件编译为二进制格式的目标代码

- 支持库

提供基本输入/输出和数学运算等常用函数

- 链接器

将库文件和目标代码整合生成可执行文件

- 仿真器

用于程序下载和调试

- 硬件仿真器驱动程序

以便电脑能够正确识别各个厂商生产的硬件仿真器

为了提高用户的开发效率，减少开发时间和开发成本，ST 公司为 STM32 微控制器设计了一套软件生态系统：STM32Cube。该系统提供了众多的软件工具和软件包，可完成芯片选型、芯片配置、应用程序的开发及调试、程序烧录和程序性能监控等操作，涵盖了嵌入式系统开发的全过程。

STM32Cube 软件生态系统主要由软件工具和嵌入式软件两个部分组成，如图 3-1 所示。

1. 软件工具（Software Tools）

一套简单易用的软件开发工具，涵盖从概念到实现的项目开发所有阶段，包括以下软件工具：

(1) STM32CubeMX

一个图形化的软件配置工具，可以完成引脚分配、系统时钟及片内外设的设置，并自动生成初始化代码和应用程序的工程框架。

(2) STM32CubeIDE

一个基于 Eclipse + GNU C/C++工具链的集成开发环境，包括代码编辑、编译、下载和调试等多个功能，同时将 STM32CubeMX 以插件形式集成在软件中。

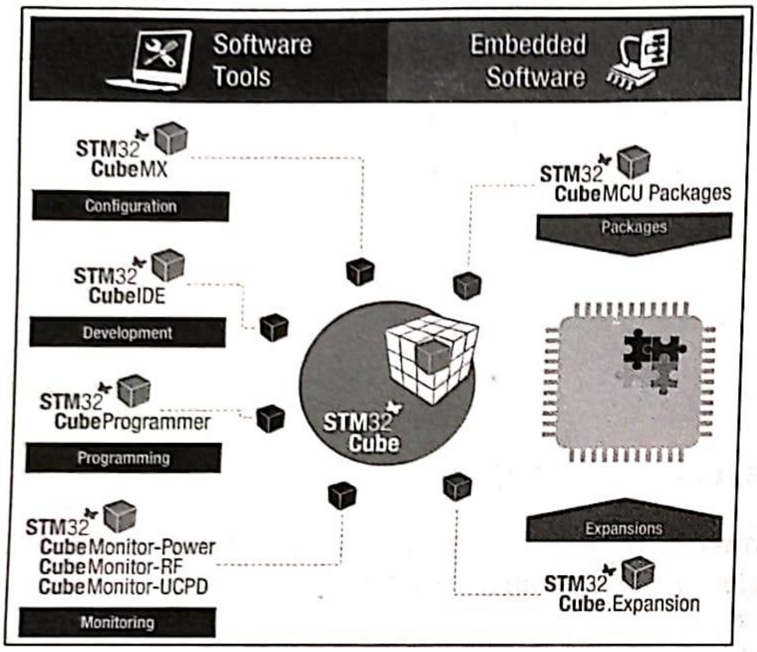


图 3-1 STM32Cube 软件生态系统

(3) STM32CubeProgrammer (STM32CubeProg)

图形化和命令行版本的编程工具，可通过多种接口（如 JTAG、SWD、UART、USB DFU、I²C，SPI 和 CAN 等）将程序代码下载到微控制器中。

(4) STM32CubeMonitor

一套监控工具，帮助开发人员实时了解应用程序的行为和性能。

2. 嵌入式软件（Embedded Software）

一套嵌入式软件，帮助加快项目开发，包括以下组件：

(1) STM32Cube MCU Packages（MCU 软件包）

MCU 软件包，按照产品系列来发布：如 STM32CubeF4 对应 STM32F4 系列 MCU，STM32CubeF1 对应 STM32F1 系列 MCU。它提供控制 STM32 片内外设所需的应用接口函数（API），是进行 STM32 开发的主要工具。

MCU 软件包主要由以下四个模块组成：

1) STM32Cube HAL APIs（HAL 库）

一套针对 MCU 硬件抽象层（Hardware Abstract Layer,HAL）的应用程序接口函数（API），如 STM32Cube_FW_F4，对应 STM32F4 系列 MCU，以确保在整个 STM32 产品组合中的可移植性，一般简称为 HAL 库。

2) STM32Cube LL APIs（LL 库）

一套针对 MCU 底层（Low Layer,LL）的应用程序接口函数（API），确保最佳的性能和最小的代码量，一般简称为 LL 库。

3) 中间件组件

包括嵌入式实时操作系统（FreeRTOS），USB 协议，TCP/IP 网络协议和图形用户界面（TouchGFX ）等。

4) 工具及例程

相关实用工具软件和与 MCU 外设应用相关的全部例程。

(2) STM32Cube Expansion Packages（STM32 扩展包）

STM32 扩展软件包是对 STM32Cube 微控制器产品的补充和扩展，提供额外的嵌入式软件功能，如蓝牙通信、电机控制算法等。STM32 扩展软件包来源于意法半导体公司及其合作伙伴，旨在围绕 STM32 打造广泛且可扩展的嵌入式软件产品。

在实际的工程应用中，通常采用的开发模式是利用 STM32CubeMX（以下简称 CubeMX）完成芯片的初始化设置，并生成工程框架，再借助第三方集成开发环境（如 MDK-ARM）完成应用程序的编写、下载和调试。

3.2 STM32CubeMX 的下载及安装

CubeMX 软件利用图形化界面完成芯片的配置，并配合 MDK-ARM 或 EWARM 等集成开发环境，生成应用程序的基本框架。用户不必对芯片底层的寄存器进行过多的了解，只需掌握软件包提供的接口函数（API）就可以完成应用程序的编写，加快了嵌入式系统的开发效率。

该软件具备以下特点：

- 涵盖整个 STM32 微控制器产品线，方便进行芯片选型
- 可以直接选择 ST 公司生产的各类开发板进行配置
- 便捷的微控制器配置（引脚、时钟树、片内外设、中间件），并同时产生对应的初始化代码
- 代码移植方便，各个系列的微控制器程序可以很容易的移植
- 根据配置的工具链，产生 MDK-ARM、EWARM 等第三方集成开发环境的工程文件
- 根据用户定义的应用程序序列计算系统的功率
- 软件自动更新和下载，并可以根据用户的配置自动下载所需的软件包和中间件
- 跨平台，支持 Windows、Linux 和 MAC OS 等多种操作系统，并可作为插件集成在其他开发工具中

3.2.1 STM32CubeMX 安装软件的下载

CubeMX 的安装文件可以在意法半导体的官方网站上下载，下载地址如下：www.st.com/stm32cubemx。

在该页面下方的“Get Software”栏，单击红色按钮“Get Software”，将进行安装文件的下载，如图 3-2 所示。目前最新的版本为 5.4.0（截止 2019/11）。在下载软件时，如果没有官方网站的账号，会提示进行网站注册，建立账号后再下载。

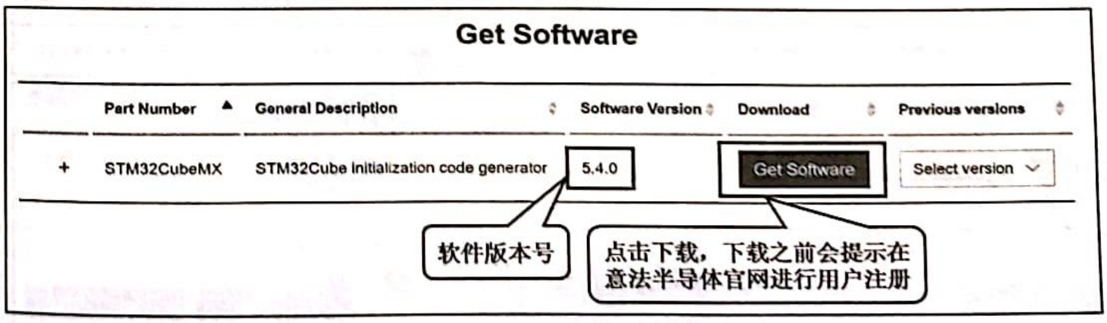


图 3-2 STM32CubeMX 安装软件的下载

3.2.2 STM32CubeMX 的安装

下载后的 CubeMX 安装文件是一个压缩包，解压后点击文件夹中以“Steup”开头的安装文件进行软件安装。安装过程比较简单，使用默认的设置即可。软件安装完成后将会在桌面生成 CubeMX 软件的快捷图标“STM32CubeMX”，如图 3-3 所示。需要特别注意的一点是：在进行软件安装路径设置时，切记不可出现中文。

如果用户的电脑中没有安装 Java 开发环境，点击安装文件后，会弹出需要安装 Java 开发环境的提示。用户根据相关提示，先下载 Java 开发环境的安装包，再运行安装包就可以完成 Java 开发环境的安装和更新。用户也可以提前下载 Java 开发环境并自行安装，下载地址如下：https://www.java.com/en/download/manual.jsp。

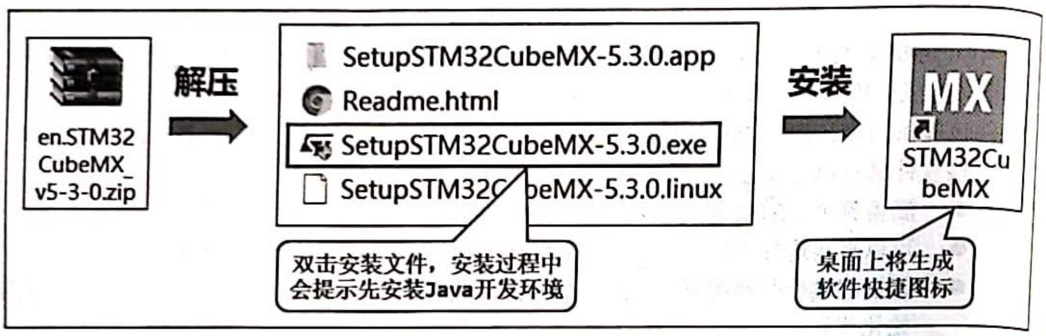


图 3-3 STM32CubeMX 软件安装过程

3.2.3 MCU 软件包的下载及安装

1. 下载 MCU 软件包

完成 CubeMX 软件的安装后，还需要安装相应的 MCU 软件包。由于本书重点介绍 STM32F4 系列微控制器，因此需要下载的 MCU 软件包为 STM32CubeF4，下载地址如下：www.st.com/stm32cubef4。

在该页面下方的“Get Software”栏，单击红色按钮“Get Software”，将进行 MCU 软件包的下载，如图 3-4 所示。目前最新的版本为 1.24.1（截止 2019/11）。下载之后的 MCU 软件包也是一个压缩文件，文件名为：en.STM32Cube_FW_F4_V x.xx.x.zip（x 表示软件包的版本号）。

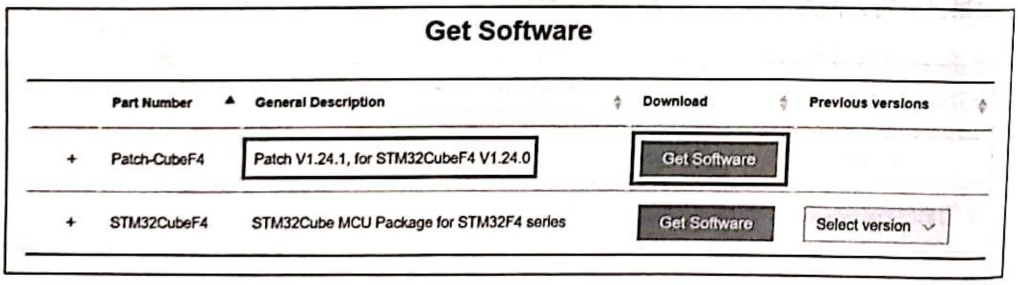


图 3-4 STM32CubeF4 软件包的下载

2． 安装 MCU 软件包

MCU 软件包的安装方法有以下三种：

方法一：在线安装

点击 CubeMX 的快捷图标，启动 CubeMX，在启动界面的菜单栏选择：Help -> Manage embedded software package，如图 3-5 所示。

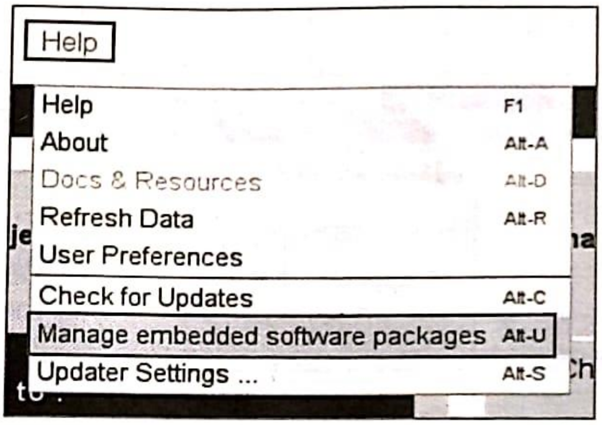


图 3-5 选择管理嵌入式软件包子菜单

在出现的管理嵌入式软件包对话框中，选择“STM32Cube MCU Package”标签页。在列表栏中，根据用户所使用的 STM32 微控制器产品系列，勾选所需的软件包。新版本的软件包修正了旧版本出现的一些错误，并增加了新的功能，因此一般选择版本号最新的软件包，然后点击“Install Now”按钮进行在线安装。操作步骤如图 3-6 中序号①～③所示。

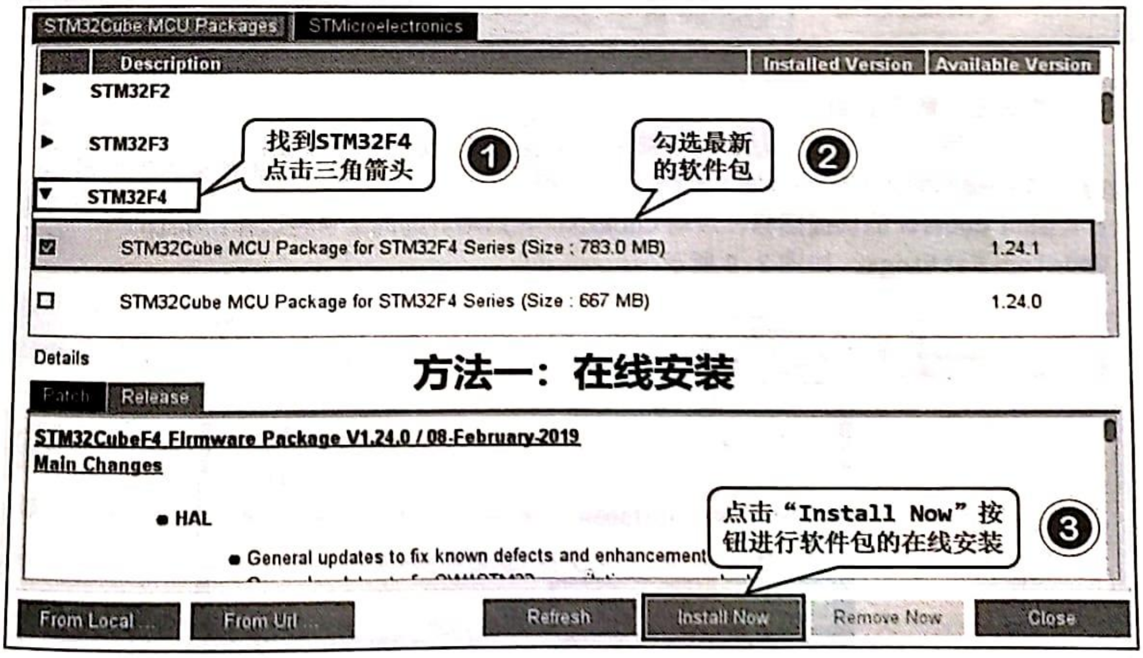


图 3-6 软件包在线安装操作步骤

方法二：离线安装

在线安装方法操作简单，但是需要通过网络下载，花费的时间较多，而离线安装只需要事先下载好 MCU 软件包，就可以直接导入，安装时间较短。

离线安装的步骤和在线安装类似，只是在出现 MCU 软件包管理窗口后，不用选择产品系列和勾选最新的软件包，只需点击“From Local”按钮，在弹出的对话框中找到之前下载的压缩文件格式的 MCU 软件包，然后点击对话框中的“打开”按钮开始软件包的导入。操作步骤如图 3-7 中序号①～③所示。

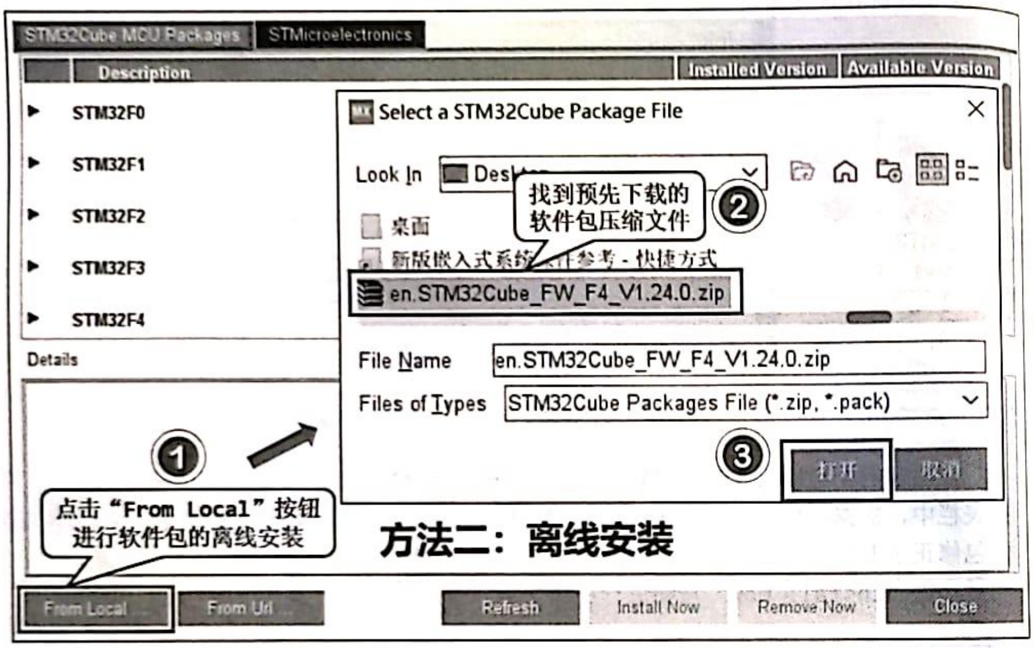


图 3-7　软件包离线安装操作步骤

方法三：解压复制

和前两种方法相比，方法三是最简单的安装方法。分成两个步骤：

(1) 找到默认的软件包安装地址

点击 CubeMX 的快捷图标，启动 CubeMX，在启动界面的菜单栏选择：Help -> Updater Settings，如图 3-8 所示。

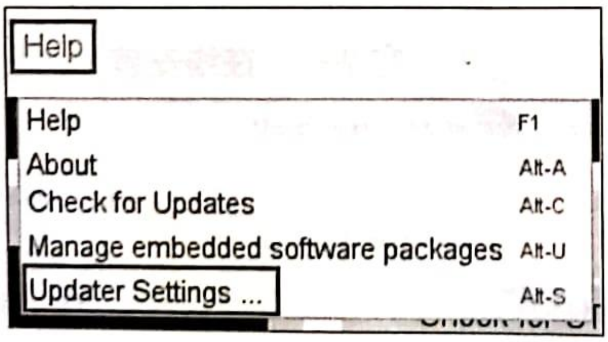


图 3-8　选择 Help 主菜单下的软件更新设置子菜单

在出现的更新设置对话框中，选择“Updater Settings”标签页，查看默认的软件包安装地址，如图 3-9 所示。

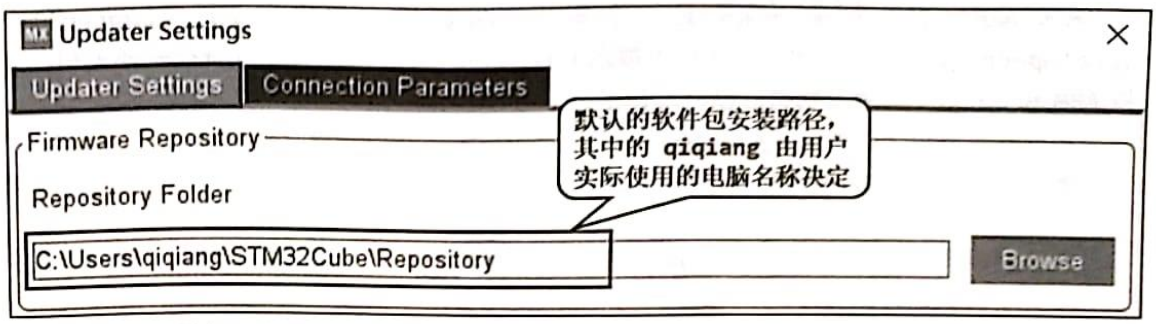


图 3-9　查看默认的软件包安装地址

(2) 解压软件包并复制到默认的软件包安装地址

将事先下载的 MCU 软件包压缩文件解压，打开解压文件夹，将文件夹中的 STM32Cube_FW_F4_Vx.xx.x 文件夹（x 表示软件包的版本号）复制到软件包安装路径中，即可完成软件包的安装。操作流程如图 3-10 所示。

注意：只复制解压文件夹中的 STM32Cube_FW_F4_Vx.xx.x 文件夹，而不是复制解压后的整个文件夹。

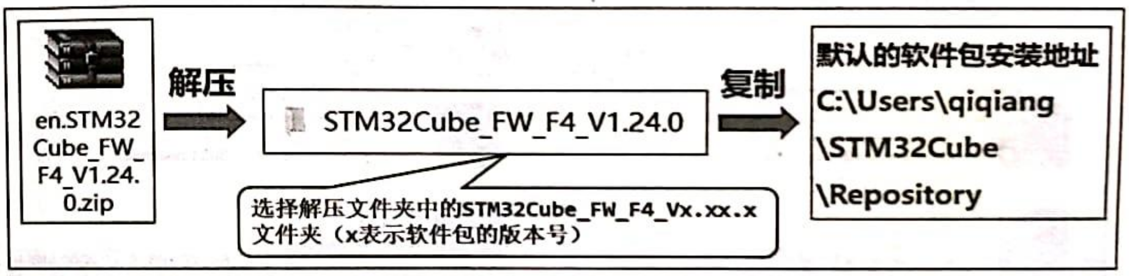


图 3-10　软件包解压复制操作流程

用户可以采用上述任何一种方法进行 MCU 软件包的安装。完成安装后，在软件包管理窗口中对应 MCU 软件包版本前面的方框将变为绿色，表明软件包已经成功安装，如图 3-11 所示。

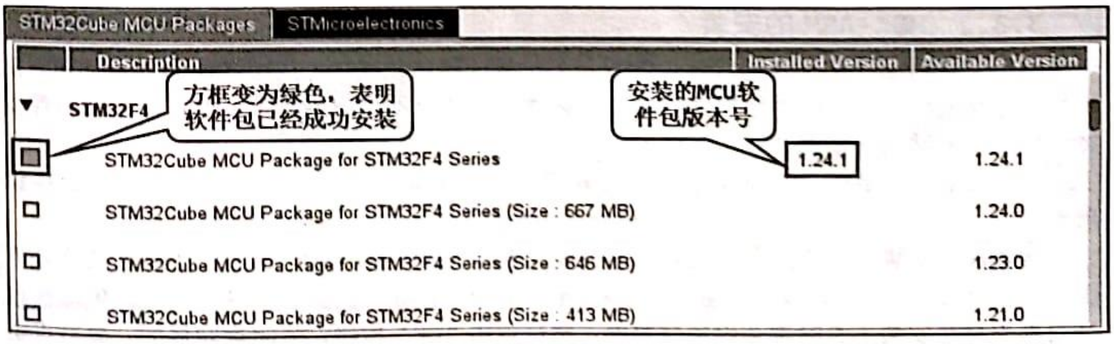


图 3-11　软件包安装成功标志

3.3 MDK-ARM 的下载及安装

MDK-ARM（以下简称 MDK）是一个集成开发环境，全称为 MicroController Development Kit for ARM。该软件由德国 Keil 公司推出，专为 32 位微控制器定制，支持 ARM 和 Cortex 等微控制器内核，是目前产业界使用最广泛的嵌入式开发工具之一。

该软件具备以下特点：

- 支持源代码编辑、编译、下载调试和模拟仿真等多种功能，涵盖嵌入式系统设计的全过程
- 将编译器和器件支持包分离，用户可以根据实际需要下载对应的器件支持包，减少了安装软件的大小
- 提供了大量的中间件，包括嵌入式操作系统、文件系统、图形用户界面和网络协议等组件，加快开发效率
- 强大的仿真功能，可以仿真整个目标硬件，包括外部信号和 I/O 仿真、中断过程仿真、片内外设仿真等

3.3.1 MDK-ARM 安装软件的下载

MDK 的安装文件可以在 Keil 公司的官方网站上下载，下载地址如下：www.keil.com/download/product/。目前最新的版本为 5.28a（截止 2019/11），如图 3-12 所示。

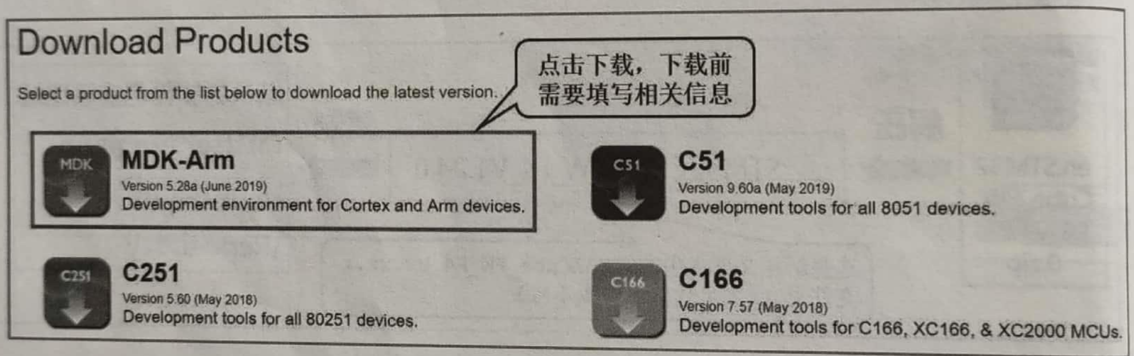


图 3-12 MDK-ARM 下载页面

点击图 3-12 中所示的“MDK-Arm”图标后，将进入下载页面，填写相关的个人信息，如邮箱，姓名等，就可以下载 MDK 的安装文件，文件名称一般为“MDK+软件版本号”的形式，如 MDK528a.exe。下载的软件为评估版，程序代码的大小不能超过 32KB。

3.3.2 MDK-ARM 的安装

点击下载后的安装文件进行软件的安装，安装过程比较简单，按照默认的配置即可。需要特别注意的一点是：在进行软件安装路径设置时，切记不可出现中文。MDK 软件及器件支持包的默认安装路径如图 3-13 所示。

软件安装完成后将会在桌面生成 MDK 软件的快捷图标“Keil uVison5”，如图 3-14 所示。

MDK 软件安装完成后，会自动弹出器件支持包安装工具（Pack installer）及欢迎界面，如图 3-15 所示。建议不勾选“Show this dialog at startup”选项，避免每次打开器件支持包安装工具时都会出现该界面，然后点击“OK”按钮，关闭欢迎界面，最后点击右上角的“×”号关闭器件支持包安装工具，操作步骤如图 3-15 中序号①~③所示。

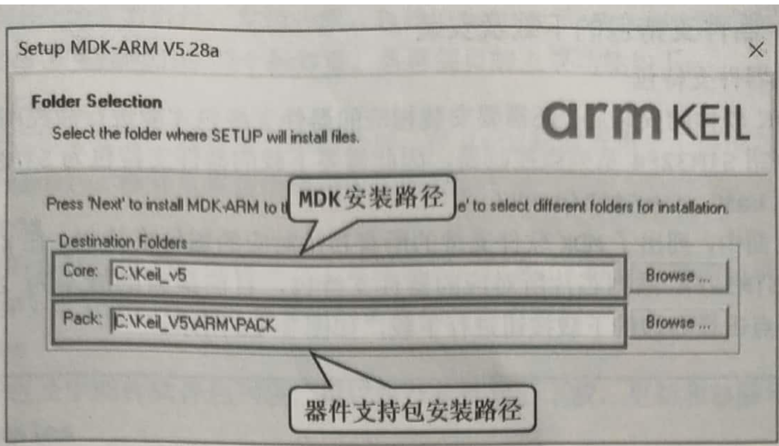


图 3-13　MDK 软件及器件支持包的安装路径

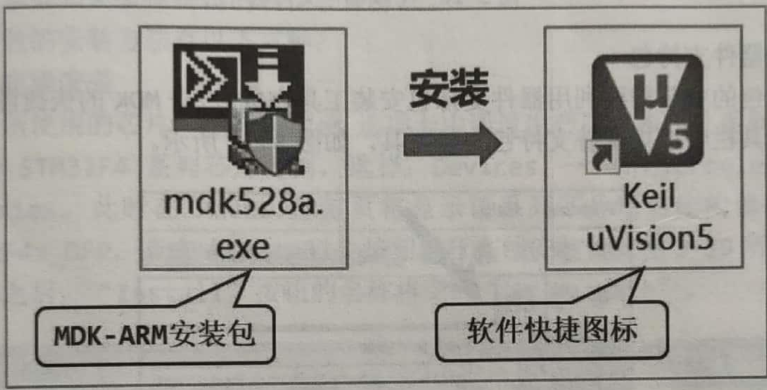


图 3-14　MDK-ARM 软件安装过程

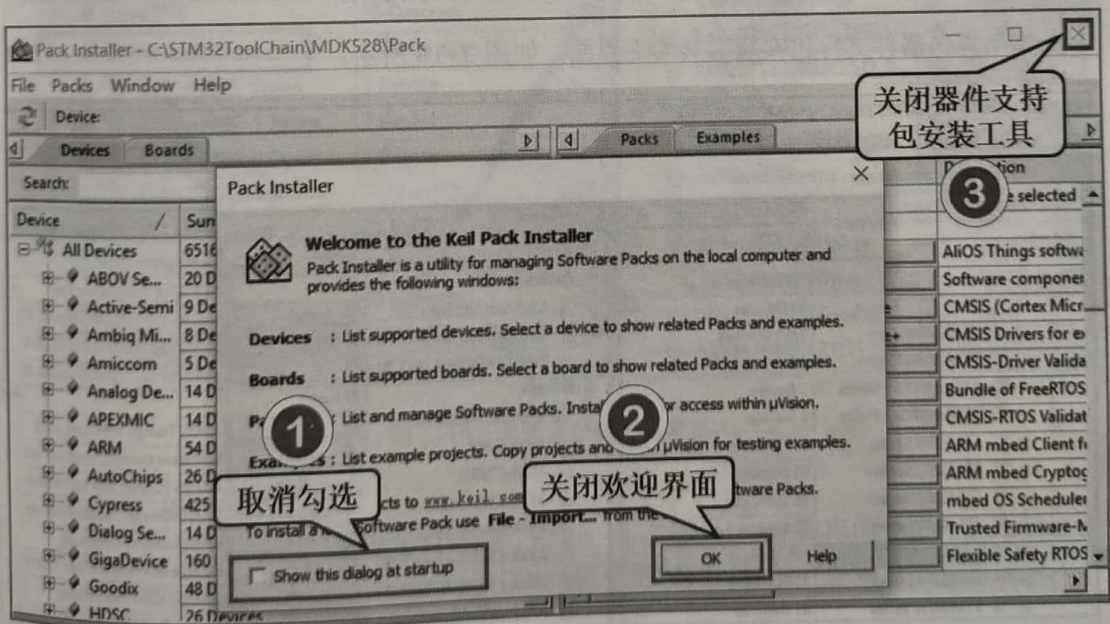


图 3-15　关闭器件支持包安装工具的操作步骤

3.3.3 器件支持包的下载及安装

1. 下载器件支持包

完成 MDK 软件的安装后，还需要安装相应的器件支持包才能进行微控制器的开发。由于本书重点介绍 STM32F4 系列微控制器，因此需要下载的器件支持包为 STM32F4，下载地址如下：www.keil.com/dd2/pack/。

在下载页面中，列出了 MDK 软件支持的所有芯片对应的器件支持包。在 Keil 公司的条目下面找到 STM32F4 系列芯片所对应的器件支持包，目前最新的版本为 2.14.0（截止 2019/11）。点击最右边的下载按钮进行下载，如图 3-16 所示。

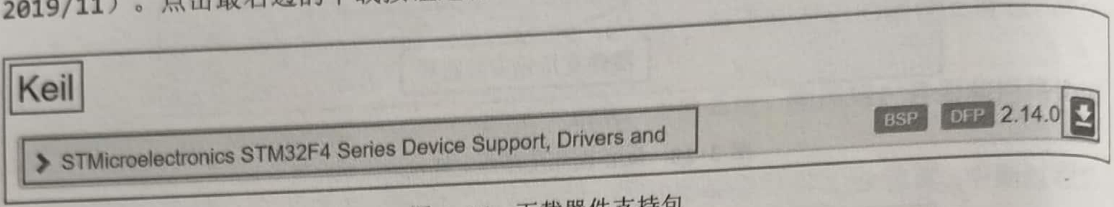


图 3-16 下载器件支持包

2. 安装器件支持包

器件支持包的安装需要利用器件支持包安装工具完成。点击 MDK 的快捷图标，启动 MDK，在主界面的工具栏中选择器件支持包安装工具，如图 3-17 所示。

图 3-17 器件支持包安装工具快捷图标

然后弹出器件支持包安装工具的主界面，如图 3-18 所示。

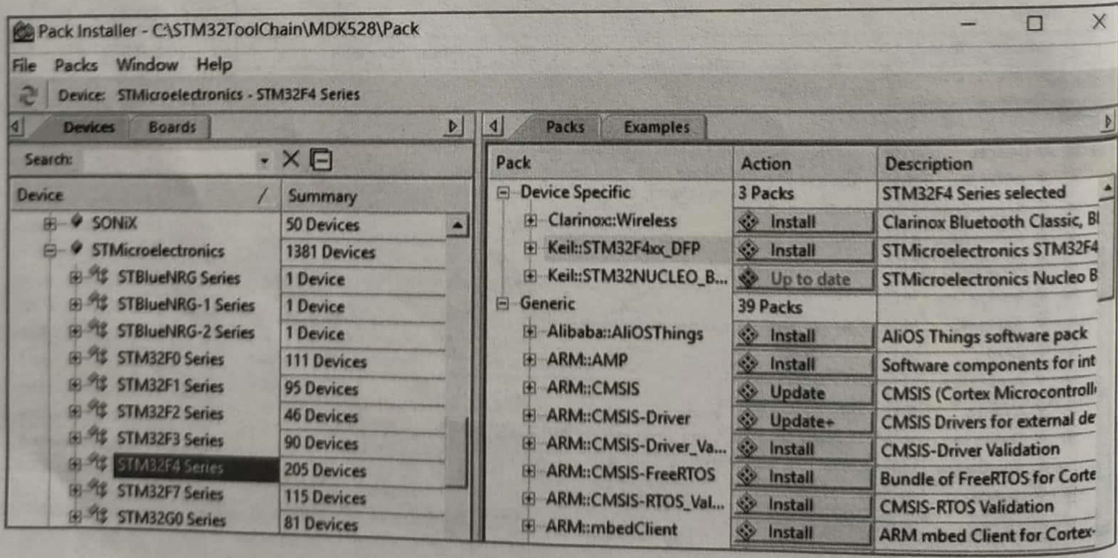


图 3-18 器件支持包安装工具的主界面

整个主界面分为左右两栏：左边一栏，包括 Devices 和 Boards 两个标签页，右边一栏则包括 Packs 和 Examples 两个标签页。各标签页的主要功能如下：

● Devices

MDK 支持的芯片列表。选择某一个芯片后，在 Pack 标签页将会出现该芯片对应的器件支持包，在 Examples 标签页将会出现相应的例程。

● Boards

MDK 支持的开发板列表。选择某一个开发板后，在 Pack 标签页将会出现该开发板的板级支持包（BSP），在 Examples 标签页将会出现相应的例程。

● Packs

器件支持包及中间件软件包列表，可以进行软件包的下载，更新和移除等操作。

● Examples

例程列表。在 Device 标签页中选择指定芯片或者 Board 标签页中选择指定开发板后，可以显示相应的参考例程。用户可以复制例程到指定的路径，通过研读这些参考例程，可以帮助我们快速掌握相关微控制器的软件设计方法。

器件支持包的安装方法有以下三种：

方法一：在线安装

根据用户所使用的芯片，在 Devices 选项卡中通过生产厂商和芯片系列来进行选择。以本书所用的 STM32F4 系列芯片为例，选择：Devices → STMicroelectronics → STM32F4 Series。此时在 Packs 标签页将显示该系列芯片所对应的器件支持件包：Keil::STM32F4x_DFP，点击“Install”按钮进行在线安装，如图 3-19 所示。

安装完成之后，“Install”按钮的名称将变成“Up to date”。

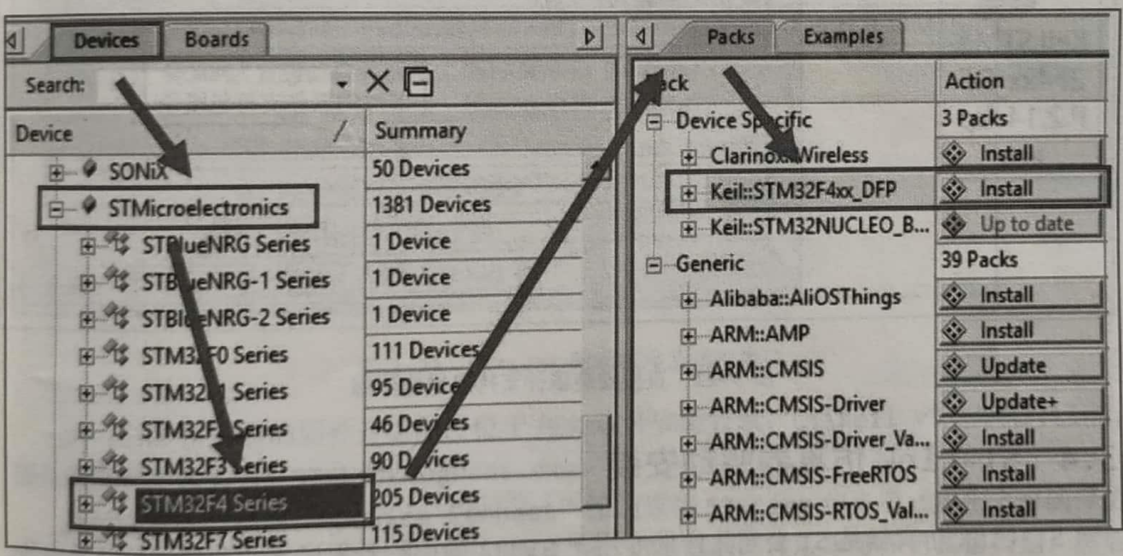


图 3-19　在线安装器件支持包

方法二：离线安装

在器件支持包安装工具的菜单栏选择：File -> Import，在弹出的 Import Packs 窗口中找到预先下载的器件支持包，然后点击对话框中的“打开”按钮开始器件支持包的导入。操作步骤如图 3-20 中序号①～③所示。

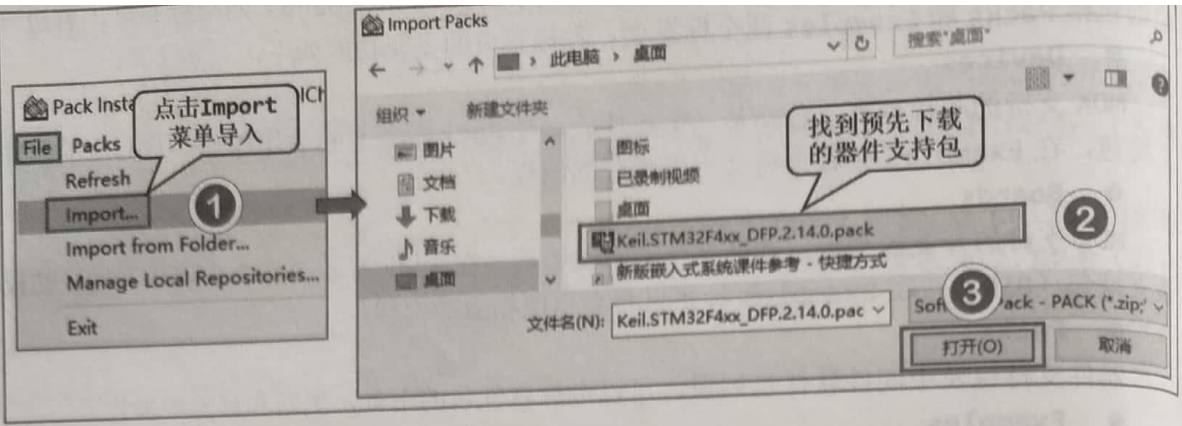


图 3-20　离线安装器件支持包操作步骤

方法三：直接安装

和前两种方法相比，方法三是最简单的安装方法。在完成 MDK 的安装后，直接点击下载的器件支持包，即可完成安装。操作步骤如图 3-21 中序号①～②所示。

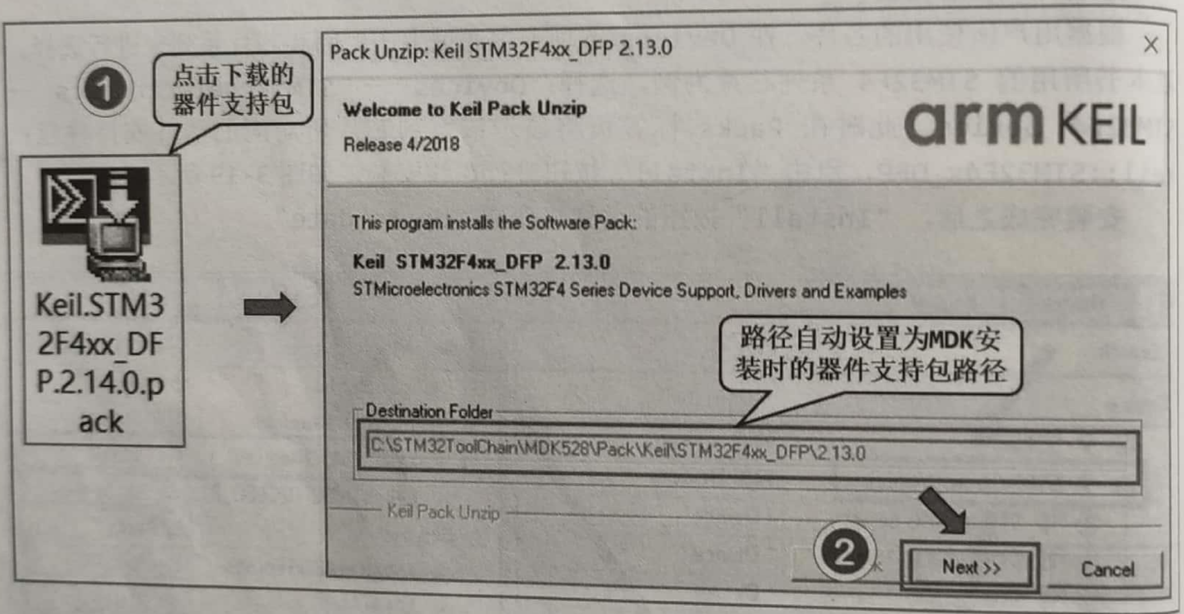


图 3-21　直接安装器件支持包操作步骤

3.4　ST-Link 仿真器驱动安装

ST-Link 仿真器是 ST 公司生产的专用于 STM32 微控制器程序下载和调试的开发工具。在 Nucleo 开发板上集成了 ST-Link 仿真器，方便用户对板载的 MCU 或外部的 MCU 进行调试。在使用 ST-Link 仿真器之前，需要安装驱动程序，以便电脑能够正确识别。

ST-Link 仿真器的驱动程序可以在 ST 官方网站通过搜索关键词“STSW-LINK007”下载。这里介绍一种简单的驱动安装方法：利用 MDK 软件自带的驱动程序。ST-Link 仿真器的驱动程序位于 MDK 软件安装目录：C:\Keil_v5\ARM\STLink\USBDriver（具体位置由用户安装 MDK 时所选择的路径决定），文件夹内容如图 3-22 所示。其中 dpinst_amd64.exe 适用于 64 位操作系统，dpinst_x86.exe 适用于 32 位操作系统，用户根据实际情况选择

对应的驱动程序进行安装。

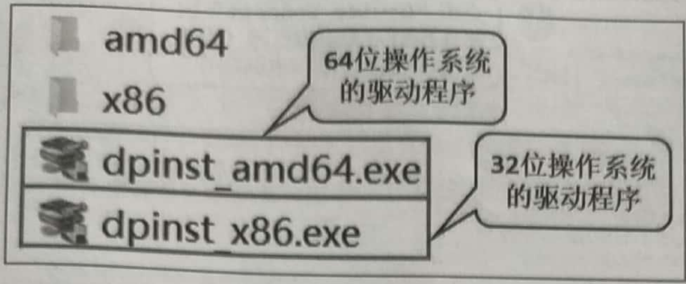


图 3-22 ST-Link 仿真器的驱动程序

如果成功安装了驱动程序，并将仿真器通过 USB 接口与电脑相连，在电脑的设备管理器窗口中的"通用串行总线设备"栏将会看到添加了 ST-Link Debug 这个驱动，同时在"端口（COM 和 LPT）"栏将会给出 ST-Link 仿真器的虚拟串口编号：COM15，如图 3-23 所示。虚拟串口的编号由系统随机分配，该编号在我们学习 UART 外设时将会用到。

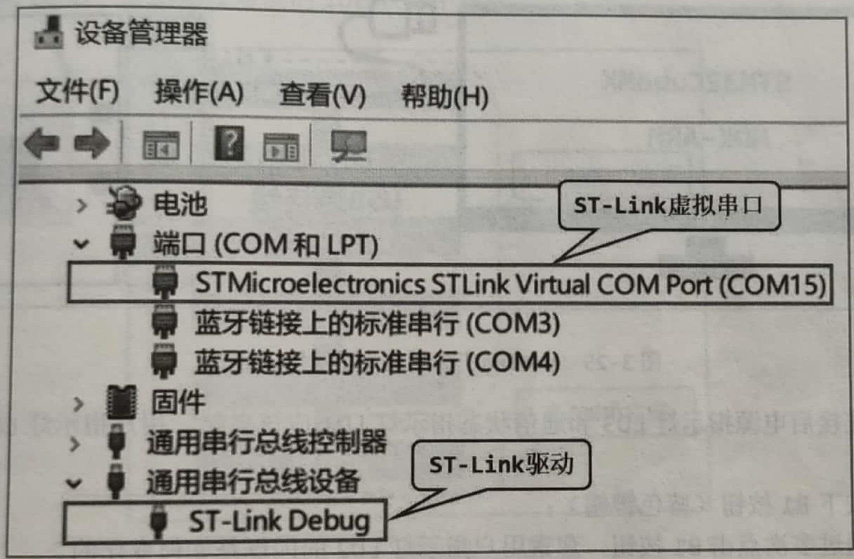


图 3-23 ST-LINK 驱动程序及虚拟串口

ST-Link 仿真器的固件升级程序位于 MDK 软件安装目录：C:\Keil_v5\ARM\STLink，固件升级程序名称为：ST-LinkUpgrade.exe。

打开该软件，点击 "Device Connect" 按钮连接 ST-Link 仿真器（点击之前确保仿真器和电脑相连），正确连接后将显示 ST-Link 仿真器的固件版本号和仿真器类型。然后点击"Yes"按钮进行固件升级，操作步骤如图 3-24 中序号①～②所示。

完成驱动安装之后，可以简单运行一下 Nucleo 开发板内置的出厂程序。

操作步骤如下：

1. 检查 Nucelo 开发板上的跳线位置：JP1 跳线短接，JP5 的跳线位于 U5V 一侧，JP6 跳线短接，CN2 两个跳线均短接（选择板载 MCU）；

2. 使用 Mini-B 型的 USB 连接线将 Nucelo 开发板与 PC 的 USB 接口相连，通过 USB 接口 CN1 给开发板供电。连接示意图如图 3-25 所示。

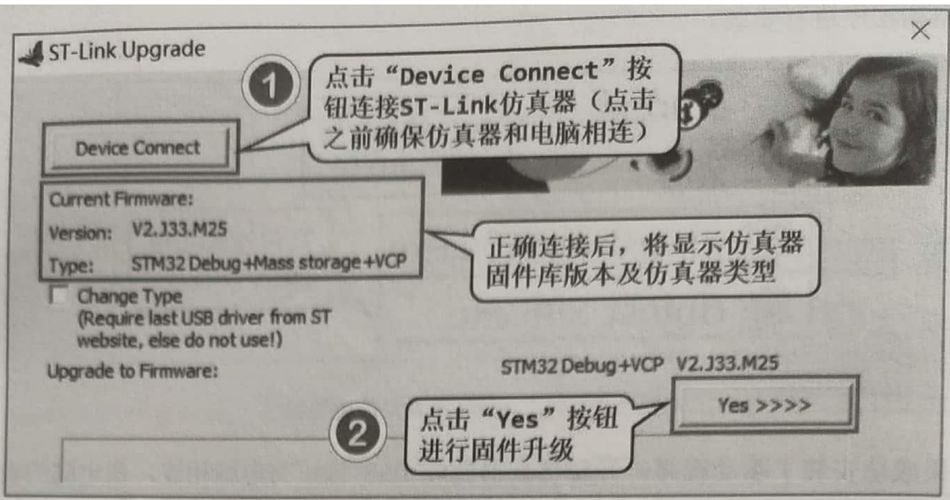


图 3-24 ST-Link 仿真器固件升级步骤

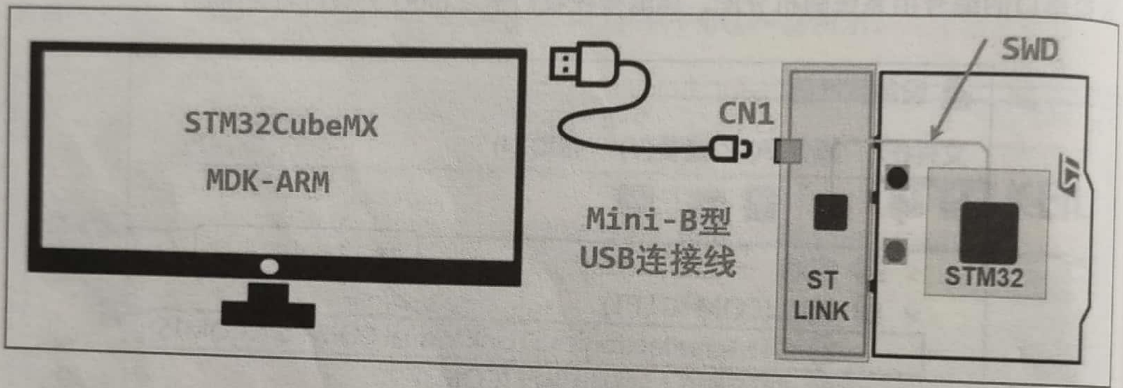


图 3-25 Nucleo 开发板与电脑连接示意图

3. 连接后电源指示灯 LD3 和通信状态指示灯 LD1 应该亮起，用户指示灯 LD2 应该闪烁；
4. 按下 B1 按钮（蓝色键帽）；
5. 通过多次点击 B1 按钮，观察用户指示灯 LD2 的闪烁是如何变化的。

3.5 本章小结

本章主要介绍了 STM32 微控制器开发环境的搭建，内容以实际操作为主，需要读者自行下载软件并按照步骤进行安装，为进一步学习 STM32 打好基础。

习题 3

1. 分析 MCU 软件包的组成以及子文件夹的作用。
2. 分析安装后的器件支持包的组成以及子文件夹的作用。
3. 下载 STM32CubeProgrammer 安装软件，进行安装，总结该软件的用法。
4. 下载 STM32CubeIDE 安装软件，进行安装，总结安装步骤。

第四章　STM32CubeMX 软件的使用

本章导读

本章重点介绍 CubeMX 软件的使用方法，操作步骤包括：目标选择 → 引脚分频 → 外设配置 → 时钟配置 → 工程配置 → 程序编写等 6 个步骤。在介绍程序编写时，通过 2 个接口函数 HAL_Delay 和 HAL_GPIO_TogglePin 的使用，让读者对 HAL 库接口函数有一个初步的认识。

本章知识点

CubeMX 软件的使用

HAL 库的接口函数

4.1 STM32CubeMX 使用流程

CubeMX 软件的使用，可以分解成 6 个步骤，如图 4-1 所示。其中前 5 个步骤在 CubeMX 软件中完成，最后一步则是在对应的 IDE 软件中完成。

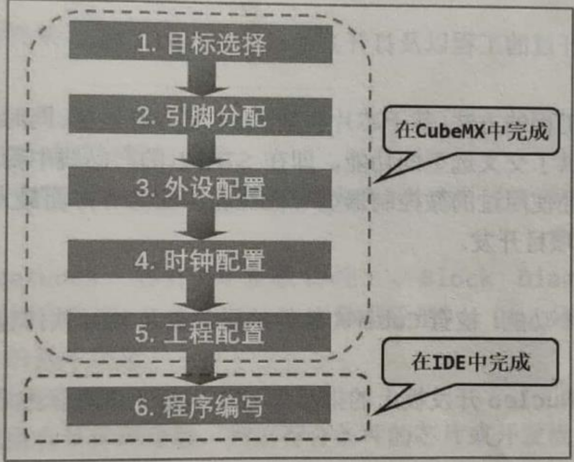


图 4-1　CubeMX 软件使用步骤

在进行 CubeMX 软件学习之前，建议读者在桌面上新建一个名为 CubeMX 的文件夹，用于存放本书所有的例程代码。对于 STM32 微控制器的外设学习，可以再建立子文件夹：按照外设名建立，如 GPIO、EXTI、TIMER、UART 等。再次强调：**CubeMX 建立的工程必须在英文路径下。**

点击桌面上的 CubeMX 软件快捷图标，启动 CubeMX 软件，进入如图 4-2 所示的 CubeMX 主界面。整个主界面主要由 4 个部分组成：

1. 菜单栏

菜单栏主要包括三个菜单，分别是 File、Window、Help。File 菜单主要包括新建工程、打开工程、退出软件等功能。Windows 菜单主要包括输出窗口的开启以及软件字体的设置。Help 菜单主要包括软件帮助文档、检查软件更新、管理 MCU 软件包等功能。

2. 已存在工程

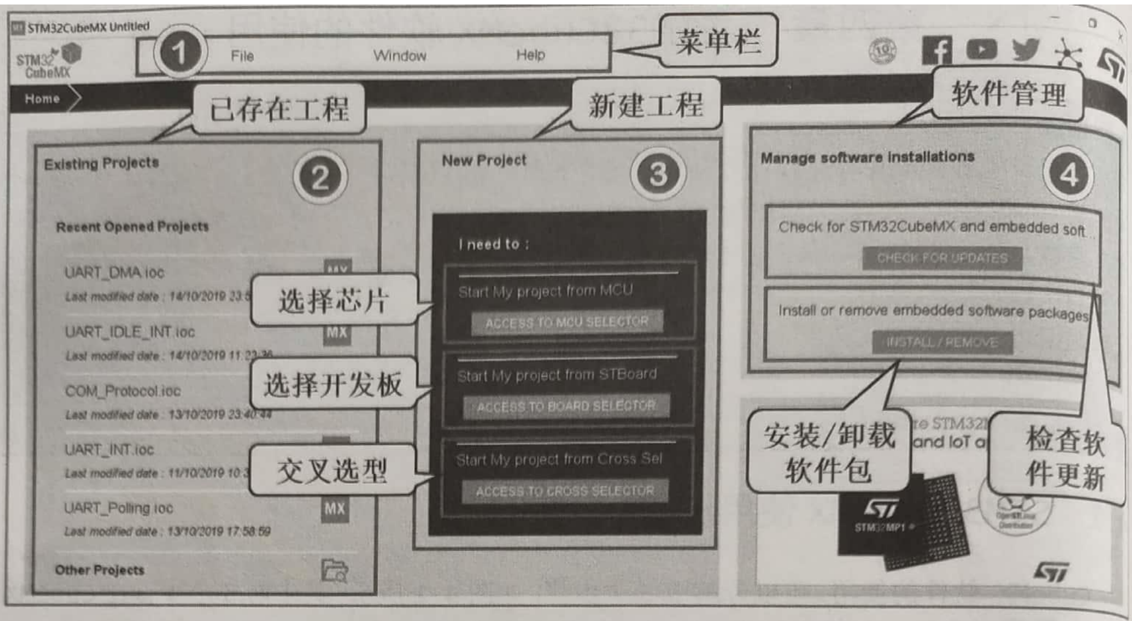


图 4-2　CubeMX 主界面

主要包括最近打开过的工程以及打开其他已经存在的工程。

3. 新建工程

提供了 2 种新建工程的方法：基于芯片和基于 ST 官方开发板。同时还为初次使用 STM32 微控制器的工程师提供了交叉选型的功能，即在 STM32 的产品线中筛选出一个芯片型号，这个型号与工程师曾经使用过的微控制器型号在性能和资源等方面最为接近，便于工程师转换到 STM32 平台进行项目开发。

4. 软件管理

软件管理包含 2 个功能：检查 CubeMX 软件的更新以及 MCU 软件包的更新；安装或者卸载 MCU 软件包。

下面我们以控制 Nucleo 开发板上的指示灯 LD2 闪烁为例来详细介绍 CubeMX 软件的 6 个使用步骤。

4.1.1 目标选择

在图 4-2 的新建工程部分提供了基于芯片和基于 ST 官方开发板的 2 种新建工程方法。由于基于 ST 官方开发板新建工程时，引脚分配、外设配置和时钟配置等工作已经自动完成，不利于初学者全面掌握 CubeMX 软件的用法，因此我们选择基于芯片的方式新建工程。

在图 4-2 中点击蓝色的按钮“Access To MCU SELECTOR”，进入芯片/开发板选择界面，如图 4-3 所示。

芯片/开发板选择界面的主要功能是选择指定的芯片/开发板型号，或者按照用户的特定要求，如处理器内核、产品系列、存储空间大小、片内外设资源等条件，筛选出满足要求的芯片/开发板型号。

芯片/开发板选择界面主要由 4 个部分组成：

1. 标签页切换

用于在 MCU/MPU SELECTOR（芯片选择器）、BOARD SELECTOR（开发板选择器）、CROSS SELECTOR（交叉选型选择器）三个标签页之间切换。芯片选择器用于选择芯片型号；开发板选择器用于选择 ST 官方的开发板；交叉选型选择器则用于初次使用 STM32 微控制器的工

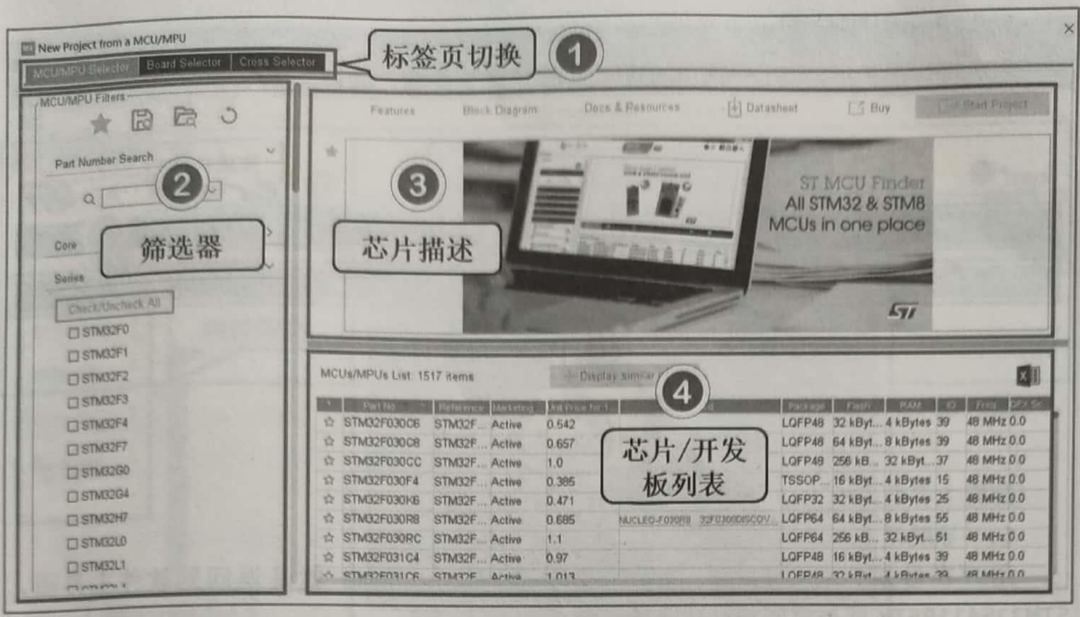


图 4-3　芯片/开发板选择界面

程师进行硬件平台的转换。

2. 筛选器

提供了 2 种筛选方法：

(1) 直接输入芯片或开发板的型号。

(2) 利用处理器内核、产品系列、存储空间大小、片内外设资源等组合条件进行筛选。

3. 芯片描述

主要提供了 Features（芯片/开发板特性）、Block Diagram（功能框图）、Doc&Resources（文档和资源）、Datasheet（数据手册）、Buy（购买渠道）等信息，方便用户快速掌握芯片的基本功能。

4. 芯片/开发板列表框

通过输入型号或组合条件筛选后，列出符合条件的芯片或开发板型号。

我们使用的 Nucleo 开发板，目标芯片为 STM32F411RECT6。直接在筛选器的 Part Number Seach（型号搜索）栏输入芯片型号：STM32F411RE。在芯片/开发板列表框中将显示对应的芯片型号。双击列表框中的芯片型号，完成目标选择。操作步骤如图 4-4 所示。

图 4-4　芯片型号选择

4.1.2 引脚分配

完成目标选择后，将进入 CubeMX 软件的配置主界面，如图 4-5 所示。

图 4-5 配置主界面

配置主界面主要由 3 个部分组成：

1. 导航栏

用户在导航栏可以切换软件的工作步骤，比如 Home 就是返回到软件主界面，STM32F411RETx 就是返回到目标选择阶段。

2. 代码生成（GENERATE CODE）

用户完成 CubeMX 软件中的全部配置步骤之后，利用该按钮生成 MDK 或者 EWARM 等集成开发环境的工程。

3. 功能标签页

功能标签页主要用于完成芯片和工程的配置，一共由 4 个标签页：

(1) Pinout & Configutation

主要用于引脚分配以及外设和中间件的配置。

(2) Clock Configuration

主要用于系统时钟的设置

(3) Project Manager

主要用于生成工程的配置

(4) Tool

主要用于系统功耗的估算

我们首先选择 Pinout & Configutation 标签页，完成引脚分配以及外设和中间件的配置。该标签页的界面如图 4-6 所示，主要由 2 个部分组成。

1) Categories（类别）

完成外设和中间件的配置。按照功能的不同，将外设及中间件分成了七大类：

① System Core

用于配置 I/O 接口、时钟源、中断系统以及系统相关外设

② Analog

用于配置模/数和数/模转换外设

③ Timer

用于配置定时器和实时时钟外设

④ Connectivity

用于配置 I^2C、SPI 和 UART 等串行通信外设

⑤ Multimedia

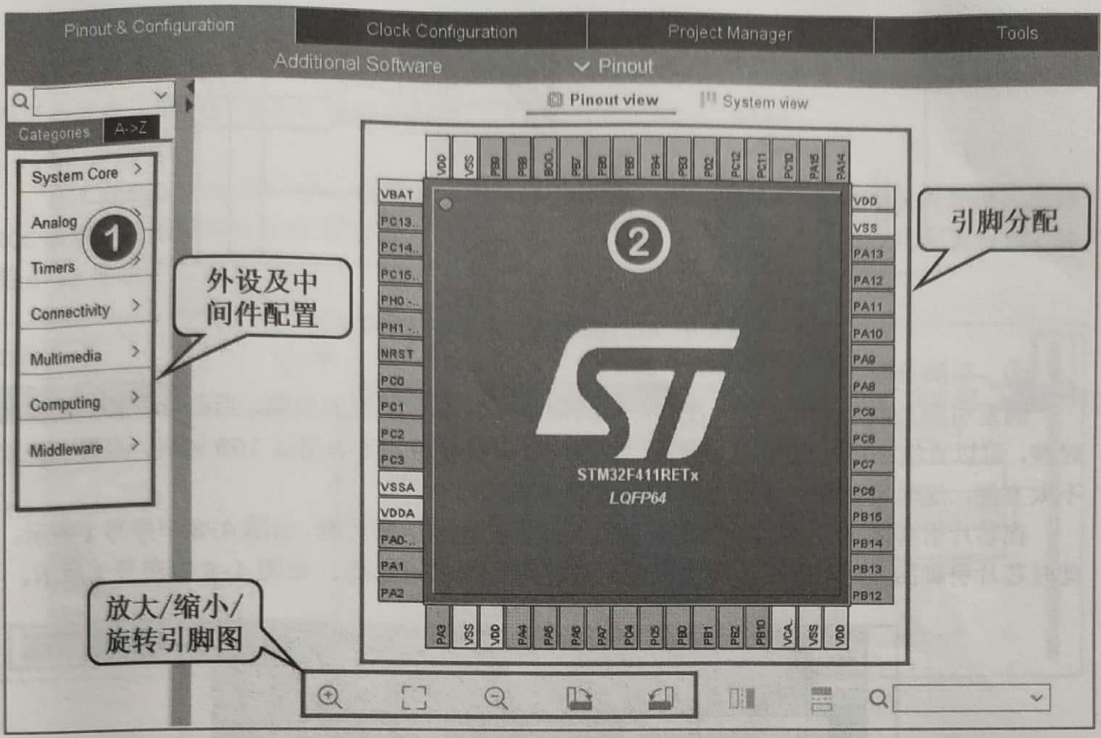


图 4-6　引脚分配标签页

⑥ Multimedia

用于配置 I^2S 等音频数据传输外设

⑦ Computing

用于配置 CRC 校验外设

⑧ Middleware

用于配置 FreeRTOS 嵌入式操作系统和 FATFS 文件系统等中间件

注意：不同型号的芯片，片内集成的外设不同，各个分类下显示的外设类型和数量也各不相同。

2) Pinout View（芯片引脚图）

用于完成引脚分配，引出了芯片的全部引脚，并通过不同的颜色显示引脚的功能。其中黄色表示电源引脚；卡其色表示 NRST（复位引脚，低电平有效）和 BOOT0（启动选择引脚），这两类引脚用户都不能修改。灰色表示 GPIO 引脚，默认的状态是复位状态 Reset_State，当用户修改某一个 GPIO 引脚的功能后，该引脚的颜色将变为亮绿色。

引脚图的大小和旋转方向可以通过其下方的几个按钮进行修改，如图 4-6 中所示。也可以点击引脚图后，利用鼠标滚轮进行放大和缩小，利用鼠标左键拖动引脚图来改变它的位置。

下面我们在芯片引脚图中进行引脚的分配，操作步骤如下：

① 确定引脚功能

由于微控制器的引脚数量有限，每一个引脚都具备多个功能。用户需要根据实际情况来选择对应的引脚功能。在 Nucleo 开发板中，控制指示灯 LD2 的引脚为 PA5，引脚功能配置为通用输入/输出功能（GPIO），方向为输出。输出高电平时，LD2 开启；输出低电平时，LD2 关闭。电路原理图如图 4-7 所示。

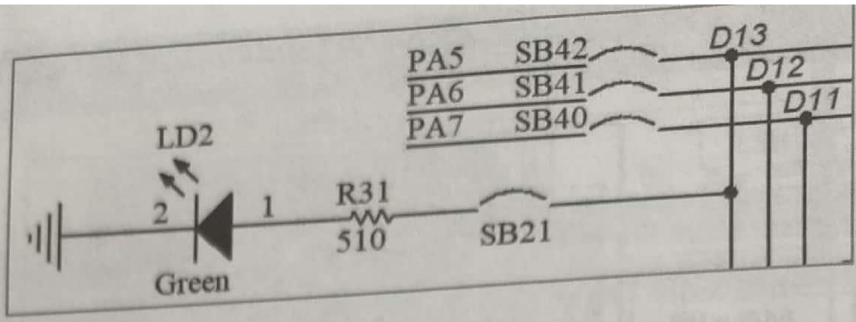


图 4-7 Nucleo 开发板指示灯驱动电路

② 引脚分配

确定引脚功能之后，就可以在芯片引脚图上找到需要配置的引脚。当芯片引脚比较少的时候，可以直接在芯片引脚图上查找。当芯片引脚较多时，比如超过 100 脚时，直接查找就不太方便，这时可以利用引脚搜索框来查找引脚。

在芯片引脚图下方的引脚搜索框中输入需要配置的引脚名称，如图 4-8 中序号 1 所示。此时芯片引脚图上对应的引脚将会闪烁，以便用户找到该引脚，如图 4-8 中序号 2 所示。

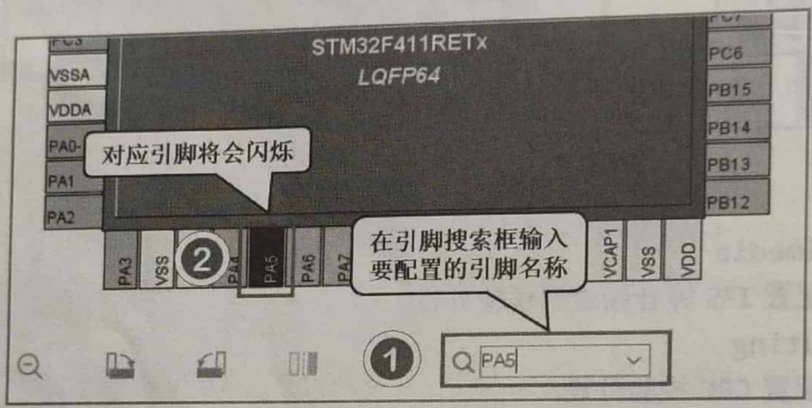


图 4-8 引脚查找

找到需要配置的引脚后，鼠标左键点击该引脚，将弹出一个下拉菜单，显示该引脚可以选择的功能。由于我们是控制指示灯 LD2，引脚功能为 GPIO，引脚方向为输出，因此选择 GPIO_Output 即可，如图 4-9 中序号 1 所示。完成功能选择后，PA5 的名称变为 GPIO_Output，引脚颜色变为亮绿色，如图 4-9 中序号 2 所示。

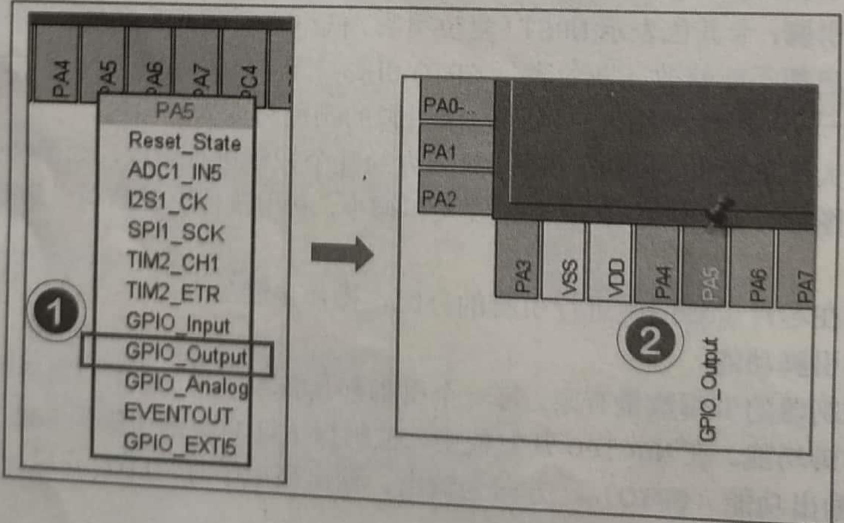


图 4-9 引脚功能设置

4.1.3 外设配置

完成引脚分配后，我们进入外设配置环节，依然还是在 Pinout & Configuration 标签页中进行。需要设置的外设如下：

1. 时钟源

在 Categories 窗口中，选择 System Core 类，点击“>”符号展开该类外设。选择 RCC 模块，将出现 RCC 的配置窗口。用户也可以利用左、右箭头展开或者收起配置窗口。如图 4-10 所示。

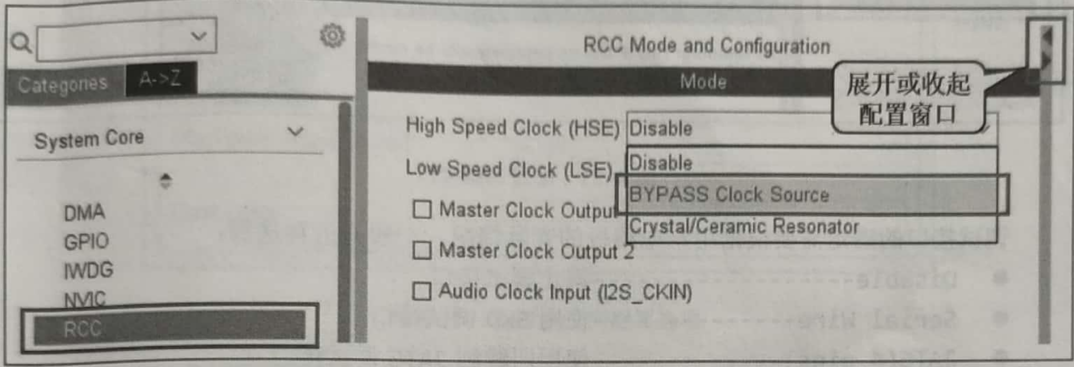


图 4-10 选择时钟源

时钟源的配置主要是根据用户电路板的实际情况选择恰当的时钟源，一般有三种选择：

(1) Disable（禁止外部时钟）

主要用于使用 MCU 内部的 RC 振荡器作为时钟源的情况。

(2) BYPASS Clock Source（旁路时钟）

主要用于直接加入外部时钟信号(可以是占空比为 40%～60%的方波、正弦波或三角波）的情况。

(3) Crystal/Ceramic Resonator（晶振/陶瓷振荡器）

主要用于使用外部震荡电路产生时钟的情况，如外接 4MHz～26MHz 的石英晶体和电容构成的时钟电路。

对于 Nucleo 开发板而言，可以使用两种方式的时钟源配置：

方式一：内部 RC 振荡器

使用了内部的 RC 振荡器作为时钟源（HSI）。在这种方式下，高速时钟（HSE）选择 Disable 选项，其他配置参数使用默认值。

方式二：旁路时钟

使用 ST-Link 仿真器提供的 8MHz 外部时钟。在这种方式下，高速时钟（HSE）选择 BYPASS Clock Source，其他配置参数使用默认值。

由于内部的 RC 振荡器一般会有 1%左右的误差，因此我们选择 BYPASS Clock Source（旁路时钟）的方式，使用 ST-Link 仿真器提供的 8MHz 信号作为时钟源。

2. 调试接口

在 Categories 窗口中，选择 System Core 类，点击“>”符号展开该类外设。选择 SYS 模块，将出现 SYS 的配置窗口。如图 4-11 所示。

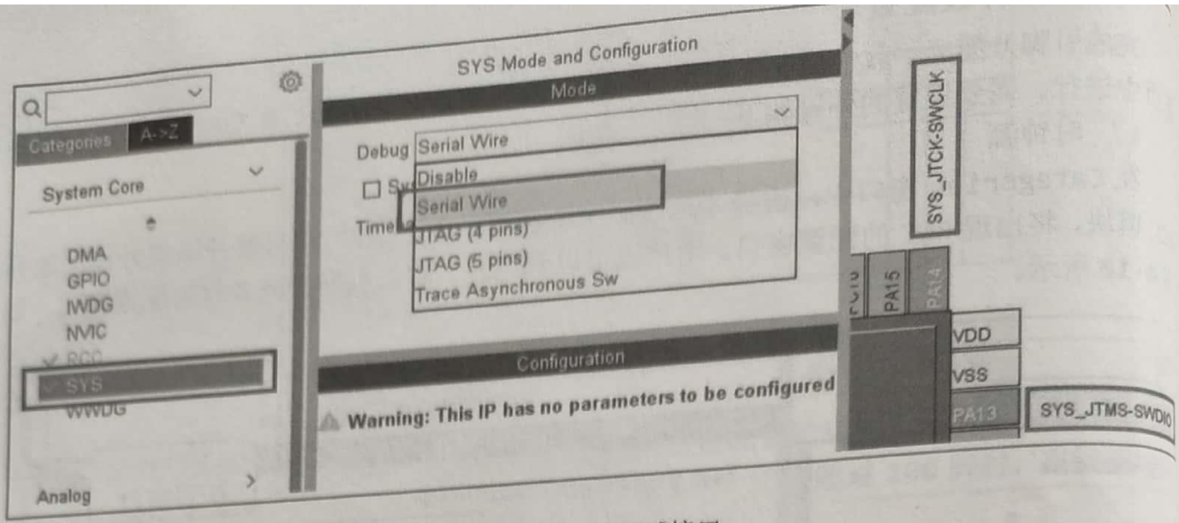


图 4-11 配置调试接口

调试接口的确定需要根据用户电路板的实际情况，一般有五种选择：

- Disable-----------------禁止调试接口
- Serial Wire--------------使用 SWD 调试接口
- JATG(4 pins)-------------使用四线制 JATG 调试接口
- JATG(5 pins)-------------使用五线制 JATG 调试接口
- Trace Asynchronous Sw----使用跟踪调试接口（SWO）

