第 3 章

通用输入/输出端口 GPIO

GPIO，即通用输入/输出端口，是 ARM 单片机可控制的引脚。GD32 芯片的 GPIO 引脚与外部设备连接起来，可实现与外部的通信、控制外部硬件或者采集外部设备数据等功能。借助 GPIO，微控制器可以实现对外围设备（如 LED、按键等）最简单、最直观的监控。除此之外，GPIO 还可以用于串行并行通信、存储器扩展等。GPIO 往往是了解、学习、开发嵌入式系统的第 1 步。

本章主要介绍 GD32F10x 系列微控制器的 I/O 端口模块的 GPIO（通用 IO）作为输入、输出口的使用方法及相关的 GPIO 库函数。


12min

3.1 芯片的常用封装

在使用 ARM 芯片时，用户肉眼所见的仅是它的外在，能和芯片内部相连的就是它的引脚了。对芯片内部程序的载入、通过程序让芯片对外围电路的控制和访问就是通过这些引脚实现的，而 ARM 芯片的生产厂家会根据市场需求生产不同引脚数量的芯片，以 GD32F10x 系列为例，它的引脚数目就有 48、64、100、144 几种规格，而具体芯片引脚的封装方式又包括 QFN、LQFP、BGA、CSP 这几种。

（1）QFN：方形扁平无引脚封装。这是表面贴装型封装之一，QFN 是日本电子机械工业会规定的名称，现在多称为 LCC。在封装的四侧配置有电极触点，由于无引脚，贴装占有面积比 QFP 小，高度比 QFP 低，但是，当印刷基板与封装之间产生应力时，在电极接触处就不可以得到缓解，因此电极触点难以做到 QFP 的引脚那样多，一般只有 14～100 个引脚。材料有陶瓷和塑料两种。当有 LCC 标记时基本上是陶瓷 QFN，如图 3-1 所示。

（2）LQFP：薄型 QFP（Low-profile Quad Flat Package）指封装本体厚度为 1.4mm 的 QFP，是日本电子机械工业会根据制定的新 QFP 外形规格所用的名称，而 QFP 这种技术的中文含义叫方形扁平式封装技术（Plastic Quad Flat Package），该技术实现的 CPU 芯片引脚之间的距离很小，引脚很细，一般大规模或超大规模集成电路采用这种封装形式，如图 3-2 所示。该技术封装的 CPU 操作方便，可靠性高，而且其封装外形尺寸较小，寄生参数减小，适合高频应用；该技术主要适合用 SMT 表面安装技术在 PCB 上安装布线。

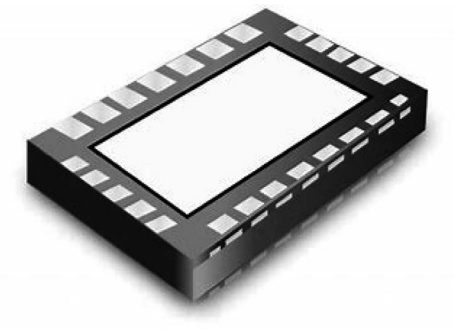

图 3-1　QFN 封装

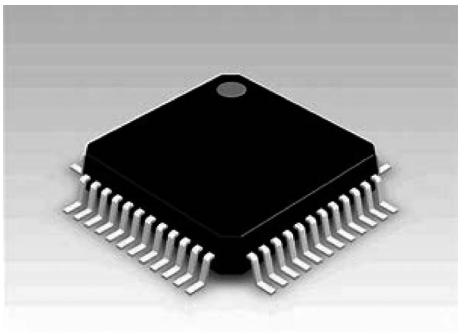

图 3-2　LQFP 封装

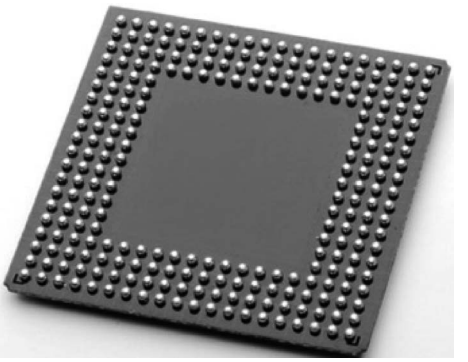

图 3-3　BGA 封装

（3）BGA：球栅阵列封装技术。该技术一出现便成为 CPU、主板南、北桥芯片等高密度、高性能、多引脚封装的最佳选择，但 BGA 封装占用基板的面积比较大，如图 3-3 所示。虽然该技术的 I/O 引脚数增多，但引脚之间的距离远大于 QFP，从而提高了组装成品率，而且该技术采用了可控塌陷芯片法焊接，从而可以改善它的电热性能。另外该技术的组装可用共面焊接，从而能大幅提高封装的可靠性，并且由该技术实现的封装 CPU 信号传输延迟小，适应频率可以提高很大。BGA 封装具有 4 个显著优点：①I/O 引脚数虽然增多，但引脚之间的距离远大于 QFP 封装方式，提高了成品率；②虽然 BGA 的功耗增加，但由于采用的是可控塌陷芯片法焊接，从而可以改善电热性能；③信号传输延迟小，适应频率大幅提高；④组装可用共面焊接，可靠性大幅提高。

（4）CSP：芯片级封装，CSP 封装是最新一代的内存芯片封装技术，其技术性能又有了新的提升。CSP 封装可以让芯片面积与封装面积之比超过 1∶1.14，已经相当接近 1∶1 的理想情况，绝对尺寸也仅有 32 平方毫米，约为普通的 BGA 的 1/3，仅仅相当于 TSOP 内存芯片面积的 1/6。与 BGA 封装相比，同等空间下 CSP 封装可以将存储容量提高三倍。

GD32F10x 系列有丰富的端口可供使用，包括多个多功能的双向 5V 兼容的快速 I/O 端口，所有 I/O 端口都可以映射到 16 个外部中断。一个 LQFP 封装的 64 脚的芯片的引脚图如图 3-4 所示。

如图 3-4 所示，LQFP64 封装的芯片 I/O 端口有 PA、PB、PC、PD 共 4 组，PA～PC 每组 16 个(0～15)，PD 只有 3 个引脚 PD0～PD2，所以 LQFP64 封装的 STM32F10x 一共有 51 个 I/O 端口。

具体到某一款具体的 ARM 芯片，其封装类型、引脚数量一般可以从其芯片命名看出。如图 3-5 所示为本书配套开发板所用芯片 GD32F103C8T6 的命名规则，从图中可以看出该芯片的引脚数量是 48 个，封装类型是 LQFP。

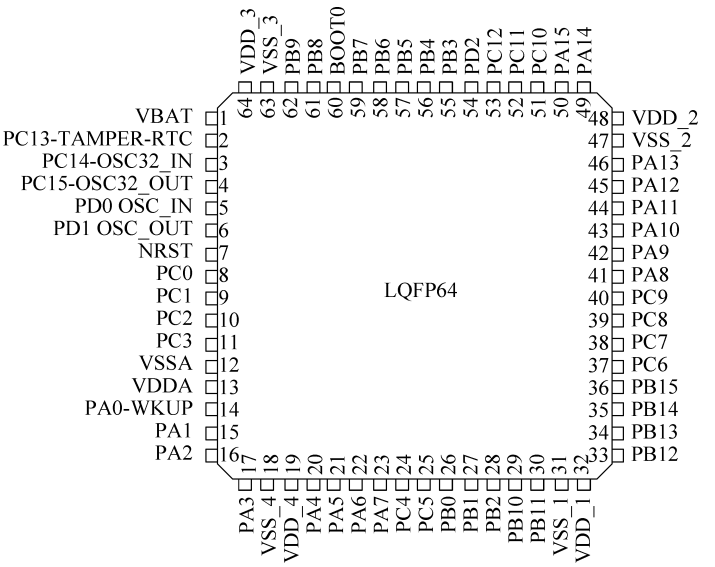

图 3-4　LQFP64 引脚图

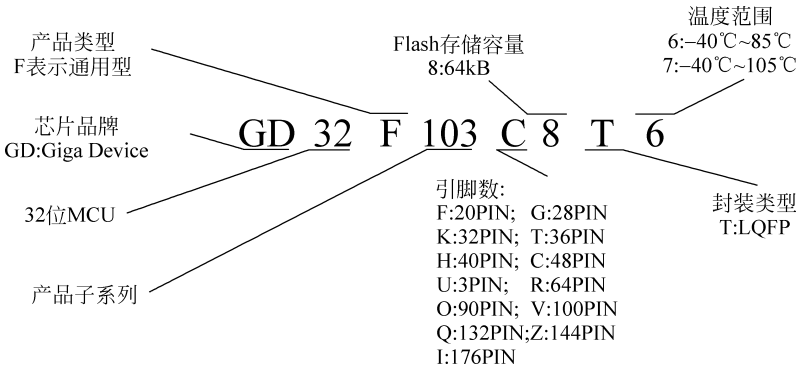

图 3-5　GD32F103C8T6 的命名规则

3.2　GPIO 工作原理

在 ARM 芯片中,GPIO 引脚既可以配置为输入也可以配置为输出,并且可以通过软件进行控制和配置。

当 GPIO 引脚被配置为输入模式时可用于接收外部设备或传感器发送的数字信号。在输入模式下,可以读取引脚的电平状态,以便进行相应处理。常见应用包括按键输入、传感器数据采集等。

当 GPIO 引脚被配置为输出模式时用于向外部设备发送数字信号,在输出模式下,可以通过设置引脚的电平状态,控制连接的外部设备的行为。常见的应用包括 LED 控制、驱动

电机、控制继电器等。

每个 I/O 端口都会有对应的配置寄存器，用户通过配置寄存器就可以设置 GPIO 引脚的功能和属性。

GPIO 引脚通常还支持中断功能，可以在引脚状态发生变化时触发中断请求。通过配置中断控制器和相关寄存器，可以实现对引脚状态变化的实时响应。这在需要高实时性和低功耗的应用中特别有用。

多数 ARM 芯片会提供多功能引脚（例如，复用引脚），可以通过配置将 GPIO 引脚用于其他功能，如串口通信、SPI 接口、I2C 接口等。这样可以根据应用需求，灵活地选择引脚的功能。

本节介绍 GPIO 的内部结构框图、输入/输出工作模式的实现原理。

3.2.1　内部结构框图

每个 GPIO 端口有两个 32 位配置寄存器（GPIOx_CRL、GPIOx_CRH）、两个 32 位数据寄存器（GPIOx_IDR、GPIOx_ODR）、一个 32 位置位/复位寄存器（GPIOx_BSRR）、一个 16 位复位寄存器（GPIOx_BRR）和一个 32 位锁定寄存器（GPIOx_LCKR）。

