STM32快速入门

作者：蒋隆宇，刘平安

# 概述

## STM32F103在51上的升级

### 系统时钟：

1. 51的时钟源只能由外部时钟决定，一般采用12M晶振和11.0592M的晶振。

2. STM32103系统可以选择三种不同的时钟源：①HSI内部（RC）振荡器时钟（8MHz）②HSE外部振荡器时钟（外接晶振4-16MHz，典型值：8M）③PLL锁相环时钟。若选择HSE为PLL来源一般经过锁相环倍频处理后STM32时钟速率在72MHZ（主要考虑到stm32f103系统时钟经过分频后给内部总线，其中AHB和APB2域的最大频率是72MHz。APB1域的最大允许频率是36MHz）。但若选择内部高速RC时钟为PLL时钟来源，系统时钟最大频率只有64MHz。

另外stm32f103具有内部低速RC时钟（40k）时钟可以用于看门狗和RTC，具有外部低速时钟接口LSE（32.768kHz）一般用于RTC时钟（一般不用）。

**注意事项**：要注意RC时钟是不稳定的，在对时间要求比较高的场合，要尽量选择外部时钟。

### IO口：

1.STM32的IO口可以设置配置为多种模式：

─ 输入浮空  
─ 输入上拉  
─ 输入下拉  
─ 模拟输入（用于ADC的模拟量的输入通道）  
─ 开漏输出  
─ 推挽式输出（驱动能力强，由程序决定高低）  
─ 推挽式复用功能（驱动能力强，由复用部件决定高低，程序不控制但程序可以设置。）如作为串口输出，PWM输出的管脚，但前提是当前管脚有对应的复用。同时外设对应的IO口并不是绝对的，例如USART1最常用PA9 PA10但其实也可以是PB6 PB7不过配置会麻烦一些，这里要参考stm32数据手册。  
─ 开漏复用功能（电平由外部决定，程序不控制但程序可以设置。）如IIC。

**2.输出模式速度**

在输出模式下的最大速度（翻转频率）在stm32f103中可以配置为 2M、10M、50M，但越大功耗越大。

### 串口：

1. 51单片机只有一个串口，且波特率设置不高,最高为19200。
2. STM32以大容量（看后续选型）型号为例，其拥有5个串口，且配置更加多样化，波特率更高，最高达4.5Mbits/s，一般配置成115200.同时还拥有一个硬件数据流控（一般不常用）
3. 在中断方式上，51只有接收发送中断，而stm32f103的中断事件有

发送期间：发送完成、发送寄存器空（发送寄存器为空中断若打开只能在中断中发送数据，因为如果不在中断中去给数据寄存器放入数据就会不停的进入中断。而完成中断是多个字节发送完成后进入中断，经验证两个字节发送的间隔时间要小于一定值，若大于这个值就也会每次都进中断，中断标志每次读取后会自动清零）。

接收期间：空闲总线检测、溢出错误、接收数据寄存器非空、校验错误、 LIN断开符号检测、噪音标志(仅在多缓冲器通信)和帧错误(仅在多缓冲器通信)。

常用中断：有发送完成中断、发送寄存器为空（一般只判断标志）、读数据寄存器非空、空闲中断

1. 发送完成中断一般用来判断用户发送的一帧数据是否结束时会置一，用户自己写0来清除。置一条件：当发送数据寄存器在150us~200us没有数据写入时，会置一。
2. 发送寄存器为空中断一般用来判断是否写入发送数据寄存器，防止当前一个字节的数据被下一个字节的数据挤掉，且一般不会开启它的中断。
3. 读数据寄存器非空中断，一般用来判断是否接收到一个字节数据，可以通过读取DR寄存器清零，也可以手动清零。
4. 空闲中断

### 定时器：

51的定时器简单来说就是只能设置一个定时时间，计数器加一的时间是不可变的且只有两个定时器。而stm32的定时器：

1. STM32103中有8个16位定时器，分别为基本定时器、通用定时器，高级定时器（其中基本定时器只向上计数，通用和高级定时器可以向上或向下计数）。
2. 定时器的时钟来源是可以选择的，并且可以分频。启动定时器也可以利用多种触发方式，也可以给其他外设提供触发（如控制AD采样频率）。
3. 通用和高级定时器具有多种工作模式，常用的有

PWM捕获：在输入捕获模式下，当相应的通道检测到跳变沿后，将使用捕获/比较寄存器来锁存计数器的值，并可产生中断或 DMA 请求。

PWM输出：对外输出脉宽（即占空比）可调的方波信号，信号频率由自动重装寄存器 ARR 的值决定，占空比由比较寄存器 CCR 的值决定。

1. 系统定时器SysTick，最常用的是做延时，可以中断，在free rtos操作系统中作为时基。
2. 介绍事件中断。

### 外部中断

EXTI—外部中断/事件控制器，管理了控制器的 20 个中断/事件线。每个中断/事件线都对应有一个边沿检测器，可以实现输入信号的上升沿检测和下降沿的检测。EXTI 可以实现对每个中断/事件线进行单独配置，可以单独配置为中断或者事件，以及触发事件的属性。

1. 每个 GPIO 都可以被设置为输入线，占用 EXTI0 至 EXTI15，如PX0（X可以为A-G）都是中断事件0，PX1（X可以为A-G）都是中断/事件1。要注意的是一个中断/事件线每次只能设置一个管脚。

3.外部中断可以作为事件给其他外设做触发（如给定时器，ADC等）。

另外几个EXTI16-EXTI19我们基本是不用的，有兴趣可以看手册了解。

## --Stm32新增功能

### ADC

STM32f103 系列有 3个 ADC，精度为 12 位，每个 ADC 最多有 16 个外部通道。其中 ADC1 和ADC2 都有 16个外部通道， ADC3 根据 CPU 引脚的不同通道数也不同，一般都有 8个外部通道，通道的管脚设置参见手册。ADC电压输入范围为VREF- ≤ VIN ≤ VREF+，一般我们直接把 VSSA 和 VREF- 接地，把 VREF+ 和 VDDA 接 3V3，得到 ADC 的输入电压范围为： 0~3.3V。若想取得更好的效果也可以把VREF接上基准。

ADC采样是可以通过定时器触发的，通过定时器控制它的采样速率结合DMA传输数据再用FFT处理数据就可以得到频谱信息，ADC外部触发控制最快可以达到750KHZ-800KHZ（要注意配置相应的时钟）。也可以连续采样（没怎么试过）

### DAC

DAC的分辨率可配置为 8 位或 12 位的数字输入信号，具有两个 DAC输出通道，这两个通道互不影响，每个通道都可以使用 DMA 功能，都具有出错检测能力，可外部触发（定时器触发）。

与ADC类似DAC 也使用 VREF+ 引脚作为参考电压，在设计原理图的时候一般把 VSSA 接地，把 VREF+ 和 VDDA 接 3.3V，可得到 DAC 的输出电压范围为： 0~3.3V。DA输出缓冲使能后，输出电流大约可达12mA，输出电阻169欧左右，未缓冲使能输出电阻大约12K。

### DMA

DMA(Direct Memory Access)—直接存储器访问。它的主要功能是用来搬运数据，但是不需要占用 CPU，即在传输数据的时候， CPU 可以干其他的事情，好像是多线程一样。数据传输支持从外设到存储器、存储器到存储器，存储器到外设，最常用就是和串口、ADC、DAC结合使用。

### IIC及SPI

STM32 的 I2C 片上外设专门负责实现 I2C 通讯协议，也即是所谓的“IIC硬件协议”只要配置得当，它就会自动根据协议要求产生通讯信号，收发数据并缓存起来， CPU 只要检测该外设的状态和访问数据寄存器，就能完成数据收发。这种由硬件外设处理 I2C 协议的方式减轻了CPU的工作，STM32F103的IIC速度最高400K，但ST为了规避飞利浦 IIC 专利问题把STM32硬件IIC设计的比较复杂，而且稳定性不怎么好，不推荐使用。

与IIC外设一样了，STM32芯片也继承了专门用于SPI协议通讯的外设。STM32F103的SPI具有8或16位传输帧格式，最高速度为所在总线速度的二分频（SPI1在APB2总线，SPI2、SPI3在APB1总线），CS管脚可以设置成软件方式或者硬件方式。

### 看门狗

STM32 有两个看门狗，一个是独立看门狗(IWDG)，另外一个是窗口看门狗(WWDG)，独立看门狗号称宠物狗，窗口看门狗号称警犬，通俗来说就是两个12位的递减计数器。但是两个看门狗计数器使用的时钟源是不一样的。

独立看门狗的时钟源使用的是LSI RC振荡器（30~60kHz），但是根据温度和工作场合会有一定的漂移，我们一般取 40KHZ，所以独立看门狗的定时时间并一定非常精确，只适用于对时间精度要求比较低的场合，窗口看门狗的时钟源使用的是PCLK1（APB1），PCLK1的最大速度为36MHz。同时独立看门狗是其计数器递减计数到零发生复位，而窗口看门狗是当递减计数器的值小于0x40，则产生复位，并且要在小于窗口值时才能被重新装载否则也将发生复位，所以造成了宠物狗和警犬的区别。

