第5章 STM32F407单片机基本原理

5.1 概述

单片机又称为单片微控制器(Microcontroller Unit,MCU),是智能电子系统的核心部件。单片机系统的基本组成如图 5.1.1 所示。随着微电子技术的发展,图 5.1.1 中的大部分功能模块都可以集成在单一的芯片中,成为真正意义的单片机。

图 5.1.1 单片机系统的基本组成

单片机的主要技术指标:①复杂指令集 CISC 还是精简指令集 RISC;②单总线(冯·诺依曼结构,也称为普林斯顿结构)还是多总线(哈佛结构);③数据总线位宽;④寻址空间;⑤最高系统时钟频率;⑥片内外设;⑦I/O 引脚数量。

由于单片机应用范围十分广泛,世界上各大半导体厂商都推出了富有特色的单片机系列,同一系列又包含多个型号。丰富的单片机品种使设计者总能找到最合适的单片机,使得所设计的系统在满足性能的前提下所需扩展的外围器件最少,从而达到小型化、高性价比。从业界使用的广泛性、性能指标、未来发展趋势等角度考虑,本书在电子系统设计中选用了 STM32F4 系列单片机。

STM32F4 系列单片机是意法半导体(STMicroelectronics,ST)公司推出的基于 ARM 内核 Cortex-M4 的 32 位微控制器。Cortex-M4 内核是为低功耗和价格敏感的应用而专门设计的,具有突出的性价比和处理速度。STM32F4 系列单片机又分为 STM32F40x、STM32F41x、STM32F42x 和 STM32F40x 等几个系列,数十个产品型号。不同型号单片机在软件和引脚方面具有良好的兼容性。本书选用的单片机型号为 STM32F40x 系列的 STM32F407VET6(以下简称 STM32F407 单片机),TQFP100 封装,其引脚排列如图 5.1.2 所示。

STM32F407 单片机的简化框图如图 5.1.3 所示,其主要片内资源有:

(1) 采用先进的 Cortex-M4 内核。带 32 位单精度硬件浮点处理单元(Floating Point Unit,FPU)。支持浮点指令集,支持 DSP 指令,可实现高效的信号处理和复杂的算法。

(2) 内含自适应实时存储器加速器(Adaptive Real-Time Memory Accelerator)。通过预取指令和分支缓存,在运行频率达到 168MHz 时,CPU 无须等待闪存,提高了系统的总体速度和能效。

(3) 丰富的片内资源。片内含有 192KB SRAM、512KB Flash ROM、带摄像头接口

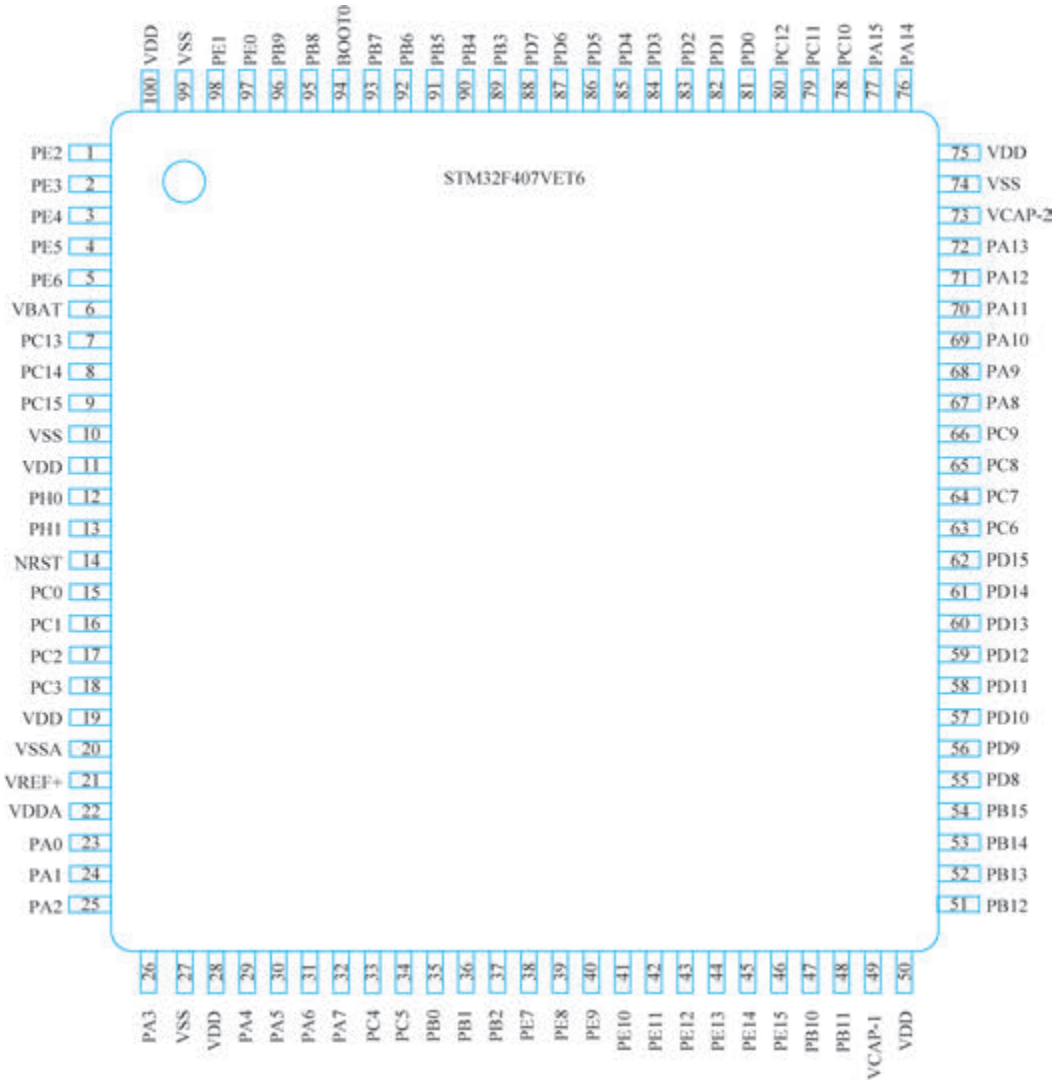


图 5.1.2　STM32F407VET6 单片机的引脚排列

(DCMI)、全速 USB OTG、真随机数发生器 RNG、3 个 12 位 ADC、2 个 12 位 DAC、12 个 16 位定时器、2 个 32 位定时器、DMA、3 个 I^2C、4 个 UART、3 个 SPI、2 个 CAN、SDIO 接口、10/100M Ethernet MAC 等。

(4) 并行总线接口(Flexible Static Memory Controller，FSMC)。

(5) 时钟系统。包括 4～26MHz 外部晶体、16MHz 内部 RC 振荡器(1%精度)、32kHz 内部低频振荡器、32kHz 外部晶体振荡器。

(6) 更低的功耗。功耗为 238μA/MHz。

从图 5.1.3 所示的框图可知，STM32F407 单片机内部有多条总线，不同的外设挂在不同的总线上。在使用片内外设时，需要了解该外设与什么总线相连、总线的带宽等信息。单片机内部的几条主要总线说明如下：

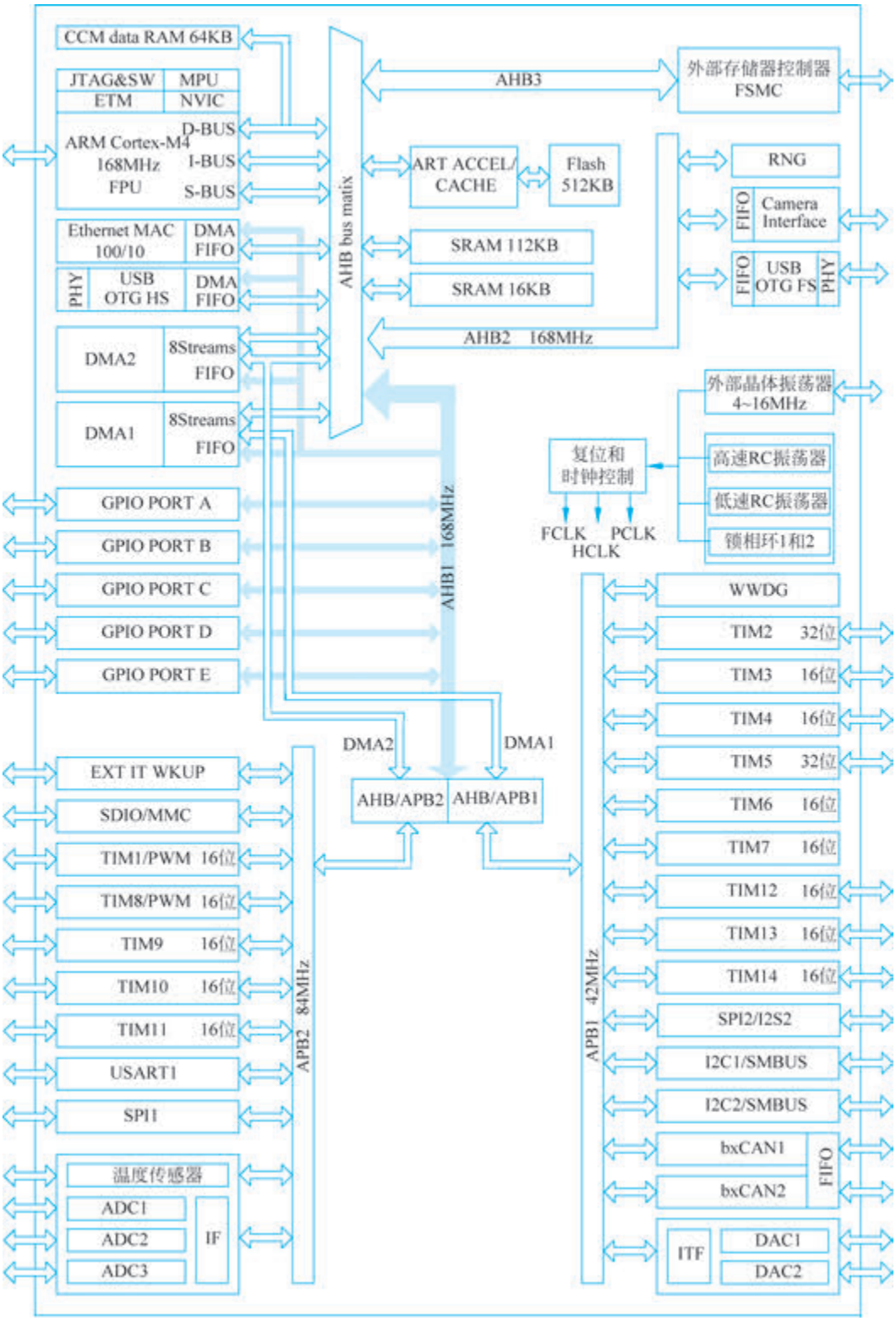


图 5.1.3 STM32F407 单片机的简化框图

（1）AHB1（Advanced High performance Bus）总线，最高时钟频率可达 168MHz。主要用于连接 GPIO 端口以及两个 AHB/APB 桥。其中两个 AHB/APB 桥与两个 DMA 控制器单独开辟了用于 DMA 传输的总线，从而大大减轻了 AHB1 总线的负担。

（2）AHB2 总线。主要用于连接随机数生成器 RNG、摄像头接口和全速 USB-OTG 单元。因为在图像应用中摄像头接口数据量很大，单独开辟总线可以避免和其他设备竞争总线造成系统反应缓慢。

（3）AHB3 总线。只连接了 FSMC 单元。FSMC 单元用于外扩存储器（包括 ROM、SRAM 和 SDRAM 等），FSMC 单元使用独立总线可获得快速的存取响应。

（4）APB1（Advanced Peripheral Bus）总线。最高时钟频率为 42MHz，用于连接 I^2C、SPI2、DAC、定时器 2～7、定时器 12～14 等片内外设。

