第十三章 跑马灯实验

本章将通过一个经典的跑马灯程序，带大家开启 STM32F103 之旅。通过本章的学习，我们将了解到 STM32F103 的 IO 口作为输出使用的方法。我们将通过代码控制开发板上的 LED 灯：LED0、LED1 交替闪烁，实现类似跑马灯的效果。

本章分为如下 4 个小节：

13.1 STM32F1 GPIO 简介
13.2 硬件设计
13.3 程序设计
13.4 下载验证

13.1 STM32F1 GPIO 简介

GPIO 是控制或者采集外部器件的信息的外设，即负责输入输出。它按组分配，每组 16 个 IO 口，组数视芯片而定。STM32F103ZET6 芯片是 144 脚的芯片，具有 GPIOA、GPIOB、GPIOC、GPIOD、GPIOE、GPIOF 和 GPIOG 七组 GPIO 口，共有 112 个 IO 口可供我们编程使用。这里重点说一下 STM32F103 的 IO 电平兼容性问题，STM32F103 的绝大部分 IO 口，都兼容 5V，至于到底哪些是兼容 5V 的，请看 STM32F103xE 的数据手册（注意是数据手册，不是中文参考手册），见表 5 大容量 STM32F103xx 引脚定义，凡是有 FT 标志的，都是兼容 5V 电平的 IO 口，可以直接接 5V 的外设（注意：如果引脚设置的是模拟输入模式，则不能接 5V！），凡是不带 FT 标志的，就建议大家不要接 5V 了，可能烧坏 MCU。

13.1.1 GPIO 功能模式

GPIO 有八种工作模式，分别是：

1、输入浮空 （上拉/下拉电阻为断开状态，该模式下 IO 口的电平完全是由外部电路决定）
2、输入上拉 上拉电阻接通（接VDD）
3、输入下拉 下拉电阻接通（接VSS）
4、模拟输入 上下拉电阻断开，施密特触发器关闭，双 MOS 管也关闭。其他外设可以通过模拟通道输入输出。（一个绕过所有的“直通车”）
5、开漏输出
6、推挽输出
7、开漏式复用功能 选择复用功能时，引脚的状态由对应的外设控制，而不是输出数据寄存器。
8、推挽式复用功能 复用输出和普通输出的根本区别：IO口电平信号控制权在谁的问题。

P-MOS常闭（高阻态/低电平）
P-MOS,N-MOS一开一关

13.1.2 GPIO 基本结构分析

我们知道了 GPIO 有八种工作模式，具体这些模式是怎么实现的？下面我们通过 GPIO 的基本结构图来分别进行详细分析，先看看总的框图，如图 13.1.2.1 所示。

寄存器：

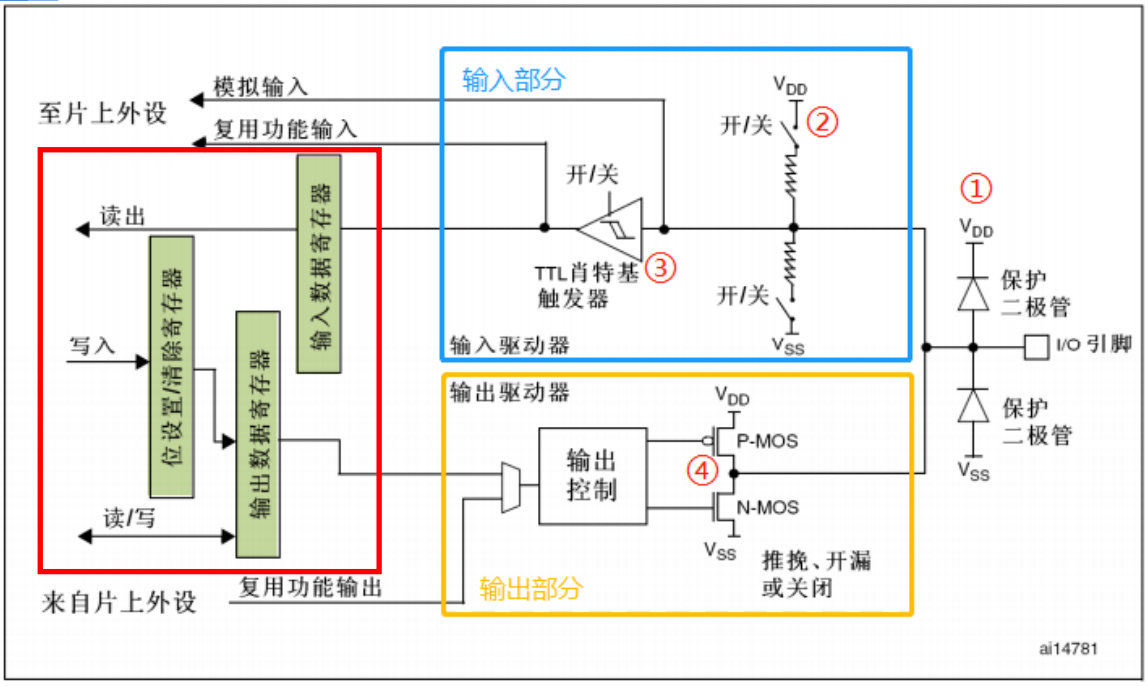

图 13.1.2.1 GPIO 的基本结构图

如上图所示，可以看到右边只有 I/O 引脚，这个 I/O 引脚就是我们可以看到的芯片实物的引脚，其他部分都是 GPIO 的内部结构。

① 保护二极管

保护二极管共有两个，用于保护引脚外部过高或过低的电压输入。当引脚输入电压高于 VDD 时，上面的二极管导通，当引脚输入电压低于 VSS 时，下面的二极管导通，从而使输入芯片内部的电压处于比较稳定的值。虽然有二极管的保护，但这样的保护却很有限，大电压大电流的接入很容易烧坏芯片。所以在实际的设计中我们要考虑设计引脚的保护电路。

② 上拉、下拉电阻

它们阻值大概在 30~50K 欧之间，可以通过上、下两个对应的开关控制，这两个开关由寄存器控制。当引脚外部的器件没有干扰引脚的电压时，即没有外部的上、下拉电压，引脚的电平由引脚内部上、下拉决定，开启内部上拉电阻工作，引脚电平为高，开启内部下拉电阻工作，则引脚电平为低。同样，如果内部上、下拉电阻都不开启，这种情况就是我们所说的浮空模式。浮空模式下，引脚的电平是不可确定的。引脚的电平可以由外部的上、下拉电平决定。需要注意的是，STM32 的内部上拉是一种“弱上拉”，这样的上拉电流很弱，如果有要求大电流还是得外部上拉。