每个 I/O 端口位都可以自由编程，然而 I/O 端口寄存器必须按 32 位字被访问（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/更改进行独立访问；这样，在读和更改访问之间产生中断请求（IRQ）时不会发生意外。如图 3-6 所示，给出了一个 I/O 端口位的内部结构。

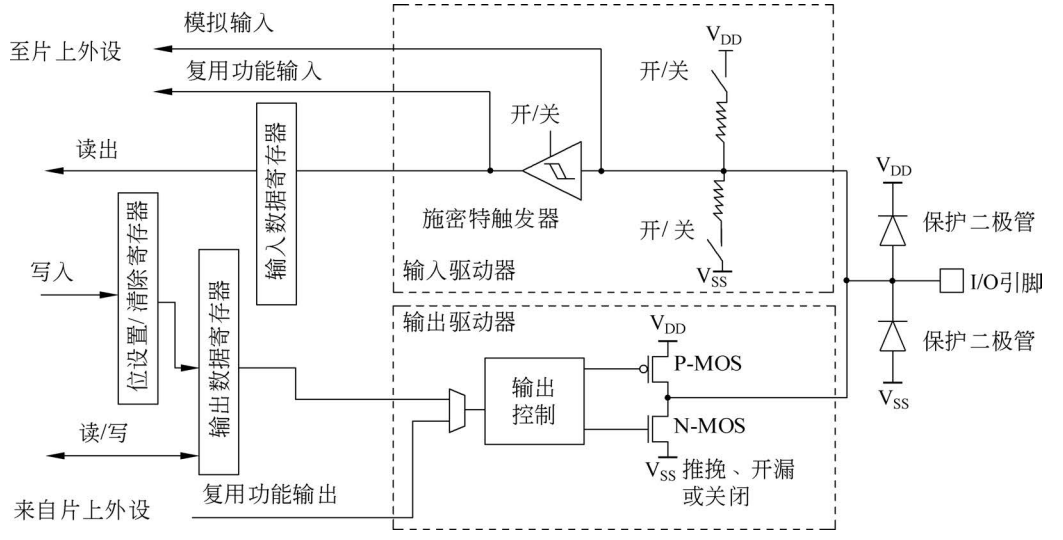

图 3-6　I/O 端口位的内部结构

如图 3-6 所示，除了最右侧的“I/O 引脚”是外界和芯片交互的出入口外，其他都是在芯片内部的。每个 I/O 端口以保护二极管、推挽开关、施密特触发器为核心实现了非常灵活

的功能。

(1) 保护二极管：I/O 引脚上下两边两个二极管用于防止引脚外部过高、过低的电压输入。当引脚电压高于 V_{DD} 时，上方的二极管导通；当引脚电压低于 V_{SS} 时，下方的二极管导通，防止不正常电压引入芯片而导致芯片烧毁。尽管如此，还是不能直接外接大功率器件，需要大功率及隔离电路驱动，防止烧坏芯片或者外接器件无法正常工作。

(2) P-MOS 管和 N-MOS 管：由 P-MOS 管和 N-MOS 管组成的单元电路使 GPIO 具有“推挽输出”和“开漏输出”模式。这里的电路会在下面详细分析。

(3) TTL 施密特触发器：信号经过触发器后，模拟信号会被转换为 0 和 1 的数字信号，但是，当 GPIO 引脚作为 ADC 采集电压的输入通道时，用其“模拟输入”功能，此时信号不再经过触发器进行 TTL 电平转换。


19min

3.2.2 输出工作模式

1. 开漏输出

在开漏输出模式下，通过设置“位设置/清除寄存器”或者“输出数据寄存器”的值，途经 N-MOS 管，最终输出到 I/O 端口，如图 3-7 所示。

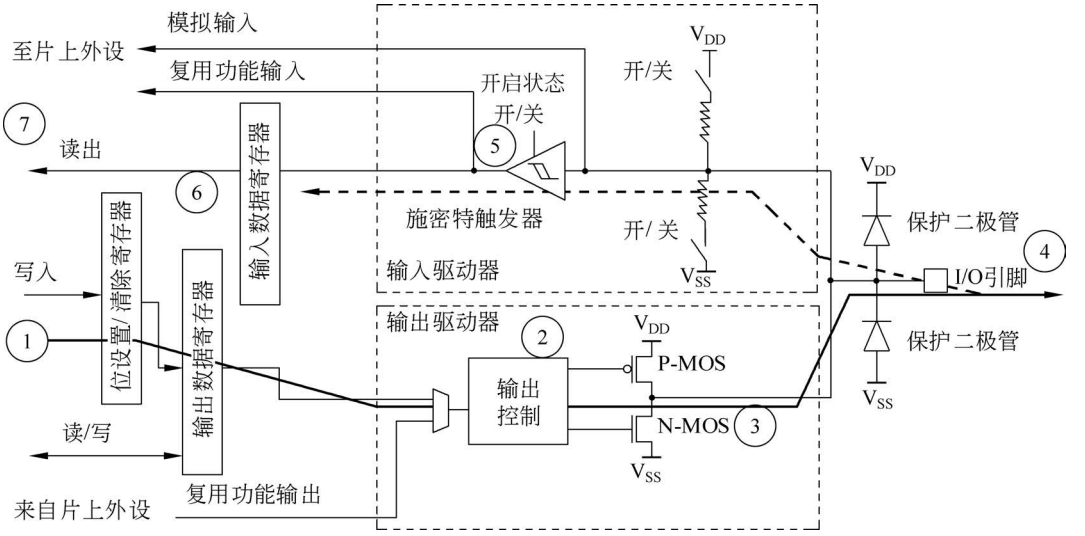

图 3-7 开漏输出模式

这里需要注意 N-MOS 管，当设置输出的值为高电平时，N-MOS 管处于关闭状态，此时 I/O 端口的电平就不会由输出的高低电平决定，I/O 此时表现为高阻态，I/O 端口的电平由外部的上拉电阻决定；当设置输出的值为低电平时，N-MOS 管处于开启状态，此时 I/O 端口的电平就是低电平。也就是说，当 I/O 端口被配置为开漏输出模式时，一般需要在外部配合上拉电阻使用。

同时，I/O 端口的电平也可以通过输入电路进行读取；当开漏输出时，I/O 端口的电平不一定是输出的电平（高电平时）。

2. 开漏复用输出

开漏复用输出模式与开漏输出模式类似。只是输出的高低电平的来源，不是让 CPU 直接写输出数据寄存器，取而代之利用片上外设模块的复用功能输出来决定。

如图 3-8 所示，①处的电平直接控制输出控制电路②，而①处的电平是由片上的外设给出的，如用作 USART、SPI 通信端口时。

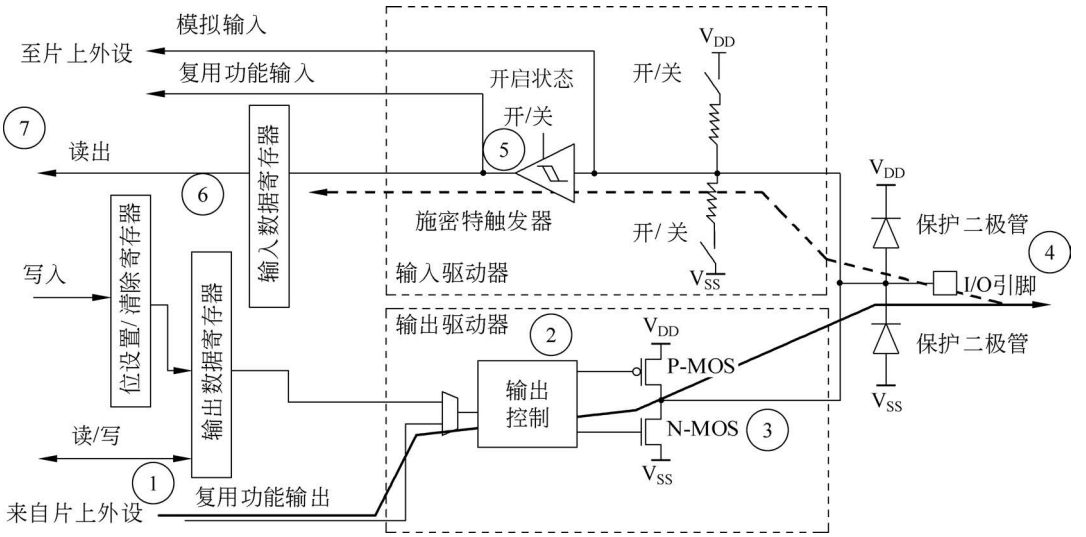

图 3-8　开漏复用输出模式

3. 推挽输出

在推挽输出模式下，通过设置“位设置/清除寄存器”或者“输出数据寄存器”的值，途经 P-MOS 管和 N-MOS 管，最终输出到 I/O 端口。

这里需要注意 P-MOS 管和 N-MOS 管状态与图 3-7 中开漏输出不同，当将输出的值设置为高电平时，P-MOS 管处于开启状态，N-MOS 管处于关闭状态，此时 I/O 端口的电平就由 P-MOS 管决定，为高电平；当将输出的值设置为低电平时，P-MOS 管处于关闭状态，N-MOS 管处于开启状态，此时 I/O 端口的电平就由 N-MOS 管决定，为低电平。

同时，I/O 端口的电平也可以通过输入电路进行读取；注意，此时读到的 I/O 端口的电平一定是输出的电平。

4. 推挽复用输出

推挽复用输出模式与推挽输出模式类似。只是输出的高低电平的来源，不是让 CPU 直接写输出数据寄存器，取而代之利用片上外设模块的复用功能输出来决定。

与图 3-8 类似，只是①处的电平直接控制输出控制电路②，而①处的电平是由片上的外设给出的，如用作 I2C 通信端口时。


24min

3.2.3 输入工作模式

GPIO的输入工作模式包括浮空输入(GPIO_Mode_IN_FLOATING)、上拉输入(GPIO_Mode_IPU)、下拉输入(GPIO_Mode_IPD)、模拟输入(GPIO_Mode_AIN)共4种，下面分别予以说明。

1. 浮空输入

在浮空输入模式下，输入驱动器中的上、下拉开关均打开，I/O端口的电平信号直接进入输入数据寄存器。也就是说，当I/O端口有电平输入时，I/O的电平状态完全由外部输入决定；如果在该引脚悬空(在无信号输入)的情况下读取该端口的电平，则是不确定的。

若I/O端口①处为低电平，②处电平和①处电平相同，也为低电平，经过TTL施密特触发器后，③处变为数字信号0，从而进入输入数据寄存器；若I/O端口①处为高电平，②处电平和①处电平相同，也为高电平，经过TTL施密特触发器后，③处变为数字信号1，从而进入输入数据寄存器；若I/O端口①处悬空，②处电平未知，经过TTL施密特触发器后，③处的数字信号未知，如图3-9所示。

