第 3 章 STM32 通用输入输出接口

CHAPTER 3

本章首先对 STM32 通用输入输出接口进行了概述，包括输入通道和输出通道。其次，详细介绍了 STM32 GPIO 的功能，涵盖普通 I/O 功能、单独的位设置或位清除、外部中断/唤醒线、复用功能(Alternate Function，AF)、软件重新映射 I/O 复用功能、GPIO 锁定机制、输入配置、输出配置、复用功能配置、模拟输入配置、STM32 的 GPIO 操作、外部中断映射和事件输出以及 GPIO 的主要特性。随后，系统讲解了 GPIO 的 HAL 驱动程序。接着，完整梳理了 STM32 的 GPIO 使用流程，包括普通 GPIO 配置和复用功能 I/O(Alternate Function I/O，AFIO)配置。最后通过采用 STM32G474 和 HAL 库的 GPIO 输出应用实例和 GPIO 输入应用实例，分别展示了 STM32G474 的 GPIO 输出应用硬件设计与软件设计以及 STM32G474 的 GPIO 输入应用硬件设计与软件设计的完整实现过程。



第 3 集
微课视频

3.1 STM32 通用输入输出接口概述

GPIO 是通用输入输出(General Purpose Input/Output)的缩写，其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，具备极强的通用性和灵活性，相关开发技术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 可以将内部的数据输出至外部设备或模块，如驱动 LED、数码管、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 来模仿 UART、SPI、I2C 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，STM32G4 系列上除特殊功能的引脚外，所有的引脚都可以作为 GPIO 使用。以常见的 LQFP144 封装的 STM32G4 系列微控制器(如 STM32G474VET6)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，STM32G4 系列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 开始依次命名为 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 等方式来指代单个的 GPIO。

以 STM32G474VET6 为例，它共有 7 个端口(PA、PB、PC、PD、PE、PF 和 PG)，每个端口 16 个 GPIO，共 7×16＝112 个 GPIO。部分 STM32G4 系列型号还可能包含 PH 等端

口，具体端口数量和 GPIO 总数因封装和型号不同而有所差异，但整体的端口分组和编号规则保持一致。

几乎所有的嵌入式系统应用都涉及开关量的输入和输出功能，例如状态指示、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通用输入/输出接口实现。

GPIO 端口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，当它输入引脚悬空时，相当于该引脚接了高电平。一般实际运用时，引脚不建议悬空，易受干扰。

（2）输入上拉：上拉就是把电压拉高，比如拉到 Vcc。上拉就是将不确定的信号通过一个电阻嵌位在高电平。电阻同时起限流作用。强弱只是上拉电阻的阻值不同，没有严格区分。

（3）输入下拉：下拉就是把电压拉低，拉到 GND，原理上与上拉原理相似。

（4）模拟输入：模拟输入是指传统方式的模拟量输入。

（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。适用于电流型驱动，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的时候另一个截止。

（7）具有上拉/下拉功能的复用功能推挽模式：可以理解为 GPIO 口被用作第二功能时的配置情况(并非作为通用 I/O 口使用)。STM32 GPIO 的推挽复用模式，复用功能模式中输出使能、输出速度可配置。这种复用模式可工作在开漏及推挽模式，但是输出信号来源于其他外设，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 实际状态，但一般直接用外设的寄存器来获取该数据信号。

（8）具有上拉/下拉功能的复用功能开漏模式：每个 I/O 可以自由编程，而 I/O 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 GPIOxBRR 寄存器支持对任何 GPIO 寄存器的独立读/更改访问，这样，在读和更改访问之间产生中断时，不会造成误操作。

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)、2 个 32 位数据寄存器(GPIOx_IDR 和 GPIOx_ODR)、1 个 32 位置位/复位寄存器(GPIOx_BSRR)、1 个 32 位配置锁存寄存器(GPIOx_LCKR)和 2 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来实现 GPIO 的配置和应用。

一个 I/O 口位的基本结构如图 3-1 所示。

STM32 系列微控制器的 GPIO 资源极为丰富，不同型号分别配备 26、37、51、80、112 个多功能双向 I/O 口，且均兼容 5V 电平，响应速度快。更重要的是，所有 I/O 口均可映射至 16 个外部中断通道，为中断触发设计提供了灵活支持。因此，GPIO 作为 STM32 最基础也最核心的外设之一，是初学者入门学习的最佳起点。

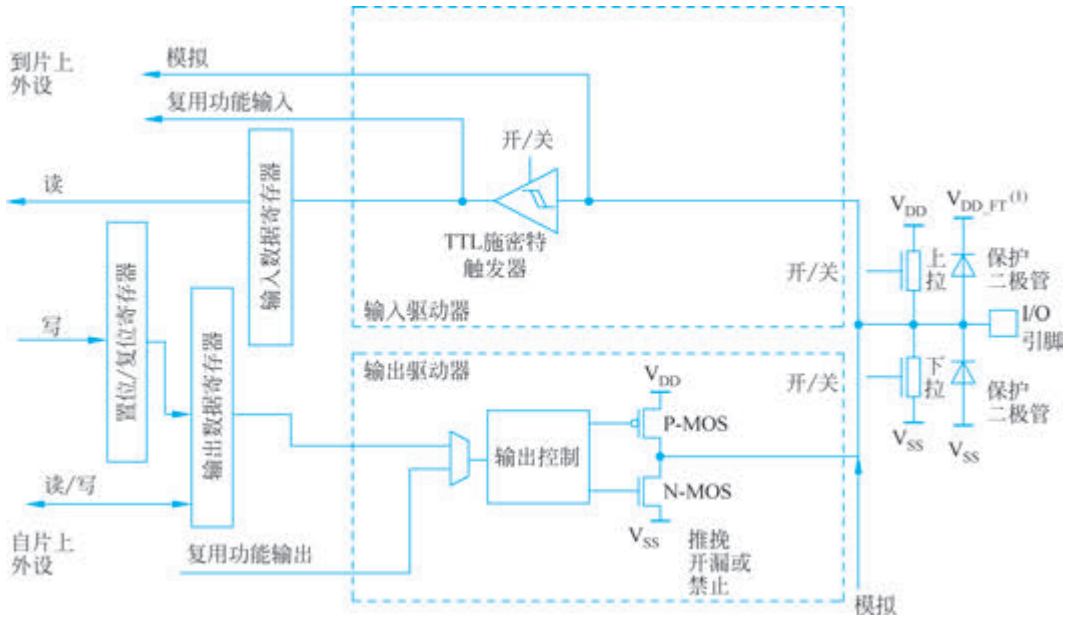

图 3-1 一个 I/O 口位的基本结构

每个 GPIO 端口通过 7 组寄存器实现配置与控制，具体如下。

(1) 2 个 32 位配置寄存器(GPIOx_CRL、GPIOx_CRH)：用于设置 I/O 口的工作模式(如输入、输出、复用功能等)和输出速率。

(2) 2 个 32 位数据寄存器(GPIOx_IDR、GPIOx_ODR)：分别用于读取输入电平状态和设置输出电平。

(3) 1 个 32 位置位/复位寄存器(GPIOx_BSRR)：通过原子操作实现 I/O 口的置位或复位，避免中断干扰。

(4) 1 个 16 位复位寄存器(GPIOx_BRR)：专门用于对输出口进行复位操作。

(5) 1 个 32 位锁定寄存器(GPIOx_LCKR)：用于锁定 I/O 口的配置状态，防止误操作修改。

这些寄存器的精准配置，是实现 GPIO 多样化功能的关键，也是深入理解 STM32 外设工作机制的基础。

GPIO 端口的每个位可以由软件分别配置成多种模式。每个 I/O 端口位可以自由编程，然而 I/O 端口寄存器必须按 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 IRQ 时不会发生危险。常用的 I/O 端口寄存器只有 4 个：CRL、CRH、IDR、ODR。CRL 和 CRH 控制着每个 I/O 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 引脚均可复用为数字或模拟外设功能。除了具有模拟输入功能的端口，所有的 GPIO 引脚都具备大电流驱动能力。

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 口位的基本结构包括以下几部分。

3.1.1 输入通道

输入通道包括输入数据寄存器和输入驱动器(图 3-1 中带虚框部分),I/O 引脚附近连接有两只保护二极管。

输入驱动器中的另外一个部件是 TTL 施密特触发器,当 I/O 口位用于开关量输入或者复用功能输入时,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 施密特触发器、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 施密特触发器、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 施密特触发器关闭。