③ 施密特触发器

对于标准施密特触发器，当输入电压高于正向阈值电压，输出为高；当输入电压低于负向阈值电压，输出为低；当输入在正负向阈值电压之间，输出不改变，也就是说输出由高电准位翻转为低电准位，或是由低电准位翻转为高电准位对应的阈值电压是不同的。只有当输入电压发生足够的变化时，输出才会变化，因此将这种元件命名为触发器。这种双阈值动作被称为迟滞现象，表明施密特触发器有记忆性。从本质上来说，施密特触发器是一种双稳态多谐振荡器。

施密特触发器可作为波形整形电路，能将模拟信号波形整形为数字电路能够处理的方波波形，而且由于施密特触发器具有滞回特性，所以可用于抗干扰，以及在闭回路正回授/负回授配置中用于实现多谐振荡器。

下面看看比较器跟施密特触发器的作用的比较，就清楚的知道施密特触发器对外部输入信号具有一定抗干扰能力，如图 13.1.2.2 所示。

高电平时，看低阈值。

低电平时，看高阈值。

过阈值变电压。

两阈值之间维持之前的状态

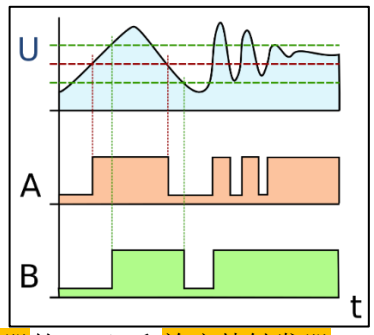

图 13.1.2.2 比较器的（A）和施密特触发器（B）作用比较

④ P-MOS 管和 N-MOS 管

这个结构控制 GPIO 的开漏输出和推挽输出两种模式。开漏输出：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。推挽输出：这两只对称的 MOS 管每次只有一只导通，所以导通损耗小、效率高。输出既可以向负载灌电流，也可以从负载拉电流。推拉式输出既能提高电路的负载能力，又能提高开关速度。

上面我们对 GPIO 的基本结构图中的关键器件做了介绍，下面分别介绍 GPIO 八种工作模式对应结构图的工作情况。

1、输入浮空

输入浮空模式：上拉/下拉电阻为断开状态，施密特触发器打开，输出被禁止。输入浮空模式下，IO 口的电平完全是由外部电路决定。如果 IO 引脚没有连接其他的设备，那么检测其输入电平是不确定的。该模式可以用于按键检测等场景。

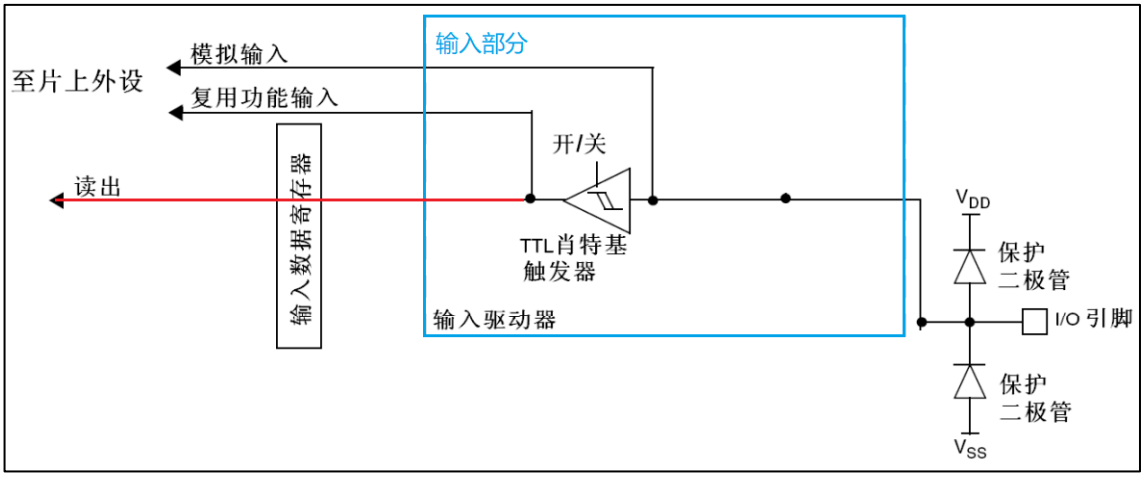

图 13.1.2.3 输入浮空模式

2、输入上拉

输入上拉模式：上拉电阻导通，施密特触发器打开，输出被禁止。在需要外部上拉电阻的时候，可以使用内部上拉电阻，这样可以节省一个外部电阻，但是内部上拉电阻的阻值较大，所以只是“弱上拉”，不适合做电流型驱动。

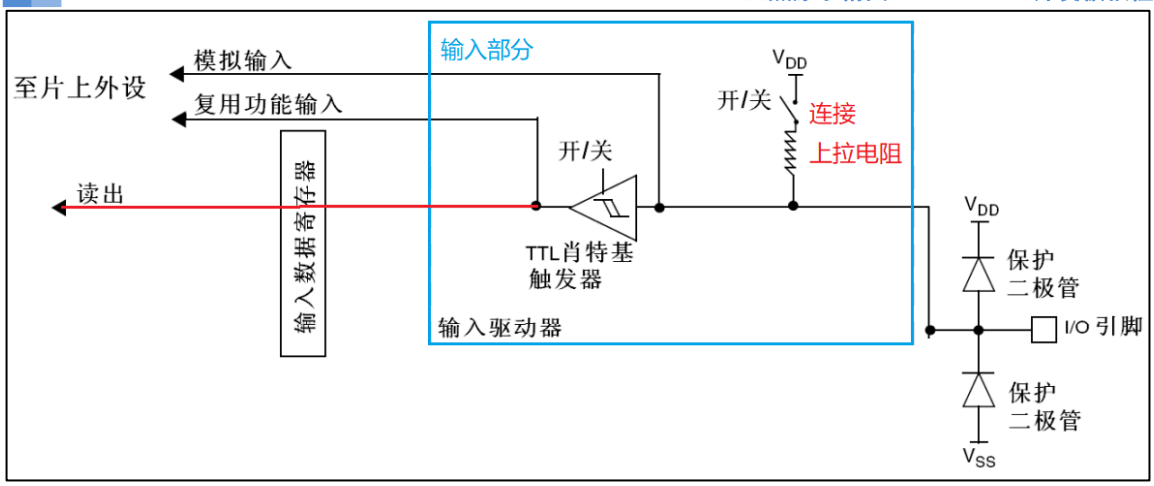

图 13.1.2.4 输入上拉模式

3、输入下拉

输入下拉模式：下拉电阻导通，施密特触发器打开，输出被禁止。在需要外部下拉电阻的时候，可以使用内部下拉电阻，这样可以节省一个外部电阻，但是内部下拉电阻的阻值较大，所以不适合做电流型驱动。