浮空输入一般多用于外部按键输入。

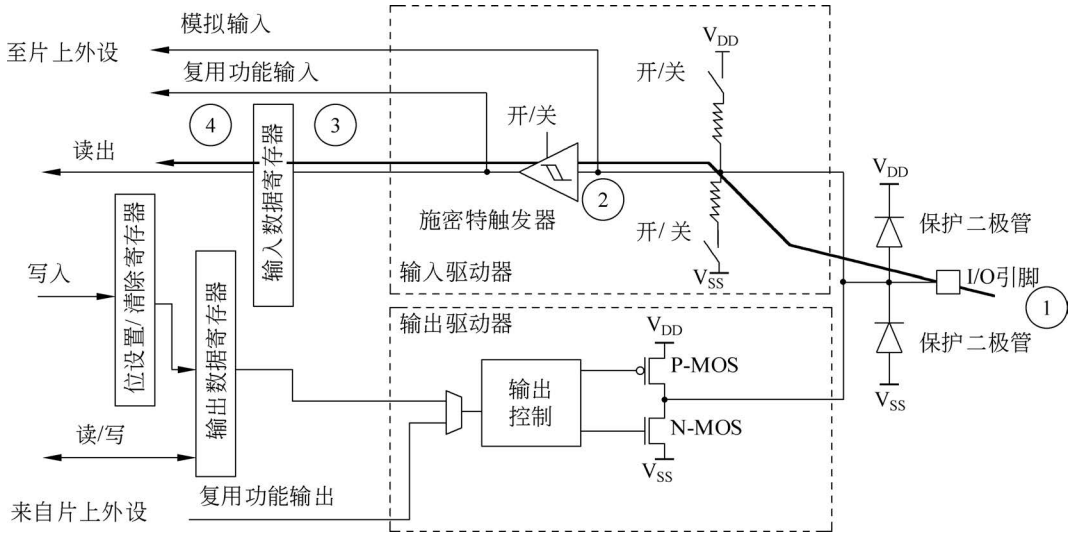

图3-9 浮空输入模式

2. 上拉输入

在上拉输入模式下，输入驱动器中的上拉开关闭合，I/O端口的电平信号直接进入输入数据寄存器，但是在I/O端口悬空(在无信号输入)的情况下，输入端的电平可以保持在高电平，并且在I/O端口输入为低电平时，输入端的电平也还是低电平。

若I/O端口①处为高或低电平的输入时，情况和浮空输入模式下相同；若I/O端口①处悬空，②处电平为高电平V_{DD}，经过TTL施密特触发器后，③处为1，如图3-10所示。

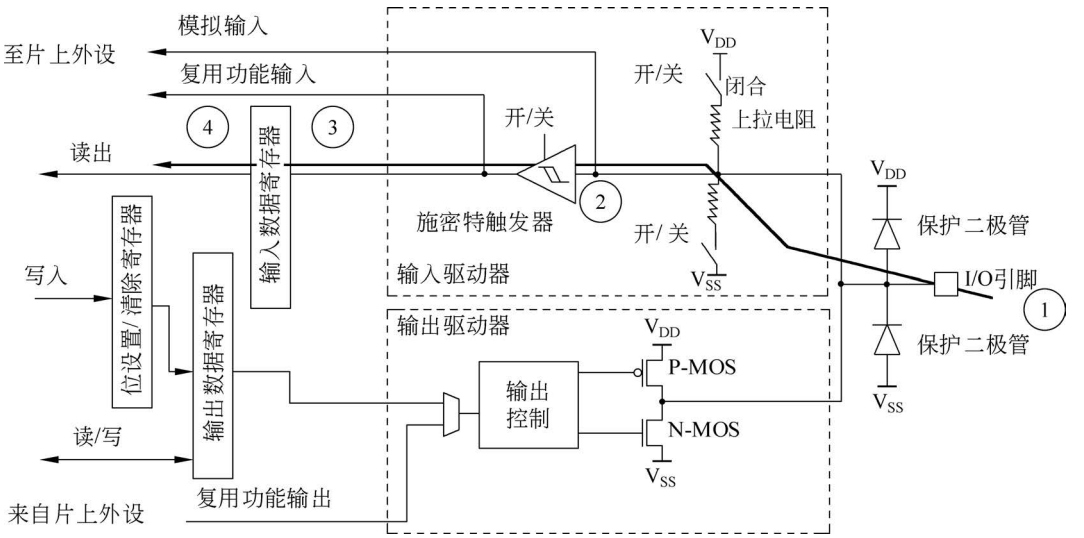

图 3-10　上拉输入模式

3. 下拉输入

在下拉输入模式下，I/O 端口的电平信号直接进入输入数据寄存器，但是在 I/O 端口悬空（在无信号输入）的情况下，输入端的电平可以保持在低电平，并且在 I/O 端口输入为高电平时，输入端的电平也还是高电平。

若 I/O 端口①处为高或低电平的输入时，情况和浮空输入模式下相同；若 I/O 端口①处悬空，②处电平为低电平 V_{SS}，经过 TTL 施密特触发器后，③处的数字信号为 0。如图 3-11 所示。

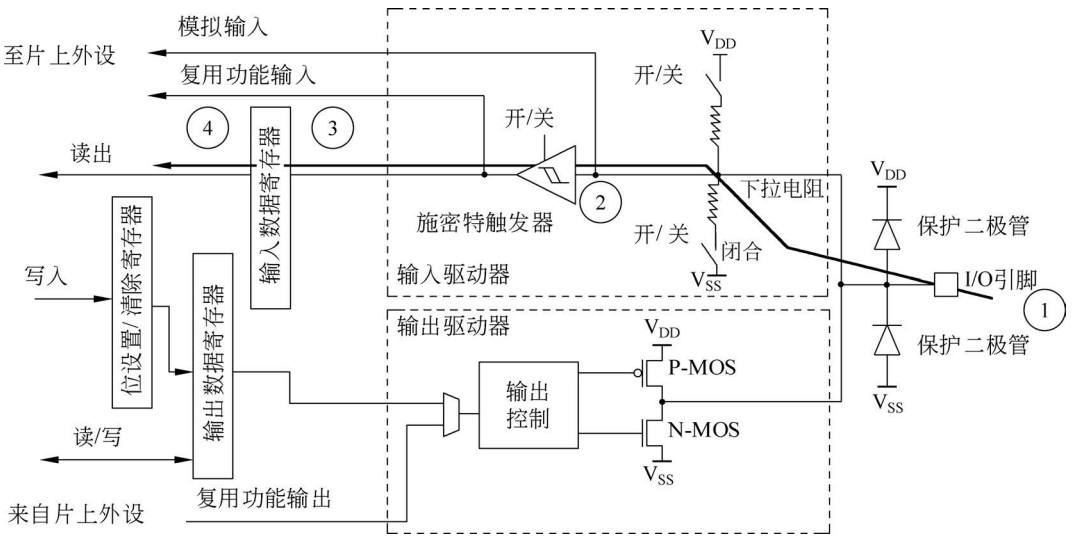

图 3-11　下拉输入模式

数字电路有3种状态：高电平、低电平和高阻状态，有些应用场合不希望出现高阻状态，可以通过上拉电阻或下拉电阻的方式使其处于稳定状态，具体选择上拉输入还是下拉输入视具体应用场景而定。

4. 模拟输入

在模拟输入模式下，I/O端口的模拟信号(电压信号，而非电平信号)直接输入片上外设模块，例如ADC模块等。

信号从I/O端口①进入，从另一端②直接进入片上模块。此时，所有的上拉、下拉电阻和施密特触发器均处于断开状态，因此“输入数据寄存器”将不能反映端口①上的电平状态，也就是说，在模拟输入配置下，CPU不能在“输入数据寄存器”上读到有效的数据。如图3-12所示。

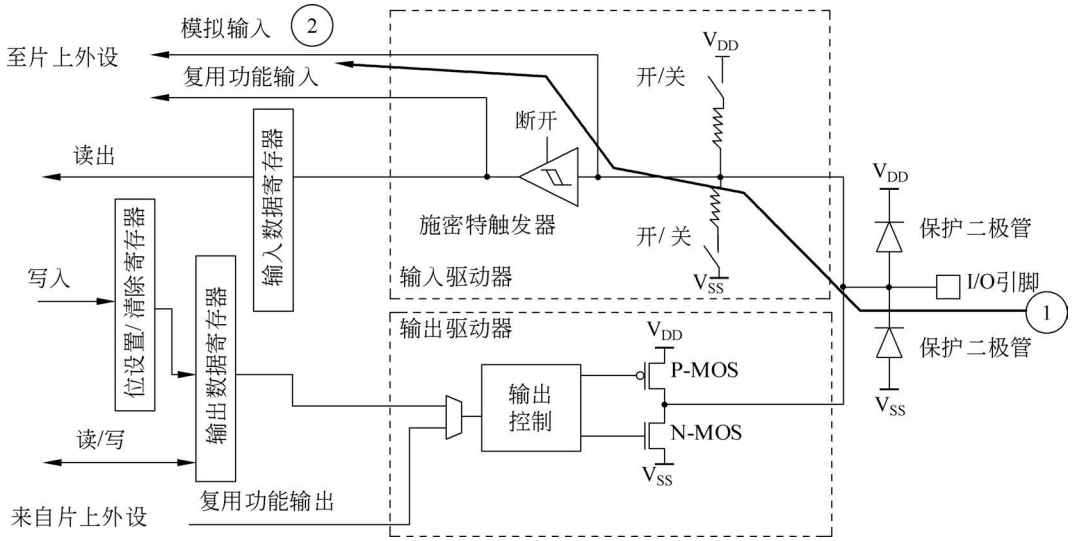

图3-12 模拟输入模式

模拟输入的信号是未经数字化处理的电压信号，可以直接供芯片内部的ADC使用。


19min

3.3 GPIO主要寄存器简介

由3.2节中的GPIO原理图可知，GD32F10x系列微控制器的GPIO模块使用了一系列寄存器来配置和控制引脚的功能和属性。每个GPIO端口都有一组寄存器与之对应，用于配置和控制该端口的引脚，这些寄存器的地址是通过基址寄存器(Base Register)和偏移量(Offset)计算得出的。

基址寄存器：每个GPIO端口都有一个基址寄存器，用于确定该端口的起始地址。在GD32F10x系列中，每个GPIO端口的基址寄存器见表3-1。

表 3-1　GPIO 端口的基址寄存器

GPIO 组	寄存器基地址
GPIOA	0x40010800
GPIOB	0x40010C00
GPIOC	0x40011000
GPIOD	0x40011400
GPIOE	0x40011800
GPIOF	0x40011C00
GPIOG	0x40012000