(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(即悬空),其电平高低完全由外部电路决定。

3.1.2 输出通道

输出通道中包括位设置/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到位设置/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 P-MOS 和 N-MOS 场效应管电路输出到引脚。通过软件设置,P-MOS 和 N-MOS 场效应管电路可以构成推挽方式、开漏方式或者关闭。

GPIO 的输出驱动器主要由多路选择器、输出控制和一对互补的 MOS 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。

(2) 输出控制逻辑和一对互补的 MOS 管。

输出控制逻辑根据用户设置通过控制 PMOS 管和 NMOS 管的状态(导通/关闭)决定 GPIO 输出模式(推挽、开漏还是关闭)。

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

PMOS 管导通，NMOS 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，NMOS 管导通，PMOS 管截止，外部输出低电平（输出电压 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} 引脚的 PMOS 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，NMOS 管导通，外部输出低电平（输出电压 0V）；当内部输出 1 时，NMOS 管截止，由于此时 PMOS 管也处于截止状态，外部输出既不是高电平，也不是低电平，而是高阻态（悬空）。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（pull-up resistor）。

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的高低。

例如，若要实现 STM32 微控制器输出 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，速度越慢，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，开漏输出可以实现电平匹配，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，适合做电流型的驱动。

3.2 STM32 GPIO 的功能

STM32 微控制器的 GPIO（通用输入/输出）引脚是非常灵活的，可以被配置为输入、输出或其他特殊功能模式。这些引脚可以用于读取数字信号、输出数字信号、连接外部设备、控制 LED、读取按钮状态等。

3.2.1 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 引脚均内置弱上拉和弱下拉电阻，当配置为输入时，二者可被激活或断开。

3.2.2 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”（GPIOx_BSRR，其中复位专用寄

存器为 GPIOx_BRR)中想要更改的位写 1 来实现的。没有被选择的位将不被更改。

3.2.3 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。

3.2.4 复用功能

使用默认复用功能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,端口位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。

3.2.5 软件重新映射 I/O 复用功能

STM32G4 微控制器的 I/O 引脚除通用功能外,还可配置为片上外设的复用功能。一个 I/O 引脚除可作为某个默认外设的复用引脚外,还能作为其他多个不同外设的复用引脚;同样,一个片上外设除默认复用引脚外,也可拥有多个备用复用引脚。在基于 STM32G4 微控制器的应用开发中,用户可根据实际需求将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。

为使不同封装器件的外设 I/O 功能数量达到最优,部分复用功能可重新映射到其他引脚上。这一操作通过软件配置 GPIOx_AFR(GPIO 复用功能寄存器)来实现,配置后,复用功能将不再映射到其原始引脚上。

STM32G4 系列通过外设复用功能编号管理不同外设的引脚映射,每个复用功能对应特定的 AF 编号,通过将 GPIO 引脚配置为相应 AF 编号,即可实现外设与引脚的关联,相比传统 AFIO 寄存器配置方式更直观灵活。

3.2.6 GPIO 锁定机制

锁定机制允许冻结 I/O 配置。当对一个端口位执行锁定(LOCK)程序,在下一次复位前,将无法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。

3.2.7 输入配置

当 I/O 口配置为输入时,会出现以下情况。

(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、下拉或浮空)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 时钟周期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 3-2 所示。

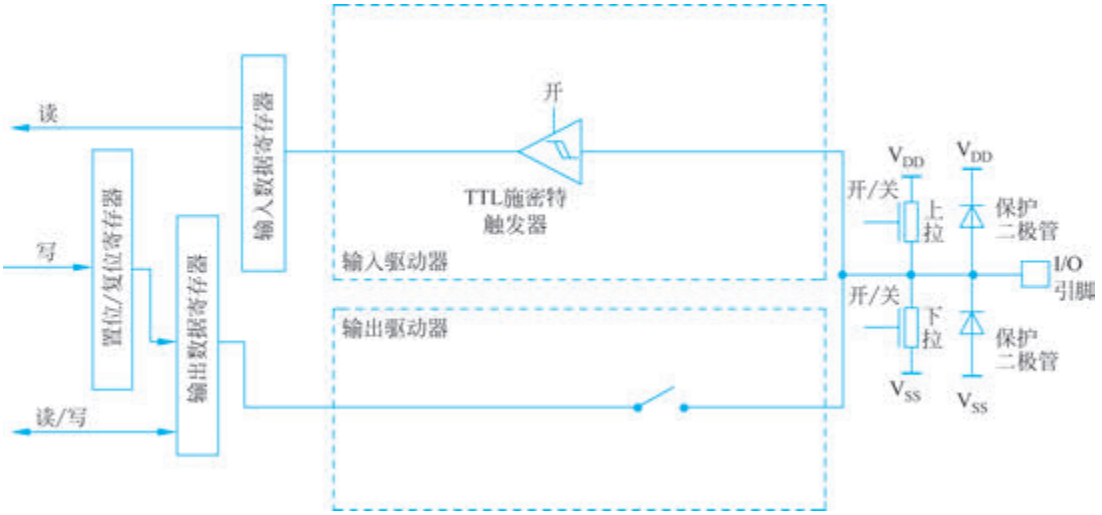

图 3-2 I/O 口位的输入配置

3.2.8 输出配置

当 I/O 口被配置为输出时,会出现以下情况。

(1) 输出缓冲器被激活。

① 开漏模式:输出寄存器上的 0 激活 N-MOS,而输出寄存器上的 1 将端口置于高阻状态(P-MOS 从不被激活)。

② 推挽模式:输出寄存器上的 0 激活 N-MOS,而输出寄存器上的 1 将激活 P-MOS。

(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 时钟周期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 3-3 所示。

3.2.9 复用功能配置

当 I/O 口被配置为复用功能时,会出现以下情况。

(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。

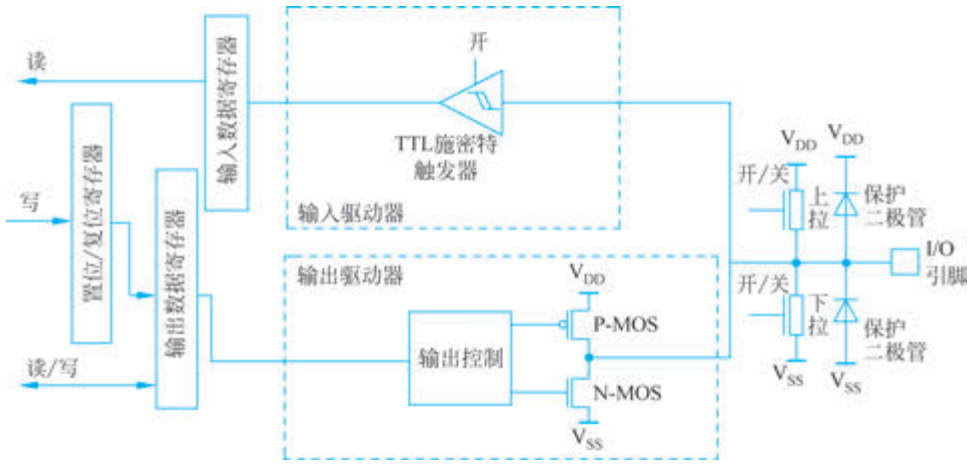

图 3-3　I/O 口位的输出配置

（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。

一组复用功能 I/O 寄存器允许用户把一些复用功能重映射到不同的引脚。

I/O 口位的复用功能配置如图 3-4 所示。

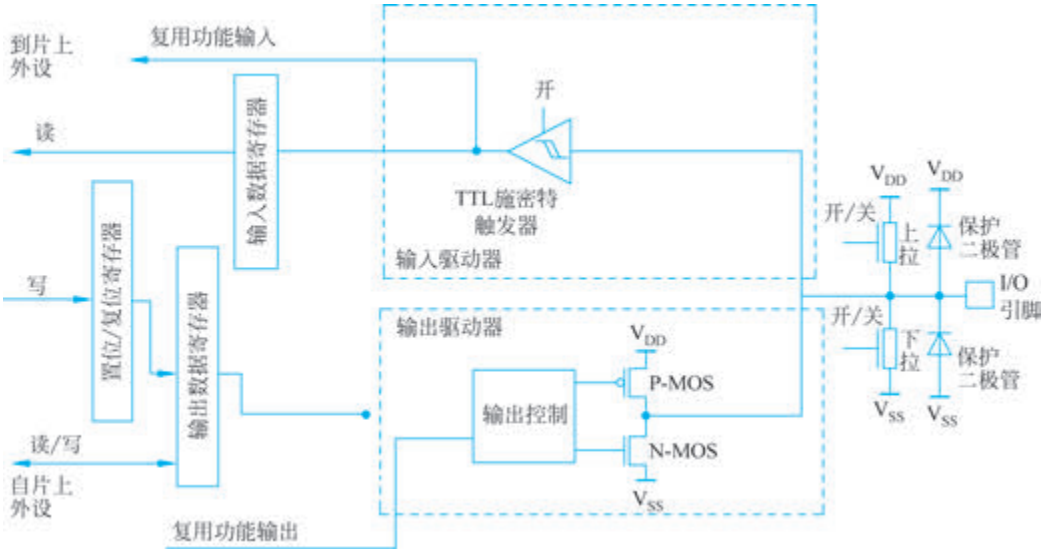

图 3-4　I/O 口位的复用功能配置

3.2.10　模拟输入配置

当 I/O 口被配置为模拟输入模式时，会出现以下情况。

（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 引脚上的零功耗。施密特触发输出值被强制置为 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 3-5 所示。

