第 5 章

STM32 GPIO

本章讲述 STM32 GPIO，包括 STM32 GPIO 接口概述、STM32 GPIO 功能、STM32 的 GPIO 常用库函数、STM32 的 GPIO 使用流程、STM32 GPIO 输出应用实例和 STM32 GPIO 输入应用实例。

5.1 STM32 GPIO 接口概述

General Purpose Input Output(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重要技能。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，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。

正是因为 GPIO 作为外设具有无与伦比的重要性，除特殊功能的引脚外，STM32 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、PC 等。每个端口最多有 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 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 7×16＝112 个 GPIO。

绝大多数嵌入式系统应用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,当输入引脚浮空时,相当于该引脚接了高电平。一般实际运用时,不建议引脚浮空,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。该模式适合做电流型的驱动,其吸收电流的能力相对较强(一般在 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(即并非作为通用 I/O 接口使用)。STM32 GPIO 的复用功能推挽模式中输出使能、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 接口被用作第二功能时的配置情况(即并非作为通用 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(IRQ)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 接口的基本结构如图 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。

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

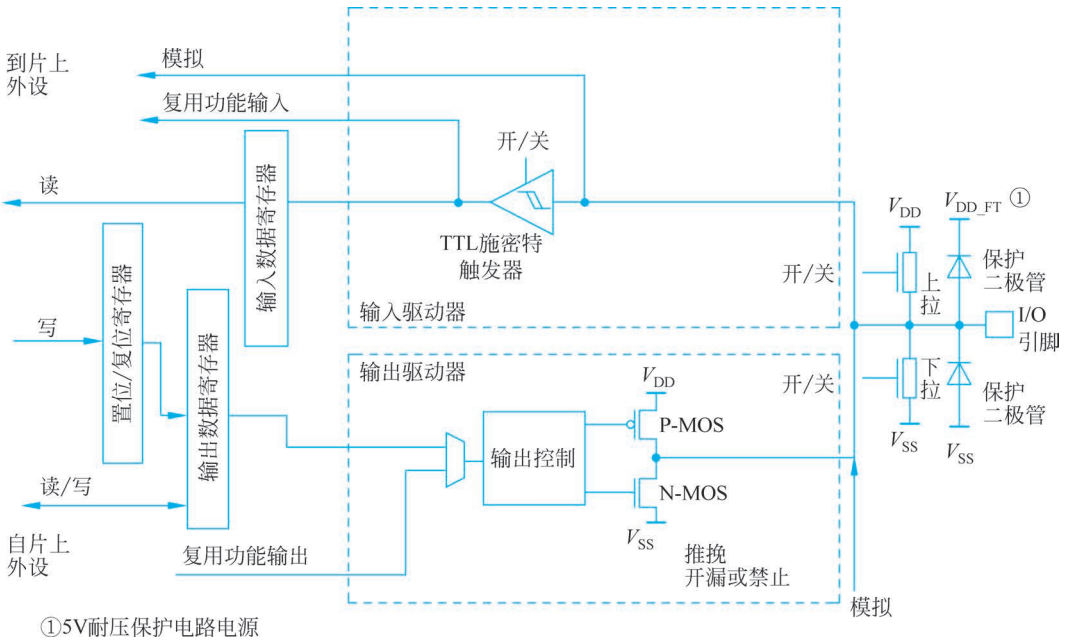

图 5-1　一个 I/O 接口的基本结构

CRL、CRH、IDR、ODR。CRL 和 CRH 寄存器控制每个 I/O 接口的模式及输出速率。

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)或复用的外设功能端口。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 接口的特定硬件特征,GPIO 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。

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最低不会低于−0.7V,最高不会高于 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,两只保护二极管都不会导通,输入驱动器中包括了两只电阻,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 接口位用作输入时,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 接口位用于开关量输入或

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

输出通道包括位设置/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位设置/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、开漏或关闭模式。

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 管组成。

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，一个引脚只能使用这些复用功能中的一个，而这个引脚对应的其他复用功能都处于禁止状态。

输出控制逻辑根据用户设置通过控制 P-MOS 管和 N-MOS 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(输出电压等于 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 0)。

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，适于输出 0 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 微控制器上对应的 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,所以负载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 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 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)上的值输出到相应的 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 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 寄存器的个别位编程时,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(GPIOx_BSRR/GPIOx_BRR)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,端口位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,端口位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,端口位必须配置复用功能输出模式(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 口位被配置为输入时:

(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 口位的输入配置如图 5-2 所示。

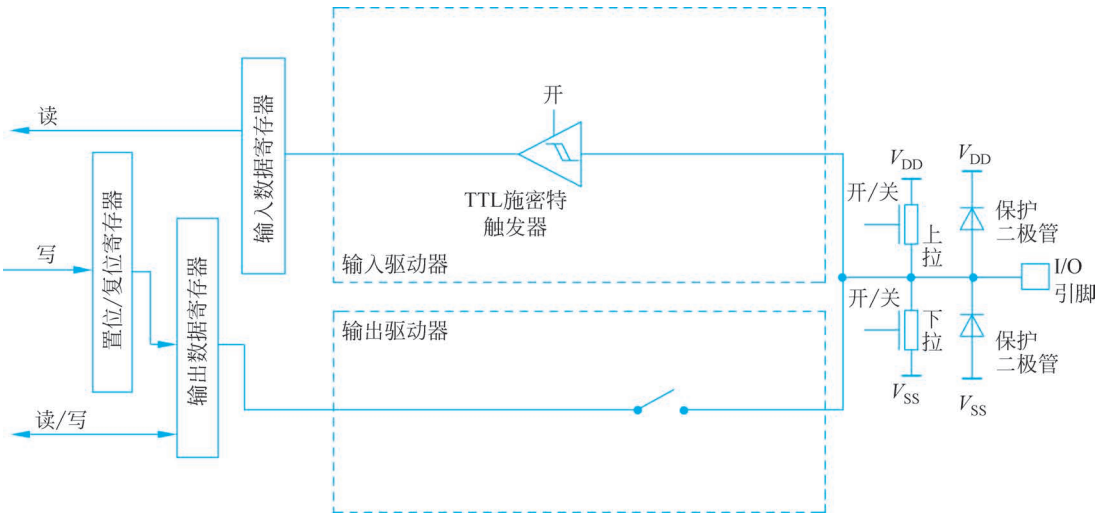

图 5-2　I/O 口位的输入配置

5.2.8　输出配置

当 I/O 口位被配置为输出时：

(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 晶体管，而输出寄存器上的 1 将端口置于高阻状态(P-MOS 晶体管从不被激活)，推挽模式下，输出寄存器上的 0 激活 N-MOS 晶体管，而输出寄存器上的 1 将激活 P-MOS 晶体管；

(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) 在开漏模式时，对输入数据寄存器的读访问可得到 I/O 状态；

(6) 在推挽式模式时，对输出数据寄存器的读访问得到最后一次写的值。

I/O 口位的输出配置如图 5-3 所示。

5.2.9　复用功能配置

当 I/O 口位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 口状态；

(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。

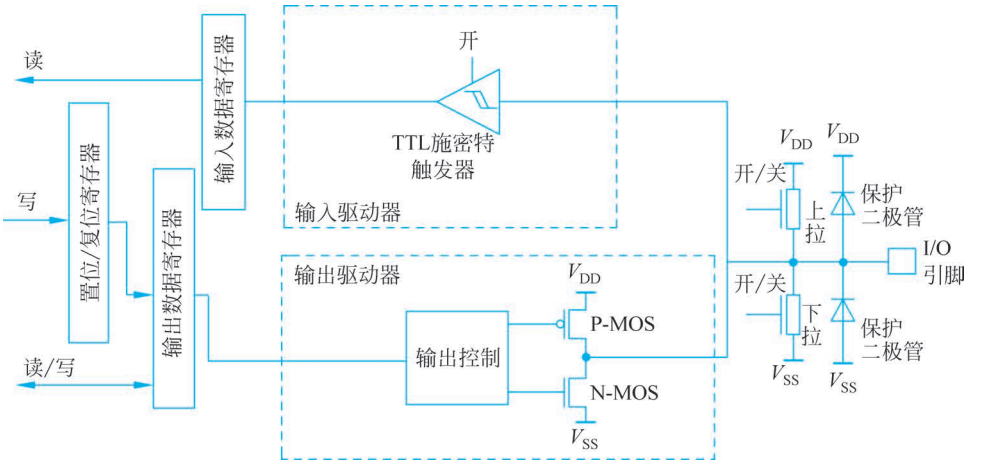

图 5-3 I/O 口位的输出配置

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 口位的复用功能配置如图 5-4 所示。

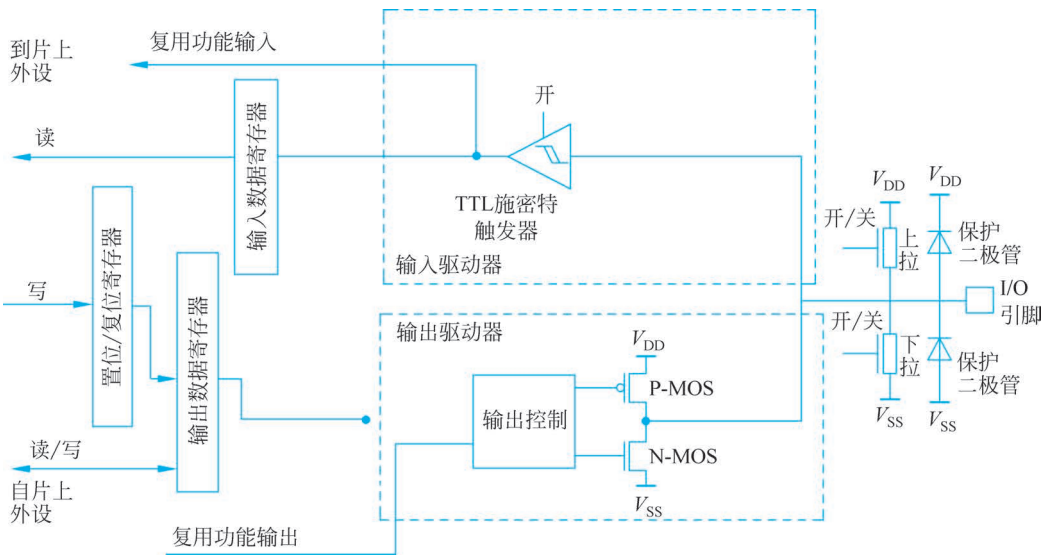

图 5-4 I/O 口位的复用功能配置

5.2.10 高阻抗模拟输入配置

当 I/O 口位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 口位的高阻抗模拟输入配置如图 5-5 所示。

