第 5 章

STM32 GPIO

本章讲述 STM32 GPIO 接口，包括 GPIO 接口概述、GPIO 功能、GPIO 的 HAL 驱动程序、GPIO 使用流程、采用 STM32CubeMX 和 HAL 库的 GPIO 输出应用实例、采用 STM32CubeMX 和 HAL 库的 GPIO 输入应用实例。

5.1 STM32 GPIO 接口概述

通用输入输出(GPIO)接口的功能是让嵌入式处理器能够通过软件灵活地读出或控制单个物理引脚上的高、低电平，实现内核和外部系统之间的信息交换。GPIO 是嵌入式处理器使用最多的外设，能够充分利用其通用性和灵活性，是嵌入式开发者必须掌握的内容。作为输入时，GPIO 可以接收来自外部的开关量信号、脉冲信号等，如来自键盘、拨码开关的信号；作为输出时，GPIO 可以将内部的数据传输给外部设备或模块，如输出到 LED、数码管、控制继电器等。另外，从理论上讲，当嵌入式处理器上没有足够的外设时，可以通过软件控制 GPIO 模拟 UART、SPI、I2C、FSMC 等各种外设的功能。

正因为 GPIO 作为外设具有无与伦比的重要性，STM32 上除特殊功能的引脚外，所有引脚都可以作为 GPIO 使用。以常见的 LQFP144 封装的 STM32F407ZGT6 为例，有 112 个引脚可以作为双向 I/O 使用。为便于使用和记忆，STM32 将它们分配到不同的“组”中，在每个组中再对其进行编号。具体来讲，每个组称为一个端口，端口号通常以大写字母命名，从 A 开始，依次简写为 PA、PB 或 PC 等。每个端口中最多有 16 个 GPIO，软件既可以读写单个 GPIO，也可以通过指令一次读写端口中全部 16 个 GPIO。每个端口内部的 16 个 GPIO 又被分别标以 0～15 的编号，从而可以通过 PA0、PB5 或 PC10 等方式指代单个的 GPIO。以 STM32F407ZGT6 为例，它共有 7 个端口(PA、PB、PC、PD、PE、PF 和 PG)，每个端口有 16 个 GPIO，共有 7×16＝112 个 GPIO。

几乎在所有嵌入式系统应用中，都涉及开关量的输入和输出功能，如状态指示、报警输出、继电器闭合和断开、按钮状态读入、开关量报警信息的输入等。这些开关量的输入和控制输出都可以通过 GPIO 实现。

GPIO 的每个位都可以由软件分别配置成以下模式。

(1) 输入浮空：浮空(Floating)就是逻辑器件的输入引脚既不接高电平，也不接低电平。由于逻辑器件的内部结构，当引脚输入浮空时，相当于该引脚接了高电平。一般实际运用时，引脚不建议浮空，易受干扰。

(2) 输入上拉：上拉就是把电压拉高，如拉到V_{CC}。上拉就是将不确定的信号通过一个电阻钳位在高电平。电阻同时起限流作用。弱强只是上拉电阻的阻值不同，没有什么严格区分。

(3) 输入下拉：下拉就是把电压拉低，拉到 GND。与上拉原理相似。

(4) 模拟输入：模拟输入是指传统方式的模拟量输入。数字输入是输入数字信号，即 0 和 1 的二进制数字信号。

(5) 具有上拉/下拉功能的开漏输出模式：输出端相当于三极管的集电极。要得到高电平状态，需要上拉电阻才行。该模式适用于电流型的驱动，其吸收电流的能力相对较强(一般 20mA 以内)。

(6) 具有上拉/下拉功能的推挽输出模式：可以输出高低电平，连接数字器件。推挽结构一般是指两个三极管分别受两个互补信号的控制，总是在一个三极管导通时另一个截止。

(7) 具有上拉/下拉功能的复用功能推挽模式：复用功能可以理解为 GPIO 被用作第二功能时的配置情况(并非作为通用 I/O 接口使用)。STM32 GPIO 的推挽复用模式中输出使能、输出速度可配置。这种复用模式可工作在开漏及推挽模式，但是输出信号是源于其他外设的，这时的输出数据寄存器 GPIOx_ODR 是无效的；而且输入可用，通过输入数据寄存器可获取 I/O 接口实际状态，但一般直接用外设的寄存器获取该数据信号。

(8) 具有上拉/下拉功能的复用功能开漏模式：复用功能可以理解为 GPIO 被用作第二功能时的配置情况(并非作为通用 I/O 接口使用)。每个 I/O 接口可以自由编程，而 I/O 接口寄存器必须按 32 位字访问(不允许半字或字节访问)。GPIOx_BSRR 和 GPIOx_BRR 寄存器允许对任何 GPIO 寄存器的读/更改的独立访问，这样，在读和更改访问之间产生中断时不会发生危险。

每个 GPIO 端口包括 4 个 32 位配置寄存器(GPIOx_MODER、GPIOx_OTYPER、GPIOx_OSPEEDR 和 GPIOx_PUPDR)、两个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR)、一个 32 位置位/复位寄存器(GPIOx_BSRR)、一个 32 位配置锁存寄存器(GPIOx_LCKR)和两个 32 位复用功能选择寄存器(GPIOx_AFRH 和 GPIOx_AFRL)。应用程序通过对这些寄存器的操作实现 GPIO 的配置和应用。

一个 I/O 端口的基本结构如图 5-1 所示。

STM32 的 GPIO 资源非常丰富，包括 26、37、51、80、112 个多功能双向 5V 兼容的快速 I/O 接口，而且所有 I/O 接口可以映射到 16 个外部中断，对于 STM32 的学习，应该从最基本的 GPIO 开始学习。

GPIO 端口的每个位可以由软件分别配置成多种模式。常用的 I/O 寄存器只有 4 个：

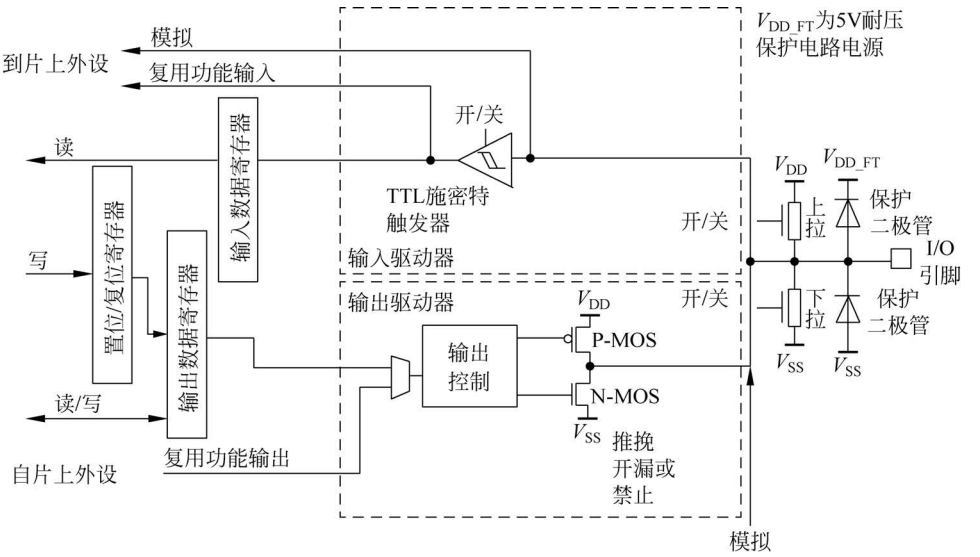

图 5-1　一个 I/O 端口的基本结构

CRL、CRH、IDR、ODR。CRL 和 CRH 控制着每个 I/O 的模式及输出速率。

每个 GPIO 引脚都可以由软件配置成输出(推挽或开漏)、输入(带或不带上拉或下拉)或复用的外设功能。多数 GPIO 引脚都与数字或模拟的复用外设共用。除了具有模拟输入功能的端口,所有 GPIO 引脚都有大电流通过能力。

根据数据手册中列出的每个 I/O 端口的特定硬件特征,GPIO 端口的每个位可以由软件分别配置成多种模式:输入浮空、输入上拉、输入下拉、模拟输入、开漏输出、推挽式输出、推挽式复用功能、开漏复用功能。

I/O 端口的基本结构包括以下几部分。

5.1.1　输入通道

输入通道包括输入数据寄存器和输入驱动器。在接近 I/O 引脚处连接了两只保护二极管,假设保护二极管的导通电压降为 V_d,则输入输入驱动器的信号 V_{in} 电压范围被钳位在

$$V_{SS}-V_d<V_{in}<V_{DD}+V_d$$

由于 V_d 的导通压降不会超过 0.7V,若电源电压 V_{DD} 为 3.3V,则输入输入驱动器的信号不会低于−0.7V,不会高于 4V,起到了保护作用。在实际工程设计中,一般都将输入信号尽可能调理到 0～3.3V,也就是说,一般情况下,两个保护二极管都不会导通,输入驱动器中包括了两个电阻,分别通过开关接电源 V_{DD}(该电阻称为上拉电阻)和地 V_{SS}(该电阻称为下拉电阻)。开关受软件的控制,用来设置当 I/O 用作输入时,选择使用上拉电阻或下拉电阻。

输入驱动器中的另一个部件是 TTL 施密特触发器,当 I/O 用于开关量输入或复用功

能输入时，TTL 施密特触发器用于对输入波形进行整形。

GPIO 的输入驱动器主要由 TTL 肖特基触发器、带开关的上拉电阻电路和带开关的下拉电阻电路组成。值得注意的是，与输出驱动器不同，GPIO 的输入驱动器没有多路选择开关，输入信号送到 GPIO 输入数据寄存器的同时也送给片上外设，所以 GPIO 的输入没有复用功能选项。

根据 TTL 肖特基触发器、上拉电阻端和下拉电阻端两个开关的状态，GPIO 的输入可分为以下 4 种。

(1) 模拟输入：TTL 肖特基触发器关闭。

(2) 上拉输入：GPIO 内置上拉电阻，此时 GPIO 内部上拉电阻端的开关闭合，GPIO 内部下拉电阻端的开关打开。该模式下，引脚在默认情况下输入为高电平。

(3) 下拉输入：GPIO 内置下拉电阻，此时 GPIO 内部下拉电阻端的开关闭合，GPIO 内部上拉电阻端的开关打开。该模式下，引脚在默认情况下输入为低电平。

(4) 浮空输入：GPIO 内部既无上拉电阻也无下拉电阻，此时 GPIO 内部上拉电阻端和下拉电阻端的开关都处于打开状态。该模式下，引脚在默认情况下为高阻态(即浮空)，其电平高低完全由外部电路决定。

5.1.2 输出通道

输出通道包括置位/清除寄存器、输出数据寄存器、输出驱动器。

要输出的开关量数据首先写入置位/清除寄存器，通过读写命令进入输出数据寄存器，然后进入输出驱动器的输出控制模块。输出控制模块可以接收开关量的输出和复用功能输出。输出的信号通过由 P-MOS 和 N-MOS 场效应管电路输出到引脚。通过软件设置，由 P-MOS 和 N-MOS 场效应管电路可以构成推挽方式、开漏方式或关闭。

GPIO 的输出驱动器主要由多路选择器、输出控制逻辑和一对互补的 MOS 晶体管组成。

1) 多路选择器

多路选择器根据用户设置决定该引脚是 GPIO 普通输出还是复用功能输出。

(1) 普通输出：该引脚的输出来自 GPIO 的输出数据寄存器。

(2) 复用功能(Alternate Function，AF)输出：该引脚的输出来自片上外设，并且一个 STM32 微控制器引脚输出可能来自多个不同外设，即一个引脚可以对应多个复用功能输出。但同一时刻，一个引脚只能使用一个复用功能，而这个引脚对应的其他复用功能都处于禁止状态。

2) 输出控制逻辑和一对互补的 MOS 晶体管

输出控制逻辑根据用户设置通过控制 P-MOS 和 N-MOS 场效应管的状态(导通/关闭)决定 GPIO 输出模式(推挽、开漏或关闭)。

(1) 推挽(Push-Pull，PP)输出：可以输出高电平和低电平。当内部输出 1 时，P-MOS

管导通，N-MOS 管截止，外部输出高电平（输出电压为 V_{DD}）；当内部输出 0 时，N-MOS 管导通，P-MOS 管截止，外部输出低电平（输出电压为 0V）。

由此可见，相比于普通输出方式，推挽输出既提高了负载能力，又提高了开关速度，适用于输出 0V 和 V_{DD} 的场合。

（2）开漏（Open-Drain，OD）输出：与推挽输出相比，开漏输出中连接 V_{DD} 的 P-MOS 管始终处于截止状态。这种情况与三极管的集电极开路非常类似。在开漏输出模式下，当内部输出 0 时，N-MOS 管导通，外部输出低电平（输出电压为 0V）；当内部输出 1 时，N-MOS 管截止，由于此时 P-MOS 管也处于截止状态，外部输出既不是高电平，也是不是低电平，而是高阻态（浮空）。如果想要外部输出高电平，必须在 I/O 引脚外接一个上拉电阻。

这样，通过开漏输出，可以提供灵活的电平输出方式——改变外接上拉电源的电压，便可以改变传输电平电压的高低。例如，如果 STM32 微控制器想要输出 5V 高电平，只需要在外部接一个上拉电阻且上拉电源为 5V，并把 STM32 微控制器对应的 I/O 引脚设置为开漏输出模式，当内部输出 1 时，由上拉电阻和上拉电源向外输出 5V 电平。需要注意的是，上拉电阻的阻值决定逻辑电平电压转换的速度。阻值越大，速度越低，功耗越小，所以上拉电阻的选择应兼顾功耗和速度。

由此可见，开漏输出可以匹配电平，一般适用于电平不匹配的场合，而且开漏输出吸收电流的能力相对较强，适合作为电流型的驱动。

5.2 STM32 的 GPIO 功能

下面讲述 STM32 的 GPIO 功能。

5.2.1 普通 I/O 功能

复位期间和刚复位后，复用功能未开启，I/O 端口被配置成浮空输入模式。

复位后，JTAG 引脚被置于输入上拉或下拉模式。

（1）PA13：JTMS 置于上拉模式。

（2）PA14：JTCK 置于下拉模式。

（3）PA15：JTDI 置于上拉模式。

（4）PB4：JNTRST 置于上拉模式。

当作为输出配置时，写到输出数据寄存器（GPIOx_ODR）的值输出到相应的 I/O 引脚。可以以推挽模式或开漏模式（当输出 0 时，只有 N-MOS 管被打开）使用输出驱动器。

输入数据寄存器（GPIOx_IDR）在每个 APB2 时钟周期捕捉 I/O 引脚上的数据。

所有 GPIO 引脚有一个内部弱上拉和弱下拉，当配置为输入时，它们可以被激活，也可以被断开。

5.2.2 单独的位设置或位清除

当对 GPIOx_ODR 的个别位编程时，软件不需要禁止中断：在单次 APB2 写操作中，可以只更改一个或多个位。这是通过对置位/复位寄存器中想要更改的位写 1 实现的。没被选择的位将不被更改。

5.2.3 外部中断/唤醒线

所有端口都有外部中断能力。为了使用外部中断线，端口必须配置成输入模式。

5.2.4 复用功能

使用默认复用功能(AF)前必须对端口位配置寄存器编程。

(1) 对于复用输入功能，端口必须配置成输入模式(浮空、上拉或下拉)且输入引脚必须由外部驱动。

(2) 对于复用输出功能，端口必须配置成复用功能输出模式(推挽或开漏)。

(3) 对于双向复用功能，端口必须配置成复用功能输出模式(推挽或开漏)。此时，输入驱动器被配置成浮空输入模式。

如果把端口配置成复用输出功能，则引脚和输出寄存器断开，并和片上外设的输出信号连接。

如果软件把一个 GPIO 引脚配置成复用输出功能，但是外设没有被激活，那么它的输出将不确定。

5.2.5 软件重新映射 I/O 复用功能

STM32F407 微控制器的 I/O 引脚除了通用功能外，还可以设置为一些片上外设的复用功能。而且，一个 I/O 引脚除了可以作为某个默认外设的复用引脚外，还可以作为其他多个不同外设的复用引脚。类似地，一个片上外设，除了默认的复用引脚，还可以有多个备用的复用引脚。在基于 STM32 微控制器的应用开发中，用户根据实际需要可以把某些外设的复用功能从默认引脚转移到备用引脚上，这就是外设复用功能的 I/O 引脚重映射。

为了使不同封装器件的外设 I/O 功能的数量达到最优，可以把一些复用功能重新映射到其他一些引脚上。这可以通过软件配置 AFIO 寄存器完成，这时，复用功能就不再映射到它们的原始引脚上了。