Nucleo 开发板的目标芯片使用了 SWD 调试接口与板载的 ST-Link 仿真器相连，因此调试接口选择 Serial Wire，其他的配置参数保持默认值。此时芯片引脚图中 SWD 调试接口所对应的引脚 PA14 将设置为 SWCLK，PA15 将设置为 SWDIO，引脚颜色也变成了亮绿色，如图 4-11 所示。

注意：用户在利用 CubeMX 进行芯片配置时，一定要使能调试接口。否则当程序第一次下载后，调试接口按照默认的“Disable”配置模式，将被禁用，无法完成后续的程序下载。

3. GPIO

在 4.1.2 节中我们完成了引脚分配，这里主要是设置引脚相关参数。在 Categories 窗口中，选择 System Core 类，点击“>”符号展开该类外设。选择 GPIO 模块，将出现 GPIO 的配置窗口，如图 4-12 所示。

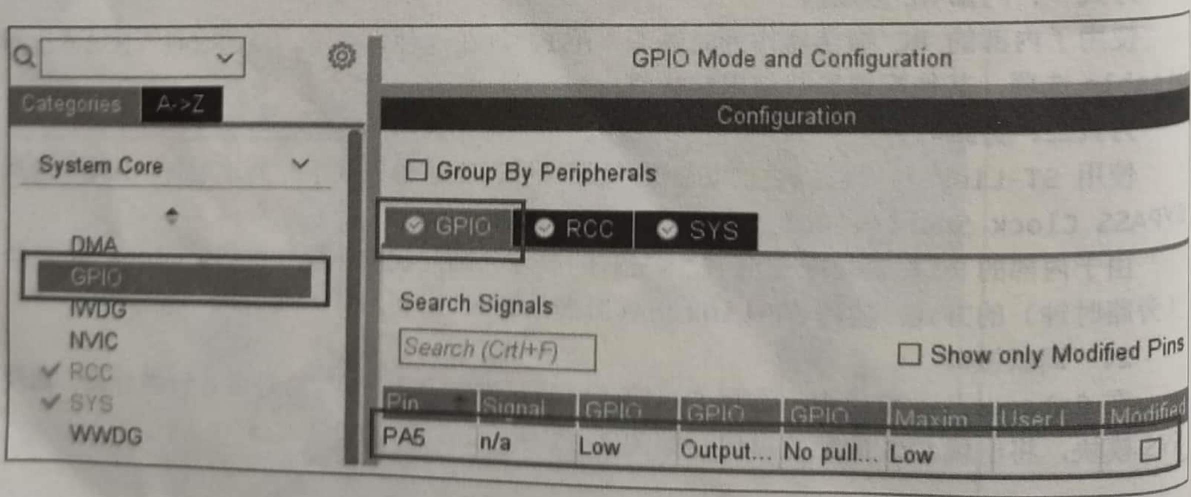


图 4-12 引脚配置设置

在 GPIO 配置窗口中，选择 GPIO 标签页，在引脚列表中选择 PA5，将显示 PA5 的具体配置选项，如图 4-13 所示。关于 GPIO 引脚的配置方法，我们将在第六章中详细介绍，这里我们暂时使用默认值。

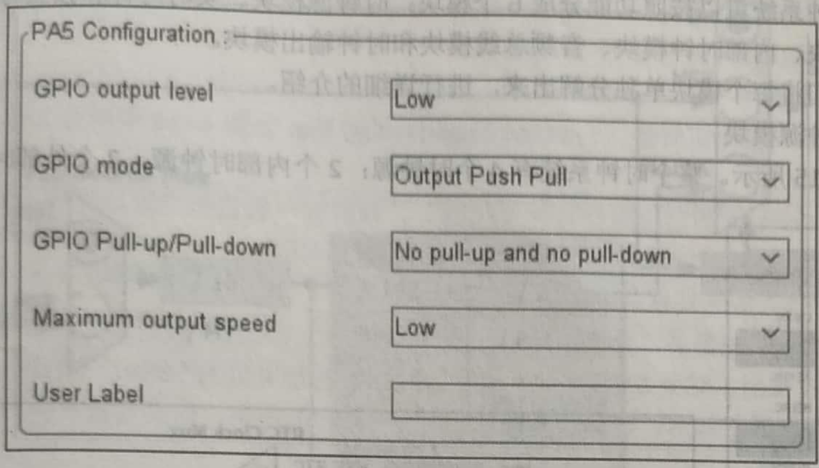


图 4-13 引脚 PA5 的配置选项

4.1.4 时钟配置

在配置主界面中选择 Clock Configuration 标签页，进行时钟配置。配置界面如图 4-14 所示。时钟配置界面显示内容的大小可以通过按下“Ctrl”键后，再利用鼠标滚轮进行缩放。

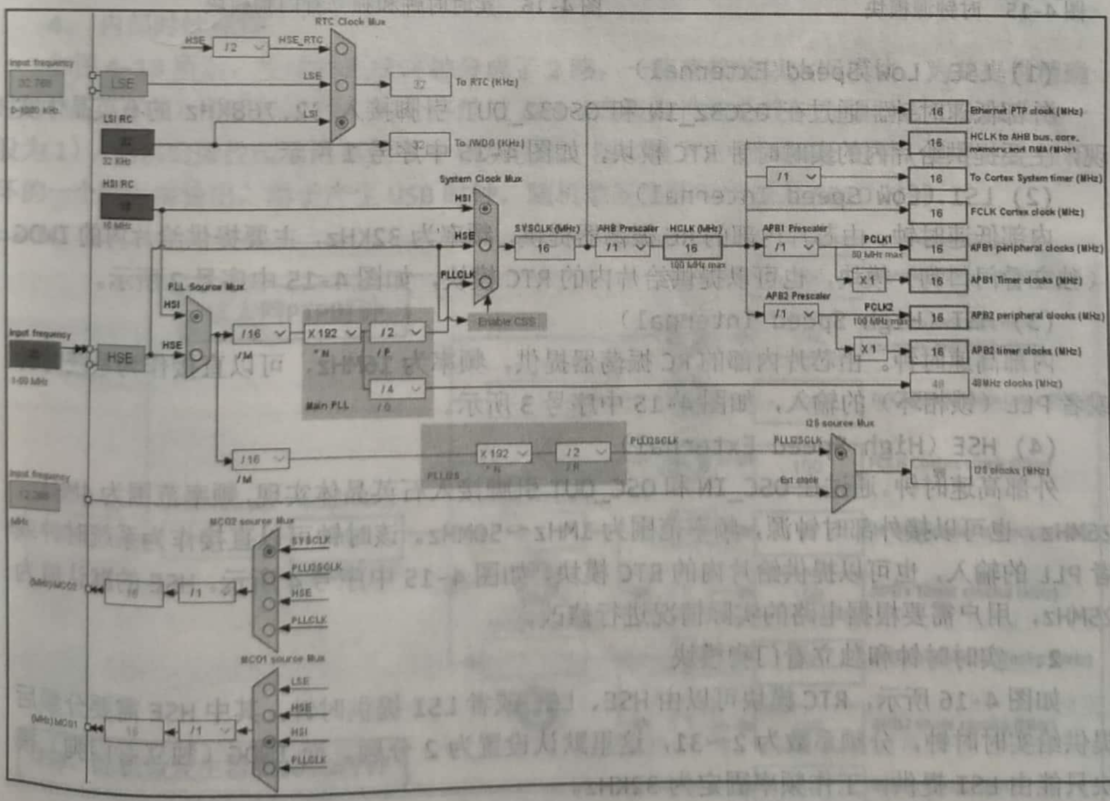


图 4-14 STM32 时钟树结构图

时钟配置界面以时钟树的形式清晰地展示了芯片内部的时钟结构，是学习和掌握时钟系统的一个辅助工具。这里我们对 STM32F411 系列芯片的时钟系统进行一个详细的分析，其他系列芯片的时钟系统也与之类似。

整个时钟系统可以按照功能分成 6 个模块：时钟源模块、实时时钟和独立看门狗模块、系统时钟模块、内部时钟模块、音频总线模块和时钟输出模块。

下面我们将每个模块单独分解出来，进行详细的介绍。

1. 时钟源模块

如图 4-15 所示。整个时钟系统有 4 个时钟源：2 个内部时钟源，2 个外部时钟源。

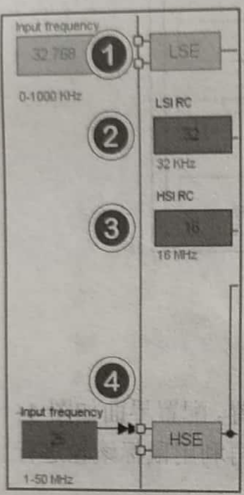


图 4-15　时钟源模块

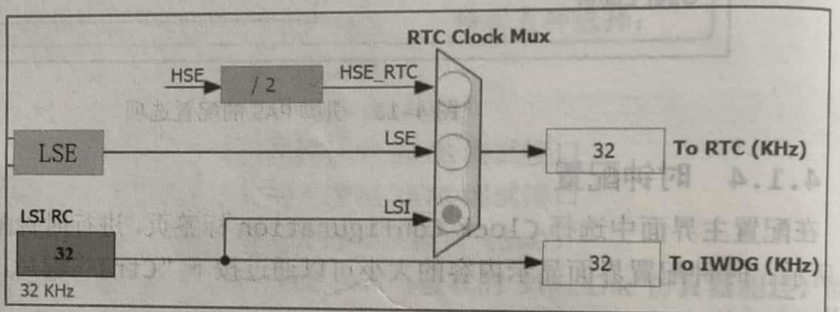


图 4-16　实时时钟和独立看门狗模块

(1) LSE（Low Speed External）

外部低速时钟。通过在 OSC32_IN 和 OSC32_OUT 引脚接入 32.768KHz 的石英晶体实现，主要提供给片内的实时时钟 RTC 模块，如图 4-15 中序号 1 所示。

(2) LSI（Low Speed Internal）

内部低速时钟。由芯片内部的 RC 振荡器提供，频率为 32KHz，主要提供给片内的 IWDG（独立看门口狗）模块，也可以提供给片内的 RTC 模块，如图 4-15 中序号 2 所示。

(3) HSI（High Speed Internal）

内部高速时钟。由芯片内部的 RC 振荡器提供，频率为 16MHz，可以直接作为系统时钟或者 PLL（锁相环）的输入，如图 4-15 中序号 3 所示。

(4) HSE（High Speed External）

外部高速时钟。通过在 OSC_IN 和 OSC_OUT 引脚接入石英晶体实现，频率范围为 4MHz~26MHz。也可以接外部时钟源，频率范围为 1MHz~50MHz。该时钟可以直接作为系统时钟或者 PLL 的输入，也可以提供给片内的 RTC 模块，如图 4-15 中序号 4 所示。HSE 的默认值为 25MHz，用户需要根据电路的实际情况进行修改。

2. 实时时钟和独立看门狗模块

如图 4-16 所示，RTC 模块可以由 HSE、LSE 或者 LSI 提供时钟，其中 HSE 需要分频后提供给实时时钟，分频系数为 2~31，这里默认设置为 2 分频。而 IWDG（独立看门狗）模块只能由 LSI 提供，工作频率固定为 32KHz。

3. 系统时钟模块

如图 4-17 所示，SYSCLK（系统时钟）可以用 HSI 和 HSE 独立产生或者二者之一经过

Main PLL（主锁相环）产生。

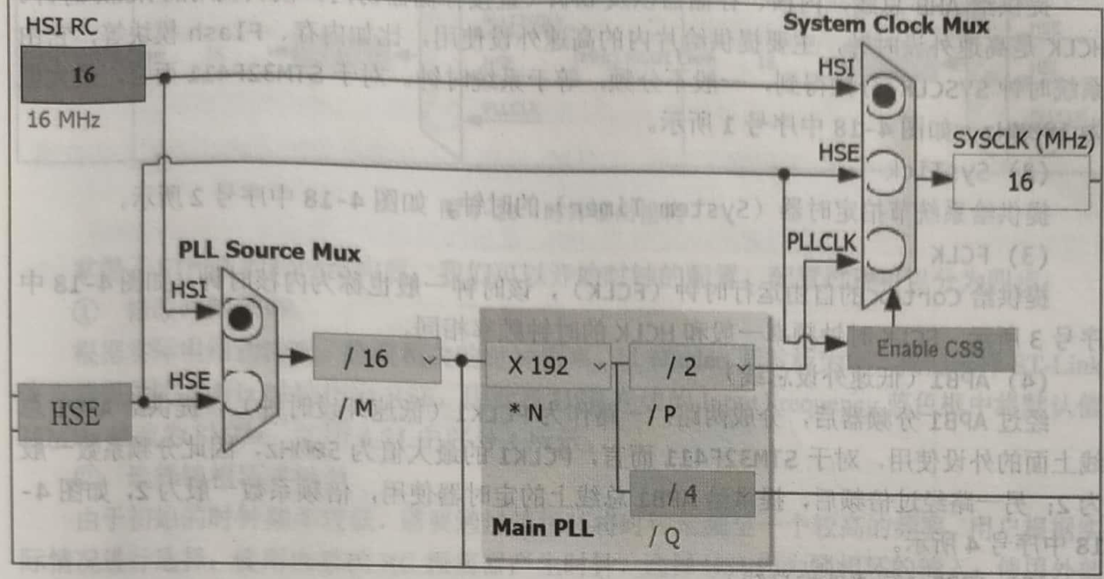


图 4-17 系统时钟模块

主锁相环用于对时钟的倍频，使得微控制器内部的系统时钟可以工作在一个较高的频率。以 STM32F411 为例，系统时钟的最大值为 100MHz。锁相环中的配置参数 M、N、R 和 Q 由 CubeMX 软件自动生成，用户无需手动计算。主锁相环一共有 2 个输出端：一个用于产生系统时钟，一个用于产生 USB 时钟、随机数发生器时钟以及用于 MicroSD 卡的 SDIO 时钟。

4. 内部时钟模块

如图 4-18 所示，生成的系统时钟分成了 2 路，一路直接给以太网模块，为其提供精确时钟协议（PTP）所需的时钟；另一路经过 AHB（高级高性能总线）分频器后（分频系数一般为 1），提供给微控制器内部的 5 个模块使用。图中最下方的 48MHz 时钟，则是由主锁相环的一个输出端输出，用于产生 USB 时钟，随机数发生器时钟以及 SDIO 时钟。

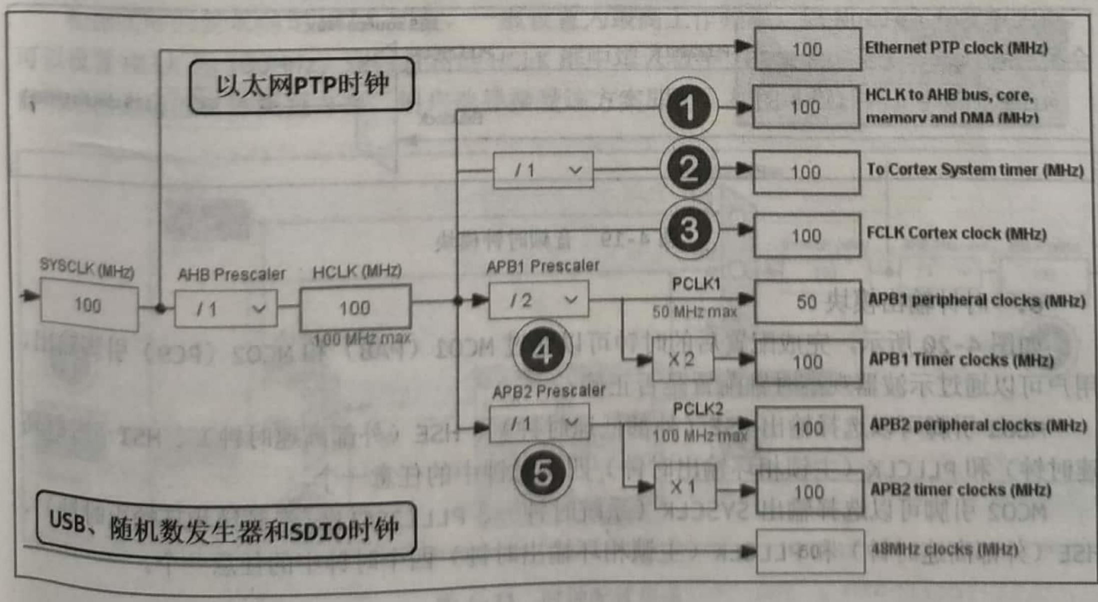


图 4-18 内部时钟模块

（1）HCLK

提供给 AHB 总线，内核、存储器以及 DMA（直接存储器访问）模块使用的 HCLK 时钟，HCLK 是高速外设时钟，主要提供给片内的高速外设使用，比如内存、Flash 模块等，它由系统时钟 SYSCLK 分频得到，一般不分频，等于系统时钟。对于 STM32F411 而言，最大值为 100MHz，如图 4-18 中序号 1 所示。

（2）SysTick

提供给系统节拍定时器（System Timer）的时钟，如图 4-18 中序号 2 所示。

（3）FCLK

提供给 Cortex 的自由运行时钟（FCLK），该时钟一般也称为内核时钟，如图 4-18 中序号 3 所示，FCLK 时钟频率一般和 HCLK 的时钟频率相同。

（4）APB1（低速外设总线）

经过 APB1 分频器后，分成两路：一路作为 PCLK1（低速外设时钟），提供给 APB1 总线上面的外设使用，对于 STM32F411 而言，PCLK1 的最大值为 50MHz，因此分频系数一般为 2；另一路经过倍频后，提供给 APB1 总线上的定时器使用，倍频系数一般为 2，如图 4-18 中序号 4 所示。

（5）APB2（高速外设总线）

经过 APB2 分频器后，分成两路：一路作为 PCLK2（高速外设时钟），提供给 APB2 总线上面的外设使用，对于 STM32F411 而言，PCLK2 的最大值为 100MHz，因此分频系数一般为 1；另一路经过倍频后，提供给 APB2 总线上的定时器使用，倍频系数一般为 1，如图 4-18 中序号 5 所示。

注意：为了降低系统功耗，挂接在 AHB 或 APB 总线上的外设都有单独的时钟使能控制，默认处于关闭状态。使用外设时，必须先开启外设的时钟，才能正常工作。

5. 音频时钟模块

如图 4-19 所示，音频时钟模块有 2 个时钟源：一个是通过 PLLI2S（音频时钟锁相环）提供，该锁相环的输入时钟来源于 HSI（内部高速时钟）或者 HSE（外部高速时钟），其中的配置参数 N 和 R 由 CubeMX 软件自动生成；另一个则由外部时钟直接提供。

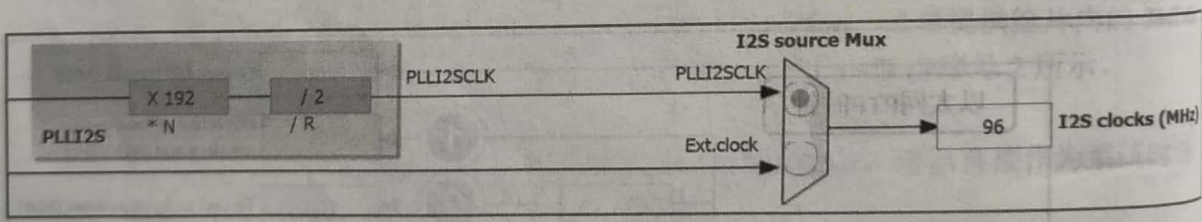


图 4-19　音频时钟模块

6. 时钟输出模块

如图 4-20 所示，完成配置后的时钟可以通过 MCO1（PA8）和 MCO2（PC9）引脚输出，用户可以通过示波器观察时钟配置是否正确。

MCO1 引脚可以选择输出 LSE（外部低速时钟）、HSE（外部高速时钟）、HSI（内部高速时钟）和 PLLCLK（主锁相环输出时钟）四个时钟中的任意一个。

MCO2 引脚可以选择输出 SYSCLK（系统时钟）、PLLI2SCLK（音频锁相环输出时钟）、HSE（外部高速时钟）和 PLLCLK（主锁相环输出时钟）四个时钟中的任意一个。

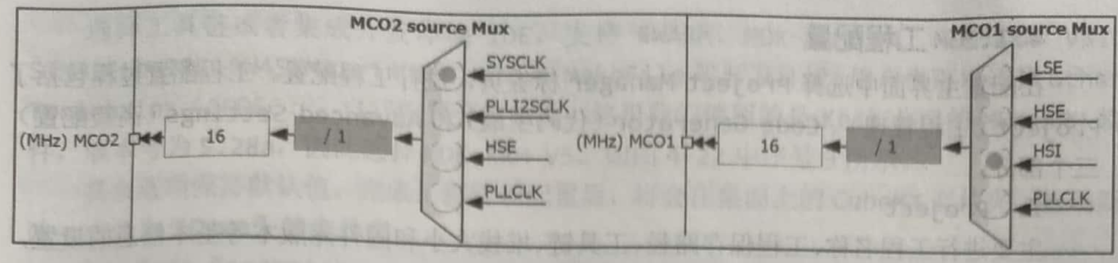


图 4-20　时钟输出模块

掌握了 STM32 的时钟结构后，我们可以开始时钟的配置。配置过程可以分为四步：

① 修改时钟频率

根据实际电路的情况，修改相关的时钟频率。以 Nucleo 开发板为例，我们选择 ST-Link 仿真器提供的 8MHz 时钟作为 HSE，因此在 HSE 左边的 Input Frequency 蓝色框中将默认值 25MHz 修改为 8MHz。如图 4-21 中序号 1 所示。

② 选择锁相环时钟源

由于初始的时钟频率较低，需要通过锁相环将时钟倍频至一个较高的频率。用户根据实际情况进行选择：使用内部的 RC 振荡器产生时钟，选择 HSI 作为锁相环的输入；使用外部震荡电路或直接接入外部时钟时，选择 HSE 作为锁相环的输入。

在第一步的设置中，我们选择 ST-Link 仿真器提供的 8MHz 时钟作为 HSE，因此在 PLL Source MUX（锁相环多路开关）中，选择 HSE 作为锁相环的输入，如图 4-21 中序号 2 所示.

③ 选择系统时钟源

使用锁相环之后，系统时钟有三个来源：HSI、HSE 和 PLLCLK（锁相环输出时钟）。由于锁相环输出时钟频率较高且更加稳定，因此在 System CLock MUX（系统时钟多路开关）中选择 PLLCLK 作为系统时钟源，如图 4-21 中序号 3 所示。此时如果相关的配置参数有误或者时钟频率超过最大范围，相应的方框将会显示为红色，提示用户配置出错。我们暂时不用理会这个错误提示。

④ 确定 HCLK 时钟频率

根据实际的需求确定 HCLK 的值，一般设置为最高工作频率。以 Nucleo 开发板为例，可以设置 HCLK 为 100MHz。在时钟树的 HCLK 框中填入数字 100，并按下回车键，软件将会自动给出正确的时钟配置方案，用户选择接受该方案即可，如图 4-21 中序号 4 所示。

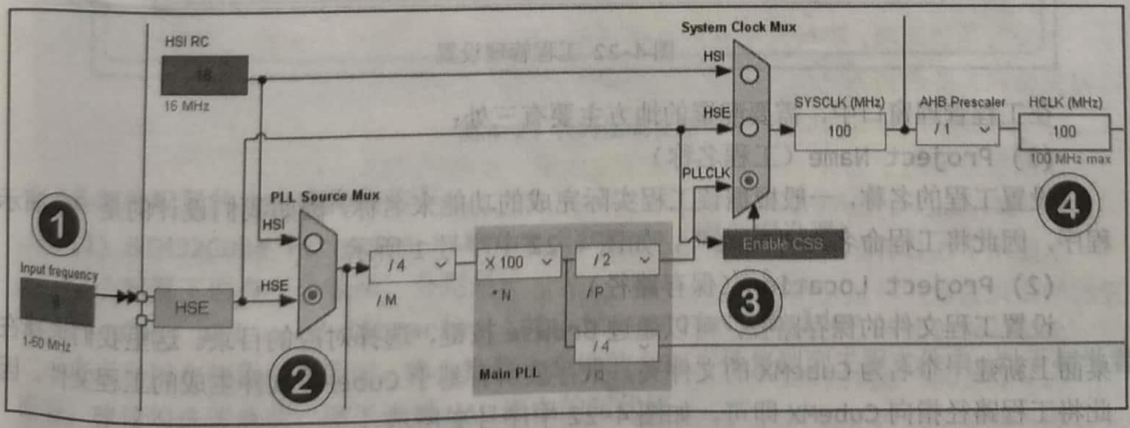


图 4-21　时钟配置步骤

4.1.5 工程配置

在配置主界面中选择 Project Manager 标签页，进行工程配置。工程配置过程包括 Project（工程管理）、Code Generator（代码生成）和 Advanced Settings（高级配置）三个部分：

1\. Project

主要进行工程名称、工程保存路径、工具链、堆栈大小和固件库版本等基本信息的设置，如图 4-22 所示。

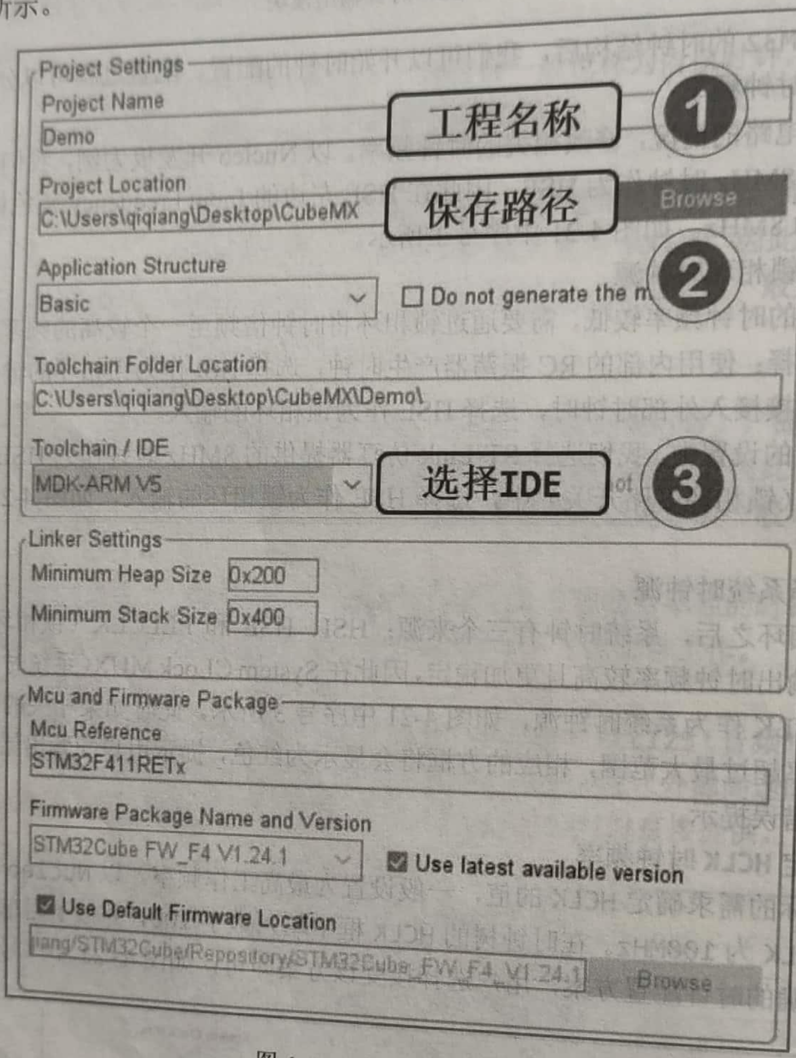


图 4-22 工程管理设置

在工程管理窗口中，需要配置的地方主要有三处：

(1) Project Name（工程名称）

设置工程的名称，一般根据该工程实际完成的功能来名称，例如我们设计的是一个演示程序，因此将工程命名为“Demo”，如图 4-22 中序号 1 所示。

(2) Project Location（保存路径）

设置工程文件的保存路径，可以通过 Browse 按键，选择对应的目录。这里我们选择在桌面上新建一个名为 CubeMX 的文件夹，来存放所有基于 CubeMX 软件生成的工程文件。因此将工程路径指向 CubeMX 即可，如图 4-22 中序号 2 所示。

(3) Toolchain / IDE（选择 IDE）

选择工具链或者集成开发环境 IDE，支持 EWARM、MDK-ARM V4、MDK-ARM V5、STM32CubeIDE、SW4STM32、TRUESTUDIO 和 MakeFile 等开发环境，用户也可以选择 Other Toolchains（GPDSC），自行配置开发软件。这里我们使用的是 Keil 公司的 MDK-ARM 软件，版本号为 5.28a，因此选择 MDK-ARM V5，如图 4-22 中序号 3 所示。

其余选项保持默认值。完成工程管理配置后，将会在桌面上的 CubeMX 文件夹下自动新建一个名为“Demo”的文件夹。

2. Code Generator

主要进行固件库设置、生成代码设置、HAL 库设置和模板设置四个部分，一般使用默认值即可，如图 4-23 所示。

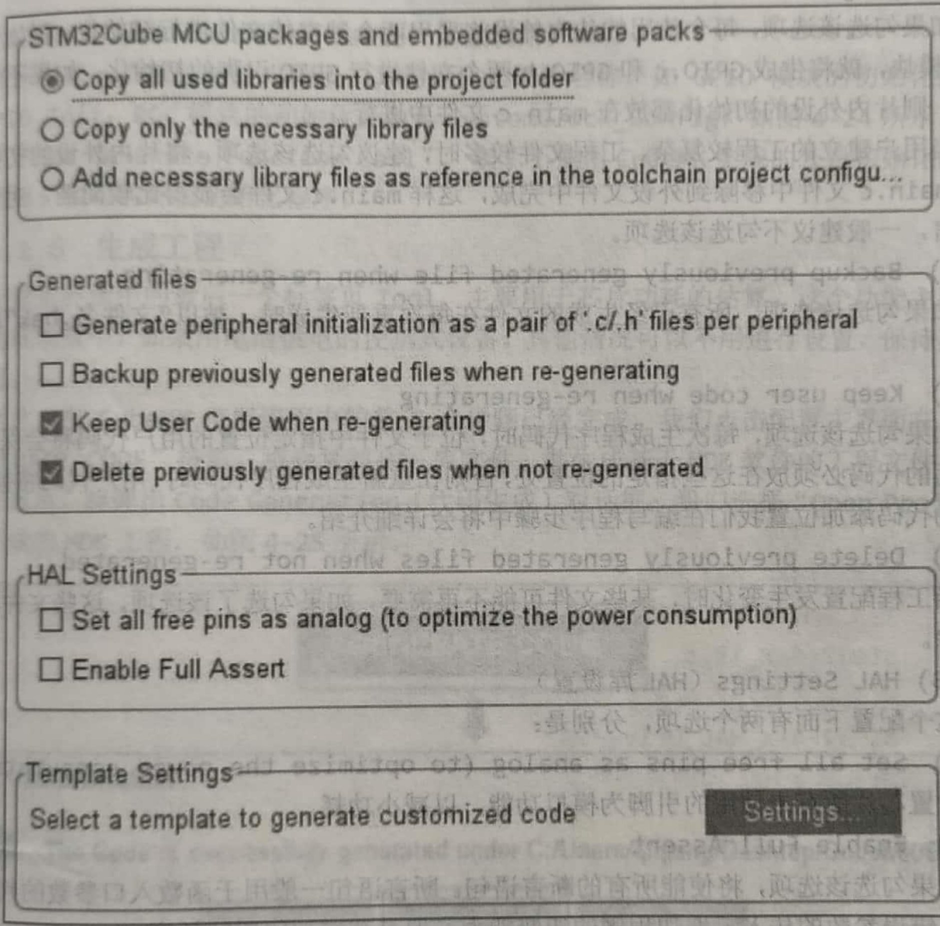


图 4-23 代码生成设置

各部分配置的含义和作用如下：

(1) STM32Cube Firmware Library Package（固件库设置）

这个配置下面有三个选项，分别是：

1) Copy all used libraries into the project folder

表示当用户使用固件库时，将会把整个库中的全部文件复制到工程文件中。对于初学者而言，建议勾选该选项，便于查找 HAL 库中的源代码。

2) Copy only the necessary library files

表示当用户使用固件库时，只复制固件库中相关联的文件到工程文件中。如我们使用

HAL 库去控制指示灯，只使用了时钟外设和 GPIO 外设，就只需要复制 HAL 库中的 RCC 模块和 GPIO 模块的文件，而不需要把整个 HAL 库都复制过去，可以减少工程的大小。

3) Add necessary library files as refence in the toolchain project configuration

表示当用户使用固件库时，相关文件不会复制到工程文件中，编译时由编译器提供固件库的引用地址。

(2) Generated files（生成文件设置）

这个配置下面有四个选项，分别是：

1) Generate peripheral initialization as a pair of '.c/.h' files per project

如果勾选该选项，每个使用的片内外设将采用两个独立的文件进行初始化。例如使用了 GPIO 模块，就将生成 GPIO.c 和 GPIO.h 两个文件进行 GPIO 引脚的初始化；如果不勾选该选项，则片内外设的初始化都放在 main.c 文件中进行。

当用户建立的工程较复杂，工程文件较多时，建议勾选该选项，将片内外设的初始化操作从 main.c 文件中移除到外设文件中完成，这样 main.c 文件会显得比较简洁。对于初学者而言，一般建议不勾选该选项。

2) Backup previously generated file when re-generating

如果勾选该选项，所有已经生成的文件在每次重新生成时，被以"文件名.bak"的形式备份。

3) Keep user code when re-generating

如果勾选该选项，每次生成程序代码时，位于文件中指定位置的用户代码将会保存。用户编写的代码必须放在这些指定的位置处，否则在重新生成程序代码时，用户代码将被清除。具体的代码添加位置我们在编写程序步骤中将会详细介绍。

4) Delete previously generated files when not re-generated

当工程配置发生变化时，某些文件可能不再需要，如果勾选了该选项，这些文件将会自动删除。

(3) HAL Settings（HAL 库设置）

这个配置下面有两个选项，分别是：

1) Set all free pins as analog (to optimize the power consumption)

设置芯片所有未使用的引脚为模拟功能，以减小功耗。

2) Enable Full Assert

如果勾选该选项，将使能所有的断言语句。断言语句一般用于函数入口参数的判断，可以避免错误参数的传入，增加程序的纠错能力，但是会增加代码的长度。

(4) Template Settings（模板设置）

该配置主要用于选择一个文件模板来生成用户定制的代码。

3. Advanced Settings

主要用于选择各个片内外设所用的固件库，如 HAL 固件库和 LL 固件库，并指明各个外设的初始化函数，一般使用默认值即可，如图 4-24 所示。

各部分配置的含义和作用如下：

(1) Driver Selector（驱动库选择）

对于使能的片内外设，可以在下拉菜单中选择使用 HAL 库还是 LL 库，如图 4-24 所示。HAL 库和 LL 库相比，执行效率稍低，但程序的可移植更好，一般使用默认的 HAL 库。

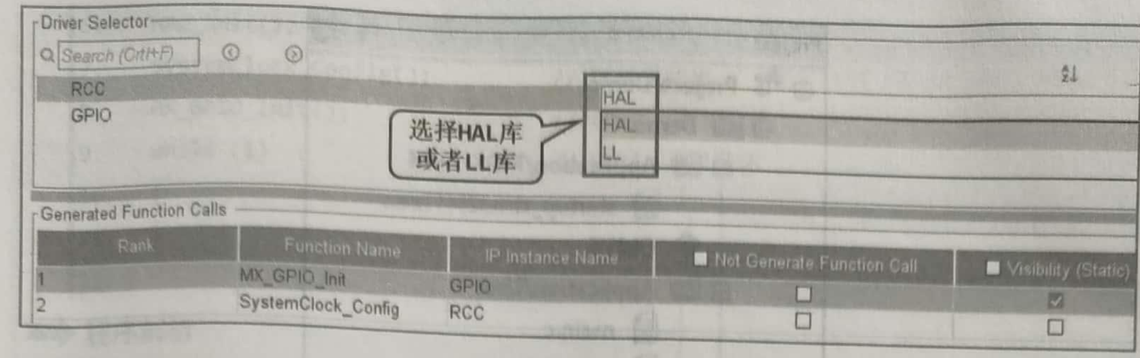


图 4-24 高级设置

(2) Generated Function Calls（函数调用）

显示了每一个使能的片内外设，其初始化函数的名称，如 GPIO 模块的初始化函数是 MX_GPIO_Init，RCC 模块的初始化函数是 SystemClock_Config，如图 4-24 所示。如果勾选了“Not Generate Function Call”，表示不使用 CubeMX 生成的外设初始化函数，而使用用户编写的外设初始化程序。

4.1.6 生成工程

配置主界面中的最后一个标签页 Tool，主要用于芯片功耗的估算。这个功能主要应用于低功耗场景中，如采用电池供电的便携式设备。其他情况可以不用进行设置，保持其默认值即可。

至此，在 CubeMX 使用流程中的前 5 个步骤已经完成，我们点击配置主界面右上角的“GENERATE CODE”按钮，将会复制 HAL 库文件，并生成基于 MDK 软件的工程文件。生成过程完成后，将弹出 Code Generation（代码生成）对话框，我们选择“Open Project”打开生成的 MDK 工程，如图 4-25 所示。

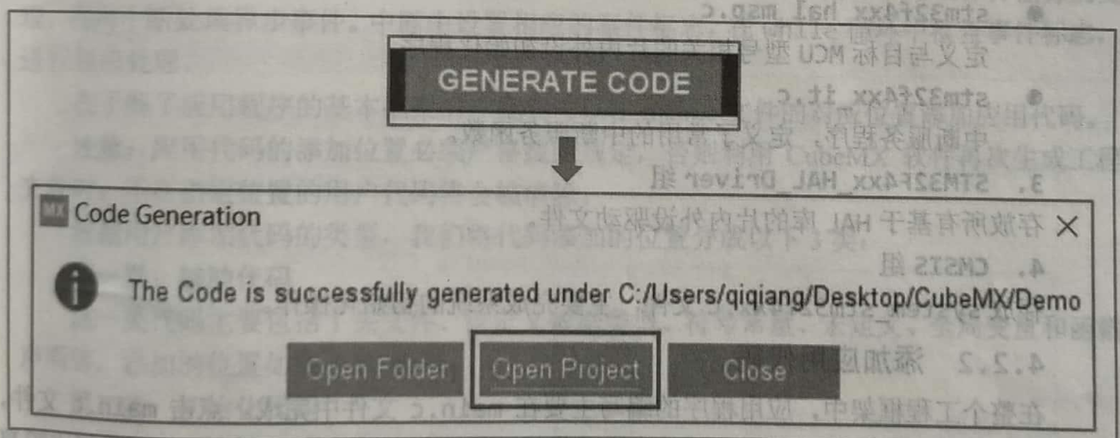


图 4-25 生成工程代码

4.2 在 MDK 中编写应用程序

4.2.1 工程框架介绍

完成在 CubeMX 中的设置后，我们开始在 MDK 软件中编写应用程序。点击工程窗口中工程名 Demo 前面的“+”号，展开整个工程。如图 4-26 所示。

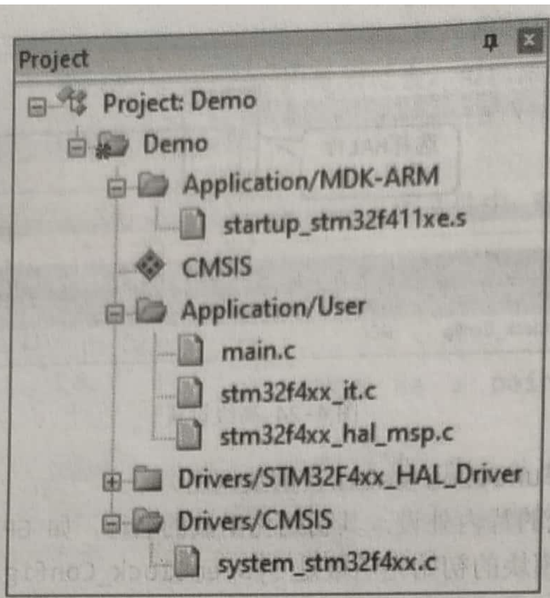


图 4-26　MDK 工程框架

工程名称为 Demo，下面分别建立了四个 Group（组），分别是：

1.　MDK-ARM 组

存放 startup_stm32f411xe.s 文件。该文件作为系统启动代码，主要功能是配置系统堆和栈的大小、初始化中断向量表。

2.　User 组

存放用户源文件，主要包含以下 3 个文件：

- **main.c**
  用户主程序，是编写应用程序的主要文件。
- **stm32f4xx_hal_msp.c**
  定义与目标 MCU 型号相关的片内外设初始化程序。
- **stm32f4xx_it.c**
  中断服务程序，定义了常用的中断服务函数。

3.　STM32F4xx_HAL_Driver 组

存放所有基于 HAL 库的片内外设驱动文件。

4.　CMSIS 组

存放 system_stm32f4xx.c 文件，主要完成系统的初始化操作。

4.2.2　添加应用代码

在整个工程框架中，应用程序的编写主要在 main.c 文件中完成。点击 main.c 文件，在代码编辑窗口中将显示 main.c 文件的源代码。整个代码包含了大量的注释，显得比较复杂。我们删除其中的注释语句以及内部函数后，程序代码如程序清单 4-1 所示。

程序清单 4-1　删除注释和内部函数后的 main.c 文件源代码

```
#include "main.h"                    // 包含头文件
void SystemClock_Config(void);       // 时钟初始化函数声明
void MX_GPIO_Init(void);             // GPIO 引脚初始化函数声明
int main(void)
{
```

```
6.    HAL_Init();                        // HAL 库初始化
7.    SystemClock_Config();              // 时钟初始化
8.    MX_GPIO_Init();                    // GPIO 引脚初始化
9.    while (1)                          // while 无限循环
10.   {
11.   }
12. }
```

程序解析

① 第 1 行是包含 main.h 文件。在 main.h 文件包含了 stm32f4xx_hal.h，这是一个所有用户文件都必须包含的文件，里面包含了 HAL 库的数据类型定义以及各类接口函数的声明。只有包含了 stm32f4xx_hal.h 文件，才能使用 HAL 库提供的接口函数。

② 第 2～3 行是内部函数的声明。由于函数 SystemClock_Config 和 MX_GPIO_Init 放在 main 函数之后定义，因此需要在 main 函数之前加上函数声明。

③ 第 6 行是调用函数 HAL_Init 完成 HAL 库的初始化。在使用 HAL 库之前，必须调用该函数，它的主要功能是：复位所有的片内外设，初始化片内 Flash 存储器接口以及系统节拍定时器 Systick。Systick 用于提供延时函数 HAL_Delay 所需的定时时间，默认的定时时间是 1ms。

④ 第 7 行调用系统时钟初始化函数 SystemClock_Config 完成系统时钟的设置。时钟的配置参数是依据用户在 CubeMX 中所做的设置（时钟设置部分）。

⑤ 第 8 行调用 GPIO 引脚初始化函数 MX_GPIO_Init 完成引脚的设置。引脚的配置参数也是依据用户在 CubeMX 中所做的设置（引脚分配和外设设置部分）。

⑥ 第 9～11 行是由 while 构成的无限循环，表明程序架构采用前后台模式。该程序架构由 CubeMX 软件生成。主程序完成系统初始化后，就在 while 循环中完成各种任务的处理，利用中断处理异步事件。中断中设置相应的事件标志，在 while 循环中检查事件标志，进行相应处理。

在了解了应用程序的基本框架后，我们可以在 main.c 文件的对应位置添加应用代码。

注意：应用代码的添加位置必须严格按照规定，否则利用 CubeMX 软件再次生成工程文件时，不在指定位置的用户代码将会被清除。

根据用户添加代码的类型，我们将代码添加的位置分成以下 3 类：

第一类：辅助代码

这一类代码主要包括了头文件、自定义数据类型、符号常量、宏定义、全局变量和函数声明等。添加的位置如图 4-27 所示。

第二类：初始化代码

这一类代码主要是用户自己编写的初始化程序，包括变量的初始化，外围电路的初始化等（片内外设的初始化一般由 CubeMX 自动生成）。添加位置如图 4-28 所示。

第三类：应用代码

这一类代码主要是执行实际操作的代码，根据项目的实际情况由用户编写。添加位置如图 4-29 所示。

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */

/* USER CODE END Includes */

/* Private typedef -----------------------------------------------------------*/
/* USER CODE BEGIN PTD */

/* USER CODE END PTD */

/* Private define ------------------------------------------------------------*/
/* USER CODE BEGIN PD */

/* USER CODE END PD */

/* Private macro -------------------------------------------------------------*/
/* USER CODE BEGIN PM */

/* USER CODE END PM */

/* Private variables ---------------------------------------------------------*/

/* USER CODE BEGIN PV */

/* USER CODE END PV */

/* Private function prototypes -----------------------------------------------*/

/* USER CODE BEGIN PFP */

/* USER CODE END PFP */
```

图 4-27 辅助代码添加位置

```
/* USER CODE BEGIN Init */
/* USER CODE END Init */

/* USER CODE BEGIN SysInit */
/* USER CODE END SysInit */
```

图 4-28 用户初始化代码添加位置

```
/* USER CODE BEGIN 0 */

/* USER CODE END 0 */

/* USER CODE BEGIN 1 */

/* USER CODE END 1 */

/* USER CODE BEGIN 2 */

/* USER CODE END 2 */

/* USER CODE BEGIN 3 */

/* USER CODE END 3 */
```

图 4-29 用户代码添加位置

要驱动 Nulceo 上的指示灯 LD2，需要引脚 PA5 间隔输出高电平和低电平：输出高电平时，指示灯开启；输出低电平时，指示灯关闭。这个功能可以利用 HAL 库中 GPIO 模块提供的引脚翻转函数 HAL_GPIO_TogglePin 实现。函数的功能描述、入口参数和返回值如表 4-1 所示。

表 4-1　引脚翻转函数

接口函数：HAL_GPIO_TogglePin	
函数原型	void HAL_GPIO_TogglePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
功能描述	翻转引脚状态
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA～GPIOK
入口参数 2	GPIO_Pin：引脚号，范围是 GPIO_PIN_0～GPIO_PIN_15
返回值	无
注意事项	无

函数 HAL_GPIO_TogglePin 实现了指示灯的开启和关闭。如果直接利用该函数驱动指示灯，由于 STM32 微控制器的引脚翻转速度太快，最高可达 50MHz，人眼无法分辨出指示灯的状态变化。因此我们需要在引脚翻转函数后面再加上一个延时函数，延时时间为 1s，这样就可以看到指示灯的状态变化了。

HAL 库中专门为用户提供了一个延时函数 HAL_Delay。该函数利用 SysTick 定时器实现以 ms 为基准的延时。函数的功能描述、入口参数和返回值如表 4-2 所示。

表 4-2　延时函数

接口函数：HAL_Delay	
函数原型	void HAL_Delay(uint32_t Delay)
功能描述	提供以 ms 为单位的延时服务
入口参数	Delay：需要的延时时间，以 ms 为单位
返回值	无
注意事项	利用系统节拍定时器 SysTick 实现，将占用该定时器资源

我们将应用代码添加在 while 循环内部的“USER CODE BEGIN 3”和“USER CODE END 3”，让切换指示灯 LD2 状态的语句循环执行，从而形成闪烁的效果。具体的代码如程序清单 4-2 所示。

程序清单 4-2　添加闪烁指示灯 LD2 的应用代码

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  // 翻转 PA5 引脚电平，控制指示灯 LD2 的开启和关闭
  HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
  HAL_Delay(1000);                              // 延时 1s
}
  /* USER CODE END 3 */
```

程序解析

应用代码非常简单，只有 2 行代码，添加的位置在 while 循环内部的“USER CODE BEGIN 3”和“USER CODE END 3”。由于控制 LD2 的引脚是 PA5，该引脚对应的端口是 GPIOA，对应的引脚是 GPIO_PIN_5，因此翻转引脚函数 HAL_GPIO_TogglePin 的 2 个入口参数分别为 GPIOA 和 GPIO_PIN_5。然后调用延时函数 HAL_Delay，入口参数为 1000，形成 1s 的延时。

4.2.3 编译程序及下载

完成应用程序的编写后，利用 MDK 主界面工具栏中的 3 个图标来完成程序的编译和下载，如图 4-30 所示。

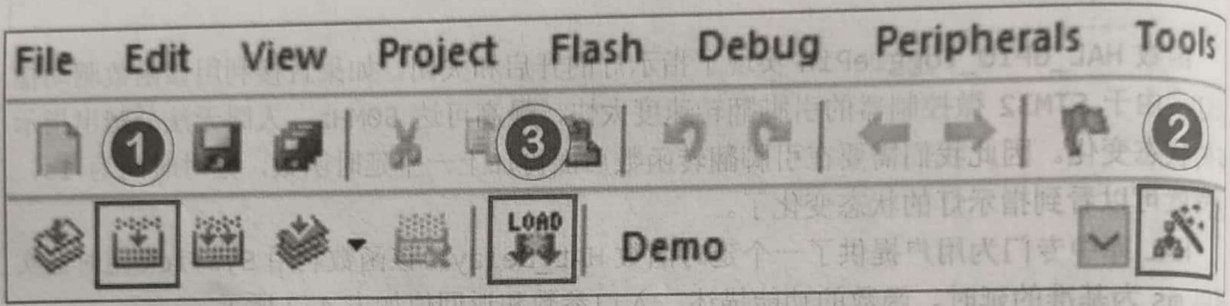


图 4-30 工具栏中用于程序编译及下载的图标

具体操作步骤如下：

1. 程序的编译和链接

点击“Build”图标，如图 4-30 中序号 1 所示，进行程序的编译和链接。如果程序编译过程中出现了错误，需要根据 MDK 主界面最下方的信息输出窗口中的错误提示修改代码，直至没有错误和警告为止。

2. 程序下载设置

点击“Options for Target”图标，如图 4-30 中序号 2 所示，进行程序下载的设置。在工程设置窗口中选择“Debug”标签页，选择下载工具为“ST-Link Debugger”。然后点击仿真器设置按钮“Settings”，在出现仿真器设置窗口中选择“Flash Download”标签页，勾选“Reset and Run”，以便让程序下载后能自动运行。如图 4-31 所示。

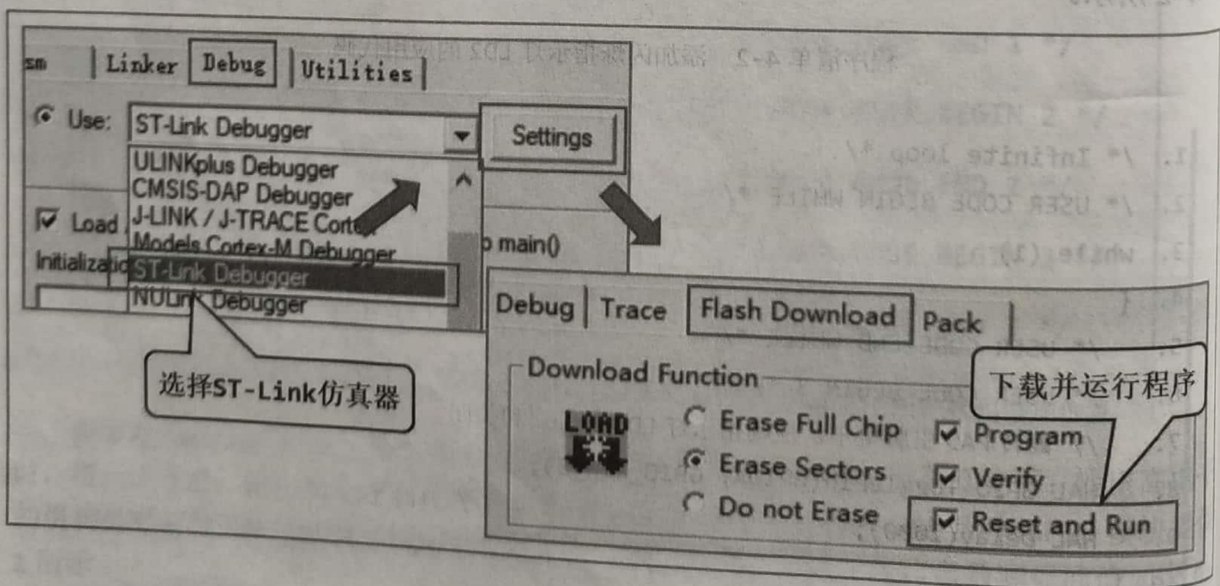


图 4-31 程序下载设置

注意：如果仿真器的复位脚没有和目标芯片的复位引脚连接，则无法设置程序“Reset and Run”，因此最稳妥的方法还是按下复位键，让程序重新执行。

3. 下载程序

点击“Download”图标，如图 4-30 中序号 3 所示，进行程序的下载。此时，Nucleo 开发板上的绿色指示灯 LD2 将会每隔 1s 闪烁一次。读者也可以修改延时函数 HAL_Delay 的延时参数，比如修改为 100，重新编译程序后下载，观察指示灯的闪烁情况。

4.3 本章小结

本章重点介绍了 CubeMX 软件的使用方法，内容以实际操作为主，需要读者多多练习，才能熟练掌握 CubeMX 软件的使用方法。建议读者尝试在工程配置中选择不同的配置选项来生成工程，对比一下生成后工程之间的差异，以便更好的理解这些配置选项的作用。

习题 4

1. 在工程设置中，尝试选择“Generate peripheral initialization as a pair of‘.c/.h’ files project”，看看生成的工程有什么变化。

2. 阅读程序源码，分析 HAL 库初始化函数 HAL_Init 的具体执行流程以及完成了哪些操作。

3. 阅读 STM32F411 参考手册，了解系统节拍定时器 SysTick 的工作原理及配置方法，结合 HAL 库中的 SysTick 初始化程序分析延时函数 HAL_Delay 是如何实现的。

4. 查阅相关资料，尝试使用 STM32CubeIDE 软件进行 STM32 的程序开发：编写一个控制指示灯闪烁的程序。

第五章 MDK-ARM 开发软件的使用

本章导读

本章重点介绍 MDK 软件的使用方法，主要包括工程设置、程序调试，以及软件的实用功能。同时介绍了 MDK 软件的 RTE 工具与 STM32CubeMX 软件联合使用的方法。

本章知识点

MDK 的工程设置

MDK 的程序调试

MDK 的实用功能

MDK 的 RTE 工具与 STM32CubeMX 的联合使用

5.1 MDK 的使用

在第四章中，我们已经介绍了利用 CubeMX 软件经过目标选择、引脚分配、外设设置、时钟设置以及工程配置等 5 个步骤后，直接生成 MDK 工程文件的过程。本章将在该工程文件的基础上详细介绍 MDK 软件的用法。

MDK 软件的使用分为 4 个步骤，如图 5-1 所示。

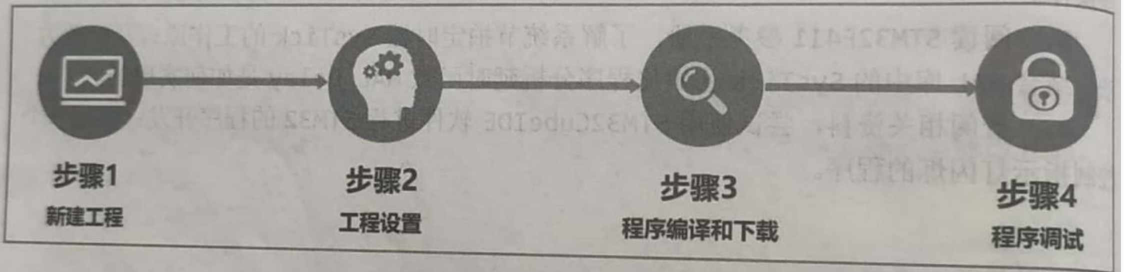


图 5-1 MDK 软件使用步骤

5.1.1 新建工程

由于在 CubeMX 软件的工程设置中，已经设置了工程名称和保存路径，如图 4-22 所示。软件将自动生成 MDK 的工程文件，用户不需要新建工程。

在 CubeMX 的文件夹下，自动生成了名为“Demo”的文件夹。在 Demo 文件夹下，生成了如图 5-2 所示的相关文件和文件夹。

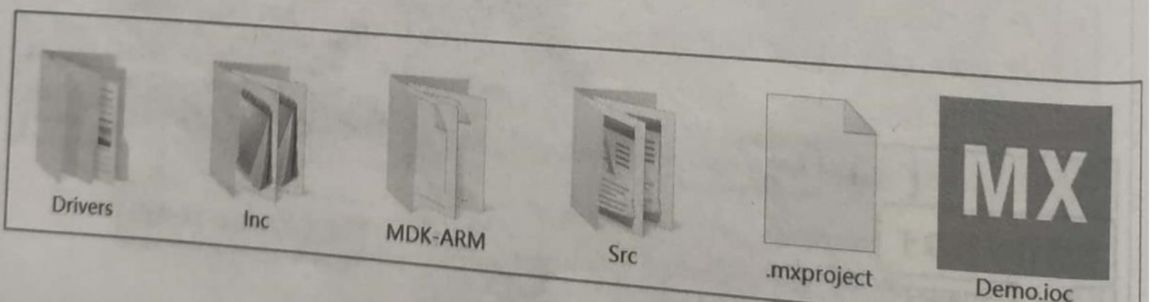


图 5-2 CubeMX 生成的工程文件夹内容

文件夹中后缀名为“.ioc”的文件“Demo.ioc”是由 CubeMX 软件生成的 CubeMX 工程文件。如果用户需要修改之前的设置，如引脚分配、外设设置等，就可以直接点击 Demo.ioc

文件重新进入到 CubeMX 软件，完成相关设置。

工程文件中还生成了 4 个文件夹，分别是 Drivers、Inc、MDK-ARM 和 Src。

1. Drivers 文件夹

用于存放和驱动相关的文件，里面包含两个子目录：

- STM32F4xx_HAL_Driver

ST 官方提供的 HAL（硬件抽象层）固件库。该文件夹下面的 Src 文件夹存放的是 STM32 微控制器片内外设的 API 函数（接口函数）。Inc 文件夹存放的是对应的头文件，头文件的内容主要是数据类型定义、宏定义和 API 函数的声明。

- CMSIS

Cortex 内核微控制器标准软件接口。该文件夹的内容由 ARM 公司和芯片生产商联合提供，主要存放 Cortex-M 内核文件、芯片头文件以及特定芯片的启动代码和系统初始化文件。

2. Src 文件夹

用于存放用户源文件，包含以下 4 个文件：

- main.c：
  用户主程序。
- stm32f4xx_hal_msp.c
  定义所有和特定 MCU 型号相关的片内外设初始化程序。
- stm32f4xx_it.c
  中断服务程序，定义了常用的中断服务函数。
- system_stm32f4xx.c
  系统初始化代码，主要完成系统初始化操作。

3. Inc 文件夹

用于存放用户头文件，包含以下 3 个文件

- main.h
  用户主程序的头文件.
- stm32f4xx_hal_conf.h
  HAL 库配置头文件。该文件是一个用户级别的配置文件，用来实现对 HAL 库的裁剪。可以进行参数的修改，如外部晶振的大小；也可以根据项目需求使能相应的片内外设。该文件一般放置在用户文件目录，不要放在 HAL 库目录中。
- stm32f4xx_it.h
  中断服务程序的头文件。

4. MDK-ARM 文件夹

用于存放基于 MDK 开发环境的工程文件，里面比较重要的是 2 个文件：

- Demo.uvprojx
  基于 MDK 开发环境的工程文件；
- startup_stm32f411xe.s
  系统启动代码，主要功能是配置系统堆和栈的大小、初始化中断向量表。

点击 MDK-ARM 文件夹下面的 Demo.uvprojx 文件，就可以启动 MDK 软件，并打开对应的工程文件。MDK 软件的主界面可以分成五个部分，如图 5-3 所示。

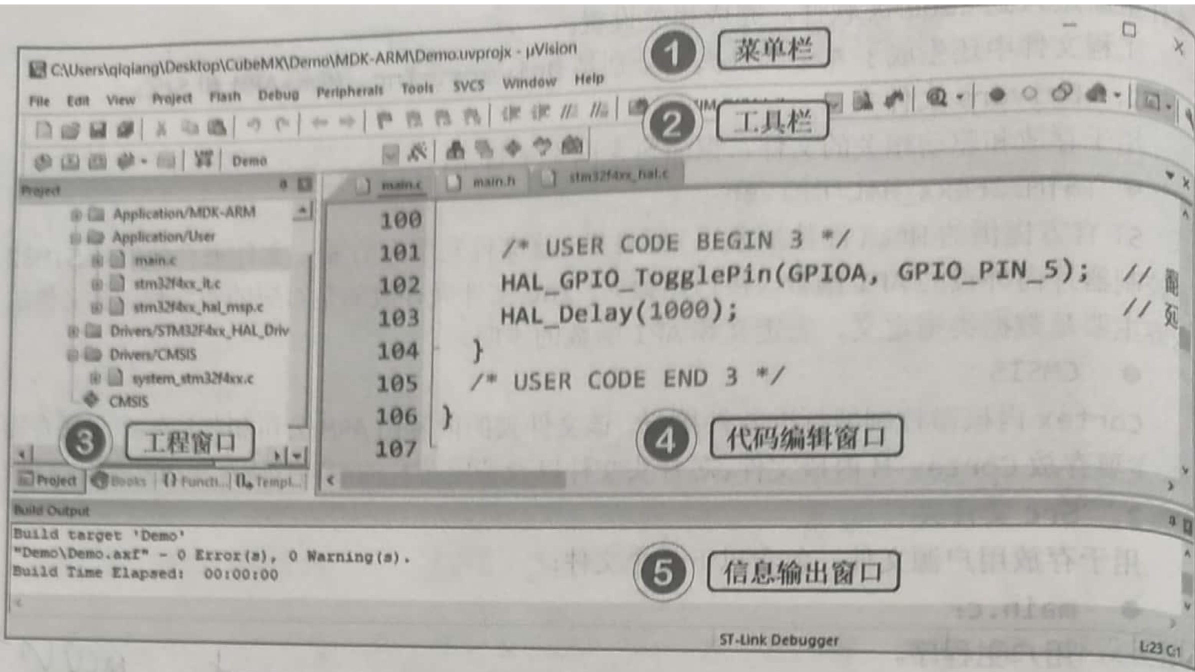


图 5-3 MDK 软件的主界面

1. 菜单栏

主界面最上方的一行是菜单栏，提供 MDK 软件的全部功能菜单，如新建文件、新建工程，如图 5-3 中序号 1 所示。

2. 工具栏

主界面的第 2 行和第 3 行是工具栏，提供 MDK 软件的常用快捷功能，包括源文件的编辑，工程的设置，程序的调试，如图 5-3 中序号 2 所示。

3. 工程窗口（Project）

主界面左边的窗口是工程窗口，以树形结构展示工程中包含的所有文件，如图 5-3 中序号 3 所示。

4. 代码编辑窗口

主界面右边的窗口是代码编辑窗口，用于源代码的编辑，如图 5-3 中序号 4 所示。

5. 信息输出窗口（Build Output）

主界面最下方的窗口是信息输出窗口，用于显示工程在编译、链接过程中出现的相关信息，并显示编译完成后代码的大小，如图 5-3 中序号 5 所示。

在工程窗口中，由 CubeMX 自动生成的工程框架如图 5-4 所示。工程的名称为 Demo，在工程内又分别建立了四个 Group（组），分别是：

- MDK-ARM 组

  存放 startup_stm32f407xx.s 文件
- STM32F4xx_HAL_Driver 组

  存放所有基于 HAL 库的片内外设驱动文件
- CMSIS 组

  存放 system_stm32f4xx.c 文件
- User 组

  存放用户文件，包括了 main.c，stm32f4xx_hal_msp.c 和 stm32f4xx_it.c。

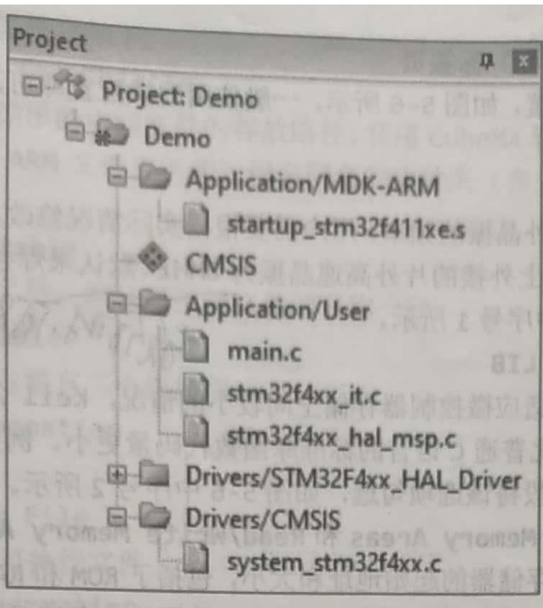


图 5-4　工程项目框架

由于 CubeMX 已经自动生成了基于 MDK 开发环境的工程，因此用户不用再新建工程。

5.1.2　工程配置

完成工程的建立后，我们需要配置工程的编译环境。在 MDK 软件主界面的工具栏中点击 按钮，进入工程配置界面（Options For Target）。在工程配置界面中，一共有十个标签页，下面我们对每个标签页做详细介绍。

1. Device（器件）标签页

用于选择微控制器的型号。一般情况下，在新建工程时已经选择好了芯片型号。如果用户想更改芯片型号，可以在 Device 标签页中进行修改，如图 5-5 中序号 1 所示。选中某一个芯片型号后，将显示该芯片的基本介绍，如图 5-5 中序号 2 所示。

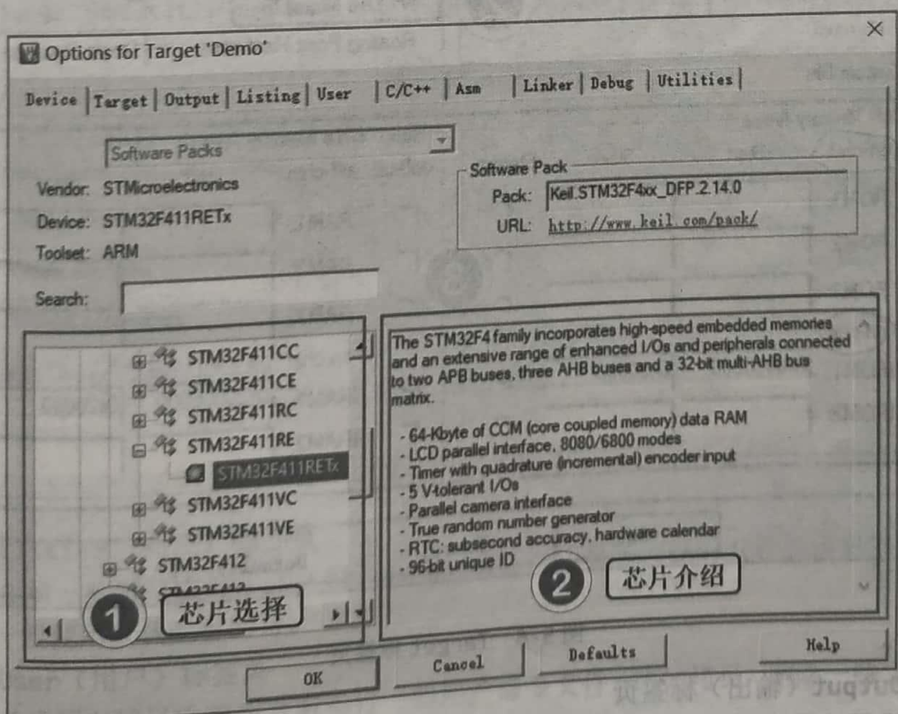


图 5-5　Device 标签页

2. Target（目标）标签页

用于硬件目标配置，如图 5-6 所示，一般使用默认配置即可，这里将几个重要的配置内容介绍一下：

- **Xtal（MHz）**

用于设置芯片片外晶振的频率，用户需要根据实际情况修改，该值主要用于程序仿真时使用。Nucleo 开发板上外接的片外高速晶振为 8MHz（默认未焊接），因此修改 Xtal（MHz）为 8MHz，如图 5-6 中序号 1 所示。

仿真时，用8MHz～

- **Use Micro LIB**

使用微库。为了适应微控制器存储空间较小的情况，Keil 公司专门开发了一套小型的 C 语言标准库函数，比普通 C 语言的标准库函数代码量更小。例如，我们使用 printf 函数时就会调用微库。一般将该选项勾选，如图 5-6 中序号 2 所示。

- **Read/Only Memory Areas 和 Read/Write Memory Areas**

设置片外、片内存储器的起始地址和大小，包括了 ROM 和 RAM。对于 Cortex-M 内核微控制器而言，一般不用进行片外存储器的扩展，使用片内存储器即可。这部分的配置值会自动根据芯片型号来分配地址，一般使用默认值即可，如图 5-6 中序号 3 所示。对于 STM32F411RET，片内 Flash 的起始地址为 0x80000000，大小为 512KB。片内 SRAM 的起始地址为 0x20000000，大小为 128KB。

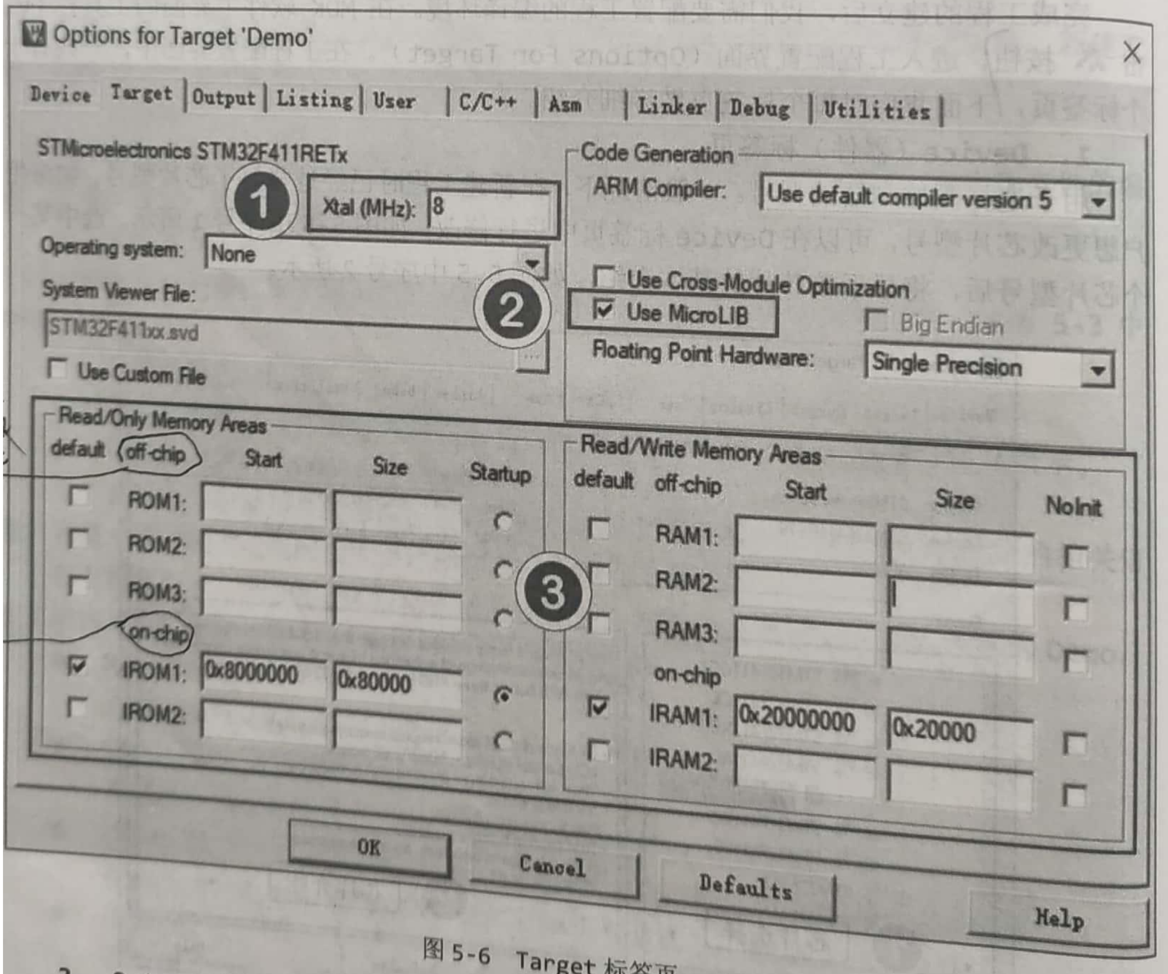


图 5-6　Target 标签页

3. Output（输出）标签页

用于设置编译过程中输出的中间文件的存放路径，如图 5-7 所示，一般采用默认配置

即可，其中几个重要的设置选项如下：

- **Select Folder for Objects**

用于选择编译过程中输出的中间文件的存放路径。利用 CubeMX 软件生成的 MDK 工程将把该路径默认设置为 MDK-ARM 文件夹下和工程名同名的文件夹（参见图 5-2），如本例中的 Demo。

- **Name of Executable**

生成的可执行文件的名称，默认和工程文件的名称一致。

- **Create Executable**

该选项默认选中。它内部有三个选择项：

■ Debug Information

生成调试信息

■ Create Hex File

生成 Hex 格式的可执行文件

■ Browse Information

生成浏览信息，如参看程序中定义的变量和函数等

默认选中了 Debug Information 和 Browse Information 两个选项来生成详细的调试信息和浏览信息。其中的 Create Hex File 选中后才能生成 Hex 格式的编程文件，用于下载到芯片运行。在使用第三方下载工具如 STM32CubeProgrammer 时，就需要 Hex 格式或者 Bin 格式的编程文件。