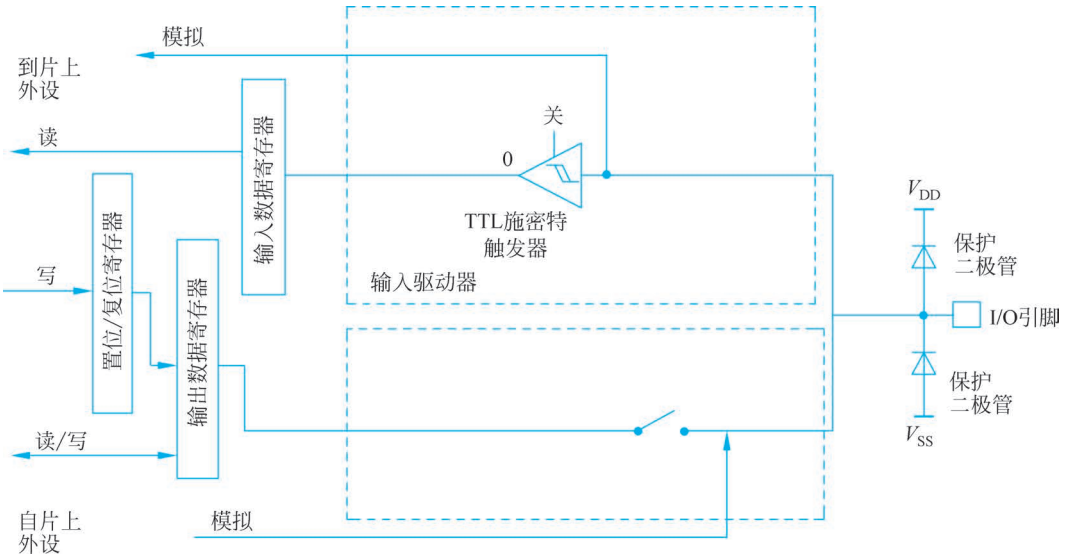

图 5-5　I/O 口位的高阻抗模拟输入配置

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 端口才会给提供时钟。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，其中 y 代表 GPIO 引脚在端口中的编号），用于选择该引脚是处于输入模式中的浮空输入模式、上拉/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 GPIO 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；在推挽模式时，互补 MOS 管对都能被打开；在开漏模式时，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；在推挽模式时，互补 MOS 管对都能被打开；在开漏模式时，只有 N-MOS 晶体管可以被打开。

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。软件可以通过对 GPIOx_IDR 寄存器某个位的直接读取，或对位带别名区中对应字的读取得到 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 位直接对应了本端口的 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-修改-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 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 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响应速度的驱动模块，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，都必须先打开 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时，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 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 从低功耗模式中唤醒。

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 引脚具有最高 84MHz 的翻转速度。30pF 时输出速度为 100MHz，15pF 时输出速度为 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 STM32 的 GPIO 常用库函数

GPIO 相关的函数和宏被定义在以下两个文件中。

(1) 头文件:stm32f4xx_gpio.h。

(2) 源文件:stm32f4xx_gpio.c。

常用库函数有初始化函数、读取输入电平函数、读取输出电平函数、设置输出电平函数、反转引脚状态函数及复用功能设置函数。

1. 初始化函数

该函数用于初始化 GPIO 的一个或多个引脚的工作模式、输出类型、输出速度及上拉/下拉方式。操作的是 4 个配置寄存器(模式寄存器、输出类型寄存器、输出速度寄存器和上拉/下拉寄存器)。语法格式为

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

初始化函数有以下两个参数。

参数 1:GPIO_TypeDef * GPIOx。该参数是操作的 GPIO 对象,是一个结构体指针。在实际使用中的参数有 GPIOA~GPIOI,被定义在头文件 stm32f4xx.h 中。例如:

```
#define GPIOA     ((GPIO_TypeDef * )GPIOA_BASE)
#define GPIOB     ((GPIO_TypeDef * )GPIOB_BASE)
…
#define GPIOI     ((GPIO_TypeDef * )GPIOI_BASE)
```

参数 2:GPIO_InitTypeDef * GPIO_InitStruct。该参数是 GPIO 初始化结构体指针。GPIO_InitTypeDef 结构体类型被定义在头文件 stm32f4xx_gpio.h 中。

```
typedef struct
{
    uint32_t  GPIO_Pin;                          //初始化的引脚
    GPIOMode_TypeDef     GPIO_Mode;              //工作模式
    GPIOSpeed_TypeDef    GPIO_Speed;             //输出速度
    GPIOOType_TypeDef    GPIO_OType;             //输出类型
    GPIOPuPd_TypeDef     GPIO_PuPd;              //上拉/下拉
}GPIO_InitTypeDef;
```

成员 uint32_t GPIO_Pin 声明需要初始化的引脚,以屏蔽字的形式出现,在头文件

stm32f4xx_gpio.h 中有如下定义。

```
#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_15     ((uint16_t)0x8000)        //Pin 15 selected
#define GPIO_Pin_All    ((uint16_t)0xFFFF)        //All pins selected
```

