第 5 章

GPIO 端口及其应用

学过 51 单片机的读者都应该知道，I/O 端口是 51 单片机的一个最基本同时也是应用最普遍的外设，STM32 单片机同样也是如此。本章就介绍 STM32 中最基本同时也是应用最普遍的外设——GPIO 端口，绝大多数基于 STM32 的应用程序开发都离不开 GPIO 端口，它是 STM32 与外部进行数据交换的通道，同时，它也是应用 STM32 内置外设的通道。

首先讲解 STM32 的 GPIO 端口的一些重要的概念和知识，包括它的 8 种工作模式及其各自的工作原理；然后介绍与 GPIO 端口相关的 7 个重要的寄存器；接着介绍 ST 官方固件库中包含的与 GPIO 端口相关的一些重要的库函数以及对它们的应用；最后讲解与 GPIO 端口相关的两个典型的应用实例，让大家在实际应用中理解 GPIO 端口的功能，并体会 ST 官方固件库在应用程序开发过程中的强大作用。

大家在对本章进行学习时，可以参考《STM32 中文参考手册》相关内容。

本章的学习目标如下：

- 理解并掌握 GPIO 端口的基础知识，重点掌握它的 8 种工作模式及其工作原理；
- 理解并掌握与 GPIO 端口相关的 7 个常用寄存器的各位的作用并掌握对它们进行配置的方法；
- 理解并掌握对 ST 官方固件库中与 GPIO 端口相关的一些常用库函数的应用方法；
- 理解并掌握 GPIO 端口的两个典型应用实例——流水灯和按键控制 LED。

5.1 GPIO 端口概述

STM32 最多可以有 7 个 GPIO(General Purpose Input/Output)端口，分别是 GPIOA、GPIOB、GPIOC、GPIOD、GPIOE、GPIOF 和 GPIOG，每个 GPIO 端口最多可以有 16 个端口位(引脚)，这样 STM32 最多可以有 112 个 GPIO 端口位。STM32 所有的 GPIO 端口位都可以被用作外部中断，学过 51 单片机的读者应该知道，51 单片机只有两个 I/O 端口引脚(P3.2 和 P3.3)可以被用作外部中断，显然，STM32 的 GPIO 端口具有更加强大的功能。

根据每个 GPIO 端口在芯片数据手册中列出的具体硬件特性，每个 GPIO 端口位都可以通过软件方式被独立地配置为以下 8 种工作模式：

- 浮空输入模式；
- 上拉输入模式；
- 下拉输入模式；

- 模拟输入模式；
- 开漏输出模式；
- 推挽输出模式；
- 复用功能的开漏输出模式；
- 复用功能的推挽输出模式。

当被配置为最后两种工作模式时，GPIO 端口位会被用作 STM32 内置外设的复用功能引脚，后面会对此进行详细介绍。

STM32 的标准 I/O 端口位的基本结构如图 5-1 所示。

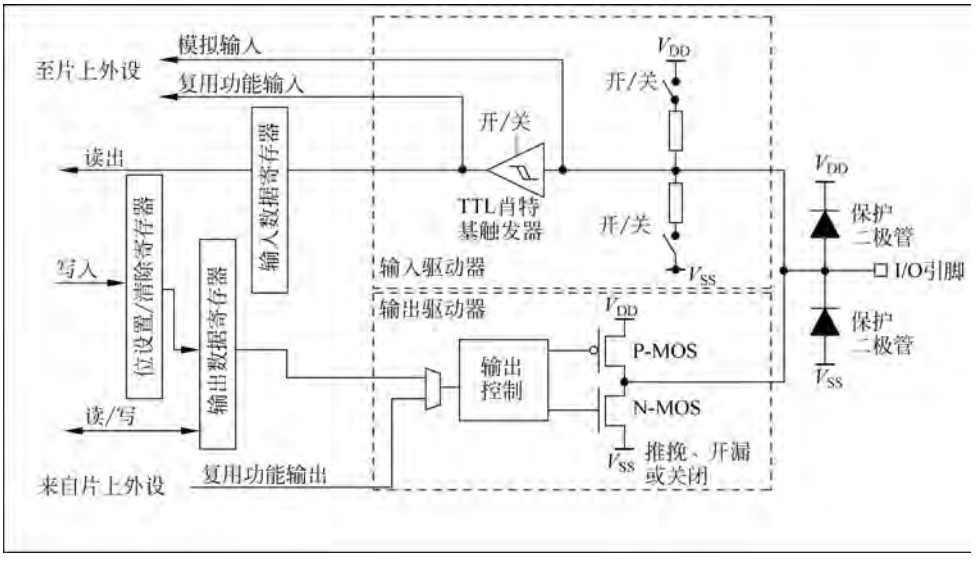

图 5-1　STM32 的标准 I/O 端口位的基本结构

STM32 单片机的每个标准的 I/O 端口位都具有如图 5-1 所示的基本结构，在此基础上，可以通过软件方式将其配置为以上 8 种工作模式之一。

大家可能对所谓的“标准”I/O 端口位不是很理解，对此进行一个简单的说明。标准 I/O 端口位，是相对于对 5V 兼容的 I/O 端口位来说的。STM32 的工作电压（V_{DD}）为 2.0～3.6V（一般选择为 3.3V），即 I/O 端口位在高电平状态下对应的电压为 3.3V，但是，有些 I/O 端口位对于 5V 电压也是能够兼容的，它们的基本结构其实与图 5-1 几乎完全相同，具体可参见《STM32 中文参考手册》第 106 页的图 14。至于具体哪些 I/O 端口位是 5V 兼容的，需要查阅芯片相关的数据手册。对于天信通 STM32F107 开发板，在其数据手册的第 3 章中有一个关于引脚定义的表格，如图 5-2 所示。

在如图 5-2 所示的引脚定义表中，每一行表示芯片的一个引脚的相关信息，其中带有“FT”标记的表示该引脚对于 5V 是兼容的。下面结合图 5-1，具体介绍 STM32I/O 端口位的 8 种工作模式的基本原理。

首先看一下输入工作模式。当 I/O 端口位被配置为浮空输入、上拉输入或下拉输入这 3 种工作模式时，其基本结构如图 5-3 所示。

在如图 5-3 所示的 I/O 端口位在浮空、上拉或下拉输入模式下的基本结构中，输入驱动

Table 5. Pin definitions

Pins			Pin name	Type[1]	I/O Level[2]	Main function[3] (after reset)	Alternate functions[4]	
BGA100	LQFP64	LQFP100					Default	Remap
A3	-	1	PE2	I/O	FT	PE2	TRACECK	-
B3	-	2	PE3	I/O	FT	PE3	TRACED0	-
C3	-	3	PE4	I/O	FT	PE4	TRACED1	-
D3	-	4	PE5	I/O	FT	PE5	TRACED2	-
E3	-	5	PE6	I/O	FT	PE6	TRACED3	-
B2	1	6	V_{BAT}	S	-	V_{BAT}	-	-
A2	2	7	PC13-TAMPER-RTC[5]	I/O	-	PC13[6]	TAMPER-RTC	-
A1	3	8	PC14-OSC32_IN[5]	I/O	-	PC14[6]	OSC32_IN	-
B1	4	9	PC15-OSC32_OUT[5]	I/O	-	PC15[6]	OSC32_OUT	-

图 5-2　STM32F107 引脚定义表

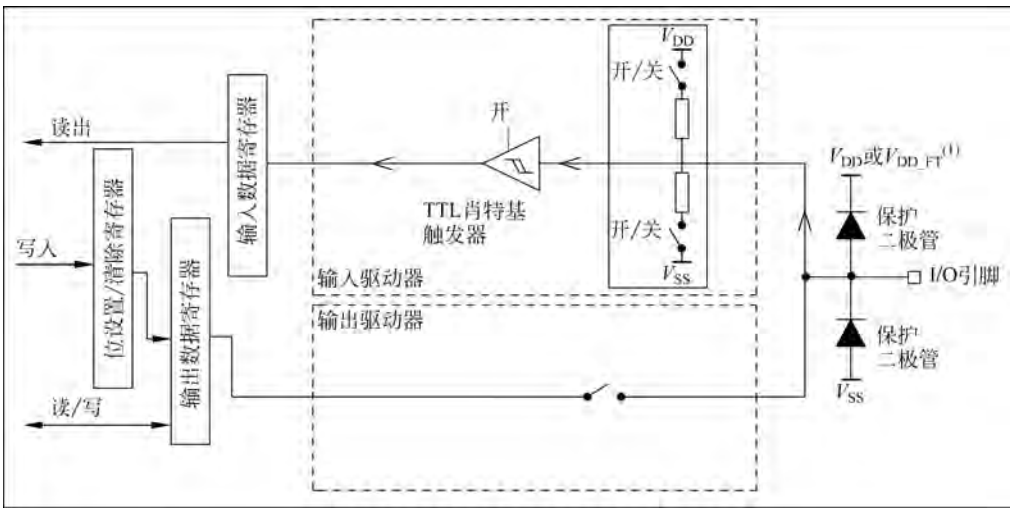

图 5-3　浮空、上拉或下拉输入模式

器工作，输出驱动器停止，且输入驱动器中的 TTL 肖特基触发器处于被激活状态，保证 I/O 端口位中的数据(1 或 0，分别对应高/低电平状态)能够被采样到输入数据寄存器中。在输入驱动器中，I/O 端口位可以通过闭合或断开相关电路的开关来选择是否通过上拉电阻连接到 V_{DD}，以及是否通过下拉电阻连接到 V_{SS}。当上拉和下拉电路的开关都断开时，即为浮空输入模式；当上拉电路开关闭合、下拉电路开关断开时，即为上拉输入模式；当下拉电路开关闭合、上拉电路开关断开时，即为下拉输入模式。通过读输入寄存器，就可以读取到 I/O 端口位的数据。

当 I/O 端口位被配置为模拟输入工作模式时，I/O 端口位的基本结构如图 5-4 所示。

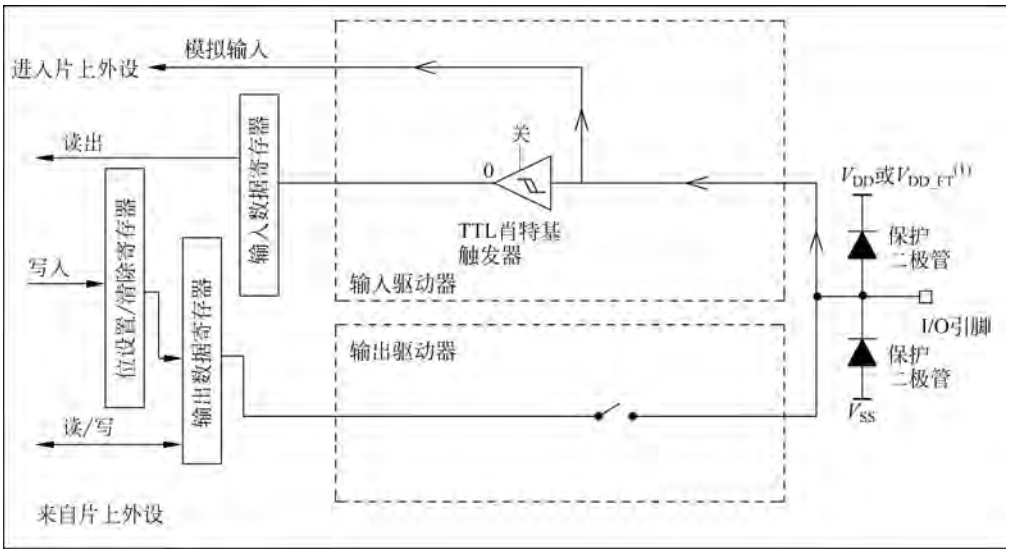

图 5-4　模拟输入模式

如图 5-4 所示的 I/O 端口位在模拟输入工作模式下的基本结构与图 5-3 最大的不同，就是它的输入驱动器中的 TTL 肖特基触发器处于被禁止的状态，它被强制输出一个 0 给输入数据寄存器，而且输入驱动器中的上拉和下拉电路都被断开（图 5-4 中未画），这样，I/O 端口位上的模拟信号会在几乎零消耗的情况下进入片上外设。