5.2.6 GPIO 锁定机制

锁定机制允许冻结 I/O 配置。当在一个端口位上执行了锁定(LOCK)程序，在下一次复位之前，将不能再更改端口位的配置。这个功能主要用于一些关键引脚的配置，防止程序

跑飞引起灾难性后果。

5.2.7 输入配置

当 I/O 配置为输入时：

(1) 输出缓冲器被禁止；

(2) 施密特触发输入被激活；

(3) 根据输入配置(上拉、下拉或浮动)的不同，弱上拉和下拉电阻被连接；

(4) 出现在 I/O 引脚上的数据在每个 APB2 时钟被采样到输入数据寄存器；

(5) 对输入数据寄存器的读访问可得到 I/O 状态。

I/O 输入配置如图 5-2 所示。

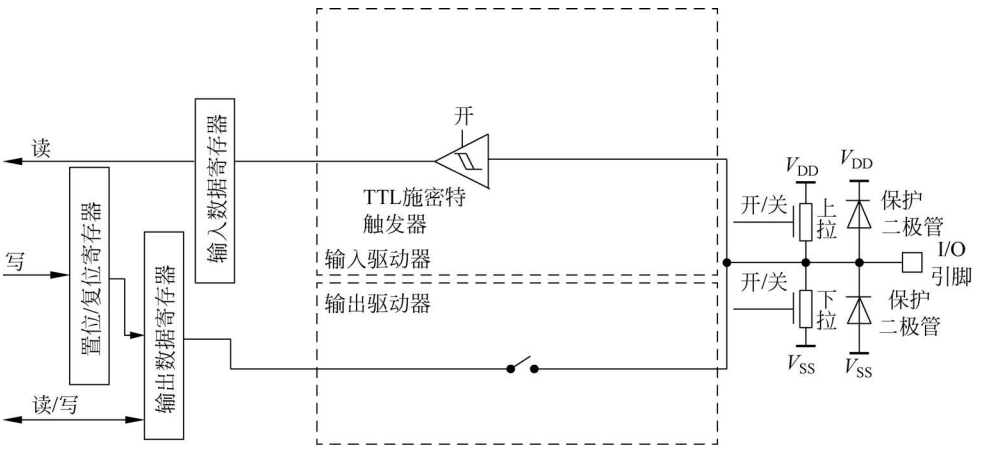

图 5-2 I/O 输入配置

5.2.8 输出配置

当 I/O 被配置为输出时：

(1) 输出缓冲器被激活，开漏模式下，输出寄存器上的 0 激活 N-MOS 管，而输出寄存器上的 1 将端口置于高阻状态(P-MOS 管从不被激活)；推挽模式下，输出寄存器上的 0 激活 N-MOS 管，而输出寄存器上的 1 将激活 P-MOS 管；

(2) 施密特触发输入被激活；

(3) 弱上拉和下拉电阻被禁止；

(4) 出现在 I/O 引脚上的数据在每个 APB2 时钟被采样到输入数据寄存器；

(5) 开漏模式下，对输入数据寄存器的读访问可得到 I/O 状态；

(6) 推挽式模式下，对输出数据寄存器的读访问得到最后一次写的值。

I/O 的输出配置如图 5-3 所示。

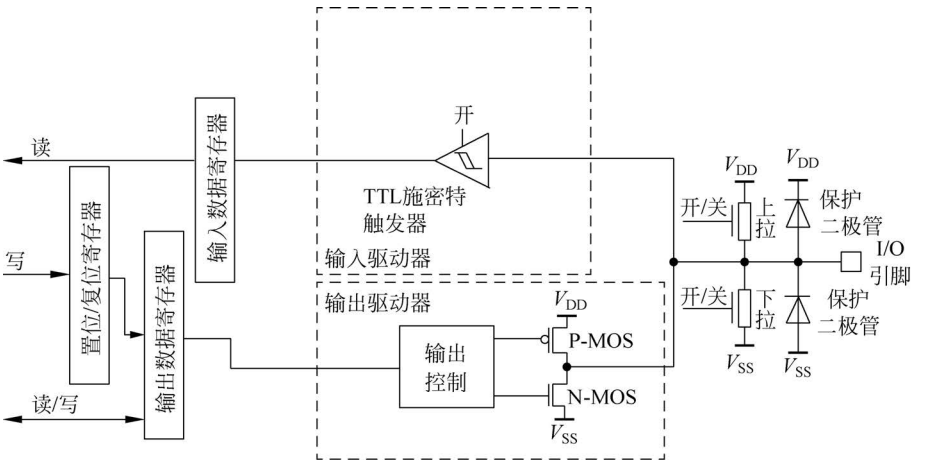

图 5-3 I/O 输出配置

5.2.9 复用功能配置

当 I/O 被配置为复用功能时：

(1) 在开漏或推挽式配置中，输出缓冲器被打开；

(2) 内置外设的信号驱动输出缓冲器(复用功能输出)；

(3) 施密特触发输入被激活；

(4) 弱上拉和下拉电阻被禁止；

(5) 在每个 APB2 时钟周期，出现在 I/O 引脚上的数据被采样到输入数据寄存器；

(6) 开漏模式时，读输入数据寄存器时可得到 I/O 状态；

(7) 在推挽模式时，读输出数据寄存器时可得到最后一次写的值。

一组复用功能 I/O 寄存器允许用户把一些复用功能重新映像到不同的引脚。

I/O 复用功能配置如图 5-4 所示。

5.2.10 模拟输入配置

当 I/O 被配置为模拟输入配置时：

(1) 输出缓冲器被禁止；

(2) 禁止施密特触发输入，实现了每个模拟 I/O 引脚上的零消耗，施密特触发输出值被强置为 0；

(3) 弱上拉和下拉电阻被禁止；

(4) 读取输入数据寄存器时数值为 0。

I/O 高阻抗模拟输入配置如图 5-5 所示。

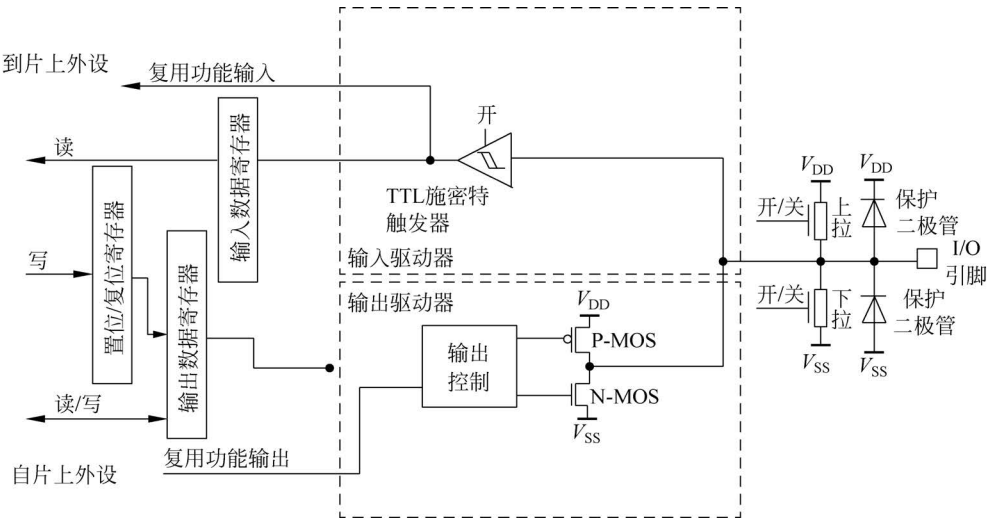

图 5-4 I/O 复用功能配置

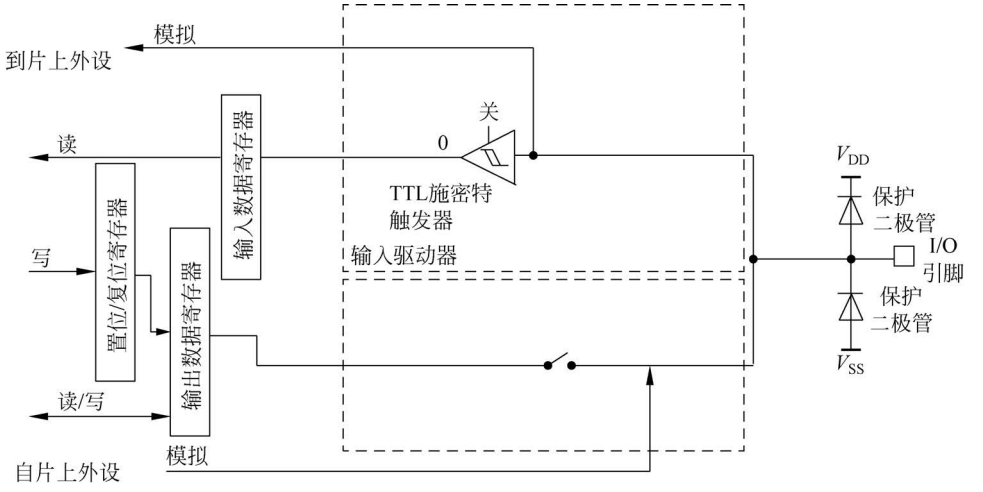

图 5-5 I/O 高阻抗模拟输入配置

5.2.11 STM32 的 GPIO 操作

下面讲述 STM32 的 GPIO 操作方法。

1. 复位后的 GPIO

为防止复位后 GPIO 引脚与片外电路的输出冲突，复位期间和刚复位后，所有 GPIO 引脚复用功能都不开启。被配置成浮空输入模式。

为了节约电能，只有被开启的 GPIO 端口才会被提供时钟。因此，复位后所有 GPIO 端口的时钟都是关断的，使用之前必须逐一开启。

2. GPIO 工作模式的配置

每个 GPIO 引脚都拥有自己的端口配置位 MODERy[1:0](模式寄存器,其中 y 代表 GPIO 引脚在端口中的编号)和 OTy[1:0](输出类型寄存器),用于选择该引脚是处于输入模式中的浮空输入模式、上位/下拉输入模式或模拟输入模式,还是输出模式中的输出推挽模式、开漏输出模式或复用功能推挽/开漏输出模式。每个 GPIO 引脚还拥有自己的端口模式位 OSPEEDRy[1:0],用于选择该引脚是处于输入模式,或是输出模式中的输出带宽(2MHz、25MHz、50MHz 和 100MHz)。

每个端口拥有 16 个引脚,而每个引脚又拥有 4 个控制位,因此需要 64 位才能实现对一个端口所有引脚的配置,它们被分置在两个字(Word)中。如果是输出模式,还需要 16 位输出类型寄存器。各种工作模式下的硬件配置总结如下。

(1) 输入模式的硬件配置:输出缓冲器被禁止;施密特触发器输入被激活;根据输入配置(上拉、下拉或浮空)的不同,弱上拉和下拉电阻被连接;出现在 I/O 引脚上的数据在每个 APB2 时钟被采样到输入数据寄存器;对输入数据寄存器的读访问可得到 I/O 状态。

(2) 输出模式的硬件配置:输出缓冲器被激活;施密特触发器输入被激活;弱上拉和下拉电阻被禁止;出现在 I/O 引脚上的数据在每个 APB2 时钟被采样到输入数据寄存器;对输入数据寄存器的读访问可得到 I/O 状态;对输出数据寄存器的读访问得到最后一次写的值;在推挽模式下,互补 MOS 管对都能被打开;在开漏模式下,只有 N-MOS 管可以被打开。

(3) 复用功能的硬件配置:在开漏或推挽式配置中,输出缓冲器被打开;片上外设的信号驱动输出缓冲器;施密特触发器输入被激活;弱上拉和下拉电阻被禁止;在每个 APB2 时钟周期,出现在 I/O 引脚上的数据被采样到输入数据寄存器;对输出数据寄存器的读访问得到最后一次写的值;在推挽模式下,互补 MOS 管对都能被打开;在开漏模式下,只有 N-MOS 管可以被打开。

3. GPIO 输入的读取

每个端口都有自己对应的输入数据寄存器 GPIOx_IDR(其中 x 代表端口号,如 GPIOA_IDR),它在每个 APB2 时钟周期捕捉 I/O 引脚上的数据。软件可以通过直接读取 GPIOx_IDR 寄存器某个位或读取位带别名区中对应字得到 GPIO 引脚状态对应的值。

4. GPIO 输出的控制

STM32 为每组 16 引脚的端口提供了 3 个 32 位的控制寄存器:GPIOx_ODR、GPIOx_BSRR 和 GPIOx_BRR(其中 x 指代 A、B、C 等端口号)。其中,GPIOx_ODR 寄存器的功能比较容易理解,它的低 16 位直接对应了端口的 16 个引脚,软件可以通过直接对这个寄存器的置位或清零,让对应引脚输出高电平或低电平。也可以利用位带操作原理,对 GPIOx_ODR 寄存器中某个位对应的位带别名区字地址执行写入操作以实现对单个位的简化操作。利用 GPIOx_ODR 寄存器的位带操作功能可以有效地避免端口中其他引脚的“读-修-写”问题,但位带操作的缺点是每次只能操作一位,对于某些需要同时操作多个引脚的应用,位带操作就显得力不从心了。STM32 的解决方案是使用 GPIOx_BSRR 和 GPIOx_BRR 两个

寄存器解决多个引脚同时改变电平的问题。

5. 输出速度

如果 STM32F407 的 I/O 引脚工作在某个输出模式下，通常还需设置其输出速度，这个输出速度指的是 I/O 驱动电路的响应速度，而不是输出信号的速度。输出信号的速度取决于软件程序。

STM32F407 的芯片内部在 I/O 的输出部分安排了多个响应速度不同的输出驱动电路，用户可以根据自己的需要，通过选择响应速度选择合适的输出驱动模块，以达到最佳噪声控制和降低功耗的目的。众所周知，高频的驱动电路噪声也高。当不需要高输出频率时，尽量选用低频响应速度的驱动电路，这样非常有利于提高系统的电磁干扰（Electro Magnetic Interference，EMI）性能。当然，如果要输出较高频率的信号，但却选用了较低频率的驱动模块，很可能会得到失真的输出信号。一般推荐 I/O 引脚的输出速度是其输出信号速度的 5～10 倍。

STM32F407 的 I/O 引脚的输出速度有 4 种选择：2MHz、25MHz、50MHz 和 100MHz。下面根据一些常见的应用，给读者一些选用参考。

（1）连接 LED、蜂鸣器等外部设备的普通输出引脚：一般设置为 2MHz。

（2）用作 USART 复用功能输出引脚：假设 USART 工作时最大比特率为 115.2kb/s，选用 2MHz 的响应速度也足够了，既省电，噪声又小。

（3）用作 I2C 复用功能的输出引脚：假设 I2C 工作时最大比特率为 400kb/s，那么 2MHz 的引脚速度或许不够，这时可以选用 10MHz 的 I/O 引脚速度。

（4）用作 SPI 复用功能的输出引脚：假设 SPI 工作时比特率为 18Mb/s 或 9Mb/s，那么 10MHz 的引脚速度显然不够，这时需要选用 50MHz 的 I/O 引脚速度。

（5）用作 FSMC 复用功能连接存储器的输出引脚：一般设置为 50MHz 或 100MHz 的 I/O 引脚速度。

5.2.12 外部中断映射和事件输出

借助 AFIO，STM32F407 微控制器的 I/O 引脚不仅可以实现外设复用功能的重映射，而且可以实现外部中断映射和事件输出。需要注意的是，如需使用 STM32F407 控制器 I/O 引脚的以上功能，必须先打开 APB2 总线上的 AFIO 时钟。

1. 外部中断映射

当 STM32 微控制器的某个 I/O 引脚被映射为外部中断线后，该 I/O 引脚就可以成为一个外部中断源，可以在这个 I/O 引脚上产生外部中断，实现对用户 STM32 运行程序的交互。

STM32 微控制器的所有 I/O 引脚都具有外部中断能力。每根外部中断线 EXTI LineXX 和所有的 GPIO 端口 GPIO[A..G].XX 共享。为了使用外部中断线，该 I/O 引脚必须配置成输入模式。

2. 事件输出

STM32 微控制器几乎每个 I/O 引脚(除端口 F 和 G 的引脚外)都可用作事件输出。例如,使用 SEV 指令产生脉冲,通过事件输出信号将 STM32 从低功耗模式唤醒。