在实际编程中，当一个 GPIO 的多个引脚被初始化为相同工作模式时，可以通过位或操作合并选择多个引脚。

例如，引脚 1、3、5 被初始化为相同工作模式时，通过位或操作合并如下。

```
GPIO_Pin_1|GPIO_Pin_3|GPIO_Pin_5
```

成员 GPIOMode_TypeDef GPIO_Mode 选择 GPIO 引脚工作模式，在头文件 stm32f4xx_gpio.h 中有如下定义。

```
typedef enum
{
    GPIO_Mode_IN = 0x00,                    //输入模式
    GPIO_Mode_OUT = 0x01,                   //输出模式
    GPIO_Mode_AF = 0x02,                    //复用功能模式
    GPIO_Mode_AN = 0x03                     //模拟功能模式
}GPIOMode_TypeDef;
```

成员 GPIOSpeed_TypeDef GPIO_Speed 选择 GPIO 引脚输出速度，在头文件 stm32f4xx_gpio.h 中有如下定义。

```
typedef enum
{
    GPIO_Low_Speed = 0x00,                  //低速
    GPIO_Medium_Speed = 0x01,               //中速
    GPIO_Fast_Speed = 0x02,                 //快速
    GPIO_High_Speed = 0x03                  //高速
}GPIOSpeed_TypeDef;
#define  GPIO_Speed_2MHz    GPIO_Low_Speed
#define  GPIO_Speed_25MHz   GPIO_Medium_Speed
#define  GPIO_Speed_50MHz   GPIO_Fast_Speed
#define  GPIO_Speed_100MHz  GPIO_High_Speed
```

成员 GPIOOType_TypeDef GPIO_OType 选择 GPIO 引脚输出类型，在头文件 stm32f4xx_gpio.h 中有如下定义。

```
typedef enum
{
    GPIO_OType_PP = 0x00,                   //推挽输出
    GPIO_OType_OD = 0x01                    //开漏输出
}GPIOOType_TypeDef;
```

成员 GPIOPuPd_TypeDef GPIO_PuPd 选择 GPIO 引脚上拉/下拉功能，在头文件 stm32f4xx_gpio.h 中有如下定义。

```
typedef enum
{
    GPIO_PuPd_NOPULL = 0x00,          //不上拉/下拉
    GPIO_PuPd_UP = 0x01,              //上拉
    GPIO_PuPd_DOWN = 0x02             //下拉
} GPIOPuPd_TypeDef;
```

例如，将 GPIO 的 PF9、PF10 引脚配置为推挽输出模式，100MHz，使能上拉功能。

```
//定义 GPIO_InitTypeDef 结构体变量
GPIO_InitTypeDef  GPIO_InitStructure;
//使能 GPIOF 时钟
RCC_AHB1PeriphClockCmd (RCC_AHB1Periph_GPIOF, ENABLE);
//GPIO 的 PF9、PF10 引脚初始化设置
GPIO_InitStructure. GPIO_Pin = GPIO_Pin_9|GPIO_Pin_10;     //需要配置的 GPIO 引脚
GPIO_InitStructure. GPIO_Mode = GPIO_Mode_OUT;             //输出模式
GPIO_InitStructure. GPIO_OType = GPIO_OType_PP;            //推挽输出
GPIO_InitStructure. GPIO_Speed = GPIO_Speed_100MHz;        //100MHz
GPIO_InitStructure. GPIO_PuPd = GPIO_PuPd_UP;              //上拉
//调用 GPIO_Init()函数，完成 GPIO 的 PF9、PF10 引脚初始化
GPIO_Init (GPIOF, &GPIO_InitStructure);
```

2. 读取输入电平函数

（1）读取某个 GPIO 的输入电平，实际操作的是输入数据寄存器。语法格式为

```
uint8_t GPIO_ReadInputDataBit (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin);
```

其中，参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象；参数 uint16_t GPIO_Pin 为操作引脚。

例如，读取 GPIO 的 PA5 引脚电平。

```
GPIO_ReadInputDataBit (GPIOA, GPIO_Pin_5);
```

该函数每次只能获取一个引脚状态。

（2）读取某组 GPIO 的输入电平，实际操作的是输入数据寄存器。语法格式为

```
uint16_t GPIO_ReadInputData (GPIO_TypeDef * GPIOx);
```

例如，读取 GPIOA 所有引脚状态。

```
GPIO_ReadInputData (GPIOA);
```

3. 读取输出电平函数

（1）读取某个 GPIO 的输出电平，实际操作的是输出数据寄存器。语法格式为

```
uint8_t GPIO_ReadOutputDataBit (GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin);
```

其中，参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象；参数 uint16_t GPIO_Pin 为操作引脚。

例如，读取 GPIO 的 PA5 引脚的输出状态。

```
GPIO_ReadOutputDataBit (GPIOA, GPIO_Pin_5);
```

（2）读取某组 GPIO 的输出电平，实际操作的是输出数据寄存器。语法格式为

```
uint16_t GPIO_ReadOutputData (GPIO_TypeDef * GPIOx);
```

例如，读取 GPIOA 组中所有引脚输出电平。

```
GPIO_ReadOutputData (GPIOA);
```

以上这两个函数不常用。

4. 设置输出电平函数

（1）设置 GPIO 引脚为高电平（1），实际操作的是置位/复位寄存器的低 16 位（BSRRL）。语法格式为

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

其中，参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象；参数 uint16_t GPIO_Pin 为操作引脚。