（5）APB2 总线。最高时钟频率为 84MHz，用于连接 SPI1、USART、ADC、定时器 1、定时器 8～11 等片内外设。

以下为了叙述方便，将单片机内部除 Cortex-M4 内核外的部件均称为片内外设，通过单片机并行总线和串行总线扩展的外部器件称为片外外设。

5.2 时钟系统

单片机时钟如同人体的心脏脉搏，单片机的内核在时钟驱动下完成指令执行，单片机的片内外设在时钟驱动下完成各种工作，时钟系统的重要性可见一斑。与早期的 8051 单片机相比，STM32F407 单片机设计了一个功能完善但却非常复杂的时钟系统，其内部时钟树和时钟源如图 5.2.1 所示。

在 STM32F407 单片机中，有以下 5 个时钟源：

（1）低速内部时钟 LSI，频率约为 32kHz，由 RC 振荡器产生。LSI 用于独立看门狗和自动唤醒单元的时钟源。

（2）低速外部时钟 LSE，频率为 32.768kHz，由石英晶体振荡器产生。LSE 主要用于实时时钟（Real Time Clock，RTC）的时钟源。

（3）高速外部时钟 HSE，频率范围为 4～26MHz，通常由石英晶体振荡器产生，或者直接由外部时钟源提供。HSE 可以直接作为系统时钟或者作为锁相环（Phase Locked Loop，PLL）输入。

（4）高速内部时钟 HSI，频率为 16MHz，由 RC 振荡器产生。经过工厂校准，RC 振荡器的频率精度可以在整个温度范围内达到 1%，HSI 可以直接作为系统时钟或者作为 PLL 输入。

（5）锁相环 PLL。STM32F407 单片机有主 PLL 和专用 PLL 两个锁相环。主 PLL 由 HSE 或 HSI 提供时钟源，产生两个不同的时钟输出。第一个输出 PLLPCLK 用于生成频率高达 168MHz 的系统时钟 SYSCLK；第二个输出 PLLQCLK 用于生成频率为 48MHz 的时钟，该时钟用于 USB OTG、随机数发生器和 SDIO 接口等外设的时钟源。专用 PLL 为 I^2S（Inter-IC Sound，用于音频数据传输的一种总线）接口提供精确时钟，以实现高品质音频性能。

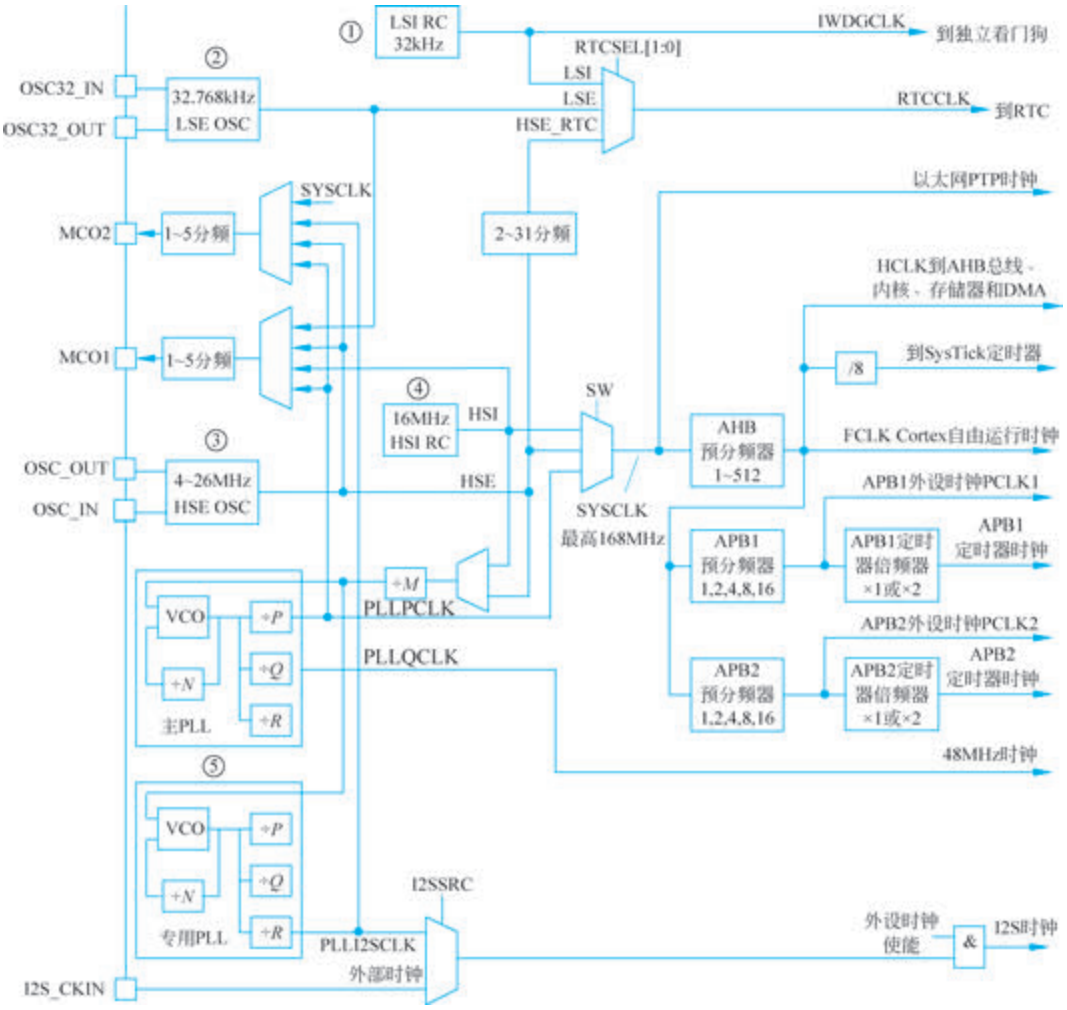
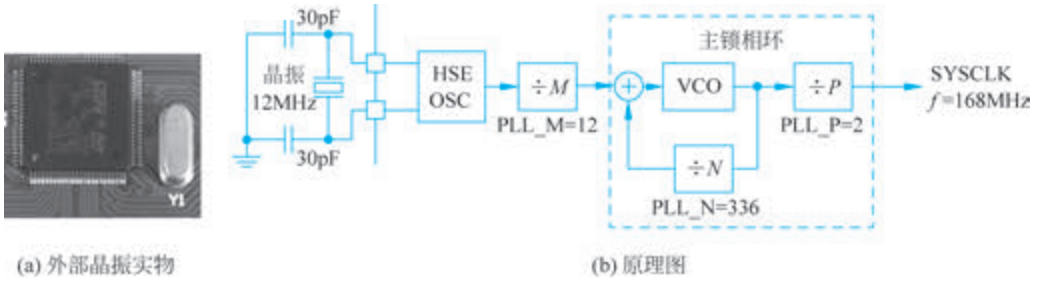


图 5.2.1　STM32F407 单片机的时钟树和时钟源

系统时钟 SYSCLK 是单片机内部最重要的时钟。从图 5.2.1 可以看到，SYSCLK 来自 3 个时钟源 HSI、HSE 和 PLLPCLK 之一，通过一个 3 选 1 的数据选择器来选择其中的一个时钟源。因为 HSI、HSE 的频率比较低，因此在实际应用中通常采用 PLLPCLK 作为 SYSCLK 的时钟源，以获得较高的时钟频率。

假设 HSE 采用外部晶体振荡器产生，将外部晶体振荡器、锁相环、分频电路合在一起，STM32F407 单片机的系统时钟产生原理图如图 5.2.2 所示。单片机有专门的两个引脚与石英晶体连接，石英晶体与单片机内部的振荡电路（HSE OSC）构成晶体振荡器，晶体振荡器产生的 12MHz 时钟信号要经过一个分频系数为 M 的分频器，然后经过倍频系数为 N 的倍频器，再经过一个分频系数为 P 的分频器，产生 PLLPCLK，然后通过 3 选 1 数据选择器（图中省略）选择 PLLPCLK 作为系统时钟 SYSCLK。

根据图 5.2.2 所示的参数，晶体振荡器频率为 12MHz，锁相环预分频系数 M 为 12，



(a) 外部晶振实物　　(b) 原理图

图 5.2.2　STM32F407 单片机系统时钟产生原理图

锁相环倍频系数 N 为 336，锁相环分频系数 P 为 2，那么系统时钟 SYSCLK 的频率为

$$\text{SYSCLK 的频率} = 12\text{MHz} \times N/(M \times P) = 12 \times 336/(12 \times 2) = 168(\text{MHz}) \tag{5.2.1}$$

图 5.2.2 中分频系数和倍频系数在 system_stm32f4××.c 文件中配置，如图 5.2.3 所示。

```
LCD.h | FONT.h | main.c | WAVEDAT.h | LCD.c | system_stm32f4xx.c | stm32f4xx.h | stm32f4xx_it.c
                                  This value must be a multiple of 0x200. */
/******************************************************************************/

/************************ PLL Parameters *************************************/
#if defined (STM32F40_41xxx) || defined (STM32F427_437xx) || defined (STM32F429_439xx) || 
/* PLL_VCO = (HSE_VALUE or HSI_VALUE / PLL_M) * PLL_N */
#define PLL_M      12
#else /* STM32F411xE */
#if defined (USE_HSE_BYPASS)
#define PLL_M      8
#else /* STM32F411xE */
#define PLL_M      16
#endif /* USE_HSE_BYPASS */
#endif /* STM32F40_41xxx || STM32F427_437xx || STM32F429_439xx || STM32F401xx */

/* USB OTG FS, SDIO and RNG Clock =  PLL_VCO / PLLQ */
#define PLL_Q      7

#if defined (STM32F40_41xxx)
#define PLL_N      336
/* SYSCLK = PLL_VCO / PLL_P */
#define PLL_P      2
#endif /* STM32F40_41xxx */
```

图 5.2.3　分频系数和倍频系数的设置

STM32F407 单片机允许的最高系统时钟频率为 168MHz，如果要降低系统时钟频率，可以直接在 system_stm32f4××.c 源代码中修改分频系数或者倍频系数来实现。

另外，石英晶体的频率也需要在 stm32f4××.h 中设置，如图 5.2.4 所示。

```
main.c | stm32f4xx_adc.h | stm32f4xx_conf.h | LCD.h | stm32f4xx.h
#if !defined  (HSE_VALUE)
  #define HSE_VALUE    ((uint32_t)12000000) /*!< Value of the External oscillator in Hz */

#endif /* HSE_VALUE */
```

图 5.2.4　设置外部石英晶体频率

STM32F407 单片机的时钟系统初始化是在 system_stm32f4××.c 中的 SystemInit()函数中完成的。SystemInit()函数的主要功能是启动 HSI 时钟、选择 HSI 作为系统时钟、调用 SetSysClock()函数来完成系统时钟关键寄存器的设置，相关代码如下。

```
void SystemInit(void)
{
  …
  RCC->CR |= (uint32_t)0x00000001;          //HSION 位置 1
  RCC->CFGR = 0x00000000;                   //复位 CFGR 寄存器
  RCC->CR &= (uint32_t)0xFEF6FFFF;          //复位 HSEON、CSSON 和 PLLON 位
  RCC->PLLCFGR = 0x24003010;                //复位 PLLCFGR 寄存器
  RCC->CR &= (uint32_t)0xFFFBFFFF;          //复位 HSEBYP 位
  RCC->CIR = 0x00000000;                    //禁止所有中断
  …
  SetSysClock();                            //调用 SetSysClock()函数
  …
}
```