5.2.13 GPIO 的主要特性

综上所述,STM32F407 微控制器的 GPIO 主要具有以下特性。

(1) 提供最多 112 个多功能双向 I/O 引脚,80%的引脚利用率。

(2) 几乎每个 I/O 引脚(除 ADC 外)都兼容 5V,每个 I/O 引脚具有 20mA 驱动能力。

(3) 每个 I/O 引脚最高具有 84MHz 的翻转速度,30 pF 时为 100 MHz 输出,15 pF 时为 80MHz 输出。

(4) 每个 I/O 引脚有 8 种工作模式,在复位时和刚复位后,复用功能未开启,I/O 引脚被配置成浮空输入模式。

(5) 所有 I/O 引脚都具备复用功能,包括 JTAG/SWD、Timer、USART、I2C、SPI 等。

(6) 某些复用功能引脚可通过复用功能重映射用作另一个复用功能,方便 PCB 设计。

(7) 所有 I/O 引脚都可作为外部中断输入,同时可以有 16 个中断输入。

(8) 几乎每个 I/O 引脚(除端口 F 和 G 外)都可用作事件输出。

(9) PA0 可作为从待机模式唤醒的引脚,PC13 可作为入侵检测的引脚。

5.3 GPIO 的 HAL 驱动程序

GPIO 引脚的操作主要包括初始化、读取引脚输入和设置引脚输出,相关的 HAL 驱动程序定义在 stm32f4xx_hal_gpio.h 文件中。GPIO 操作相关函数如表 5-1 所示,表中只列出了函数名,省略了函数参数。

表 5-1 GPIO 操作相关函数

函 数 名	功 能 描 述
HAL_GPIO_Init()	GPIO 引脚初始化
HAL_GPIO_DeInit()	GPIO 引脚解除初始化,恢复为复位后的状态
HAL_GPIO_WritePin()	使引脚输出 0 或 1
HAL_GPIO_ReadPin()	读取引脚的输入电平
HAL_GPIO_TogglePin()	翻转引脚的输出
HAL_GPIO_LockPin()	锁定引脚配置,而不是锁定引脚的输入或输出状态

使用 STM32CubeMX 生成代码时,GPIO 引脚初始化的代码会自动生成,用户常用的 GPIO 操作函数是进行引脚状态读写的函数。

1. 初始化函数 HAL_GPIO_Init()

HAL_GPIO_Init()函数用于对一个端口的一个或多个相同功能的引脚进行初始化设置,包括输入/输出模式、上拉或下拉等。原型定义如下。

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

其中,第 1 个参数 GPIOx 是 GPIO_TypeDef 类型的结构体指针,它定义了端口的各个寄存器的偏移地址,实际调用 HAL_GPIO_Init()函数时使用端口的基地址作为 GPIOx 的值,在 stm32f407xx.h 文件中定义了各个端口的基地址,如

```
#define  GPIOA    ((GPIO_TypeDef * )GPIOA_BASE)
#define  GPIOB    ((GPIO_TypeDef * )GPIOB_BASE)
#define  GPIOC    ((GPIO_TypeDef * )GPIOC_BASE)
#define  GPIOD    ((GPIO_TypeDef * )GPIOD_BASE)
```

第 2 个参数 GPIO_Init 是一个 GPIO InitTypeDef 类型的结构体指针,它定义了 GPIO 引脚的属性,这个结构体的定义如下。

```
typedef  struct
{
    uint32_t  Pin;                    //要配置的引脚,可以是多个引脚
    uint32_t  Mode;                   //引脚功能模式
    uint32_t  Pull;                   //上拉或下拉
    uint32_t  Speed;                  //引脚最高输出频率
    uint32_t  Alternate;              //复用功能选择
}GPIO_InitTypeDef;
```

这个结构体的各个成员变量的意义及取值如下。

(1) Pin 是需要配置的 GPIO 引脚,在 stm32f4xx hal_gpio.h 文件中定义了 16 个引脚的宏。如果需要同时定义多个引脚的功能,就用这些宏的或运算进行组合。

```
#define  GPIO_PIN_0   ((uint16_t)0x0001)    /*  Pin  0  selected   */
#define  GPIO_PIN_1   ((uint16_t)0x0002)    /*  Pin  1  selected   */
#define  GPIO_PIN_2   ((uint16_t)0x0004)    /*  Pin  2  selected   */
#define  GPIO_PIN_3   ((uint16_t)0x0008)    /*  Pin  3  selected   */
#define  GPIO_PIN_4   ((uint16_t)0x0010)    /*  Pin  4  selected   */
#define  GPIO_PIN_5   ((uint16_t)0x0020)    /*  Pin  5  selected   */
#define  GPIO_PIN_6   ((uint16_t)0x0040)    /*  Pin  6  selected   */
#define  GPIO_PIN_7   ((uint16_t)0x0080)    /*  Pin  7  selected   */
#define  GPIO_PIN_8   ((uint16_t)0x0100)    /*  Pin  8  selected   */
#define  GPIO_PIN_9   ((uint16_t)0x0200)    /*  Pin  9  selected   */
#define  GPIO_PIN_10  ((uint16_t)0x0400)    /*  Pin  10  selected  */
#define  GPIO_PIN_11  ((uint16_t)0x0800)    /*  Pin  11  selected  */
#define  GPIO_PIN_12  ((uint16_t)0x1000)    /*  Pin  12  selected  */
#define  GPIO_PIN_13  ((uint16_t)0x2000)    /*  Pin  13  selected  */
#define  GPIO_PIN_14  ((uint16_t)0x4000)    /*  Pin  14  selected  */
#define  GPIO_PIN_15  ((uint16_t)0x8000)    /*  Pin  15  selected  */
#define  GPIO_PIN_All  ((uint16_t)0xFFFF)   /*  All  pins  selected  */
```

(2) Mode 是引脚功能模式设置,其可用常量定义如下。

```
#define  GPIO_MODE_INPUT                  0x00000000U          //输入浮空模式
#define  GPIO_MODE_OUTPUT_PP              0x00000001U          //推挽输出模式
#define  GPIO_MODE_OUTPUT_OD              0x000000110          //开漏输出模式
#define  GPIO_MODE_AF_PP                  0x00000002U          //复用功能推挽模式
#define  GPIO_MODE_AF_OD                  0x00000012U          //复用功能开漏模式
#define  GPIO_MODE_ANALOG                 0x000000030          //模拟信号模式
#define  GPIO_MODE_IT_RISING              0x10110000U          //外部中断,上升沿触发
#define  GPIO_MODE_IT_FALLING             0x10210000U          //外部中断,下降沿触发
#define  GPIO_MODE_IT_RISING_FALLING      0x10310000U          //上升、下降沿触发
```

(3) Pull 定义是否使用内部上拉或下拉电阻,其可用常量定义如下。

```
#define  GPIO_NOPULL          0x00000000U                    //无上拉或下拉
#define  GPIO_PULLUP          0x00000001U                    //上拉
#define  GPIO_PULLDOWN        0x00000002U                    //下拉
```

(4) Speed 定义输出模式引脚的最高输出频率,其可用常量定义如下。

```
#define  GPIO_SPEED_FREQ_LOW          0x00000000U          //2MHz
#define  GPIO_SPEED_FREQ_MEDIUM       0x00000001U          //12.5～50MHz
#define  GPIO_SPEED_FREQ_HIGH         0x00000002U          //25～100MHz
#define  GPIO_SPEED_FREQ_VERY_HIGH    0x000000030          //50～200MHz
```

(5) Alternate 定义引脚的复用功能,在 stm32f4xx hal gpio_ex.h 文件中定义了这个参数的可用宏定义,这些复用功能的宏定义与具体的 MCU 型号有关,部分定义示例如下。

```
#define  GPIO_AF1_TIM1    ((uint8_t)0x01)        // TIM1 复用功能映射
#define  GPIO_AF1_TIM2    ((uint8_t)0x01)        // TIM2 复用功能映射
#define  GPIO_AF5_SPI1    ((uint8_t)0x05)        // SPI1 复用功能映射
#define  GPIO_AF5_SPI2    ((uint8_t)0x05)        // SPI2 复用功能映射
#define  GPIO_AF7_USART1  ((uint8_t)0x07)        // USART1 复用功能映射
#define  GPIO_AF7_USART2  ((uint8_t)0x07)        // USART2 复用功能映射
#define  GPIO_AF7_USART3  ((uint8_t)0x07)        // USART3 复用功能映射
```

2. 设置引脚输出的 HAL GPIO_WritePin()函数

使用 HAL_GPIO_WritePin()函数向一个或多个引脚输出高电平或低电平。原型定义如下。

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

其中,参数 GPIOx 是具体的端口基地址; GPIO_Pin 是引脚号; PinState 是引脚输出电平,它是 GPIO_PinState 枚举类型,在 stm32f4xx_hal_gpio.h 文件中的定义如下。

```
typedef enum
{
```

```
    GPIO_PIN_RESET = 0,
    GPIO_PIN_SET
}GPIO_PinState;
```

GPIO_PIN_RESET 表示低电平，GPIO_PIN_SET 表示高电平。例如，要使 PF9 和 PF10 输出低电平，可使用如下代码。

```
HAL_GPIO_WritePin (GPIOF,GPIO_PIN_9|GPIO_PIN_10,GPIO_PIN_RESET);
```

若要输出高电平，只需修改为如下代码。

```
HAL_GPIO_WritePin(GPIOF,GPIO_PIN_9|GPIO_PIN_10,GPIO_PIN_SET);
```

3. 读取引脚输入的 HAL_GPIO_ReadPin()函数

HAL_GPIO_ReadPin()函数用于读取一个引脚的输入状态。原型定义如下。

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

函数的返回值是 GPIO_PinState 枚举类型。GPIO_PIN_RESET 表示输入为 0(低电平)，GPIO_PIN SET 表示输入为 1(高电平)。

4. 翻转引脚输出的 HAL_GPIO_TogglePin()函数

HAL_GPIO_TogglePin()函数用于翻转引脚的输出状态。例如，引脚当前输出为高电平，执行此函数后，引脚输出为低电平。原型定义如下，只需传递端口号和引脚号。

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

5.4 STM32 的 GPIO 使用流程

根据 I/O 端口的特定硬件特征，I/O 端口的每个引脚都可以由软件配置成多种工作模式。在运行程序之前，必须对每个用到的引脚功能进行配置。

(1) 如果某些引脚的复用功能没有使用，可以先配置为 GPIO。

(2) 如果某些引脚的复用功能被使用，需要对复用的 I/O 端口进行配置。

(3) I/O 端口具有锁定机制，允许冻结 I/O 端口。当在一个端口位上执行了锁定(LOCK)程序后，在下一次复位之前，将不能再更改端口位的配置。

5.4.1 普通 GPIO 配置

GPIO 是最基本的应用，其基本配置方法如下。

(1) 配置 GPIO 时钟，完成初始化。

(2) 利用 HAL_GPIO_Init()函数配置引脚，包括引脚名称、引脚传输速率、引脚工作模式。

(3) 完成 HAL_GPIO_Init()函数的设置。

5.4.2 I/O 复用功能 AFIO 配置

I/O 复用功能 AFIO 常对应到外设的输入输出功能。使用时,需要先配置 I/O 为复用功能,打开 AFIO 时钟,然后再根据不同的复用功能进行配置。对应外设的输入输出功能有以下 3 种情况。

(1) 外设对应的引脚为输出:需要根据外围电路的配置选择对应的引脚为复用功能的推挽输出或复用功能的开漏输出。

(2) 外设对应的引脚为输入:根据外围电路的配置可以选择浮空输入、带上拉输入或带下拉输入。

(3) ADC 对应的引脚:配置引脚为模拟输入。

5.5 采用 STM32CubeMX 和 HAL 库的 GPIO 输出应用实例

本 GPIO 输出应用实例是使用固件库点亮 LED。

5.5.1 STM32 的 GPIO 输出应用硬件设计

STM32F407 与 LED 的连接如图 5-6 所示。这是一个 RGB LED 灯,由红、蓝、绿 3 个 LED 构成,使用 PWM 控制时可以混合成不同的颜色。

这些 LED 的阴极都连接到 STM32F407 的 GPIO 引脚,只要控制 GPIO 引脚的电平输出状态,即可控制 LED 的亮灭。如果使用的开发板中 LED 的连接方式或引脚不一样,只需修改程序的相关引脚即可,程序的控制原理相同。

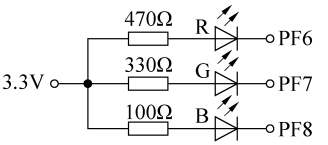

图 5-6 STM32F407 与 LED 的连接

LED 电路是由外接 3.3V 电源驱动的。当 GPIO 引脚输出为 0 时,LED 点亮;输出为 1 时,LED 熄灭。

在本实例中,根据图 5-6 的电路设计一个示例,使 LED 循环显示如下。

(1) 红灯亮 1s,灭 1s;

(2) 绿灯亮 1s,灭 1s;

(3) 蓝灯亮 1s,灭 1s;

(4) 红灯亮 1s,灭 1s;

(5) 轮流显示,红、绿、蓝、黄、紫、青、白各亮 1s;

(6) 关灯 1s。

5.5.2 STM32 的 GPIO 输出应用软件设计

1. 通过 STM32CubeMX 新建工程

通过 STM32CubeMX 新建工程的步骤如下。

1）新建文件夹

在D盘根目录下新建Demo文件夹，这是保存所有工程的地方，在该目录下新建LED文件夹，这是保存本实例新建工程的文件夹。

2）新建STM32CubeMX工程

如图5-7所示，在STM32CubeMX开发环境中执行File→New Project菜单命令或通过STM32CubeMX启动界面中的New Project提示窗口新建工程。

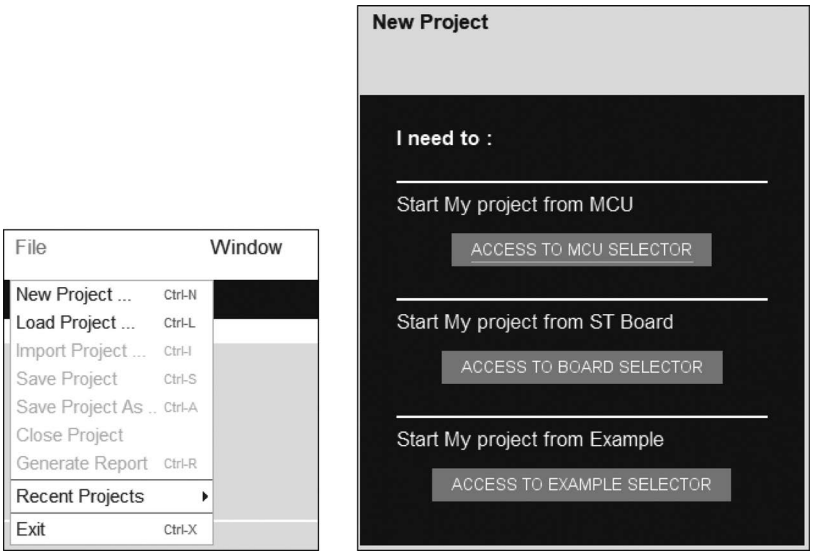

图5-7 STM32CubeMX新建工程

3）选择MCU或开发板

以MCU为例，Commercial Part Number选择STM32F407ZGT6，如图5-8所示。

单击Start Project按钮，启动工程后的界面如图5-9所示。

4）保存STM32Cube MX工程

执行File→Save Project菜单命令，保存工程到LED文件夹，如图5-10所示。生成的STM32CubeMX文件为LED.ioc。

此处直接配置工程名称和保存位置，后续生成的工程应用结构（Application Structure）为Advanced模式，即Inc、Src文件夹存放于Core文件夹下，如图5-11所示。

5）生成工程报告

执行File→Generate Report菜单命令生成当前工程的报告文件LED.pdf，如图5-12所示。

6）配置MCU时钟树

在Pinout & Configuration工作页面下，选择System Core→RCC，根据开发板实际情况，High Speed Clock（HSE）选择为Crystal/Ceramic Resonator（晶体/陶瓷晶振），如图5-13所示。