### RTC时钟

TM32 的 RTC 外设（Real Time Clock），实质是一个掉电后还继续运行的定时器（VBAT供电）。从定时器的角度来说，相对于通用定时器 TIM 外设，它十分简单，只有很纯粹的计时和触发中断的功能；但从掉电还继续运行的角度来说，它却是 STM32 中唯一具有如此强大功能的外设。所以 RTC外设的复杂之处并不在于它的定时功能，而在于它掉电还继续运行的特性。它的时钟来源可以为LSE振荡器（一般就是32.768KHz）、LSI振荡器、HSE振荡器在128分频，并可以设置分频。

以上所说的掉电，是指主电源 VDD 断开的情况，为了 RTC 外设掉电继续运行，必须接上锂电池。

相当于将一个DS1302集成在芯片上，但RTC就是一个32位的计数器，需要自己转换成年月日，有溢出中断可以检测溢出。

### 电源管理PWR（低功耗）

对于电子设备来说，电源对于其重要性不言而喻，而保证系统能够稳定工作后，低功耗的要求也随之提出。STM32的低功耗模式总共有三种分别为睡眠模式，停止模式以及待机模式。三种模式对应不同的低功耗量级，可根据自己的需求选择。

### FSMC

**STM32F1 系列芯片使用 FSMC 外设来管理扩展的存储器， FSMC 是 Flexible Static Memory Controller 的缩写，译为灵活的静态存储控制器。它可以用于驱动包括 SRAM、 NOR FLASH 以及NANDFLSAH 类型的存储器，不能驱动如 SDRAM 这种动态的存储器。**

**对于本实验室常用用法是驱动TFT屏幕，进行高速刷新显示，以此来解决串口屏刷新速度慢的问题。**

### CAN

STM32 的芯片中具有 bxCAN 控制器 (Basic Extended CAN)，它支持 CAN 协议 2.0A 和 2.0B 标准。

该 CAN 控制器支持最高的通讯速率为 1Mb/s；可以自动地接收和发送 CAN 报文，支持使用标准ID 和扩展 ID 的报文；外设中具有 3 个发送邮箱，发送报文的优先级可以使用软件控制，还可以记录发送的时间；具有 2 个 3 级深度的接收 FIFO，可使用过滤功能只接收或不接收某些 ID 号的报文；可配置成自动重发；不支持使用 DMA 进行数据收发。

但是对于本实验室来说，基本不常用，只需要大概了解即可。

### USB

### SDIO

STM32F10x 系列控制器有一个 SDIO 主机接口，它可以与 MMC 卡、 SD 卡、 SD I/O 卡以及 CE-ATA 设备进行数据传输。MMC 卡可以说是 SD 卡的前身，现阶段已经用得很少。

SD I/O 卡本身不是用于存储的卡，它是指利用 SDIO 传输协议的一种外设。比如 Wi-Fi Card，它主要是提供 Wi-Fi 功能，有些 Wi-Fi 模块是使用串口或者 SPI 接口进行通信的，但 Wi-Fi SDIO Card 是使用 SDIO 接口进行通信的。并且一般设计 SD I/O 卡是可以插入到 SD 的插槽。

对于本实验室来说，基本不常用，简单了解即可。

## Stm32f103选型

以下为本实验室常用的几款STM32F1系列芯片的产品的功能以及外设配置，详细说明看相关数据手册规格说明相关章节：



表格

描述已自动生成

在进行芯片选型时，首先需要考虑到，需要用到哪些外设，其次为使用的外设数量，然后在选择合适的管脚数量，最后在判断需要的flash以及SRAM的大小。

STM32F103的后缀:

第一位代表管脚数C:48 R:64 V:100 Z:144

第二位代表存储器容量：8:64K B:128K C:256K D:384K E:512K

第三位表示封装：H=BGA T=LQFP Y=WLCSP64

第四位表示温度范围：6 = 工业级温度范围， -40°C~85°C

7 = 工业级温度范围， -40°C~105°C

在本实验室中，一般STM32RCT6基本就可以满足大部分需求，因为它属于大容量产品，而且其管脚数量为64，适用绝大部分场景。

## 相关工具

### KEIL

Keil 是单片机C语言软件开发系统工具，里面有C编译器、宏汇编、链接器、库管理和一个功能强大的仿真调试器。

其中在线仿真调试工具是一个非常强大而且方便的工具，在我们寻找程序错误中能够提供给我们非常大的便利，所以实验室的同学一定要熟悉在线仿真调试。

在相关工具文件夹中，我们附加了其安装包和详细的使用教程，可以请同学们翻阅。

### STM32CubeMX

STM32CubeMX是一种图形工具，通过分步过程可以非常轻松地配置STM32微控制器和微处理器，以及为Arm Cortex-M内核或面向Arm Cortex-A内核的特定Linux设备树生成相应的初始化C代码。

且在STM32F429之后，STM32没有标准库，能使用CubeMX行配置。

我们实验室的初学者在第一次接触到STM32时，并不是很推荐使用CubeMX行外设配置，因为这样会导致对STM32的寄存器理解不是很深入，而且CubeMX成的代码的运行效率不高，而且比较臃肿，不易理解。

但是我们在学习时钟配置时，由于库函数很抽象，建议使用CubeMX行简单的图形化配置，这样能够加深我们对STM32时钟数的理解。

在相关工具文件夹中，我们附加了其安装包和详细的使用教程，可以请同学们翻阅。

# 部件详细介绍

## RCC时钟配置

### 功能

时钟是单片机运行的基础，时钟信号推动单⽚机内各个部分执行相应的指令。时钟系统就是CPU的脉搏，决定cpu速率，像人的⼼跳⼀样，只有有了心跳，⼈才能做其他的事情，而单片机有了时钟，才能够运行执行指令，才能够做其他的处理 (点灯，串⼝，ADC)，时钟的重要性不言而喻。

#### STM32的时钟系统框图

图示, 示意图

描述已自动生成

乍一看很吓⼈，但其实很好理解，我们从系统时钟SYSCLK看起，左边选择系统时钟来源来源，系统时钟SYSCLK 的右边通过AHB预分频器，给各个外设提供时钟来源。

从左到右可以简单理解为 各个时钟源（外部晶振，内部RC）--->系统时钟来源的设置--->各个外设时钟的设置

#### 时钟系统

##### 各个时钟源（左边部分）

STM32 有4个独⽴时钟源:HSI、HSE、LSI、LSE。

①HSI是高速内部时钟，RC振荡器，频率为8MHz，精度不高。

②HSE是高速外部时钟，可接⽯英/陶瓷谐振器，或者接外部时钟源，频率范围为4MHz~16MHz。

③LSI是低速内部时钟，RC振荡器，频率为40kHz，提供低功耗时钟。

④LSE是低速外部时钟，接频率为32.768kHz的石英晶体。

其中LSI是作为IWDGCLK(独立看门狗)时钟源和RTC时钟源，独立使用。

HSI高速内部时钟 HSE高速外部时钟 PLL锁相环时钟 这三个经过分频或者倍频 作为系统时钟来使用。

PLL为锁相环倍频输出，其时钟输入源可选择为HSI/2、HSE或者HSE/2。倍频可选择为2~16倍。 通过倍频之后作为系统时钟的时钟源。

举个例子：Keil编写程序是默认的时钟为72Mhz，其实是这么来的：外部晶振(HSE)提供的8MHz（与电路板上的晶振的相关）通过PLLXTPRE分频器后，进⼊PLLSRC选择开关，进而通过PLLMUL锁相环进行倍频（x9）后，为系统提供72MHz的系统时钟（SYSCLK），倍频（X16）可以超频到128MHz。之后是AHB预分频器对时钟信号进行分频，然后为低速外设提供时钟。或者内部RC振荡器(HSI) 为8MHz /2 为4MHz 进⼊PLLSRC选择开关，通过PLLMUL锁相环进⾏倍频（x16）后 为64MHz。

##### 系统时钟SYSCLK

系统时钟SYSCLK可来源于三个时钟源：

①、HSI振荡器时钟

②、HSE振荡器时钟

③、PLL时钟

图示, 示意图

描述已自动生成

##### USB时钟

图示

描述已自动生成

STM32中有⼀个全速功能的USB模块，其串行接口引擎需要一个频率为48MHz的时钟源。该时钟源只能从PLL输出端获取（唯⼀的），，可以选择为1.5分频或者1分频，也就是，当需要使⽤USB模块时，PLL必须使能，并且时钟频率配置为48MHz或72MHz。

##### 把时钟信号输出到外部

图示

描述已自动生成

STM32可以选择⼀个时钟信号输出到MCO脚(PA8)上，可以选择为PLL输出的2分频、HSI、HSE、或者系统时钟。可以把时钟信号输出供外部使用。

当我们在使用串口时，发现我们设置的波特率与实际的波特率不对时，可以把时钟信号输出到外部，观察我们的系统频率是不是准确的。

##### 系统时钟通过AHB分频器给外设提供时钟（重重点）

时钟配置从左到右可以简单理解为：系统时钟--->AHB分频器--->各个外设分频倍频器 ---> 外设时钟的设置。

右边部分为：系统时钟SYSCLK通过AHB分频器分频后送给各模块使⽤，AHB分频器可选择1、2、4、8、16、64、128、256、512分频。一般直接1分频。

其中AHB分频器输出的时钟送给5⼤模块使用：该绿色部分不是很重要，适当了解即可。

1.内核总线：送给AHB总线（最大72MHz）、内核、内存和DMA使⽤的HCLK时钟。

2.Tick定时器：通过8分频后送给Cortex的系统定时器时钟。

3.I2S总线：直接送给Cortex的空闲运行时钟FCLK。

4.APB1外设（最大为36MHz）：送给APB1分频器。APB1分频器可选择1、2、4、8、16分频，其输出⼀路供APB1外设使用(PCLK1，最⼤频率36MHz)，另⼀路送给通用和基本定时器使用。当APB1分频器分频数为2时，给定时器的频率乘2，时钟输出供定时器2-7使用，时钟仍然为72MHz。