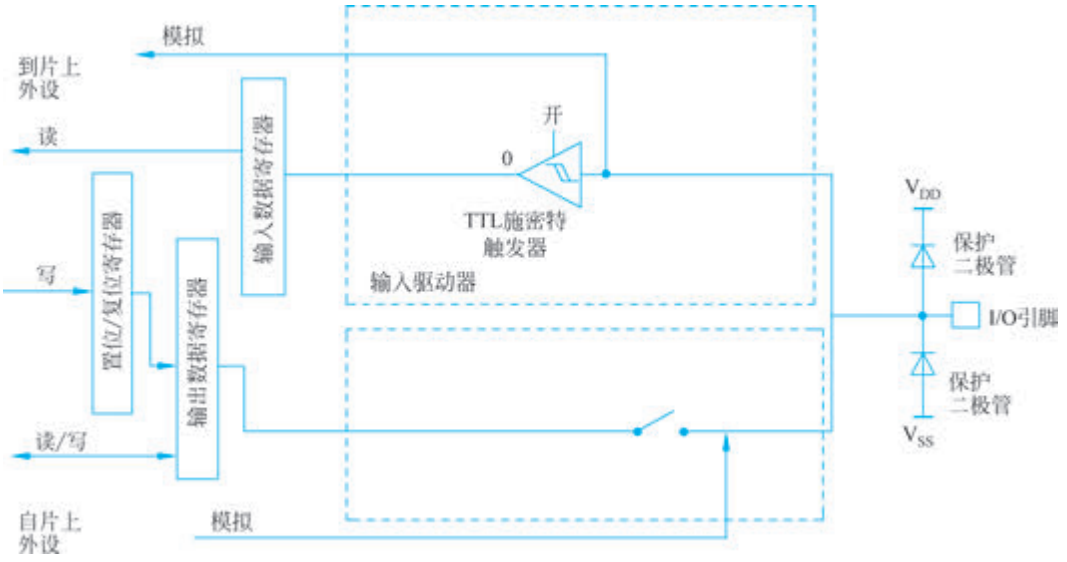

图 3-5 I/O 口位的高阻抗模拟输入配置

3.2.11 STM32 的 GPIO 操作

在 STM32G4 微控制器中操作 GPIO(通用输入 / 输出)引脚涉及几个关键步骤,包括配置 GPIO 模式、设置引脚状态以及读取引脚输入。

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。

为节约电能,只有被开启的 GPIO 端口才会被供给时钟。因此复位后所有 GPIO 接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[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(浮空、上拉/下拉或者模拟输入),还是输出模式(推挽输出、开漏输出或者复用功能推挽/开漏输出)。每个 GPIO 引脚还拥有自己的端口模式位 OSPEEDRy[1:0],用于选择该引脚在输出模式中的输出速度(低速、中速、高速和超高速)。

每个端口拥有 16 个引脚,每个引脚的上述控制位需要 5 位(MODERy[1:0]、OTy[0]、OSPEEDRy[1:0]),因此需要 80 位才能实现对一个端口所有引脚的配置,它们被分置在多个寄存器中。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 APB 时钟周期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APB时钟周期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NMOS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APB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NMOS管可以被打开。

3. GPIO输入的读取

每个端口都有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APB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，得到GPIO引脚状态对应的值。

4. GPIO输出的控制

STM32G4为每组16引脚的端口提供了32位的控制寄存器GPIOx_ODR、GPIOx_BSRR（其中x指代A、B、C等端口号）。其中GPIOx_ODR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，软件可以通过对该寄存器执行置位或清零操作，使对应引脚输出高电平或低电平。

利用GPIOx_BSRR寄存器可以解决多个引脚同时改变电平的问题，该寄存器的低16位用于置位操作，高16位用于清零操作，可实现对多个引脚的同时置位或清零，避免了“读-修改-写”操作带来的潜在问题。

5. 输出速度

如果STM32G4的I/O引脚工作处于某个输出模式下，通常还需设置其输出速度，这个输出速度指的是I/O口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。

STM32G4的芯片内部在I/O口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，通过选择不同的响应速度档位匹配合适的输出驱动模块，以达到优化噪声控制、降低功耗的目的。高频的驱动电路噪声也相对较高，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（Electromagnetic Interference，EMI）性能。当然如果要输出较高频率的信号，但选用了较低频率响应速度的驱动模块，则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。

STM32G4的I/O引脚的输出速度有4种选择：低速（2MHz）、中速（25MHz）、高速（50MHz）和超高速（100MHz）。

下面根据一些常见的应用，提供一些选用参考。

（1）连接LED、蜂鸣器等外部设备的普通输出引脚：一般设置为低速（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12.5Mb/s，选用中速（25MHz）的响应速度即可满足需求，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 1Mb/s，选用中速(25MHz)的 I/O 引脚速度即可满足需求。

(4) 用作 SPI 复用功能的输出引脚：假设 SPI 工作时比特率为 50Mb/s，需要选用超高速(100MHz)的 I/O 引脚速度。

(5) 用作高速通信接口的输出引脚：根据实际通信速率，可选用高速(50MHz)或超高速(100MHz)的 I/O 引脚速度。

3.2.12 外部中断映射和事件输出

STM32G4 微控制器的 I/O 引脚可实现外部中断映射和事件输出功能，无须通过 AFIO 模块配置实现(STM32G4 系列已取消 AFIO 模块，相关功能整合至 SYSCFG 模块)。若需使用这些功能，必须先开启 APB2 上的 SYSCFG 时钟。

1. 外部中断映射

当 STM32G4 微控制器的某个 I/O 引脚被映射为外部中断线后，该 I/O 引脚即可作为外部中断源，通过引脚上的电平变化或边沿触发产生外部中断，实现与用户程序的交互。

STM32G4 微控制器的所有 I/O 引脚均具备外部中断能力。外部中断线 EXTI Linexx 可与所有 GPIO 端口(GPIOA～GPIOI)的对应引脚(xx 号引脚)关联。使用外部中断线时，需将对应 I/O 引脚配置为输入模式，并通过 SYSCFG_EXTICR 寄存器配置中断线与 GPIO 端口的映射关系。

2. 事件输出

STM32G4 微控制器的多数 I/O 引脚可配置为事件输出。例如，该功能可通过特定事件触发产生脉冲信号，用于将 STM32G4 从低功耗模式中唤醒，或实现其他事件联动功能。事件输出功能需通过配置相关寄存器实现，以确定事件源与输出引脚的映射关系。

3.2.13 GPIO 的主要特性

STM32G4 微控制器的 GPIO 主要具有以下特性：

(1) 提供最多 144 个多功能双向 I/O 引脚，引脚利用率高，可满足复杂应用的接口需求。

(2) 多达 114 个 I/O 引脚兼容 5V，每个 I/O 引脚具有 20mA 驱动能力(部分引脚支持更高电流驱动)，满足多数外设的驱动需求。

(3) 每个 I/O 引脚最高支持 100MHz 翻转速度，可适应高速信号传输场景。

(4) 每个 I/O 引脚有 8 种工作模式(输入模式包括浮空、上拉、下拉、模拟输入；输出模式包括推挽、开漏；复用功能包括推挽、开漏)，复位时复用功能未开启，I/O 引脚默认配置为浮空输入模式。