再来看输出配置。当 I/O 端口位被配置为开漏输出工作模式时，其基本结构如图 5-5 所示。

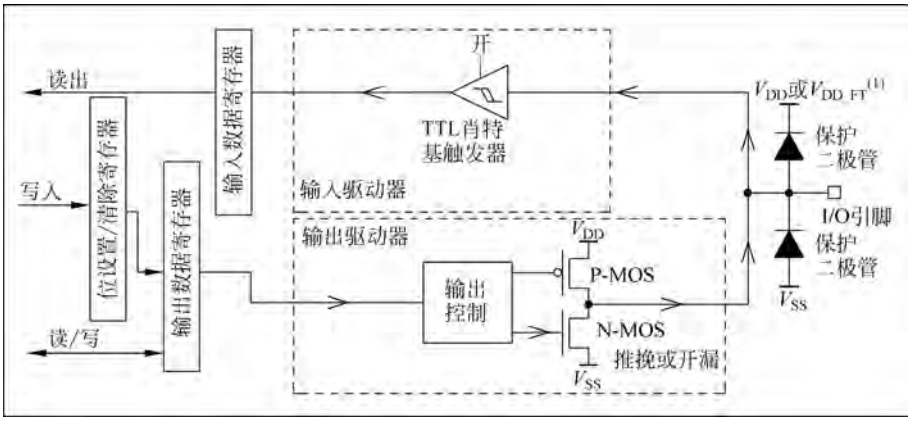

图 5-5　开漏输出模式

在如图 5-5 所示的 I/O 端口位在开漏输出模式下的基本结构中，输入、输出驱动器均启动，可以通过位设置/清除寄存器或直接对输出数据寄存器（稍后会介绍这些寄存器）进行写操作，来设置 I/O 端口位中的数据，数据经过输出控制后，传递给其后面的相关 MOS 管电路。在输出控制的后面，上面的 P-MOS 管不工作，只有下面的 N-MOS 管工作。如果对数据写 0，则 N-MOS 管导通，数据 0 会被传递到 I/O 端口位；如果对数据写 1，则 N-MOS 管

截止,I/O 端口位处于高阻状态,即它可能是 0,也可能是 1。在输入驱动器中,TTL 肖特基触发器处于被激活状态,可以通过输入数据寄存器读出 I/O 端口位的数据。也可以读输出数据寄存器,但读取的值不一定就是 I/O 端口位中的数据(读取的值为 0 的时候是,为 1 的时候则不一定)

当 I/O 端口位被配置为推挽输出模式时,其基本结构如图 5-6 所示。

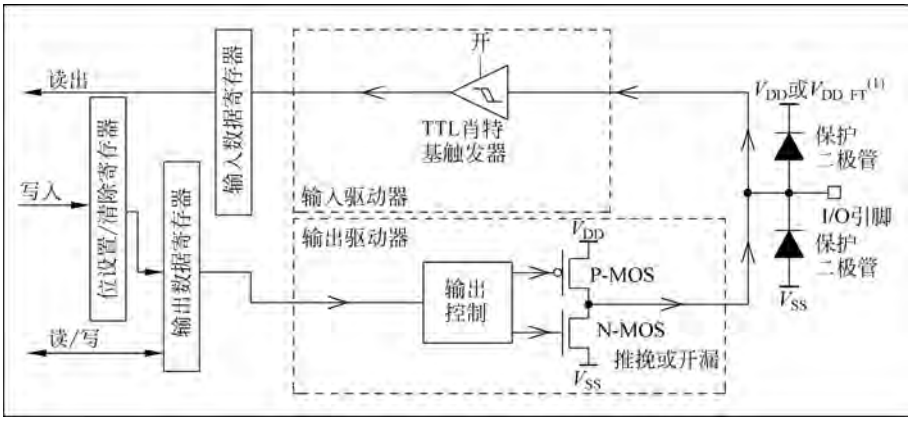

图 5-6　推挽输出模式

如图 5-6 所示的 I/O 端口位在推挽输出模式下的基本结构与图 5-5 的最大的不同,就是它输出驱动器中输出控制后面的 P-MOS 管和 N-MOS 管都会工作。当向 I/O 端口位的输出数据寄存器写 0 时,N-MOS 管导通,P-MOS 管截止,数据 0 会被传递到 I/O 端口位;当向 I/O 端口位的输出数据寄存器写 1 时,N-MOS 管截止,P-MOS 管导通,数据 1 会被传递到 I/O 端口位。这样,向输出数据寄存器写的数据总能够被传递到 I/O 端口位,而 P-MOS 管和 N-MOS 管相当于各占半个工作周期,“推挽输出”一词也由此而来。在这种工作模式下,既可以通过输入数据寄存器,也可以通过输出数据寄存器来读出 I/O 端口位中的数据。

当 I/O 端口位被配置为复用功能的开漏输出模式时,其基本结构如图 5-7 所示。

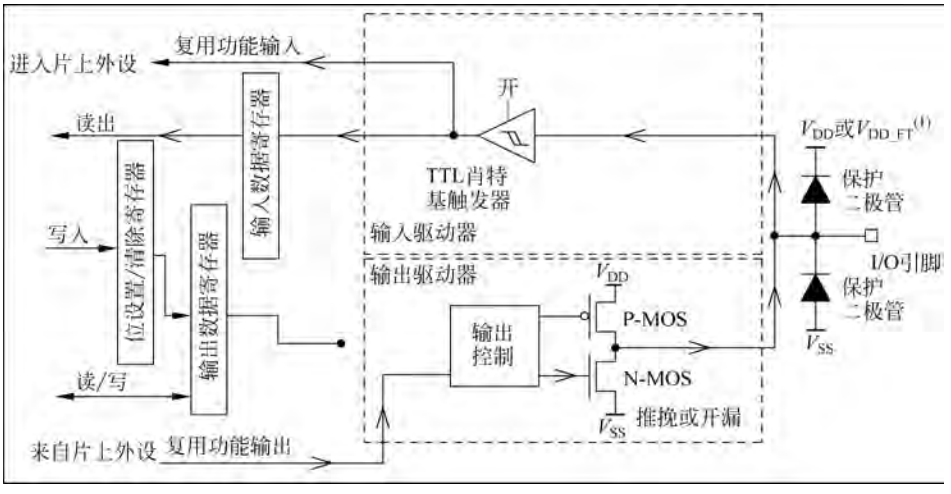

图 5-7　复用功能的开漏输出模式

如图 5-7 所示的 I/O 端口位在复用功能开漏输出模式下的基本结构与如图 5-5 所示的 I/O 端口位在开漏输出模式下的最大的不同，就是它的输出驱动器中输出控制的数据，来自片上外设的复用功能输出，并且它的输入驱动器中 TTL 肖特基触发器输出的数据还会作为复用功能的输入进入片上外设。

当 I/O 端口位被配置为复用功能的推挽输出模式时，其基本结构如图 5-8 所示。

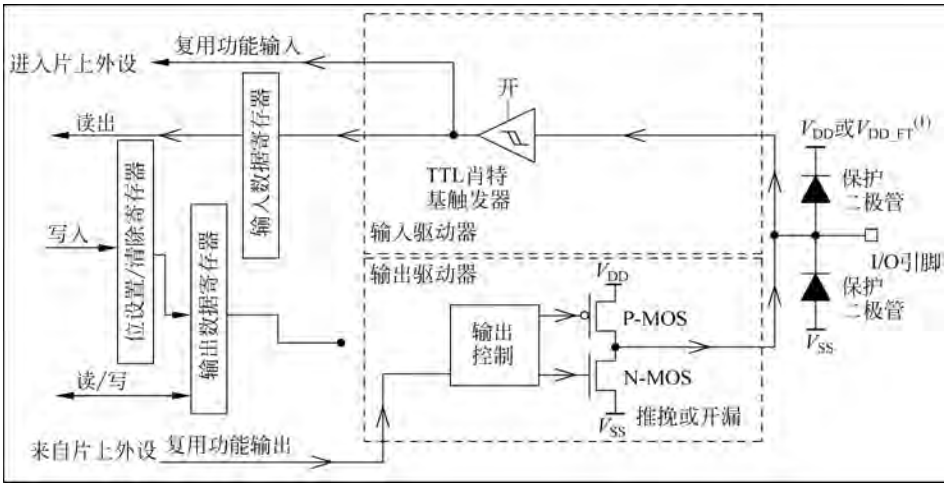

图 5-8　推挽复用输出模式

如图 5-8 所示的 I/O 端口位在复用功能推挽输出模式下的基本结构与如图 5-6 所示的 I/O 端口位在推挽输出模式下的最大的不同，就是它的输出驱动器中输出控制的数据来源，来自片上外设的复用功能输出，并且它的输入驱动器中 TTL 肖特基触发器输出的数据还会作为复用功能的输入进入片上外设。

以上就是 STM32 的 I/O 端口位的 8 种工作模式。当芯片上电复位后，除了与 JTAG 相关的 I/O 端口位之外，其他所有的 I/O 端口位都会被配置为浮空输入模式。而与 JTAG 相关的 I/O 端口位则会被配置为上拉或下拉输入模式，具体如下：

PA15——JTDI 置于上拉输入模式；

PA14——JTCK 置于下拉输入模式；

PA13——JTMS 置于上拉输入模式；

PB4——JNRTST 置于上拉输入模式。

5.2　GPIO 端口的相关寄存器

STM32 最多有 7 个 GPIO 端口(GPIOA～GPIOG)，每个 GPIO 端口有 7 个相关的寄存器，因此，STM32 最多有 7×7＝49 个 GPIO 端口相关的寄存器，分别是：GPIOx_CRL(x＝A,...,G)，GPIOx_CRH(x＝A,...,G)，GPIOx_IDR(x＝A,...,G)，GPIOx_ODR(x＝A,...,G)，GPIOx_BSRR(x＝A,...,G)，GPIOx_BRR(x＝A,...,G)，GPIOx_LCKR(x＝A,...,G)，它们都是 32 位的寄存器，并且都只能以 32 位(字)的形式被访问，不能以 16 位(半字)或 8 位(字节)的形式被访问。

5.2.1 端口配置低寄存器

端口配置低寄存器(GPIOx_CRL)(x=A,...,G)中各位的含义如图 5-9 所示。

偏移地址：0x00

复位值：0x4444 4444

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) 软件通过这些位配置相应的I/O端口 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) 软件通过这些位配置相应的I/O端口 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 5-9 端口配置低寄存器(GPIOx_CRL)(x=A,...,G)

如图 5-9 所示的端口配置低寄存器(GPIOx_CRL)(x=A,...,G)共有 32 位，可读/写。从第 0 位开始，到第 31 位，每 4 位为一组，一共可以分为 8 组，每组包含端口 x 的一组模式位 MODEy[1:0](y=0,...,7)和一组配置位 CNFy[1:0](y=0,...,7)，可以通过设置它们的值来配置端口 x 的第 y 位的工作模式。

当 MODEy[1:0]的值被设置为 00(二进制，下同)时，端口 x 的第 y 位被配置为输入模式；当 MODEy[1:0]的值分别被设置为 01、10 或 11 时，端口 x 的第 y 位被配置为输出模式，且对应的最大输出速度分别为 10MHz、2MHz 和 50MHz。

当 MODEy[1:0]的值被设置为 00 时(输入模式)，当 CNFy[1:0]的值分别被设置为 00、01 或 10 时，端口 x 的第 y 位分别被配置为模拟输入、浮空输入或上拉/下拉输入模式(至于究竟是上拉还是下拉输入模式，需要通过设置后面将要介绍的 GPIOx_ODR 寄存器来确定)，CNFy[1:0]的值为 11 的情况保留未被使用。

当 MODEy[1:0]的值被设置为 01、10 或 11 时(输出模式)，当 CNFy[1:0]的值分别被设置为 00、01、10 或 11 时，端口 x 的第 y 位分别被配置为推挽输出、开漏输出、复用功能推