- **Create Executable Lib**

生成 lib 库文件。用户可以根据自己的需要选择是否生成库文件，一般的应用是不生成库文件的。如果希望隐藏自己的源代码，则可以生成库文件。

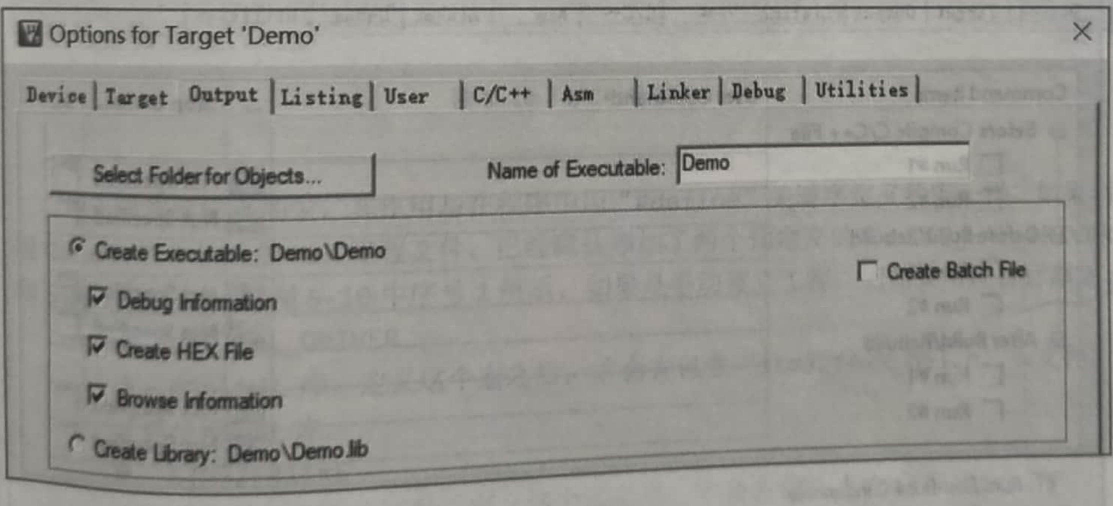


图 5-7 Output 标签页

4. Listing（列表）标签页

用于生成一些链接文件，比如编译后代码的详细信息和调试时查看的信息等，如图 5-8 所示，一般使用默认配置即可。

5. User（用户）标签页

主要是设置在编译之前或之后运行一些用户命令文件，比如拷贝，删除一些文件等批处理操作，如图 5-9 所示，一般使用默认配置即可。

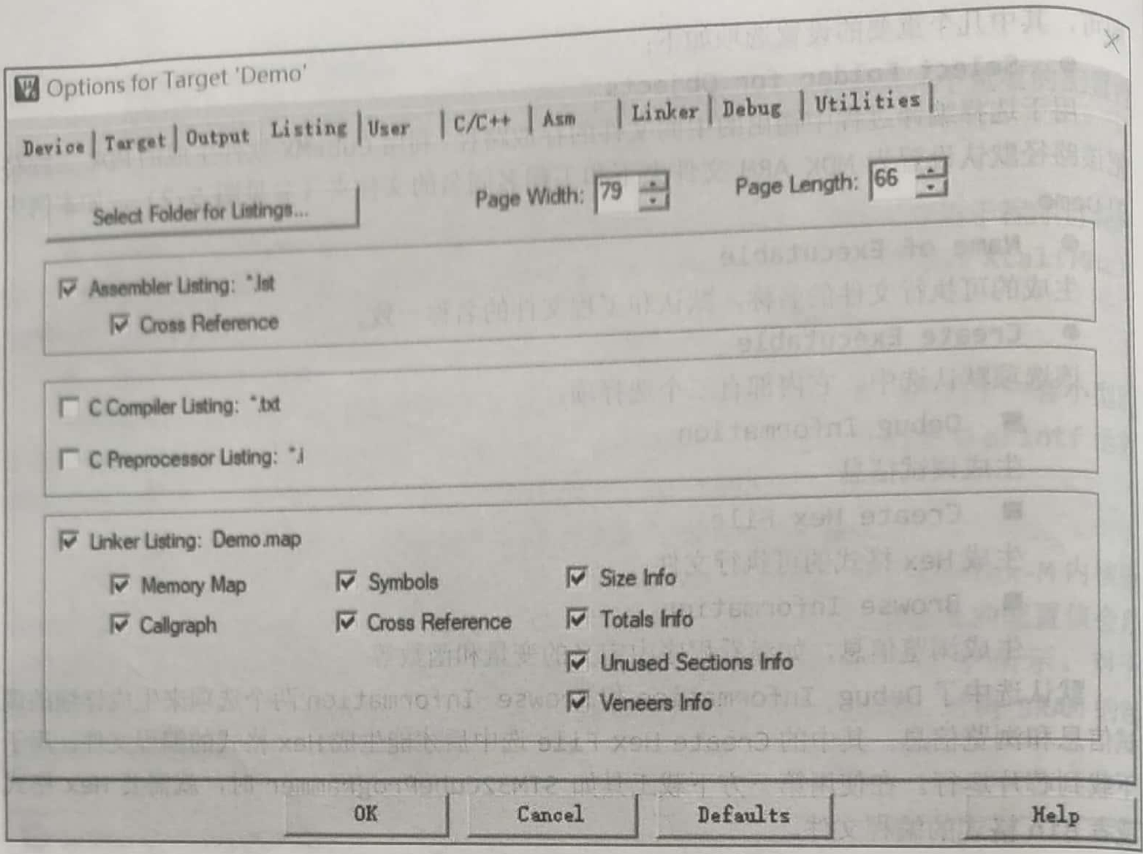


图 5-8　Listing 标签页

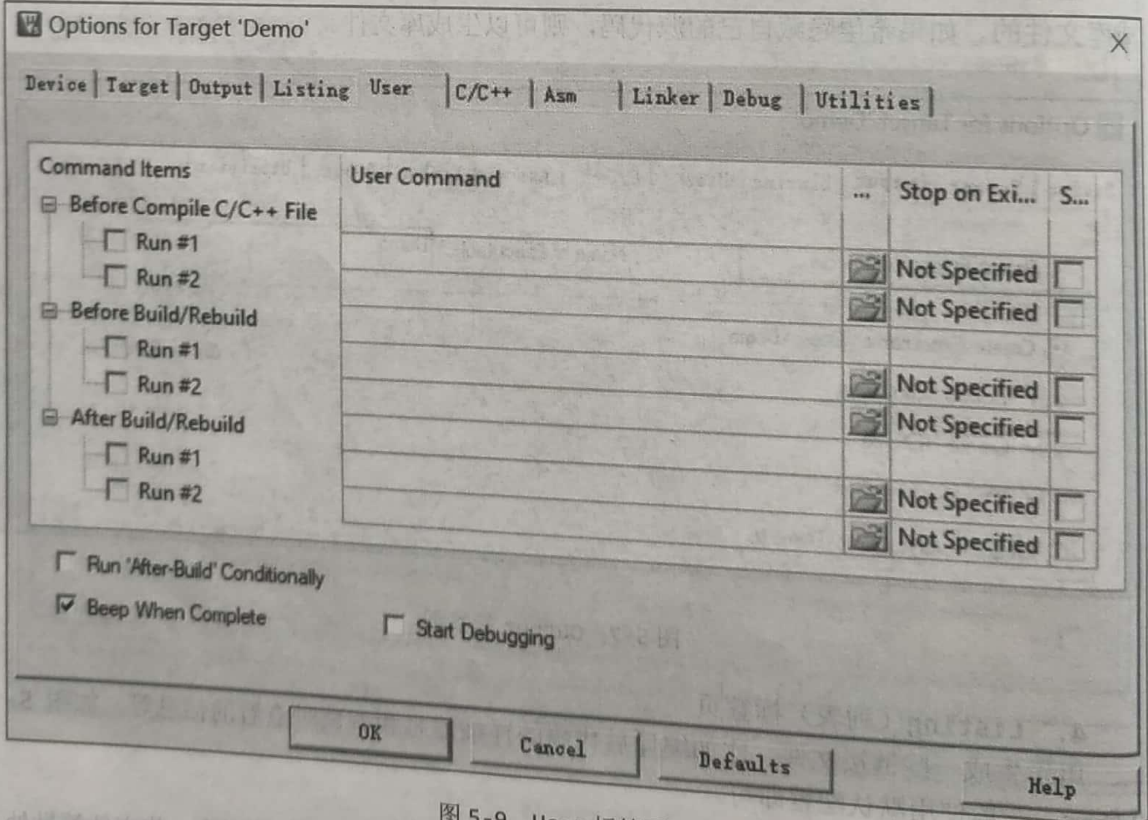


图 5-9　User 标签页

6. C/C++标签页

用于在 C 或者 C++程序中设置预定义的宏、程序优化等级以及头文件的包含路径等。如图 5-10 所示。其中几个重要的设置选项如下：

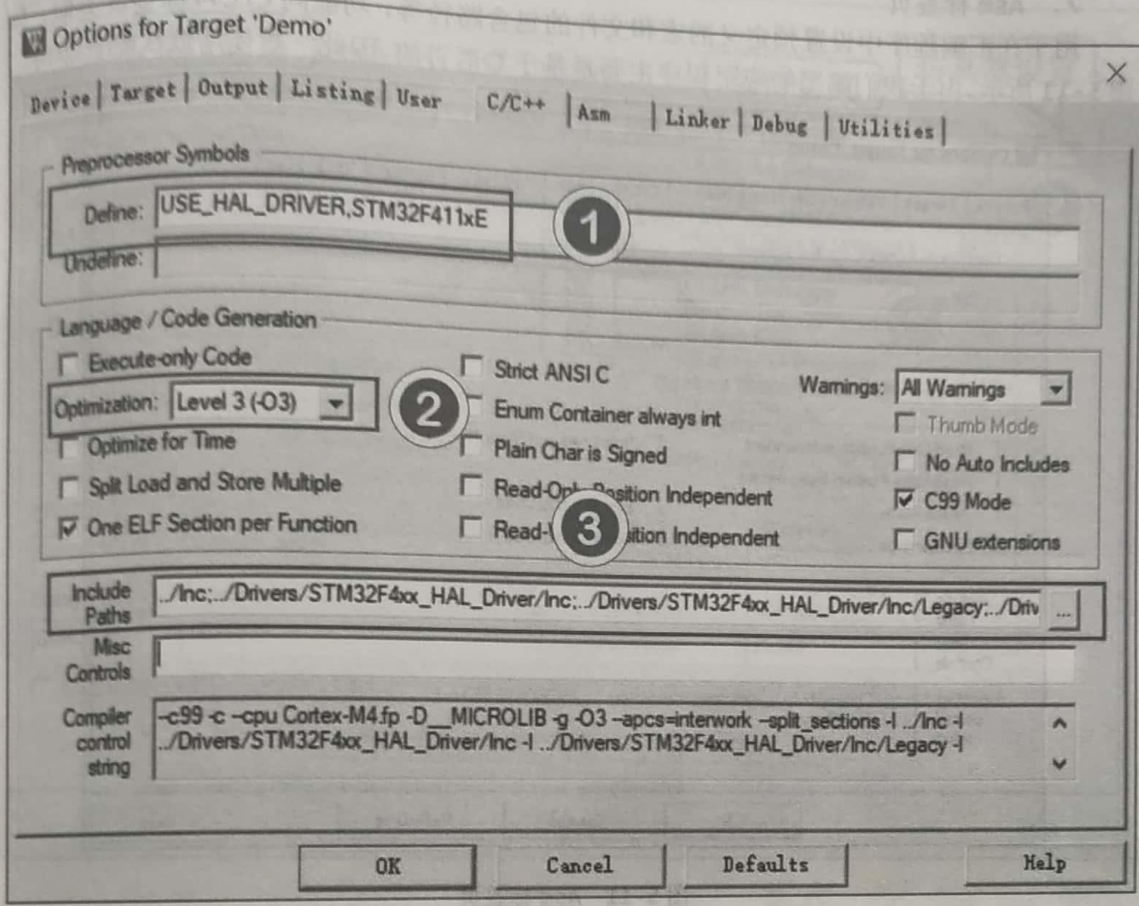


图 5-10 C/C++标签页

● Define

用于设置预定义的宏，其作用与在程序中用“#define”关键字定义的宏一样。如果利用 CubeMX 软件生成的 MDK 工程文件，已经默认添加了两个预定义的宏：USE_HAL_DRIVER 和 STM32F411xE，如图 5-10 中序号 1 所示。如果是手动建立工程，则需要用户自行添加。

■ USE_HAL_DRIVER

表示使用 HAL 库，定义这个宏之后，才会去包含“stm32f4xx_hal.h”头文件，才能使用 HAL 库。

■ STM32F411xE

表示具体使用的芯片型号，定义这个宏之后，才会去包含“stm32f411xe.h”，才能使用芯片片内硬件寄存器的定义。

● Optimization

用于设置程序的优化等级，默认是等级最低的 Level 3 等级，如图 5-10 中序号 2 所示。用户可以根据自己的实际情况进行修改。

● Include Paths

头文件的包含路径，如图 5-10 中序号 3 所示。如果利用 CubeMX 软件生成的 MDK 工程文件，会自动添加所需头文件的路径。如果是直接利用 MDK 软件建立工程或者用户在当前

工程的基础上新建组，则需要用户自行添加对应的头文件路径。如果没有正确设置头文件的路径，在编译时将会报错：无法找到该文件。

7. Asm 标签页

用于在汇编程序中设置预定义的宏和文件的包含路径等，功能同 C/C++标签页类似，如图 5-11 所示。由于我们编写的应用程序主要是基于 C 语言的，因此一般使用默认配置即可。

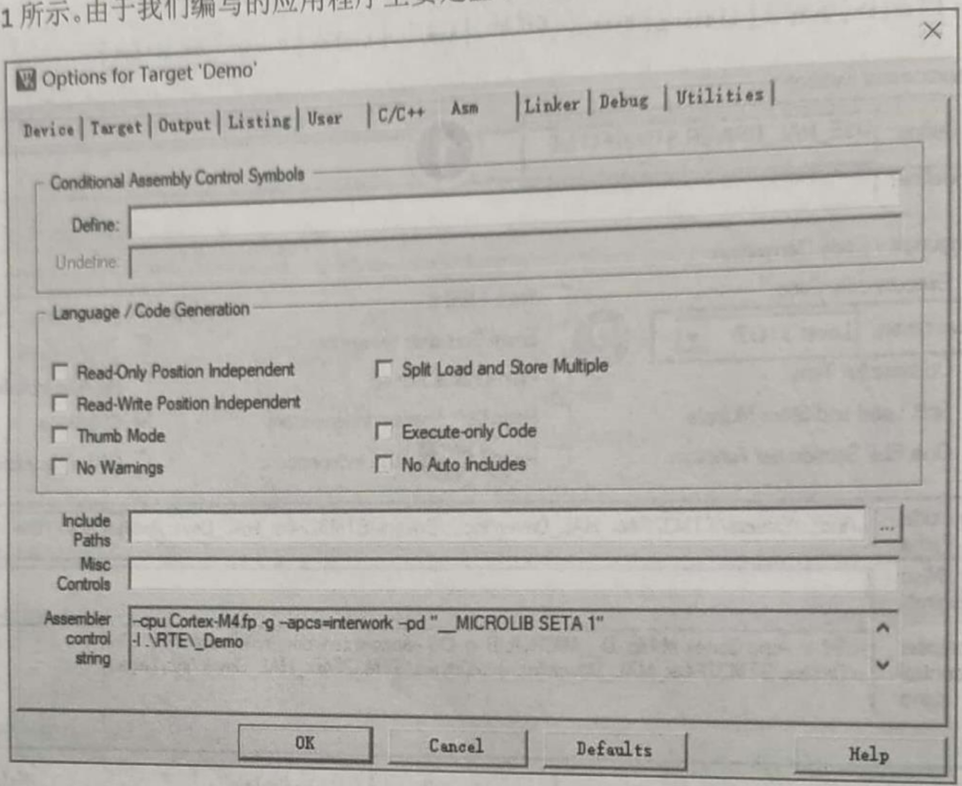


图 5-11　Asm 标签页

8. Linker（链接器）标签页

用于设置程序的链接地址和分散加载文件，如图 5-12 所示。一般使用默认配置即可。

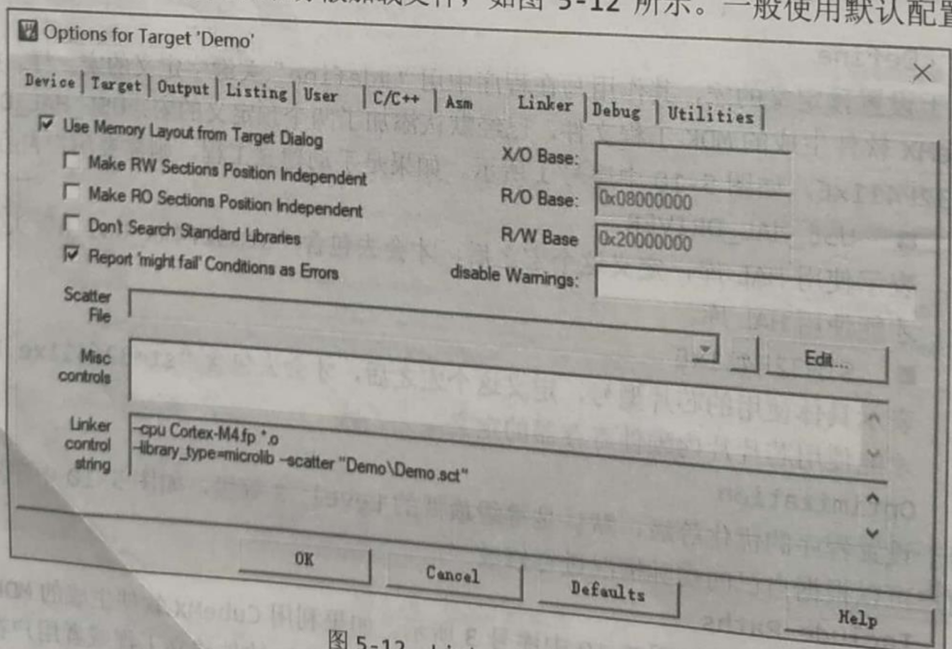


图 5-12　Linker 标签页

9. Debug（调试）标签页

用于程序仿真方式的设置，如图 5-13 所示。

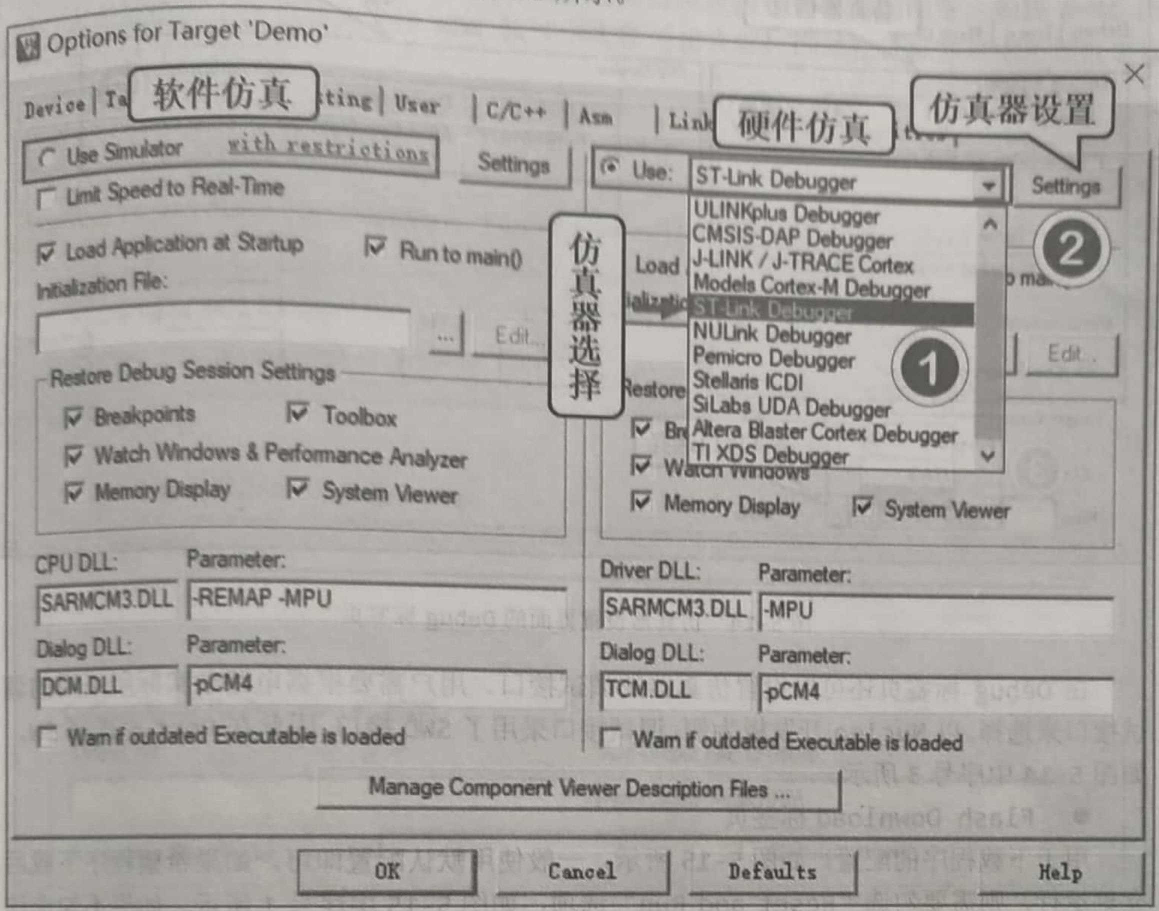


图 5-13　Debug 标签页

MDK 支持软件仿真和硬件仿真两种仿真方式。软件仿真主要用于验证算法和程序流程是否正确。在工程应用中，我们更多的是采用硬件仿真进行程序的调试和错误的排查。

使用硬件仿真时，首先要选择所使用的仿真器：如果使用官方开发板，如 Nucleo 开发板，板上集成了 ST-Link 仿真器，则选择 ST-Link Debugger；如果是自己设计的电路板，则根据实际使用的仿真器来选择，如工程中常用的 J-Link 仿真器，则选择 J-LINK/J-Trace Crotex。MDK 软件所支持的仿真器选项如图 5-13 中序号 1 所示。

选择好仿真器后，还需要进行仿真器的设置。点击仿真器右边的“Setting”按钮，如图 5-13 中序号 2 所示，将进入仿真器设置窗口。

仿真器设置界面分成了 Debug、Trace、Flash Download 和 Pack 四个标签页，如图 5-14 所示。用户只需要设置 Debug 和 Flash Download 标签页即可。

- Debug 标签页

用于显示仿真器的名称、序列号、硬件版本号和固件版本号等基本信息，如图 5-14 中序号 1 所示。如果仿真器正确连接，用户的电路板设计正确并上电，将会在右边的 SW Device 窗口中显示芯片的 ID 号，如图 5-14 中序号 2 所示。如果无法显示仿真器的相关信息，可以检查一下仿真器是否插紧，仿真器的驱动程序是否安装。如果无法显示芯片的 ID 号，则先检查一下电路板是否上电，如果电路板已经上电还是无法检测，则要考虑芯片最小系统的电路设计是否有问题，可以检查一下时钟电路、复位电路和调试电路来排查错误。

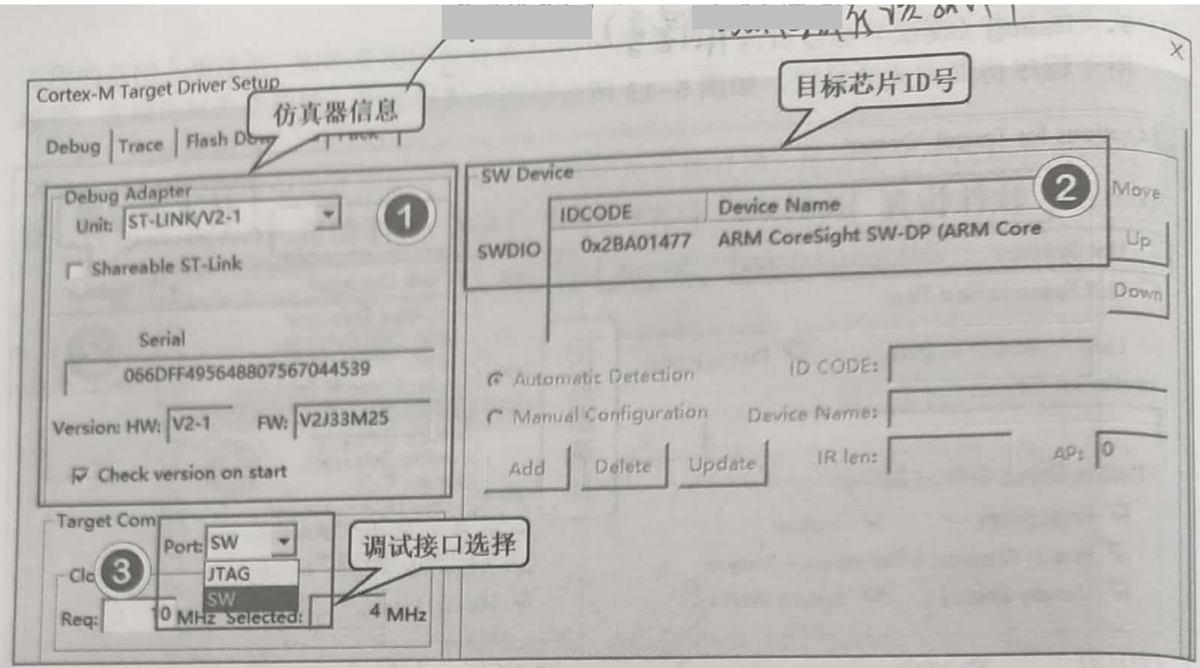


图 5-14　仿真器设置界面的 Debug 标签页

在 Debug 标签页还可以设置仿真器的调试接口。用户需要根据电路中实际所采用的调试接口来选择。以 Nucleo 开发板为例，调试接口采用了 SWD 接口，因此在 Port 栏选择 SW，如图 5-14 中序号 3 所示。

- Flash Download 标签页

用于下载程序的配置，如图 5-15 所示，一般使用默认配置即可。如果希望程序下载后立刻运行，则需要勾选“Reset and Run”选项，如图 5-15 中序号 1 所示。如果不勾选该选项，则下载程序后，需要按一下电路板上的复位按钮，程序才能运行。在 Flash 编程算法的设置上，一旦确定了芯片型号，将会自动添加所选微控制器片内 Flash 的编程算法，如图 5-15 中序号 2 所示。

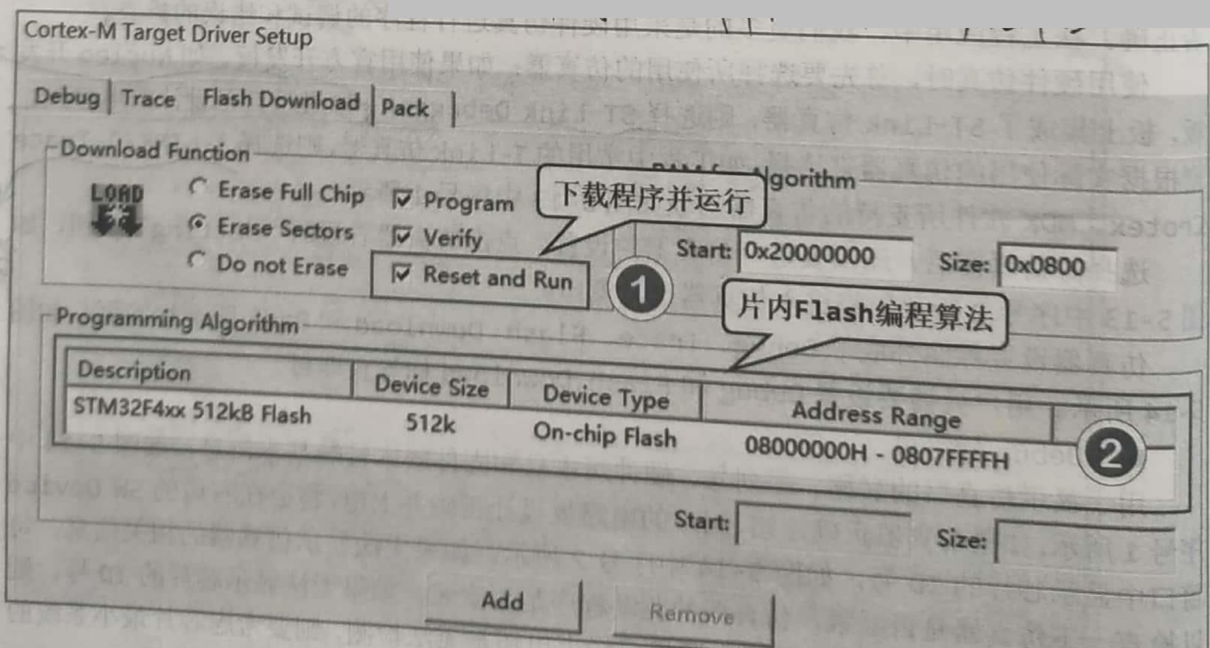


图 5-15　仿真器设置界面的 Flash Download 标签页

10. Utilities（工具）标签页

用于选择 MDK 软件中没有支持的第三方仿真器以及进行仿真器的配置，如图 5-16 所示。大多数情况下我们都会选择 MDK 软件中支持的仿真器，因此在该标签页中不做任何修改，使用默认配置即可。

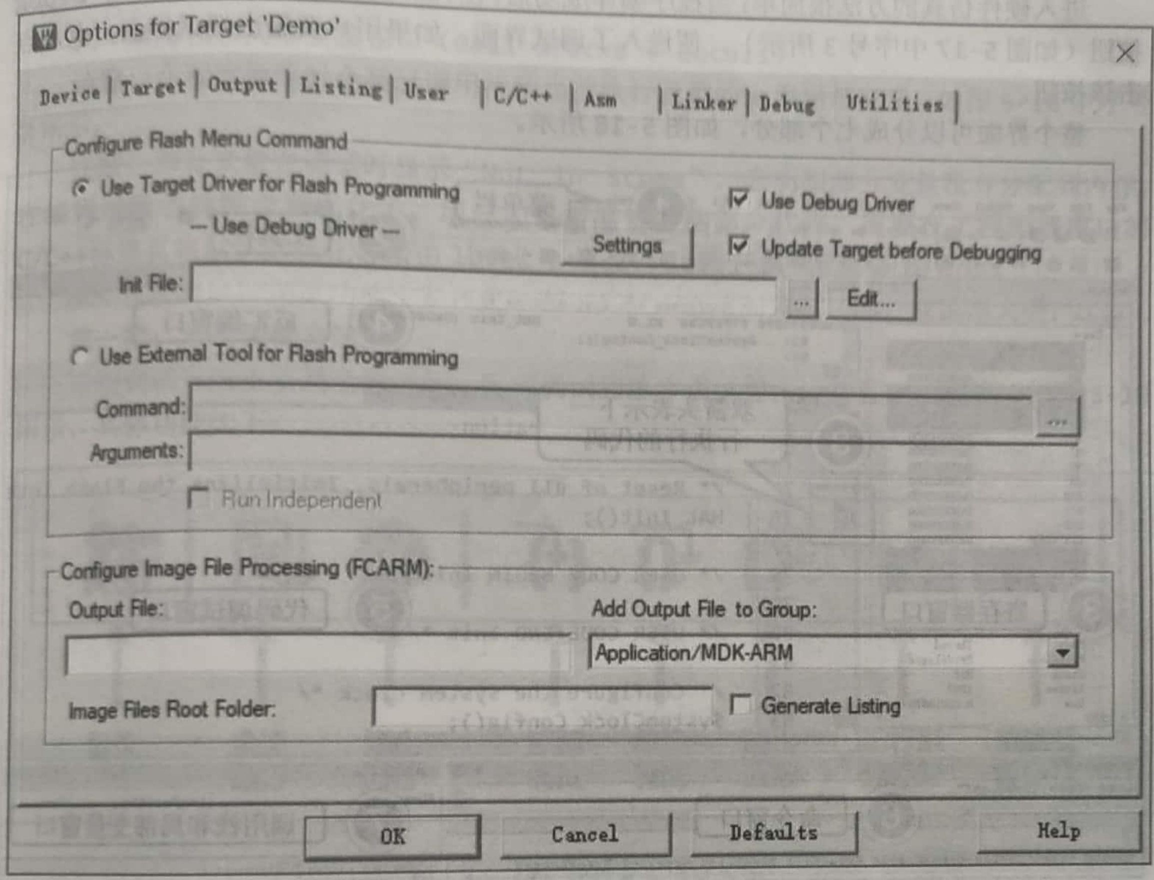


图 5-16　Utilities 标签页

注意：完成所有标签页的配置后，一定记得点击“OK”按钮，才能使做所的配置生效。

5.1.3　程序编译和下载

完成工程配置之后，就可以进行程序的编译和下载。在 MDK 主界面的工具栏上点击“Build”图标，如图 5-17 中序号 1 所示，进行程序的编译和链接。如果程序有误，需要根据信息输出窗口中的提示信息，返回源文件进行修改，直至没有错误（Error）和警告（Warning）为止 。完成编译后，点击工具栏中的“Download”图标，进行程序的下载，如图 5-17 中序号 2 所示。

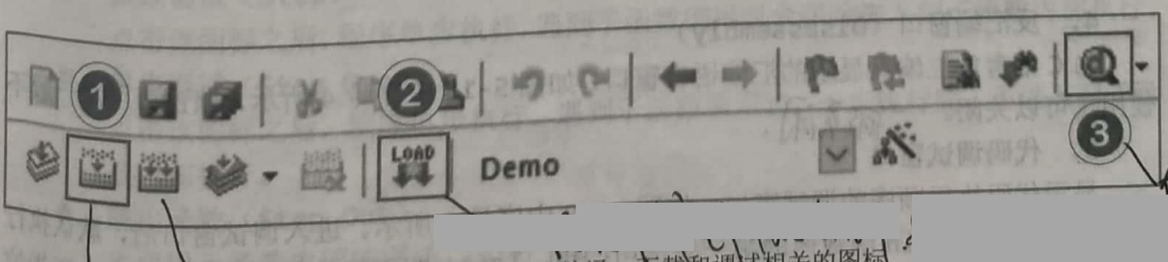


图 5-17　主界面中与程序编译、下载和调试相关的图标

5.1.4 程序调试

MDK 软件支持了强大的调试功能，几乎能够仿真芯片所有的外设及硬件寄存器。MDK 软件提供了软件仿真和硬件仿真两种手段，软件仿真主要用于验证算法和程序流程是否正确，在工程应用中，我们更多的采用硬件仿真进行程序的调试和错误的排查。

进入硬件仿真的方法很简单：当程序编译成功后，在 MDK 主界面的工具栏上点击“Deubg”按钮（如图 5-17 中序号 3 所示），便进入了调试界面。如果用户要退出调试界面，也是点击该按钮。

整个界面可以分成七个部分，如图 5-18 所示。

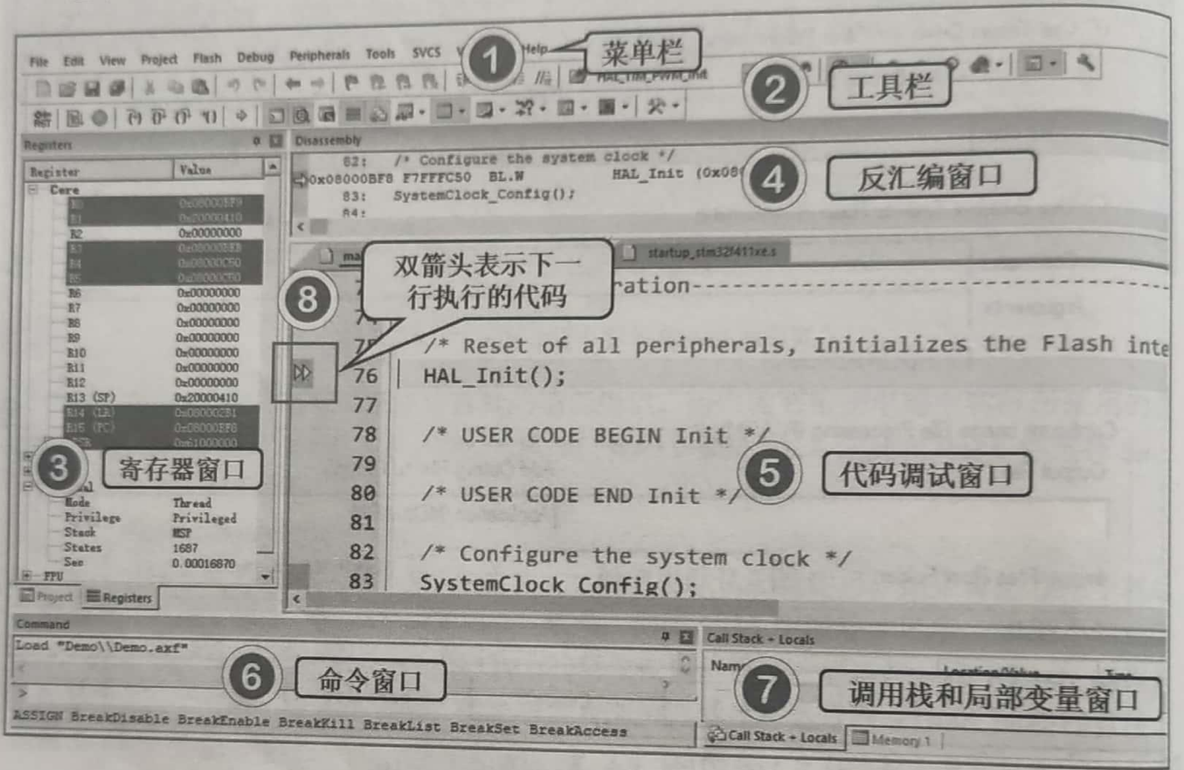


图 5-18 MDK 软件的调试界面

1. 菜单栏

提供调试模式下的全部功能菜单，如图 5-18 中序号 1 所示。

2. 工具栏

提供调试模式下的常用快捷功能，包括全速、跟踪、单步和断点等调试方法以及变量窗口、外设窗口等可视化调试窗口，如图 5-18 中序号 2 所示。

3. 寄存器窗口（Register）

显示微控制器片内寄存器、系统堆栈，处理器的工作模式以及程序的运行时间等信息，如图 5-18 中序号 3 所示。

4. 反汇编窗口（Disassembly）

由 C 语言反汇编后显示的汇编语言窗口，如图 5-18 中序号 4 所示。该窗口用户一般不使用，可以关闭。一般关闭

5. 代码调试窗口

显示代码执行顺序的调试窗口，如图 5-18 中序号 5 所示。进入调试窗口后，默认执行的第一条语句是 main 函数内的第一条语句：HAL_Init，即双箭头所指位置。双箭头所指位

置表示了下一条将要执行的语句的位置，如图 5-18 中序号 8 所示。在进行单步或者断点调试时，双箭头会始终指向下一步将要执行的语句。

6. 命令窗口（Command）

用于显示调试过程中的一些系统信息，如仿真器的状态等信息，如图 5-18 中序号 6 所示。

7. 调用栈和局部变量窗口（Call Stacks + Locals）

该窗口在程序暂停时会显示调用栈和当前执行函数的全部局部变量，如图 5-18 中序号 7 所示。

注意：部分变量在调试时显示“Not in scope”，表明则部分变量没有分配到内存，在编译过程中被编译器优化了。用户如果需要查看这部分代码，需要在工程配置窗口的 C/C++标签页将默认的优化等级由 level 3 修改为 level 0，如图 5-10 所示。

MDK 软件提供的调试功能主要在集中在工具栏，常用的调试功能分成了四大类：

第一类：常规调试方法

常规调试方法主要是全速运行、跟踪调试和单步调试等，一共有 8 个图标，如图 5-19 所示，具体功能如下：

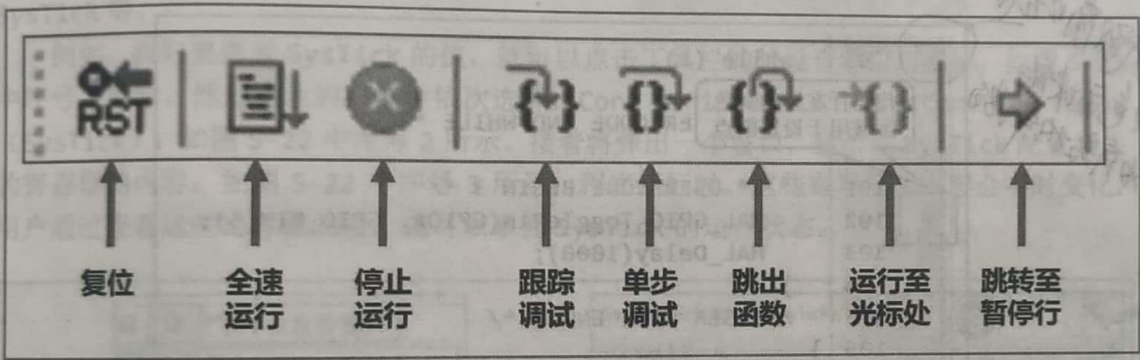


图 5-19 MDK 软件常规调试方法

- 复位（Reset）

  其功能等同于硬件复位，让程序复位到起点，代码会重新从头开始执行。
- 全速运行（Run）

  点击该图标之后，程序全速运行，可以观察程序运行的实际效果。如果运行中遇到用户设置的断点，将会在断点处暂停。
- 停止运行（Stop）

  点击该图标之后，程序会暂停执行，进入单步调试状态：双箭头指向下一条将要执行的代码。该图标在程序全速执行的时候有效。
- 跟踪调试（Step）

  点击该图标之后，程序单步执行，遇到子函数调用时会跟踪进入到子函数内部执行。
- 单步调试（Step Over）

  点击该图标之后，程序单步执行，遇到子函数调用时，一步执行完，不进入到子函数内部执行。
- 跳出函数（Step Out）

  主要用于函数内部的调试。当用户不想再执行函数内部的剩余代码时，可以点击该图标跳出函数，回到函数被调用的位置。

- 执行至光标处（Run to Cursor Line）

  该图标可以迅速使程序运行到光标处，功能类似于全速运行至断点。两者的区别在于：断点可以设置多个，但是光标所在处只有一个。

- 跳转至暂停行（Show Next Statement）

  点击该图标，可以跳转到程序暂停时的所在行。主要用于打开多个文件后，不知道程序的暂停位置时使用。该图标在程序停止运行时有效。

第二类：断点

断点是程序调试的重要手段之一，可以帮助用户快速执行到程序错误处，进行问题的排查。在调试界面中设置断点的方法如下：

- 添加断点

在代码调试窗口中，单击语句行号旁边的灰色区域，将出现一个小红点，表明已经在该语句处添加了一个断点。程序执行到断点时就会暂停，用户可以结合调试窗口中的相关信息来排查程序中的错误。

例如，在第 102 行代码处单击灰色区域，出现一个小红点，表明已经添加了断点，程序执行到 102 行时就会暂停，如图 5-20 所示。

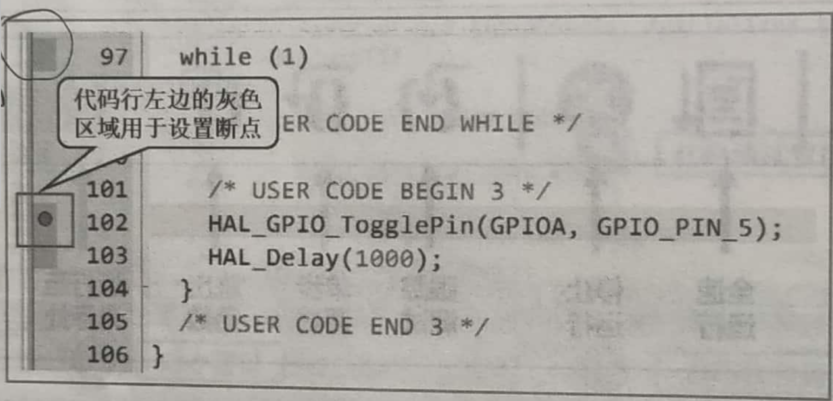


图 5-20　添加断点

- 取消断点

在需要取消的断点处，点击断点的小红点标记，小红点标记将消失，表明完成了断点的取消，程序再次执行到此处时，将不会暂停。

第三类：观察窗口（Watch Windows）

主要用于查看全局变量的值。在程序调试过程中我们常常需要查看很多变量的值，来判断程序运行是否正确，而调用栈和局部变量（Call Stacks + Locals）窗口只能查看程序中局部变量的值。如果要查看全局变量的值，就需要通过观察窗口来完成。如果要在观察窗口中查看局部变量的值，必须要程序运行到局部变量所在的函数时才能观察。

将变量添加至观察窗口的过程如下：

① 在源码调试窗口中选中需要观察的变量。

② 单击鼠标右键，在出现的菜单中选择“Add ‘value’ to”，例如这里定义的全局变量“value”；并在二级子菜单中选择观察窗口“Watch 1”或者“Watch 2”，如图 5-21 中序号 1 所示；

③ 在调试界面的右下角将会出现观察窗口，显示变量名、变量值和变量的数据类型等基本信息，如图 5-21 中序号 2 所示；；

④ 在观察窗口中变量的值默认为 16 进制，用户可以右键点击变量，修改变量的显示格式为十进制，即去掉“Hexadecimal Display”，如图 5-21 中序号 3 所示。

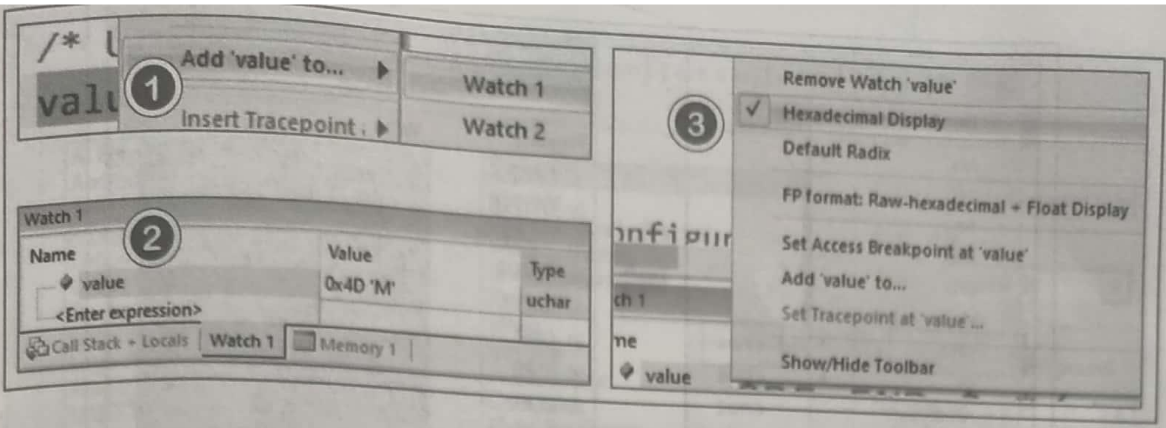


图 5-21　添加变量至观察窗口

第四类：外设查看窗口（System Viewer Windows）

主要提供各种片内外设的寄存器的查看窗口，便于用户了解外设的状态信息。片内外设分成了两大类：

一类是 Cortex-M 内核固有外设，如嵌套向量中断控制器 NVIC、系统节拍定时器 SysTick 等。

例如，我们要查看 SysTick 的值，就可以点击工具栏的外设查看窗口图标，如图 5-22 中序号 1 所示。然后出现的菜单中依次选择：Core Peripherals -> System Tick Timer（SysTick），如图 5-22 中序号 2 所示。接着将弹出一个窗口，显示和 SysTick 配置相关的寄存器的内容，如图 5-22 中序号 3 所示。程序运行时，这些寄存器的内容会实时变化，用户通过查看这些寄存器的值，就可以掌握 SysTick 的运行状态。

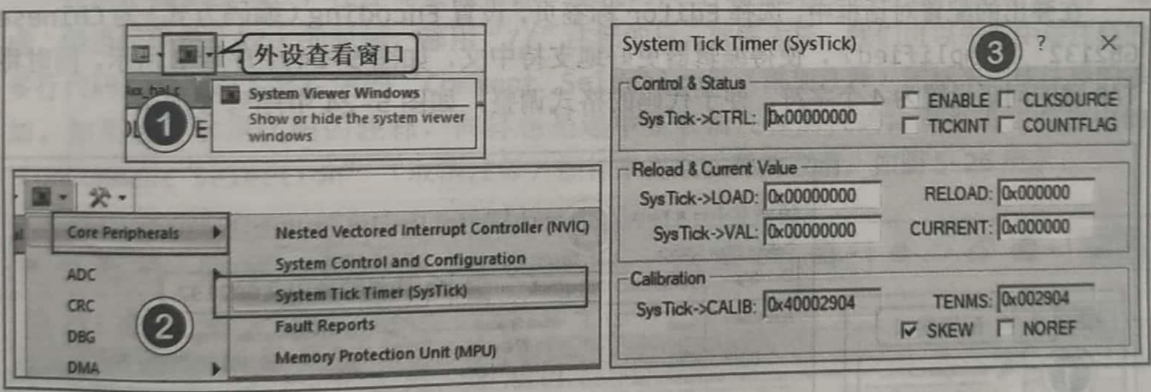


图 5-22　查看系统节拍定时器的寄存器

另一类是目标芯片的片内外设，如 GPIO、定时器和串口等，具体可显示的片内外设会根据所选择的芯片型号发生变化。

例如，我们查看端口 GPIOA 对应的寄存器，点击外设查看窗口图标后，在出现的菜单中依次选择：GPIO -> GPIOA，接着将弹出一个窗口，显示与端口 GPIOA 端口配置相关的全部寄存器的内容，如图 5-23 所示。

程序运行时，这些寄存器的内容会实时变化，用户通过查看这些寄存器的值，就可以掌握端口 GPIOA 各个引脚的相关状态。

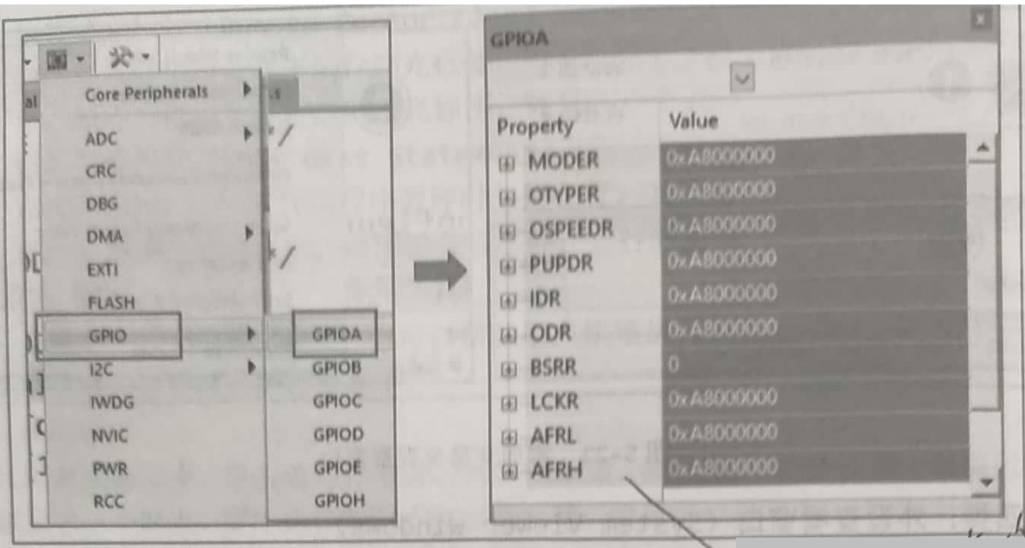


图 5-23　查看端口 GPIOA 的寄存器

5.2　MDK 软件的实用功能

MDK 开发软件还提供了一些实用功能，方便用户进行程序的开发和调试，这里把几个常用的功能给大家介绍一下。

1. 编辑器配置

编辑器设置主要选择文本的编码方式、代码的字体大小、关键字的颜色等。在 MDK 主界面的工具栏中，选择配置（Configutation）图标，如图 5-24 中序号 1 所示。

在弹出的配置对话框中，选择 Editor 标签页，设置 Encoding（编码方式）为 Chinese GB2132 （Simplified），使得编辑器更好地支持中文，如图 5-24 中序号 2 所示。同时将 TAB 键的间隔设置为 4 个字符，便于代码的格式调整，如图 5-24 中序号 3 所示。

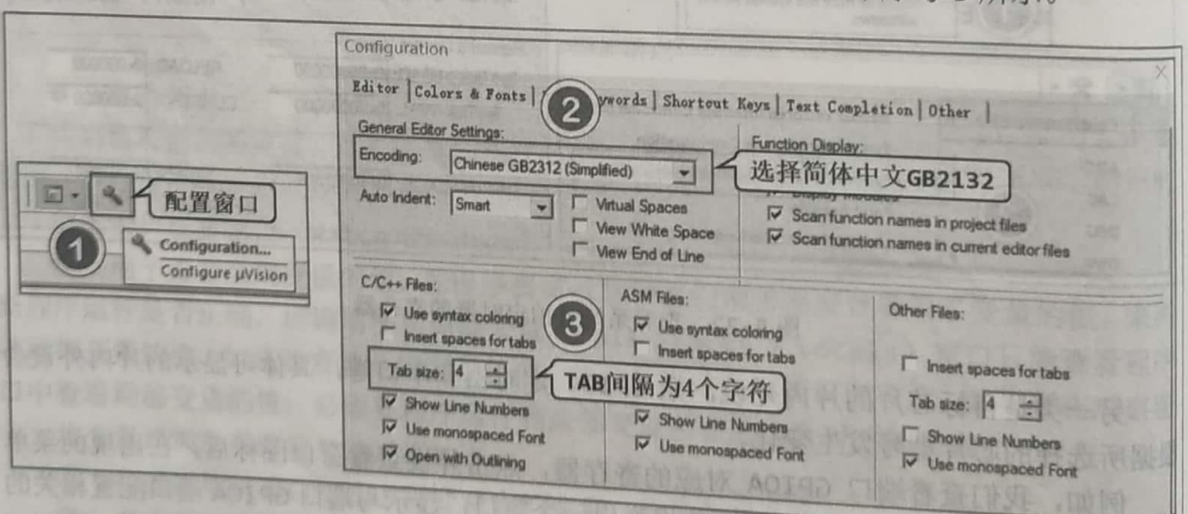


图 5-24　配置编码方式及 TAB 键间隔字符

在 Color&Font 标签页中，可以设置代码的字体和颜色。由于我们使用的是 C 语言编程，所以在 Windows 栏选择“C/C++ Editor”，然后在 Element 栏可以选择文本、数字、C 语言关键字、字符等多个元素，在 Font 栏去设置对应元素的字体类型、字体大小和字体颜色等参数。用户可以根据自己的编程习惯进行设置，如图 5-25 所示。

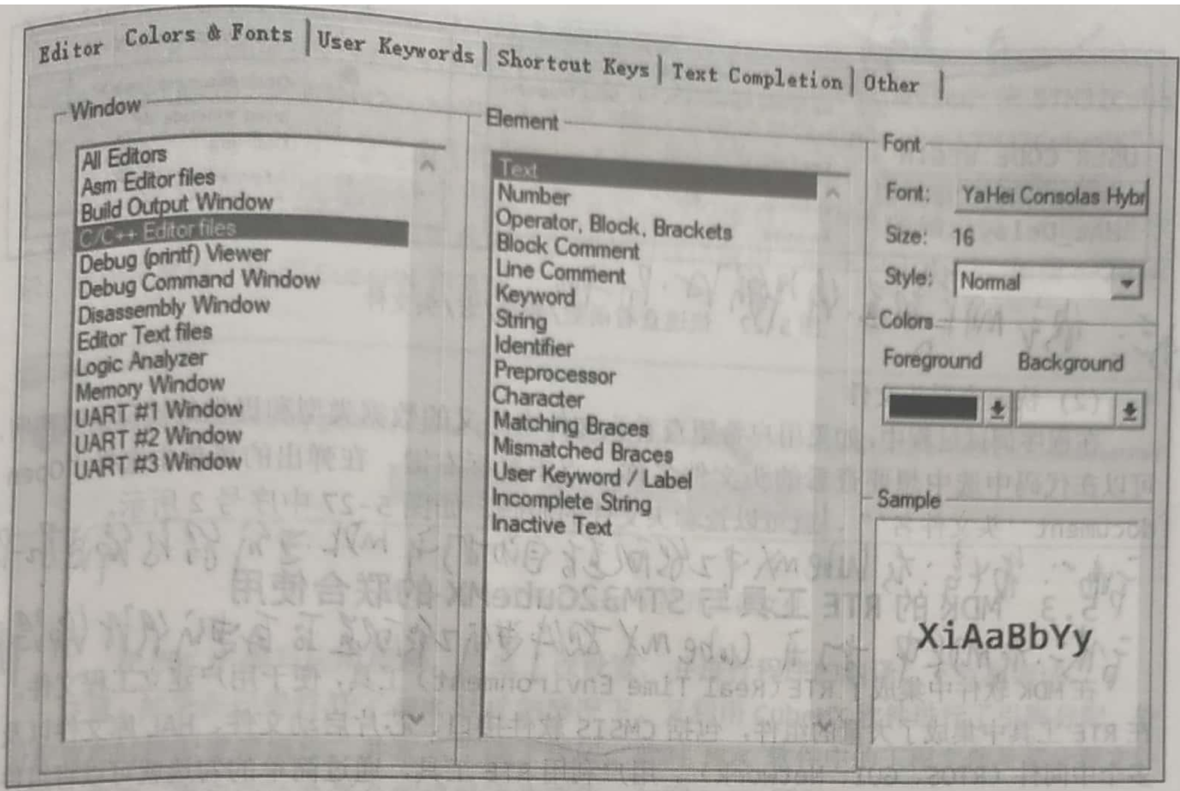


图 5-25　配置字体类型、大小及颜色

2. 批量注释代码和取消注释

在程序调试过程中，往往需要注释掉一些代码，当代码比较少的时候可以采用“//”完成，但当代码量比较大的时候，都用“//”注释的话，效率太低。我们可以先选中要注释的多行代码，再点击工具栏中的“Comment Selection”（添加注释）图标来完成注释的添加。如果想要取消代码的注释，同样也是选中要取消注释的代码，再点击工具栏中的“UnComment Selection”（取消注释）图标来完成注释的取消。如图 5-26 所示。

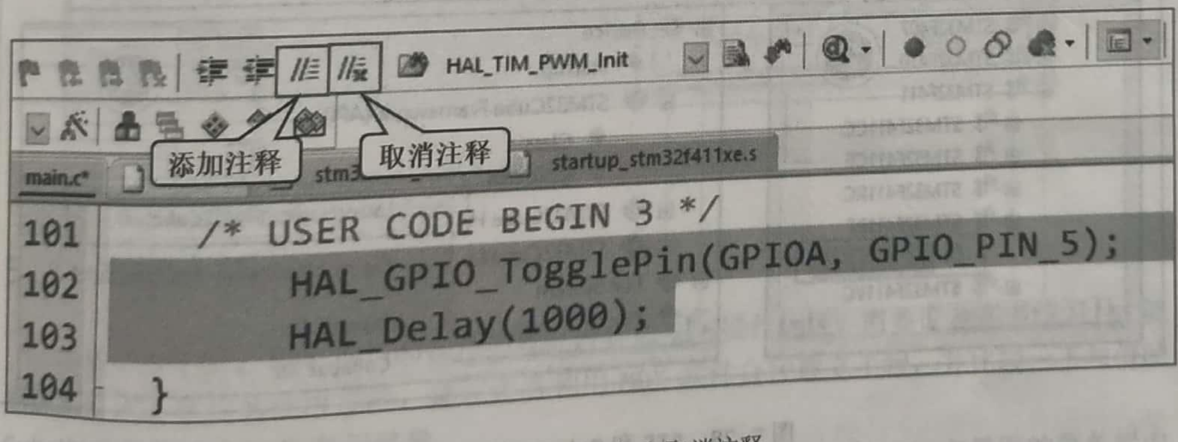


图 5-26　批量添加和取消注释

3. 快速查看

(1) 快速查看函数/变量/宏的定义

在程序调试过程中，遇到某些函数、变量和宏时，往往需要查看它们的原始定义。可以先选中函数、变量或宏，点击鼠标右键，在弹出的菜单中选择“Go To Definition of '函数名/变量名/宏名'”，就可以查看到对应的原始定义，如图 5-27 中序号 1 所示。

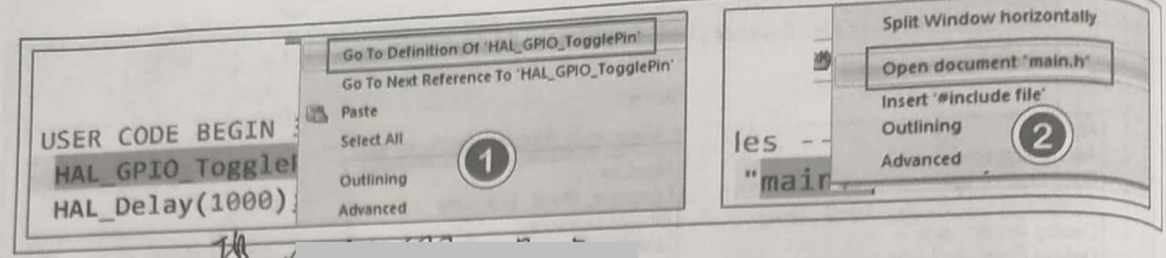


图 5-27　快速查看函数/变量/宏/头文件

(2) 快速查看头文件

在程序调试过程中，如果用户希望查看头文件中定义的数据类型和提供的接口函数声明，可以在代码中选中想要查看的头文件名称，点击鼠标右键，在弹出的菜单中选择“Open document '头文件名'”，就可以查看头文件的源码，如图 5-27 中序号 2 所示。

5.3　MDK 的 RTE 工具与 STM32CubeMX 的联合使用

在 MDK 软件中集成了 RTE（Real Time Environment）工具，便于用户建立工程文件。在 RTE 工具中集成了大量的组件，包括 CMSIS 软件接口、芯片启动文件、HAL 库文件以及多个中间件（RTOS、GUI、Network）。用户利用 RTE 工具，通过简单的勾选就可以将组件自动添加到自己的工程中，不需要手动移植，并且还可以和 CubeMX 软件联合使用来建立工程。

这里简单介绍一下 RTE 工具和 CubeMX 软件联合创建工程的步骤，如图 5-28 所示：

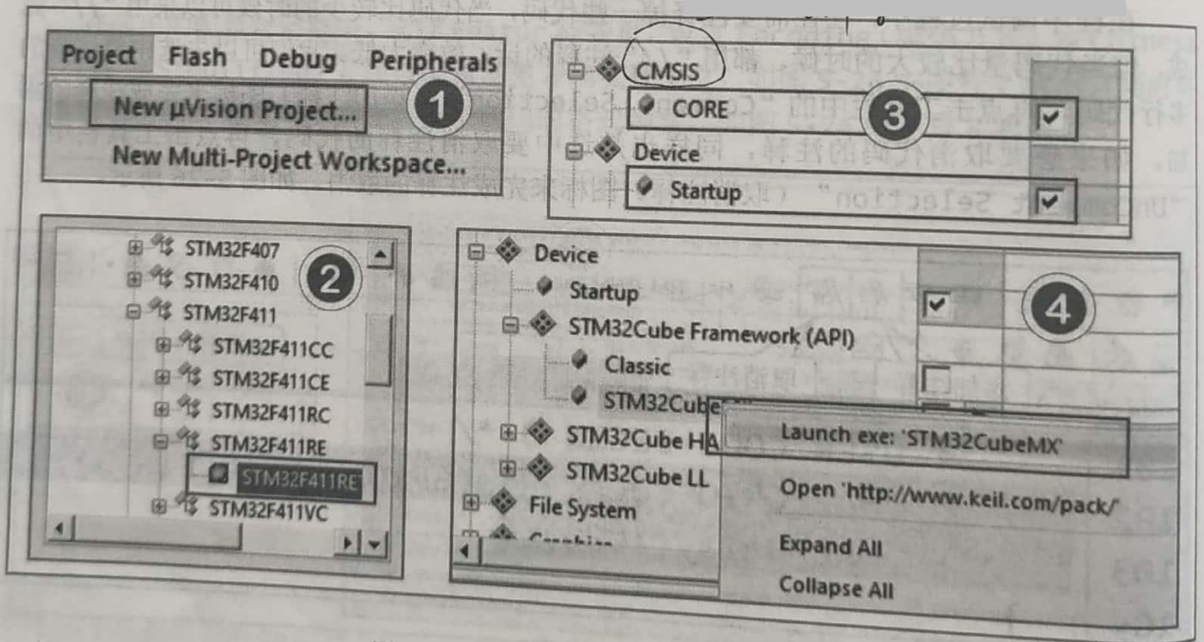


图 5-28　RTE 和 CubeMX 的联合使用

1. 在桌面上新建一个名为 RTE 的文件夹。
2. 启动 MDK 软件，在主界面的菜单栏选择：Project -> New uVsion Project，选择新建的 RTE 文件夹，输入工程名：RTE，如图 5-28 中序号 1 所示。
3. 选择目标 MCU，如 Nucleo 开发板上的 STM32F411RE，如图 5-28 中序号 2 所示。
4. 芯片选择后，将自动启动 RTE 工具。在 RTE 窗口中勾选 CMSIS：CORE 和 Device:

Startup 两个组件，如图 5-28 中序号 3 所示。

5. 在 RTE 环境中，勾选 STM32CubeMX ，然后启动 CubeMX [redacted] Device → STM32Cube Framework（API） → STM32CubeMX。先勾选，然后右键选择“Lanch exe STM32CubeMX”，如图 5-28 中序号 4 所示。[redacted]

6. 在启动的 CubeMX 软件中完成引脚分配，外设配置，时钟配置等工作（不需要进行工程配置），并生成代码。代码生成后，在弹出的对话框中，选择“Close”，如图 5-29 所示。完成设置后，关闭 CubeMX 和 RTE。

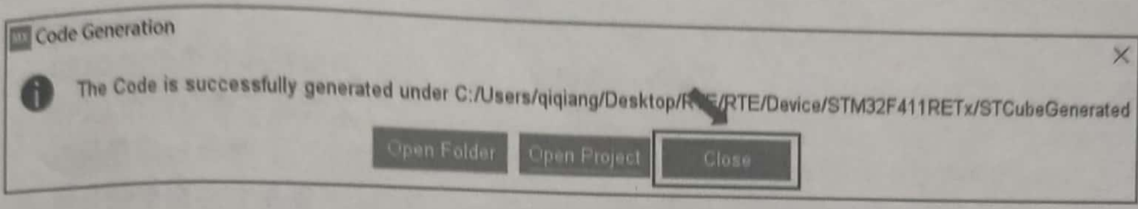


图 5-29 选择 Close 不打开工程文件

7. 在 MDK 中编写应用程序，完成工程设置，并进行程序的编译、下载和调试。

注意： 如果在已经打开了 MDK 软件的情况下，又利用 CubeMX 软件进行了引脚分配、外设配置和时钟配置等操作，并重新生成了代码。此时 MDK 软件中的工程文件及部分源文件将发生更改，用户应该选择全部接受这些变化，如图 5-30 所示。

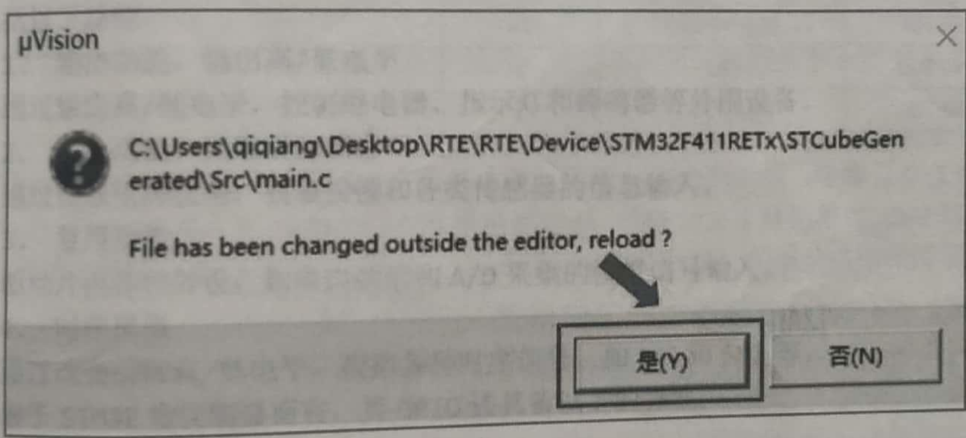


图 5-30 选择接受代码变化

5.4 本章小结

本章重点介绍了 MDK 软件的使用方法，内容以实际操作为主，需要读者多多练习，才能熟练掌握 MDK 软件的使用方法。建议读者利用 MDK 软件自行建立工程，并比较一下与借助 CubeMX 软件来建立工程有何差异。

关于 MDK 软件更为详细的介绍，可以参考软件的帮助手册。在 MDK 主界面的菜单栏中选择 help 菜单，即可找到相关的参考资料。

习题 5

1. 利用 MDK 软件的调试功能，调试在第四章中生成的指示灯闪烁程序，熟悉复位、全

速执行、单步调试、跟踪调试、断点调试及变量查看等常用的调试手段。

2. 利用 MDK 软件的调试功能，调试在第四章中生成的指示灯闪烁程序，分析调试过程中端口 GPIOA 的哪些寄存器发生了变化。

3. 查询 MDK 软件的帮助手册，在 MDK 的调试界面中利用逻辑分析窗口显示引脚 PA5 输出的波形。

第六章 通用输入输出接口（GPIO）

本章导读

本章重点介绍通用输入/输出接口（GPIO）的基本原理、电路结构和工作模式。在编程方式上，先从 GPIO 外设的寄存器入手，介绍寄存器的编程方式，再过渡到基于 HAL 库的库函数编程方式。最后介绍硬件抽象层的设计思想，并以指示灯驱动设计为例介绍具体实现方法。

本章知识点

GPIO 的作用和电路结构

GPIO 的硬件寄存器含义

库函数设计思想

硬件抽象层设计思想

6.1 概述

6.1.1 GPIO 功能及特性

GPIO 是通用输入/输出接口（General Purpose Input Output）的简称，主要用于工业现场需要数字量输入/输出的场合，是微控制器中使用最广泛，也是最重要的外设，主要包括以下功能：

1. 输出功能：输出高/低电平

通过输出高/低电平，控制继电器、指示灯和蜂鸣器等外围设备。

2. 输入功能：读取引脚状态

通过读取引脚状态，获取按键和各类传感器的信息输入。

3. 复用功能

驱动片内各种外设，如串口通信和 A/D 采集的模拟信号输入。

4. 时序模拟

通过改变引脚高/低电平，模拟各种时序信号，如 I²C 和 SPI、UART 等。

对于 STM32 微控制器而言，其 GPIO 还具备以下的特性：

● 多种工作模式

提供输入模式、输出模式、模拟模式和复用模式等 4 种工作模式。

● 灵活复用

STM32 微控制器内部集成了很多外设，这些外设的外部引脚都是与 GPIO 模块的引脚共用。引脚可以通过寄存器的配置，选择为 GPIO 或者外设功能中的一种，比如作为串口通信引脚或者 A/D 采集的输入通道等。

● 5V 电压容限

大部分引脚具备 5V 电压容限，可以接受 5V 电压的输入，方便与 5V 供电的外围电路连接，具体的引脚信息可查阅芯片的数据手册。

● 外部中断

每一个 I/O 引脚都具备外部中断能力。

学习 GPIO 外设时要注意两个重要的概念：端口和引脚。

1. 端口（PORT）

STM32 微控制器将 GPIO 分成了很多个组，每一组称为端口，如 GPIOA、GPIOB 和 GPIOH 等。各组端口可以看成独立的子模块，同一组端口的工作模式可以通过相同的多个寄存器来进行配置。

2. 引脚（PIN）

每一组端口一般有 16 个引脚，如端口 GPIOA 对应的 16 个引脚为 PA0～PA15，其中 PA0 表示该端口的第 0 个引脚。这些引脚在硬件上对应 MCU 的一个管脚，同时也对应到控制该组端口的多个寄存器的某一位或某几位，如 PA0 对应 STM32F411RET6 芯片的 14 脚，也对应到控制端口 GPIOA 的输出数据寄存器 GPIOA_ODR 的第 0 位（bit0）。

从上面的描述可以看出：端口相当于一个独立的外设子模块，由多个寄存器构成。而引脚则对应 MCU 的一个管脚。一个端口可以控制多个引脚，通过对相关寄存器的配置来设定引脚的工作模式。

不同型号和规格的 STM32 微控制器所包含的 I/O 引脚数量不同。STM32F411RET6 一共有 50 个 I/O 引脚，分成了 GPIOA、GPIOB、GPIOC、GPIOD 和 GPIOH 等 5 组端口，其端口和引脚的配置如表 6-1 所示。

表 6-1 STM32F411RET6 的端口和引脚配置

端口	引脚	数量	备注
GPIOA	PA0～PA15	16	—
GPIOB	PB0～PB15	15	PB11 引脚被 VCAP_1 引脚取代
GPIOC	PC0～PC15	16	—
GPIOD	PD2	1	
GPIOH	PH0、PH1	2	

50 个 I/O 引脚中，PH0 和 PH1 一般用于接入外部高速时钟 HSE，PA13 和 PA14 用于串行调试接口 SWD。如果用到了 RTC 模块，还需要使用 PC14 和 PC15 接入外部低速时钟 LSE。因此，实际留给用户使用的 I/O 引脚只有 44 个。

6.1.2 GPIO 的电路结构和工作模式

STM32F4 微控制器的 GPIO 电路结构如图 6-1 所示。

整个电路结构从左到右可以看做由 3 部分组成：

1. 端口控制寄存器组

图 6-1 的左边是控制 GPIO 端口的寄存器组，用于设置属于该组端口的所有引脚的工作模式。这里只展示了 3 个寄存器，分别是置位/复位寄存器、输出数据寄存器和输入数据寄存器。用户通过设置这 3 个寄存器的内容，并利用相应的电路元件，就可以在该端口指定的引脚上输出高/低电平或者读取指定引脚的电平状态。

2. 输入/输出驱动器

图 6-1 的中间是输入/输出驱动器。驱动器中包含的电路元件可以将输出寄存器中的 0 或 1 数据转换为引脚上的高/低电平，或者将引脚电平状态转换为输入寄存器中的 0 或 1 数据。

● 输入驱动器

输入驱动器的主要电路元件是施密特触发器，其电路原理是当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低。I/O 引脚的电压经过触发器后，将转换为 0 或 1 的数字信号。

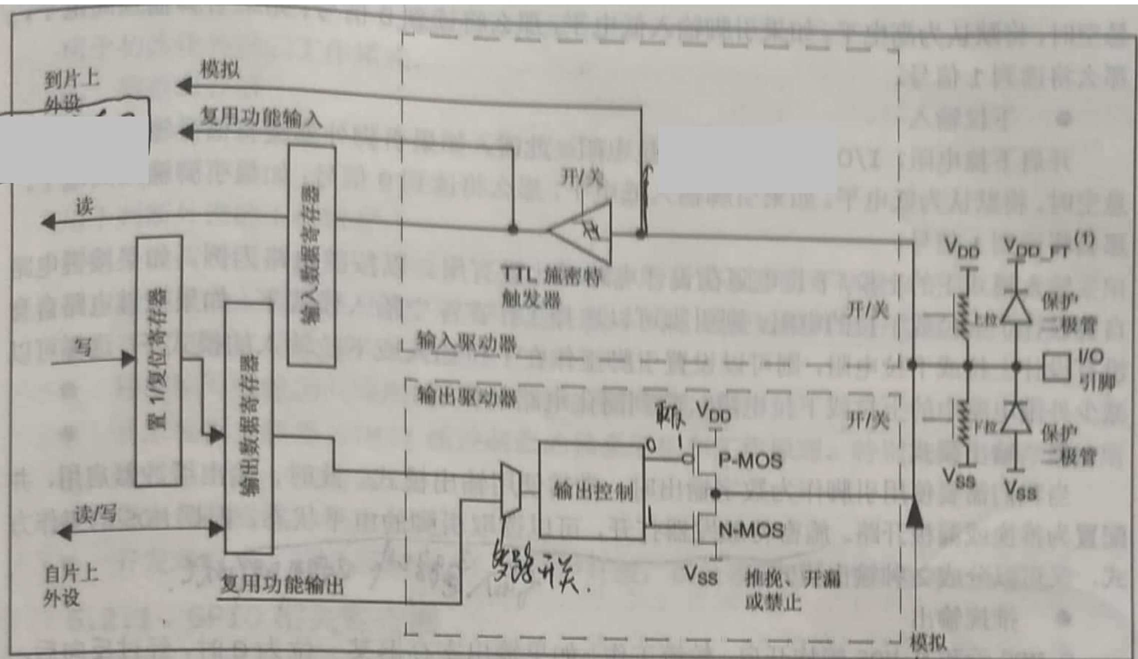


图 6-1　STM32F4 微控制器 GPIO 电路结构

● 输出驱动器

输出驱动器的主要电路元件是一对 P-MOS 管和 N-MOS 管。上方的 P-MOS 管低电平导通，高电平关闭；下方的 N-MOS 高电平导通，低电平关闭。根据两个 MOS 管的工作方式，使得 GPIO 具有“推挽输出”和“开漏输出”的工作模式。

3. 引脚电路

图 6-1 的右边是引脚电路。引脚电路由 2 组电路元件组成：

● 保护二极管

I/O 引脚上下两边的 2 个二极管用于防止引脚外部过高或过低的电压输入。当引脚电压高于 VDD_FT（5V 电压容限）时，上方的二极管导通；当引脚电压低于 VSS 时，下方的二极管导通，防止不正常电压引入芯片导致芯片烧毁。

● 上拉/下拉电阻

I/O 引脚上下两边的 2 个电阻用于控制引脚默认状态时的电压。上拉电阻连接在引脚和电源 VDD 之间，下拉电阻连接在引脚与地 VSS 之间。上拉/下拉电阻可以利用寄存器控制：当开启上拉电阻时，引脚默认电压为高电平；当开启下拉电阻时，引脚默认电压为低电平。

根据 GPIO 的电路结构，引脚可以分为 4 种模式：

1. 输入模式

当我们需要读取引脚的电平状态时，常常使用输入模式。此时，输出缓冲器被禁用，施密特触发器打开，引脚的输入信号经过施密特触发器转换后，送入到输入数据寄存器。根据是否开启上拉/下拉电阻，又可以分成 3 种输入模式：

● 浮空输入

不开启上拉/下拉电阻。此时，I/O 引脚的电平状态不确定，由外部输入决定。MCU 上电复位后，GPIO 默认的工作模式就是浮空输入。

● 上拉输入

开启上拉电阻，I/O 引脚内部接上拉电阻。此时，如果引脚外部没有信号输入或者引脚

悬空时，将默认为高电平。如果引脚输入低电平，那么将读到 0 信号；如果引脚输入高电平，那么将读到 1 信号。

● 下拉输入

开启下拉电阻，I/O 引脚内部接下拉电阻。此时，如果引脚外部没有信号输入或者引脚悬空时，将默认为低电平。如果引脚输入低电平，那么将读到 0 信号；如果引脚输入高电平，那么将读到 1 信号。