偏移量：每个 GPIO 端口的寄存器地址与基址寄存器之间的偏移量确定了特定寄存器的地址。在 GD32F10x 系列中，常用的 GPIO 寄存器偏移量及对应功能见表 3-2。

表 3-2　GPIO 寄存器偏移量及对应功能

寄存器类型	偏移量(功能)
CTL0	0x00(配置低 8 位引脚)
CTL1	0x04(配置高 8 位引脚)
ISTAT	0x08(输入数据寄存器)
OCTL	0x0C(端口输出控制寄存器)
BOP	0x10(端口位操作寄存器)
BC	0x14(位清除寄存器)
LOCK	0x18(端口配置锁定寄存器)

通过将基址寄存器与偏移量相加，即可计算得到特定寄存器的地址。例如，要访问 GPIOA 的 CTL0 寄存器，可以使用以下地址计算公式：

GPIOA_ CTL0 = GPIOA_BASE + 0x00 = 0x40010800 + 0x00 = 0x40010800

同样地，要访问 GPIOA 的 OCTL 寄存器，可以使用以下地址计算公式：

GPIOA_OCTL = GPIOA_BASE + 0x0C = 0x40010800 + 0x0C = 0x4001080C

需要注意的是，以上地址计算公式适用于 GD32F10x 系列微控制器中的 GPIO 模块，其他型号的 GD32 系列微控制器可能存在微小差异。在实际编程中，需要参考 GD32F10x 的技术手册和参考手册中提供的基址寄存器和偏移量定义，或者直接调用 GD 官方提供的固件库中相应的接口函数实现。

3.3.1　端口控制寄存器

1. 端口控制寄存器 0(GPIOx_CTL0，x=A..G)

端口控制寄存器 0(GPIOx_CTL0)用来配置 GPIOx 的低 8 个引脚的工作状态，如图 3-13 所示。该寄存器的地址偏移量为 0x00，复位值为 0x4444 4444，该寄存器只能按字(32 位)访问。

端口控制寄存器 0 的位描述见表 3-3。

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
CTL7[1:0]	MD7[1:0]	CTL6[1:0]	MD6[1:0]	CTL5[1:0]	MD5[1:0]	CTL4[1:0]	MD4[1:0]
rw	rw	rw	rw	rw	rw	rw	rw
15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
CTL3[1:0]	MD3[1:0]	CTL2[1:0]	MD2[1:0]	CTL1[1:0]	MD1[1:0]	CTL0[1:0]	MD0[1:0]
rw	rw	rw	rw	rw	rw	rw	rw

图 3-13　端口控制寄存器 0

表 3-3　端口控制寄存器 0 的位描述

位/位域	名　　称	描　　述
31:30	CTL7[1:0]	Port 7 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
29:28	MD7[1:0]	Port 7 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
27:26	CTL6[1:0]	Port 6 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述
25:24	MD6[1:0]	Port 6 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
23:22	CTL5[1:0]	Port 5 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述
21:20	MD5[1:0]	Port 5 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
19:18	CTL4[1:0]	Port 4 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述
17:16	MD4[1:0]	Port 4 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
15:14	CTL3[1:0]	Port 3 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述
13:12	MD3[1:0]	Port 3 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
11:10	CTL2[1:0]	Port 2 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述
9:8	MD2[1:0]	Port 2 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
7:6	CTL1[1:0]	Port 1 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述

续表

位/位域	名　　称	描　　述
5:4	MD1[1:0]	Port 1 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述
3:2	CTL0[1:0]	Port 零配置位，该位由软件置位和清除。 输入模式(MD[1:0]=00)。 00：模拟输入 01：浮空输入 10：上拉输入 /下拉输入 11：保留 输出模式 (MD[1:0]>00)。 00：GPIO 推挽输出 01：GPIO 开漏输出 10：AFIO 推挽输出 11：AFIO 开漏输出
1:0	MD0[1:0]	Port 0 模式位，该位由软件置位和清除。 00：输入模式 (复位状态) 01：输出模式(10MHz) 10：输出模式(20MHz) 11：输出模式(50MHz)

例如，要将 GPIOA 的 Port0 引脚配置为模拟输入，需要将 GPIOA_CTL0 寄存器的第1～0bit(MD0[1:0])设置为 00(代表输入模式)，再将 GPIOA_CTL0 寄存器的第 3～2 位(CTL0[1:0])设置为 00(当 MD0[1:0]为 00 时代表模拟输入)。

2. 端口控制寄存器 1(GPIOx_CTL1，x=A..G)

端口控制寄存器 1(GPIOx_CTL1)用来配置 GPIOx 的高 8 个引脚的工作状态，如图 3-10 所示。该寄存器的地址偏移量为 0x04，复位值为 0x4444 4444，该寄存器只能按字(32 位)访问。

31 30	29 28	27 26	25 24	23 22	21 20	19 18	17 16
CTL15[1:0]	MD15[1:0]	CTL14[1:0]	MD14[1:0]	CTL13[1:0]	MD13[1:0]	CTL12[1:0]	MD12[1:0]
rw	rw	rw	rw	rw	rw	rw	rw
15 14	13 12	11 10	9 8	7 6	5 4	3 2	1 0
CTL11[1:0]	MD11[1:0]	CTL10[1:0]	MD10[1:0]	CTL9[1:0]	MD9[1:0]	CTL8[1:0]	MD8[1:0]
rw	rw	rw	rw	rw	rw	rw	rw

图 3-14　端口控制寄存器 1

端口控制寄存器 1 的位描述见表 3-4。

表 3-4　端口控制寄存器 1 的位描述

位/位域	名　　称	描　　述
31:30	CTL15[1:0]	Port 15 配置位，该位由软件置位和清除。 参考 CTL0[1:0]的描述
29:28	MD15[1:0]	Port 15 模式位，该位由软件置位和清除。 参考 MD0 [1:0]的描述

续表

位/位域	名　　称	描　　述
27:26	CTL14[1:0]	Port 14 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
25:24	MD14[1:0]	Port 14 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述
23:22	CTL13[1:0]	Port13 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
21:20	MD13[1:0]	Port 13 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述
19:18	CTL12[1:0]	Port 12 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
17:16	MD12[1:0]	Port 12 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述
15:14	CTL11[1:0]	Port 11 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
13:12	MD11[1:0]	Port 11 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述
11:10	CTL10[1:0]	Port 10 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
9:8	MD10[1:0]	Port10 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述
7:6	CTL9[1:0]	Port 9 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
5:4	MD9[1:0]	Port 9 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述
3:2	CTL8[1:0]	Port 8 配置位,该位由软件置位和清除。 参考 CTL0[1:0]的描述
1:0	MD8[1:0]	Port 8 模式位,该位由软件置位和清除。 参考 MD0 [1:0]的描述

GPIOx_CTL1 的使用方法与 GPIOx_CTL0 类似。

3.3.2　端口输入状态寄存器(GPIOx_ISTAT,x=A..G)

端口输入状态寄存器(GPIOx_ISTAT)用来标识 GPIOx 引脚的输入状态,如图 3-15 所示。该寄存器的地址偏移量为 0x08,复位值为 0x0000 XXXX,该寄存器只能按字(32 位)访问。

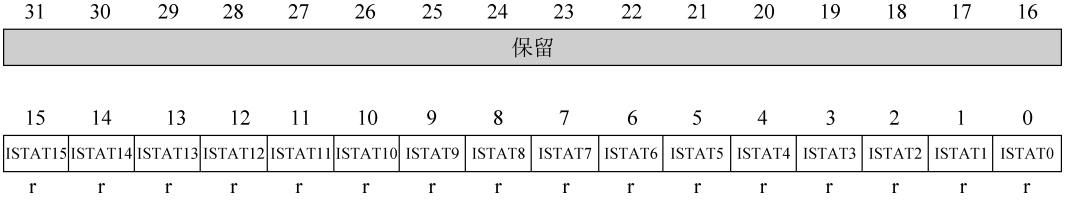

图 3-15　端口输入状态寄存器

端口输入状态寄存器的位描述见表 3-5。

表 3-5　端口输入状态寄存器的位描述

位/位域	名　　称	描　　述
31:16	保留	必须保持复位值
15:0	ISTATy	端口输入状态位(y=0…15),这些位由软件置位和清除。 0：引脚输入信号为低电平 1：引脚输入信号为高电平

例如,当寄存器 GPIOA_ISTAT 的 ISTAT7 位为 1 时表示 GPIOA 的 PIN7 引脚的输入信号为高电平。

3.3.3　端口输出控制寄存器(GPIOx_OCTL,x=A..G)

端口输出控制寄存器(GPIOx_OCTL)用来控制 GPIOx 引脚的输出状态,如图 3-16 所示。该寄存器的地址偏移量为 0x0C,复位值为 0x0000 0000,该寄存器只能按字(32 位)访问。

图 3-16　端口输出控制寄存器

端口输出控制寄存器的位描述见表 3-6。

表 3-6　端口输出控制寄存器的位描述

位/位域	名　　称	描　　述
31:16	保留	必须保持复位值
15:0	OCTLy	端口输出控制位(y=0…15),这些位由软件置位和清除。 0：引脚输出低电平 1：引脚输出高电平

例如,若要 GPIOA 的 PIN7 引脚输出高电平,只需将 GPIOA_ISTAT 的 OCTL7 位置为 1。

3.3.4　端口位操作与位清除寄存器

1. 端口位操作寄存器(GPIOx_BOP,x=A..G)

端口位操作寄存器(GPIOx_BOP)用来控制 GPIOx_OCTL 寄存器,如图 3-17 所示。该寄存器的地址偏移量为 0x10,复位值为 0x0000 0000,该寄存器只能按字(32 位)访问。

端口位操作寄存器的位描述见表 3-7。

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CR15	CR14	CR13	CR12	CR11	CR10	CR9	CR8	CR7	CR6	CR5	CR4	CR3	CR2	CR1	CR0
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
BOP15	BOP14	BOP13	BOP12	BOP11	BOP10	BOP9	BOP8	BOP7	BOP6	BOP5	BOP4	BOP3	BOP2	BOP1	BOP0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

图 3-17　端口位操作寄存器

表 3-7　端口位操作寄存器的位描述

位/位域	名　　称	描　　述
31:16	CRy	端口清除位 y(y=0…15),这些位由软件置位和清除。 0：相应的 OCTLy 位没有改变 1：将相应的 OCTLy 位清除为 0
15:0	BOPy	端口置位 y(y=0…15),这些位由软件置位和清除。 0：相应的 OCTLy 位没有改变 1：将相应的 OCTLy 位设置为 1

例如,若想让 GPIOA 的 PIN7 引脚的输出值变为 0,而其他位不变,只需将 GPIOA_BOP 的 CR7 置为 1,将其他的 CR 位置为 0；若想要 GPIOA 的 PIN7 引脚的输出值变为 1,而其他位不变,只需将 GPIOA_BOP 的 BOP7 置为 1,将其他的 BOP 位置为 0。