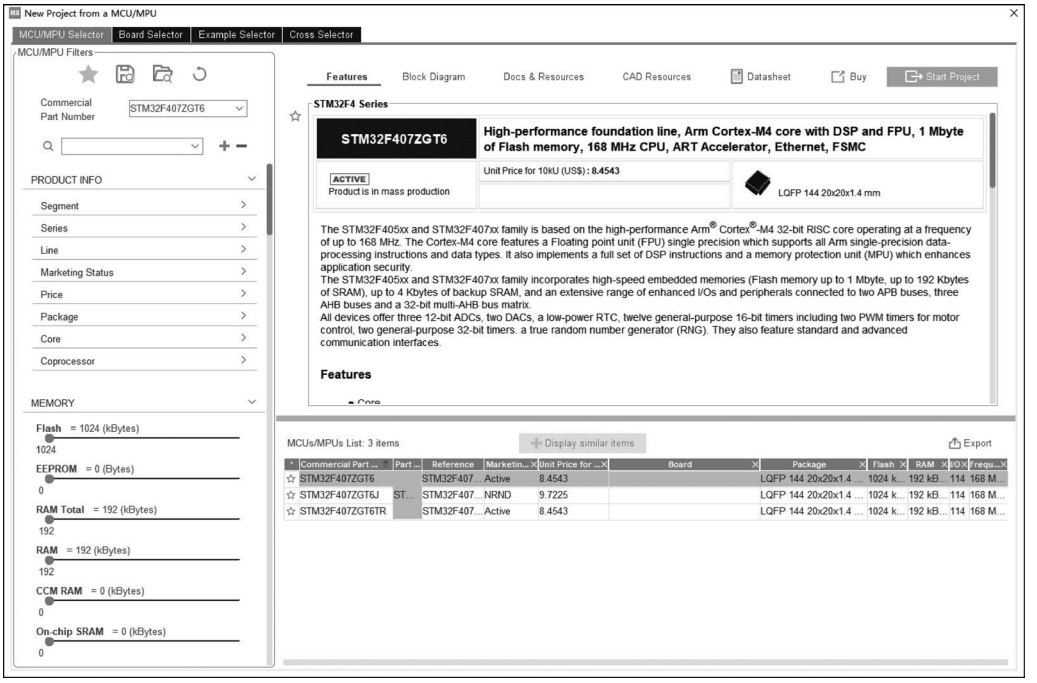

图 5-8　选择 STM32F407ZGT6

图 5-9　启动工程后的界面

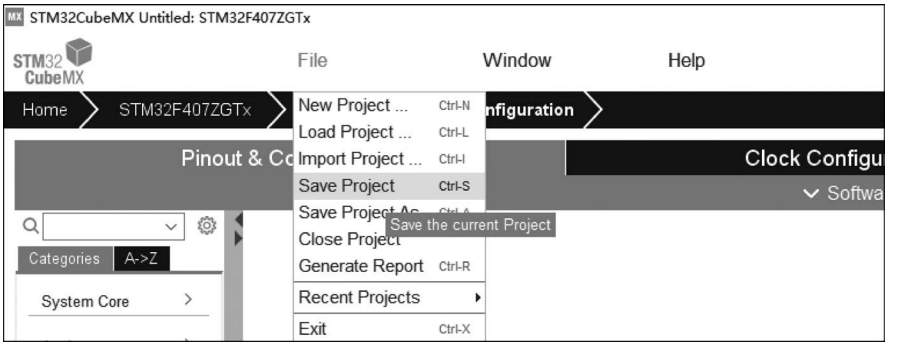

图 5-10 保存工程

图 5-11 工程应用结构

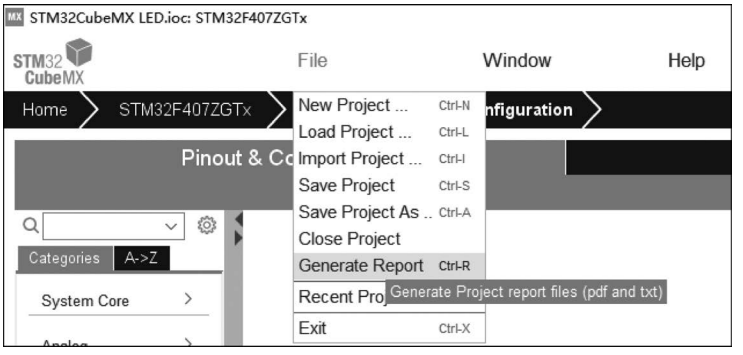

图 5-12 生成工程报告

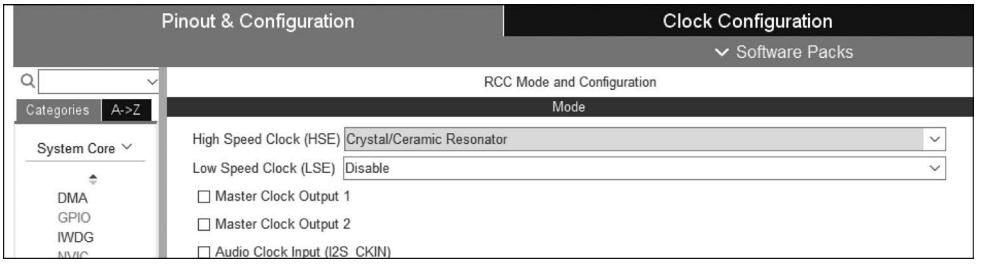

图 5-13 HSE 选择 Crystal/Ceramic Resonator

切换到 Clock Configuration 工作页面。根据开发板外设情况配置总线时钟。此处配置 Input frequency 为 25MHz，PLL Source Mux 为 HSE，分频系数为 25，PLLMul 倍频为 336MHz，PLLCLK 2 分频后为 168MHz，System Clock Mux 为 PLLCLK，APB1 Prescaler 为/4，APB2 Prescaler 为/2，其余默认设置即可。配置完成的时钟树如图 5-14 所示。

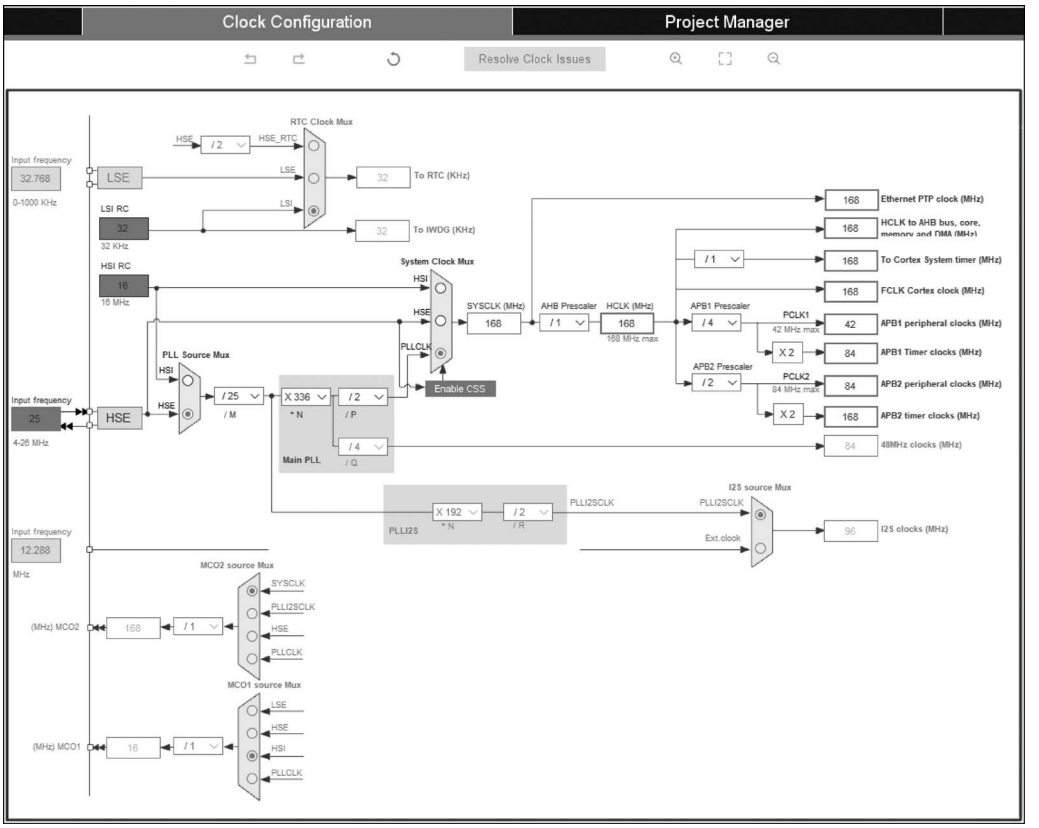

图 5-14　配置完成的时钟树

7）配置 MCU 外设

根据 LED 电路，整理出 MCU 连接的 GPIO 引脚的输入/输出配置，如表 5-2 所示。

表 5-2　MCU 引脚配置

用户标签	引脚名称	引脚功能	GPIO 模式	上拉或下拉	端口速率
LED1_RED	PF6	GPIO_Output	推挽输出	上拉	高
LED2_GREEN	PF7	GPIO_Output	推挽输出	上拉	高
LED3_BLUE	PF8	GPIO_Output	推挽输出	上拉	高

根据表 5-2 进行 GPIO 引脚配置。在引脚视图上，单击相应的引脚，在弹出的菜单中选择引脚功能。与 LED 连接的引脚是输出引脚，设置引脚功能为 GPIO_Output，具体步骤如下。

在 Pinout & Configuration 工作页面选择 System Core→GPIO，此时可以看到与 RCC 相关的两个 GPIO 已自动配置完成，如图 5-15 所示。

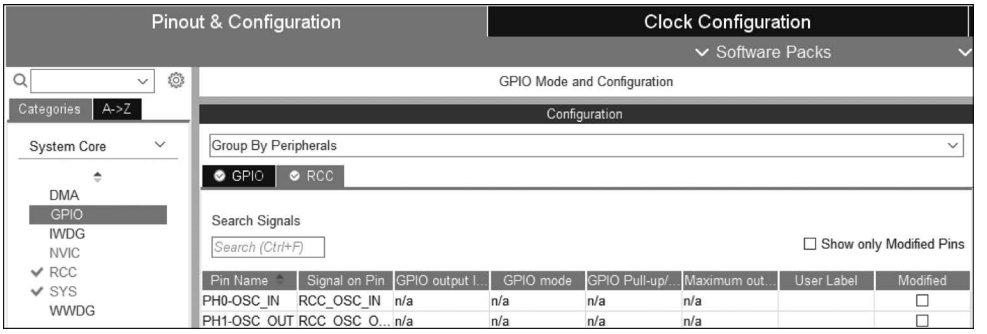

图 5-15 RCC 相关 GPIO 配置

以控制红色 LED 的 PF6 引脚为例，通过搜索框搜索可以定位 I/O 的引脚位置或在引脚视图上选择 PF6，视图中会闪烁显示，配置 PF6 引脚属性为 GPIO_Output，如图 5-16 所示。

图 5-16 配置 PF6 引脚属性为 GPIO_Output

在GPIO组件的模式和配置(GPIO Mode and Configuration)界面对每个GPIO引脚进行更多的设置。例如,GPIO输入引脚是上拉还是下拉,GPIO输出引脚是推挽输出还是开漏输出,按照表5-2的内容设置引脚的用户标签。所有设置是通过下拉列表选择的。GPIO输出引脚的最高输出速率指的是引脚输出变化的最高频率。初始输出设置根据电路功能确定,此工程LED默认输出高电平,即灯不亮状态。具体步骤如下。

如图5-17所示,配置PF6引脚属性,GPIO output level选择High,GPIO mode选择Output Push Pull,GPIO Pull-up/Pull-down选择Pull-up,Maximum output speed选择High,User Label定义为LED1_RED。

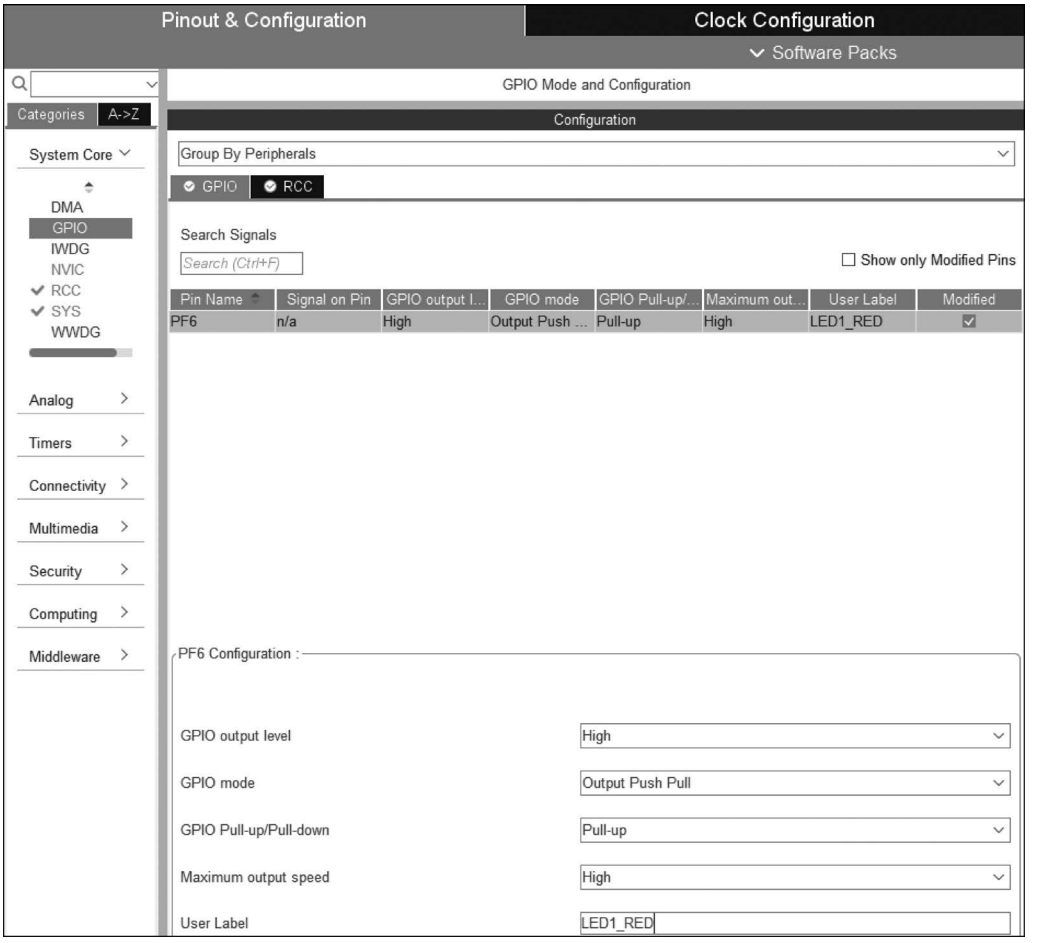

图5-17 PF6引脚配置

用同样方法配置LED2_GREEN(PF7)和LED3_BLUE(PF8)。

我们为引脚设置了用户标签,在生成代码时,STM32CubeMX会在main.h文件中为这些引脚定义宏定义符号,然后在GPIO初始化函数中使用这些符号。

配置完成后的GPIO引脚如图5-18所示。

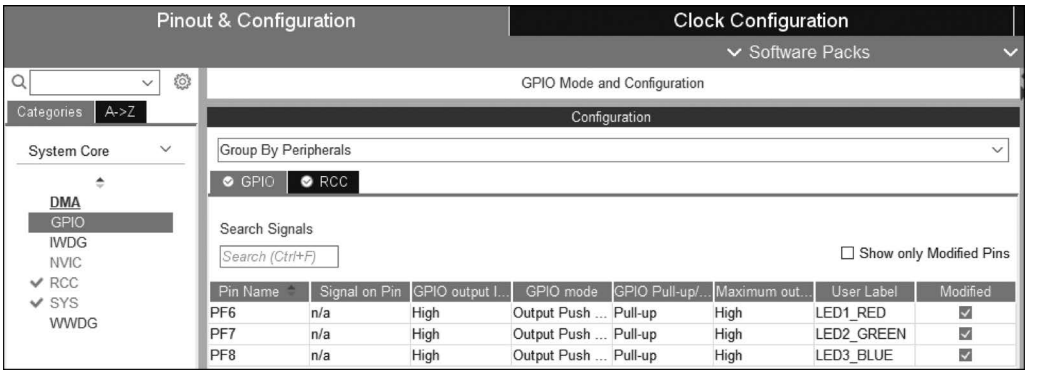

图 5-18 配置完成后的 GPIO 引脚

8）配置工程

在 Project Manager 工作页面 Project 栏，选择 Toolchain/IDE 为 MDK-ARM，Min Version 选择 V5，可生成 Keil MDK 工程；选择 STM32CubeIDE，可生成 STM32CubeIDE 工程。其余配置保持默认即可，如图 5-19 所示。