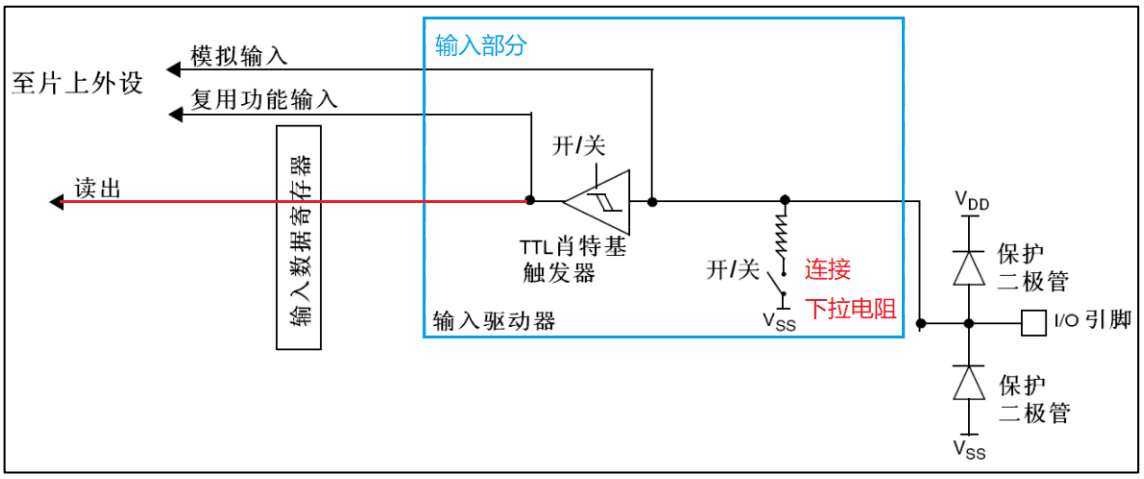

图 13.1.2.5 输入下拉模式

4、模拟功能

近乎全部关闭

模拟功能：上下拉电阻断开，施密特触发器关闭，双 MOS 管也关闭。其他外设可以通过模拟通道输入输出。该模式下需要用到芯片内部的模拟电路单元单元，用于 ADC、DAC、MCO 这类操作模拟信号的外设。

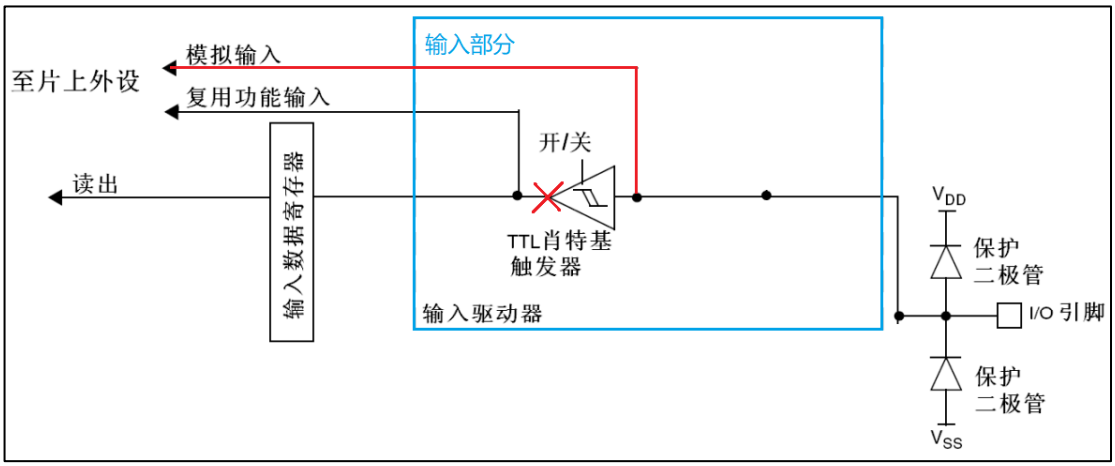

图 13.1.2.6 模拟功能

高阻态：P-MOS与N-MOS都关闭，对外表现为高电阻几近绝缘状态，引脚上的电压完全由外部电路决定，芯片内部不施加任何影响。
注意：输出高阻态本身并不决定外部信号能否进入输入电路，它只是保证了输出部分不会“捣乱”（和输入是两个概念）

P-MOS一直关闭，N-MOS控制。
输出：低电平/高阻态

5、开漏输出

开漏输出模式：STM32 的开漏输出模式是数字电路输出的一种，从结果上看它只能输出低电平 Vss 或者高阻态，常用于 IIC 通讯（IIC_SDA）或其它需要进行电平转换的场景。根据《STM32F10xxx 参考手册_V10（中文版）.pdf》第 108 页关于“GPIO 输出配置”的描述，我们可以知道开漏模式下，IO 是这样工作的：

- P-MOS 被“输出控制”控制在截止状态，因此 IO 的状态取决于 N-MOS 的导通状况；
- 只有 N-MOS 还受控制于输出寄存器，“输出控制”对输入信号进行了逻辑非的操作；
- 施密特触发器是工作的，即可以输入，且上下拉电阻都断开了，可以看成浮空输入；

根据参考手册的描述，同时为了方便大家理解，我们在“输出控制”部分做了等效处理，如图 13.1.2.7 所示。图 13.1.2.7 中写入输出数据寄存器①的值怎么对应到 IO 引脚的输出状态②是我们最关心的。

根据参考手册的描述：开漏输出模式下 P-MOS 一直在截止状态，即不导通，所以 P-MOS 管的栅极相当于一直接 V_{DD}。如果输出数据寄存器①的值为 0，那么 IO 引脚的输出状态②为低电平，这是我们需要的控制逻辑，怎么做到的呢？是这样的，输出数据寄存器的逻辑 0 经过“输出控制”的取反操作后，输出逻辑 1 到 N-MOS 管的栅极，这时 N-MOS 管就会导通，使得 IO 引脚连接到 V_{SS}，即输出低电平。如果输出数据寄存器的值为 1，经过“输出控制”的取反操作后，输出逻辑 0 到 N-MOS 管的栅极，这时 N-MOS 管就会截止。又因为 P-MOS 管是一直截止的，使得 IO 引脚呈现高阻态，即不输出低电平，也不输出高电平。因此要 IO 引脚输出高电平就必须接上拉电阻。 又由于 F1 系列的开漏输出模式下，内部的上下拉电阻不可用，所以只能通过接芯片外部上拉电阻的方式，实现开漏输出模式下输出高电平。如果芯片外部不接上拉电阻，那么开漏输出模式下，IO 无法输出高电平。

在开漏输出模式下，施密特触发器是工作的，所以 IO 口引脚的电平状态会被采集到输入数据寄存器中，如果对输入数据寄存器进行读访问可以得到 IO 口的状态。也就是说开漏输出模式下，我们可以读取 IO 引脚状态。