例如，设置 GPIO 的 PA3、PA5 引脚为高电平。

```
GPIO_SetBits(GPIOA,GPIO_Pin_3|GPIO_Pin_5);
```

该函数可同时设置多个引脚的状态。

（2）设置 GPIO 引脚为低电平（0），实际操作的是置位/复位寄存器的高 16 位（BSRRH）。语法格式为

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

例如，设置 GPIO 的 PA2、PA4 引脚为低电平。

```
GPIO_ResetBits(GPIOA,GPIO_Pin_2|GPIO_Pin_4);
```

该函数可同时设置多个引脚的状态。

（3）设置某个 GPIO 引脚为特定电平，实际操作的是置位/复位寄存器。语法格式为

```
void GPIO_WriteBit(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin,Bit Action BitVal);
```

其中，参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象；参数 uint16_t GPIO_Pin 为操作引脚；参数 Bit Action BitVal 为引脚状态，取值为 Bit_SET 或 Bit_RESET。

例如，设置 GPIO 的 PA3 引脚为高电平，设置 GPIO 的 PA5 引脚为低电平。

```
GPIO_WriteBit (GPIOA, GPIO_Pin_3, Bit_SET);
GPIO_WriteBit (GPIOA, GPIO_Pin_5, Bit_RESET);
```

该函数只能设置一个引脚状态。

(4) 设置某个 GPIO 所有引脚为特定电平,实际操作的是输出数据寄存器。语法格式为

```
void GPIO_Write(GPIO_TypeDef * GPIOx,uint16_t Port Val);
```

其中,参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象;参数 uint16_t Port Val 为 16 位的无符号数据,据每位对应控制一个引脚的输出状态,0 代表对应引脚输出低电平,1 代表对应引脚输出高电平。

例如,设置 GPIOA 所有引脚为高电平。

```
GPIO_Write(GPIOA,0xFFFF);
```

5. 反转引脚状态函数

该函数将 GPIO 引脚状态反转,使用位异或操作输出数据寄存器。语法格式为

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

其中,参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象,用初始化函数的参数 1 定义;参数 uint16_t GPIO_Pin 为操作引脚。

例如,设置 GPIO 的 PA3、PA5 引脚状态反转。

```
GPIO_ToggleBits(GPIOA,GPIO_Pin_3|GPIO_Pin_5);
```

6. 复用功能设置函数

该函数将 GPIO 的某个引脚设置为特定的复用功能,操作的是复用功能低位寄存器或复用功能高位寄存器。语法格式为

```
void GPIO_PinAFConfig(GPIO_TypeDef *  GPIOx,uint16_t GPIO_PinSource,uint8_tGPIO_AF);
```

其中,参数 GPIO_TypeDef * GPIOx 为 GPIO 操作对象,用初始化函数的参数 1 定义;参数 uint16_t GPIO_PinSource 为复用引脚,在 stm32f4xx_gpio.h 文件中有如下定义。

```
#define GPIO_PinSource0     ((uint8_t)0x00)
#define GPIO_PinSource1     ((uint8_t)0x01)
#define GPIO_PinSource2     ((uint8_t)0x02)
…
#define GPIO_PinSource14    ((uint8_t)0x0E)
#define GPIO_PinSource15    ((uint8_t)0x0F)
```

参数 uint8_tGPIO_AF 为复用对象,在 stm32f4xx_gpio.h 文件中有片上外设引脚复用宏定义。例如,USART1 的引脚复用宏定义如下。

```
#define GPIO_AF_USART1     ((uint8_t)0x07)
```

例如,将 PA9、PA10 分别复用为 USART1 的发送(TX)引脚和接收(RX)引脚。PA9、PA10 可复用的功能如表 5-1 所示。

表 5-1 PA9、PA10 可复用的功能

引 脚	复 用 功 能
PA9	TIM1_CH2、I2C3_SMBA、USART1_TX、DCMI_D0、EVENTOUT
PA10	TIM1_CH3、USART1_RX、OTG_FS_ID、DCMI_D1、EVENTOUT

USART1 对应的复用标号是 GPIO_AF_USART1(AF7),因此实现程序如下。

```
//PA9 连接 AF7,复用为 USART1_TX
GPIO_PinAFConfig(GPIOA,GPIO_PinSource9,GPIO_AF_USART1);
//PA10 连接 AF7,复用为 USART1_RX
GPIO_PinAFConfig(GPIOA,GPIO_PinSource10,GPIO_AF_USART1);
```

5.4 STM32 的 GPIO 使用流程

使用库函数实现 GPIO 的应用,一般需要以下几步。

(1) 使能 GPIO 的时钟(非常重要),涉及 stm32f4xx_rcc.h 头文件和 stm32f4xx_rcc.c 源文件。

使用的主要函数如下。

```
RCC_AHB1PeriphClockCmd(uint32_t RCC_AHB1Periph,FunctionalState NewState);
```

片上外设一般被设计为数字时序电路,需要驱动时钟才能工作。片上外设大都被挂载在 AHB1、AHB2、APB1、APB2 这 4 条总线上,因此,工作时钟由对应总线时钟驱动。微控制器为每个片上外设设置了一个时钟开关,可以控制片上外设的运行和禁止。通过操作 4 个外设时钟使能寄存器 RCC_AHB1ENR、RCC_AHB2ENR、RCC_APB1ENR、RCC_APB2ENR 相应的位段实现。