输入模式下的上拉/下拉电阻在设计电路时比较有用。以按键电路为例，如果按键电路自身设计了上拉或下拉的电阻，则引脚可以选择工作在浮空输入模式下。如果按键电路自身没有设计上拉或下拉电阻，则可以设置引脚工作在上拉输入或下拉输入的模式下。这样可以减少外围电路中的上拉或下拉电阻，起到简化电路的作用。

2. 输出模式

当我们需要使用引脚作为数字输出时，常常使用输出模式。此时，输出缓冲器启用，并配置为推挽或漏极开路。施密特触发器打开，可以读取引脚的电平状态。根据 MOS 的工作方式，又可以分成 2 种输出模式：

● 推挽输出

P-MOS 管和 N-MOS 管均开启，轮流工作。如果输出寄存器某一位为 0 时，经过反向后，P-MOS 管关闭，N-MOS 管导通，对应引脚输出低电平。如果输出寄存器某一位为 1 时，经过反向后，P-MOS 管导通，N-MOS 管关闭，对应引脚输出高电平。引脚电路部分的上拉/下拉电阻的作用是控制在没有输出时的引脚电平。

● 开漏输出

P-MOS 管关闭，只有 N-MOS 管工作。如果输出寄存器某一位为 0 时，经过反向后，P-MOS 管关闭，N-MOS 管导通，对应引脚输出低电平。如果输出寄存器某一位为 1 时，经过反向后，P-MOS 管和 N-MOS 管均关闭，对应引脚无法输出高电平。此时，I/O 引脚的电平由引脚电路部分的上拉/下拉电阻决定。如果使能上拉电阻，输出高电平；如果使能下拉电阻，输出低电平；上拉/下拉都不使能，引脚处于悬空状态。

在输出模式中，推挽输出是最常用的模式，用于指示灯、蜂鸣器、继电器及电机等外围电路的控制。开漏输出则一般用于 I^2C 等需要“线与”功能的总线中。

3. 模拟模式

当我们需要使用 ADC 外设对外界信号进行采样或者通过 DAC 外设生成模拟信号时，可以使用模拟模式。此时，施密特触发器被旁路，输出缓冲器以及上拉/下拉电阻被禁用。模拟模式下，I/O 引脚的功耗最小，因此在工程应用中，常常将没有用到的 I/O 引脚设置为模拟模式，以减少系统功耗，提高抗干扰能力。

4. 复用模式

复用模式用于将引脚分配给 STM32 微控制器的片内外设使用。此时，引脚的电平状态不再受端口寄存器组的控制，而是由片内外设控制。STM32 片内集成了大量外设，如 UART、SPI 和 I^2C 等。这些外设通过多路复用器与 GPIO 引脚互连，用户可以修改多路复用器，重新分配片内外设所映射的 I/O 引脚。

6.2 基于寄存器方式控制 GPIO

微控制器通过寄存器来控制片内集成的众多外设，按照寄存器的用途可以大致分为三类：

1. 配置寄存器

用于初始化外设的工作模式。

2. 数据寄存器

用于外设的数据交换：向外设写入数据或者从外设读取数据。

3. 状态寄存器

用于判断外设的工作状态。

控制外设按照用户设定的要求来工作，其本质是设置不同的寄存器来完成，因此，采用寄存器方式进行嵌入式系统开发，是最传统、最本质、最接近硬件运行原理的开发方式。

基于寄存器方式开发嵌入式系统具有如下特点：

- 程序编写直接面向底层的寄存器，与硬件密切相关。
- 要求编程者熟悉 STM32 微控制器的体系架构和工作原理，特别是每个寄存器的用途及其中每一位的含义。
- 程序代码简练，执行效率高。
- 开发难度大，开发周期长，后期维护升级，以及硬件平台的移植等工作量较大。

6.2.1 GPIO 相关寄存器

如前所述，STM32 微控制器的 GPIO 分成了多组端口，每组端口的工作模式通过相同的多个寄存器来进行配置，寄存器中的对应位控制该端口中对应的引脚。STM32F411 分成了 5 组端口：GPIOA、GPIOB、GPIOC、GPIOD 和 GPIOH。

每组 GPIO 端口由 10 个 32 位的寄存器来控制，其中有 4 个配置寄存器（GPIOx_MODER、GPIOx_OTYPER、GPIOx_OSPEEDR 和 GPIOx_PUPDR），2 个数据寄存器（GPIOx_IDR 和 GPIOx_ODR），1 个置位/复位寄存器（GPIOx_BSRR），1 个锁定寄存器（GPIOx_LCKR）和 2 个复用功能选择寄存器（GPIOx_AFRH 和 GPIOx_AFRL）。在这 10 个寄存器中，和 GPIO 工作模式相关的主要有以下 7 个寄存器：

1. 模式寄存器（GPIOx_MODER）

该寄存器用于控制 GPIOx（x 代表具体的端口号，如 GPIOA、GPIOD 等，下同）端口各引脚的工作模式。寄存器各位的描述见表 6-2 所示。

表 6-2 端口模式寄存器 GPIOx_MODER

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

模式寄存器一共有 32 位，每 2 位一组，对应着 1 个不同的 I/O 引脚。这些位通过软件写入，用于配置 I/O 引脚的模式：

- 00：输入模式（复位值）
- 01：通用输出模式
- 10：复用功能模式
- 11：模拟模式

例如：向 GPIOA_MODER 寄存器的 bit1 和 bit0 写入 01，表示将 PA0 引脚设置为输出

模式；向 GPIOC_MODER 寄存器的 bit31 和 bit30 写入 11，表示将 PC15 引脚设置为模拟模式。

2. 输出类型寄存器（GPIOx_OTYPER）

该寄存器用于控制 GPIOx 端口各引脚的输出类型，该寄存器在引脚配置为输出模式后才有效，其他模式下无效。寄存器各位的描述见表 6-3 所示。

表 6-3 端口输出类型寄存器 GPIOx_OTYPER

31～16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
OT15	OT14	OT13	OT12	OT11	OT10	OT9	OT8	OT7	OT6	OT5	OT4	OT3	OT2	OT1	OT0

输出类型寄存器一共有 32 位，高 16 位保留，低 16 位的每 1 位对应一个不同的 I/O 引脚。这些位通过软件写入，用于配置 I/O 引脚的输出类型：

- 0：推挽输出（复位值）
- 1：开漏输出

例如：向 GPIOB_MODER 寄存器的 bit1 写入 1，表示将 PB1 引脚设置为开漏输出模式。

3. 输出速度寄存器（GPIOx_OSPEEDR）

该寄存器用于控制 GPIOx 端口各引脚的输出速度模式，该寄存器在引脚配置为输出模式后才有效，其他模式下无效。寄存器各位的描述见表 6-4 所示。

表 6-4 输出速度寄存器 GPIOx_OSPEEDR

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

输出速度寄存器一共有 32 位，每 2 位一组，对应着 1 个不同的 I/O 引脚。这些位通过软件写入，用于配置 I/O 引脚的输出速度：

- 00：低速（复位值）
- 01：中速
- 10：高速
- 11：超高速

例如：向 GPIOD_MODER 寄存器的 bit15 和 bit14 写入 11，表示将 PD7 引脚的输出速度设置为超高速。

4. 上拉/下拉寄存器（GPIOx_PUPDR）

该寄存器用于控制 GPIOx 端口各引脚的上拉/下拉电阻，可用于输入和输出模式。寄存器各位的描述见表 6-5 所示。

上拉/下拉寄存器一共有 32 位，每 2 位一组，对应着 1 个不同的 I/O 引脚。这些位通过软件写入，用于配置 I/O 引脚的上拉/下拉电阻：

表 6-5　上拉/下拉寄存器 GPIOx_PUPDR

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

- 00：既无上拉也无下拉（复位值）
- 01：上拉
- 10：下拉
- 11：保留

例如：向 GPIOE_MODER 寄存器的 bit17 和 bit16 写入 01，表示设置 PD8 引脚的上拉电阻功能。

5. 输入数据寄存器（GPIOx_IDR）

该寄存器用于读取 GPIOx 端口各引脚的电平，可用于输入和输出模式。寄存器各位的描述见表 6-6 所示。

表 6-6　端入数据寄存器 GPIOx_IDR

31～16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0

输入数据寄存器一共有 32 位，高 16 位保留，低 16 位的每 1 位对应一个不同的 I/O 引脚。该位的不同取值反映了引脚输入的电平情况：

- 0：对应引脚输入低电平
- 1：对应引脚输入高电平

例如：如果读到 GPIOA_IDR 寄存器的 bit7 为 1，表示 PA7 引脚输入低电平。

注意：输入数据寄存器是只读模式，不能通过软件写入。并且当引脚设置为输出模式时，也可以通过该寄存器读取该引脚当前的电平状态。

6. 输出数据寄存器（GPIOx_ODR）

该寄存器用于控制 GPIOx 端口各引脚的输出电平，仅用于输出模式。寄存器各位的描述见表 6-7 所示。

表 6-7　端出数据寄存器 GPIOx_ODR

31～16															
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0

(8) GPIOx_LCKR：锁定寄存器
GPIOx_AFRL (7)
复用 I/O 功能寄存器

输出数据寄存器一共有 32 位，高 16 位保留，低 16 位的每 1 位对应一个不同的 I/O 引脚。该位的不同取值反映了输出引脚的电平情况：

- 0：对应引脚输出低电平
- 1：对应引脚输出高电平

例如：向 GPIOA_ODR 寄存器的 bit5 写入 1，表示 PA5 引脚输出~~低~~高电平。

输出数据寄存器在使用时要注意以下两点：

(1) 设置某一个引脚的输出电平时，只修改该引脚对应的位，而不要修改其他的位，可以利用 C 语言中的位操作完成。

(2) 设置输出数据寄存器某一个位为 1，并不代表该位对应的引脚一定输出高电平。有可能该引脚在外围电路的作用下拉到了低电平。因此读取 GPIOx_ODR 寄存器的内容并不一定反映了引脚的真实电平情况，而应该通过读取 GPIOx_IDR 寄存器来获取引脚的真实电平情况。

7. 置位/复位寄存器（GPIOx_BSRR）

该寄存器用于置位或复位 GPIOx 端口的各引脚，功能和 GPIOA_ODR 寄存器类似，都可以控制引脚输出高/低电平。寄存器各位的描述见表 6-8 所示。

表 6-8　置位/复位数据寄存器 GPIOx_~~ODR~~BSRR

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
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

置位/复位寄存器一共有 32 位，分成高 16 位和低 16 位两组：

- 高 16 位用于控制对应的 I/O 引脚输出低电平：写入 1 对应引脚输出低电平，写入 0，没有任何作用。
- 低 16 位用于控制对应的 I/O 引脚输出高电平：写入 1 对应引脚输出高电平，写入 0，没有任何作用。

和 GPIOx_ODR 寄存器相比，使用 GPIOx_BSRR 寄存器控制输出引脚有 2 个优势：

(1) GPIOx_BSRR 寄存器的操作是原子操作，不会被中断所打断；

(2) 在控制多个引脚同时输出高/低电平时更加方便。

例如，要求 PA0 输出低电平，PA15 输出高电平。

方法一：使用 GPIOx_ODR 寄存器实现

如果采用 GPIOx_ODR 寄存器实现时，要按照“保存 → 修改 →写入”的方式进行。先读取 GPIOA_ODR 寄存器的值，保存在预先定义的变量中，然后利用位操作将读取数据的 bit0 置为 0，bit15 置为 1，最后再重新写入 GPIOA_ODR 寄存器。

方法一：使用 GPIOx_BSRR 寄存器实现

如果采用 GPIOx_BSRR 寄存器实现时，则可以直接利用位操作，将 GPIOA_BSRR 寄存器的 bit16 和 bit15 同时置为 1，或者直接向 GPIOx_BSRR 寄存器写入 0x00018000。

注意：利用置位/复位寄存器输出的高/低电平，同样会受到外围电路的影响，因此也需要通过读取 GPIOx_IDR 寄存器来获取引脚的真实电平情况。

6.2.2 利用指针访问单个寄存器

在 C 语言程序设计中，要访问某一个变量，可以通过变量的名称或者变量的地址。同样的，要访问 GPIO 模块对应的寄存器，必须要知道这些寄存器的地址。

STM32 微控制器是一个 32 位的处理器，可以寻址的内存空间容量为 4GB，对应的 16 进制地址为：0x00000000～0xFFFFFFFF，片内的 Flash、SRAM 及外设对应的寄存器均在这个 4GB 的地址空间中。通过查阅芯片的参考手册，GPIO 模块对应的寄存器组所占用的地址空间如表 6-9 所示。

表 6-9　GPIO 模块寄存器的地址空间

端口号	寄存器组地址范围	备注
GPIOA	0x4002 0000 ～ 0x4002 03FF	每组端口占用了 0x000～0x3FF 共 1K 字节的存储空间，而实际每组端口由 10 个 32 位的寄存器控制，只需要 40 个字节的存储空间，多余的存储空间预留给芯片升级使用。
GPIOB	0x4002 0400 ～ 0x4002 07FF	
GPIOC	0x4002 0800 ～ 0x4002 0BFF	
GPIOD	0x4002 0C00 ～ 0x4002 0FFF	
GPIOE	0x4002 1000 ～ 0x4002 13FF	
GPIOH	0x4002 1C00 ～ 0x4002 1FFF	

从表 6-9 可以看到，端口 GPIOA 对应的地址空间为：0x40020000 ～ 0x400203FF，起始地址为 0x40020000。该端口由 10 个寄存器控制，每个寄存器 32 位，占用 40 个字节。这 10 个寄存器的起始地址以及相对于端口 GPIOA 地址（0x40020000）的偏移量如表 6-10 所示。

表 6-10　GPIO 模块寄存器的地址空间

寄存器名称	寄存器起始地址	地址偏移量
GPIOA_MODER	0x4002 0000	0x00
GPIOA_OTYPER	0x4002 0004	0x04
GPIOA_OSPEEDR	0x4002 0008	0x08
GPIOA_PUPDR	0x4002 000C	0x0C
GPIOA_IDR	0x4002 0010	0x10
GPIOA_ODR	0x4002 0014	0x14
GPIOA_BSRR	0x4002 0018	0x18
GPIOA_LCKR	0x4002 001C	0x1C
GPIOA_AFRL	0x4002 0020	0x20
GPIOA_AFRH	0x4002 0024	0x24

知道寄存器的地址后，还无法直接访问寄存器。对于 C 语言的编译器而言，这些地址只是单纯地表示了一个 16 进制的常数，无法识别为地址，还需要利用强制类型转换，将这些地址值转换为指针的形式，再利用指针的解引用来访问。

以模式寄存器 GPIOA_MODER 为例，其定义如下：

```
#define GPIOA_MODER   *(volatile unsigned int *)(0x40020000UL)
```

➡ **定义解析**

① 0x40020000 表示模式寄存器 GPIOA_MODER 的起始地址，后缀 UL 表示无符号长整型。由于常数在 C 语言中默认为 signed int 型，最高位当做符号位，而地址值不具备符号特性，因此用 UL 做一个强制类型转换，将 signed int 转为 unsigned long。在 32 位处理器中，unsigned int 和 unsigned long 都是 4 个字节，因此使用 U 或者 UL 做后缀都可以。

② 在地址之前加上（unsigned int *）修饰，表示将该地址强制转换为一个指向无符号整型数据的指针。这里要注意一个概念：**指针的基类型**。指针的基类型表示指针所指向数据的类型，这个类型决定了指针的访问范围，即从指针所指向的地址开始，可以访问多少个字节的内存单元。如果指针指向 char 类型，表明只访问当前地址这一个字节的内存单元；如果指针指向 int 类型，表明从当前地址开始，可以访问 4 个字节的内存单元；如果指针指向 struct 类型，表明从当前地址开始，可以访问该结构体所占有的全部内存单元。由于 GPIOA_MODER 寄存器是 32 位的，实际占用的地址空间为：0x40020000～0x40020003，因此需要用 unsign int *类型的指针来指向。这样，在访问 0x40020000 这个地址时，将访问从 0x40020000 地址开始的 4 字节内存单元。如果是 8 位的寄存器，则应该转换为 unsiged char *类型的指针。

③ volatile 关键字表示每次读取该内存单元的数据时都要到内存单元处读取，而不是读取它在内存中的备份值。由于控制外设的寄存器需要与外围电路交互，这些寄存器中存放的数据受外围电路影响，如输入数据寄存器 GPIOx_IDR 的值就由外围电路决定。

我们通过一个程序片段来说明这个问题：

```
while ( ( unsigned char * ) 0x20 )
```

在使用 while 循环时，系统可能不会真正读取 0x20 单元的值，而是使用第一次读出的值。这样，可能导致 while 循环变成一个死循环。用了 volatile 修饰后，则要求每次都去读 0x20 单元的实际内容。

④ 最外面的*号表示指针的解引用，即访问指针所指向地址的内容。由于指针的类型是 unsigned int *，因此表示从内存地址 0x40020000 开始的连续 4 字节内存单元中读取数据或者写入数据。此时，*(volatile unsigned int *)(0x40020000UL)这个整体就相当于一个变量，我们可以对它进行读/写操作。

在 C 语言中也有类型的用法：在定义 int a; int *p = &a 后，*p 就和变量 a 等价，相当于变量 a 的一个别名，对*p 的操作就是对变量 a 的操作。

⑤ 使用*(volatile unsigned int *)(0x40020000UL)的方式来进行读/写操作显得比较繁琐，用户需要记忆该部分内存单元的起始地址，程序可读性较低，使用不方便。如同利用变量名来访问变量一样，我们可以根据该部分内存单元所完成的功能取一个别名，这个别名就是我们前面介绍的寄存器名。本质上，这个别名可以采用任何形式的字母组合来定义，比如 UESTC 等。之所以采用寄存器名称作为别名，是为了表明该部分内存单元的功能，并方便用户记忆。

经过上述的转换过程，我们就可以通过寄存器名称来访问对应的内存单元，因此下列代码表示：

```
GPIOA_MODER = 0x00000003;   // 向 GPIOA_MODER 寄存器中写入 0x03
unsigned int  a ;           // 定义无符号整型变量 a
a = GPIOA_MODER;            // 读取 GPIOA_MODER 寄存器的内容，并存入变量 a
```

下面我们在 MDK 开发环境中，采用寄存器的方式来编写一个控制 Nucleo 开发板指示灯 LD2 的程序。具体操作步骤如下：

步骤一：新建工程

① 在桌面上新建一个 Register 文件夹；

② 启动 MDK 软件，在 MDK 主界面的菜单栏：Project -> New uVsion Project，选择新建的 Register 文件夹，输入工程名：LED。如图 6-2 中序号 1 所示；

③ 选择目标 MCU，如图 6-2 中序号 2 所示；

④ 在 RTE 环境中勾选 CMSIS：CORE 和 Device：Startup 两个组件，如图 6-2 中序号 3 所示。

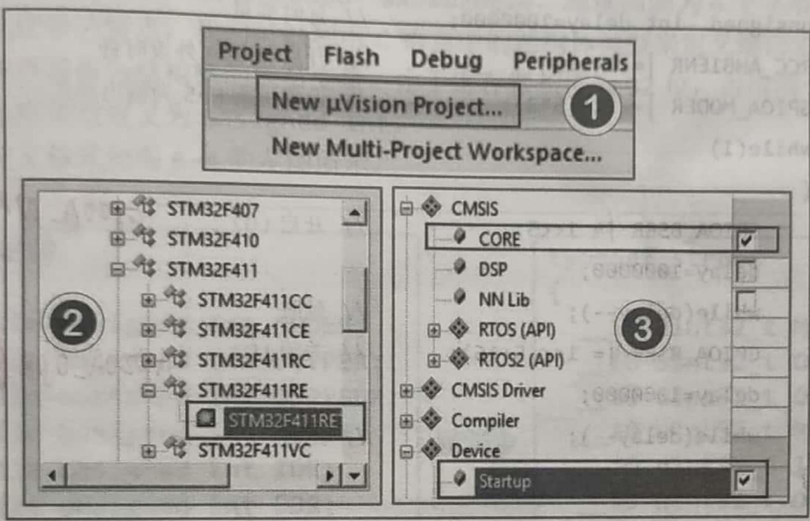


图 6-2　新建工程

步骤二：新建源文件

① 工程窗口 → 右键 Source Group1 → Add New to Group 'Source Group1'

② 选择 C File 类型文件，输入文件名为 main.c，利用"Add"按钮加入到工程中。

步骤二的关键操作步骤如图 6-3 所示。

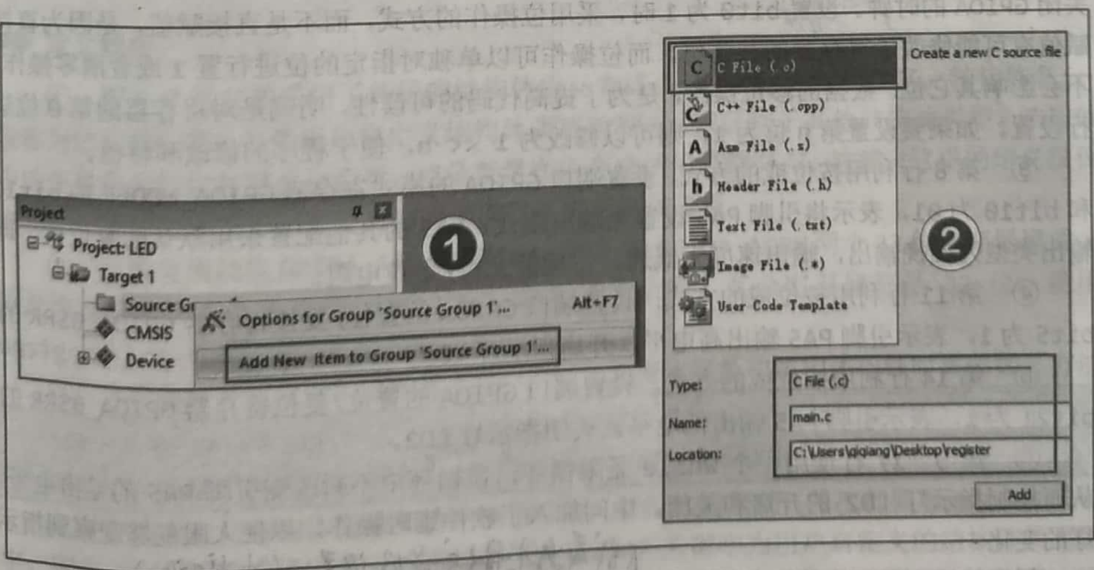


图 6-3　新建源文件

步骤三：编写代码并下载验证

打开 main.c 文件，编写程序。采用寄存器方式控制指示灯的程序见程序清单 6-1。

程序清单 6-1　采用寄存器方式控制指示灯

```
1.  #define GPIOA_MODER     *(unsigned  int *)(0x40020000UL)
2.  #define GPIOA_BSRR      *(unsigned  int *)(0x40020018UL)
3.  #define RCC_AHB1ENR     *(unsigned  int *)(0x40023830UL)
4.  int main()
5.  {
6.      unsigned  int delay=1000000;       // 延时变量
7.      RCC_AHB1ENR |= 1<<0;               // 开启 GPIOA 外设时钟
8.      GPIOA_MODER |= 1<<(5*2);           // 设置引脚 PA5 为输出模式
9.      while(1)
10.     {
11.         GPIOA_BSRR |= 1<<5;            // 开启 LD2
12.         delay=1000000;
13.         while(delay--);                // 延时
14.         GPIOA_BSRR |= 1<<(5+16);       // 关闭 LD2
15.         delay=1000000;
16.         while(delay--);                // 延时
17.     }
18. }
```

程序解析

① 第 1～3 行采用指针和宏定义的方式，定义了寄存器的别名。相关寄存器的地址可以查找芯片的参考手册。

② 第 7 行用于开启端口 GPIOA 的时钟。端口 GPIOA 是挂接在高速系统总线 AHB1 上，RCC_AHB1ENR 寄存器的 bit0 控制 GPIOA 的时钟：为 1 表示开启 GPIOA 的时钟，为 0 表示关闭 GPIOA 的时钟。设置 bit0 为 1 时，采用位操作的方式，而不是直接赋值，是因为直接赋值有可能修改了寄存器的其它位，而位操作可以单独对指定的位进行置 1 或者清零操作，不会影响其它位。数据的移位操作，是为了提高代码的可读性，明确是对寄存器的第 0 位进行设置。如果要设置第 n 位为 1，则可以修改为 1 << n，便于程序的修改和移植。

③ 第 8 行利用按位或的方式，设置端口 GPIOA 的模式寄存器 GPIOA_MODER 的 bit11 和 bit10 为 01，表示将引脚 PA5 设置为输出模式。PA5 的其他配置采用默认的复位值，即输出类型为推挽输出，输出速度为低速，不使能上拉/下拉电阻。

④ 第 11 行利用按位或的方式，设置端口 GPIOA 的置位/复位寄存器 GPIOA_BSRR 的 bit5 为 1，表示引脚 PA5 输出高电平，开启指示灯 LD2。

⑤ 第 14 行利用按位或的方式，设置端口 GPIOA 的置位/复位寄存器 GPIOA_BSRR 的 bit21 为 1，表示引脚 PA5 输出低电平，关闭指示灯 LD2。

⑥ 第 9～17 行使用一个 while 无限循环，在循环中不断改变引脚 PA5 的输出电平，从而控制指示灯 LD2 的开启和关闭。中间加入了软件延时操作，以便人眼能够观察到指示灯的变化。

程序编写完成后，进行编译链接，并通过 ST-Link 仿真器下载到 Nucleo 开发板上，

观察指示灯 LD2 的闪烁情况，用户还可以通过修改延时变量的值来改变 LD2 的闪烁频率。

6.2.3 利用结构体指针访问寄存器组

从上一小节，我们看到，使用指针访问寄存器时，需要重新定义每一个寄存器，并且一次定义只能访问一个寄存器，操作比较繁琐。对于某一个外设而言，与之相关的寄存器一般都有多个，能不能一次定义就能够访问与该外设相关的全部寄存器呢？

解决这个问题可以分成两个步骤：

步骤一：定义数据结构

以 GPIO 外设为例，从表 6-10 中可以发现，与端口 GPIOA 相关的 10 个寄存器在内存空间所占的地址是连续的：0x40020000～0x40020024，地址间隔为 4 个字节。

基于地址连续的特点，我们可以利用 C 语言中的结构体类型来定义端口 GPIOA 的全部寄存器，将寄存器作为结构体的成员变量。由于寄存器长度为 32 位，占用四个字节，因此成员变量的数据类型定义为 unsigned int。

具体的定义格式如图 6-4 的左图所示。

```
struct GPIO
{
  volatile unsigned int MODER;
  volatile unsigned int OTYPER;
  volatile unsigned int OSPEEDR;
  volatile unsigned int PUPDR;
  volatile unsigned int IDR;
  volatile unsigned int ODR;
  volatile unsigned int BSRR;
  volatile unsigned int LCKR;
  volatile unsigned int AFR[2];
};
```

```
typedef struct
{
   __IO uint32_t MODER;
   __IO uint32_t OTYPER;
   __IO uint32_t OSPEEDR;
   __IO uint32_t PUPDR;
   __IO uint32_t IDR;
   __IO uint32_t ODR;
   __IO uint32_t BSRR;
   __IO uint32_t LCKR;
   __IO uint32_t AFR[2];
} GPIO_TypeDef;
```

图 6-4 利用结构体定义寄存器组

定义解析

① 图 6-7 的左图采用了传统的结构体定义方式：struct + 结构体名。结构体名一般称为结构体标签，作为用户自定义结构体类型的标志，以区别于其他结构体类型。结构体中的变量称为结构体的成员变量，成员变量在内存空间中连续分布。**注意**：这里的定义仅仅声明了一种结构体类型，并没有定义该类型下的结构体变量。

② 引脚复用功能 GPIOA_AFR 由 GPIOA_AFRL 和 GPIOA_AFRH 两个 32 位寄存器组成，因此定义为包含两个元素的 unsigned int 类型数组。其余寄存器都是 32 位，使用 unsigned int 类型定义。

③ 采用传统方式定义的结构体类型，在定义结构体变量和结构体指针时按照如下方法：

```
struct GPIO  GPIOA;       // 定义结构体变量
struct GPIO *pGPIOA;      // 定义指向结构体变量的指针
```

这种方式在定义结构体变量和结构体指针时，显得比较繁琐，每次都要写 struct 关键字。因此，在实际的编程应用中，往往利用 typedef 关键字为用户自定义的结构体类型取一个别名，如图 6-7 的右图所示：用 GPIO_TypeDef 作为整个结构体类型的一个别名。这

样，在定义该结构体类型下的结构体变量和结构体指针时，就可以简化为：

```
GPIO_TypeDef GPIOA;        // 定义结构体变量
GPIO_TypeDef *pGPIOA;      // 定义指向结构体变量的指针
```

④ 图 6-7 的右图定义中，__IO 就是 volatile 关键字的一个别名。使用__IO 来替换 volatile，可以增加程序的可读性，表明这里定义的变量是一个可读可写的硬件寄存器。具体定义如下：

```
#define     __IO     volatile
```

⑤ 图 6-7 的右图定义中，uint32_t 是 unsigned int 类型的一个别名，使用 typedef 进行类型重定义。uint32_t 和 unsigned int 定义的变量都表示无符号整型变量。对标准数据类型取别名的方式可以提高程序的可读性，表明该变量占用 4 字节 32 位的内存空间。具体定义如下：

```
typedef unsigned  int uint32_t;
```

步骤二：强制类型转换

采用结构体方式完成端口 GPIOA 的定义后，还需要将 GPIOA 的起始地址，也就是第一个寄存器 GPIOA_MODER 的地址 0x40020000，采用强制类型转换的方法，转换为指向结构体的指针。

端口 GPIOA 的定义如下：

```
#define GPIOA    ((GPIO_TypeDef *) 0x40020000UL)
```

➡ 定义解析

① 地址 0x40020000 表示端口 GPIOA 的起始地址；

② 利用强制类型转换，将端口 GPIOA 的起始地址 0x40020000 转换为 GPIO_TypeDef 类型的结构体指针，表示在访问 0x40020000 这个地址时，可以从该地址开始，连续访问 40 个字节（具体的字节数由结构体中定义的成员变量的数量和类型决定）的内存单元。访问的方式是结构体指针加成员变量：

```
(GPIO_TypeDef *)(0x40020000UL) -> MODER
```

③ 为了提高程序的可读性，利用宏定义为该指针取一个别名，一般用外设的名称作为别名。这样，在访问该外设对应的寄存器时，就可以采用结构体指针加成员变量的方式。

示例如下：

```
GPIOA -> MODER   等价于 *(volitile unsigned  int *)(0x40020000UL)
GPIOA -> OTYPER  等价于 *(volitile unsigned  int *)(0x40020004UL)
```

利用同样的方式可以将 GPIO 外设所有的端口进行定义，这样就访问 GPIO 外设的全部寄存器。

```
#define GPIOB          ((GPIO_TypeDef *) 0x40020400UL)
#define GPIOC          ((GPIO_TypeDef *) 0x40020800UL)
#define GPIOD          ((GPIO_TypeDef *) 0x40020C00UL)
#define GPIOE          ((GPIO_TypeDef *) 0x40021000UL)
#define GPIOH          ((GPIO_TypeDef *) 0x40021C00UL)
```

为了便于用户使用，ST 公司将 STM32 微控制器片内所有外设的寄存器都采用上述的方法进行了定义，以.h 文件的格式提供给用户。用户在使用时，只需包含该头文件，就可以

通过外设的结构体指针访问外设的相关寄存器。以 STM32F411 系列微控制器为例，对应的.h 文件为 stm32f411xe.h。

采用结构体指针方式控制指示灯的程序见程序清单 6-2。

程序清单 6-2 采用结构体指针方式控制指示灯

stm32f411xe.h"

```
#include "stm32f4xx.h"            // 头文件中包含所有外设寄存器的定义
int main()
{
  uint32_t delay=1000000;         // 延时变量
  RCC->AHB1ENR |= 1<<0;           // 开启 GPIOA 外设时钟
  GPIOA->MODER |= 1<<(5*2);       // 设置引脚 PA5 为输出模式
  while(1)
  {
     GPIOA->BSRR |= 1<<5;         // 开启 LD2
     delay=1000000;
     while(delay--);              // 延时
     GPIOA->BSRR |= 1<<(5+16);    // 关闭 LD2
     delay=1000000;
     while(delay--);              // 延时
  }
}
```

程序解析

程序清单 6-2 和程序清单 6-1 的功能一样，均是通过控制 PA5 引脚输出高/低电平来实现指示灯的开启和关闭，只是在访问外设寄存器时，采用了结构体指针加成员变量的方式。

6.3 基于 HAL 库方式控制 GPIO

6.3.1 库函数设计思想

从 6.2 节的介绍可以看到，采用寄存器方式进行 GPIO 外设控制时，需要掌握每个寄存器的用途及其每一位的含义。由于 STM32 微控制器片内集成了众多外设，如果都采用寄存器方式编程，其开发难度可想而知。因此在产业界，一般都采用库函数方式进行开发。

那么库函数的设计思想是什么呢？从前面的介绍中，我们可以看到，对外设的配置实质就是向相关的寄存器中写入配置参数。如果寄存器数量较多，这个配置过程就比较繁琐。

如何简化这个配置过程呢？我们可以考虑在内存中分配一块存储区域，并对其进行初始化，初始化的值就是我们需要写入寄存器的配置参数。然后利用一个初始化函数，将这些参数复制到外设对应的寄存器中，就可以完成对外设的配置，如图 6-5 所示。

在这个配置过程中，关键的步骤有 2 个：一个是构造存储区域，也就是设计一个数据类型，该数据类型用来存放配置到外设寄存器的参数(h文件)；第二个是编写初始化函数，该函数的功能是将数据类型中的参数写入到对应的外设寄存器中。(.c文件)

那么，如何实现数据类型和初始化函数呢？这正是库函数所完成的工作之一。ST 公司基于上述的设计思想，针对不同的外设构造了不同的数据类型以及操作这些数据类型的接口函数。开发人员只需要用这些数据类型定义对应的变量，并写入相应的配置参数，然后调用

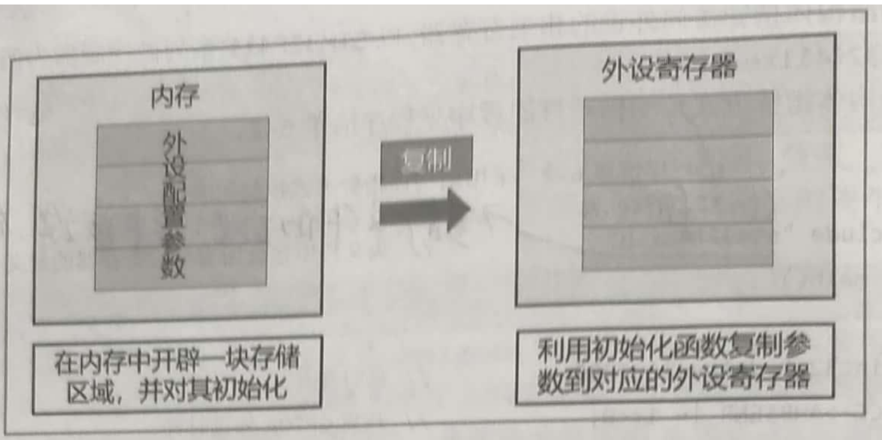


图 6-5　寄存器初始化示意图

初始化函数完成外设的初始化，调用控制函数控制外设的行为，调用状态函数判断外设的状态。整个开发工作就转变为了解数据类型的定义，以及掌握接口函数的用法，而不需要掌握外设寄存器的功能和每一位的含义，以及直接访问外设寄存器。

基于库函数方式开发嵌入式系统具有如下特点：

- 程序编写直接面向库函数。编程者只需要了解库函数中相关接口函数的功能，并按照要求传入参数，利用返回值完成操作即可，不需要过多了解底层硬件。
- 由于考虑了程序的稳健性、扩充性和可移植性，程序代码比较繁琐和臃肿，执行效率较低。
- 开发难度较小，开发周期较短，后期的维护升级、以及硬件平台的移植等工作量较小。

下面我们以 GPIO 外设为例，来深入了解一下库函数的具体实现。

与 GPIO 外设控制相关的数据类型定义和接口函数定义位于 HAL 库的源文件 stm32f4xx_hal_gpio.h 以及 stm32f4xx_hal_gpio.c 中。该文件可以在下载的 MCU 软件包中找到，文件路径为：STM32Cube_FW_F4_Vx.x\Drivers\STM32F4xx_HAL_Driver，也可以直接打开 MDK 的工程查看。

stm32f4xx_hal_gpio.h 文件中主要是数据类型的定义以及提供给用户调用的接口函数声明。stm32f4xx_hal_gpio.c 文件中主要是接口函数的具体实现。

6.3.2　GPIO 外设数据类型定义

GPIO 外设的数据类型定义主要包括引脚初始化数据定义、引脚电平状态定义以及引脚所属端口定义。

1. 引脚初始化数据定义（GPIO_InitTypeDef）

GPIO 引脚的初始化数据采用结构体类型定义，具体定义见程序清单 6-3。

程序清单 6-3　引脚初始化数据定义

```
typedef struct
{
  uint32_t Pin;   // 指定需要配置的 GPIO 引脚，该参数可以是 GPIO_pins 的值之一
  uint32_t Mode;  // 指定所选引脚的工作模式，该参数可以是 GPIO_mode 的值之一
  uint32_t Pull;  // 指定所选引脚的上拉/下拉电阻，该参数可以是 GPIO_pull 的值之一
  uint32_t Speed; // 指定所选引脚的速度，该参数可以是 GPIO_speed 的值之一
```

```
    // 将外设连接至所选择的引脚，该参数可以是 Alternate_function_selection 的值之一
    uint32_t Alternate;
}GPIO_InitTypeDef;
```

定义解析

引脚初始化结构体类型中定义引脚的序号、工作模式、上拉/下拉电阻、输出速度和复用功能等成员变量，这些变量作为引脚的配置参数，由用户配置后，再调用引脚初始化函数写入到对应的硬件寄存器中。

这些成员变量的取值没有使用单纯的数值，而是采用宏定义的方式重新封装，以提高程序的可读性，具体的宏定义参见表 6-11～表 6-14。

- Pin

Pin 表示引脚定义，按照该引脚在所属端口的位置来确定它的常量值，以 PA5 为例，属于端口 GPIOA 的第 5 脚。该端口有 16 个引脚，对应 16 位二进制数 0000 0000 0000 0000。第 5 脚对应这个二进制数的 bit5，将 bit5 置为 1，对应的常量值即为 0x0020。加上 uint16_t 的修饰后，表示将 0x0020 进行强制类型转换，转换为无符号短整型 unsigned short int，占用 2 个字节 16 位。为了便于用户使用，提高程序的可读性，又采用宏定义的方式对这个常数取了一个别名 GPIO_PIN_5。采用这种方式的定义，可以让属于同一个端口且配置属性相同的引脚通过按位或的形式同时配置。

成员变量 Pin 的取值范围如表 6-11 所示。

表 6-11　引脚定义 GPIO_pins

宏常量定义	含义	常数值
GPIO_PIN_0	选择该端口的引脚 0	((uint16_t)0x0001)
GPIO_PIN_1	选择该端口的引脚 1	((uint16_t)0x0002)
GPIO_PIN_2	选择该端口的引脚 2	((uint16_t)0x0004)
GPIO_PIN_3	选择该端口的引脚 3	((uint16_t)0x0008)
GPIO_PIN_4	选择该端口的引脚 4	((uint16_t)0x0010)
GPIO_PIN_5	选择该端口的引脚 5	((uint16_t)0x0020)
GPIO_PIN_6	选择该端口的引脚 6	((uint16_t)0x0040)
GPIO_PIN_7	选择该端口的引脚 7	((uint16_t)0x0080)
GPIO_PIN_8	选择该端口的引脚 8	((uint16_t)0x0100)
GPIO_PIN_9	选择该端口的引脚 9	((uint16_t)0x0200)
GPIO_PIN_10	选择该端口的引脚 10	((uint16_t)0x0400)
GPIO_PIN_11	选择该端口的引脚 11	((uint16_t)0x0800)
GPIO_PIN_12	选择该端口的引脚 12	((uint16_t)0x1000)
GPIO_PIN_13	选择该端口的引脚 13	((uint16_t)0x2000)
GPIO_PIN_14	选择该端口的引脚 14	((uint16_t)0x4000)
GPIO_PIN_15	选择该端口的引脚 15	((uint16_t)0x8000)
GPIO_PIN_All	选择该端口所有引脚	((uint16_t)0xFFFF)

- Mode

Mode 表示引脚工作模式。GPIO 外设的每一个引脚具备 4 种工作模式，这里根据开漏模

式和推挽模式又将输出模式和复用模式进行了二次划分，构造出 6 种工作模式的宏定义。成员变量 Mode 的取值范围如表 6-12 所示。

表 6-12　引脚工作模式 GPIO_mode

宏常量定义	含义
GPIO_MODE_INPUT	浮空输入模式
GPIO_MODE_OUTPUT_PP	推挽输出模式
GPIO_MODE_OUTPUT_OD	开漏输出模式
GPIO_MODE_AF_PP	复用功能推挽模式
GPIO_MODE_AF_OD	复用功能开漏模式
GPIO_MODE_ANALOG	模拟模式

- Pull

Pull 表示引脚上拉/下拉电阻，取值范围如表 6-13 所示。

表 6-13　引脚上拉/下拉 GPIO_pull

宏常量定义	含义
GPIO_NOPULL	没有上拉或下拉电阻激活
GPIO_PULLUP	上拉电阻激活
GPIO_PULLDOWN	下拉电阻激活

- Speed

Speed 表示引脚输出速度，取值范围如表 6-14 所示。

表 6-14　引脚输出速度 GPIO_speed

宏常量定义	含义
GPIO_SPEED_FREQ_LOW	引脚输出速度 2MHz
GPIO_SPEED_FREQ_MEDIUM	引脚输出速度 12.5MHz～50MHz
#define GPIO_SPEED_FREQ_HIGH	引脚输出速度 25MHz～100MHz
GPIO_SPEED_FREQ_VERY_HIGH	引脚输出速度 50MHz～200MHz

- Alternate

Alternate 表示引脚的复用功能。由于不同型号的 STM32 微控制器片内集成的外设不同，因此该成员变量的取值范围由芯片型号决定。该成员变量的取值一般通过 CubeMX 软件分配，不需要用户手动设置。感兴趣的读者可以查阅 stm32f4xx_hal_gpio_ex.h 文件了解 Alternate 的取值范围。

2. 引脚电平状态定义（GPIO_PinState）

GPIO 引脚的电平状态采用枚举类型定义，具体定义见程序清单 6-4

程序清单 6-4　引脚电平状态定义

```
typedef enum
{
    GPIO_PIN_RESET = 0,     // 引脚低电平状态
```

```
    GPIO_PIN_SET                        // 引脚高电平状态
}GPIO_PinState;
```

➡ 定义解析

引脚电平状态采用了枚举类型来定义，枚举类型代表了一组整型常量的集合，与单纯使用整型常量 0 或者 1 相比，枚举类型可以提高程序的可读性。

例如，GPIO_PIN_RESET 的值为 0，就比较直观地表示向引脚输出低电平或者从引脚读到低电平；GPIO_PIN_SET 的值为 1，则表示向引脚输出高电平或者从引脚读到高电平。

3. 引脚所属端口定义（GPIO_Port）

除了在 stm32f4xx_hal_gpio.h 文件定义的 2 个数据类型以外，还有一个在配置 GPIO 引脚时常用的入口参数：引脚所属端口。该类型定义在以芯片型号命名的 h 文件中，如 STM32F411 系列对应的头文件 stm32f411xe.h。引脚所属端口是一个指向结构体类型的指针，该结构体类型定义了控制端口的寄存器组，具体的定义如图 6-4 所示。

引脚所属端口的取值范围如表 6-15 所示。

表 6-15　引脚端口 GPIO_Port

宏常量定义	含义	常数值
GPIOA	选择端口 A	((GPIO_TypeDef *) GPIOA_BASE)
GPIOB	选择端口 B	((GPIO_TypeDef *) GPIOB_BASE)
GPIOC	选择端口 C	((GPIO_TypeDef *) GPIOB_BASE)
GPIOD	选择端口 D	((GPIO_TypeDef *) GPIOD_BASE)
GPIOE	选择端口 E	((GPIO_TypeDef *) GPIOE_BASE)
GPIOH	选择端口 H	((GPIO_TypeDef *) GPIOH_BASE)

➡ 定义解析

引脚所属端口的定义主要根据各端口寄存器组的起始地址来定义，如 GPIOA_BASE 就表示控制端口 GPIOA 的寄存器组的起始地址 0x40020000。各端口寄存器组的起始地址如表 6-9 所示。在定义时，利用强制类型转换将起始地址转换为指向端口寄存器结构体类型（GPIO_TypeDef）的指针，然后利用端口号加成员变量的方式访问控制端口的寄存器组，如：GPIOA->MODER 就可以访问控制端口 GPIOA 的引脚模式寄存器。

4. 引脚配置步骤

在具体配置一个 GPIO 引脚的工作模式时，一般先利用引脚初始化结构体类型 GPIO_InitTypeDef 定义一个结构体变量，然后按照引脚的工作模式，依次对该结构体的成员变量赋值，最后调用初始化函数 HAL_GPIO_Init 将配置参数写入到对应的寄存器中。如果需要配置多个引脚，则需要多次修改结构体中成员变量的数据，重复调用初始化函数来配置。如果多个引脚属于同一个端口，且工作模式一致，则可以利用位操作同时配置。

5. 引脚配置实例

设置 PA5 和 PA15 为推挽输出模式，无上拉/下拉电阻。设置 PC13 为浮空输入模式，配置程序见程序清单 6-5。

程序清单 6-5　引脚初始化程序

```
GPIO_InitTypeDef GPIO_InitStruct = {0}; // 初始化为0
// 配置引脚 PA5 和 PA15：推挽输出，无上拉/下拉，输出低速
```

```
3.  GPIO_InitStruct.Pin   = GPIO_PIN_5|GPIO_PIN_15;
4.  GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
5.  GPIO_InitStruct.Pull  = GPIO_NOPULL;
6.  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
7.  HAL_GPIO_Init(GPIOA, &GPIO_InitStruct); // 初始化函数
8.  // 配置引脚 PC13：浮空输入，无上拉/下拉
9.  GPIO_InitStruct.Pin = GPIO_PIN_13;
10. GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
11. GPIO_InitStruct.Pull = GPIO_NOPULL;
12. HAL_GPIO_Init(GPIOC, &GPIO_InitStruct); // 初始化函数
```

程序解析

① 第 1 行定义了引脚初始化结构体变量 GPIO_InitStruct，并将成员变量全部初始化为 0；

② 第 3~6 行按照引脚的配置要求，向对应的成员变量写入初始化参数。

③ 第 7 行调用初始化函数 HAL_GPIO_Init 将初始化参数写入对应的硬件寄存器。由于 PA5 和 PA15 同属端口 GPIOA，且配置参数相同，因此可以采用按位或的方式同时配置，这样可以减少调用引脚初始化函数的次数。

④ 第 9~12 行配置 PC13 引脚，重新修改各成员变量的参数，然后调用初始化函数 HAL_GPIO_Init 配置引脚。

6.3.3 GPIO 外设接口函数定义

定义了 GPIO 外设的数据类型后，还需要设计相应的接口函数，来完成 GPIO 外设的初始化和相关控制。与 GPIO 外设控制相关的接口函数一共有 6 个：

1. 引脚初始化函数（HAL_GPIO_Init）

引脚初始化函数 HAL_GPIO_Init 主要用于引脚的初始化，将用户设定的引脚参数写入到对应的硬件寄存器。具体描述如表 6-16 所示。

表 6-16 引脚初始化函数

接口函数：HAL_GPIO_Init	
函数原型	void HAL_GPIO_Init (GPIO_TypeDef * GPIOx, GPIO_InitTypeDef * GPIO_Init)
功能描述	引脚初始化
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA~GPIOK
入口参数 2	GPIO_Init：指向 GPIO_InitTypeDef 的结构体指针，包含指定引脚的配置参数
返回值	无
注意事项	该函数由 CubeMX 自动生成

2. 引脚复位函数（HAL_GPIO_DeInit）

引脚复位函数 HAL_GPIO_DeInit 主要用于引脚的配置状态初始化至复位值。具体描述如表 6-17 所示。

表 6-17 引脚复位函数

接口函数：HAL_GPIO_DeInit

函数原型	void HAL_GPIO_DeInit (GPIO_TypeDef * GPIOx, uint32_t GPIO_Pin)
功能描述	复位引脚到初始状态
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA～GPIOK
入口参数 2	GPIO_Pin：引脚号，范围是 GPIO_PIN_0～GPIO_PIN_15
返回值	无
注意事项	~~该函数由 CubeMX 自动生成~~

3. 读取引脚函数（HAL_ReadPin）

读取引脚函数 HAL_ReadPin 主要用于读取指定引脚的电平状态。具体描述如表 6-18 所示。

表 6-18　读取引脚函数

接口函数：HAL_ReadPin	
函数原型	GPIO_PinState HAL_GPIO_ReadPin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
功能描述	读取引脚的电平状态
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA～GPIOK
入口参数 2	GPIO_Pin：引脚号，范围是 GPIO_PIN_0～GPIO_PIN_15
返回值	GPIO_PinState：引脚电平状态，取值是以下参数之一： ● GPIO_PIN_SET 表示高电平 ● GPIO_PIN_RESET 表示低电平
注意事项	该函数由用户调用

4. 写入引脚函数（HAL_WritePin）

写入引脚函数 HAL_WritePin 主要用于设置指定引脚输出高电平或者低电平。具体描述如表 6-19 所示。

表 6-19　写入引脚函数

接口函数：HAL_WritePin	
函数原型	void HAL_GPIO_WritePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin, GPIO_PinState PinState)
功能描述	设置引脚输出高/低电平
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA～GPIOK
入口参数 2	GPIO_Pin：引脚号，范围是 GPIO_PIN_0～GPIO_PIN_15
入口参数 3	GPIO_PinState：引脚电平状态，取值是以下参数之一： ● GPIO_PIN_SET 表示高电平 ● GPIO_PIN_RESET 表示低电平
返回值	无
注意事项	该函数由用户调用

5. 引脚翻转函数（HAL_GPIO_TogglePin）

引脚翻转函数 HAL_GPIO_TogglePin 主要用于翻转引脚的状态：原来是高电平，则变为低电平；原来是低电平，则变为高电平。具体描述如表 6-20 所示。

表 6-20　引脚翻转函数

接口函数：HAL_GPIO_TogglePin	
函数原型	void HAL_GPIO_TogglePin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
功能描述	翻转引脚状态
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA～GPIOK
入口参数 2	GPIO_Pin：引脚号，范围是 GPIO_PIN_0～GPIO_PIN_15
返回值	无
注意事项	该函数由用户调用

6. 引脚锁定函数（HAL_GPIO_LockPin）

引脚锁定函数 HAL_GPIO_LockPin 主要用于锁定引脚配置寄存器。具体描述如表 6-21 所示。

表 6-21　引脚锁定函数

接口函数：HAL_GPIO_LockPin	
函数原型	HAL_StatusTypeDef HAL_GPIO_LockPin (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
功能描述	锁定引脚配置寄存器
入口参数 1	GPIOx：引脚端口号，范围是 GPIOA ~ GPIOK
入口参数 2	GPIO_Pin：引脚号，范围是 GPIO_PIN_0 ~ GPIO_PIN_15
返回值	HAL_StatusTypeDef：HAL_OK 表示引脚锁定成功，HAL_ERROR 表示引脚锁定失败
注意事项	该函数由用户调用

6.3.4　GPIO 外设接口函数源码分析

了解了 GPIO 外设定义的数据类型和接口函数之后，我们可以尝试去看一下这些接口函数是如何实现的，了解一下官方的代码编写规范，对于提高自身的编程能力有很大的帮助。下面我们以最常用的 3 个接口函数为例，分析它们的源码。

1. 读取引脚函数（HAL_ReadPin）

该函数的源码如程序清单 6-6 所示。

程序清单 6-6　HAL_ReadPin 函数源码

```
/*
  * @brief  Reads the specified input port pin.
  * @param  GPIOx where x can be (A..K) to select the GPIO peripheral for
  *                STM32F429X device or
                   x can be (A..I) to select the GPIO peripheral for
```

```
6.  *          STM32F40XX and STM32F427X devices.
7.  * @param  GPIO_Pin specifies the port bit to read.
8.  *         This parameter can be GPIO_PIN_x where x can be (0..15).
9.  * @retval The input port pin value.
10. */
11. GPIO_PinState HAL_GPIO_ReadPin(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
12. {
13.   GPIO_PinState bitstatus;
14.   /* Check the parameters */
15.   assert_param(IS_GPIO_PIN(GPIO_Pin));
16.   if((GPIOx->IDR & GPIO_Pin) != (uint32_t)GPIO_PIN_RESET)
17.   {
18.     bitstatus = GPIO_PIN_SET;
19.   }
20.   else
21.   {
22.     bitstatus = GPIO_PIN_RESET;
23.   }
24.   return bitstatus;
25. }
```

程序解析

① 第1～10行是函数的说明。介绍了函数的功能(brief)、入口参数(param)和出口参数(retval)，并详细说明了入口参数的取值范围。

② 第13行定义了一个GPIO_PinState类型的枚举变量bitstatus，作为函数的返回值，返回读取的引脚电平状态，取值范围是：GPIO_PIN_SET或GPIO_PIN_RESET。

③ 第15行是参数检查，确保入口参数GPIO_Pin的值在GPIO_PIN_0和GPIO_PIN_15之间。

④ 第16～23行读取引脚状态。通过读取输入数据寄存器GPIOx_IDR的值，与入口参数GPIO_Pin做一个按位与的操作，如果结果不为0 (GPIO_PIN_RESET)，说明GPIOx_IDR对应的位是1，即对应引脚状态是高电平，因此将返回值bitstatus赋值为GPIO_PIN_SET。如果结果为0，说明GPIOx_IDR对应的位是0，即对应引脚状态是低电平，因此将返回值bitstatus赋值为GPIO_PIN_RESET。

⑤ 第24行返回读取的引脚电平状态。

2. 写入引脚函数（HAL_WritePin）

该函数的源码如程序清单6-7所示。

程序清单6-7　HAL_WritePin函数源码

```
1. /*
2.  * @brief  Sets or clears the selected data port bit.
3.  *
4.  * @note   This function uses GPIOx_BSRR register to allow atomic
5.  *         read/modify accesses. In this way, there is no risk of
```

```
6.  *          an IRQ occurring between the read and the modify access.
7.  *
8.  * @param  GPIOx where x can be (A..K) to select the GPIO peripheral for
                         STM32F429X device or
9.  *                    x can be (A..I) to select the GPIO peripheral for
10. *                    STM32F40XX and STM32F427X devices.
11. *
12. * @param  GPIO_Pin specifies the port bit to be written.
13. *         This parameter can be one of GPIO_PIN_x where x can be (0..15).
14. * @param  PinState specifies the value to be written to the selected bit.
15. *         This parameter can be one of the GPIO_PinState enum values:
16. *         @arg GPIO_PIN_RESET: to clear the port pin
17. *         @arg GPIO_PIN_SET: to set the port pin
18. * @retval None
19. */
20. void HAL_GPIO_WritePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, GPIO_PinState
                          PinState)
21. {
22.   /* Check the parameters */
23.   assert_param(IS_GPIO_PIN(GPIO_Pin));
24.   assert_param(IS_GPIO_PIN_ACTION(PinState));
25.   if(PinState != GPIO_PIN_RESET)
26.   {
27.     GPIOx->BSRR = GPIO_Pin;
28.   }
29.   else
30.   {
31.     GPIOx->BSRR = (uint32_t)GPIO_Pin << 16U;
32.   }
33. }
```

程序解析

① 第 1～19 行是函数的说明。介绍了函数的功能(brief)、入口参数(param)和出口参数(retval)，并详细说明了入口参数的取值范围。与上一个函数 HAL_ReadPin 相比，增加了函数的注意事项(note)：设置引脚输出电平是通过置位/复位寄存器 GPIOx_BSRR 完成，这个设置过程是原子操作，不会被中断所打断。

② 第 23～24 行是参数检查。利用断言语句 assert_param 检查入口参数 GPIO_Pin 的值是否在 GPIO_PIN_0 和 GPIO_PIN_15 之间，PinState 的值是否在 GPIO_PIN_RESET 和 GPIO_PIN_SET 之间。

③ 第 25～32 行设置引脚输出电平。首先判断需要设置的电平状态：如果是高电平，则对 GPIOx_BSRR 寄存器的低 16 位中的对应位写入 1；如果是低电平，则对 GPIOx_BSRR 寄存器的高 16 位中的对应位写入 1。

④ 第 27 行和 31 行在设置 GPIOx_BSRR 寄存器时，采用了直接赋值的方式，而不是位操作。这是因为 GPIOx_BSRR 寄存器写入 0 不起任何作用，因此可以直接赋值。否则还是

要采用位操作，以避免改写寄存器的其他位。

⑤ 第 31 行是设置输出低电平。由于输出低电平是由 GPIOx_BSRR 寄存器的高 16 位控制，因此对数据做了左移 16 位的操作。

3. 翻转引脚函数（HAL_TogglePin）

该函数的源码如程序清单 6-8 所示。

程序清单 6-8 HAL_TogglePin 函数源码

```
1. /*
2.   * @brief  Toggles the specified GPIO pins.
3.   * @param  GPIOx Where x can be (A..K) to select the GPIO peripheral for
4.   *                STM32F429X device or
5.   *                x can be (A..I) to select the GPIO peripheral for
6.   *                STM32F40XX and STM32F427X devices.
7.   * @param  GPIO_Pin Specifies the pins to be toggled.
8.   * @retval None
9.   */
10. void HAL_GPIO_TogglePin(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin)
11. {
12.   /* Check the parameters */
13.   assert_param(IS_GPIO_PIN(GPIO_Pin));
14.   if ((GPIOx->ODR & GPIO_Pin) == GPIO_Pin)
15.   {
16.     GPIOx->BSRR = (uint32_t)GPIO_Pin << 16U;
17.   }
18.   else
19.   {
20.     GPIOx->BSRR = GPIO_Pin;
21.   }
22. }
```

➡ 程序解析

① 第 1~9 行是函数的说明。介绍了函数的功能(brief)、入口参数(param)和出口参数(retval)，并详细说明入口参数的取值范围。

② 第 13 行是参数检查，确保入口参数 GPIO_Pin 的值在 GPIO_PIN_0 和 GPIO_PIN_15 之间。

③ 第 14~21 行翻转引脚状态。通过读取输出数据寄存器 GPIOx_ODR 的值，与入口参数 GPIO_Pin 做一个按位与的操作。如果结果等于 GPIO_Pin，说明 GPIOx_ODR 对应的位是 1，即原来引脚输出状态是高电平，需要将引脚状态翻转至低电平，因此向 GPIOx_BSRR 寄存器的高 16 位中的对应位写入 1。如果结果不等于 GPIO_Pin，说明 GPIOx_ODR 对应的位是 0，即原来引脚输出状态是低电平，需要将引脚状态翻转至高电平，因此向 GPIOx_BSRR 寄存器的低 16 位中的对应位写入 1。

6.3.5 HAL 库设计思想

在了解 GPIO 外设的数据类型和接口函数之后，我们和之前介绍的库函数设计思想相结合，可以发现 HAL 库的设计思想和面向对象的设计思想很相似：相关的数据以及跟这些数据有内在联系的操作组成的一个模块。

以 GPIO 模块的 HAL 库函数设计为例。可以将引脚看做一个对象，引脚的编号、工作模式、上拉/下拉电阻、输出速度、复用功能等都可以看作是对象的各个属性。而相关的接口函数，如引脚初始化函数 HAL_GPIO_Init，读取引脚函数 HAL_GPIO_ReadPin，写入引脚函数 HAL_GPIO_WritePin 等，可以看作是对对象进行的各种操作。

HAL 库正是采用这种类似面向对象的设计思想，将微控制器片内集成的外设作为对象，定义相关的数据类型作为对象的各种属性，设计相关的接口函数执行对对象的各种操作：初始化、控制、读取状态。这种设计思想在后续章节的学习中还会深入的介绍。

下面我们总结一下 GPIO 模块的使用步骤：

1. 使能引脚所属端口的系统总线时钟（AHB）：__HAL_RCC_GPIOx_CLK_ENABLE（x 表示端口号），该函数是时钟模块提供的接口函数，具体可以参见 stm32f4xx_hal_rcc.h 文件。

2. 利用引脚初始化类型 GPIO_InitTypeDef 定义结构体变量，根据具体需求配置 GPIO 引脚：

- 通过成员变量 Mode 配置输入、输出、模拟等模式；
- 通过成员变量 Pull 配置上拉/下拉电阻；
- 通过成员变量 Speed 配置引脚输出速度；
- 通过成员变量 Alternate 配置引脚复用模式。

3. 使用初始化函数 HAL_GPIO_Init 完成 GPIO 引脚配置，将配置参数写入对应的硬件寄存器；

4. 使用对应的控制函数完成引脚的控制：函数 HAL_GPIO_ReadPin 读取引脚状态；函数 HAL_GPIO_WritePin 设置引脚状态；函数 HAL_GPIO_TogglePin 翻转引脚状态。

从上面的使用步骤可以看到，GPIO 外设的使用过程还是比较繁琐，特别是引脚的初始化部分，需要根据需求去配置不同的成员变量，并且要了解这些配置参数的含义。一旦遇到比 GPIO 更复杂的外设，如定时器和串口，相关的寄存器数量更多，需要掌握的成员变量和配置参数将成倍增加。开发人员在使用时不得不经常查找库函数的使用手册，开发效率变得很低。

那么，有没有更好的方法来简化外设的初始化过程呢？

答案就是 CubeMX。由于外设的初始化过程，只在启动外设的时候进行一次，其他时候一般不需要修改，因此 ST 公司将外设的初始化工作交给 CubeMX 软件来完成。从第四章的学习我们可以看到，开发人员只需要在外设配置时，选择相应的参数，不需要编写任何代码，就可以自动生成外设的初始化代码。

以 GPIO 外设为例，利用 CubeMX 软件完成引脚的初始化设置后，开发人员只需要熟悉端口宏定义：GPIOx，引脚宏定义：GPIO_PIN_x，引脚电平状态：GPIO_PIN_SET 和 GPIO_PIN_RESET，以及 3 个基本的接口函数：HAL_GPIO_ReadPin、HAL_GPIO_WritePin 和 HAL_GPIO_TogglePin，就可以随心所欲地控制 GPIO 引脚。

因此将库函数和 CubeMX 软件相结合，才是真正的嵌入式开发利器。

6.4 任务实践

在第四章里，我们详细介绍了 CubeMX 软件的使用流程，如图 6-6 所示。

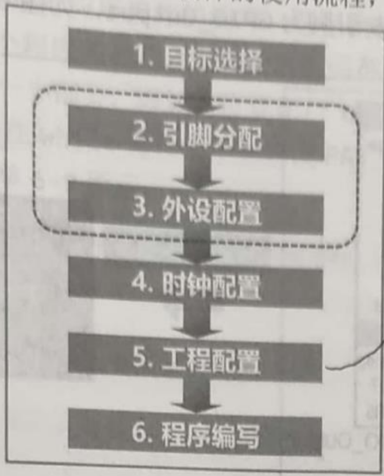


图 6-6 CubeMX 使用流程

在整个开发流程中，步骤 1～步骤 5 在 CubeMX 软件中完成，步骤 6 则在 MDK 软件中完成。对于同一个嵌入式硬件系统而言，步骤 1，步骤 4 和步骤 5 的设置一般都是相同的，而步骤 6 是在 MDK 开发环境中进行。因此，在使用 CubeMX 软件进行设计时，如果用户需要增加新的外设或者修改已有外设的配置，只需要在步骤 2 和步骤 3 中进行引脚的重新分配和外设的重新配置即可。因此我们在后续的设计任务中重点介绍步骤 2 和步骤 3 的设置过程，以及步骤 6 的应用程序编写。

【小贴士】任务学习建议

本书按照阶梯式的学习方法，设置了基础型、进阶型和挑战型等多种任务。建议读者在练习每一个任务时，单独建立一个文件夹，存放与该任务相关的工程文件。

CubeMX 的工程文件不需要每次都重新建立，只需要将之前建立的工程整体复制到新的文件夹即可。具体操作步骤如下：

1. 新建一个文件夹，文件夹的名称和具体的工程应用相关，如驱动指示灯的应用，就可以将文件夹命名为 LED；
2. 将之前建立的工程内容整个复制到新的文件夹中；
3. 删除文件夹中的 MDK-ARM 文件夹；
4. 修改 CubeMX 生成的.ioc 文件名，和新的工程名称一致，如 LED.ioc；
5. 点击重命名的 IOC 文件，启动 CubeMX 软件。按照新任务的要求完成引脚分配和外设配置后，重新生成 MDK 工程；
6. 修改工程相关设置：如仿真器设置和下载并执行代码设置等；
7. 编写应用代码。

6.4.1 基础任务：驱动指示灯

➡ **任务目标**：掌握 CubeMX 软件配置 GPIO 外设的方法

➡ **任务内容**：控制 Nucleo 开发板上的指示灯每隔 100ms 闪烁

➡ **硬件原理**：指示灯 LD2 由 PA5 引脚控制：PA5 输出高电平，开启指示灯 LD2；PA5 输出

低电平，关闭指示灯 LD2。电路原理图参见第二章的图 2-16。

任务实现

1. 引脚分配

在芯片引脚窗口中选择 PA5 引脚为 GPIO_Output，如图 6-7 所示。

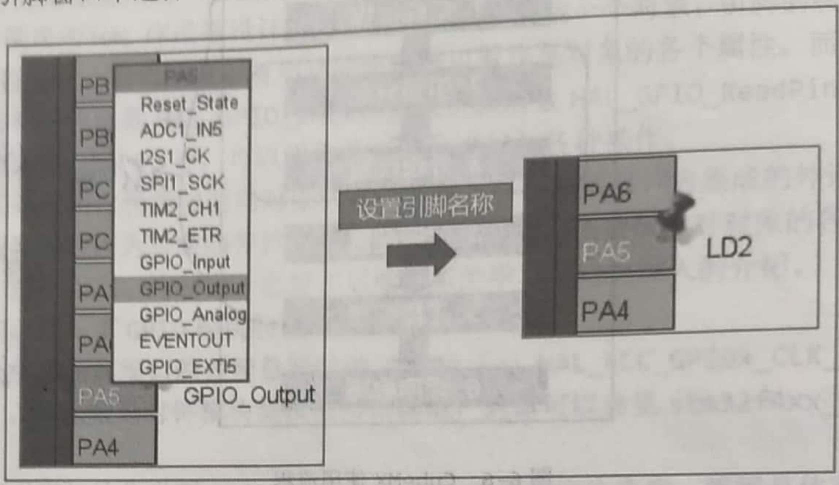


图 6-7 引脚分配

2. 外设配置

在外设设置窗口中选择：SystemCore → GPIO。在出现的引脚配置窗口中选择 GPIO 标签页。点击 PA5 引脚一栏，在展开的参数配置窗口中对 PA5 作如下配置：

- GPIO output level（初始电平）：Low（低电平）
- GPIO mode（工作模式）：Output Push Pull（推挽输出）
- GPIO Pull-up and /Pull-down（上拉/下拉电阻）：No pull-up and no pull-down（无下拉/下拉电阻）
- Maximum output speed（输出速度）：Low（低速）
- User Label（引脚名称）：LD2

具体配置如图 6-8 所示。

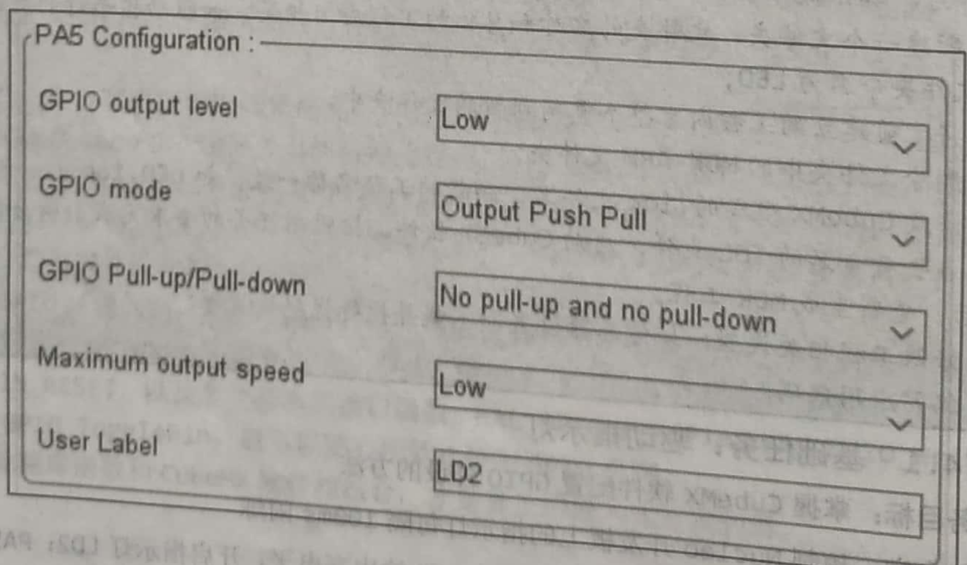


图 6-8 外设配置

注意：在设置引脚名称后，芯片引脚窗口中 PA5 引脚的名称将由 GPIO_Output 变为 LD2，如图 6-7 所示。修改引脚名称的目的主要是为了提高程序的可读性。

3. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。这里我们将完整的 main 函数呈现给大家，以便了解让大家了解整个程序的流程。大部分的代码都是由 CubeMX 生成，用户编写的指示灯控制代码只是其中的一小部分，添加的位置在/* USER CODE BEGIN 3 */和/* USER CODE END 3*/之间，主要是在 while 循环中不断修改 PA5 引脚的电平，从而达到控制 LD2 的目的。具体程序如程序清单 6-9 所示。

程序清单 6-9 驱动指示灯

```
/*
 * @brief  The application entry point.
 * @retval int
 */
int main(void)
{
  /* USER CODE BEGIN 1 */

  /* USER CODE END 1 */
  /* MCU Configuration-------------------------------------------------------*/
  /* Reset of all peripherals, Initializes the Flash interface and the Systick. */
  HAL_Init();
  /* USER CODE BEGIN Init */

  /* USER CODE END Init */
  /* Configure the system clock */
  SystemClock_Config();
  /* USER CODE BEGIN SysInit */

  /* USER CODE END SysInit */
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  /* USER CODE BEGIN 2 */

  /* USER CODE END 2 */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
  while (1)
  {
    /* USER CODE END WHILE */
    /* USER CODE BEGIN 3 */
```

```
32.     HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);  // 开启 LD2
33.     HAL_Delay(100);                                           // 延时 100ms
34.     HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);// 关闭 LD2
35.     HAL_Delay(100);                                           // 延时 100ms
36.   }
37.   /* USER CODE END 3 */
38. }
```

程序解析

① 第 1～4 行是对 main 函数的基本描述。

② 第 7～9 行是用户代码添加的位置：/* USER CODE BEGIN 1 */和/* USER CODE END 1*/之间。

③ 第 12 行是调用函数 HAL_Init()完成 HAL 库的初始化。

④ 第 13～15 行是用户代码添加的位置：/* USER CODE BEGIN Init */和/* USER CODE END Init*/，主要用于添加用户编写的初始化代码。

⑤ 第 17 行调用系统时钟初始化函数 SystemClock_Config 完成系统时钟的设置。

⑥ 第 18～20 行是用户代码添加的位置：/* USER CODE BEGIN SysInit */和/* USER CODE END SysInit*/，主要用于添加用户编写的系统初始化代码。

⑦ 第 22 行调用 GPIO 引脚初始化函数 MX_GPIO_Init 完成引脚的设置。引脚的配置参数是依据用户在 CubeMX 的引脚分配部分所做的设置。

⑧ 第 23～25 行是用户代码添加的位置：/* USER CODE BEGIN 2 */和/* USER CODE END 2*/，主要用于添加片内外设的启动代码，比如定时器的启动等。

⑨ 第 28～36 行是由 while 构成的无限循环。

⑩ 第 32 行控制 LD2 开启。通过调用写入引脚函数 HAL_GPIO_WritePin 设置 PA5 引脚输出高电平，开启指示灯 LD2。

⑪ 第 34 行控制 LD2 关闭。通过调用写入引脚函数 HAL_GPIO_WritePin 设置 PA5 引脚输出低电平，关闭指示灯 LD2。

⑫ 第 33 行和第 35 行调用延时函数函数 HAL_Delay，延时 100ms，以便形成指示灯闪烁的效果。

⑬ 写入引脚函数 HAL_GPIO_WritePin 函数入口参数 LD2_GPIO_Port 和 LD2_Pin 在 main.h 文件中已经自动进行了定义，以提高程序的可读性。

```
#define LD2_Pin GPIO_PIN_5
#define LD2_GPIO_Port GPIOA
```

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上，可以观察到绿色指示灯 LD2 每隔 100ms 闪烁一次。我们也可以通过修改延时函数 HAL_Delay 的入口参数来改变 LD2 闪烁的频率。例如，入口参数为 1000，则 LD2 每隔 1000ms 闪烁一次。

在本任务中，引脚 PA5 的初始化代码由 CubeMX 按照图 6-8 的设置自动生成，函数名为 MX_GPIO_Init，在 main 函数中自动调用。函数 MX_GPIO_Init 完成了使能引脚所在端口的时钟，配置引脚工作模式等操作，具体代码如程序清单 6-10 所示。

程序清单 6-10 自动生成的引脚初始化代码

```
/*
 * @brief GPIO Initialization Function
 * @param None
 * @retval None
 */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
  /*Configure GPIO pin : LD2_Pin */
  GPIO_InitStruct.Pin     = LD2_Pin;
  GPIO_InitStruct.Mode    = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull    = GPIO_NOPULL;
  GPIO_InitStruct.Speed   = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
}
```

同样的，系统时钟初始化代码由 CubeMX 按照用户所做的时钟设置自动生成，函数名为 SystemClock_Config，在 main 函数中自动调用。函数 SystemClock_Config 根据用户所选择的时钟源和锁相环配置参数，设置系统时钟为目标值。感兴趣的读者可以通过 MDK 工程查看源码，这里就不再赘述。

你的任务

任选 Nucleo 开发板上的 4 个 I/O 引脚控制 4 个指示灯：LED0、LED1、LED2 和 LED3。假设 PA4 控制 LED0，PB0 控制 LED1，PC1 控制 LED2，PC0 控制 LED3。请编程实现 4 个指示灯的轮流点亮：LED0 → LED1 → LED2 → LED3。点亮下一个指示灯时，上一个指示灯熄灭。

提示：在 CubeMX 软件中选择 4 个 I/O 引脚为 GPIO_Output，并参考图 6-8 进行 GPIO 的配置。控制 4 个指示灯轮流点亮，实质是 4 个引脚不断送出以下电平状态：0001 → 0010 → 0100 → 1000，然后开始下一次循环。

6.4.2 基础任务：离散信号检测

➡ **任务目标**：掌握 CubeMX 软件配置 GPIO 外设的方法

➡ **任务内容**：利用 Nucleo 开发板上的按键 B1 模拟离散信号输入。当读到信号值为低电平时，开启指示灯 LD2；当读到信号值为高电平时，关闭指示灯 LD2。

➡ **硬件原理**：按键 B1 由 PC13 控制：按键按下时为低电平，释放时为高电平。指示灯 LD2 由 PA5 控制：PA5 输出高电平，开启指示灯；PA5 输出低电平，关闭指示灯。