挽输出或复用功能开漏输出的工作模式。

注意，GPIOx_CRL 在芯片上电复位后的初值为 0x44444444，即在芯片上电复位后，GPIOx_CRL 的每一组模式位 MODEy[1:0]和每一组配置位 CNFy[1:0]的值会被分别设置为 00 和 01，因此，STM32 的 I/O 端口位（除 JTAG 相关的之外）在初始状态下会被配置为浮空输入工作模式。

5.2.2　端口配置高寄存器

端口配置高寄存器（GPIOx_CRH）（x＝A，…，G）的内容如图 5-10 所示。

偏移地址：0x04

复位值：0x4444 4444

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) 软件通过这些位配置相应的I/O端口 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=8,...,15) 软件通过这些位配置相应的I/O端口 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 5-10　端口配置高寄存器（GPIOx_CRH）（x＝A，…，G）

如图 5-10 所示的端口配置高寄存器（GPIOx_CRH）（x＝A，…，G）和如图 5-9 所示的端口配置低寄存器（GPIOx_CRL）（x＝A，…，G）非常相似，所不同的只是 GPIOx_CRL 对应的是端口 x 的低 8 位（第 0 位～第 7 位）的工作模式的配置，而 GPIOx_CRH 对应的是端口 x 的高 8 位（第 8 位～第 15 位）的工作模式的配置。这样通过设置 GPIOx_CRL 和 GPIOx_CRH，就可以配置端口 x 的全部 16 位（0～15）的工作模式。

5.2.3　端口输入数据寄存器

端口输入数据寄存器（GPIOx_IDR）（x＝A，…，G）的内容如图 5-11 所示。

图 5-11　端口输入数据寄存器(GPIOx_IDR)(x=A,...,G)

在图 5-11 所示的端口输入数据寄存器(GPIOx_IDR)(x=A,...,G)中，第 0～15 位分别对应端口 x 的 16 位的输入电平状态(1 对应输入高电平，0 对应输入低电平)，这些位只能读并且只能以字的形式被读出。第 16～31 位保留，它们始终被读为 0。

在芯片上电复位后，该寄存器的初值被设置为 0x0000XXXX，因为端口 x 的 16 位默认处于浮空输入模式，所以它们的电平状态也处于浮空状态，不能确定它们输入的是高电平还是低电平。

5.2.4　端口输出数据寄存器

端口输出数据寄存器(GPIOx_ODR)(x=A,...,G)的内容如图 5-12 所示。

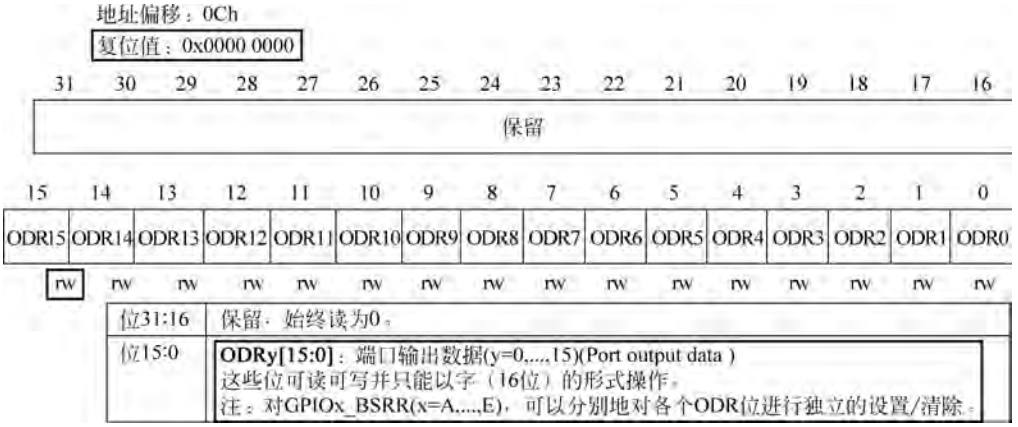

图 5-12　端口输出数据寄存器(GPIOx_ODR)(x=A,...,G)

在图 5-12 所示的端口输出数据寄存器(GPIOx_ODR)(x=A,...,G)中，第 0～15 位分别对应端口 x 的 16 位的输出电平状态(1 对应输出高电平，0 对应输出低电平)，这些位可读可写并且只能以字(16 位)的形式进行操作，第 16～31 位保留，它们始终被读为 0。

在芯片上电复位后，该寄存器的初值被设置为 0x00000000，即在输出模式下，端口 x 的 16 位默认都输出低电平。

前面在介绍 GPIOx_CRL(x=A,...,G)寄存器时,曾经提到过,当 I/O 端口位被配置为上拉/下拉输入模式时,究竟是选择上拉还是下拉输入模式,需要通过设置 GPIOx_ODR(x=A,...,G)寄存器来进一步确定,现在给出《STM32 中文参考手册》中的关于 I/O 端口位工作模式与 GPIOx_CRL 以及 GPIOx_ODR 寄存器关系的两个表,如图 5-13 所示。

端口位配置表

配置模式		CNF1	CNF0	MODE1	MODE0	PxODR寄存器
通用输出	推挽(Push-Pull)	0	0	01 10 11 见表18		0或1
	开漏(Open-Drain)		1			0或1
复用功能输出	推挽(Push-Pul)	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

输出模式位表

MODE[1:0]	意义
00	保留
01	最大输出速度为10MHz
10	最大输出速度为2MHz
11	最大输出速度为50MHz

图 5-13　端口位配置表和输出模式位表

5.2.5　端口位设置/清除数据寄存器

端口位设置/清除数据寄存器(GPIOx_BSRR)(x=A,...,G)的内容如图 5-14 所示。

地址偏移：0x10

复位值：0x0000 0000

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) 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) 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 5-14　端口位设置/清除数据寄存器(GPIOx_BSRR)(x=A,...,G)

在如图 5-14 所示的寄存器(GPIOx_BSRR)(x=A,...,G)中,第 0～15 位为对应端口 x 的第 0～15 位的设置位,第 16～31 位为对应端口 x 的第 0～15 位的清除位,这些位只能被写入,并且只能以字(16 位)的形式被操作。

当对设置位的第 y(y=0,...,15)位 BSy 进行操作时,如果对该位写 0,则不会对 GPIOx_ODR 寄存器的第 y 位产生影响;如果对该位写 1,则清除对应的 ODRy 位为 0。

当对清除位的第 y(y=0,...,15)位 BRy 进行操作时,如果对该位写 0,则不会对 GPIOx_ODR 寄存器的第 y 位产生影响;如果对该位写 1,设置对应的 ODRy 位为 1。

这个寄存器的主要作用在于:通过对它进行设置,可以单独地设置或清除 GPIOx_ODR 寄存器中的相关位,以达到更改某个端口位的输出电平状态而不影响其他端口位的效果,其实在如图 5-12 所示的 GPIOx_ODR 寄存器的表中的最后进行了相关的说明,大家可以回去看一下。最后,需要注意的是,如果同时设置了 GPIOx_BSRR 寄存器中相对应的 BSy 和 BRy 位,则 BSy 位起作用。

5.2.6 端口位清除数据寄存器

端口位清除数据寄存器(GPIOx_BRR)(x=A,...,G)的内容如图 5-15 所示。

地址偏移:0x14

复位值:0x0000 0000

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
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

位31:16	保留
位15:0	**BRy**:清除端口x的位y(y=0,...,15) 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

图 5-15 端口位清除数据寄存器(GPIOx_BRR)(x=A,...,G)

在如图 5-15 所示的寄存器(GPIOx_BSRR)(x=A,...,G)中,第 0～15 位对应端口 x 的第 0～15 位的清除位,这些位只能被写入,并且只能以字(16 位)的形式被操作。

当对其中的第 y 位 BRy 进行操作时,如果对该位写 0,则不会对 GPIOx_ODR 寄存器的第 y 位产生影响;如果对该位写 1,清除对应的 ODRy 位为 0。GPIOx_BSRR 寄存器的低 16 位与 GPIOx_BSRR 寄存器的高 16 位的功能基本相同,它的高 16 位则保留。

以上介绍了与 GPIOx(x=A,...,G)相关的 6 个寄存器,还有一个端口配置锁定寄存器(GPIOx_LCKR)(x=A,...,G),用得不是很多,这里就不介绍了,有兴趣的读者可以参考《STM32 中文参考手册》中 8.2.7 节的内容进行学习。这里顺便说明一下,关于 STM32 的 GPIO 端口的内容在《STM32 中文参考手册》的 8.1 节和 8.2 节中有详细介绍,整个 8.2 节全部是在介绍 GPIO 端口相关的 7 个寄存器,在该手册中,一般在每章的最后一节,都会将该章主讲的知识点所涉及的全部寄存器都详细地列出,因为本章还有另一个重要知识点 AFIO,因此 GPIO 相关的寄存器放在了 8.2 节中。

5.3　GPIO 端口的相关库函数

通过 5.2 节的学习，大家应该对 GPIO 端口相关的几个主要的寄存器有了一个总体的认识和基本的了解。本节将在 5.2 节的基础上介绍 ST 官方固件库中提供的与 GPIO 端口相关的一些重要的库函数，为 5.4 节 GPIO 端口的应用实例打下基础。

首先打开在 3.4 节中建立的工程模板 Template，并在它的 FWLIB 子文件夹下找到 stm32f10x_gpio.c 文件，如图 5-16 所示。

打开 stm32f10x_gpio.c 文件，在开始处的头文件包含区域，找到与该文件相对应的 stm32f10x_gpio.h 头文件被包含的预处理命令，然后对其右击，在弹出的快捷菜单中选择 Open document ‘stm32f10x_gpio.h’或 Go to Headerfile ‘stm32f10x_gpio.h’命令，如图 5-17 所示。

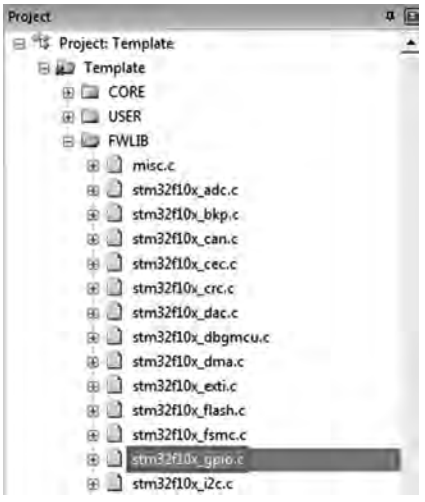

图 5-16　在 FWLIB 子文件夹下找到 stm32f10x_gpio.c 文件

图 5-17　打开 stm32f10x_gpio.h 头文件

另一个打开 stm32f10x_gpio.h 头文件的方法是在工程的 USER 或 FWLIB 子文件夹下的任何一个源文件所包含的头文件列表中找到并打开它。以 main.c 为例，如图 5-18 所示，当然，这需要首先对工程进行编译。

在 stm32f10x_gpio.h 头文件的最后，可以看到一系列以“GPIO”开头的函数声明，如图 5-19 所示。

它们都是与 GPIO 端口相关的函数，这些函数的定义都在 stm32f10x_gpio.c 源文件中。在 3.1.3 节介绍 ST 官方固件库包时，曾经讲到，函数库文件夹 Libraries 中包含一个名为 STM32F10x_StdPeriph_Driver 的文件夹，该文件夹中又包含了 inc 和 src 两个文件夹，它们又分别包含着 ST 官方提供的各种库函数相关的源文件(.c 文件)和头文件(.h 文件)，且一个源文件对应一个相关的头文件。现在可以告诉大家，关于库函数的定义基本上都在相关的源文件中，而其声明基本上都在相关的头文件中。