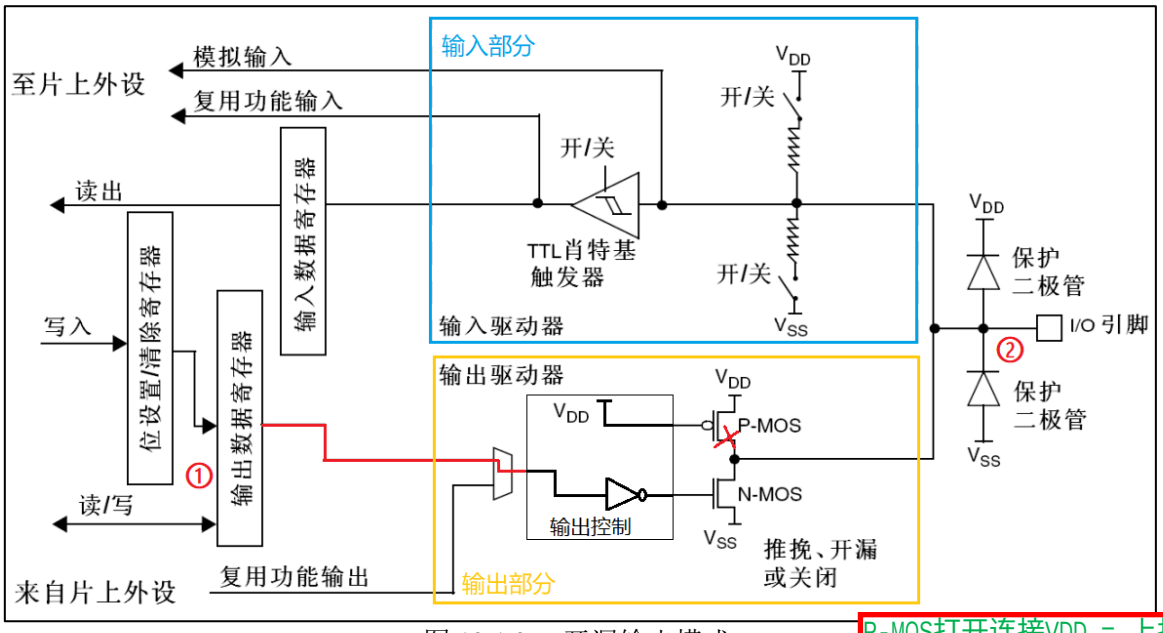

图 13.1.2.7 开漏输出模式

P-MOS打开连接VDD = 上拉
N-MOS打开连接VSS = 下拉

6、推挽输出

推挽输出模式：STM32 的推挽输出模式，从结果上看它会输出低电平 V_{SS} 或者高电平 V_{DD}。推挽输出跟开漏输出不同的是，推挽输出模式 P-MOS 管和 N-MOS 管都用上。同样地，我们根据参考手册推挽模式的输出描述，可以得到等效原理图，如图 13.1.2.8 所示。根据手册描述可以把“输出控制”简单地等效为一个非门。

推挽输出 = 你开我关，你关我开

如果输出数据寄存器①的值为 0，经过“输出控制”取反操作后，输出逻辑 1 到 P-MOS 管的栅极，这时 P-MOS 管就会截止，同时也会输出逻辑 1 到 N-MOS 管的栅极，这时 N-MOS 管就会导通，使得 IO 引脚接到 V_{SS}，即输出低电平。

如果输出数据寄存器的值为 1 ，经过“输出控制”取反操作后，输出逻辑 0 到 N-MOS 管的栅极，这时 N-MOS 管就会截止，同时也会输出逻辑 0 到 P-MOS 管的栅极，这时 P-MOS 管就会导通，使得 IO 引脚接到 V_{DD}，即输出高电平。

由上述可知，推挽输出模式下，P-MOS 管和 N-MOS 管同一时间只能有一个管是导通的。当 IO 引脚在做高低电平切换时，两个管子轮流导通，一个负责灌电流，一个负责拉电流，使其负载能力和开关速度都有较大的提高。

另外在推挽输出模式下，施密特触发器也是打开的，我们可以读取 IO 口的电平状态。

由于推挽输出模式下输出高电平时，是直接连接 V_{DD}，所以驱动能力较强，可以做电流型驱动，驱动电流最大可达 25mA，但是芯片的总电流有限，所以并不建议这样用，最好还是使用芯片外部的电源。

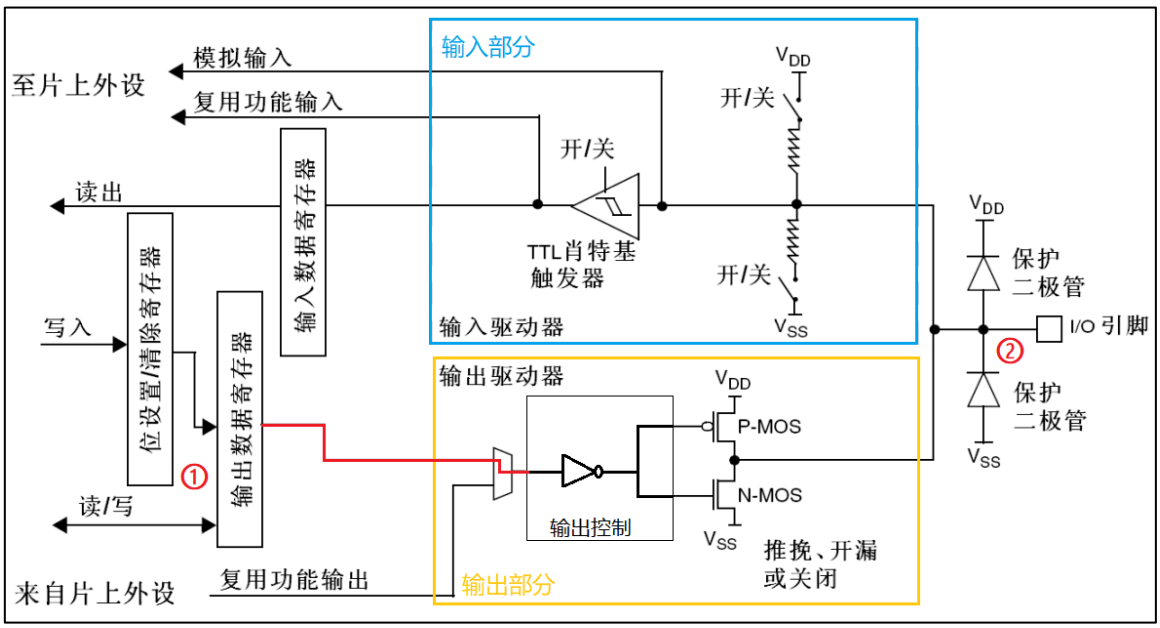

图 13.1.2.8 推挽输出模式

7、开漏式复用功能

开漏式复用功能：一个 IO 口可以是通用的 IO 口功能，还可以是其他外设的特殊功能引脚，这就是 IO 口的复用功能。一个 IO 口可以是多个外设的功能引脚，我们需要选择作为其中一个外设的功能引脚。当选择复用功能时，引脚的状态是由对应的外设控制，而不是输出数据寄存器。除了复用功能外，其他的结构分析请参考开漏输出模式。