电路原理图参见第二章的图 2-16 和图 2-18。

任务实现

1. 引脚分配

在芯片引脚窗口中选择 PA5 引脚为 GPIO_Output，PC13 引脚为 GPIO_Input，如图 6-9 所示。

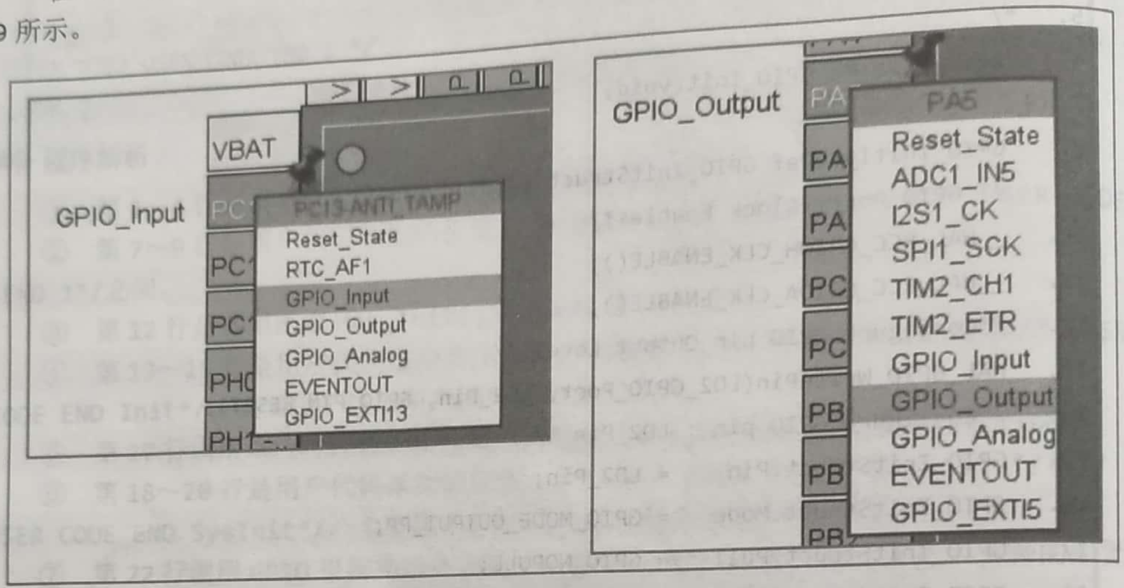


图 6-9 引脚分配

2. 外设配置

在外设设置窗口中选择：SystemCore → GPIO。在出现的引脚配置窗口中选择 GPIO 标签页。

点击 PA5 引脚一栏，在展开的参数配置窗口中对 PA5 作如下配置：

- GPIO output level（初始电平）：Low（低电平）
- GPIO mode（工作模式）：Output Push Pull（推挽输出）
- GPIO Pull-up and /Pull-down（上拉/下拉电阻）：No pull-up and no pull-down（无下拉/下拉电阻）
- Maximum output speed（输出速度）：Low（低速）
- User Label（引脚名称）：LD2

点击 PC13 引脚一栏，在展开的参数配置窗口中对 PC13 作如下配置：

- GPIO mode（工作模式）：Input mode（输入模式）
- GPIO Pull-up and /Pull-down（上拉/下拉电阻）：No pull-up and no pull-down（无下拉/下拉电阻）
- User Label（引脚名称）：B1

具体配置如图 6-10 所示。

注意：在设置引脚名称后，芯片引脚窗口中 PA5 引脚的名称将由 GPIO_Output 变为 LD2，PC13 引脚的名称将由 GPIO_Input 变为 B1。

3. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。在 main.c 文件的 while 循环中添加应用代码，如程序清单 6-11 所示。

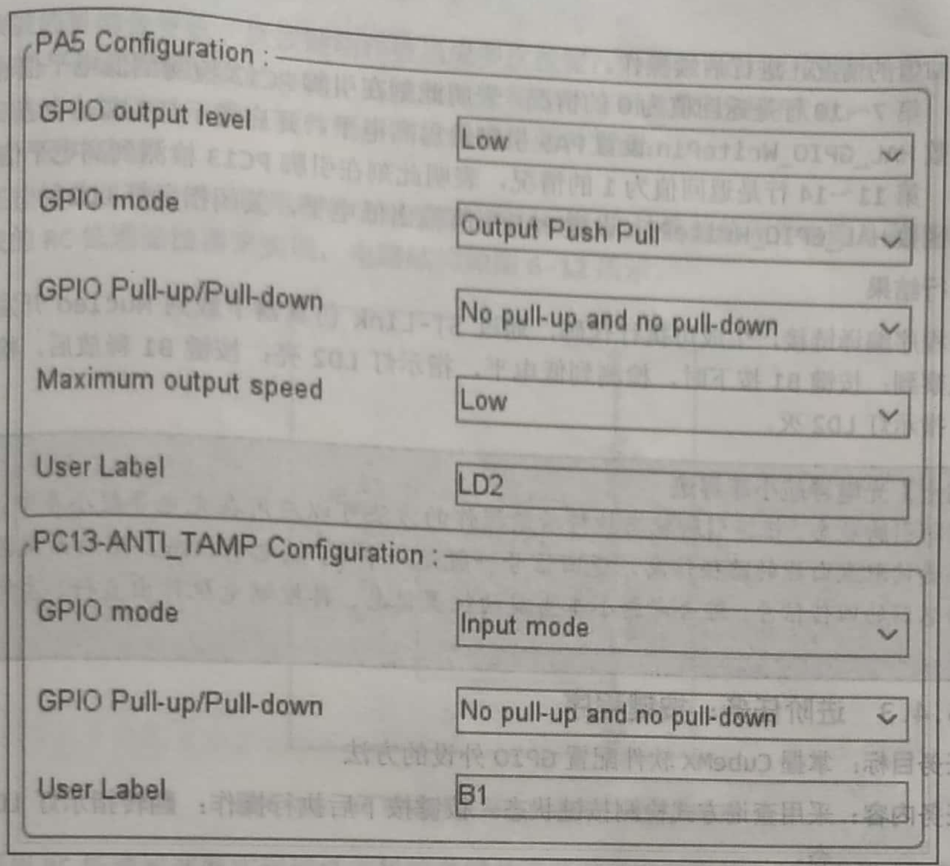


图 6-10　外设配置

程序清单 6-11　离散信号检测

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET) //检测高电平
  {
     HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);  //开启 LD2
  }
                                                              //检测低电平
  else
  {
     HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET); //关闭 LD2
  }
}
  /* USER CODE END 3 */
```

程序解析

① 程序采用if else两分支选择结构实现，利用函数HAL_GPIO_ReadPin检测信号，

根据返回值的情况，进行后续操作。

② 第 7～10 行是返回值为 0 的情况，表明此刻在引脚 PC13 检测到低电平信号，因此调用函数 HAL_GPIO_WritePin 设置 PA5 引脚输出高电平，开启指示灯 LD2。

③ 第 11～14 行是返回值为 1 的情况，表明此刻在引脚 PC13 检测到高电平信号，因此调用函数 HAL_GPIO_WritePin 设置 PA5 引脚输出低电平，关闭指示灯 LD2。

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上，可以观察到：按键 B1 按下时，检测到低电平，指示灯 LD2 亮；按键 B1 释放后，检测到高电平，指示灯 LD2 灭。

【小贴士】光电导航小车寻迹

判断引脚状态，根据引脚状态执行后续操作的方法可以应用在光电导航小车中。利用光电传感器读取黑白线的路径信息，返回信号一般是一个高/低电平状态。然后系统根据多个光电传感器的回传信号，综合考虑小车当前的位置信息，再控制电机作出直行、左转和右转的操作。

6.4.3 进阶任务：按键程序

任务目标：掌握 CubeMX 软件配置 GPIO 外设的方法

任务内容：采用查询方式检测按键状态，按键按下后执行操作：翻转指示灯 LD2 的状态。

硬件原理：按键 B1 由 PC13 控制：按键按下时为低电平，释放时为高电平。指示灯 LD2 由 PA5 控制：PA5 输出高电平，开启指示灯；PA5 输出低电平，关闭指示灯。电路原理图参见第二章的图 2-16 和图 2-18。

任务实现

本任务所使用的引脚和外设与离散信号检测任务的相同，因此省去引脚分配和外设配置的过程。

1. 按键检测原理

按键内部采用机械弹性开关，由于机械触点的弹性作用，按键在按下后并不会马上稳定地接通，在释放时也不会立刻断开，因而在按下及释放的瞬间均伴随有一连串的抖动，如图 6-11 所示。抖动时间的长短由按键的机械特性决定，一般为 5ms～10ms，而用户按键的整个时长为零点几秒至数秒。

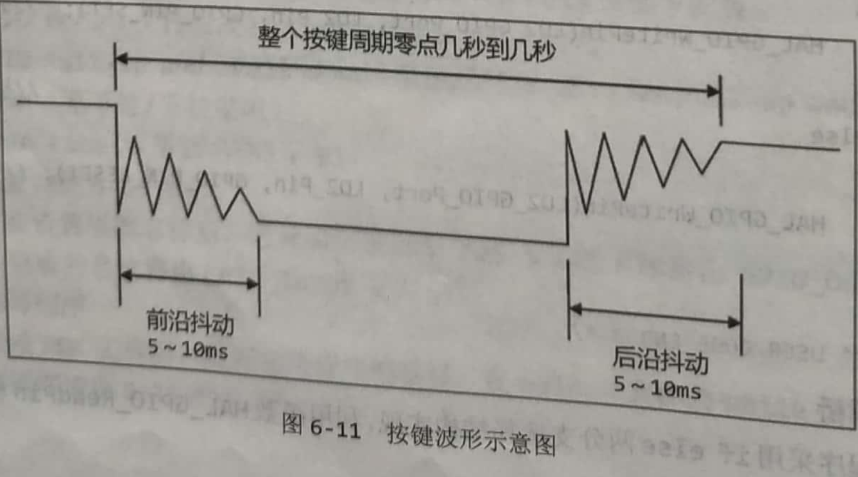


图 6-11 按键波形示意图

按键的抖动会导致一次按键动作被当成多次按键，为确保 MCU 对按键的一次闭合仅作一次处理，必须消除按键的抖动，在按键处于稳定状态时读取按键的状态。按键的去抖动，一般有硬件消抖和软件消抖两种方式：

● 硬件消抖

在按键数目较少时，可用硬件方法消除按键抖动。比较简单的硬件电路是利用电阻和电容构成的 RC 低通滤波器来实现，电路结构如图 6-12 所示。

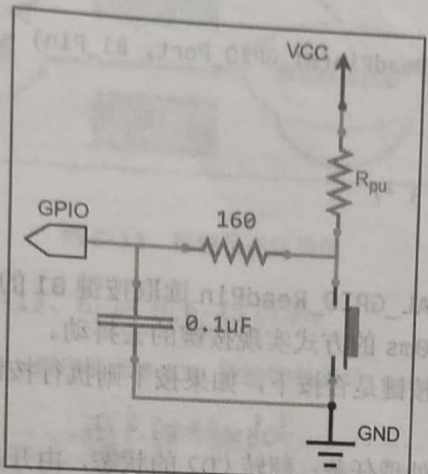


图 6-12　RC 低通滤波器构成的硬件去抖动电路

采用 RC 低通滤波器实现硬件去抖的关键是选择适当的电阻值 R 和电容值 C。在工程应用中，由于人按键的频率是低于 1KHz 的，而按键抖动的频率一般是几十/几百 KHz，因此可以将低通滤波器的截止频率选择为 10KHz，按照计算公式 6-1，可以选取电阻值为 160 欧，电容值为 0.1UF 构成的 RC 网络。

$$F_{LP} = 1 / 2\pi RC \tag{6-1}$$

● 软件消抖

如果按键数目较多，常采用软件方法去抖动，即检测出按键闭合后执行一个延时程序，产生 5ms～10ms 左右的延时，让前沿抖动消失后，再次检测按键的状态。如果仍保持闭合状态，则确认为按键真正按下，然后执行按键处理任务。同样，对于按键的释放，也需要采用延时函数去掉后沿的抖动。

2. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。在 main.c 文件的 while 循环中添加应用代码，如程序清单 6-12 所示。

程序清单 6-12　按键程序

```
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET) //按键按下
  {
```

```
9.      HAL_Delay(10);                                              //延时去抖动
10.     // 按键依然按下
11.     if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET)
12.     {
13.       HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);               //翻转 LD2
14.     }
15.     // 等待按键释放
16.     while(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET);
17.   }
18. }
19.   /* USER CODE END 3 */
```

➡ 程序解析

① 第 7 行调用函数 HAL_GPIO_ReadPin 读取按键 B1 的状态。

② 第 9 行采用延时 10ms 的方式实现按键的去抖动。

③ 第 11 行再次判断按键是否按下，如果按下则执行按键处理任务。否则退出本次按键检测过程。

④ 第 13 行执行按键处理任务：翻转 LD2 的状态，由开启变成关闭或者由关闭变成开启。调用引脚翻转函数 HAL_GPIO_TogglePin 实现。

⑤ 第 16 行采用死循环方式等待按键 B1 释放。

➡ 运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上，可以观察到：按键 B1 每按下一次，指示灯 LD2 的状态就翻转一次。

6.4.4 挑战任务：状态机设计思想

在程序清单6-12中，按键的去抖动采用了延时的方式，利用函数HAL_Delay延时10ms。这种方式在一般的应用场合是可以的，但是在实际的工程应用中，延时 10ms 会占用处理器的执行时间，降低处理器的利用效率。因为在延时的时候，处理器无法处理其他事务。在实际的工程应用中往往采用状态机的设计思想，并借助定时中断来完成按键的处理。

➡ 任务目标： 掌握状态机设计思想

➡ 任务内容： 利用状态机设计思想编写按键处理程序，按键按下后执行操作：翻转指示灯 LD2 的状态。

➡ 硬件原理： 按键 B1 由 PC13 控制：按键按下时为低电平，释放时为高电平。指示灯 LD2 由 PA5 控制：PA5 输出高电平，开启指示灯；PA5 输出低电平，关闭指示灯。电路原理图参见第二章的图 2-16 和图 2-18。

➡ 任务实现： 设计思路如下：

定义三个按键状态：按键检测状态 KEY_CHECK、按键确认状态 KEY_COMFIRM 和按键释放状态 KEY_RELEASE。利用定时器产生 10ms 的中断，在定时中断服务程序中检测按键引脚的电平，根据电平值切换按键状态。同时设置一个全局变量 KeyFlag，如果经过了按键的三种状态周期，则 KeyFlag 置 1，主程序在 while 循环中不断检测 KeyFlag，为 1 则执行按键的处理任务。按键的状态转换过程及转换条件如图 6-13 所示。

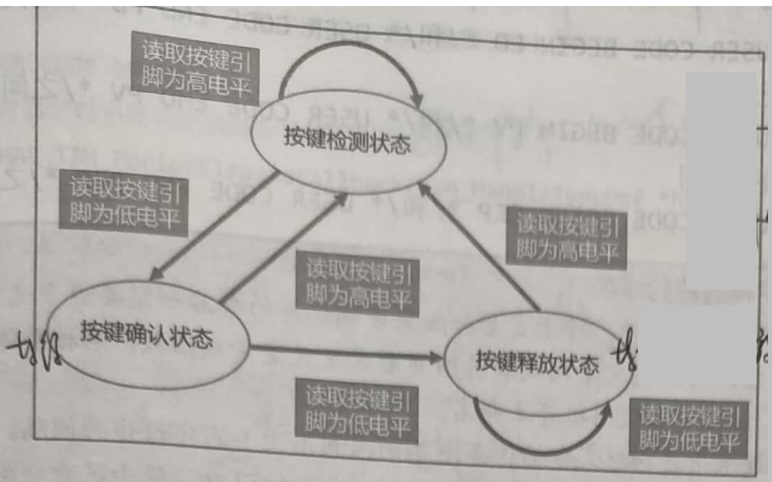


图 6-13　按键状态转换图

具体的程序如程序清单 6-13、6-14 和 6-15 所示。

程序清单 6-13　数据类型定义

```
1. /* Private typedef --------                  ----------------------\
2. /* USER CODE BEGIN PTD */
3. typedef enum
4. {
5.     KEY_CHECK = 0,  // 按键检测状态
6.     KEY_COMFIRM,    // 按键确认状态
7.     KEY_RELEASE     // 按键释放状态
8. }KEY_STATE;
9. /* USER CODE END PTD */
10. /* Private variables ----
11. /* USER CODE BEGIN PV */
12. KEY_STATE KeyState    = 0; // 按键状态
13. uint8_t   KeyFlag     = 0; // 按键值有效标志，1：有效；0：无效
14. /* USER CODE END PV */
```

程序解析

① 第 3～8 行定义了一个枚举数据类型 KEY_STATE，用于说明按键状态。采用枚举类型的目的是提高程序的可读性，并限制了数据的范围。

② 第 12 行利用枚举数据类型 KEY_STATE 定义了一个枚举变量 KeyState，用于存放按键状态。

③ 第 13 行利用一个标志变量 KeyFlag，为 1 表示按键有效，为 0 表示按键无效。

注意：用户定义的数据类型必须添加在对应的位置：

- 用户头文件

添加在/* USER CODE BEGIN Includes */和/* USER CODE END Includes*/之间。

- 用户自定义数据类型

添加在/* USER CODE BEGIN PTD */和/* USER CODE END PTD*/之间。

- 用户宏定义

添加在/* USER CODE BEGIN PD */和/* USER CODE END PD */之间。

- 用户变量

添加在/* USER CODE BEGIN PV */和/* USER CODE END PV */之间。

- 用户函数声明

添加在/* USER CODE BEGIN PFP */和/* USER CODE END PFP */之间。

> **【小贴士】变量和函数的命名**
>
> 开发人员在程序设计时，对于变量和函数等标识符的命名一定要遵守通用的编程习惯：
>
> 1. 见名知意：利用英文单词或者其缩写形式定义变量或函数，名称要体现变量的作用或函数的功能，切记不要使用拼音来命名。
>
> 2. 变量一般采用名词形式命名，多个单词间利用大小写字母作为间隔。全局变量的首字母大写，如 KeyFlag；局部变量的首字母小写，如 keyFlag，便于区分全局变量和局部变量。
>
> 3. 函数一般采用动宾结构命名，首字母大写，也是利用大小写字母作为间隔，如 GetValue 等。
>
> 4. 宏定义和用户自定义数据类型全部采用大写字母，利用下划线作为间隔，如 KEY_STATE 等。

程序清单 6-14　按键处理程序

```
1.  /* USER CODE BEGIN 2 */
2.  HAL_TIM_Base_Start_IT(&htim10);
3.  /* USER CODE END 2 */
4.  /* Infinite loop */
5.  /* USER CODE BEGIN WHILE */
6.  while (1)
7.  {
8.   /* USER CODE END WHILE */
9.
10.  /* USER CODE BEGIN 3 */
11.   if( KeyFlag == 1)                          // 检测按键有效标志
12.   {
13.      KeyFlag = 0;                            // 清除标志
14.      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);  // 按键执行任务：翻转 LD2
15.   }
16. }
17. /* USER CODE END 3 */
```

程序解析

① 第 2 行调用函数 HAL_TIM_Base_Start_IT 启动定时器 10 工作，每隔 10ms 中断一次，定时器的介绍参见第八章。

② 第 11～15 在 while 循环中不断检测 KeyFlag 标志变量是否置 1，置 1 则清除标志变量，然后执行按键任务：翻转指示灯 LD2 的状态。

程序清单 6-15　按键状态程序

```
/* USER CODE BEGIN 4 */
// 定时器定时中断的回调函数
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
  if(htim->Instance == htim10.Instance)
  {
    switch(KeyState)
    {
      case KEY_CHECK:
      {
        // 读到低电平，进入等待按键确认状态
        if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET)
        {
          KeyState = KEY_COMFIRM;
        }
        break;
      }
      case KEY_COMFIRM:
      {
        // 读到低电平，进入等待按键释放状态
        if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_RESET)
        {
          KeyState = KEY_RELEASE;
        }
        // 读到高电平，可能是干扰信号，返回初始状态
        else
        {
          KeyState = KEY_CHECK;
        }
        break;
      }
      case KEY_RELEASE:
      {
        // 读到高电平，说明按键释放，返回初始状态
        if(HAL_GPIO_ReadPin(B1_GPIO_Port, B1_Pin) == GPIO_PIN_SET)
        {
          KeyState = KEY_CHECK;
          KeyFlag  = 1;          // 设置有效按键标志
        }
        break;
      }
```

```
42.         default: break;
43.     }
44.   }
45. }
46. /* USER CODE END 4 */
```

程序解析

① 第 1 行和第 46 行之间是用户的应用程序添加的位置：/* USER CODE BEGIN 4 */ 和/* USER CODE END 4 */之间。

② 第 3 行是定时器的定时中断回调函数，该函数每隔 10ms 执行一次。关于中断回调函数的介绍参见第七章。

③ 第 5 行是判断定时中断是由哪一个定时器触发的，htim10 是定时器的句柄定义，封装了定时器的寄存器定义及初始化参数定义，具体可参见第八章的介绍。

④ 第 7～43 行利用 switch-case 语句判断按键状态的变化。

⑤ 第 9～17 行是在按键检测状态下判断按键是否按下。如果按下，则由按键检测状态转入按键确认状态，否则保持当前状态。

⑥ 第 18～31 行是在按键确认状态下判断按键是否按下。如果按下，则由按键确认状态转入按键释放状态。否则，说明可能出现干扰信号，修改状态至初始状态：按键检测状态。

⑦ 第 32～41 行是在按键释放状态下判断按键是否释放。如果释放，则设置按键有效标志 KeyFlag 为 1，并将按键状态转入按键检测状态，准备新一轮的检测。否则保持当前状态。

你的任务

任选 Nucleo 开发板上的 4 个 I/O 引脚控制 4 个按键：KEY0、KEY1、KEY2 和 KEY3，4 个 I/O 引脚控制 4 个指示灯：LED0、LED1、LED2 和 LED3，编程实现 4 个按键的处理程序：KEY0 控制 LED0 翻转，KEY1 控制 LED1 翻转，KEY2 控制 LED2 翻转，KEY3 控制 LED3 翻转。

提示：在设计一个键盘缓冲区，存放读取的按键值。定时中断中不断扫描按键，将有效的按键值存放到键盘缓冲区，主程序中不断从键盘缓冲区读取键值进行处理。

6.5 硬件抽象层设计

在第一章里，我们介绍了硬件抽象层的设计思想：屏蔽底层硬件的差异，方便应用程序的移植，提高开发效率。

板级驱动包 BSP 是硬件抽象层的具体体现，主要由针对具体电路板上外围电路相关的驱动程序组成。BSP 直接和硬件交互，采用寄存器或者库函数的方式控制片内外设，并根据电路板的具体硬件设计封装出驱动程序，提供给应用层或操作系统调用。比如利用 GPIO 的库函数封装出指示灯的驱动程序。

BSP 一般以外围电路模块为单位构成，比如指示灯、按键和液晶显示屏等。每一个外围电路模块的驱动程序单独用一个文件夹存放，如 LED、KEY 和 LCD 等。文件夹内包含 2 个文件：头文件包含本模块的数据类型以及提供给外部调用的接口函数的说明，C 文件主要是本模块的硬件引脚定义以及接口函数的具体实现。

本节将以指示灯的设计作为切入点，介绍硬件抽象层的具体实现。

6.5.1 指示灯 BSP 的实现

我们假设有电路板上有 4 个指示灯，分别由 PA0、PB0、PC0 和 PD0 这 4 个 GPIO 引脚控制，控制方式采用高电平驱动：引脚输出高电平开启指示灯。

对于指示灯模块，我们需要设计 BSP_LED.h 和 BSP_LED.c 两个文件。下面将分别给出具体的设计实现。

1. BSP_LED.h 文件

BSP_LED.h 文件主要是用户自定义的数据类型以及提供给外部调用的接口函数的说明。具体代码如程序清单 6-16 所示。

程序清单 6-16　BSP_LED.h 文件

```
/*****************************************************************
* 模块名称 : 指示灯驱动模块
* 文件名   : BSP_LED.h
* 说明     : 头文件，提供自定义数据类型及外部调用的接口函数
* 版 本 号 : V2.0
* 修改记录 :
*              版本号          日期            作者
*              V2.0         2018-05-02        qiqiang
*****************************************************************/
#ifndef __BSP_LED_H
#define __BSP_LED_H
/*****************************************************************
                        用户自定义数据类型
*****************************************************************/
typedef enum
{
  LED0 = 0,
  LED1 = 1,
  LED2 = 2,
  LED3 = 3,
}LED_INDEX;
/*****************************************************************
                      本模块提供给外部调用的函数
*****************************************************************/
extern void BSP_LED_Init    ( LED_INDEX Led );  // 指示灯初始化
extern void BSP_LED_On      ( LED_INDEX Led );  // 开启指示灯
extern void BSP_LED_Off     ( LED_INDEX Led );  // 关闭指示灯
extern void BSP_LED_Toggle  ( LED_INDEX Led );  // 翻转指示灯状态
#endif
/****************************** (END OF FILE) *******************/
```

程序解析

① 第 1～9 行是模块的功能说明，版本号以及修改记录等，方便其他开发人员快速了解该程序模块的用途。这是在工程应用中常用的做法，大家在编程时也应该有类似的程序模块介绍，养成良好的编程习惯。

② 第 10～11 行是利用#ifdef 或 ifndef 条件编译指令来避免头文件的重复包含。如果在一个工程中，某一个模块用#include 指令包含 BSP_LED.h 文件。假设是第一次引用，由于宏常量__BSP_LED_H 没有定义，因此 ifndef 条件成立，将复制后面的语句到该模块中，包括定义宏常量__BSP_LED_H，以及后面的数据类型和接口函数的声明。如果该工程中还有其他模块包含 BSP_LED.h 文件，由于__BSP_LED_H 已经定义了，将不会在重复复制这些内容到该模块。这种方法在用户编写自己的头文件时经常使用。

③ 第 15～21 行定义了一个枚举数据类型 LED_INDEX，用于表示指示灯的序号，其中的枚举变量 LED0、LED1、LED2 和 LED3 作为用户调用指示灯接口函数时的入口参数，用来控制对应的指示灯。采用枚举类型的目的是提高程序的可读性，并限制了数据的范围。

④ 第 25～28 行是指示灯驱动模块提供给外部调用的接口函数的声明语句，包括指示灯初始化函数BSP_LED_Init，开启指示灯函数BSP_LED_On，关闭指示灯函数BSP_LED_Off，以及翻转指示灯状态函数 BSP_LED_Toggle。4 个函数的入口参数均为枚举类型 LED_INDEX，取值范围为 LED0～LED3。

⑤ 第 29 号#endif 作为第 9 行的条件编译指令#ifndef 对应的结束指令，表明条件编译到该语句结束。

2. BSP_LED.c 文件

BSP_LED.c 文件主要是控制指示灯的硬件引脚定义以及接口函数的具体实现。具体代码如程序清单 6-17 所示。

程序清单 6-17 BSP_LED.c 文件

```
/*****************************************************************************
 * 模块名称：指示灯驱动模块
 * 文件名  ：BSP_Led.c
 * 说明    ：驱动 LED 指示灯
 * 版 本 号：V2.0
 * 修改记录：1、将 LED 驱动采用结构体指针数组的形式进行改写，修改只需要修改 LED
              的数量，引脚定义以及存放端口和引脚的两个数组。 2018/05/02
 *           版本号          日期          作者
 *            V2.0        2018-05-02     qiqiang
******************************************************************************/
/*****************************************************************************
                              模块移植性说明
******************************************************************************/
/*
1、如果用户的 LED 指示灯个数小于等于 4 个，可以将多余指示灯的引脚定义全部定义为
   和第 1 个指示灯一样，其余代码不做任何修改
2、如果用户的 LED 指示灯个数大于 4 个，需要修改如下内容：
   (1) 增加.h 文件中 LED_INDEX 类型的枚举变量 LEDx
```

```
    (2) 修改指示灯数量定义 LEDn
    (3) 修改控制引脚的定义
    (4) 修改端口和引脚两个数组：GPIO_PORT[LEDn]和 GPIO_Pin[LEDn]
    (5) 修改引脚的时钟控制宏定义 LEDx_GPIO_CLK_ENABLE(__INDEX__)的内容
*/
/**************************************************************************
                              INCLUDES
**************************************************************************/
#include "stm32f4xx_hal.h"
#include "BSP_LED.h"
/**************************************************************************
                           指示灯硬件接口定义
**************************************************************************/
#define LEDn                        4               // 指示灯数量
// 指示灯 0
#define LED0_PIN                    GPIO_PIN_0      // PA0
#define LED0_GPIO_PORT              GPIOA
#define LED0_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOA_CLK_ENABLE()
// 指示灯 1
#define LED1_PIN                    GPIO_PIN_0      // PB0
#define LED1_GPIO_PORT              GPIOB
#define LED1_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOB_CLK_ENABLE()
// 指示灯 2
#define LED2_PIN                    GPIO_PIN_0      // PC0
#define LED2_GPIO_PORT              GPIOC
#define LED2_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOC_CLK_ENABLE()
// 指示灯 3
#define LED3_PIN                    GPIO_PIN_0      // PD0
#define LED3_GPIO_PORT              GPIOD
#define LED3_GPIO_CLK_ENABLE()      __HAL_RCC_GPIOD_CLK_ENABLE()
 // 端口数组，数组中存放的都是指示灯控制引脚所属端口
GPIO_TypeDef *GPIO_PORT[LEDn] = {
                                LED0_GPIO_PORT,
                                LED1_GPIO_PORT,
                                LED2_GPIO_PORT,
                                LED3_GPIO_PORT,
                              };
// 引脚数组，数组中存放的都是指示灯控制引脚
const uint16_t GPIO_PIN[LEDn] = {
                                LED0_PIN,
                                LED1_PIN,
                                LED2_PIN,
```

```c
                                   LED3_PIN,
                                 };
// 端口时钟使能宏定义
#define LEDx_GPIO_CLK_ENABLE(__INDEX__) do{ \
                                    if((__INDEX__) == 0)       \
                                    LED0_GPIO_CLK_ENABLE();    \
                                    else if((__INDEX__) == 1)  \
                                    LED1_GPIO_CLK_ENABLE();    \
                                    else if((__INDEX__) == 2)  \
                                    LED2_GPIO_CLK_ENABLE();    \
                                    else if((__INDEX__) == 3)  \
                                    LED3_GPIO_CLK_ENABLE();    \
                                    } while(0)
/*********************************************************************
                            本模块提供外部调用的函数
*********************************************************************/
/*********************************************************************
 * @name        BSP_LED_Init
 * @brief       Configures LED GPIO.
 * @param[in]   Led: Specifies the Led to be configured.
 *              This parameter can be one of following parameters:
 * @arg         LED0~LED3
 * @return      None
 * @note        This function is called by BSP_Init()
*********************************************************************/
void BSP_LED_Init(LED_INDEX Led)
{
  GPIO_InitTypeDef  GPIO_InitStruct;
  /* Enable the GPIO_LED Clock */
  LEDx_GPIO_CLK_ENABLE(Led);
  /* Configure the GPIO_LED pin */
  GPIO_InitStruct.Pin   = GPIO_PIN[Led];
  GPIO_InitStruct.Mode  = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull  = GPIO_NOPULL;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
  HAL_GPIO_Init(GPIO_PORT[Led], &GPIO_InitStruct);
  HAL_GPIO_WritePin(GPIO_PORT[Led], GPIO_PIN[Led], GPIO_PIN_RESET);
}
/*********************************************************************
 * @name        BSP_LED_On
 * @brief       Turns selected LED On
 * @param[in]   Led: Specifies the Led to be set on.
```

```
103.  *               This parameter can be one of following parameters:
104.  * @arg          LED0~LED3
105.  * @return       None
106.  * @note
107. ****************************************************************************/
108. void BSP_LED_On(LED_INDEX Led)
109. {
110.   HAL_GPIO_WritePin(GPIO_PORT[Led], GPIO_PIN[Led], GPIO_PIN_SET);
111. }
112. /***************************************************************************
113.  * @name         BSP_LED_Off
114.  * @brief        Turns selected LED Off
115.  * @param[in]    Led: Specifies the Led to be set on. off
116.  *               This parameter can be one of following parameters:
117.  * @arg          LED0~LED3
118.  * @return       None
119.  * @note
120. ****************************************************************************/
121. void BSP_LED_Off(LED_INDEX Led)
122. {
123.   HAL_GPIO_WritePin(GPIO_PORT[Led], GPIO_PIN[Led], GPIO_PIN_RESET);
124. }
125. /***************************************************************************
126.  * @name         BSP_LED_Toggle
127.  * @brief        Toggles the selected LED.
128.  * @param[in]    Led: Specifies the Led to be toggled.
129.  *               This parameter can be one of following parameters:
130.  * @arg          LED0~LED3
131.  * @return       None
132.  * @note
133. ****************************************************************************/
134. void BSP_LED_Toggle(LED_INDEX Led)
135. {
136.   HAL_GPIO_TogglePin(GPIO_PORT[Led], GPIO_PIN[Led]);
137. }
138. /******************************* (END OF FILE) *************************/
```

程序解析

① 第1～10行是模块的功能说明，版本号以及修改记录等。

② 第11～23行是模块的移植说明，即该模块移植到其他硬件平台上需要修改的地方。

③ 第24～28行是模块所包含的头文件。由于要使用HAL库函数，因此包含了头文件"STM32f4xx_hal.h"。由于要使用指示灯枚举类型作为接口函数的入口参数，因此包含了头文件"BSP_LED.h"。

④ 第 32 行定义指示灯的数量，默认为 4 个。移植时，如果目标电路板上的指示灯数量小于等于 4 个，可以不做修改；如果大于 4 个，则修改为指示灯的实际数量。

⑤ 第 33～48 行是指示灯控制引脚的定义，包括引脚序号、引脚所属端口和端口时钟使能三部分。为了提高程序的可读性，将原有的端口号和引脚号利用宏定义取了一个别名，如 GPIO_PIN_0 重定义为 LED0_PIN，GPIOA 重定义为 LED_GPIO_PORT。其中端口的时钟使能语句：__HAL_RCC_GPIOC_CLK_ENABLE()是 HAL 库中时钟模块提供的接口函数，具体可以参见 stm32f4xx_hal_rcc.h 文件。这部分代码在移植时需要根据目标电路板上控制指示灯的实际引脚来修改。

⑥ 第 49～55 行定义了一个端口数组，存放指示灯控制引脚所属端口。移植时，如果目标电路板上的指示灯数量小于等于 4 个，可以不做修改，如果大于 4 个，则需要添加新的端口定义。

⑦ 第 56～62 行定义了一个引脚数组，存放指示灯控制引脚。移植时，如果目标电路板上的指示灯数量小于等于 4 个，可以不做修改，如果大于 4 个，则需要添加新的引脚定义。

⑧ 第 63～73 行定义了用于端口时钟使能的宏函数，入口参数为__INDEX__，用于使能指定端口的时钟。宏函数中各语句末尾的“\”号表示宏定义的换行，避免一行代码过长，方便代码的阅读。移植时，如果目标电路板上的指示灯数量小于等于 4 个，可以不做修改，如果大于 4 个，则需要添加新的端口时钟使能语句。

⑨ 第 77～98 行定义了指示灯初始化函数 BSP_LED_Init。其中，第 77～85 行是函数的功能描述、入口参数、返回值和注意事项的说明；第 86～98 行是初始化函数的具体实现:先在函数体中定义了一个 GPIO 初始化结构体变量 GPIO_InitStruct，然后使能端口时钟，并在结构体的成员变量中填入配置参数：引脚号、推挽输出、无上拉\下拉电阻和低速输出等，最后调用函数 HAL_GPIO_Init 完成指定引脚的初始化。第 97 行调用函数 HAL_GPIO_WritePin 设置指定引脚默认输出低电平，使指示灯在系统复位后为关闭状态。

注意：指示灯初始化函数 BSP_LED_Init 一次只能初始化 1 个指示灯控制引脚，多个控制引脚需要多次调用该函数。

⑩ 第 99～111 行定义了指示灯开启函数 BSP_LED_On。函数的入口参数为枚举变量 LED_INDEX，取值范围为 LED0～LED3。函数内部调用函数 HAL_GPIO_WritePin 使指定引脚输出高电平，开启对应的指示灯。

⑪ 第 112～124 行定义了指示灯关闭函数 BSP_LED_Off。函数的入口参数为枚举变量 LED_INDEX，取值范围为 LED0～LED3。函数内部调用函数 HAL_GPIO_WritePin 使指定引脚输出低电平，关闭对应的指示灯。

⑫ 第 125～137 行定义了指示灯翻转函数 BSP_LED_Toggle。函数的入口参数为枚举变量 LED_INDEX，取值范围为 LED0～LED3。函数内部调用引脚翻转函数 HAL_GPIO_TogglePin 翻转指定引脚的电平，从而翻转对应指示灯的状态。

BSP 设计小结

从上述指示灯模块的设计中，我们可以总结出以下的设计原则：

- h 文件设计方法

h 文件作为模块对外的基本说明文件，其内部只包括数据类型的说明以及提供给外部调用的接口函数的说明。一般不定义变量以及硬件引脚的说明，也不包含任何其他的头文件，这样可以确保头文件的通用性。在进行程序移植时，用户不需要修改 BSP_LED.h 文件，并且由于应用层只是调用.h 文件提供的 4 个接口函数，因此也不需要修改应用层的代码。用

户只需要修改 BSP_LED.c 文件，即可完成移植。

● C 文件设计方法

C 文件主要完成接口函数的实现，其内部需要包含与硬件相关的全部定义，以及相关的头文件。在具体实现时，调用 HAL 库提供的接口函数，进行二次封装，提供出一个可读性更强，移植性更好的模块驱动程序。

6.5.2 指示灯 BSP 的应用

完成指示灯模块的设计后，我们需要将该模块应用到具体的工程中。我们以 Nucleo 开发板为目标电路板，将指示灯的 BSP 移植到该平台，并完成指示灯的驱动任务。

整个移植步骤可以分成修改 BSP、添加 BSP 至用户工程和修改应用程序等 3 个步骤：

1. 修改 BSP

根据指示灯 BSP 模块的移植说明，移植该模块要区分两种情况：

(1) 目标电路板指示灯数量小于等于 4

此时，需要修改指示灯引脚控制定义，并将多余指示灯的引脚定义全部修改为与第一个指示灯一样。其余代码不做任何修改。

(2) 目标电路板指示灯数量大于 4

此时，需要进行如下步骤的修改：

1) 增加 BSP_LED.h 文件中 LED_INDEX 类型的枚举变量；
2) 修改 BSP_LED.c 文件中指示灯数量定义 LEDn 为目标电路板指示灯的实际数量；
3) 修改 BSP_LED.c 文件中指示灯控制引脚的定义，并增加相关指示灯控制引脚的定义；
4) 修改 BSP_LED.c 文件与端口和引脚相关的两个数组：GPIO_PORT[LEDn]和 GPIO_PIN[LEDn]，根据指示灯的数量修改数组元素；
5) 修改 BSP_LED.c 文件中与引脚相关的时钟控制宏定义：LEDx_GPIO_CLK_ENABLE (__INDEX__)，增加相关指示灯的时钟控制语句。

对于 NucleoF411RE 开发板而言，板载指示灯只有 1 个，由 PA5 引脚控制，属于第 1 种移植情况，因此只需要将 BSP_LED.c 文件中关于 LED0 的控制引脚、端口号和端口时钟使能进行修改，并将多余的 3 个指示灯 LED1、LED2 和 LED3 的控制引脚、端口号和端口时钟使能全部修改为与 LED0 一致即可。具体修改内容如下所示：

```
// LED 接口的硬件定义
// 指示灯 0
#define LED0_PIN                   GPIO_PIN_5    // PA5
#define LED0_GPIO_PORT             GPIOA
#define LED0_GPIO_CLK_ENABLE()     __HAL_RCC_GPIOA_CLK_ENABLE()
// 指示灯 1
#define LED1_PIN                   GPIO_PIN_5    // PA5
#define LED1_GPIO_PORT             GPIOA
#define LED1_GPIO_CLK_ENABLE()     __HAL_RCC_GPIOA_CLK_ENABLE()
// 指示灯 2
#define LED2_PIN                   GPIO_PIN_5    // PA5
#define LED2_GPIO_PORT             GPIOA
#define LED2_GPIO_CLK_ENABLE()     __HAL_RCC_GPIOA_CLK_ENABLE()
```

```
// 指示灯 3
#define LED3_PIN                GPIO_PIN_5    // PA5
#define LED3_GPIO_PORT          GPIOA
#define LED3_GPIO_CLK_ENABLE()  __HAL_RCC_GPIOA_CLK_ENABLE()
```

2. 添加 BSP 至用户工程

使用 CubeMX 软件建立工程时，已经建立了基本的工程框架。我们在这个工程框架的基础上添加指示灯的 BSP。

(1) 复制工程。将基本任务中驱动指示灯的工程整体复制到新建的文件夹 LED_BSP 中，然后修改 CubeMX 工程文件 xxx.ioc 为 LED_BSP.ioc，并删除其中的 MDK-ARM 文件夹。同时新建一个 BSP 文件夹，并在 BSP 文件夹下面新建一个 LED 文件夹，将利用编辑软件编写的 BSP_LED.c 和 BSP_LED.h 两个文件复制到 LED 文件夹下面，如图 6-14 所示。

图 6-14 修改工程

(2) 点击 LED_BSP.ioc 文件，启动 CubeMX 软件。不需要进行任何配置的修改，直接生成 MDK 的工程。

(3) 添加 BSP 文件。在 MDK 主界面的工具栏点击工程文件管理图标，在弹出的工程文件管理窗口中，选择 Groups 栏。利用新建组图标，新建一个名为 BSP 的组。选中 BSP 组，然后在 Files 栏中，利用“Add Files”按钮找到 BSP\LED 下面的 BSP_LED.c 和 BSP_LED.h 文件，添加到 BSP 组。最后点击“OK”按钮退出。完成文件添加后，工程窗口中将增加 BSP 组，以及组中包含的文件 BSP_LED.c 和 BSP_LED.h。相关操作步骤如图 6-15 所示。

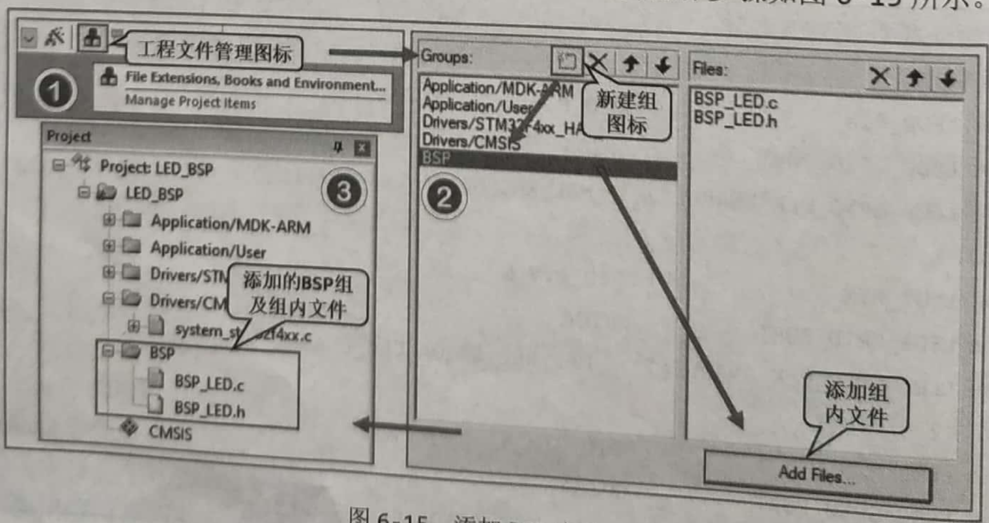


图 6-15 添加 BSP 文件

(4) 添加头文件包含路径。在 MDK 主界面的工具栏点击工程设置图标，在弹出的工程配置窗口中，选择"C/C++"选项卡。选择 Include Paths，点击最右边的省略号图标，进入设置头文件路径窗口。在该窗口中，利用新建头文件路径图标，找到 BSP_LED.h 文件所在的路径，添加到头文件路径窗口中。最后点击"OK"按钮退出。相关操作步骤如图 6-16 所示。

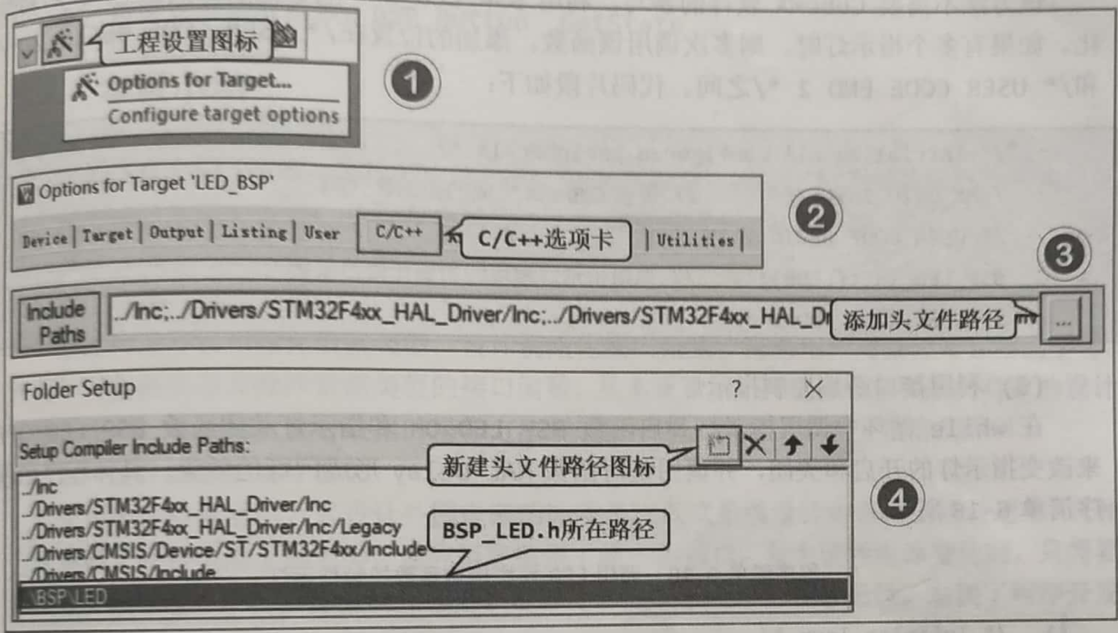


图 6-16　添加头文件路径

(5) 工程设置。按照第五章中的介绍，进行工程设置，主要是选择仿真器以及设置程序下载后运行（Reset and Run）等配置选项。

3. 修改应用程序

完成 BSP 文件的添加及工程配置后，就可以应用程序的编写了。

(1) 添加头文件

由于应用程序在调用 LED 模块提供的接口函数时，需要接口函数的声明及入口参数 LEDx（用于表明控制哪一个指示灯）的枚举数据类型说明，因此需要包含 BSP_LED.h 文件。我们将该文件添加到 main.h 文件的头文件添加代码位置，main.c 文件自动会包含 main.h 文件，这样就间接包含了 BSP_LED.h。代码片段如下：

```
/* Private includes ----------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "BSP_LED.h"            // 包含 LED 模块的头文件
/* USER CODE END Includes */
```

(2) 修改初始化代码

指示灯的初始化可以采用两种方式实现：一种是利用 CubeMX 软件生成的初始化函数 MX_GPIO_Init；另一种是调用 LED 模块提供的指示灯初始化接口函数 BSP_LED_Init。用户可以任选其一来完成指示灯的初始化。

方法一：利用 CubeMX 软件

该方法同前面设置指示灯控制引脚一样，在 CubeMX 软件的 GPIO 配置窗口中设置引脚的初始化电平、工作模式、上拉/下拉电阻、输出速度和引脚名称等参数，应用程序中自动调用初始化函数 MX_GPIO_Init 完成指示灯控制引脚的初始化。

方法二：利用指示灯初始化函数

该方法不需要 CubeMX 软件的参与，利用 BSP_LED_Init 完成指示灯控制引脚的初始化。如果有多个指示灯时，则多次调用该函数。添加的位置在/* USER CODE BEGIN 2 */和/* USER CODE END 2 */之间。代码片段如下：

```
/* Initialize all configured peripherals */
//MX_GPIO_Init();        // 屏蔽 CubeMX 生成的初始化代码
/* USER CODE BEGIN 2 */
BSP_LED_Init(LED0);      // 调用指示灯模块的初始化接口函数
/* USER CODE END 2 */
```

(3) 利用接口函数控制指示灯

在 while 循环中调用指示灯开启函数 BSP_LED_On 和指示灯关闭函数 BSP_LED_Off 来改变指示灯的开启和关闭，并调用延时函数 HAL_Delay 形成闪烁的效果。具体程序如程序清单 6-18 所示。

程序清单 6-18　调用 LED 模块接口函数控制指示灯

```
1. /* Infinite loop */
2. /* USER CODE BEGIN WHILE */
3. while (1)
4. {
5.   /* USER CODE END WHILE */
6.   /* USER CODE BEGIN 3 */
7.   // 调用 LED 模块的接口函数控制指示灯
8.   BSP_LED_On(LED0);    // 开启 LD2
9.   HAL_Delay(1000);     // 延时 100m0s
10.  BSP_LED_Off(LED0);;  // 关闭 LD2
11.  HAL_Delay(1000);     // 延时 100m0s
12. }
13.  /* USER CODE END 3 */
```

程序解析

① 第 8 行调用指示灯模块提供的指示灯开启函数 BSP_LED_On(LED0)，开启 LED0 参数对应的指示灯，即控制引脚 PA5 输出高电平，开启 LD2。

② 第 10 行调用指示灯模块提供的指示灯关闭函数 BSP_LED_Off(LED0)，关闭 LED0 参数对应的指示灯，即控制引脚 PA5 输出低电平，关闭 LD2。

我们将使用基于 BSP 模式设计的应用程序和直接调用 HAL 库的应用程序相比，可以发现，前者的程序更简练，可读性更强。如果硬件平台发生变化，主程序几乎不用修改，只需要修改底层的 BSP 程序。读者可以尝试将这段代码移植到 NucleoF103 开发板或其他厂商

的开发板上，来验证一下程序的移植过程。

你的任务

借鉴指示灯模块的设计方法，尝试编写按键模块的 BSP 驱动程序：BSP_Button.c 和 BSP_Button.h，提供以下接口函数：

- 按键初始化函数：BSP_Button_Init
- 按键状态读取函数：BSP_Button _GetState

6.6 本章小结

本章重点介绍了 GPIO 模块的电路结构、相关寄存器、HAL 库相关的数据类型和接口函数。采用循序渐进的方式，从寄存器编程入手，逐渐过渡到 HAL 库的编程。

本章有两个重要的设计思想：

1. 库函数设计思想

通过构造外设初始化数据类型，设计初始化接口函数，屏蔽用户对硬件寄存器的学习，转而学习数据类型及操作数据类型的接口函数。从本质上讲，其设计思想和面向对象的设计思想很相似：将数据以及对数据的操作方法封装在一起，作为一个相互依存的整体。

2. 硬件抽象层设计思想

采用硬件抽象层的方式设计外围电路的驱动是嵌入式系统设计中常用的设计思想。这种方式屏蔽了硬件电路上的差异，为应用层提供了统一的接口。发生硬件电路变化时，只需要修改硬件抽象层，不需要修改应用程序。这种方式提高了程序的可移植性，加快了程序开发的效率。

这些设计思想需要读者深入体会，并应用到自己的编程实践中。

习题 6

1. 编写流水灯程序。假定有四个指示灯 LED0～LED3，分别由 PA1、PB1、PC1 和 PD1 等 4 个引脚控制。设计两种形式的流水灯。第一种是逐个点亮：LED0 → LED1 → LED2 → LED3 → LED0；第二种是逐次增加： LED0 → LED0 LED1 → LED0 LED1 LED2 → LED0 LED1 LED2 LED3，然后全部熄灭后再循环。时间间隔设置为 500ms。
2. 利用指示灯模块的 BSP 驱动程序，改写习题 1 中的流水灯程序。
3. 借鉴指示灯模块的设计方法，编写按键模块的 BSP 驱动程序，提供按键初始化函数和按键状态读取函数。
4. 利用按键模块的 BSP 驱动程序，改写进阶任务和挑战任务中的按键程序。
5. 借鉴 HAL 库的 GPIO 模块源代码，自行编写 GPIO 的库函数，提供以下接口函数：
   - 设置引脚方向函数　　GPIO_SetDir
   - 控制引脚输出高电平　　GPIO_SetPin
   - 控制引脚输出低电平　　GPIO_ClrPin
   - 从指定引脚读取引脚状态　　GPIO_ReadPin

第七章　外部中断

本章导读

本章主要介绍 STM32 的外部中断。先介绍中断的基本概念和 STM32 的中断系统，然后以外部中断为例，详细分析了 HAL 库对中断的处理过程。最后介绍了 HAL 库外部中断的数据类型和接口函数，并通过多个任务实例来掌握接口函数的用法。

本章知识点

中断的基本概念和作用

STM32 中断系统

HAL 库中断处理流程

外部中断的 HAL 库数据类型及接口函数

7.1　中断概述

在计算机系统设计中，处理器常常需要与外部设备进行数据的传输，用于了解外部设备的状态或者控制外部设备的运行。常见的数据传输方式有以下 4 种：

1.　无条件方式

处理器不需要了解外部设备状态，直接进行数据传输，适用于指示灯和按键等简单设备。

2.　查询方式

常用于处理器与慢速外部设备交换数据。处理器与外部设备传输数据之前，先检查外设状态，如果外设处于“准备好”状态（输入设备）或“空闲”状态（输出设备），才进行数据传输，否则将一直查询。在查询过程中处理器无法执行其他任务，利用率较低。

3.　中断方式

处理器不主动查询外部设备的状态，而是让外部设备在数据准备好之后，再通知处理器。这样，处理器在没接到外部设备通知前只管做自己的事情，只有接到通知时才执行与外部设备的数据传输工作。在中断方式中，处理器和外部设备之间可以并行工作，从而大大提高微处理器的利用率。

4.　直接存储器访问方式（DMA）

DMA 方式是在处理器内部建立片内外设和内存之间的数据传输通道，传输过程不需要处理器参与。DMA 方式由硬件实现，特别适合批量数据传输的场合。

7.1.1　中断相关概念

1.　中断的定义

中断是计算机中的一个处理过程。当 CPU 在处理某一事件 A 时，发生了另一事件 B，请求 CPU 迅速去处理（中断发生）；CPU 接到请求后，如果同意该请求，就暂时中断当前的工作，转去处理事件 B；待 CPU 将事件 B 处理完毕后，再回到原来事件 A 被中断的地方继续处理事件 A（中断返回），这一过程称为中断。

在这个过程中：提出请求的事件 B，称为中断源；当前正在执行的事件 A 称为主程序；同意事件 B 的请求，称为开放中断。处理事件 B 的过程称为中断响应和中断处理；事件 A 被中断的地方称为断点。实现上述中断功能的部件称为中断系统。

中断过程的示意图如图 7-1 所示。

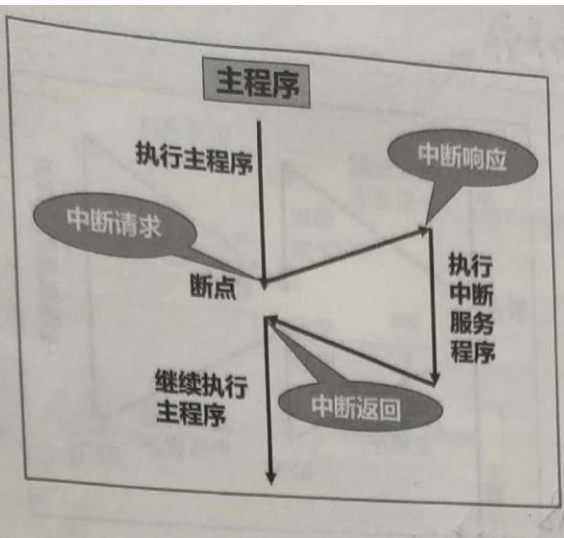


图 7-1　中断过程示意图

2. 中断的作用

最初，中断技术引入计算机系统，只是为了解决快速的 CPU 与慢速的外部设备之间传送数据的矛盾。例如：打印输出时，CPU 传送数据的速度较快，而打印机打印的速度较慢。如果不采用中断技术，CPU 将经常处于等待状态，效率极低。采用中断技术后，CPU 可以正常进行其它的工作，只在打印机打印完当前数据，并发出中断请求之后，才予以响应。这时，CPU 暂时停止执行当前程序而转去执行向打印机传送数据的程序，传送完成后又返回原来的程序执行，从而大大地提高了处理器的工作效率。

随着计算机技术的应用，人们发现中断技术不仅解决了 CPU 和外部设备的数据传送问题，而且还具有如下优点：

- 分时操作

  CPU 可以分时为多个外部设备服务，提高了计算机的利用率。

- 实时响应

  CPU 能够及时处理应用系统的随机事件，系统的实时性大大增强。

- 可靠性高。

  CPU 具有处理设备故障及掉电等突发性事件能力，从而使系统可靠性提高。

3. 中断优先级和中断嵌套

在实际应用中，处理器需要处理多个来自不同中断源的中断申请，不同中断源的中断申请在紧迫度上可能不尽相同。因此处理器需要在系统中设置不同中断申请的优先等级，也就是中断优先级。

在某一个时刻有多个中断源同时发出中断申请时，处理器只处理其中优先级最高的中断源。当处理器正在运行某一个中断服务程序期间出现了另一个中断源的申请时。如果后者的优先级低于前者，处理器将不予理睬。反之，处理器应立即响应后者,这种方式也称为中断的嵌套。中断嵌套的执行过程如图 7-2 所示。

4. 中断向量表

在响应一个特定中断的时候，处理器会执行一个函数，该函数一般称为中断处理程序(interrupt handler)或中断服务程序(interrupt service routine，ISR)。需要响应的每一类中断都编写有相应的中断服务程序，并预先装入内存。中断服务程序在内存中的入口地址称为中断向量。把系统中所有的中断向量集中起来放到存储器的某一区域内，这个存放中断向量的存储区就叫中断向量表或中断服务程序入口地址表。

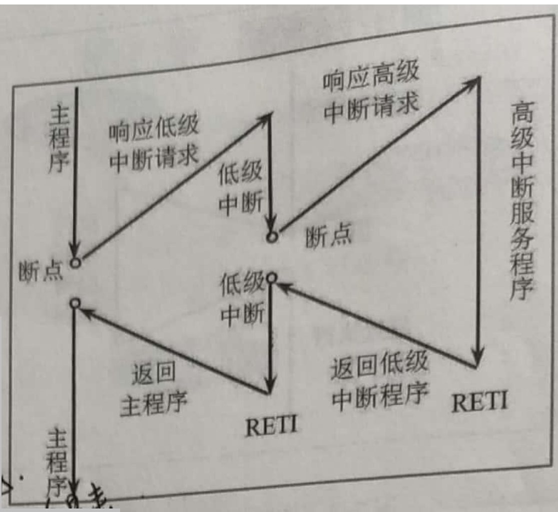


图 7-2 中断嵌套示意图

为了便于在中断向量表中找到中断向量，计算机系统会对每一个中断源进行编号，这个号码称为中断类型号。已知一个中断类型号，通过两次地址转换（中断类型号到中断向量表地址；中断向量表地址到中断处理程序入口地址）后，处理器就可转入中断服务程序开始执行相应的中断处理。

5. 中断响应过程

了解上述的中断基本概念后，我们可以总结出中断响应的完整步骤：

(1) 中断源发出中断请求；

(2) 判断处理器是否允许中断，以及该中断源是否被屏蔽；

(3) 优先级排队；

(4) 处理器暂定当前程序，保护断点地址和处理器的当前状态，根据中断类型号，查找中断向量表，转到对应的中断服务程序；

(5) 执行中断服务程序；

(6) 恢复被保护的状态，执行中断返回指令，回到被中断的程序。

7.1.2 STM32 中断系统

1. 嵌套向量中断控制器

STM32 把能够打断当前代码执行流程的事件分为异常(exception)和中断(interrupt)两类。二者的区别在于：中断是由内核外部产生的，通过中断通道送入内核，一般由硬件引起，比如定时器中断和外部中断等。而异常通常是内核自身产生的，大多是软件引起的，比如除法出错异常，预取值失败等。

STM32 使用嵌套向量中断控制器（NVIC）来进行中断管理。NVIC 属于 Cortex-M 内核的组成单元，所有基于 Cortex-M 内核设计的微控制器内部都集成了该单元。

NVIC、内核以及片内外设之间的关系如图 7-3 所示。

2. STM32 中断优先级设置

在 STM32 中，所有的异常和中断都通过中断向量表管理。优先级从-3~6 的中断向量称为系统异常，用户不能修改优先级，如用于时钟安全系统（CSS）的不可屏蔽中断（NMI）。优先级从 7 开始的称为内核外中断，用户可以设置优先等级。

STM32 将优先等级又细分出两个优先级：抢占优先级（preempition priority）和子优先级（sub priority）。

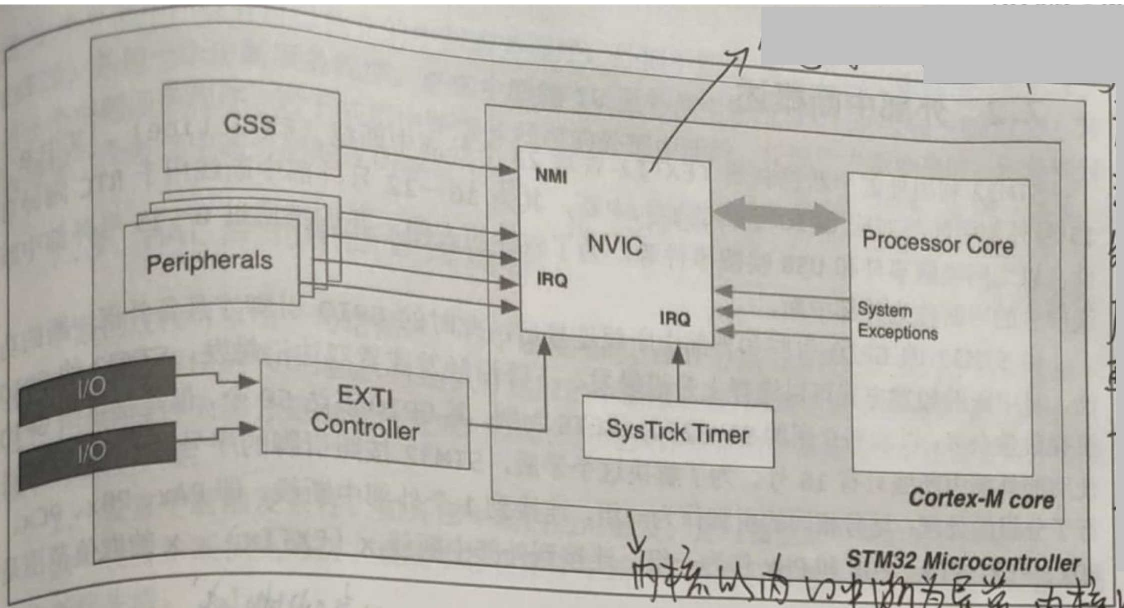


图 7-3　NVIC、内核及片内外设之间的关系

内核以内的中断为异常．内核以外的中断为外部中断．

在判断一个中断的优先级时，需要综合考虑这两个优先级。当多个中断同时提出中断申请时，抢占优先级高的中断会优先得到执行。如果抢占优先级相同，则比较子优先级，子优先级高的中断会优先得到执行。如果抢占优先级和子优先级都相同的话，就比较他们的中断编号，编号越小，优先级越高。所有异常和中断的中断类型编号都定义在与芯片型号对应的头文件中，如 stm32f411xe.h，以枚举类型 IRQn_Type 进行了说明，感兴趣的读者可以自行查阅。

在具体设置中断优先级时，是通过中断优先级寄存器中的高 4 位完成。这 4bit 的中断优先级可以分成 5 组：

- 第 0 组：所有 4bit 用于指定抢占优先级。
- 第 1 组：最高 1 位用于指定抢占优先级，后面 3 位用于指定子优先级。
- 第 2 组：最高 2 位用于指定抢占优先级，后面 2 位用于指定子优先级。
- 第 3 组：最高 3 位用于指定抢占优先级，后面 1 位用于指定子优先级。
- 第 4 组：所有 4 位用于指定抢占优先级。

不同分组情况下，抢占优先级和子优先级的等级划分如表 7-1 所示。

表 7-1　STM32 中断优先级分组

优先级分组	抢占优先级	子优先级
第 4 组：NVIC_PriorityGroup_4	4 位/16 级	0 位/0 级
第 3 组：NVIC_PriorityGroup_3	3 位/8 级	1 位/2 级
第 2 组：NVIC_PriorityGroup_2	2 位/4 级	2 位/4 级
第 1 组：NVIC_PriorityGroup_1	1 位/2 级	3 位/8 级
第 0 组：NVIC_PriorityGroup_0	0 位/0 级	4 位/16 级

在 HAL 库的初始化过程中，HAL 库初始化函数 HAL_Init 将优先级分组设置为第 4 组，即有 0~15，共 16 级抢占式优先级，没有子优先级。编号越小的优先级越高：0 号为最高，15 号为最低。

7.2 外部中断概述

STM32 利用外部中断控制器（EXTI）管理 23 个外部中断线（EXTI Line），其中 0～15 号外部中断线对应 GPIO 引脚的输入中断，其余 16～22 号外部中断线用于 RTC 闹钟事件、以太网唤醒事件和 USB 唤醒事件等。为了叙述的方便，我们把通过 0～15 号外部中断线产生的中断称为外部中断。

当 STM32 的 GPIO 引脚和外部中断线连接后，此时的 GPIO 引脚才具备外部中断的功能，且中断的触发方式可以选择上升沿触发、下降沿触发或者双边沿触发。STM32 的 GPIO 引脚数量众多，以本书介绍的 STM32F411RET6 为例，其 GPIO 多达 50 个，但是提供给 GPIO 使用的外部中断线只有 16 号。为了解决这个矛盾，STM32 按照引脚的序号对 GPIO 引脚进行了分组的处理：尾号相同的引脚作为一组，连接到 1 个外部中断线，即 PAx、PBx、PCx、PDx、PEx、PFx、PGx 和 PHx 作为一组，连接到外部中断线 x（EXTIx），x 的取值范围是 0～15。

GPIO 引脚和外部中断线的连接关系如图 7-4 所示。

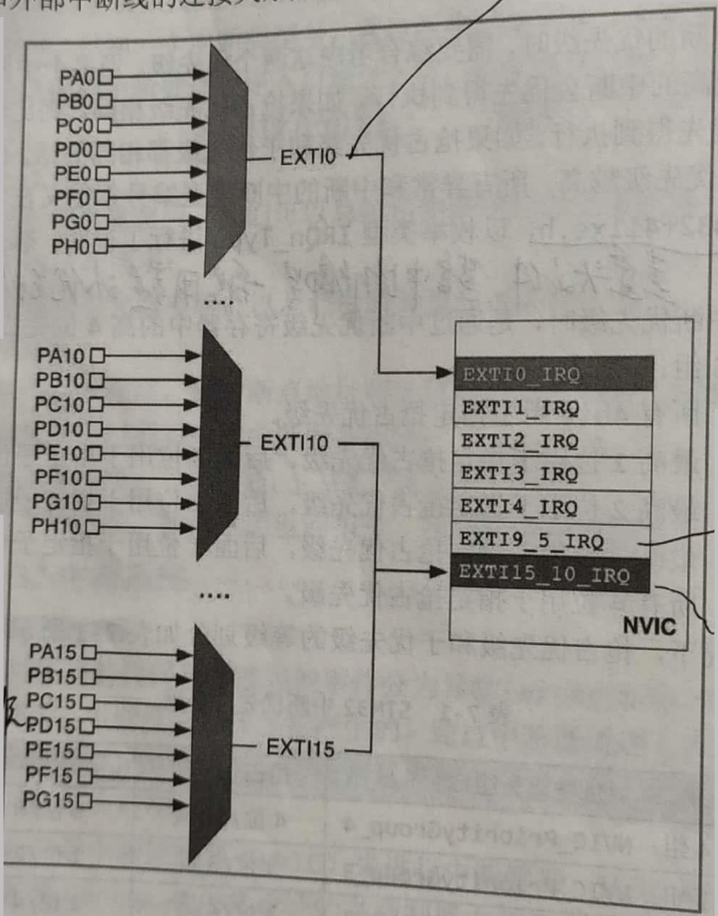


图 7-4 GPIO 与外部中断线的映射关系

由于 GPIO 引脚按组进行了分类，同组的引脚只能使用其中 1 个与外部中断线连接，作为外部中断引脚。例如，PA0、PB0、PC0、PD0、PE0、PF0、PG0 和 PH0 这些引脚作为 1 组。如果我们使用 PA0 引脚作为外部中断引脚，那么其他引脚就不能作为外部中断引脚使用。因此从本质上讲，可供用户同时使用的外部中断引脚最多只有 16 个。

在图 7-2 中还可以看出，16 个外部中断线中，外部中断线 0（EXTI0）、外部中断线 1（EXTI1）、外部中断线 2（EXTI2）、外部中断线 3（EXTI3）和外部中断线 4（EXTI4）

这 5 个外部中断线有自己独立的中断服务程序。外部中断线 5 至外部中断线 9（EXTI5～EXTI9）共用一个中断服务程序。外部中断线 10 至外部中断线 15（EXTI10～EXTI15）共用一个中断服务程序。对于共用中断服务程序的外部中断线，在进行中断处理时，还需要判断是哪一个外部中断引脚触发的本次中断。

7.3 HAL 库的中断处理流程

中断的过程可以用一句话描述：当中断发生时，MCU 暂停当前程序，进入中断向量表中查找中断服务程序的入口地址，进而执行相应的中断服务程序，最后返回断点处执行原程序。虽然上述的操作大部分是由硬件完成，但是在针对具体外设的中断处理中，还需要用户完成以下的编程步骤：

1. 设置中断触发条件。如外部中断的边沿触发，定时器更新中断的时间间隔等。
2. 设置中断优先等级。系统中存在多个中断时，需要根据任务的紧迫程度，设置不同的中断优先级。
3. 使能外设的某个中断。MCU 片内集成的外设一般都有多个中断源，这些中断源的使能由每个外设的寄存器控制。例如，串口通信的发送数据寄存器空中断 TXE 和发送完成中断 TC，就是由串口控制寄存器 CR1 中的 TXEIE 位和 TCIE 位控制。
4. 编写中断服务程序。在启动文件 startup_stm32fxxx.s（xxx 表示芯片型号）文件中预先为每个中断编写了一个中断服务程序，只是这些中断函数都是死循环的，目的只是初始化中断向量表。因此实际的中断服务程序还需要用户重新编写，统一存放在 stm32fxxx_it.c 文件中。用户在编写中断服务程序时还需要注意：中断服务程序的函数名必须和启动文件中定义的中断服务程序的函数名一致，否则将不能正确跳转到用户编写的中断服务程序中。
5. 清除中断标志。发生中断后，相应的中断标志位置位，以便 MCU 查询。这些中断标志在中断处理过程要注意及时清除，避免重复进入中断。

上述的 5 个中断编程步骤中，借助 CubeMX 可以以图形化的设置方式完成前 3 个步骤，而第 4 和第 5 个步骤则可以借助 HAL 库进一步简化。

HAL 库将中断进行了重新封装，并做了如下的处理：

- 统一规定处理各个外设的中断处理程序 HAL_PPP_IRQHandler（PPP 代表外设名称）。
- 在中断处理程序 HAL_PPP_IRQHandler 完成中断标志的判断和清除。
- 将具体的中断处理任务以回调函数的形式提供给用户。

【小贴士】回调函数

回调函数是一个通过函数指针调用的函数。如果把函数的指针（地址）作为参数传递给另一个函数，当这个指针被用来调用其所指向的函数时，我们就称该函数为回调函数。

回调函数不是由该函数的实现方直接调用，而是在特定的事件或条件发生时由另外的一方调用的，用于对该事件或条件进行响应。

下面我们以外部中断为例，来详细分析一下 HAL 库的中断处理过程。

假设微控制器芯片为 STM32F411，设置 PC0 和 PC13 引脚为外部中断，使能对应的外部中断：外部中断线 0 和外部中断线 10～15。当 PC0 或 PC13 出现脉冲边沿时，将触发外部中断，具体的执行过程如下：

(1) 微控制器暂定当前运行的程序，由内部的硬件机制跳转到外部中断线 0 所对应的中断服务程序 EXTI0_IRQHandler 执行或者外部中断线 10～15 所对应的中断服务程序 EXTI15_10_IRQHandler 执行，具体执行哪一个中断服务程序由中断发生的先后顺序及中断的优先级决定。启动文件 startup_stm32f411xe.s 的中断向量表中预先定义了所有外部中断线对应的中断服务程序的函数名称，如表 7-2 所示。这些定义的函数都加入了 weak 属性。weak 属性的函数表示如果该函数没有在其他文件中定义，则使用该函数。如果用户定义了该函数，则使用用户定义的函数。

表 7-2 外部中断对应的中断服务程序

外部中断线	中断服务程序的函数名称
外部中断线 0(EXTI Line 0)	EXTI0_IRQHandler
外部中断线 1(EXTI Line 1)	EXTI1_IRQHandler
外部中断线 2(EXTI Line 2)	EXTI2_IRQHandler
外部中断线 3(EXTI Line 3)	EXTI3_IRQHandler
外部中断线 4(EXTI Line 4)	EXTI4_IRQHandler
外部中断线 5～9(EXTI Line[9:5])	EXTI9_5_IRQHandler
外部中断线 10～15(EXTI Line[15:10])	EXTI15_10_IRQHandler

(2) 启动文件 startup_stm32f411xe.s 中定义的中断服务程序默认为死循环。由于这些函数定义为 weak 属性，因此用户可以在其他文件中重新定义与该函数同名的中断服务程序。HAL 库专门提供了一个 stm32f4xx_it.c 文件用于存放用户定义的中断服务程序。这样，当中断发生时，将转入在 stm32f4xx_it.c 文件中预先定义的中断服务程序中。该文件的内容由 CubeMX 生成，如程序清单 7-1 所示。

进入中断服务程序 EXTI0_IRQHandler 或者 EXTI15_10_IRQHandler 后，都将调用 HAL 库提供的外部中断入口函数 HAL_GPIO_EXTI_IRQHandler，该函数作为所有外部中断发生后的统一入口。

程序清单 7-1 外部中断服务程序调用 HAL_GPIO_EXTI_IRQHandler

```
/*
 * @brief This function handles EXTI line0 interrupt.
 */
void EXTI0_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_0);    // 调用统一的中断入口函数
}
/*
 * @brief This function handles EXTI line[15:10] interrupts.
 */
void EXTI15_10_IRQHandler(void)
{
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_13);   // 调用统一的中断入口函数
}
```

(3) 外部中断入口函数 HAL_GPIO_EXTI_IRQHandler 是 HAL 库提供的外部中断接口函数，该函数在 stm32f4xx_hal_gpio.c 文件中定义。函数内部将判断外部中断是由哪一号外部中断线产生的，并且清除对应的中断标志。最后调用外部中断回调函数 HAL_GPIO_EXTI_Callback，将产生外部中断的 GPIO 引脚 GPIO_Pin 作为入口参数传入。中断入口函数 HAL_GPIO_EXTI_IRQHandler 的源代码如程序清单 7-2 所示。

程序清单 7-2　外部中断处理函数 HAL_GPIO_EXTI_IRQHandler 源代码

```
/*
 * @brief  This function handles EXTI interrupt request.
 * @param  GPIO_Pin Specifies the pins connected EXTI line
 * @retval None
 */
void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
{
  /* EXTI line interrupt detected */
  if(__HAL_GPIO_EXTI_GET_IT(GPIO_Pin) != RESET)  // 检测对应的中断标志是否置位
  {
    __HAL_GPIO_EXTI_CLEAR_IT(GPIO_Pin);          // 清除对应的中断标志
    HAL_GPIO_EXTI_Callback(GPIO_Pin);            // 调用回调函数执行中断任务
  }
}
```

(4) 外部中断回调函数 HAL_GPIO_EXTI_Callback 用于处理具体的中断任务。该函数也是 HAL 库提供的外部中断接口函数，在 stm32f4xx_hal_gpio.c 文件中被定义为 weak 属性，用户可以重新定义该函数。默认的外部中断回调函数没有任何可执行代码，仅仅有一个避免 GCC 编译器警告的语句：UNUSED(GPIO_Pin)。回调函数的源代码如程序清单 7-3 所示。

程序清单 7-3　外部中断回调函数 HAL_GPIO_EXTI_Callback 源代码

```
/*
 * @brief  EXTI line detection callbacks.
 * @param  GPIO_Pin Specifies the pins connected EXTI line
 * @retval None
 */
__weak void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  /* Prevent unused argument(s) compilation warning */
  UNUSED(GPIO_Pin);
  /* NOTE:This function Should not be modified, when the callback is needed,
           the HAL_GPIO_EXTI_Callback could be implemented in the user file
   */
}
```

(5) 用户需要在应用程序 main.c 中重新定义 HAL_GPIO_EXTI_Callback 函数，并编写具体的中断任务处理代码。由于所有的外部中断最终都会调用该函数，因此在函数内部需要根据入口参数 GPIO_Pin 判断是哪一个外部中断线产生的外部中断，再根据不同的外部中断执行不同的中断任务处理代码。由于要判断多个外部中断线，一般常常使用 switch-case 语句实现。示例代码如程序清单 7-4 所示。

程序清单 7-4　外部中断回调函数 HAL_GPIO_EXTI_Callback 编写示例