所有 GPIO 挂载在 AHB1 总线上,使能 GPIO 的工作时钟,操作的是外设时钟使能寄存器 RCC_AHB1ENR。

例如,使能 GPIOA 的工作时钟使用的函数如下。

```
RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOA,ENABLE);
```

该函数有以下两个参数。

参数 1 为一个片上外设时钟对应于外设时钟使能寄存器中的使能屏蔽字,被定义在 stm32f4xx_rcc.h 文件中。

例如,GPIOA 的时钟使能位在 RCC_AHB1ENR 中的 0 位,因此有如下屏蔽字定义。

```
#define RCC_AHB1Periph_GPIOA        ((uint32_t) 0x00000001)
```

参数 2 为使能/禁止片上外设时钟。ENABLE(使能时钟)和 DISABLE(禁止时钟)以枚举类型分别被定义为 0 和非 0。RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_

GPIOA,ENABLE)函数实现的功能,就是根据参数 2 的使能状态,将 RCC_AHB1ENR 的 0 位置 1(ENABLE,使能时钟)或清零(DISABLE,禁止时钟)。

(2) 设置对应于片上外设使用的 GPIO 工作模式。

(3) 如果使用复用功能,需要单独设置每个 GPIO 引脚的复用功能。

(4) 在应用程序中读取引脚状态、控制引脚输出状态或使用复用功能完成特定功能。

5.5 STM32 GPIO 输出应用实例

本节 GPIO 输出应用实例是使用固件库点亮 LED 灯。

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。

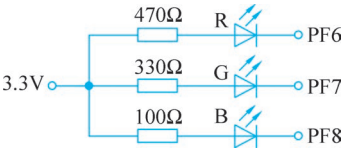

图 5-6 LED 灯硬件电路

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。如果读者使用的开发板中 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。

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;当输为 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 1s。

(6) 关灯 1s。

5.5.2 STM32 的 GPIO 输出应用软件设计

为了使工程更加有条理,把 LED 控制相关的代码独立分开存储,方便以后移植。在工程模板上新建 bsp_led.c 及 bsp_led.h 文件,其中 bsp 即 Board Support Packet 的缩写(板级支持包)。

1. bsp_led.h 头文件

```
#ifndef __LED_H
#define  __LED_H

#include "stm32f4xx.h"

//引脚定义
/*******************************************************/
//R 红色灯
#define LED1_PIN                    GPIO_Pin_6
#define LED1_GPIO_PORT              GPIOF
#define LED1_GPIO_CLK               RCC_AHB1Periph_GPIOF

//G 绿色灯
#define LED2_PIN                    GPIO_Pin_7
#define LED2_GPIO_PORT              GPIOF
#define LED2_GPIO_CLK               RCC_AHB1Periph_GPIOF

//B 蓝色灯
#define LED3_PIN                    GPIO_Pin_8
#define LED3_GPIO_PORT              GPIOF
#define LED3_GPIO_CLK               RCC_AHB1Periph_GPIOF
/*******************************************************/

/* 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  0
#define OFF 1

/* 带参宏,可以像内联函数一样使用 */
#define LED1(a)  if (a) \
                    GPIO_SetBits(LED1_GPIO_PORT,LED1_PIN);\
                    else    \
                    GPIO_ResetBits(LED1_GPIO_PORT,LED1_PIN)

#define LED2(a)  if (a) \
                    GPIO_SetBits(LED2_GPIO_PORT,LED2_PIN);\
                    else    \
                    GPIO_ResetBits(LED2_GPIO_PORT,LED2_PIN)

#define LED3(a)  if (a) \
                    GPIO_SetBits(LED3_GPIO_PORT,LED3_PIN);\
                    else    \
                    GPIO_ResetBits(LED3_GPIO_PORT,LED3_PIN)

/* 直接操作寄存器的方法控制 I/O */
#define  digitalHi(p,i)     {p->BSRRL=i;}          //设置为高电平
#define digitalLo(p,i)      {p->BSRRH=i;}          //输出低电平
```

```
#define digitalToggle(p,i)      {p->ODR ^= i;}        //输出反转状态

/* 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
void LED_GPIO_Config(void);
#endif /* __LED_H */
```

这部分宏控制 LED 亮灭的操作是直接向 BSRR、BRR 和 ODR 这 3 个寄存器写入控制指令实现的。对 BSRR 寄存器写 1 输出高电平，对 BRR 寄存器写 1 输出低电平，对 ODR 寄存器某位进行异或操作可反转位的状态。

RGB 彩灯可以实现混色。

上述代码中的\是 C 语言中的续行符语法，表示续行符的下一行与续行符所在的代码是同一行。因为代码中宏定义关键字 #define 只对当前行有效，所以使用续行符连接起来。以下的代码是等效的。

```
#define LED_YELLOW LED1_ON; LED2_ON; LED3_OFF
```