另外在开漏式复用功能模式下，施密特触发器也是打开的，我们可以读取 IO 口的电平状态，同时外设可以读取 IO 口的信息。

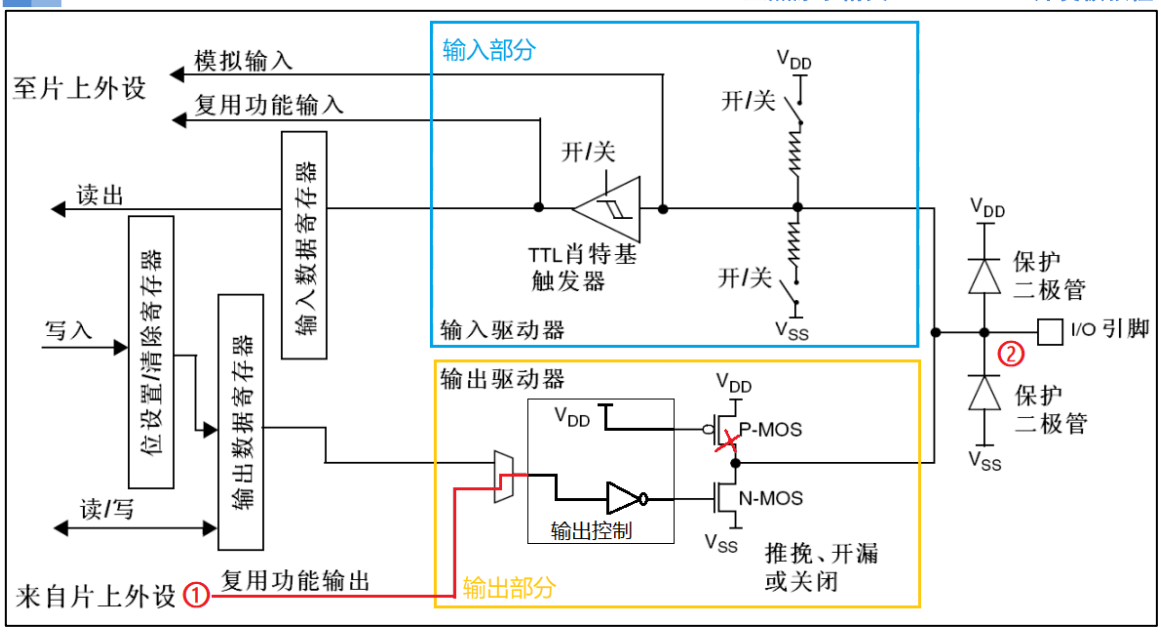

图 13.1.2.9 开漏式复用功能

8、推挽式复用功能

推挽式复用功能：复用功能介绍请查看开漏式复用功能，结构分析请参考推挽输出模式，这里不再赘述。

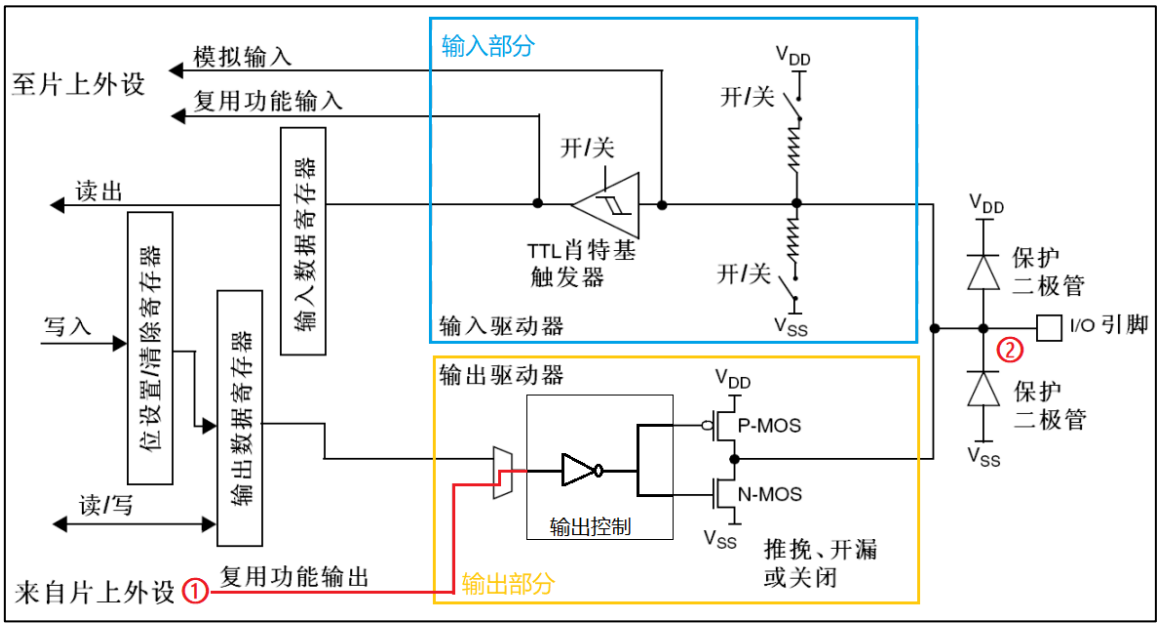

图 13.1.2.10 推挽式复用功能

13.1.3 GPIO 寄存器介绍

STM32F1 每组（这里是 A~D）通用 GPIO 口有 7 个 32 位寄存器控制，包括 ：

2 个 32 位端口配置寄存器（CRL 和 CRH）
2 个 32 位端口数据寄存器（IDR 和 ODR）
1 个 32 位端口置位/复位寄存器（BSRR）
1 个 16 位端口复位寄存器（BRR）
1 个 32 位端口锁定寄存器 (LCKR)

下面我们将带大家理解本章用到的寄存器，没有介绍到的寄存器后面用到会继续介绍。这里主要是带大家学会怎么理解这些寄存器的方法，其他寄存器理解方法是一样的。因为寄存器

太多不可能一个个列出来讲，以后基本就是只会把重要的寄存器拿出来讲述，希望大家尽快培养自己学会看手册的能力。下面先看 GPIO 的 2 个 32 位配置寄存器：

● **端口配置寄存器（GPIOx_CRL 和 GPIOx_CRH）**

这两个寄存器都是 GPIO 口配置寄存器，不过 CRL 控制端口的低八位，CRH 控制端口的高八位。寄存器的作用是控制 GPIO 口的工作模式和工作速度，寄存器描述如图 13.1.3.1 和图 13.1.3.2 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