图 5-18　包含的头文件列表

```
349 void GPIO_DeInit(GPIO_TypeDef* GPIOx);
350 void GPIO_AFIODeInit(void);
351 void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct);
352 void GPIO_StructInit(GPIO_InitTypeDef* GPIO_InitStruct);
353 uint8_t GPIO_ReadInputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
354 uint16_t GPIO_ReadInputData(GPIO_TypeDef* GPIOx);
355 uint8_t GPIO_ReadOutputDataBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
356 uint16_t GPIO_ReadOutputData(GPIO_TypeDef* GPIOx);
357 void GPIO_SetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
358 void GPIO_ResetBits(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
359 void GPIO_WriteBit(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin, BitAction BitVal);
360 void GPIO_Write(GPIO_TypeDef* GPIOx, uint16_t PortVal);
361 void GPIO_PinLockConfig(GPIO_TypeDef* GPIOx, uint16_t GPIO_Pin);
362 void GPIO_EventOutputConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
363 void GPIO_EventOutputCmd(FunctionalState NewState);
364 void GPIO_PinRemapConfig(uint32_t GPIO_Remap, FunctionalState NewState);
365 void GPIO_EXTILineConfig(uint8_t GPIO_PortSource, uint8_t GPIO_PinSource);
366 void GPIO_ETH_MediaInterfaceConfig(uint32_t GPIO_ETH_MediaInterface);
```

图 5-19　GPIO 端口相关的函数声明

5.3.1　GPIO_Init()函数

首先来介绍 GPIO 端口的相关初始化函数 GPIO_Init()。在 ST 官方固件库中,一般名字以“_Init”结尾的函数都是初始化函数。该函数的声明如下所示:

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

在该函数声明处对 GPIO_Init 右击,在弹出的快捷菜单中选择 Go To Definition Of ‘GPIO_Init’命令,如图 5-20 所示。

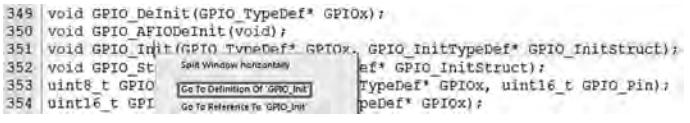

图 5-20　Go To Definition Of ‘GPIO_Init’命令

程序会跳转到 stm32f10x_gpio.c 源文件中的该函数定义处,如图 5-21 所示。

```
void GPIO_Init(GPIO_TypeDef* GPIOx, GPIO_InitTypeDef* GPIO_InitStruct)
{
  uint32_t currentmode = 0x00, currentpin = 0x00, pinpos = 0x00, pos = 0x00;
  uint32_t tmpreg = 0x00, pinmask = 0x00;
  /* Check the parameters */
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
  assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));

/*---------------------------- GPIO Mode Configuration -----------------------
  currentmode = ((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x0F);
  if ((((uint32_t)GPIO_InitStruct->GPIO_Mode) & ((uint32_t)0x10)) != 0x00)
  {
    /* Check the parameters */
    assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
    /* Output mode */
```

图 5-21　GPIO_Init()函数的定义

图 5-21 只是截取了该函数定义中的前几行代码,关于该函数的具体实现过程,此处不做详细介绍,有兴趣的读者可以自己尝试,其实该函数最终就是通过设置 GPIOx_CRL 或 GPIOx_CRH 这两个寄存器来实现配置 GPIO 端口引脚工作模式的功能,相关的代码如下:

```
GPIOx->CRL = tmpreg;
GPIOx->CRH = tmpreg;
```

下面重点介绍关于该函数的应用。

可以看到，该函数的两个参数分别是 GPIO_TypeDef 和 GPIO_InitTypeDef 的指针类型的。用刚才的方法单击进入 GPIO_TypeDef 的定义，程序会跳转到 stm32f10x.h 头文件中的如下代码处：

```
typedef struct
{
  __IO uint32_t CRL;
  __IO uint32_t CRH;
  __IO uint32_t IDR;
  __IO uint32_t ODR;
  __IO uint32_t BSRR;
  __IO uint32_t BRR;
  __IO uint32_t LCKR;
} GPIO_TypeDef;
```

这是一个结构体类型，它有 7 个成员变量，实际上它们分别对应着与 GPIO 端口相关的 7 个寄存器，大家从它们的名字中也能看出它们之间是互相对应的。该结构体类型实际上对应的是一个 GPIO 端口。

再单击进入 GPIO_InitTypeDef 的定义，程序会跳转到 stm32f10x_gpio.h 头文件中的如下代码处：

```
typedef struct
{
  uint16_t GPIO_Pin;
  GPIOSpeed_TypeDef GPIO_Speed;
  GPIOMode_TypeDef GPIO_Mode;
}GPIO_InitTypeDef;
```

这也是一个结构体类型，它有 3 个成员变量，其中，GPIO_Speed 和 GPIO_Mode 分别是 GPIOSpeed_TypeDef 和 GPIOMode_TypeDef 类型的。再分别进入它们的定义，可以看到，它们的定义都在 stm32f10x_gpio.h 头文件中。

GPIOSpeed_TypeDef 的定义如下所示：

```
typedef enum
{
  GPIO_Speed_10MHz = 1,
  GPIO_Speed_2MHz,
  GPIO_Speed_50MHz
}GPIOSpeed_TypeDef;
```

可以看出，GPIOSpeed_TypeDef 是一个枚举类型，其枚举值分别对应端口引脚在输出模式下的输出速度为 10MHz、2MHz 以及 50MHz。

GPIOMode_TypeDef 的定义如下所示：

```
typedef enum
{
GPIO_Mode_AIN = 0x0,
  GPIO_Mode_IN_FLOATING = 0x04,
  GPIO_Mode_IPD = 0x28,
  GPIO_Mode_IPU = 0x48,
  GPIO_Mode_Out_OD = 0x14,
  GPIO_Mode_Out_PP = 0x10,
  GPIO_Mode_AF_OD = 0x1C,
  GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;
```

可以看出，GPIOSpeed_TypeDef 也是一个枚举类型，其枚举值分别对应 GPIO 端口的 8 种工作模式，依次为模拟输入、浮空输入、下拉输入、上拉输入、开漏输出、推挽输出、复用功能开漏输出以及复用功能推挽输出。

结合它们的名字和注释(可以参考具体代码)，不难猜出它们的含义：

GPIO_Pin——要被初始化的 GPIO 端口的引脚；

GPIO_Speed——要被初始化的 GPIO 端口引脚的(输出)速度；

GPIO_Mode——要被初始化的 GPIO 端口引脚的工作模式。

现在，相信大家应该对该函数有一个大体的了解了。下面通过一个例子来具体说明对该函数的应用及其应用过程中的一些重要技巧。

例如，现在要将 GPIOA 的第 5 个引脚 PA5 配置为推挽输出工作模式，那么可以编写如下的程序代码：

```
GPIO_InitTypeDef GPIO_InitStructure;
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;                    //端口引脚号为 5
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;             //推挽输出模式
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;            //输出速度为 50MHz
GPIO_Init(GPIOA, &GPIO_InitStructure);                       //GPIO 端口 A
```

首先，需要说明的是，这只是一段孤立的代码，实际应用中当然还需要添加其他相关代码或命令，这里只是为了讲解怎样应用这个函数(下同)。

这段代码首先定义了一个 GPIO_InitTypeDef 结构体类型的变量 GPIO_InitStructure，然后对 GPIO_InitStructure 的 3 个成员变量分别进行赋值，最后调用 GPIO_Init()函数完成相关的初始化操作。注意，因为 GPIO_Init()函数的相应形参是 GPIO_InitTypeDef 指针类型的，而定义的变量 GPIO_InitStructure 是 GPIO_InitTypeDef 类型的，所以，在 GPIO_Init()函数的调用过程中，要用“&GPIO_InitStructure”来作相应的实参。

可能大家会有疑问，在上面的代码中，对 GPIO_Init()函数的实参 GPIOA 以及 GPIO_InitStructure 的成员变量 GPIO_Pin 的赋值要到哪里去找呢？下面就来回答这个问题同时以该函数为例介绍应用 ST 官方库函数的一些重要技巧。

首先，在 Template 工程的 USER 子文件夹下打开 main.c 文件，并将其中 main()函数

中的代码全部注释起来。注释的快速方法是选中要注释的代码，然后在 MDK 工具栏中单击 Edit→Advanced→Comment Selection 命令，取消注释的快速方法则是相应地单击 Edit→Advanced→Uncomment Selection 命令。

然后可以试着在 main()函数中逐条输入以上的 5 条代码。因为要调用的 GPIO_Init()函数的声明之前已经在 stm32f10x_gpio.h 头文件中找到，所以可以先输入这个函数的名称，然后再分别确定它的相关实参，如下所示。

```
GPIO_Init();
```

对 GPIO_Init()函数的实参的选取，实际上可以通过如图 5-20 所示该函数定义中的前几行对函数形参进行有效性验证的代码来实现。具体来说，在如图 5-20 所示的 GPIO_Init()函数的定义中，在定义了相关的局部变量之后，多次通过调用 assert_param()函数来对 GPIO_Init()函数的各个形参或其成员变量进行有效性验证，相关代码如下：

```
assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
assert_param(IS_GPIO_MODE(GPIO_InitStruct->GPIO_Mode));
assert_param(IS_GPIO_PIN(GPIO_InitStruct->GPIO_Pin));
assert_param(IS_GPIO_SPEED(GPIO_InitStruct->GPIO_Speed));
```

只需要对其中的宏 IS_GPIO_ALL_PERIPH、IS_GPIO_MODE、IS_GPIO_PIN 和 IS_GPIO_SPEED 右击，在弹出的快捷菜单中选择 Go To Definition Of 命令，进入它们的定义，就可以查看到相关取值的定义。

单击进入 IS_GPIO_ALL_PERIPH 的定义，可以看到，程序跳转到 stm32f10x_gpio.h 头文件中的如下代码处：

```
#define IS_GPIO_ALL_PERIPH(PERIPH) (((PERIPH) == GPIOA) || \
                                    ((PERIPH) == GPIOB) || \
                                    ((PERIPH) == GPIOC) || \
                                    ((PERIPH) == GPIOD) || \
                                    ((PERIPH) == GPIOE) || \
                                    ((PERIPH) == GPIOF) || \
                                    ((PERIPH) == GPIOG))
```

它的作用就是判断 PERIPH 是否是 GPIOA～GPIOG 中的一个，这里的“\”表示下一行代码与该行是连续的。从中可以很容易地找到需要的 GPIO 端口，并将它作为被调用的 GPIO_Init()函数的第一个实参写入。

对于 GPIO_Init()函数的第二个实参，因为它对应的形参是一个 GPIO_InitTypeDef 结构体类型的变量，因此，需要先定义一个该结构体类型的变量：

```
GPIO_InitTypeDef GPIO_InitStructure;
```

然后，需要对该结构体类型变量的各成员分别进行赋值，在 GPIO_InitStructure 的后面输入“.”，可以看到，GPIO_InitStructure 的 3 个成员变量会自动显示出来，如图 5-22 所示。

这样,就可以很容易地写出 GPIO_InitStructure 的 3 个成员变量,再分别对它们进行赋值。

图 5-22 GPIO_InitStructure 的 3 个成员变量

然后,用前面的方法,单击进入 IS_GPIO_ALL_PERIPH 的定义,程序会跳转到 stm32f10x_gpio.h 头文件中的如下代码处:

```
#define IS_GPIO_MODE(MODE) (((MODE) == GPIO_Mode_AIN) || \
((MODE) == GPIO_Mode_IN_FLOATING) || \
                    ((MODE) == GPIO_Mode_IPD) || \
((MODE) == GPIO_Mode_IPU) || \
                    ((MODE) == GPIO_Mode_Out_OD) ||\
                    ((MODE) == GPIO_Mode_Out_PP) || \
                    ((MODE) == GPIO_Mode_AF_OD) ||\
                    ((MODE) == GPIO_Mode_AF_PP))
```

与刚才 IS_GPIO_ALL_PERIPH(PERIPH)的情况相似,从中可以很容易地找到需要配置的 GPIO 端口的工作模式,并将其赋值给 GPIO_InitStructure 的成员变量 GPIO_Mode,如下所示:

```
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
```