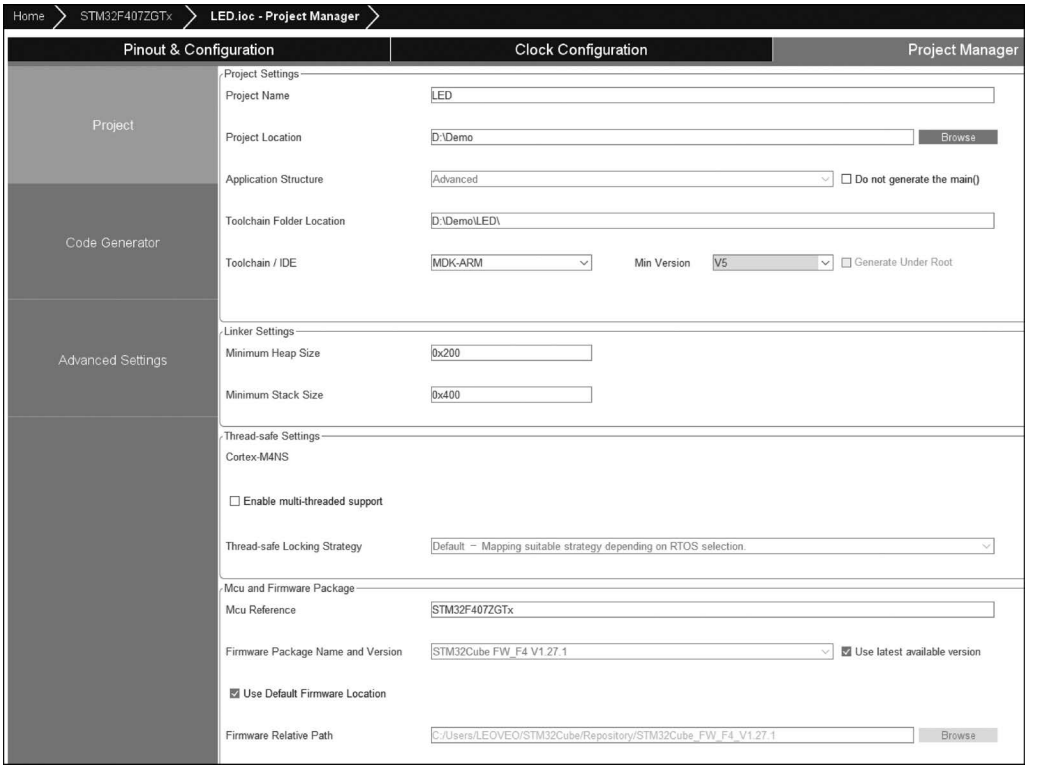

图 5-19 配置工程

若前面已经保存过工程，生成的工程应用结构默认为 Advanced 模式，此处不可再次修改；若前面未保存过工程，此处可修改工程名，存放位置等信息，生成的工程应用结构为 Basic 模式，即 Inc、Src 为单独的文件夹，不存放于 Core 文件夹内，如图 5-20 所示。

图 5-20　Basic 模式工程应用结构

在 Project Manager 工作页面 Code Generator 栏，按图 5-21 勾选 Generated files 选项。

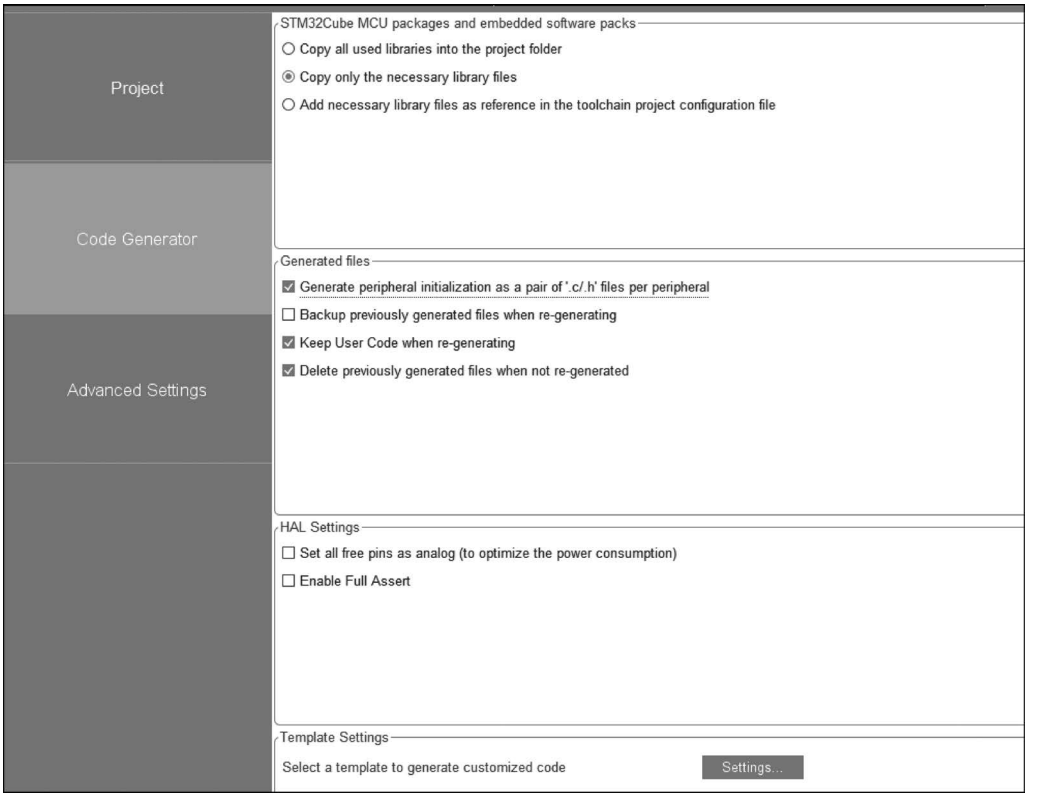

图 5-21　Generated files 配置

9）生成 C 代码工程

返回 STM32CubeMX 主页面，单击 GENERATE CODE 按钮生成 C 代码工程，分别生成 MDK-ARM 和 STM32CubeIDE 工程。

2. 通过 Keil MDK 实现工程

通过 Keil MDK 实现工程的步骤如下。

1）打开工程

打开 LED/MDK-ARM 文件夹下的工程文件 LED.uvprojx，如图 5-22 所示。

图 5-22 MDK-ARM 文件夹内容

2）编译 STM32CubeMX 自动生成的 MDK 工程

在 MDK 开发环境中执行 Project→Rebuild all target files 菜单命令或单击工具栏 Rebuild 按钮 编译工程。

3）STM32CubeMX 自动生成的 MDK 工程

main.c 文件中 main()函数依次调用了以下 3 个函数。

（1）HAL_Init()函数：HAL 库的初始化函数，用于复位所有外设、初始化 Flash 接口和 SysTick 定时器。HAL_Init()函数是在 stm32f4xx_hal.c 文件中定义的函数，它的代码调用了 MSP 函数 HAL_MspInit()，用于对具体 MCU 进行初始化处理。HAL_MspInit()函数在项目的用户程序文件 stm32f4xx_hal_msp.c 中重新实现，实现的代码举例如下，功能是开启各个时钟系统。

```
void HAL_MspInit(void)
{
    __HAL_RCC_SYSCFG_CLK_ENABLE();
    __HAL_RCC_PWR_CLK_ENABLE();
    /* System interrupt init */
}
```

（2）SystemClock_Config()函数：在 main.c 文件中定义和实现，它是根据 STM32CubeMX 里的 RCC 和时钟树的配置自动生成的代码，用于配置各种时钟信号频率。

```
void SystemClock_Config(void)
{
    RCC_OscInitTypeDef RCC_OscInitStruct = {0};
    RCC_ClkInitTypeDef RCC_ClkInitStruct = {0};

    /** Configure the main internal regulator output voltage
    */
    __HAL_RCC_PWR_CLK_ENABLE();
    __HAL_PWR_VOLTAGESCALING_CONFIG(PWR_REGULATOR_VOLTAGE_SCALE1);

    /** Initializes the RCC Oscillators according to the specified parameters
```

```
    * in the RCC_OscInitTypeDef structure
    */
    RCC_OscInitStruct.OscillatorType = RCC_OSCILLATORTYPE_HSI;
    RCC_OscInitStruct.HSIState = RCC_HSI_ON;
    RCC_OscInitStruct.HSICalibrationValue = RCC_HSICALIBRATION_DEFAULT;
    RCC_OscInitStruct.PLL.PLLState = RCC_PLL_NONE;
    if (HAL_RCC_OscConfig(&RCC_OscInitStruct) != HAL_OK)
    {
      Error_Handler();
    }

    /** Initializes the CPU, AHB and APB buses clocks
    */
    RCC_ClkInitStruct.ClockType = RCC_CLOCKTYPE_HCLK|RCC_CLOCKTYPE_SYSCLK
                                  |RCC_CLOCKTYPE_PCLK1|RCC_CLOCKTYPE_PCLK2;
    RCC_ClkInitStruct.SYSCLKSource = RCC_SYSCLKSOURCE_HSI;
    RCC_ClkInitStruct.AHBCLKDivider = RCC_SYSCLK_DIV1;
    RCC_ClkInitStruct.APB1CLKDivider = RCC_HCLK_DIV1;
    RCC_ClkInitStruct.APB2CLKDivider = RCC_HCLK_DIV1;

    if (HAL_RCC_ClockConfig(&RCC_ClkInitStruct, Flash_LATENCY_0) != HAL_OK)
    {
      Error_Handler();
    }
}
```

(3) GPIO 端口初始化函数 MX_GPIO_Init():在 gpio.h 文件中定义的 GPIO 引脚初始化函数,它是 STM32CubeMX 中 GPIO 引脚图形化配置的实现代码。

在 main()函数中,HAL_Init()和 SystemClock_Config()是必然调用的两个函数,再根据使用的外设情况,调用各个外设的初始化函数,然后进入 while 死循环。

在 STM32CubeMX 中,为 LED 连接的 GPIO 引脚设置了用户标签,这些用户标签的宏定义在 main.h 文件中。代码如下。

```
/* Private defines -----------------------------------------------------------*/
#define LED1_RED_Pin GPIO_PIN_6
#define LED1_RED_GPIO_Port GPIOF
#define LED2_GREEN_Pin GPIO_PIN_7
#define LED2_GREEN_GPIO_Port GPIOF
#define LED3_BLUE_Pin GPIO_PIN_8
#define LED3_BLUE_GPIO_Port GPIOF
/* USER CODE BEGIN Private defines */
```

在 STM32CubeMX 中设置的一个 GPIO 引脚用户标签,会在此生成两个宏定义,分别是端口宏定义和引脚号宏定义,如 PF6 引脚设置的用户标签为 LED1_RED,就生成了 LED1_RED_Pin 和 LED1_RED_GPIO_Port 两个宏定义。

GPIO 引脚初始化文件 gpio.c 和 gpio.h 是 STM32CubeMX 生成代码时自动生成的用

户程序文件。注意，必须在 STM32CubeMX Project Manager 工作界面 Code Generator 栏中勾选“生成. c/. h 文件对”选项，才会为一个外设生成. c/. h 文件对，如图 5-23 所示。

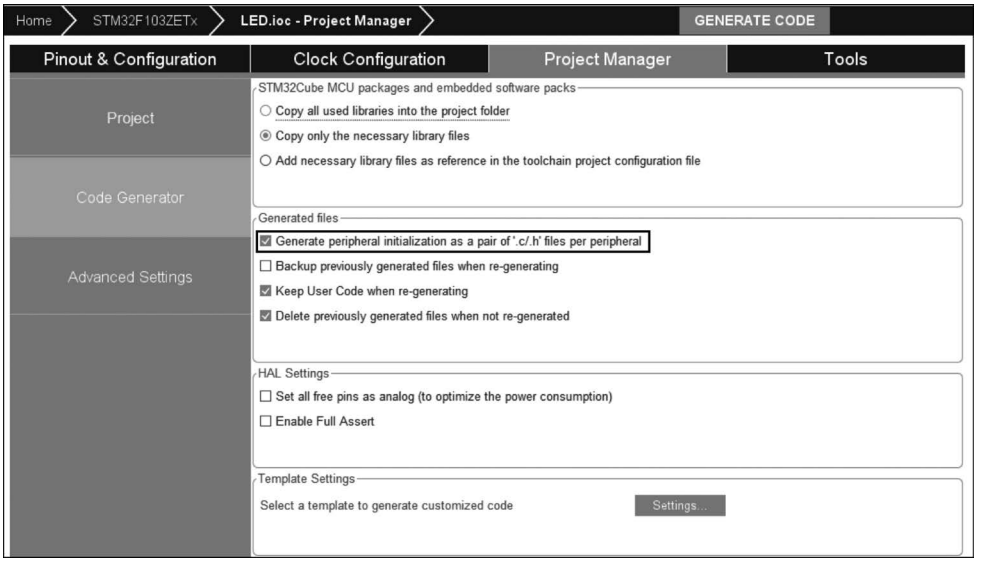

图 5-23 生成. c/. h 文件对配置项

gpio. h 文件定义了一个 MX_GPIO_Init()函数，这是在 STM32CubeMX 中图形化设置的 GPIO 引脚的初始化函数。

gpio. h 文件的代码如下，定义了 MX_GPIO_Init()函数原型。

```
#include "main.h"
void MX_GPIO_Init(void);
```

gpio. c 文件包含了 MX_GPIO_Init()函数的实现代码，具体如下。

```
#include "gpio.h"
void MX_GPIO_Init(void)
{
    GPIO_InitTypeDef GPIO_InitStruct = {0};
    /* GPIO Ports Clock Enable */
    __HAL_RCC_GPIOF_CLK_ENABLE();
    __HAL_RCC_GPIOH_CLK_ENABLE();
    /* Configure GPIO pin Output Level */
    HAL_GPIO_WritePin(GPIOF, LED1_RED_Pin|LED2_GREEN_Pin|LED3_BLUE_Pin, GPIO_PIN_SET);
    /* Configure GPIO pins : PFPin PFPin PFPin */
    GPIO_InitStruct.Pin = LED1_RED_Pin|LED2_GREEN_Pin|LED3_BLUE_Pin;
    GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
    GPIO_InitStruct.Pull = GPIO_PULLUP;
    GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
    HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);
}
```

GPIO 引脚初始化需要开启引脚所在端口的时钟，然后使用一个 GPIO_InitTypeDef 结构体变量设置引脚的各种 GPIO 参数，再调用 HAL_GPIO_Init()函数进行 GPIO 引脚初始化配置。使用 HAL_GPIO_Init()函数可以对一个端口的多个相同配置的引脚进行初始化，而不同端口或不同功能的引脚需要分别调用 HAL_GPIO_Init()函数进行初始化。在 MX_GPIO_Init()函数的代码中，使用了 main.h 文件中为各个 GPIO 引脚定义的宏。这样编写代码的好处是程序可以很方便地移植到其他开发板上。

4）新建用户文件

在 LED/Core/Src 文件夹下新建 bsp_led.c 文件，在 LED/Core/Inc 文件夹下新建 bsp_led.h 文件。将 bsp_led.c 文件添加到 Application/User/Core 文件夹下，如图 5-24 所示。

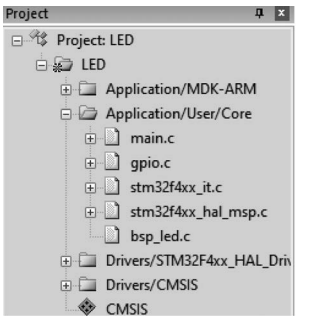

图 5-24　添加文件到 MDK 工程

5）编写用户代码

如果用户想在生成的初始项目的基础上添加自己的应用程序代码，只需把用户代码写在代码沙箱段内，就可以在 STM32CubeMX 中修改 MCU 设置，重新生成代码，而不会影响用户已经添加的程序代码。沙箱段一般以 USER CODE BEGIN 和 USER CODE END 标识。此外，用户自定义的文件不受 STM32CubeMX 生成代码影响。

```
/* USER CODE BEGIN */
用户自定义代码
/* USER CODE END */
```

为了方便控制 LED，把 LED 常用的亮、灭及状态翻转的控制也直接定义成宏，定义在 bsp_led.h 文件中。

```
/** 控制 LED 灯亮灭的宏
  * LED 低电平亮，设置 ON = 0，OFF = 1
  * 若 LED 高电平亮，把宏设置成 ON = 1 ，OFF = 0 即可
  */
#define ON  GPIO_PIN_RESET
#define OFF GPIO_PIN_SET

/* 带参宏，可以像内联函数一样使用 */
#define LED1(a)  HAL_GPIO_WritePin(LED1_GPIO_PORT,LED1_PIN,a)
#define LED2(a)  HAL_GPIO_WritePin(LED2_GPIO_PORT,LED2_PIN,a)
#define LED3(a)  HAL_GPIO_WritePin(LED2_GPIO_PORT,LED3_PIN,a)

/* 直接操作寄存器的方法控制 I/O */
#define digitalHi(p,i)      {p->BSRR = i;}                  //设置为高电平
#define digitalLo(p,i)      {p->BSRR = (uint32_t)i << 16;}  //输出低电平
#define digitalToggle(p,i)  {p->ODR ^= i;}                  //输出翻转状态
```

```
/* 定义控制 I/O 的宏 */
#define LED1_TOGGLE    digitalToggle(LED1_GPIO_PORT,LED1_PIN)
#define LED1_OFF       digitalHi(LED1_GPIO_PORT,LED1_PIN)
#define LED1_ON        digitalLo(LED1_GPIO_PORT,LED1_PIN)

#define LED2_TOGGLE    digitalToggle(LED2_GPIO_PORT,LED2_PIN)
#define LED2_OFF       digitalHi(LED2_GPIO_PORT,LED2_PIN)
#define LED2_ON        digitalLo(LED2_GPIO_PORT,LED2_PIN)

#define LED3_TOGGLE    digitalToggle(LED3_GPIO_PORT,LED3_PIN)
#define LED3_OFF       digitalHi(LED3_GPIO_PORT,LED3_PIN)
#define LED3_ON        digitalLo(LED3_GPIO_PORT,LED3_PIN)

/* 基本混色,后面高级用法使用 PWM 可混出全彩颜色,且效果更好 */

//红
#define LED_RED  \
                    LED1_ON;\
                    LED2_OFF\
                    LED3_OFF
//绿
#define LED_GREEN  \
                    LED1_OFF;\
                    LED2_ON\
                    LED3_OFF
//蓝
#define LED_BLUE\
                    LED1_OFF;\
                    LED2_OFF\
                    LED3_ON
//黄(红 + 绿)
#define LED_YELLOW  \
                    LED1_ON;\
                    LED2_ON\
                    LED3_OFF
//紫(红 + 蓝)
#define LED_PURPLE  \
                    LED1_ON;\
                    LED2_OFF\
                    LED3_ON
//青(绿 + 蓝)
#define LED_CYAN  \
                    LED1_OFF;\
                    LED2_ON\
                    LED3_ON
//白(红 + 绿 + 蓝)
#define LED_WHITE  \
                    LED1_ON;\
                    LED2_ON\
```

```
                    LED3_ON
//黑(全部关闭)
#define LED_RGBOFF  \
                    LED1_OFF;\
                    LED2_OFF\
                    LED3_OFF
```