图 13.1.3.1 GPIOx_CRL 寄存器描述

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

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

图 13.1.3.2 GPIOx_CRH 寄存器描述

每组 GPIO 下有 16 个 IO 口，一个寄存器共 32 位，每 4 个位控制 1 个 IO，所以才需要两个寄存器完成。我们看看这个寄存器的复位值，然后用复位值举例说明一下这样的配置值代表什么意思。比如 GPIOA_CRL 的复位值是 0x44444444，4 位为一个单位都是 0100，以寄存器低四位说明一下，首先位 1：0 为 00 即是设置为 PA0 为输入模式，位 3：2 为 01 即设置为浮空输入模式。所以假如 GPIOA_CRL 的值是 0x44444444，那么 PA0~PA7 都是设置为输入模式，而且是浮空输入模式。

上面这 2 个配置寄存器就是用来配置 GPIO 的相关工作模式和工作速度，它们通过不同的配置组合方法，就决定我们所说的 8 种工作模式。下面，我们来列表阐述，如表 13.1.3.1。

配置模式		CNF1	CNF0	MODE1	MODE0	PxODR寄存器
通用输出	推挽(Push-Pull)	0	0	01 最大输出速度10MHz 10 最大输出速度2MHz 11 最大输出速度50MHz		0 或 1
	开漏(Open-Drain)		1			0 或 1
复用功能输出	推挽(Push-Pull)	1	0			不使用
	开漏(Open-Drain)		1			不使用
输入	模拟输入	0	0	00		不使用
	浮空输入		1			不使用
	下拉输入	1	0			0
	上拉输入					1

表 13.1.3.1 4 个配置寄存器组合下的 8 种工作模式

因为本章需要 GPIO 作为输出口使用，所以我们再来看看端口输出数据寄存器。

- **端口输出数据寄存器（GPIOx_ODR）**

该寄存器用于控制 GPIOx 的输出高电平或者低电平，寄存器描述如图 13.1.3.3 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

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

图 13.1.3.3 GPIOx_ODR 寄存器描述

该寄存器低 16 位有效，分别对应每一组 GPIO 的 16 个引脚。当 CPU 写访问该寄存器，如果对应的某位写 0(ODRy=0)，则表示设置该 IO 口输出的是低电平，如果写 1(ODRy=1)，则表示设置该 IO 口输出的是高电平，y=0~15。

此外，除了 ODR 寄存器，还有一个寄存器也是用于控制 GPIO 输出的，它就是 BSRR 寄存器。

- **端口置位/复位寄存器（GPIOx_BSRR）**

该寄存器也用于控制 GPIOx 的输出高电平或者低电平，寄存器描述如图 13.1.3.4 所示。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
BR15	BR14	BR13	BR12	BR11	BR10	BR9	BR8	BR7	BR6	BR5	BR4	BR3	BR2	BR1	BR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
BS15	BS14	BS13	BS12	BS11	BS10	BS9	BS8	BS7	BS6	BS5	BS4	BS3	BS2	BS1	BS0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

位31:16	**BRy**: 清除端口x的位y (y = 0…15) (Port x Reset bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响　　1：清除对应的ODRy位为0 注：如果同时设置了BSy和BRy的对应位，BSy位起作用。
位15:0	**BSy**: 设置端口x的位y (y = 0…15) (Port x Set bit y) 这些位只能写入并只能以字(16位)的形式操作。 0：对对应的ODRy位不产生影响　　1：设置对应的ODRy位为1

图 13.1.3.4 GPIOx_BSRR 寄存器描述

低16位写1高电平
高16位写1低电平

为什么有了 ODR 寄存器，还要这个 BSRR 寄存器呢？我们先看看 BSRR 的寄存器描述，首先 BSRR 是只写权限，而 ODR 是可读可写权限。BSRR 寄存器 32 位有效，对于低 16 位（0-15），我们往相应的位写 1(BSy=1)，那么对应的 IO 口会输出高电平，往相应的位写 0(BSy=0)，对 IO 口没有任何影响，高 16 位（16-31）作用刚好相反，对相应的位写 1(BRy=1)会输出低电平，写 0(BRy=0)没有任何影响，y=0~15。

也就是说，对于 BSRR 寄存器，你写 0 的话，对 IO 口电平是没有任何影响的。我们要设置某个 IO 口电平，只需要相关位设置为 1 即可。而 ODR 寄存器，我们要设置某个 IO 口电平，我们首先需要读出来 ODR 寄存器的值，然后对整个 ODR 寄存器重新赋值来达到设置某个或者某些 IO 口的目的，而 BSRR 寄存器直接设置即可，这在多任务实时操作系统中作用很大。BSRR 寄存器还有一个好处，就是 BSRR 寄存器改变引脚状态的时候，不会被中断打断，而 ODR 寄存器有被中断打断的风险。

13.2 硬件设计

1. 例程功能

LED 灯：LED0 和 LED1 每过 500ms 一次交替闪烁，实现类似跑马灯的效果。

2. 硬件资源

1）LED 灯

LED0 - PB5
LED1 - PE5

3. 原理图

本章用到的硬件用到 LED 灯：LED0 和 LED1。电路在开发板上已经连接好了，所以在硬件上不需要动任何东西，直接下载代码就可以测试使用。其连接原理图如图 13.2.1 所示：

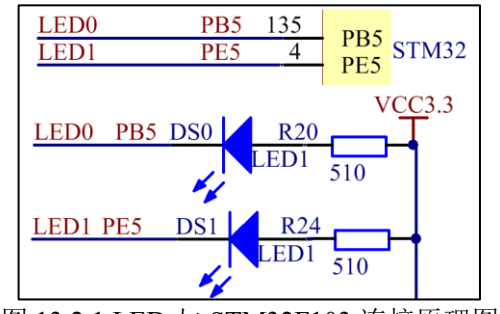

图 13.2.1 LED 与 STM32F103 连接原理图

13.3 程序设计

了解了 GPIO 的结构原理和寄存器，还有我们的实验功能，下面开始设计程序。

13.3.1 GPIO 的 HAL 库驱动分析

HAL 库中关于 GPIO 的驱动程序在 stm32f1xx_hal_gpio.c 文件以及其对应的头文件。

1. HAL_GPIO_Init 函数

要使用一个外设我们首先要对它进行初始化，所以我们先看外设 GPIO 的初始化函数。其声明如下：

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

- **函数描述：**

  用于配置 GPIO 功能模式，还可以设置 EXTI 功能。

- **函数形参：**

  形参 1 是端口号，可以有以下的选择：

```
#define GPIOA                ((GPIO_TypeDef *) GPIOA_BASE)
#define GPIOB                ((GPIO_TypeDef *) GPIOB_BASE)
#define GPIOC                ((GPIO_TypeDef *) GPIOC_BASE)
#define GPIOD                ((GPIO_TypeDef *) GPIOD_BASE)
#define GPIOE                ((GPIO_TypeDef *) GPIOE_BASE)
#define GPIOF                ((GPIO_TypeDef *) GPIOF_BASE)
#define GPIOG                ((GPIO_TypeDef *) GPIOG_BASE)
```