在这段代码的前面,实际上就是 GPIOMode_TypeDef 的定义,如图 5-23 所示。

```
typedef enum
{ GPIO_Mode_AIN = 0x0,
  GPIO_Mode_IN_FLOATING = 0x04,
  GPIO_Mode_IPD = 0x28,
  GPIO_Mode_IPU = 0x48,
  GPIO_Mode_Out_OD = 0x14,
  GPIO_Mode_Out_PP = 0x10,
  GPIO_Mode_AF_OD = 0x1C,
  GPIO_Mode_AF_PP = 0x18
}GPIOMode_TypeDef;

#define IS_GPIO_MODE(MODE) (((MODE) == GPIO_Mode_AIN) || ((MODE) == GPIO_Mode_IN_FLOATING) ||
                            ((MODE) == GPIO_Mode_IPD) || ((MODE) == GPIO_Mode_IPU) || \
                            ((MODE) == GPIO_Mode_Out_OD) || ((MODE) == GPIO_Mode_Out_PP) || \
                            ((MODE) == GPIO_Mode_AF_OD) || ((MODE) == GPIO_Mode_AF_PP))
```

图 5-23 GPIOMode_TypeDef 及 IS_GPIO_MODE(MODE)的定义

再单击进入 IS_GPIO_ALL_PERIPH 的定义,程序会跳转到 stm32f10x_gpio.h 头文件中的相关代码处,如图 5-24 所示。

在图 5-24 中,在 IS_GPIO_ALL_PERIPH(PIN)定义的前面,就是对所有 GPIO 端口引脚的定义 GPIO_Pin_0~GPIO_Pin_15,以及一个对全部 16 个 GPIO 端口引脚都选中的定义,从中可以很容易地找到需要的 GPIO 端口引脚,并将其赋值给 GPIO_InitStructure 的成员变量 GPIO_Pin,如下所示:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5;
```

IS_GPIO_ALL_PERIPH(PIN)的定义如下:

```
#define IS_GPIO_PIN(PIN) ((((PIN) & (uint16_t)0x00) == 0x00) && ((PIN) != (uint16_t)0x00))
```

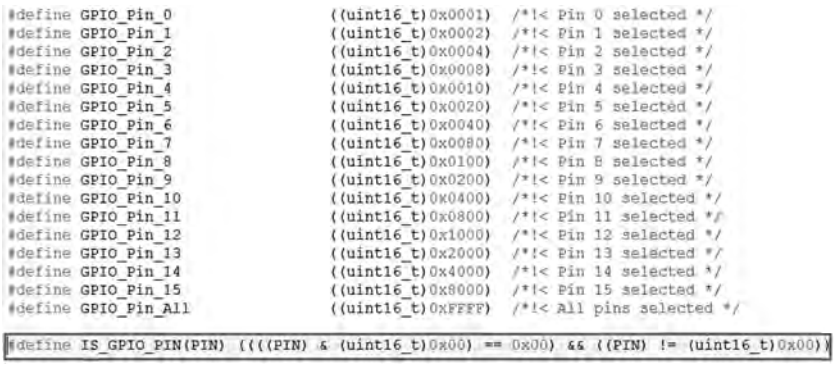

图 5-24　所有 GPIO 引脚及 IS_GPIO_ALL_PERIPH(PIN)的定义

它实际上就是验证 PIN 是否来自前面定义的 16 个 GPIO 端口引脚中的一个或多个。因为 16 个引脚中的每一个都对应 16 位二进制数的其中 1 位，16 个引脚刚好对应 16 位，所以，这 16 个引脚中的每一个是否被选中与其他引脚是否被选中彼此之间是相互独立的。如果想一次选中多个引脚，则可以用“|”操作符将它们“捆绑”起来。例如，这里如果还想将 GPIOA 的第 6 和第 7 个引脚 PA6 和 PA7 也配置为与 PA5 相同的工作模式，那么只需将相关的代码修改为：

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_5 | GPIO_Pin_6 | GPIO_Pin_7;
```

而不必对 PA6 和 PA7 都像上面 PA5 那样赋值一次。

最后，单击进入 IS_GPIO_SPEED 的定义，如图 5-25 所示。

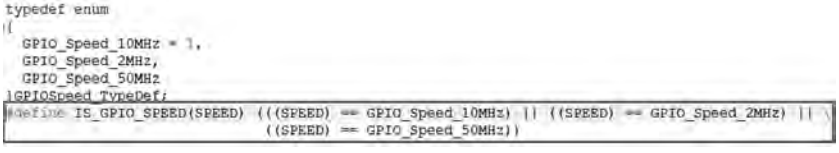

图 5-25　GPIOSpeed_TypeDef 及 IS_GPIO_SPEED(SPEED)的定义

在图 5-25 中，在 IS_GPIO_SPEED(SPEED)的前面，就是对 GPIOSpeed_TypeDef 的定义，从中也可以很容易地找到我们需要的端口引脚的输出速度，这里选择 50MHz，并将其对应的枚举成员赋值给 GPIO_InitStructure 的成员变量 GPIO_Speed，如下所示：

```
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
```

通过上面这 3 个例子，可以看出，ST 官方固件库中很多关于取值的相关定义就在对该值进行相关有效性验证的前面。而且，ST 官方固件库提供的代码有很好的可读性，基本上从被定义的某值所对应的宏的名字，就可以明白它所表示的含义，而当选取某值时，只需要选取它所对应的宏就可以了，其他工作 ST 官方固件库都已经为我们做好了，这样既减轻了大家编程的工作量，又提高了代码的可读性。

在确定了对 GPIO_InitStructure 的 3 个成员变量的赋值之后，再将“&GPIO_InitStructure”

作为被调用的 GPIO_Init()函数的第二个实参写入,这样,这段代码就全部写完了。很显然,通过这种方式来调用 GPIO_Init()函数并选取它的相关实参,可以使编程更加高效和便捷。

一般情况下,在 ST 官方提供的每个库函数定义中的起始部分,在相关局部变量的定义之后,都会通过 assert_param()函数来对该库函数的形参进行相关的有效性验证,而通过它就可以很容易地获得对该函数的相关实参的选取。这个对库函数的使用技巧非常重要,在后面会再次提到它,到时大家可以结合实际应用来体会它的具体作用。

5.3.2 GPIO_SetBits()函数和 GPIO_ResetBits()函数

下面介绍与 GPIO 端口相关的其他重要函数。首先来看 GPIO_SetBits()和 GPIO_ResetBits(),它们的声明分别如下所示:

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

对于这两个函数,相信大家在经过了刚才对 GPIO_Init()函数的学习之后,再结合这两个函数的名字及其形参,不难猜出它们要实现的功能。没错,它们的作用分别是将端口 GPIOx 的引脚 GPIO_Pin 上的输出电平设置为高电平和低电平,或者说,将该电平对应的二进制数据置 1 和清 0。它们的定义或具体实现过程分别如下所示:

```
void GPIO_SetBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
{
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  GPIOx->BSRR = GPIO_Pin;
}
void GPIO_ResetBits(GPIO_TypeDef * GPIOx, uint16_t GPIO_Pin)
{
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  assert_param(IS_GPIO_PIN(GPIO_Pin));
  GPIOx->BRR = GPIO_Pin;
}
```

可以看出,可以分别通过设置 GPIOx_BSR 和 GPIOx_BRR 寄存器来实现其相应的功能。

如果想要将 PA5 引脚上的输出电平置 1 或清 0,则可以在通过调用 5.3.2 节中介绍的 GPIO_Init()函数对其进行相关的初始化操作后,再通过调用本节介绍的这两个函数来实现上述的要求,如下所示:

```
GPIO_SetBits(GPIOA, GPIO_Pin_5);
GPIO_ResetBits(GPIOA, GPIO_Pin_5);
```

5.3.3 GPIO_Write()函数和 GPIO_WriteBit()函数

在 5.2 节中曾经讲到,对 GPIOx_BSR 和 GPIOx_BRR 寄存器的设置实际上最终的结

果是设置 GPIOx_ODR 寄存器。在 stm32f10x_gpio.c 源文件中，当然也定义了直接对 GPIOx_ODR 寄存器进行写操作的相关库函数——GPIO_Write()函数，它的声明如下：

```
void GPIO_Write(GPIO_TypeDef * GPIOx, uint16_t PortVal);
```

显然，只能一次性对端口 GPIOx 上的输出电平数据赋值 PortVal，其具体实现过程如下所示：

```
void GPIO_Write(GPIO_TypeDef * GPIOx, uint16_t PortVal)
{
  assert_param(IS_GPIO_ALL_PERIPH(GPIOx));
  GPIOx->ODR = PortVal;
}
```

还有一个名字与之类似的函数 GPIO_WriteBit()，声明如下：

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

该函数的功能实际上是 GPIO_SetBits()和 GPIO_ResetBits()这两个函数的结合，即对端口 GPIOx 的引脚 GPIO_Pin 上的输出电平数据赋值 BitVal。

5.3.4 GPIO_ReadInputDataBit()函数、GPIO_ReadInputData()函数、GPIO_ReadOutputDataBit()函数和GPIO_ReadOutputData()函数

还有 4 个分别对 GPIOx_IDR 和 GPIOx_ODR 寄存器进行读操作的库函数，它们的声明分别如下：

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

这些函数的作用分别是：返回读取的端口 GPIOx 的引脚 GPIO_Pin 上的输入电平数据、返回读取的整个 GPIOx 端口(即端口 GPIOx 上所有引脚的)的输入电平数据、返回读取的端口 GPIOx 的引脚 GPIO_Pin 上的输出电平数据以及返回读取的整个 GPIOx 端口(即端口 GPIOx 上所有引脚的)的输出电平数据。

5.3.5 GPIO_DeInit()函数

还有一个与 GPIO_Init()函数名字类似的函数 GPIO_DeInit()，它的声明如下：

```
void GPIO_DeInit(GPIO_TypeDef * GPIOx);
```

该函数的作用是将端口 GPIOx 的所有相关寄存器的值都初始化到芯片上电复位后的默认状态。这个函数用得不是很多，在这里给大家介绍它，是想告诉大家，在 ST 官方固件

库中,一般名字以"_DeInit"结尾的函数,都是初始化到默认状态的相关函数。

这样就将与 GPIO 端口相关的常用库函数全部都介绍完了。这些函数实际上都是通过对底层的相关寄存器进行操作来实现相应的功能的。

目前,对大家的要求是只需掌握怎样使用它们就可以了。如果以后想进行更深层次的开发,可以尝试着学习这些库函数的具体实现,这样可以加深对这些库函数的理解。此外,以后可能需要自己编写相关芯片的外设驱动函数等,学习 ST 官方库函数的具体实现过程,是很有帮助的。

最后需要说明的是,在 5.2 节介绍 GPIO 端口相关的寄存器的内容,是为了让大家对本节介绍的 GPIO 端口相关的库函数有更深刻的理解。

当用 STM32 进行实际项目开发时,通过 ST 官方提供的库函数就可以实现许多功能,但如果只会简单的调用,而不理解其底层操作的原理,那相当于只学会了皮毛,而没有掌握其内在的实质——单片机的应用开发实际上就是通过对它的底层寄存器进行相关的操作来实现其相应的功能,这样以后在较复杂的项目开发过程中很难做到举一反三,因为 STM32 有许多功能无法只靠 ST 官方提供的库函数来实现。因此,只有在理解了 STM32 底层寄存器工作原理的基础上,再去调用 ST 官方提供的库函数,才能在实际项目开发的过程中取得事半功倍的效果。

5.4 GPIO 端口的应用实例

通过前面几节的学习,相信大家应该对 STM32 的 GPIO 端口的工作原理、相关寄存器以及相关库函数都有了一定程度的理解。本节将在前面学习的基础上,并在本书相关的硬件开发平台的支持下,应用 STM32 的 GPIO 端口实现两个典型的案例。