SetSysClock()函数的主要功能是使能 HSE 振荡器，等待 HSE 就绪，配置 AHB、APB1、APB2 时钟相关的分频因子，打开主 PLL 时钟，然后设置主 PLL 作为系统时钟 SYSCLK 时钟源。SetSysClock()函数相关代码如下。

```
static void SetSysClock(void)
{
  __IO uint32_t StartUpCounter = 0, HSEStatus = 0;
  RCC->CR |= ((uint32_t)RCC_CR_HSEON);      //使能 HSE
  do                                        //等待 HSE 工作稳定，如超时则退出
  {
    HSEStatus = RCC->CR & RCC_CR_HSERDY;
    StartUpCounter++;
  } while((HSEStatus == 0) && (StartUpCounter != HSE_STARTUP_TIMEOUT));
  if ((RCC->CR & RCC_CR_HSERDY) != RESET)
  {
    HSEStatus = (uint32_t)0x01;             //HSE 工作稳定
  }
  else
  {
    HSEStatus = (uint32_t)0x00;             //HSE 未达到工作稳定
  }
  if (HSEStatus == (uint32_t)0x01)          //注 1
  {
    RCC->APB1ENR |= RCC_APB1ENR_PWREN;
    PWR->CR |= PWR_CR_VOS;
    RCC->CFGR |= RCC_CFGR_HPRE_DIV1;        //HCLK = SYSCLK / 1
    RCC->CFGR |= RCC_CFGR_PPRE2_DIV2;       //PCLK2 = HCLK / 2
    RCC->CFGR |= RCC_CFGR_PPRE1_DIV4;       //PCLK1 = HCLK / 4
    RCC->PLLCFGR = PLL_M | (PLL_N << 6) | (((PLL_P >> 1) - 1) << 16) |
        (RCC_PLLCFGR_PLLSRC_HSE) | (PLL_Q << 24);
    RCC->CR |= RCC_CR_PLLON;                //使能主 PLL
    while((RCC->CR & RCC_CR_PLLRDY) == 0)   //等待主 PLL 工作稳定
  …
  }
}
```

从时钟系统的初始化程序 SystemInit()可知，由于产生 HSI 时钟为 RC 振荡器，起振较快，所以在单片机刚上电时，默认使用 HSI。HSE 由于采用晶体振荡器，需要一定的时间才能达到稳定状态，单片机通过软件检测到 HSE 时钟稳定后就切换到 HSE。如果 HSE 在一定的时间内不能达到稳定状态（如外部晶振不能稳定或者没有外部晶振），那么仍然将 HSI 作为系统时钟。

当时钟切换到 HSE 后，SetSysClock()就执行注 1 所示的这段代码。这一段代码描述了如何从 SYSCLK 分频得到 AHB、APB2 和 APB1 的总线时钟，可以用图 5.2.5 所示的示意图来说明。

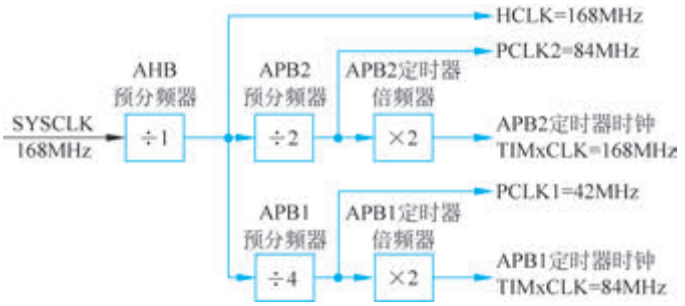


图 5.2.5　总线时钟的产生

从图 5.2.5 中可知，AHB 总线时钟 HCLK 频率为 168MHz；APB1 总线时钟 PCLK1 频率为 42MHz；APB2 总线时钟 PCLK2 频率为 84MHz；APB1 定时器时钟频率为 84MHz；APB2 定时器时钟频率为 168MHz。图 5.2.5 中有两个×2 的倍频器，其输出是专门给定时器提供时钟的。这两个倍频器只有当前面预分频器分频系数不为 1 时才起作用。设置这个倍频器的目的是，在其他外设使用较低时钟频率时，定时器仍能得到较高的时钟频率。


视频

STM32F407 单片机的所有片内外设都需要工作时钟，因此，编程时需要了解片内外设使用什么时钟以及时钟的频率。

5.3　通用输入输出端口

1. GPIO 的基本结构

GPIO(General-Purpose I/O Port)是通用输入输出端口的简称。STM32F407 单片机通过 GPIO 引脚与外部设备连接起来，从而实现与外部通信、控制以及数据采集功能。

从图 5.1.2 可知，STM32F407VET6 单片机有 GPIOA、GPIOB、GPIOC、GPIOD、GPIOE 5 个 16 位通用 I/O 端口。每位 I/O 端口基本结构如图 5.3.1 所示。每位 I/O 端口含有两只保护二极管以及可选择的上拉电阻和下拉电阻。图中上半部分为输入通道，当 I/O 引脚用作数字输入引脚时，数字信号经过施密特触发器后存储在输入数据寄存器。施密特触发器用于输入信号的整形和抗干扰。当 I/O 引脚用作模拟输入引脚时，上拉电阻和下拉电阻断开，施密特触发器关闭，模拟信号直接送到片内 ADC。图中下半部分为输出通道。输出通道中包含了由 NMOS 管和 PMOS 管构成的单元电路。通过输出

控制，可以将 I/O 引脚设置成推拉输出、漏极开路输出和高阻输出。当处于推拉式输出模式时，两只管子轮流导通。当处于漏极开路输出模式时，PMOS 管始终处于截止状态。当处于高阻输出时，NMOS 管和 PMOS 管都截止。在输出通道中，有两只寄存器：置位/复位寄存器、输出数据寄存器。使用置位/复位寄存器可以方便快速地实现对端口某些特定位的操作，而不影响其他位的状态。

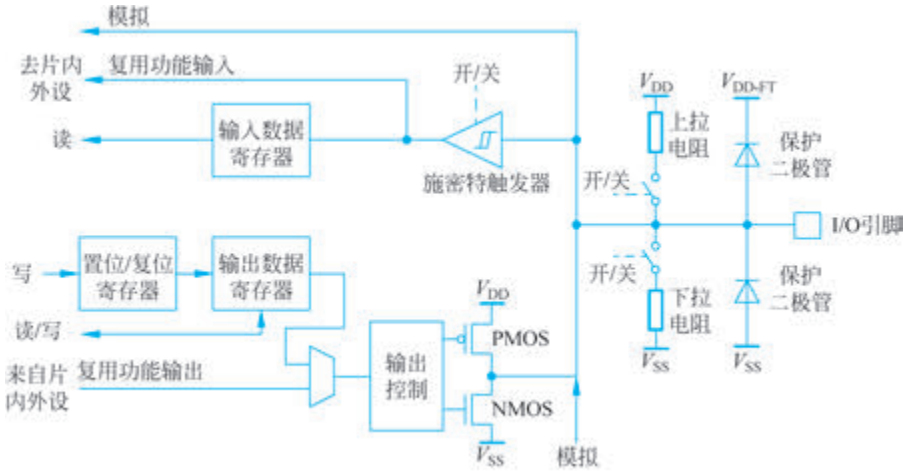


图 5.3.1　I/O 端口基本结构

STM32F407 单片机有很多片内外设，这些外设的功能引脚都是与 GPIO 复用的。当这个 GPIO 给片内外设使用时，就叫作复用(Alternate Functions)。复用器实际上是一个 16 选 1 数据选择器，其示意图如图 5.3.2 所示。任何时刻只允许一个外设连接到对应的 I/O 引脚，以确保共用同一个 I/O 引脚的外设之间不会发生冲突。通过 GPIOx_AFRL 和 GPIOx_AFRH 寄存器的配置，选择其中一个外设连接到对应的 I/O 引脚。

图 5.3.2 所示的复用器不能理解为所有的外设可以与任何一个 I/O 引脚相连。例如，单片机的 MCO1、USART1、I2C3、OTG、EVENTOUT 等外设只能与 PA8 引脚相连。某片内外设究竟能与哪些 I/O 引脚相连，应查阅 STM32F407 单片机的数据手册。

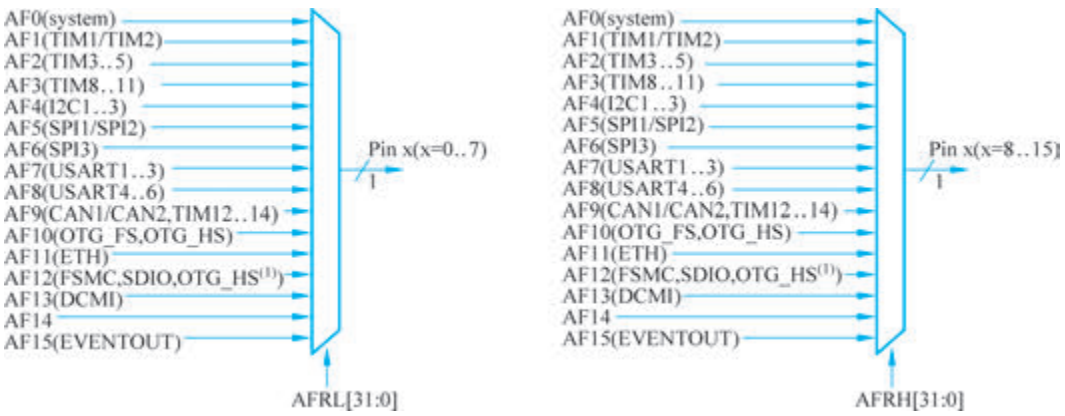


图 5.3.2　复用器示意图

2. GPIO 的初始化

在使用 GPIO 之前，首先应该对 GPIO 进行初始化。对 GPIO 的初始化有两种方法：一种是寄存器配置，另一种是库函数配置。早期的单片机通常直接操作寄存器来进行外设的初始化，这种方法需要掌握每个寄存器的用法。对 STM32F407 单片机来说，理解数百个寄存器谈何容易，于是单片机生产厂家推出了官方固件库（也称为库函数），将寄存器的底层操作都封装起来，开发者一般不需要知道操作的是哪个寄存器，只需要调用哪些函数就可以。GPIO 的初始化就是采用以下库函数配置。

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

GPIO 初始化结构体为

```
typedef struct
{
    uint32_t   GPIO_Pin;
    GPIOMode_TypeDef   GPIO_Mode;
    GPIOSpeed_TypeDef   GPIO_Speed;
    GPIOOType_TypeDef   GPIO_OType;
    GPIOPuPd_TypeDef   GPIO_PuPd;
}GPIO_InitTypeDef;
```

GPIO 初始化结构体中各参数的含义如表 5.3.1 所示。

表 5.3.1　GPIO 初始化结构体中各参数含义

控制寄存器	参　数	说　明
GPIO_Mode	GPIO_Mode_IN	输入模式
	GPIO_Mode_OUT	输出模式
	GPIO_Mode_AF	复用模式
	GPIO_Mode_AN	模拟模式
GPIO_OType	GPIO_OType_PP	推拉式输出
	GPIO_OType_OD	OD 输出
GPIO_PuPd	GPIO_PuPd_NOPULL	无上拉下拉
	GPIO_PuPd_UP	上拉
	GPIO_PuPd_DOWN	下拉
GPIO_Speed	GPIO_Speed_2MHz	低速
	GPIO_Speed_25MHz	中速
	GPIO_Speed_50MHz	快速
	GPIO_Speed_100MHz	高速

从表 5.3.1 可知，GPIO 有多种工作模式，应根据具体应用来选择。例如，I/O 引脚用于产生方波，则应选择推拉式输出模式。I/O 引脚用于按键输入或者外部中断输入，则应设成输入模式，同时，为了避免引脚悬空，可以选择内部上拉电阻。如果 I/O 引脚用于 ADC 的输入，则应设成模拟输入。

GPIO 的引脚速度也要跟应用相匹配。速度配置越高，噪声越大，功耗越高。使用合适的引脚速度可以降低功耗和噪声。比如 USART 串口，若最大波特率只需 115.2kb/s，

那用 2MHz 的速度就够了；对于 I^2C 接口，若使用 400kHz 的时钟，可以选用 25MHz 速度；对于 SPI 接口，若使用 10MHz 以上的时钟速率，需要选用 50MHz 速度。

例 5.3.1 MCP4802 是一片双路电压输出 8 位串行 D/A 转换器，与单片机连接如图 5.3.3(a)所示。图 5.3.3(b)为 MCP4802 的时序图，每次传送 16 位数据，在时钟信号的上升沿将数据送入 MCP4802。请编写程序实现图 5.3.3(b)的时序。