这是库里面的选择项，对于本芯片来说，我们都拥有以上 5 组 IO 口。

形参 2 是 GPIO_InitTypeDef 类型的结构体变量，其定义如下：

```
typedef struct
{
  uint32_t Pin;       /* 引脚号 */
  uint32_t Mode;      /* 模式设置 */
  uint32_t Pull;      /* 上拉下拉设置 */
  uint32_t Speed;     /* 速度设置 */
} GPIO_InitTypeDef;
```

该结构体很重要，下面对每个成员介绍一下。

成员 Pin 表示引脚号，范围：GPIO_PIN_0 到 GPIO_PIN_15，另外还有 GPIO_PIN_All 和 GPIO_PIN_MASK 可选。

成员 Mode 是 GPIO 的模式选择，有以下选择项：

```
#define  GPIO_MODE_INPUT                 (0x00000000U)   /* 输入模式 */
#define  GPIO_MODE_OUTPUT_PP             (0x00000001U)   /* 推挽输出 */
#define  GPIO_MODE_OUTPUT_OD             (0x00000011U)   /* 开漏输出 */
#define  GPIO_MODE_AF_PP                 (0x00000002U)   /* 推挽式复用 */
#define  GPIO_MODE_AF_OD                 (0x00000012U)   /* 开漏式复用 */
#define  GPIO_MODE_AF_INPUT              GPIO_MODE_INPUT

#define  GPIO_MODE_ANALOG                (0x00000003U)   /* 模拟模式 */

#define  GPIO_MODE_IT_RISING             (0x10110000u)  /* 外部中断，上升沿触发检测 */
#define  GPIO_MODE_IT_FALLING            (0x10210000u)  /* 外部中断，下降沿触发检测 */
/* 外部中断，上升和下降双沿触发检测 */
#define  GPIO_MODE_IT_RISING_FALLING     (0x10310000u)

#define  GPIO_MODE_EVT_RISING            (0x10120000U)   /*外部事件，上升沿触发检测 */
#define  GPIO_MODE_EVT_FALLING           (0x10220000U)   /*外部事件，下降沿触发检测 */
/* 外部事件，上升和下降双沿触发检测 */
#define  GPIO_MODE_EVT_RISING_FALLING    (0x10320000U)
```

成员 Pull 用于配置上下拉电阻，有以下选择项：

```
#define  GPIO_NOPULL                (0x00000000U)   /* 无上下拉 */
#define  GPIO_PULLUP                (0x00000001U)   /* 上拉 */
#define  GPIO_PULLDOWN              (0x00000002U)   /* 下拉 */
```

成员 Speed 用于配置 GPIO 的速度，有以下选择项：

```
#define  GPIO_SPEED_FREQ_LOW         (0x00000002U)   /* 低速 */
#define  GPIO_SPEED_FREQ_MEDIUM      (0x00000001U)   /* 中速 */
#define  GPIO_SPEED_FREQ_HIGH        (0x00000003U)   /* 高速 */
```

- **函数返回值：**

  无

- **注意事项：**

  HAL 库的 EXTI 外部中断的设置功能整合到此函数里面，而不是单独独立一个文件。这个我们到外部中断实验再细讲。

2. HAL_GPIO_WritePin 函数

HAL_GPIO_WritePin 函数是 GPIO 口的写引脚函数。其声明如下：

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

- **函数描述：**

  用于设置引脚输出高电平或者低电平，通过 BSRR 寄存器复位或者置位操作。

- **函数形参：**

  形参 1 是端口号，可以选择范围：GPIOA~GPIOG。

  形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。

  形参 3 是要设置输出的状态，是枚举型有两个选择：GPIO_PIN_SET 表示高电平，GPIO_PIN_RESET 表示低电平。

- **函数返回值：**

  无

3. HAL_GPIO_TogglePin 函数

HAL_GPIO_TogglePin 函数是 GPIO 口的电平翻转函数。其声明如下：

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

- **函数描述：**

  用于设置引脚的电平翻转，也是通过 BSRR 寄存器复位或者置位操作。

- **函数形参：**

  形参 1 是端口号，可以选择范围：GPIOA~GPIOG。

  形参 2 是引脚号，可以选择范围：GPIO_PIN_0 到 GPIO_PIN_15。

- **函数返回值：**

  无

本实验我们用到上面三个函数，其他的 API 函数后面用到再进行讲解。

GPIO 输出配置步骤

1）使能对应 GPIO 时钟

STM32 在使用任何外设之前，我们都要先使能其时钟（下同）。本实验用到 PB5 和 PE5 两个 IO 口，因此需要先使能 GPIOB 和 GPIOE 的时钟，代码如下:

```
__HAL_RCC_GPIOB_CLK_ENABLE();
__HAL_RCC_GPIOE_CLK_ENABLE();
```

2）设置对应 GPIO 工作模式（推挽输出）

本实验 GPIO 使用推挽输出模式，控制 LED 亮灭，通过函数 HAL_GPIO_Init 设置实现。

3）控制 GPIO 引脚输出高低电平

在配置好 GPIO 工作模式后，我们就可以通过 HAL_GPIO_WritePin 函数控制 GPIO 引脚输出高低电平，从而控制 LED 的亮灭了。

13.3.2 程序流程图

程序流程图能帮助我们更好的理解一个工程的功能和实现的过程，对学习和设计工程有很好的主导作用。本实验的程序流程图如下：

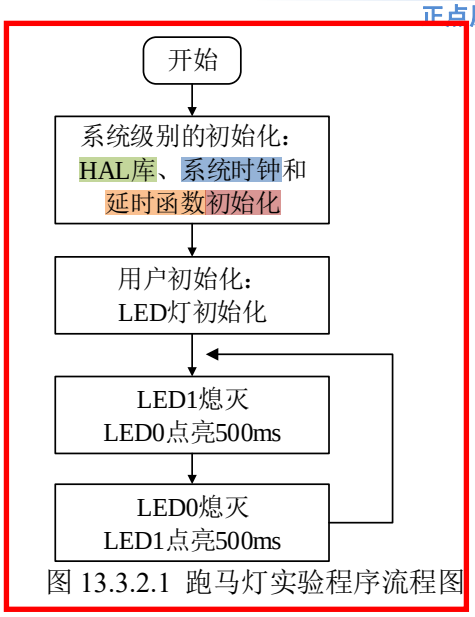

图 13.3.2.1 跑马灯实验程序流程图

13.3.3 程序解析

1. led 驱动代码

这里我们只讲解核心代码，详细的源码请大家参考光盘本实验对应源码。LED 驱动源码包括两个文件：led.c 和 led.h（正点原子编写的外设驱动基本都是包含一个.c 文件和一个.h 文件，下同）。