首先,需要说明的是,硬件开发平台——天信通 STM32F107 开发板的主控芯片 STM32F107VCT6 只有 5 个 GPIO 端口,即 GPIOA～GPIOE,每个 GPIO 端口上都有 16 个端口位,一共有 80 个 GPIO 端口位。

5.4.1 流水灯

单片机 I/O 端口的一个典型应用就是通过输出高低电平来实现控制 LED(Light Emitting Diode,发光二极管)的亮灭,并可以进一步实现流水灯。如果大家学过 51 单片机,对此应该不会感到陌生。本节应用 STM32 的 GPIO 端口来实现流水灯,并以这个典型实例作为本书所有应用实例的开始。

1. 实例描述

本实例通过控制开发板的相关 GPIO 端口引脚不断地输出高低电平来使得开发板中的 4 个 LED 依次亮,在每一个 LED 亮时,其他 LED 灭,从而实现流水灯的效果。

2. 硬件电路

本实例相关的硬件电路如图 5-26 所示。

在图 5-26 中,发光二极管 D2、D3、D4 和 D5 的正极分别通过限流电阻 $R54$、$R53$、$R52$ 和 $R51$ 连接到 3.3V,它们的负极分别连接到开发板的 PE9、PE11、PE13 和 PE14 引脚。根据二极管的单向导电性,只有当正极与负极的电压差不小于开启电压时,二极管才能导通。

因此，在该电路中，当 PE9、PE11、PE13 或 PE14 引脚输出低电平时，与其相对应的发光二极管（即 D2、D3、D4 或 D5）会亮；而当 PE9、PE11、PE13 或 PE14 引脚输出高电平时，与其相对应的发光二极管不会亮。

3. 软件设计

下面进行本应用实例的软件设计。

首先，新建一个名为“流水灯”的文件夹，并在其中用 3.4 节中介绍的基于固件库新建工程模板的方法新建一个名为 LED 的工程。在 MDK 中，在向工程 LED 的 FWLIB 子目录下添加文件时，可以只添加 stm32f10x_gpio.c 和 stm32f10x_rcc.c 这两个文件，因为本实例只会用到这两个文件中的库函数。在 main.c 文件中，可以暂时只保留一个空的 main()函数。

然后，在“流水灯”文件夹中新建一个名为 HARDWARE 的文件夹，再在 HARDWARE 文件夹中新建一个名为 LED 的文件夹，如图 5-27 所示。

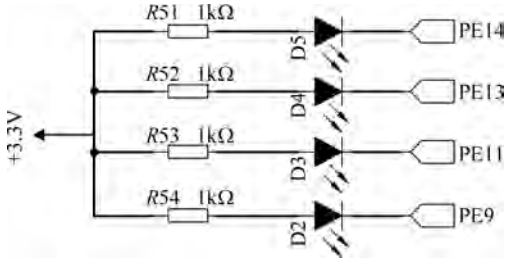

图 5-26　LED 相关的硬件电路

图 5-27　HARDWARE 文件夹中的 LED 文件夹

现在，在 MDK 的菜单栏上单击 File→New 命令，或者在 MDK 的工具栏上单击其相关的图标，新建两个文件，然后在菜单栏上单击 File→Save 命令，或者在工具栏上单击图标，将它们保存在刚才新建的 LED 文件夹中，并分别将它们命名为 led.c 和 led.h。注意，这一步也可以直接在如图 5-27 所示的 LED 文件夹中通过右击，在弹出的快捷菜单中选择“新建”→“文本文档”命令的方式来进行创建，只需要将原本文件名的扩展名.txt 相应地修改为.c 和.h 即可，这里是为了给大家介绍用 MDK 新建文件的基本应用。

然后，在 MDK 中将 led.c 源文件添加到 LED 工程的 HARDWARE 子目录中。并且在 LED 工程选项中，将 led.h 头文件的路径添加到 LED 工程要包含的头文件的路径列表中。如果对以上操作不是很清楚，可以参考 3.4 节中的相关内容。

现在，LED 工程在 MDK 中的目录结构如图 5-28 所示。

在图 5-28 中，除了给出了 LED 工程的文件夹结构，还对 LED 工程的各子文件夹的内容做了简单的介绍，其中的一部分内容曾经在 3.1.3 节介绍 STM32 官方固件库时提到过，但那时大家会感觉有些抽象，现在结合实际例程，再次对它们进行讲解，相信大家会对它们有更深刻的理解。

1）USER 子文件夹

USER 子文件夹中共包含 3 个文件：main.c、stm32f10x_it.c 和 system_stm32f10x.c。其中，main.c 文件主要用来编写 main 函数；stm32f10x_it.c 文件主要用来定义部分中断服务函数；system_stm32f10x.c 文件主要用来定义 SystemInit 时钟初始化函数。

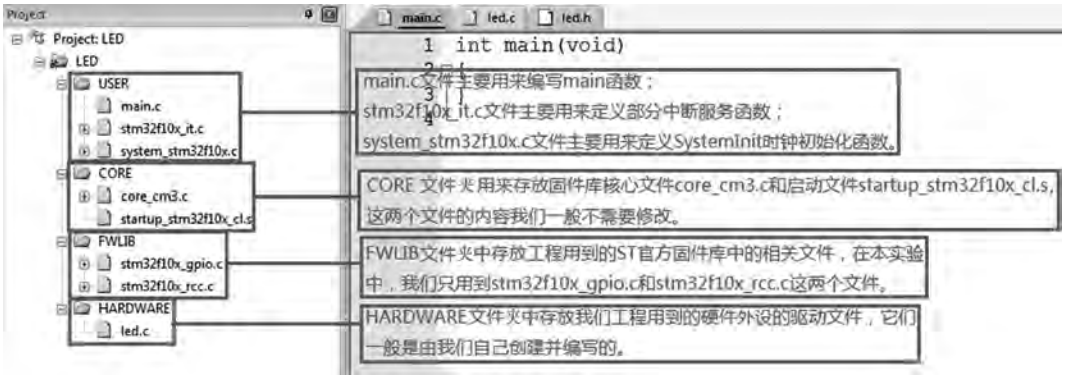

图 5-28　LED 工程的文件夹结构及其各子文件夹内容简介

2）CORE 子文件夹

CORE 子文件夹用来存放固件库核心文件 core_cm3. c 和启动文件 startup_stm32f10x_cl. s，这两个文件的内容一般不需要进行修改。

3）FWLIB 子文件夹

FLWIB 子文件夹用来存放工程用到的 ST 官方固件库中的相关文件，一般是用到哪个就添加哪个（避免工程过于庞大），本实例中，只用到与 STM32 的 GPIO 端口相关的 stm32f10x_gpio. c 文件以及与 STM32 的时钟相关的 stm32f10x_rcc. c 文件。一般情况下，所有的工程都需要添加 stm32f10x_rcc. c 文件，因为在系统初始化以及使用外设的过程中都会调用到时钟相关的库函数。

4）HARDWARE 子文件夹

HARDWARE 子文件夹用来存放工程相关的硬件外设的驱动文件，它们一般需要由自己进行创建并编写，如本工程中的 led. c 文件。

现在，开始编写程序代码。

第一步，在 led. h 头文件中添加如下程序代码：

```
#ifndef __LED_H
#define __LED_H
#include "stm32f10x.h"
void Init_LED(void);
#endif
```

如果大家是用前面介绍的第二种方法新建的 led. c 和 led. h 文件，则可以通过在 MDK 的菜单栏中单击 File→Open 命令的方法打开 led. h 文件。

下面简单解释一下这段代码的含义。

首先，文件通过条件编译预处理命令，避免了编译过程中出现 led. h 头文件被重复定义的错误，如下所示：

```
#ifndef __LED_H
#define __LED_H
...
#endif
```

上面的这段预处理命令被称为是条件编译预处理命令。所谓预处理命令，顾名思义，即在编译之前会预先处理的命令。引入这段条件编译预处理命令的目的是防止 led.h 头文件被重复定义。大家以后在实际工程中可以看到几乎所有的被包含的头文件的开头都会采用这种形式，实际上 ST 官方固件库提供的所有头文件的开头也都采用了这种形式。它的具体含义为：如果未定义宏__LED_H，则定义它，并执行它后面的一直到＃endif 之前的代码；反之，不会执行＃endif 之前的代码。宏__LED_H 实际上对应的就是该头文件，通过这种方式，可避免 led.h 头文件因被多次包含而重复定义所带来的编译错误。

然后，在条件编译预处理命令之间，文件通过文件包含预处理命令包含了 stm32f10x.h 头文件，如下所示：

```
#include "stm32f10x.h"
```

前面介绍过，这个头文件包含了许多重要的结构体以及宏的定义，此外，它还包含了系统寄存器定义声明以及包装内存操作，它几乎在整个 STM32 应用程序开发的过程中都会被使用到。可以在其右键快捷菜单中选择“Open document ‘stm32f10x.h’”命令打开该文件，查看其中的相关定义。

此外，文件还声明了 LED 相关的初始化函数 LED_Init()，如下所示，这个函数会在 led.c 文件中定义。

```
void LED_Init(void);
```

至此，led.h 文件中的所有程序代码就全部讲解完了。

第二步，在 led.c 文件中添加如下程序代码：

```
#include "led.h"

void LED_Init(void)
{
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
 GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
 GPIO_Init(GPIOE, &GPIO_InitStructure);
 GPIO_SetBits(GPIOE, GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14 );
}
```

在该文件中，首先包含了 led.h 头文件，这相当于同时包含了 stm32f10x.h 头文件。在该文件中，主要对 LED_Init()函数进行定义，该函数的主要功能是初始化 LED 相关的 GPIO 端口位，即配置 LED 相关的 GPIO 端口位的工作模式。

在 4.2 节中曾经讲到，对于 STM32，使用它的任何一个外设之前都必须首先使能与它相关的时钟。因此，在 LED_Init()函数中，首先需要做的就是初始化将要用到的 GPIO 端口的时钟。另外，所有的通用端口（即 GPIO 端口）的时钟都来自 APB2 时钟。

这里可以通过设置相关的寄存器来实现 GPIO 端口的时钟使能，当然，更简单的方法还是通过调用 ST 官方提供的库函数来实现。

ST 官方提供的时钟相关的库函数，都被定义在 stm32f10x_rcc.c 源文件中，打开相应的 stm32f10x_rcc.h 头文件，在该文件的最后，可以看到一系列时钟相关的函数声明，从中可以找到 RCC_APB2PeriphClockCmd()函数的声明，如下所示：

```
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState);
```

该函数的功能就是使能挂在 APB2 高级外设总线上的外设的时钟，单击进入它的定义，如图 5-29 所示。

```
void RCC_APB2PeriphClockCmd(uint32_t RCC_APB2Periph, FunctionalState NewState)
{
  /* Check the parameters */
  assert_param(IS_RCC_APB2_PERIPH(RCC_APB2Periph));
  assert_param(IS_FUNCTIONAL_STATE(NewState));
  if (NewState != DISABLE)
  {
    RCC->APB2ENR |= RCC_APB2Periph;
  }
  else
  {
    RCC->APB2ENR &= ~RCC_APB2Periph;
  }
}
```