5.APB2外设（最大为72MHz）：送给APB2分频器。APB2分频器可选择1、2、4、8、16分频，其输出⼀路供APB2外设使用(PCLK2，最⼤频率72MHz)，另⼀路送给高级定时器。该倍频器可选择1或者2倍频，时钟输出供定时器1和定时器8使用。

另外，APB2分频器还有⼀路输出供ADC分频器使用，分频后送给ADC模块使用。ADC分频器可选择为2、4、6、8分频。ADC类型为逐次逼近型。

需要注意的是，如果 APB 预分频器分频系数是 1，则定时器时钟频率 (TIMxCLK) 为 PCLKx。否则，定时器时钟频率将为 APB 域的频率的两倍：TIMxCLK = 2xPCLKx。

APB1和APB2 的对应外设：

图示

低可信度描述已自动生成

APB1上面连接的是低速外设，包括电源接⼝、备份接口、CAN、USB、I2C1、I2C2、USART2、USART3、UART4、UART5、SPI2、SPI3等；

⽽APB2上面连接的是高速外设，包括UART1、SPI1、Timer1、ADC1、ADC2、ADC3、所有的普通I/O⼝（PA-PE）、第二功能I/O（AFIO）⼝等。

### 设置

#### RCC相关寄存器

RCC 寄存器结构，RCC\_TypeDeff，在⽂件“stm32f10x.h”中定义如下：

1059⾏~1081⾏

typedef struct

{

vu32 CR; //HSI,HSE,CSS,PLL等的使能

vu32 CFGR; //PLL等的时钟源选择以及分频系数设定

vu32 CIR; // 清除/使能 时钟就绪中断

vu32 APB2RSTR; //APB2线上外设复位寄存器

vu32 APB1RSTR; //APB1线上外设复位寄存器

vu32 AHBENR; //DMA，SDIO等时钟使能

vu32 APB2ENR; //APB2线上外设时钟使能

vu32 APB1ENR; //APB1线上外设时钟使能

vu32 BDCR; //备份域控制寄存器

vu32 CSR;

} RCC\_TypeDef;

#### RCC的初始化

这⾥我们使⽤HSE(外部时钟），正常使⽤的时候也都是使⽤外部时钟  
使⽤HSE时钟，程序设置时钟参数流程：  
1、将RCC寄存器重新设置为默认值 RCC\_DeInit;  
2、打开外部高速时钟晶振HSE RCC\_HSEConfig(RCC\_HSE\_ON);  
3、等待外部高速时钟晶振⼯作 HSEStartUpStatus = RCC\_WaitForHSEStartUp();  
4、设置AHB时钟 RCC\_HCLKConfig;  
5、设置高速AHB时钟 RCC\_PCLK2Config;（APB2）  
6、设置低速AHB时钟 RCC\_PCLK1Config;（APB1）  
7、设置PLL RCC\_PLLConfig;  
8、打开PLL RCC\_PLLCmd(ENABLE);  
9、等待PLL⼯作 while(RCC\_GetFlagStatus(RCC\_FLAG\_PLLRDY) == RESET)  
10、设置系统时钟 RCC\_SYSCLKConfig;  
11、判断是否PLL是系统时钟 while(RCC\_GetSYSCLKSource() != 0x08)  
12、打开要使⽤的外设时钟 RCC\_APB2PeriphClockCmd()/RCC\_APB1PeriphClockCmd()

代码实现：  
对RCC的配置函数(使⽤外部8MHz晶振)  
系统时钟72MHz，APH 72MHz，APB2 72MHz，APB1 32MHz，USB 48MHz TIMCLK=72M

void RCC\_Configuration(void)  
{  
 //----------使⽤外部RC晶振-----------  
 RCC\_DeInit(); //初始化为缺省值  
 RCC\_HSEConfig(RCC\_HSE\_ON); //使能外部的⾼速时钟  
 while(RCC\_GetFlagStatus(RCC\_FLAG\_HSERDY) == RESET); //等待外部⾼速时钟使能就绪  
 FLASH\_PrefetchBufferCmd(FLASH\_PrefetchBuffer\_Enable); //Enable Prefetch Buffer  
 FLASH\_SetLatency(FLASH\_Latency\_2); //Flash 2 wait state  
 RCC\_HCLKConfig(RCC\_SYSCLK\_Div1); //HCLK = SYSCLK  
 RCC\_PCLK2Config(RCC\_HCLK\_Div1); //PCLK2 = HCLK  
 RCC\_PCLK1Config(RCC\_HCLK\_Div2); //PCLK1 = HCLK/2  
 RCC\_PLLConfig(RCC\_PLLSource\_HSE\_Div1,RCC\_PLLMul\_9); //PLLCLK = 8MHZ \* 9 =72MHZ  
 RCC\_PLLCmd(ENABLE); //Enable PLLCLK  
 while(RCC\_GetFlagStatus(RCC\_FLAG\_PLLRDY) == RESET); //Wait till PLLCLK is ready  
 RCC\_SYSCLKConfig(RCC\_SYSCLKSource\_PLLCLK); //Select PLL as system clock  
 while(RCC\_GetSYSCLKSource()!=0x08); //Wait till PLL is used as system clock source  
 //---------打开相应外设时钟--------------------（选写）  
 RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA,ENABLE); //使能APB2外设的GPIOA时钟  
}

也就是我们时钟树框图从左到右的配置

### 拓展

#### 时钟监视器CSS

图示

描述已自动生成

STM32还提供了⼀个时钟监视系统（CSS），用于监视高速外部时钟（HSE）的⼯作状态。倘若HSE失效，会自动切换（高速内部时钟）HSI作为系统时钟的输⼊，保证系统的正常运行。

### 注意事项

时钟的合适的配置能够最大程度减少单片机的功耗，不同外设的时钟上限也是不一样的。

1. STM32时钟系统主要的目的就是给相对独立的外设模块提供时钟，也是为了降低整个芯片的耗能。
2. 系统时钟，是处理器运⾏时间基准（每⼀条机器指令一个时钟周期）
3. 时钟是单片机运行的基础，时钟信号推动单片机内各个部分执⾏相应的指令。
4. 一个单片机内提供多个不同的系统时钟，可以适应更多的应用场合。
5. 不同的功能模块会有不同的时钟上限，因此提供不同的时钟，也能在一个单⽚机内放置更多的功能模块。对不同模块的时钟增加开启和关闭功能，可以降低单片机的功耗
6. STM32为了低功耗，他将所有的外设时钟都设置为disable(不使能)，用到什么外设，只要打开对应外设的时钟就可以， 其他的没⽤到的可以还是disable(不使能)，这样耗能就会减少。 这就是为什么不管你配置什么功能都需要先打开对应的时钟的原因。

## GPIO

### 功能

如下图所示IO端口位的基本结构，stm32共有八种模式。

**输入**:上拉输入、下拉输入、浮空输入、模拟输入

输入模式比较容易理解除了模拟输入外就是上拉下拉的区别，模拟输入模式是专门为供ADC/DAC模式使用

**输出**: 推挽输出、开漏输出、 复用推挽输出、 复用开漏输出

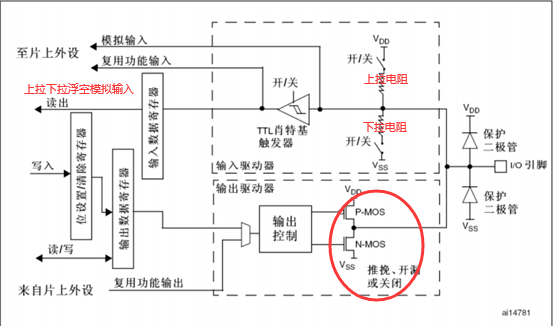
**推挽输出**时驱动能力比较强，如下图Pmos导通，N-mos截⽌，IO⼝等价直接连接在VDD上，所以IO⼝电平是⾼电平。同理输出寄存器的值为0时，P-mos截⽌，N-mos导通，IO⼝直接连接在VSS上，所有IO⼝电平为低电平。

**开漏输出**与推挽输出唯⼀的区别就是开漏输出只有⼀个N-mos管。当输出寄存器的值为0的时候，n-mos导通，IO⼝直接连接VSS，输出为低电平。当输出寄存器为1的时候，n-mos截⽌，IO⼝直接和输出端断开了，处于浮空状态。电平状态不可控制。

**复用推挽\开漏**和（推挽\开漏输出）区别在于起点不⼀样，复用输出来源⽚上外设，由复用部件决定高低，程序不控制但程序可以设置（直接改变寄存器的值，寄存器的值会改变，但是管脚的电平还是不会改变）。

用处：复用开漏输出---片内外设功能（MOSI,MISO,SCK，I2C的SCL,SDA）

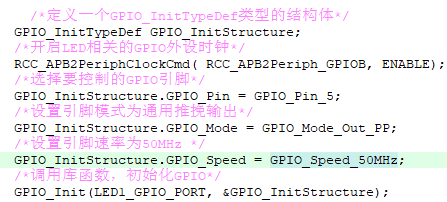
用推挽输出---片内外设功能（串口：TX）



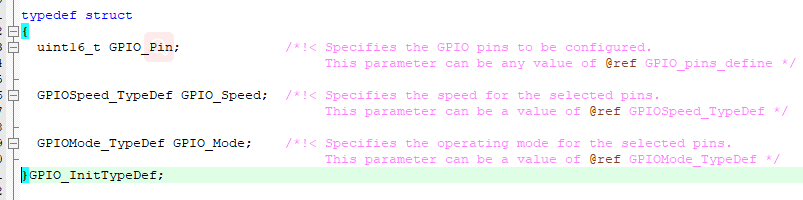
使用外设时具体的GPIO模式请参考《STM32F10xx参考手册》的8.1.11外设的GPIO配置章节。