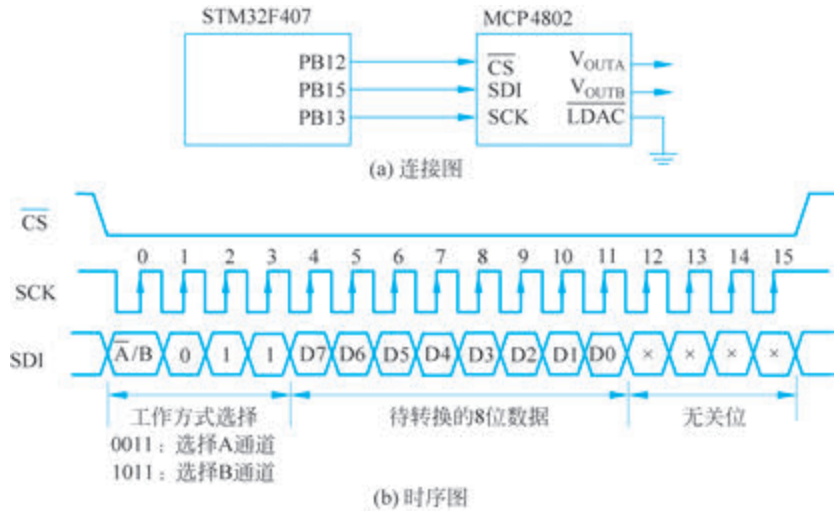


图 5.3.3 MCP4802 的连接图和时序图

解：(1) 主程序

```
#define SPI_SCK_LOW() GPIO_ResetBits(GPIOB, GPIO_Pin_13)      //时钟信号置低
#define SPI_SCK_HIGH() GPIO_SetBits(GPIOB, GPIO_Pin_13)       //时钟信号置高
#define SPI_MOSI_LOW() GPIO_ResetBits(GPIOB, GPIO_Pin_15)     //输出数据线置低
#define SPI_MOSI_HIGH() GPIO_SetBits(GPIOB, GPIO_Pin_15)      //输出数据线置高
#define SPI_CS_LOW() GPIO_ResetBits(GPIOB, GPIO_Pin_12)       //片选信号置低
#define SPI_CS_HIGH() GPIO_SetBits(GPIOB, GPIO_Pin_12)        //片选信号置高
GPIO_InitTypeDef GPIO_InitStructure;
u16 dacdat;
u8 AV;
main(void)
{
    …
    dacdat = AV;                                              //待转换的 8 位数据
    dacdat = (dacdat << 4)|0x3000;                            //把最高位置 0,选择 A 通道
    Write_MCP4802(dacdat);
    …
}
```

(2) 写串行 D/A 子程序

```
void Write_MCP4802(u16 dat)
{
    u8 i;
```

```
    SPI_CS_LOW();                                          //将片选信号置成低电平
    for(i = 0;i < 16;i++)                                  //向 DAC 写入 16 位数据
    {
       SPI_SCK_LOW();                                      //时钟信号置低电平
      if ((dat&0x8000) == 0x8000)
      {
        SPI_MOSI_HIGH();                                   //数据线置高电平
      }
      else
      {
        SPI_MOSI_LOW();                                    //数据线置低电平
      }
       dat << = 1;
       SPI_SCK_HIGH();                                     //产生时钟信号上升沿
     }
      SPI_CS_HIGH();                                       //片选信号恢复成高电平
}
```

（3）GPIO 初始化程序

```
void GPIO_Configuration(void)
{
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE); //使能 GPIOB 时钟
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_12 | GPIO_Pin_13 | GPIO_Pin_15;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
}
```

本例通过软件的方法在 I/O 引脚输出时序信号，这是单片机学习中必须掌握的技能。

例 5.3.2　如何将系统时钟从 I/O 引脚输出？

解：MCO（Microcontroller Clock Output）时钟输出功能是 STM32F407 单片机一项非常实用的功能，它可以将内部时钟信号输出到外部引脚，为外部设备提供时钟。从图 5.2.1 可知，单片机内部时钟信号可以通过 MCO 引脚输出，以便为电子系统中的其他芯片提供精确的时钟信号。根据 STM32F407 单片机的数据手册，MCO1 从单片机的 PA8 引脚输出。相关的程序代码如下。

1）初始化 PA8

```
void GPIO_Configuration(void)
{
      GPIO_PinAFConfig(GPIOA,GPIO_PinSource8,GPIO_AF_MCO);
      GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;         //选择复用模式
      GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
      GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;     //无上拉下拉
      GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
      GPIO_InitStructure.GPIO_Pin = GPIO_Pin_8;
      GPIO_Init(GPIOA, &GPIO_InitStructure);
}
```

在 PA8 的初始化程序中，调用了函数 GPIO_PinAFConfig，该函数入口第 1、第 2 个参数用于确定是哪个 I/O 端口，对于第 3 个参数选择哪个复用外设。单片机的复用外设在函数 stm32f4××_gpio.h 中非常详细地列出来了，如图 5.3.4 所示。

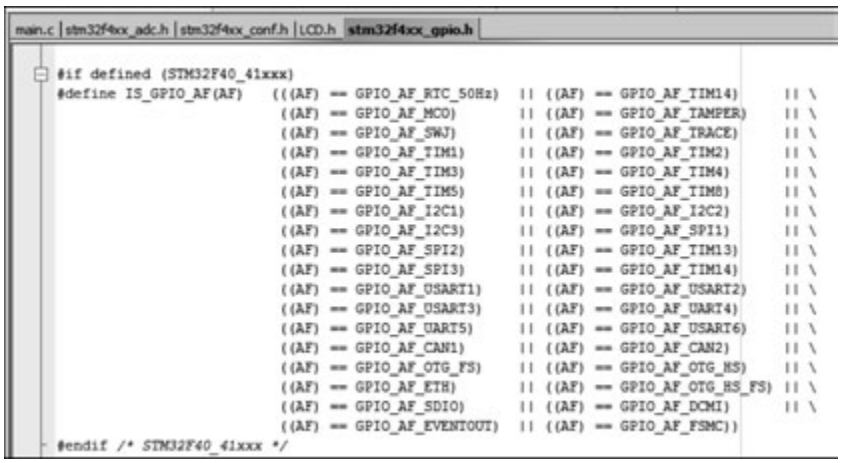


图 5.3.4　复用外设预定义

2）选择 MCO 时钟源

```
RCC_MCO1Config(RCC_MCO1Source_HSE,RCC_MCO1Div_4);
```

该函数表示 MCO1 选择高速外部时钟 HSE，并对 HSE 进行 4 分频。假设高速外部时钟采用 12MHz 晶振，则 PA8 输出 3MHz 时钟信号。

例 5.3.3　通过按键控制信号灯示意图如图 5.3.5 所示。要求每按一次键，信号灯状态改变一次。所谓状态改变，就是由点亮变为熄灭，或者由熄灭到点亮。请编写相关程序。

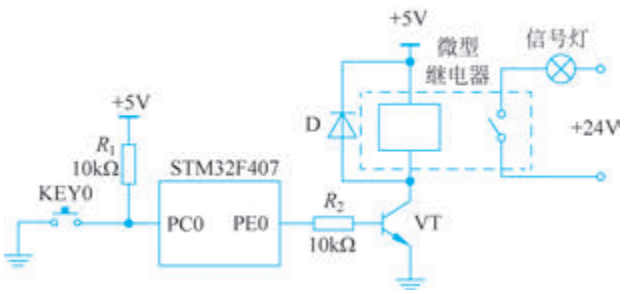


图 5.3.5　按键控制信号灯示意图

解：独立式键盘的按键识别通过软件实现。按键识别首先判断是否有键按下，由于按键闭合时机械抖动，还需要消抖处理。按键的检测和消抖可以用图 5.3.6 所示的时序图来说明。如果检测到低电平，单片机软件延时 10ms 再检测一次按键电平，如果还是低电平，说明按键已经稳定闭合，置键有效标志。因为按键的闭合时间通常大于 10ms，为了避免按键重复执行，在程序中设置了 key_up 标志。一旦检测到键有效，就将 key_up 标志置 0，直到按键释放后，将 key_up 标志置 1，为下一次检测按键做好准备。

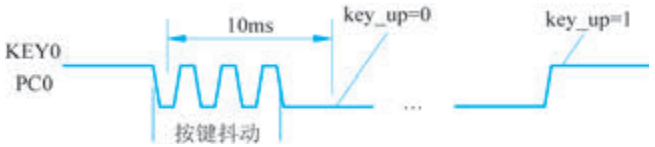


图 5.3.6　按键读取时序

1）主程序

```
#define KEY0 GPIO_ReadInputDataBit(GPIOC, GPIO_Pin_0) //读 PC0 引脚电平
u8 key_up,keysign, posbit;                            //定义 3 个 8 位无符号变量
main(void)
{
    …
    GPIO_Configuration();                             //对 PC0、PE0 两个 I/O 端口初始化
    while (1)
    {
        if (key_up&&key0 == 0)                        //如果 key_up = 1 而且按键闭合
        {
            delay_ms(10);                             //延时 10ms
            key_up = 0;
            if(key0 == 0) keysign = 1;                //置键有效标志
        }
        else if (key0 == 1) key_up = 1;
        if (keysign == 1)
        {
            keysign = 0;                              //清键有效标志
            PE0Tog();                                 //通过 PE0 改变信号状态
        }
    }
    …
}
```

2）GPIO 的初始化程序

```
void GPIO_Configuration(void)
{
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOC, ENABLE);   //使能 GPIOC 时钟
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOE, ENABLE);   //使能 GPIOE 时钟
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_OUT;           //输出模式
    GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;          //推拉输出
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL ;       //无上拉下拉
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;               //初始化 PE0
    GPIO_Init(GPIOE, &GPIO_InitStructure);
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;            //输入模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;            //注 1
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_Init(GPIOC, &GPIO_InitStructure);                  //初始化 PC0
}
```

注 1：这条语句启用 PC0 口的内部上拉电阻，图 5.3.5 中的上拉电阻 R_1 实际上可以省略。

3）PE0 取反子程序

```
void PE0Tog(void)
{
    posbit = ~posbit;                          //posbit 在 0x00 和 0xFF 之间切换
    if (posbit == 0xFF)
    {
      GPIO_SetBits(GPIOE, GPIO_Pin_0);         //将 PE0 置高电平
    }
    else
    {
      GPIO_ResetBits(GPIOE, GPIO_Pin_0);       //将 PE0 置低电平
    }
}
```

4）软件延时程序

```
void delay_ms(volatile u16 time)
{
    volatile u16 i = 0;
    while (time--)
    {
      i = 18660;                 //通过改变 i 的值来调节软件延时
    while (i--);
    }
}
```


视频

5.4 定时器

1. STM32M407 单片机内部定时器

STM32F407 单片机总共有 14 个定时器之多，分为高级定时器、通用定时器和基本定时器 3 类，具体如表 5.4.1 所示。高级定时器、通用定时器和基本定时器形成了上下级的关系。通用定时器包含了基本定时器的所有功能，同时增加了向下、向上/向下计数器、PWM 生成、输出比较、输入捕获等功能；而高级定时器又包含了通用定时器的所有功能外，还增加了死区互补输出、刹车信号、加入重复计数器等功能。

表 5.4.1 各个定时器特性

定时器类型	名称	计数器位数/位	计数器类型	预分频系数	DMA 请求生成	捕获/比较通道	互补输出	最高定时器时钟频率/MHz
高级	TIM1 TIM8	16	递增、递减、递增/递减	1～65636	有	4	有	168
通用	TIM2 TIM5	32	递增、递减、递增/递减	1～65636	有	4	无	84
	TIM3 TIM4	16	递增、递减、递增/递减	1～65636	有	4	无	84
	TIM9	16	递增	1～65636	无	2	无	168
	TIM10 TIM11	16	递增	1～65636	无	1	无	168