2. 端口位清除寄存器(GPIOx_BC,x=A..G)

端口位清除寄存器(GPIOx_BC)用来控制 GPIOx_OCTL 寄存器,如图 3-18 所示。该寄存器的地址偏移量为 0x14,复位值为 0x0000 0000,该寄存器只能按字(32 位)访问。

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
CR15	CR14	CR13	CR12	CR11	CR10	CR9	CR8	CR7	CR6	CR5	CR4	CR3	CR2	CR1	CR0
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w

图 3-18　端口位清除寄存器

端口位清除寄存器的位描述见表 3-8。

表 3-8　端口位清除寄存器的位描述

位/位选	名　　称	描　　述
31:16	保留	必须保持复位值
15:0	CRy	端口清除位 y(y=0…15),这些位由软件置位和清除。 0：相应的 OCTLy 位没有改变 1：清除相应的 OCTLy 位

该寄存器的用法与 GPIOx_BOP 的 CR 部分的用法相同。

3.3.5　端口配置锁定寄存器(GPIOx_LOCK,x=A,B)

端口配置锁定寄存器(GPIOx_ LOCK)用来锁定特定的 I/O 端口是否可以被改变,如图 3-19 所示。该寄存器的地址偏移量为 0x18,复位值为 0x0000 0000,该寄存器只能按字(32 位)访问。

通过设置 GPIOx_LOCK 寄存器的特定值,可以锁定相关的 GPIO 寄存器,防止对其进行进一步的配置更改。这可以防止意外或恶意地修改 GPIO 配置,以确保 GPIO 的稳定性和可靠性。

图 3-19　端口配置锁定寄存器

端口配置锁定寄存器的位描述见表 3-9。

表 3-9　端口配置锁定寄存器的位描述

位/位域	名　　称	描　　述
31:17	保留	必须保持复位值
16	LKK	锁定序列键。 该位只能通过 Lock Key 写序列设置,始终可读。 0: GPIO_LOCK 寄存器和端口配置没有锁定 1: 直到下一次 MCU 复位前,GPIO_LOCK 寄存器被锁定 LOCK Key 写序列: 写 1→写 0→写 1→读 0→读 1。 注意: 在 LOCK Key 写序列期间,LK[15:0]的值必须保持
15:0	LKy	端口锁定位 y(y=0…15),这些位由软件置位和清除。 0: 相应的端口位配置没有锁定 1: 当 LKK 位置 1 时,相应的端口位配置被锁定

当执行正确的写序列设置了 LKK 位时,该寄存器用来锁定端口位的配置。位[15:0]用于锁定 GPIO 端口的配置。在规定的写入操作期间,不能改变 LCKP[15:0]。当对相应的端口位执行了 LOCK 序列后,在下次系统复位之前将不能再更改端口位的配置。

3.3.6　AFIO 端口配置寄存器 0(AFIO_PCF0)

AFIO 端口配置寄存器 0(AFIO_PCF0)是与芯片的引脚重映射(Pin Remap)相关的寄存器。该寄存器的地址偏移量为 0x04,复位值为 0x0000 0000,该寄存器只能按字(32 位)访问。

中密度、高密度和超高密度产品寄存器内存映射和位定义如图 3-20 所示。

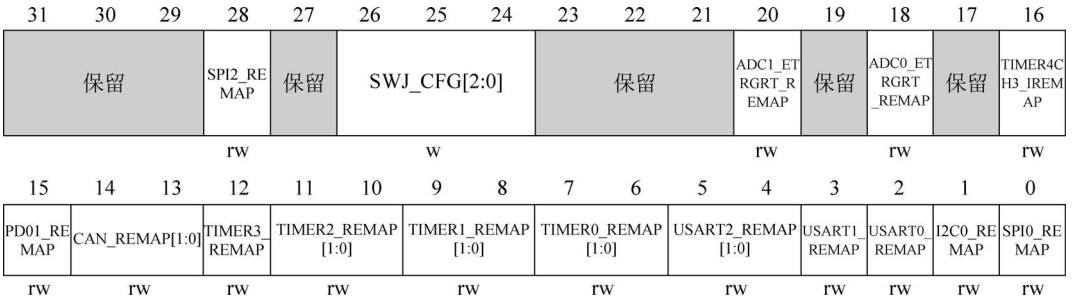

图 3-20　寄存器内存映射和位定义

AFIO 端口配置寄存器 0 的位描述见表 3-10。

表 3-10　AFIO 端口配置寄存器 0 的位描述

位/位域	名　　称	描　　述
31:29	保留	必须保持复位值
28	SPI2_REMAP	SPI2/I2S2 重映射。 该位由软件置位和清除。 0：关闭重映射功能(SPI2_NSS-I2S2_WS/PA15，SPI2_SCK-I2S2_CK/PB3，SPI2_MISO/PB4，SPI2_MOSI-I2S_SD/PB5) 1：完全开启重映射功能(SPI2_NSS-I2S2_WS/PA4，SPI2_SCK-I2S2_CK/PC10，SPI2_MISO/PC11，SPI2_MOSI-I2S_SD/PC12) 注意：该位只在高密度产品和超高密度产品中可用，在其他系列中为保留位
27	保留	必须保持复位值
26:24	SWJ_CFG[2:0]	串行线 JTAG 配置。 这些位只写(如果读这些位，则将返回未定义值)。 000：JTAG-DP 使能和 SW-DP 使能(复位状态) 001：JTAG-DP 使能和 SW-DP 使能但没有 NJTRST 010：JTAG-DP 禁用和 SW-DP 使能 100：JTAG-DP 禁用和 SW-DP 禁用 其他：未定义
23:21	ADC1_ETRGRT_REMAP	ADC 1 常规转换外部触发重映射。 0：连接 ADC1 常规转换外部触发与 EXTI11 1：连接 ADC1 常规转换外部触发与 TIM7_TRGO
19	保留	必须保持复位值
18	ADC0_ETRGRT_REMAP	ADC 0 常规转换外部触发重映射。 该位由软件置位和清除。 0：连接 ADC0 常规转换外部触发与 EXTI11 1：连接 ADC0 常规转换外部触发与 TIM7_TRGO
17	保留	必须保持复位值

续表

位/位域	名　　称	描　　述
16	TIMER4CH3_IREMAP	TIMER 四通道 3 内部重映射。 该位由软件置位和清除。 0：连接 TIMER4_CH3 与 PA3 1：连接 TMER4_CH3 与 IRC40K 内部时钟，用于对 IRC40K 进行校准 注意：该位在高密度和超高密度产品线中可用
15	PD01_REMAP	OSC_IN/OSC_OUT 重映射到 Port D0/Port D1。 该位由软件置位和清除。 0：关闭重映射功能 1：OSC_IN 重映射到 PD0，OSC_OUT 重映射到 PD1
14:13	CAN _ REMAP [1:0]	CAN 接口重映射。 这些位由软件置位和清除。 00：关闭重映射功能(CAN_RX/ PA11，CAN_TX/PA12) 01：没有使用 10：开启重映射部分功能(CAN_RX/PB8，CAN_TX/PB9) 11：完全开启重映射功能(CAN_RX/PD0，CAN_TX/PD1)
12	TIMER3_REMAP	TIMER3 重映射。 该位由软件置位和清除。 0：关闭重映射功能(TIMER3_CH0/PB6，TIMER3_CH1/PB7，TIMER3_CH2/PB8，TIMER3_CH3/PB9) 1：完全开启重映射功能(TIMER3_CH0/PD12，TIMER3_CH1/PD13，TIMER3_CH2/PD14，TIMER3_CH3/PD15)
11:10	TIMER2_REMAP [1:0]	TIMER2 重映射。 这些位由软件置位和清除。 00：关闭重映射功能(TIMER2 _ CH0/PA6，TIMER2 _ CH1/PA7，TIMER2_CH2/PB0，TIMER2_CH3/PB1) 01：没有使用 10：开启重映射部分功能(TIMER2 _ CH0/PB4，TIMER2 _ CH1/PB5，TIMER2_CH2/PB0，TIMER2_CH3/PB1) 11：完全开启重映射功能(TIMER2_CH0/PC6，TIMER2_CH1/PC7，TIMER2_CH2/PC8，TIMER2_CH3/PC9)
9:8	TIMER1_REMAP [1:0]	TIMER1 重映射。 这些位由软件置位和清除。 00：关闭重映射功能(TIMER1 _ CH0/TIMER1 _ ETI/PA0，TIMER1 _ CH1/PA1，TIMER1_CH2/PA2，TIMER1_CH3/PA3) 01：开启重映射部分功能(TIMER1 _ CH0/TIMER1 _ ETI/PA15，TIMER1_CH1/PB3，TIMER1_CH2/PA2，TIMER1_CH3/PA3) 10：开启重映射部分功能(TIMER1_CH0/TIMER1_ETI/PA0，TIMER1_CH1/PA1，TIMER1_CH2/PB10，TIMER1_CH3/PB11) 11：完全开启重映射功能(TIMER1 _ CH0/TIMER1 _ ETI/PA15，TIMER1_CH1/PB3，TIMER1_CH2/PB10，TIMER1_CH3/PB11)

续表

位/位域	名　　称	描　　述
7:6	TIMER0_REMAP[1:0]	TIMER0 重映射。 这些位由软件置位和清除。 00：关闭重映射功能（TIMER0_ETI/PA12，TIMER0_CH0/PA8，TIMER0_CH1/PA9，TIMER0_CH2/PA10，TIMER0_CH3/PA11，TIMER0_BKIN/PB12，TIMER0_CH0_ON/PB13，TIMER0_CH1_ON/PB14，TIMER0_CH2_ON/PB15） 01：开启重映射部分功能（TIMER0_ETI/PA12，TIMER0_CH0/PA8，TIMER0_CH1/PA9，TIMER0_CH2/PA10，TIMER0_CH3/PA11，TIMER0_BKIN/PA6，TIMER0_CH0_ON/PA7，TIMER0_CH1_ON/PB0，TIMER0_CH2_ON/PB1） 10：没有使用 11：完全开启重映射功能（TIMER0_ETI/PE7，TIMER0_CH0/PE9，TIMER0_CH1/PE11，TIMER0_CH2/PE13，TIMER0_CH3/PE14，TIMER0_BKIN/PE15，TIMER0_CH0_ON/PE8，TIMER0_CH1_ON/PE10，TIMER0_CH2_ON/PE12）
5:4	USART2_REMAP[1:0]	USART2 重映射。 这些位由软件置位和清除。 00：关闭重映射功能（USART2_TX/PB10，USART2_RX/PB11，USART2_CK/PB12，USART2_CTS/PB13，USART2_RTS/PB14） 01：开启重映射部分功能（USART2_TX/PC10，USART2_RX/PC11，USART2_CK/PC12，USART2_CTS/PB13，USART2_RTS/PB14） 10：没有使用 11：完全开启重映射功能（USART2_TX/PD8，USART2_RX/PD9，USART2_CK/PD10，USART2_CTS/PD11，USART2_RTS/PD12）
3	USART1_REMAP	USART1 重映射。 该位由软件置位和清除。 0：关闭重映射功能（USART1_CTS/PA0，USART1_RTS/PA1，USART1_TX/PA2，USART1_RX/PA3，USART1_CK/PA4） 1：开启重映射功能（USART1_CTS/PD3，USART1_RTS/PD4，USART1_TX/PD5，USART1_RX/PD6，USART1_CK/PD7）
2	USART0_REMAP	USART0 重映射。 该位由软件置位和清除。 0：关闭重映射功能（USART0_TX/PA9，USART0_RX/PA10） 1：开启重映射功能（USART0_TX/PB6，USART0_RX/PB7）
1	I2C0_REMAP	I2C0 重映射。 该位由软件置位和清除。 0：关闭重映射功能（I2C0_SCL/PB6，I2C0_SDA/PB7） 1：开启重映射功能（I2C0_SCL/PB8，I2C0_SDA/PB9）