```
/* USER CODE BEGIN 4 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
  switch(GPIO_Pin)
  {
    case GPIO_PIN_0:                          // 外部中断线 0 对应的中断任务
         /* GPIO_PIN_0 EXTI handling */
         break;
    case GPIO_PIN_13:
         /*GPIO_PIN_13 EXTI handling */      // 外部中断线 13 对应的中断任务
         break;
    case GPIO_PIN_x:                          // 外部中断线 x 对应的中断任务
        /* GPIO_PIN_x EXTI handling */
        break;
    ......
    default:  break;
  }
}
/* USER CODE END 4 */
```

整个外部中断的处理流程可以用图 7-5 来表示。

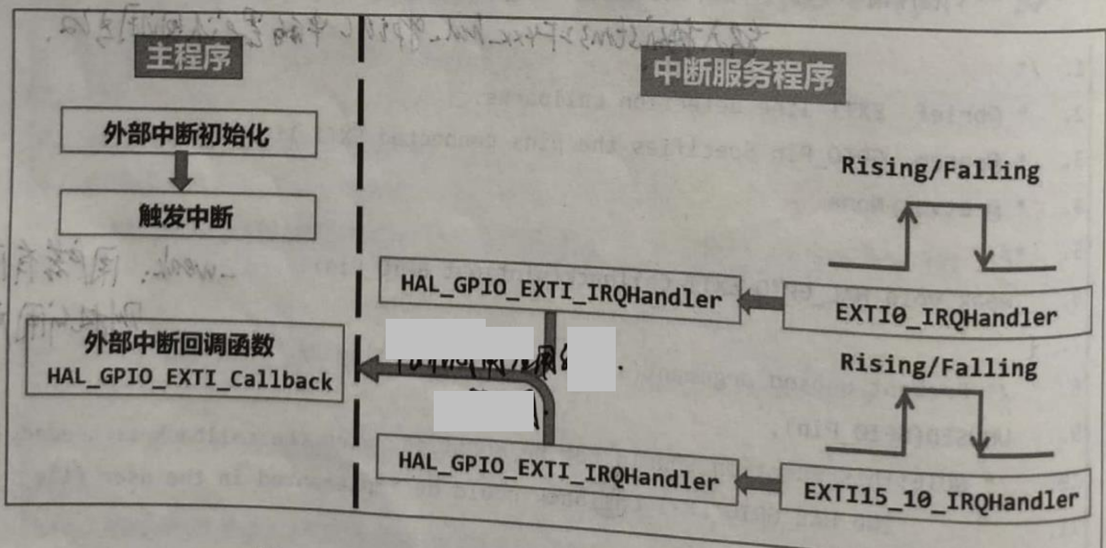


图 7-5　从函数调用角度看 HAL 库的外部中断处理流程

如果从发生中断后，函数调用的时间角度分析，我们可以勾勒一个更加清晰的中断处理流程，如图 7-6 所示。

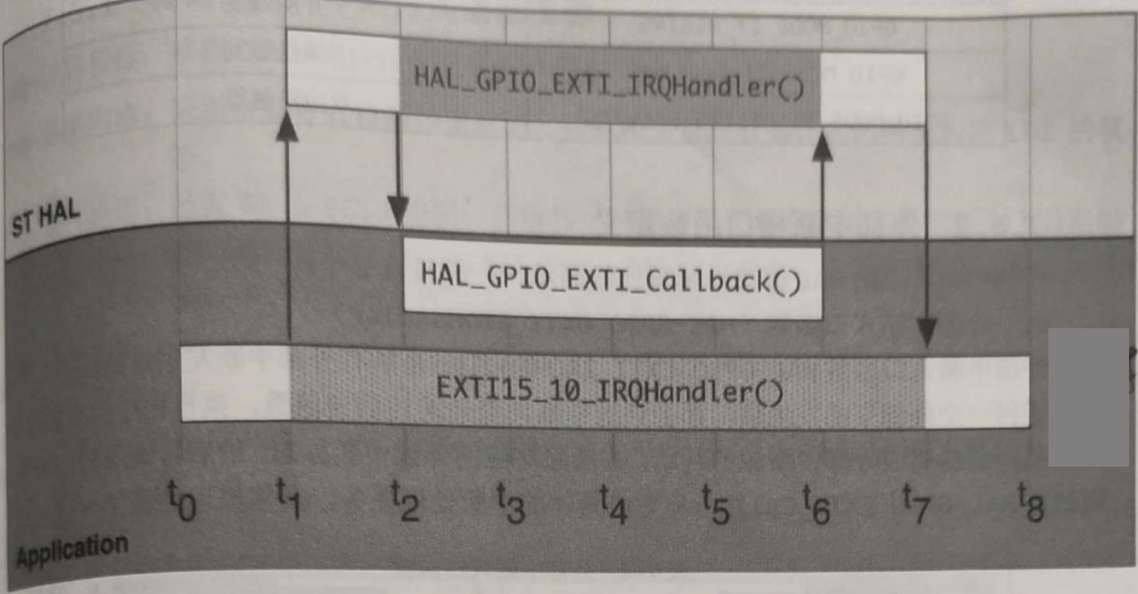


图 7-6　从时间角度看 HAL 库的外部中断处理流程

图 7-6 可以划分为上下两个部分，上半部分是由 HAL 库处理，下半部分则是用户程序。

假设在 t1 时刻发生了外部中断，进入到中断服务程序 EXTI15_10_IRQHandler，该函数将调用 HAL 库提供的外部中断入口函数 HAL_GPIO_EXTI_IRQHandler。该函数执行到 t2 时刻时，将调用用户编写的外部中断回调函数 HAL_GPIO_EXTI_Icallback 去执行具体的中断处理任务，执行时间从 t2～t6。当回调函数执行完中断处理任务后，返回到外部中断处理函数 HAL_GPIO_EXTI_IRQHandler，该函数执行到 t7 时刻时，返回到最初的中断服务程序 EXTI15_10_IRQHandler，执行中断返回的相关操作，在 t8 时刻返回主程序。

在函数的调用过程中，函数 HAL_GPIO_EXTI_Icallback 是放在 main.c 文件中，作为应用程序的部分，而它的调用却是由中断触发的，因此称为回调函数。

从上述的外部中断流程分析可以看到，借助 CubeMX 软件和 HAL 库，用户的中断编程步骤大大简化。只需要利用 CubeMX 软件进行中断的相关配置，包括中断触发条件、中断优先级、使能中断等。最后在中断回调函数中编写与中断任务处理相关的代码即可。

7.4　外部中断的 HAL 库定义

由于外部中断都是利用 GPIO 引脚实现，因此与外部中断相关的 HAL 库定义都放在 GPIO 外设文件中：外部中断的数据类型定义放在 stm32f4xx_hal_gpio.h 文件中，外部中断的接口函数放在 stm32f4xx_hal_gpio.c 文件中。

7.4.1　外部中断数据类型定义

外部中断的数据类型定义比较简单，主要是外部中断触发类型的宏定义。这些宏常量在使用时，作为引脚初始化数据类型 GPIO_InitTypeDef 中成员变量 Mode 的取值范围，用来选择外部中断的触发方式：上升沿触发、下降沿触发或者双边沿触发，具体定义如表 7-3 所示。

表 7-3　外部中断触发类型 Mode

宏常量定义	含义
GPIO_MODE_IT_RISING	上升沿触发
GPIO_MODE_IT_FALLING	下降沿触发
GPIO_MODE_IT_RISING_FALLING	双边沿触发

7.4.2　外部中断接口函数定义

外部中断对应的 HAL 库接口函数比较简单，一共只有 2 个：

1.　外部中断入口函数（HAL_GPIO_EXTI_IRQHandler）

外部中断入口函数 HAL_GPIO_EXTI_IRQHandler 是所有外部中断发生后的统一入口函数。任何一个外部中断发生后，都会通过中断向量表跳转到该函数，执行相应的中断处理。该函数内部会完成外部中断源的判断，并清除对应的外部中断标志，最后再调用外部中断回调函数 HAL_GPIO_EXTI_Callback 完成具体的中断处理任务。具体描述如表 7-4 所示。

所有外部中断均自动调用此函数
用户看不见此函数

表 7-4　外部中断入口函数

接口函数：HAL_GPIO_EXTI_IRQHandler	
函数原型	void HAL_GPIO_EXTI_IRQHandler(uint16_t GPIO_Pin)
功能描述	所有外部中断发生后的统一入口函数
入口参数	GPIO_Pin：连接到对应外部中断线的引脚，范围是 GPIO_PIN_0～GPIO_PIN_15
返回值	无
注意事项	1. 所有外部中断服务程序均调用该函数完成中断处理 2. 函数内部根据 GPIO_Pin 的取值判断中断源，并清除对应外部中断线的中断标志 3. 函数内部调用外部中断回调函数 HAL_GPIO_EXTI_Callback 完成实际的处理任务 4. 该函数由 CubeMX 自动生成

2.　外部中断回调函数（HAL_GPIO_EXTI_Callback）

外部中断回调函数 HAL_GPIO_EXTI_Callback 主要用于处理具体的中断任务，即发生该中断后应该完成的具体操作。该函数作为 16 个外部中断发生时统一的任务处理函数，在函数内部需要先判断是哪一个外部中断，然后执行对应的中断任务处理程序。具体描述如表 7-5 所示。

表 7-5　外部中断回调函数

接口函数：HAL_GPIO_EXTI_Callback	
函数原型	void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
功能描述	外部中断回调函数，用户在该函数内编写实际的中断任务程序
入口参数	GPIO_Pin：连接到对应外部中断线的引脚，范围是 GPIO_PIN_0～GPIO_PIN_15 (16个中
返回值	无
注意事项	1. 该函数由 HAL_GPIO_EXTI_IRQHandler 函数调用，完成所有外部中断的任务处理 2. 函数内部根据 GPIO_Pin 的取值判断中断源，然后执行对应的的中断任务程序 3. 该函数由用户根据实际任务编写

7.5 任务实践

7.5.1 基础任务：中断方式读取按键

➡ 任务目标：掌握 CubeMX 软件配置外部中断的方法

➡ 任务内容：采用中断方式检测按键状态，按键按下后执行操作：翻转指示灯 LD2 的状态。

➡ 硬件原理：按键 B1 由 PC13 控制：按键按下时为下降沿触发方式，释放时为上升沿触发方式。指示灯 LD2 由 PA5 控制：PA5 输出高电平，开启指示灯；PA5 输出低电平，关闭指示灯。电路原理图参见第二章的图 2-16 和图 2-18。

➡ 任务实现

1. 引脚分配

在芯片引脚窗口中选择 PA5 引脚为 GPIO_Output，PC13 引脚为 GPIO_EXTI13，如图 7-7 所示。

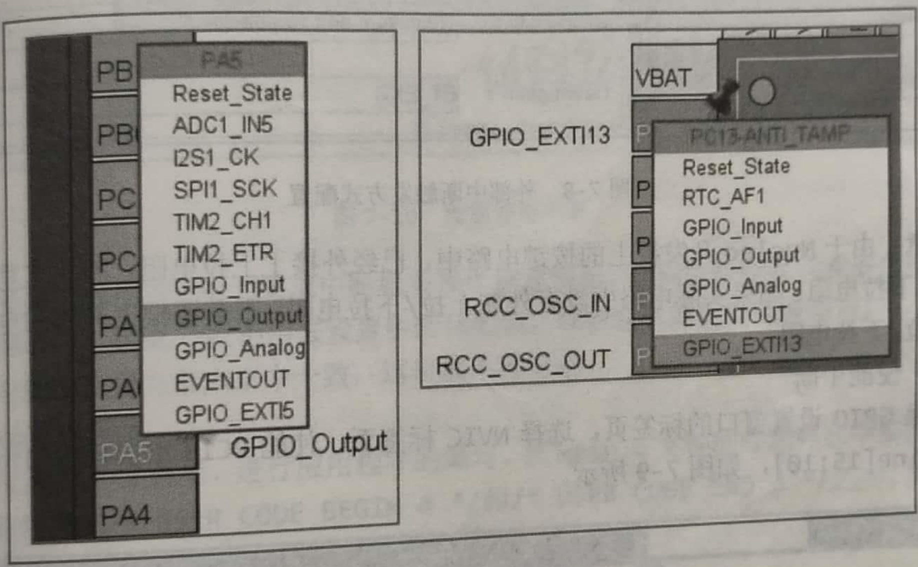


图 7-7 引脚分配

2. 外设配置

外部中断的外设配置与 GPIO 模块相比，除了完成参数配置以外，还增加使能中断以及设置中断优先级的步骤。

(1) 参数配置

在外设设置窗口中选择：SystemCore → GPIO，在出现的引脚配置窗口中，选择 GPIO 标签页。点击 PC13 引脚一栏，在展开的参数配置窗口中对 ~~PA13~~ PC13 作如下配置：

- GPIO mode（工作模式）：External Interrupt Mode with Rising edge trigger detection（上升沿触发）
- GPIO Pull-up and /Pull-down（上拉/下拉电阻）：No pull-up and no pull-down（无下拉/下拉电阻）
- User Label（引脚名称）：B1_EXTI

具体配置如图 7-8 所示。

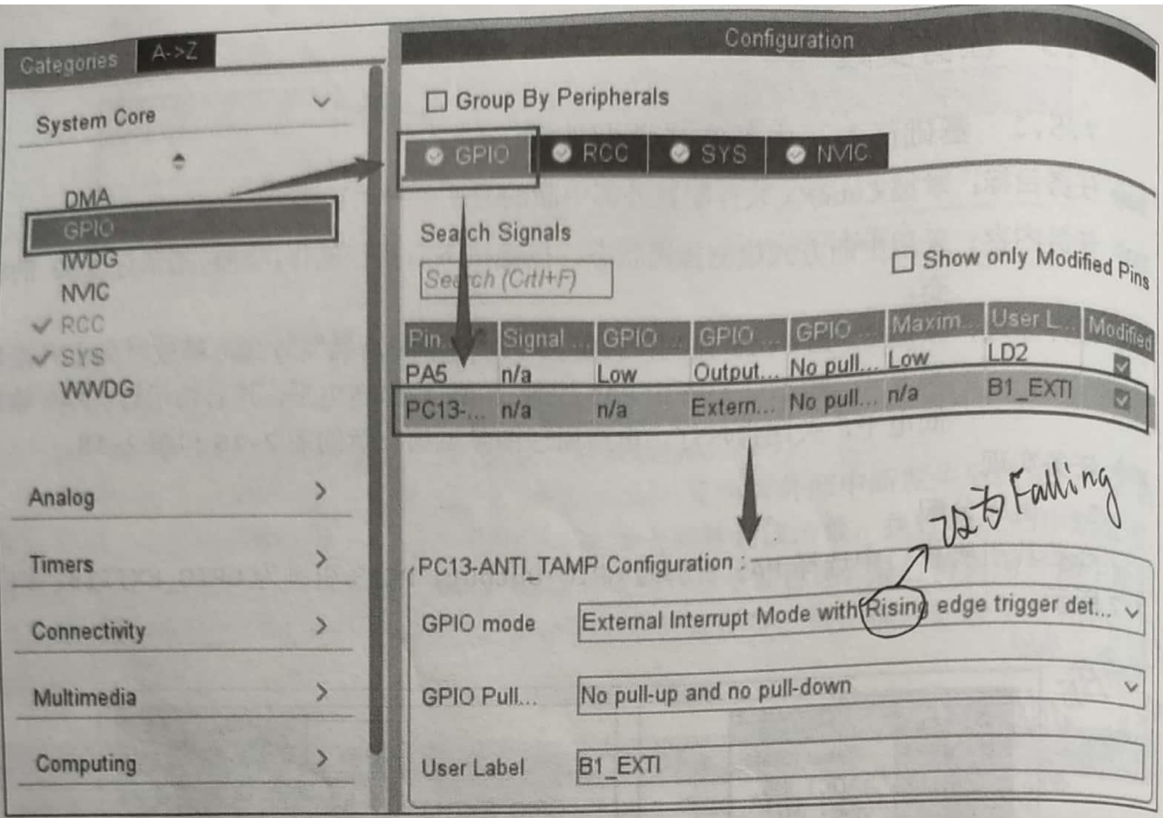


图 7-8　外部中断触发方式配置

注意：由于 Nucleo 开发板上的按键电路中，已经外接了上拉电阻，因此这里可以不开启上拉/下拉电阻。如果实际电路中没有外加上拉/下拉电阻，则需要根据具体的电路连接来选择上拉/下拉电阻。

(2) 使能中断

切换 GPIO 设置窗口的标签页，选择 NVIC 标签页，使能 PC13 引脚对应的外部中断线 EXTI line[15:10]，如图 7-9 所示。

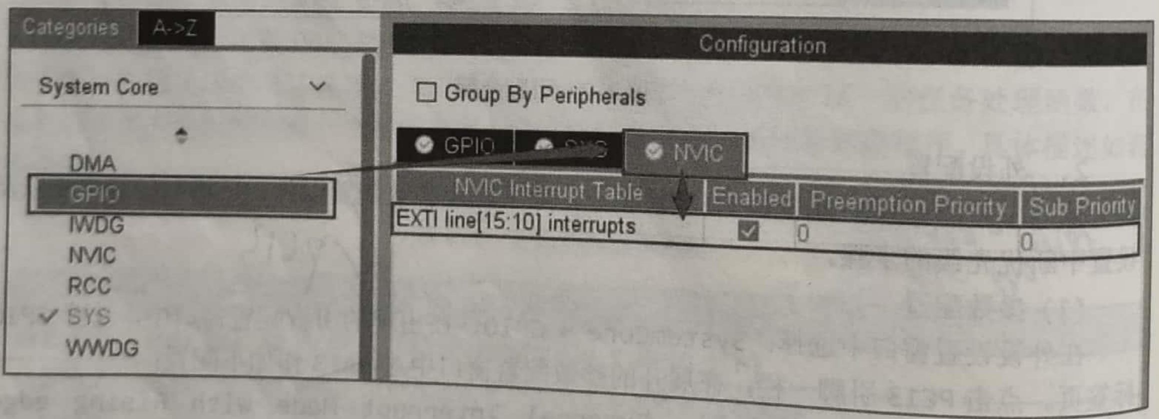


图 7-9　使能外部中断

(3) 设置中断优先级

由于在 HAL 初始化时，已经将中断优先级分组设置为第 4 组，即由 16 级抢占式优先级，0 级子优先级。系统节拍定时器 Systick 的抢占式优先级设置为 0，即最高优先级。这里我们将外部中断线 10～15 的抢占式优先级设置为 15，即最低优先级。

在外设设置窗口中选择：SystemCore → NVIC，在出现的配置窗口中，选择 NVIC 标签页。点击 EXIT Line[15:10] interrrupt 一栏，设置它的抢占优先级（Preemption priority）为 15，如图 7-10 所示。

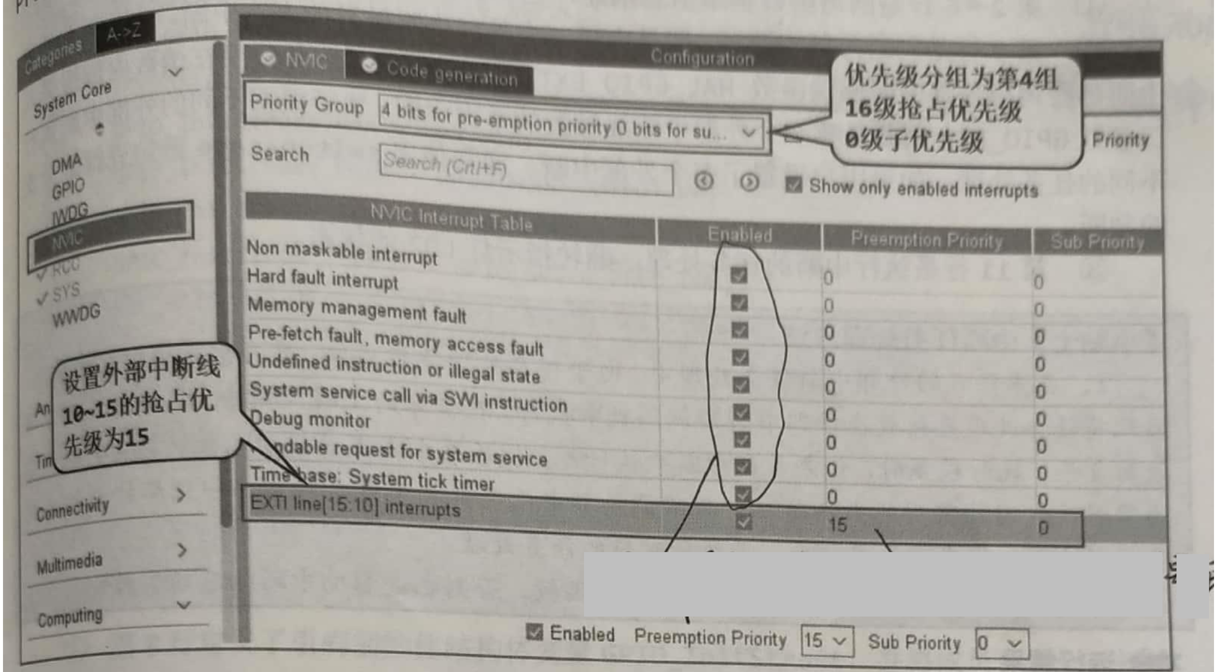


图 7-10　使能外部中断

在大多数的应用中，由于要求中断服务程序快进快出，出现中断嵌套或者多个中断同时到达的几率较小，因此一般不用去设置中断优先级，保持默认的优先等级即可。

PA5 引脚的配置与第六章中一致，这里就不再赘述。

3. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。在 main.c 文件的添加外部中断的回调函数，添加的位置在/* USER CODE BEGIN 4 */和/* USER CODE END 4 */之间。具体的程序如代码清单 7-5 所示。

程序清单 7-5　外部中断回调函数

```
/* USER CODE BEGIN 4 */
/*
 * @brief  EXTI line detection callbacks.
 * @param  GPIO_Pin: Specifies the pins connected EXTI line.
 * @retval None
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
   if( GPIO_Pin == B1_EXTI_Pin )                    // 判断外部中断源
   {
      HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin); // 翻转 LD2 状态
   }
}
```

```
14. /* USER CODE END 4 */
```

程序解析

① 第 2～6 行是回调函数基本信息描述，包括函数功能、入口参数和返回值。

② 第 9 行是判断外部中断源，即是由哪一个外部中断触发的本次中断。由于所有外部中断都会调用外部中断回调函数 HAL_GPIO_EXTI_Callback，因此需要在函数内部根据入口参数 GPIO_Pin 判断是哪一个外部中断线触发的本次中断。然后根据不同的外部中断执行不同的任务处理。如果用户设置了多个外部中断，通常使用 switch-case 语句进行多分支的判断。

③ 第 11 行是执行中断的任务处理：翻转指示灯 LD2 的状态。

> **【小贴士】中断任务处理方法**
>
> 1．在本任务的外部中断任务处理中，由于任务处理流程简单，只是翻转指示灯的状态，因此将任务处理直接放在外部中断回调函数中执行。在实际的工程应用中，当中断任务处理较为复杂，耗时较多时，如果放在中断中执行，会使后续中断无法及时响应，影响中断响应的实时性。因此常用的做法是在中断回调函数中设置标志变量，主程序中循环检测标志变量。当检测到标志变量置位后，再执行对应的任务处理。
>
> 2．在中断回调函数中一定不要调用延时函数，否则也会影响中断响应的实时性

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。可以观察到：按键 B1 每按下一次，指示灯 LD2 的状态就翻转一次。

在本任务中，PC13 引脚设置为外部中断后，其初始化代码由 CubeMX 按照图 7-3 和图 7-4 的设置自动生成。外部中断的初始化代码没有单独生成函数，而是在 GPIO 的初始化函数 MX_GPIO_Init 中完成，其初始化的过程和 GPIO 几乎一致。除了在初始化工作模式的成员变量 mode 时，取值必须为表 7-3 中定义的中断触发方式。具体代码如程序清单 7-6 所示。

程序清单 7-6 自动生成的外部中断初始化代码

```
/**
  * @brief GPIO Initialization Function
  * @param None
  * @retval None
  */
static void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
  /*Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
```

```
15. /*Configure GPIO pin : B1_EXTI_Pin */
16. GPIO_InitStruct.Pin = B1_EXTI_Pin;
17. GPIO_InitStruct.Mode = GPIO_MODE_IT_RISING;
18. GPIO_InitStruct.Pull = GPIO_NOPULL;
19. HAL_GPIO_Init(B1_EXTI_GPIO_Port, &GPIO_InitStruct);
20. /*Configure GPIO pin : LD2_Pin */
21. GPIO_InitStruct.Pin = LD2_Pin;
22. GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
23. GPIO_InitStruct.Pull = GPIO_NOPULL;
24. GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_LOW;
25. HAL_GPIO_Init(LD2_GPIO_Port, &GPIO_InitStruct);
26. /* EXTI interrupt init*/
27. HAL_NVIC_SetPriority(EXTI15_10_IRQn, 15, 0);
28. HAL_NVIC_EnableIRQ(EXTI15_10_IRQn);
29. }
30. /* USER CODE END 4 */
```

程序解析

① 第8行定义了引脚初始化结构体变量GPIO_InitStruct，并将成员变量全部初始化为0。

② 第10~12行使能对应GPIO端口的时钟。

③ 第16~19行是对PC13引脚的配置，此时PC13作为外部中断使用。按照图7-8的配置要求，向对应的成员变量写入初始化参数，方法和GPIO引脚类似，除了成员变量Mode设置为外部中断的触发方式。

④ 第19行调用初始化函数HAL_GPIO_Init将初始化参数写入对应的硬件寄存器，完成PC13引脚的配置。

⑤ 第20~25行是对PA5引脚的初始化，此时PA5作为GPIO引脚使用。

⑥ 第27行是中断优先级的设置，这里按照图7-10的中断优先级设置：将外部中断线10~15的抢占优先级设置为15，子优先级设置为0。

⑦ 第28行使能外部中断线EXTI line[15:10]，确保外部中断EXTI13能够响应。

你的任务

任务一：改变外部中断触发方式

将外部中断的触发方式改为上升沿触发，重新生成MDK工程并下载到Nucleo开发板上。注意观察按键B1按下和释放时，指示灯LD2状态的变化。与采用下降沿触发的外部中断相比，有什么不同？为什么会出现这样的差别？

任务二：GPIO引脚模拟外部中断

在基本任务中，外部中断是通过按键B1触发的，由于按键的机械抖动，会导致一次按键产生多个脉冲，从而可能触发多次中断，使指示灯LD2翻转多次。请使用一个GPIO引脚输出10Hz的方波来模拟外部中断信号，通过杜邦线与PC13引脚连接，PC13引脚设置为下降沿触发，在中断中执行翻转指示灯LD2状态的任务，请编写相应的程序。进一步思考，如果将PC13引脚的触发方式改为双边沿触发，指示灯LD2的翻转状态有什么变化？

7.5.2 进阶任务：按键控制指示灯闪烁频率

任务目标： 掌握 CubeMX 软件配置外部中断的方法

任务内容： 利用按键 B1 控制指示灯 LD2 的闪烁频率，闪烁频率设置为 3 档：初始状态时，LD2 按照 ~~2Hz~~ 1Hz 的频率闪烁；第一次按键后，LD2 按照 ~~10Hz~~ 5Hz 的频率闪烁；第二次按键后，LD2 按照 20Hz 的频率闪烁。再次按键后让 LD2 恢复以 2Hz 的频率闪烁，并重复上述过程。按键的检测采用中断方式。

硬件原理： 按键 B1 由 PC13 控制：按键按下时为下降沿触发方式，释放时为上升沿触发方式。指示灯 LD2 由 PA5 控制：PA5 输出高电平，开启指示灯；PA5 输出低电平，关闭指示灯。电路原理图参见第二章的图 2-16 和图 2-18。

任务实现

本任务所使用的引脚和外设与基础任务中所使用的相同，因此省去引脚分配和外设配置的过程。

1. 设计思路

考虑定义一个全局变量 Speed，该变量在由按键触发的外部中断服务程序中改变，主程序在 while 循环中不断判断 Speed 的值，根据取值的不同设置不同的指示灯闪烁频率。

2. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。具体的程序如程序清单 7-7 和 7-8 所示。

程序清单 7-7　主程序

```
/* USER CODE BEGIN PV */
volatile uint8_t Speed = 0;  // 指示灯闪烁频率：0-->1Hz 1-->5Hz 2-->20Hz
/* USER CODE END PV */
/* .................. */    // 省去中间部分代码
/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
  /* USER CODE END WHILE */
  /* USER CODE BEGIN 3 */
  if( Speed == 0 )              // 1Hz 闪烁频率
  {
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    HAL_Delay(1000);
  }
  else if( Speed == 1 )         // 5Hz 闪烁频率
  {
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
    HAL_Delay(200);
  }
  else
  {                             // 20Hz 闪烁频率
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);
```

```
23.     HAL_Delay(50);
24.   }
25.   }
26. }
27.   /* USER CODE END 3 */
```

程序解析

① 第 2 行定义了一个 uint8_t（无符号字符型）类型全局变量 Speed，用来存放指示灯的闪烁频率：Speed=0 表示 1Hz 的闪烁频率；Speed=1 表示 5Hz 的闪烁频率；Speed=2 表示 20Hz 的闪烁频率。在 uint8_t 之前加上了 volatile 关键字修饰，确保变量不会被编译器优化。

② 第 11～25 行是 While 循环中执行的代码，采用 if-else 的分支语句，根据 Speed 的不同取值，改变指示灯的闪烁频率。

③ 第 11～15 行是判断 Speed 为 0 时，设置指示灯的闪烁频率为 1Hz，即延时时间为 1000ms。

④ 第 16～20 行是判断 Speed 为 1 时，设置指示灯的闪烁频率为 5Hz，即延时时间为 200ms。

⑤ 第 21～25 行是判断 Speed 为 2 时，设置指示灯的闪烁频率为 20Hz，即延时时间为 50ms。

程序清单 7-8　外部中断回调函数

```
/* USER CODE BEGIN 4 */
/*
 * @brief  EXTI line detection callbacks.
 * @param  GPIO_Pin: Specifies the pins connected EXTI line
 * @retval None
 */
void HAL_GPIO_EXTI_Callback(uint16_t GPIO_Pin)
{
   if(GPIO_Pin == B1_EXTI_Pin)         // 判断外部中断源
   {
   Speed++;                            // 修改 Speed 变量的值
   if( Speed == 3)                     // 限制 Speed 变量的变化范围：0~2
   {
     Speed = 0;
    }
   }
}
/* USER CODE END 4 */
```

程序解析

在外部中断回调函数 HAL_GPIO_EXTI_Callback 中完成具体的中断处理任务：对 Speed 的值做+1 操作，并判断 Speed 是否等于 3。如果 Speed==3，则需要对 Speed 清零，重新开始下一轮的修改。

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上，可以观察到：上电后，指示灯 LD2 按照 1Hz 频率闪烁。按下按键 B1 后，指示灯 LD2 闪烁频率加快，按照 5Hz 频率闪烁。再次按下按键 B1 后，指示灯 LD2 闪烁频率更快，按照 20Hz 频率闪烁。第三次按下按键后，指示灯恢复到 1Hz 的闪烁频率，并重复上述过程。

7.6 本章小结

本章重点介绍了中断的概念和作用以及 STM32 的中断系统。本章最为重要的部分是 HAL 库的中断处理流程，读者需要通过追踪中断的执行过程，深入到源代码内部去了解整个流程。图 7-5 和 7-6 可以帮助读者更好的理解 HAL 库的中断处理流程。在后续章节的学习中，如定时器和串口通信，也可以利用本章介绍的 HAL 库中断处理流程来分析定时器中断和串口中断的处理过程。

中断系统经过 HAL 封装之后，用户不需要判断中断标志以及清除中断标志，只需编写对应中断的回调函数即可，简化了中断服务程序的编写。

习题 7

1. 处理器与外部设备进行数据传输的方式有几种？各有什么特点？适用于什么场合？
2. 什么是中断？它在计算机系统中有什么作用？
3. 什么是中断向量表？发生中断时，程序是如何跳转到中断服务程序处执行的？
4. 请用自己的话描述 HAL 库的中断处理流程。分析这种处理方式的优点和缺点。
5. 处理多个外部中断与处理单个外部中断在编写回调函数时有什么不同？
6. 设计一个多路抢答器，假设有 4 路按键通过外部中断引入，请在 CubeMX 中完成配置并编写应用程序。

第八章　定时器

本章导读

本章主要介绍 STM32 的定时器。首先介绍定时器的基本概念：定时和计数。然后介绍 STM32 的定时器分类及各自特点。通过分析定时器的五大基本功能：定时、计数、输入捕获、输出比较和脉宽调制（Pulse Width Modulation，PWM）来全面掌握定时器的用法。每一种功能在介绍时，先分析该功能的结构框图和相关寄存器的作用，然后介绍 HAL 库中对应数据类型及接口函数的定义，并通过阶梯式任务实践来掌握它们的用法。

本章知识点

定时和计数的基本概念

HAL 库外设模块设计方法

定时器的基本定时器功能

定时器的 PWM 输出功能

8.1 定时器概要

定时功能的应用在我们日常生活中随处可见：交通灯各个信号的定时转换；微波炉和洗衣机的定时操作；智能手表和智能手环提供的时间功能。我们把用于实现定时功能的电路单元称为定时器。那么定时器的工作原理是什么？它又是如何实现的？

8.1.1 定时器工作原理

1. 定时和计数

定时器的设计是基于计数器实现的。计数器是一种对电脉冲进行计数的电路结构，向计数器输入一个脉冲信号，计数器的值就会加 1。如果我们将这个脉冲信号换成周期固定的信号，比如频率为 1Hz，周期为 1s 的方波，再让计数器的值从 0 开始计数。随着周期信号的持续输入，计数器的值不断增加。由于脉冲信号的周期为 1s，相当于计数器每增加 1 个数，时间就是 1s。这样，通过计数值，我们可以换算出时间值。比如，当前计数器的值是 10，就表示时间为 10s。因此，定时和计数本质上是一个概念，定时只是计数的一个特例。

2. 工作原理

从上面的描述我们可以推论出：定时器的核心就是一个计数器模块，可以进行加 1 或减 1 计数。每出现一个计数信号，计数器的值就自动加 1 或减 1。当计数值从 0 递增到最大值或者从最大值递减到 0 时，定时器可以向处理器发送中断请求，完成后续的操作。计数信号的来源可以选择周期性的内部时钟信号或者非周期的外部输入信号。这两种不同的计数信号，也决定了定时器两种最基本的工作模式：定时模式和计数模式。

学习定时器时，一定要注意以下的基本概念：

- **计数模式**

对引脚输入的外部脉冲信号计数，是定时器基本工作模式之一。

- **定时模式**

对处理器内部的周期性时钟信号计数，是定时器基本工作模式之一。

- **计数器**

定时器内部的核心模块。计数器可以递增计数或者递减计数。16 位计数器的最大计数值为 65535，32 位的最大值为 4294967295。

- 计数频率

定时器模式下，单位时间内计数器的计数次数。

- 计数周期

定时器模式下，计数器记一次数所花费的时间，为计数频率的倒数。

由于处理器内部的时钟频率是一个固定值，所以可以根据计数器的计数值计算出定时时间：

$$定时时间 = 计数值 / 计数频率$$

$$定时时间 = 计数值 \times 计数周期$$

例如，计数频率为 1MHz，计数周期则为 1/1000000s，也就是每经过 1 微秒（us），计数器加一（或减一）。此时 16 位计数器的最大定时时间为 65535us，即 65.535ms。

8.1.2 STM32 定时器介绍

STM32 微控制器的定时功能非常强大，定时器的种类也比较多。按照核内、核外的标准可以划分为内核定时器和外设定时器 2 大类。其中的外设定时器按照功能又可以划分为专用定时器和常规定时器。具体的分类如图 8-1 所示。

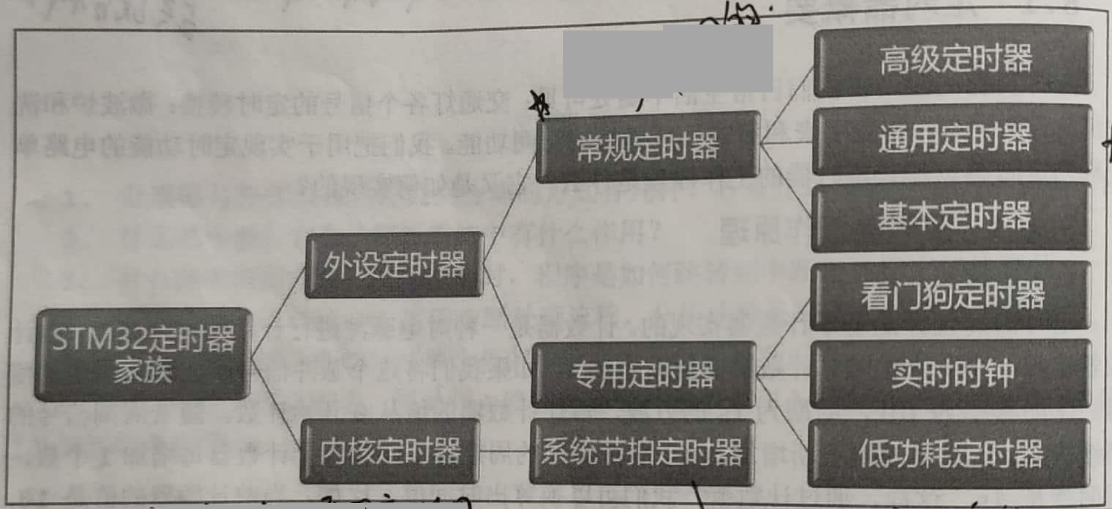


图 8-1 STM32 定时器分类

在众多的定时器中，常规定时器是工程应用中使用最广泛的。按照功能的不同分成了基本定时器、通用定时器和高级定时器三类：

- 基本定时器

几乎没有任何对外输入/输出，常用作时基，实现基本的计数、定时功能。

- 通用定时器

除了基本定时器的时基功能外，还具备输入捕获、输出比较等功能，以及连接其它传感器的接口，如编码器和霍尔传感器等。

- 高级定时器

此类定时器的功能最为强大，除了具备通用定时器的功能外，还包含一些与电机控制和数字电源应用相关的功能，比如带死区控制的互补信号输出、紧急刹车关断输入控制等。

不同型号的 STM32 微控制器芯片，片内所集成的常规定时器数量和种类各不相同，以本书介绍的 STM32F411 为例，片内集成了 8 个定时器。其中有 1 个 16 位高级定时器、2 个 32 位通用定时器和 5 个 16 位通用定时器。各个定时器拥有的硬件资源彼此独立，互不干

扰，如捕获/比较通道等。

定时器的类型和功能如表 8-1 所示。

表 8-1　STM32F411 定时器类型及功能

类型	名称	计数器位数	计数方式	预分频因数	产生 DMA 请求	捕获/比较通道	时钟频率	挂接总线
高级定时器	TIM1	16 位	递增/递减/中心对齐	1～65336	是	4	100MHz	APB2
通用定时器	TIM2 TIM5	32 位	递增/递减/中心对齐	1～65336	是	4	100MHz	APB1
	TIM3 TIM4	16 位	递增/递减/中心对齐	1～65336	是	4	100MHz	APB1
	TIM9	16 位	递增	1～65336	否	2	100MHz	APB2
	TIM10 TIM11	16 位	递增	1～65336	否	1	100MHz	APB2

定时器的时钟频率由定时器所挂接的外设总线时钟 APB 决定。挂接在外设总线 APB1 的定时器包括 TIM2、TIM3、TIM4 和 TIM5，APB1 时钟频率最高为 100MHz；挂接在外设总线 APB2 的定时器包括 TIM1、TIM9、TIM10 和 TIM11，APB2 时钟频率最高为 100MHz。在 CubeMX 的时钟设置部分，我们已经将 APB1 和 APB2 设置为 100MHz，因此所有定时器的时钟频率为 TIM_CLK = 100MHz。外设总线的时钟配置如 8-2 所示。

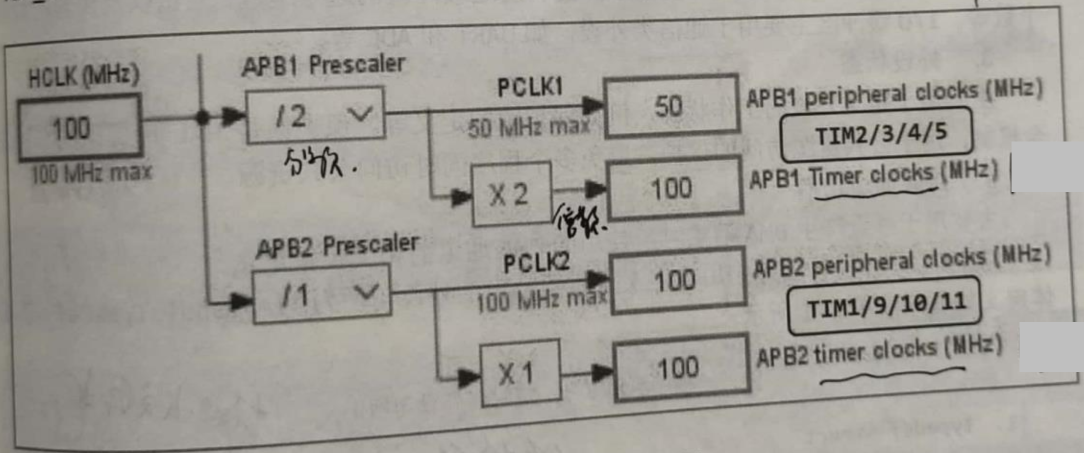


图 8-2　定时器时钟频率设置

无论哪一种定时器，其基本的功能都是定时和计数，在这两个功能上又衍生出其他的功能。在实际的工程应用中，最常用的定时器功能包括以下 5 种：

- 基本定时功能：可用于产生时间基准
- 基本计数功能：可用于脉冲个数的测量
- 输入捕获功能：可用于脉冲宽度的测量
- 输出比较功能：可用于输出各种类型的方波
- PWM 输出功能：可用于电机驱动、灯光亮度调节及 DC/DC 转换

由于定时器每一种功能所对应的数据类型及接口函数都比较多，我们将按照定时器的功能模块为单元，分阶段的学习。

备注：定时器数据类型的定义以及接口函数的声明放在 stm32f4xx_hal_timer.h 文件中，定时器接口函数的定义则放在 stm32f4xx_hal_timer.c 文件中。

8.2 HAL 库外设模块设计方法

在第 6 章 GPIO 中，我们介绍了 GPIO 外设对应的数据类型定义，只使用一个引脚初始化数据类型 GPIO_InitTypeDef 来描述引脚的各个属性，如引脚编号、工作模式、上拉/下拉电阻及输出速度等。这种方法对于简单的外设是能够实现的，如 GPIO 和外部中断等。但是对于功能比较复杂的外设而言，这种设计方法就有些力不从心了。

以定时器为例，除了最基本的定时功能外，要有输入捕获、输出比较和 PWM 输出等功能，每一种功能的实现都需要单独设计对应的初始化数据类型。因此，对于复杂外设的初始化数据类型及相关控制操作，HAL 库采用了外设句柄设计、外设编程模型以及外设通用接口函数等设计方法。

8.2.1 外设句柄设计

对于定时器（TIMER）、串口（UART）和模/数转换器（ADC）等功能较为复杂的外设，HAL 库设计了一个名为外设句柄的结构体类型 PPP_ HandleTypeDef（PPP 表示外设名称）。外设句柄一般由以下部分组成：

1. 初始化配置参数

主要包括用于外设初始化的结构体类型，以及外设寄存器的基地址定义。

2. I/O 缓冲区

主要用于定义数据传输时，接收缓冲区和发送缓冲区的起始地址、大小以及传输的数据个数等。I/O 缓冲区主要用于通信类外设，如 UART 和 ADC 等。

3. 外设状态

主要用于定义外设的工作状态，例如保护锁的定义等。保护锁是 HAL 库提供的一种安全机制，用于公共资源访问的保护，避免多个程序同时访问公共资源。

4. DMA 通道句柄

主要用于外设产生 DMA 请求时所使用的 DMA 通道的相关参数定义。

以定时器为例，HAL 库中用于定义定时器外设的外设句柄为 TIM_HandleTypeDef，其具体定义如程序清单 8-1 所示。

程序清单 8-1　定时器外设句柄定义

```
typedef struct
{
    TIM_TypeDef                  *Instance;   // 定时器寄存器的基地址定义
    TIM_Base_InitTypeDef         Init;        // 定时器定时功能的配置参数定义
    HAL_TIM_ActiveChannel        Channel;     // 输入捕获/输出比较的通道定义
    DMA_HandleTypeDef            *hdma[7];    // DMA 请求处理定义
    HAL_LockTypeDef              Lock;        // 保护锁类型定义
    __IO HAL_TIM_StateTypeDef    State;       // 定时器状态定义
} TIM_HandleTypeDef;
```

➡ 定义解析

● TIM_TypeDef *Instance

定时器寄存器的基地址定义，用于访问硬件寄存器。具体的做法是：先将定时器的所有寄存器封装为名为 TIM_TypeDef 的结构体类型，然后将定时器外设的首地址强制转换为该结构体类型的指针，再通过结构体指针加成员变量的形式访问各个硬件寄存器。这种方法在介绍 GPIO 外设时已经介绍过，感兴趣的读者可自行查询源码。

该结构体指针也常常称为外设实例，如定时器 1 对应的实例名：TIM1。利用 TIM1->CR1 就可以访问定时器 1 的 CR1 寄存器。

● TIM_Base_InitTypeDef Init

定义定时器基本定时功能的配置参数，采用结构体形式定义。具体含义将在 8.3 小节介绍。

● HAL_TIM_ActiveChannel Channel

定义定时器在输入捕获或输出比较时所使用的通道。

● DMA_HandleTypeDef *hdma[7]

定义定时器产生 DMA 请求时所使用的 DMA 通道的句柄。

● HAL_LockTypeDef Lock

定义保护锁类型。保护锁是 HAL 库中提供的一种安全机制，避免对定时器等资源型外设的并发访问。

● __IO HAL_TIM_StateTypeDef State

定义定时器的工作状态，提供给状态接口函数读取。

8.2.2 外设编程模型

HAL 库根据微处理器和外设的数据传输方式，设计了轮询、中断和 DMA 三种编程模型，用于外设的控制。

以定时器的基本定时功能为例，三种编程模型下的接口函数分别是：

1. 轮询方式

```
HAL_StatusTypeDef HAL_TIM_Base_Start(TIM_HandleTypeDef *htim);
HAL_StatusTypeDef HAL_TIM_Base_Stop(TIM_HandleTypeDef *htim);
```

2. 中断方式

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

3. DMA 方式

```
HAL_StatusTypeDef HAL_TIM_Base_Start_DMA(TIM_HandleTypeDef *htim,
uint32_t *pData, uint16_t Length);
HAL_StatusTypeDef HAL_TIM_Base_Stop_DMA(TIM_HandleTypeDef *htim);
```

从上述的接口函数定义，我们可以看出以下的一些规律：

● 不同编程模型下的接口函数，通过函数后缀加以区别，如轮询方式的接口函数不加后缀；中断方式的接口函数后缀为 IT，DMA 方式的接口函数后缀为 DMA。

● 接口函数的入口参数都是外设句柄的地址（指针）。

8.2.3 外设通用接口函数设计

为了确保 HAL 库函数在整个 STM32 微控制器系列之间能够无缝移植，HAL 库提供了通

用的接口函数，根据功能可以分成以下四类：

1. 初始化函数

根据用户配置的参数完成外设的初始化操作，例如我们在 GPIO 部分学习的 GPIO 初始化函数：HAL_GPIO_Init。

2. I/O 交互式函数

这类函数主要与外设进行数据交换，按照轮询、中断和 DMA 三种编程模型分类。相关函数的示例在 8.2.2 小节已经介绍了。

3. 控制函数

这类函数主要用于动态配置外设参数。例如，使用定时器产生 PWM 信号时，如果用户需要在定时器运行过程中，动态修改 PWM 信号的占空比，就需要调用比较/捕获寄存器设置函数__HAL_TIM_SET_COMPARE 来修改比较/捕获寄存器中的内容。

4. 状态函数

这类函数主要用于获取外设的工作状态以及出错信息等。例如，定时器状态获取函数 HAL_TIM_Base_GetState。

这些通用接口的函数设计既方便了用户快速掌握接口函数的功能，也确保了程序在 STM32 各个产品线之间能够无缝移植，提高开发效率。

8.3 定时器基本定时功能

8.3.1 时基单元

STM32 的常规定时器主要包括基本定时器、通用定时器和高级定时器。不论哪一类定时器，都有一个共同的计数定时单元，我们称之为时基单元。该单元主要由三个模块组成：预分频模块、计数模块和自动重载模块。时基单元的功能框架如图 8-3 所示。

在图 8-3 中，时钟信号 TIM_CLK 是由定时器所挂接的外设总线 AHB 的时钟频率决定，这里我们把它称为定时时钟。时钟信号 CK_CNT 是经过预分频模块后输入计数模块的时钟，这里我们把它称为计数时钟。

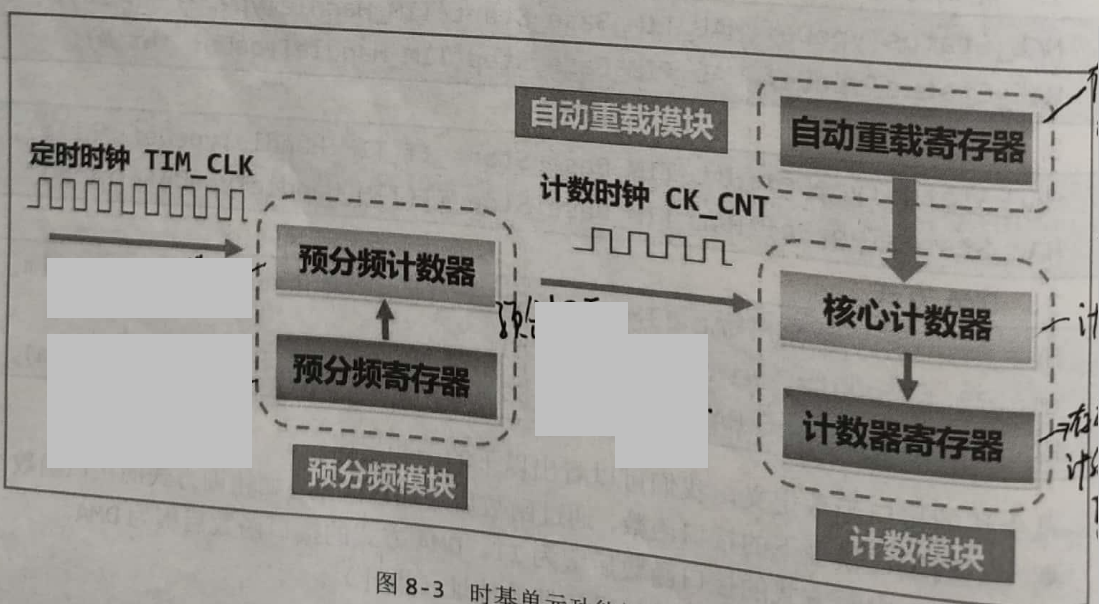


图 8-3 时基单元功能框架

1. 预分频模块

预分频模块由分频计数器和预分频寄存器 TIMx_PSC（x 表示定时器编号 1～11，下同）组成：分频计数器用来对由内部时钟送出的定时时钟 TIM_CLK 进行分频，而预分频寄存器用来设置和存放预分频系数。

当定时器的时钟源为内部时钟时，其频率一般都比较高，以 STM32F411 的通用定时器 TIM2 为例，其挂接的外设总线 APB1 的时钟频率最大为 100MHz，即 TIM_CLK = 100000000。体现在 16 位定时器上的效果就是从 0 计数到 65535，发生上溢的时间只需要 ~~0.065~~ 0.65ms 毫秒。如果我们需要更长时间的定时间隔，就需要设置预分频寄存器对定时时钟 TIM_CLK 进行分频处理，以降低计数时钟（CK_CNT）的频率。

除此之外，也可以通过设置预分频寄存器，来获取想要的计数时钟。假设外设总线 APB1 的时钟频率为 72MHz，需要获取一个精确的 1ms 定时中断。如果不分频，72MHz 的时钟对应的计数周期 1/72us，十分不利于计算。这时候，通过设置预分频寄存器将其 72 分频后为 1MHz，计数周期为 1us，1000 个计时周期即为 1ms，这样既便于计算，定时也更加精确。

预分频模块分频的工作原理是：预分频器计数器的初值为 0，TIM_CLK 每来一个时钟，预分频器计数器的值就加 1，直到达到预分频寄存器的设定值，然后再来一个时钟后，预分频器计数器的值归零，开始下一次计数。

由此可以看出，因为达到最大值后还要再来一个时钟才归零，所以分频系数应该为 PSC+1，其中 PSC 为用户写入预分频寄存器的值。例如，要对内部时钟进行 100 分频，则预分频寄存器的值应设置为 99。

2. 计数模块

计数模块由核心计数器和计数器寄存器 TIMx_CNT 组成：核心计数器用来对预分频模块输出的计数时钟 CK_CNT 进行二次计数。核心计数器的初值为 0，每来 1 个计数时钟，核心计数器自动加 1 或减 1。计数器寄存器则用来存放核心计数器实时运行时的计数值，便于用户读取。

3. 自动重载模块

自动重载模块由自动重载寄存器 TIMx_ARR 构成，用来设置核心计数器的计数终值或初值，决定计数脉冲的多少（计数模式）或计时周期（定时模式）的长短。当定时器设置为递增计数模式时，TIMx_ARR 的值作为核心计数器的计数终值，表示记到多少发生溢出；当定时器设置为递减计数模式时，TIMx_ARR 的值作为作为核心计数器的计数初值，表示从多少开始往下计数。自动重载寄存器 TIMx_ARR 的具体含义及使用需要配合核心计数器的计数模式予以确定。

【小贴士】预装载功能和影子寄存器

自动重载寄存器 TIMx_ARR 和预分频寄存器 TIMx_PSC 都具有预装载功能，即每类寄存器具有双寄存器机制，分别由各自的影子寄存器和预装载寄存器组成。影子寄存器是真正起作用的寄存器，预装载寄存器为影子寄存器提供缓冲，用户只能操作预装载寄存器。

预装载功能可由软件开启或关闭。关闭预装载功能时，写入预装载寄存器的内容将立即传入影子寄存器，使修改的内容生效。开启预装载功能时，写入预装载寄存器的内容将在更新事件（计数器溢出）发生时才传入影子寄存器并生效。

预装载功能在多个定时器同时输出信号时比较有用，可以确保多个定时器的输出信号在同一个时刻变化，实现同步输出。

4. **计数模式**

核心计数器支持 3 种计数模式：递增计数、递减计数和中心对齐计数，并产生更新事件，作为中断请求。

(1) 递增计数

核心计数器从 0 计数到自动重载值（TIMx_ARR 寄存器的内容），然后重新从 0 开始计数并生成计数器上溢事件。

(2) 递减计数

核心计数器从自动重载值（TIMx_ARR 寄存器的内容）开始递减计数到 0，然后重新从自动重载值开始计数并生成计数器下溢事件。

(3) 中心对齐计数（递增/递减计数）

核心计数器从 0 开始计数到自动重载值（TIMx_ARR 寄存器的内容）-1，生成计数器上溢事件；然后从自动重载值开始向下计数到 1，并生成计数器下溢事件。之后再从 0 开始重新计数。

注意：中心对齐计数在一次计数周期中可以产生 2 个溢出事件。

三种计数模式的计数过程如图 8-4 所示。

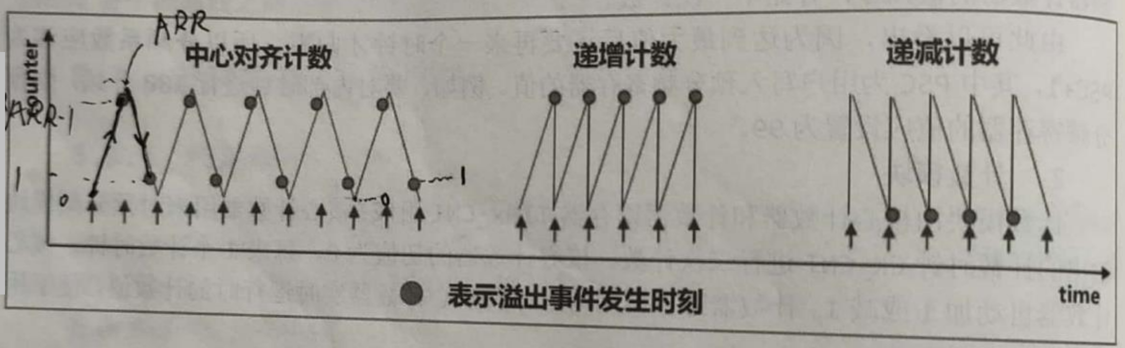


图 8-4 三种计数模式计数过程示意图

结合自动重载模块，我们可以得到不同计数模式下，核心计数器的溢出点 CNT 与自动重载值 ARR 之间的关系，如表 8-2 所示。

表 8-2 溢出点 CNT 与自动重载值 ARR 的关系列表

计数模式	溢出点	计数器重载值
递增计数	CNT = ARR	CNT = 0
递减计数	CNT = 0	CNT = ARR
中心对齐计数	CNT = ARR-1	CNT = ARR
	CNT = 1	CNT = 0

5. **定时时间计算公式**

定时器的定时时间由送入时基单元的定时时钟 TIM_CLK 的频率、预分频系数 PSC（TIMx_PSC 寄存器的内容）和自动重载值 ARR（TIMx_ARR 寄存器的内容）三者决定。

假设 PSC = 1，ARR = 36，采用递增计数模式，计数器寄存器的初值 CNT = 0。定时器的时序图如图 8-5 所示。

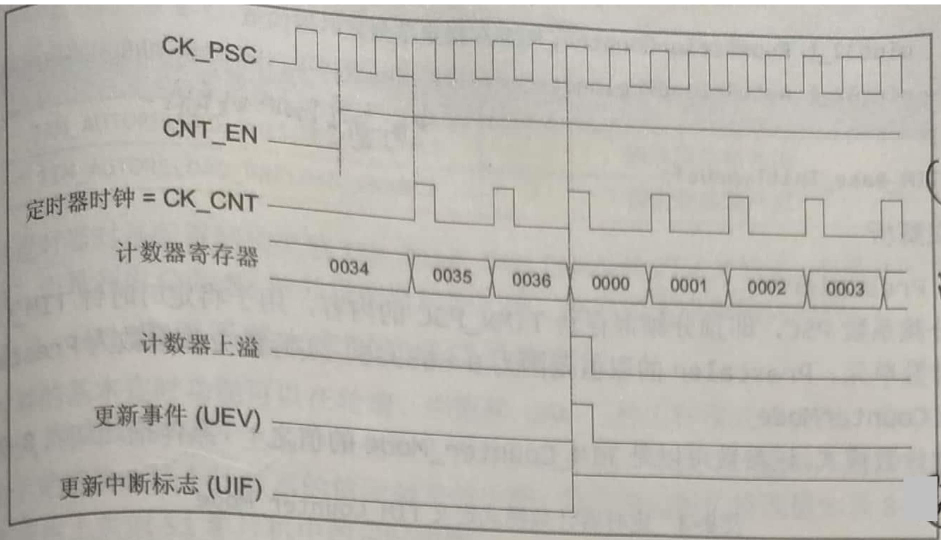


图 8-5　定时器时序图

从图 8-5 可以看出以下几点：

(1) CK_PSC 为送入预分频模块的时钟，该时钟的频率等于定时时钟 TIM_CLK 的频率。

(2) 当 PSC = 1 时，输入预分频模块的时钟 CK_PSC 进行了 2 分频，得到计数时钟 CK_CNT。因此，预分频模块的实际分频系数为 PSC + 1。

(3) 当 ARR = 36 时，由表 8-2 可知，当计数值 CNT = 36 时，生成计数器计数器上溢事件。由于计数值 CNT 的初始值为 0，从 0 开始计数，且计数值 0 会保持 1 个完整的计数脉冲。因此计算实际的计数脉冲个数时，需要加 1。当 ARR = 36 时，实际的计数脉冲个数为 ARR + 1 = 37。

综上所述，我们可以得出定时器的更新中断频率计算公式：

```
F_UpdateEvent = TIM_CLK / [( PSC + 1 ) x ( ARR + 1 )]          (8-1)
```

应用举例

假设，定时器挂接的总线 APB1 的时钟频率为 100MHz，预分频寄存器的值 PSC 等于 9999，自动重载寄存器的值 ARR 也等于 9999。将参数带入公式 8-1，我们可以得到定时器的更新中断（定时时间）频率为：

```
F_UpdateEvent = 100000000 / [(9999+1) x (9999 + 1)] = 1Hz = 1s
```

8.3.2 HAL 库用于基本定时的数据类型定义

定时器用于基本定时的数据类型是时基配置结构类型 TIM_Base_InitTypeDef，其定义如程序清单 8-2 所示。

程序清单 8-2　定时器时基配置结构定义

```
typedef struct
{
  uint32_t Prescaler;          // 指定预分频系数 PSC，即 TIMx_PSC 寄存器的内容
  uint32_t CounterMode;        // 指定计数模式
  uint32_t Period;             // 指定自动重载值 ARR，即 TIMx_ARR 寄存器的内容
  uint32_t ClockDivision;      // 指定定时时钟 TIM_CLK 的分频值
```

```
    uint32_t RepetitionCounter;    // 指定重复定时器的值
    uint32_t AutoReloadPreload;    // 设置自动重载寄存器 TIMx_ARR 内容生效时刻
}
TIM_Base_InitTypeDef;
```

[illegible]PWM 时有用.

定义解析

- Prescaler

预分频系数 PSC，即预分频寄存器 TIMx_PSC 的内容，用于将定时时钟 TIM_CLK 分频后输入计数单元。Prescaler 的取值范围为 0～63335。实际的分频系数为 Prescaler+1。

- CounterMode

指定计数模式，该参数可以是 TIM_Counter_Mode 的值之一，具体描述如表 8-3 所示。

表 8-3　定时器计数模式定义 TIM_Counter_Mode

宏常量定义	含义
TIM_COUNTERMODE_UP	递增计数模式
TIM_COUNTERMODE_DOWN	递减计数模式
TIM_COUNTERMODE_CENTERALIGNED1	中心对齐计数模式 1
TIM_COUNTERMODE_CENTERALIGNED2	中心对齐计数模式 2
TIM_COUNTERMODE_CENTERALIGNED3	中心对齐计数模式 3

- Period

自动重载值 ARR，即自动重载寄存器 TIMx_ARR 的内容，作为计数器的边界值。如果是 16 位计数器，Period 的取值范围为 0～63335；如果是 32 位计数器，Period 的取值范围为 $0\sim2^{32}-1$。由于计数脉冲要算上 0，实际的计数脉冲数目是 Period+1。

注意：如果 Period 设置为 0，计数器将不会计数，即定时器不会启动。

- ClockDivision

指定定时时钟 TIM_CLK 的分频值，分频后的时钟作为引脚输入信号的采样时钟，实现对输入信号的滤波。该参数可以是 TIM_ClockDivision 的值之一，具体描述如表 8-4 所示。

表 8-4　定时器定时时钟分频值定义 TIM_ClockDivision

宏常量定义	含义
TIM_CLOCKDIVISION_DIV1	对定时时钟 TIM_CLK 进行 1 分频
TIM_CLOCKDIVISION_DIV2	对定时时钟 TIM_CLK 进行 2 分频
TIM_CLOCKDIVISION_DIV4	对定时时钟 TIM_CLK 进行 4 分频

- RepetitionCounter

用于设置重复计数器，重复计数器是高级定时器中才具备的电路单元，其作用是当计数器发生上溢/下溢事件时，重复计数器的值减 1。当减到零时，才会生成更新事件。

- AutoReloadPreload

用于设置自动重载寄存器 TIMx_ARR 的预装载功能，即自动重装寄存器的内容是更新事件产生时写入有效还是立即写入有效。该参数可以是 TIM_AutoReloadPreload 的值之一，具体描述如表 8-5 所示。

表 8-5　定时器预装载功能定义 TIM_AutoReloadPreload

宏常量定义	含义
TIM_AUTORELOAD_PRELOAD_DISABLE	预装载功能关闭
TIM_AUTORELOAD_PRELOAD_ENABLE	预装载功能开启

虽然定时器时基配置结构类型 TIM_Base_InitTypeDef 的定义较多，但是并不需要用户自行配置，而是利用 CubeMX 的外设配置功能完成，并自动生成初始化代码。

8.3.3 HAL 库用于基本定时的接口函数定义

定时器的基本定时功能可以在轮询、中断和 DMA 三种工作模式下实现，其中最常用的是中断模式。基本定时所产生的中断类型主要是更新中断，即核心计数器不断计数至发生溢出时，由于重新修改核心计数器的值而触发的中断。发生溢出时的修改值如表 8-2 所示。更新中断在应用上类似 51 单片机中的定时中断。

下面我们将重点介绍中断模式下与基本定时功能相关的接口函数。

1. 定时器时基单元初始化函数（HAL_TIM_Base_Init）

定时器时基单元初始化函数 HAL_TIM_Base_Init 主要用于时基单元的初始化，将定时器句柄中的参数写入对应的硬件寄存器。具体描述如表 8-6 所示。

表 8-6　定时器时基单元初始化函数

接口函数：HAL_TIM_Base_Init	
函数原型	HAL_StatusTypeDef HAL_TIM_Base_Init(TIM_HandleTypeDef *htim)
功能描述	按照定时器句柄中指定的参数初始化定时器时基单元
入口参数	*htim：定时器句柄的地址
返回值	HAL_StatusTypeDef：HAL_OK 表示初始化成功，HAL_ERROR 表示初始化失败
注意事项	1. 调用 MCU 相关的初始化函数 HAL_TIM_Base_MspInit 完成引脚、时钟和中断的设置 2. 该函数由 CubeMX 自动生成

注意：在函数 HAL_TIM_Base_Init 初始化时基单元时，会执行一个软件更新语句：TIMx->EGR = TIM_EGR_UG，这个语句将触发更新事件，并置位更新中断标志。如果用户采用中断方式启动定时器时，一旦使能了更新中断，将立即进入更新中断服务程序。所以我们在使能定时器更新中断之前，必须先做更新中断标志的清除操作，调用宏函数：__HAL_TIM_CLEAR_IT(&htimn,TIM_IT_UPDATE)，n 表示定时器编号。

2. 定时器中断模式启动函数（HAL_TIM_Base_Start_IT）

定时器中断模式启动函数 HAL_TIM_Base_Start_IT 主要用于使能定时器的更新中断，并启动定时器运行。该函数由用户在定时器初始化之后调用。具体描述如表 8-7 所示。

表 8-7　定时器启动函数

接口函数：HAL_TIM_Base_Start_IT	
函数原型	HAL_StatusTypeDef HAL_TIM_Base_Start_IT(TIM_HandleTypeDef *htim)
功能描述	使能定时器的更新中断，并启动定时器运行
入口参数	*htim：定时器句柄的地址
返回值	HAL_StatusTypeDef：固定返回 HAL_OK 表示启动成功

注意事项	1. 该函数在定时器初始化完成之后调用 2. 函数需要由用户调用，用于使能定时器的更新中断，并启动定时器运行 3. 启动前调用宏函数__HAL_TIM_CLEAR_IT(&htimn,TIM_IT_UPDATE)清除更新中断标志

3. 定时器中断入口函数（HAL_TIM_IRQHandler）

定时器中断入口函数 HAL_TIM_IRQHandler 是定时器所有中断发生后的统一入口函数。任何一个定时器的相关中断发生后，如更新中断或捕获中断等，都会通过中断向量表跳转到该函数。在该函数的内部会进行中断类型的判断，并清除对应的中断标志，然后再根据不同的中断类型，调用不同的回调函数完成中断处理任务。具体描述如表 8-8 所示。

表 8-8 定时器中断入口函数

接口函数：HAL_TIM_IRQHandler	
函数原型	void HAL_TIM_IRQHandler(TIM_HandleTypeDef *htim);
功能描述	定时器所有中断发生后的统一入口函数
入口参数	*htim：定时器句柄的地址
返回值	无
注意事项	1. 函数内部先判断中断类型，并清除对应的中断标志，最后调用回调函数完成中断处理 2. 该函数由 CubeMX 自动生成

4. 定时器更新中断回调函数（HAL_TIM_PeriodElapsedCallback）

定时器更新中断 HAL_TIM_PeriodElapsedCallback 用于处理发生更新中断后的具体任务。该函数作为所有定时器发生更新中断时的统一任务处理函数，在函数内部需要先判断是哪一个定时器产生的更新中断，然后执行对应的任务处理操作。具体描述如表 8-9 所示。

表 8-9 定时器更新中断回调函数

接口函数：HAL_TIM_PeriodElapsedCallback	
函数原型	void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
功能描述	回调函数，用于处理所有定时器的更新中断，用户在该函数内编写实际的任务处理程序
入口参数	*htim：定时器句柄的地址
返回值	无
注意事项	1. 函数由 HAL_TIM_IRQHandler 函数调用，完成所有定时器的更新中断的任务处理 2. 函数内部需要根据定时器句柄的实例来判断是哪一个定时器产生的更新中断 3. 函数由用户根据具体的处理任务编写

5. 定时器中断标志清除函数（__HAL_TIM_CLEAR_IT）

定时器中断标志清除函数__HAL_TIM_CLEAR_IT 是用于清除定时器各类中断标志。该函数采用带参数的宏实现（宏函数），与普通函数相比，不需要函数调用的过程，直接进行宏的替换，执行效率较高。

宏函数的定义如下：

```
#define          __HAL_TIM_CLEAR_IT(__HANDLE__,                    __INTERRUPT__)
            ((__HANDLE__)->Instance->SR = ~(__INTERRUPT__))
```

其中的入口参数__HANDLE__代表定时器句柄的地址，入口参数__INTERRUPT__代表要清除的中断标志。替换部分的语句直接访问定时器状态寄存器 SR，根据中断标志的类型

宏：替换

清除 SR 寄存器中对应的位。具体的实现方式就是利用定时器句柄 htim 的成员变量外设实例 instance 来访问对应的硬件寄存器。

常用的几个中断标志定义如表 8-10 所示。

表 8-10 定时器常用中断标志定义

宏常量定义	含义
TIM_IT_UPDATE	更新中断标志
TIM_IT_CC1	通道 1 的捕获/比较中断标志
TIM_IT_CC2	通道 2 的捕获/比较中断标志
TIM_IT_CC3	通道 3 的捕获/比较中断标志
TIM_IT_CC4	通道 4 的捕获/比较中断标志

8.3.4 基础任务：定时闪烁指示灯

➡ **任务目标**：掌握 CubeMX 软件配置定时器实现基本定时的方法

➡ **任务内容**：控制 Nucleo 开发板上的指示灯 LD2 每隔 1s 闪烁

➡ **硬件原理**：指示灯 LD2 由 PA5 引脚控制：PA5 输出高电平，开启指示灯；PA5 输出低电平，关闭指示灯。电路原理图参见第二章的图 2-16。

➡ **任务实现**

1. 设计思路

本任务使用定时器 10 产生 1s 的定时中断，定时器 10 挂接在 APB2 总线上，因此送入定时器的定时时钟 TIM_CLK 为 100MHz，我们可以设置预分频系数 PSC 为 9999，自动重载值 ARR 为 9999，将 TIM_CLK、PSC 和 ARR 都带入公式 8-1，刚好得到 1Hz 的更新中断，即 1s 的定时时间。

2. 引脚分配

本任务的引脚分配主要是指示灯控制引脚 PA5 的设置，设置参数第五章中一样，这里就不再赘述。

3. 外设配置

在外设设置窗口中选择：Timers → TIM10，将展开 TIM10 的配置窗口。

(1) 在配置窗口上方的 Mode 栏，勾选 Activated，以使能定时器 10（TIM10），如图 8-6 中序号 1 所示。

(2) 在配置窗口下方的 Configuration 栏，选择 Parameter Settings 标签页进行如下的参数设置，如图 8-6 中序号 2 所示。

- Prescaler（预分频系数 PSC）：9999
- Counter Mode（计数模式）：Up（递增计数）
- Counter Period（自动重载值 ARR）：9999
- Internal Clock Division（时钟分频）：No Division（不分频）
- auto-reload preload（预装载功能）：Disable（预装载功能关闭）

(3) 切换到 NVIC 栏使能 TIM10 的中断，如图 8-6 中序号 3 所示。

注意：定时器 10 的全局中断和定时器 1 的更新中断共享同一个中断通道。

(4) 中断优先级不进行设置，使用默认的优先级即可。

具体的配置如图 8-6 所示。

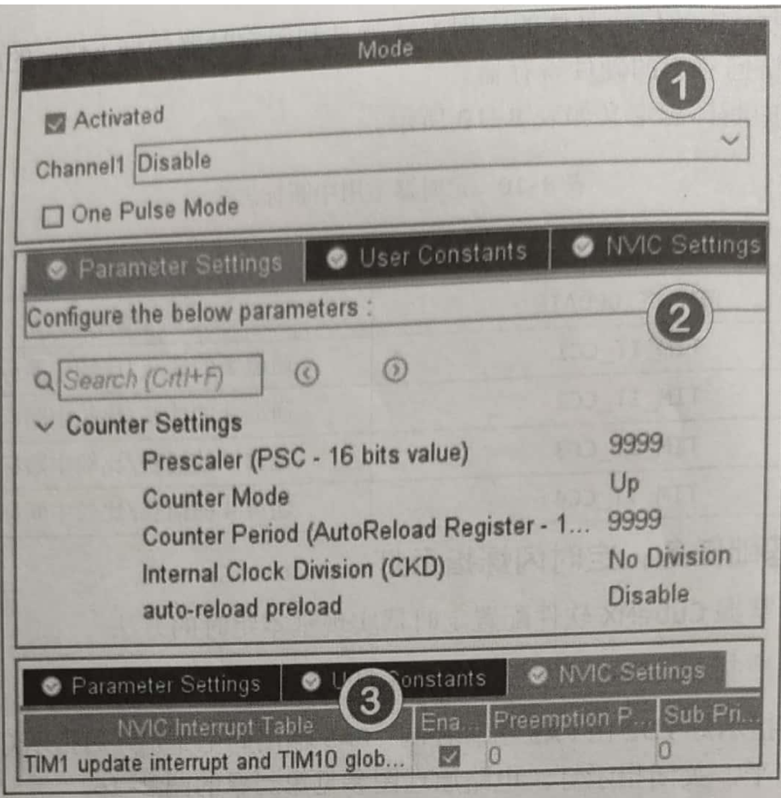


图 8-6　定时器 10 配置

4. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。主程序部分将会自动定义一个定时器 10 的句柄 htim10，并且自动生成定时器 10 的初始化函数 MX_TIM10_Init。完成定时器 10 的初始化之后，调用中断模式下的定时器启动函数 HAL_TIM_Base_Start_IT，使能定时器 10 的更新中断，并启动定时器运行。While 循环中不进行任何操作，指示灯的驱动任务交给定时器的更新中断回调函数完成。主程序部分的程序如程序清单 8-3 所示。

程序清单 8-3　定时闪烁指示灯任务的主程序

```
/* Private variables ---------------------------------------------------------*/
TIM_HandleTypeDef htim10;                  // 定时器 10 的句柄 htim10
/* Private function prototypes -- -------------------------------------------*/
void SystemClock_Config(void);             // 系统时钟初始化函数声明
static void MX_GPIO_Init(void);            // GPIO 初始化函数声明
static void MX_TIM10_Init(void);           // 定时器 10 初始化函数声明
int main(void)
{
  HAL_Init();                              // HAL 初始化
  SystemClock_Config();                    // 系统时钟初始化
  MX_GPIO_Init();                          // GPIO 初始化
  MX_TIM10_Init();                         // 定时器 10 初始化
  /* USER CODE BEGIN 2 */
  // 清除更新中断标志，避免定时器一启动就进入中断
  __HAL_TIM_CLEAR_IT(&htim10, TIM_IT_UPDATE);
```

```
16.  HAL_TIM_Base_Start_IT(&htim10);            // 使能定时器 10 更新中断并启动定时器
17.  /* USER CODE END 2 */
18.  /* Infinite loop */
19.  /* USER CODE BEGIN WHILE */
20.  while (1)                                   // while 循环
21.  {
22.    /* USER CODE END WHILE */
23.    /* USER CODE BEGIN 3 */
24.  }
25.  /* USER CODE END 3 */
26. }
```

程序解析

① 第 2 行是由 CubeMX 自动生成的定时器 10 的句柄 htim10。

② 第 4~6 行是片内外设初始化函数的声明，由 CubeMX 自动生成。

③ 第 9~12 行是系统初始化操作，包括 HAL 库初始化，系统时钟初始化，GPIO 初始化和定时器 10 初始化，由 CubeMX 自动生成。

④ 第 15 行清除更新中断标志，避免定时器一启动就进入中断。

⑤ 第 16 行是调用中断模式下的定时器启动函数 HAL_TIM_Base_Start_IT，使能定时器 10 的更新中断，并启动定时器运行，由用户添加。注意调用时传入的入口参数是定时器句柄的地址：&htim10。

⑥ 第 20~24 行是 while 循环，循环中没有添加任何代码。

更新中断回调函数的程序如程序清单 8-4 所示。

程序清单 8-4 定时闪烁指示灯任务的更新中断回调函数