这部分宏控制 LED 亮灭的操作是通过直接向 BSRR 寄存器写入控制指令实现的，对 BSRR 寄存器低 16 位写 1 输出高电平，对 BSRR 寄存器高 16 位写 1 输出低电平，对 ODR 寄存器某位进行异或操作可翻转位的状态。

利用上面的宏，bsp_led.c 文件实现 LED 的初始化函数 LED_GPIO_Config()。此处仅关闭 RGB 灯，用户可根据需要初始化 RGB 灯的状态。

```
void LED_GPIO_Config(void)
{
    /* 关闭 RGB 灯 */
    LED_RGBOFF;
}
```

在 main.c 文件中添加对 bsp_led.h 文件的引用。

```
/* Private includes ---------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include "bsp_led.h"
/* USER CODE END Includes */
```

在 main()函数中添加对 LED 的控制。调用前面定义的 LED_GPIO_Config()函数初始化 LED，然后直接调用各种控制 LED 灯亮灭的宏实现 LED 灯的控制，延时采用库自带的基于滴答时钟延时函数 HAL_Delay()，单位为 ms，直接调用即可，这里 HAL_Delay(1000)表示延时 1s。

```
int main(void)
{
  /* MCU Configuration--------------------------------------------------------*/
  /* Reset of all peripherals, initializes the Flash interface and the Systick */
  HAL_Init();
  /* Configure the system clock */
  SystemClock_Config();
  /* Initialize all configured peripherals */
  MX_GPIO_Init();
  /* USER CODE BEGIN 2 */
  /* LED 端口初始化 */
  LED_GPIO_Config();
  /* USER CODE END 2 */
  /* Infinite loop */
  /* USER CODE BEGIN WHILE */
```

```
  while (1)
  {
      LED1( ON );                 // 亮
      HAL_Delay(1000);
      LED1( OFF );                // 灭
      HAL_Delay(1000);

      LED2( ON );                 // 亮
      HAL_Delay(1000);
      LED2( OFF );                // 灭

      LED3( ON );                 // 亮
      HAL_Delay(1000);
      LED3( OFF );                // 灭

      /* 轮流显示红绿蓝黄紫青白 */
      LED_RED;
      HAL_Delay(1000);

      LED_GREEN;
      HAL_Delay(1000);

      LED_BLUE;
      HAL_Delay(1000);

      LED_YELLOW;
      HAL_Delay(1000);

      LED_PURPLE;
      HAL_Delay(1000);

      LED_CYAN;
      HAL_Delay(1000);

      LED_WHITE;
      HAL_Delay(1000);

      LED_RGBOFF;
      HAL_Delay(1000);
    /* USER CODE END WHILE */

    /* USER CODE BEGIN 3 */
  }
  /* USER CODE END 3 */
}
```

开发板上的 RGB LED 可以实现混色，最后一段代码控制各种颜色的实现。

6）重新编译工程

重新编译添加代码后的 MDK 工程，如图 5-25 所示。

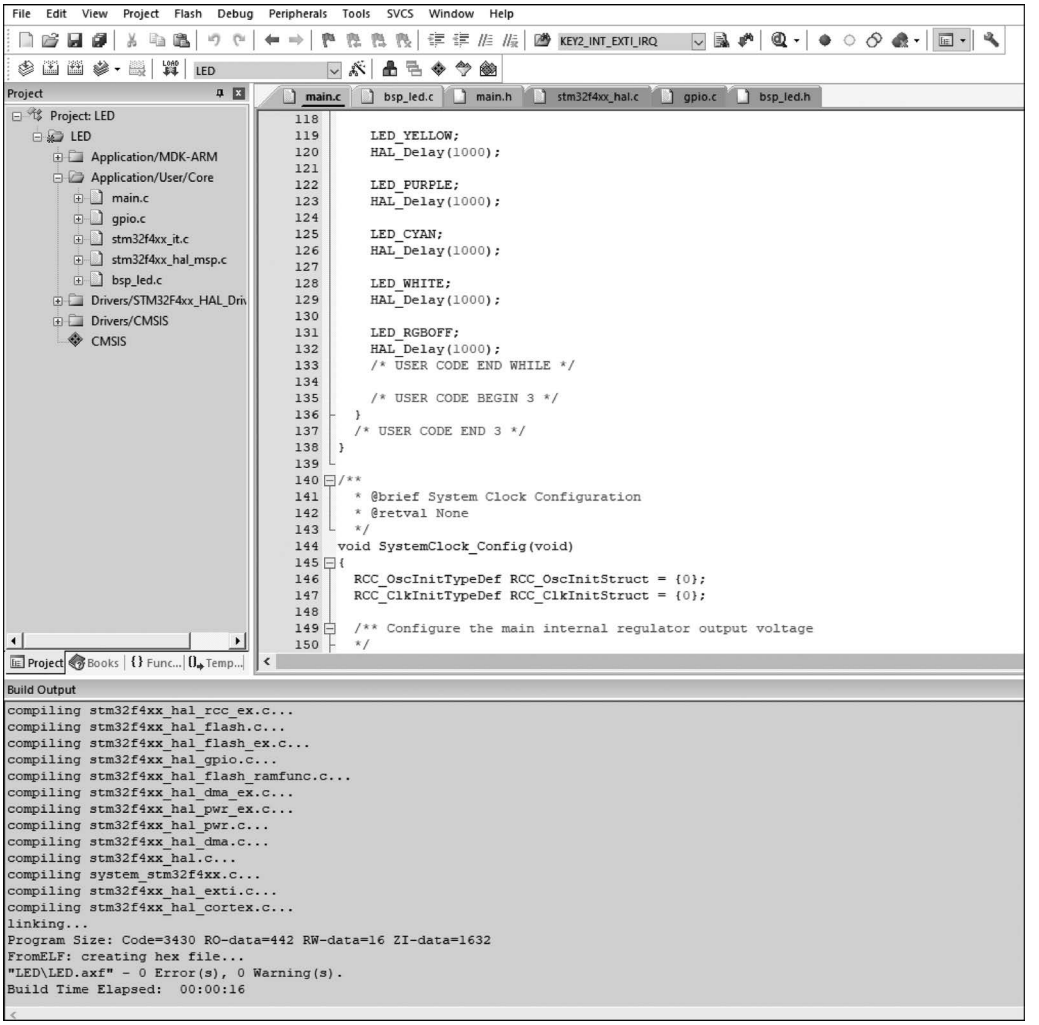

图 5-25　重新编译 MDK 工程

7）配置工程仿真与下载项

在 MDK 开发环境中执行 Project→Options for Target 菜单命令或单击工具栏 按钮配置工程，如图 5-26 所示。

切换至 Debug 选项卡，选择使用的仿真下载器 ST-Link Debugger。配置 Flash Download，勾选 Reset and Run 复选框，单击“确定”按钮，如图 5-27 所示。

8）下载工程

连接好仿真下载器，开发板上电。

图 5-26 配置 MDK 工程

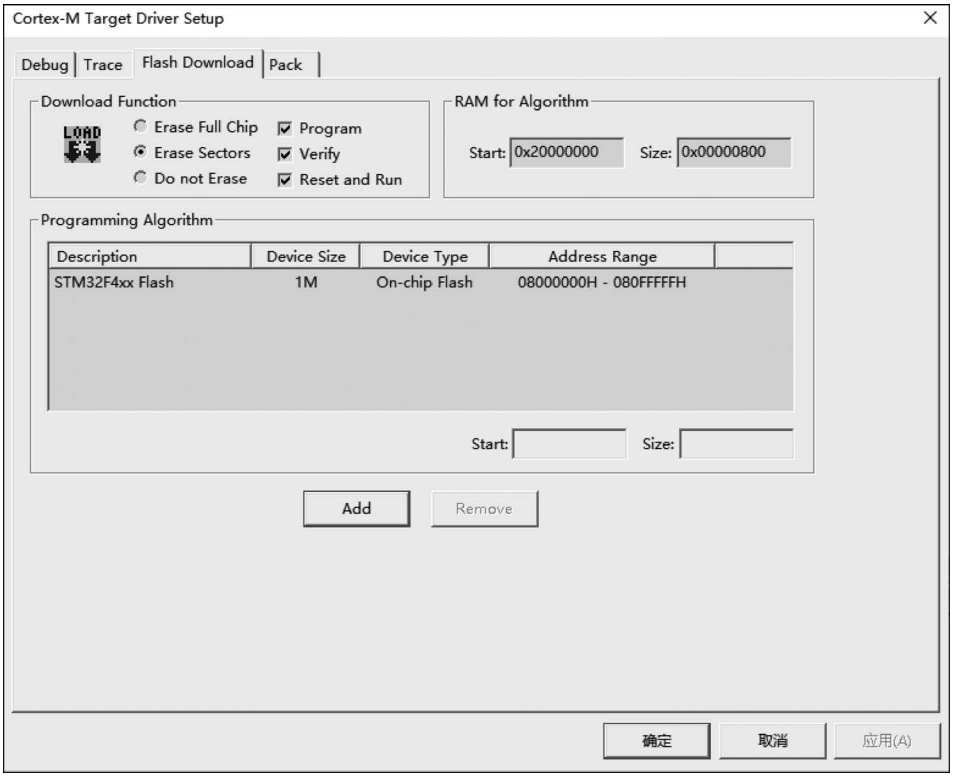

图 5-27 配置 Flash Download 选项

在 MDK 开发环境中执行 Flash→Download 菜单命令或单击工具栏 按钮下载工程，如图 5-28 所示。

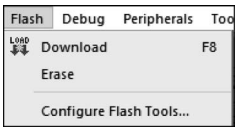

图 5-28　下载工程

工程下载成功提示如图 5-29 所示。

工程下载完成后，观察开发板上 LED 的闪烁状态，RGB 彩灯轮流显示不同的颜色。

```
Build Output
Build started: Project: LED
*** Using Compiler 'V5.06 update 7 (build 960)', folder: 'C:\Keil_v5\ARM\ARMCC\Bin'
Build target 'LED'
"LED\LED.axf" - 0 Error(s), 0 Warning(s).
Build Time Elapsed:  00:00:00
Load "LED\\LED.axf"
Erase Done.
Programming Done.
Verify OK.
Application running ...
Flash Load finished at 11:43:53
```

图 5-29　工程下载成功提示

3. 通过 STM32CubeIDE 实现工程

通过 STM32CubeIDE 实现工程的步骤如下。

1）打开工程

打开 LED/STM32CubeIDE 文件夹下的.project 工程文件，如图 5-30 所示。

图 5-30　STM32CubeIDE 工程文件夹内容

2）编译工程

在 STM32CubeIDE 开发环境中执行 Project→Build All 菜单命令或单击工具栏 Build All 按钮 编译工程。

STM32CubeMX 自动生成的 STM32CubeIDE 工程与其自动生成的 MDK 工程是一样的，可参考前面的代码讲述。

3）新建用户文件

在 LED/Core/Src 文件夹下新建 bsp_led.c 文件，在 LED/Core/Inc 文件夹下新建 bsp_led.h 文件。将 bsp_led.c 文件添加到 Application/User/Core 文件夹下。右击 Core 文件夹，在弹出的快捷菜单中选择 Import→File System，选择路径 LED/Core/Src，勾选 bsp_led.c 文件。作为链接形式勾选 Create links in workspace 复选项，单击 Finish 按钮，如图 5-31 所示。然后添加文件到 STM32CubeIDE 工程，如图 5-32 所示。

图 5-31 导入添加文件

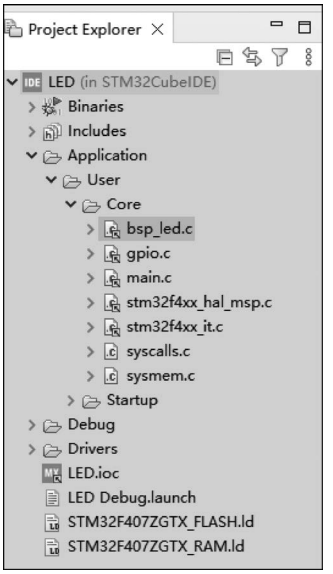

图 5-32 添加文件到 STM32CubeIDE 工程

4）配置文件编码格式

为了防止 STM32CubeIDE 代码编辑器中文显示乱码或串口输出乱码的问题，进行如下操作。

执行 Project→Properties 菜单命令，弹出 Properties for LED 对话框，依次单击 C/C++ Build→Settings→MCU GCC Compiler→Miscellaneous，单击 按钮新建 GCC 编译指令，如图 5-33 所示。

```
-fexec-charset=GBK
-finput-charset=UTF-8
```

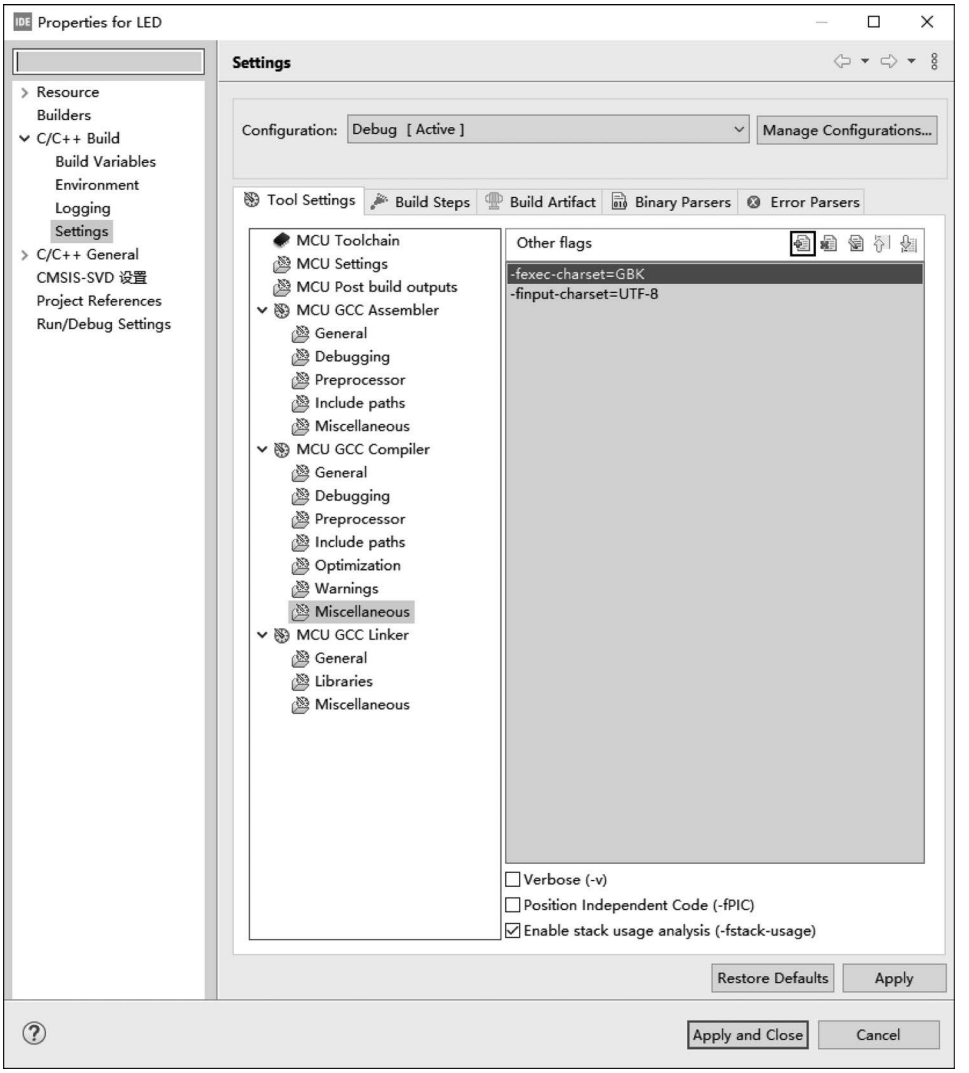

图 5-33　配置编译 GCC 编译指令

执行 Edit→Set Encoding 菜单命令，在 Other 下拉列表中选择 GBK（如果没有，手动输入），如图 5-34 所示。

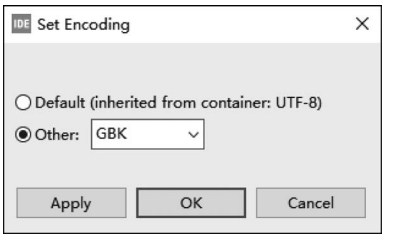

图 5-34 配置编码

当.c 文件中用到中文（非注释部分）时，需要设置编码格式为 UTF-8，且 STM32CubeIDE 重新生成代码后，需注意中文是否乱码。STM32CubeIDE 对中文的支持不友好，当显示乱码时，需进行相应编码格式的切换。