续表

定时器类型	名称	计数器位数/位	计数器类型	预分频系数	DMA 请求生成	捕获/比较通道	互补输出	最高定时器时钟频率/MHz
通用	TIM12	16	递增	1～65636	无	2	无	84
	TIM13 TIM14	16	递增	1～65636	无	1	无	84
基本	TIM6 TIM7	16	递增	1～65636	有	0	无	84

2. 基本定时器

基本定时器功能少，结构简单，是理解通用寄存器和高级寄存器的基础。基本定时器主要用于定时，生成时基或触发数模转换器。

基本定时器的原理框图如图 5.4.1 所示。基本定时器的计数过程主要涉及 3 个 16 位寄存器，分别是计数器寄存器 TIMx_CNT、预分频寄存器 TIMx_PSC、自动重载寄存器 TIMx_ARR。基本定时器时钟 TIMxCLK 来自如图 5.2.1 所示的单片机时钟系统。由于基本定时器挂在 APB1 总线上，因此，TIMxCLK 的最高时钟频率为 84MHz。

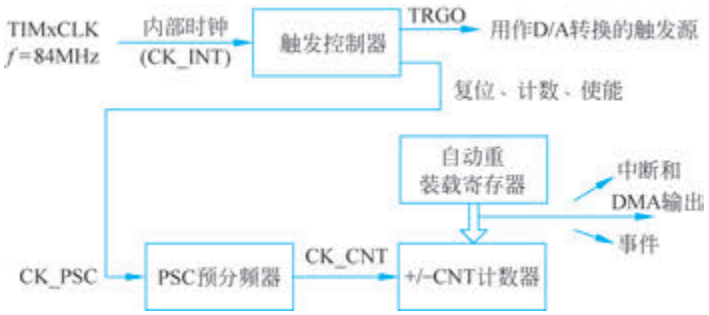


图 5.4.1　基本定时器的原理框图

TIMxCLK 经过预分频寄存器 TIMx_PSC 分频后得到计数器时钟 CK_CNT，其频率由下式确定

$$CK\_CNT = TIMx\_CLK/(PSC+1)$$

式中，PSC 就是存放在 TIMx_PSC 中的值，范围为 0～65535。通过设置 PSC 的值可以得到不同频率的 CK_CNT。

自动重装载寄存器 TIMx_ARR 用来存放与计数器值比较的数值，范围为 1～65535。定时器开始计数时，每来一个 CK_CNT 脉冲，计数器 TIMx_CNT 值加 1。当 TIMx_CNT 值与 TIMx_ARR 的设定值相等时就自动生成事件，同时 TIMx_CNT 清零，然后重新开始计数。由此可见，只要设置 TIMx_PSC 和 TIMx_ARR 两个寄存器的值，就可以控制生成事件的间隔时间。定时器在生成事件的同时，产生中断和 DMA 输出。定时时间可以用下式计算

$$T = \frac{(PSC+1)\times(ARR+1)}{TIMxCLK(MHz)}(\mu s) \tag{5.4.1}$$

定时器的初始化结构体为

```
typedef struct
{
  uint16_t TIM_Prescaler;
  uint16_t TIM_CounterMode;
  uint32_t TIM_Period;
  uint16_t TIM_ClockDivision;
  uint8_t TIM_RepetitionCounter;
} TIM_TimeBaseInitTypeDef;
```

TIM_Prescaler：定时器预分频器设置。它设定 TIMx_PSC 寄存器的值。

TIM_CounterMode：定时器计数模式。分别为向上计数、向下计数和中央对齐模式。向上计数即 TIMx_CNT 从 0 向上累加到重载寄存器 TIMx_ARR 的值，产生上溢事件。向下计数即 TIMx_CNT 从 TIMx_ARR 的值累减至 0，产生下溢事件。中央对齐模式为向上计数模式和向下计数模式的结合体，TIMx_CNT 先从 0 向上累加到重载寄存器 TIMx_ARR 的值减 1 时，产生一个上溢事件，然后向下计数到 1 时，产生一个下溢事件，再从 0 开始重新计数。

TIM_Period：定时器周期，实质是存储到重载寄存器 TIMx_ARR 的值。

TIM_ClockDivision：时钟分频因子。该参数只对计数器使用外部时钟源时才有影响。基本定时器只采用内部时钟源，因此该参数不需设置。

TIM_RepetitionCounter：重复计数器，属于高级定时器的专用寄存器。

例 5.4.1 利用基本定时器 TIM7 中断在 PE0 产生 50Hz 的方波。

解：要在 PE0 产生 50Hz 的方波，TIM7 的定时时间常数应设为 10ms。TIM7 的时钟频率为 84MHz。将 TIM7 的预分频器设置为 83，定时器周期设为 9999，则根据式(5.4.1)定时时间为

$$T=(9999+1)\times(83+1)/84=10000(\mu s)=10(ms)$$

1) TIM7 初始化程序

```
void TIM7_init(void)
{
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM7, ENABLE);   //使能 TIM7 时钟
  TIM_TimeBaseStructInit(&TIM_TimeBaseStructure);
  TIM_TimeBaseStructure.TIM_Period = 9999;               //设置自动重装载寄存器的值
  TIM_TimeBaseStructure.TIM_Prescaler = 83;              //设置预分频值
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //设为向上计数
  TIM_TimeBaseInit(TIM7, &TIM_TimeBaseStructure);        //初始化 TIM7
  TIM_Cmd(TIM7, ENABLE);                                 //使能 TIM7 计数器
}
```

2) TIM7 中断初始化程序

```
void TIM7INT_init(void)
{
  NVIC_InitStructure.NVIC_IRQChannel = TIM7_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 1;
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 1;
```

```
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
  NVIC_Init(&NVIC_InitStructure);
  TIM_ITConfig(TIM7,TIM_IT_Update,ENABLE);                    //允许溢出中断
}
```

3）TIM7 中断服务程序

```
void TIM7_IRQHandler(void)
{
   if(TIM_GetITStatus(TIM7,TIM_IT_Update)!= RESET)
   {
    TIM_ClearITPendingBit(TIM7,TIM_IT_Update);                //清中断标志
    PE0Tog();                                                 //PE0 口取反程序,参见例 5.3.3
   }
}
```

3. 通用定时器

通用定时器的原理框图如图 5.4.2 所示。与基本定时器相比，通用定时器增加了以下功能。

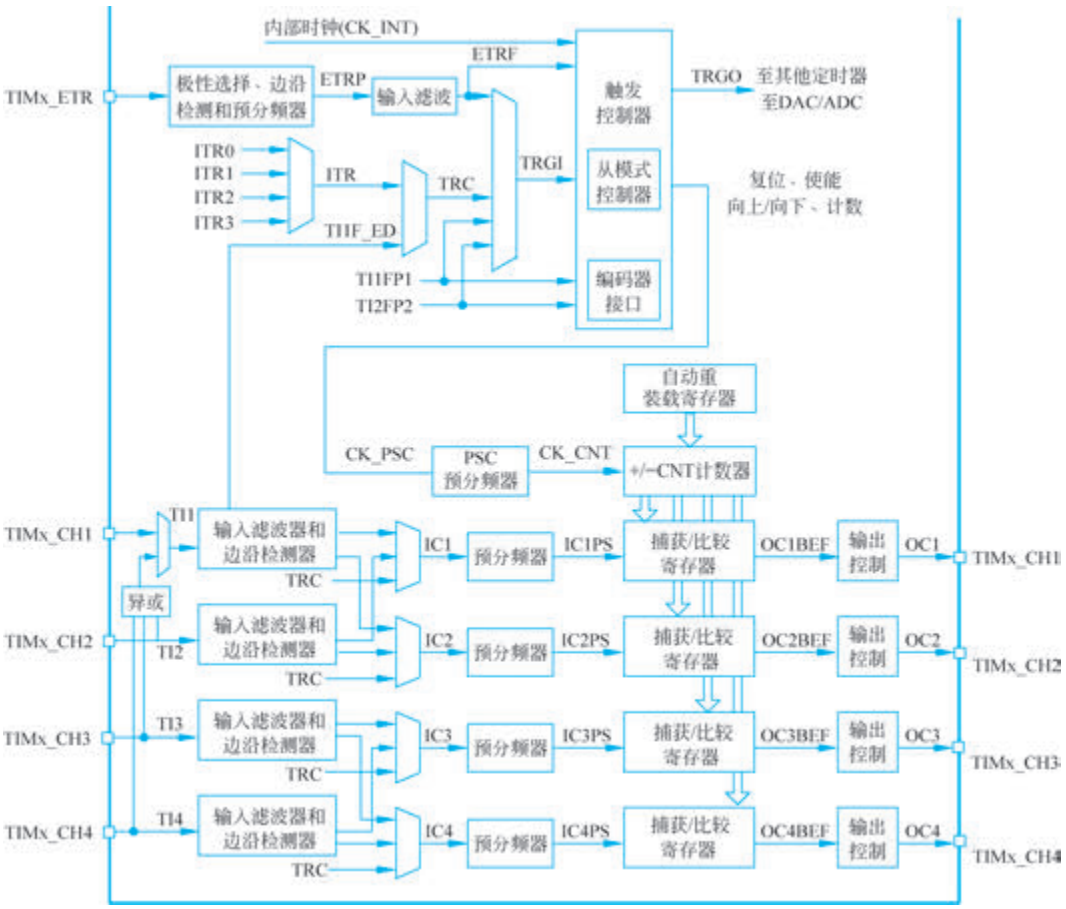


图 5.4.2　通用定时器原理框图

（1）通用定时器有多个时钟源：内部时钟源 CK_INT，该时钟源与基本定时器相同；外部输入引脚 TIx，就是图 5.4.2 中的 TI1～TI4，TIx 引脚的上升沿或者下降沿可以产生计数时钟；外部触发输入 TIMx_ETR，通过极性选择、边沿检测和预分频后可以作为时钟；内部触发输入 ITRx，就是图 5.4.2 中的 ITR0、ITR1、ITR2、ITR3。利用该时钟源可以实现一个定时器作为另一个定时器的预分频器，从而大大延长定时时间。

（2）增加了捕获/比较寄存器 TIMx_CCR。在脉冲输入时，TIMx_CCR 用于捕获（存储）在输入脉冲电平翻转时计数器 TIMx_CNT 的当前计数值，从而实现脉冲的频率测量或者脉宽测量。在产生脉冲时，TIMx_CCR 用于存储一个脉冲数值，把这个数值与计数器 TIMx_CNT 的当前计数值进行比较，根据比较结果进行不同的电平输出。

通用定时器具有 4 个独立通道，这些通道可以用来作为输入捕获、输出比较、PWM 生成和单脉冲模式输出。

以下事件发生时产生中断或 DMA：

（1）更新：计数器向上溢出/向下溢出，计数器初始化（通过软件或者内部/外部触发）；

（2）触发事件（计数器启动、停止、初始化或者由内部/外部触发计数）；

（3）输入捕获；

（4）输出比较。

通用定时器除了基本定时功能外，还可以用于测量输入信号的脉冲宽度（输入捕获）或者产生输出波形（产生 PWM 波形）等。

例 5.4.2 利用定时器 TIM4 输出比较功能，在 PB6 产生 1kHz、占空比为 10%的 PWM 信号，如图 5.4.3 所示。

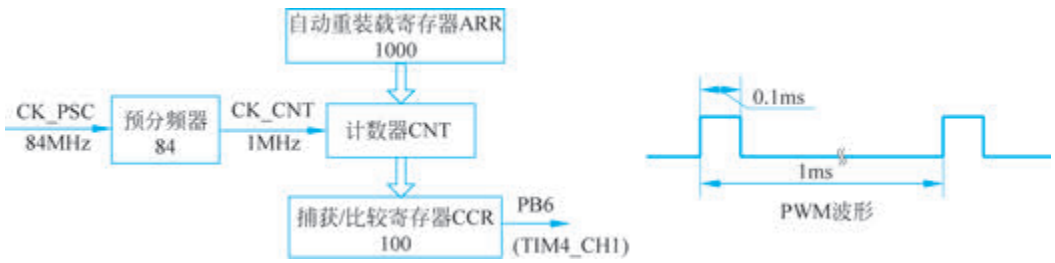


图 5.4.3 例 5.4.2 图

解： PWM 信号从 TIM4 的通道 1（TIM4_CH1）输出。根据 STM32F407VET6 单片机的数据手册，TIM4 的通道 1 与 PB6 对应。产生 PWM 波形的主要任务就是控制频率和占空比。频率和占空比分别通过自动重装寄存器 ARR 和捕获/比较寄存器 CCR 控制。计数器计到 ARR 寄存器的值后就清零并重新开始计数，这样 PWM 信号的频率就是 CK_CNT/(ARR+1)。在计数过程中，计数器的值会不停地与 CCR 中的数值进行比较。如果计数器的值小于 CCR 中的值，PB6 输出高电平，否则，PB6 输出低电平。可见，CCR 的值就控制了占空比。从图 5.4.3 可知，频率为 84MHz 的时钟信号首先通过预分频器进行 84 分频得到 1MHz 的计数器时钟 CK_CNT。ARR 设为 999，因此，PWM 信号