(5) 所有 I/O 引脚都具备复用功能，包括 SWD 调试接口、定时器、USART、I2C、SPI、CAN FD 等外设接口。

(6) 复用功能通过 AF 编号管理，每个引脚可映射多个外设功能，通过配置 GPIOx_AFR 寄存器实现功能切换，方便 PCB 布局设计。

(7) 所有 I/O 引脚都可作为外部中断输入，支持 16 条外部中断线，可配置边沿或电平触发方式。

(8) 多数 I/O 引脚可配置为事件输出，支持事件触发与低功耗模式唤醒等功能。

（9）特定引脚具备专用功能，例如部分引脚可作为从待机模式唤醒的引脚，支持低功耗场景。

3.3 GPIO 的 HAL 驱动程序

GPIO 引脚的操作主要包括初始化、引脚输入读取和引脚输出设置，相关的 HAL 驱动程序定义在文件 stm32g4xx_hal_gpio.h 中，GPIO 操作相关函数如表 3-1 所示，表中只列出了函数名，省略了函数参数。

表 3-1 GPIO 操作相关函数

函 数 名	函数功能描述
HAL_GPIO_Init()	GPIO 引脚初始化
HAL_GPIO_DeInit()	GPIO 引脚去初始化，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()	翻转引脚的输出状态
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 操作函数是引脚状态读写函数。

1. 初始化函数 HAL_GPIO_Init()

函数 HAL_GPIO_Init()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，配置内容包括输入/输出模式、上拉或下拉等。其原型定义如下：

```
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，该类型结构体定义了端口的各个寄存器的偏移地址，实际调用函数 HAL_GPIO_Init()时使用端口的基地址作为参数 GPIOx 的值，在文件 stm32g4xx.h 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,在文件 stm32g4 xx hal_gpio.h 中定义了 16 个引脚的宏。

如果需要同时配置多个引脚的功能,可将这些宏进行按位或运算组合。

```
#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 用于配置引脚功能模式,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              0x00000011U      //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,上跳沿触发
#define  GPIO_MODE_IT_FALLING             0x10210000U      //外部中断,下跳沿触发
#define  GPIO_MODE_IT_RISING_FALLING      0x10310000U      //上、下跳沿触发
```

(3) Pull 用于配置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 用于配置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,在文件 stm32g4xx hal gpio_ex.h 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,下面是其中的部分定义示例:

```
#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/I2S2 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. 设置引脚输出的函数 HAL_GPIO_WritePin()

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

```
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，是枚举类型 GPIO_PinState，在 stm32g4xx_hal_gpio.h 文件中的定义如下：

```
typedef enum
{
GPIO_PIN_RESET = 0,
GPIO_PIN_SET
}GPIO_PinState;
```

枚举常量 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. 读取引脚输入的函数 HAL_GPIO_ReadPin()

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

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

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

4. 翻转引脚输出的函数 HAL_GPIO_TogglePin()

函数 HAL_GPIO_TogglePin()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。其原型定义如下，只需传递端口号和引脚号：

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

3.4 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。

运行程序前，必须对所有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 具有锁定机制，允许冻结 I/O 配置。当在一个端口位上执行了锁定（LOCK）操作后，在下一次复位之前，将不能再更改端口位的配置。

3.4.1 普通 GPIO 配置

GPIO 操作是最基本的外设应用，其基本配置方法如下。

（1）配置 GPIO 时钟，完成初始化。

（2）利用函数 HAL_GPIO_Init 配置引脚，包括引脚编号、引脚输出速度、引脚工作模式。

（3）完成配置。

3.4.2 复用功能 I/O 配置

I/O 复用功能常对应外设的输入输出功能。使用时，需要先配置 IO 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下述三种情况。

（1）外设对应的引脚为输出：需要根据外围电路的配置将对应的引脚设置为复用推挽输出或复用开漏输出。

（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模式。

（3）ADC 对应的引脚：配置引脚为模拟输入。

3.5 采用 STM32G474 和 HAL 库的 GPIO 输出应用实例

本实例基于 STM32G474，实现两个 LED（LED0 和 LED1）交替闪烁的跑马灯效果。通过配置 GPIO 引脚为推挽输出模式，控制引脚电平实现 LED 点亮与熄灭。程序基于 HAL 库完成初始化、系统时钟配置及延时模块配置，在主循环中每 500ms 翻转一次 LED 状态。开发环境为 Keil MDK，程序结构清晰，资源占用低，适合嵌入式初学者学习基本 I/O 控制和硬件应用方法。

3.5.1 STM32G474 的 GPIO 输出应用硬件设计

STM32G474 与 LED 的硬件电路如图 3-6 所示。

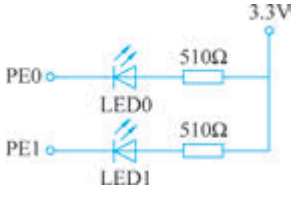

图 3-6 STM32G474 与 LED 的硬件电路

这些 LED 的阴极都连接到 STM32G474 的 GPIO 引脚，只需控制 GPIO 引脚的输出电平状态，即可控制 LED 的亮灭。若所用开发板的 LED 连接方式或对应引脚不同，仅需修改程序中的相关引脚定义即可，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 引脚输出低电平(0)时，LED 点亮；输出高电平(1)时，LED 熄灭。

在本实例中，根据图 3-6 所示的电路设计一个程序：LED0 和 LED1 每 500ms 交替闪烁一次，实现类似跑马灯的效果。

3.5.2 STM32G474 的 GPIO 输出应用软件设计

本程序基于 STM32G474 微控制器，实现两个 LED(LED0 和 LED1)交替闪烁的跑马灯效果。头文件 led.h 定义了 LED 对应的 GPIO 端口、引脚及时钟使能宏，并提供点亮、熄灭和翻转控制的宏接口。驱动文件 led.c 实现 GPIO 端口时钟使能和引脚配置，将 LED 引脚设置为推挽输出、高速、上拉模式，初始化后将 LED 置于关闭状态。主程序中，首先进行 HAL 库初始化、系统时钟配置和延时模块初始化，随后调用 led_init()完成 LED 硬件初始化。进入主循环后，通过反复切换两个 LED 的点亮和熄灭状态，配合 500ms 延时，实现 LED 交替闪烁的跑马灯效果。项目通过 Keil MDK 开发环境完成编译和下载，资源占用合理，程序无编译错误。调试运行时，板载 LED 按预期交替闪烁，展现 STM32 基本 I/O 控制及系统时钟和延时管理功能，适合作为嵌入式入门学习和硬件控制演示的示例。

1. led.h

```
/*
 * @file        led.h
 * @brief       LED 驱动代码
 */
#ifndef __LED_H
#define __LED_H
#include "./SYSTEM/sys/sys.h"
/*****************************************************************************/
/* 引脚定义 */
#define LED0_GPIO_PORT                  GPIOE
#define LED0_GPIO_PIN                   GPIO_PIN_0
#define LED0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
/* PE 口时钟使能 */
#define LED1_GPIO_PORT                  GPIOE
#define LED1_GPIO_PIN                   GPIO_PIN_1
#define LED1_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
/* LED 端口定义 */
#define LED0(x)    do{ x ? \
                       HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_SET) : \
                       HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_RESET); \
                   }while(0)       /* LED0 亮灭控制 */

#define LED1(x)    do{ x ? \
                       HAL_GPIO_WritePin(LED1_GPIO_PORT, LED1_GPIO_PIN, GPIO_PIN_SET) : \
                       HAL_GPIO_WritePin(LED1_GPIO_PORT, LED1_GPIO_PIN, GPIO_PIN_RESET); \
                   }while(0)       /* LED1 亮灭控制 */
```

```
/* LED 取反定义 */
#define LED0_TOGGLE()    do{ HAL_GPIO_TogglePin(LED0_GPIO_PORT, LED0_GPIO_PIN); }while(0)
/* LED0 = !LED0 */
#define LED1_TOGGLE()    do{ HAL_GPIO_TogglePin(LED1_GPIO_PORT, LED1_GPIO_PIN); }while(0)
/* LED1 = !LED1 */
/* 外部接口函数 */
void led_init(void);
/* 初始化 */
#endif
```

led.h 头文件是针对 STM32G474 微控制器设计的 LED 驱动代码接口，功能如下：

(1) 定义 LED 对应的 GPIO 端口和引脚。

宏定义 LED0 和 LED1 各自连接的 GPIO 端口(GPIOE)及引脚宏(GPIO_PIN_0、GPIO_PIN_1)。

例如：