续表

位/位域	名　　称	描　　述
0	SPI0_REMAP	SPI0 重映射。 该位由软件置位和清除。 0：关闭重映射功能（SPI0_NSS/PA4，SPI0_SCK/PA5，SPI0_MISO/PA6，SPI0_MOSI/PA7） 1：开启重映射功能（SPI0_NSS/PA15，SPI0_SCK/PB3，SPI0_MISO/PB4，SPI0_MOSI/PB5）

AFIO（Alternate Function I/O）提供了一种机制，使单个 GPIO 引脚可以具备多种功能。每个 GPIO 引脚通常具有一个默认的功能，例如作为普通的数字输入/输出，但通过 AFIO 功能，可以将 GPIO 引脚配置为另外的功能，例如模拟输入、定时器输入/输出、串行通信接口等。

通过 AFIO，用户可以灵活地重新分配 GPIO 引脚的功能，以满足不同应用的需求。这种引脚功能的重新分配称为引脚重映射（Pin Remap）。引脚重映射允许用户将 GPIO 引脚与特定的功能模块相连接，从而实现所需的输入或输出操作。

3.4　GPIO 常用库函数介绍

针对 GD32F10x，GD 官方的标准库中有一系列与 GPIO 相关的库函数，这些库函数为开发者提供了一种高层次的抽象，使在编程时不需要直接操作底层硬件寄存器，而是通过简单的函数调用来配置和操作 GPIO 端口的输入/输出、中断等功能，常用的库函数见表 3-11。

表 3-11　GPIO 常用的库函数

库函数名称	库函数描述
gpio_deinit	复位外设 GPIOx
gpio_afio_deinit	复位 AFIO
gpio_init	GPIO 参数初始化
gpio_bit_set	置位引脚值
gpio_bit_reset	复位引脚值
gpio_bit_write	将特定的值写入指定的引脚
gpio_port_write	将特定的值写入指定的一组端口
gpio_input_bit_get	获取引脚的输入值
gpio_input_port_get	获取一组端口的输入值
gpio_output_bit_get	获取引脚的输出值
gpio_output_port_get	获取一组端口的输出值
gpio_pin_remap_config	配置 GPIO 引脚重映射
gpio_exti_source_select	选择哪个引脚作为 EXTI 源
gpio_event_output_config	配置事件输出
gpio_event_output_enable	事件输出使能
gpio_event_output_disable	事件输出除能

续表

库函数名称	库函数描述
gpio_pin_lock	相应的引脚配置被锁定
gpio_ethernet_phy_select	以太网 MII 或 RMII PHY 选择

概括起来有 4 类：①GPIO 初始化，可以设置 GPIO 引脚的输入/输出模式、引脚类型、上下拉电阻、输出速度等参数；②GPIO 输入/输出控制，其作用是允许开发者在程序中灵活地控制和读取 GPIO 引脚的状态；③GPIO 中断控制，可以帮助开发者配置和控制 GPIO 引脚的中断功能，借此实现响应外部事件的功能，例如，按键按下、传感器状态变化等(有关中断知识会在第 4 章讲解)；④GPIO 状态读取，使开发者可以方便地获取 GPIO 引脚的输入或输出状态，从而了解外部设备或信号的当前状态。具体的库函数名称及其简介见表 3-11，需要注意的是本书所介绍的标准库函数版本号为 2.1，随着时间的推移 GD 官方的库函数的名称、参数等可能会发生变化。

3.4.1 初始化函数

1. gpio_init()

gpio_init()是 GPIO 初始化的函数，参数见表 3-12。

表 3-12 gpio_init()函数参数

参 数	描 述
函数原型	void gpio_init(uint32_t gpio_periph, uint32_t mode, uint32_t speed, uint32_t pin);
功能描述	GPIO 参数初始化
输入 1：gpio_periph	GPIO 端口 输入值：GPIOx：端口选择(x = A,B,C,D,E,F,G)
输入 2：mode	GPIO 引脚模式 输入值： GPIO_MODE_AIN //模拟输入模式 GPIO_MODE_IN_FLOATING //浮空输入模式 GPIO_MODE_IPD //下拉输入模式 GPIO_MODE_IPU //上拉输入模式 GPIO_MODE_OUT_OD //开漏输出模式 GPIO_MODE_OUT_PP //推挽输出模式 GPIO_MODE_AF_ODAFIO //复用开漏输出模式 GPIO_MODE_AF_PPAFIO //复用推挽输出模式
输入 3：speed	GPIO 输出最大频率 输入值： GPIO_OSPEED_10MHZ //输出最大频率为 10MHz GPIO_OSPEED_2MHZ //输出最大频率为 2MHz GPIO_OSPEED_50MHZ //输出最大频率为 50MHz
输入 4：pin	GPIO 引脚 输入值： GPIO_PIN_x //引脚选择 (x=0…15) GPIO_PIN_ALL //所有引脚

示例代码如下：

```
/* 将 PA0 设置为模拟输入 */
gpio_init(GPIOA, GPIO_MODE_AIN, GPIO_OSPEED_50MHZ, GPIO_PIN_0);
```

2. gpio_deinit()

gpio_deinit()将 GPIO 恢复为默认值的函数，参数见表 3-13。

表 3-13　gpio_deinit()函数参数

参　数	描　述
函数原型	void gpio_deinit(uint32_t gpio_periph)；
功能描述	将外设 GPIOx 寄存器重设为默认值
输入：gpio_periph	GPIO 端口 输入值： GPIOx　//端口选择(x = A,B,C,D,E,F,G)

示例代码如下：

```
/* 将 GPIOA 置为默认值 */
gpio_deinit(GPIOA);
```

3.4.2　输入输出控制函数

1. gpio_bit_set()

gpio_bit_set()将指定的 GPIO 引脚置为高电平状态，函数参数见表 3-14。

表 3-14　gpio_bit_set()函数参数

参　数	描　述
函数原形	void gpio_bit_set(uint32_t gpio_periph, uint32_t pin)；
功能描述	将指定 GPIO 端口的引脚置为高电平状态
输入 1：gpio_periph	GPIO 外设编号 输入值： GPIOx　//端口选择(x = A,B,C,D,E,F,G)
输入 2：pin	GPIO 引脚 输入值： GPIO_PIN_x　　//引脚选择(x = 0…15) GPIO_PIN_ALL　//所有引脚

示例代码如下：

```
/* 将 PA0 引脚置为高电平 */
gpio_bit_set(GPIOA, GPIO_PIN_0);
```

2. gpio_bit_reset()

gpio_bit_reset()将指定的 GPIO 引脚置为低电平状态，函数参数见表 3-15。

表 3-15 gpio_bit_reset ()函数参数

参　数	描　述
函数原形	void gpio_bit_reset(uint32_t gpio_periph, uint32_t gpio_pin);
功能描述	将指定 GPIO 端口的引脚置为低电平状态
输入 1：gpio_periph	GPIO 外设编号 输入值： GPIOx　//端口选择(x = A,B,C,D,E,F,G)
输入 2：pin	GPIO 引脚 输入值： GPIO_PIN_x　　//引脚选择(x = 0…15) GPIO_PIN_ALL　//所有引脚

示例代码如下：

```
/* 将 PA0 引脚置为低电平 */
gpio_bit_reset(GPIOA, GPIO_PIN_0);
```

3. gpio_port_write()

gpio_port_write()一次性指定一组 GPIO 引脚的状态，函数参数见表 3-16。

表 3-16 gpio_port_write()函数参数

参　数	描　述
函数原形	void gpio_port_write(uint32_t gpio_periph, uint16_tdata);
功能描述	该函数的作用是一次性设置指定 GPIO 外设的 16 个引脚的电平状态，以达到快速设置多个引脚的目的。注意，该函数会覆盖之前对这 16 个引脚的任何设置
输入：gpio_periph	GPIO 外设编号 输入值： GPIOx　//端口选择(x = A,B,C,D,E,F,G)

示例代码如下：

```
/* 将"1010 0101"写入 GPIOA */
gpio_port_write (GPIOA, 0xA5);
```

4. gpio_bit_write()

gpio_bit_write()将某个 GPIO 引脚置为输入的电平值，函数参数见表 3-17。

表 3-17 gpio_bit_write()函数参数

参　数	描　述
函数原型	void gpio_bit_write(uint32_t gpio_periph,uint32_t pin, bit_status bit_value);
功能描述	将特定的值写入某个指定引脚
输入 1：gpio_periph	GPIO 外设编号 输入值： GPIOx　//端口选择(x = A,B,C,D,E,F,G)

续表

参　　数	描　　述
输入 2：pin	GPIO 引脚 输入值： GPIO_PIN_x　　//引脚选择(x = 0…15)
输入 3：bit_value	待写入引脚的值 输入值： Bit_RESET　//清除数据端口位(置低电平) Bit_SET　　//设置数据端口位(置高电平)

示例代码如下：

```
/* 将 GPIOA 的第 15 引脚置为高电平 */
gpio_bit_write(GPIOA, GPIO_PIN_15, Bit_SET);
```