的周期为 1000μs。CCR 的值设为 100，因此，PWM 信号的高电平持续时间设为 100μs。

1）主程序相关代码

```
…
TIM4_PWM_Init(999,83);                                  //设置 PWM 信号频率
TIM_SetCompare1(TIM4,100);                              //设置 CCR 的值
…
```

2）PWM 初始化程序

PWM 初始化程序包括 PB6 初始化和 TIM4 初始化，源程序介绍如下：

```
void TIM4_PWM_Init(u32 arr,u32 psc);
{
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE);
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE);
  GPIO_PinAFConfig(GPIOB,GPIO_PinSource6,GPIO_AF_TIM4);
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_OType = GPIO_OType_PP;
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_NOPULL;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
  GPIO_Init(GPIOB,&GPIO_InitStructure);

  TIM_DeInit(TIM4);
  TIM_TimeBaseStructure.TIM_Prescaler = psc;                    //预分频
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //加计数
  TIM_TimeBaseStructure.TIM_Period = arr;                       //定时器的周期值
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;       //时钟分割
  TIM_TimeBaseInit(TIM4,&TIM_TimeBaseStructure);

  TIM_OCInitStructure.TIM_OCMode = TIM_OCMode_PWM1;             //配置为 PWM 模式 1
  TIM_OCInitStructure.TIM_OutputState = TIM_OutputState_Enable;
  TIM_OCInitStructure.TIM_Pulse = 0;                            //注 1
  TIM_OCInitStructure.TIM_OCPolarity = TIM_OCPolarity_High;     //注 2
  TIM_OC1Init(TIM4,&TIM_OCInitStructure);                       //输出比较通道初始化
  TIM_Cmd(TIM4,ENABLE);                                         //使能定时器 4
}
```

注 1：该语句用于设置 CRR 的值，当计数器计到这个值时，电平发生跳变。CRR 的值实际上就是脉冲宽度。这里将 CRR 的值设为 0，是因为 CRR 的值将在主程序中通过 TIM_SetCompare1(TIM4,100)函数来设定。

注 2：该语句表示当计数值小于跳变值时 PWM 信号设为高电平。

比较例 5.4.1 和例 5.4.2 两种产生方波的方法，例 5.4.2 的方法不需要软件开销，而且占空比可调。

例 5.4.3 利用定时器 TIM4 的输入捕获功能，测量方波信号的频率。输入捕获电路测频的框图如图 5.4.4 所示。

解：频率测量最简单的方法是将被测信号作为外部中断源，然后在外部中断服务程序中读取定时器中的计数值，将相邻两次的定时器值相减就是被测信号的周期，从而得

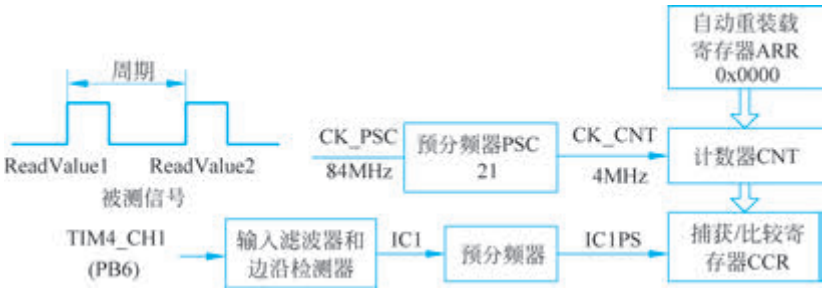


图 5.4.4　例 5.4.3 图

到被测信号的频率。由于外部中断处理需要时间，会影响测量精度。使用通用定时器的捕获功能，在指定脉冲边沿的时刻及时地将此时的计数器计数值锁存在“捕获/比较寄存器”中，从而有效地避免了上面提到的方法中进入中断时延造成的计时误差。

假设第 1 个上升沿时得到的计数值存在 ReadValue1 中，第 2 个上升沿时，得到的计数值存在 ReadValue2 中，则被测信号周期为

$$T = \text{ReadValue2} - \text{ReadValue1}$$

由于计数器的时钟 CK_CNT 频率为 4MHz，则被测信号的频率为

$$f = \frac{4000000}{T}(\text{Hz})$$

1）主程序相关代码

```
void TIM4_CH1_Cap_Init(u32 arr,u16 psc);
u16 Period;                                          //存放周期值
u16 ReadValue1,ReadValue2;
…
TIM4_CH1_Cap_Init(0xffff,21 - 1);                    //以 84MHz/21 = 4MHz 的频率计数
while(1)
    {
     s = 4000000/ Period;
     if (count == 32)                                //每 0.5s 显示 1 次
      {
         count = 0;
         LCD_ShowStringBig(170,160," ",YELLOW);      //清除上次显示值
         LCD_ShowNumBig(170,160,s,YELLOW);           //显示当前频率值
       }
     }
…
```

2）捕获中断服务程序

```
void TIM4_IRQHandler(void)
{
    if (TIM_GetITStatus(TIM4, TIM_IT_CC1) != RESET)      //捕获 1 发生捕获事件
      {
        ReadValue2 = TIM_GetCapture1(TIM4);
        Period = (ReadValue2 - ReadValue1);
        ReadValue1 = ReadValue2;
```

```
    }
    TIM_ClearITPendingBit(TIM4, TIM_IT_CC1|TIM_IT_Update);      //清除中断标志位
}
```

3）TIM4 初始化程序

```
void TIM4_CH1_Cap_Init(u32 arr,u16 psc)
{
  GPIO_InitTypeDef GPIO_InitStructure;
  TIM_TimeBaseInitTypeDef TIM_TimeBaseStructure;
  NVIC_InitTypeDef NVIC_InitStructure;
  TIM_ICInitTypeDef TIM4_ICInitStructure;
  RCC_APB1PeriphClockCmd(RCC_APB1Periph_TIM4,ENABLE);   //TIM4 时钟使能
  RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB,ENABLE);  //使能 PORTB 时钟

  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;      //速度为 50MHz
  GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_DOWN;         //下拉
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
  GPIO_Init(GPIOB,&GPIO_InitStructure);                  //初始化 PB6
  GPIO_PinAFConfig(GPIOB,GPIO_PinSource6,GPIO_AF_TIM4); //PB6 复用

  TIM_TimeBaseStructure.TIM_Prescaler = psc;             //定时器分频
  TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up; //向上计数模式
  TIM_TimeBaseStructure.TIM_Period = arr;                //自动重装载值
  TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
  TIM_TimeBaseInit(TIM4,&TIM_TimeBaseStructure);

  TIM4_ICInitStructure.TIM_Channel = TIM_Channel_1;      //选择输入端 IC1 映射到 TI1 上
  TIM4_ICInitStructure.TIM_ICPolarity = TIM_ICPolarity_Rising;          //上升沿捕获
  TIM4_ICInitStructure.TIM_ICSelection = TIM_ICSelection_DirectTI;      //映射到 TI1 上
  TIM4_ICInitStructure.TIM_ICPrescaler = TIM_ICPSC_DIV1;     //配置输入分频,不分频
  TIM4_ICInitStructure.TIM_ICFilter = 0x00;           //IC1F = 0000 配置输入滤波器,不滤波
  TIM_ICInit(TIM4, &TIM4_ICInitStructure);               //初始化 TIM4
  TIM_ITConfig(TIM4,TIM_IT_Update|TIM_IT_CC1,ENABLE);   //允许更新和捕获中断
  TIM_Cmd(TIM4,ENABLE);                                  //使能定时器 4

  NVIC_InitStructure.NVIC_IRQChannel = TIM4_IRQn;
  NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 2;      //抢占优先级 2
  NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;     //子优先级 0
  NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;        //IRQ 通道使能
  NVIC_Init(&NVIC_InitStructure);                       //根据指定的参数初始化 NVIC 寄存器
  }
```

视频

5.5 中断系统

单片机在正常运行程序时，由于内部或外部事件引起暂时中止现行程序，转去执行请求单片机为其服务的那个外设或事件的服务程序，等该服务程序执行完成后又返回被中止的地方继续运行程序，这个过程称为中断。中断过程示意如图 5.5.1 所示。

中断系统是单片机的重要组成部分。实时控制、及时处理紧急任务、故障处理、单片

机与外设之间的数据交换都需要依靠中断系统。如果没有中断，则单片机的工作效率会大打折扣。以键盘输入程序为例，何时按下按键是随机的，如果单片机采用不断查询按键状态的方式，则单片机几乎做不了其他事情。如果采用中断的方式，则有键按下时，键盘接口电路向单片机发出中断请求，单片机响应中断，执行键盘中断服务程序，完成按键处理后再返回主程序继续执行。通过中断这种机制来实现按键输入，工作效率自然就提高了。

图 5.5.1　中断过程示意图

中断系统通常由中断源、中断使能控制、中断优先级控制等几部分组成。中断源是指产生中断的外设。通常每个中断源都分配一个固定的中断入口地址，这个地址称为中断向量。单片机响应中断时，就跳转到中断源对应的中断入口地址，而在中断入口地址上，放置一条调用中断服务程序的语句，从而使单片机转而执行该中断源的中断服务程序。每个中断源都可以通过中断使能控制被允许或被禁止。中断源之间有优先级之分，高优先级可以中断低优先级程序。例如，假设发生了一个优先级比较低的中断，单片机转到其中断服务函数去执行，在执行过程中，发生更高优先级的中断，那么，单片机同样中止当前的代码，转到高优先级的中断源对应的中断入口去执行中断服务函数，当高优先级中断服务函数执行完成后再返回原来被中断的低优先级的中断服务函数断点处继续运行，运行完成后，返回主程序的断点片继续运行。中断源之间的优先级通过中断优先级控制电路来实现。

1. STM32F407 单片机的中断源

STM32F407 单片机总共有 92 个中断源，包括 10 个内核中断源和 82 个可屏蔽中断源。部分常用可屏蔽中断源如表 5.5.1 所示。中断向量名称在 stm32f4××.h 定义，中断服务程序名在 startup_stm32f407.s 中定义。

表 5.5.1　STM32F407 单片机部分常用可屏蔽中断源

中　断　源	中断向量	中断服务程序名
EXTI 线 0 中断	EXTI0_IRQn	EXTI0_IRQHandler
EXTI 线 1 中断	EXTI1_IRQn	EXTI1_IRQHandler
ADC1、ADC2 和 ADC3 全局中断	ADC_IRQn	ADC_IRQHandler
CAN1 TX 中断	CAN1_TX_IRQn	CAN1_TX_IRQHandler
CAN1 RX0 中断	CAN1_RX0_IRQn	CAN1_RX0_IRQHandler
TIM1 更新中断和 TIM10 全局中断	TIM1_UP_TIM10_IRQn	TIM1_UP_TIM10_IRQHandler
TIM1 捕获/比较中断	TIM1_CC_IRQn	TIM1_CC_IRQHandler
TIM2 全局中断	TIM2_IRQn	TIM2_IRQHandler
TIM3 全局中断	TIM3_IRQn	TIM3_IRQHandler
TIM4 全局中断	TIM4_IRQn	TIM4_IRQHandler
I2C1 事件中断	I2C1_EV_IRQn	I2C1_EV_IRQHandler
I2C1 错误中断	I2C1_ER_IRQn	I2C1_ER_IRQHandler
SPI1 全局中断	SPI1_IRQn	SPI1_IRQHandler