```
#define LED0_GPIO_PORT GPIOE
#define LED0_GPIO_PIN  GPIO_PIN_0
```

(2) 定义 GPIO 端口时钟使能宏。

LED0_GPIO_CLK_ENABLE()和 LED1_GPIO_CLK_ENABLE()均用于开启 GPIOE 端口的时钟，使 GPIO 引脚可以正常工作。

例如：

```
#define LED0_GPIO_CLK_ENABLE() do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
```

(3) 定义 LED 灯控制宏。

LED0(x)和 LED1(x)根据传入参数 x 值控制对应引脚的输出电平，从而控制 LED 的亮灭。

当 x 为非零时，引脚输出高电平，LED 熄灭(具体逻辑由硬件连接方式决定)；当 x 为 0 时，输出低电平，点亮 LED。

例如：

```
#define LED0(x) do { x ? HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_SET) :
HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_RESET); } while(0)
```

(4) 定义 LED 灯电平取反宏。

LED0_TOGGLE() 和 LED1_TOGGLE()用于翻转对应 LED 引脚的电平，实现 LED 状态反转(亮变灭，灭变亮)。

例如：

```
#define LED0_TOGGLE() do { HAL_GPIO_TogglePin(LED0_GPIO_PORT, LED0_GPIO_PIN); } while(0)
```

(5) 声明 LED 初始化函数接口。

声明 led_init()函数，初始化 LED 所使用的 GPIO 端口和引脚，完成硬件配置。这样，头文件统一管理 LED 引脚定义和控制接口，方便主程序使用和维护。具体实现通常在 led.c 文件中。

2. led.c

```
/*
 * @file        led.c
 * @brief       LED 驱动代码
 */
#include "./BSP/LED/led.h"
/*
 * @brief       初始化 LED 相关 IO 口, 并使能时钟
 * @param       无
 * @retval      无
 */
void led_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;

    LED0_GPIO_CLK_ENABLE();                             /* LED0 时钟使能 */
    LED1_GPIO_CLK_ENABLE();                             /* LED1 时钟使能 */

    gpio_init_struct.Pin = LED0_GPIO_PIN;               /* LED0 引脚 */
    gpio_init_struct.Mode = GPIO_MODE_OUTPUT_PP;        /* 推挽输出 */
    gpio_init_struct.Pull = GPIO_PULLUP;                /* 上拉 */
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /* 高速 */
    HAL_GPIO_Init(LED0_GPIO_PORT, &gpio_init_struct);   /* 初始化 LED0 引脚 */

    gpio_init_struct.Pin = LED1_GPIO_PIN;               /* LED1 引脚 */
    HAL_GPIO_Init(LED1_GPIO_PORT, &gpio_init_struct);   /* 初始化 LED1 引脚 */

    LED0(1);                                            /* 关闭 LED0 */
    LED1(1);                                            /* 关闭 LED1 */
}
```

led.c 文件的代码是针对 STM32G474 微控制器的 LED 驱动实现，功能是初始化两个 LED 对应的 GPIO 引脚，具体说明如下：

(1) 使能 GPIO 端口时钟。

调用宏 LED0_GPIO_CLK_ENABLE()和 LED1_GPIO_CLK_ENABLE()，打开 LED0 和 LED1 对应 GPIO 端口(此处均为 GPIOE)的时钟，保证 GPIO 端口能够工作。

(2) 配置 LED 相关 GPIO 引脚。

使用 GPIO_InitTypeDef 结构体配置引脚的具体参数：

Pin：指定当前初始化的引脚(LED0 或 LED1 对应的 GPIO 引脚)。

Mode：设置为推挽输出模式(GPIO_MODE_OUTPUT_PP)，方便输出高低电平以控制 LED。

Pull：内部上拉电阻使能(GPIO_PULLUP)，确保引脚状态稳定。

Speed：设置为高速模式(GPIO_SPEED_FREQ_HIGH)，保证 GPIO 响应速度。

分别调用 HAL_GPIO_Init()初始化 LED0 和 LED1 对应的 GPIO 端口引脚。

(3) 初始关闭 LED。

通过调用 LED0(1)和 LED1(1)宏，将两个 LED 初始状态设为关闭(高电平)，避免上电时 LED 误亮。

该 led_init()函数通过如下步骤完成对 LED 相关硬件接口的初始化工作。

① 使能 GPIO 端口时钟；

② 设置 GPIO 引脚参数为推挽输出、高速、上拉；

③ 初始化对应引脚；

④ 关闭两个 LED。

led_init()函数是对 led.h 中声明函数的具体实现，确保主程序进入后可以通过 LED 控制接口正常控制灯的状态。

3. main.c

```
/*
 *@file        main.c
 *@brief       跑马灯 实验
 */
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"
#include "./BSP/LED/led.h"

int main(void)
{
    HAL_Init();                             /*初始化 HAL 库*/
    sys_stm32_clock_init(85, 2, 2, 4, 8);   /*设置时钟,170MHz*/
    delay_init(170);                        /*延时初始化*/
    led_init();                             /*初始化 LED*/

    while (1)
    {
        LED0(0);                            /*LED0 亮*/
        LED1(1);                            /*LED1 灭*/
        delay_ms(500);
        LED0(1);                            /*LED0 灭*/
        LED1(0);                            /*LED1 亮*/
        delay_ms(500);
    }
}
```

这段代码是基于 STM32G474 微控制器的主程序，用来实现一个简单的“跑马灯”效果（交替点亮两个 LED 灯）。下面是对代码功能和执行过程的详细说明。

该程序的功能是使两个 LED 灯（LED0 和 LED1）交替闪烁，即一个 LED 亮时另一个灭，反之亦然，间隔时间约为 500ms，从而形成交替亮灭的跑马灯效果。具体流程如下。

（1）初始化 HAL 库。

```
HAL_Init();
```

调用 HAL 库初始化函数，配置底层硬件抽象层，初始化中断向量表和系统滴答定时器等基础设施。

（2）配置系统时钟。

```
sys_stm32_clock_init(85, 2, 2, 4, 8);
```

设置系统时钟，利用 PLL 倍频参数，将主频配置为 170MHz，保证 CPU 及外设高速稳

定运行。

(3) 初始化延时功能。

```
delay_init(170);
```

根据系统主频 170MHz 初始化延时模块,确保 delay_ms 能实现准确的毫秒延时。

(4) 初始化 LED 引脚。

```
led_init();
```

配置 LED0 与 LED1 对应的 GPIO 为输出模式,为后续控制 LED 亮灭做准备。

(5) 运行主循环(跑马灯)。

```
while (1)
{
    LED0(0);    //LED0 亮
    LED1(1);    //LED1 灭
    delay_ms(500);