3.4.3　状态查询函数

1. gpio_input_bit_get()

gpio_input_bit_get()读取某个 GPIO 引脚的输入电平状态，函数参数见表 3-18。

表 3-18　gpio_input_bit_get()函数参数

参　　数	描　　述
函数原型	FlagStatus gpio_input_bit_get(uint32_t gpio_periph, uint32_t pin);
功能描述	该函数只能用于读取 GPIO 端口的输入电平状态，如果要读取 GPIO 口的输出电平状态，则可使用 gpio_output_bit_get 函数
输入 1：gpio_periph	GPIO 外设编号 输入值： GPIOx　　//端口选择(x = A,B,C,D,E,F,G)
输入 2：pin	GPIO 引脚 输入值： GPIO_PIN_x　　//引脚选择(x = 0…15)
返回	类型：FlagStatus 可能返回值： SET/RESET

示例代码如下：

```
/* 读取 PA0 的状态 */
FlagStatus bit_state;
bit_state = gpio_input_bit_get(GPIOA, GPIO_PIN_0);
```

2. gpio_input_port_get()

gpio_input_port_get()读取某一组 GPIO 所有引脚的输入电平状态，函数参数见表 3-19。

表 3-19 gpio_input_port_get()函数参数

参　数	描　述
函数原型	uint16_t gpio_input_port_get(uint32_t gpio_periph);
功能描述	该函数的返回值为一个 16 位的整数,表示指定 GPIO 端口的 16 个引脚的输入电平状态。该值的每个位代表相应的 GPIO 引脚,0 表示输入低电平,1 表示输入高电平。注意,该函数只能用于读取 GPIO 端口的输入电平状态,如果要读取 GPIO 端口的输出电平状态,则可使用 gpio_output_port_get 函数
输入:gpio_periph	GPIO 外设编号 输入值: GPIOx　　//端口选择(x = A,B,C,D,E,F,G)
返回	类型:uint16_t 可能返回值: 0x00-0xFF

代码如下:

```
/* 读取端口组 GPIOA 的输入值 */
uint16_t port_state;
port_state = gpio_input_port_get(GPIOA);
```

3. gpio_output_bit_get()

gpio_output_bit_get()读取某个 GPIO 引脚的输出电平状态,函数参数见表 3-20。

表 3-20 gpio_output_bit_get ()函数参数

参　数	描　述
函数原形	FlagStatus gpio_output_bit_get(uint32_t gpio_periph, uint32_t pin);
功能描述	该函数只能用于读取 GPIO 端口的输出电平状态,如果要读取 GPIO 端口的输入电平状态,则可使用 gpio_input_bit_get 函数
输入 1:gpio_periph	GPIO 外设编号 输入值: GPIOx　　//端口选择(x = A,B,C,D,E,F,G)
输入 2:pin	GPIO 引脚 输入值: GPIO_PIN_x　　//引脚选择(x = 0…15)
返回	类型:FlagStatus 可能返回值: SET/RESET

示例代码如下:

```
/* 读取引脚 PA0 的输出电平 */
FlagStatus bit_state;
bit_state = gpio_output_bit_get(GPIOA, GPIO_PIN_0);
```

4. gpio_output_port_get()

gpio_output_port_get 读取某组 GPIO 引脚的输出电平状态,函数参数见表 3-21。

表 3-21　gpio_output_port_get()函数参数

参　数	描　述
函数原形	uint16_t gpio_output_port_get(uint32_t gpio_periph);
功能描述	该函数用于读取一组 GPIO 端口的输出电平状态
输入：gpio_periph	GPIO 外设编号 输入值： GPIOx　　//端口选择(x = A,B,C,D,E,F,G)
返回	类型：uint16_t 可能返回值： 0x00-0xFF

示例代码如下：

```
/* 获取 GPIOA 的输出值 */
uint16_t port_state;
port_state = gpio_output_port_get (GPIOA);
```

3.5　GPIO 案例：按键控制 LED 亮灭

36min

3.5.1　案例需求

开发板上的按键 A、B 分别用于控制 LED1、LED2 的亮灭状态翻转，即①当按键 A 被按下时，若 LED1 当前为点亮状态，则熄灭，若 LED1 当前为熄灭状态，则点亮；②当按键 B 被按下时，若 LED2 当前为点亮状态，则熄灭，若 LED2 当前为熄灭状态，则点亮。开发板如图 3-21 所示。

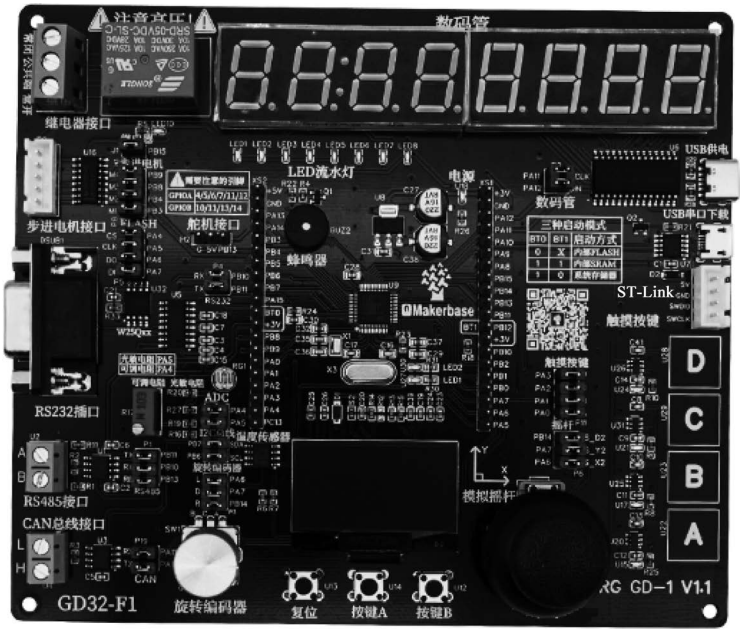

图 3-21　按键 A、B 分别控制 LED1、LED2 状态翻转

3.5.2 案例方法

欲实现案例目标,大致需要经过两个步骤:①先对要用到的I/O端口的工作模式进行初始化,即需要分别将LED1(U27)和LED2(U30)对应的PB0、PB1设置为推挽输出,将按键A、按键B对应的PA0、PA1设置为上拉输入;②进入一个while(TRUE)的循环,在循环中查询按键A、按键B的状态,若按键被按下(对应I/O引脚输入一个低电平),则将该按键控制的LED的亮灭状态翻转。LED1、LED2和按键A、按键B对应的电路原理如图3-22所示。

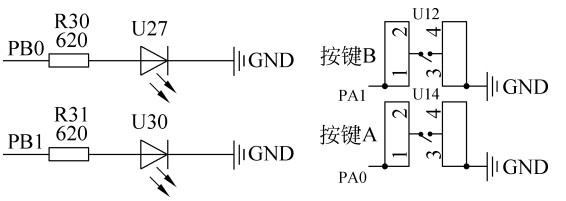

图3-22 LED1、LED2和按键A、按键B对应的电路原理图

整体的流程图如图3-23所示。

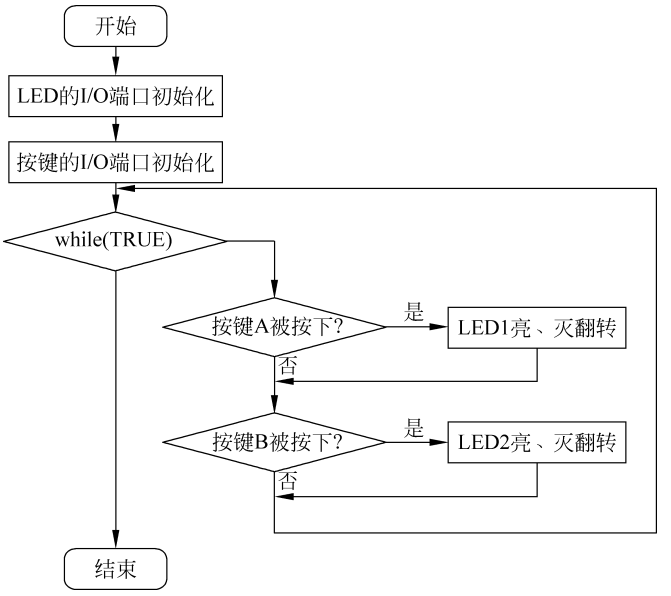

图3-23 按键控制LED状态翻转的流程图

3.5.3 案例代码

为实现案例目标,需要定义三个模块:①LED控制模块,能够初始化LED对应的I/O引脚状态并能够控制LED的亮、灭状态;②按键控制模块,能够初始化按键对应的I/O引脚状态并能够查询按键的状态;③主模块,调用LED模块,按键模块接口函数实现预期的功能。

LED 控制模块由 LED. h、LED. c 两个文件组成，按键控制模块由 KEY. h、KEY. c 两个文件构成，主模块由 main. c 文件构成，整个工程的文件视图如图 3-24 所示。

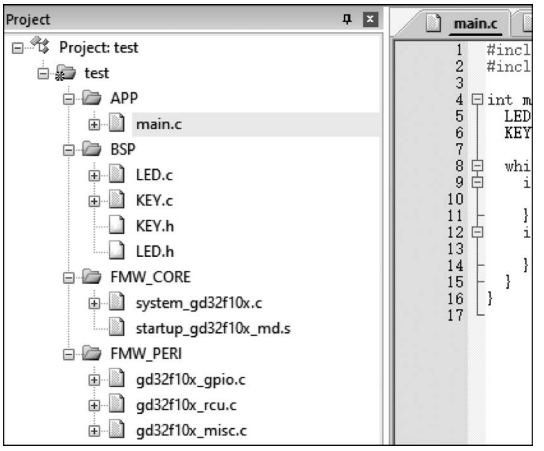

图 3-24　按键控制 LED 工程视图

LED 控制模块，LED. h 文件的代码如下：

```
/*!
    \file    第 3 章\3.5 案例\LED.h
*/

#ifndef _LED_H
#define _LED_H                          //防止头文件被重复包含或定义

#include "gd32f10x.h"
#include <stdio.h>

#define LED1_PIN   GPIO_PIN_0           //LED 对应 I/O 引脚的宏定义
#define LED2_PIN   GPIO_PIN_1
#define LED_PORT   GPIOB

void LED_Init(void);                    //LED 相关的 I/O 端口初始化

void LED1_Toggle(void);                 //LED1 状态翻转
void LED2_Toggle(void);                 //LED2 状态翻转

#endif
```

LED 控制模块，LED. c 文件的代码如下：

```
/*!
    \file    第 3 章\3.5 案例\LED.c
*/
#include "LED.h"
```

```
//LED 对应 I/O 引脚的初始化
void LED_Init(){
    rcu_periph_clock_enable(RCU_GPIOB);

    gpio_init(LED_PORT, GPIO_MODE_OUT_PP, GPIO_OSPEED_50MHZ, LED1_PIN|LED2_PIN);
    gpio_bit_reset(LED_PORT, LED1_PIN|LED2_PIN);   //初始化为熄灭状态
}

//反转 LED1 点亮或熄灭的状态
void LED1_Toggle(void){
    gpio_bit_write(LED_PORT, LED1_PIN, (bit_status)!gpio_output_bit_get(LED_PORT,LED1_
PIN));
}

//反转 LED2 点亮或熄灭的状态
void LED2_Toggle(void){
    gpio_bit_write(LED_PORT, LED2_PIN, (bit_status)!gpio_output_bit_get(LED_PORT,LED2_
PIN));
}
```