### 设置-GPIO

如下图为GPIOB的PIN5配置为推挽输出的过程



在配置过程中首先需要打开对应GPIO的时钟；配置GPIO\_InitTypeDef结构体，如下图所示GPIO\_InitTypeDef结构体包括管脚序号、模式、管脚速度；然后调取GPIO\_Init函数初始化即可完成配置。（GPIO\_Init输入参数有端口和GPIO\_InitTypeDef结构体）



GPIO\_Init初始化函数实际上就是配置如下图所示的寄存器-GPIO\_TypeDef结构体中的CRL、CRH、BRR、BSRR的数据。

CRL与CRH：端口配置寄存器H代表管脚8-15，L代表管脚0-7；

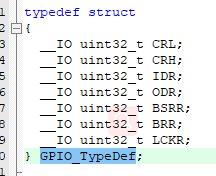
IDR：端口输入寄存器

ODR：端口输出寄存器

BSRR:端口位设置清除寄存器（32位寄存器每个管脚占两个位分别控制置高和清除）

BRR：端口位清除寄存器（16位寄存器控制每个管脚的清除）

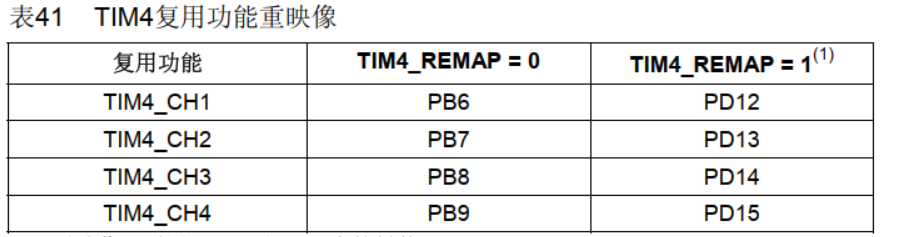
LCKR：端口配置锁定寄存器（没用过）



### 扩展-重映射

外设对应的管脚一般都是默认的，如果不想用默认管脚，要参考《参考手册》 “复用功能I/O和调试配置”章节以及数据手册，这里举个例子：

如图所示TIM4定时器的外部比较/捕获通道，一般是CH1-PB6、CH2-PB7、CH3-PB8、CH4-PB9，经过重映射是可以变为CH1-PD12、CH2- PD13、CH3- PD14、CH4- PD15。



**在配置重映射**时要先打开复用时钟，然后再调用重映射函数。如TIM4通道的配置过程就是先用RCC\_APB2PeriphClockCmd（RCC\_APB2Periph\_AFIO，ENABLE）打开复用时钟，然后用GPIO\_PinRemapConfig（GPIO\_Remap\_TIM4，ENABLE）即可配置完成。

### 扩展GPIO位操作

#### 位操作简介

位操作就是可以单独的对一个比特位读和写，这个在 51 单片机中非常常见。 51 单片机中通过关键字 sbit 来实现位定义， STM32 没有这样的关键字，而是通过访问位带别名区来实现。

在 STM32 中，有两个地方实现了位带，一个是 SRAM 区的最低 1MB 空间，令一个是外设区最低 1MB 空间。这两个 1MB 的空间除了可以像正常的 RAM 一样操作外，他们还有自己的位带别名区，位带别名区把这 1MB 的空间的每一个位膨胀成一个 32 位的字，当访问位带别名区的这些字时，就可以达到访问位带区某个比特位的目的。

**什么意思呢，简单来说就是这2MB的每一个位都有一个地址与之对应，直接写对应地址或者读对应地址的数据就是完成相关的位操作**。

#### 位操作别名区地址说明

对于片上外设位带区的某个比特，记它所在字节的地址为 A, 位序号为 n(0<=n<=7)，则该比特在别名区的地址如下图，0X42000000 是外设位带别名区的起始地址，0x40000000 是外设位带区的起始地址，（A-0x40000000）表示该比特前面有多少个字节，一个字节有 8 位，所以\*8，一个位膨胀后是 4 个字

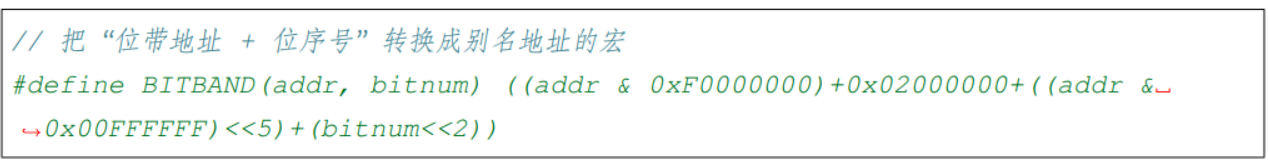
节，所以\*4，n表示该比特在A地址的序号，因为一个位经过膨胀后是四个字节，所以也\*4。



对于 SRAM 位带区的某个比特，记它所在字节的地址为 A, 位序号为 n(0<=n<=7)，则该比特在别名区的地址为：



为了方便操作，我们可以把这两个公式合并成一个公式，把“位带地址 + 位序号”转换成别名区地址统一成一个宏addr就是寄存器地址，bitnum就是位号。



#### 常用GPIO位操作过程

位操作一般就是给GPIO做位操作

过程如下：

1. 先定义一个宏可以输入地址和位后代表对应的位带别名区地址。



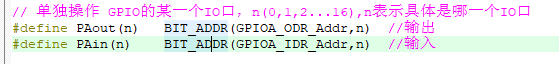
1. 再定义一个宏把地址强行转换为指针，如下图这个宏就是代表addr这个地址的数据。



1. 有了上边两个宏，我们就可以把它们结合如下图，这个宏就直接输入寄存器地址和位去代表对应位的数据。



4．以PA端口举例，如下图所示以PAout和PAin都对应宏上一步的BIT\_ADDR；它们都有一个输入参数n在表的是那一位，不一样的是后面的地址；PAout后边的地址是GPIOA\_ODR\_Addr这里也是一个宏对应的是PA端口输出寄存器的地址，这样我们将对应管脚初始化后可以用PAout（n）=0/1直接对PA的某一个管脚置一或者清零；而PAin后面的输入地址是PA端口的输入寄存器，初始化后可以读取PAin（n）的值就可以知道对应管脚的输入电平。



### 注意事项-默认外设管脚

**要注意有的端口管脚默认并不是普通的IO口。**

#### PD0，PD1

LQFP64封装的引脚5和引脚6在芯片复位后默认配置为OSC\_IN和OSC\_OUT功能脚。软件可以重新设置这两个引脚为PD0和PD1功能，而100脚以上PD0 PD1与OSC\_IN OSC\_OUT不冲突。

#### PB3、PB4、PA13、PA14、PA15

PB3、PB4、PA13、PA14、PA15默认都是用作JTAG/SWD仿真器调试接口的管脚，其中PA13，PA14分别作为SWD调试的SWIO和SWCLK；PB3，PB4，PA13，PA14，PA15共同用于JTAG。都是需要重映射后才可以当做普通IO口。

1.GPIO\_Remap\_SWJ\_JTAGDisable： /\* JTAG-DP Disabled and SW-DP Enabled \*/ 即能用PB3，PB4，PA15做普通IO，PA13&14用于SWD调试。

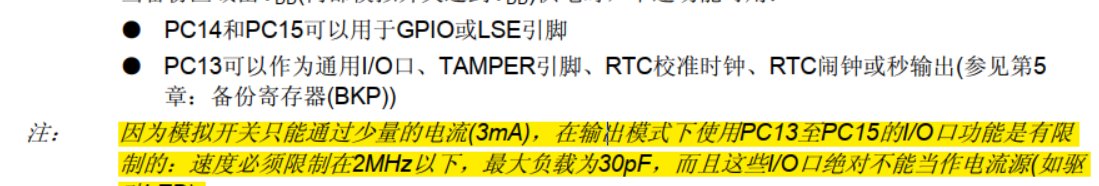
2.GPIO\_Remap\_SWJ\_Disable： /\* Full SWJ Disabled (JTAG-DP + SW-DP) \*/ 5个引脚全为普通引脚，但不能再用JTAG&SWD仿真器调试，只能用st-link调试 。

3.GPIO\_Remap\_SWJ\_NoJTRST： /\* Full SWJ Enabled (JTAG-DP + SW-DP) but without JTRST \*/PB4可为普通IO口，JTAG&SWD正常使用，但JTAG没有复位。