    LED0(1);    //LED0 灭
    LED1(0);    //LED1 亮
    delay_ms(500);
}
```

进入无限循环,交替点亮和熄灭两个 LED。

LEDx(0)点亮 LED0,LEDx(1)熄灭 LED1,状态保持 500ms。

交替闪烁形成跑马灯效果。

这样,整段程序通过初始化和运行主循环,驱动两个 LED 交替闪烁,演示了 STM32 的基本 IO 控制和系统时钟配置流程。

该主程序利用 STM32G474 微控制器的 GPIO 控制功能,结合系统时钟初始化与延时管理,实现两个 LED 交替点亮的简单跑马灯效果。程序结构清晰,调用了硬件抽象层初始化、时钟配置、延时初始化和 LED 引脚初始化等功能模块,展示了嵌入式系统中常见的基本硬件控制流程。

4. 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 G:\程序代码\3-1 跑马灯实例\Projects\MDK-Arm 文件夹下的工程文件 atk_g474.uvprojx,如图 3-7 所示。



图 3-7 打开工程

（2）编译 Keil MDK 工程。

在 Keil MDK 开发环境中通过菜单 Project→Rebuild all target files 或工具栏 Rebuild 按钮编译工程，如图 3-8 所示。

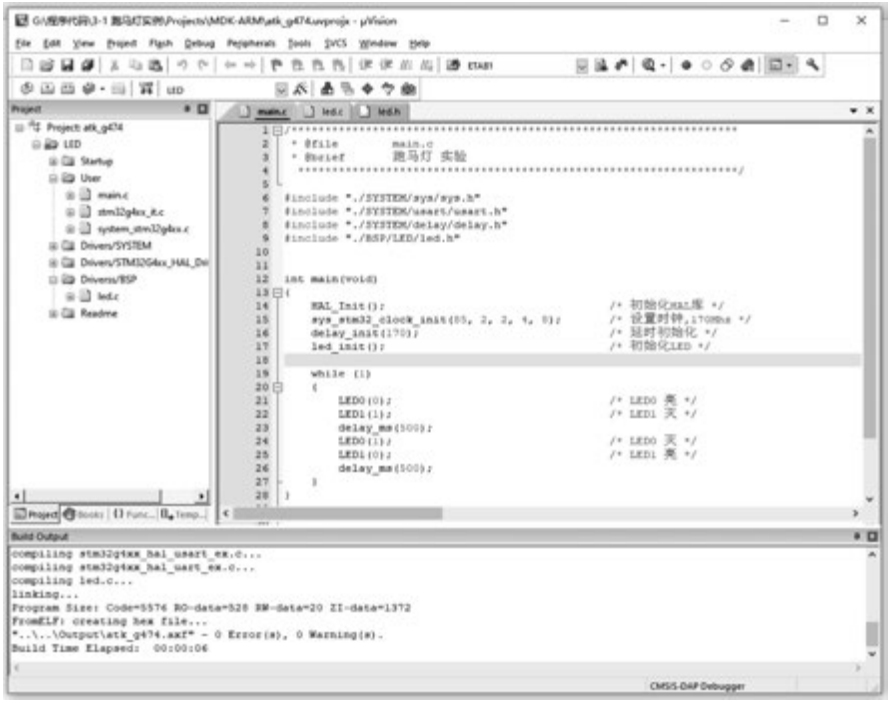

图 3-8 编译 Keil MDK 工程

（3）配置 Keil MDK 工程与 Flash Download 选项。

在 Keil MDK 开发环境中通过菜单 Project→Options for Target 或工具栏配置工程，如图 3-9 所示。

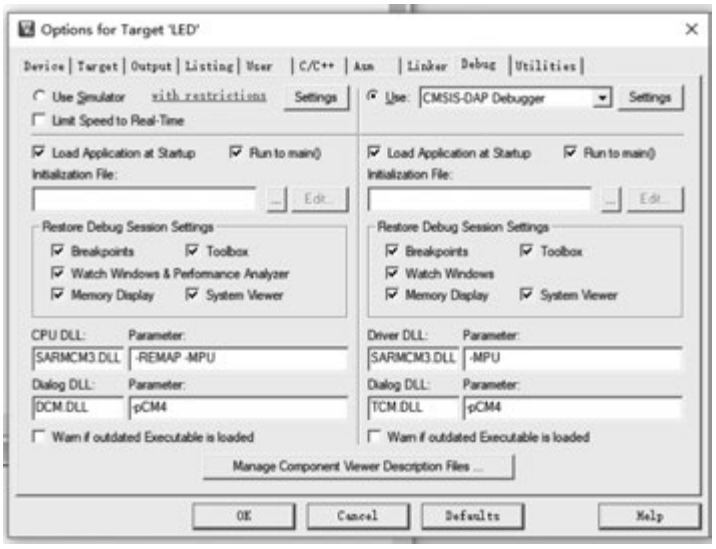

图 3-9 配置 Keil MDK 工程

打开 Debug 选项卡，选择使用的仿真器 CMSIS-DAP Debugger。在 Flash Download 的 Download Function 下勾选 Reset and Run 选项。单击 OK 完成配置，如图 3-10 所示。

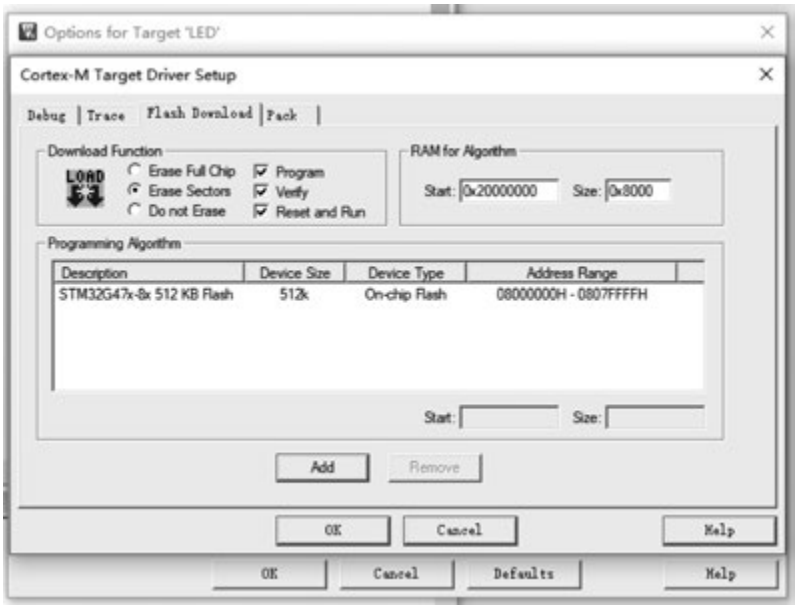

图 3-10　配置 Flash Download 选项

(4) 下载工程。

连接好仿真器，给开发板上电。

在 Keil MDK 开发环境中通过菜单 Flash→Download 或工具栏下载工程，如图 3-11 所示。



图 3-11　下载工程

工程下载成功提示如图 3-12 所示。



图 3-12　工程下载成功提示

从编译信息中可以看到，代码编译无任何错误和警告。程序的资源占用情况如下。

Code=5576：代表程序代码的大小，单位通常是字节(Bytes)。这里是指，程序的机器代码和常量数据占用了 5576 字节的闪存空间。

RO-data=528：只读数据(Read-Only Data)，即常量数据(例如 const 修饰的变量)，这

里是指，数据存储在闪存中，占用了 528 字节。

RW-data＝20：可读写数据(Read-Write Data)，即已初始化的全局变量和静态变量，这里是指，数据在程序开始运行时会从闪存复制到 RAM，占用了 20 字节的 RAM 空间。

ZI-data＝1372：零初始化数据(Zero-Initialized Data)，即未初始化的全局变量和静态变量(如未赋值的全局变量)，这里是指，变量在运行时被初始化为零，占用了 1372 字节的 RAM 空间。

需要特别注意的是，程序实际占用的 Flash 大小是 Code＋RO-data 的总和，而不是 .hex 文件的大小。

完成编译后，接下来就可以使用调试器进行程序下载和验证。这里推荐使用 DAP 仿真器(也可以使用其他支持的调试器)进行烧录。

下载成功后，可以观察到 LED0 和 LED1 交替闪烁，实现了预期的跑马灯效果。

3.6 采用 STM32G474 和 HAL 库的 GPIO 输入应用实例

本实例基于 STM32G474 微控制器，实现通过三个按键控制 LED 的翻转功能。硬件采用配置为内部上拉、高速输入模式的 GPIO 按键电路，软件设计包括按键初始化、状态扫描及消抖处理，确保按键输入稳定可靠。扫描函数支持多按键优先级机制和单次/连续触发模式，提升应用灵活性。主程序通过检测按键状态，实现 KEY0 控制 LED0、KEY1 控制 LED1 翻转、KEY2 控制两个 LED 同步翻转的功能。开发环境为 Keil MDK，程序结构合理，响应稳定、无误触发，适合嵌入式入门学习和实际按键控制应用，软硬件接口设计合理。

3.6.1 STM32G474 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，按键开关不会立刻稳定接通，也不会立刻完全断开，使用按键时会产生抖动信号，需要进行软件消抖处理。本实例开发板连接的按键检测电路如图 3-13 所示。

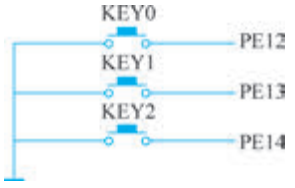

图 3-13 按键检测电路

本实例基于图 3-13 所示的电路设计程序，通过开发板上的三个独立按键控制 LED 灯：KEY0 控制 LED0 翻转，KEY1 控制 LED1 翻转，KEY2 控制 LED0、LED1 同时翻转。

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

3.6.2 STM32G474 的 GPIO 输入应用软件设计

基于 STM32G474 微控制器的按键输入驱动系统通过 key.h 和 key.c 文件实现了对三个按键(KEY0、KEY1、KEY2)的 GPIO 硬件资源管理、初始化配置及状态扫描功能。key.h 中定义了按键对应的 GPIO 端口和引脚，并提供时钟使能及状态读取的宏接口，同时声明了按键初始化和扫描函数。key.c 实现了按键初始化，将相关 GPIO 引脚配置为输入模式、内部上拉及高速状态；按键扫描函数支持消抖处理，防止多次误触发，并设置了按键优先级机制，保证多按键同时按下时返回优先级最高的键值。扫描函数还分别支持单次触发和连续触发两种模式，灵活满足不同应用需求。

在主程序 main.c 中，完成系统时钟、HAL 库、延时、LED 和按键模块的初始化后，程序进入主循环持续检测按键状态。根据按键输入，实现 LED 状态的切换：KEY0 控制 LED0，KEY1 控制 LED1，KEY2 则使两个 LED 同时切换。程序采用按键消抖和延时机制，保证响应稳定且避免误触发。通过 Keil MDK 开发环境完成编译、下载与仿真，调试验证过程中，按键操作与 LED 状态变化完全符合预期效果，证明代码设计合理且功能稳定。该方案为嵌入式系统按键输入控制提供了结构清晰、易维护的软硬件实现方案，适合教学和实际应用。

1. key.h