应用续行符时要注意，在\后面不能有任何字符(包括注释、空格)，只能直换行。

2. bsp_led.c 源文件

```
#include "./led/bsp_led.h"

 /*
  * @brief  初始化控制 LED 的 I/O
  * @param  无
  * @retval 无
  */
void LED_GPIO_Config(void)
{
    /* 定义一个 GPIO_InitTypeDef 类型的结构体 */
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启 LED 相关的 GPIO 外设时钟 */
    RCC_AHB1PeriphClockCmd ( LED1_GPIO_CLK|LED2_GPIO_CLK|
    LED3_GPIO_CLK, ENABLE);

    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStructure.GPIO_Pin = LED1_PIN;

    /* 设置引脚模式为输出模式 */
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;

    /* 设置引脚的输出类型为推挽输出 */
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;

    /* 设置引脚为上拉模式 */
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
```

```
    /* 设置引脚速率为 2MHz */
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;

    /* 调用库函数,使用上面配置的 GPIO_InitStructure 初始化 GPIO */
    GPIO_Init(LED1_GPIO_PORT, &GPIO_InitStructure);

    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStructure.GPIO_Pin = LED2_PIN;
    GPIO_Init(LED2_GPIO_PORT, &GPIO_InitStructure);

    /* 选择要控制的 GPIO 引脚 */
    GPIO_InitStructure.GPIO_Pin = LED3_PIN;
    GPIO_Init(LED3_GPIO_PORT, &GPIO_InitStructure);

    /* 关闭 RGB 灯 */
    LED_RGBOFF;
}
```

初始化 GPIO 端口时钟时采用了 STM32 库函数,函数执行流程如下。

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量,以便下面用于存储 GPIO 配置。

(2) 调用库函数 RCC_AHB1PeriphClockCmd()使能 LED 灯的 GPIO 端口时钟,如果是直接向 RCC 寄存器赋值使能时钟的,不如这样直观。该函数有两个输入参数,第 1 个参数用于指示要配置的时钟,如本例中的 RCC_AHB1Periph_GPIOH 和 RCC_AHB1Periph_GPIOD,应用时使用|操作同时配置 3 个 LED 的时钟;函数的第 2 个参数用于设置状态,可输入 DISABLE 关闭或 ENABLE 使能时钟。

(3) 向 GPIO 初始化结构体赋值,把引脚初始化成推挽输出模式,其中的 GPIO_Pin 使用宏 LEDx_PIN 赋值,使函数的实现方便移植。

(4) 使用以上初始化结构体的配置,调用 GPIO_Init()函数向寄存器写入参数,完成 GPIO 的初始化,这里的 GPIO 端口使用宏 LEDx_GPIO_PORT 赋值,也是为了程序移植方便。

(5) 使用同样的初始化结构体,只修改控制的引脚和端口,初始化其他 LED 使用的 GPIO 引脚。

(6) 使用宏控制 RGB 灯默认关闭。

编写完 LED 的控制函数后,就可以在 main 函数中测试了。

3. main.c 文件

```
#include "stm32f4xx.h"
#include "./led/bsp_led.h"

void Delay(__IO u32 nCount);

/*
  * @brief  主函数
  * @param  无
```

```
  * @retval 无
  */
int main(void)
{
  /* LED 端口初始化 */
  LED_GPIO_Config();

  /* 控制 LED 灯 */
  while (1)
  {
    LED1( ON );                          // 亮
    Delay(0xFFFFFF);
    LED1( OFF );                         // 灭

    LED2( ON );                          // 亮
    Delay(0xFFFFFF);
    LED2( OFF );                         // 灭

    LED3( ON );                          // 亮
    Delay(0xFFFFFF);
    LED3( OFF );                         // 灭

    /*轮流显示红绿蓝黄紫青白*/
    LED_RED;
    Delay(0xFFFFFF);

    LED_GREEN;
    Delay(0xFFFFFF);

    LED_BLUE;
    Delay(0xFFFFFF);

    LED_YELLOW;
    Delay(0xFFFFFF);

    LED_PURPLE;
    Delay(0xFFFFFF);

    LED_CYAN;
    Delay(0xFFFFFF);

    LED_WHITE;
    Delay(0xFFFFFF);

    LED_RGBOFF;
    Delay(0xFFFFFF);
  }
}

void Delay(__IO uint32_t nCount)        //简单的延时函数
{
  for(; nCount != 0; nCount--);
}
```

在 main 函数中，调用定义的 LED_GPIO_Config()函数初始化 LED 的控制引脚，然后直接调用各种控制 LED 亮灭的宏实现 LED 的控制。

以上就是一个使用 STM32 标准软件库开发应用的流程。

把编译好的程序下载到开发板并复位，可以看到 RGB 彩灯轮流显示不同的颜色。

5.6 STM32 GPIO 输入应用实例

本节 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一下子断开，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 5-7 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。

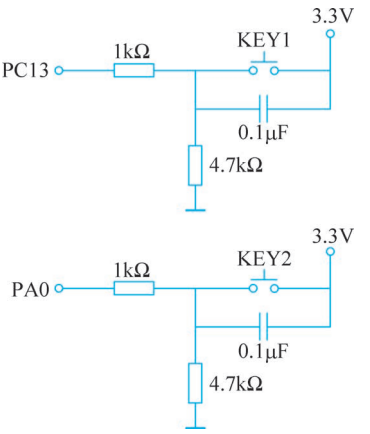

图 5-7 按键检测电路

从按键检测电路可知，这些按键在没有被按下时，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)。只要按键检测引脚的输入电平，即可判断按键是否被按下。

若读者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 5-7 电路设计一个示例，通过按键控制 LED，具体如下。