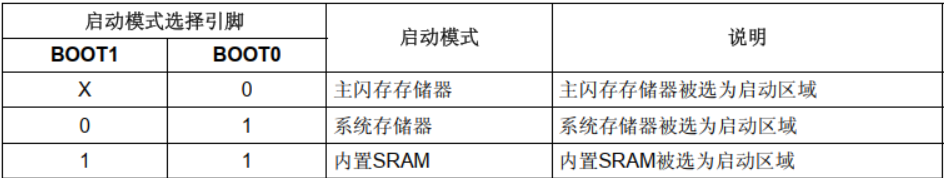
如果你用到所有的五个引脚当做普通IO口，那么上述步骤二中的重映射配置应写为GPIO\_PinRemapConfig(GPIO\_Remap\_SWJ\_Disable, ENABLE); 如果你用PB3，PB4，PA15做普通IO，PA13&14用于SWD调试，则重映射配置应写为GPIO\_PinRemapConfig(GPIO\_Remap\_SWJ\_JTAGDisable, ENABLE); 同理可配置只用PB4可为普通IO口的情况。

#### PC13,PC14,PC15

PC13， PC14和PC15引脚也比较特殊和RTC外设相关通过电源开关供电。一般PC14、PC15是默认接外部晶振32.86KHZ用的（其实也可以直接用）。因为开关只吸收有限的电流(3 mA)，在输出模式是受限的:速率不超过2MHz，最大负载30pf。这些I/O口不能用作电流源（例如驱动LED，但试验过用外部接高电平驱动时是可以接LED的，上电后是直接可以当IO口使用的）。



PB2是BOOT1管脚要注意上电时这个管脚的电平，但上电后程序开始运行后就是普通IO口了。（BOOT1 BOOT0控制启动方式如下图所示）。



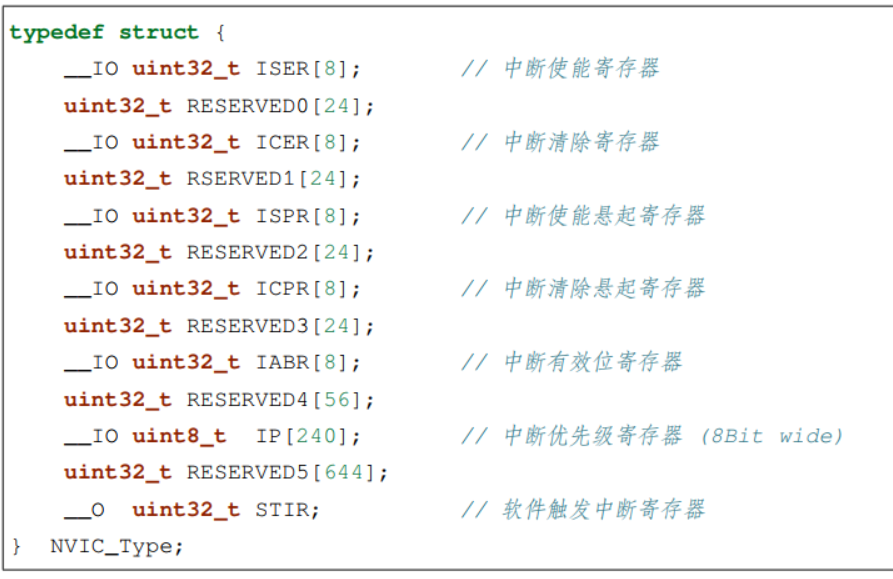
## 中断

### 功能

F103 在内核水平上搭载了一个异常响应系统，支持为数众多的系统异常和外部中断。其中系统异常有 8 个（如果把 Reset复位 和 HardFault硬件错误也算上的话就是 10 个），外部中断有 60 个。除了个别异常的优先级被定死外，其它异常的优先级都是可编程的。有关具体的系统异常和外部中断可在标准库文件 stm32f10x.h 这个头文件查询到，在 IRQn\_Type 这个结构体里面包含了 F103 系列全部的异常声明（异常也就是中断）。在基本使用中除了系统滴答定时器中断我们一般是不会用到系统中断的，在操作系统中切换任务就会用到一些系统异常，对相关中断有兴趣的可以看CM3权威指南相关章节。

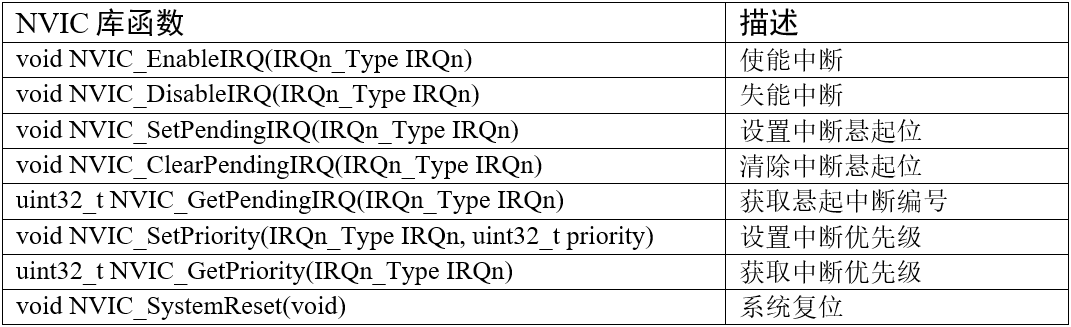
### NVIC寄存器简介

什么是NVIC呢？向量中断控制器，简称 NVIC，是 Cortex-M3 不可分离的一部分，它与 CM3 内核的逻辑紧密耦合，有一部分甚至水乳交融在一起。 NVIC 与 CM3 内核同声相应，同气相求，相辅相成，里应外合，共同完成对中断的响应，通过NVIC去控制中断的使能、失能以及优先级等。如下图在固件库中NVIC—Type结构体中就是NVIC中一些相关寄存器的映射。ISER中断使能寄存器每一位都代表一个中断，当然在103中它的位是没有用完的而且这里是不包括系统中断的配置，其他的详细介绍请看CM3权威指南。（注意结构体定义的和实际寄存器名字并不是一样的，如下图的ISE8[8]对应实际寄存器SETENA0- SETENA7；）。



**中断悬起、软件触发说明**：如果中断发生时，正在处理同级或高优先级异常，或者被掩蔽，则中断不能立即得到响应。此时中断被悬起。中断的悬起状态可以通过“中断设置悬起寄存器”和“中断悬起清除寄存器”来读取，还可以写它们来手工悬起中断，目前也还没有直接操作过。软件触发中断寄存器相关手册中对它的描述也比较少，应该不常用目前还没有直接操作过，手册中的解释是-软件中断，包括手工产生的普通中断，能以多种方式产生。最简单的就是使用相应的中断挂起寄存器；而更专业更快捷的作法，则是通过使用软件触发中断寄存器STIR 。

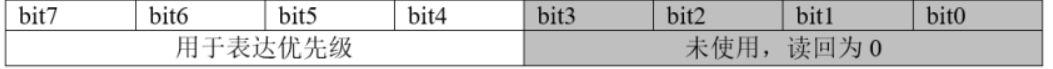
固件库文件 core\_cm3.h 的最后，还提供了 NVIC 的一些函数，这些函数遵循 CMSIS 规则，只要是 Cortex-M3 的处理器都可以使用，具体如下：



### 中断优先级说明

* 1. 上边我们大概先了解下NVIC是什么，这里我们详细说下其中的优先级配置。

在 NVIC 有一个专门的寄存器：中断优先级寄存器 NVIC\_IPRx，用来配置外部中断的优先级， IPR宽度为 8bit，原则上每个外部中断可配置的优先级为 0~255，数值越小，优先级越高。但是绝大多数 CM3 芯片都会精简设计，以致实际上支持的优先级数减少，在 F103 中，只使用了高 4bit，如下所示：



用于表达优先级的这 4bit，又被分组成抢占优先级和子优先级。如果有多个中断同时响应，抢占优先级高的就会抢占抢占优先级低的优先得到执行，如果抢占优先级相同，就比较子优先级。如果抢占优先级和子优先级都相同的话，就比较他们的硬件中断编号，编号越小，优先级越高。抢占优先级高的是可以打断低优先级的中断的，子优先级高是不可以打断中断的。

优先级分组优先级的分组由内核外设 SCB 的应用程序中断及复位控制寄存器 AIRCR 的 PRIGROUP[10:8] 位决定， F103 分为了 5 组。设置优先级分组可调用库函数 NVIC\_PriorityGroupConfig() 实现，有关 NVIC 中断相关的库函数都在库文件 misc.c 和 misc.h 中；函数输入参数中可以选择分组，有分组0-4选择第几分组抢占优先级就占几位（从高位看起）。

### EXTI简单介绍

#### 概述

每个 GPIO 都可以被设置为输入线，占用 EXTI0 至 EXTI15，如PX0（X可以为A-G）都是中断事件0，PX1（X可以为A-G）都是中断/事件1。要注意的是一个中断/事件线每次只能设置一个管脚。 即EXTI0 可以通过 AFIO 的外部中断配置寄存器选择配置为 PA0、 PB0、 PC0、PD0、PE0、PF0、PG0、PH0 或者PI0。

如下图所示为EXTI功能框图。配置过程简单来说就是配置为上升沿中断，下降沿中断，上升沿和下降沿触发中的一个，然后选择触发后产生中断还是给其他外设产生一个脉冲事件。

图示

描述已自动生成

#### 中断的传输路径

图中的蓝色虚线箭头，标出了外部中断信号的传输路径。首先，外部信号从编号1的芯片管脚进入，经过编号2的边沿检测电路，通过编号3的或门进入中断挂起请求寄存器，最后经过编号4的与门输出到NVIC中断检测电路。