```
/*
 *@file         key.h
 *@brief        按键输入 驱动代码
 */
#ifndef __KEY_H
#define __KEY_H
#include "./SYSTEM/sys/sys.h"
/*引脚定义*/
#define KEY0_GPIO_PORT                  GPIOE
#define KEY0_GPIO_PIN                   GPIO_PIN_12
#define KEY0_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
/*PE口时钟使能*/

#define KEY1_GPIO_PORT                  GPIOE
#define KEY1_GPIO_PIN                   GPIO_PIN_13
#define KEY1_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
/*PE口时钟使能*/

#define KEY2_GPIO_PORT                  GPIOE
#define KEY2_GPIO_PIN                   GPIO_PIN_14
#define KEY2_GPIO_CLK_ENABLE()          do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
/*PE口时钟使能*/
#define KEY0      HAL_GPIO_ReadPin(KEY0_GPIO_PORT, KEY0_GPIO_PIN)   /*读取KEY0引脚*/
#define KEY1      HAL_GPIO_ReadPin(KEY1_GPIO_PORT, KEY1_GPIO_PIN)   /*读取KEY1引脚*/
#define KEY2      HAL_GPIO_ReadPin(KEY2_GPIO_PORT, KEY2_GPIO_PIN)   /*读取KEY2引脚*/

#define KEY0_PRES     1             /*KEY0按下*/
#define KEY1_PRES     2             /*KEY1按下*/
#define KEY2_PRES     3             /*KEY2按下*/

void key_init(void);                /*按键初始化函数*/
uint8_t key_scan(uint8_t mode);     /*按键扫描函数*/

#endif
```

key.h 头文件是针对 STM32G474 微控制器设计的按键输入驱动接口，功能说明如下：

(1) 定义按键对应的 GPIO 端口和引脚。

为三个按键 KEY0、KEY1、KEY2 分别定义 GPIO 端口(均为 GPIOE)及对应的引脚宏(GPIO_PIN_12、GPIO_PIN_13、GPIO_PIN_14)，方便代码中直接使用宏名称引用按键硬件资源。

(2) 定义 GPIO 端口时钟使能宏。

为每个按键定义对应 GPIO 端口时钟开启宏(KEYx_GPIO_CLK_ENABLE())，保证

按键对应的 GPIO 端口时钟被使能，GPIO 引脚可正常使用。

(3) 定义读取按键状态的宏。

通过 KEY0、KEY1、KEY2 宏直接调用 HAL_GPIO_ReadPin 读取对应引脚的电平，实现对按键当前状态的快速读取。

(4) 定义按键状态宏。

定义了宏 KEY0_PRES、KEY1_PRES、KEY2_PRES 对应按键被按下时的返回值(分别为 1、2、3)，用于程序中判定哪个按键被按下。

(5) 声明按键初始化及扫描函数接口。

void key_init(void)：用于初始化按键所需的 GPIO 配置(输入模式及相关设置)。

uint8_t key_scan(uint8_t mode)：扫描按键状态的函数，带参数控制扫描方式(具体实现细节在 key.c 中)。

key.h 通过统一管理按键硬件资源定义、提供读取状态的宏接口，并声明初始化与扫描函数，方便主程序实现按键输入功能，增强代码的可读性和可维护性。

2. key.c

```
/*
 *@file         key.c
 *@brief        按键输入 驱动代码
 */
#include "./BSP/KEY/key.h"
#include "./SYSTEM/delay/delay.h"
/**
 *@brief        按键初始化函数
 *@param        无
 *@retval       无
 */
void key_init(void)
{
    GPIO_InitTypeDef gpio_init_struct;                  /*GPIO 配置参数存储变量*/
    KEY0_GPIO_CLK_ENABLE();                             /*KEY0 时钟使能*/
    KEY1_GPIO_CLK_ENABLE();                             /*KEY1 时钟使能*/
    KEY2_GPIO_CLK_ENABLE();                             /*KEY2 时钟使能*/

    gpio_init_struct.Pin = KEY0_GPIO_PIN;               /*KEY0 引脚*/
    gpio_init_struct.Mode = GPIO_MODE_INPUT;            /*输入*/
    gpio_init_struct.Pull = GPIO_PULLUP;                /*上拉*/
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /*高速*/
    HAL_GPIO_Init(KEY0_GPIO_PORT, &gpio_init_struct);   /*KEY0 引脚模式设置,上拉输入*/

    gpio_init_struct.Pin = KEY1_GPIO_PIN;               /*KEY1 引脚*/
    gpio_init_struct.Mode = GPIO_MODE_INPUT;            /*输入*/
    gpio_init_struct.Pull = GPIO_PULLUP;                /*上拉*/
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /*高速*/
    HAL_GPIO_Init(KEY1_GPIO_PORT, &gpio_init_struct);   /*KEY1 引脚模式设置,上拉输入*/

    gpio_init_struct.Pin = KEY2_GPIO_PIN;               /*KEY2 引脚*/
    gpio_init_struct.Mode = GPIO_MODE_INPUT;            /*输入*/
    gpio_init_struct.Pull = GPIO_PULLUP;                /*上拉*/
    gpio_init_struct.Speed = GPIO_SPEED_FREQ_HIGH;      /*高速*/
```

```
    HAL_GPIO_Init(KEY2_GPIO_PORT, &gpio_init_struct);  /* KEY2 引脚模式设置,上拉输入 */
}

/**
 * @brief       按键扫描函数
 * @note        该函数有子优先级(同时按下多个按键): KEY0 > KEY1 > KEY2!!
 * @param       mode:0 / 1, 具体含义如下。
 *    @arg      0,  不支持连续按(当按键按下不放时, 只有第一次调用会返回键值,
 *                  必须松开以后, 再次按下才会返回其他键值)
 *    @arg      1,  支持连续按(当按键按下不放时, 每次调用该函数都会返回键值)
 * @retval      键值, 定义如下。
 *              KEY0_PRES, 1, KEY0 按下
 *              KEY1_PRES, 2, KEY1 按下
 *              KEY2_PRES, 3, KEY2 按下
 */
uint8_t key_scan(uint8_t mode)
{
    static uint8_t key_up = 1;          /* 按键松开标志 */
    uint8_t keyval = 0;

    if (mode) key_up = 1;               /* 支持连按 */

    if (key_up && (KEY0 == 0 || KEY1 == 0 || KEY2 == 0))  /* 按键松开标志为 1, 且有任意一
个按键按下 */
    {
        delay_ms(10);                   /* 去抖动 */
        key_up = 0;
        if (KEY0 == 0)  keyval = KEY0_PRES;
        if (KEY1 == 0)  keyval = KEY1_PRES;
        if (KEY2 == 0)  keyval = KEY2_PRES;
    }
    else if (KEY0 == 1 && KEY1 == 1 && KEY2 == 1)  /* 没有任何按键按下, 标记按键松开 */
    {
        key_up = 1;
    }

    return keyval;                      /* 返回键值 */
}
```

key.c 代码实现了 STM32G474 微控制器上的按键驱动功能,主要包括按键的初始化和扫描。详细说明如下。

(1) 按键初始化函数 key_init()。

负责配置按键所连接的 GPIO 引脚为输入模式。

依次为 KEY0、KEY1、KEY2 所用的 GPIO 端口使能时钟(此处均为 GPIOE)。

设置每个按键对应引脚为输入模式(GPIO_MODE_INPUT)、内部上拉电阻(GPIO_PULLUP)、高速 I/O 速度(GPIO_SPEED_FREQ_HIGH)。

通过调用 HAL_GPIO_Init()完成引脚初始化,保证 MCU 可以正确读取按键电平状态。

(2) 按键扫描函数 key_scan(uint8_t mode)。

用于检测按键是否被按下,并返回对应按键的标识值。

函数内部采用静态变量 key_up 标记按键释放状态,防止按键多次触发(去抖动及按键

状态控制)。

按键优先级：KEY0>KEY1>KEY2，即如果多个按键同时按下，返回优先级最高的按键值。

参数 mode 用来控制是否支持连续按：

当 mode=0 时，只在按键第一次按下时返回值，按键被释放后才能再次触发；

当 mode=1 时，支持连续按，按键按下时每次调用都返回按键值。

检测逻辑：

当 key_up 为 1 且检测到任一按键电平为低(按下有效电平假设为 0)时，延时 10ms 消除抖动，更新 key_up 为 0 并记录按下键值；

当所有按键电平均为高时(抬起状态)，重置 key_up 为 1，允许下一次检测。

返回当前检测到的按键值，未检测到按键则返回 0。

这是嵌入式系统中常见的按键驱动基础代码，为用户交互功能提供了软硬件接口支持。

3. main.c