图 5-29　RCC_APB2PeriphClockCmd()函数的定义

在对 RCC_APB2PeriphClockCmd()函数进行调用时，可以按照 5.3 节中介绍的方法来选取相应的实参。找到如图 5-29 所示的该函数定义中的通过 assert_param()对函数形参进行有效性验证的相关代码，然后分别右击其中的 IS_RCC_APB2_PERIPH 和 IS_FUNCTIONAL_STATE，在弹出的快捷菜单中选择 Go to Definition of 命令。对 IS_RCC_APB2_PERIPH 进行这样的操作，程序会跳转到 stm32f10x_rcc.h 文件中的相关代码处，如图 5-30 所示。

```
493 /** @defgroup APB2_peripheral
494   * @{
495   */
496
497 #define RCC_APB2Periph_AFIO              ((uint32_t)0x00000001)
498 #define RCC_APB2Periph_GPIOA             ((uint32_t)0x00000004)
499 #define RCC_APB2Periph_GPIOB             ((uint32_t)0x00000008)
500 #define RCC_APB2Periph_GPIOC             ((uint32_t)0x00000010)
501 #define RCC_APB2Periph_GPIOD             ((uint32_t)0x00000020)
502 #define RCC_APB2Periph_GPIOE             ((uint32_t)0x00000040)
503 #define RCC_APB2Periph_GPIOF             ((uint32_t)0x00000080)
504 #define RCC_APB2Periph_GPIOG             ((uint32_t)0x00000100)
505 #define RCC_APB2Periph_ADC1              ((uint32_t)0x00000200)
506 #define RCC_APB2Periph_ADC2              ((uint32_t)0x00000400)
507 #define RCC_APB2Periph_TIM1              ((uint32_t)0x00000800)
508 #define RCC_APB2Periph_SPI1              ((uint32_t)0x00001000)
509 #define RCC_APB2Periph_TIM8              ((uint32_t)0x00002000)
510 #define RCC_APB2Periph_USART1            ((uint32_t)0x00004000)
511 #define RCC_APB2Periph_ADC3              ((uint32_t)0x00008000)
512 #define RCC_APB2Periph_TIM15             ((uint32_t)0x00010000)
513 #define RCC_APB2Periph_TIM16             ((uint32_t)0x00020000)
514 #define RCC_APB2Periph_TIM17             ((uint32_t)0x00040000)
515 #define RCC_APB2Periph_TIM9              ((uint32_t)0x00080000)
516 #define RCC_APB2Periph_TIM10             ((uint32_t)0x00100000)
517 #define RCC_APB2Periph_TIM11             ((uint32_t)0x00200000)
518
519 #define IS_RCC_APB2_PERIPH(PERIPH) ((((PERIPH) & 0xFFC00002) == 0x00) && ((PERIPH) != 0x00))
```

图 5-30　APB2 总线上所有外设时钟及 IS_RCC_APB2_PERIPH(PERIPH)的定义

在图 5-30 中，在 IS_RCC_APB2_PERIPH(PERIPH)定义的前面，是对 APB2 总线上所有外设时钟的定义。对于这些宏定义，基本上通过它们的名字就可以知道它们所对应的外设，从中可以很容易地找到要使能的端口 GPIOE 的时钟所对应的宏定义 RCC_APB2Periph_GPIOE，并将它作为要被调用的 RCC_APB2PeriphClockCmd()函数的相应实参，剩下的工作 ST 官方固件库都已经为我们做好了。

IS_RCC_APB2_PERIPH(PERIPH)的定义如下所示：

```
#define IS_RCC_APB2_PERIPH(PERIPH) ((((PERIPH) & 0xFFC00002) == 0x00) && ((PERIPH) != 0x00))
```

它的作用就是检验 PERIPH 是否来自前面定义的挂在 APB2 总线上的外设时钟。可以看到，每一个挂在 APB2 总线上的外设时钟都被定义为一个 32 位二进制数的其中一位，与前面 GPIO 端口引脚的定义相似，它们当中每一个外设时钟是否被选中与其他外设时钟是否被选中彼此之间都是相互独立的。因此，当需要同时使能多个挂在 APB2 总线上的外设时钟时，也可以像之前对 GPIO 端口引脚的操作那样，用“|”将它们“捆绑”起来。

这里对 APB2 总线上所有外设时钟的定义，实际上相当于给出了一种查找某一个外设挂在哪个外设总线上的方法，当要使能一个外设的时钟而不知道它挂在哪个外设总线上的时候，就可以通过这种方法进行查找。大家可以在如图 5-30 所示的这段代码的前后分别看到挂在 AHB 总线和 APB1 总线上的外设定义。

然后，在 IS_FUNCTIONAL_STATE 的右键快捷菜单中选择 Go to Definition of 命令，程序会跳转到 stm32f10x.h 文件中的相关代码处，如图 5-31 所示。

```
typedef enum {DISABLE = 0, ENABLE = !DISABLE} FunctionalState;
#define IS_FUNCTIONAL_STATE(STATE) (((STATE) == DISABLE) || ((STATE) == ENABLE))
```

图 5-31　FunctionalState 及 IS_FUNCTIONAL_STATE(STATE)的定义

很显然，在这里应该选择 ENABLE 作为 RCC_APB2PeriphClockCmd()函数的相应实参。

这样，就通过调用 RCC_APB2PeriphClockCmd()函数完成了对 GPIOE 时钟的使能，如下所示：

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
```

下面继续来看 LED_Init()函数中的代码。在初始化了 GPIOE 的时钟后，下一步需要初始化本应用实例中用到的相关 I/O 端口位，这可以通过调用 GPIO_Init()函数来实现。关于该函数及其具体应用，在 5.3 节中有过详细的讲解，此处不再赘述。函数中的相关代码如下所示：

```
GPIO_InitTypeDef GPIO_InitStructure;
RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOE, ENABLE);
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14;
GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_Init(GPIOE, &GPIO_InitStructure);
```

关于这段代码,应注意以下几点:

第一,局部变量 GPIO_InitStructure 的定义一定要在函数的最开始,要在刚才讲解的 RCC_APB2PeriphClockCmd()函数调用语句的前面;

第二,因为端口引脚 PE9、PE11、PE13 和 PE14 都要被配置为推挽输出工作模式,而且它们来自同一个 GPIO 端口,所以这里可以通过如下代码来简化编码工作:

```
GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14;
```

第三,GPIO_Init()函数的第二个形参是 GPIO_InitTypeDef 的指针类型的,而定义的变量 GPIO_InitStructure 是 GPIO_InitTypeDef 类型的,因此,在调用该函数时,应先对 GPIO_InitStructure 取地址,再将其作为 GPIO_Init()函数的第二个实参。

最后,再来看看 LED_Init()函数中的最后一条语句:

```
GPIO_SetBits(GPIOE, GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14 );
```

GPIO_SetBits()函数在 5.3 节中也曾经介绍过,它的作用就是将相关 GPIO 端口引脚的输出数据置 1,因为本应用实例中的所有 LED,都是在其相关引脚输出低电平时亮,高电平时不亮,所以,在这里将它们的输出都设置为高电平,即让它们一开始都不亮。

至此,led.c 文件中的所有程序代码就全部讲解完了。

第三步,在 main.c 文件中添加如下程序代码:

```
#include "stm32f10x.h"
#include "led.h"

void Delay(u32 count)
{
    while(count--);
}

int main(void)
{
 LED_Init();
 while(1)
 {
      GPIO_ResetBits(GPIOE, GPIO_Pin_9);
      GPIO_SetBits(GPIOE, GPIO_Pin_11 | GPIO_Pin_13 | GPIO_Pin_14 );
      Delay(14400 * 200);

      GPIO_ResetBits(GPIOE, GPIO_Pin_11);
      GPIO_SetBits(GPIOE, GPIO_Pin_9 | GPIO_Pin_13 | GPIO_Pin_14 );
      Delay(14400 * 200);

      GPIO_ResetBits(GPIOE, GPIO_Pin_13);
      GPIO_SetBits(GPIOE, GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_14 );
      Delay(14400 * 200);
```

```
        GPIO_ResetBits(GPIOE, GPIO_Pin_14);
        GPIO_SetBits(GPIOE, GPIO_Pin_9 | GPIO_Pin_11 | GPIO_Pin_13 );
        Delay(14400 * 200);
    }
}
```

这段代码相对来说比较简单。

首先，在文件起始处包含了 stm32f10x.h 和 led.h 头文件；其次，定义了一个很简单的延时函数；最后，在 main()函数中，先调用 LED_Init()函数对 LED 相关的 GPIO 端口引脚进行相关的初始化，然后在 while(1)死循环中，让 4 个 LED——D2、D3、D4 和 D5 依次亮起，且在每个 LED 亮时其他 LED 不亮，并且通过延时函数让每个 LED 亮的状态持续一段固定的时间。要想让某个 LED 亮，就需要使其相关的引脚输出低电平，这里是通过 GPIO_ResetBits()函数来实现的，这个函数在 5.3 节也曾经介绍过，它的作用是将相关的 GPIO 端口位清 0。最后，需要注意的是，延时函数是必不可少的，而且延时的时间也不能过短，因为人眼能够分辨事物变化的最小时间间隔大约为 20ms，所以延时的时间不能小于这个数。经测验，延时函数 Delay()的实参大约为 14 400 时，对应的延时时间大约为 1ms，就以 14 400×200 作为 Delay()函数的实参，这样，延时时间近似为 200ms。

至此，本应用例程的编程工作就全部完成了。

4. 例程下载验证

将该例程下载到开发板，来验证它是否实现了流水灯的效果。通过 JLINK 下载方式将其下载到开发板，并按 RESET 按键对其复位，可以看到，D2、D3、D4 和 D5 以大约 200ms 的时间间隔循环往复地不停闪烁，实现了流水灯的效果。

5.4.2　按键控制 LED

5.4.1 节用 GPIO 端口的输出工作模式实现了流水灯。本节将用 GPIO 端口的输入工作模式来实现它的另一个典型应用——按键。

1. 实例描述

本应用实例将会通过按键来控制 LED 的亮灭，开发板共有 4 个开关按键和 4 个 LED，编程实现用这 4 个开关按键分别控制这 4 个 LED 的亮灭。即对某一个按键按下一次，它所控制的 LED 的亮灭状态就改变一次。

2. 硬件电路

本实例相关的硬件电路分别如图 5-32 和图 5-33 所示。

如图 5-32 所示的 LED 相关的硬件电路在 5.4.1 节已经介绍过，此处不再赘述。

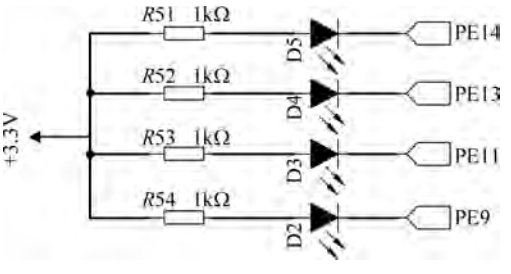

图 5-32　LED 相关的硬件电路

在如图 5-33 所示的按键相关的硬件电路中，开关按键 S1、S3、S4 和 S5 的一端都接 DGND，另一端分别接 PC6、PC7、PC8 和 PC9 引脚，且分别通过上拉电阻 $R34$、$R82$、$R87$

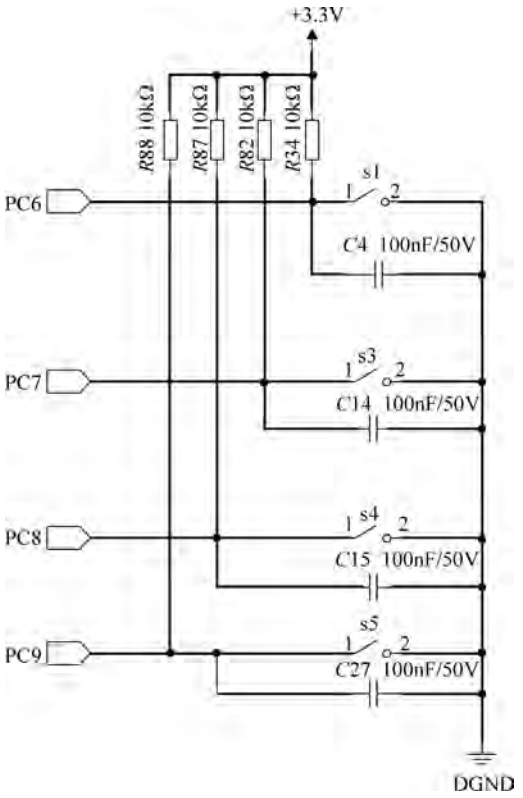

图 5-33　按键相关的硬件电路

和 $R88$ 接到 3.3V 高电平，其中电容 $C4$、$C14$、$C15$ 和 $C27$ 的作用是按键消抖滤波。当按键未被按下时，相关的引脚被拉高到 3.3V 高电平；当按键被按下时，相关的引脚被 DGND 拉低到低电平。因此，通过检测各开关按键相关引脚的输入电平，就可以判断它们是否被按下。

3. 软件设计

下面进行本应用实例的软件设计。

因为本应用实例是在流水灯应用实例的基础上完成的，所以，为了讲解方便，就直接在流水灯应用例程上进行修改。对复制的流水灯应用例程的工程文件夹及相关工程文件的名称进行修改，将 USER 文件夹中的 LED. uvprojx 和 LED. uvoptx 分别重命名为 KEY 加相关的后缀名，并将 OBJ 文件夹中的所有文件删除。然后，进入工程，将工程的目录名由原来的 LED 改为 KEY，最后在工程选项的 OUTPUT 标签下，将 Name of Executable 中的内容由原来的 LED 改为 KEY，这样就完成了对整个工程的重命名操作。

下面正式开始对“按键控制 LED”应用例程的实现。

首先，按照在流水灯应用例程实现过程中讲解的方法，在工程文件夹的 HARDWARE 文件夹中新建一个名为 KEY 的文件夹，并在其中新建 2 个文件——key. c 和 key. h，然后将 key. c 添加到工程 KEY 的 HARDWARE 子文件夹中，将 key. h 文件的路径添加到工程要包含头文件的路径列表中。

然后，开始编写程序代码。

第一步，在 key. h 文件中，添加如下程序代码：