首先是编号2的边沿检测电路：这个边沿检测电路受上升沿或下降沿选择寄存器控制，用户可以使用这两个寄存器控制需要哪一个边沿产生中断，因为选择上升沿或下降沿是分别受2个平行的寄存器控制，所以用户可以同时选择上升沿或下降沿，而如果只有一个寄存器控制，那么只能选择一个边沿了。

接下来，是编号3的或门，这个或门的另一个输入是软件中断/事件寄存器，从这里可以看出，软件可以优先于外部信号请求一个中断或事件，即当软件中断/事件寄存器的对应位为"1"时，不管外部信号如何，编号3的或门都会输出有效信号。

然后，一个中断或事件请求信号经过编号3的或门后，进入挂起请求寄存器，到此之前，中断和事件的信号传输通路都是一致的，也就是说，挂起请求寄存器中记录了外部信号的电平变化。

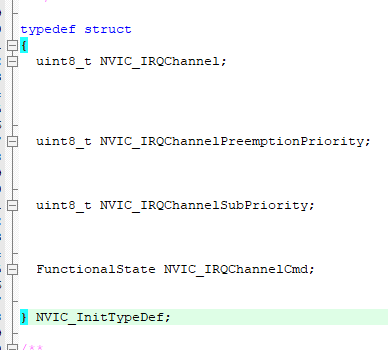
外部请求信号最后经过编号4的与门，向NVIC中断控制器发出一个中断请求，如果中断屏蔽寄存器的对应位为"0"，则该请求信号不能传输到与门的另一端，实现了中断的屏蔽。

#### 事件的传输路径

明白了外部中断的请求机制，就很容易理解事件的请求机制了。图中红色虚线箭头，标出了外部事件信号的传输路径，外部请求信号经过编号3的或门后，进入编号5的与门，这个与门的作用与编号4的与门类似，用于引入事件屏蔽寄存器的控制;最后脉冲发生器的一个跳变的信号转变为一个单脉冲，输出到芯片中的其它功能模块。

### EXTI中断编程过程

大概了解了NVIC和优先级分组。这里我们以外部中断举例说明中断的配置过程；**一开始要先调用NVIC\_PriorityGroupConfig函数设置中断分组；然后调用NVIC\_Init函数根据NVIC\_InitTypeDef结构体的数据去配置相关寄存器**，如下图为NVIC\_InitTypeDef结构体其中NVIC\_IRQChannel是中断源，NVIC\_IRQChannelPreemptionPriority是抢占优先级，NVIC\_IRQChannelSubPriority是子优先级，NVIC\_IRQChannelCmd是中断源使能；



配置EXTI中断分组以及优先级的相关代码如下，这里我选择中断分组1，中断源选择外部0中断，抢占优先级设置为1，子优先级也为1，并使能外部中断源。

void ange\_extinvicinit()

{

NVIC\_InitTypeDef nvicinit;

NVIC\_PriorityGroupConfig(NVIC\_PriorityGroup\_1);

nvicinit.NVIC\_IRQChannel=EXTI0\_IRQn;

nvicinit.NVIC\_IRQChannelPreemptionPriority=1;

nvicinit.NVIC\_IRQChannelSubPriority=1;

nvicinit.NVIC\_IRQChannelCmd=ENABLE;

NVIC\_Init(&nvicinit);

}

**当然肯定也要把要用的GPIO管脚初始化，配置成浮空输入，或者上下拉模式。**

**要注意的一定要打开复用时钟，然后使用GPIO\_EXTILineConfig函数选择中断源对应的管脚。**

**最后调用EXTI\_Init根据EXTI\_InitTypeDef中的数据写入相关寄存器完成对EXTI的配置。**

配置PA0管脚外部中断相关代码：

void ange\_extiinit()

{

GPIO\_InitTypeDef gpioinit;

EXTI\_InitTypeDef extiinit;

ange\_extinvicinit(); //NVIC初始化

RCC\_APB2PeriphClockCmd(RCC\_APB2Periph\_GPIOA|RCC\_APB2Periph\_AFIO,ENABLE); //要开复用时钟

gpioinit.GPIO\_Pin=GPIO\_Pin\_0;

gpioinit.GPIO\_Mode=GPIO\_Mode\_IN\_FLOATING; //浮空输入

GPIO\_Init(GPIOA,&gpioinit);

GPIO\_EXTILineConfig(GPIO\_PortSourceGPIOA,GPIO\_PinSource0); //配置AFIO->EXTICR寄存器选择EXTIx外部中断的输入源

extiinit.EXTI\_Line=EXTI\_Line0; //选择EXTI0

extiinit.EXTI\_Mode=EXTI\_Mode\_Interrupt; //产生中断

extiinit.EXTI\_Trigger=EXTI\_Trigger\_Rising; //上升沿触发

extiinit.EXTI\_LineCmd=ENABLE; //配置中断屏蔽，事件屏蔽寄存器

EXTI\_Init(&extiinit);

}

这样配置完成后就可以编写中断服务函数了，要注意在stm32中相关中断服务函数的名字已经是被命名好的，如下是EXTI0中断服务函数，一般情况下进入服务函数后都会先判断下中断标志位，然后**去手动清除标志位**，不清除标志位是会一直进中断的，而且有的中断服务函数是包含好几个中断的比如EXTI9\_5\_IRQn就是EXTI5-EXTI9的中断服务函数，如果开启这其中的多个中断，那么在函数中通过标志位去判断是那条中断线产生的中断就显得很有必要了。

void EXTI0\_IRQHandler()

{

if(EXTI\_GetITStatus(EXTI\_Line0)!=RESET)

{

EXTI\_ClearITPendingBit(EXTI\_Line0); //EXTI\_Line0

}

}

### 扩展中断与事件的区别

#### 中断与事件的概念

事件：是表示检测到某一动作（电平边沿）触发事件发生了。

中断：有某个事件发生并产生中断，并跳转到对应的中断处理程序中。

中断有可能被更优先的中断屏蔽，事件不会。

事件本质上就是一个触发信号,是用来触发特定的外设模块或核心本身(唤醒)。

事件还是其它一些操作的触发源，比如DMA，还有TIM中影子寄存器的传递与更新；

简单点就是中断一定要有中断服务函数,但是事件却没有对应的函数。

事件可以在不需要CPU干预的情况下,执行这些操作，但是中断则必须要CPU介入.。

#### 区别

从外部激励信号来看，中断和事件的产生源都可以是一样的。之所以分成2个部分，由于中断是需要CPU参与的，需要软件的中断服务函数才能完成中断后产生的结果。但是事件，是靠脉冲发生器产生一个脉冲，进而由硬件自动完成这个事件产生的结果，当然相应的联动部件需要先设置好，比如引起DMA操作，AD转换等。

　　简单举例：外部I/O触发AD转换，来测量外部物品的重量;如果使用传统的中断通道，需要I/O触发产生外部中断，外部中断服务程序启动AD转换，AD转换完成中断服务程序提交最后结果;要是使用事件通道，I/O触发产生事件，然后联动触发AD转换，AD转换完成中断服务程序提交最后结果;相比之下，后者不要软件参与AD触发，并且响应速度也更快。要是使用事件触发DMA操作，就完全不用软件参与就可以完成某些联动任务了。

#### 总结

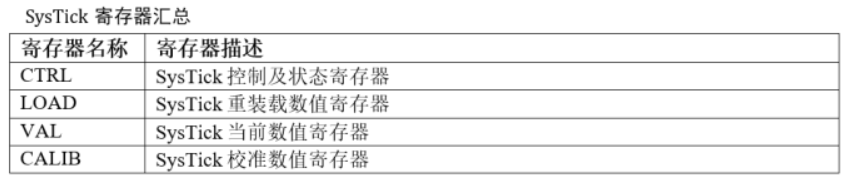
可以这样简单的认为，事件机制提供了一个完全由硬件自动完成的触发到产生结果的通道，不要软件的参与，降低了CPU的负荷，节省了中断资源，提高了响应速度(硬件总快于软件)，是利用硬件来提升CPU芯片处理事件能力的一个有效方法。

## SysTick

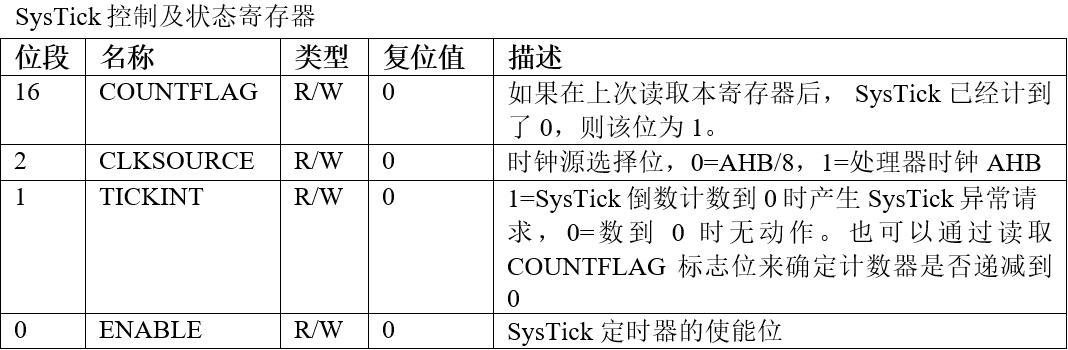
### SysTick简介

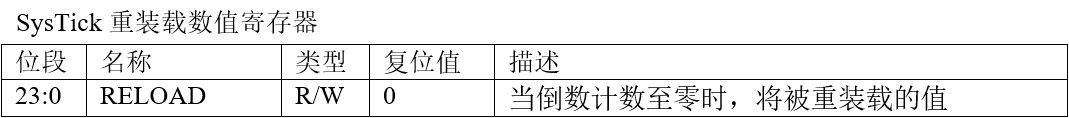
SysTick—系统定时器是属于 CM3 内核中的一个外设，内嵌在 NVIC 中。系统定时器是一个 24bit的向下递减的计数器，计数器每计数一次的时间为 1/SYSCLK，一般我们设置系统时钟 SYSCLK等于 72M。当重装载数值寄存器的值递减到 0 的时候，系统定时器就产生一次中断，以此循环往复。