```
/* USER CODE BEGIN 4 */
/*
 * @brief  Period elapsed callback in non-blocking mode
 * @param  htim TIM handle
 * @retval None
 */
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{                                               // 判断发生更新中断的定时器
  if( htim ->Instance==TIM10 )
  {
    HAL_GPIO_TogglePin(LD2_GPIO_Port, LD2_Pin);  // 翻转指示灯 LD2 状态
  }
}
/* USER CODE END 4 */
```

程序解析

整个更新中断的回调函数由用户编写。进入该回调函数的流程如下：

① 定时器 10 递增计数，从 0 开始记到自动重载值 ARR 时，产生计数器上溢，触发

更新中断；

② 处理器响应更新中断，进入启动文件 startup_stm32f411xe.s 中的中断向量表，根据中断类型号找到对应的中断服务程序 TIM1_UP_TIM10_IRQHandler，转入该函数执行；

③ 在 TIM1_UP_TIM10_IRQHandler 函数中调用定时器发生中断时的统一中断入口函数 HAL_TIM_IRQHandler，该函数位于 stm32f4xx_it.c 文件中；

④ 在函数 HAL_TIM_IRQHandler 中判断中断类型，并清除对应的中断标志，然后调用更新中断回调函数 HAL_TIM_PeriodElapsedCallback。

以上 4 个步骤的代码由 CubeMX 自动生成。经过上述 4 个步骤后，进入定时器更新中断回调函数 HAL_TIM_PeriodElapsedCallback。在函数内部，先根据定时器句柄的实例 Instance 判断是哪一个定时器发生的更新中断，然后执行任务处理：翻转指示灯 LD2 的状态。

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。可以观察到：指示灯 LD2 的状态每隔 1s 翻转一次。

在本任务中，定时器的初始化代码由 CubeMX 按照图 8-6 的设置自动生成，对应的初始化函数名为 MX_TIM10_Init，在 main 函数中自动调用。函数 MX_TIM10_Init 向句柄 htim10 的成员变量中写入初始化参数，最后调用定时器时基单元初始化函数 HAL_TIM_Base_Init 将参数写入对应的硬件寄存器。具体代码如程序清单 8-5 所示。

程序清单 8-5　自动生成的定时器初始化代码

```
/*
 * @brief TIM10 Initialization Function
 * @param None
 * @retval None
 */
static void MX_TIM10_Init(void)
{
  /* USER CODE BEGIN TIM10_Init 0 */
  /* USER CODE END TIM10_Init 0 */
  /* USER CODE BEGIN TIM10_Init 1 */
  /* USER CODE END TIM10_Init 1 */
  htim10.Instance = TIM10;
  htim10.Init.Prescaler = 9999;
  htim10.Init.CounterMode = TIM_COUNTERMODE_UP;
  htim10.Init.Period = 9999;
  htim10.Init.ClockDivision = TIM_CLOCKDIVISION_DIV1;
  htim10.Init.AutoReloadPreload = TIM_AUTORELOAD_PRELOAD_DISABLE;
  if (HAL_TIM_Base_Init(&htim10) != HAL_OK)
  {
    Error_Handler();
  }
  /* USER CODE BEGIN TIM10_Init 2 */
```

```
23.    /* USER CODE END TIM10_Init 2 */
24. }
```

程序解析

① 第 12 行向句柄的成员变量外设实例 Instance 赋值 TIM10，用于访问定时器 10 对应的硬件寄存器。

② 第 13～17 行向句柄的成员变量 Init 赋值，即对时基配置结构赋值，主要是预分频系数、计数模式、自动重载值等参数。

③ 第 18 行调用定时器时基单元初始化函数 HAL_TIM_Base_Init 完成初始化。

④ 第 8～11 行以及第 22～23 行之间可以由用户自行添加所需的初始化代码。

你的任务

任务一：软件计数器

如果不使用预分频寄存器，只使用核心计数器，如何实现 1s 的定时中断？

提示：定义一个软件计数器变量，定时器实现 ~~1ms~~ 0.1ms 定时，利用软件计数器扩展到 1s。

任务二：定时中断处理按键

完成第六章中采用状态机思想设计的按键处理程序。

8.3.5 挑战任务：实现电子时钟

任务目标： 掌握 CubeMX 软件配置定时器实现基本定时的方法

任务内容： 设计电子时钟，从 00:00:00 开始计时，并将计时信息通过串口传给 PC 显示。

硬件原理： 利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

任务实现

本任务所使用的引脚和外设与基础任务中所使用的相同，因此省去引脚分配和外设配置的过程。关于串口通信的设置请参考第九章的串口重定向部分。

1. 设计思路

定义一个结构体变量，包含时、分、秒三个成员变量。设置定时器产生 1s 的更新中断，在更新中断回调函数中，修改时、分、秒的值。主程序在 while 循环中每隔 1s 调用 printf 函数将时间信息发送到 PC 显示。

2. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。限于篇幅关系，这里重点把用户编写的代码列出来。主程序的具体代码如程序清单 8-6 所示。

程序清单 8-6　电子时钟任务的主程序

```
/* USER CODE BEGIN PTD */
// 定义时钟结构体类型，包含时、分、秒 3 个成员变量
typedef struct
{
    uint8_t hour;
    uint8_t minute;
    uint8_t second;
```

latile

```
8.  }CLOCK_TypeDef;
9.  /* USER CODE END PTD */
10. /* USER CODE BEGIN PV */
11. CLOCK_TypeDef clock = {0} ;                    // 定义时钟结构体变量，初始化为 0
12. /* USER CODE END PV */
13. int main(void)
14. {
15.   /* USER CODE BEGIN 2 */
16.   // 清除更新中断标志，避免定时器一启动就进入中断
17.   __HAL_TIM_CLEAR_IT(&htim10, TIM_IT_UPDATE);
18.   HAL_TIM_Base_Start_IT(&htim10);     // 使能定时器 10 更新中断并启动定时器
19.   /* USER CODE END 2 */
20.   /* Infinite loop */
21.   /* USER CODE BEGIN WHILE */
22.   while (1)
23.   {
24.     /* USER CODE END WHILE */
25.     /* USER CODE BEGIN 3 */
26.     // 每隔 1s 显示时间信息
27.     printf("Time:%02d:%02d:%02d.\r\n",clock.hour,clock.minute,clock.second);
28.     HAL_Delay(1000);
29.   }
30.   /* USER CODE END 3 */
31. }
```

程序解析

① 第 3～8 行利用类型重定义 Typedef 定义了时间的结构体类型 CLOCK_TypeDef，包含时、分、秒三个成员变量。

② 第 11 行利用新类型 CLOCK_TypeDef 定义时间变量 clock，并初始化为 0。

③ 第 17 行清除更新中断标志，避免定时器一启动就进入中断。

④ 第 18 行使能定时器 10 更新中断，并启动定时器。

⑤ 第 27～28 行每隔 1s，向串口输出时间信息，%02d 表示输出占两个字符宽度，不足补 0。

定时器更新中断回调函数的具体代码如程序清单 8-7 所示。

程序清单 8-7 电子时钟任务的定时器更新中断回调函数

```
/* USER CODE BEGIN 4 */
/*
  * @brief  Period elapsed callback in non-blocking mode
  * @param  htim TIM handle
  * @retval None
  */
```

```c
void HAL_TIM_PeriodElapsedCallback(TIM_HandleTypeDef *htim)
{
    if( htim ->Instance==TIM10 )
    {
        clock.second++;
        if( clock.second == 60)
        {
            clock.second = 0;
            clock.minute++;
            if( clock.minute == 60 )
            {
                clock.minute = 0;
                clock.hour++;
                if( clock.hour == 24)
                {
                    clock.hour = 0;
                }
            }
        }
    }
}
* USER CODE END 4 */
```

➡ 程序解析

时钟更新中断回调函数的程序比较简单，主要是根据 24 小时的时间格式，修改 clock 的成员变量：当 1s 定时时间到时，成员变量 second 加 1。当 second 加到 60 时，second 清零，成员变量 minute 加 1。当 minute 加到 60 时，minute 清零，成员变量 hour 加 1。当 hour 加到 24 时，hour 清零。

8.4 定时器 PWM 输出功能

脉冲宽度调制（PWM）是一种对模拟信号电平进行数字编码的方法。通过高分辨率计数器的使用，方波的占空比被调制后，可以用来对一个具体模拟信号的电平进行编码。PWM 技术广泛应用于机械、通信、功率控制等领域，如电机转速控制、灯光亮度控制、DC-DC 转换器以及信号调制等场合。

PWM 信号有两个重要的概念：周期和占空比。

- 周期（Period）：一个完整 PWM 波形持续的时间。
- 占空比（Duty）：高电平持续时间（T_{on}）与周期时间（Period）的比值。

占空比的计算公式如下：

$$Duty = (T_{on} / Perion) \times 100\% \tag{8-2}$$

图 8-7 给出了 50%、20%和 80%三种不同占空比时的 PWM 信号。

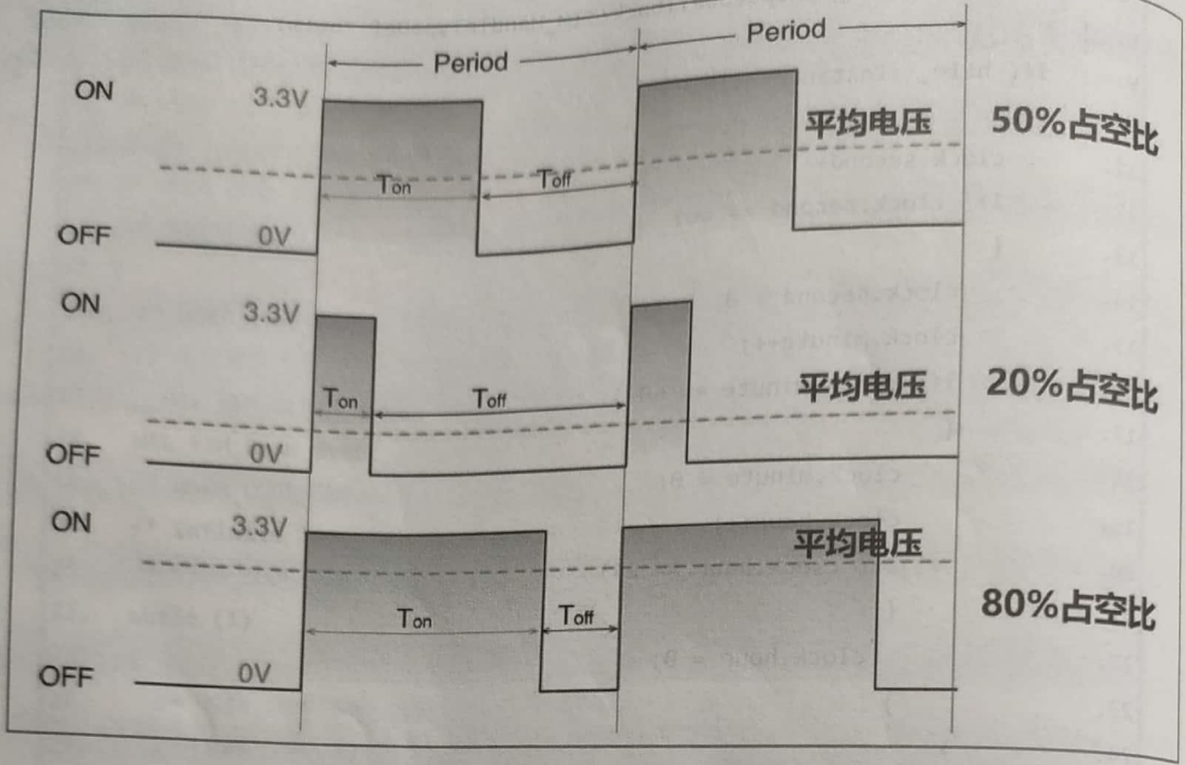


图 8-7　三种不同占空比的 PWM 信号

从图 8-7 还可以看到，不同占空比的 PWM 信号，对应的平均电压不同，电压幅值从 0V~VDD（对于 STM32 而言，VDD 就是 I/O 口的输出电压 3.3V）范围内变化，这也正是 PWM 可以进行电机转速控制和灯光亮度控制的原因。

8.4.1　定时器通道结构

每个 STM32 的高级定时器和通用定时器在硬件结构上配备了称为通道的电路结构。每个定时器最多可以拥有 4 个独立的通道，这些通道我们通常用 TIMx_CH1、TIMx_CH2 来标识。每个通道都有对应的硬件寄存器进行控制，也有对应的 GPIO 引脚作为通道的输入和输出，其电路结构如图 8-8 所示。

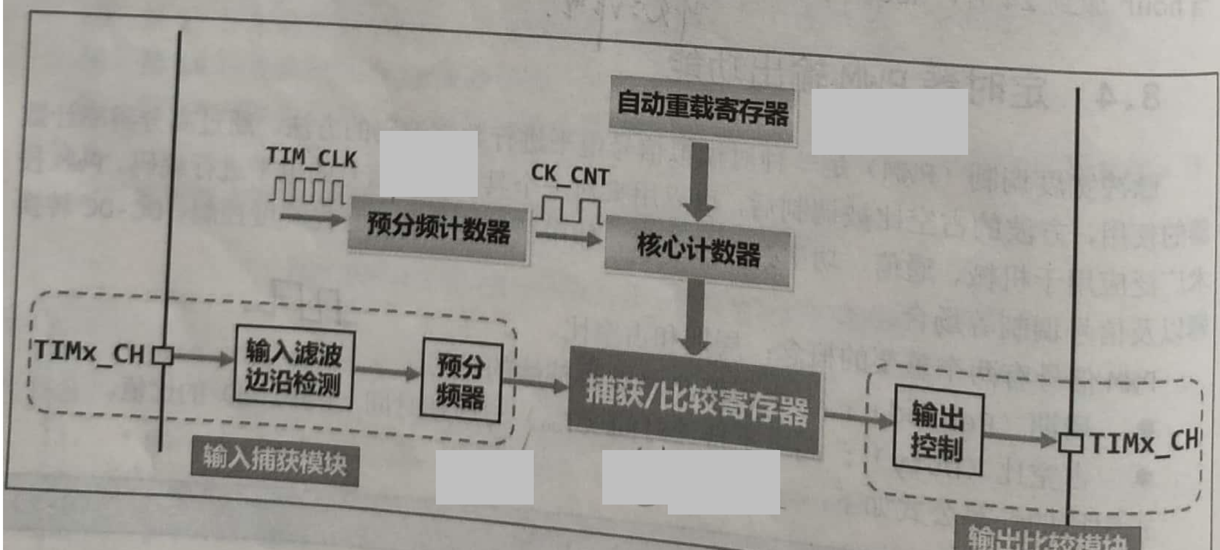


图 8-8　定时器通道结构

从图 8-8 可以看到，在原来时基单元的基础上，增加了名为通道的电路结构。通道的电路结构由 3 个模块组成：输入捕获模块、比较/捕获寄存器、输出比较模块。

1. 输入捕获模块

输入捕获模块用于捕获外部触发信号，捕获方式可以设置为上升沿捕获、下降沿捕获和双边沿捕获三种。发生捕获事件时，将此刻计数器的值锁存到比较/捕获寄存器中，以供用户读取，同时可以产生捕获中断。输入捕获功能主要用于信号测量，可以测量信号的周期、频率和占空比等参数。

2. 捕获/比较寄存器

捕获/比较寄存器 TIMx_CCR 是通道电路结构中最重要的寄存器，寄存器的位数是 16 位。它由影子寄存器和预装载寄存器组成，具备预装载功能，可由软件选择开启或关闭。用户通过访问预装载寄存器获取 TIMx_CCR 的值。

3. 输出比较模块

输出比较模块用于信号输出。定时器通过将预设的比较值与计数器的值做匹配比较，实现各类输出，如 PWM 输出、电平翻转、单脉冲输出、强制输出等。预设的比较值存放在捕获/比较寄存器 TIMx_CCR 中。

8.4.2 PWM 实现原理

为了阐明 PWM 模式的实现原理，我们举一个简单的例子。假设，我们需要生成一个周期为1ms，占空比为 47.5% 的 PWM 信号，其波形图如图 8-9 所示。

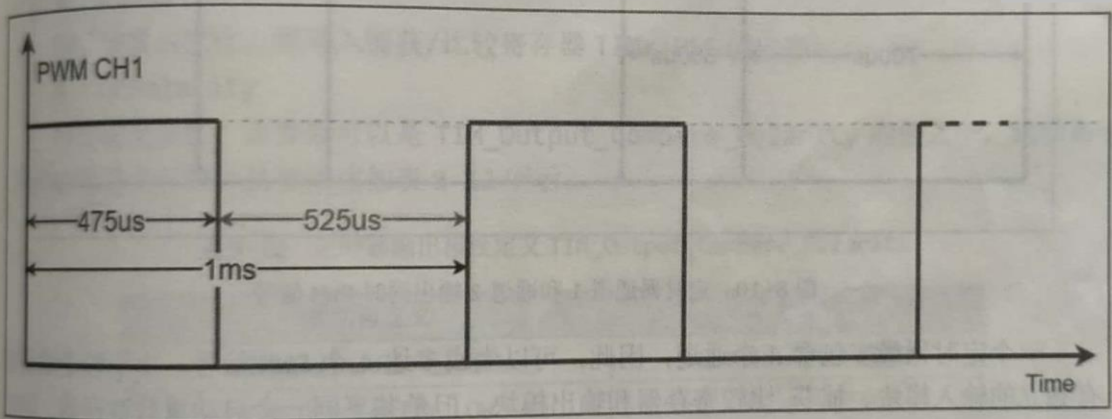


图 8-9 定时器通道 1 输出 PWM 信号

要产生这样一个 PWM 信号，首先要确保 1ms 的周期。假设送入定时器的时钟为 100MHz，我们可以设置预分频系数 PSC 为 99，将时钟分频到 1MHz，然后设置自动重载值 ARR 为 999，则每隔 1ms 有一个更新中断，这个时间就作为 PWM 的周期。

假设使用通道 1 输出 PWM 信号，对应的引脚为 PWM_CH1。按照占空比的要求，我们设置捕获/比较寄存器 TIMx_CCR1 为 475。这样，计数器从 0 开始计数，计数过程中，不断将计数值和 TIMx_CCR1 寄存器的值比较。当两者相等时，即计数值 CNT == 475 时，输出电平翻转。计数器继续计数，到计数值 CNT 等于重载值 ARR 时，输出电平再次翻转，开始下一个周期的输出。

因此，要使定时器输出指定周期和占空比的 PWM 信号，主要就是设置预分频系数 PSC、自动重载值 ARR 和匹配值 CCR。PWM 信号的周期由送入定时器的时钟 TIMx_CLK 和自动重载寄存器 TIMx_ARR 所确定。PWM 信号的占空比由捕获/比较寄存器 TIMx_CRR 确定。

综上所述，我们可以 PWM 信号频率及占空比的计算公式：

```
F_PWM = TIM_CLK / ( PSC + 1 ) x ( ARR + 1 )        (8-3)
Duty  = ( CRR / ( ARR + 1 ) ) x 100%                (8-4)
```

由于定时器包含多个通道，我们也可以利用同一个定时器产生具有相同周期不同占空比的另一个 PWM 信号。例如，使用通道 2，将 TIMx_CCR2 配置为 700。这时，将在通道 2 对应的引脚 PWM_CH2 上输出周期为 1ms，占空比为 70%的 PWM 信号。如图 8-10 所示。

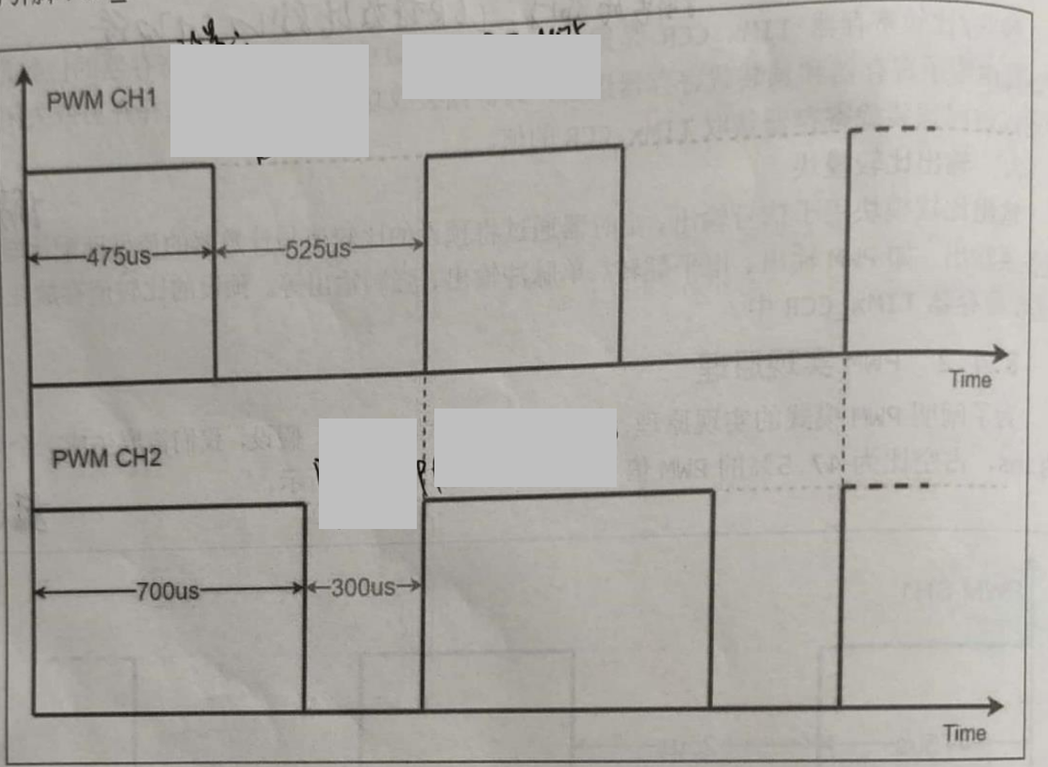


图 8-10　定时器通道 1 和通道 2 输出同时 PWM 信号

一个定时器最多包含 4 个通道，因此，可以生成多达 4 个 PWM 信号。由于每个通道都有独立的输入模块、捕获/比较寄存器和输出模块，但是共享同一个自动重载寄存器。因此这些 PWM 信号具有相同的周期和不同的占空比。

8.4.3　HAL 库用于 PWM 输出的数据类型定义

定时器用于 PWM 输出的数据类型是输出比较配置结构类型 TIM_OC_InitTypeDef，其定义如程序清单 8-8 所示。

程序清单 8-8　定时器输出比较配置结构定义

```
typedef struct
{
    uint32_t OCMode;              // 指定定时器输出模式
    uint32_t Pulse;               // 指定装入捕获/比较寄存器 CRR 的值
    uint32_t OCPolarity;          // 指定输出极性
    uint32_t OCNPolarity;         // 指定互补极性，该参数只对高级定时器有效
    uint32_t OCFastMode;          // 快速输出模式使能，该参数只对 PWM 输出有效
```

```
    uint32_t OCIdleState;        // 指定空闲状态下输出比较引脚的电平状态
    uint32_t OCNIdleState;       // 指定空闲状态下互补输出引脚的电平状态
} TIM_OC_InitTypeDef;
```

定义解析

- OCMode

指定输出模式，该参数可以是 TIM_Output_Compare_and_PWM_modes 的值之一，具体描述如表 8-11 所示。

表 8-11　定时器输出模式定义 TIM_Output_Compare_and_PWM_modes

宏常量定义	含义
TIM_OCMODE_TIMING	计数模式
TIM_OCMODE_ACTIVE	匹配时设置通道为有效电平
TIM_OCMODE_INACTIVE	匹配时设置通道为无效电平
TIM_OCMODE_PWM1	PWM 模式 1
TIM_OCMODE_PWM1	PWM 模式 2
TIM_OCMODE_FORCED_ACTIVE	强制为有效电平
TIM_OCMODE_FORCED_INACTIVE	强制为无效电平

- Pulse

用于设置占空比，即写入捕获/比较寄存器 TIMx_PSC 的内容。

- OCPolarity

指定输出极性，该参数可以是 TIM_Output_Compare_Polarity 的值之一，选择高电平或者低电平有效。具体描述如表 8-12 所示。

表 8-12　定时器输出极性定义 TIM_Output_Compare_Polarity

宏常量定义	含义
TIM_OCPOLARITY_HIGH	输出高电平有效
TIM_OCPOLARITY_LOW	输出低电平有效

- OCFastMode

快速输出模式使能，该参数可以是 TIM_Output_Fast_State 的值之一。具体描述如表 8-13 所示。

表 8-13　定时器快速输出模式定义 TIM_Output_Fast_State

宏常量定义	含义
TIM_OCFAST_DISABLE	不使能快速输出模式
TIM_OCFAST_ENABLE	使能快速输出模式

输出比较配置结构体中的另外三个成员变量 OCNPolarity、OCIdleState、OCNIdleState 是高级定时器才有的属性，主要用于互补输出。这里就不再赘述，感兴趣的读者可以自行查阅芯片的参考手册。

8.4.4 HAL 库用于 PWM 输出的接口函数定义

定时器的 PWM 输出功能可以在轮询、中断和 DMA 三种工作模式下实现。在大多数应用中，主要是产生指定周期和占空比的 PWM 信号，因此主要工作在轮询模式下，下面我们将重点介绍轮询模式下与 PWM 输出功能相关的接口函数。

1. 定时器 PWM 输出启动函数（HAL_TIM_PWM_Start）

定时器 PWM 输出启动函数 HAL_TIM_PWM_Start 主要用于在轮询模式下启动 PWM 输出。具体描述如表 8-14 所示。

表 8-14 定时器 PWM 输出启动函数

接口函数：HAL_TIM_PWM_Start	
函数原型	HAL_StatusTypeDef HAL_TIM_PWM_Start(TIM_HandleTypeDef *htim, uint32_t Channel)
功能描述	启动指定定时器的指定通道输出 PWM 信号
入口参数	htim：定时器句柄，范围是 htim1～htim11
返回值	HAL_StatusTypeDef：HAL_OK 表示初始化成功，HAL_ERROR 表示初始化失败
注意事项	1. 该函数在定时器初始化完成之后调用 2. 函数需要由用户调用，用于启动定时器指定通道输出 PWM 信号

2. 定时器比较/捕获寄存器设置函数（__HAL_TIM_SET_COMPARE）

定时器比较/捕获寄存器设置函数__HAL_TIM_SET_COMPARE 是用于设置比较/捕获寄存器中的内容，采用带参数的宏实现（宏函数）。具体描述如表 8-15 所示。

表 8-15 定时器比较/捕获寄存器设置函数

接口函数：__HAL_TIM_SET_COMPARE	
函数原型	__HAL_TIM_SET_COMPARE(__HANDLE__, __CHANNEL__, __COMPARE__)
功能描述	设置定时器的捕获/比较寄存器 CCR，一般用于 PWM 输出时，改变 PWM 信号的占空比
宏参数 1	__HANDLE__：定时器句柄
宏参数 2	__CHANNEL__：定时器通道号，该参数值可以是 TIM_CHANNEL_1～TIM_CHANNEL_4 之一
宏参数 3	__COMPARE__：写入捕获比较寄存器的值
返回值	无
注意事项	1. 该函数是宏函数，进行宏替换，不发生函数调用 2. 函数需要由用户调用，用于 PWM 输出时，改变 PWM 信号的占空比

8.4.5 基础任务：输出 PWM 信号

➡ **任务目标**：掌握 CubeMX 软件配置定时器输出 PWM 信号的方法

➡ **任务内容**：输出周期为 200ms，占空比 50%的方波信号控制 Nucleo 开发板上的指示灯

➡ **硬件原理**：设置 PA5 引脚输出 PWM 信号，控制指示灯 LD2。

➡ **任务实现**

1. 设计思路

由于我们要输出 PWM 信号控制指示灯，因此在引脚上要选择指示灯对应的控制引脚 PA5 作为定时器的输出通道。在 CubeMX 软件的芯片引脚窗口中点击 PA5 引脚，可以看到它的复用功能，其中对应定时器输出功能的是 TIM2_CH1，即定时器 2 的输出通道 1。因此我们选

择定时器 2 的通道 1 来输出 PWM 信号。

本任务要求的 PWM 信号周期为 200ms。由于定时器 2 挂接在 APB2 总线上，送入定时器的定时时钟 TIM_CLK 为 100MHz。我们可以设置预分频系数 PSC 为 9999，自动重载值 ARR 为 1999，将 TIM_CLK、PSC 和 ARR 都带入公式 8-3，刚好得到 5Hz 的更新中断频率，即 200ms 的 PWM 周期。

确定周期之后，由于占空比为 50%，根据公式 8-4，可以得到捕获/比较寄存器 CCR 的值为 1000。

2. 引脚分配

本任务的引脚分配主要是 PWM 信号输出通道的设置。在芯片引脚窗口中点击 PA5 引脚，选择定时器输出功能 TIM2_CH1。

注意：此时的引脚颜色为橙色，表示只完成了引脚分配，但对应的外设还没有使能。如图 8-11 所示

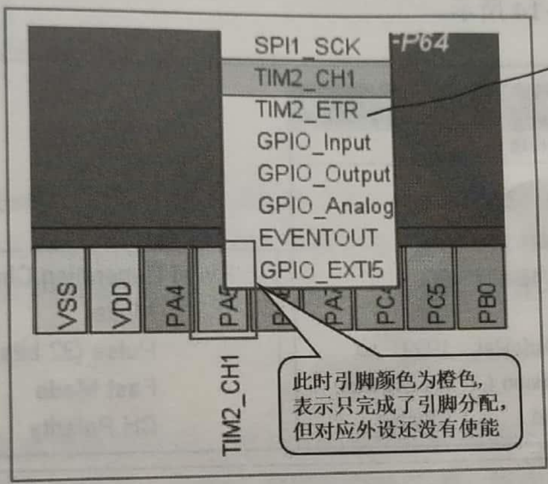


图 8-11 选择 PA5 引脚功能为 TIM2_CH1

3. 外设配置

在外设设置窗口中选择：Timers → TIM2，将展开 TIM2 的配置窗口。

(1) 在配置窗口上方的 Mode 栏，进行如下选择：

- Clock Source（时钟源）：Internal Clock（选择内部时钟）
- Channel（通道 1）：PWM Generation CH1（选择通道 1 为 PWM 输出）

具体配置如图 8-12 所示。此时，引脚 PA5 的颜色变为亮绿色，表示已经完成 PWM 引脚配置，如图 8-13 所示。

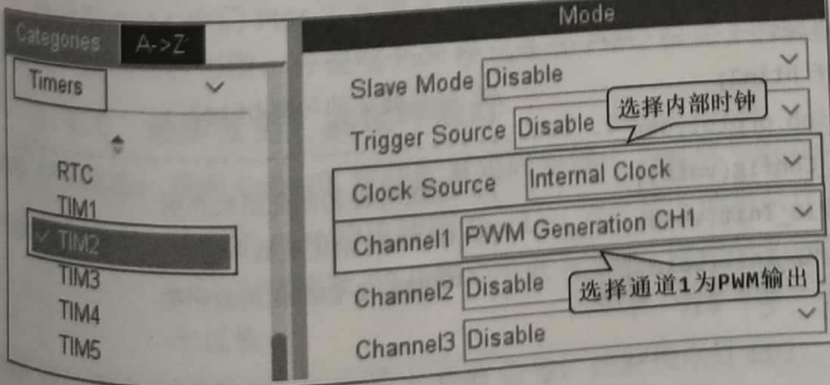


图 8-12 设置定时器 2 的工作模式参数

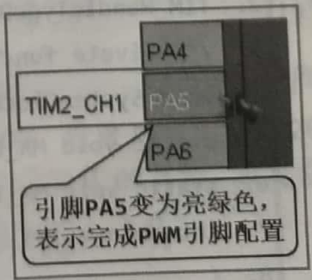


图 8-13 PA5 引脚设置完成

(2) 在配置窗口下方的 Configuration 栏，选择 Parameter Settings 标签页。

在 Counter Setting（计数器设置）部分进行如下的参数设置：

- Prescaler（预分频系数 PSC）：9999
- Counter Mode（计数模式）：Up（递增计数）
- Counter Period（自动重载值 ARR）：1999
- Internal Clock Division（时钟分频）：No Division（不分频）
- auto-reload preload（预装载功能）：Disable（预装载功能关闭）

在 PWM Generation Channel1（PWM 输出通道 1）部分进行如下的参数设置：

- Mode（输出模式）：PWM mode 1（输出模式 1）
- Pulse（占空比）：1000（50%占空比）
- Fast Mode（快速输出模式）：Disable（不使能快速输出模式）
- CH Polarity（输出极性）：输出高电平有效

具体的配置如图 8-14 所示。

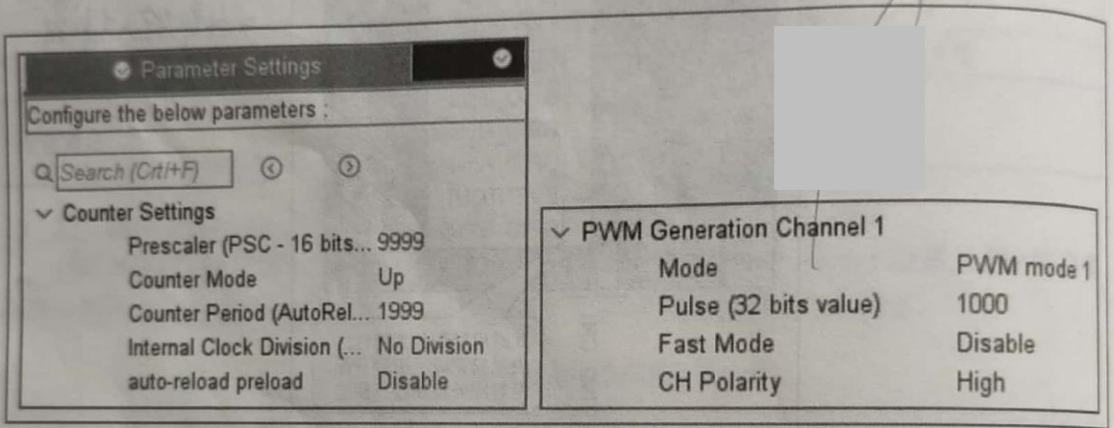


图 8-14 设置定时器 2 的 PWM 输出参数

4. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。主程序部分将会自动定义一个定时器 2 的句柄 htim2，并且自动生成定时器 2 的初始化函数 MX_TIM2_Init。完成定时器 2 的初始化之后，调用轮询模式下的定时器 PWM 输出启动函数 HAL_TIM_PWM_Start，使定时器的通道 1 输出 PWM 信号。While 循环中不进行任何操作。主程序部分的程序如程序清单 8-9 所示。

程序清单 8-9 输出 PWM 信号任务的主程序

```
/* Private variables -------------------------------------------------------*/
TIM_HandleTypeDef htim2;                    // 定时器 2 的句柄 htim2
/* Private function prototypes -- ------------------------------------------*/
void SystemClock_Config(void);              // 系统时钟初始化函数声明
static void MX_GPIO_Init(void);             // GPIO 初始化函数声明
static void MX_TIM2_Init(void);             // 定时器 2 初始化函数声明
int main(void)
{
  HAL_Init();                               // HAL 初始化
```

```
10. SystemClock_Config();                          // 系统时钟初始化
11. MX_GPIO_Init();                                // GPIO 初始化
12. MX_TIM2_Init();                                // 定时器 2 初始化
13. /* USER CODE BEGIN 2 */
14. HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1);//启动定时器 2 的通道 1 输出 PWM 信号
15. /* USER CODE END 2 */
16. /* Infinite loop */
17. /* USER CODE BEGIN WHILE */
18. while (1)                                      // while 循环
19. {
20.   /* USER CODE END WHILE */
21.   /* USER CODE BEGIN 3 */
22. }
23. /* USER CODE END 3 */
24. }
```

➡ 程序解析

① 第 2 行是由 CubeMX 自动生成的定时器 2 的句柄 htim2。

② 第 14 行是调用轮询模式下的定时器 PWM 输出启动函数 HAL_TIM_PWM_Start，启动定时器 2 的通道 1 输出 PWM 信号，由用户添加。注意调用时传入的入口参数是句柄的地址：&htim2。

➡ 运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。可以观察到：由于输出周期 200ms，占空比 50%的 PWM 信号，使得在高电平周期指示灯 LD2 开启，在低电平周期指示灯关闭，形成每隔 100ms 的指示灯闪烁效果。

在本任务中，定时器的初始化代码由 CubeMX 按照图 8-14 的设置自动生成，对应的初始化函数名为 MX_TIM2_Init，在 main 函数中自动调用。初始化函数中，将会定义输出比较配置结构体变量 sConfigOC，它的成员变量用于存放初始化参数，然后调用 PWM 输出通道配置函数 HAL_TIM_PWM_ConfigChannel 将参数写入对应的硬件寄存器，最后调用 MCU 相关初始化函数 HAL_TIM_MspPostInit 将引脚 PA5 设置为定时器 2 的通道 1 输出。感兴趣的读者可以自行阅读源代码。

你的任务

修改图 8-14 中的 PWM 工作模式和输出极性，利用示波器观察一下各种配置下输出波形的变化，绘制出波形图，并说明 PWM 模式 1 和 PWM2 模式 2 的区别。

8.4.6 进阶任务：实现呼吸灯

➡ 任务目标： 掌握 CubeMX 软件配置定时器输出 PWM 信号的方法

➡ 任务内容： 请利用 PWM 输出控制 Nucleo 开发板上的指示灯 LD2。设置 PWM 周期为 20 ms，占空比从 0%开始，步进为 20%。递增到 100%后，又从 0%开始，重复整个过程。

➡ 硬件原理： 设置 PA5 引脚输出 PWM 信号，控制指示灯 LD2。

➡ 任务实现

1. 设计思路

定义 2 个变量：占空比 Duty 和步进值 Step。Duty 从 0 逐次步进到 100，步进值为 20，步进时间间隔设置为 100ms。PWM 信号的周期为 20ms，定时器 2 的定时时钟 TIM2_CLK 为 100MHz，根据公式 8-1 可以设置预分频系数 PSC 为 9999，自动重载值 ARR 为 199。

由于占空比从 0%开始，因此写入捕获/比较寄存器的初值为 0，然后在 while 循环中调用定时器比较/捕获寄存器设置函数__HAL_TIM_SET_COMPARE 修改 CCR 中的内容，从 0 开始，逐渐增加到 ~~100~~ 200，步进为 ~~20~~，然后重复该过程。
+0 40

2. 引脚分配

本任务所使用的引脚与基础任务中所使用的相同，因此省去引脚分配的过程。

3. 外设设置

根据设计思路中的分析，设置预分频系统 PSC 为 9999，自动重载值 ARR 为 199，占空比 Pulse 为 0。具体配置参数如图 8-15 所示。

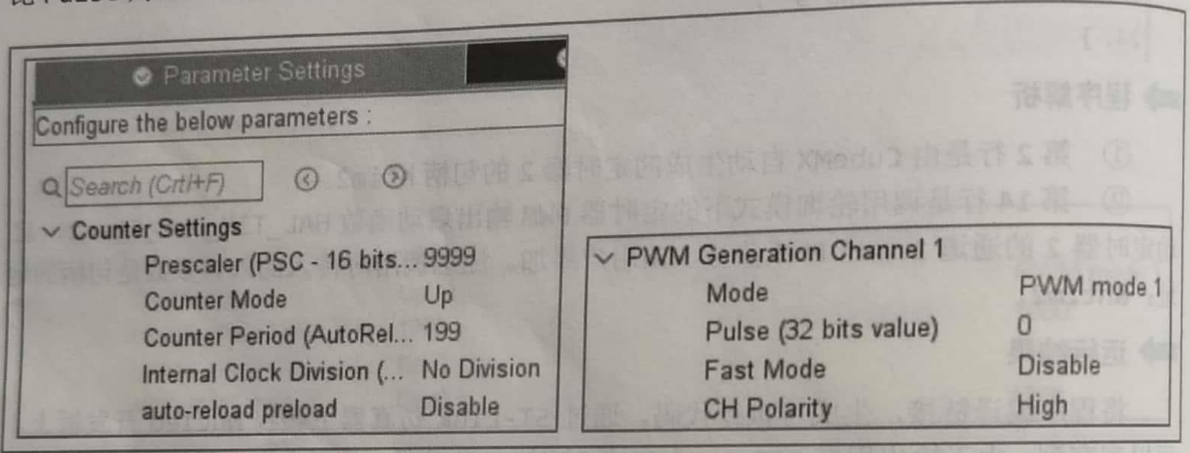


图 8-15 设置定时器 2 的 PWM 输出参数

4. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。限于篇幅关系，这里重点把用户编写的代码列出来。主程序的具体代码如程序清单 8-10 所示。

程序清单 8-10 呼吸灯任务的主程

```
/* USER CODE BEGIN PV */
uint16_t Duty  = 0;      // 占空比
uint16_t Step  = 20;     // 步进值
/* USER CODE END PV */
/* USER CODE BEGIN 2 */
HAL_TIM_PWM_Start(&htim2, TIM_CHANNEL_1); //启动定时器 2 的通道 1 输出 PWM 信号
/* USER CODE END 2 */
while (1)
{
/* USER CODE BEGIN 3 */
   for( Duty = 0 ; Duty <= 100 ; Duty = Duty + Step )
   {
```

```
13.     __HAL_TIM_SET_COMPARE(&htim2,TIM_CHANNEL_1,Duty); // 修改占空比
14.     HAL_Delay(100);                                   // 延时 100ms
15.   }
16. }
17. /* USER CODE END 3 */
```

程序解析

① 第 2~3 行是用户定义的变量：Duty 表示 PWM 信号的占空比，Step 表示占空比每次变化的步进值。

② 第 6 行是调用轮询模式下的定时器 PWM 输出启动函数 HAL_TIM_PWM_Start，启动定时器 2 的通道 1 输出 PWM 信号，由用户添加。

③ 第 11~15 行利用 for 循环改变占空比，循环初值为 0，循环终值 100，步进 20。表示占空比从 0%变化到 100%，步进 20%，其中第 13 行调用定时器比较/捕获寄存器设置函数__HAL_TIM_SET_COMPARE 修改 CCR 中的内容。

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上，可以观察到指示灯 LD2 从熄灭状态，然后亮度逐渐增强，最后到最亮，并循环该过程。

你的任务

修改程序，使指示灯从灭到亮的渐变后，又从亮到灭的渐变，并重复该过程。

提示：在 while 循环增加一个 for 循环，循环初值为 200，终值为 0，步进值为-20。

8.5 本章小结

本章重点介绍了 STM32 的定时器模块。以定时器的功能框图为引入，重点介绍了定时器的基本定时器功能和 PWM 输出功能，以及在 HAL 库中的数据类型和接口函数的定义。

本章以定时器为例，介绍了 HAL 库中外设模块的设计方法，包括外设句柄，外设编程模型和外设通用接口函数，这些方法需要读者在学习其他外设时深入体会。

PWM 输出功能在工程应用中使用非常广泛，读者可以利用 Nucleo 开发板的 Arduino 接口接入一个舵机，利用 PWM 信号控制舵机的转动。

由于 STM32 定时器的功能非常丰富，本章只是介绍了几个常用的功能，其他的功能，如编码器接口、互补输出、单脉冲输出、输入捕获等，还需要读者仔细阅读芯片的参考手册来掌握它们的使用方法。

习题 6

1. 分析定时器的中断流程，并与外部中断的中断流程比较。
2. 分别计算定时 1us、10us、100us、1ms、10ms、100ms 对应的预分频系数 PSC 和自动重载值 ARR，总结出 PSC 和 ARR 的计算方法。
3. 完成定时器计数的测试，利用按键 B1 作为脉冲的触发，按键按下后，连续发出 100 个脉冲，送到定时器的计数引脚计数，并通过串口发送到 PC 显示。
4. 利用定时器实现微秒级的延时。即编写一个延时函数，功能类似于 HAL 库提供的延时函数 HAL_Delay。函数的入口参数是延时时间，单位为 us。

5. HAL 的延时函数 HAL_Delay 是利用系统节拍定时器 Systick 实现的，请仔细阅读 HAL 库的初始化函数 HAL_Init，找到其中与系统节拍定时器设置相关的代码，写出每一行代码的作用。Systick 模块的寄存器介绍可以查阅芯片的参考手册。

6. 尝试编写软件定时器的 BSP。

第九章　串口通信

本章导读

本章主要介绍 STM32 的串口。首先介绍通信的基本概念。然后通过 STM32 串口通信的结构框图分析其工作原理。同时以串口通信为例，介绍 HAL 库中外设初始化的设计思想。在具体的通信方式上，介绍了轮询、中断和 DMA 三种方式，以及 HAL 库中对应的数据类型和接口函数的定义，并通过阶梯式任务实践来掌握它们的用法。

本章知识点

通信的基本概念

HAL 库外设初始化设计思想

轮询方式的串口通信

中断方式的串口通信

DMA 方式的串口通信

9.1　概要

9.1.1　计算机通信

在实际应用中，计算机的 CPU 常常要和它的外部设备之间进行信息的交换，同时一台计算机和另一台计算机之间往往也要交换信息。所有的信息交换我们都可称为“通信”。

通信的基本方式一般分为并行和串行两种方式。

1. 并行通信

并行通信是指将数据字节的各位用多条数据线同时进行传送的通信方式。这种方式的特点是控制简单、传输速度快，但由于传输线较多，长距离传送时硬件成本较高。并行通信的传输示意图如图 9-1 所示。

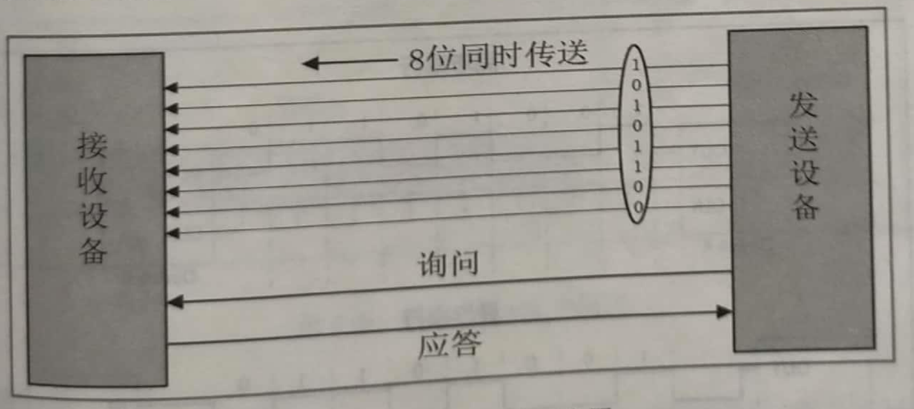


图 9-1　同步通信方式示意图

2. 串行通信

并行通信是指将数据字节分成一位一位的形式在一条传输线上逐个地传送。串行通信的特点是传输线少，长距离传送时成本低，但数据的传送控制比并行通信复杂。串行通信的传输示意图如图 9-2 所示。

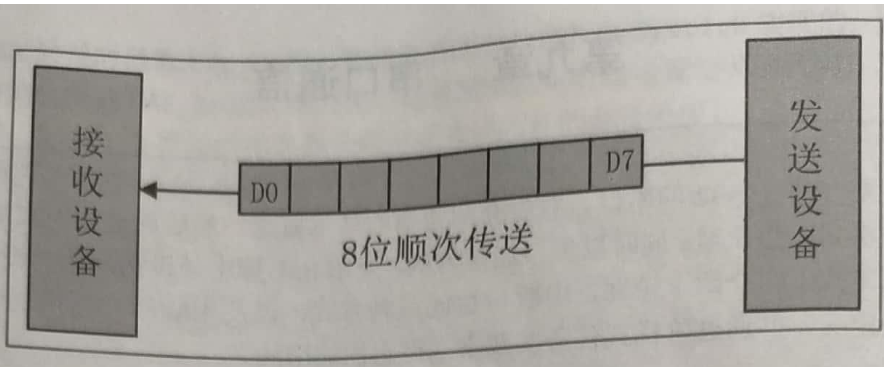


图 9-2 串行通信方式示意图

9.1.2 串行通信基本概念

1. 异步通信与同步通信

在现在的计算机通信中，几乎所有的通信都是串行通信方式，比如我们常用的网络通信和 USB 通信。由于数据串行发送，收发双方需要考虑数据的同步问题：即数据什么时候开始传输，什么时候结束传输，数据中的每一位持续的时间是多久。

按照实现数据同步的不同方法，串行通信又可以分成同步串行和异步串行两类。

- 同步串行

同步通信的收发双方通过建立专门的时钟信号使双方达到完全同步。产生时钟的一方称为主机（Master），接收时钟的一方称为从机（Slaves）。此类通信的示例包括 SPI、I^2C 和 USB。

同步串行通信的优点是可以实现高速度、大容量的数据传送，缺点是要求发送时钟和接收时钟保持严格同步，通信协议及硬件电路较为复杂。

- 异步串行

异步通信的收发双方没有专门的时钟信号，而是依靠事先约定的字符格式和传输波特率来完成通信。此类通信的典型例子就是我们的异步串口（UART）。

同步通信和异步通信的传输示意图如图 9-3 所示。

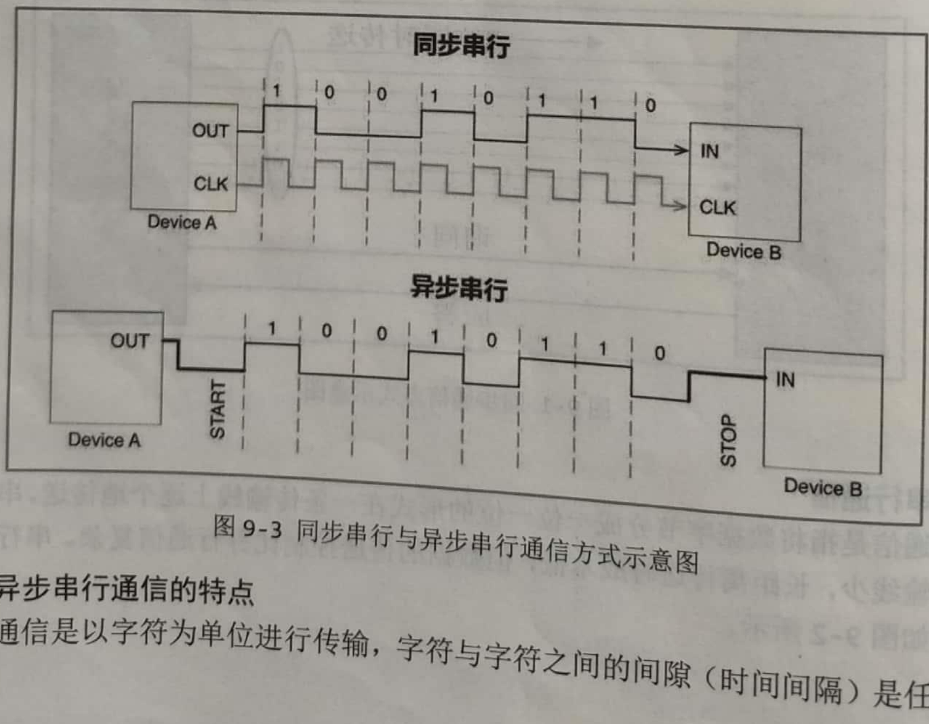


图 9-3 同步串行与异步串行通信方式示意图

2. 异步串行通信的特点

异步通信是以字符为单位进行传输，字符与字符之间的间隙（时间间隔）是任意的，但

每个字符内部的各位是以固定的时间间隔传送的，即每一位在通信线路上持续的时间是相同的。异步通信的传输示意图如图 9-4 所示。

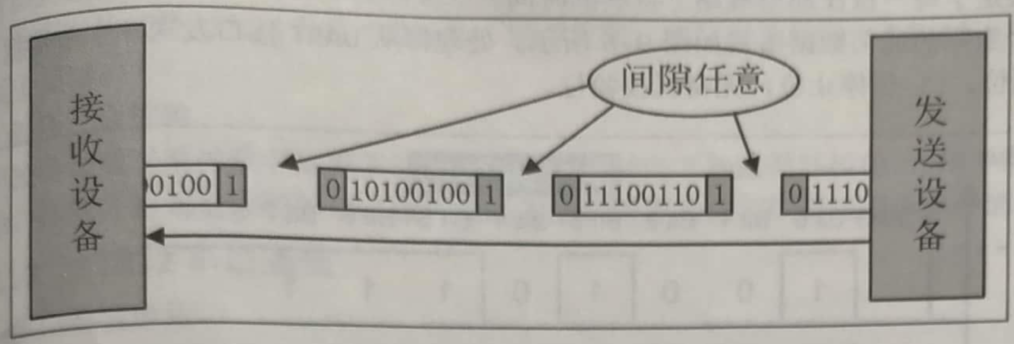


图 9-4 异步通信方式示意图

(1) 数据格式

由于字符的发送是随机的，因此收发双方必须规定通信字符的格式来判断数据的开始和结束。

异步串行通信的一个字符一般由四部分组成：起始位、数据位、奇偶校验位和停止位。一个字符由起始位开始，停止位结束。起始位为低电平（逻辑 0），占用一位，用来通知接收设备，一个新的字符开始到来。线路上在不传输数据的时候应该一直保持高电平（逻辑 1）。接收端不断检测线路的状态。若在连续的 1 之后检测到一个 0，就知道一个新的字符出现了，马上开始准备接收。起始位以后是数据位，数据位的位数可以由实际需要的通信速率来决定，一般为 8 位。在发送数据时，一般是低位在前，高位在后。奇偶校验位只占用一位，用于检测数据是否出错。停止位只占用一位，用来表征一个字符的结束，它一定是高电平（逻辑 1）。接收端接收到停止位时，就表示这一个字符已经结束。同时也为下一个字符的接收做好准备（只要再接收到 0 就表示新的字符来到）。图 9-5 给出了串行通信的数据格式。

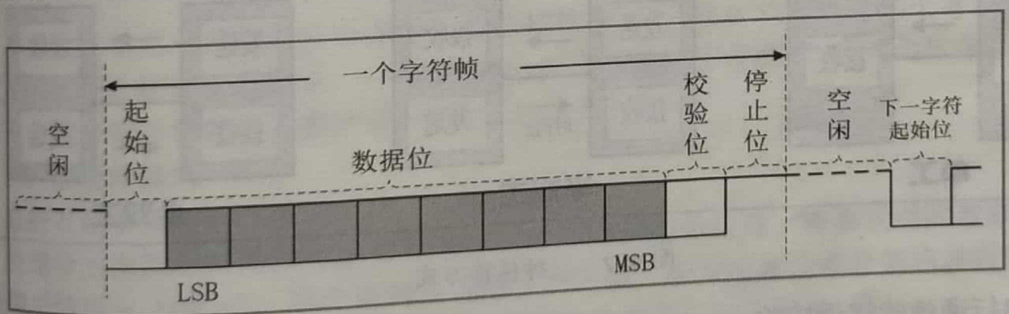


图 9-5 串行通信的数据格式

(2) 波特率

在串行通信中还有一个重要的指标叫波特率。它的定义是每秒钟传送二进制数码的位数，以位/秒为单位。波特率反映了串行通信的速率，在实际中，它为每秒传送的字符数和每个字符位数的乘积。

例如：波特率为 115200bit/s 就表示每秒传输 115200 位。假设传输的字符格式为 1 位起始位、8 位数据位、1 位停止位、无奇偶校验，即一个字符包含了 10 位，因此每秒可以传输字符的速率为 11520 字符/s。由于一个字符中的有效数据为 8 位，即一个字节，因此也可以说每秒传输 11520 个字节数据（起始位和停止位不算有效数据，由发送方自动添加）。

从上面的叙述我们可以看到：在异步串行通信中，收发双方能够准确传输数据的关键就是要约定两件事：一是通信字符的格式；二是通信的波特率。前者决定每一个字符的位数，后者则决定了每一位在通信线路上持续的时间。

一个实际的通信数据格式如图 9-6 所示。处理器从 UART 接口发送一字节数据 0xE9，8 位数据位，1 位停止位，无奇偶校验位。

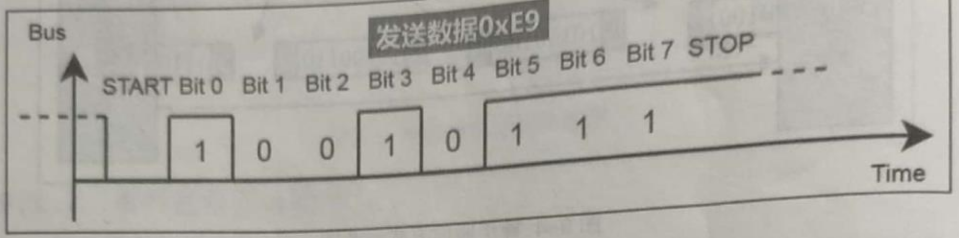


图 9-6 串行数据时序（0xE9）

2. 串行通信的传输方向

在串行通信中按照数据传输的方向，还可以分成以下三类：

(1) 单工

单工是指数据传输仅能沿一个方向，不能实现反向传输，只有一条通行线路。

(2) 半双工

半双工是指数据传输可以沿两个方向，但需要分时进行，也是只有一条通信线路。

(3) 全双工

全双工是指数据可以同时进行双向传输，一般有两条通信线路。我们的 UART 就属于全双工方式。

三种通信方式的示意图如图 9-7 所示。

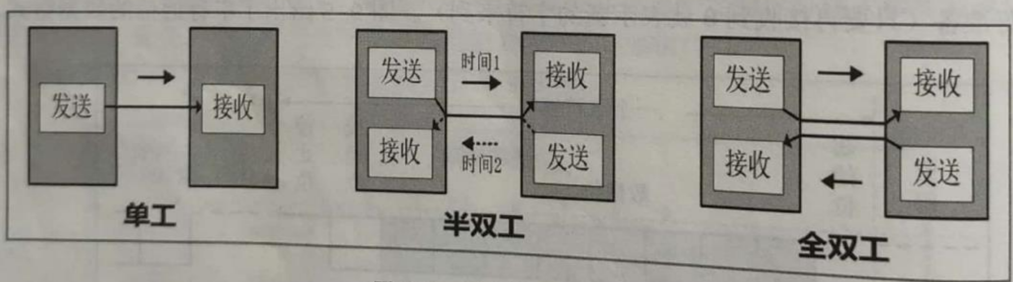


图 9-7 三种传输方式

3. 串行通信的错误校验

(1) 奇偶校验

在发送数据时，数据位尾随的 1 位为奇偶校验位（1 或 0）。奇校验时，数据中“1”的个数与校验位“1”的个数之和应为奇数；偶校验时，数据中“1”的个数与校验位“1”的个数之和应为偶数。接收字符时，对“1”的个数进行校验，若发现不一致，则说明传输数据过程中出现了差错。奇偶检验只能检验出单个位数出错的情况。

例如，数据位为 8 位，传输内容为 00110101，数据中有 4 个 1。如果采用奇校验，则校验位应为 1，以保证数据位和校验位中“1”的个数之和为奇数。如果采用偶校验，则校验位应为 0，以保证数据位和校验位中“1”的个数之和为偶数。校验位的产生一般由硬件自动完成，用户只需要使能相应的控制位即可。

(2) 代码和校验

代码和校验是发送方将所发数据块求和（或各字节异或），产生一个字节的校验字符（校验和）附加到数据块末尾。接收方接收数据同时对数据块（除校验字节外）求和（或各字节异或），将所得的结果与发送方的“校验和”进行比较，相符则无差错，否则即认为传送过程中出现了差错。

(3) 循环冗余校验

这种校验是通过某种数学运算实现有效信息与校验位之间的循环校验，常用于对磁盘信息的传输、存储区的完整性校验等。这种校验方法纠错能力强，广泛应用于串行通信中。

9.1.3 STM32 串口通信

1. 串口收发单元

STM32 串口的核心部分为串口收发单元，其功能框图如图 9-8 所示。

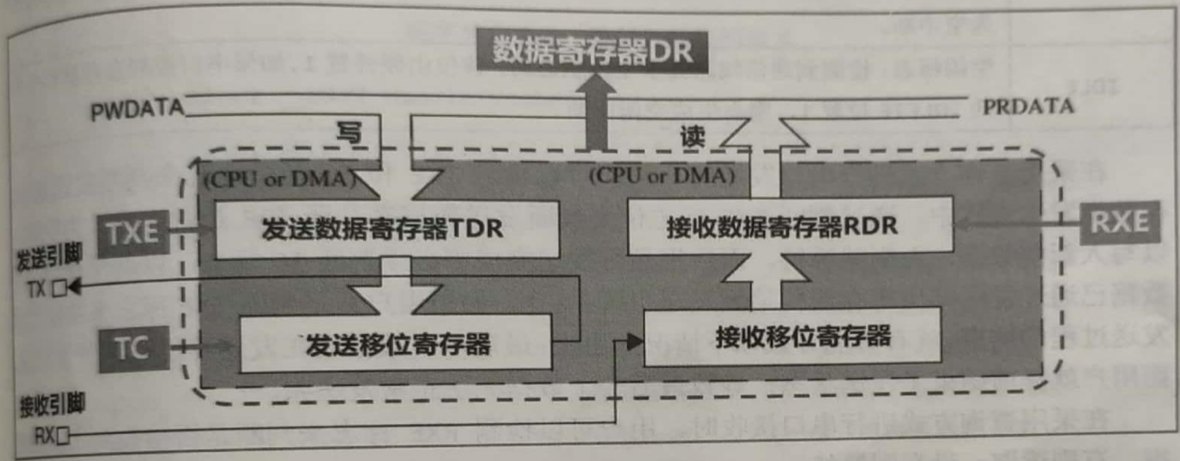


图 9-8 串口收发单元功能框图

串口收发单元对应的硬件寄存器为数据寄存器 DR，该寄存器包含两个寄存器，一个用于发送的发送数据寄存器 TDR，一个用于接收的接收数据寄存器 RDR，因此具有双重功能（读和写），两个寄存器通过数据的读/写操作来区分。

⑴ 双缓冲模式

数据发送单元和接收单元采用了双缓冲模式。

发送单元由发送数据寄存器和发送移位寄存器两部分组成。需要发送的数据由 CPU 通过数据总线写入 TDR 后，再传送到发送移位寄存器，由发送移位寄存器将并行数据转换为串行数据，按照低位在前，高位在后的方式逐位从数据发送引脚 TX 输出。

发送单元采用双缓冲模式的好处是：当 TDR 寄存器的数据送入发送移位寄存器后，在数据发送的同时，可以把新的数据写入 TDR 寄存器。这样写入新数据和发送数据可以同时进行，提高了数据发送的效率。

接收单元由接收数据寄存器和接收移位寄存器两部分组成。接收移位寄存器从接收引脚 RX 上将数据逐位移入寄存器内，当 8 位数据都接收完成后，再一起送入接收数据寄存器 RDR，CPU 通过数据总线在 RDR 寄存器中读取接收到的数据。

⑵ 收发状态标志

在数据的收发过程中，CPU 如何知道什么时候可以送入新的数据？什么时候可以读取接收的数据？这就需要一个能够反映收发状态的寄存器。在 STM32 的串口外设中设计了一个

状态寄存器 SR，该寄存器内设计了 4 个标志来反映收发状态。具体描述如表 9-1 所示。

表 9-1　收发标志状态

标志位名称	含义
TXE	发送数据寄存器空标志。当 TDR 寄存器的内容已经传送到移位寄存器时，该位由硬件置 1。如果串口控制寄存器 CR1 中的 TXEIE 位置 1，则会生成发送数据寄存器空中断。注意：当 TXE 置 1 时，数据有可能还在发送。
TC	发送完成标志。当发送移位寄存器的内容发送完成，同时发送数据寄存器也为空时，表明没有新的数据写入，数据已经彻底传输完成，则此位由硬件置 1。如果串口控制寄存器 CR1 中的 TCIE 位置 1，则会生成发送完成中断。注意：当 TC 置 1 时，数据已真正完成发送。
RXE	接收数据寄存器不为空标志。当移位寄存器的内容已传输到接收数据寄存器 RDR 时，该位由硬件置 1。如果串口控制寄存器 CR1 中的 RXNEIE 位置 1，则会生成接收数据寄存器不为空中断。
IDLE	空闲标志。检测到通信线路处于空闲状态时，该位由硬件置 1.如果串口控制寄存器 CR1 中的 IDLEIE 位置 1，则会生成空闲中断。

在采用查询方式进行串口发送时，用户可以检测 TXE 和 TC 两个标志来判断发送状态。在数据发送过程中，通过判断 TXE 标志位来判断发送数据寄存器 TDR 是否为空，为空则可以写入新的数据，否则就等待。而数据是否发送完成则必须判断 TC 标志，以确保最后一个数据已通过发送移位寄存器传输到发送引脚 TX 上。如果用户只是判断 TXE 标志来决定本次发送过程的结束，就有可能导致以下情况的出现：最后一个数据还在发送移位寄存器里发送，而用户就提前结束了发送状态，导致最后一个数据无法正常发送完。

在采用查询方式进行串口接收时，用户可以检测 RXE 标志来判断是否接收到了新的数据，有则读取，没有则等待。

在中断发送下：如果串口控制寄存器 CR1 中 TXEIE 位置 1，则会生成发送寄存器空中断；TCIE 位置 1，则会生成发送完成中断；RXNEIE 位置 1，则会生成数据接收中断。用户在串口中断服务程序中再检测相应的标志位，判断出中断的类型，然后进行后续的任务处理：如果是 TXE 为 1，则表示 CPU 可以写入新的数据；如果是 TC 为 1，则表示本次数据发送完成；如果是 RXE 为 1，则表示 CPU 可以读取新的数据。

2.　**串口通信引脚**

STM32 的串口收发引脚 RX 和 TX 与 GPIO 引脚复用，可以有多个引脚选择。STM32F411 芯片具有 3 个串口，编号分别是 UART1、UART2 和 UART6。各串口对应的可选引脚如表 9-2 所示。

表 9-2　STM32F411 的 UART 引脚

串口号	TX 引脚	RX 引脚
UART1	PA9/PB6	PA10/PB7
UART2	PA2	PA3
UART6	PC6	PC7

在 Nucleo 开发板上，默认使用了 UART2，对应的 TX 引脚是 PA2，RX 引脚是 PA3。这两个引脚接入到 ST-Link 仿真器，由仿真器完成串口通信到 USB 通信的转换。

9.2 HAL 库外设初始化设计思想

在 GPIO 的初始化过程中，调用了函数 HAL_GPIO_Init 完成引脚的初始化，初始化过程只要将引脚初始化结构类型 GPIO_InitTypeDef 的成员变量写入对应的硬件寄存器即可。而对于串口等较为复杂的外设，其初始化过程则分成了抽象和重载两个步骤实现。

9.2.1 串口外设句柄定义

在介绍串口的初始化过程之前，我们先看看串口句柄和串口初始化数据类型的定义。串口数据类型的定义以及接口函数的声明放在 stm32f4xx_hal_uart.h 文件中，串口接口函数的定义则放在 stm32f4xx_hal_uart.c 文件中。

HAL 库中用于定义串口外设的外设句柄为 UART_HandleTypeDef，其具体定义如程序清单 9-1 所示。

程序清单 9-1　串口外设句柄定义

```
typedef struct __UART_HandleTypeDef
{
    USART_TypeDef                   *Instance;      // 串口寄存器的基地址定义
    UART_InitTypeDef                Init;           // 串口通信参数
    uint8_t                         *pTxBuffPtr;    // 串口发送缓冲区首地址
    uint16_t                        TxXferSize;     // 串口待发送数据个数
    __IO uint16_t                   TxXferCount;    // 串口发送数据计数器
    uint8_t                         *pRxBuffPtr;    // 串口接收缓冲区首地址
    uint16_t                        RxXferSize;     // 串口待接收数据个数
    __IO uint16_t                   RxXferCount;    // 串口接收数据计数器
    DMA_HandleTypeDef               *hdmatx;        // 串口 DMA 发送时的句柄
    DMA_HandleTypeDef               *hdmarx;        // 串口 DMA 接收时的句柄
    HAL_LockTypeDef                 Lock;           // 保护锁类型定义
    __IO HAL_UART_StateTypeDef      gState;         // 串口全局状态和发送状态信息
    __IO HAL_UART_StateTypeDef      RxState;        // 串口接收状态信息
    __IO uint32_t                   ErrorCode;      // 串口错误代码
} UART_HandleTypeDef;
```

定义解析

- USART_TypeDef　*Instance

串口寄存器的基地址定义，用于访问硬件寄存器。具体的做法是：先将串口的所有寄存器封装为名为 USART_TypeDef 的结构体类型，然后将串口外设的首地址强制转换为该结构体类型的指针，再通过结构体指针加成员变量的形式访问各个硬件寄存器。这种方法在介绍 GPIO 外设时已经介绍过，感兴趣的读者可自行查询源码。

该结构体指针也常常称为外设实例，如串口 1 对应的实例名：UART1。利用 UART1->SR 就可以访问串口 1 的 SR 寄存器。

- UART_InitTypeDef　Init

定义串口通信的初始化参数，采用结构体形式定义。具体含义将在下一小节介绍。

- *pTxBuffPtr、TxXferSize、TxXferCount

这三个成员变量与串口发送缓冲区相关。pTxBuffPtr 指向发送缓冲区的首地址，TxXferSize 表示需要发送的数据个数，TxXferCount 用于在发送过程中记录已经发送的数据个数。

- *pRxBuffPtr、RxXferSize、RxXferCount

这三个成员变量与串口接收缓冲区相关。pRxBuffPtr 指向接收缓冲区的首地址，RxXferSize 表示待接收的数据个数，RxXferCount 用于在接收过程中记录已经接收的数据个数。

- *hdmatx 和*hdmarx

定义串口产生 DMA 请求时所使用的 DMA 通道的句柄。

- HAL_LockTypeDef   Lock

定义保护锁类型。保护锁是 HAL 库中提供的一种安全机制，避免对串口等资源型外设的并发访问。

- gState 和 RxState

定义串口的工作状态，提供给状态接口函数读取。

- __IO uint32_t   ErrorCode

定义串口的错误代码。

9.2.2 串口外设初始化数据类型定义

串口外设用于通信参数初始化的数据类型是 UART_InitTypeDef，其定义如程序清单 9-2 所示。

程序清单 9-2 串口通信参数初始化结构定义

```
typedef struct
{
  uint32_t BaudRate;          // 指定通信波特率
  uint32_t WordLength;        // 指定通信字符中数据位的位数
  uint32_t StopBits;          // 指定发送时停止位的位数
  uint32_t Parity;            // 指定奇偶校验模式
  uint32_t Mode;              // 指定接收或发送模式是否使能或禁能
  uint32_t HwFlowCtl;         // 指定硬件流控是否使能或禁能
  uint32_t OverSampling;      // 指定采样频率和信号传输频率的比例
} UART_InitTypeDef;
```

定义解析

- BaudRate

通信波特率的设置，单位 bit/s，表示串口通信的传输速率。

- WordLength

用于设置串口发送字符或接收字符中数据位所占长度，可以是 8 位或者 9 位。该参数可以是 UART_Word_Length 的值之一，具体描述如表 9-3 所示。

表 9-3 串口通信数据位长度 UART_Word_Length

宏常量定义	含义
UART_WORDLENGTH_8B	数据位长度为 8 位

UART_WORDLENGTH_9B	数据位长度为 9 位

- StopBits

用于设置串口发送字符或接收字符中停止位所占长度，可以是 1 位或者 2 位。该参数可以是 UART_Stop_Bits 的值之一，具体描述如表 9-4 所示。

表 9-4 串口通信停止位长度 UART_Stop_Bits

宏常量定义	含义
UART_STOPBITS_1	停止位长度为 1 位
UART_STOPBITS_2	停止位长度为 2 位

- Parity

用于设置奇偶校验模式，可以选择无奇偶校验、偶校验、奇校验三种模式，具体描述如表 9-5 所示。

表 9-5 串口通信奇偶检验设置 UART_Parity

宏常量定义	含义
UART_PARITY_NONE	无奇偶校验
UART_PARITY_EVEN	偶校验
UART_PARITY_ODD	奇校验

- Mode

用于使能发送或接收模式，该参数可以是 UART_Mode 的值之一，具体描述如表 9-6 所示。

表 9-6 串口通信模式设置 UART_Mode

宏常量定义	含义
UART_MODE_RX	串口仅处于接收模式，只能接收数据，不能发送数据
UART_MODE_TX	串口仅处于发送模式，只能发送数据，不能接收数据
UART_MODE_TX_RX	串口处于接收和发送模式，可以同时收发数据

- HwFlowCtl

用于硬件流控是否使能或禁能，该参数可以是 UART_Hardware_Flow_Control 的值之一，具体描述如表 9-7 所示。

表 9-7 串口通信硬件流控设置 UART_Hardware_Flow_Control

宏常量定义	含义
UART_HWCONTROL_NONE	无硬件流控
UART_HWCONTROL_RTS	使能“请求发送（RTS）”引脚
UART_HWCONTROL_CTS	使能“允许发送（CTS）”引脚
UART_HWCONTROL_RTS_CTS	使能“请求发送（RTS）”和“允许发送（CTS）”引脚

- OverSampling

指定采样频率和信号传输频率的比例，该参数可以是 UART_Over_Sampling 的值之一，

具体描述如表 9-8 所示。

表 9-8　串口通信过采样频率设置 UART_Over_Sampling

宏常量定义	含义
UART_OVERSAMPLING_16	采样频率是信号传输频率的 16 倍
UART_OVERSAMPLING_8	采样频率是信号传输频率的 8 倍

9.2.3 串口初始化过程

对于定时器（TIMER）、串口（UART）和模/数转换器（ADC）等功能较为复杂的外设，其初始化过程不仅包括配置参数的初始化，还包括引脚复用功能的初始化。因此在复杂外设的初始化过程中，采用了从抽象到承载的设计思想。

我们以串口为例来介绍这种设计思想。串口的初始化过程包含了两个步骤：

1. **串口的抽象**

串口的抽象是指与 MCU 无关的参数，比如串口通信中的数据位数、波特率以及奇偶校验位等。这些参数的设置属于通用配置，与使用什么样的 MCU 没有任何关系，可以使用 STM32F1 系列的 MCU，也可以是 STM32F4 系列的 MCU，甚至可以是 PC 上的串口。

2. **串口的承载**

串口的承载是指与具体 MCU 相关的初始化，将之前抽象的串口在具体的 MCU 上实现，主要是硬件引脚的配置。以 STM32F411 为例，使用 UART2 与 PC 进行通信，就需要配置引脚 PA2 作为 UART2 的数据发送引脚，引脚 PA3 作为 UART2 的数据接收引脚。

对于外设的抽象，HAL 库提供了外设通用初始化函数 HAL_PPP_Init（PPP 表示外设名称），该函数位于 stm32f4xx_hal_ppp.c 文件中。对于外设的承载，HAL 库提供了外设的 MCU 相关初始化函数 HAL_PPP_MspInit，其中的 MSP 表示 MCU Support，该函数位于 stm32f4xx_hal_msp.c 文件中。具体的初始化流程如图 9-9 所示。

图 9-9　外设分级初始化流程

HAL 库中对串口的初始化步骤如下：

(1) 定义外设句柄 huartn，n 表示串口号；

(2) 初始化外设句柄的各个参数，然后调用串口通用初始化函数 HAL_UART_Init 将参数设置写入硬件寄存器；

(3) 在函数 HAL_UART_Init 内部再调用串口 MCU 相关初始化函数 HAL_UART_MspInit 初始化串口所使用的相关引脚。

假设使用 UART2，设置波特率 115200，8 位数据位，1 位停止位，无奇偶校验，它的初始化示例程序如程序清单 9-3 所示。

程序清单 9-3　串口 2 初始化代码