5）编写用户代码

如果用户想在生成的初始项目的基础上添加自己的应用程序代码，只需把用户代码写在代码沙箱段内，就可以在 STM32CubeMX 中修改 MCU 设置，重新生成代码，而不会影响用户已经添加的程序代码。沙箱段一般以 USER CODE BEGIN 和 USER CODE END 标识。此外，用户自定义的文件不受 STM32CubeMX 生成代码影响。可参考通过 Keil MDK 实现工程的编写用户代码。

6）重新编译工程

重新编译添加代码后的 STM32CubeIDE 工程，如图 5-35 所示。

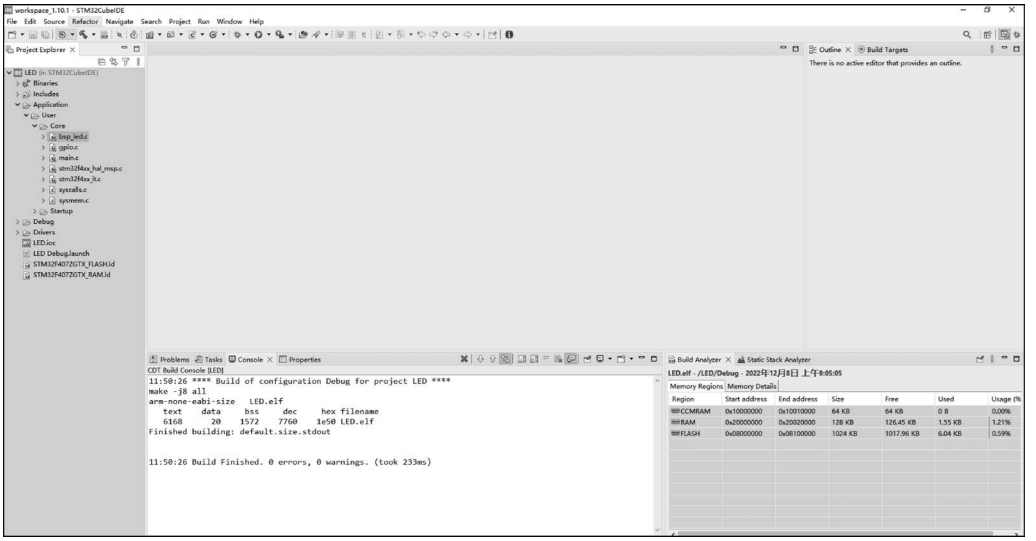

图 5-35 重新编译 STM32CubeIDE 工程

如果在编译 STM32CubeIDE 工程时，出现如图 5-36 所示的路径错误，则执行 Project→Clean 菜单命令，如图 5-37 所示，可以解决编译 STM32CubeIDE 工程时出现的路径错误。

产生这种错误的原因是：如果将在 D 盘建立的 STM32CubeIDE 工程复制到其他盘，如 F 盘，再次编译 STM32CubeIDE 工程会出现路径错误。

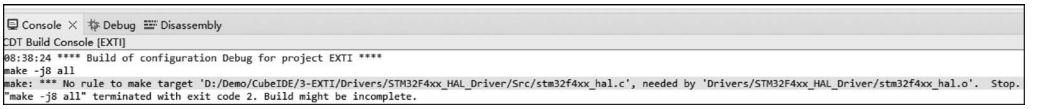

图 5-36　编译 STM32CubeIDE 工程时出现的路径错误

7）下载工程

连接好仿真下载器，开发板上电。

执行 Run→Run 菜单命令或单击工具栏 按钮，首次运行时会弹出配置对话框，选择调试探头为 ST-LINK，接口为 JTAG，其余选项采用默认设置，如图 5-38 所示。

工程下载完成后，观察开发板上 LED 的闪烁状态，RGB 彩灯轮流显示不同的颜色。下载 STM32CubeIDE 工程后提示信息如图 5-39 所示。

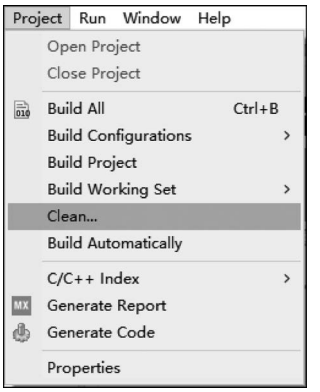

图 5-37　执行 Project→Clean 菜单命令

4. 通过 STM32CubeProgrammer 下载工程

也可以使用 STM32CubeProgrammer 下载工程，步骤如下。

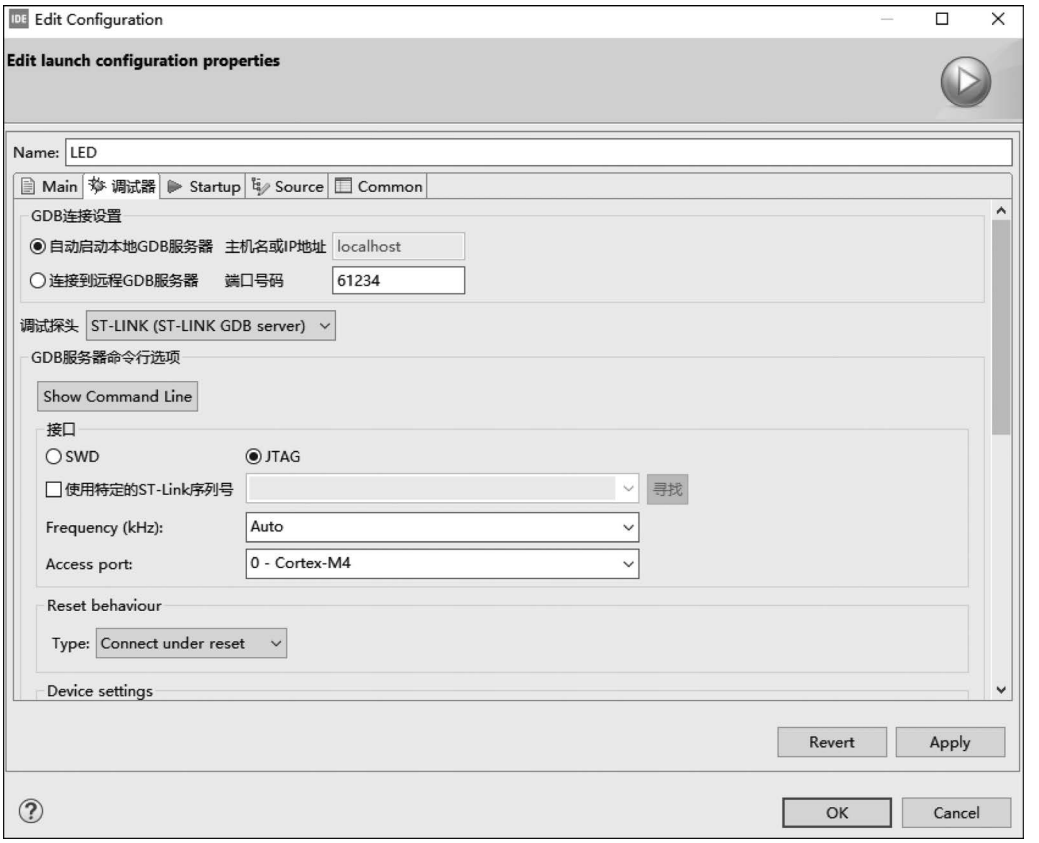

图 5-38　配置 STM32CubeIDE 工程调试器

图 5-39　下载 STM32CubeIDE 工程后提示信息

（1）连接好仿真下载器，开发板上电。

（2）打开 STM32CubeProgrammer，配置工具为 ST-LINK，选择 Port 为 JTAG，如图 5-40 所示。

（3）单击 Connect 按钮，STM32CubeProgrammer 连接 ST-LINK，如图 5-41 所示。

（4）单击左边栏图标进入 Erasing & Programming 页面，单击 Browse 按钮选择 LED/STM32CubeIDE/Debug 文件夹下的 LED.elf 文件，如图 5-42 所示。

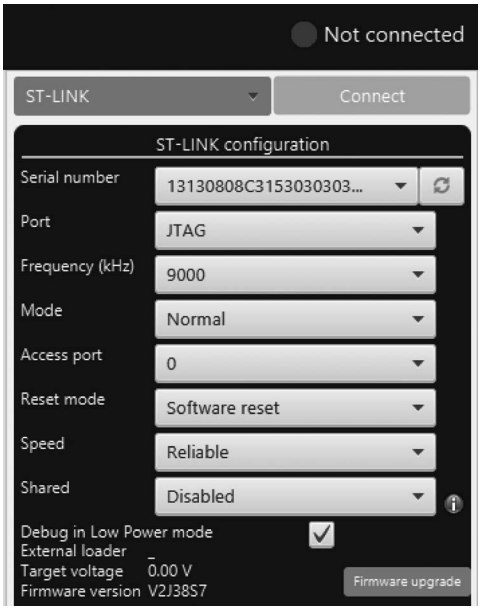

图 5-40　STM32CubeProgrammer 配置 ST-LINK

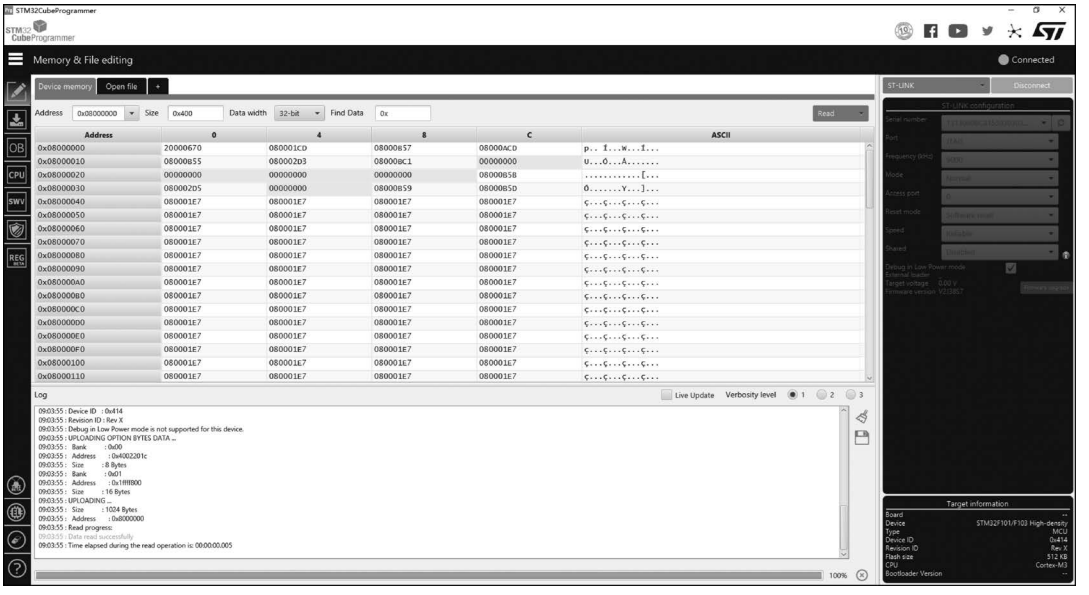

图 5-41　STM32CubeProgrammer 连接 ST-LINK

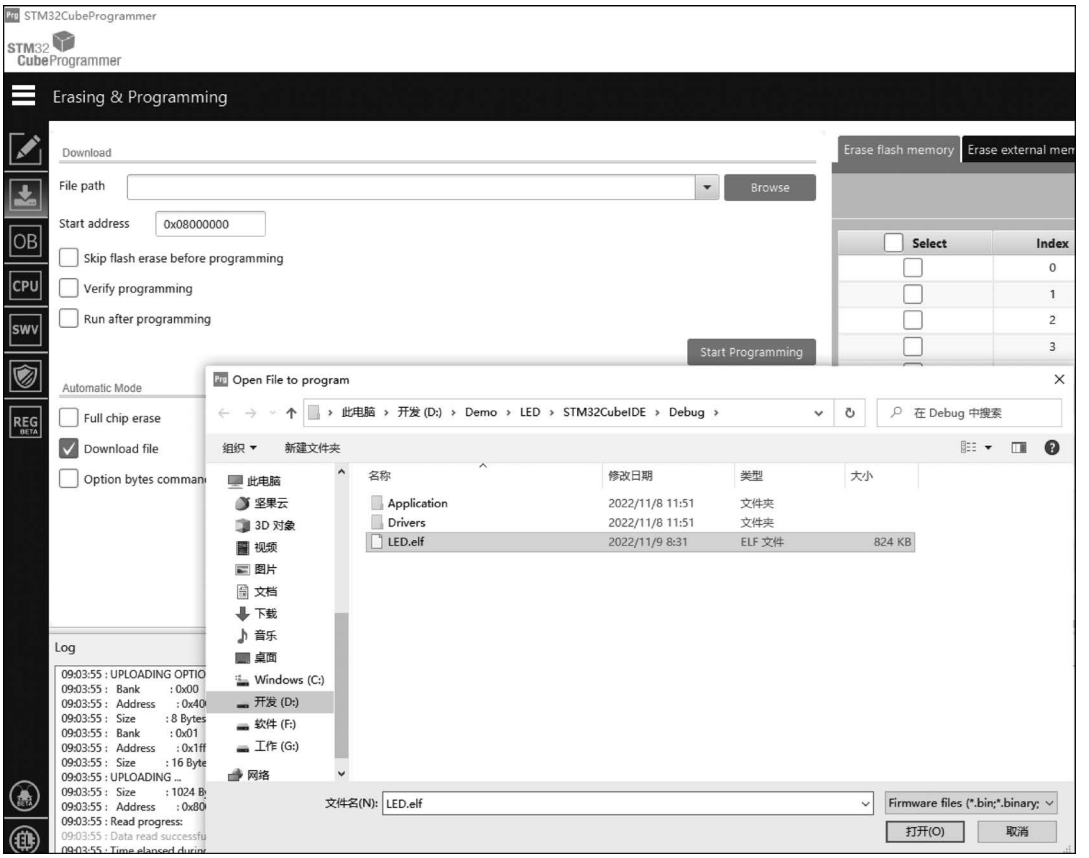

图 5-42　STM32CubeProgrammer 选择下载文件

(5) 勾选 Verify programming 和 Run after programming 复选框，单击 Start Programming 按钮，开始下载工程，如图 5-43 所示。

工程下载成功提示如图 5-44 所示。

工程下载完成后，观察开发板上 LED 的闪烁状态，RGB 彩灯轮流显示不同的颜色。

特别提示：如果 STM32 的外设(如 SPI1)与 JTAG 程序下载接口共用了引脚，则单击 按钮下载工程后执行程序，会出现如图 5-45 所示的“Target is not responding, retrying...”的问题。

产生该问题的原因是 JTAG 引脚与 SPI1 引脚复用了，在下载程序时 JTAG 正常连接，下载完成后程序运行，端口作为 SPI 复用功能，STM32CubeIDE 原先建立的 JTAG 连接失效，因此有对应的“Target is not responding, retrying...”提示。解决的办法是 STM32 的外设(如 SPI1)不与 JTAG 程序下载接口复用。