下面我们先解析 led.h 的程序，我们把它分两部分功能进行讲解。

- **LED 灯引脚宏定义**

由硬件设计小节，我们知道 LED 灯在硬件上分别连接到 PB5 和 PE5，再结合 HAL 库，我们做了下面的引脚定义。

```
/* 引脚 定义 */                                    do{...} while(0) :0为假 = 执行一次
#define LED0_GPIO_PORT              GPIOB
#define LED0_GPIO_PIN               GPIO_PIN_5
/* PB 口时钟使能 */
#define LED0_GPIO_CLK_ENABLE()      do{ __HAL_RCC_GPIOB_CLK_ENABLE(); }while(0)

#define LED1_GPIO_PORT              GPIOE
#define LED1_GPIO_PIN               GPIO_PIN_5
/* PE 口时钟使能 */
#define LED1_GPIO_CLK_ENABLE()      do{ __HAL_RCC_GPIOE_CLK_ENABLE(); }while(0)
```

这样的好处是进一步隔离底层函数操作，移植更加方便，函数命名更亲近实际的开发板。比如：当我们看到 LED0_GPIO_PORT 这个宏定义，我们就知道这是灯 LED0 的端口号；看到 LED0_GPIO_PIN 这个宏定义，就知道这是灯 LED0 的引脚号；看到 LED0_GPIO_CLK_ENABLE 这个宏定义，就知道这是灯 LED0 的时钟使能函数。大家后面学习时间长了就会慢慢熟悉这样的命名方式。

__HAL_RCC_GPIOx_CLK_ENABLE 函数是 HAL 库的 IO 口时钟使能函数，x=A 到 G。

- **LED 灯操作函数宏定义**

为了后续对 LED 灯进行便捷的操作，我们为 LED 灯操作函数做了下面的定义。

```
/* 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 翻转 */

#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 翻转 */

/* 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 */
```

LED0 和 LED1 这两个宏定义，分别是控制 LED0 和 LED1 的亮灭。例如：对于宏定义标识符 LED0(x)，它的值是通过条件运算符来确定：当 x=0 时，宏定义的值为 HAL_GPIO_WritePin(LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_RESET)，也就是设置 LED0_GPIO_PORT(PB5)输出低电平，当 n!=0 时，宏定义的值为 HAL_GPIO_WritePin (LED0_GPIO_PORT, LED0_GPIO_PIN, GPIO_PIN_SET)，也就是设置 LED0_GPIO_PORT(PB5)输出高电平。所以如果要设置 LED0 输出低电平，那么调用宏定义 LED0(0)即可，如果要设置 LED0 输出高电平，调用宏定义 LED0(1)即可。宏定义 LED1(x)同理。

LED0_TOGGLE 和 LED1_TOGGLE 这三个宏定义，分别是控制 LED0 和 LED1 的翻转。这里利用 HAL_GPIO_TogglePin 函数实现 IO 口输出电平取反操作。

下面我们再解析 led.c 的程序，这里只有一个函数 led init，这是 LED 灯的初始化函数，其定义如下：

```
/**
 * @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 */
}
```

对 LED 灯的两个引脚都设置为中速上拉的推挽输出。最后关闭 LED 灯的输出，防止没有操作就亮了。

2. main.c 代码

在 main.c 里面编写如下代码：

```
int main(void)
{
    HAL_Init();                                 /* 初始化 HAL 库 */
    sys_stm32_clock_init(RCC_PLL_MUL9);         /* 设置时钟, 72Mhz */
    delay_init(72);                             /* 延时初始化 */
    led_init();                                 /* 初始化 LED */

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

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

首先是调用系统级别的初始化：初始化 HAL 库、系统时钟和延时函数。接下来，调用 led_init 来初始化 LED 灯。最后在无限循环里面实现 LED0 和 LED1 间隔 500ms 交替闪烁一次。

13.4 下载验证

我们先来看看编译结果，如图 13.4.1 所示。

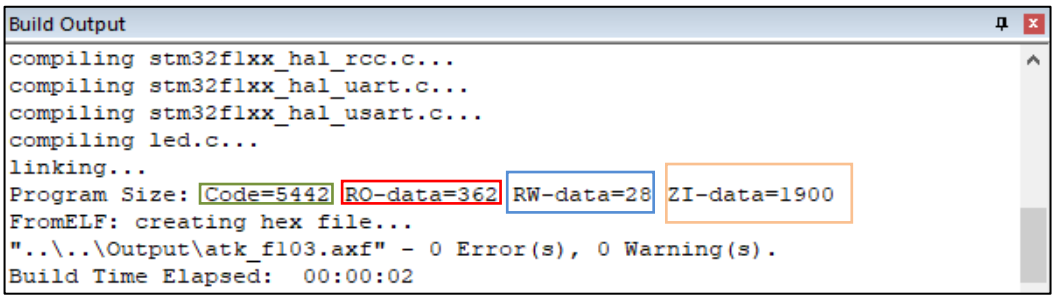

图 13.4.1 编译结果

可以看到没有 0 错误，0 警告。从编译信息可以看出，我们的代码占用 FLASH 大小为：5804 字节（5442+362+28），所用的 SRAM 大小为：1928 个字节（28+1900）。这里我们解释一下，编译结果里面的几个数据的意义：

Code：表示程序所占用 FLASH 的大小（FLASH）。

RO-data：即 Read Only-data，表示程序定义的常量（FLASH）。

RW-data：即 Read Write-data，表示已被初始化的变量（FLASH + RAM）

ZI-data：即 Zero Init-data，表示未被初始化的变量(RAM)

有了这个就可以知道你当前使用的 flash 和 ram 大小了，所以，一定要注意的是程序的大小不是.hex 文件的大小，而是编译后的 Code 和 RO-data 之和。

接下来，大家就可以下载验证了。这里我们使用 DAP 仿真器（也可以使用其他调试器）下载。

下载完之后，运行结果如图 13.4.2 所示，可以看到 LED 灯的 LED0 和 LED1 交替亮。

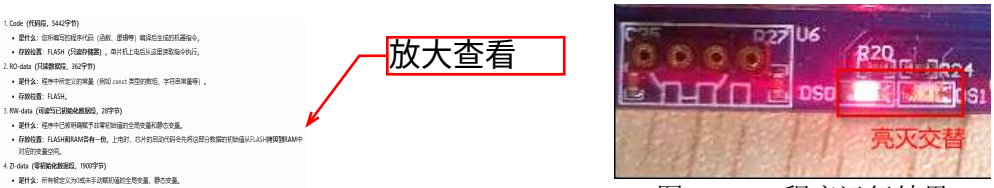

图 13.4.2 程序运行结果

至此，我们的跑马灯实验的学习就结束了，本章介绍了 STM32F103 的 IO 口的使用及注意事项，是后面学习的基础，希望大家好好理解。