按键控制模块,KEY.h 文件的代码如下:

```
/*!
    \file    第 3 章\3.5 案例\KEY.h
*/
#ifndef _KEY_H
#define _KEY_H                //防止重复包含或定义

#include "gd32f10x.h"
#include <stdio.h>

#define KEY_A_PIN  GPIO_PIN_0
#define KEY_B_PIN  GPIO_PIN_1
#define KEY_PORT   GPIOA

void KEY_Init(void);

bool KEY_A_Pressed(void);      //按键 A 是否被按下
bool KEY_B_Pressed(void);      //按键 B 是否被按下

#endif
```

按键控制模块,KEY.c 文件的代码如下:

```
/*!
    \file    第 3 章\3.5 案例\KEY.c
*/
#include "KEY.h"
```

```
void KEY_Init(void){
    rcu_periph_clock_enable(RCU_GPIOA);

    gpio_init(KEY_PORT, GPIO_MODE_IPU, GPIO_OSPEED_50MHZ, KEY_A_PIN|KEY_B_PIN);
}

/*
功能:判断按键 A 是否被按下(加了软件消抖)
返回:如果按键 A 被按下,则返回值为 TRUE,否则返回值为 FALSE
*/
bool KEY_A_Pressed(void){
    if(gpio_input_bit_get(KEY_PORT, KEY_A_PIN) == RESET){
        return TRUE;
    }
    return FALSE;
}

/*
功能:判断按键 B 是否被按下(没加软件消抖)
返回:如果按键 B 被按下,则返回值为 TRUE,否则返回值为 FALSE
*/
bool KEY_B_Pressed(void){
    if(gpio_input_bit_get(KEY_PORT, KEY_B_PIN) == RESET){
        return TRUE;
    }
    return FALSE;
}
```

主模块,main.c 的代码如下:

```
/*!
    \file    第 3 章\3.5 案例\main.c
*/
#include "LED.h"
#include "KEY.h"

int main(){
    LED_Init();
    KEY_Init();

    while(1){
        if(KEY_A_Pressed()){
            LED1_Toggle();
        }
        if(KEY_B_Pressed()){
            LED2_Toggle();
        }
    }
}
```

3.5.4 效果分析

工程编译成功后将得到的可执行文件下载到开发板后，按下按键 A，LED1 被点亮，再次按下按键 A，LED1 熄灭；按下按键 B，LED2 被点亮，再次按下按键 B，LED2 熄灭。

但是，按下按键 A 或按键 B 并不是每次都能很好地控制 LED1 或 LED2 的亮灭状态，有时会出现控制失灵的现象。原因是，作为机械按键，在被按下或弹起时会有抖动而产生纹波，如图 3-25 所示。若要按键较好地控制 LED，需要进行消抖，消抖的方法将在 3.8 节的实验中介绍。

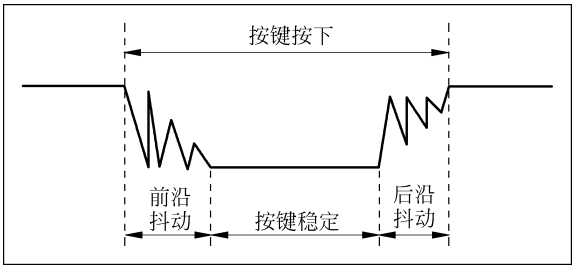

图 3-25 机械按键的抖动

3.6 小结

本章详细介绍了 GPIO 的内部工作原理，分析了 GPIO 的原理框图。分输入、输出两类对 GPIO 的 8 种工作模式的工作原理进行了详细分析。只有掌握了 GPIO 每个工作模式的内部原理才能在实际开发中灵活应用。最后，给出了一个通过按键控制 LED 亮灭的案例工程，可以根据按键按下或弹起的状态来改变 LED 点亮和熄灭的状态，通过这个应用案例应该掌握如何使用 GD 提供的库函数对 GD32 的 GPIO 进行初始化、输入查询、输出控制等操作。

在一般的应用场景中，用户按下按键这个动作并不是很频繁，所以在 main 函数的无限循环中不停地查看 PA0 的状态并不是很必要。第 4 章将介绍 GD32 的中断和事件。

3.7 练习题

(1) 简述 GD32 常见的封装方式。

(2) 在嵌入式系统研发工作中，该如何选择芯片的封装方式？

(3) GPIO 的输入工作模式有哪些？

(4) GPIO 的输出工作模式有哪些？

(5) 简述 GPIO 模拟输入工作模式中信号的流向。

(6) 简述 GPIO 开漏输出模式中信号的流向。

（7）说明 GPIO 使用的流程及其对应步骤的实现方法。

（8）整理 GPIO 各种工作模式的适用场景。

3.8　实验：物理按键软件消抖

3.8.1　实验目标

根据 3.5 节按键控制 LED 亮灭的实验结果，物理按键的前沿抖动、后沿抖动会影响按键的工作效果。本次实验的目标是，通过软件方法避开前沿抖动、后沿抖动阶段，在按键稳定的时间段内控制按键的按下、弹起的状态。

3.8.2　实验方法分析

根据常识，按键被按下的过程一般会持续 100ms 左右，前沿抖动、后沿抖动一般会持续 10ms 左右。以按键 A（接入 PA0 口，并且被设置为上拉输入模式）为例，按键被按下前后的 I/O 端口输入如图 3-26 所示。

由图 3-26 可知，若要避开前沿抖动，则只需在 PA0 刚开始输入为 0 时等待 10ms 左右等按键进入稳定时段再次读取 PA0 的输入，若此时 PA0 输入依然为 0，则表示按键确实被按下且已经进入稳定时段，此后等待按键被弹起（当等到 PA0 的输入再次变为 1 时按键被弹起），可以确认按键被按下了；若不是在前沿抖动时 PA0 检测到一个 0 输入，等待 10ms 后，按键不会进入稳定按下的状态（PA0 的输入会变为 1），表示按键没有被按下。将按键软件消抖的过程整理为流程图，如图 3-27 所示。

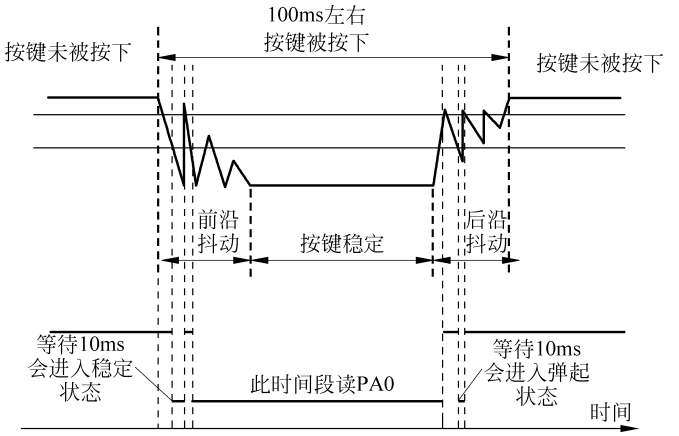

图 3-26　物理按键 A 被按下过程对应 PA0 引脚输入电平分析

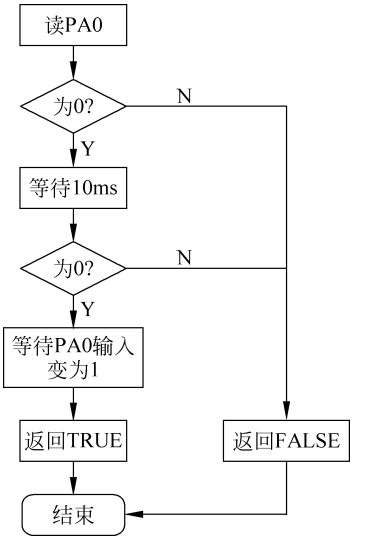

图 3-27　物理按键 A 软件消抖流程图

3.8.3 实验代码

若要实现图 3-27 所示的软件消抖流程,则需要在 3.5 节的案例的基础上再增加一个延时等待的模块以实现 10ms 的等待。因为芯片运行一条指令需要一定时间,所以可以通过运行若干条空指令以实现芯片延时的效果,空指令的运行时长和芯片的时钟频率、指令集等因素相关,针对 GD32F103C8T6 不太精准的毫秒级延时函数的实现代码如下:

```
//ms 延时函数
//输入:time_count,表示要延时 time_count 毫秒
void delay_ms(uint16_t time_count){
    while(time_count--){
        uint16_t i = 20000;
        while(i--){
            ; //空指令
        }
    }
}
```

有了延时函数,相对于 3.5.3 节中的代码,只需将判断按键状态的函数更改为可进行软件消抖的代码便可以达到实验目的。KEY.c 的代码更改如下:

```
/*!
    \file    第 3 章\3.8 实验\KEY.h
*/
#include "KEY.h"
#include "DELAY.h"

void KEY_Init(void){
    rcu_periph_clock_enable(RCU_GPIOA);

    gpio_init(KEY_PORT, GPIO_MODE_IPU, GPIO_OSPEED_50MHZ, KEY_A_PIN|KEY_B_PIN);
}

/*
功能:判断按键 A 是否被按下(加了软件消抖)
返回:如果按键 A 被按下,则返回值为 TRUE,否则返回值为 FALSE
*/
bool KEY_A_Pressed(void){
    if(gpio_input_bit_get(KEY_PORT, KEY_A_PIN) == RESET){
        //软件消抖
        delay_ms(15);
        if(gpio_input_bit_get(KEY_PORT, KEY_A_PIN) == RESET){
            while(gpio_input_bit_get(KEY_PORT, KEY_A_PIN) == RESET);
            return TRUE;
        }
    }
    return FALSE;
```

```
}

/*
功能:判断按键 B 是否被按下(没加软件消抖)
返回:如果按键 B 被按下,则返回值为 TRUE,否则返回值为 FALSE
*/
bool KEY_B_Pressed(void){
    if(gpio_input_bit_get(KEY_PORT, KEY_B_PIN) == RESET){
        //软件消抖
        delay_ms(15);
        if(gpio_input_bit_get(KEY_PORT, KEY_B_PIN) == RESET){
            while(gpio_input_bit_get(KEY_PORT, KEY_B_PIN) == RESET);
            return TRUE;
        }
    }
    return FALSE;
}
```

3.8.4　实验现象

加入软件消抖的代码后,按键控制 LED 的亮灭状态更加精准。