该问题不影响程序的运行，可以忽略。

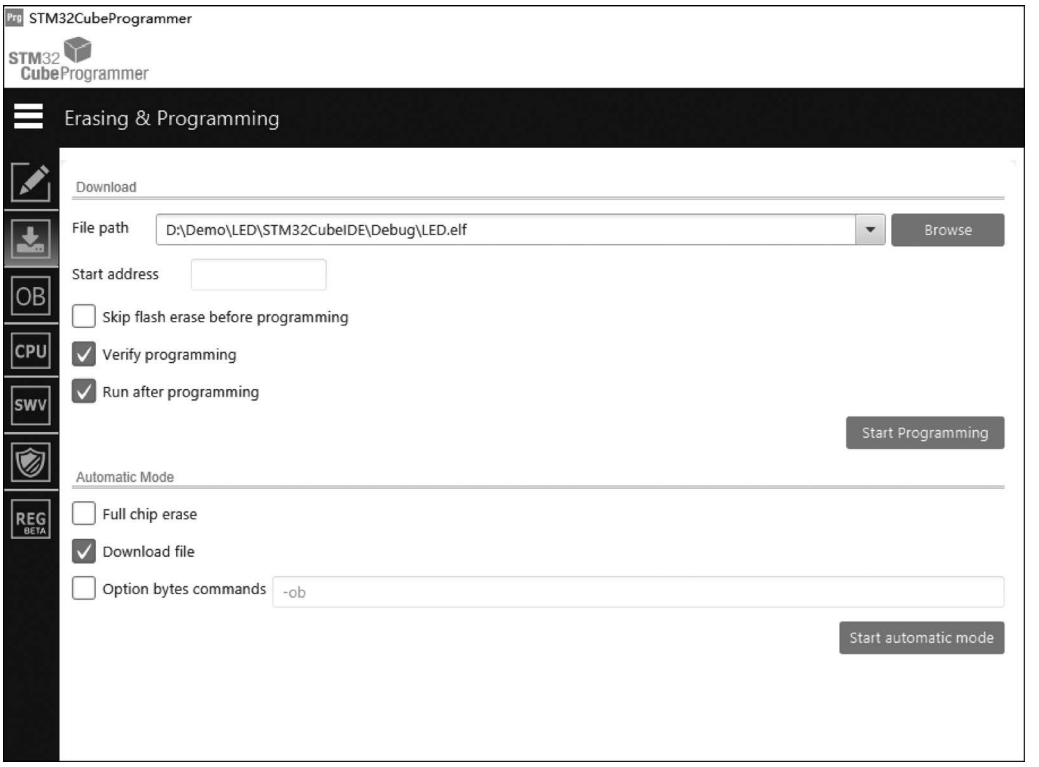

图 5-43　STM32CubeProgrammer 下载工程

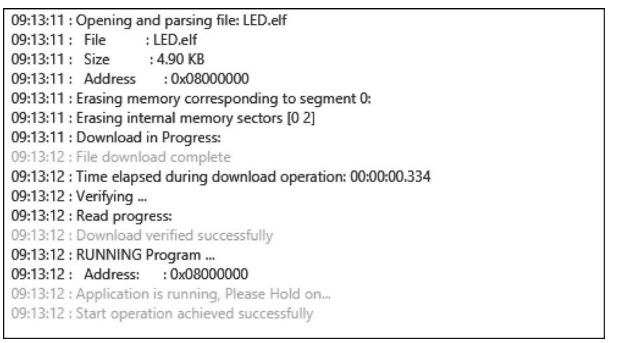

图 5-44　STM32CubeProgrammer 工程下载成功提示

图 5-45　“Target is not responding，retrying…”问题提示

5.6 采用 STM32CubeMX 和 HAL 库的 GPIO 输入应用实例

本 GPIO 输入应用实例是使用固件库的按键检测。

5.6.1 STM32 的 GPIO 输入应用硬件设计

按键机械触点断开、闭合时，由于触点的弹性作用，按键开关不会马上稳定接通或立即断开，使用按键时会产生抖动信号，需要用软件消抖处理滤波，不方便输入检测。本实例开发板连接的按键附带硬件消抖功能，如图 5-46 所示。它利用电容充放电的延时消除了波纹，从而简化软件的处理，软件只需要直接检测引脚的电平即可。

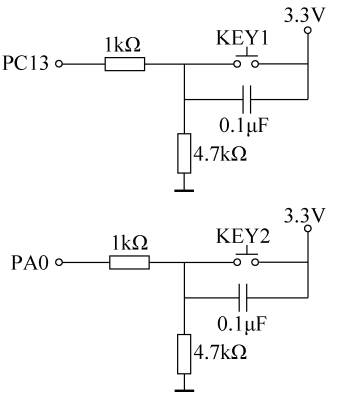

图 5-46 按键检测电路

由按键检测电路可知，当按键没有被按下时，GPIO 引脚的输入状态为低电平（按键所在的电路不通，引脚接地）；当按键按下时，GPIO 引脚的输入状态为高电平（按键所在的电路导通，引脚接到电源）。只要检测按键引脚的输入电平，即可判断按键是否被按下。

若使用的开发板按键的连接方式或引脚不一样，只需根据工程修改引脚即可，程序的控制原理相同。

在本实例中，根据图 5-46 的电路设计一个实例，通过按键控制 LED，功能如下。

（1）按下 KEY1，红灯翻转。

（2）按下 KEY2，绿灯翻转。

5.6.2 STM32 的 GPIO 输入应用软件设计

编程要点如下。

（1）使能 GPIO 端口时钟。

（2）初始化 GPIO 目标引脚为输入模式（浮空输入）。

（3）编写简单测试程序，检测按键的状态，实现按键控制 LED。

1. 通过 STM32CubeMX 新建工程

（1）在 Demo 目录下新建 KEY 文件夹，这是保存本实例新建工程的文件夹。

（2）在 STM32CubeMX 开发环境中新建工程。

（3）选择 MCU 或开发板。选择 STM32F407ZGT6 型号，启动工程。

（4）执行 File→Save Project 菜单命令，保存工程。

（5）执行 File→Generate Report 菜单命令生成当前工程的报告文件。

（6）配置 MCU 时钟树。在 Pinout & Configuration 工作页面，选择 System Core→RCC，根据开发板实际情况，High Speed Clock（HSE）选择为 Crystal/Ceramic Resonator

(晶体/陶瓷晶振)。

切换到 Clock Configuration 工作页面,根据开发板外设情况配置总线时钟。此处配置 Input frequency 为 25MHz,PLL Source Mux 为 HSE,分配系数为 25,PLLMul 倍频为 336MHz,PLLCLK 2 分频后为 168MHz,System Clock Mux 为 PLLCLK,APB1 Prescaler 为/4,APB2 Prescaler 为/2,其余采用默认设置即可。

(7) 配置 MCU 外设

根据 LED 和 KEY 电路,整理出 MCU 连接的 GPIO 引脚的输入/输出配置,如表 5-3 所示。

表 5-3 MCU 引脚的配置

用户标签	引脚名称	引脚功能	GPIO 模式	上拉或下拉	端口速率
LED1_RED	PF6	GPIO_Output	推挽输出	上拉	最高
LED2_GREEN	PF7	GPIO_Output	推挽输出	上拉	最高
LED3_BLUE	PF8	GPIO_Output	推挽输出	上拉	最高
KEY1	PA0	GPIO_Input	浮空输入	无	—
KEY2	PC13	GPIO_Input	浮空输入	无	—

再根据表 5-3 进行 GPIO 引脚配置。在引脚视图上单击相应的引脚,在弹出的菜单中选择引脚功能。与 LED 连接的引脚是输出引脚,设置引脚功能为 GPIO_Output;与 KEY 连接的引脚是输入引脚,设置引脚功能为 GPIO_Input,具体步骤如下。

在 Pinout & Configuration 工作页面选择 System Core→GPIO,对使用的 GPIO 进行设置。LED 输出引脚:LED1_RED(PF6)、LED2_GREEN(PF7)和 LED3_BLUE(PF8);按键输入引脚:KEY1(PA0)和 KEY2(PC13)。配置完成后的 GPIO 引脚页面如图 5-47 所示。

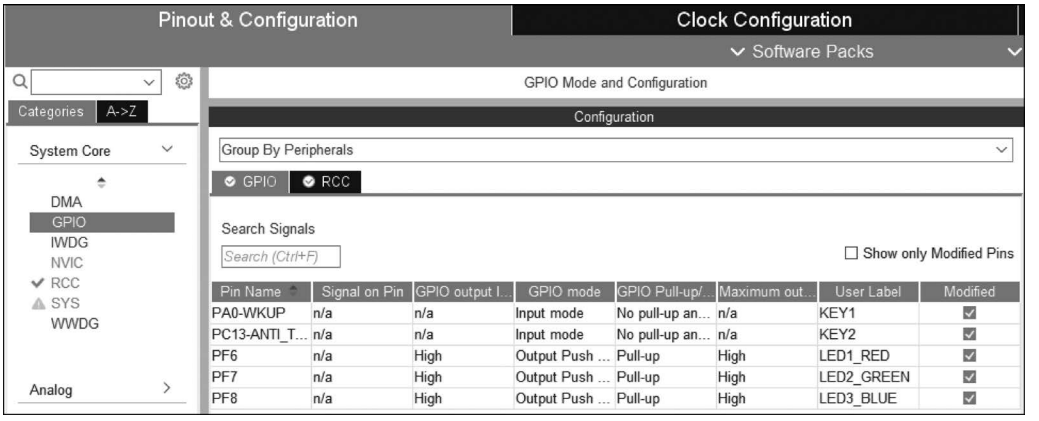

图 5-47 配置完成后的 GPIO 引脚页面

(8) 配置工程。

在 Project Manager 工作页面 Project 栏，Toolchain/IDE 选择为 MDK-ARM，Min Version 选择为 V5，可生成 Keil MDK 工程；选择为 STM32CubeIDE，可生成 STM32CubeIDE 工程。

(9) 生成 C 代码工程。

返回 STM32CubeMX 主页面单击 GENERATE CODE 按钮生成 C 代码工程。

2. 通过 Keil MDK 实现工程

通过 Keil MDK 实现工程的步骤如下。

(1) 打开 KEY/MDK-ARM 文件夹下的工程文件。

(2) 编译 STM32CubeMX 自动生成的 MDK 工程。

在 MDK 开发环境中执行 Project→Rebuild all target files 菜单命令或单击工具栏 Rebuild 按钮 编译工程。

(3) STM32CubeMX 自动生成的 MDK 工程如下。

main.c 文件中 main()函数依次调用了：HAL_Init()函数，用于复位所有外设，初始化 Flash 接口和 SysTick 定时器；SystemClock_Config()函数，用于配置各种时钟信号频率；MX_GPIO_Init()函数，用于初始化 GPIO 引脚。

在 STM32CubeMX 中，为 LED 和按键连接的 GPIO 引脚设置了用户标签，这些用户标签的宏定义在 main.h 文件里。代码如下。

```
/* Private defines -----------------------------------------------------------*/
#define KEY2_Pin GPIO_PIN_13
#define KEY2_GPIO_Port GPIOC
#define LED1_RED_Pin GPIO_PIN_6
#define LED1_RED_GPIO_Port GPIOF
#define LED2_GREEN_Pin GPIO_PIN_7
#define LED2_GREEN_GPIO_Port GPIOF
#define LED3_BLUE_Pin GPIO_PIN_8
#define LED3_BLUE_GPIO_Port GPIOF
#define KEY1_Pin GPIO_PIN_0
#define KEY1_GPIO_Port GPIOA
/* USER CODE BEGIN Private defines */
```

gpio.c 文件包含了 MX_GPIO_Init()函数的实现代码，具体如下。

```
void MX_GPIO_Init(void)
{
  GPIO_InitTypeDef GPIO_InitStruct = {0};
  /* GPIO Ports Clock Enable */
  __HAL_RCC_GPIOC_CLK_ENABLE();
  __HAL_RCC_GPIOF_CLK_ENABLE();
  __HAL_RCC_GPIOH_CLK_ENABLE();
  __HAL_RCC_GPIOA_CLK_ENABLE();
```

```
  /* Configure GPIO pin Output Level */
  HAL_GPIO_WritePin(GPIOF, LED1_RED_Pin|LED2_GREEN_Pin|LED3_BLUE_Pin, GPIO_PIN_SET);

  /* Configure GPIO pin : PtPin */
  GPIO_InitStruct.Pin = KEY2_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(KEY2_GPIO_Port, &GPIO_InitStruct);

  /* Configure GPIO pins : PFPin PFPin PFPin */
  GPIO_InitStruct.Pin = LED1_RED_Pin|LED2_GREEN_Pin|LED3_BLUE_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_OUTPUT_PP;
  GPIO_InitStruct.Pull = GPIO_PULLUP;
  GPIO_InitStruct.Speed = GPIO_SPEED_FREQ_HIGH;
  HAL_GPIO_Init(GPIOF, &GPIO_InitStruct);

  /* Configure GPIO pin : PtPin */
  GPIO_InitStruct.Pin = KEY1_Pin;
  GPIO_InitStruct.Mode = GPIO_MODE_INPUT;
  GPIO_InitStruct.Pull = GPIO_NOPULL;
  HAL_GPIO_Init(KEY1_GPIO_Port, &GPIO_InitStruct);
}
```

(4) 新建用户文件。

在 KEY/Core/Src 文件夹下新建 bsp_led.c、bsp_key.c 文件，在 KEY/Core/Inc 文件夹下新建 bsp_led.h、bsp_key.h 文件。将 bsp_led.c 和 bsp_key.c 文件添加到 Application/User/Core 文件夹下。

(5) 编写用户代码。

bsp_led.h 和 bsp_led.c 文件实现 LED 操作的宏定义和 LED 初始化。

bsp_key.h 文件实现按键检测引脚相关的宏定义。

```
/** 按键按下标志宏
  * 若按键按下为高电平，设置 KEY_ON = 1, KEY_OFF = 0
  * 若按键按下为低电平，把宏设置成 KEY_ON = 0 ，KEY_OFF = 1 即可
  */
#define KEY_ON  1
#define KEY_OFF 0
```

bsp_key.c 文件实现按键扫描函数 Key_Scan()。GPIO 引脚的输入电平可通过读取 IDR 寄存器对应的数据位感知，而 STM32HAL 库提供了库函数 HAL_GPIO_ReadPin()获取位状态，该函数输入 GPIO 端口及引脚号，返回该引脚的电平状态，高电平返回 1，低电平返回 0。Key_Scan()函数中将 HAL_GPIO_ReadPin()函数的返回值与自定义的宏 KEY_ON 进行对比，若检测到按键按下，则使用 while 循环持续检测按键状态，直到按键释放，按键释放后 Key_Scan()函数返回一个 KEY_ON 值；若没有检测到按键按下，则函数直接返

回 KEY_OFF。若按键的硬件没有做消抖处理，需要在这个 Key_Scan()函数中做软件滤波，防止波纹抖动引起误触发。

```
uint8_t Key_Scan(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin)
{
    /* 检测是否有按键按下 */
    if(HAL_GPIO_ReadPin(GPIOx,GPIO_Pin) == KEY_ON )
    {
        /* 等待按键释放 */
        while(HAL_GPIO_ReadPin(GPIOx,GPIO_Pin) == KEY_ON);
        return KEY_ON;
    }
    else
        return KEY_OFF;
}
```

在 main.c 文件中添加对用户自定义头文件的引用。

```
/* Private includes ---------------------------------------------------------*/
/* USER CODE BEGIN Includes */
#include " bsp_led.h"
#include "bsp_key.h"
/* USER CODE END Includes */
```

在 main.c 文件中添加对 LED 的初始化和对按键的控制。KEY1 控制 LED1_RED，KEY2 控制 LED2_GREEN。按一次按键，LED 状态就翻转一次。

```
/* USER CODE BEGIN 2 */
/* LED 端口初始化 */
LED_GPIO_Config();
/* USER CODE END 2 */

/* Infinite loop */
/* USER CODE BEGIN WHILE */
while (1)
{
    if( Key_Scan(KEY1_GPIO_PORT,KEY1_PIN) == KEY_ON  )
    {
        /* LED1 翻转 */
        LED1_TOGGLE;
    }

    if( Key_Scan(KEY2_GPIO_PORT,KEY2_PIN) == KEY_ON  )
    {
        /* LED2 翻转 */
        LED2_TOGGLE;
    }
    /* USER CODE END WHILE */
```

初始化 LED 及按键后，在 while 循环里不断调用 Key_Scan()函数，并判断其返回值，若返回值表示按键按下，则翻转 LED 的状态。

(6) 重新编译添加代码后的工程。

(7) 配置工程仿真与下载项。

在 MDK 开发环境中执行 Project→Options for Target 菜单命令或单击工具栏 按钮配置工程。

切换至 Debug 选项卡中选择使用的仿真下载器 ST-Link Debugger。配置 Flash Download，勾选 Reset and Run 复选框。

(8) 下载工程。

连接好仿真下载器，开发板上电。

在 MDK 开发环境中执行 Flash→Download 菜单命令或单击工具栏 按钮下载工程。

工程下载完成后，操作按键，观察开发板上 LED 的状态。