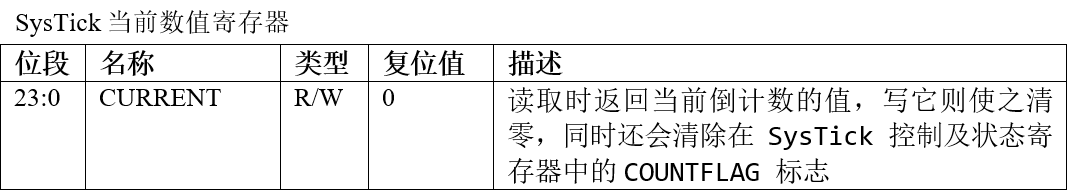
SysTick—系统定时器有 4 个寄存器，简要介绍如下。在使用 SysTick 产生定时的时候，只需要配置前三个寄存器，最后一个校准寄存器不需要使用。



### SysTick寄存器介绍







### SysTick配置过程

Stm32f103中直接是有相关的配置函数的，调用SysTick\_Config函数，输入参数会被放入重装载数值寄存器，并将数值寄存器清零，同时函数中会将SysTick控制及状态寄存器的低三位置一-所达到的效果就是选择AHB为时钟源（一般就等于系统时钟72M），可以自行修改函数改为AHB/8；然后就是控制寄存器位1置一会产生中断，这个SysTick\_Config函数默认是开启中断的，如果不想产生中断也可以自行修改这个函数；把位0置一开始计数。

目前使用最常用的就是用来做延时，可以通过判断中断次数来实现延时，或者是配置完后直接去判断控制寄存器位16（计数到零置一）。这里举一个例子：这个函数就是一个利用SysTick的简单延时函数，最小是1MS；将SysTick重装载值设置成72000也就是每1ms计数到0，后边是个循环等待要多少毫秒就等待多少次，最后又把SysTick失能了（停止计数）。

void SysTick\_delayms(uint32\_t n) //1ms

{

u32 i;

SysTick\_Config(72000);

for(i=0;i<n;i++)

{

while(((SysTick->CTRL)&(1<<16))==0);

}

SysTick->CTRL&=~0x01;

}

### 注意事项

要注意在主程序和中断中都使用SysTick定时器做延时，可能冲突，比如在主程序中正在延时突然进入中断，而中断中也用了这个延时方式可能会重新配置或者改变延时变量的值，从而改变主程序的时序。

## TIM

### 基础概述

#### 定时器分类

STM32F1 大容量产品中，共有 8 个定时器（中容量产品有所不同），分为基本定时器，通用定时器和高级定时器。基本定时器 TIM6 和 TIM7 是一个 16 位的只能向上计数的定时器，只能定时，没有外部IO。通用定时器 TIM2/3/4/5 是一个 16 位的可以向上/下计数的定时器，可以定时，可以输出比较，可以输入捕捉，每个定时器有四个外部 IO。高级定时器 TIM1/8 是一个 16 位的可以向上/下计数的定时器，可以定时，可以输出比较，可以输入捕捉，还可以有三相电机互补输出信号，每个定时器有 8 个外部 IO，详细介绍看下图。

因为定时器涉及到的知识点太多了，这里就先不做总体介绍了，后续根据要实现的功能再做相关的详细介绍。



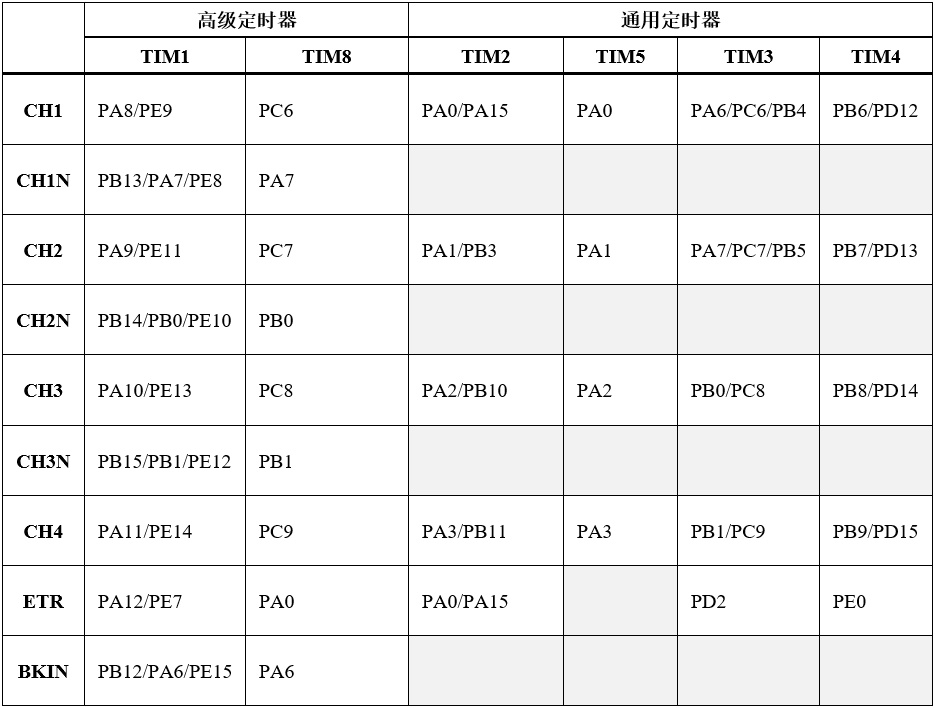
### 基本定时功能

#### 基本定时功能说明

基本定时功能的核心是时基，不仅基本定时器有，通用定时器和高级定时器也有。定时器时钟源默认使用的是内部时钟源，定时器2-7是在APB1总线上；定时器1和8是在APB2总线上。虽然在系统时钟为72MHz时APB1是36MHz它上边的定时器的内部时钟仍然是72MHz，这里是因为APB1和APB2预分频器不为1定时器频率是乘以2的，所以stm32f103中定时器内部时钟频率一般都是72MHz。（扩展stm32f407定时器内部时钟并不是全是168MHz）

基本定时器只有向上计数，高级和通用定时器可以向上向下计数。

STM32定时器相关的管脚如下图所示，CHx是外部通道区实现捕获以及PWM输出等功能，CHxN是互补通道，ETR是外部时钟引脚（看从模式和时钟源选择），BKIN是刹车输入引脚（看后续输出比较）



#### 相关寄存器

**TIMx\_PSC寄存器：定时器预分频**

定时器时钟经过 PSC 预分频器之后，即 CK\_CNT，用来驱动计数器计数。 PSC 是一个 16 位的预分频器，可以对定时器时钟 TIMxCLK 进行 1~65536 之间的任何一个数进行分频。具体计算方式为：CK\_CNT=TIMxCLK/(PSC+1)。

**TIMx\_CNT计数器：定时器计数器**

计数器 CNT 是一个 16 位的计数器，最大计数值为 65535。当计数达到自动重装载寄存器的时候产生更新事件，然后从头开始计数。

**TIMx\_ARR寄存器：TIM自动重装载寄存器**

自动重装载寄存器 ARR 是一个 16 位的寄存器，这里面装着计数器能计数的最大或者最小数值。当计数到这个值的时候，如果使能了中断的话，定时器就产生溢出中断。

**TIMx\_RCR寄存器：TIM重复计数寄存器**

重复计数器，只有高级定时器存在，假设原来设置的是10MS进入一次更新中断，但如果重复计数寄存器里面的值是99就会溢出99+1次才会进入更新中断，那么现在时间就变成了10\*（99+1）MS=1S。

#### 定时时间计算

定时器分频在写入寄存器时要在想要分频数值的基础上减一，那么计数器在 CK\_CNT 的驱动下，按定时器时钟为72M计算计一个数的时间则是 72M/(PSC+1);如果向上计数自动重装载值为ARR那么定时时间就是

（ARR+1）\*72M/(PSC+1)。

#### 定时器初始化结构体

如下图为定时器基本初始化结构体

TIM\_Prescaler：配置TIMx\_PSC寄存器决定预分频值；

TIM\_CounterMode:是计数方式包括向上计数、向下计数以及三种中心对齐模式。基本定时器只能是向上计数；

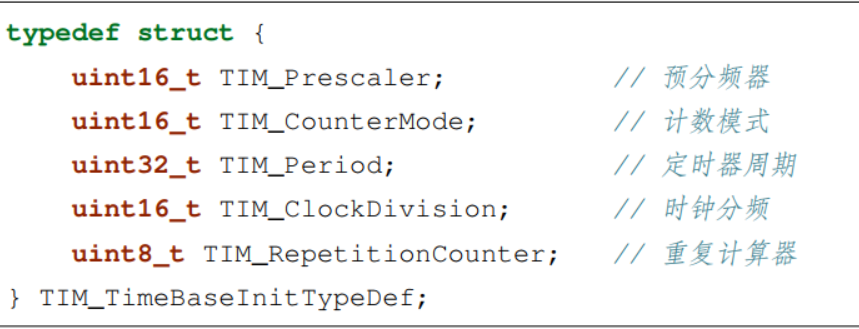
TIM\_Period：配置TIMx\_ARR寄存器决定定时周期；

TIM\_ClockDivision ：时钟分频，设置定时器时钟 CK\_INT 频率与数字滤波器采样时钟频率分频比和数字滤波器和PWM死区（看后续解释）输出有关，这里不需要不用设置