```
#ifndef __KEY_H
#define __KEY_H
#include "stm32f10x.h"
#include "bitband.h"

#define KEY1 GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_6)
#define KEY2 GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_7)
#define KEY3 GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_8)
#define KEY4 GPIO_ReadInputDataBit(GPIOC,GPIO_Pin_9)

void Delay(u32 count);
void KEY_Init(void);
u8 KEY1_Scan(void);
u8 KEY2_Scan(void);
u8 KEY3_Scan(void);
u8 KEY4_Scan(void);
#endif
```

这段代码定义了 4 个宏 KEY1、KEY2、KEY3 和 KEY4，它们分别表示 4 个按键所对应 GPIO 端口引脚的输入电平，这是通过调用 GPIO_ReadInputDataBit()函数来实现的，这个函数在 5.3 节中曾经简单介绍过，它的功能就是读取某个 GPIO 端口引脚上的输入数据。

此外，还声明了延时函数 Delay()、按键初始化函数 KEY_Init()以及 4 个按键扫描函数 KEY1_Scan()、KEY2_Scan()、KEY3_Scan()和 KEY4_Scan()。

第二步，在 key.c 文件中，添加如下程序代码：

```
#include "key.h"

void Delay(u32 count)
{
 while(count--);
}

void KEY_Init()
{
 GPIO_InitTypeDef GPIO_InitStructure;
 RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC,ENABLE);
 GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6|GPIO_Pin_7|GPIO_Pin_8|GPIO_Pin_9;
 GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IPU;
 GPIO_Init(GPIOE, &GPIO_InitStructure);
}

u8 KEY1_Scan()
{
 static u8 key_up1 = 1;
 if(key_up1 && KEY1 == 0)
```

```
{
      Delay(14400 * 10);
      if(KEY1 == 0)
      {
           key_up1 = 0;
           return 1;
      }
 }
 else if(KEY1 == 1)
     key_up1 = 1;
 return 0;
}

u8 KEY2_Scan()
{
 static u8 key_up2 = 1;
 if(key_up2 && KEY2 == 0)
 {
      Delay(14400 * 10);
      if(KEY2 == 0)
      {
           key_up2 = 0;
           return 1;
      }
 }
 else if(KEY2 == 1)
      key_up2 = 1;
 return 0;
}

u8 KEY3_Scan()
{
 static u8 key_up3 = 1;
 if(key_up3 && KEY3 == 0)
 {
      Delay(14400 * 10);
      if(KEY3 == 0)
      {
           key_up3 = 0;
           return 1;
      }
 }
 else if(KEY3 == 1)
      key_up3 = 1;
 return 0;
}

u8 KEY4_Scan()
{
 static u8 key_up4 = 1;
```

```
if(key_up4 && KEY4 == 0)
 {
      Delay(14400 * 10);
      if(KEY4 == 0)
      {
           key_up4 = 0;
           return 1;
      }
 }
 else if(KEY4 == 1)
      key_up4 = 1;
 return 0;
}
```

这段代码首先定义了延时函数 Delay()，因为在后面的 4 个按键扫描函数中会用到它，然后定义了按键初始化函数 KEY_Init()，它与流水灯实验例程中的 KEY_Init()函数相似，此处不再赘述。需要注意的是，这里按键相关的 GPIO 端口引脚的工作模式应配置为输入上拉模式。

下面重点介绍这 4 个按键扫描函数，它们的实现过程完全相同，就以 KEY1_Scan()为例来对它们进行讲解。

众所周知，按键有支持连续按下操作的，比如计算机的键盘；也有不支持连续按下操作的，比如电视机遥控器的按键，它在被按下一次后必须经过一次弹起才能进行下一次被按下的操作，否则，即使保持一直被按下的状态，也只能实现按下一次的效果。很显然，这里要实现的按键属于第二种，否则，按下一次相关按键，它所控制的 LED 会不停地在亮灭状态之间快速转换，因为转换时间太短，甚至看不出它的变化，而只会看到它亮，一直到松开按键，而这时按键的亮灭状态几乎完全是随机的，根本无法控制。

因此，在 KEY1_Scan()函数中定义了一个 static 类型的变量 key1_up，它被用来标记当检测到按键 S1 当前处于被按下状态时，它在此之前是否已经处于被按下状态。当它的值为 1 时，表明它之前未处于被按下状态，即它是从弹起状态被按下的，则此次按下有效；当它的值为 0 时，表明它之前已处于被按下状态，则此次按下无效。因为 static 类型的变量被存储在静态存储区，所以在 KEY1_Scan()函数每次被调用完成之后，key1_up 对应的内存不会被释放，因此它具有“记忆”功能，用它可以标记在本次检测前按键 S1 是否处于被按下状态。需要说明的是，在 KEY1_Scan()函数中，对 key1_up 的初始化操作只会被执行一次，当再次调用该函数时，相关语句不会再次被执行：

```
static u8 key_up1 = 1;
```

在 KEY1_Scan()函数中，当检测到按键 S1 之前处于未被按下的状态且当前它被按下时，首先需要通过延时函数来消除抖动，如下所示：

```
if(key_up1 && KEY1 == 0)
{
 Delay(14400 * 10);
```

```
...
}
```

这是因为在按键被按下的实际过程中,其相关的引脚并不是从高电平直接变为低电平,而是会经历一个按键抖动的过程,如图 5-34 所示。

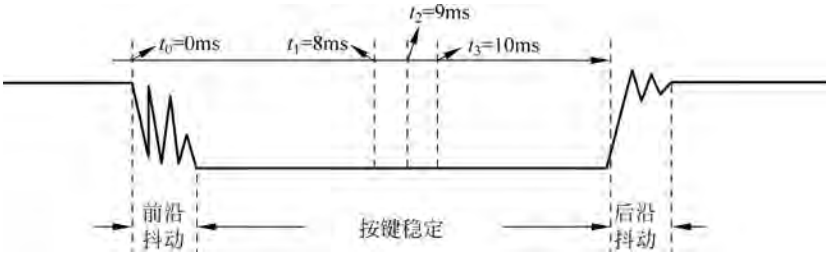

图 5-34 按键抖动过程

按键抖动有前沿抖动和后沿抖动,分别对应按键在被按下和松开的过程中按键的抖动过程。一般来说,按键抖动过程持续的时间为 5～10ms。这里用延时函数延时 10ms 来消除按键在按下过程中的前沿抖动。

在消除按键的抖动后,再判断 S1 是否被按下,如果是,则将 key_up 的值清 0 标记 S1 已处于被按下状态,然后函数返回 1,表明 S1 被按下,如下所示:

```
if(KEY4 == 0)
{
 key_up4 = 0;
 return 1;
}
```

当 S1 处于未被按下状态时,需将 key_up 的值置 1,以保证当它被按下时能够及时地检测到,如下所示:

```
else if(KEY4 == 1)
 key_up4 = 1;
```

最后,函数返回 0,则表示未检测到 S1 被按下,如下所示:

```
return 0;
```

其他几个按键扫描函数的实现过程与 KEY1_Scan()函数完全相同,此处不再赘述。

至此,key.c 文件中的所有程序代码就全部讲解完了。

第三步,在 main.c 文件中,将原来的代码删除,添加如下程序代码:

```
#include "led.h"
#include "key.h"

int main(void)
```

```
{
 vu8 key = 0;
 LED_Init();
 KEY_Init();
 while(1)
 {
      if(KEY1_Scan())
      {
           if(GPIO_ReadOutputDataBit(GPIOE,GPIO_Pin_9))
           {
               GPIO_ResetBits(GPIOE,GPIO_Pin_9);
           }
           else
           {
               GPIO_SetBits(GPIOE,GPIO_Pin_9);
           }
      }

      if(KEY2_Scan())
      {
           if(GPIO_ReadOutputDataBit(GPIOE,GPIO_Pin_11))
           {
               GPIO_ResetBits(GPIOE,GPIO_Pin_11);
           }
           else
           {
               GPIO_SetBits(GPIOE,GPIO_Pin_11);
           }
      }

      if(KEY3_Scan())
      {
           if(GPIO_ReadOutputDataBit(GPIOE,GPIO_Pin_13))
           {
               GPIO_ResetBits(GPIOE,GPIO_Pin_13);
           }
           else
           {
               GPIO_SetBits(GPIOE,GPIO_Pin_13);
           }
      }

      if(KEY4_Scan())
      {
           if(GPIO_ReadOutputDataBit(GPIOE,GPIO_Pin_14))
           {
               GPIO_ResetBits(GPIOE,GPIO_Pin_14);
           }
           else
           {
```

```
                GPIO_SetBits(GPIOE,GPIO_Pin_14);
            }
        }
        Delay(14400 * 10);
    }
}
```

这段代码相对来说比较简单。首先定义一个 vu8(volatile u8)类型的变量 key,表示它可能会被意想不到地改变。然后分别调用 LED_Init()和 KEY_Init()函数初始化 LED 和按键相关的 GPIO 端口引脚。最后在一个 while(1)的死循环中,每隔大约 10ms,依次调用 4 个按键扫描函数来检测相关按键是否被按下,如果有,则将相关按键所对应的 LED 的亮灭状态改变一次。

这里通过调用 GPIO_ReadOutputDataBit()函数来判断相关 LED 当前的亮灭状态,该函数在 5.3 节曾经介绍过,它的作用是返回相关 GPIO 端口引脚的输出数据,它与在 key.h 文件中调用的 GPIO_ReadInputDataBit()函数相似,后者是返回相关 GPIO 端口引脚的输入数据。然后,根据 LED 当前的亮灭状态,即相关 GPIO 端口引脚的输出数据,相应地将它当前的亮灭状态改变一次,这当然是通过 GPIO_ResetBits()和 GPIO_SetBits()函数来实现的。

至此,本应用例程的编程工作就全部完成了。

4. 例程下载验证

将该例程下载到开发板,来验证它是否实现了按键控制 LED 的效果。通过 JLINK 下载方式将其下载到开发板,并按 RESET 按键对其复位,可以看到,分别通过按下一次按键 S1、S3、S4 和 S5,可以分别将它们所对应的 LED,即 D2、D3、D4 和 D5 的亮灭状态改变一次,我们的例程实现了按键控制 LED 的效果。

本章小结

本章介绍 STM32 中最基本同时也是应用最普遍的外设——GPIO 端口,它是 STM32 与外部进行数据交换的通道,同时,它也是应用 STM32 内置外设的通道。本章介绍了 STM32 的 GPIO 端口的知识,8 种工作模式以及它们各自的工作原理;介绍了与 GPIO 端口相关的 7 个寄存器及库函数以及应用;通过与 GPIO 端口相关的两个典型的应用实例,在应用中理解 GPIO 端口的功能,了解固件库在应用程序开发过程中的强大作用。