续表

中　断　源	中断向量	中断服务程序名
SPI2 全局中断	SPI2_IRQn	SPI2_IRQHandler
USART1 全局中断	USART1_IRQn	USART1_IRQHandler
TIM8 更新中断和 TIM13 全局中断	TIM8_UP_TIM13_IRQn	TIM8_UP_TIM13_IRQHandler
TIM8 捕获/比较中断	TIM8_CC_IRQn	TIM8_CC_IRQHandler
TIM5 全局中断	TIM5_IRQn	TIM5_IRQHandler
TIM6 全局中断 DAC1 和 DAC2 下溢错误中断	TIM6_DAC_IRQn	TIM6_DAC_IRQHandler
TIM7 全局中断	TIM7_IRQn	TIM7_IRQHandler

2. 嵌套矢量中断控制器(NVIC)

NVIC 是 Cortex-M4 的一个内部器件。所有含有 Cortex-M4 内核的单片机的 NVIC 是完全相同的，NVIC 的配置函数也是由 ARM 公司提供。NVIC 功能非常强大，在中断处理上效率很高，优先级配置也很灵活。NVIC 含有以下这些寄存器：

(1) 中断使能寄存器组 ISER[8]：ISER(Interrupt Set-Enable Register)由 8 个 32 位寄存器组成，寄存器中的每一位对应一个中断，因此，总共可以支持 256 个中断。STM32F407 总共有 92 个中断，因此，只需要用到 3 个 32 位寄存器就可以了。要使能某个中断，只须将 ISER 寄存器中的相应位置 1 即可。

(2) 中断除能寄存器组 ICER[8]：ICER(Interrupt Clear-Enable Register)与 ISER 的作用刚好相反，是用来清除某个中断使能的。这里专门设置了一个 ICER 来清除中断位，而不是对 ISER 写 0 来清除，这是因为这些寄存器都是写 1 有效的，写 0 是无效的。

(3) 中断挂起控制寄存器组 ISPR[8]：ISPR(Interrupt Set-Pending Registers)通过置 1，将已经产生中断请求但无法马上执行的中断挂起，等可以执行中断服务程序时，再执行挂起的中断。例如，当高、低级别的中断同时发生时，就挂起低级别中断，等高级别中断程序执行完，再执行低级别中断。

(4) 中断解挂控制寄存器组 ICPR[8]：ICRP(Interrupt Clear-Pending Registers)的作用与 ISPR 相反，通过置 1，可以将正在进行的中断解挂。

(5) 中断激活标志寄存器组 IABR[8]：IABR(Interrupt Active Bit Registers)某位置 1，表示该位所对应的中断正在被执行。这是一个只读寄存器。通过它可以知道当前在执行的中断是哪个。当中断执行完后由硬件自动清零。

(6) 中断优先级控制的寄存器组 IPR[240]：IPR(Interrupt Priority Registers)用于设置每个中断的抢占优先级(Preemption Priority)和子优先级(Sub Priority)。IPR 由 240 个 8 位寄存器组成，不过 STM32F407 单片机只用到其中的 82 个寄存器 IPR[0]～IPR[81]，而每个 8 位寄存器又只用到了其中的高 4 位，用来指定每个中断源的两种优先级。

对于上述寄存器，只需要作一般性了解，因为在实际编程中，通常不是直接对寄存器操作，而是调用相应的库函数来对寄存器间接操作。

3. 中断优先级

NVIC 中断优先级总共可以分为 5 个组，如表 5.5.2 所示。

表 5.5.2 NVIC 中断优先级

组	调 用 函 数	分 配 结 果
0	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_0)	0 位抢占优先级，4 位子优先级
1	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_1)	1 位抢占优先级，3 位子优先级
2	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2)	2 位抢占优先级，2 位子优先级
3	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_3)	3 位抢占优先级，1 位子优先级
4	NVIC_PriorityGroupConfig(NVIC_PriorityGroup_4)	4 位抢占优先级，0 位子优先级

假设中断优先级设在第 2 组，根据表 5.5.2，每个中断可以设置抢占优先级 0～3，子优先级亦为 0～3，数值越小所代表的优先级越高。

抢占优先级和子优先级遵循以下原则：高抢占优先级可以打断正在进行的低抢占优先级中断。抢占优先级相同的中断，高子优先级不可以打断低子优先级的中断，只有在两个中断同时发生的情况下，高子优先级中断先执行。如果两个中断的抢占优先级和子优先级都是一样的，则看哪个中断先发生就先执行。如果两个中断的抢占优先级和子优先级都是一样的，而且这两个中断同时到达，则根据它们在中断表中的排位顺序决定先处理哪个。

4. 外部中断

STM32F407 的中断控制器支持 23 个外部中断/事件请求。STM32F407 的 23 个外部中断为：

EXTI 0～15：对应外部 I/O 引脚的输入中断。

EXTI 16：连接到 PVD 输出。

EXTI 17：连接到 RTC 闹钟事件。

EXTI 18：连接到 USB OTG FS 唤醒事件。

EXTI 19：连接到以太网唤醒事件。

EXTI 20：连接到 USB OTG HS(在 FS 中配置)唤醒事件。

EXTI 21：连接到 RTC 入侵和时间戳事件。

EXTI 22：连接到 RTC 唤醒事件。

EXTI 0～15 这 16 个外部中断请求从单片机的 I/O 引脚输入，EXTI 16～22 这 7 个外部中断请求来自单片机内部相关外设。例如，EXTI 16 的中断请求来自可编程电压检测器 PVD(Programmable Voltage Detector)。PVD 将单片机的电源电压 V_{DD} 和参考电压比较，将比较结果作为 EXTI 16 的中断请求信号，这样就可以在中断服务程序中对电源电压异常时进行处理。

STM32F407 单片机的每个 I/O 引脚都可以作为外部中断 EXTI 0～15 的中断输入口。由于 STM32F407 单片机供 I/O 引脚使用的中断线只有 16 根，而 I/O 引脚却远不止 16 根，因此，每根中断线对应了多个 I/O 引脚，如图 5.5.2 所示。以线 EXTI 0 为例，它对应了 PA0、PB0、PC0、PD0、PE0 等多根 I/O 引脚，通过配置来决定对应的中断线配

置到哪根 I/O 引脚上。

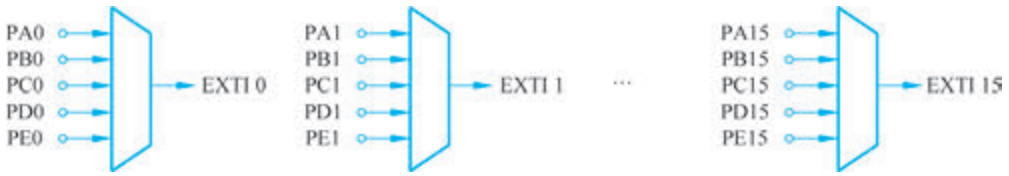


图 5.5.2　I/O 引脚跟中断线的映射关系

STM32F407 单片机外部中断原理框图如图 5.5.3 所示。

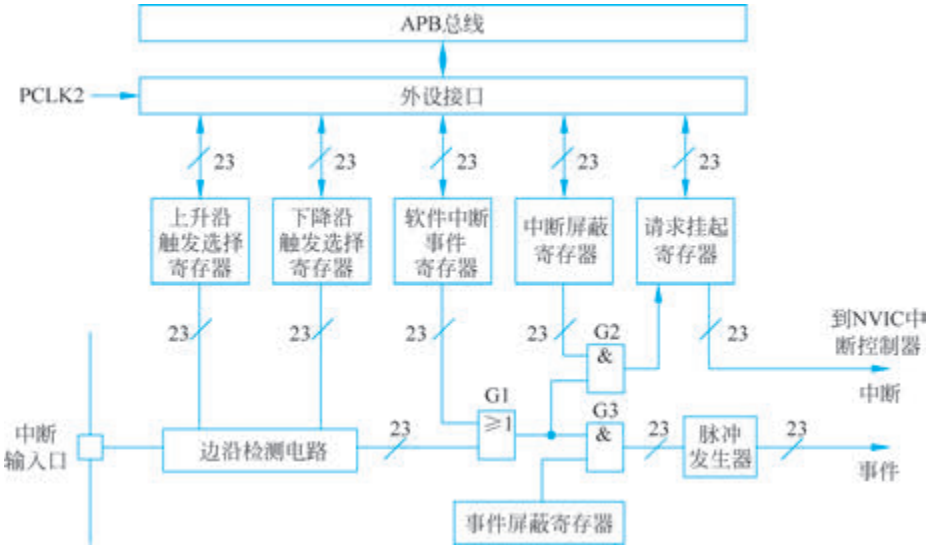


图 5.5.3　STM32F407 单片机外部中断原理框图

当中断输入口出现上升沿或者下降沿（究竟是上升沿有效还是下降沿有效，或者两者都有效，由图中的两个触发选择寄存器选择）时，边沿检测电路产生中断事件信号。中断事件信号经过或门 G1 后，同时送到两个与门 G2 和 G3。中断事件信号能否通过 G2 受中断屏蔽寄存器控制，如果中断屏蔽寄存器相应为置 1，中断事件信号就能通过 G2 形成中断请求信号，并保存到挂起寄存器中。如果 CPU 没有正在执行同级或更高级别的中断，挂起寄存器中的中断请求信号就会送到 NVIC 中。

中断事件信号能否通过 G3 受事件屏蔽寄存器控制，如果事件屏蔽寄存器相应为置 1，中断事件信号就能通过 G3 形成事件信号。脉冲发生器将事件信号转换成一个脉冲信号。该脉冲信号可以给其他电路使用，如启动 A/D 转换或者 DMA 传输等。

从图 5.5.3 可知，外部中断源同时产生了中断和事件，这里有必要对事件和中断的区别作一点说明。中断一定要有中断服务函数，但是事件却没有对应的函数。中断必须要 CPU 介入，但是事件可以在不需要 CPU 干预的情况下，执行一些操作。以外部 I/O 引脚触发 A/D 转换为例，如果使用中断通道，需要 I/O 引脚触发产生外部中断，外部中断服务程序启动 A/D 转换，A/D 转换完成，通过 A/D 中断服务程序读取转换结果。如果使用事件通道，I/O 引脚触发产生事件，然后事件触发 A/D 转换，A/D 转换完成，通过

A/D 中断服务程序读取转换结果。相比之下，事件触发 A/D 转换，响应速度更快，软件开销更小。可见，事件机制提供了一个完全由硬件自动完成的触发到产生结果的通道，提高了响应速度，是利用硬件来提升单片机处理事件能力的一种有效方法。

例 5.5.1 双音频信号发生器框图如图 5.5.4 所示。单片机在通过内部 DAC 在 PA4 引脚产生 1000Hz 和 800Hz 的正弦波，每种信号持续时间为 2s。

图 5.5.4 双音频信号发生器框图

解： 该双音频信号发生器需要采用以下 3 个中断源：

(1) 按键中断的外部中断(INT0)。通过键盘中断读取 4 位键值。4×4 键盘中有任一个键按下时，键盘编码电路的 DAV 引脚产生由高到低的跳变，向单片机发出中断请求。

(2) 定时器 1(TIM1)中断。通过 TIM1 中断控制 DAC 在 PA4 口产生频率为 1000Hz 和 800Hz 的正弦波信号。

(3) 定时器 7(TIM7)中断。通过 TIM7 中断实现 2s 的定时。

假设正弦波的每个周期由 256 点数据构成，为了得到 800Hz 和 1000Hz 的正弦波，TIM1 的定时时间计算如下：

$$T_1=\frac{10^6}{1000\times 256}\approx 3.906(\mu s),\quad T_2=\frac{10^6}{800\times 256}\approx 4.883(\mu s)$$