TIM\_RepetitionCounter：重复计数器，属于高级控制寄存器专用寄存器。这里不用

设置。

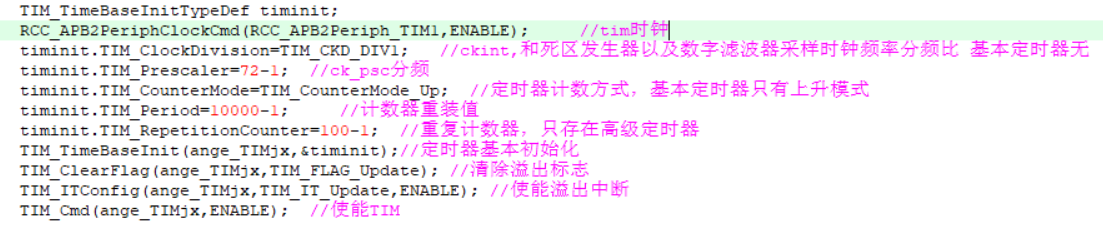
中心对齐：计数器从 0 开始递增计数，直到计数值等于 (ARR-1)，然后从 ARR 值开始递减计数直到 1如此循环，每次发生计数器上溢和下溢事件都会生成更新事件。具体看扩展。



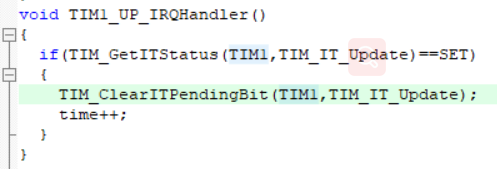
#### 基本定时功能设置过程

1. 如果要开启更新中断，首先还是要配置NVIC相关寄存器，NVIC配置参考外部中断。
2. 开启定时器时钟。
3. 然后定义配置TIM\_TimeBaseInitTypeDef结构体变量，用TIM\_TimeBaseInit函数初始化变量。
4. 要开启中断就使能溢出中断。
5. 使能定时器让定时器开始计数。

如下图是TIM1的配置过程首先开启在APB2总线的定时器时钟，定时器预分频设置为72写入71，定时器重装载值为9999周期是10000，重复计数器值为99重复100次。最后开启中断并使能计数。这里计算可以试着计算下进入更新中断的间隔时间t=1\*72/72M\*10000\*100=1000000US=1S



定时器更新中断服务函数如下，（TIM1的中断服务函数可不止一个，），在写各种中断时最好去中断向量表看一下。



### PWM捕获与输入捕获

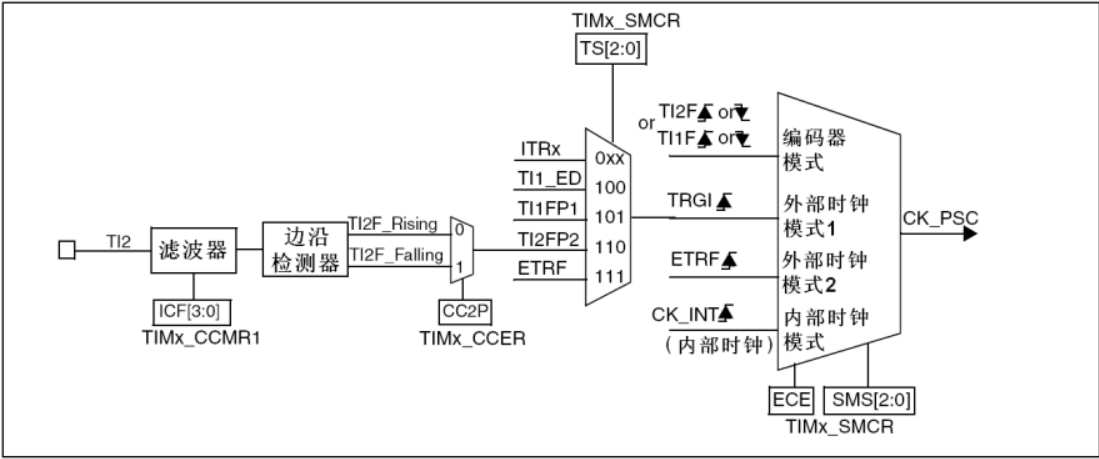
### PWM输出与输出比较

### 时钟源选择

定时器的计数时钟来源在一般情况下都是直接选择内部时钟72M，

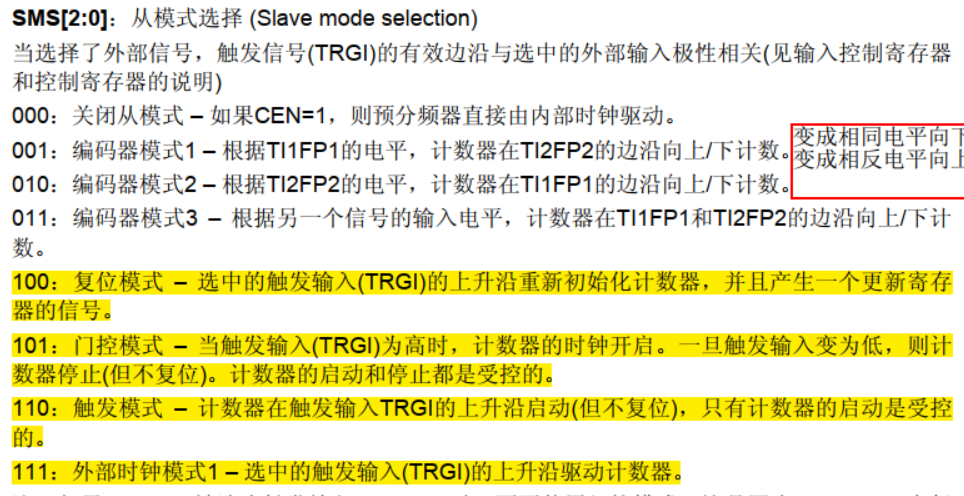
但它还有另外两种方式，这都时和定时器从模式有关（看后续）

#### 外部时钟源模式1

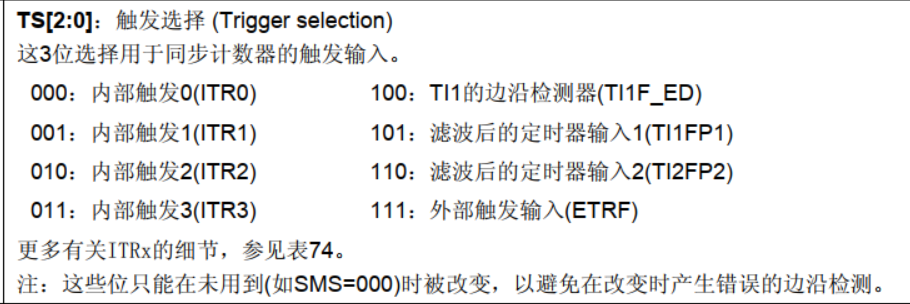


### 定时器从模式（很厉害）

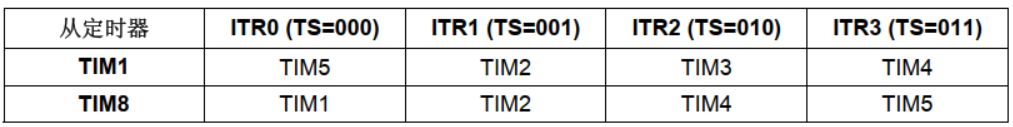
定时器能够在多种模式下和一个外部的触发同步：复位模式、门控模式和触发模式。这都和SMCR从模式控制寄存器有关，外部时钟源的选择，定时器同步，PWM输入捕获，编码器模式都会用到这个寄存器。



#### 触发源选择

定时器的触发输入（TRGI）有多种选择，如下图所示也是在SMCR寄存器中配置，不同定时器的内部触发ITRx是不同的，在中文参考手册SMCR寄存器的后面会有各个定时器ITRx的对应。

如下图所示是高级定时器内部触发的连接，例如TIM1定时器的内部触发ITR0来源是TIM5，在选择内部触发ITR0后还要配置TIM5的触发输出，触发输出的选择在CR2寄存器中配置。例如最常用的触发输出就是更新，当定时器计数器到达重装载值后溢出时，给外部做一次触发，具体请看手册。触发输入的选择使用函数TIM\_SelectInputTrigger；触发输出的选择使用函数TIM\_SelectOutputTrigger



#### 复位模式

复位模式的效果是选中的触发输入(TRGI)的上升沿重新初始化计数器，并且产生一个更新寄存器的信号

### 扩展定时器同步

### 扩展影子寄存器

### 扩展中心对齐

### 扩展编码器模式

## DMA

## ADC

## DAC

## IIC

## SPI

## RTC时钟

## 低功耗模式

# 应用示例