(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

为了使工程更加有条理，把 LED 控制相关的代码独立分开存储，方便以后移植。在工程模板上新建 bsp_key.c 及 bsp_key.h 文件，其中的 bsp 即 Board Support Packet 的缩写(板级支持包)。

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. bsp_key.h 头文件

```
#ifndef __KEY_H
#define  __KEY_H

#include "stm32f4xx.h"

//引脚定义
/*******************************************************/
#define KEY1_PIN                  GPIO_Pin_0
#define KEY1_GPIO_PORT            GPIOA
#define KEY1_GPIO_CLK             RCC_AHB1Periph_GPIOA

#define KEY2_PIN                  GPIO_Pin_13
#define KEY2_GPIO_PORT            GPIOC
#define KEY2_GPIO_CLK             RCC_AHB1Periph_GPIOC
/*******************************************************/
/* 按键按下标志宏
 * 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

void Key_GPIO_Config(void);
uint8_t Key_Scan(GPIO_TypeDef * GPIOx,u16 GPIO_Pin);

#endif /* __LED_H */
```

2. bsp_key.c 源文件

```
#include "./key/bsp_key.h"
/// 不精确的延时
void Key_Delay(__IO u32 nCount)
{
    for(; nCount != 0; nCount--);
}

/*
 * @brief  配置按键用到的 I/O 口
 * @param  无
 * @retval 无
 */
void Key_GPIO_Config(void)
{
    GPIO_InitTypeDef GPIO_InitStructure;

    /* 开启按键 GPIO 的时钟 */
    RCC_AHB1PeriphClockCmd(KEY1_GPIO_CLK|KEY2_GPIO_CLK,ENABLE);

    /* 选择按键 1 的引脚 */
    GPIO_InitStructure.GPIO_Pin = KEY1_PIN;

    /* 设置引脚为输入模式 */
```

```
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;

    /* 设置引脚不上拉也不下拉 */
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;

    /* 使用上面的结构体初始化按键 2 */
    GPIO_Init(KEY1_GPIO_PORT, &GPIO_InitStructure);

    /* 选择按键 2 的引脚 */
    GPIO_InitStructure.GPIO_Pin = KEY2_PIN;

    /* 使用上面的结构体初始化按键 2 */
    GPIO_Init(KEY2_GPIO_PORT, &GPIO_InitStructure);
}
```

函数执行流程如下。

(1) 使用 GPIO_InitTypeDef 定义 GPIO 初始化结构体变量,以便下面用于存储 GPIO 配置。

(2) 调用库函数 RCC_AHB1PeriphClockCmd()使能按键的 GPIO 端口时钟,调用时使用|操作同时配置两个按键的时钟。

(3) 向 GPIO 初始化结构体赋值,把引脚初始化成浮空输入模式,其中的 GPIO_Pin 使用宏 KEYx_PIN 赋值,使函数的实现方便移植。由于引脚的默认电平受按键电路影响,所以设置成浮空、上拉、下拉模式均没有区别。

(4) 使用以上初始化结构体的配置,调用 GPIO_Init()函数向寄存器写入参数,完成 GPIO 的初始化,这里的 GPIO 端口使用宏 KEYx_GPIO_PORT 赋值,也是为了程序移植方便。

(5) 使用同样的初始化结构体,只修改控制的引脚和端口,初始化其他按键检测时使用的 GPIO 引脚。

```
/*
  * @brief    检测是否有按键按下
  * @param    GPIOx:具体的端口, x 可以是 A~K
  * @param    GPIO_PIN:具体的端口位, 可以是 GPIO_PIN_x(x 可以是 0~15)
  * @retval   按键的状态
  * @arg KEY_ON:按键按下
  * @arg KEY_OFF:按键未按下
  */
uint8_t Key_Scan(GPIO_TypeDef * GPIOx,uint16_t GPIO_Pin)
{
    /* 检测是否有按键按下 */
    if(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON )
    {
        /* 等待按键释放 */
        while(GPIO_ReadInputDataBit(GPIOx,GPIO_Pin) == KEY_ON);
        return   KEY_ON;
    }
    else
        return KEY_OFF;
}
```

这里定义了一个 Key_Scan()函数,用于扫描按键状态。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 STM32 标准库提供了库函数 GPIO_ReadInputDataBit()获取位状态,该函数输入 GPIO 端口及引脚号,函数返回该引脚的电平状态,高电平返回 1,低电平返回 0。Key_Scan()函数中以 GPIO_ReadInputDataBit()函数的返回值与自定义的宏 KEY_ON 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 Key_Scan()函数中做软件滤波,防止波纹抖动引起误触发。

3. main.c 程序

```
#include "stm32f4xx.h"
#include "./led/bsp_led.h"
#include "./key/bsp_key.h"

/*
 * @brief  主函数
 * @param  无
 * @retval 无
 */
int main(void)
{
    /* LED 端口初始化 */
    LED_GPIO_Config();

    /* 初始化按键 */
    Key_GPIO_Config();

    /* 轮询按键状态,若按键按下则翻转 LED */
    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;
        }
    }
}
```

代码中初始化 LED 及按键后,在 while 循环中不断调用 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。

把编译好的程序下载到开发板并复位,按下 KEY1 和 KEY2 分别可以控制 LED 的亮、灭状态。