为了提高定时精度，采用高级定时器 TIM1，其计数时钟频率可达 168MHz。可计算得到 T_1 和 T_2 的定时时间常数为

$$TIMPR_1=3.906\mu s\times 168MHz\approx 656,\quad TIMPR_2=4.883\mu s\times 168MHz\approx 820$$

为了避免正弦波信号的失真，将 TIM1 中断优先级设为最高，TIM7 的中断优先级次之，INT0 的中断优先级最低。将中断优先级组设为 2，在中断初始化程序中将 INT0 的抢占优先级为 2，子优先级为 2；TIM7 中断的抢占优先级为 1，子优先级为 1；TIM1 中断的抢占优先级为 0，子优先级为 0。那么这 3 个中断的优先级顺序为：TIM1 中断＞TIM7 中断＞INT0 中断。

程序流程图如图 5.5.5 所示。

(1) 主程序

```
int main(void)
{
    …
    NVIC_PriorityGroupConfig(NVIC_PriorityGroup_2);        //中断分组
    EXTI0_init();                                          //外部中断初始化
    TIM1_init();                                           //TIM1 初始化
    TIM1INT_init();                                        //TIM1 中断初始化
```

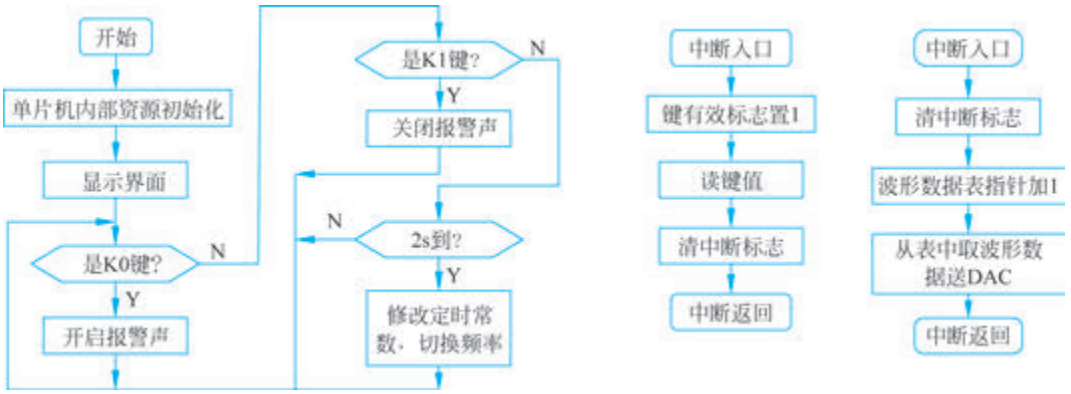


图 5.5.5　双音频信号发生器流程图

```
TIM7_init();                                          //TIM7 初始化
TIM7INT_init();                                       //TIM7 中断初始化
DAC_init();                                           //DAC 初始化
keysign = 0;
funsign = 0;
while(1)
{
   if(keysign == 1)
   {
      keysign = 0;                                    //键有效标志清零
      switch(keycode)
      {
         case 0x00:                                   //K0 键
         {
            TIM_Cmd(TIM1,ENABLE);                     //开启 TIM1 计数;
            break;
         }
         case 0x01:                                   //K1 键
         {
            TIM_Cmd(TIM1, ,DISABLE);                  //关闭 TIM1 计数;
            break;
         }
       }
    }
    if (count == 200)                                 //定时两秒
    {
     count = 0;
     funsign = ~funsign;                              //funsign 取反
     if(funsign == 0x00)
     {
        TIM_TimeBaseStructure.TIM_Period = TIMPR1;
        TIM_TimeBaseStructure.TIM_Prescaler = 0;      //设置时钟频率除数的预分频值
        TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
     }
     else
     {
```

```
                TIM_TimeBaseStructure.TIM_Period = TIMPR2;
                TIM_TimeBaseStructure.TIM_Prescaler = 0;
                TIM_TimeBaseInit(TIM1, &TIM_TimeBaseStructure);
            }
        }
    }
...
}
```

(2) 外部中断初始化函数

```
void EXTI0_init(void)
{
    RCC_AHB1PeriphClockCmd(RCC_AHB1Periph_GPIOB, ENABLE); //使能 GPIOB 时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_SYSCFG, ENABLE);//使能 SYSCFG 时钟
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN;          //将 I/O 引脚设置成输入模式
    GPIO_InitStructure.GPIO_PuPd = GPIO_PuPd_UP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_2MHz;
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_0;
    GPIO_Init(GPIOB, &GPIO_InitStructure);
    SYSCFG_EXTILineConfig(EXTI_PortSourceGPIOB, EXTI_PinSource0);     //注 1
    EXTI_InitStructure.EXTI_Line = EXTI_Line0;                        //注 2
    EXTI_InitStructure.EXTI_Mode = EXTI_Mode_Interrupt;               //注 3
    EXTI_InitStructure.EXTI_Trigger = EXTI_Trigger_Falling;           //注 4
    EXTI_InitStructure.EXTI_LineCmd = ENABLE;                         //注 5
    EXTI_Init(&EXTI_InitStructure);
    EXTI_ClearFlag(EXTI_Line0);

    NVIC_InitStructure.NVIC_IRQChannel = EXTI0_IRQn;                  //写入中断向量
    NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0x02;      //抢占优先级
    NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0x02;             //子优先级
    NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;                   //使能该中断
    NVIC_Init(&NVIC_InitStructure);
}
```

注 1：该语句用于配置 I/O 引脚与中断线的映射关系的函数，其功能是将 EXTI 0 连到 PB0。该函数在 stm32f4xx_syscfg.h 文件中。

注 2：该语句用于设置中断线的标号，其取值范围为 EXTI_Line0～EXTI_Line15。

注 3：该语句用于设置中断模式，可选值为中断 EXTI_Mode_Interrupt 和事件 EXTI_Mode_Event。

注 4：该语句用于设置触发方式，可选值为上升沿触发 EXTI_Trigger_Rising，下降沿触发 EXTI_Trigger_Falling，或者任意边沿触发 EXTI_Trigger_Rising_Falling。

注 5：该语句用于设置中断使能，可选值为 ENABLE 和 DISABLE。

(3) TIM1 初始化程序

```
void TIM1_init(void)
{
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_TIM1,ENABLE); //使能 TIM1
    TIM_TimeBaseStructure.TIM_Period = TIMPR1;          //设置自动重装载寄存器周期的值
    TIM_TimeBaseStructure.TIM_Prescaler = 0;            //设置时钟频率除数的预分频值
```

```
    TIM_TimeBaseStructure.TIM_ClockDivision = TIM_CKD_DIV1;
    //设置时钟分割 TIM_CKD_DIV1 = 0x0000
    TIM_TimeBaseStructure.TIM_RepetitionCounter = 0x00;
    //设置 RCR 寄存器值(这个只有高级定时器中有)
      TIM_TimeBaseStructure.TIM_CounterMode = TIM_CounterMode_Up;   //TIM1 向上计数
      TIM_TimeBaseInit(TIM1,&TIM_TimeBaseStructure);    //初始化 TIM1
      TIM_Cmd(TIM1,ENABLE);                             //开启 TIM1 计数
     //TIM_ClearFlag(TIM1, TIM_FLAG_Update);            //清溢出标志
}
```

(4) TIM1 中断初始化程序

```
void TIM1INT_init(void)
{
      NVIC_InitStructure.NVIC_IRQChannel = TIM1_UP_TIM10_IRQn;
      NVIC_InitStructure.NVIC_IRQChannelPreemptionPriority = 0;
      NVIC_InitStructure.NVIC_IRQChannelSubPriority = 0;
      NVIC_InitStructure.NVIC_IRQChannelCmd = ENABLE;
      NVIC_Init(&NVIC_InitStructure);
      TIM_ITConfig(TIM1,TIM_IT_Update,ENABLE);          //允许溢出中断
}
```

(5) TIM7 初始化程序和中断初始化程序见例 5.4.1。

(6) DAC 初始化程序参考 8.4 节有关内容。

(7) INT0(键盘中断)的中断服务程序

```
void EXTI0_IRQHandler(void)
{
    keycode = KEY_RAM;                      //读 4 位键值,KEY_RAM 为键盘接口的片选地址
    keycode &= 0x0F;
    keysign = 1;                            //设置键值有效标志
    EXTI_ClearITPendingBit(EXTI_Line0);     //清中断标志
}
```

(8) TIM1 的中断服务程序

```
void TIM1_UP_TIM10_IRQHandler(void)
{
  if (TIM_GetITStatus(TIM1, TIM_IT_Update) != RESET)     //是否发生定时器更新中断
  {
      TIM_ClearITPendingBit(TIM1, TIM_FLAG_Update);      //清除中断待处理位
      DACDAT = sindata[k];                               //读波形数据
      DACDAT = DACDAT << 4;                              //注 1
      k++;
      DAC_SetChannel1Data(DAC_Align_12b_R,DACDAT);       //通过波形数据到 DAC1
   }
}
```

注 1：DACDAT 为预先定义的 16 位无符号变量,从波形数据表中读取的数据为 8 位,通过这条左移 4 位指令,将 8 位波形数据转化为 12 位的 DAC 数据。

(9) TIM7 的中断服务程序

```
void TIM7_IRQHandler(void)
```

```
{
   if(TIM_GetITStatus(TIM7,TIM_IT_Update)!= RESET)
   {
      TIM_ClearITPendingBit(TIM7,TIM_IT_Update);
      count++;      //每隔 10ms 软件计数器 count 加 1
   }

}
```

通过本例，在编程中断有关的程序时，应注意以下几点：

（1）中断分组函数应放在所有中断源的中断初始化程序之前。

（2）如果分组为 2，则抢占优先级设为 4 级，子优先级设为 4 级。如果将某中断源的抢占优先级设为 0x0F，则相当于设为 0x03；如果将某中断源的抢占优先级设为 0x08，则相当于设为 0x00；总之，只有低两位有效。

（3）中断服务程序名可以通过查找表 5.5.1 得到。中断服务程序中要有清除中断标志的指令。

（4）中断服务函数编写原则：快进快出，在中断不要执行占用 CPU 较长时间的代码。


视频

思考题

1. 选择题。

（1）STM32F407 单片机的最高时钟频率为多少？

A. 42MHz　　B. 72MHz　　C. 84MHz　　D. 168MHz

（2）当 STM32F407 单片机与速度较慢的外设连接时，如何降低系统时钟的频率？

A. 减小锁相环倍频系数 PLL-N 的值

B. 增大锁相环倍频系数 PLL-N 的值

C. 增大锁相环分频系数 PLL-P 的值

D. 采用低速外部时钟 LSE

（3）关于 STM32F407 单片机，以下哪种说法是错误的？

A. 内部锁相环的作用是为了获得较高的系统时钟频率

B. 内部定时器 TIM1 的时钟频率可以为 168MHz

C. 使用外部晶振时，外部晶振的频率不能低于 4MHz

D. 采用复杂指令集 CISC

（4）STM32F407 单片机如果没有安装外部晶振，那么单片机的系统时钟来自哪里？

A. HSI　　B. LSI　　C. LSE　　D. HSE

2. STM32F407 单片机内部有哪几种总线？分别连接什么外设？

3. STM32F407 单片机有哪些时钟源？各有什么用途？

4. STM32F407 单片机内部有多少个定时器？可分成哪几类？

5. 计数时钟为 84MHz 时，基本定时器 TIM7 的最大定时时间是多少？

6. 用单片机在 I/O 引脚产生方波有哪几种方法？分别有什么优点？

7. 根据图 5.4.4 所示的参数设置，要求频率测量精度不低于 1%，允许被测信号的频率范围为多少？

8. 简述你对中断和事件这两个概念的理解。

9. 在编写 STM32F407 单片机的中断服务程序时，其函数名为什么不能任意命名？

10. 抢占优先级和子优先级有什么区别？