```
/*
 *@file        main.c
 *@brief       按键输入实验
 */
#include "./SYSTEM/sys/sys.h"
#include "./SYSTEM/usart/usart.h"
#include "./SYSTEM/delay/delay.h"
#include "./BSP/LED/led.h"
#include "./BSP/KEY/key.h"

int main(void)
{
    uint8_t key;

    HAL_Init();                             /*初始化 HAL 库*/
    sys_stm32_clock_init(85, 2, 2, 4, 8);   /*设置时钟,170MHz*/
    delay_init(170);                        /*延时初始化*/
    led_init();                             /*初始化 LED*/
    key_init();                             /*初始化按键*/

    while (1)
    {
        key = key_scan(0);                  /*得到键值*/

        if (key)
        {
            switch (key)
            {
                case KEY0_PRES:             /*控制 LED0(RED)翻转*/
                    LED0_TOGGLE();          /*LED0 状态取反*/
                    break;

                case KEY1_PRES:             /*控制 LED1(GREEN)翻转*/
                    LED1_TOGGLE();          /*LED1 状态取反*/
                    break;
```

```
                case KEY2_PRES:              /* 同时控制 LED0,LED1 翻转 */
                    LED0_TOGGLE();           /* LED0 状态取反 */
                    LED1_TOGGLE();           /* LED1 状态取反 */
                    break;
                default : break;
            }
        }
        else
        {
            delay_ms(10);
        }
    }
}
```

该 STM32G474 微控制器主程序 main.c 功能和执行过程的详细说明如下。

本程序实现了通过三个按键控制两个 LED 指示灯状态切换的功能：

按下 KEY0 时，LED0(红灯)状态取反(亮灭切换)；

按下 KEY1 时，LED1(绿灯)状态取反；

按下 KEY2 时，LED0 和 LED1 状态同时取反。

执行过程详细说明如下。

(1) 系统初始化。

HAL_Init()：初始化 HAL 库，配置 MCU 基础外设和中断。

sys_stm32_clock_init(85,2,2,4,8)：设置系统时钟，配置主频为 170MHz，保证高速运行。

delay_init(170)：根据 170MHz 主频初始化延时模块，确保延时函数精确。

led_init()：初始化 LED 所用的 GPIO，引脚配置为推挽输出，上电时两个 LED 均关闭。

key_init()：初始化按键所用的 GPIO，引脚配置为上拉输入，确保可以读取按键状态。

(2) 主循环。

进入无限循环 while(1)，持续检测按键输入。

调用 key_scan(0)函数扫描按键状态。

按键处理逻辑为若 key 不为零(表示有按键按下)，则进入 switch 判断：

KEY0_PRES(1)表示调用 LED0_TOGGLE()，切换 LED0 状态(亮→灭或灭→亮)；

KEY1_PRES(2)表示调用 LED1_TOGGLE()，切换 LED1 状态；

KEY2_PRES(3)表示同时调用 LED0_TOGGLE()和 LED1_TOGGLE()，两个 LED 同时状态取反。

若无按键按下，执行 delay_ms(10)，短暂停顿，降低 CPU 资源占用。

该主程序架构简洁清晰，实现了基于 STM32G474 的按键控制 LED 的功能：

(1) 完善初始化(涵盖 HAL 库、时钟、延时、LED、按键)；

(2) 循环检测按键，兼顾消抖和子优先级；

(3) 根据不同按键输入触发对应 LED 的状态切换，直观反映按键功能。

4. Keil MDK 工程搭建与下载

通过 Keil MDK 搭建与下载工程的步骤如下。

(1) 打开工程。

打开 G:\程序代码\3-2 按键输入实例\Projects\MDK-Arm 文件夹下的工程文件 atk_g474.uvprojx,如图 3-14 所示。



图 3-14 打开工程

(2) 编译 Keil MDK 工程。

在 Keil MDK 开发环境中通过菜单 Project→Rebuild all target files 或工具栏 Rebuild 按钮编译工程,如图 3-15 所示。

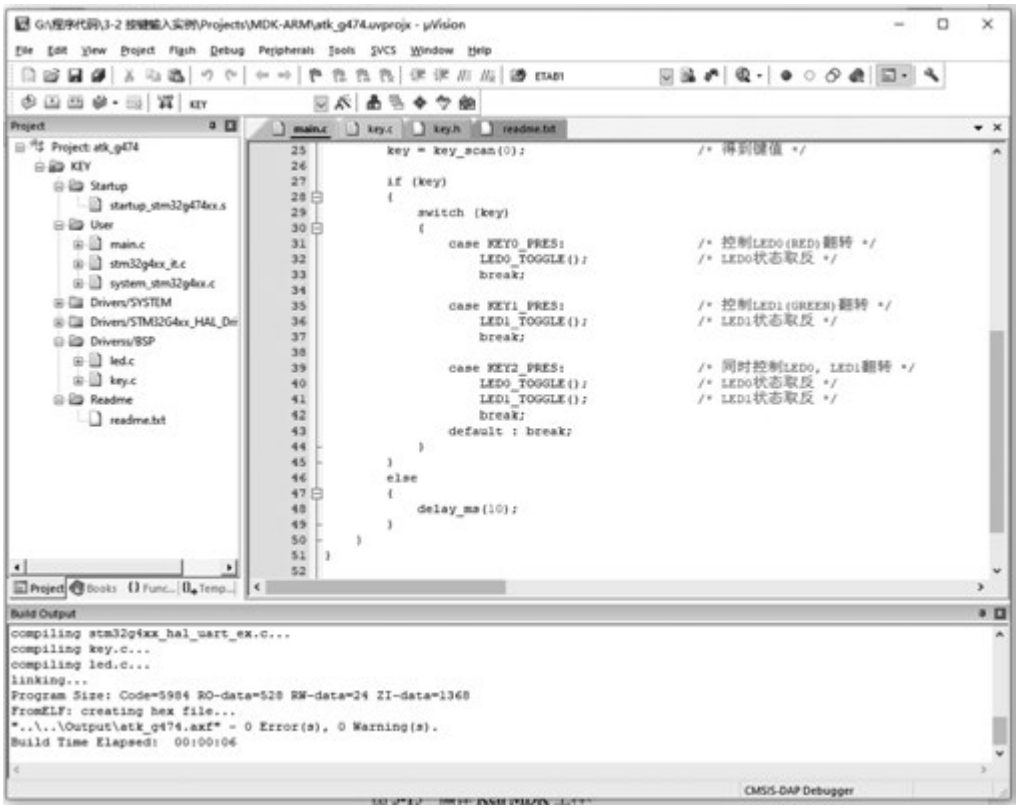

图 3-15 编译 Keil MDK 工程

(3) 配置 Keil MDK 工程与 Flash Download 选项。

在 Keil MDK 开发环境中通过菜单 Project→Options for Target 或工具栏配置工程。

在 Flash Download 的 Download Function 下勾选 Reset and Run 选项。单击 OK 完成配置。

(4) 下载工程。

连接好仿真器,给开发板上电。

在 Keil MDK 开发环境中通过菜单 Flash→Download 或工具栏下载工程。

完成程序编译与烧录后,观察到板载 LED(即 LED0、LED1)初始处于熄灭状态。此时,按下并松开一次 KEY0,LED0 的亮灭状态随之翻转一次;按下并松开一次 KEY1,LED1 的亮灭状态也相应翻转一次;而按下并松开一次 KEY2 时,LED0 和 LED1 则同时翻转。以上实验现象与程序设计的预期效果完全一致,充分证明了程序的正确性和稳定性。