```
1. int main(void)
2. {
3.   UART_HandleTypeDef huart2;                              // 定义串口 2 句柄
4.   HAL_Init();                                             // 初始化 HAL 库
5.   SystemClock_Config();                                   // 初始化时钟
6.   // 串口 2 通信参数配置
7.   huart2.Instance              = USART2;                  // 配置的串口号
8.   huart2.Init.BaudRate         = 115200;                  // 通信波特率
9.   huart2.Init.WordLength       = UART_WORDLENGTH_8B;      // 数据位为 8 位
10.  huart2.Init.StopBits         = UART_STOPBITS_1;         // 停止位为 1 位
11.  huart2.Init.Parity           = UART_PARITY_NONE;        // 无奇偶校验
12.  huart2.Init.Mode             = UART_MODE_TX_RX;         // 使能接收和发送模式
13.  huart2.Init.HwFlowCtl        = UART_HWCONTROL_NONE;     // 无硬件流控
14.  huart2.Init.OverSampling = UART_OVERSAMPLING_16; // 16 倍过采样
15.  HAL_UART_Init(&huart2);
16.  ...
17. }
```

➡ 程序解析

① 第 3 行定义串口 2 的句柄 huart2。

② 第 4 行是 HAL 库的初始化。

③ 第 5 行是时钟系统的初始化。

④ 第 7～14 行进行串口参数设置，将参数写入句柄的串口初始化成员变量 Init 中。具体设置为：波特率 115200，8 位数据位，1 位停止位，无奇偶校验，使能接收和发送模式，无硬件流控，16 倍过采样。

⑤ 第 15 行调用函数 HAL_UART_Init 完成串口初始化，入口参数为串口 2 句柄 huart2 的地址。

在函数 HAL_UART_Init 内部再调用串口 MCU 相关初始化函数 HAL_UART_MspInit 初始化串口所使用的相关引脚，具体代码如程序清单 9-4 所示。

程序清单 9-4　串口 2 引脚初始化

```
void HAL_UART_MspInit(UART_HandleTypeDef* huart)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};     // 定义引脚初始化结构体，并赋值为 0
  if(huart->Instance==USART2)                 // 判断串口号
  {
    __HAL_RCC_USART2_CLK_ENABLE();            // 使能串口 2 时钟
    __HAL_RCC_GPIOA_CLK_ENABLE();             // 使能端口 GPIOA 时钟
    // 串口引脚复用，设置 PA2 为 USART2_TX 引脚，PA3 为 USART2_RX 引脚
    GPIO_InitStruct.Pin = GPIO_PIN_2|GPIO_PIN_3;
    GPIO_InitStruct.Mode = GPIO_MODE_AF_PP;
```

```
11.     GPIO_InitStruct.Pull = GPIO_PULLUP;
12.     GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_VERY_HIGH;
13.     GPIO_InitStruct.Alternate = GPIO_AF7_USART2;
14.     HAL_GPIO_Init(GPIOA, &GPIO_InitStruct);
15.   }
16. }
```

程序解析

① 第 3 行定义引脚初始化结构体变量 GPIO_InitStruct，初值赋为 0。

② 第 4 行判断是对哪一个串口进行初始化。

③ 第 6 行使能 UART2 的时钟。

④ 第 7 行使能端口 GPIOA 的时钟。

⑤ 第 9～13 行初始化引脚结构体变量，设置 PA2 和 PA3 为复用功能推挽模式，上拉电阻使能，输出超高速，PA2 复用为 USART2_TX，PA3 复用为 USART2_RX。

⑥ 第 14 行调用引脚初始化函数 HAL_GPIO_Init 完成串口引脚的初始化。

9.3 轮询方式的串口通信

STM32 的串口通信支持轮询、中断和 DMA 三种方式。轮询方式要求 CPU 不断检测串口的状态标志 TXE、TC 和 RXE 等来判断数据收发的情况，当标志置位后进行后续的处理。轮询方式的程序设计比较简单，但是由于 CPU 在检测标志位期间，无法执行其他任务，因此 CPU 的利用率较低。

9.3.1 轮询方式相关的 HAL 库函数

1. 串口初始化函数（HAL_UART_Init）

函数 HAL_UART_Init 主要用于串口初始化，将串口外设句柄中的参数写入对应的硬件寄存器，并调用 MCU 相关初始化函数 HAL_UART_MspInit 完成时钟、引脚和中断等与特定 MCU 相关的初始化操作。具体描述如表 9-9 所示。

表 9-9 串口初始化函数

接口函数：HAL_UART_Init	
函数原型	HAL_StatusTypeDef HAL_UART_Init(UART_HandleTypeDef *huart)
功能描述	按照串口句柄中指定的参数初始化串口
入口参数	huart：串口句柄的地址
返回值	HAL_StatusTypeDef：HAL_OK 表示初始化成功，HAL_ERROR 表示初始化失败
注意事项	1. 调用 MCU 相关的初始化函数 HAL_UART_MspInit 完成引脚、时钟和中断的设置 2. 该函数由 CubeMX 自动生成

2. 串口轮询方式发送函数（HAL_UART_Transmit）

函数 HAL_UART_Transmit 主要用于在轮询方式下发送一定数量的数据。具体描述如表 9-10 所示。

表 9-10 串口轮询方式发送函数

接口函数：HAL_UART_Transmit	
函数原型	HAL_StatusTypeDef HAL_UART_Transmit(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)
功能描述	在轮询方式下发送一定数量的数据
入口参数 1	huart：串口句柄的地址
入口参数 2	pData：待发送数据的首地址
入口参数 3	Size：待发送数据的个数
入口参数 4	Timeout：超时等待时间
返回值	HAL_StatusTypeDef：HAL_OK 表示发送成功；HAL_ERROR 表示参数错误；HAL_BUSY 表示串口被占用；HAL_TIMEOUT 表示发送超时
注意事项	1. 连续发送数据，发送过程中判断 TXE 标志发送下一个数据，发送完成则判断 TC 标志 2. 如果在超时时间内没发送完成，则不再发送，返回超时标志（HAL_TIMEOUT） 3. 该函数由用户调用

3. 串口轮询方式接收函数（HAL_UART_Receive）

函数 HAL_UART_Receive 主要用于在轮询方式下接收一定数量的数据。具体描述如表 9-11 所示。

表 9-11 串口轮询方式接收函数

接口函数：HAL_UART_Receive	
函数原型	HAL_StatusTypeDef HAL_UART_Receive(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size, uint32_t Timeout)
功能描述	在轮询方式下接收一定数量的数据
入口参数 1	huart：串口句柄的地址
入口参数 2	pData：存放接收数据的首地址
入口参数 3	Size：待接收数据的个数
入口参数 4	Timeout：超时等待时间
返回值	HAL_StatusTypeDef：HAL_OK 表示接收成功；HAL_ERROR 表示参数错误；HAL_BUSY 表示串口被占用；HAL_TIMEOUT 表示接收超时
注意事项	1. 连续接收数据，接收过程中通过判断 RXNE 标志来接收新的数据 2. 如果在超时时间内没接收完成，则不再接收数据，返回超时标志（HAL_TIMEOUT） 3. 该函数由用户调用

9.3.2 基础任务：固定长度的数据收发

➡ 任务目标：掌握 CubeMX 软件配置串口实现轮询方式通信的方法

➡ 任务内容：从 PC 上发送 5 个字符到 Nucleo 开发板，Nucleo 开发板收到后再回传到 PC。

➡ 硬件原理：利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

➡ 任务实现

1. 设计思路

定义一个数组 uint8_t RecBuf[5]，作为接收缓冲区，数组大小为 5 个字节。

2. 引脚分配

本任务的引脚分配在外设配置中完成。

3. 外设配置

在外设设置窗口中选择：Connectivity → USART，将展开 UART2 的配置窗口。

(1) 在配置窗口上方的 Mode 栏中的 Mode(工作模式)参数设置中，选择 Asynchronous（异步模式），使 URAT2 工作在异步串行模式下。在 Hardware Flow Control（硬件流控）参数设置中选择 Disable，不使能硬件流控。

(2) 在配置窗口下方的 Configuration 栏，选择 Parameter Settings 标签页。

在 Basic Parameters（基本参数）部分进行如下的参数设置：

- Baud Rate（波特率）：115200
- Word Length（数据位长度）：8 Bits(including Parity)
- Parity（奇偶校验位）：None
- Stop Bits（停止位长度）：1

在 Advanced Parameters（高级参数）部分进行如下的参数设置：

- Data Direction（数据方向）：Receive and Transmit
- Over Sampling（过采样）：16 Samples

以上配置将 UART2 配置为异步串口模式，无硬件流控，115200 波特率，8 位数据位，1 位停止位，无奇偶校验，使能接收和发送模式，16 倍过采样率。具体的配置如图 9-10 所示。

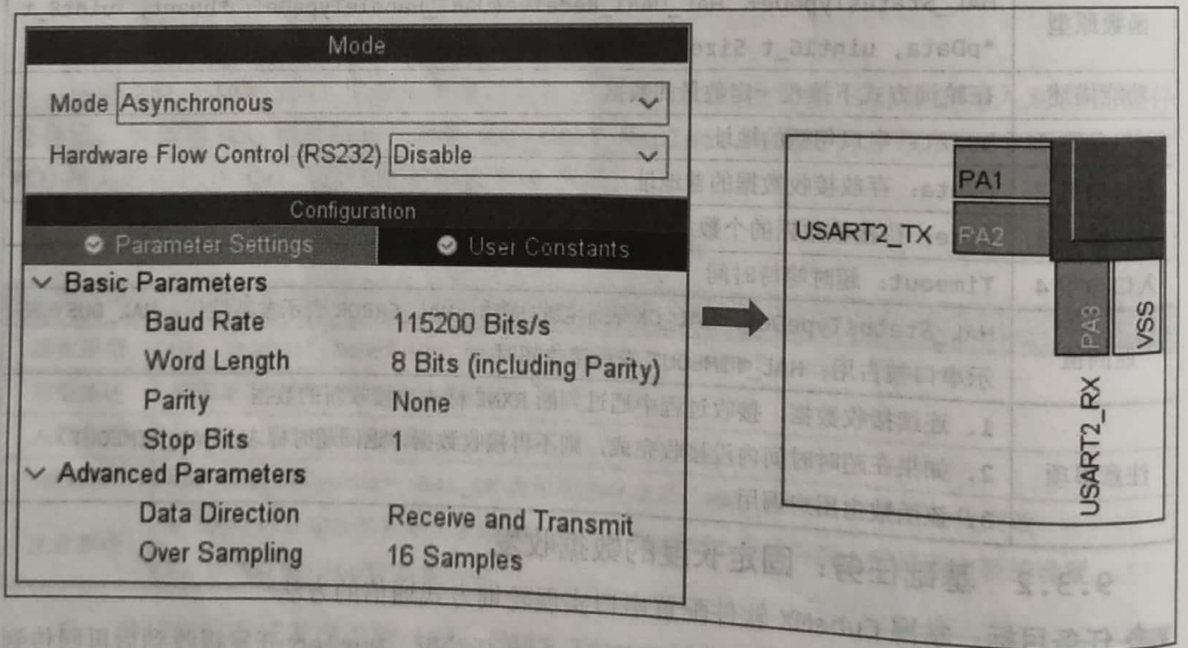


图 9-10 轮询方式的串口配置

完成串口外设配置后，在芯片引脚窗口中，PA2 引脚复用为 USART2_TX，作为串口 2 的数据发送引脚；PA3 引脚复用为 USART2_RX，作为串口 2 的数据接收引脚，如图 9-10 所示。

4. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。主程序部分将会自动定义一个串口 2 的句柄 huart2，并且自动生成串口 2 的初始化函数 MX_USART2_UART_Init，完成串口 2 的初始化操作。用户需要定义一个 uint8_t 类型的数组 RecBuf[5]存放接收的数据，并在 while 循环中编写数据的收发程序。主程序部分的程序如程序清单 9-5 所示。

程序清单 9-5　轮询方式下固定长度数据收发的主程序

```
1. #include "main.h"
2. /* USER CODE BEGIN PM */
3. uint8_t RecBuf[5];                              // 接收缓冲区定义
4. /* USER CODE END PM */
5. UART_HandleTypeDef huart2;                      // 串口 2 句柄定义
6. void SystemClock_Config(void);                  // 时钟系统初始化函数声明
7. static void MX_USART2_UART_Init(void);          // 串口 2 初始化函数声明
8. int main(void)
9. {
10.   HAL_Init();                                  // HAL 库初始化
11.   SystemClock_Config();                        // 时钟系统初始化
12.   MX_USART2_UART_Init();                       // 串口 2 初始化
13.   while (1)
14.   {
15.     /* USER CODE BEGIN 3 */
16.     // 接收 5 个字符完成
17.     if( HAL_UART_Receive(&huart2,RecBuf,5,100) == HAL_OK )
18.     {
19.        HAL_UART_Transmit(&huart2,RecBuf,5,100);  // 把接收的字符重新发送
20.     }
21.   }
22.   /* USER CODE END 3 */
23. }
```

➡ 程序解析

① 第 3 行是用户定义的数据接收缓冲区。

② 第 5 行是由 CubeMX 自动生成的定时器 2 的句柄 htim2。

③ 第 17 行是调用轮询模式下的串口接收函数 HAL_UART_Receive 接收数据，接收数据存放的首地址为 Recbuf（数组名代表数组的首地址），接收数据个数为 5，接收超时等待时间 100ms。程序使用了 if 语句判断接收数据是否成功接收，函数返回值为 HAL_OK，表明已经成功接收，进入下一步处理。否则，开始下一次循环接收。

④ 第 19 行是调用轮询模式下的串口发送函数 HAL_UART_Transmit 将接收到的数据原样发回。待发送数据存放的首地址为 Recbuf，发送数据个数为 5，发送超时等待时间 100ms。

➡ 运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。

然后打开串口调试助手，按照如下步骤进行参数的设置：

(1) 查找串口号。右键我的电脑 -> 属性 -> 设备管理器 -> 端口(COM 和 LPT)。

(2) 设置 PC 串口参数。端口：COM15；波特率：115200；数据位：8；校验位：无；停止位：1。如果串口处于关闭状态，还需要点击按钮"打开串口"来开启串口。

(3) 设置接收区的参数。如果用户需要显示 ASCII 码字符，则不勾选"16 进制"。如果用户需要显示 ASCII 码对应的 16 进制数，则勾选"16 进制"。勾选"自动清空"，以便接收数据超过一屏后自动清除。

(4) 设置收发参数。勾选"显示保存发送"，将显示发送的数据。勾选"帧换行"，使得每接收一次数据，就自动换行，便于用户观察接收的信息。如果发送的信息中自身包含了回车换行信息，则不要勾选。

(5) 设置发送区 1 属性。不勾选"16 进制"，则数据以 ASCII 码格式符发送。如果用户需要以 16 进制发送数据，则勾选"16 进制"。

(6) 设置发送内容。在发送区 1 中输入待发送的字符"UESTC"。然后点击"手动发送"按钮，将数据通过 USB 接口发送到 Nucleo 开发板。

(7) 收发信息显示。此时在串口调试助手的接收区将会显示发送的数据，以及接收的数据。

整个设置步骤如图 9-11 所示。

图 9-11 串口调试助手的设置

你的任务

修改程序，PC 一次发送 10 个字符到 Nucleco 开发板，开发板将这 10 个字符中的第 0/2/4/6/8 个字符按位取反之后发送回 PC。

9.3.3 进阶任务：实现 printf 函数重定向

在 PC 上进行 C 语言开发时，我们可以利用 C 语言的格式化输出函数 printf 将程序的运行结果送到显示屏上显示。在嵌入式系统的开发中，我们也希望能够将程序结果或者调试

信息发送到 PC 上显示。HAL 库提供的串口发送函数功能比较简单，不能进行格式化的输出。如果要使用类似 C 语言中的格式化输出函数 printf，我们需要把 printf 函数重新定向到串口中。这里的重定向是指用户可以自己重写 C 语言的库函数，当链接器检查到用户编写了与 C 库函数相同名字的函数时，就优先采用用户编写的函数，这样用户就可以实现对库函数的修改。为了实现重定向 printf 函数，我们需要重写 fputc 这个 C 标准库函数。

➡ 任务目标：实现 printf 函数重定向

➡ 任务内容：利用 printf 函数发送“Hello world!”字符串到 PC。

➡ 硬件原理：利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

➡ 任务实现

1. 设计思路

要实现串口重定向功能，只需要重新编写 fputc 函数，然后在 main.h 文件中包含 C 语言的标准输入/输出头文件 stdio.h 即可。

2. 引脚分配

本任务的引脚分配在外设配置中完成。

3. 外设配置

在外设设置窗口中选择：Connectivity → USART，将展开 UART2 的配置窗口。在配置窗口中按照图 9-12 所示参数进行配置。

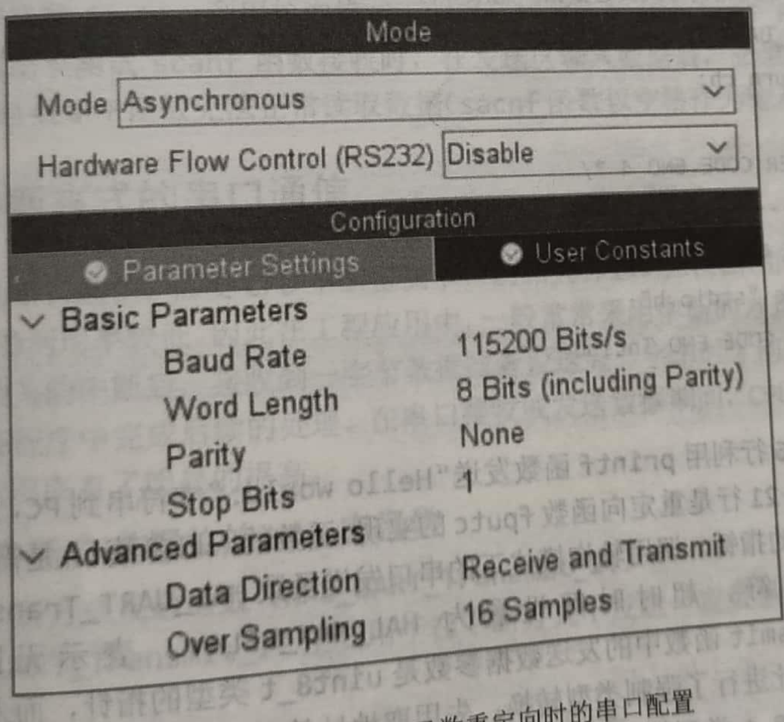


图 9-12 实现 printf 函数重定向时的串口配置

4. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。在 main.c 文件的/* USER CODE BEGIN 4 */和/* USER CODE END 4 */之间添加 fputc 函数的实现。然后在 while 循环中利用 printf 函数编写数据的收发程序。限于篇幅，这里只给出用户添加的代码，具体内容如程序清单 9-6 所示。

程序清单 9-6　实现 printf 函数重定向的关键代码

```
1. /*------------------------- main.c ------------------------------*/
2. while (1)
3. {
4.   /* USER CODE BEGIN 3 */
5.   printf ("Hello MCU World!\r\n");  // 通过串口输出字符串
6.   HAL_Delay(1000);                  // 延时 1s
7. }
8.   /* USER CODE END 3 */
9. /*---------------------------------------------------------------*/
10. /* USER CODE BEGIN 4 */
11. /**
12.  * @brief  Retargets the C library printf function to the USART.
13.  * @param  None
14.  * @retval None
15.  */
16. int fputc(int ch, FILE *f)
17. {
18.    // 采用轮询方式发送数据，无限等待
19.    HAL_UART_Transmit(&huart2, (uint8_t *)&ch, 1, HAL_MAX_DELAY);
20.    return ch;
21. }
22.  /* USER CODE END 4 */
23. /*------------------------- main.h ------------------------------*/
24. /* USER CODE BEGIN Includes */
25. #include "stdio.h"
26. /* USER CODE END Includes */
```

程序解析

① 第5～6行利用 printf 函数发送“Hello world.”字符串到 PC，发送间隔为 1s。

② 第16～21行是重定向函数 fputc 的实现，函数的入口参数 ch 是需要发送的字符，f 表示文件类型的指针。调用轮询模式下的串口发送函数 HAL_UART_Transmit 向串口发送 1 个无符号字符，超时时间设置为 HAL_MAX_DELAY，表示无限等待。由于 HAL_UART_Transmit 函数中的发送数据参数是 uint8_t 类型的指针，而入口参数 ch 是有符号字符型，因此进行了强制类型转换：先用取地址符号获得 ch 变量的地址，然后将地址强制转换为 uint8_t 类型的指针。

③ 第25行是在 main.h 文件中包含 C 语言的标准输入/输出头文件 stdio.h。如果不加入该头文件，编译器会报错，无法找到 printf 函数的定义。因为 printf 函数是在 stdio.h 文件中声明的。

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。

然后打开串口调试助手。由于 printf 函数中加入了回车换行符“\r\n”，因此可以在串口调试助手的第 4 步设置中，去掉对“帧换行”的勾选。具体的运行结果如图 9-13 所示。

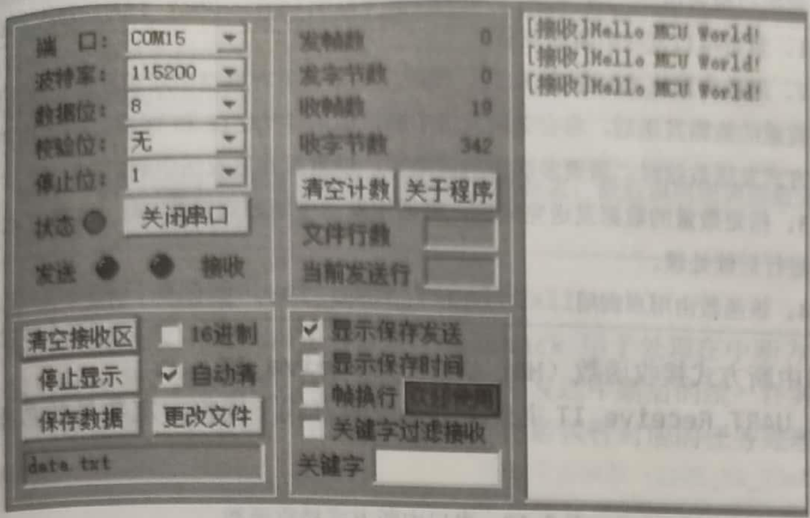


图 9-13 printf 函数输出的字符串

你的任务

实现 scanf 函数的重定向，并测试相关的数据接收。

提示：改写函数 fgetc，利用轮询模式下的串口接收函数 HAL_UART_Receive 实现。在利用串口调试助手测试 scanf 函数接收时，在发送区输入数据后，必须以空格结束，然后点击发送。否则 scanf 函数无法正常读取数据（sacnf 函数以空格作为输入数据的结束）。

9.4 中断方式的串口通信

轮询方式的串口通信，需要 CPU 不断检测串口的相关标志。在检测期间，CPU 无法执行其他任务，CPU 的利用率较低。因此在工程应用中，一般常常采用中断的方式进行串口通信。串口在使能了相关的中断后，接收到一字节数据或者发送完一字节时才向 CPU 申请中断，然后在中断服务程序中完成后续的处理。在串口接收或发送数据期间，CPU 还可以执行其他任务，CPU 的利用率有了明显的提高。

9.4.1 中断方式相关的 HAL 库函数

1. 串口中断方式发送函数（HAL_UART_Transmit_IT）

函数 HAL_UART_Transmit_IT 主要用于在中断方式下发送一定数量的数据。具体描述如表 9-12 所示。

表 9-12 串口中断方式发送函数

接口函数：HAL_UART_Transmit_IT	
函数原型	HAL_StatusTypeDef HAL_UART_Transmit_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
功能描述	在中断方式下发送一定数量的数据
入口参数 1	huart：串口句柄的地址
入口参数 2	pData：待发送数据的首地址

入口参数 3	Size：待发送数据的个数
返回值	HAL_StatusTypeDef：HAL_OK 表示发送成功；HAL_ERROR 表示参数错误；HAL_BUSY 表示串口被占用
注意事项	1. 负责串口发送中断的开启。 2. 函数将置位 TXEIE 和 TCIE，使能发送数据寄存器空中断和发送完成中断。完成指定数量的数据发送后，将会关闭发送中断，即清零 TXEIE 和 TCIE。因此用户多次采用中断方式发送数据时，需要多次调用该函数，以重新开启发送中断。 3. 指定数量的数据发送完成后，将调用发送中断回调函数 HAL_UART_TxCpltCallback 进行后续处理。 4. 该函数由用户调用。

2. 串口中断方式接收函数（HAL_UART_ Receive_IT）

函数 HAL_UART_Receive_IT 主要用于在中断方式下接收一定数量的数据。具体描述如表 9-13 所示。

表 9-13 串口中断方式接收函数

接口函数：HAL_UART_Receive_IT	
函数原型	HAL_StatusTypeDef HAL_UART_Receive_IT(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
功能描述	在中断方式下接收一定数量的数据
入口参数 1	huart：串口句柄的地址
入口参数 2	pData：存放接收数据的首地址
入口参数 3	Size：待接收数据的个数
返回值	HAL_StatusTypeDef：HAL_OK 表示接收成功；HAL_ERROR 表示参数错误；HAL_BUSY 表示串口被占用
注意事项	1. 负责串口接收中断的开启。 2. 函数将置位 RXEIE，使能接收数据寄存器非空中断 RXE。完成指定数量的数据接收后，将会关闭接收中断，即清零 RXEIE。因此用户多次采用中断方式接收数据时，需要多次调用该函数，以重新开启接收中断。 3. 指定数量的数据发送完成后，将调用接收中断回调函数 HAL_UART_RxCpltCallback 进行后续处理。 4. 该函数由用户调用。

3. 串口中断入口函数（HAL_UART_IRQHandler）

串口中断入口函数 HAL_UART_IRQHandler 是串口所有中断发生后的统一入口函数。任何一个串口的相关中断发生后，如发送数据寄存器空中断 TXE 或接收数据寄存器非空中断 RXE 等，都会通过中断向量表跳转到该函数。在该函数的内部会进行中断类型的判断，并清除对应的中断标志，然后再根据不同的中断类型，调用不同的回调函数完成中断处理任务。具体描述如表 9-14 所示。

表 9-14　串口中断入口函数

接口函数：HAL_UART_IRQHandler	
函数原型	void HAL_UART_IRQHandler(UART_HandleTypeDef *huart)
功能描述	串口所有中断发生后的统一入口函数
入口参数	*huart：串口句柄的地址
返回值	无
注意事项	1. 函数内部先判断中断类型，并清除对应的中断标志，最后调用回调函数完成中断处理 2. 该函数由 CubeMX 自动生成

4. 串口发送中断回调函数（HAL_UART_TxCpltCallback）

串口发送中断回调函数 HAL_UART_TxCpltCallback 用于处理在中断方式下发送完指定数量数据后的后续任务。该函数作为所有串口在发生发送中断后的统一任务处理函数，在函数内部需要先判断是哪一个串口产生的发送中断，然后执行对应的任务处理操作。具体描述如表 9-15 所示。

表 9-15　串口发送中断回调函数

接口函数：　HAL_UART_TxCpltCallback	
函数原型	void HAL_UART_TxCpltCallback(UART_HandleTypeDef *huart)
功能描述	回调函数，用于处理所有串口的发送中断，用户在该函数内编写实际的任务处理程序
入口参数	*huart：串口句柄的地址
返回值	无
注意事项	1. 函数由 HAL_UART_IRQHandler 函数调用，完成所有串口的发送中断任务处理 2. 函数内部需要根据串口句柄的实例来判断是哪一个串口产生的发送中断 3. 函数由用户根据具体的处理任务编写

5. 串口接收中断回调函数（HAL_UART_RxCpltCallback）

串口接收中断回调函数 HAL_UART_RxCpltCallback 用于处理在中断方式下接收完指定数量数据后的后续任务。该函数作为所有串口在发生接收中断后的统一任务处理函数，在函数内部需要先判断是哪一个串口产生的接收中断，然后执行对应的任务处理操作。具体描述如表 9-16 所示。

表 9-16　串口接收中断回调函数

接口函数：　HAL_UART_RxCpltCallback	
函数原型	void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
功能描述	回调函数，用于处理所有串口的接收中断，用户在该函数内编写实际的任务处理程序
入口参数	*huart：串口句柄的地址
返回值	无
注意事项	1. 函数由 HAL_UART_IRQHandler 函数调用，完成所有串口的接收中断任务处理 2. 函数内部需要根据串口句柄的实例来判断是哪一个串口产生的接收中断 3. 函数由用户根据具体的处理任务编写

6. 串口中断使能函数（__HAL_UART_ENABLE_IT）

串口中断函数__HAL_UART_ENABLE_IT 是用于使能对应的串口中断类型，采用带参数的宏实现（宏函数）。具体描述如表 9-17 所示。

表 9-17 串口中断使能函数

接口函数：__HAL_UART_ENABLE_IT	
函数原型	__HAL_UART_ENABLE_IT(__HANDLE__, __INTERRUPT__)
功能描述	使能对应的串口中断类型，如发送数据寄存器空中断、空闲中断等
宏参数 1	__HANDLE__：串口句柄
宏参数 2	__ INTERRUPT __：串口中断类型，该参数值几个常用的取值如下： UART_IT_TXE：发送数据寄存器空中断 UART_IT_TC：发送完成中断 UART_IT_RXNE：接收数据寄存器非空中断 UART_IT_IDLE：线路空闲中断
返回值	无
注意事项	1. 该函数是宏函数，进行宏替换，不发生函数调用 2. 函数需要由用户调用

7. 串口中断标志查询函数（__HAL_UART_GET_FLAG）

串口中断标志查询函数__HAL_UART_GET_FLAG 是用于查询对应的串口中断标志是否置位，采用带参数的宏实现（宏函数）。具体描述如表 9-18 所示。

表 9-18 串口中断标志查询函数

接口函数：__HAL_UART_GET_FLAG	
函数原型	__HAL_UART_GET_FLAG (__HANDLE__, __INTERRUPT__)
功能描述	查询对应的串口中断标志是否置位，如发送数据寄存器空中断标志、空闲中断标志等
宏参数 1	__HANDLE__：串口句柄
宏参数 2	__ INTERRUPT __：串口中断类型，该参数值几个常用的取值如下： UART_IT_TXE：发送数据寄存器空中断 UART_IT_TC：发送完成中断 UART_IT_RXNE：接收数据寄存器非空中断 UART_IT_IDLE：线路空闲中断
返回值	无
注意事项	1. 该函数是宏函数，进行宏替换，不发生函数调用 2. 函数需要由用户调用

8. 空闲中断标志清除函数（__HAL_UART_CLEAR_IDLEFLAG）

空闲中断标志查询函数__HAL_UART_CLEAR_IDLEFLAG 专门用于清除串口的空闲中断标志，采用带参数的宏实现（宏函数）。具体描述如表 9-19 所示。

表 9-19　空闲中断标志清除函数

接口函数：__HAL_UART_CLEAR_IDLEFLAG	
函数原型	__HAL_UART_CLEAR_IDLEFLAG (__HANDLE__)
功能描述	清除串口的空闲中断标志
宏参数 1	__HANDLE__：串口句柄
返回值	无
注意事项	1. 该函数是宏函数，进行宏替换，不发生函数调用 2. 函数需要由用户调用

9.4.2 基础任务：固定长度的数据收发

➡ 任务目标：掌握 CubeMX 软件配置串口实现中断方式通信的方法

➡ 任务内容：从 PC 上发送 10 个字符到 Nucleo 开发板，Nucleo 开发板收到后再回传到 PC。

➡ 硬件原理：利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

➡ 任务实现

1. 设计思路

定义一个数组 uint8_t RecBuf[10]，作为接收缓冲区，数组大小为 10 个字节。

2. 引脚分配

本任务的引脚分配在外设配置中完成。

3. 外设配置

外设配置部分的参数设置同轮询方式下的设置一致，如 9.3 节中的图 9-10 所示。唯一不同的是需要使能串口 2 的中断。

在外设设置窗口中选择：Connectivity → USART，将展开 UART2 的配置窗口。

(1) 在配置窗口下方的 Configuration 栏，选择 NVIC Settings 标签页，使能 UART2 全局中断，如图 9-14 所示。

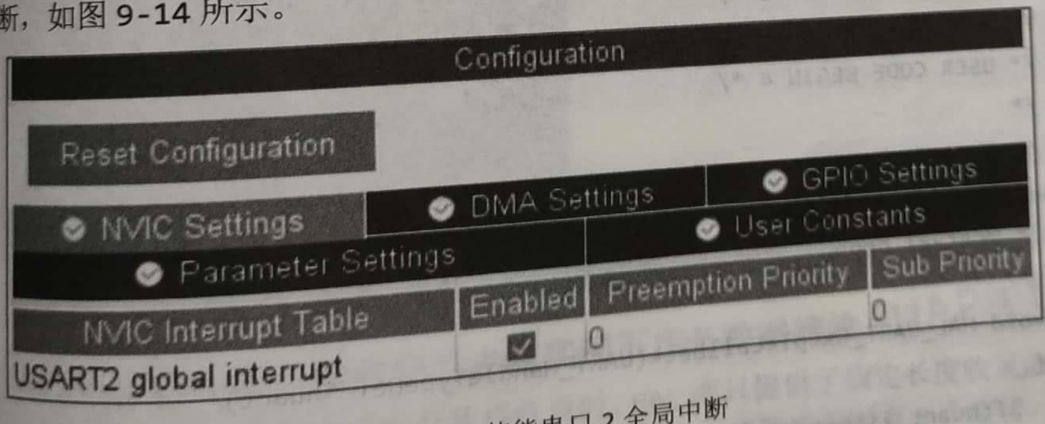


图 9-14　使能串口 2 全局中断

(2) 中断优先级不进行设置，使用默认的优先级即可。

4. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。用户需要定义一个 uint8_t 类型的数组 RecBuf[10]存放接收的数据，然后在串口初始化完成后，调用函数 HAL_UART_Receive_IT 使能接收中断。接收到 10 个数据后，在接收中断回调函数 HAL_UART_RxCpltCallback 中

置位接收完成标志 RxFlag。while 循环中不断检测 RxFlag 标志，发现为 1 后，调用串口中断方式发送函数 HAL_UART_Transmit_IT 将收到数据原样发送到 PC。限于篇幅，这里只给出用户添加的代码，具体内容如程序清单 9-7 所示。

程序清单 9-7　中断方式下固定长度数据收发的主程序

```
/* USER CODE BEGIN PV */
uint8_t RxBuffer[10];                                        // 接收缓冲区
volatile uint8_t RxFlag = 0;                                 // 数据接收完成标志
/* USER CODE END PV */
/*------------------------------------------------------------------------*/
/* USER CODE BEGIN 2 */
printf("******  UART commucition using IT  ******\r\n"); // 发送提示信息
printf("Please enter 10 characters:\r\n");
HAL_UART_Receive_IT(&huart2,(uint8_t*)RxBuffer,10);          // 使能接收中断
/* USER CODE END 2 */
/*------------------------------------------------------------------------*/
while (1)
{
  /* USER CODE BEGIN 3 */
  if( RxFlag == 1)
  {
      RxFlag = 0;                                            // 清除标志位
      printf("Recevie Success!\r\n");
      HAL_UART_Transmit_IT(&huart2,(uint8_t*)RxBuffer,10); // 数据回发
  }
}
  /* USER CODE END 3 */
/*------------------------------------------------------------------------*/
/* USER CODE BEGIN 4 */
/*
 * @brief  UART receive interrupt call back function
 * @param  None
 * @retval None
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if(huart ->Instance == USART2)
  {
    RxFlag = 1;                                              // 置位标志位
    HAL_UART_Receive_IT(&huart2,(uint8_t*)RxBuffer,10); // 重新使能接收中断
  }
}
```

```
38. /* USER CODE END 4 */
```

程序解析

① 第 2 行是用户定义的数据接收缓冲区。

② 第 3 行是用户定义的接收完成标志 RxFlag 并初始化为 0。标志位含义：为 1 表示接收完成，为 0 表示接收未完成。

③ 第 7~8 行利用 printf 函数发送提示信息，以提高程序的交互性。

④ 第 9 行在串口初始化完成后，调用函数 HAL_UART_Receive_IT 使能接收中断。

⑤ 第 15~20 行在 while 循环中不断检测 RxFlag 是否为 1，一旦 RxFlag 为 1，则清除标志位，发送接收成功的提示信息，并调用函数 HAL_UART_Transmit_IT 将收到数据原样发送到 PC。

⑥ 第 30~37 行是串口接收中断回调函数。函数内部先根据串口句柄的实例 Instance 判断是哪一个串口发生的接收中断，然后执行任务处理：置位 RxFlag，同时使能接收中断，准备下一轮的数据接收。

运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。然后打开串口调试助手，在发送区 1 输入需要发送的 10 个字符，如“1234567890”。具体的测试结果如图 9-15 所示。

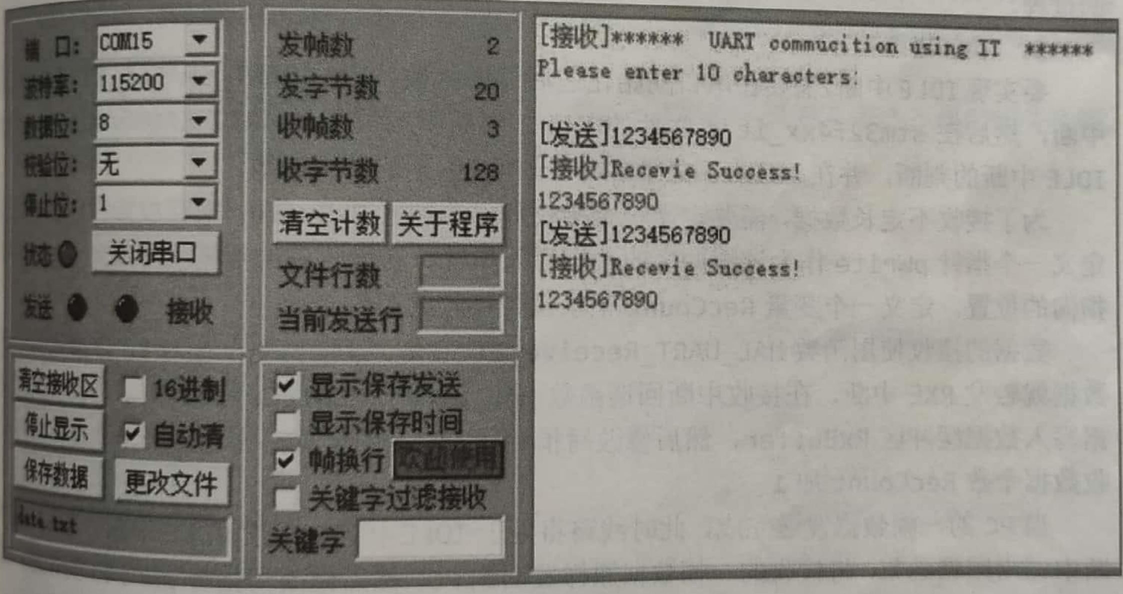


图 9-15　中断方式下固定长度数据的收发测试

9.4.3　进阶任务：利用空闲中断实现不定长数据接收

在采用轮询方式和中断方式进行数据收发时，HAL 库只提供了固定长度收发的接口函数。然而在工程应用中，往往会遇到不定长数据的接收，即一帧数据有时有 8 个字符，有时又只有 5 个字符，长度不固定，这种不定长的数据接收使用现有的 HAL 库接口函数就无法实现。

在 STM32 的串口中断中，提供了一个线路空闲中断 IDLE。当一帧数据传输结束之后，线路会维持高电平态，此时，就可以触发 IDLE 中断。

假设有一帧数据由 2 个字符构成，分别是 0xaa、0x55。在通信线路上传输的波形如图 9-16 所示。

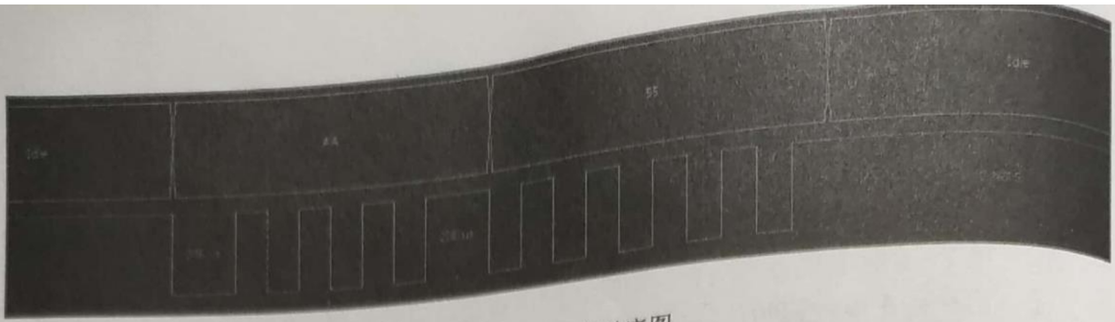


图 9-16　空闲状态时序图

在发送该帧数据之前和之后，数据线处于 IDLE 的状态。

从图 9-16 中我们可以看到，在该帧有效数据发送过程中，0xaa 字节与 0x55 字节之间，没有 IDLE 状态出现，因此不会出现 IDLE 误触发的情况，这样利用 IDLE 状态就可以判断不定长数据的发送结束。

任务目标：利用 IDLE 中断实现不定长数据的接收。

任务内容：PC 发送任意长度的数据到 Nucleo 开发板，开发板将数据原样回传到 PC。

硬件原理：利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

任务实现

本任务所使用的引脚和外设与基础任务中所使用的相同，因此省去引脚分配和外设配置的过程。

1. 设计思路

要实现 IDLE 中断，需要在串口初始化之后调用函数__HAL_UART_ENABLE_IT 使能 IDLE 中断，然后在 stm32f4xx_it.c 文件的串口 2 中断服务程序 USART2_IRQHandler 中添加 IDLE 中断的判断，并在 main.c 文件中编写 IDLE 中断的回调函数。

为了接收不定长数据，需要定义一个接收数据缓冲区 RxBuffer 来存放接收的数据，并定义一个指针 pWrite 作为该缓冲区的数据写入指针，新接收到的数据就写入到 pWrite 所指向的位置。定义一个变量 RecCount 记录已经接收的数据个数。

数据的接收使用函数 HAL_UART_Receive_IT 完成，接收长度设置为 1，每接收到一个数据就触发 RXE 中断，在接收中断回调函数 HAL_UART_RxCpltCallback 中将接收到的数据写入数据缓冲区 RxBuffer，然后修改写指针 pWrite 的位置，指向下一个地址，同时接收数据个数 RecCount 加 1。

当 PC 的一帧数据发送完成，此时线路将处于 IDLE 状态，触发 IDLE 中断，在 IDLE 中断中调用回调函数，将接收的一帧数据原样发回到 PC，然后修改写指针 pWrite 的位置，指向接收数据缓冲区的首地址，并清零接收数据个数 RecCount，准备下一轮的数据接收。

2. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。用户需要修改 2 个文件：stm32f4xx_it.c 文件中添加 IDLE 中断处理程序；main.c 文件中完成全部的变量定义和 IDLE 中断回调函数。限于篇幅，这里只给出用户添加的代码。

stm32f4xx_it.c 文件中添加的内容如程序清单 9-8 所示。

程序清单 9-8　添加 IDLE 中断服务程序

```
/*
  * @brief This function handles USART2 global interrupt.
```

```
  */
void USART2_IRQHandler(void)
{
  /* USER CODE BEGIN USART2_IRQn 0 */
  /* USER CODE END USART2_IRQn 0 */
  HAL_UART_IRQHandler(&huart2);
  /* USER CODE BEGIN USART2_IRQn 1 */
  // 添加 IDLE 中断处理
  if(__HAL_UART_GET_FLAG(&huart2, UART_FLAG_IDLE) != RESET)
  {
     __HAL_UART_CLEAR_IDLEFLAG(&huart2);     // 清除 Idle 中断标志
     HAL_UART_IdleCpltCallback(&huart2);     // 用户编写的 IDLE 中断回调函数
  }
  /* USER CODE END USART2_IRQn 1 */
}
```

程序解析

在串口 2 中断服务程序 USART2_IRQHandler 中添加的代码比较简单，主要是调用宏函数__HAL_UART_GET_FLAG 判断是否产生了 IDLE 中断。如果产生了 IDLE 中断则调用宏函数__HAL_UART_CLEAR_IDLEFLAG 清除 IDLE 中断，然后调用用户编写的 IDLE 中断回调函数完成后续的处理。

注意:如果用户使用的其他串口，则需要在该串口对应的中断服务程序中添加相关代码。

在 main.c 文件中添加的内容如程序清单 9-9 所示。

程序清单 9-9 实现不定长数据接收程序

```
/* USER CODE BEGIN PM */
#define   LENGTH   100                          // 接收缓冲区大小
/* USER CODE END PM */
/* USER CODE BEGIN PV */
uint8_t RxBuffer[LENGTH];                       // 接收缓冲区
uint8_t *pWrite;                                // 接收缓冲区写入数据位置
uint8_t RecCount;                               // 接收数据个数
/* USER CODE END PV */
/*-------------------------------------------------------------*/
/* USER CODE BEGIN 2 */
// 发送提示信息
printf("******  UART commucition using IDLE IT  ******\r\n");
printf("Please enter arbitrary length characters:\r\n");
pWrite = RxBuffer;                              // 写指针指向接收缓冲区首地址
RecCount = 0;                                   // 接收数据个数初始化为 0
__HAL_UART_ENABLE_IT(&huart2, UART_IT_IDLE);    // 使能 IDLE 中断
HAL_UART_Receive_IT(&huart2,pWrite,1);          // 使能接收中断，准备接收第一个数据
/* USER CODE END 2 */
```

```
--------------------------------------------------------------------*/
19. /*--------------------------------------------------------------
20. /* USER CODE BEGIN 4 */
21. /*
22.  * @brief  UART receive interrupt call back function
23.  * @param  huart
24.  * @retval None
25.  */
26. void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
27. {
28.     if(huart ->Instance == USART2)
29.     {
30.       // 修改写指针，具体的写入过程由 HAL_UART_Receive_IT 函数完成
31.       pWrite++;
32.       RecCount++;                          //接收数据个数加 1
33.       HAL_UART_Receive_IT(&huart2,pWrite,1);//重新使能接收中断，接收下一个数据
34.     }
35. }
36. /**
37.   * @brief  Rx Transfer IDLE callbacks.
38.   * @param  huart
39.   * @retval None
40.   */
41. void HAL_UART_IdleCpltCallback(UART_HandleTypeDef *huart)
42. {
43.    HAL_UART_Transmit(huart, RxBuffer, RecCount, 100);          // 原样回发
44.    //接收完一帧数据,指针指向接收缓冲区首地址,接收数据个数清零，准备下一次接收
45.    pWrite   = RxBuffer;
46.    RecCount = 0;
47. }
48. /* USER CODE END 4 */
```

程序解析

① 第 2 行是用户定义的宏常量 LENGTH，作为数据接收缓冲区的大小。采用宏定义的目的是方便用户修改缓冲区的大小。

② 第 5 行是用户定义的数据接收缓冲区，大小由宏常量 LENGTH 决定。

③ 第 6 行是用户定义的指针 pWrite，表明接收数据写入缓冲区的位置。

④ 第 7 行是用户定义的变量 RecCount，记录接收过程中接收数据的个数

⑤ 第 12～13 行利用 printf 函数发送提示信息，以提高程序的交互性。

⑥ 第 14～15 行完成变量的初始化：写指针 pWrite 指向数据接收缓冲区的首地址，接收数据个数 RecCount 初始化为 0。

⑦ 第 16 行使能串口 IDLE 中断。

⑧ 第 17 行在串口初始化完成后，调用函数 HAL_UART_Receive_IT 使能接收中断，准备开始接收数据。**注意：**存放接收数据的地址为写指针 pWrite，每次接收的数据个数为

⑨ 第26～35行是串口接收中断回调函数。函数内部先根据串口句柄的实例Insatnce判断是哪一个串口发生的接收中断，然后执行任务处理：修改写指针的位置和接收数据的个数，同时使能接收中断，准备下一个数据的接收。

⑩ 第41～47行是串口空闲中断回调函数。由于该回调函数在串口2的中断服务程序USART2_IRQHandler中调用，只用于UART2，因此在回调函数内部不用判断是哪一个串口发生的IDLE中断。发生IDLE中断时，说明一帧数据已经发送完成，此时执行处理任务：调用串口中断方式发送函数HAL_UART_Transmit_IT将该帧数据原样发回到PC。然后写指针pWrite重新指向接收缓冲区的首地址，接收数据个数清零，准备下一帧数据的接收。

由于在stm32f4xx_it.c文件中调用了main.c文件中的串口空闲中断回调函数HAL_UART_IdleCpltCallback，因此需要将该函数在main.h文件中作一个函数声明，以便提供给外部文件使用。stm32f4xx_it.c文件中通过包含main.h文件来间接包含空闲中断回调函数的声明。

在main.h文件中添加的内容如程序清单9-10所示。

程序清单9-10 提供给外部文件使用的函数声明

```
/* USER CODE BEGIN EFP */
void HAL_UART_IdleCpltCallback(UART_HandleTypeDef *huart);    // 函数声明
/* USER CODE END EFP */
```

➡ 程序解析

这里的代码主要是添加了提供给外部文件使用的函数声明，注意函数声明添加的位置必须在/* USER CODE BEGIN EFP */和/* USER CODE BEGIN EFP */之间。

➡ 运行结果

将程序编译链接，生成可执行代码，通过ST-Link仿真器下载到Nucleo开发板上。然后打开串口调试助手进行如下设置：接收区设置为16进制显示，勾选帧换行和显示保存发送，发送区1设置为16进制格式。用户测试时可动态修改发送数据的数量，观察数据的回传情况。具体的测试结果如图9-17所示。

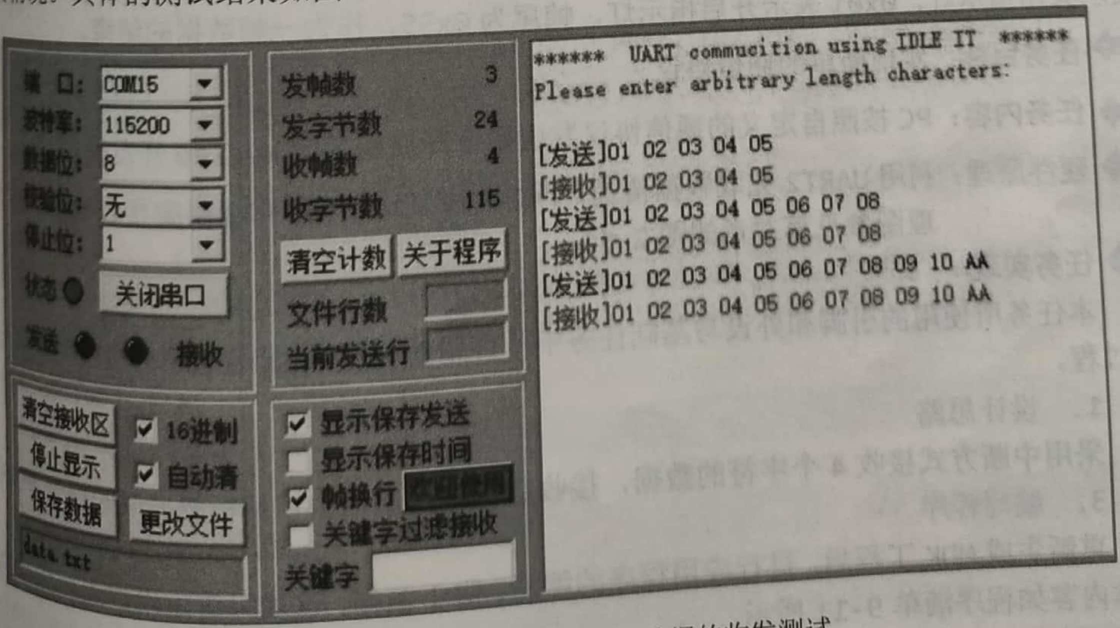


图9-17 IDLE中断实现不定长数据的收发测试

9.4.4 挑战任务：通信协议的实现

在前面的串口信息传递中，我们仅仅是完成了数据的发送和回传。在工程应用中，PC和控制器之间一般是通过标准的通信协议进行信息的传递。那么，什么是通用协议？它又有什么作用呢？

打一个比方，人与人的交流是通过某种语言，通信协议也就好像是机器与机器之间沟通的语言。通信协议通常由多个字符构成，以帧为单位。一帧数据中，不同字段的字符代表不同的含义。发送方按照规定的协议发送一帧数据，接收发接收下这一帧数据后，再按照通信协议进行解析，然后完成对应的操作。

以工控领域最常用的通信协议 Modbus 为例，其一帧的数据构成如表 9-20 所示。

表 9-20 Modbus 通信协议格式

起始符	设备地址	功能代码	数据	校验	结束符
1 个字符	2 个字符	1 个字符	n 个字符	2 个字符	1 个字符

- 起始符----------表示一帧数据的开始
- 设备地址--------指定需要进行信息传递的设备
- 功能代码--------指定需要完成的操作
- 数据------------传输的数据
- 校验------------用于通信中的错误校验
- 结束符----------表示一帧数据的结束

我们仿照 Modbus 的通信协议格式，设计一个简单的通信协议，如表 9-21 所示。

表 9-21 自定义通信协议格式

帧头	设备代码	功能代码	帧尾
0xaa	1 个字符	1 个字符	0x55

在这个通信协议里面，一共有 4 个字符，采用 16 进制定义。帧头为 0xaa，作为一帧数据的开始。接下来是设备代码，我们用 0x01 表示指示灯。在功能代码部分，我们用 0x00 表示关闭指示灯，0x01 表示开启指示灯。帧尾为 0x55，作为一帧数据的结束。

➡ **任务目标**：实现简单的通信协议

➡ **任务内容**：PC 按照自定义的通信协议发送指令开启或关闭 Nucleo 开发板的 LD2。

➡ **硬件原理**：利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

➡ **任务实现**

本任务所使用的引脚和外设与基础任务中所使用的相同，因此省去引脚分配和外设配置的过程。

1. 设计思路

采用中断方式接收 4 个字符的数据，接收后再根据通信协议解析要完成的指令并执行。

3. 编写程序

重新生成 MDK 工程后，进行应用程序的编写。限于篇幅，这里只给出用户添加的代码，具体内容如程序清单 9-11 所示。

程序清单 9-11　简单通信协议的实现

```c
/* USER CODE BEGIN PV */
uint8_t RxBuffer[4];                                   // 接收缓冲区
volatile uint8_t RxFlag = 0;                           // 数据接收标志
uint8_t ErrFlag = 0;                                   // 接收错误标志
/* USER CODE END PV */
/*-------------------------------------------------------------------------*/
/* USER CODE BEGIN 2 */
// 发送提示信息
printf("******  Commucition Protocol  ******\r\n");
printf("Please enter instruction:\r\n");
printf("Head->0xaa  Device->0x01  Operation->0x0/0x01  Head->0x55.\r\n");
HAL_UART_Receive_IT(&huart2,(uint8_t*)RxBuffer,4);     // 使能接收中断
/* USER CODE END 2 */
/*-------------------------------------------------------------------------*/
while (1)
{
  /* USER CODE BEGIN 3 */
  if( RxFlag == 1)
  {
     RxFlag = 0;                                       // 清除标志位
     // 通信协议解析
     if( RxBuffer[0]==0xaa && RxBuffer[3]==0x55)       // 判断帧头帧尾
     {
        if( RxBuffer[1]==0x01 )                        // 判断设备码
        {
           if(RxBuffer[2]==0x00 )                      // 判断功能码
           {
             HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_RESET);
             printf("LD2 is close!\r\n");
           }
           else if(RxBuffer[2]==0x01 )                 // 判断功能码
           {
             HAL_GPIO_WritePin(LD2_GPIO_Port, LD2_Pin, GPIO_PIN_SET);
             printf("LD2 is open!\r\n");
           }
           else
           {                                           // 接收错误
             ErrFlag = 1;
           }
        }
        else
```

```
42.         {
43.           ErrFlag = 1;                                        // 接收错误
44.         }
45.       }
46.       else
47.       {
48.         ErrFlag = 1;                                          // 接收错误
49.       }
50.       if( ErrFlag==1 )
51.       {
52.         printf("Communication Protocol Error! Please send again!\r\n");
53.       }
54.       // 清除接收缓冲区和出错标志，准备下一次接收
55.       ErrFlag     = 0;
56.       RxBuffer[0] = 0;
57.       RxBuffer[1] = 0;
58.       RxBuffer[2] = 0;
59.       RxBuffer[3] = 0;
60.   }
61. }
62.   /* USER CODE END 3 */
63. /*-----------------------------------------------------------------------*/
64. /* USER CODE BEGIN 4 */
65. /*
66.   * @brief  UART receive interrupt call back function
67.   * @param  None
68.   * @retval None
69.   */
70. void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
71. {
72.   if(huart ->Instance == USART2)
73.   {
74.       RxFlag = 1;                                           // 置位标志位
75.       HAL_UART_Receive_IT(&huart2,(uint8_t*)RxBuffer,4);    // 重新使能接收中断
76.   }
77. }
78. /* USER CODE END 4 */
```

程序解析

① 第 2 行是用户定义的一个数据接收缓冲区。

② 第 3 行是用户定义的接收完成标志 RxFlag 并初始化为 0。标志位含义：为 1 表示接收完成，为 0 表示接收未完成。

③ 第4行是用户定义的接收错误标志 ErrFlag 并初始化为0。标志位含义：为1表示接收错误，为0表示接收正确。

④ 第9~11行利用 printf 函数发送提示信息，以提高程序的交互性。

⑤ 第12行在串口初始化完成后，调用函数 HAL_UART_Receive_IT 使能接收中断。

⑥ 第18~20行在 while 循环中不断检测 RxFlag 是否为1，一旦 RxFlag 为1，则清除标志位，开始通信帧的解析。

⑦ 第22~49行是整个通行帧的解析过程。首先判断帧头和帧尾是否是 0xaa 和 0x55；然后判断设备码是否是 0x01，接着再判断功能码是否是 0x00 或 0x01。以上判断有任何一个不满足，则置位接收错误标志 ErrFlag。条件都满足，则根据功能码的内容执行相应的操作：开启或者关闭指示灯 LD2。

⑧ 第50~53行是判断接收错误标志 ErrFlag，给出错误提示信息。

⑨ 第55~59行是清除接收错误标志 ErrFlag 和接收缓冲区 RxBuffer，准备下一次接收。

⑩ 第70~77行是串口接收中断回调函数。函数内部先根据串口句柄的实例 Instance 判断是哪一个串口发生的接收中断，然后执行任务处理：置位 RxFlag，同时使能接收中断，准备下一轮的数据接收。

➡ 运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。然后打开串口调试助手，接收区设置为 ASCII 码显示（不勾选16进制），取消帧换行和显示保存发送。发送区 1 设置为 16 进制格式。当发送正确的通信协议时可以开启或关闭 Nucleo 开发板上指示灯 LD2，并回传提示信息。如果发送错误的通信协议则会给出错误提示。具体的测试结果如图 9-18 所示。

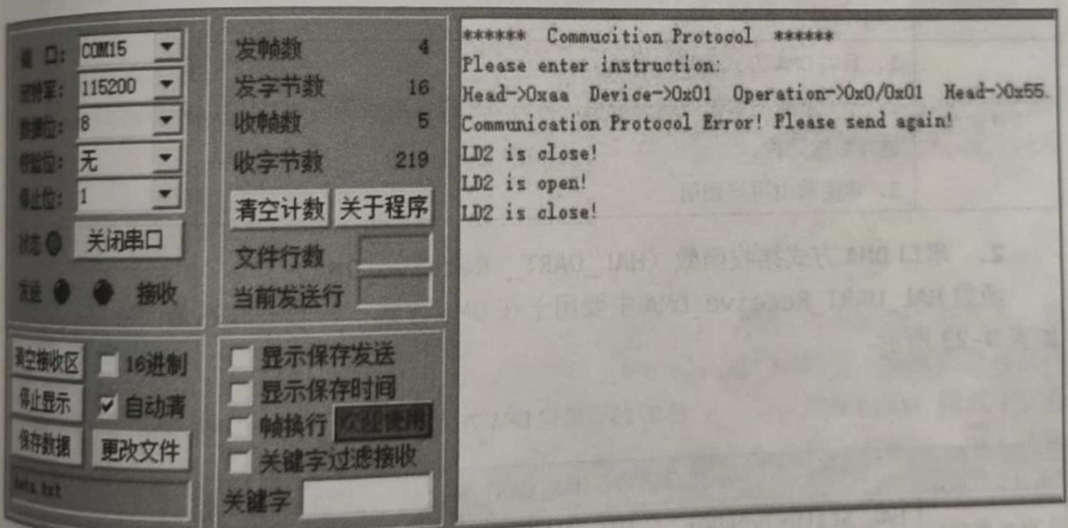


图 9-18 简单通信协议的测试结果

9.5 DMA 方式的串口通信

DMA 是一种不经过 CPU 干预而直接从存储器存取数据的数据交换模式。在 DMA 模式下，CPU 只需要向 DMA 控制器下达指令，数据传递由 DMA 自动完成，这样能够较少对 CPU 的占用。DMA 已成为现代高性能微控制器的标准配置。

在串口通信的三种方式中，DMA 方式是传输效率最高的一种。使用 DMA 方式进行串口通信时，只需要在启动 DMA 传输前，设置 DMA 通道、存储器的首地址、传输数据的个数等参数。传输过程中不需要 CPU 的干预，DMA 控制器会自动将串口接收的数据写入存储器或者将存储器的数据通过串口发送。接收或发送到指定的数据个数后，再产生 DMA 中断，由 CPU 进行后续处理。

DMA 方式和中断方式最大的不同在于：中断方式下，每接收或发送一个数据，都会产生中断，要求 CPU 进行处理；而 DMA 方式下，当接收或发送指定数量的数据后，才产生 DMA 中断，要求 CPU 进行处理，收发过程由 DMA 控制器完成，无需 CPU 干预。

9.5.1 DMA 方式相关的 HAL 库函数

1. 串口 DMA 方式发送函数（HAL_UART_Transmit_DMA）

函数 HAL_UART_Transmit_DMA 主要用于在 DMA 方式下发送一定数量的数据。具体描述如表 9-22 所示。

表 9-22　串口 DMA 方式发送函数

接口函数：HAL_UART_Transmit_DMA	
函数原型	HAL_StatusTypeDef HAL_UART_Transmit_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
功能描述	在 DMA 方式下发送一定数量的数据
入口参数 1	huart：串口句柄的地址
入口参数 2	pData：待发送数据的首地址
入口参数 3	Size：待发送数据的个数
返回值	HAL_StatusTypeDef：HAL_OK 表示发送成功；HAL_ERROR 表示参数错误；HAL_BUSY 表示串口被占用
注意事项	1. 启动 DMA 方式的串口发送。 2. 指定数量的数据发送完成后，将调用发送中断回调函数 HAL_UART_TxCpltCallback 进行后续处理。 3. 该函数由用户调用。

2. 串口 DMA 方式接收函数（HAL_UART_ Receive_DMA）

函数 HAL_UART_Receive_DMA 主要用于在 DMA 方式下接收一定数量的数据。具体描述如表 9-23 所示。

表 9-23　串口 DMA 方式接收函数

接口函数：HAL_UART_Receive_DMA	
函数原型	HAL_StatusTypeDef HAL_UART_Receive_DMA(UART_HandleTypeDef *huart, uint8_t *pData, uint16_t Size)
功能描述	在 DMA 方式下接收一定数量的数据
入口参数 1	huart：串口句柄的地址
入口参数 2	pData：存放接收数据的首地址
入口参数 3	Size：待接收数据的个数

返回值	HAL_StatusTypeDef：HAL_OK 表示接收成功；HAL_ERROR 表示参数错误；HAL_BUSY 表示串口被占用
注意事项	1. 启动 DMA 方式的串口接收。 2. 指定数量的数据接收完成后，将调用接收中断回调函数 HAL_UART_RxCpltCallback 进行后续处理。 3. 该函数由用户调用。

9.5.2 基础任务：固定长度的数据收发

UART6交件表.

任务目标： 掌握 CubeMX 软件配置串口实现 DMA 方式通信的方法

任务内容： 从 PC 上发送 10 个字符到 Nucleo 开发板，Nucleo 开发板收到后再回传到 PC。

硬件原理： 利用 UART2 完成数据传递，PA2 引脚发送数据，PA3 引脚接收数据。电路原理图参见第二章的图 2-11。

任务实现

1. 设计思路

定义一个数组 uint8_t RecBuf[10]，作为接收缓冲区，数组大小为 10 个字节。

2. 引脚分配

本任务的引脚分配在外设配置中完成。

3. 外设配置

在外设设置窗口中选择：Connectivity → USART，将展开 UART2 的配置窗口。

(1) Mode 栏以及 Configuration 栏的 Parameter Settings 标签页的参数设置与轮询方式下的设置一致，如 9.3 节中的图 9-10 所示。

(2) 在配置窗口下方的 Configuration 栏，选择 DMA Settings 标签页，按照如下步骤进行 DMA 通道的设置：

1) 点击 ADD 按钮；

2) 在下拉菜单中分别选择 USART2_RX 和 USART_TX，添加串口接收和发送的 DMA 通道。DMA 控制器一共有 7 个通道，具体使用哪一个由 CubeMX 自动分配。数据的传输方向和 DMA 通道的优先级也使用默认的配置。

3) DMA 通道参数设置，包括以下参数：

- Mode

用于设置 DMA 工作模式，分为 Normal 和 Circular 两种模式。在 Normal 模式下，当 DMA 传输结束时，即指定数量的数据已经完成传输，将不再传输新的数据。如果需要再次传输数据，需要重新启动 DMA 传输。这种方式相当于单次传输方式。在 Circular 模式下，当 DMA 传输结束时，传输数据个数将自动重载为初值，数据缓冲区重新指向首地址，DMA 传输将继续运行。这种方式相当于连续传输方式。

Circular 模式主要用于连续地、大批量的数据传递，如 A/D 转换中的扫描模式。这里的任务要求是由 PC 发送数据到 Nulceo 开发板，Nulceo 开发板是作为数据的消费者，而不是数据的生产者，因此选择 Normal 模式。

- Increment Address

用于设置地址递增。外设作为单一设备，地址固定，因此地址设置为不递增。而存储器包括多个存储单元，完成一个数据传输后，地址应该自动递增，指向下一个待传输数据的地址，因此设置为自动增加。

● Data Width

用于设置数据宽度，有 Byte（字节）、Half Word（2 字节）和 Word（4 字节）三种选择。由于串口一次只能收发一个字节，因此选择 Byte。

● Use Fifo

用于设置是否使能 Fifo。Fifo 作为看做 DMA 传输过程中的缓冲区，用于数据的缓存。一般选择不使能。

DMA 的操作步骤及配置情况如图 9-19 所示。

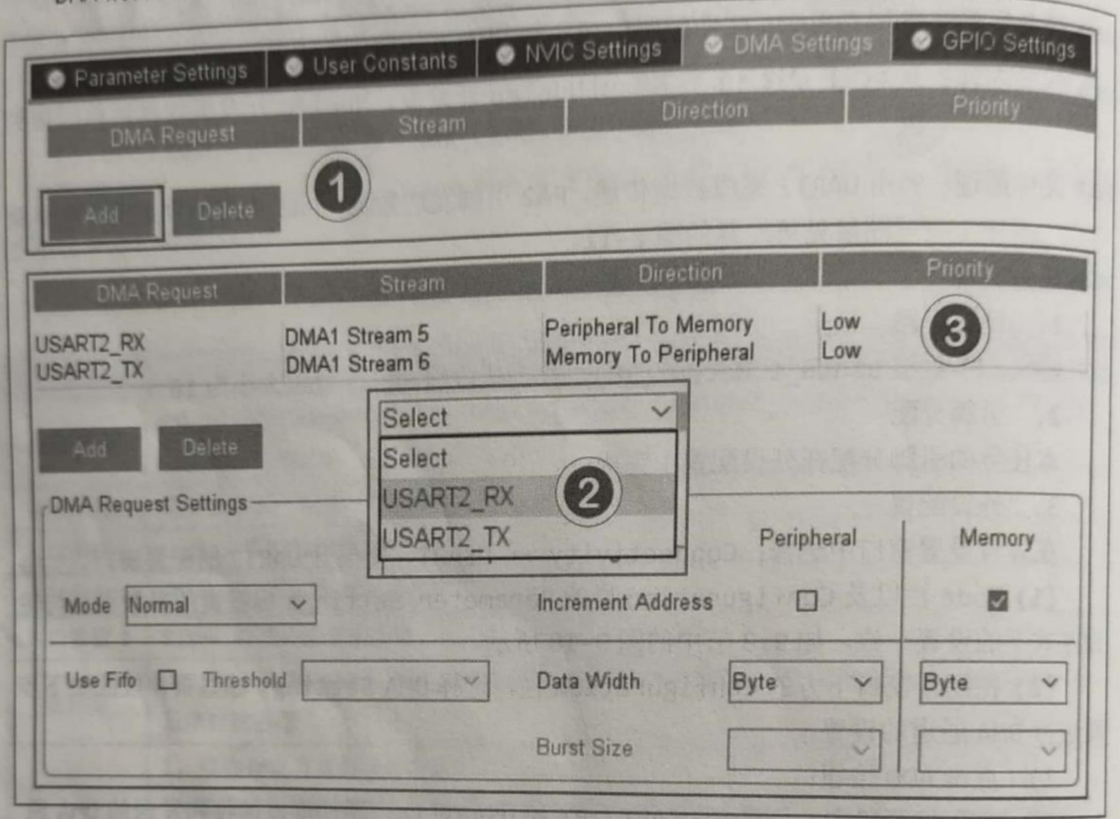


图 9-19　串口 DMA 方式的参数设置

(3) 在配置窗口下方的 Configuration 栏，选择 NVIC Settings 标签页，使能 UART2 全局中断，DMA 通道的中断使能由 CubeMX 自动勾选，如图 9-20 所示。

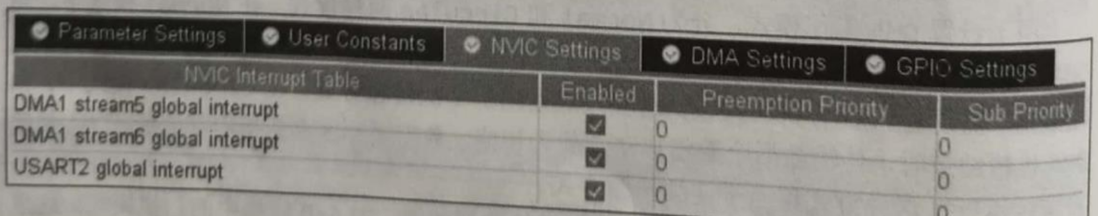


图 9-20　使能串口 2 全局中断和 DMA 中断

注意：DMA 方式的串口通信，也必须使能串口中断，这样才能进入对应的发送中断回调函数 HAL_UART_TxCpltCallback 和接收中断回调函数 HAL_UART_RxCpltCallback 来判断 DMA 的传输结束。如果不使能串口中断，只能发送第一次的数据。

(4) 中断优先级不进行设置，使用默认的优先级即可。

4. 编写程序

重新生成MDK工程后，进行应用程序的编写。CubeMX将自动生成用于DMA传输的句柄hdma_usart2_rx和hdma_usart2_tx，以及函数MX_DMA_Init来完成DMA通道的初始化。

程序的设计思路和中断方式下的类似，只需要将中断方式下的数据接收函数HAL_UART_Receive_IT改为DMA方式下的数据接收函数HAL_UART_Receive_DMA；将中断方式下的数据发送函数HAL_UART_Transmit_IT改为DMA方式下的数据发送函数HAL_UART_Transmit_DMA即可。虽然两种方式的程序设计思路类似，但传输效率完全不同。限于篇幅，这里只给出用户添加的代码，具体内容如程序清单9-12所示。

程序清单9-12　DMA方式下固定长度数据收发的主程序

```
/* USER CODE BEGIN PV */
uint8_t RxBuffer[10];                                  // 接收缓冲区
volatile uint8_t RxFlag = 0;                           // 数据接收完成标志
/* USER CODE END PV */
/*------------------------------------------------------------------------*/
/* USER CODE BEGIN 2 */
printf("******  UART commucition using DMA  ******\r\n"); // 发送提示信息
printf("Please enter 10 characters:\r\n");
HAL_UART_Receive_DMA(&huart2,(uint8_t*)RxBuffer,10);     // 使能DMA接收
/* USER CODE END 2 */
/*------------------------------------------------------------------------*/
while (1)
{
  /* USER CODE BEGIN 3 */
  if( RxFlag == 1)
  {
        RxFlag = 0;                                    // 清除标志位
        printf("DMA Recevie Success!\r\n");
        HAL_UART_Transmit_DMA(&huart2,(uint8_t*)RxBuffer,10); // 数据回发
  }
}
  /* USER CODE END 3 */
/*------------------------------------------------------------------------*/
/* USER CODE BEGIN 4 */
/*
 * @brief  UART receive interrupt call back function
 * @param  None
 * @retval None
 */
void HAL_UART_RxCpltCallback(UART_HandleTypeDef *huart)
{
  if(huart ->Instance == USART2)
```

```
  {                                                        // 置位标志位
      RxFlag = 1;
      HAL_UART_Receive_DMA(&huart2,(uint8_t*)RxBuffer,10); // 重新使能 DMA 接收
  }
}
/* USER CODE END 4 */
```

➡ 程序解析

程序的设计思路和中断方式下的串口收发程序几乎一致，只是将中断方式下的数据收发改为了 DMA 方式下的数据收发。**注意：**当 DMA 接收完成后，使用的回调函数依然是接收中断回调函数 HAL_UART_RxCpltCallback。

➡ 运行结果

将程序编译链接，生成可执行代码，通过 ST-Link 仿真器下载到 Nucleo 开发板上。然后打开串口调试助手，在发送区 1 中输入需要发送的 10 个字符，如“Hello MCU!”。具体的测试结果如图 9-21 所示。

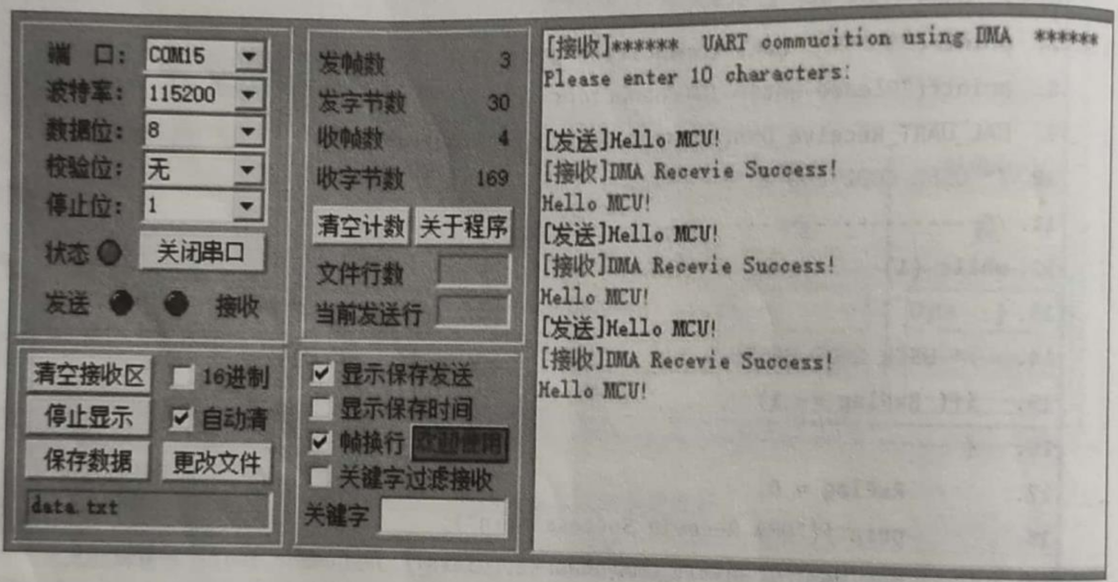


图 9-21　DMA 方式下固定长度数据的收发测试

你的任务

如何将 DMA 方式和 IDLE 中断结合起来，实现不定长数据的收发。

9.6　本章小结

本章重点介绍了 STM32 的串口模块。以串口的功能框图为引入，介绍了串口模块的工作原理，以及 HAL 库中轮询、中断和 DMA 三种通信方式下的数据类型和接口函数定义。

本章以串口为例，介绍了 HAL 库中外设模块的初始化设计思想：从抽象到承载。抽象部分是外设的通用属性，如波特率和数据位数等。而承载部分则是与具体 MCU 相关的配置，如硬件引脚和中断配置等。这些设计思想需要读者在学习时深入体会。

在实际的工程应用中，由于数据的发送方为主机，数据的接收方为从机。因此，常用的

通信方式是：发送时采用轮询方式，接收时采用中断方式。当需要大批量数据传递时，才使用 DMA 方式。

在实际的通信格式设计中，可以采用产业界标准的通信协议，如 Modbus，也可以由用户自定义的通信协议来实现。

习题 9

1. 在串行通信中，假设传送的波特率为 4800，每个数据占 10 位，问传输 2KB 的数据需要多少时间？
2. 实现 scanf 函数的重定向。
3. 编写通信协议，协议格式如表 9-21 所示。增加两个功能码：0x03 表示指示灯每隔 1s 闪烁一次；0x04 表示指示灯以呼吸灯形式变化，周期为 20ms，步进值为 20%。
4. 编写 DMA 方式和 IDLE 中断相结合实现不定长数据收发的程序。
5. 编写使用 Modbus 通信协议控制指示灯的程序。